Structure and Interpretation of Signals and Systems

Edward A. Lee / Pravin Varaiya

Structure and Interpretation of Signals and Systems

Edward A. Lee and Pravin Varaiya
eal@eecs.berkeley.edu, varaiya@eecs.berkeley.edu
Electrical Engineering & Computer Science
University of California, Berkeley

July 4, 2000

Contents

Preface

This book is a reaction to a trauma in our discipline. We have all been aware for some time that "electrical engineering" has lost touch with the "electrical." Electricity provides the impetus, the pressure, the *potential*, but not the body. How else could microelectromechanical systems (MEMS) become so important in EE? Is this not a mechanical engineering discipline? Or signal processing. Is this not mathematics? Or digital networking. Is this not computer science? How is it that control system are applied equally comfortably to aeronautical systems, structural mechanics, electrical systems, and options pricing?

Like so many engineering schools, Berkeley used to have an introductory course entitled "Introduction to Electrical Engineering" that was about analog circuits. This quaint artifact spoke more about the origins of the discipline that its contemporary reality. Like engineering topics in schools of mining (which Berkeley's engineering school once was), ours has evolved more rapidly than the institutional structure around it.

Abelson and Sussman, in *Structure and Interpretation of Computer Programs* (MIT Press), a book that revolutionized computer science education, faced a similar transition in their discipline.

> "Underlying our approach to this subject is our conviction that 'computer science' is not a science and that its significance has little to do with computers."

Circuits used to be the heart of electrical engineering. It is arguable that today it is the analytical techniques that emerged from circuit theory that are the heart of the discipline. The circuits themselves have become an area of specialization. It is an important area of specialization, to be sure, with high demand for students, who command high salaries. But it is a specialization nonetheless.

Before Abelson and Sussman, computer programming was about getting computers to do your bidding. In the preface to *Structure and Interpretation of Computer Programs*, they say

> "First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."

In the origins of our discipline, a *signal* was a voltage that varies over time, an electromagnetic waveform, or an acoustic waveform. Now it is likely to be a sequence of discrete messages sent over the internet using TCP/IP. The *state* of a system used to be adequately captured by variables in a differential equation. Now it is likely to be the registers and memory of a computer, or more abstractly, a process continuation, or a set of concurrent finite state machines. A *system* used to be well-modeled by a linear time-invariant transfer function. Now it is likely to be a computation in a Turing-complete computational engine. Despite these fundamental changes in the medium with which we operate, the methodology remains robust and powerful. It is the methodology, not the medium, that defines our field. Our graduates are more likely to write software than to push electrons, and yet we recognize them as electrical engineers.

Fundamental limits have also changed. Although we still face thermal noise and the speed of light, we are likely to encounter other limits before we get to these, such as complexity, computability, chaos, and, most commonly, limits imposed by other human constructions. A voiceband data modem, for example, faces the telephone network, which was designed to carry voice, and offers as immutable limits such non-physical constraints as its 3 kHz bandwidth. DSL modems face regulatory constraints that are more limiting than their physical constraints. Computer-based audio systems face latency and jitter imposed by the operating system.

The mathematical basis for the discipline has also shifted. Although we still use calculus and differential equations, we frequently need discrete math, set theory, and mathematical logic. Indeed, a major theme of this book is to illustrate that formal techniques can be used in a very wide range of contexts. Whereas the mathematics of calculus and differential equations evolved to describe the physical world, the world we face as system designers often has non-physical properties that are not such a good match to this mathematics. Instead of abandoning formality, we need to broaden the mathematical base.

Again, Abelson and Sussman faced a similar conundrum.

> "... we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems."

This book is about signals and systems, not about large software systems. But it takes a computational view of signals and systems. It focuses on the methods "used to control the intellectual complexity," rather than on the physical limitations imposed by the implementations of old. Appropriately, it puts emphasis on discrete-time modeling, which is pertinent to the implementations in software and digital logic that are so common today. Continuous-time models describe the physical world, with which our systems interact. But fewer of the systems we engineer operate directly in this domain.

If imitation is the highest form of flattery, then it should be obvious whom we are flattering. Our title is a blatant imitation of *Structure and Interpretation of Computer Programs*. The choice of title reflects partly a vain hope that we might (improbably) have as much influence as they have. But more realistically, it reflects a sympathy with their cause. Like us, they faced an identity crisis in their discipline.

> "The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is what might best be called *procedural epistemology* – the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of 'what is.' Computation provides a fram-work for dealing precisely with notions of 'how to'."

Indeed, a major theme in our book is the connection between imperative (computational) and declarative (mathematical) descriptions of signals and systems. The laboratory component of the text, in our view, is an essential part of a complete study. The web content, with its extensive applets illustrating computational concepts, is an essential complement to the mathematics. Traditional electrical engineering has emphasized the declarative view. Modern electrical engineering has much closer ties to computer science, and has to complement this declarative view with an imperative one.

Besides being a proper part of the intellectual discipline, an imperative view has another key advantage. It enables a much closer connection with "real signals and systems," which are often too messy for a complete declarative treatment. While a declarative treatment can easily handle a sinusoidal signal, an imperative treatment can easily handle a voice signal. One of our objectives in designing this text and the course on which it is based is to illustrate concepts with real signals and systems *at every step*.

This is quite hard to do in a textbook. The print medium biases authors towards the declarative view simply because its static nature is better suited to the declarative view than to the imperative view. Our solution to this problem has been heavily influenced by the excellent and innovative textbooks by Steiglitz, *A Digital Signal Processing Primer – with Applications to Digital Audio and Computer Music* (Addison-Wesley), and McClellan, Schafer and Yoder, *DSP First – A Multimedia Approach*. Steiglitz leverages natural human interest in the very human field of music to teach, spectacularly gently, very sophisticated concepts in signals and systems. McClellan, Schafer and Yoder beautifully integrate web-based content and laboratory exercises, complementing the traditional mathematical treatment with accessible and highly motivational manipulation of real signals. If you are familiar with these books, you will see their influence all over the laboratory exercises and the web content.

Notes to Instructors

The course begins by describing signals as functions, focusing on characterizing the domain and the range. Systems are also described as functions, but now the domain and range are sets of signals. Characterizing these functions is *the* topic of this course. Sets and functions provide the unifying notation and formalism.

We begin by describing systems using the notion of state, first using automata theory and then progressing to linear systems. Frequency domain concepts are introduced as a complementary toolset, different from that of state machines, and much more powerful when applicable. Frequency decomposition of signals is introduced using psychoacoustics, and gradually developed until all four Fourier transforms (the Fourier series, the Fourier transform, the discrete-time Fourier transform, and the DFT) have been described. We linger on the first of these, the Fourier series, since it is conceptually the easiest, and then quickly present the others as simple generalizations of the Fourier series. Finally, the course closes by using these concepts to study sampling and aliasing, which helps bridge the computational world with the physical world.

This text has evolved to support a course we now teach regularly at Berkeley to about 500 students per year. An extensive accompanying web page is organized around Berkeley's 15 week semester, although we believe it can be adapted to other formats. The course organization at Berkeley is as follows:

Week 1 – Signals as Functions – Chapters 1 and 2. The first week motivates forthcoming material by illustrating how signals can be modeled abstractly as functions on sets. The emphasis is on characterizing the domain and the range, not on characterizing the function itself. The startup sequence of a voiceband data modem is used as an illustration, with a supporting applet that plays the very familiar sound of the startup handshake of V32.bis modem, and examines the waveform in both the time and frequency domain. The domain and range of the following signal types is given: sound, images, position in space, angles of a robot arm, binary sequences, word sequences, and event sequences.

Week 2 – Systems as Functions – Chapters 1 and 2. The second week introduces systems as functions that map functions (signals) into functions (signals). Again, it focuses not on how the function is defined, but rather on what is its domain and range. Block diagrams are defined as a visual syntax for composing functions. Applications considered are DTMF signaling, modems, digital voice, and audio storage and retrieval. These all share the property that systems are required to convert domains of functions. For example, to transmit a digital signal through the telephone system, the digital signal has to be converted into a signal in the domain of the telephone system (i.e., a bandlimited audio signal).

Week 3 – State – Chapter 3. Week 3 is when the students start seriously the laboratory component of the course. The first lecture in this week is therefore devoted to the problem of relating declarative and imperative descriptions of signals and systems. This sets the framework for making the intellectual connection between the labs and the mathematics.

The purpose of this first lab exercise is to explore arrays in Matlab and to use them to construct audio signals. The lab is designed to help students become familiar with the fundamentals of Matlab, while applying it to synthesis of sound. In particular, it introduces the vectorization feature of the Matlab programming language. The lab consists of explorations with sinusoidal sounds with exponential envelopes, relating musical notes with frequency, and introducing the use of discrete-time (sampled) representations of continuous-time signals (sound).

Note that there is some potential confusion because Matlab uses the term "function" somewhat more loosely than the course does when referring to mathematical functions. Any Matlab command that takes arguments in parentheses is called a function. And most have a well-defined domain and range, and do, in fact, define a mapping from the domain to the range. These can be viewed formally as a (mathematical) functions. Some, however, such as `plot` and `sound` are a bit harder to view this way. The last exercise in the lab explores this relationship.

The rest of the lecture content in the third week is devoted to introducing the notion of state and state machines. State machines are described by a function *update* that, given the current state and input, returns the new state and output. In anticipation of composing state machines, the concept of *stuttering* is introduced. This is a slightly difficult concept to introduce at this time because it has no utility until you compose state machines. But introducing it now means that we don't have to change the rules later when we compose machines.

Week 4 – Nondeterminism and Equivalence – Chapter 3. The fourth week deals with nondeterminism and equivalence in state machines. Equivalence is based on the notion of simulation, so simulation relations and bisimulation are defined for both deterministic and nondeterministic machines. These are used to explain that two state machines may be equivalent even if they have a different number of states, and that one state machine may be an abstraction of another, in that it has all input/output behaviors of the other (and then some).

During this week, students do a second set of lab exercises that explore the representation of images in Matlab, relating the Matlab use of color maps with a formal functional model. It discusses the file formats for images, and explores the compression that is possible with colormaps and with more sophisticated techniques such as JPEG. The students construct a simple movie, reinforcing the notions of sampling introduced in the previous lab. They also blur an image and create a simple edge detection algorithm for the same image. This lab also reinforces the theme of the previous one by asking students to define the domain and range of mathematical models of the relevant Matlab functions. Moreover, it begins an exploration of the tradeoffs between vectorized functions and lower-level programming constructs such as for loops. The edge detection algorithm is challenging (and probably not practical) to design using only vectorized functions. As you can see, the content of the labs lags the lecture by about one week so that students can use the lab to reinforce material that they have already had some exposure to.

Week 5 – Composition – Chapter 4. This week is devoted to composition of state machines. The deep concepts are synchrony, which gives a rigorous semantics to block diagrams, and feedback. The most useful concept to help subsequent material is that feedback loops with delays are always well formed.

The lab in this week (C.3) uses Matlab as a low-level programming language to construct state machines according to a systematic design pattern that will allow for easy composition. The theme of the lab is establishing the correspondence between pictorial representations of finite automata, mathematical functions giving the state update, and software realizations.

The main project in this lab exercise is to construct a virtual pet. This problem is inspired by the Tamagotchi virtual pet made by Bandai in Japan. Tamagotchi pets, which translate as "cute little eggs," were extremely popular in the late 1990's, and had behavior considerably more complex than that described in this exercise. The pet is cat that behaves as follows:

> It starts out *happy*. If you *pet* it, it *purrs*. If you *feed* it, it *throws up*. If *time passes*, it gets *hungry* and *rubs* against your legs. If you feed it when it is hungry, it purrs and gets happy. If you pet it when it is hungry, it *bites* you. If time passes when it is hungry, it *dies*.

The italicized words and phrases in this description should be elements in either the state space or the input or output alphabets. Students define the input and output alphabets and give a state transition diagram. They construct a function in Matlab that returns the next state and the output given the current state and the input. They then write a program to execute the state machine until the user types 'quit' or 'exit.'

Next, the students design an open-loop controller that keeps the virtual pet alive. This illustrates that systematically constructed state machines can be easily composed.

This lab builds on the flow control constructs (for loops) introduced in the previous labs and introduces string manipulation and the use of M files.

Week 6 – Linear Systems. We consider linear systems as state machines where the state is a vector of reals. Difference equations and differential equations are shown to describe such state machines. The notions of linearity and superposition are introduced.

In the previous lab, students were able to construct an open-loop controller that would keep their virtual pet alive. In this lab (C.4), they modify the pet so that its behavior is nondeterministic. In particular, they modify the cat's behavior so that if it is hungry and they feed it, it sometimes gets happy and purrs (as it did before), but it sometimes stays hungry and rubs against your legs. They then attempt to construct an open-loop controller that keeps the pet alive, but of course no such controller is possible without some feedback information. So they are asked to construct a state machine that can be composed in a feedback arrangement such that it keeps the cat alive.

The semantics of feedback in this course are consistent with tradition in signals systems. Computer scientists call this style "synchronous composition," and define the behavior of the feedback system as a (least or greatest) fixed point of a monotonic function on a partial order. In a course at this level, we cannot go into this theory in much depth, but we can use this example to explore the subtleties of synchronous feedback.

In particular, the controller composed with the virtual pet does not, at first, seem to have enough information available to start the model executing. The input to the controller, which is the output of the pet, is not available until the input to the pet, which is the output of the controller, is available. There is a bootstrapping problem. The (better) students learn to design state machines that can resolve this apparent paradox.

Most students find this lab quite challenging, but also very gratifying when they figure it out. The concepts behind it are deep, and the better students realize that. The weaker students, however, just get by, getting something working without really understanding how to do it systematically.

Week 7 – Response of Linear Systems. Matrices and vectors are used to compactly describe systems with linear and time-invariant state updates. Impulses and impulse response are introduced. The deep concept here is linearity, and the benefits it brings, specifically being able to write the state and output response as a convolution sum.

We begin to develop frequency domain concepts, using musical notes as a way to introduce the idea that signals can be given as sums of sinusoids.

There is no lab exercise this week, giving students more time to prepare for the first midterm exam.

Week 8 – Frequency Domain. This week introduces frequency domain concepts and the Fourier series. Periodic signals are defined, and Fourier series coefficients are calculated by inspection for certain signals. The frequency domain decomposition is motivated by the linearity of systems considered last week (using the superposition principle), and by psychoacoustics and music.

In the lab in this week (C.5), the students build on the previous exercise by constructing state machine models (now with infinite states and linear update equations). They build stable, unstable, and marginally stable state machines, describing them as difference equations.

The prime example of a stable system yields a sinusoidal signal with a decaying exponential envelope. The corresponding state machine is a simple approximate model of the physics of a plucked string instrument, such as a guitar. It is also the same signal that the students generated in the first lab by more direct (and more costly) methods. They compare the complexity of the state machine model with that of the sound generators that they constructed in the first lab, finding that the state machine model yields sinusoidal outputs with considerably fewer multiplies and adds than direct calculation of trigonometric and exponential functions.

The prime example of a marginally stable system is an oscillator. The students discover that an oscillator is just a boundary case between stable and unstable systems.

Week 9 – Frequency Response. In this week, we consider linear, time-invariant (LTI) systems, and introduce the notion of frequency response. We show that a complex exponential is an eigenfunction of an LTI system. The Fourier series is redone using complex exponentials, and frequency response is defined in terms of this Fourier series, for periodic inputs.

The purpose of the lab in week 9 (C.6) is to experiment with models of continuous-time systems that are described as differential equations. The exercises aim to solidify state-space concepts while giving some experience with software that models continuous-time systems.

The lab uses Simulink, a companion to Matlab. The lab is self contained, in the sense that no additional documentation for Simulink is needed. Instead, we rely on the on-line help facilities. However, these are not as good for Simulink as for Matlab. The lab exercise have to guide the students extensively, trying to steer clear of the more confusing parts. As a result, this lab is bit more "cookbook-like" than the others.

Simulink is a block-diagram modeling environment. As such, it has a more declarative flavor than Matlab, which is imperative. You do not specify exactly how signals are computed in Simulink. You simply connect together blocks that represent systems. These blocks declare a relationship between the input signal and the output signal. One of the reasons for using Simulink is to expose students to this very different style of programming.

Simulink excels at modeling continuous-time systems. Of course, continuous-time systems are not directly realizable on a computer, so Simulink must discretize the system. There is quite a bit of sophistication in how this is done, but the students are largely unaware of that. The fact that they do not specify how it is done underscores the observation that Simulink has a declarative flavor.

Week 10 – Filtering. The use of complex exponentials is further explored, and phasors and negative frequencies are discussed. The concept of filtering is introduced, with the terms lowpass, bandpass, and highpass, with applications to audio and images. Composition of LTI systems is introduced, with a light treatment of feedback.

The purpose of the lab (C.7) is to learn to examine the frequency domain content of signals. Two methods are used. The first method is to plot the discrete Fourier series coefficients of finite signals. The second is to plot the Fourier series coefficients of finite segments of time-varying signals, creating a spectrogram.

The students have, by this time, done quite a bit with Fourier series, and have established the relationship between finite signals and periodic signals and their Fourier series.

Matlab does not have any built-in function that directly computes Fourier series coefficients, so an implementation using the FFT is given to the students. The students construct a chirp, listen to it, study its instantaneous frequency, and plot its Fourier series coefficients. They then compute a time-varying discrete-Fourier series using short segments of the signal, and plot the result in a waterfall plot. Finally, they render the same result as a spectrogram, which leverages their study of color maps in lab 2. The students also render the spectrogram of a speech signal.

The lab concludes by studying beat signals, created by summing sinusoids with closely spaced frequencies. A single Fourier series analysis of the complete signal shows its structure consisting of two distinct sinusoids, while a spectrogram shows the structure that corresponds better with what the human ear hears, which is a sinusoid with a low-frequency sinusoidal envelope.

Week 11 – Convolution. We describe signals as sums of weighted impulses and then use linearity and time invariance to derive convolution. FIR systems are introduced, with a moving average being the prime example. Implementation of FIR systems in software and hardware is discussed, and signal flow graphs are introduced. Causality is defined.

The purpose of the lab (C.8) is to use a comb filter to deeply explore concepts of impulse response and frequency response, and to lay the groundwork for much more sophisticated musical instrument synthesis done in the next lab. The "sewer pipe" effect of a comb filter is distinctly heard, and the students are asked to explain the effect in physical terms by considering sound propagation in a cylindrical pipe. The comb filter is analyzed as a feedback system, making the connection to the virtual pet.

The lab again uses Simulink, this time for discrete-time processing. Discrete-time processing is not the best part of Simulink, so some operations are awkward. Moreover, the blocks in the block libraries that support discrete-time processing are not well organized. It can be difficult to discover how to do something as simple as an N-sample delay or an impulse source. The lab has to identify the blocks that the students need, which again gives it a more "cookbook-like" flavor. The students cannot be expected to wade through the extensive library of blocks, most of which will seem utterly incomprehensible.

Week 12 – Fourier Transforms. We relate frequency response and convolution, building the bridge between time and frequency domain views of systems. We introduce the DTFT and the continuous-time Fourier transform and derive various properties. These transforms are described as generalizations of the Fourier series where the signal need not be be periodic.

There is no lab exercise in this week, to allow time to prepare for the second midterm.

Week 13 – Sampling and Aliasing. We discuss sampling and aliasing as a major application of Fourier analysis techniques. Emphasis is on intuitive understanding of aliasing and its relationship to the periodicity of the DTFT. The Nyquist-Shannon sampling theorem is stated and related to this intuition, but its proof is not emphasized.

The purpose of the lab (C.9) is to experiment with models of a plucked string instrument, using it to deeply explore concepts of impulse response, frequency response, and spectrograms. The methods discussed in this lab were invented by Karplus and Strong [1]. The design of the lab itself was inspired by the excellent book of Steiglitz [5].

The lab uses Simulink, modifying the comb filter of the previous lab in three ways. First, the comb filter is initialized with random state, leveraging the concept of zero-input state response, studied previously with state-space models. Then it adds a lowpass filter to the feedback loop to create a dynamically varying spectrum, and it uses the spectrogram analysis developed in previous labs to show the effect. Finally, it adds an allpass filter to the feedback loop to precisely tune the resulting sound by adjusting the resonant frequency.

Week 14 – Filter Design. This week begins a review that focuses on how to apply the techniques of the course in practice. Filter design is considered with the objective of illustrating how frequency response applies to real problems, and with the objective of enabling educated use of filter design software. The modem startup sequence example is considered again in some detail, zeroing in on detection of the answer tone to illustrate design tradeoffs.

The purpose of this lab (C.10) is to use frequency domain concepts to study amplitude modulation. This is motivated, of course, by talking about AM radio, but also about digital communication systems, including digital cellular telephones, voiceband data modems, and wireless networking devices.

The students are given the following problem scenario:

> Assume we have a signal that contains frequencies in the range of about 100 to 300 Hz, and we have a channel that can pass frequencies from 700 to 1300 Hz. The task is to modulate the first signal so that it lies entirely within the channel passband, and then to demodulate to recover the original signal.

The test signal is a chirp. The frequency numbers are chosen so that every signal involved, even the demodulated signal with double frequency terms, is well within the audio range at an 8 kHz sample rate. Thus, students can reinforce the visual spectral displays with sounds that illustrate clearly what is happening.

A secondary purpose of this lab is to gain a working (users) knowledge of the FFT algorithm. In fact, they get enough information to be able to fully understand the algorithm that they were previously given to compute discrete Fourier series coefficients.

In this lab, the students also get an introductory working knowledge of filter design. They construct a specification and a filter design for the filter that eliminates the double frequency terms. This lab requires the Signal Processing Toolbox of Matlab for filter design.

Week 15 – Comprehensive Examples. This week develops applications that combine techniques of the course. The precise topics depend on the interests and expertise of the instructors, but we have specifically covered the following:

- Speech analysis and synthesis, using a historical Bell Labs recording of the Voder and Vocoder from 1939 and 1940 respectively, and explaining how the methods illustrated there (parametric modeling) are used in today's digital cellular telephones.
- Digital audio, with emphasis on encoding techniques such as MP3. Psychoacoustic concepts such as perceptual masking are related to the frequency domain ideas in the course.
- Vehicle automation, with emphasis on feedback control systems for automated highways. The use of discrete magnets in the road and sensors on the vehicles provides a superb illustration of the risks of aliasing.

The purpose of this lab (C.11) is to study the relationship between discrete-time and continuous-time signals by examining sampling and aliasing. Of course, a computer cannot directly deal with continuous-time signals. So instead, we construct discrete-time signals that are defined as samples of continuous-time signals, and then operate entirely on them, downsampling them to get new signals with lower sample rates, and upsampling them to get signals with higher sample rates. The upsampling operation is used to illustrate oversampling, as commonly used in digital audio players such as compact disk players. Once again, the lab is carefully designed so that all phenomena can be heard.

Discussion

The first few times we offered this course, automata appeared after frequency domain concepts. The new ordering, however, is far better. In particular, it introduces mathematical concepts gradually.

Specifically, the mathematical concepts on which the course relies are, sets and functions, matrix multiplication, complex numbers, and series and integrals. In particular, note that although students need to be comfortable with matrix multiplication, most of linear algebra is not required. We never mention an eigenvalue nor a matrix inverse, for example. The calculus required is also quite simple. The few exercises in the text that require calculus provide any integration formulas that a student might otherwise look up. Although series figure prominently, we only lightly touch on convergence, raising but not resolving the issue.

Some instructors may be tempted to omit the material on automata. We advise strongly against this. First, it gets students used to formally characterizing signals and systems in the context of a *much simpler* framework than linear systems. Most students find this material quite easy. Moreover, the methods apply much more broadly than frequency domain analysis, which applies primarily to LTI systems. **Most systems are not LTI.** Thus, inclusion of this material properly reflects the breadth of electrical engineering, which includes such specialties as data networks, which have little to with LTI systems. Even in specializations that heavily leverage frequency domain concepts, such as signal processing and communications, practitioners find that a huge fraction of their design effort deals with control logic and software-based services. Regrettably, classically trained electrical engineers harbor the misapprehension that these parts of their work are not compatible with rigor. This is wrong.

Notation

The notation we use is somewhat unusual when compared to standard notation in the vast majority of texts on signals and systems. However, we believe that the standard notation is seriously flawed. As a community, we have been able to get away with it for many years because signals and systems dealt only with continuous-time LTI systems. But to be useful, the discipline must be much broader now. Our specific complaints about the standard notation include:

Domains and Ranges

It is all too common to use the form of the argument of a function to define the function. For example, $x(n)$ is a discrete-time signal, while $x(t)$ is a continuous-time signal. This leads to mathematical nonsense like the $x(n) = x(nT)$ to define sampling. Similarly, many authors use ω for frequency in radians per second (unnormalized) and Ω for frequency in radians per sample (normalized). This means that $X(\Omega) \neq X(\omega)$ even when $\Omega = \omega$. The same problem arises when using the form $X(j\omega)$ for the continuous-time Fourier transform and $X(e^{j\omega})$ for the discrete-time Fourier transform. Worse, these latter forms are used specifically to establish the relationship to the Laplace and Z transforms. So $X(j\omega) = X(s)$ when $s = j\omega$, but $X(j\omega) \neq X(e^{j\omega})$ when $e^{j\omega} = j\omega$.

The intent in using the form of the argument is to indicate what the domain of the function is. However, the form of the argument is not the best way to do this. Instead, we treat the domain of a function as an integral part of its definition. Thus, for example, a discrete-time (real-valued) signal is a function $x\colon$ *Ints* $\to$ *Reals*, and it has a discrete-time Fourier transform that is also a function

$X\colon \textit{Reals} \to \textit{Comps}$. The DTFT itself is a function whose domain and range are sets of functions

$$\textit{DTFT}\colon [\textit{Ints} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Comps}].$$

Thus, we can write $X = \textit{DTFT}(x)$.

Functions as Values

Most texts call the expression $x(t)$ a function. A better interpretation is that $x(t)$ is an element in the range of the function x. The difficulty with the former interpretation becomes obvious when talking about systems. Many texts pay lip service to the notion that a system is a function by introducing a notation like $y(t) = T(x(t))$. This makes no distinction between the value of the function at t and the function y itself.

Why does this matter? Consider our favorite type of system, an LTI system. We write $y(t) = x(t) * h(t)$ to indicate convolution. Under any reasonable interpretation of mathematics, this would seem to imply that $y(t-\tau) = x(t-\tau) * h(t-\tau)$. But it is not so! How is a student supposed to conclude that $y(t-2\tau) = x(t-\tau) * h(t-\tau)$? This sort of sloppy notation could easily undermine the students' confidence in mathematics.

In our notation, a function is the element of a set of functions, just as its value for a given element in the domain is an element of its range. Convolution is a function whose domain is the cross product of two sets of functions. Continuous-time convolution, for example, is

$$\begin{aligned} \textit{Convolution} &: [\textit{Reals} \to \textit{Reals}] \times [\textit{Reals} \to \textit{Reals}] \\ &\to [\textit{Reals} \to \textit{Reals}]. \end{aligned}$$

We then introduce the notation $*$ as a shorthand,

$$x * y = \textit{Convolution}(x, y),$$

and define the convolution function by

$$(x * y)(t) = \int\limits_{-\infty}^{\infty} x(\tau) y(t-\tau) d\tau.$$

Note the careful parenthesization.

A major advantage of our notation is that it easily extends beyond LTI systems to the sorts of systems that inevitably arise in any real world application. For example, the events generated by the buttons of an audio component are a signal given as a function,

$$\textit{Commands}\colon \textit{Nats} \to \{\textit{Rec}, \textit{Play}, \textit{Stop}, \textit{FastFwd}, \textit{Rewind}\},$$

where *Nats* is the set of natural numbers. This is now a signal! With traditional notation, it is a whole new animal.

Names of Functions

We have chosen to use long names for functions and variables when they have a concrete interpretation. Thus, instead of x we might use *Sound*. This follows a long-standing tradition in software, where readability is considerably improved by long names. By giving us a much richer set of names to use, this helps us avoid some of the pitfalls we cite above. For example, to define sampling of an audio signal, we might write

$$\mathit{SampledSound} = \mathit{Sampler}_T(\mathit{Sound}).$$

It also helps bridge the gap between realizations of systems (which are often software) and their mathematical models. How to manage and understand this gap is a major theme of our approach.

Structure and Interpretation of Signals and Systems

Chapter 1

Signals and Systems

Signals convey information. Systems transform signals. This book is about developing a deeper understanding of both. We gain this understanding by dissecting their structure (their **syntax**) and by examining their interpretation (their **semantics**). For systems, we look at the relationship between the input and output signals (this relationship is a **declarative** description of the system) and the procedure for converting an input signal into an output signal (this procedure is an **imperative** description of the system).

A sound is a signal. We leave the physics of sound to texts on physics, and instead, show how a sound can be usefully decomposed into components that themselves have meaning. A musical chord, for example, can be decomposed into a set of notes. An image is a signal. We do not discuss the biophysics of visual perception, but instead show that an image can be usefully decomposed. We can use such decomposition, for example, to examine what it means for an image to be sharp or blurred, and thus to determine how to blur or sharpen an image.

Signals can be more abstract (less physical) than sound or images. They can be, for example, a sequence of commands or a list of names. We develop models for such signals and the systems that operate on them, such as a system that interprets a sequence of commands from a musician and produces a sound.

One way to get a deeper understanding of a subject is to formalize it, to develop mathematical models. Such models admit manipulation with a level of confidence not achievable with less formal models. We know that if we follow the rules of mathematics, then a transformed model still relates strongly to the original model. There is a sense in which mathematical manipulation preserves "truth" in a way that is elusive with almost any other intellectual manipulation of a subject. We can leverage this truth-preservation to gain confidence in the design of a system, to extract hidden information from a signal, or simply to gain insight.

Mathematically, we model both signals and systems as functions. A **signal** is a function that maps a domain, often time or space, into a range, often a physical measure such as air pressure or light intensity. A **system** is a function that maps signals from its domain — its input signals — into signals in its range — its output signals. The domain and the range are both sets of signals (**signal spaces**). Thus, systems are functions that operate on functions.

We use the mathematical language of sets and functions to make our models unambiguous, precise, and manipulable. This language has its own notation and rules, which are reviewed in appendix A. Depending on the situation, we represent physical quantities such as time, voltage, current, light intensity, air pressure, or the content of a memory location by variables that range over appropriate sets. For example, discrete time may be represented by a variable $n \in \mathit{Nats}$, the natural numbers, or $n \in \mathit{Ints}$, the integers; continuous time may be represented by a variable $t \in \mathit{Reals}_+$, the non-negative real numbers, or $t \in \mathit{Reals}$, the real numbers. Light intensity may be represented by a continuous variable $x \in [0, I]$, a range of real numbers from zero to I, where I is some maximum value of the intensity; a variable in a logic circuit may be represented as $x \in \mathit{Bin}$, the binary digits. A binary file is an element of Bin^*, the set of sequences of binary digits. A computer name such as `cory.eecs.Berkeley.EDU` may be assigned to a variable in $Char^*$, the set of sequences of characters.

1.1 Signals

Signals are functions that carry information, often in the form of temporal and spatial patterns. These patterns may be embodied in different media; radio and broadcast TV signals are electromagnetic waves, and images are spatial patterns of light intensities of different colors. In digital form, images and video are bit strings. Sensors of physical quantities (such as speed, temperature, or pressure) often convert those quantities into electrical voltages, which are then often converted into digital numbers for processing by a computer. In this text we will study systems that store, manipulate, and transmit signals.

In this section we study signals that occur in human and machine perception, in radio and TV, in telephone and computer networks, and in the description of physical quantities that change over time or over space. The most common feature of these signals is that their domains are sets representing time and space. However, we also study signals that are represented as sequences of symbols, where position within the sequence has no particular association with the physical notions of time or space. Such signals are often used to represent sequences of commands or sequences of events.

We will model signals as functions that map a domain (a set) into a range (another set). Our interest for now is to understand through examples how to select the domain and range of signals and how to visualize them. To fully describe a signal, we need to specify not only its domain and range, but also the rules by which the function assigns values. Because the domain and range are often infinite sets, specification of the rules is rarely trivial. Much of the emphasis of subsequent chapters is on how to characterize these functions.

1.1.1 Audio signals

Our ears are sensitive to sound, which is physically just rapid variations in air pressure. Thus sound can be represented as a function

$$\boxed{\mathit{Sound} \colon \mathit{Time} \to \mathit{Pressure}}$$

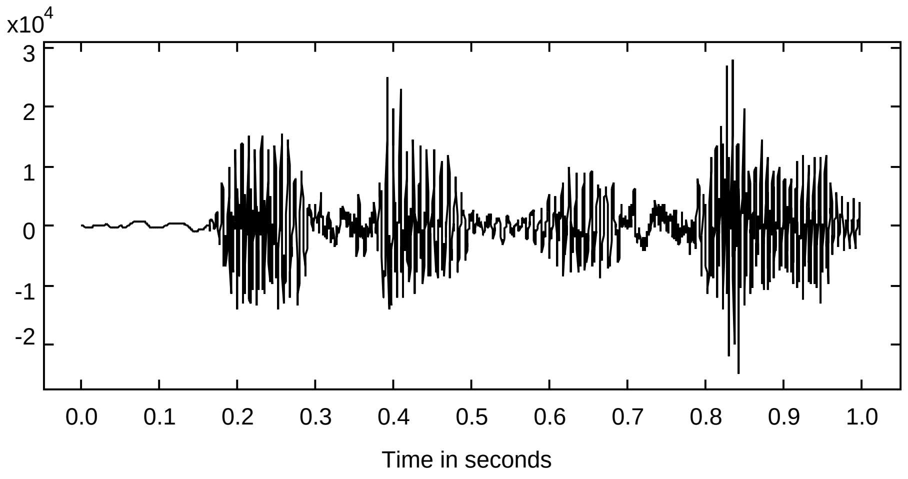


Figure 1.1: Waveform of a speech fragment.

where *Pressure* is a set consisting of possible values of air pressure, and *Time* is a set representing the time interval over which the signal lasts.[*]

Example 1.1: For example, a one-second segment of a voice signal is a function of the form

$$\textit{Voice}\colon [0,1] \to \textit{Pressure},$$

where $[0,1]$ represents one second of time. An example of such a function is plotted in figure 1.1. Such a plot is often called a **waveform**.

In figure 1.1, the vertical axis does not directly represent air pressure. This is obvious because air pressure cannot be negative. In fact, the possible values of the function *Voice* as shown in figure 1.1 are 16-bit integers, suitable for storage in a computer. Let us call the set of 16-bit integers $\textit{Ints16} = \{-32768, ..., 32767\}$. The audio hardware of the computer is responsible for converting members of the set *Ints16* into air pressure.

Notice also that the signal in figure 1.1 varies over positive and negative values, averaging approximately zero. But air pressure cannot be negative. It is customary to normalize the representation of sound by removing (subtracting) the ambient air pressure (about 100,000 newtons per square meter) from the range. Our ears, after all, are not sensitive to constant ambient air pressure. Thus, we take $\textit{Pressure} = \textit{Reals}$, the real numbers, where negative pressure means a drop in pressure relative to ambient air pressure. The numbers in the computer representation, *Ints16*, are a subset of the *Reals*. The units of air pressure in this representation are arbitrary, so to convert to units of newtons per square meter, we would need to multiply these numbers by some constant. The value of this constant depends on the audio hardware of the computer and on its volume setting.

[*]For a review of the notation of sets and functions, see appendix A.

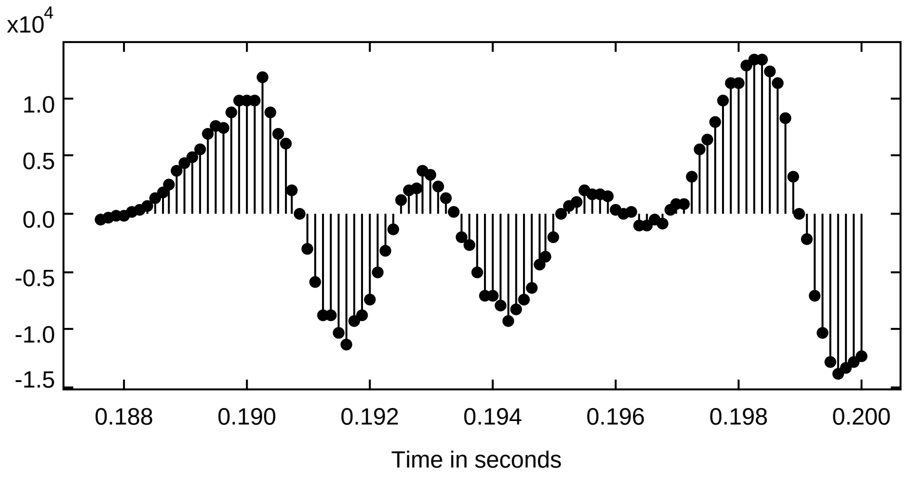


Figure 1.2: Discrete-time representation of a speech fragment.

The horizontal axis in figure 1.1 suggests that time varies continuously from zero to 1.0. However, a computer cannot directly handle such a continuum. The sound is represented not as a continuous waveform, but rather as a list of numbers (for voice-quality audio, 8,000 numbers for every second of speech).[†] A close-up of a section of the speech waveform is shown in figure 1.2. That plot shows 100 data points (called **samples**). For emphasis, that plot shows a dot for each sample rather than a continuous curve, and a stem connecting the dot to the horizontal axis. Such a plot is called a **stem plot**. Since there are 8,000 samples per second, the 100 points in figure 1.2 represent 100/8,000 seconds, or 12.5 milliseconds of speech.

Such signals are said to be **discrete-time signals** because they are defined only at discrete points in time. A discrete-time one-second voice signal in a computer is a function

$$\boxed{\mathit{ComputerVoice}\colon \mathit{DiscreteTime} \to \mathit{Ints16},}$$

where $\mathit{DiscreteTime} = \{0, 1/8000, 2/8000, \dots, 8000/8000\}$ is the set of sampling times.

By contrast, **continuous-time signals** are functions defined over a continuous interval of time (technically, a continuum in the set *Reals*). The audio hardware of the computer is responsible for converting the *ComputerVoice* function into a function of the form $\mathit{Sound}\colon \mathit{Time} \to \mathit{Pressure}$. That hardware, which converts an input signal into a different output signal, is a system.

Example 1.2: The sound emitted by a precisely tuned and idealized 440 Hz tuning fork over the infinite time interval $\mathit{Reals} = (-\infty, \infty)$ is the function

$$\mathit{PureTone}\colon \mathit{Reals} \to \mathit{Reals},$$

where the time-to-(normalized) pressure assignment is[‡]

$$\forall\, t \in \mathit{Reals}, \quad \mathit{PureTone}(t) = P \sin(2\pi \times 440t),$$

[†]In a compact disc (CD), there are 44,100 numbers per second of sound per stereo channel.
[‡]If the notation here is unfamiliar, see appendix A.

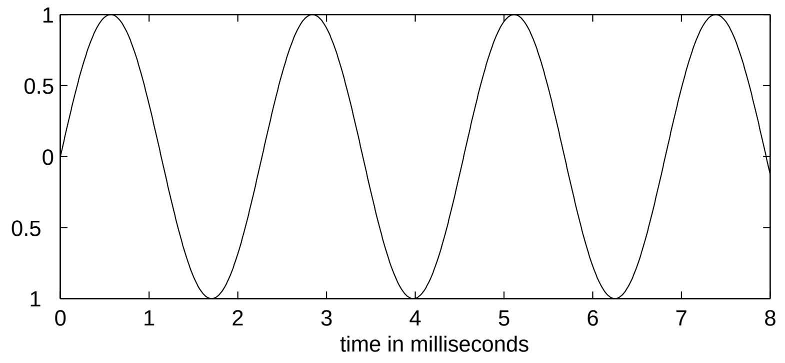


Figure 1.3: Portion of the graph of a pure tone with frequency 440 Hz.

> Here, P is the **amplitude** of the sinusoidal signal *PureTone*. It is a real-valued constant. Figure 1.3 is a graph of a portion of this pure tone (showing only a subset of the domain, *Reals*). In the figure, $P = 1$.

The number 440 in this example is the **frequency** of the sinusoidal signal shown in figure1.3, in units of **cycles per second** or **Hertz**, abbreviated **Hz**.[§] It simply asserts that the sinusoid completes 440 cycles per second. Alternatively, it completes one cycle in 1/440 seconds or about 2.3 milliseconds. The time to complete one cycle, 2.3 milliseconds, is called the **period**.

The *Voice* signal in figure 1.1 is much more irregular than *PureTone* in figure 1.3. An important theorem, which we will study in subsequent chapters, says that, despite its irregularity, a function like *Voice* is a sum of signals of the form of *PureTone*, but with different frequencies. A sum of two pure tones of frequencies, say 440 Hz and 660 Hz, is the function *SumOfTones*: *Reals* → *Reals* given by

$$\forall\, t \in \mathit{Reals}, \quad \mathit{SumOfTones}(t) = P_1 \sin(2\pi \times 440t) + P_2 \sin(2\pi \times 660t)$$

Notice that summing two signals amounts to adding the values of their functions at each point in the domain. Of course, two signals can be added only if they have the same domain and range, and the range must be such that addition is defined. The two components are shown in figure1.4. Notice that at any point on the horizontal axis, the value of the sum is simply the addition of the values of the two components.

1.1.2 Images

We see because the eye is sensitive to impulses of light (photons). If the image is a grayscale picture on a 11×8.5 inch sheet of paper, the picture is represented by a function

$$\mathit{Image}\colon [0, 11] \times [0, 8.5] \to [0, B_{max}], \tag{1.1}$$

[§]The unit of frequency called Hertz is named after physicist Heinrich Rudolf Hertz (1857-94), for his research in electromagnetic waves.

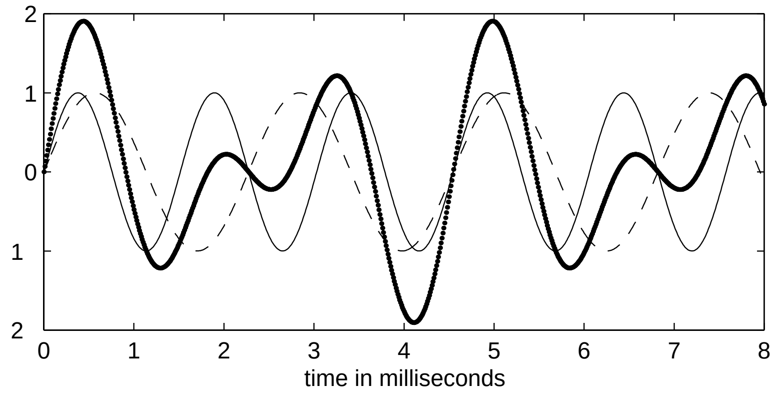


Figure 1.4: Sum of two pure tones, one at 440 Hz and the other at 660 Hz.

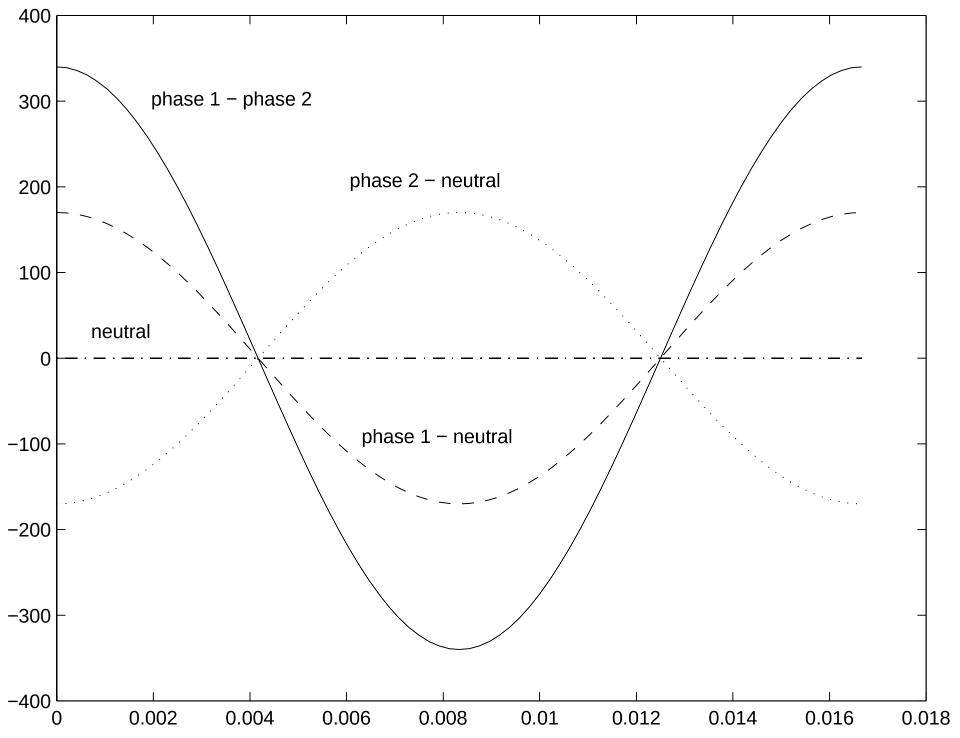


Figure 1.5: The voltages between the two hot wires and the neutral wire and between the two hot wires in household electrical power in the U.S.

Probing further: Household electrical power

In the U.S., household current is delivered on three wires, a **neutral wire** and two **hot wires**. The voltage between either hot wire and the neutral wire is around 110 to 120 volts, RMS (root mean square). The voltage between the two hot wires is around 220 to 240 volts, RMS. The higher voltage is used for appliances that need more power, such as air conditioners. Here, we examine exactly how this works.

The voltage between the hot wires and the neutral wire is sinusoidal with a frequency of 60 Hz. Thus, for one of the hot wires, it is a function $x\colon \mathit{Reals} \to \mathit{Reals}$ where the domain represents time and the range represents voltage, and

$$\forall\, t \in \mathit{Reals}, \quad x(t) = 170\cos(60 \times 2\pi t).$$

This 60 Hertz sinusoidal waveform completes one cycle in a period of $P = 1/60$ seconds. Why is the amplitude 170, rather than 120? Because the 120 voltage is **RMS** (**root mean square**). That is,

$$\mathit{voltage}_{RMS} = \sqrt{\frac{\int_0^P x^2(t)dt}{P}},$$

the square root of the average of the square of the voltage. This evaluates to 120.

The voltage between the second hot wire and the neutral wire is a function $y\colon \mathit{Reals} \to \mathit{Reals}$ where

$$\forall\, t \in \mathit{Reals}, \quad y(t) = -170\cos(60 \times 2\pi t) = -x(t).$$

It is the negative of the other voltage at any time t. This sinusoidal signal is said to have a **phase shift** of 180 degrees, or π radians, compared to the first sinusoid. Equivalently, it is said to be 180 degrees **out of phase**.

We can now see how to get the higher voltage for power-hungry appliances. We simply use the two hot wires rather than one hot wire and the neutral wire. The voltage between the two hot wires is the difference, a function $z\colon \mathit{Reals} \to \mathit{Reals}$ where

$$\forall\, t \in \mathit{Reals}, \quad z(t) = x(t) - y(t) = 340\cos(60 \times 2\pi t).$$

This corresponds to 240 volts RMS. A plot is shown in figure 1.5.

The neutral wire should not be confused with the ground wire in a three-prong plug. The ground wire is not connected to electrical company facilities. The neutral wire is. The ground wire is typically connected to the plumbing or some metallic conductor that goes underground. It is a safety feature to allow current to flow into the earth rather than, say, through a person.

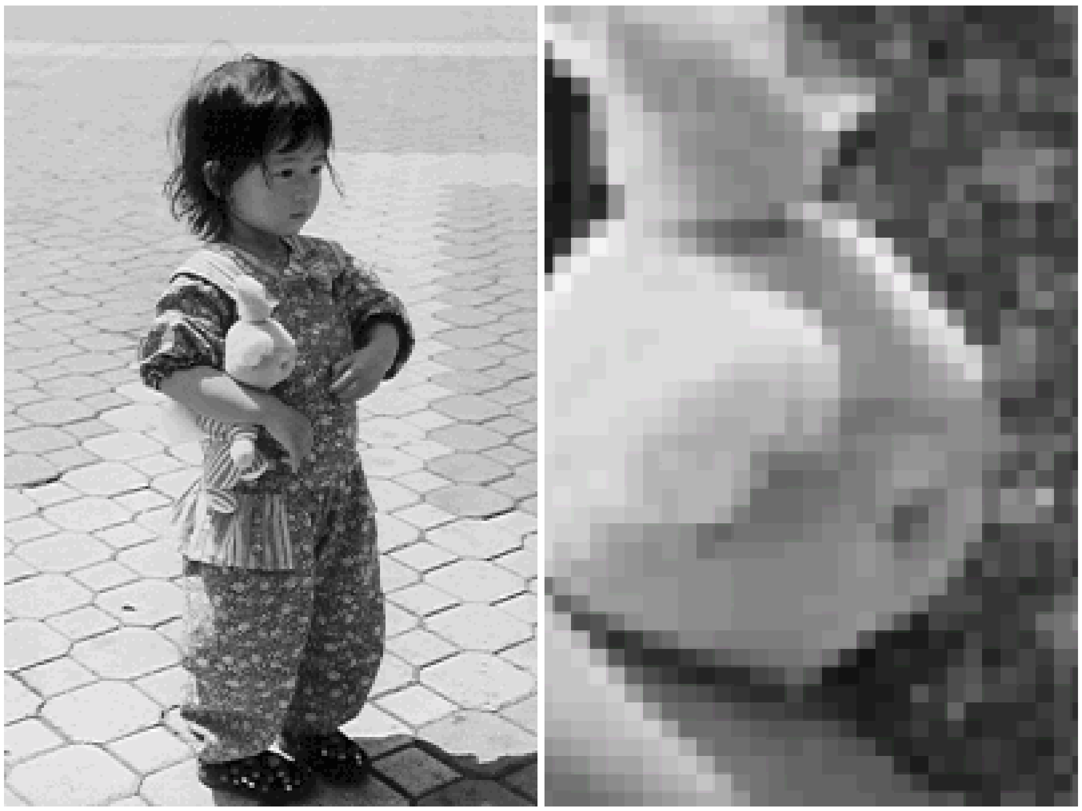


Figure 1.6: Grayscale image on the left, and its enlarged pixels on the right.

where B_{max} is the maximum grayscale value (0 is black and B_{max} is white). The set $[0, 11] \times [0, 8.5]$ defines the space of the sheet of paper.[¶] More generally, a grayscale image is a function

$$\boxed{\textit{Image}\colon \textit{VerticalSpace} \times \textit{HorizontalSpace} \to \textit{Intensity},}$$

where $\textit{Intensity} = [\textit{black}, \textit{white}]$ is the intensity range from *black* to *white* measured in some scale. An example is shown in figure 1.6.

For a color picture, the reflected light is sometimes measured in terms of its RGB values (i.e. the magnitudes of the red, green, and blue colors), and so a color picture is represented by a function

$$\boxed{\textit{ColorImage}\colon \textit{VerticalSpace} \times \textit{HorizontalSpace} \to \textit{Intensity}^3.}$$

The RGB values assigned by *ColorImage* at any point (x, y) in its domain is the triple $(r, g, b) \in \textit{Intensity}^3$ given by

$$(r, g, b) = \textit{ColorImage}(x, y).$$

Different images will be represented by functions with different spatial domains (the size of the image might be different), different ranges (we may consider a more or less detailed way of repre-

[¶]Again, see appendix A if this notation is unfamiliar.

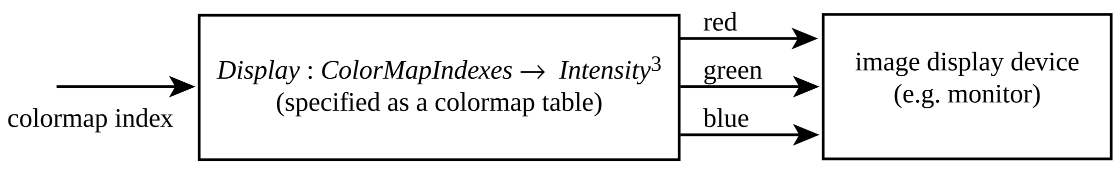

colormap index
Display : *ColorMapIndexes* → *Intensity*³
(specified as a colormap table)
red
green
blue
image display device
(e.g. monitor)

Figure 1.7: In a computer representation of a color image that uses a colormap, pixel values are elements of the set *ColorMapIndexes*. The function *Display* converts these indexes to an RGB representation.

senting light intensity and color than grayscale or RGB values), and differences in the assignment of color values to points in the domain.

Since a computer has finite memory and finite wordlength, an image is stored by discretizing both the domain and the range. So, for example, your computer may represent an image by storing a function of the form

$$\mathit{ComputerImage}\colon \mathit{DiscreteVerticalSpace} \times \mathit{DiscreteHorizontalSpace} \to \mathit{Ints8}$$

where

$$\begin{aligned} \mathit{DiscreteVerticalSpace} &= \{1, 2, \cdots, 300\}, \\ \mathit{DiscreteHorizontalSpace} &= \{1, 2, \cdots, 200\}, \text{and} \\ \mathit{Ints8} &= \{0, 1, \cdots, 255\}. \end{aligned}$$

It is customary to say that *ComputerImage* stores 300×200 pixels, where a **pixel** is an individual picture element. The value of a pixel is $\mathit{ComputerImage}(\mathit{row}, \mathit{column}) \in \mathit{Ints8}$ where $\mathit{row} \in \mathit{DiscreteVerticalSpace}$, $\mathit{column} \in \mathit{DiscreteHorizontalSpace}$. In this example the range *Ints8* has 256 elements, so in the computer these elements can be represented by an 8-bit word (hence the name *Ints8*). An example of such an image is shown in figure 1.6, where the right-hand version of the image is magnified to show the discretization implied by the individual pixels.

A computer could store a color image in one of two ways. One way is to represent it as a function

$$\mathit{ColorComputerImage}\colon \mathit{DiscreteVerticalSpace} \times \mathit{DiscreteHorizontalSpace} \to \mathit{Ints8}^3 \tag{1.2}$$

so each pixel value is an element of $\{0, 1, \cdots, 255\}^3$. Such a pixel can be represented as three 8-bit words. A common method which saves memory is to use a **colormap**. Define the set $\mathit{ColorMapIndexes} = \{0, \cdots, 255\}$, together with a *Display* function,

$$\mathit{Display}\colon \mathit{ColorMapIndexes} \to \mathit{Intensity}^3. \tag{1.3}$$

Display assigns to each element of *ColorMapIndexes* the three (r, g, b) color intensities. This is depicted in the block diagram in figure 1.7. Use of a colormap reduces the required memory to store an image by a factor of three because each pixel can now be represented by a single 8-bit number. But only 256 colors can be represented in any given image. The function *Display* is typically represented in the computer as a lookup table (see lab C.2).

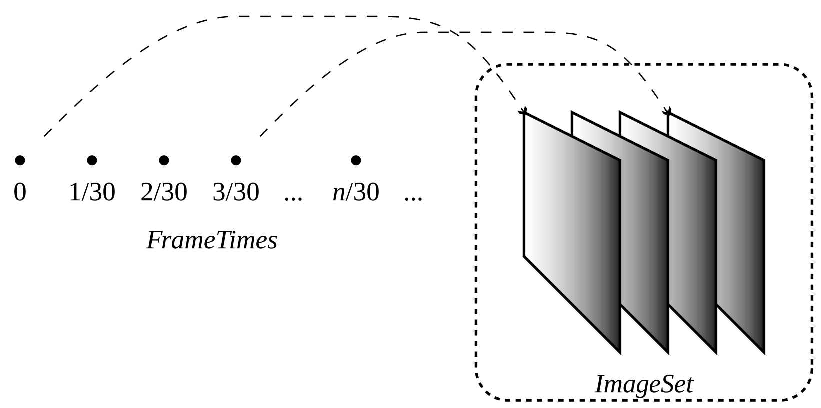


Figure 1.8: Illustration of the function *Video*.

1.1.3 Video signals

A **video** is a sequence of images displayed at a certain rate. NTSC video (the standard for analog video followed in the U.S.) displays 30 images (called **frames**) per second. Our eye is unable to distinguish between successive images displayed at this frequency, and so a TV broadcast appears to be continuously varying in time.

Thus the domain of a video signal is discrete time, *FrameTimes* $= \{0, 1/30, 2/30, \cdots\}$, and its range is a set of images, *ImageSet*. For **analog video**, each image in *ImageSet* is a function of the form

$$VideoFrame\colon DiscreteVerticalSpace \times HorizontalSpace \to Intensity^3.$$

An analog video signal is discretized in the vertical direction, but not in the horizontal direction.[‖] The image is composed of a set of horizontal lines called **scan lines**, where the intensity varies along the line. The horizontal lines are stacked vertically to form an image.

A video signal, therefore, is a function

$$Video\colon FrameTimes \to ImageSet. \tag{1.4}$$

For any time $t \in$ *FrameTimes*, the image $Video(t) \in$ *ImageSet* is displayed. The signal *Video* is illustrated in figure 1.8.

An alternative way of specifying a video signal is by the function *AltVideo* whose domain is a product set as follows:

$$AltVideo\colon FrameTimes \times DiscreteVerticalSpace \times HorizontalSpace \to Intensity^3.$$

[‖]This is actually somewhat of a simplification. Most analog video images are **interlaced**, meaning that successive frames use different sets for *DiscreteVerticalSpace* so that scan lines in one frame lie between the scan lines of the previous frame. Also, the range *Intensity*^{3} has a curious structure that ensures compatibility between black-and-white and color television sets.

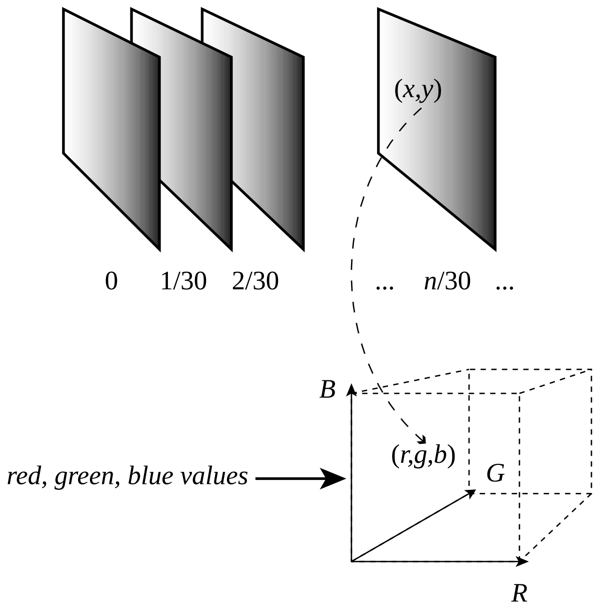


Figure 1.9: Illustration of the function *AltVideo*.

Similarly to figure 1.8, we can depict *AltVideo* as in figure 1.9. The RGB value assigned to a point (x, y) at time t is

$$(r, g, b) = \mathit{AltVideo}(t, x, y). \tag{1.5}$$

If the signals specified in (1.4) and (1.5) represent the same video, then for all $t \in \mathit{FrameTimes}$ and $(x, y) \in \mathit{DiscreteVerticalSpace} \times \mathit{HorizontalSpace}$,

$$(\mathit{Video}(t))(x, y) = \mathit{AltVideo}(t, x, y). \tag{1.6}$$

It is worth pausing to understand the notation used in (1.6). *Video* is a function of t, so $\mathit{Video}(t)$ is an element in its range *ImageSet*. Since elements in *ImageSet* themselves are functions, $\mathit{Video}(t)$ is a function. The domain of $\mathit{Video}(t)$ is the product set $\mathit{DiscreteVerticalSpace} \times \mathit{HorizontalSpace}$, so $(\mathit{Video}(t))(x, y)$ is the value of this function at the point (x, y) in its domain. This value is an element of $\mathit{Intensity}^3$. On the right-hand side of (1.6) *AltVideo* is a function of (t, x, y) and so $AltVideo(t, x, y)$ is an element in its range, $\mathit{Intensity}^3$. The equality (1.6) asserts that for all values of t, x, y the two sides are the same. On the left-hand side of (1.6) the parentheses enclosing $\mathit{Video}(t)$ are not necessary; we could equally well write $\mathit{Video}(t)(x, y)$. However, the parentheses improve readability.

1.1.4 Signals representing physical attributes

The change over time in the attributes of a physical object or device can be represented as functions of time or space.

Example 1.3: The position of an airplane can be expressed as

$$\textit{Position}\colon \textit{Time} \to \textit{Reals}^3,$$

where for all $t \in \textit{Time}$,

$$\textit{Position}(t) = (x(t), y(t), z(t))$$

is its position in 3-dimensional space at time t. The position and velocity of the airplane is a function

$$\textit{PositionVelocity}\colon \textit{Time} \to \textit{Reals}^6, \tag{1.7}$$

where

$$\textit{PositionVelocity}(t) = (x(t), y(t), z(t), v_x(t), v_y(t), v_z(t)) \tag{1.8}$$

gives its position and velocity at $t \in \textit{Time}$.

The position of the pendulum shown in the left panel of figure 1.10 is represented by the function

$$\theta\colon \textit{Time} \to [-\pi, \pi],$$

where $\theta(t)$ is the angle with the vertical made by the pendulum at time t.

The position of the upper and lower arms of a robot depicted in the right panel of figure 1.10 can be represented by the function

$$(\theta_u, \theta_l)\colon \textit{Time} \to [-\pi, \pi]^2,$$

where $\theta_u(t)$ is the angle at the elbow made by the upper arm with the vertical, and $\theta_l(t)$ is the angle made by the lower arm with the upper arm at time t. Note that we can regard (θ_u, θ_l) as a single function with range as the product set $[-\pi, \pi]^2$ or as two functions θ_u and θ_l each with range $[-\pi, \pi]$. Similarly, we can regard *PositionVelocity* in (1.7) as a single function with range *Reals*^{6} or as a collection of six functions, each with range *Reals*, as suggested by (1.8).

Example 1.4: The spatial variation of temperature over some volume of space can be represented by a function

$$\textit{AirTemp}\colon X \times Y \times Z \to \textit{Reals}$$

where $X \times Y \times Z \subset \textit{Reals}^3$ is the volume of interest, and $AirTemp(x, y, z)$ is the temperature at the point (x, y, z).

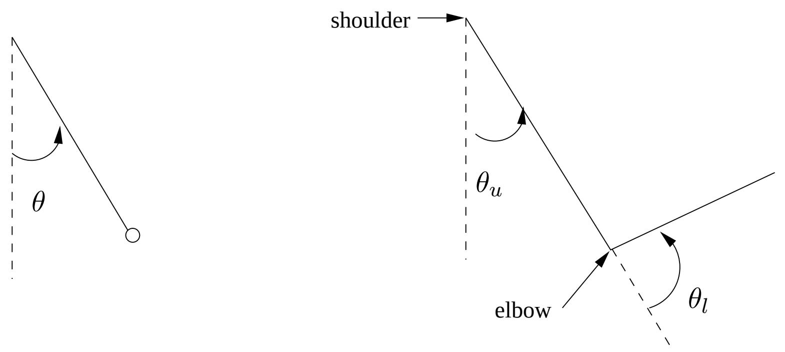


Figure 1.10: Position of a pendulum (left) and upper and lower arms of a robot (right).

1.1.5 Sequences

Above we studied examples in which temporal or spatial information is represented by functions of a variable representing time or space. The domain of time or space may be continuous as in *Voice* and *Image* or discrete as in *ComputerVoice* and *ComputerImage*.

In many situations, information is represented as **sequences** of symbols rather than as functions of time or space. These sequences occur in two ways: as a representation of **data** or as a representation of an **event stream**.

Examples of data represented by sequences are common. A file stored in a computer in binary form is a sequence of bits, or binary symbols, i.e. a sequence of 0's and 1's. A text, like this book, is a sequence of words. A sheet of music is a sequence of notes.

Example 1.5: Consider an N-bit long binary file,

$$b_1, b_2, \cdots, b_N,$$

where each $b_i \in \mathit{Bin} = \{0, 1\}$. We can regard this file as a function

$$\mathit{File}\colon \{1, 2, \cdots, N\} \to \mathit{Bin},$$

with the assignment $\mathit{File}(n) = b_n$ for every $n \in \{1, \cdots, N\}$.

If instead of *Bin* we take the range to be *EnglishWords*, then an N-word long English text is a function

$$\mathit{EnglishText}\colon \{1, 2, \cdots, N\} \to \mathit{EnglishWords}.$$

In general, data sequences are functions of the form

$$\boxed{\mathit{Data}\colon \mathit{Indices} \to \mathit{Symbols},} \tag{1.9}$$

where *Indices* $\subset$ *Nats*, where *Nats* is the set of natural numbers, is an appropriate index set such as $\{1, 2, \cdots, N\}$, and *Symbols* is an appropriate set of symbols such as *Bin* or *EnglishWords*.

One advantage of the representation (1.9) is that we can then interpret *Data* as a discrete-time signal, and so some of the techniques that we will develop in later chapters for those signals will automatically apply to data sequences. However, the domain *Indices* in (1.9) does not represent uniformly spaced instances of time. All we can say is that if m and n are in *Indices* with $m < n$, then the m-th symbol *Data*(m) occurs in the data sequence *before* the n-th symbol *Data*(n), but we cannot say how much time elapses between the occurrence of those two symbols.

The second way in which sequences arise is as representations of event streams. An **event stream** or **trace** is a record or log of the significant events that occur in a system of interest. Here are some everyday examples.

Example 1.6: When you call someone by phone, the normal sequence of events is

LiftHandset, HearDialTone, DialDigits, HearTelephoneRing, HearCalleeAnswer, $\cdots$

but if the other phone is busy, the event trace is

LiftHandset, HearDialTone, DialDigits, HearBusyTone, $\cdots$

When you send a file to be printed the normal trace of events is

CommandPrintFile, FilePrinting, PrintingComplete

but if the printer has run out of paper, the trace might be

CommandPrintFile, FilePrinting, MessageOutofPaper, InsertPaper, $\cdots$

Example 1.7: When you enter your car the starting trace of events might be

StartEngine, SeatbeltSignOn, BuckleSeatbelt, SeatbeltSignOff, $\cdots$

Thus event streams are functions of the form

$$\boxed{\mathit{EventStream}\colon \mathit{Indices} \to \mathit{EventSet}.}$$

We will see in chapter 3 that the behavior of finite state machines is best described in terms of event traces, and that systems that operate on event streams are often best described as finite state machines.

1.1.6 Discrete signals and sampling

Voice and *PureTone* are said to be continuous-time signals because their domain *Time* is a continuous interval of the form $[\alpha, \beta] \subset$ *Reals*. The domain of *Image*, similarly, is a continuous 2-dimensional rectangle of the form $[a, b] \times [c, d] \subset \mathit{Reals}^2$. The signals *ComputerVoice* and *ComputerImage* have domains of time and space that are discrete sets. *Video* is also a discrete-time signal, but in principle it could be a function of a space continuum. We can define a function *ComputerVideo* where all three sets that are composed to form the domain are discrete.

Discrete signals often arise from signals with continuous domains by **sampling**. We briefly motivate sampling here, with a detailed discussion to be taken up later. Continuous domains have an infinite number of elements. Even the domain $[0, 1] \subset \mathit{Time}$ representing a finite time interval has an infinite number of elements. The signal assigns a value in its range to each of these infinitely many elements. Such a signal cannot be stored in a finite digital memory device such as a computer or CD-ROM. If we wish to store, say, *Voice*, we must approximate it by a signal with a finite domain.

A common way to approximate a function with a continuous domain like *Voice* and *Image* by a function with a finite domain is by uniformly sampling its continuous domain.

Example 1.8: If we sample a 10-second long domain of *Voice*,

$$\mathit{Voice}\colon [0, 10] \to \mathit{Pressure},$$

10,000 times a second (i.e. at a frequency of 10 kHz) we get the signal

$$\mathit{SampledVoice}\colon \{0, 0.0001, 0.0002, \cdots, 9.9998, 9.9999, 10\} \to \mathit{Pressure}, \tag{1.10}$$

with the assignment

$$\mathit{SampledVoice}(t) = \mathit{Voice}(t), \text{ for all } t \in \{0, 0.0001, 0.0002, \cdots, 9.9999, 10\}. \tag{1.11}$$

Notice from (1.10) that uniform sampling means picking a uniformly spaced subset of points of the continuous domain $[0, 10]$.

In the example, the **sampling interval** or **sampling period** is 0.0001 sec, corresponding to a **sampling frequency** or **sampling rate** of 10,000 Hz. Since the continuous domain is 10 seconds long, the domain of *SampledVoice* has 100,000 points. A sampling frequency of 5,000 Hz would give the domain $\{0, 0.0002, \cdots, 9.9998, 10\}$, which has half as many points. The sampled domain is finite, and its elements are discrete values of time.

Notice also from (1.11) that the pressure assigned by *SampledVoice* to each time in its domain is the same as that assigned by *Voice* to the same time. That is, *SampledVoice* is indeed obtained by sampling the *Voice* signal at discrete values of time.

Figure 1.11 shows an exponential function $\mathit{Exp}\colon [-1, 1] \to \mathit{Reals}$ defined by

$$\mathit{Exp}(x) = e^x.$$

SampledExp is obtained by sampling with a sampling interval of 0.2. So its domain is

$$\{-1, -0.8, \cdots, 0.8, 1.0\}.$$

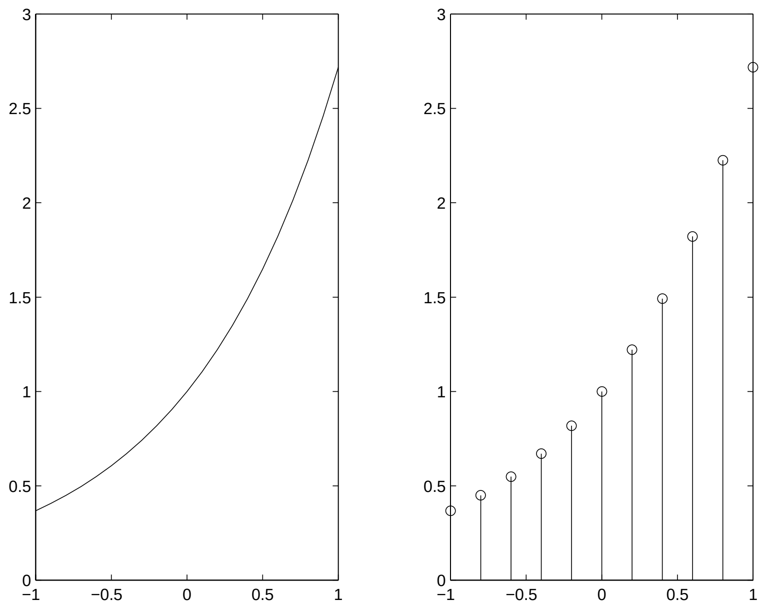


Figure 1.11: The exponential functions *Exp* and *SampledExp*, obtained by sampling with a sampling interval of 0.2.

The continuous domain of *Image* given by (1.1), which describes a grayscale image on an 8.5 by 11 inch sheet of paper, is the rectangle $[0, 11] \times [0, 8.5]$, representing the space of the page. In this case, too, a common way to approximate *Image* by a signal with finite domain is to sample the rectangle. Uniform sampling with a **spatial resolution** of say, 100 dots per inch, in each dimension, gives the finite domain $D = \{0, 0.01, \cdots, 8.49, 8.5\} \times \{0, 0.01, \cdots, 10.99, 11.0\}$. So the sampled grayscale picture is

$$\textit{SampledImage}\colon D \to [0, B_{max}]$$

with

$$\textit{SampledImage}(x, y) = \textit{Image}(x, y), \text{ for all } (x, y) \in D.$$

As mentioned before, each sample of the image is called a **pixel**, and the size of the image is often given in pixels. The size of your computer screen display, for example, may be 600×800 or 768×1024 pixels.

Sampling and approximation

Let f be a continuous-time function, and let *Sampledf* be the discrete-time function obtained by sampling f. Suppose we are given *Sampledf*, as, for example, in the left panel of figure1.12. Can we reconstruct or recover f from *Sampledf*? This question lies at the heart of digital storage and communication technologies. The general answer to this question tells us, for example, what audio quality we can obtain from a given discrete representation of a sound. The format for a compact disc is based on the answer to this question. We will discuss it in much detail in later chapters.

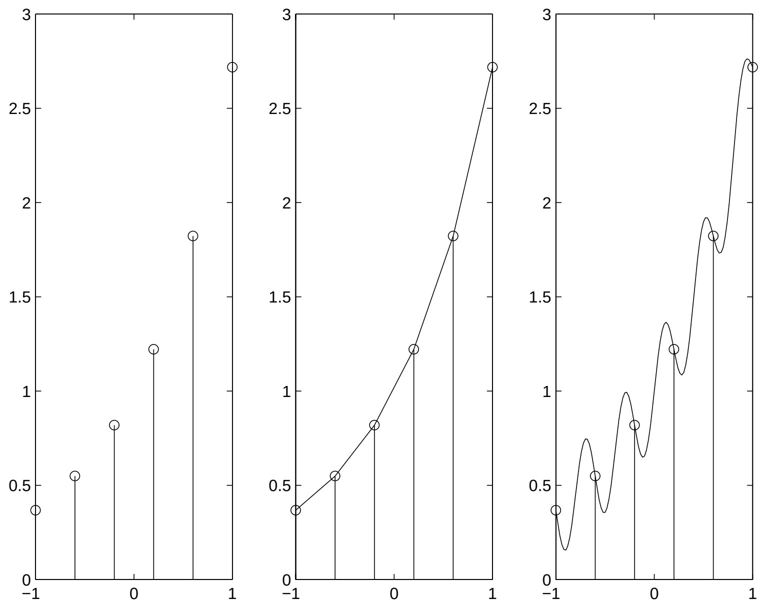


Figure 1.12: The discrete-time signal on the left is obtained by sampling the continuous-time signal in the middle or the one on the right.

For the moment, let us note that the short answer to the question above is no. For example, we cannot tell whether the discrete-time function in the left panel of figure 1.12 was obtained by sampling the continuous-time function in the middle panel or the function in the right panel. Indeed there are infinitely many such functions, and one must make a choice. One option is to connect the sampled values by straight line segments, as shown in the middle panel. Another choice is shown in the right panel. The choice made by your CD player is different from both of these, as explored further in chapter 10.

Similarly, an image like *Image* cannot be uniquely recovered from its sampled version *SampledImage*. Several different choices are commonly used.

Digital signals and quantization

Even though *SampledVoice* in example 1.8 has a finite domain, we may yet be unable to store it in a finite amount of memory. To see why, suppose that the range *Pressure* of the function *SampledVoice* is the continuous interval $[a, b]$. To represent every value in $[a, b]$ requires infinite **precision**. In a computer, where data are represented digitally as finite collections of bits, such precision would require an infinite number of bits for just one sample. But a finite digital memory has a finite wordlength in which we can store only a finite number of values. For instance, if a word is 8 bits long, it can have $2^8 = 256$ different values. So we must approximate each number in the range $[a, b]$ by one of 256 values. The most common approximation method is to **quantize** the signal. A common approach is to choose 256 uniformly-spaced values in the range $[a, b]$, and to approximate

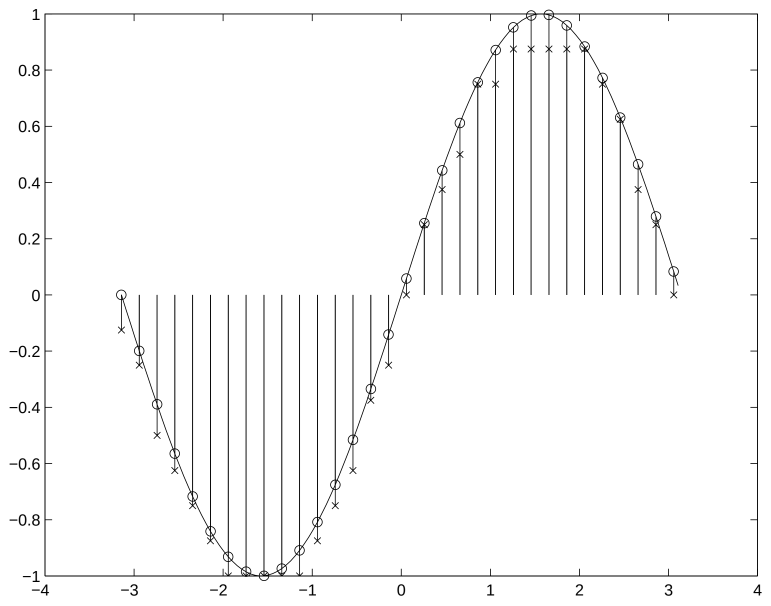


Figure 1.13: *PureTone* (continuous curve), *SampledPureTone* (circles), and *DigitalPureTone* signals (x's).

each value in $[a, b]$ by the one of these 256 values that is closest. An alternative approximation, called **truncation**, is to choose the largest of the 256 values that is less than or equal to the desired value.

Example 1.9: Figure 1.13 shows a *PureTone* signal, *SampledPureTone* obtained after sampling, and a quantized *DigitalPureTone* obtained using 4-bit or 16-level truncation. *PureTone* has continuous domain and continuous range, while *SampledPureTone* (depicted with circles) has discrete domain and continuous range, and *DigitalPureTone* (depicted with x's) has discrete domain and discrete range. Only the last of these can be precisely represented on a computer.

It is customary to call a signal with continuous domain and continuous range like *PureTone* an **analog signal**, and a signal with discrete domain and range, like $Digital$*PureTone*, a **digital signal**.

Example 1.10: In digital telephones, voice is sampled every 125μsec, or at a sampling frequency of 8,000 Hz. Each sample is quantized into an 8-bit word, or 256 levels. This gives an overall rate of $8,000 \times 8 = 64,000$ bits per second. The worldwide digital telephony network, therefore, is composed primarily of channels capable of carrying 64,000 bits per second, or multiples of this (so that multiple telephone channels can be carried together). In cellular phones, voice samples are further compressed to bit rates of 8,000 to 32,000 bits per second.

1.2 Systems

Systems are functions that transform signals. There are many reasons for transforming signals. A signal carries information. A transformed signal may carry the same information in a different way. For example, in a live concert, music is represented as sound. A recording system may convert that sound into a pattern of magnetic fields on a magnetic tape. The original signal, the sound, is difficult to preserve for posterity. The magnetic tape has a more persistent representation of the same information. Thus, **storage** is one of the tasks accomplished by systems.

A system may transform a signal into a form that is more convenient for **transmission**. Sound signals cannot be carried by the Internet. There is simply no physical mechanism in the Internet for transporting rapid variations in air pressure. The Internet provides instead a mechanism for transporting sequences of bits. A system must convert a sound signal into a sequence of bits. Such a system is called an **encoder** or **coder**. At the far end, of course, a **decoder** is needed to convert the sequence back into sound. When a coder and a decoder are combined into the same physical device, the device is often called a **codec**.

A system may transform a signal to hide its information so that snoops do not have access to it. This is called **encryption**. To be useful, we need matching **decryption**.

A system may **enhance** a signal by emphasizing some of the information it carries and deemphasizing some other information. For example, an **audio equalizer** may compensate for poor room acoustics by reducing the magnitude of certain low frequencies that happen to resonate in the room. In transmission, signals are often degraded by **noise** or distorted by physical effects in the transmission medium. A system may attempt to reduce the noise or reverse the distortion. When the signal is carrying digital information over a physical channel, the extraction of the digital information from the degraded signal is called **detection**.

Systems are also designed to **control** physical processes such as the heating in a room, the ignition in an automobile engine, the flight of an aircraft. The state of the physical process (room temperature, cylinder pressure, aircraft speed) is sensed. The sensed signal is processed to generate signals that drive actuators, such as motors or switches. Engineers design a system called the **controller** which, on the basis of the processed sensor signal, determines the signals that control the physical process (turn the heater on or off, adjust the ignition timing, change the aircraft flaps) so that the process has the desired behavior (room temperature adjusts to the desired setting, engine delivers more torque, aircraft descends smoothly).

Systems are also designed for **translation** from one format to another. For example, a command sequence from a musician may be transformed into musical sounds. Or the detection of risk of collision in an aircraft might be translated into control signals that perform evasive maneuvers.

1.2.1 Systems as functions

Consider a system S that transforms input signal x into output signal y. The system is a function, so $y = S(x)$. Suppose $x\colon D \to R$ is a signal with domain D and range R. For example, x might be a sound, $x\colon \mathit{Reals} \to \mathit{Pressure}$. The domain of S is the set X of all such sounds, which we write

$$\boxed{X = [D \to R] = \{x \mid x\colon D \to R\}} \tag{1.12}$$

This notation reads "X, also written $[D \to R]$, is the set of all x such that x is a function from D to R." This set is called a **signal space** or **function space**. A signal or function space is a set of all functions with a given domain and range.

Example 1.11: The set of all sound segments with duration [0,1] and range *Pressure* is written

$$[[0,1] \to \mathit{Pressure}].$$

Notice that square brackets are used for both a range of reals, as in $[0,1]$, and for a function space, as in $[D \to R]$, although obviously the meanings of these two notations are very different.

The set *ImageSet* considered in section 1.1.3 is the function space

$$\mathit{ImageSet} = [\mathit{DiscreteVerticalSpace} \times \mathit{HorizontalSpace} \to \mathit{Intensity}^3].$$

Since this is a set, we can define functions that use it as a domain or range, as we have done above with

$$\mathit{Video}\colon \mathit{FrameTimes} \to \mathit{ImageSet}.$$

Similarly, the set of all binary files of length N is

$$\mathit{BinaryFiles} = [\mathit{Indices} \to \mathit{Bin}].$$

where $\mathit{Indices} = \{1, \cdots, N\}$.

A system S is a function mapping a signal space into a signal space,

$$\boxed{S\colon [D \to R] \to [D' \to R'].}$$

Systems, therefore, are much like signals, except that their domain and range are both signal spaces. Thus, if $x \in [D \to R]$ and $y = S(x)$, then it must be that $y \in [D' \to R']$.

1.2.2 Telecommunications systems

We give some examples of systems that occur in or interact with the global telecommunications network. This network is unquestionably one of the most remarkable accomplishments of humankind. It is astonishingly complex, composed of hundreds of distinct corporations and linking billions of people. We often think of it in terms of its basic service, **POTS**, or plain-old telephone service. POTS is a voice service, but the telephone network is in fact a global, high-speed digital network that carries not just voice, but also video, images, and computer data, including much of the traffic in the Internet.

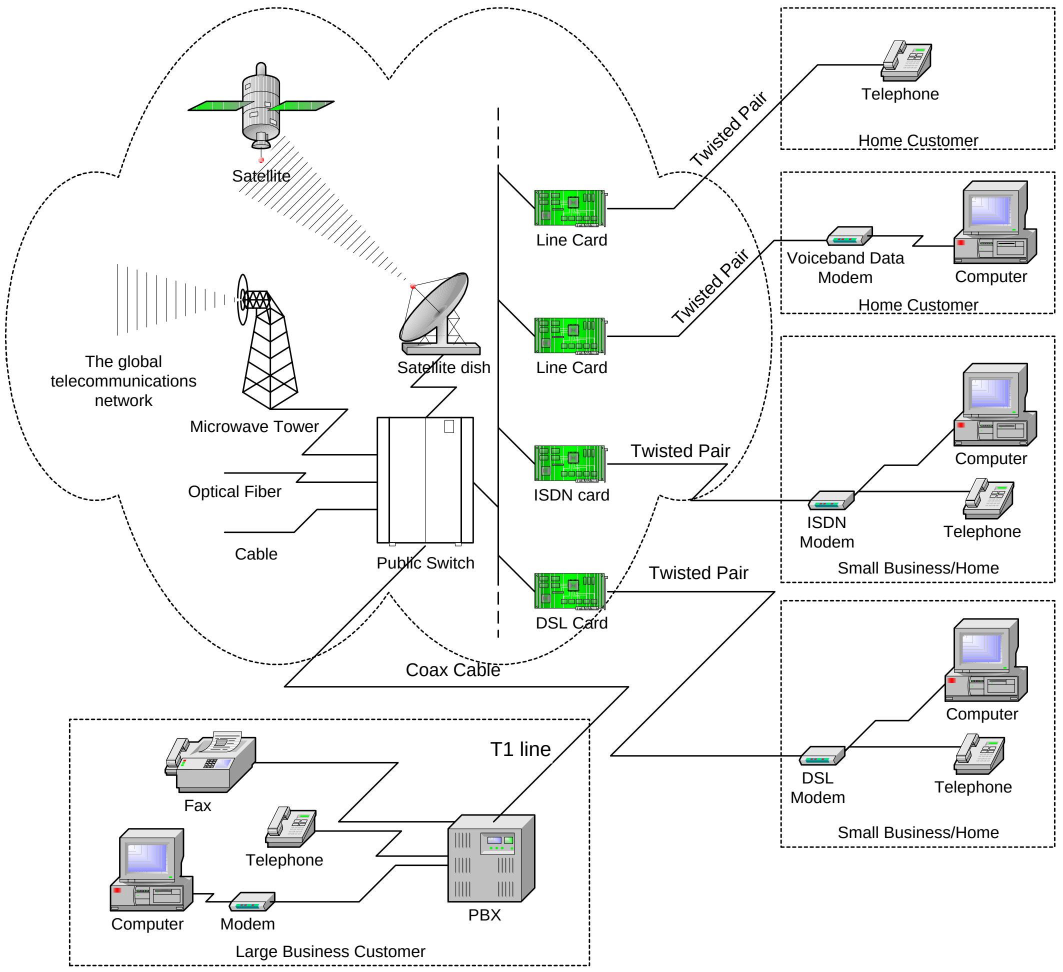


Figure 1.14: A portion of the global telecommunications network.

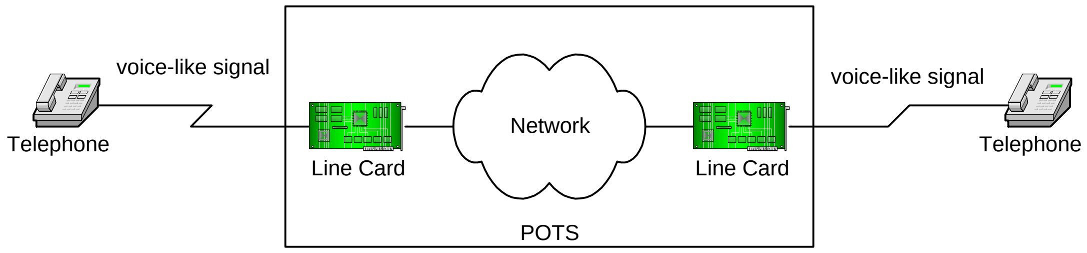


Figure 1.15: Abstraction of plain-old telephone service (POTS).

Figure 1.14 depicts a small portion of the global telecommunications network. POTS service is represented at the upper right, where a **twisted pair** of copper wires connects a central office to a home telephone. This twisted pair is called the **local loop** or **subscriber line**. At the central office, the twisted pair is connected to a **line card**, which usually converts the signal from the telephone immediately into digital form. The line card, in turn, is connected to a **switch**, which routes incoming and outgoing telephone connections. The Berkeley central office is located on Bancroft, between Oxford and Shattuck.

The digital representation of a voice signal, a sequence of bits, is routed through the telephone network. Usually it is combined with other bit sequences, which are other voices or computer data, and sent over high-speed links implemented with optical fiber, microwave radio, coaxial cable, or satellites.

Of course, a telephone conversation usually involves two parties, so the network delivers to the same line card a digital sequence representing the far-end speaker. That digital sequence is decoded and delivered to the telephone via the twisted pair. The line card, therefore, includes a codec.

The telephone itself, of course, is a system. It transforms the electrical signal that propagates down the twisted pair into a sound signal, and transforms a local sound signal into an electrical signal that can propagate down the twisted pair.

POTS can be abstracted as shown in figure 1.15. The entire network is reduced to a model that accepts an electrical representation of a voice signal and transports it to a remote telephone. In this abstraction, the digital nature of the telephone network is irrelevant. The system simply transports (and degrades somewhat) a voice signal.

DTMF

Even in POTS, not all of the information transported is voice. At a minimum, the telephone needs to be able to convey to the central office a telephone number in order to establish a connection. A telephone number is not a voice signal. It is intrinsically discrete. Since the system is designed to carry voice signals, one option is to convert the telephone number into a voice-like signal. A system is needed with the structure shown in figure 1.16. The block labeled "DTMF" is a system that transforms a sequence of numbers (coming from the keypad on the left) into a voice-like signal.

Probing further: Wireless communication

Recently, the telephone network has been freeing itself of its dependence on wires. **Cellular telephones**, which came into widespread use in the 1990s, use radio waves to connect a small, hand-held telephone to a nearby **base station**. The base station connects directly to the telephone network.

There are three major challenges in the design of cellular networks. First, radio spectrum is scarce. Frequencies are allocated by regulatory bodies, often constrained by international treaties. Finding frequencies for new technologies is difficult. Thus, wireless communication devices have to be extremely efficient in their use the available frequencies. Second, the power available to drive a cellular phone is limited. Cellular phones must operate for reasonably long periods of time using only small batteries that fit easily within the handset. Although battery technology has been improving, the power that these batteries can deliver severely limits the range of a cellular phone (how far it can be from a base station) and the processing complexity (the microprocessors in a cellular phone consume considerable power). Third, networking is complicated. In order to be able to route telephone calls to a cellular phone, the network needs to know where the phone is (or more specifically, which base station is closest). Moreover, the network needs to support phone calls in moving vehicles, which implies that a phone may move out of range of one base station and into the range of another during the course of a telephone call. The network must **hand off** the call seamlessly.

Although “radio telephones” have existed for a long time, particularly for maritime applications where wireline telephony is impossible, it was the cellular concept that made it possible to offer radio telephony to large numbers of users. The concept is simple. Radio waves propagating along the surface of the earth lose power approximately proportionally to the inverse of the fourth power of distance. That is, if at distance d meters from a transmitter your receive w watts of radio power, then at distance $2d$ you will receive approximately $w/2^4 = w/16$ watts of radio power. This fourth-power propagation loss was traditionally considered only to be a hindrance to wireless communication. It had to be overcome by greatly boosting the transmitted power. The cellular concept turns this hindrance into an advantage. It observes that since the loss is so high, beyond a certain distance the same frequencies can be re-used without significant interference. Thus, the service area is divided into cells. A second benefit of the cellular concept is that, at least in urban areas, a cellular phone is never far from a base station. Thus, it does not need to transmit a high-power radio signal to reach a base station. This makes it possible to operate on a small battery.

Probing further: LEO telephony

Ideally, a cellular phone, with its one phone number, could be called anywhere in the world, wherever it happens to be, without the caller needing to know where it is. Unfortunately, the technological and organizational infrastructure is not quite there yet. When a phone “roams” out of its primary service area, it has to negotiate with the service provider in a new area for service. That service may be incomplete, for example allowing outgoing but not incoming calls. Charges may be exorbitant, and technical glitches may prevent smooth operation.

One candidate technology for solving these problems is a suite of global telephony services based on low-earth-orbit (**LEO**) satellites. One such project is the Iridium project, spearheaded by Motorola, and so named because in the initial conception, there would be 77 satellites. The iridium atom has 77 electrons. The idea is that enough satellites are put into orbit that one is always near enough to communicate with a hand-held telephone. When the orbit is low enough that a hand-held telephone can reach the satellite (a few hundred kilometers above the surface of the earth), the satellites move by fairly quickly. As a consequence, during the course of a telephone conversation, the connection may have to be handed off from one satellite to another. In addition, in order to be able to serve enough users simultaneously, each satellite has to re-use frequencies according to the cellular concept. To do that, it focuses multiple beams on the surface of the earth using multi-element antenna arrays.

There is some debate about whether this approach is economically viable. The investment already has been huge, with at least one high-profile bankruptcy already, so the risks are high. Better networking of terrestrial cellular services may provide formidable competition. The LEO approach, however, has one advantage that terrestrial services cannot hope to match anytime soon: truly worldwide service. The satellites provide service essentially everywhere, even in remote wilderness areas and at sea.

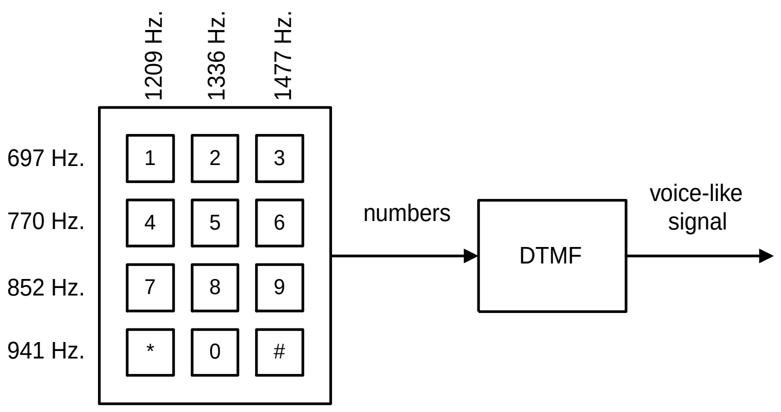


Figure 1.16: DTMF converts numbers from a keypad into a voice-like signal.

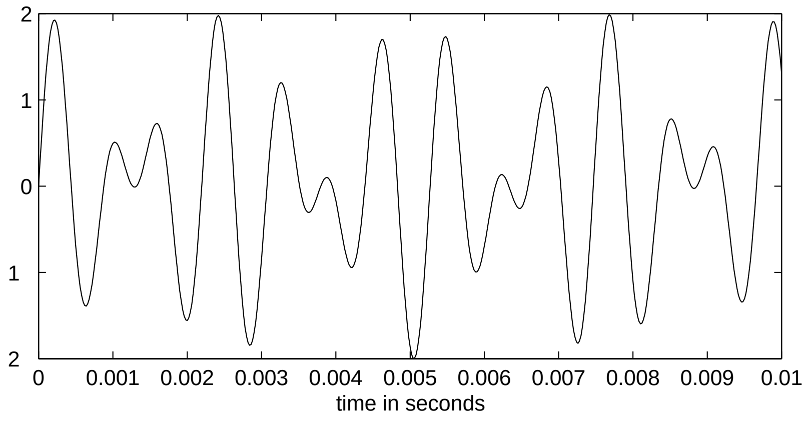


Figure 1.17: Waveform representing the “0” key in DTMF.

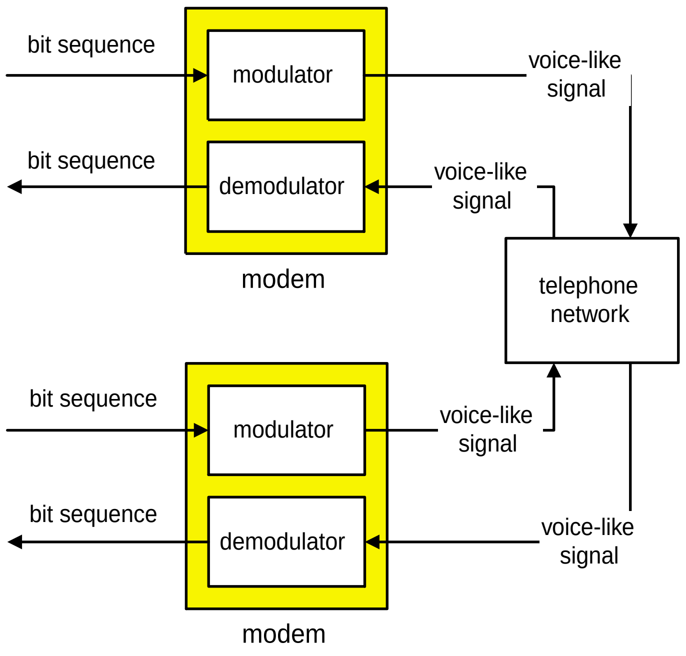


Figure 1.18: Voiceband data modems.

The **DTMF** standard — dual-tone, multi-frequency — provides precisely such a mechanism. As indicated at the left in the figure, when the customer pushes one of the buttons on the telephone keypad, a sound is generated that is the sum of two sinusoidal signals. The frequencies of the two sinusoids are given by the row and column of the key. Thus, for example, a "0" is represented as a sum of two sinusoids with frequencies 941 Hz and 1336 Hz. The waveform for such a sound is shown in figure 1.17. The line card in the central office measures these frequencies to determine which digit was dialed.

Modems

Because POTS is ubiquitous, it is attractive to find a way for it to carry computer data. Like the numbers on a keypad, computer data is intrinsically discrete. Computer data are represented by bit sequences, which are functions of the form

$$\boxed{\mathit{BitSequence}\colon \mathit{Indices} \to \mathit{Bin},}$$

where $\mathit{Indices} \subset \mathit{Nats}$, the natural numbers, and $\mathit{Bin} = \{0, 1\}$. Like keypad numbers, in order for a bit sequence to traverse a POTS phone line, it has to be transformed into something that resembles a voice signal. Further, a system is needed to transform the voice-like signal back into a bit sequence. A system that does that is called a **voiceband data modem**, shown just below the upper right in figure 1.14. The word **modem** is a contraction of modulator, demodulator. A typical arrangement is shown in figure 1.18.

The voice-like signal created by modern modems does not sound like the discrete tones of DTMF, but rather sounds more like hiss. This is a direct consequence of the fact that modern modems carry much more data per second than DTMF can (up to 54,000 bits per second rather than just a few digits per second).

Most line cards involved in POTS service convert the voice signal into a digital bit stream at the rate of 64,000 bits per second. This bit stream is then transported by the digital network. A voiceband data modem gains access to the digital network rather indirectly, by first constructing a voice-like signal to send to the line card. This gives the voiceband data modem the universality that it has. It works anywhere because the telephone network is designed to carry voice, and it is making the digital data masquerade as voice. It would be nice to get more direct access to the digital network.

Digital networks

The first widely available service that gave direct access to the global digital telephone network was **ISDN** — integrated services digital network. The ISDN service required that a different line card be installed at the central office; it was therefore not as universally available as POTS. In fact, it took nearly 10 years in the U.S. for ISDN become widely installed after it was developed in the early 1980s.

The configuration for ISDN is shown below the voiceband data modem in figure 1.14. It requires a special modem on the customer side as well as a special line card in the central office. ISDN typically provides two channels at rates of 64,000 bits per second plus a third control channel with a rate of 16,000 bits per second. One of the 64 kbps channels can be used for voice while simultaneously the other two channels are used for data.

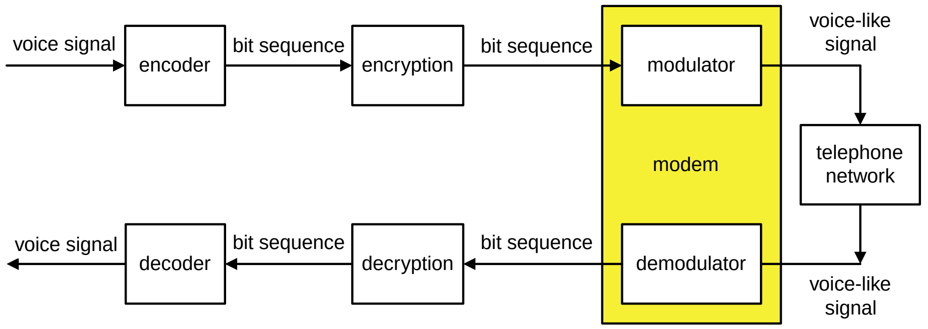
A more modern service is **DSL** — digital subscriber line. As shown at the lower right in figure 1.14, the configuration is similar to ISDN. Specialized modems and line cards are required. **ADSL**, asymmetric DSL, is a variant that provides an asymmetric bit rate, with a much higher rate in the direction from the central office to the customer than from the customer to the central office. This asymmetry recognizes the reality of most Internet applications, where relatively little data flows from the client, and torrents of data (including images and video) flow from the server.

Modems are used for many other channels besides the voiceband channel. Digital transmission over radio, for example, requires that the bit sequence be transformed into a radio signal that conforms with regulatory constraints on that radio channel. Digital transmission over electrical cable requires transforming the bit sequence into a form that propagates well over such cable and that does not radiate too much radio-frequency interference. Digital transmission over optical fiber requires transforming the bit sequence into a light signal, usually with the intensity being modulated at very high rates. At each stage in the telephone network, therefore, a voice signal has a different physical form with properties that are well suited to the medium through which the signal propagates. For example, voice, which in the form of sound only travels well over the short distances, is converted to an electrical signal that carries well over copper wires for the few kilometers. Copper wires, however, are not as well matched for long distances as optical fiber. Most long distance communication channels today use optical fiber, although satellites still have certain advantages.

Probing further: Encrypted speech

Pairs of modems are used at opposite ends of a telephone connection, each with a transmitter and a receiver to achieve bidirectional (called **full duplex**) communication. Once such modems are in place, and once they have been connected via the telephone network, then they function as a bidirectional "bit pipe." That bit pipe is then usable by other systems, such as your computer.

One of the strangest uses is to transmit digitally represented and encrypted voice signals. Here is a depiction of this relatively complicated arrangement:



What is actually sent through the telephone network sounds like hiss, which by itself provides a modicum of privacy. Casual eavesdroppers will be unable to understand the encoded speech. However, this configuration also provides protection against sophisticated listeners. A listener that is able to extract the bit sequence from this sound will still not be able to reconstruct the voice signal because the bit sequence is encrypted.

Only one end is shown. The encoder and decoder, which convert voice signals to and from bit sequences, are fairly sophisticated systems, as are the encryption and decryption systems. The fact that such an approach is cost effective has more to do with economics and sociology than technology.

Signal degradation

A voice received via the telephone network is different from the original in several respects. These differences can be modeled by a system that degrades the voice signal.

First, there is a loss of information because of sampling and quantization in the encoder, as discussed in the section 1.1.6. Moreover, the media that carry the signal, such as the twisted pair, are not perfect. They distort the signal. One cause of distortion is addition of **noise** to the signal. Noise, by definition, is any undesired component in the signal. Noise in the telephone network is sometimes audible as background hiss, or as **crosstalk**, i.e., leakage from other telephone channels into your own. Another degradation is that the medium attenuates the signal, and this attenuation depends on the signal frequency. The line card, in particular, usually contains a **bandlimiting filter** that discards the high frequency components of the signal. This is why telephone channels do not transport music well. Finally, the signal propagates over a physical medium at a finite speed, bounded by the speed of light, and so there is a delay between the time you say something and the time when the person at the other end hears what you say. Light travels through 1 km of optical fiber in approximately 5 μs, so the 5,000 km between Berkeley and New York causes a delay of about 25 ms, which is not easily perceptible.[**]

Communications engineering is concerned with how to minimize the degradation for all kinds of communication systems, including radio, TV, cellular phones, and computer networks (such as the Internet).

1.2.3 Audio storage and retrieval

We have seen how audio signals can be represented as sequences of numbers. Digital audio storage and retrieval is all about finding a physical and persistent representation for these numbers. These numbers can be converted into a single sequence of bits (binary digits) and then “printed” onto some physical medium from which they can later be read back. The transformation of sound into its persistent representation can be modeled as a system, as can the reverse or playback process.

> **Example 1.12:** In the case of compact discs (CDs), the physical medium is a layer of aluminum on a platter into which tiny pits are etched. In the playback device, a laser aimed at the platter uses an interference pattern to determine whether or not a pit exists at a particular point in the platter. These pits, thus, naturally represent binary digits, since they can have two states (present or not present).
>
> While a voiceband data modem converts bit sequences into voice-like signals, a musical recording studio does the reverse, creating a representation of the sound that is a bit

[**] A phone conversation relayed by satellite has a much larger delay. Most satellites traditionally used in the telecommunications network are **geosynchronous**, meaning that they hover at the same point over the surface of the earth. To do that, they have to orbit at a height of 22,300 miles or 35,900 kilometers. It takes a radio signal about 120 ms to traverse that distance; since a signal has to go up and back, there is an end-to-end delay of at least 240 ms (not counting delays in the electronics). If you are using this channel for a telephone conversation, then the round-trip delay from when you say something to when you get a reaction is a minimum of 480 ms. This delay can be quite annoying, impeding your ability to converse until you got used to it. If you use Internet telephony, the delays are even larger, and they can be irregular depending upon how congested the Internet is when you call.

sequence,

$$\mathit{RecordingStudio}\colon \mathit{Sounds} \to \mathit{BitStreams}.$$

There is a great deal of engineering in the details, however. For instance, CDs are vulnerable to surface defects, which may arise in manufacturing or in the hands of the user. These defects may obscure some of the pits, or fool the reading laser into detecting a pit where there is none. To guard against this, a very clever error-correcting code called a Reed-Solomon code is used. The coding process can be viewed as a function

$$\mathit{Encoding}\colon \mathit{BitStreams} \to \mathit{RedundantBitStreams}.$$

where $\mathit{RedundantBitStreams} \subset \mathit{BitStreams}$ is the set of all possible encoded bit sequences. These bit sequences are redundant, in that they contain more bits than are necessary to represent the original bit sequence. The extra bits are used to detect errors, and (sometimes) to correct them. Of course, if the surface of the CD is too badly damaged, even this clever scheme fails, and the audio data will not be recoverable.

CDs also contain **meta data**, which is extra information about the audio signal. This information allows the CD player to identify the start of a musical number and its length, and sometimes the title and the artist.

The CD format can also be used to contain purely digital data. Such a CD is called a **CD ROM** (read-only memory). It is called this because, like a computer memory, it contains digital information. But unlike a computer memory, that information cannot be modified.

DVD (**digital video discs**) take this concept much further, including much more meta data. They may eventually replace CDs. They are entirely compatible, in that they can contain exactly the same audio data that a CD can. DVD players can play CDs, but not the reverse, however. DVDs can also contain digital video information and, in fact, any other digital data. **DAT** (**digital audio tape**) is also a competitor to CDs, but has failed to capture much of a market.

1.2.4 Modem negotiation

A very different kind of system is the one that manages the establishment of a connection between two voiceband data modems. These two modems are at physically different locations, are probably manufactured by different manufacturers, and possibly use different communication standards. Both modems convert bit streams to and from voice-like signals, but other than that, they do not have much in common.

When a connection is established through the telephone network, the answering modem emits a tone that announces "I am a modem." The initiating modem listens for this tone, and if it fails to detect it, assumes that no connection can be established and hangs up. If it does detect the tone, then it answers with a voice-like signal that announces "I am a modem that can communicate according to ITU standard x," where x is one of the many modem standard published by the **International Telecommunication Union**, or **ITU**.

The answering modem may or may not recognize the signal from the initiating modem. The initiating modem, for example, may be a newer modem using a standard that was established after the answering modem was manufactured. If the answering modem does recognize the signal, then it responds with a signal that says “good, I too can communication using standard x, so let’s get started.” Otherwise, it remains silent. The initiating modem, if it fails to get a response, tries another signal, announcing “I am a modem that can communicate according to ITU standard y,” where y is typically now an older (and slower) standard. This process continues until the two modems agree on a standard.

Once agreement is reached, the modems need to make measurements of the telephone channel to compensate for its distortion. They do this by sending each other pre-agreed signals called **training signals**, defined by the standard. The training signal is distorted by the channel, and, since the receiving modem knows the signal, it can measure the distortion. It uses this measurement to set up a device called an **adaptive equalizer**. Once both modems have completed their setup, they begin to send data to one another.

As systems go, modem negotiation is fairly complex. It involves both event sequences and voice-like signals. The voice like signals need to be analyzed in fairly sophisticated ways, sometimes producing events in the event sequences. It will take this entire book to analyze all parts of this system. The handling of the event sequences will be treated using finite state machines, and the handling of the voice-like signals will be treated using frequency-domain concepts and filtering.

1.2.5 Feedback control system

Feedback control systems are composite systems where a **plant**, the nature of which we have little control over, is fed a control signal. A plant may be a mechanical device, such as the power train of a car, or a chemical process, or an aircraft with certain inertial and aerodynamic properties, for example. Sensors attached to the plant produce signals that are fed to the controller, which then generates the control signal. This arrangement, where the plant feeds the controller and the controller feeds the plant, is a complicated sort of composite system called a **feedback control system**. It has extremely interesting properties which we will explore in much more depth in subsequent chapters.

In this chapter, we construct a model of a feedback control system using the syntax of block diagrams. Each system model consists of several interconnected components. We will identify the input and output signals of each component and how the components are interconnected, and we will argue on the basis of a common-sense physics how the overall system will behave. In later chapters we consider mathematical specifications of these component systems from which we can mathematically *analyze* the system behavior.

Example 1.13: Consider a forced air heating system, which heats a room in a home or office to a desired temperature. Our first task is to identify the individual components of the heating system. These are

- a furnace/blower unit (which we will simply call the heater) that heats air and blows the hot air through vents into a room,
- a temperature sensor that measures the temperature in a room, and

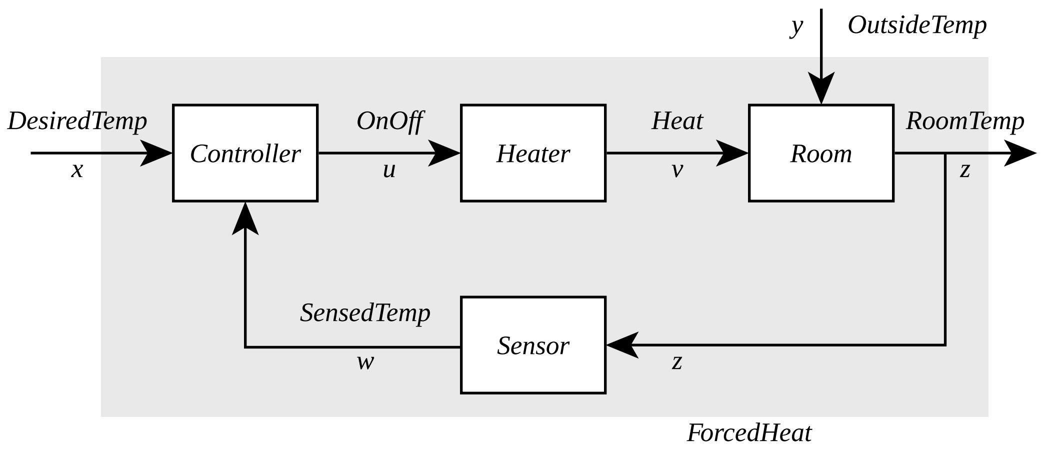


Figure 1.19: The interconnected components of a forced air heating system.

- the control system that compares the specified desired temperature with the sensed temperature and turns the furnace/blower unit on or off depending on whether the sensed temperature is below or above the demanded temperature.

The interconnection of these components is shown in figure 1.19.

Our second task is to specify the input and output signals of each component (the domain and range of the function), ensuring the input-output matching conditions. The heater produces hot air depending on whether it is turned on or off. So its input signal is simply a function of time which takes one of two values, *On* or *Off*. We call input to the heater (a signal) *OnOff*,

$$\mathit{OnOff}\colon \mathit{Time} \to \{\mathit{On}, \mathit{Off}\},$$

and we take $\mathit{Time} = \mathit{Reals}_+$, the non-negative reals. So the input signal space is

$$\mathit{OnOffProfiles} = [\mathit{Reals}_+ \to \{\mathit{On}, \mathit{Off}\}].$$

(Recall that the notation $[D \to R]$ defines a function space, as explained in section 1.2.1.) When the heater is turned on it produces heat at some rate that depends on the capacity of the furnace and blower. We measure this heating rate in BTUs per hour. So the output signal of the heater, which we name *Heat* is of the form

$$\mathit{Heat}\colon \mathit{Reals}_+ \to \{0, B_c\},$$

where B_c is the heating capacity measured in BTU/hour. If we name the output signal space *HeatProfiles*, then

$$\mathit{HeatProfiles} = [\mathit{Reals}_+ \to \{0, B_c\}].$$

Thus the *Heater* system is described by a function

$$\mathit{Heater}\colon \mathit{OnOffProfiles} \to \mathit{HeatProfiles}. \tag{1.13}$$

Common-sense physics tells us that when the heater is turned on the room will begin to warm up and when the heater is turned off the room temperature will fall until it reaches the outside temperature. So the room temperature depends on both the heat delivered by the heater and the outside temperature. Thus the input signal to the room is the pair $(\mathit{Heat}, \mathit{OutsideTemp})$. We can take *OutsideTemp* to be of the form

$$\mathit{OutsideTemp}\colon \mathit{Reals}_+ \to [\mathit{min}, \mathit{max}],$$

where $[\mathit{min}, \mathit{max}]$ is the range of possible outside temperatures, measured in degrees Celsius, say. The output signal of the room is of course the room temperature, $\mathit{RoomTemp}\colon \mathit{Reals}_+ \to [\mathit{min}, \mathit{max}]$. If we denote

$$\mathit{OutsideTempProfiles} = [\mathit{Reals}_+ \to [\mathit{min}, \mathit{max}]],$$

and

$$\mathit{RoomTempProfiles} = [\mathit{Reals}_+ \to [\mathit{min}, \mathit{max}]],$$

then the behavior of the *Room* system is described by a function

$$\mathit{Room}\colon \mathit{HeatProfiles} \times \mathit{OutsideTempProfiles} \to \mathit{RoomTempProfiles} \tag{1.14}$$

In a similar manner, the *Sensor* system is described by a function

$$\mathit{Sensor}\colon \mathit{RoomTempProfiles} \to \mathit{SensedTempProfiles} \tag{1.15}$$

with input signal space *RoomTempProfiles* and output signal space

$$\mathit{SensedTempProfiles} = [\mathit{Reals}_+ \to [\mathit{min}, \mathit{max}]].$$

The *Controller* is described by the function

$$\mathit{Controller}\colon \mathit{DesiredTempProfile} \times \mathit{SensedTempProfile} \to \mathit{OnOffProfile}, \tag{1.16}$$

where

$$\mathit{DesiredTempProfiles} = [\mathit{Reals}_+ \to [\mathit{min}, \mathit{max}]].$$

We have constructed a model where the input-output matching condition is satisfied everywhere.

The overall forced air heating system (the shaded part of figure 1.19) has a pair of input signals, desired temperature and outside temperature, and one output signal, room temperature. So it is described by the function

$$\mathit{ForcedHeat}\colon \mathit{DesiredTempProfiles} \times \mathit{OutsideTempProfiles} \to \mathit{RoomTempProfiles}.$$

If we are given the input signal values x of desired temperature and the value y of outside temperature, we can compute the value $z = \mathit{ForcedHeat}(x, y)$ by solving the following four simultaneous equations

$$\begin{aligned} u &= \mathit{Controller}(x, w) \\ v &= \mathit{Heater}(u) \\ z &= \mathit{Room}(y, v) \\ w &= \mathit{Sensor}(z) \end{aligned} \tag{1.17}$$

Given x and y, an we must solve these four equations to determine the four unknown functions u, v, w, z of which u, v, w are the internal signals, and z is the output signal.

Of course to solve these simultaneous equations, we need to specify the four system functions. So far we have simply given names to those functions and identified their domain and range. To complete the specification we must describe how those functions assign output signals to input signals. The various ways of doing that are the subject of later chapters.

1.3 Summary

Signals are functions that represent information. We studied examples of three classes of signals. In the first class are functions of discrete or continuous time and space that occur in human and machine perception. In the second class are functions of time and space representing attributes of physical objects or devices. The third class of signals consist of sequences of symbols representing data or the occurrences of events.

Systems are functions that transform signals. We looked at telecommunication systems, where a network that was originally designed for carrying voice signals is used for many other kinds of signals today. One way to accomplish this is to design systems such as modems that transform signals so that they masquerade as voice-like signals. We also looked at system models for signal degradation and for storage of signals. We looked at systems that are primarily concerned with discrete events and command sequences, and we examined feedback control systems.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **C** For each of the continuous-time signals below, represent the signal in the form of $f \colon X \to Y$ and as a sketch like figure 1.1. Carefully identify the range and domain in each case.
   (a) The voltage across a car battery,
   (b) The closing prices on each day of a share of a company,
   (c) The position of a moving vehicle on a straight road,
   (d) The simultaneous position of two moving vehicles on the same straight road, and
   (e) The sound heard in both of your ears.
2. **C** Represent the following examples of spatial information as a signal in the form of $f \colon X \to Y$. Carefully identify the range and domain in each case.
   (a) An image impressed on photgraphic paper,

   (b) An image from a scanner stored in computer memory,
   (c) The height of points on the surface of the earth,
   (d) The location of the chairs in a room.

3. **C** Represent these examples as data or event sequences. Identify the range and domain in each case.
   (a) The result of 100 tosses of a coin,
   (b) The log of button presses inside an elevator,
   (c) The log of the main events in a soda vending machine,
   (d) What you would say to a motorist asking directions,
   (e) A play-by-play account of a game of chess.

4. **C** Formulate the following items of information as functions. Identify the domain and range in each case.
   (a) The population of U.S. cities,
   (b) The white pages in a phone book (careful: the white pages may list two identical names, and may list the same phone number under two different names),
   (c) The birth dates of students in class,
   (d) The broadcast frequencies of AM radio stations,
   (e) The broadcast frequencies of FM radio stations, (look at your radio dial, or at the web page:

   `http://www.eecs.berkeley.edu/˜eal/eecs20/sidebars/radio/index.html.`

5. **E** Use Matlab to plot the graph of the following continuous-time functions defined over $[-1, 1]$, and on the same plot display 11 uniformly spaced samples (0.2 seconds apart) of these functions. Are these samples good representations of the waveforms?
   (a) $f\colon [-1, 1] \to \textit{Reals}$, where for all $x \in [-1, 1]$, $f(x) = e^{-x}\sin(10\pi x)$.
   (b) $\textit{Chirp}\colon [-1, 1] \to \textit{Reals}$, where for all $t \in [-1, 1]$, $\textit{Chirp}(t) = \cos(10\pi t^2)$.

6. **T** There is a large difference between the sets X, Y, and $[X \to Y]$. This exercise explores some of that difference.
   (a) Suppose $X = \{a, b, c\}$ and $Y = \{0, 1\}$. List all the functions from X to Y, i.e. all the elements of $[X \to Y]$
   (b) If X has m elements and Y has n elements, how many elements does $[X \to Y]$ have?
   (c) Suppose

$$\begin{aligned}\textit{ColormapImages} = [&\textit{DiscreteVerticalSpace} \times \textit{DiscreteHorizontalSpace}\\ &\to \textit{ColorMapIndexes}].\end{aligned}$$

   Suppose the domain of each image in this set has 6,000 pixels and the range has 256 values. How many distinct images are there? Give an approximate answer in the form of 10^n. **Hint**: $a^b = 10^{b\log_{10}(a)}$.

Chapter 2

Defining Signals and Systems

The previous chapter describes the modeling of signals and systems as functions, concentrating on how to select the domain and range. This chapter is concerned with how to give more complete definitions of these functions. In particular, we need an **assignment rule**, which specifies how to assign an element in the range to each element in the domain.

There are many ways to give an assignment rule. A theme of this chapter is that these different ways have complementary uses. Procedural descriptions of the assignment rule, for example, are more convenient for synthesizing signals or constructing implementations of a system in software or hardware. Mathematical descriptions are more convenient for analyzing signals and systems and determining their properties.

Given the complementary uses of descriptions of assignment rules, we find that in practice it is often necessary to use several in combination. Much of what a practicing engineer does in designing systems is reconciling these diverse views, for example to ensure that particular piece of software indeed implements a system that is specified mathematically. We begin with a discussion of functions in general, and then specialize to signals and systems.

2.1 Defining functions

A function $f\colon X \to Y$ assigns to each element in X an element in Y. This assignment can be defined by declaring the mathematical relationship between the value in X and the value in Y, by graphing or enumerating the possible assignments, by giving a procedure for determining the value in Y given a value in X, or by composing simpler functions. We go over each of these in more detail in this section.

2.1.1 Declarative assignment

Consider the function *Square*: *Reals* $\to$ *Reals* given by

$$\forall\, x \in \textit{Reals}, \quad \textit{Square}(x) = x^2. \tag{2.1}$$

In (2.1), we have used the universal quantifier symbol '$\forall$', which means 'for all' or 'for every' to declare the relationship between values in the domain of the function and values in the range. Statement (2.1) is read: "for every value of x in *Reals*, the function *Square* evaluated at x is assigned the value x^2." The expression "$\mathit{Square}(x) = x^2$" in (2.1) is an assignment.[1]

Expression (2.1) is an instance of the following prototype for defining functions. Define $f\colon X \to Y$ by

$$\boxed{\forall\, x \in X, \quad f(x) = \text{expression in } x.} \tag{2.2}$$

In this prototype, f is the name of the function to be defined, such as *Square*, X is the domain of f, Y is the range of f, and 'expression in x' specifies the value in Y assigned to $f(x)$.

The prototype (2.2) does not say how the 'expression in x' is to be evaluated. In the *Square* example above, it was specified by the algebraic expression $\mathit{Square}(x) = x^2$. Such a definition of a function is said to be **declarative**, because it declares properties of the function without directly explaining how to construct the function.

Example 2.1: Here are some examples of functions of complex variables.[2]

The magnitude of a complex number is given by $\mathit{abs}\colon \mathit{Comps} \to \mathit{Reals}_+$, where *Comps* is the set of complex numbers and Reals_+ is the set of set of non-negative real numbers, and

$$\forall\, z = x + iy \in \mathit{Comps}, \quad \mathit{abs}(z) = \sqrt{(x^2 + y^2)}$$

The complex conjugate of a number, $\mathit{conj}\colon \mathit{Comps} \to \mathit{Comps}$, is given by

$$\forall\, z = x + iy \in \mathit{Comps}, \quad \mathit{conj}(z) = x - iy$$

The exponential of a complex number, $exp\colon \mathit{Comps} \to \mathit{Comps}$, is given by

$$\forall\, z \in \mathit{Comps}, \quad exp(z) = \sum_{n=0}^{\infty} \frac{z^n}{n}$$

It is worth emphasizing that the last definition is declarative. In particular, it does not give a procedure for calculating the exponential function, since the sum is infinite. Such a calculation would never terminate.

Example 2.2: The signum function gives the sign of a real number, $\mathit{signum}\colon \mathit{Reals} \to \{-1, 0, 1\}$,

$$\forall\, x \in \mathit{Reals}, \quad \mathit{signum}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.3}$$

The right side of this assignment tabulates three expressions for three different subsets of the domain. Below we will consider a more extreme case of this where every value in the domain is tabulated with a value in the range.

[1] See appendix A for a discussion of the use of "=" as an assignment, vs. its use as an assertion.
[2] See appendix B for a review of complex variables.

Example 2.3: The size of a matrix, $\mathit{size}\colon \mathit{Matrices} \to \mathit{Nats} \times \mathit{Nats}$, is given by

$$\forall\, M \in \mathit{Matrices}, \quad \mathit{size}(M) = (m, n),$$

where m is the number of rows of the matrix M, n is the number of columns of M, and *Matrices* is the set of all matrices.

This definition relies not only on formal mathematics, but also on the English sentence that defines m and n. Without that sentence, the assignment would be meaningless.

2.1.2 Graphs

Consider a function $f\colon X \to Y$. To each $x \in X$, f assigns the value $f(x)$ in Y. The pair $(x, f(x))$ is an element of the product set $X \times Y$. The set of all such pairs is called the **graph** of f, written $\mathit{graph}(f)$. Using the syntax of sets, $\mathit{graph}(f)$ is the subset of $X \times Y$ defined by

$$\mathit{graph}(f) = \{(x, y) \mid x \in X \text{ and } y = f(x)\}, \tag{2.4}$$

or slightly more simply,

$$\mathit{graph}(f) = \{(x, f(x)) \mid x \in X\}.$$

The vertical bar | is read "such that," and the expression after it is a **predicate** that defines the set.[3]

When $X \subset \mathit{Reals}$ and $Y \subset \mathit{Reals}$, we can plot $\mathit{graph}(f)$ on a page. For example,

$$\mathit{graph}(\mathit{Square}) = \{(x, x^2) \mid x \in [-1, 1]\}$$

is plotted in figure 2.1. In that figure, the horizontal and vertical axes represent the domain and the range, respectively (more precisely, a subset of the domain and the range). The rectangular region enclosed by these axes represents the product of the domain and the range (every point in that region is a member of $(\mathit{Reals} \times \mathit{Reals})$). The graph is visually rendered by placing a black dot at every point in that region that is a member of $\mathit{graph}(\mathit{Square})$. The resulting picture is the familiar plot of the *Square* function.

While the graph of $f\colon X \to Y$ is a subset of $X \times Y$, it is a very particular sort of subset. For each element $x \in X$, there is exactly one element $y \in Y$ such that $(x, y) \in \mathit{graph}(f)$. In particular, there cannot be more than one such $y \in Y$, and there cannot be no such $y \in Y$. This is, in fact, what we mean when we say that f is a function.

Example 2.4: Let $X = \{1, 2\}$ and $Y = \{a, b\}$. Then

$$\{(1, a), (2, a)\}$$

is the graph of a function, but

$$\{(1, a), (1, b)\}$$

is not. Neither is

$$\{(1, a)\}.$$

[3] See appendix A for a review of this notation.

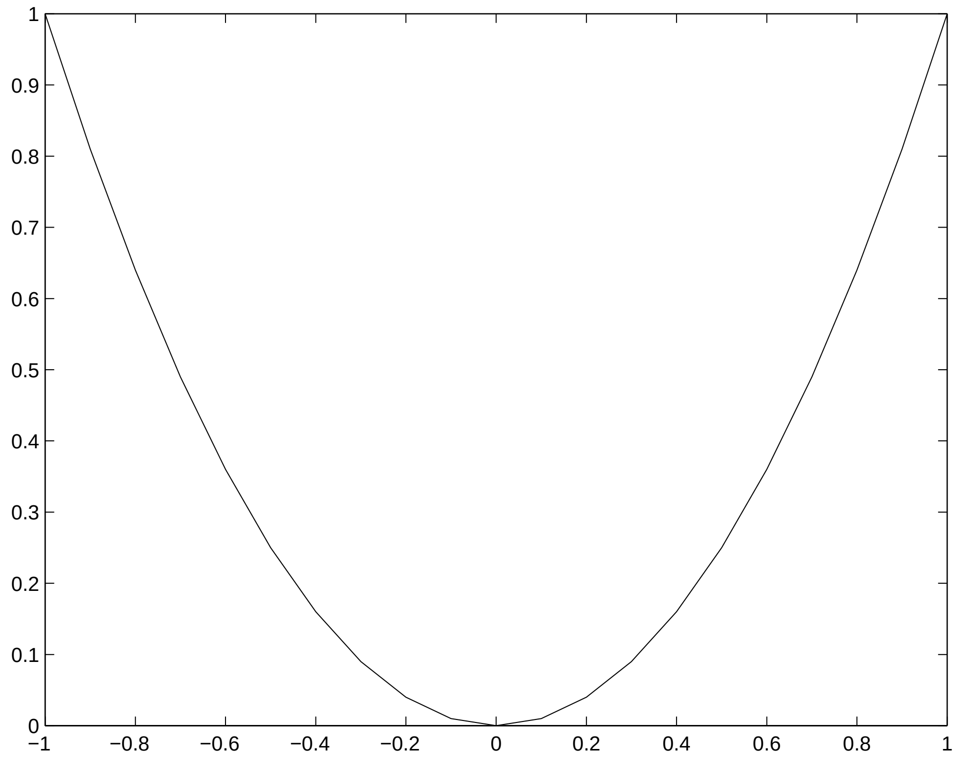


Figure 2.1: Graph of *Square*

Probing further: Relations

The graph of a function $f: X \to Y$ is a subset of $X \times Y$, as defined in (2.4). An arbitrary subset of $X \times Y$ is called a **relation**. Thus, a function is a special kind of relation. For relations, it is common to call X the **domain** and Y the **codomain**. A relation is a function if for every $x \in X$ there is exactly one $y \in Y$ such that (x, y) is an element of the relation. So a relation $R \subset X \times Y$ is a function if for every $x \in X$ there is a $y_1 \in Y$ such that $(x, y_1) \in R$, and if in addition $(x, y_2) \in R$, then $y_1 = y_2$.

Name	Marks
John Brown	90.0
Jane Doe	91.2
...	...

Table 2.1: Tabular representation of *Score*.

The graph of *Square*, *graph*(*Square*), is given by the algebraic expression for $Square(x) = x^2$. In other cases, no such algebraic expression exists. For example, *Voice* is specified through its graph in figure 1.1, not through an algebraic expression. Thus, graphs can be used to define functions that cannot be conveniently given by declarative assignments.

Consider again the prototype in (2.2),

$$\forall\, x \in X, \quad f(x) = \text{ expression in } x$$

The graph of f is

$$graph(f) = \{(x, y) \in X \times Y \mid y = \text{ expression in } x\}.$$

The expression '$y =$ expression in x' is a predicate in the variable (x, y) and so this prototype definition conforms to the prototype new set constructor given in (A.4) of appendix A:

$$\boxed{NewSet = \{z \in Set \mid Pred(z)\}.}$$

Since the graph of f is a set, we can define the function f via its graph using the same techniques we use to define sets.

2.1.3 Tables

If $f: X \to Y$ has finite domain, then $graph(f) \subset X \times Y$ is a finite set, so it can be specified simply by a list of all its elements. This list can be put in the form of a table. This table defines the function.

Example 2.5: Suppose the function

$$Score\colon Students \to [0, 100]$$

gives the outcome of the first midterm exam for each student in the class. Obviously, this function cannot be given by an algebraic declarative assignment. But it can certainly be given as a table, as shown in table 2.1.

Example 2.6: The command nslookup on a networked computer is a function that maps hostnames into their IP (Internet) address. For example, if you type:

```
nslookup cory.eecs.berkeley.edu
```

you get the IP address 128.32.134.240. The **domain name server** attached to your machine stores the nslookup function as a table.

2.1.4 Procedures

Sometimes the value $f(x)$ that a function f assigns to an element $x \in domain(f)$ is obtained by executing a procedure.

Example 2.7: Here is a Matlab procedure to compute the factorial function

$$fact\colon \{1, \cdots, 10\} \to Nats,$$

where *Nats* is the natural numbers:

```
fact(1) = 1;
for n = 2:10
   fact(n) = n * fact(n-1);
 end
```

Unlike previous mechanisms for defining a function, this one gives a constructive method to determine an element in the range given an element in the domain. This style is called **imperative** to distinguish it from declarative. The relationship between these two styles is interesting, and quite subtle. It is explored further in section 2.1.6.

2.1.5 Composition

Functions can be combined to define new functions. The simplest mechanism is to connect the output of one function to the input of another. We have been doing this informally to define systems by connecting components in block diagrams such as figure 1.18.

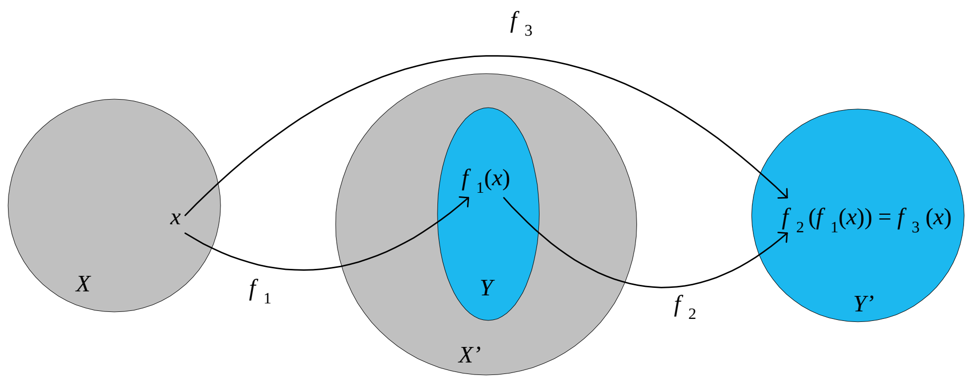


Figure 2.2: Function composition: $f_3 = f_2 \circ f_1$.

If the first function is f_1 and the second is f_2, then we write the composed function as $f_2 \circ f_1$. That is, for every x in the domain of f_1,

$$\boxed{(f_2 \circ f_1)(x) = f_2(f_1(x)).}$$

This is called **function composition**. A fundamental requirement for such a composition to be valid is that the range of f_1 must be a subset of the domain of f_2. In other words, any output from the first function must be in the set of possible inputs for the second. Thus, for example, the output of *modulator* in figure 1.18 is a voice-like signal, which is precisely what the system *telephone network* is able to accept as an input. Thus, we can compose the modulator with the telephone network. Without this input-output **connection restriction**, the interconnection would be meaningless.

It is worth pausing to study the notation $f_2 \circ f_1$. Assume $f_1 \colon X \to Y$ and $f_2 \colon X' \to Y'$. Then if $Y \subset X'$, we can define

$$f_3 = f_2 \circ f_1,$$

where $f_3 \colon X \to Y'$ such that

$$\forall\, x \in X, \quad f_3(x) = f_2(f_1(x)) \tag{2.5}$$

Why is f_1 listed second in $f_2 \circ f_1$? This convention simply mirrors the ordering of $f_2(f_1(x))$ in (2.5). We can visualize f_3 as in figure 2.2.

Example 2.8: Consider the representation of a color image using a colormap. The decoding of the image is depicted in figure 1.7. The image itself might be given by the function

$$\mathit{ColormapImage} \colon \mathit{DiscreteVerticalSpace} \times \mathit{DiscreteHorizontalSpace} \to \mathit{ColorMapIndexes}.$$

The function

$$\mathit{Display} : \mathit{ColorMapIndexes} \to \mathit{Intensity}^3$$

decodes the colormap indexes. If *ColorMapIndexes* has 256 values, it could be identified with the set *Ints8* of all 8-bit words, as we have seen. If we compose these functions

$$\mathit{ColorComputerImage} = \mathit{Display} \circ \mathit{ColormapImage}$$

then we get the decoded representation of the image

$$\mathit{ColorComputerImage} \colon \mathit{DiscreteVerticalSpace} \times \mathit{DiscreteHorizontalSpace} \to \mathit{Intensity}^3.$$

ColorComputerImage describes how an image looks when it is displayed, whereas $ColormapImage$ describes how it is stored in the computer.

If $f : X \to X$, i.e. the domain and range of f are the same, we can form the function

$$\boxed{f^2 = f \circ f.}$$

We can compose f^2 with f to form f^3, and so on.

Example 2.9: Consider the function $S \colon \mathit{Reals}^2 \to \mathit{Reals}^2$, where the assignment $(y_1, y_2) = S(x_1, x_2)$ is defined by matrix multiplication,

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{2.6}$$

The function $S^2 = S \circ S \colon \mathit{Reals}^2 \to \mathit{Reals}^2$ is also defined by matrix multiplication, and the corresponding matrix is the square of the matrix in (2.6).

To see this, let $(y_1, y_2) = S(x_1, x_2)$ and $(z_1, z_2) = S(y_1, y_2) = (S \circ S)(x_1, x_2)$. Then we see that

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}}_{A} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$= \underbrace{\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}}_{A^2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = A^2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

Example 2.10: Consider another example in the context of the telephone system. Let *Voices* be the set of possible voice input signals of the form

$$\mathit{Voice} \colon \mathit{Time} \to \mathit{Pressure}.$$

Voices is a function space,

$$\mathit{Voices} = [\mathit{Time} \to \mathit{Pressure}].$$

A telephone converts a *Voice* signal into a signal in the set

$$\mathit{LineSignals} = [\mathit{Time} \to \mathit{Voltages}].$$

Thus, we could define

$$\mathit{Mouthpiece}\colon \mathit{Voices} \to \mathit{LineSignals}.$$

The twisted wire pair may distort this signal, so we define a function

$$\mathit{LocalLoop}\colon \mathit{LineSignals} \to \mathit{LineSignals}.$$

The input to the line card therefore is

$$(\mathit{LocalLoop} \circ \mathit{Mouthpiece})(\mathit{Voice}).$$

Similarly let *BitStreams* be the set of possible bitstreams of the form:

$$\mathit{BitStream}\colon \mathit{DiscreteTime} \to \mathit{Bin}$$

where $\mathit{DiscreteTime} = \{0, 1/64,000, 2/64,000, \cdots\}$, since there are 64,000 bits/sec. So,

$$\mathit{BitStreams} = [\mathit{DiscreteTime} \to \mathit{Bin}].$$

The encoder in a line card can be mathematically described as a function

$$\mathit{Encoder}\colon \mathit{LineSignals} \to \mathit{BitStreams}$$

or, with more detail, as a function

$$\mathit{Encoder}\colon [\mathit{Time} \to \mathit{Voltages}] \to [\mathit{DiscreteTime} \to \mathit{Bin}].$$

The digital telephone network itself might be modeled as a function

$$\mathit{Network}\colon \mathit{BitStreams} \to \mathit{BitStreams}.$$

We can continue in this fashion until we model the entire path of a voice signal through the telephone network as the function composition

$$\mathit{Earpiece} \circ \mathit{LocalLoop}_2 \circ \mathit{Decoder} \circ \mathit{Network} \circ \mathit{Encoder} \circ \mathit{LocalLoop}_1 \circ \mathit{Mouthpiece}. \quad (2.7)$$

Given a complete definition of each of these functions, we would be well equipped to understand the degradations experienced by a voice signal in the telephone network.

Probing further: Declarative interpretation of imperative definitions

The declarative approach establishes a relation between the domain and the range of a function. For example, the equation

$$y = \sin(x)/x$$

can be viewed as defining a subset of *Reals* $\times$ *Reals*. This subset is the graph of the function *Sinc*: *Reals* $\to$ *Reals*.

The imperative approach also establishes a function, but it is a function that maps the program state before the statement is executed into a program state after the statement is executed. Consider for example the Java statement

```
     y = Math.sin(x)/x;
```

Considering only this statement (rather than a larger program), the program state is the value of the two variables, `x` and `y`. Suppose that these have been declared to be of type `double`, which in Java represents double-precision floating-point numbers encoding according to an IEEE standard. Let the set *Doubles* be the set of all numbers so encoded, and note that `NaN` $\in$ *Doubles*, not a number, the result of division by zero. The set of possible program states is therefore *Doubles* $\times$ *Doubles*. The Java statement therefore defines a function

$$\textit{Statement}\colon (\textit{Doubles} \times \textit{Doubles}) \to (\textit{Doubles} \times \textit{Doubles}).$$

2.1.6 Declarative vs. imperative

Declarative definitions of functions assert a relationship between elements in the domain and elements in the range. Imperative definitions give a procedure for finding an element in the range given one in the domain. Often, both types of specifications can be given for the same function. However sometimes the specifications are subtly different.

Consider the function

$$\mathit{SquareRoot}\colon \mathit{Reals}_+ \to \mathit{Reals}_+$$

defined by the statement "$\mathit{SquareRoot}(x)$ is the unique value of $y \in \mathit{Reals}_+$ such that $y^2 = x$." This declarative definition of *SquareRoot* does not tell us how to calculate its value at any point in its domain. Nevertheless, it defines *SquareRoot* perfectly well. By contrast, an imperative definition of *SquareRoot* would give us a procedure, or algorithm, for calculating $\mathit{SquareRoot}(x)$ for a given x. Call the result of such an algorithm $\hat{y}$. Since the algorithm would yield an approximation in most cases, $\hat{y}^2$ would not be exactly equal to x. So the declarative and imperative definitions are not always the same.

Any definition of a function following the prototype (2.2) is a declarative definition. It does not give a procedure for evaluating 'expression in x'.

Example 2.11: As another example where declarative and imperative definitions differ in subtle ways, consider the following mathematical equation:

$$y = \frac{\sin(x)}{x}. \tag{2.8}$$

Consider the following Java statement:

```
y = Math.sin(x)/x;
```

or an equivalent Matlab statement

```
y = sin(x)/x
```

Superficially, these look very similar to (2.8). There are minor differences in syntax in the Java statement, but otherwise, it is hard to tell the difference. But there are differences. For one, the mathematical equation (2.8) has meaning if y is known and x is not. It declares a relationship between x and y. The Java and Matlab statements define a procedure for computing `y` given `x`. Those statements have no meaning if y is known and x is not.

The mathematical equation (2.8) can be interpreted as a predicate that defines a function, for example the function $\mathit{Sinc}\colon \mathit{Reals} \to \mathit{Reals}$, where

$$\mathit{graph}(\mathit{Sinc}) = \{(x, y) \mid x \in \mathit{Reals}, y = \sin(x)/x\}. \tag{2.9}$$

> The Java and Matlab statements can be interpreted as imperative definitions of a function[4]. That is, given an element in the domain, they specify how to compute an element in the range. However, these two statements do not define the same function as in (2.9). To see this, consider the value of y when $x = 0$. Given the mathematical equation, it is not entirely trivial to determine the value of y. You can verify that $y = 1$ when $x = 0$ using l'Hôpital's rule[5] In contrast, the meaning of the Java and Matlab statements is that $y = 0/0$ when $x = 0$, which Java and Matlab (and most modern languages) define to be `NaN`, not a number. Thus, given $x = 0$, the procedures yield different values for y than the mathematical expression.

We can see from the above example some of the strengths and weaknesses of imperative and declarative approaches. Given only a declarative definition, it is difficult for a computer to determine the value of y. Symbolic mathematical software, such as Maple and Mathematica, is designed to deal with such situations, but these are very sophisticated programs. In general, using declarative definitions in computers requires quite a bit more sophistication than using imperative definitions.

Imperative definitions are easier for computers to work with. But the Java and Matlab statements illustrate one weakness of the imperative approach: it is arguable that $y =$ `NaN` is the wrong answer, so the Java and Matlab statements have a bug. This bug is unlikely to be detected unless, in testing, these statements happen to be executed with the value $x = 0$. A correct Java program might look like this:

```
if (x == 0.0) y = 1.0;
else y = Math.sin(x)/x;
```

Thus, the imperative approach has the weakness that ensuring correctness is more difficult. Humans have developed a huge arsenal of techniques and skills for thoroughly understanding declarative definitions (thus lending confidence in their correctness), but we are only beginning to learn how to ensure correctness in imperative definitions.

2.2 Defining signals

Signals are functions. Thus, both declarative and imperative approaches can be used to define them.

[4]Confusingly, many programming languages, including Matlab, use the term "function" to mean something a bit different from a mathematical function. They use it to mean a **procedure** that can compute an element in the range of a function given an element in its domain. Under certain restrictions (avoiding global variables for example), Matlab functions do in fact compute mathematical functions. But in general, they do not.

[5]**l'Hôpital's rule** states that if $f(a) = g(a) = 0$, then

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f'(x)}{g'(x)},$$

if the limit exists, where $f'(x)$ is the derivative of f with respect to x.

2.2.1 Declarative definitions

Consider for example an audio signal s, a pure tone at 440 Hz (middle A on the piano keyboard). Recall that audio signals are functions $\mathit{Sound}\colon \mathit{Time} \to \mathit{Pressure}$, where the set $\mathit{Time} \subset \mathit{Reals}$ represents a range of time and the set *Pressure* represents air pressure.[6] To define this function, we might give the declarative description

$$\forall\, t \in \mathit{Time}, \quad s(t) = \sin(440 \times 2\pi t). \tag{2.10}$$

In many texts, you will see the shorthand

$$s(t) = \sin(440 \times 2\pi t)$$

used as the definition of the function s. Using the shorthand is only acceptable when the domain of the function is well understood from the context. This shorthand can be particularly misleading when considering systems, and so we will only use it sparingly. A portion of the graph of the function (2.10) is shown in figure 1.3.

2.2.2 Imperative definitions

We can also give an imperative description of such a signal. When thinking of signals rather than more abstractly of functions, there is a subtle question that arises when we attempt to construct an imperative definition. Do you give the value of $s(t)$ for a particular t? Or for all t in the domain? Suppose we want the latter, which seems like a more complete definition of the function. Then we have a problem. The domain of this function may be any time interval, or all time! Suppose we just want one second of sound. Define $t = 0$ to be the start of that one second. Then the domain is $[0, 1]$. But there are an (uncountably) infinite number of values for t in this range! No Java or Matlab program could provide the value of $s(t)$ for all these values of t.

Since a signal is function, we give an imperative description of the signal exactly as we did for functions. We give a procedure that has the potential of providing values for $s(t)$, given any t.

Example 2.12: We could define a Java **method** as follows:[7]

```
double s(double t) {
    return (Math.sin(440*2*Math.PI*t));
}
```

Calling this method with a value for `t` as an argument yields a value for `s(t)`. Java (and most object-oriented languages) use the term "method" for most procedures.

[6]Recall further that we normalize *Pressure* so that zero represents the ambient air pressure. We also use arbitrary units, rather than a physical unit such as millibars.

[7]It is uncommon, and not recommended, to use single-character names for methods and variables in programs. We do it here only to emphasize the correspondence with the declarative definition. In mathematics, it is common to use single-character names.

Another alternative is to provide a set of samples of the signal.

> **Example 2.13:** In Matlab, we could define a vector `t` that gives the values of time that we are interested in:
>
> ```
> t = [0:1/8000:1];
> ```
>
> In the vector `t` there are 8001 values evenly spaced between 0 and 1, so our sample rate is 8000 samples per second. Then we can compute values of `s` for these values of `t` and listen to the resulting sound:
>
> ```
> s = cos(2*pi*440*t);
> sound(s,8000)
> ```
>
> The vector `s` also has 8001 elements, representing evenly spaced samples of one second of A-440.

2.2.3 Physical modeling

An alternative way to define a signal is to construct a model for a physical system that produces that signal.

> **Example 2.14:** A pure tone might be defined as a solution to a differential equation that describes the physics of a tuning fork.
>
> A tuning fork consists of a metal finger (called a **tine**) that is displaced by striking it with a hammer. After being displaced, it vibrates. If we assume that the tine has no friction, then it will vibrate forever. We can denote the displacement of the tine after being struck at time zero as a function $x\colon \mathit{Reals}_+ \to \mathit{Reals}$. If we assume that the initial displacement introduced by the hammer is one unit, then using our knowledge of physics we can determine that for all $t \in \mathit{Reals}_+$, the displacement satisfies the differential equation
>
> $$\ddot{x}(t) = -\omega_0^2 x(t) \tag{2.11}$$
>
> where ω_0 is constant that depends on the mass and stiffness of the tine, and and where $\ddot{x}(t)$ denotes the second derivative with respect to time of x (see box).
>
> It is easy to verify that x given by
>
> $$\forall\, x \in \mathit{Reals}_+, \quad x(t) = cos(\omega_0 t) \tag{2.12}$$
>
> is a solution to this differential equation (just take its second derivative). Thus, the displacement of the tuning fork is sinusoidal. This displacement will couple directly with air around the tuning fork, creating vibrations in the air (sound). If we choose materials for the tuning fork so that $\omega_0 = 2\pi \times 440$, then the tuning fork will produce the tone of A-440 on the musical scale.

Probing further: Physics of a Tuning Fork

A tuning fork consists of two fingers called tines, as shown in figure 2.3. If you displace one of these tines by hitting it with a hammer, it will vibrate with a nearly perfect sinusoidal characteristic. As it vibrates, it pushes the air, creating a nearly perfect sinusoidal variation in air pressure that propogates as sound. Why does it vibrate this way?

Suppose the displacement of the tine (relative to its position at rest) at time t is given by $x(t)$, where $x\colon \textit{Reals} \to \textit{Reals}$. There is a force on the tine pushing it towards its at-rest position. This is the restorative force of the elastic material used to make the tine. The force is proportional to the displacement (the greater the displacement, the greater the force), so

$$F(t) = -kx(t),$$

where k is the proportionality constant that depends on the material and geometry of the tine. In addition, Newton's second law of motion tells us the relationship between force and acceleration,

$$F(t) = ma(t),$$

where m is the mass and $a(t)$ is the acceleration at time t. Of course,

$$a(t) = \frac{d^2}{dt}x(t) = \ddot{x}(t),$$

so

$$m\ddot{x}(t) = -kx(t)$$

or

$$\ddot{x}(t) = -(k/m)x(t).$$

Comparing with (2.11), we see that $\omega_0^2 = k/m$.

A solution to this equation needs to be some signal that is proportional to its own second derivative. A sinusoid as in (2.12) has exactly this property. The sinusoidal behavior of the tine is called **simple harmonic motion**.

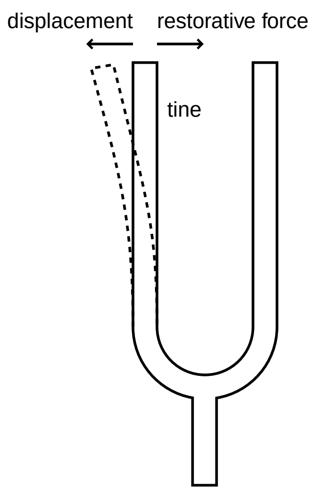


Figure 2.3: A tuning fork.

2.3 Defining systems

All of the methods that we have discussed for defining functions can be used, in principle, to define systems. However, in practice, the situation is much more complicated for systems than for signals. Recall from section 1.2.1 that a system is a function where the domain and range are sets of signals called signal spaces. Elements of these domains and ranges are considerably more difficult to specify than, say, an element of *Reals* or *Ints*. For this reason, it is almost never reasonable to use a graph or a table to define a system. Much of the rest of this book is devoted to giving precise ways to define systems where some analysis is possible. Here we consider some simple techniques that can be immediately motivated. Then we show how more complicated systems can be constructed from simpler ones using block diagrams. We give a rigorous meaning to these block diagrams so that we can use them without resorting to perilous intuition to interpret them.

Consider a system S where

$$S\colon [D \to R] \to [D' \to R']. \tag{2.13}$$

Suppose that $x \in [D \to R]$ and $y = S(x)$. Then we call the pair (x, y) a **behavior** of the system. A behavior is an input, output pair. The set of all behaviors is

$$\boxed{\textit{Behaviors}(S) = \{(x, y) \mid x \in [D \to R] \text{ and } y = S(x)\}.}$$

Giving the set of behaviors is one way to define a system. Explicitly giving the set *Behaviors*, however, is usually impractical, because it is a huge set, typically infinite (see boxes on pages305 and 306). Thus, we seek other ways of talking about the relationship between a signal x and a signal y when $y = S(x)$.

To describe a system, one must specify its domain (the space of input signals), its range (the space of output signals), and the rule by which the system assigns an output signal to each input signal. This assignment rule is more difficult to describe and analyze than the input and output signals themselves. A table is almost never adequate, for example. Indeed for most systems we do not have effective mathematical tools for describing or understanding their behavior. Thus, it is useful to restrict our system designs to those we can understand. We first consider some simple examples.

2.3.1 Memoryless systems

A system $F : [\mathit{Reals} \to Y] \to [\mathit{Reals} \to Y]$ is **memoryless** if there is a function $f : Y \to Y$ such that

$$\forall\, t \in \mathit{Reals} \text{ and } x \in [\mathit{Reals} \to Y], \quad (F(x))(t) = f(x(t)).$$

Specification of a memoryless system reduces to specification of the function f. If Y is finite, then a table may be adequate.

Example 2.15: Consider a time-domain system with input x and output y, where for all $t \in$ *Reals*,

$$y(t) = x^2(t).$$

This example defines a simple system, where the value of the output signal at each time depends only on the value of the input signal at that time. Such systems are said to be memoryless because you do not have to remember previous values of the input in order to determine the current value of the output.

2.3.2 Differential equations

Consider a class of systems given by functions $S\colon \mathit{ContSignals} \to \mathit{ContSignals}$ where *ContSignals* is a set of **continuous-time signals**. Depending on the scenario, we could have $\mathit{ContSignals} = [\mathit{Time} \to \mathit{Reals}]$ or $\mathit{ContSignals} = [\mathit{Time} \to \mathit{Comps}]$, where $\mathit{Time} = \mathit{Reals}$ or $\mathit{Time} = \mathit{Reals}_+$. These are often called **continuous-time systems** because they operate on continuous-time signals. Frequently, such systems can be defined by **differential equations** that relate the input signal to the output signal.

Example 2.16: Consider a continuous-time system with input x and output $y = F(x)$ such that $\forall\, t \in$ *Reals*,

$$y(t) = \frac{1}{M}\int_{t-M}^{t} x(\tau)d\tau.$$

By a change of variables this can also be written

$$y(t) = \frac{1}{M}\int_{0}^{M} x(t-\tau)d\tau.$$

Once again, this is clearly not memoryless. Such a system also has the effect of smoothing a signal. We will study it and many related systems in detail.

Example 2.17: Consider a particle constrained to move forward or backwards along a straight line with an externally imposed acceleration. We will consider this particle to be a system where the output is its position and the externally imposed acceleration is the input.

Denote the position of the particle by $x\colon \mathit{Time} \to \mathit{Reals}$, where $\mathit{Time} = \mathit{Reals}_+$. By considering only the non-negative reals, we are assuming that the model has a starting time. Denote the acceleration by $a\colon \mathit{Time} \to \mathit{Reals}$. By the definition of acceleration, the two signals are related by the differential equation

$$\forall\, t \in \mathit{Reals}_+, \quad \ddot{x}(t) = a(t),$$

where $\ddot{x}(t)$ denotes the second derivative with respect to time of x. If we know the initial position $x(0)$ and initial speed $\dot{x}(0)$ of the particle at time 0, and if we are given the acceleration a, we can evaluate the position at any t by integrating this differential equation

$$x(t) = x(0) + \dot{x}(0)t + \int_0^t [\int_0^s a(\tau)d\tau]ds. \tag{2.14}$$

We can regard the initial position and velocity as inputs, together with acceleration, in which case the system is a function

$$\mathit{Particle}\colon \mathit{Reals} \times \mathit{Reals} \times [\mathit{Reals}_+ \to \mathit{Reals}] \to [\mathit{Reals}_+ \to \mathit{Reals}],$$

where for any inputs $(x(0), \dot{x}(0), a)$, $x = \mathit{Particle}(x(0), \dot{x}(0), a)$ must satisfy (2.14).

Suppose for example that the input is $(1, -1, a)$ where $\forall\, t \in \mathit{Reals}_+$, $a(t) = 1$. We can calculate the position by carrying out the integration in (2.14) to find that

$$\forall\, t \in \mathit{Reals}_+, \quad x(t) = 1 - t + 0.5t^2.$$

Suppose instead that $x(0) = \dot{x}(0) = 0$ and $\forall\, t \in \mathit{Reals}_+$, $a(t) = \cos(2\pi f t)$, where f is some fixed number, the frequency of the input acceleration. Again, we can carry out the integration to get

$$\int_0^t \int_0^s \cos(2\pi f u)du ds = -\frac{1}{4}\frac{\cos 2t\pi f - 1}{\pi^2 f^2}.$$

Notice that the position of the particle is sinusoidal. Notice further that the amplitude of this sinusoid decreases as f increases. Intuitively, this has to be the case. If the externally imposed acceleration is varying more rapidly back and forth, the particle has less time to respond to each direction of acceleration, and hence its excursion is less. In subsequent chapters, we will study how the response of certain kinds of systems varies with the frequency of the input.

2.3.3 Difference equations

Consider a class of systems given by functions $S\colon \mathit{DiscSignals} \to \mathit{DiscSignals}$ where *DiscSignals* is a set of **discrete-time signals**. Depending on the scenario, we could have $\mathit{DiscSignals} = [\mathit{Ints} \to$

Reals] or *DiscSignals* $=$ [*Ints* $\to$ *Comps*], or even *DiscSignals* $=$ [*Nats*$_0$ $\to$ *Reals*], or *DiscSignals* $=$ [*Nats*$_0$ $\to$ *Comps*]. These are often called **discrete-time systems** because they operate on discrete-time signals. Frequently, such systems can be defined by **difference equations** that relate the input signal to the output signal.

Example 2.18: Consider a system

$$S\colon [\textit{Nats}_0 \to \textit{Reals}] \to [\textit{Nats}_0 \to \textit{Reals}]$$

where for all $x \in [\textit{Nats}_0 \to \textit{Reals}]$, $S(x) = y$ is given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = (x(n) + x(n-1))/2.$$

The output at each index is the average of two of the inputs. This is a simple example of a **moving average** system, where typically more than two input values get averaged to produce an output value.

Suppose that $x = u$, the **unit step** function, defined by

$$\forall\, n \in \textit{Ints}, \quad u(n) = \begin{cases} 1 & \text{if } n \ge 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

We can easily calculate the output y,

$$\forall\, n \in \textit{Ints}, \quad y(n) = \begin{cases} 1 & \text{if } n \ge 1 \\ 1/2 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

The system smoothes the transition of the unit step a bit. In fact, systems are popular on Wall Street for smoothing the daily fluctuations of stock prices in an effort to discern trends in stock prices.

A slightly more interesting input is a sinusoidal signal given by

$$\forall\, n \in \textit{Ints}, \quad x(n) = \cos(2\pi f n).$$

The output is given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = (\cos(2\pi f n) + \cos(2\pi f(n-1)))/2.$$

By trigonometric identities,[8] this can be written as

$$y(n) = R\cos((2\pi f n + \theta)$$

[8]

$$\begin{aligned} A\cos(\theta+\alpha) + B\cos(\theta+\beta) &= C\cos\theta - S\sin\theta \\ &= R\cos(\theta+\phi) \end{aligned}$$

where

$$\begin{aligned} C &= A\cos\alpha + B\cos\beta \\ S &= A\sin\alpha + B\sin\beta \\ R &= \sqrt{C^2+S^2} \\ \phi &= \arctan(S/C) \end{aligned}$$

where

$$\begin{aligned}\theta &= \arctan(\frac{\sin(-2\pi f)}{1+\cos(-2\pi f)})/2, \\ R &= \sqrt{2+2\cos(2\pi f)}\end{aligned}$$

As in the previous example, a sinusoidal input stimulates a sinusoidal output with the same frequency. In this case, the amplitude of the output varies (in a fairly complicated way) as a function of the input frequency. We will examine this phenomenon in more detail in subsequent chapters by studying the *frequency response* of such systems.

Example 2.19: The general form for a moving average is given by

$$\forall\, n \in Ints, \quad y(n) = \frac{1}{M}\sum_{k=0}^{M-1} x(n-k),$$

where x is the input and y is the output. This system is called an M-point **moving average**, since at any n it gives the average of the M most recent values of the input. It computes an average, just like example 2.16 but the integral has been replaced by its discrete counterpart, the sum.

Moving averages are widely used on Wall Street to smooth out momentary fluctuations in stock prices to try to determine general trends. We will study the smoothing properties of this system. We will also study more general forms of difference equations of which the moving average is a special case.

The examples above give declarative definitions of systems. Imperative definitions require giving a procedure for computing the output signal given the input signal. It is clear how to do that with the memoryless system, assuming that an imperative definition of the function f is available, and with the moving average. The integral equation, however, is harder to define imperatively. An imperative description of such systems that is suitable for computation on a computer requires approximation via solvers for differential equations. Simulink, for example, which is part of the Matlab package, provides such solvers. Alternatively, an imperative description can be given in terms of analog circuits or other physical systems that operate directly on the pertinent continuous domain. Discrete-time systems often have reasonable imperative definitions as **state machines**, considered in detail in the next chapter.

2.3.4 Composing systems using block diagrams

We have been using **block diagrams** informally to describe systems. A block diagram is a **visual syntax** for describing a system as an interconnection of other (component) systems, each of which emphasizes one particular input-to-output transformation of a signal. A block diagram is a collection of blocks interconnected by arrows. Arrows are labeled by signals. Each block represents an individual system that transforms an incoming or **input signal** into an outgoing or **output signal**.

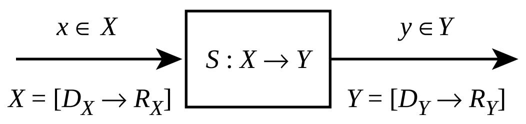


Figure 2.4: The simplest block diagram represents a function S that maps an input signal $x \in X$ to an output signal $y \in Y$. The domain and range of the input are D_X and R_X, repectively, and of the output, D_Y and R_Y.

For example, *modulator* in figure 1.18 is a system that transforms a bit sequence into a voice-like signal.

We can use function composition, as discussed in section 2.1.5, to give a precise meaning to block diagrams. A block represents a function, and the connection of an output from one block to the input of another represents the composition of their two functions. The only syntactic requirement for interconnecting two blocks is that the output of the first block must be an acceptable input for the second. Thus, for example, the output of *modulator* in figure 1.18 is a voice-like signal, which is precisely what the system *telephone network* is able to accept as an input.

Block diagrams can be more readable than symbolic function composition, particularly for complicated interconnections. They also offer a natural hierarchy, where we can combine blocks to hide certain signals and component systems and to emphasize others. For instance, the *telephone network* in figure 1.18 hides the details shown in figure 1.14. This emphasizes the POTS capabilities of the telephone network, while hiding its other features.

The simplest block diagram has a single block, as in figure 2.4. The block represents a system with input signal x and output signal y. Here, x denotes a variable over the set X, and y denotes a variable over the set Y. The system is described by the function $S\colon X \to Y$. Both X and Y are sets of functions or signals. Therefore the variables x and y themselves denote functions.

Cascade Composition

The voice path in (2.7) is an example of cascade composition of functions. In general, a system obtained by a **cascade composition** of two blocks is given by the composition of the functions describing those blocks. In figure 2.5 the function S describes the system obtained by connecting the systems S_1 and S_2, with $S = S_2 \circ S_1$, i.e.

$$\forall\, x \in X, \quad S(x) = S_2(S_1(x)).$$

The combined system has input signal x, output signal z, and internal signal y. Of course, the range of the first system must be be contained by the domain of the second for this composition to make sense. In the figure, the typical case is shown where this range and domain are the same.

Probing further: Composition of graphs

We suggest a general method for writing down a declarative specification of the interconnected system S in figure 2.5 in terms of the subsystems S_1 and S_2 and the **connection restriction** that the output of S_1 be acceptable as an input of S_2.

We describe S_1 and S_2 by their graphs,

$$\textit{graph}(S_1) = \{(x, y_1) \in X \times Y \mid y_1 = S_1(x)\},$$

$$\textit{graph}(S_2) = \{(y_2, z) \in Y \times Z \mid z = S_2(y_2)\},$$

and we specify the connection restriction as the predicate

$$y_1 = y_2.$$

Note that we use different dummy variables y_1 and y_2 to distinguish between the two systems and the connection restriction.

The graph of the combined system S is then given by

$$\begin{aligned}\textit{graph}(S) \quad = \quad & \{(x, z) \in X \times Z \mid \exists y_1, \exists y_2 \\ & \quad (x, y_1) \in \textit{graph}(S_1) \wedge (y_2, z) \in \textit{graph}(S_2) \wedge y_1 = y_2\}.\end{aligned}$$

Here, $\wedge$ denotes logical conjunction, "and." It is now straightforward to show that $\textit{graph}(S) = \textit{graph}(S_2 \circ S_1)$ so that $S = S_2 \circ S_1$.

In the case of the cascade composition of figure 2.5 this elaborate method is unnecessary, since we can write down $S = S_2 \circ S_1$ simply by inspecting the figure. For feedback connections, we may not be able to write down the combined system directly.

There are three other reasons to understand this method. First, we use it later to obtain a description of interconnected state machines from their component machines. Second, this method is used to describe electronic circuits. Third, if we want a computer to figure out the description of the interconnected system from a description of the subsystems and the connection restrictions, we have to design an algorithm that the computer must follow. Such an algorithm can be based on this general method.

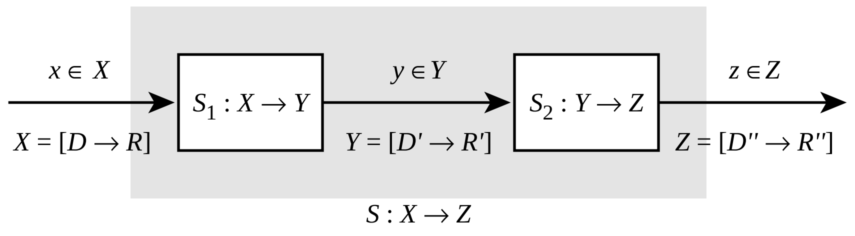


Figure 2.5: The cascade composition of the two systems is described by $S = S_2 \circ S_1$.

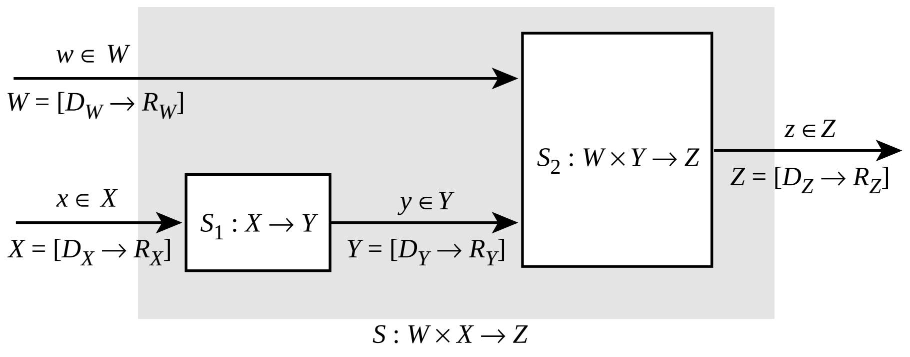


Figure 2.6: The combined system with input signals x, w and output signal z is described by the function S, where $\forall\ (x, w), \quad S(x, w) = S_2(w, S_1(x))$.

More complex block diagrams

Consider two more block diagrams with slightly more complicated structure. Figure 2.6 is similar to figure 2.5. The system described by S_1 is the same as before, but the system described by S_2 has a pair of input signals $(w, y) \in W \times Y$. The combined system has the pair $(x, w) \in X \times W$ as input signal, z as output signal, y as internal signal, and it is described by the function $S: X \times W \to Z$, where

$$\forall (x, w) \in X \times W, \quad S(x, w) = S_2(w, S_1(x)). \tag{2.16}$$

The system of figure 2.7 is obtained from that of figure 2.6 by connecting the output signal z to the input signal w. As a result the new system has input signal x, output signal z, internal signals y and w, and it is described by the function $S': X \to Z$, where

$$\forall\ x \in X, \quad S'(x) = S_2(S'(x), S_1(x)). \tag{2.17}$$

The connection of z to w is called a **feedback** connection because the output z is fed back as input w. Of course, such a connection has meaning only if Z, the range of S_2, is a subset of W.

There is one enormous difference between (2.16) and (2.17). Expression (2.16) serves as a definition of the function S: to every (x, w) in its domain S assigns the value given by the right-hand side

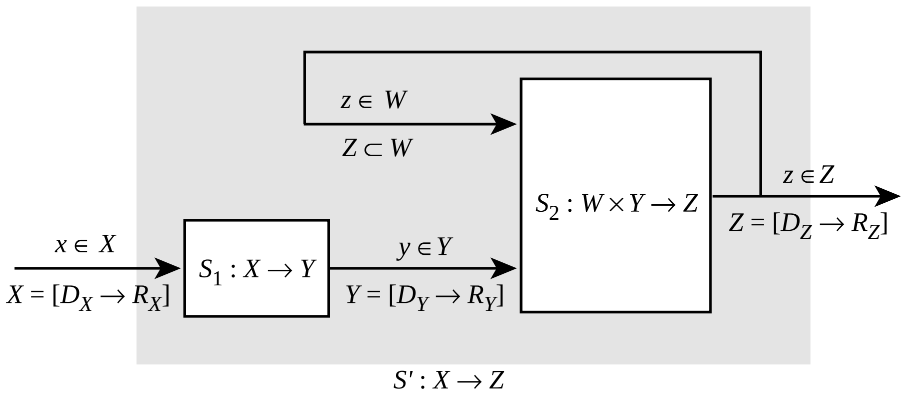


Figure 2.7: The combined system is described by the function S', where $S'(x) = S_2(S'(x), S_1(x))$.

which is uniquely determined by the given functions S_1 and S_2. But in expression (2.17) the value $S'(x)$ assigned to x may not be determined by the right-hand side of (2.17), since the right-hand side itself depends on $S'(x)$. In other words, (2.17) is an *equation* that must be solved to determine the value of $S'(x)$ for a given x; i.e. $S'(x) = y$ where y is a solution of

$$y = S_2(y, S_1(x)). \tag{2.18}$$

Such a solution, if it exists, is called a **fixed point**. We now face the difficulty that this equation may have no solution, exactly one solution, or several solutions. Another difficulty is that the value y that solves (2.18) is not a number but a function. So it will not be easy to solve such an equation. Since feedback connections always arise in control systems, we will study how to solve them. We will first solve them in the context of state machines, which are introduced in the next chapter.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, and requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** The broadcast signal of an AM radio station located at 110 on your dial has a carrier frequency of 110 kHz. An AM signal that includes the carrier has the form

$$\forall\, t \in \mathit{Time}, \quad \mathit{AMSignal}(t) = (1 + m(t)) \sin(2\pi \times 110,000t),$$

where m is an audio signal like *Voice* in figure 1.1, except that $\forall\, t \in \mathit{Time}$, $|m(t)| < 1$. Since you cannot easily plot such a high frequency signal, give an expression for and plot *AMSignal* (using Matlab) for the case where $\mathit{Time} = [0, 1]$, $m(t) = \cos(\pi t)$, and the carrier frequency is 20 Hz.

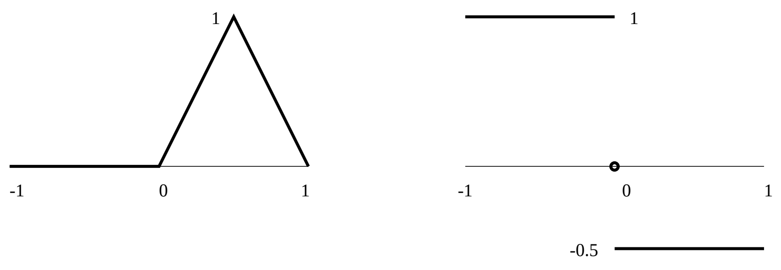


Figure 2.8: Graphs of two functions. The bold line is the graph.

2. **T** This problem studies the relationship between the notion of **delay** and the graph of a function.

   (a) Consider two functions f and g from *Reals* into *Reals* where $\forall\, t \in$ *Reals*, $f(t) = t$ and $g(t) = f(t - t_0)$, where t_0 is a fixed number. Sketch a plot of f and g for $t_0 = 1$ and $t_0 = -1$. Observe that if $t_0 > 0$ then *graph*(g) is obtained by moving *graph*(f) to the right, and if $t_0 < 0$ by moving it to the left.

   (b) Show that if f: *Reals* $\to$ *Reals* is any function whatsoever, and $\forall\, t,\ \ g(t) = f(t - t_0)$, then if $(t, y) \in$ *graph*(f), then $(t + t_0, y) \in$ *graph*(g). This is another way of saying that if $t_0 > 0$ then the graph is moved to the right, and if $t_0 < 0$ then the graph is moved to the left.

   (c) If t represents time, and if $t_0 > 0$, we say that g is obtained by *delaying* f. Why is it reasonable to say this?

3. **E** Figure 2.8 shows graphs of two different functions of the form $f\colon [-1, 1] \to [-1, 1]$. For each case, define f by giving an algebraic expression for its value at each point in its domain. This expression will have several parts, similar to the definition of the *signum* function in (2.3). Note that $f(0) = 0$ for the graph on the right. Call f_l the function on the left and f_r the function on the right. Plot *graph*$(f_l \circ f_r)$ and *graph*$(f_r \circ f_l)$.

4. **T** Let $X = \{a, b, c\}$, $Y = \{1, 2\}$. For each of the following subsets $G \subset X \times Y$, determine whether G is the graph of a function from X to Y, and if it is, describe the function as a table.

   (a) $G = \{(a, 1), (b, 1), (c, 2)\}$

   (b) $G = \{(a, 1), (a, 2), (b, 1), (c, 2)\}$

   (c) $G = \{(a, 1), (b, 2)\}$

5. **C** A router in the Internet is a switch with several input ports and several output ports. A packet containing data arrives at an input port at an arbitrary time, and the switch forwards the packet to one of the outgoing ports. The ports of different routers are connected by transmission links. When a packet arrives at an input port, the switch examines the packet, extracting from it a destination address d. The switch then looks up the output port in its routing table, which contains entries of the form $(d, outputPort)$. It then forwards the packet to the specified output port. The Internet works by setting up the routing tables in the routers.

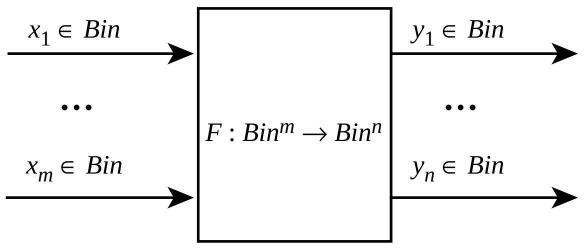


Figure 2.9: The logic circuit has m binary inputs $(x_1, \cdots, x_m)$ and n binary outputs $(y_1, \cdots, y_n)$.

Consider a simplified router with one input port and and two output ports, named O_1, O_2. Let D be the set of destination addresses.

(a) Explain why the routing table can be described as a subset $T \subset D \times \{O_1, O_2\}$.

(b) Is it reasonable to constrain T to be the graph of a function from $D \to \{O_1, O_2\}$? Why?

(c) Assume the signal at the input port is a sequence of packets. How would you describe the space of input signals to the router and output signals from the router?

(d) How would you describe the switch as a function from the space of input signals to the space of output signals?

6. **C** For each of the following expressions, state whether it can be interpreted as an assignment, an assertion, or a predicate. More than one choice may be valid because the full context is not supplied.

   (a) $x = 5$,

   (b) $A = \{5\}$,

   (c) $x > 5$,

   (d) $3 > 5$,

   (e) $x > 5 \wedge x < 3$.

7. **T** A logic circuit with m binary inputs and n binary outputs is shown in figure 2.9. It is described by a function $F\colon X \to Y$ where $X = Bin^m$ and $Y = Bin^n$. (In a circuit, the signal values 1 and 0 in *Bin* correspond to voltage *High* and *Low*, respectively.) How many such distinct logic functions F are there?

8. **T** The following system S takes a discrete-time signal $x \in X$ and transforms it into a discrete-time signal $y \in Y$ whose value at index n is the average of the previous 4 values of x. Such a system is called a **moving average**. Suppose that $X = Y = [Nats \to Reals]$, where *Nats* is the set of natural numbers. More precisely, the system is described by the function S such that for any $x \in X$, $y = S(x)$ is given by

$$y(n) = \begin{cases} [x(1) + \cdots + x(n)]/4 & \text{for } 1 \le n < 4 \\ [x(n-3) + x(n-2) + x(n-1) + x(n)]/4 & \text{for } n \ge 4 \end{cases}$$

Notice that the first three samples are averages only if we assume that samples prior to those that are available have value zero. Thus, there is an initial **transient** while the system collects enough data to begin computing meaningful averages.

Write a Matlab program to calculate and plot the output signal y for time $1 \le n \le 20$ for the following input signals:

(a) x is a unit step delayed by 10, i.e. $x(n) = 0$ for $n \le 9$ and $x(n) = 1$ for $n \ge 10$.

(b) x is a unit step delayed by 15.

(c) x alternates between +1 and -1, i.e. $x(n) = 1$ for n odd, and $x(n) = -1$ for n even. **Hint**: Try computing $\cos(\pi n)$ for $n \in \mathit{Nats}_0$.

(d) Comment on what this system does. Qualitatively, how is the output signal different from the input signal?

9. **T** The following system is similar to the one in the previous problem, but time is continuous. Now $X = Y = [\mathit{Reals} \to \mathit{Reals}]$ and the system $F\colon X \to Y$ is defined as follows. For all $x \in X$ and $t \in \mathit{Reals}$

$$(F(x))(t) = \frac{1}{10}\int_{t-10}^{t} x(s)ds$$

Show that if x is the sinusoidal signal

$$\forall\, t \in \mathit{Reals} \quad x(t) = \sin(\omega t),$$

then y is also sinusoidal

$$\forall\, t \in \mathit{Reals}, \quad y(t) = A\sin(\omega t + \phi).$$

You do not need to give precise expressions for A and ϕ, but just show that the result has this form. Also, show that as ω gets large, the amplitude of the output gets small. Higher frequencies, which represent more abrupt changes in the input, are more attenuated by the system than lower frequencies.

Hint: The following fact from calculus may be useful:

$$\int_a^b \sin(\omega s)ds = \frac{1}{\omega}(\cos(\omega a) - \cos(\omega b)).$$

Also, the identity in the footnote on page 55 might be useful to show that the output is sinusoidal with the appropriate frequency.

10. **E** Suppose that $f\colon \mathit{Reals} \to \mathit{Reals}$ and $g\colon \mathit{Reals} \to \mathit{Ints}$ such that for all $x \in \mathit{Reals}$,

$$g(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

and

$$f(x) = 1 + x.$$

(a) Define $h = g \circ f$.

(b) Suppose that

$$\begin{aligned} F\colon [\textit{Reals} \to \textit{Reals}] &\to [\textit{Reals} \to \textit{Reals}] \\ G\colon [\textit{Reals} \to \textit{Reals}] &\to [\textit{Reals} \to \textit{Ints}] \end{aligned}$$

such that for all $s \in [\textit{Reals} \to \textit{Reals}]$ and $x \in \textit{Reals}$,

$$\begin{aligned} (F(s))(x) &= f(s(x)) \\ (G(s))(x) &= g(s(x)) \end{aligned}$$

where f and g are as given above. Sketch a block diagram for $H = G \circ F$, where you have one block for each of G and F. Label the inputs and outputs of your blocks with the domain and range of the functions in the blocks.

(c) Let $s \in [\textit{Reals} \to \textit{Reals}]$ be such that for all $x \in \textit{Reals}$,

$$s(x) = \cos(\pi x).$$

Define u where

$$u = (G \circ F)(s).$$

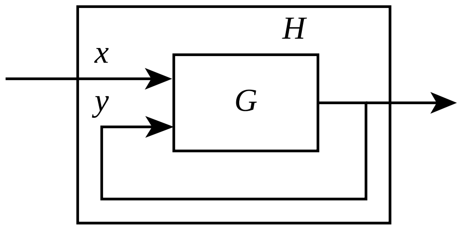
11. **T** Let $D = \textit{DiscSignals} = [\textit{Ints} \to \textit{Reals}]$ and let

$$G\colon D \times D \to D$$

such that for all $x, y \in D$ and for all $n \in \textit{Ints}$,

$$(G(x, y))(n) = x(n) - y(n-1).$$

Now suppose we construct a new system H as follows:



Define H (as much as you can).

Chapter 3

State-Space Models

Systems are functions that transform signals. The domain and the range of these functions are both signal spaces, which significantly complicates specification of the functions. A broad class of systems can be characterized using the concept of **state** and the idea that a system evolves through a sequence of changes in state, or **state transitions**. Such characterizations are called **state-space models**.

A state-space model describes a system procedurally, giving a sequence of step-by-step operations for the evolution of a system. It shows how the input signal drives changes in state, and how the output signal is produced. It is thus an imperative description. Implementing a system described by a state-space model in software or hardware is straightforward. The hardware or software simply needs to sequentially carry out the steps given by the model. Conversely, given a piece of software or hardware, it is often useful to describe it using a state-space model, which yields better to analysis than more informal descriptions.

In this chapter, we introduce state-space models by discussing systems with a finite (and relatively small) number of states. Such systems typically operate on event streams, often implementing control logic. For example, the decision logic of modem negotiation described in chapter1 can be modeled using a finite state model. Such a model is much more precise than the English-language descriptions that are commonly used for such systems.

3.1 State machines

A description of a system as a function involves three entities: the set of input signals, the set of output signals, and the function itself, $F\colon InputSignals \to OutputSignals$. For a **state machine**, the input and output signals have the form

$$EventStream\colon Nats_0 \to Symbols,$$

where $Nats_0 = \{0, 1, 2, \cdots\}$, and *Symbols* is an arbitrary set. The domain of these signals represents *ordering* but not necessarily time (neither discrete nor continuous time). The ordering of the domain means that we can say that one event occurs *before* or *after* another event. But we cannot say how much time elapsed between these events. In chapter 5 we will study how state-space models can be used with functions of time.

A state machine constructs the output signal one element at a time by observing the input signal one element at a time. Specifically, a state machine *StateMachine* is a 5-tuple,

$$\mathit{StateMachine} = (\mathit{States}, \mathit{Inputs}, \mathit{Outputs}, \mathit{update}, \mathit{initialState}) \tag{3.1}$$

where $\mathit{States}, \mathit{Inputs}, \mathit{Outputs}$ are sets, *update* is a function, and $\mathit{initialState} \in \mathit{States}$. The meaning of these names is:

States is the **state space**,
Inputs is the **input alphabet**,
Outputs is the **output alphabet**,
$\mathit{initialState} \in \mathit{States}$ is the **initial state**, and
$\mathit{update}\colon \mathit{States} \times \mathit{Inputs} \to \mathit{States} \times \mathit{Outputs}$ is the **update function**.

This five-tuple is called the **sets and functions model** of a state machine.

Inputs and *Outputs* are the sets of possible input and output values. The set of **input signals** consists of all infinite sequences of input values,

$$\mathit{InputSignals} = [\mathit{Nats}_0 \to \mathit{Inputs}].$$

The set of **output signals** consists of all infinite sequences of output values,

$$\mathit{OutputSignals} = [\mathit{Nats}_0 \to \mathit{Outputs}].$$

Let $x \in \mathit{InputSignals}$ be an input signal. A particular element in the signal can be written $x(n)$ for any $n \in \mathit{Nats}_0$. We write the entire input as a sequence

$$(x(0), x(1), \cdots, x(n), \cdots).$$

This sequence defines the function x in terms of elements $x(n) \in \mathit{Inputs}$, which represent particular input values.

We reiterate that the index n in $x(n)$ does not refer to time, but rather to the **step number**. This is an **ordering constraint** only: step n occurs after step $n-1$ and before step $n+1$. The state machine evolves (i.e. moves from one state to the next) in **steps**.[1]

[1] Of course the steps could last a fixed duration of time, in which case there would be a simple relationship between step number and time. The relationship may be a mixed one, where some inputs are separated by a fixed amount of time and some are not.

3.1.1 Updates

The interpretation of *update* is this. If $s(n) \in \textit{States}$ is the current state at step n, and $x(n) \in \textit{Inputs}$ is the current input, then the current output and the next state are given by

$$(s(n+1), y(n)) = \textit{update}(s(n), x(n)).$$

Thus the *update* function makes it possible for the state machine to construct the output signal step by step by observing the input signal step by step.

The state machine *StateMachine* of (3.1) defines a function

$$F \colon \textit{InputSignals} \to \textit{OutputSignals} \tag{3.2}$$

such that for any input signal $x \in \textit{InputSignals}$ the corresponding output signal is $y = F(x)$. However, it does much more than just define this function. It also gives us a procedure for evaluating this function on a particular input signal. The **state response** $(s(0), s(1), \cdots)$ and output y are constructed as follows:

$$\begin{aligned} s(0) &= \textit{initialState}, & (3.3)\\ \forall n \geq 0,\ (s(n+1), y(n)) &= \textit{update}(s(n), x(n)), & (3.4) \end{aligned}$$

Observe that if the initial state is changed, the function F will change, so the initial state is an essential part of the definition of a state machine.

Each evaluation of (3.4) is called a **reaction** because it defines how the state machine reacts to a particular input symbol. Note that exactly one output symbol is produced for each input symbol. Thus, it is not necessary to have access to the entire input sequence to start producing output symbols. This feature proves extremely useful in practice, since it is usually impractical to have access to the entire input sequence (it is infinite in size!). The procedure summarized by (3.3)–(3.4) is **causal**, in that the next state $s(n+1)$ and current output $y(n)$ depend only on the initial state $s(0)$ and current and past inputs $x(0), x(1), \cdots, x(n)$.

3.1.2 Stuttering

A state machine produces exactly one output symbol for each input symbol. For each input symbol, it may also change state (of course, it could also remain in the same state by changing back to the same state). This means that with no input symbol, there is neither an output symbol nor a change of state.

Later, when we compose simpler state machines to construct more complicated ones, it will prove convenient to be explicit in the model about the fact that no input triggers no output and no state change. We do that by insisting that the input and output symbol sets include a **stuttering element**, typically denoted *absent*. That is,

$$\textit{absent} \in \textit{Inputs}, \text{ and } \textit{absent} \in \textit{Outputs}.$$

Moreover, we require that for any $s \in \textit{States}$,

$$\textit{update}(s, \textit{absent}) = (s, \textit{absent}). \tag{3.5}$$

This is called a **stuttering reaction** because no progress is made. An absent input triggers an absent output and no state change. Now any number of *absent* elements may be inserted into the input sequence, anywhere, without changing the non-absent outputs.

Example 3.1: Consider a 60-minute parking meter. There are three (non-stuttering) inputs: *in5* and *in25* which represent feeding the meter 5 and 25 cents respectively, and *tick* which represents the passage of one minute. The meter displays the time in minutes remaining before the meter expires. When *in5* occurs, this time is incremented by 5, and when *in25* occurs it is incremented by 25, up to a maximum of 60 minutes. When *tick* occurs, the time is decremented by 1, down to a minimum of 0. When the remaining time is 0, the display reads *expired*.

We can construct a finite-state machine model for this parking meter. The set of states is

$$\textit{States} = \{0, 1, 2, ..., 60\}.$$

The input and output alphabets are

$$\textit{Inputs} = \{\textit{in5}, \textit{in25}, \textit{tick}, \textit{absent}\}.$$

$$\textit{Outputs} = \{\textit{expired}, 1, 2, ..., 60, \textit{absent}\}.$$

The initial state is

$$\textit{initialState} = 0.$$

The update function

$$\textit{update} \colon \textit{States} \times \textit{Inputs} \to \textit{States} \times \textit{Outputs}$$

is given by, $\forall\ s \in \textit{States},\ x \in \textit{Inputs}$,

$$\textit{update}(s, x) = \begin{cases} (0, \textit{expired}) & \text{if} \quad x = \textit{tick} \wedge (s = 0 \vee s = 1) \\ (s-1, s-1) & \text{if} \quad x = \textit{tick} \wedge s > 1 \\ (\min(s+5, 60), \min(s+5, 60)) & \text{if} \quad x = \textit{in5} \\ (\min(s+25, 60), \min(s+25, 60)) & \text{if} \quad x = \textit{in25} \\ (s, \textit{absent}) & \text{if} \quad x = \textit{absent} \end{cases}$$

where min is a function that returns the minimum of its arguments.

If the input sequence is $\textit{in25}, \textit{tick}^{20}, \textit{in5}, \textit{tick}^{10}, ...$, for example, then the output sequence is[2]

$$\textit{expired}, 25, 24, ..., 6, 5, 10, 9, 8, ...2, 1, \textit{expired}.$$

[2]We are using the common notation $\textit{tick}^{10}$ to mean a sequence of 10 consecutive *tick*s.

3.2 Finite state machines

Often, *States* is a finite set. In this case, the state machine is called a **finite state machine**, abbreviated **FSM**. FSMs yield to powerful analytical techniques because, in principle, it is possible to explore all possible sequences of states. The parking meter example above is a finite state machine.

When the number of states is small, and the input and output alphabets are finite (and small), we can describe the state machine using a very readable and intuitive diagram called a **state transition diagram**. (Figure 3.6(a) is the state machine diagram of the parking meter.)

> **Example 3.2:** A verbal description of an automatic telephone answering machine might go like this.
>
> > When a call arrives, the phone rings. If the phone is not picked up, then on the third ring, the machine answers. It plays a pre-recorded greeting requesting that the caller leave a message ("Hello, sorry I can't answer your call right now ... Please leave a message after the beep"), then records the caller's message, and then automatically hangs up. If the phone is answered before the third ring, the machine does nothing.
>
> Figure 3.1 shows a state transition diagram for the state machine model of this answering machine.

Without any further explanation, the reader can probably read the diagram in figure3.1. It is sufficiently intuitive. Nonetheless, we will explain it precisely.

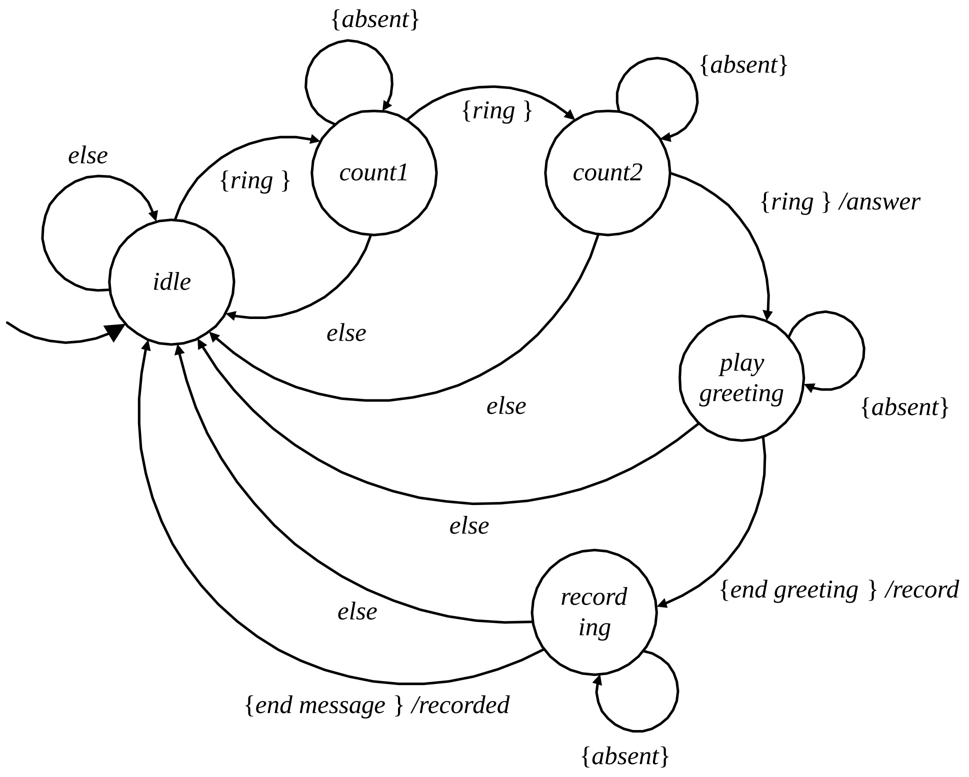
3.2.1 State transition diagrams

Figure 3.1 consists of bubbles linked by arcs. (The arcs are also called arrows.) In this bubbles-and-arcs syntax each bubble represents one state of the answering machine, and each arc represents a **transition** from one state to another. The bubbles and arcs are annotated, i.e. they are labeled with some text. The execution of the state machine consists of a sequence reactions, where each reaction involves a transition from one state to another (or back to the same state) along one of the arcs. The tables at the bottom of the figure are not part of the state transition diagram, but they improve our understanding of the diagram by giving the meanings of the names of the states, inputs, and outputs.

The notation for state transition diagrams, which we will fully explain in this chapter and the next one, is summarized in figure 3.2. Each bubble is labeled with the name of the **state** it represents. The state names can be anything, but they must be distinct. The state machine of figure3.1 has five states. The state names define the state space,

$$States = \{idle, count1, count2, play\ greeting, recording\}.$$

Each arc is labeled by a **guard** and (optionally) an **output**. If an output is given, it is separated from the guard by a slash, as in the example $\{ring\}/answer$ going from state *count2* to *play greeting*. A

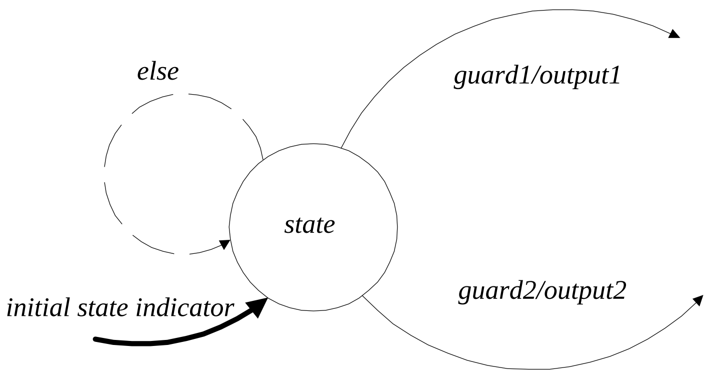


states
idle: nothing is happening
count1: one ring has arrived
count2: two rings have arrived
play greeting: playing the greeting message
recording : recording the message

inputs
ring - incoming ringing signal
offhook - a telephone extension is picked up
end greeting - greeting message is finished playing
end message - end of message detected (e.g. dialtone)
absent - no input of interest.

outputs
answer - answer the phone and start the greeting message
record - start recording the incoming message
recorded- recorded an incoming message
absent - default output when there is nothing interesting to say

Figure 3.1: State transition diagram for the telephone answering machine.



state machine:
$(\textit{States}, \textit{Inputs}, \textit{Outputs}, \textit{update}, \textit{initialState})$
$\textit{update}\colon \textit{States} \times \textit{Inputs} \to \textit{States} \times \textit{Outputs}$
$\textit{initialState} \in \textit{States}$

elements:
$\textit{state} \in \textit{States}$
$\textit{output1}, \textit{output2} \in \textit{Outputs}$
$\textit{guard1}, \textit{guard2} \subset \textit{Inputs}$
$\textit{else} = \{x \in \textit{Inputs} \mid x \notin (\textit{guard1} \cup \textit{guard2})\}$

determinacy: (There at most one possible reaction to an input)
$\textit{guard1} \cap \textit{guard2} = \emptyset$

Figure 3.2: Summary of notation in state transition diagrams, shown for a single state with two outgoing arcs and one self loop.

guard specifies which inputs might trigger the associated transition. It is a subset of the *Inputs*, the input alphabet, which for the answering machine is

$$\textit{Inputs} = \{\textit{ring}, \textit{offhook}, \textit{end greeting}, \textit{end message}, \textit{absent}\}.$$

In figure 3.1, some guards are labeled "*else*." This special notation designates an arc that is taken when there is no match on any other guard emerging from a given state. The arc with the guard *else* is called the **else arc**. Thus, *else* is the set of all inputs not included in any other guard emerging from the state. For the example in figure 3.2, *else* is defined by

$$\textit{else} = \{x \in \textit{Inputs} \mid x \notin (\textit{guard1} \cup \textit{guard2})\}.$$

In general, for a given state, *else* is the complement with respect to *Inputs* of the union of the guards on emerging arcs. If no *else* arc is specified, and the set *else* is not empty, then the *else* arc is implicitly a **self loop**, as shown by the dashed arc in figure 3.2. A self loop is an arc that transitions back to the same state. When the else arc is a self loop, then the stuttering element may be a member of the set *else*.

Initially, the system of figure 3.1 is in the *idle* state. The **initial state** is indicated by the bold arc on the left that leads into the state *idle*. Each time an input arrives, the state machine reacts. It checks the guards on arcs going out of the current state and determines which of them contains the input. It then takes that transition.

Two problems might occur.

- The input may not be contained in the guard of any outgoing arc. In our state machine models, for every state, there is at least one outgoing transition that matches the input (because of the *else* arc). This property is called **receptiveness**; it means that the machine can always react to an input. That is, there is always a transition out of the current state that is enabled by the current input. (The transition may lead back to current state if it is a self loop.) Our state machines are said to be **receptive**.

- More than one guard going out from the current state may contain the input. A state machine that has such a structure is said to be **nondeterministic**. The machine is free to choose any arc whose guard contains the input, so more than one behavior is possible for the machine. Nondeterministic state machines will be discussed further below. Until then, we assume that the guards are always defined to give deterministic state machines. Specifically, the guards on outgoing arcs from any state are mutually exclusive. In other words, the intersection of any two guards on outgoing arcs of a state is empty, as indicated in figure 3.2. Of course, by the definition of the *else* set, for any *guard* that is not *else*, it is true that $\textit{guard} \cap \textit{else} = \emptyset$.

A sequence of inputs thus triggers a sequence of state transitions. The resulting sequence of states is called the **state response**.

Example 3.3: In figure 3.1, if the input sequence is

$$(\textit{ring}, \textit{ring}, \textit{offhook}, \cdots)$$

then the state response is

$$(idle, count1, count2, idle, \cdots).$$

The ellipsis ("$\cdots$") are there because the answering machine generally responds to an infinite input sequence, and we are showing only the beginning of that response. This behavior can be compactly represented by a **trace**,

$$idle \xrightarrow{ring} count1 \xrightarrow{ring} count2 \xrightarrow{offhook} idle \cdots$$

A trace represents the state response together with the input sequence that triggers it. This trace describes the behavior of the answering machine when someone picks up a telephone extension after two rings.[3]

A more elaborate trace illustrates the behavior of the answering machine when it takes a message:

$$\begin{aligned} &idle \xrightarrow{ring} count1 \xrightarrow{ring} count2 \xrightarrow{ring} play\ greeting \\ &\xrightarrow{end\ greeting} recording \xrightarrow{end\ message} idle \quad \cdots \end{aligned} \tag{3.6}$$

A state machine also produces outputs. In figure 3.1, the output alphabet is

$$Outputs = \{answer, record, recorded, absent\}.$$

An output is produced as part of a reaction. The output that is produced is indicated after a slash on an arc. If the arc annotation shows no output, then the output is **absent**.

Example 3.4: The output sequence for the trace (3.6) is

$$(absent, absent, answer, record, recorded, \cdots).$$

There is an output symbol for every input symbol, and some of the output symbols are *absent*.

It should be clear how to obtain the state response and output sequence for any input sequence. We begin in the initial state and then follow the state transition diagram to determine the successive state transitions for successive inputs. Knowing the sequence of transitions, we also know the sequence of outputs.

[3]When you lift the handset of a telephone to answer, your phone sends a signal called 'offhook' to the telephone switch. The reason for the name 'offhook' is that in the earliest telephone designs, the handset hung from a hook on the side of the phone. In order to answer, you had to pick the handset off the hook. When you finished your conversation you replaced the handset on the hook, generating an 'onhook' signal. The onhook signal is irrelevant to the answering machine, so it is not included in the model.

Shorthand

State transition diagrams can get very verbose, with many arcs with complicated labels. A number of shorthand options can make a diagram clearer by reducing the clutter.

- If no guard is specified on an arc, then that transition is always taken when the state machine reacts and is in the state from which arc emerges, as long as the input is not the stuttering element. That is, giving no guard is equivalent to giving the entire set *Inputs* as a guard, minus the stuttering element. The stuttering element, recall, always triggers a transition back to the same state, and always produces a stuttering element on the output.
- Any clear notation for specifying subsets can be used to specify guards. For example, if *Inputs* $= \{a, b, c\}$, then the guard $\{b, c\}$ can be given by $\neg a$ (read "not a").
- An *else* transition for a state need not be given explicitly. It is an implied self-loop if it is not given. This is why it is shown with a dashed line in figure 3.2. The output is the stuttering element of *Outputs* if it is not given.

These shorthand notations are not always a good idea. For example, the *else* transitions often correspond to exceptional (unexpected) input sequences, and staying in the same state might not be the right behavior. For instance, in figure 3.1, all *else* transitions are shown explicitly, and all exceptional input sequences result in the machine ending up in state *idle*. This is probably reasonable behavior, allowing the machine to recover. Had we left the *else* transitions implicit, we would likely have ended up with less reasonable behavior. Use your judgment in deciding whether or not to explicitly include *else* transitions.

3.2.2 Update table

An alternative way to describe a finite state machine is by an **update table**. This is simply a tabular representation of the state transition diagram.

For the diagram of figure 3.1, the table is shown in figure 3.3. The first column lists the current state. The remaining columns list the next state and the output for each of the possible inputs.

The first row, for example, corresponds to the current state *idle*. If the input is *ring*, the next state is *count1* and the output is *absent*. Under any of the other inputs, the state remains *idle* and the output remains *absent*.

Types of State Machines

The type of state machines introduced in this section are known as **Mealy machines**, after G. H. Mealy, who studied them in 1955. Their distinguishing feature is that outputs are associated with state transitions. That is, when a transition is taken, an output is produced. Alternatively, we could have associated outputs with states, resulting in a model known as **Moore machines**, after F. Moore, who studied them in 1956. In a Moore machine, an output is produced while the machine is in a

current state	(*next state*, *output*) under specified input				
	ring	*offhook*	*end greeting*	*end message*	*absent*
idle	(*count1*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)
count1	(*count2*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*count1*, *absent*)
count2	(*play greeting*, *answer*)	(*idle*, *absent*)	(*idle*, *absent*)	(*idle*, *absent*)	(*count2*, *absent*)
play greeting	(*idle*, *absent*)	(*idle*, *absent*)	(*recording*, *record*)	(*idle*, *absent*)	(*play greeting*, *absent*)
recording	(*idle*, *recorded*)	(*idle*, *absent*)	(*idle*, *recorded*)	(*idle*, *recorded*)	(*recording*, *absent*)

Figure 3.3: Update table for the telephone answering machine specifies next state and current output as a function of current state and current input.

particular state. Mealy machines turn out to be more useful when they are composed synchronously, as we will do in the next chapter. This is the reason that we choose this variant of the model.

It is important to realize that state machine models, like most models, are not unique. A great deal of engineering judgment goes into a picture like figure 3.1, and two engineers might come up with very different pictures for what they believe to be the same system. Often, the differences are in the amount of detail shown. One picture may show the operation of a system in more detail than another. The less detailed picture is called an **abstraction** of the more detailed picture. Also likely are differences in the names chosen for states, inputs and outputs, and even in the meaning of the inputs and outputs. There may be differences in how the machine responds to exceptional circumstances (input sequences that are not expected). For example, what should the answering machine do if it gets the input sequence $(\textit{ring}, \textit{end greeting}, \textit{end message})$? This probably reflects a malfunction in the system. In figure 3.1, the reaction to this sequence is easy to see: the machine ends up in the *idle* state.

Given these likely differences, it becomes important to be able to talk about **abstraction relations** and **equivalence relations** between state machine models. This turns out to be a fairly sophisticated topic, one that we touch upon below in section 3.3.

The meaning of state

We discussed three equivalent ways of describing a state machine: sets and functions, the state transition diagram, and the update table. These descriptions have complementary uses. The table makes obvious the sparsity of outputs in the answering machine example. The table and the diagrams are both useful for a human studying the system to follow its behavior in different circumstances. The sets and functions and the table are useful for building the state machine in hardware or software.

The sets and functions description is also useful for mathematical analysis.

Of course, the tables and the transition diagram can be used only if there are finitely many states and finitely many inputs and outputs, i.e. if the sets *States*, *Inputs*, and*Outputs* are finite. The sets and functions description is often equally comfortable with finite and infinite state spaces. We will discuss infinite-state systems in chapter 5.

Like any state machine, the telephone answering machine is a **state-determined** system. Once we know its current state, we can tell what its future behavior is for any future inputs. We do not need to know what inputs in the past led to the current state in order to predict how the system will behave in the future. In this sense we can say the current state of the system summarizes the past history of the system. This is, in fact, the key intuitive notion of state.

The number of states equals the number of patterns we need to summarize the past history. If this is intrinsically finite, then a finite-state model exists for the system. If it is intrinsically infinite, then no finite-state model exists. We can often determine which of these two situations applies using simple intuition. We can also show that a system has a finite-state model by finding one. Showing that a system does not have a finite-state model is a bit more challenging.

Example 3.5: Consider the example of a system called *CodeRecognizer* whose input and output signals are binary sequences (with arbitrarily inserted stuttering elements, which have no effect). The system outputs *recognize* at the end of every subsequence 1100 in the input, and otherwise it outputs *absent*. If the input x is given by a sequence

$$(x(0), x(1), \cdots),$$

and the output y is given by the sequence

$$(y(0), y(1), \cdots),$$

then, if none of the inputs is *absent*,

$$y(n) = \begin{cases} \textit{recognize} & \text{if } (x(n-3), x(n-2), x(n-1), x(n)) = (1,1,0,0) \\ \textit{absent} & \text{otherwise} \end{cases} \tag{3.7}$$

Intuitively, in order to determine $y(n)$, it is enough to know whether the previous pattern of (non-*absent*) inputs is 0, 1, 11, 110, 1100. If this intuition is correct, we can implement *CodeRecognizer* by a state machine with five states that remember the patterns 0, 1, 11, 110, 1100. The machine of figure 3.4 does the job. The fact that we have a finite-state machine model of this system shows that this is a finite-state system.

The relationship in this example between the number of states and the number of input patterns that need to be stored suggests how to construct functions mapping input sequences to output sequences that cannot be realized by finite state machines. Here is a particularly simple example of such a function called *Equal*.

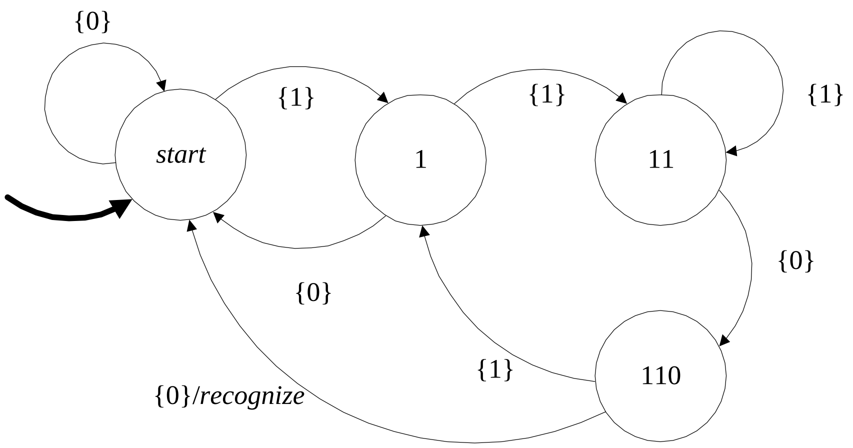


Figure 3.4: A machine that implements *CodeRecognizer*. It outputs *recognize* at the end of every input subsequence 1100, otherwise it outputs *absent*.

Example 3.6: The input and output signals of *Equal* are binary sequences (again with stuttering elements arbitrarily inserted). At each step, *Equal* outputs 1 if the previous inputs contain an equal number of 0's and 1's; otherwise *Equal* outputs 0. If the input sequence x is the sequence $(x(0), x(1), \cdots)$, then the output $y = F(x)$ is given by

$$\forall\, n \in \mathit{Nats}_0, \quad y(n) = \begin{cases} \mathit{equal} & \text{if number of 1's same as 0's in } x(0), \cdots, x(n) \\ \mathit{notEqual} & \text{otherwise} \end{cases} \tag{3.8}$$

Intuitively, in order to realize *Equal*, the machine must remember the difference between the number of 1's and 0's that have occurred in the past. Since these numbers can be arbitrarily large, the machine must have infinite memory, and so *Equal* cannot be realized by a finite-state machine.

We give a mathematical argument to show that *Equal* cannot be realized by any finite-state machine. The argument uses contradiction.

Suppose that a machine with N states realizes *Equal*. Consider an input sequence that begins with N 1's, $(1, \cdots, 1, x(N), \cdots)$. Let the state response be

$$(s(0), s(1), \cdots, s(N), \cdots).$$

Since there are only N distinct states, and the state response is of length at least $N+1$, the state response must visit at least one state twice. Call that state α. Suppose $s(m) = s(n) = \alpha$, with $m < n$. Then the two sequences $1^m 0^m$ and $1^n 0^m$ must lead to the same state, hence yield the same last output on entering state α.[4] But the last output for $1^m 0^m$ should be *equal*, and the last output for $1^n 0^m$ should be *notEqual*, which is a contradiction. So our hypothesis that a finite-state machine realizes *Equal* must be wrong! Exercise 6 asks you to construct an infinite state machine that realizes *Equal*.

[4]Recall that 1^m means a sequence of m consecutive 1's, similarly for 0^m.

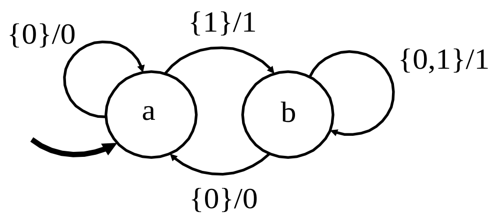


Figure 3.5: A simple nondeterministic state machine.

3.3 Nondeterministic state machines

There are situations in which it is sufficient to give an incomplete model of a system. Such models are more compact than complete models because they hide inessential details. This compactness will often make them easier to understand.

A useful form of incomplete model is a **nondeterministic state machine**. A nondeterministic state machine often has fewer states and transitions than would be required by a complete model. The state machines we have studied so far are **deterministic**.

3.3.1 State transition diagram

The state transition diagram for a state machine has one bubble for each state and one arc for each state transition. Nondeterministic machines are no different. Each arc is labeled with by "*guard*/*output*," where

$$\textit{guard} \subset \textit{Inputs}.$$

In a deterministic machine, the guards on arcs emerging from any given state are mutually exclusive. That is, they have no common elements. This is precisely what makes the machine deterministic. For nondeterministic machines, we relax this constraint. Guards can overlap. Thus, a given input value may appear in the guard of more than one transition, which means that more than one transition can be taken when that input arrives. This is precisely what makes the machine nondeterministic.

Example 3.7: Consider the state machine shown in figure 3.5. It begins in state a and transitions to state b the first time it encounters a 1 on the input. It then stays in state b arbitrarily long. If it receives a 1 at the input, it must stay in state b. If it receives a 0, then it can either stay in b or transition to a. Given the input sequence

$$0, 1, 0, 1, 0, 1, \cdots$$

then the following are all possible state responses and output sequences:

$$a, a, b, a, b, a, b, \cdots$$
$$0, 1, 0, 1, 0, 1, \cdots$$

$$a, a, b, b, b, a, b, \cdots$$
$$0, 1, 1, 1, 0, 1, \cdots$$

$$a, a, b, b, b, b, b, \cdots$$
$$0, 1, 1, 1, 1, 1, \cdots$$

$$a, a, b, a, b, b, b, \cdots$$
$$0, 1, 0, 1, 1, 1, \cdots$$

Nondeterminism can be used to construct an **abstraction** of a complicated machine, which is a simpler machine that has all the behaviors of the more complicated machine.

Example 3.8: Consider again the 60-minute parking meter. Its input alphabet is

$$\mathit{Inputs} = \{\mathit{coin5}, \mathit{coin25}, \mathit{tick}, \mathit{absent}\}.$$

Upon arrival of *coin5*, the parking meter increments its count by five, up to a maximum of 60 minutes. Upon arrival of *coin25*, it increments its count by 25, again up to a maximum of 60. Upon arrival of *tick*, it decrements its count by one, down to a minimum of zero.

A deterministic state machine model is illustrated schematically in figure 3.6(a). The state space is

$$\mathit{States} = \{0, 1, \cdots, 60\},$$

which is too many states to draw conveniently. Thus, patterns in the state space are suggested with ellipsis "$\cdots$".

Suppose that we are interested in modeling the interaction between this parking meter and a police officer. The police officer does not care what state the parking meter is in, except to determine whether the meter has expired or not. Thus, we need only two nonstuttering outputs, so

$$\mathit{Outputs} = \{\mathit{safe}, \mathit{expired}, \mathit{absent}\}.$$

The value *expired* is produced whenever the machine enters state 0.

The model has enough states that a full state transition diagram is tedious and complex enough that it might not be useful for generating insight about the design. Moreover, the detail that is modeled may not add insight about the interaction with a police officer.

Figure 3.6(b) is a nondeterministic model of the same parking meter. It has three states,

$$\mathit{States} = \{0, 1, \mathit{more}\}.$$

The inputs *coin5* and *coin25* in state 0 or 1 cause a transition to state *more*. The input *tick* in state *more* nondeterministically moves the state to 1 or leaves it in *more*.

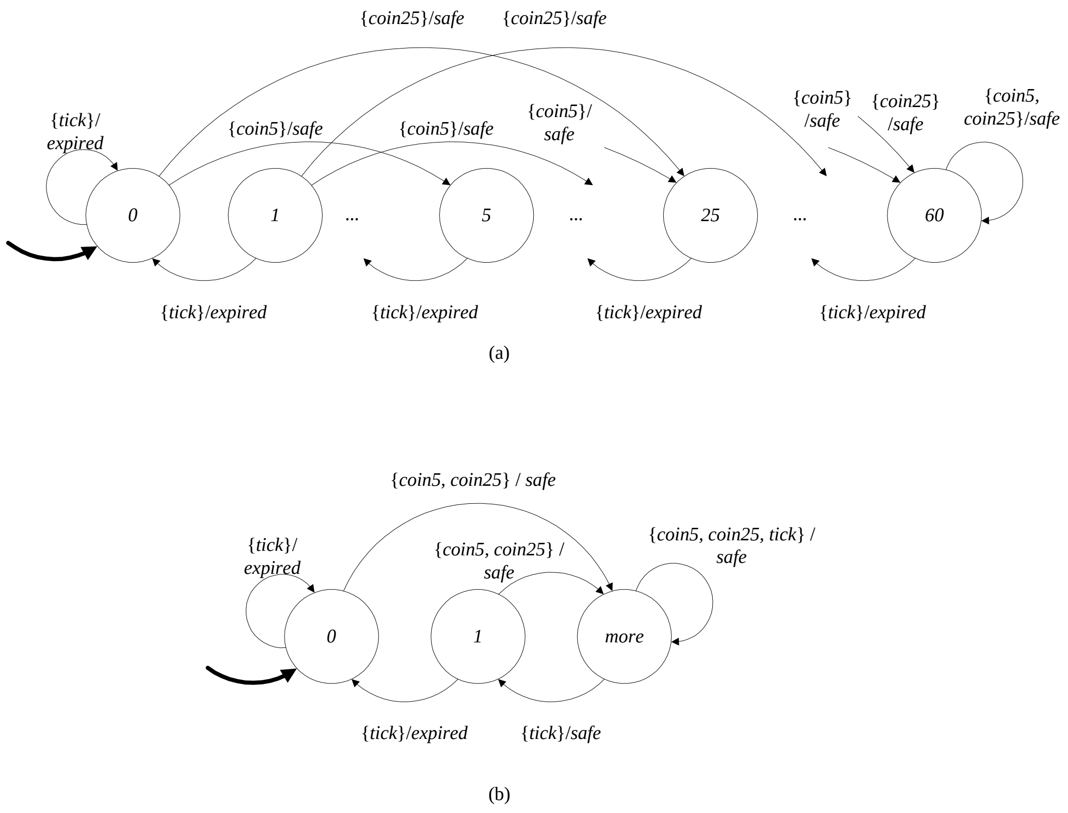


Figure 3.6: Deterministic and nondeterministic models for a 60 minute parking meter.

The top state machine has more detail than the bottom machine. Shortly, we will give a precise meaning to the phrase 'has more detail' using the concept of *simulation*. For the moment, note that the bottom machine can generate any output sequence that the top machine generates, for the same input sequence. But the bottom machine can also generate output sequences that the top machine cannot. For example, the sequence

$$(\mathit{expired}, \mathit{safe}, \mathit{safe}, \mathit{expired}, \cdots,$$

in which there are two *safe* outputs between two *expired* outputs is not a possible output sequence of the top machine, but it is a possible output sequence of the bottom machine. In the top machine, successive *expired* outputs must be separated by 0 or at least five *safe* outputs. This detail is not captured by the bottom machine. Indeed, in modeling the interaction with a police officer, this detail may not be important, so omitting it may be entirely appropriate.

The machines that we design and build, including parking meters, are usually deterministic. However, the state space of these machines is often very large, much larger than in this example, and it can be difficult to understand their behavior. We use simpler nondeterministic machine models that hide inessential details of the deterministic machine. The analysis of the simpler model reveals some properties, but not all properties, of the more complex machine. The art, of course, is in choosing the model that reveals the properties of interest.

3.3.2 Sets and functions model

The state machines we have been studying, with definitions of the form (3.1), are deterministic. If we know the initial state and the input sequence, then the entire state trajectory and output sequence can be determined. This is because any current state s and current input x uniquely determine the next state and output $(s', y) = \mathit{update}(s, x)$.

In a nondeterministic state machine, the next state is not completely determined by the current state and input. For a given current state s and input x, there may be more than one next state. So it no longer makes sense to characterize the machine by the function $\mathit{update}(s, x)$ because there is no single next state. Instead, we define a function *possibleUpdates* so that $\mathit{possibleUpdates}(s, x)$ is the set of possible next states and outputs. Whereas a deterministic machine has update function

$$\mathit{update}\colon \mathit{States} \times \mathit{Inputs} \to \mathit{States} \times \mathit{Outputs},$$

a nondeterministic machine has a (nondeterministic) state transition function

$$\mathit{possibleUpdates}\colon \mathit{States} \times \mathit{Inputs} \to P(\mathit{States} \times \mathit{Outputs}), \tag{3.9}$$

where $P(\mathit{State} \times \mathit{Outputs})$ is the **power set** of $\mathit{States} \times \mathit{Outputs}$. Recall that the power set is the set of all subsets. That is, any subset of $\mathit{States} \times \mathit{Outputs}$ is an element of $P(\mathit{States} \times \mathit{Outputs})$.

In order for a nondeterministic machine to be **receptive**, it is necessary that

$$\forall\, s \in \mathit{States}, x \in \mathit{Inputs} \quad \mathit{possibleUpdates}(s, x) \neq \emptyset.$$

Recall that a receptive machine accepts any input value in any state, makes a state transition (possibly back to the same state), and produces an output. That is, there is no situation where the reaction to an input is not defined.

Operationally, a nondeterministic machine involves arbitrarily selecting from among the possible next states given a current state and an input. The model says nothing about how the selection is made.

Similar to deterministic machines, we can collect the specification of a nondeterministic state machine into a 5-tuple

$$\mathit{StateMachine} = (\mathit{States}, \mathit{Inputs}, \mathit{Outputs}, \mathit{possibleUpdates}, \mathit{initialState}). \tag{3.10}$$

The *possibleUpdates* function is different from the *update* function of a deterministic machine.

Deterministic state machines of the form (3.1) are a special case of nondeterministic machines in which $\mathit{possibleUpdates}(s, x)$ consists of a single element, namely $\mathit{update}(s, x)$. In other words,

$$\mathit{possibleUpdates}(s, x) = \{\mathit{update}(s, x)\}.$$

Thus, any deterministic machine, as well as any nondeterministic machine, can be given by (3.10).

In the nondeterministic machine of (3.10), a single input sequence may give rise to many state responses and output sequences. If $x(0), x(1), x(2), \cdots$ is an input sequence, then $s(0), s(1), s(2), \cdots$ is a (possible) state trajectory and $y(0), y(1), y(2), \cdots$ is a (possible) output sequence provided that

$$\begin{aligned} s(0) &= \mathit{initialState} \\ \forall n \geq 0, \quad (s(n+1), y(n)) &\in \mathit{possibleUpdates}(s(n), x(n)). \end{aligned}$$

A deterministic machine defines a function from an input sequence to an output sequence,

$$F \colon \mathit{InputSignals} \to \mathit{OutputSignals},$$

where

$$\mathit{InputSignals} = [\mathit{Nats}_0 \to \mathit{Inputs}],$$

and

$$\mathit{OutputSignals} = [\mathit{Nats}_0 \to \mathit{Outputs}].$$

We define a **behavior** of the machine to be a pair (x, y) such that $y = F(x)$, i.e., a behavior is possible input, output pair. A deterministic machine is such that for each $x \in \mathit{InputSignals}$, there is exactly one $y \in \mathit{OutputSignals}$ such that (x, y) is a behavior.

We define the set

$$\mathit{Behaviors} \subset \mathit{InputSignals} \times \mathit{OutputSignals}, \tag{3.11}$$

where

$$\mathit{Behaviors} = \\ \{(x, y) \in \mathit{InputSignals} \times \mathit{OutputSignals} \mid y \text{ is a possible output sequence for input } x\}.$$

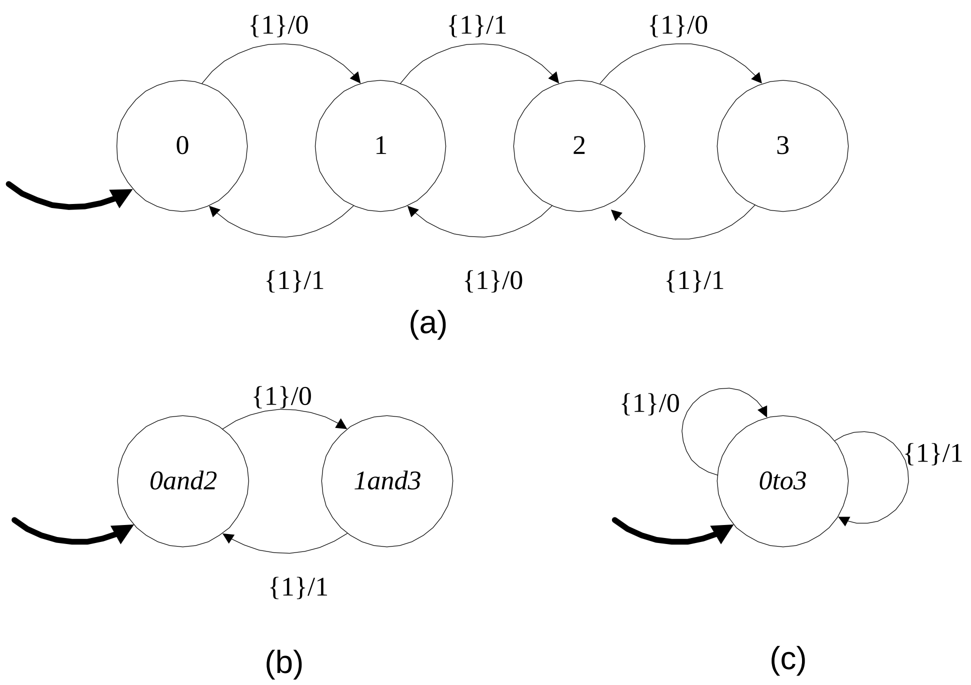


Figure 3.7: Three state machines where (a) and (b) simulate one another and (c) simulates (a) and (b).

The set *Behaviors* is the graph of the function F.

For a nondeterministic machine, for each $x \in$ *InputSignals*, there may be more than one $y \in$ *OutputSignals* such that (x, y) is a behavior. The set *Behaviors*, therefore, is no longer the graph of a function. Instead, it defines a **relation**, which is a generalization of a function where there can be two or more distinct elements that share the same element in the domain. Its interpretation is still simple, however. If $(x, y) \in$ *Behaviors*, then input x may produce output y.

3.4 Simulation and bisimulation

Two state machines that have the same input and output alphabets may have related behaviors. Suppose for example that given the same input sequence, they produce the same output sequence. They may be very different state machines and still be equivalent in this sense.

Example 3.9: Consider the three state machines in figure 3.7, where the input and output alphabets are

$$\textit{Inputs} = \{1, \textit{absent}\} \text{ and } \textit{Outputs} = \{0, 1, \textit{absent}\}.$$

Machine (a) is the most complicated, in that it has the most states. However, its behavior is identical to that of the state machine in (b). Both machines produce an alternating sequence of 0 and 1 each time they receive a 1 at the input. The machine in (a) is nondeterministic: for example, from state 1 it can move under input 1 to state 0 or state 2). Yet its input/output behavior is identical to that of the deterministic machine in (b).

The nondeterministic machine in (c) is capable of behaving like those in (a) and (b). That is, it can produce an alternating sequence of 0 and 1. However, it can also produce other output sequences that neither (a) nor (b) can produce. It is, in this sense, more general, or more abstract, than the machines in (a) and (b).

To study the relationships between the machines in figure 3.7 we introduce the concepts of **simulation** and **bisimulation**. The machine in (c) is said to **simulate** (b) and (a). The machine in (b) is said to **bisimulate** (a), or to **be bisimilar to** (a). Bisimulation can be viewed as a form of equivalence between state machines. Simulation can be viewed as a form of abstraction of state machines.

Example 3.10: In figure 3.6, the bottom machine can generate any output sequence that the top machine generates, for the same input sequence. The reverse is not true; there are output sequences that the bottom machine can generate that the top machine cannot. The bottom machine is an abstraction of the top one.

We will see that the bottom machine simulates the top machine (but not vice versa).

To understand simulation, it is easiest to consider a "matching" game between one machine and the other, more abstract, machine that simulates the first. The game starts with both machines in their initial states. The first machine is allowed to react to an input. If this machine is nondeterministic, it may have more than one possible reaction; it is permitted to choose any one of these reactions. The second, more abstract, machine must react to the same input such that it produces the same output. If it is non-deterministic, it is free to pick from among the possible reactions any one that matches the output of the first machine. The second machine "wins" this matching game if it can always match the output of the first machine. We then say that the second machine simulates the first one. If the first machine can produce an output that the second one cannot match, then the second machine does not simulate the first machine.

Example 3.11: Suppose we wish to determine whether (c) simulates (b) in figure 3.7. The game starts with the two machines in their initial states, which we jointly denote by the pair

$$s_0 = (\mathit{0and2}, \mathit{0to3}) \in \mathit{States}_b \times \mathit{States}_c.$$

Machine (b) (the one being simulated) moves first. Given an input, it reacts. If it is nondeterministic, then it is free to react in any way possible, although in this case, (b) is deterministic, so it will have only one possible reaction. Machine (c) then has to **match** the move taken by (b); given the same input, it must react such that it produces the same output.

There are two possible inputs to machine (b), 1 and *absent*. If the input is *absent*, the machine reacts by stuttering. Machine (c) can match this by stuttering as well. For this example, it will always to match stuttering moves, so we will not consider them further.

Excluding the stuttering input, there is only one possible input to machine (b), 1. The machine reacts by producing the output 0 and changing to state *1and3*. Machine (c) can match this by taking one of the two possible transitions out of its current state, the one that produces output 0. The resulting states of the two machines are denoted

$$s_1 = (\mathit{1and3}, \mathit{0to3}) \in \mathit{States}_b \times \mathit{States}_c.$$

From here, again there is only one non-stuttering input possible, so (b) reacts by moving back to *0and3* and producing the output 1. Again, (c) can match this by choosing the transition that produces 1. The resulting states are s_0, back where we started.

The "winning" strategy of the second machine can be summarized by the set

$$S_{b,c} = \{s_0, s_1\} \subset \mathit{States}_b \times \mathit{States}_c.$$

The set $S_{b,c}$ in the previous example is called a **simulation relation**; it shows how (c) simulates (b). A simulation relation associates states of the two machines. Suppose we have two state machines, X and Y, which may be deterministic or nondeterministic. Let

$$X = (\mathit{States}_X, \mathit{Inputs}, \mathit{Outputs}, \mathit{possibleUpdates}_X, \mathit{initialState}_X),$$

and

$$Y = (\mathit{States}_Y, \mathit{Inputs}, \mathit{Outputs}, \mathit{possibleUpdates}_Y, \mathit{initialState}_Y).$$

Notice that the two machines have the same input and output alphabets. If either machine is deterministic, then its *possibleUpdates* function always returns a set with only one element in it.

If Y simulates X, the simulation relation is given as a subset of $\mathit{States}_X \times \mathit{States}_Y$. Note the ordering here; the machine that moves first in the game, X, the one being simulated, is first in $\mathit{States}_X \times \mathit{States}_Y$.

To consider the reverse scenario, if X simulates Y, then the relation is given as a subset of $\mathit{States}_Y \times \mathit{States}_X$. In this version of the game Y must move first.

If we can find simulation relations in both directions, then the machines are bisimilar.

We can state the "winning" strategy mathematically. We say that Y **simulates** X if there is a subset $S \subset \mathit{States}_X \times \mathit{States}_Y$ such that

1. $(\mathit{initialState}_X, \mathit{initialState}_Y) \in S$, and
2. If $(s_X, s_Y) \in S$, then $\forall\ x \in \mathit{Inputs}$, and $\forall\ (s'_X, y_X) \in \mathit{possibleUpdates}_X(s_X, x)$, there is a $(s'_Y, y_Y) \in \mathit{possibleUpdates}_Y(s_Y, x)$ such that:
   (a) $(s'_X, s'_Y) \in S$, and
   (b) $y_X = y_Y$.

This set S, if it exists, is called the simulation relation. It establishes a correspondence between states in the two machines.

Example 3.12: Consider again the state machines in figure 3.7. The machine in (b) simulates the one in (a). The simulation relation is a subset

$$S_{a,b} \subset \{0, 1, 2, 3\} \times \{\mathit{0and2}, \mathit{1and3}\}.$$

The names of the states in (b) (which are arbitrary) are suggestive of the appropriate simulation relation. Specifically,

$$S_{a,b} = \{(0, \mathit{0and2}), (1, \mathit{1and3}), (2, \mathit{0and2}), (3, \mathit{1and3})\}.$$

The first condition of a simulation relation, that the initial states match, is satisfied because $(0, \mathit{0and2}) \in S_{a,b}$. The second condition can be tested by playing the game, but starting in each pair of states in $S_{a,b}$.

Start with the two machines in one pair of states in $S_{a,b}$, such as the initial states $(0, \mathit{0and2})$. Then consider the moves that machine (a) can make in a reaction. Ignoring stuttering, if we start with $(0, \mathit{0and2})$, (a) must move to state 1 (given input 1). Given the same input, can (b) match the move? To match the move, it must react to the same input, produce the same output, and move to a state so that the new state of (a) paired with the new state of (b) is in $S_{a,b}$.

Indeed, (b) can match either of the moves (a) can take from state 0. It is easy (albeit somewhat tedious) to check that this matching can be done from any starting point in $S_{a,b}$.

The above example shows how to use the game to check that a particular subset of $\mathit{States}_X \times \mathit{States}_Y$ is a simulation relation. Thus, the game can be used either to construct a simulation relation or to check whether a particular set is a simulation relation.

For the examples in figure 3.7, we have shown that (c) simulates (b) and that (b) simulates (a). Simulation is **transitive**, meaning that we can immediately conclude that (c) simulates (a). In particular, if we are given simulation relations $S_{a,b} \subset \mathit{States}_a \times \mathit{States}_b$ ((b) simulates (a)) and $S_{b,c} \subset \mathit{States}_b \times \mathit{States}_c$ ((c) simulates (b)), then

$$S_{a,c} = \{(s_a, s_c) \in \mathit{States}_a \times \mathit{States}_c \mid \text{ there exists } s_b \in S_b \text{ where } (s_a, s_b) \in S_{a,b} \text{ and } (s_b, s_c) \in S_{b,c}\} \tag{3.12}$$

is the simulation relation showing that (c) simulates (a).

Example 3.13: For the examples in figure 3.7, we have already determined

$$S_{a,b} = \{(0, \mathit{0and2}), (1, \mathit{1and3}), (2, \mathit{0and2}), (3, \mathit{1and3})\}.$$

and

$$S_{b,c} = \{(\mathit{0and2}, \mathit{0to3}), (\mathit{1and3}, \mathit{0to3})\}.$$

From (3.12) we can conclude that

$$S_{a,c} = \{(0, \mathit{0to3}), (1, \mathit{0to3}), (2, \mathit{0to3}), (3, \mathit{0to3})\},$$

which further supports the suggestive choices of state names.

Simulation relations are not (necessarily) symmetric.

Example 3.14: For the examples in figure 3.7, (b) does not simulate (c). To see this, we can attempt to construct a simulation relation by playing the game. Starting in the initial states,

$$s_0 = (\mathit{0to3}, \mathit{0and2}),$$

we allow (c) to move first. Presented with a nonstuttering input, 1, it has two choices of moves. One of these, which produces an output 0, (b) can match. However, the other move, which produces the output 1, (b) cannot match. In this state, (b) has no way to produce the output 1. Thus, the game gets stuck, and we fail to construct a simulation relation.

Although simulation relations are not necessarily symmetric, it is possible to have two state machines that simulate each other. Obviously, if the two machines are identical, they simulate each other. But they may simulate each other even if they are not identical.

Example 3.15: In figure 3.7, not only does state machine (b) simulate (a), but also (a) simulates (b). We can easily verify this by determining that not only can (b) match any move (a) makes, but (a) can match any move (b) makes. In fact, since (a) is nondeterministic, in two of its states it has two distinct ways of matching the moves of (b). Since it moves second, it can arbitrarily choose from among these possibilities.

If from state 1 it always chooses to return to state 0, then the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1)\}.$$

Otherwise, if from state 2 it always chooses to return to state 1, then the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1), (\mathit{0and2}, 2)\}.$$

Otherwise, the simulation relation is

$$S_{b,a} = \{(\mathit{0and2}, 0), (\mathit{1and3}, 1), (\mathit{0and2}, 2), (\mathit{1and3}, 3)\}.$$

Thus, the simulation relation is not unique.

When simulation is bidirectional, the relationship is called **bisimulation**. The machines are said to be **bisimilar**. This is a rather strong form of equivalence. Usually, any state machine may be replaced by a bisimilar state machine. Thus, the machine in (a) may be replaced by the simpler one in (b).

A common use of simulation is to establish a relationship between a more abstract model and a more detailed model. In the example above, (c) is a more abstract model of either (b) or (a). It is more abstract in the sense that it loses detail. For example, it has lost the property that 0's and 1's alternate in the output sequence. We now give a more compelling example of such abstraction, where the abstraction dramatically reduces the number of states while still preserving some properties of interest.

Example 3.16: In the case of the parking meter, the bottom machine in figure 3.6 simulates the top machine. Let A denote the top machine, and let B denote the bottom machine. We will now identify the simulation relation.

The simulation relation is a subset $S \subset \{0, 1, \cdots, 60\} \times \{0, 1, \mathit{more}\}$. It is intuitively clear that 0 and 1 of the bottom machine correspond to 0 and 1, respectively, of the top machine. Thus, $(0,0) \in S$ and $(1,1) \in S$. It is also intuitive that *more* corresponds to all of the remaining states $2, \cdots 60$ of the top machine. So we propose to define the simulation relation as

$$S = \{(0,0), (1,1)\} \cup \{(s_A, \mathit{more}) \mid 2 \le s_A \le 60\} \tag{3.13}$$

We now check that S is indeed a simulation relation, as defined above.

The first condition of a simulation relation, that the initial states match, is satisfied because $(0,0) \in S$. The second condition is more tedious to verify. It says that for each pair of states in S, and for each input, the two machines can transition to a pair of new states that is also in S, and that these two transitions produce the same output. Since machine A is deterministic, there is no choice about which transition it takes and which output it produces. In machine B, there are choices, but all we require is that one of the choices match.

The only state of machine B that actually offers choices is *more*. Upon receiving *tick*, the machine can transition back to *more* or down to 1. In either case, the output is *safe*. It is easy to see that these two choices are sufficient for state *more* to match states $2, 3, ...60$ of machine A.

Thus the bottom machine indeed simulates the top machine with the simulation relation (3.13).

3.4.1 Relating behaviors

A simulation relation establishes a correspondence between two state machines, one of which is typically much simpler than the other. The relation lends confidence that analyzing the simpler machine indeed reveals properties of the more complicated machine.

This confidence rests on a theorem and corollary that we will develop in this section. These results relate the input/output behaviors of state machines that are related by simulation.

Given an input sequence $x = (x(0), x(1), x(2), \cdots)$, if a state machine can produce the output sequence $y = (y(0), y(1), y(2), \cdots)$, then (x, y) is said to be a **behavior** of the state machine. The set of all behaviors of a state machine obviously satisfies

$$\mathit{Behaviors} \subset \mathit{InputSignals} \times \mathit{OutputSignals}.$$

Theorem Let B simulate A. Then

$$\mathit{Behaviors}_A \subset \mathit{Behaviors}_B.$$

This theorem is easy to prove. Consider a behavior $(x, y) \in \textit{Behaviors}_A$. We need to show that $(x, y) \in \textit{Behaviors}_B$.

Let the simulation relation be S. Find all possible state responses for A

$$s_A = (s_A(0), s_A(1), \cdots)$$

that result in behavior (x, y). (If A is deterministic, then there will be only one.) The simulation relation assures us that we can find a state response for B

$$s_B = (s_B(0), s_B(1), \cdots)$$

where $(s_A(i), s_B(i)) \in S$, such that given input x, B produces y. Thus, $(x, y) \in \textit{Behaviors}_B$.

Intuitively, the theorem simply states that B can match every move of A and produce the same output sequence. It also implies that if B cannot produce a particular output sequence, then neither can A. This is stated formally in the following corollary.

Corollary Let B simulate A. Then if

$$(x, y) \notin \textit{Behaviors}_B$$

then

$$(x, y) \notin \textit{Behaviors}_A.$$

The theorem and corollary are useful for analysis. The general approach is as follows. We have a deterministic state machine A. We wish to show that its input-output function satisfies some property. That is, every behavior satisfies some condition. We construct a simpler nondeterministic machine B whose input-output relation satisfies the same property, where B simulates A. The theorem guarantees that A will satisfy this property, too. That is, since all behaviors of B satisfy the property, all behaviors of A must also. This technique is useful since it is often easier to understand a simple FSM than a complex FSM with many states.

Conversely, if there is some property that we must assure that no behavior of A has, it is sufficient to show that no behavior of the simpler machine B has it. This scenario is typical of a **safety** problem, where we must show that dangerous outputs from our system are not possible.

Example 3.17: For the parking meter of figure 3.6, for example, we can use the nondeterministic machine to show that if a coin is inserted at step n, the output at steps n and $n + 1$ is *safe*. By the corollary, it is sufficient to show that the nondeterministic machine cannot do any differently.

It is important to understand what the theorem says, and what it does not say. It does not say, for example, that if $\textit{Behaviors}_A \subset \textit{Behaviors}_B$ then B simulates A. In fact, this statement is not true. Consider the two machines in figure 3.8, where

$$\textit{Inputs} = \{1, \textit{absent}\},$$

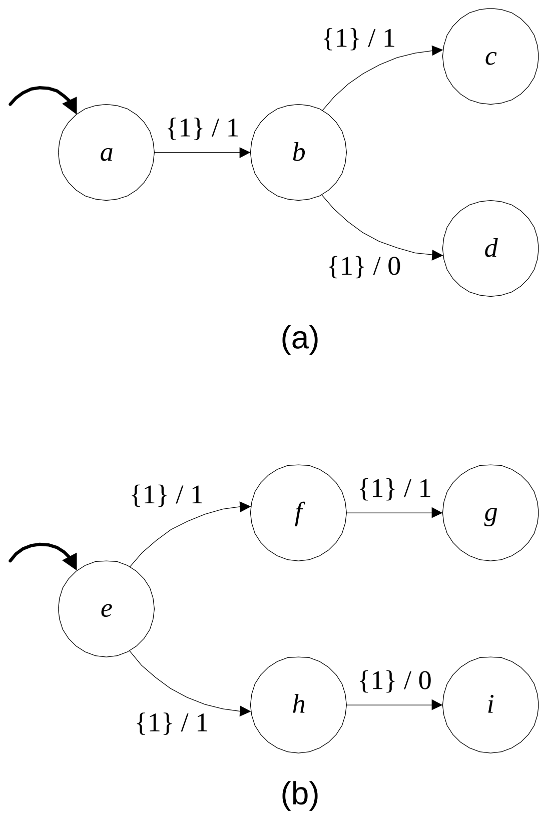


Figure 3.8: Two state machines with the same behaviors that are not bisimilar.

$$\textit{Outputs} = \{0, 1, \textit{absent}\}.$$

These two machines have the same behaviors. The non-stuttering outputs are $(1, 0)$ or $(1, 1)$, selected nondeterministically, assuming the input sequence has at least two non-stuttering elements. However, they are not bisimilar. In particular, (b) does not simulate (a). To see this, we play the matching game. Machine (a) is allowed to move first. Ignoring stuttering, it has no choice but to move from a to b and produce output 1. Machine (b) can match this two ways; it has no basis upon which to prefer one way to match it over another, so it picks one, say moving to state f. Now it is the turn of machine (a), which has two choices. If it choses to move to d, then machine (b) cannot match its move. (A similar argument works if (b) picks state h.) Thus, machine (b) does not simulate machine (a), despite the fact that $\textit{Behaviors}_A \subset \textit{Behaviors}_B$.[5]

Exercises

In some of the following exercises you are asked to design state machines that carry out a given task. The design is simple and elegant if the state space is properly chosen. Although the state space is not unique, there often is a natural choice. As usual, each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** A state machine with
$$\textit{Inputs} = \{a, b, c, d, \textit{absent}\},$$
has a state s with two emerging arcs with guards
$$\textit{guard1} = \{a\}$$
and
$$\textit{guard2} = \{a, b, d\}.$$
   (a) Is this state machine deterministic?
   (b) Define the set *else* for state s and specify the source and destination state for the *else* arc.

2. **E** For the answering machine example of figure 3.1, assume the input is
$$(\textit{offhook}, \textit{offhook}, \textit{ring}, \textit{offhook}, \textit{ring}, \textit{ring}, \textit{ring}, \textit{offhook}, \cdots).$$
This corresponds to a user of the answering machine making two phone calls, answering a third after the first ring, and answering a second after the third ring.
   (a) Give the state response of the answering machine.
   (b) Give the trace of the answering machine.

[5]Recall that in our notation $\subset$ allows the two sets to be equal.

(c) Give the output.

3. **E** Consider the alphabets

$$\mathit{Inputs} = \mathit{Outputs} = \mathit{Bin} = \{0, 1\}.$$

(a) Construct a state machine that uses these alphabets such that if $x(0), x(1), \cdots$ is any input sequence, the output sequence is

$$\forall\, n \in \mathit{Nats}_0, \quad y(n) = \begin{cases} 1 & \text{if } n \geq 2 \wedge (x(n-2), x(n-1), x(n)) = (1,1,1) \\ 0 & \text{otherwise} \end{cases}$$

In words, the machine outputs 1 if the three previous inputs are all 1's, otherwise it outputs 0.

(b) For the same input and output alphabet, construct a state machine that outputs 1 if the three previous inputs are either $(1, 1, 1)$ or $(1, 0, 1)$, and otherwise it outputs 0.

4. **E** A modulo N counter is a device that can output any integer between 0 and $N - 1$. The device has three inputs, *increment*, *decrement*, and *reset*, plus, as always, a stuttering element *absent*; *increment* increases the output integer by 1; *decrement* decreases this integer by 1; and *reset* sets the output to 0. Here *increment* and *decrement* are modulo N operations.

Note: Modulo N numbers work as follows. For any integer m, $m \; mod \; N = k$ where $0 \leq k \leq N - 1$ is the unique integer such that N divides $(m - k)$. Thus there are only N distinct modulo-N numbers, namely, $0, \cdots, N - 1$.

(a) Give the state transition diagram of this counter for $N = 4$.

(b) Give the update table of this counter for $N = 4$.

(c) Give a description of the state machine by specifying the five entities that appear in (3.1); again assume $N = 4$.

(d) Take $N = 3$. Calculate the state response for the input sequence

$$\mathit{increment}^4, \mathit{decrement}^3, \cdots$$

starting with initial state 1, where s^n means s repeated n times.

5. **T** The state machine *UnitDelay* is defined to behave as follows. On the first non-stuttering reaction (when the first non-stuttering input arrives), the output a is produced. On subsequent reactions (when subsequent inputs arrive), the input that arrived at the previous non-stuttering reaction is produced as an output.

(a) Assume the input and output alphabets are

$$\mathit{Inputs} = \mathit{Outputs} = \{a, b, c, \mathit{absent}\}.$$

Give a finite state machine that implements *UnitDelay* for this input set. Give both a state transition diagram and a definition of each of the components in (3.1).

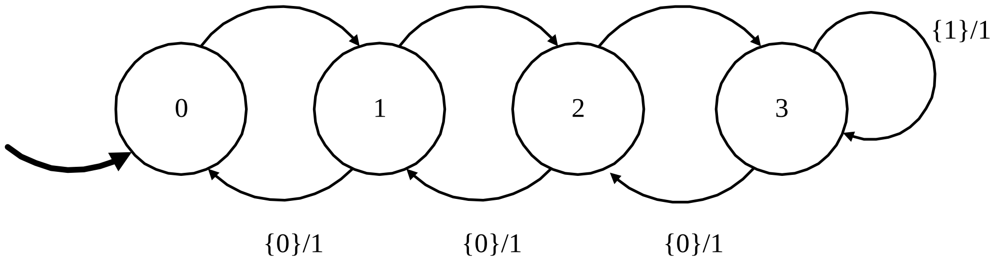


Figure 3.9: Machine that outputs at least one 1 between any two 0's.

(b) Assume the input and output set is

$$\textit{Inputs} = \textit{Nats} \cup \{\textit{absent}\},$$

and that on the first non-stuttering reaction, the machine produces 0 instead of a. Give an (informal) argument that no finite state machine can implement *UnitDelay* for this input set. Give an infinite state machine by defining each of the components in (3.1).

6. **T** Construct an infinite state machine that realizes *Equal*.

7. **C** An elevator connects two floors, 1 and 2. It can go up (if it is on floor 1), down (if it is on floor 2) and stop on either floor. Passengers at any floor may press a button requesting service. Design a controller for the elevator so that (1) every request is served, and (2) if there is no pending request, the elevator is stopped. For simplicity, do not be concerned about responding to requests from passengers inside the elevator.

8. **T** The state machine in figure 3.9 has the property that it outputs at least one 1 between any two 0's. Construct a two-state nondeterministic state machine that simulates this one and preserves that property.

9. **T** For the nondeterministic state machine in figure 3.10 the input and output alphabets are

$$\textit{Inputs} = \textit{Outputs} = \{0, 1, \textit{absent}\}.$$

(a) Define the *possibleUpdates* function (3.9) for this state machine.

(b) Define the relation *Behaviors* in (3.11) for this state machine. Part of the challenge here is to find a way to describe this relation compactly. For simplicity, ignore stuttering; i.e. assume the input is never *absent*.

10. **E** The state machine in figure 3.11 implements *CodeRecognizer*, but has more states than the one in figure 3.4. Show that it is equivalent by giving a bisimulation relation with the machine in figure 3.4.

11. **E** The state machine in figure 3.12 has input and output alphabets

$$\textit{Inputs} = \{1, a\},$$

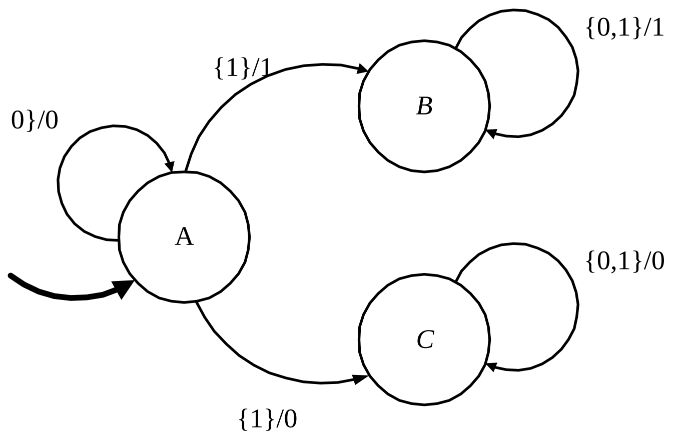


Figure 3.10: Nondeterministic state machine for exercise 9.

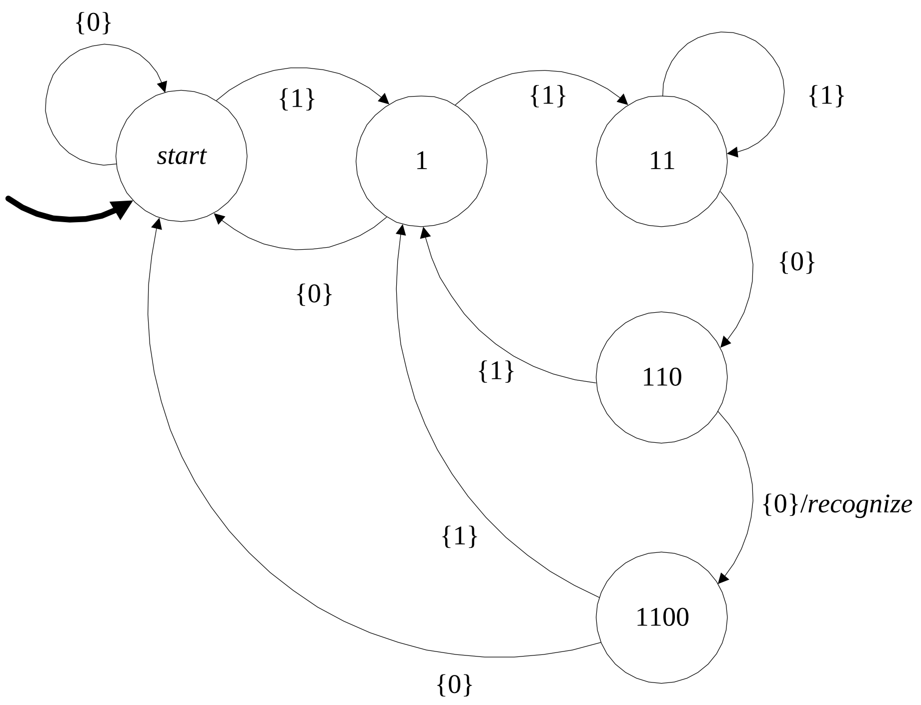


Figure 3.11: A machine that implements *CodeRecognizer*, but has more states than the one in figure 3.4.

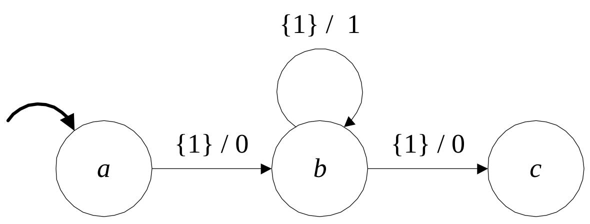


Figure 3.12: State machine for problem 11.

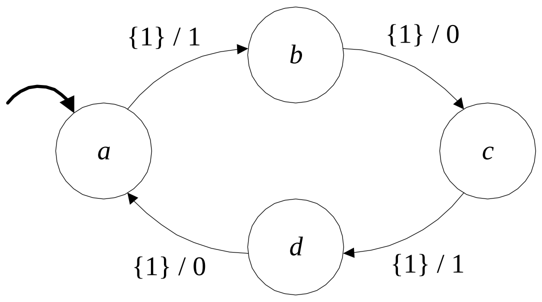


Figure 3.13: A machine that has more states than it needs.

$$\mathit{Outputs} = \{0, 1, a\},$$

where a (short for *absent*) is the stuttering element. State whether each of the following is in the set *Behaviors* for this machine. In each of the following, the ellipsis "$\cdots$" means that the last element is repeated forever. Also, in each case, the input and output signals are given as sequences.

(a) $((1, 1, 1, 1, 1, \cdots), (0, 1, 1, 0, 0, \cdots))$
(b) $((1, 1, 1, 1, 1, \cdots), (0, 1, 1, 0, a, \cdots))$
(c) $((a, 1, a, 1, a, \cdots), (a, 1, a, 0, a, \cdots))$
(d) $((1, 1, 1, 1, 1, \cdots), (0, 0, a, a, a, \cdots))$
(e) $((1, 1, 1, 1, 1, \cdots), (0, a, 0, a, a, \cdots))$

12. **E** The state machine in figure 3.13 has

$$\mathit{Inputs} = \{1, \mathit{absent}\},$$

$$\mathit{Outputs} = \{0, 1, \mathit{absent}\}.$$

Find a bisimilar state machine with only two states, and give the bisimulation relation.

13. **E** You are told that state machine A has

$$\mathit{Inputs} = \{1, 2, \mathit{absent}\},$$

$$\mathit{Outputs} = \{1, 2, \mathit{absent}\},$$

$$\textit{States} = \{a, b, c, d\}.$$

but you are told nothing further. Do you have enough information to construct a state machine B that simulates A? If so, give such a state machine, and the simulation relation.

Chapter 4

Composing State Machines

Design of interesting systems involves composing components. Their complexity is built up out of simpler parts. Since systems are functions, their composition is function composition, as discussed above in section 2.1.5. State machines, however, are not given directly as functions, so composing them is not totally trivial. The state machine model, however, is amenable to composition, in the sense that we can define a new state machine from a composition of simpler ones. That is the topic of this chapter.

In section 2.3.4 we used a block diagram syntax to define compositions of systems. We will use the same syntax here. In that section, we saw that feedback compositions have a more subtle meaning than those without feedback. We will encounter similar subtlety with feedback compositions of state machines.

4.1 Synchrony

Consider a set of interconnected components, where each component is a state machine. By "interconnected" we mean that the outputs of one component may be inputs of another. We wish to construct a state machine model for the composition of components. We choose a particular style of interconnection called **synchrony**. This style dictates that each state machine in the composition reacts *simultaneously* and *instantaneously*. Thus, a reaction of the composite machine consists of a set of simultaneous reactions of each of the component machines.

A reaction of the composite machine is triggered by inputs. Thus, when a reaction occurs is externally determined, not determined by the composite machine itself. This is the same as a single machine. Even the stuttering element of the input alphabet is provided externally. The environment provides *absent* at the input, for example, and demands a reaction when the state machine is composed with another machine, and a reaction is needed from the other machine.

Such systems are said to be **reactive**, which means that they react at a rate determined by the environment in which they operate. Because they are synchronous, they are often called **synchronous/reactive** systems.

The reactions of the component machines and of the composite are viewed as being instantaneous. That is, a reaction does not take time. This creates some interesting subtleties. In particular, the output from a state machine is viewed as being *simultaneous* with the input. We will discuss the ramifications of this interpretation for each specific composition below.

Synchrony is a very useful model of the behavior of physical systems. Digital circuits, for example, are almost always designed using the model. Circuit elements are viewed as taking no time to calculate their outputs given their inputs, and time overall is viewed as progressing in a sequence of discrete time steps according to ticks of a clock. Of course, the time that it takes for a circuit element to produce an output cannot ultimately be ignored, but the model is useful because for most circuit elements in a complex design, it can be ignored. Only the delay of those circuit elements along a **critical path** affects the overall performance of the circuit.

More recently than for circuits, synchrony has come to be used in software as well. Concurrent software modules interact according to the synchronous model. Languages built on this principle are called **synchronous languages**. They are used primarily in real-time embedded system[1] design.

4.2 Side-by-side composition

Consider the composition of two state machines shown in figure 4.1. This is called a **side-by-side composition**. The two state machines have no interaction with one another, but nonetheless we wish to define a single state machine representing the synchronous execution of the two component state machines.

The state space of the composite state machine is simply

$$\mathit{States} = \mathit{States}_A \times \mathit{States}_B.$$

(We could take the cross product in the other order, resulting in a different but bisimilar composite state machine.) The initial state is

$$\mathit{initialState} = (\mathit{initialState}_A, \mathit{initialState}_B).$$

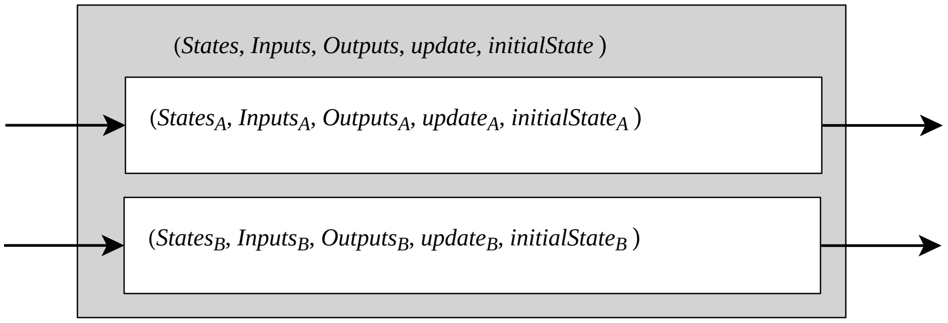
The input and output alphabets are

$$\mathit{Inputs} = \mathit{Inputs}_A \times \mathit{Inputs}_B, \tag{4.1}$$

$$\mathit{Outputs} = \mathit{Outputs}_A \times \mathit{Outputs}_B. \tag{4.2}$$

Recall that Inputs_A and Inputs_B include a stuttering element. This is convenient because it allows a reaction of the composite when we really want only one of the machines to react. The composite also has a stuttering element, which is the pair of stuttering elements.

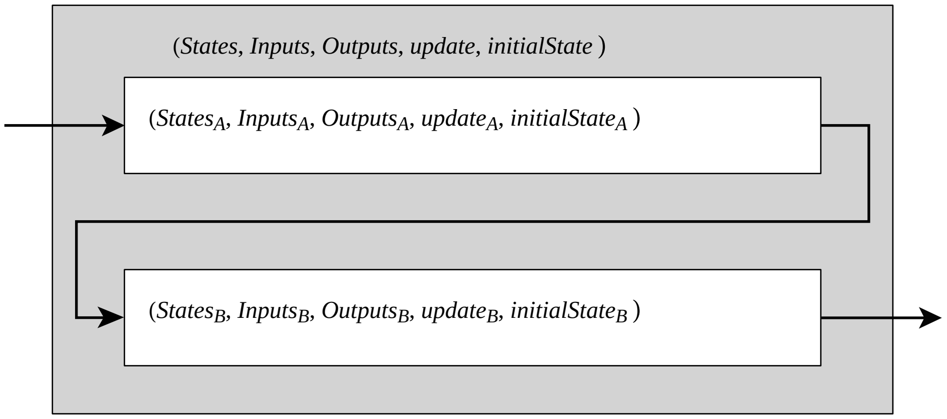
[1] An **embedded system** is a computing system that is not first-and-foremost a computer. For example, a cellular telephone, which contains several on-board computers, is an embedded system. The ignition controls of recent cars are embedded systems. In fact, most modern controllers of physical systems are embedded systems.


(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
(States_B, Inputs_B, Outputs_B, update_B, initialState_B)

Definition of the composite machine:
$\mathit{States} = \mathit{States}_A \times \mathit{States}_B$
$\mathit{Inputs} = \mathit{Inputs}_A \times \mathit{Inputs}_B$
$\mathit{Outputs} = \mathit{Outputs}_A \times \mathit{Outputs}_B$
$\mathit{initialState} = (\mathit{initialState}_A, \mathit{initialState}_B)$
$\mathit{update}((s_A, s_B), (x_A, x_B)) = ((s'_A, s'_B), (y_A, y_B))$
where

$$(s'_A, y_A) = \mathit{update}_A(s_A, x_A) \text{ and } (s'_B, y_B) = \mathit{update}_B(s_B, x_B)$$

Figure 4.1: Summary of side-by-side composition of state machines.


(*States*, *Inputs*, *Outputs*, *update*, *initialState*)
(*States*$_A$, *Inputs*$_A$, *Outputs*$_A$, *update*$_A$, *initialState*$_A$)
(*States*$_B$, *Inputs*$_B$, *Outputs*$_B$, *update*$_B$, *initialState*$_B$)

Assumptions about the component machines:
$\mathit{Outputs}_A \subset \mathit{Inputs}_B$

Definition of the composite machine:
$\mathit{States} = \mathit{States}_A \times \mathit{States}_B$
$\mathit{Inputs} = \mathit{Inputs}_A$
$\mathit{Outputs} = \mathit{Outputs}_B$
$\mathit{initialState} = (\mathit{initialState}_A, \mathit{initialState}_B)$
$\mathit{update}((s_A, s_B), x) = ((s'_A, s'_B), y_B)$
where

$$(s'_A, y_A) = \mathit{update}_A(s_A, x) \text{ and } (s'_B, y_B) = \mathit{update}_B(s_B, y_A).$$

Figure 4.2: Summary of cascade composition of state machines.

The update function of the composite, with this assumption, is quite simple:

$$\mathit{update}((s_A, s_B), (x_A, x_B)) = ((s'_A, s'_B), (y_A, y_B)),$$

where

$$(s'_A, y_A) = \mathit{update}_A(s_A, x_A),$$

and

$$(s'_B, y_B) = \mathit{update}_B(s_B, x_B).$$

4.3 Cascade composition

We consider two state machines shown in figure 4.2, where the output of one is the input of the other. This is called a **cascade composition**. We wish to compose these state machines so that they react together, synchronously, as one state machine. This is fairly easy to do, but there are some subtleties.

The two state machines are given by

$$\mathit{StateMachine}_A = (\mathit{States}_A, \mathit{Inputs}_A, \mathit{Outputs}_A, \mathit{update}_A, \mathit{initialState}_A)$$

and

$$\mathit{StateMachine}_B = (\mathit{States}_B, \mathit{Inputs}_B, \mathit{Outputs}_B, \mathit{update}_B, \mathit{initialState}_B).$$

Let the composition be given by

$$\mathit{StateMachine} = (\mathit{States}, \mathit{Inputs}, \mathit{Outputs}, \mathit{update}, \mathit{initialState}).$$

Clearly, for a composition like that in figure 4.2 to be possible we must have

$$\mathit{Outputs}_A \subset \mathit{Inputs}_B.$$

That is, any output produced by machine A needs to be in the input alphabet for machine B. As a result,

$$\mathit{OutputSignals}_A \subset \mathit{InputSignals}_B.$$

This is analogous to a **type constraint** in programming languages, where in order for two pieces of code to interact, they must use compatible data types.

We wish to construct a state machine model for this series connection. As noted in the figure, the input alphabet of the composite is given by

$$\mathit{Inputs} = \mathit{Inputs}_A.$$

The stuttering element of *Inputs*, of course, is just the stuttering element of Inputs_A. The output alphabet of the composite is

$$\mathit{Outputs} = \mathit{Outputs}_B.$$

The state space of the composite state machine is simply

$$\mathit{States} = \mathit{States}_A \times \mathit{States}_B. \tag{4.3}$$

This asserts that the composite state machine is in state (s_A, s_B) when $\mathit{StateMachine}_A$ is in state s_A and $\mathit{StateMachine}_B$ is in state s_B. The initial state is therefore

$$\mathit{initialState} = (\mathit{initialState}_A, \mathit{initialState}_B).$$

Note that we could have equally well defined the states of the composite state machine in the opposite order,

$$\mathit{States} = \mathit{States}_B \times \mathit{States}_A.$$

This would result in a *different* but *bisimilar* state machine description (either one simulates the other). Intuitively, it does not matter which of this two choices we make, so we choose (4.3).

To complete the definition of the composite machine, we need to define the *update* function in terms of the component machines. Here, a slight subtlety arises. Since this is a synchronous composition, the output of machine A is viewed as being *simultaneous* with its input. Thus, in a reaction, the output of machine A in that reaction must be available to machine B in the same reaction. This seems intuitive, but it has some counterintuitive consequences. Although the reactions of machine A and B are simultaneous, we must determine the reaction of A before we can determine the reaction of B. This apparent paradox underlies most of what is interesting in synchronous composition, and will become a major issue with feedback composition. In feedback composition, it is not immediately evident which reactions need to be determined first.

For now, it is intuitively obvious what we need to do. First, we determine the reaction of machine A. Assume that the current input is x and the current state is $s = (s_A, s_B)$, where s_A is the state of machine A and s_B is the state of machine B. Machine A reacts by updating its state and producing output y_A,

$$(s'_A, y_A) = \textit{update}_A(s_A, x). \tag{4.4}$$

Its output y_A becomes the input to machine B. Thus, machine B reacts by updating its state and producing output y_B,

$$(s'_B, y_B) = \textit{update}_B(s_B, y_A). \tag{4.5}$$

The output of the composite machine, of course, is just the output of machine B, and the next state of the composite machine is just (s'_A, s'_B), so

$$\textit{update}((s_A, s_B), x) = ((s'_A, s'_B), y_B),$$

where s'_A, s'_B, and y_B are given by (4.4) and (4.5). The definition of the composite machine summarized in figure 4.2.

Example 4.1: Consider the cascade composition in figure4.3. The composite machine has state space

$$\textit{States} = \{(0,0), (0,1), (1,0), (1,1))\}$$

and alphabets

$$\textit{Inputs} = \textit{Outputs} = \{0, 1, \textit{absent}\}.$$

The initial state is

$$\textit{initialState} = (0,0).$$

The *update* function is given by the table:

current	(*next state*, *output*) for input		
state	0	1	*absent*
(0,0)	((0,0),0)	((1,1),1)	((0,0), *absent*)
(0,1)	((0,0),1)	((1,1),0)	((0,1), *absent*)
(1,0)	((1,1),1)	((0,0),0)	((1,0), *absent*)
(1,1)	((1,1),0)	((0,0),1)	((1,1), *absent*)

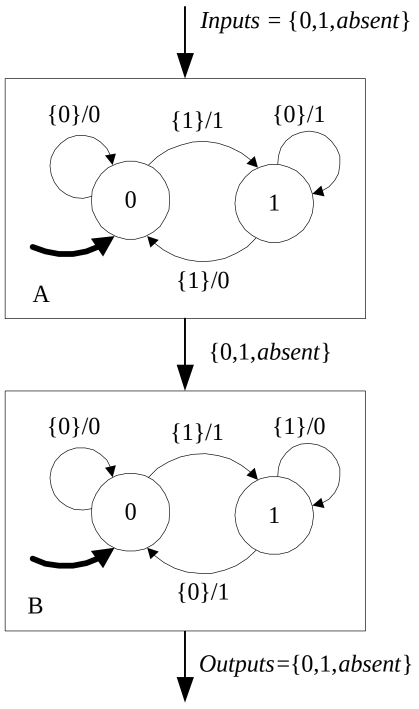


Figure 4.3: Example of a cascade composition.

Note from this table, however, that states $(0,1)$ and $(1,0)$ are not reachable from the initial state. A state s is said to be **reachable** if some sequence of inputs can take the state machine from the initial state to s. This suggests that a simpler machine with fewer states would exhibit the same input/output behavior. In fact, notice from the table that the input is always equal to the output! Thus, a totally trivial machine would exhibit the same input output behavior.

The simple behavior of the composite machine is not totally obvious from figure4.3. We have to systematically construct the composite machine to derive this simple behavior. In fact, this composite machine can be viewed as an encoder and decoder, because the input bit sequence is encoded by a distinctly different bit sequence (the intermediate signal in figure 4.3), and then the second machine, given the intermediate signal, reconstructs the original.

This particular encoder is known as a **differential precoder**. It is “differential” in the sense that when the input is 0, the intermediate signal sample is unchanged from the previous sample (whether it was 0 or 1), and when the input is 1, the sample is changed. Thus, the intermediate signal indicates *change* in the input with a 1, and *no change* with a 0. Differential precoders are used when it is important that the average number of 1’s and 0’s is the same, regardless of the data that is encoded.

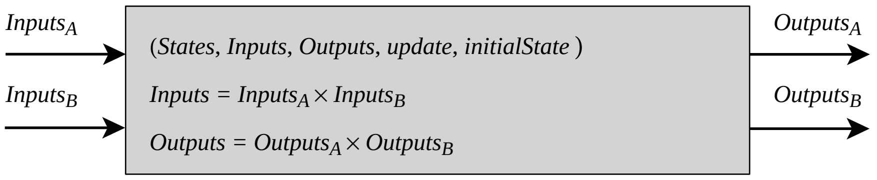

$Inputs_A$
$Inputs_B$
(States, Inputs, Outputs, update, initialState)
$Inputs = Inputs_A \times Inputs_B$
$Outputs = Outputs_A \times Outputs_B$
$Outputs_A$
$Outputs_B$

Figure 4.4: State machine with product-form inputs and outputs.

4.4 Product-form inputs and outputs

In the state machine model of (3.1), at each step the environment selects one input to which the machine reacts and produces one output. Sometimes we wish to model the fact that some input values are selected by one part of the environment, while other input values are simultaneously selected by another part. Similarly, some output values are sent to one part of the environment, while other output values are simultaneously sent to another part. The product-form composition of this section permits these models.

The machine in figure 4.1 is shown as a block with two distinct input and output arrows. The figure suggests that the machine receives inputs from two sources and sends outputs to two destinations. Furthermore, the two source inputs and the two destination outputs happen simultaneously. In the answering machine example of chapter 3, for instance, the *end greeting* input value might originate in a physically different piece of hardware in the machine than the *offhook* value.

The distinct arrows into and out of a block are called *ports*. Each port has a set of values called the **port alphabet** associated with it, as shown in figure 4.4. Each port alphabet must include a stuttering element. The set *Inputs* of input values to the state machine is the product of the input sets associated with the ports. Of course, the product can be constructed in any order, but each ordering results in a distinct (but bisimilar) state machine model.

For example, in figure 4.4, there are two input ports and two output ports. The upper input port can present to the state machine any value in the alphabet $Inputs_A$, which includes *absent*, its stuttering element. The lower port can present any value in the set $Inputs_B$, which also includes *absent*. The input value actually presented to the state machine in a reaction is taken from the set

$$Inputs = Inputs_A \times Inputs_B.$$

The stuttering element for this alphabet is the tuple $(absent, absent)$. The output value produced by a reaction is taken from the set

$$Outputs = Outputs_A \times Outputs_B.$$

If the output is (y_A, y_B), then the upper port shows y_A and the lower port shows y_B. These can now be separately presented as inputs to downstream state machines. Again, the stuttering element is $(absent, absent)$.

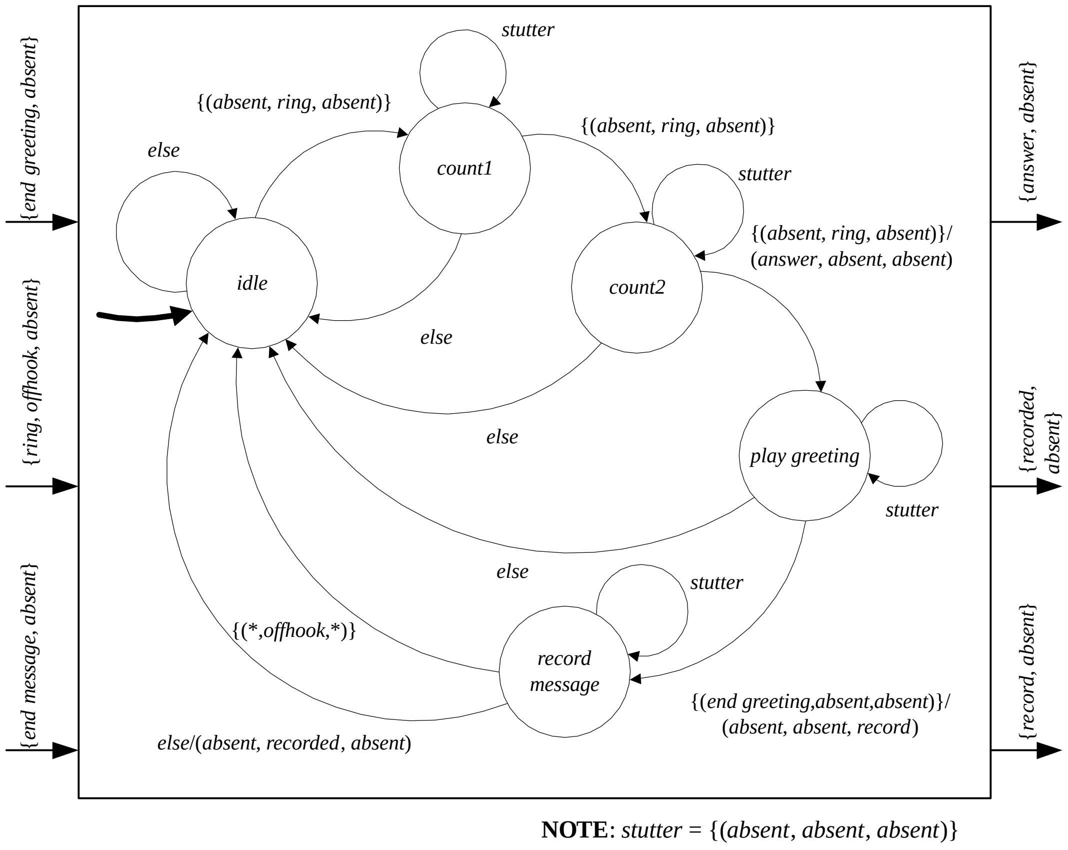


Figure 4.5: Answering machine with product-form inputs and outputs.

Example 4.2: The answering machine of figure 3.1 has input alphabet

$$Inputs = \{ring, offhook, end\ greeting, end\ message\}.$$

In a typical realization of an answering machine, *ring* and *offhook* come from a subsystem (often an **ASIC**, or **application-specific integrated circuit**) that interfaces to the telephone line. The value *end greeting* comes from a subsystem, such as a magnetic tape machine or digital audio storage device, that plays the answer message. The value *end message* comes from a similar, but distinct, subsystem that records incoming messages. So a more convenient model might show three separate factors for the inputs, as in figure 4.5. That figure also shows the outputs in product form, anticipating that the distinct output values will need to be sent to distinct subsystems.

Several features distinguish the diagram in figure 4.5 from that of figure 3.1. First, each state except the *idle* state has acquired a self-loop labeled *stutter*, which is a shorthand for

$$stutter = (absent, absent, absent).$$

This self loop prevents the state machine from returning to the idle state when nothing interesting is happening on the inputs. Usually, there will not be a reaction if nothing interesting is happening on the inputs, but because of synchrony, this machine may be composed with others, and all machines have to react at the same time. Thus, if anything interesting is happening anywhere in the system, then this machine has to react even though nothing interesting is happening here. Such a reaction is called a **stutter**. The state does not change, and the output produced is the stuttering element of the output alphabet.

Second, each guard now consists of a set of triples, since the product-form input has three components. The shorthand "(*, *offhook*, *)" on the arc from *record message* to *idle* represents the set

$$\{(absent, offhook, absent), (end\ greeting, offhook, absent),$$
$$(absent, offhook, end\ message), (end\ greeting, offhook, end\ message)\}.$$

The "*" is a **don't care** or **wildcard** notation. Anything in its position will trigger the guard.

The outputs are also triples, but most of them are implicitly $(absent, absent, absent)$.

4.5 General feedforward composition

Given that state machines can have product-form inputs and outputs, it is easy to construct a composition of state machines that has the features of both the cascade composition of figure 4.2 and the side-by-side composition of figure 4.1. An example is shown in figure 4.6. In that figure,

$$\begin{aligned} Outputs_A &= Outputs_{A1} \times Outputs_{A2} \\ Inputs_B &= Inputs_{B1} \times Inputs_{B2}. \end{aligned}$$

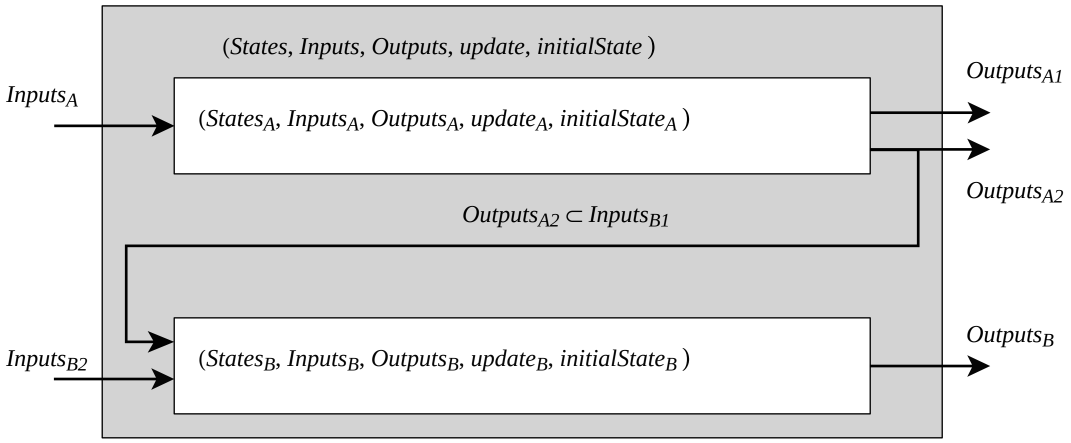

(States, Inputs, Outputs, update, initialState)
Inputs_A
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A1
Outputs_A2
Outputs_A2 ⊂ Inputs_B1
Inputs_B2
(States_B, Inputs_B, Outputs_B, update_B, initialState_B)
Outputs_B

Figure 4.6: More complex composition.

Notice that the bottom port of machine A goes both to the output of the composite machine and to the top port of machine B. Sending a value to multiple destinations like this is called **forking**. In exercise 1 at the end of this chapter you are asked to define the composite machine for this example.

Example 4.3:

We compose the answering machine of figure 4.5 with a playback system, shown in figure 4.7, which plays messages that have been recorded by the answering machine. The playback system receives the *recorded message* input from the answering machine whenever the answering machine is done recording a message. Its task is to light an indicator that a message is pending, and to wait for a user to push a button on the answering machine to request that pending messages be played back. When that button is pressed, all pending messages are played back. When they are done being played back, then the indicator light is turned off.

The composition is shown in figure 4.8. The figure shows a number of other components, not modeled as state machines, to help understand how everything works in practice. These other components are shown as three-dimensional objects, to emphasize their physicality. We have simplified the figure by omitting the *absent* elements of all the sets. They are implicit.

A telephone line interface provides *ring* and *offhook* when these are detected. Detection of one of these can trigger a reaction of the composite machine. In fact, any output from any of the physical components can trigger a reaction of the composite machine. When *AnsweringMachine* generates the *answer* output, then the “greeting playback device” plays back the greeting. From the perspective of the state machine model, all that happens is that time passes (during which some reactions may occur),

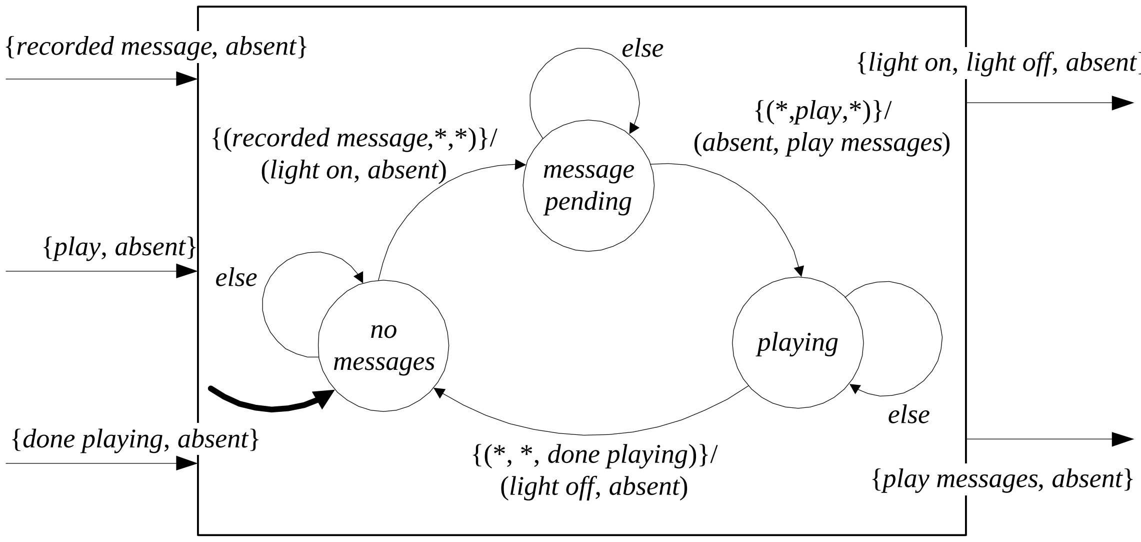

{recorded message, absent}
{play, absent}
{done playing, absent}
{light on, light off, absent}
{play messages, absent}
else
{(recorded message,*,*)}/ (light on, absent)
message pending
{(*,play,*)}/ (absent, play messages)
else
no messages
playing
else
{(*, *, done playing)}/ (light off, absent)

Figure 4.7: Playback system for composing with the answering machine.

and then an *end greeting* input is received. The recording device works similarly. When *AnsweringMachine* generates a *recorded message* output, then the *Playback* machine will respond by lighting the indicator light. When a user presses the "play button" the input *play* is generated, the composite machine reacts, and the *Playback* machine issues a *play messages* output to the "message playback device." This device also allows time to pass, then generates a *done playing* input to the composite state machine.

If we wish to model the playback or recording subsystem in more detail using finite state machines, then we need to be able to handle feedback compositions. These are considered below.

4.6 Hierarchical composition

By using the compositions discussed above, we can now handle any composition of state machines that does not have feedback. Consider for example the cascade of three state machines shown in figure 4.9. So far, all the composition techniques we have talked about involved only two state machines. It is easy to generalize the composition in figure 4.2 to handle three state machines (see exercise 2), but a more systematic method might be to apply the composition of figure 4.2 to compose two of the state machines, and then apply it again to compose the third state machine with the result of the first composition. This is called **hierarchical composition**.

In general, given a collection of interconnected state machines, there are several ways to hierarchically compose them. For example, in figure 4.9, we could first compose machines A and B to get, say, machine D, and then compose D with C. Or we could alternatively first compose B and C to get, say, machine E, and then compose E and A. These two techniques result in different but

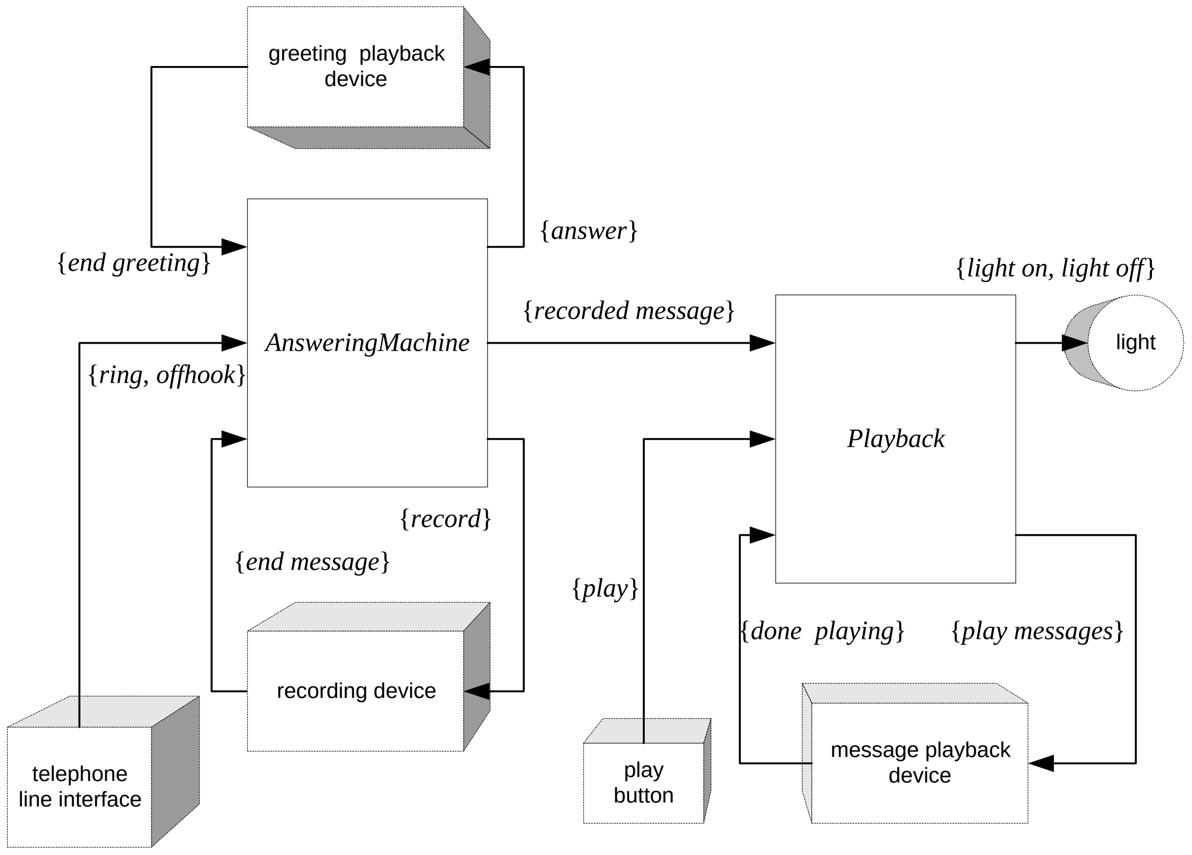


Figure 4.8: Composition of an answering machine with a message playback machine. The three-dimensional boxes are physical components that are not modeled as state machines. They are the sources of some inputs and the destinations of some outputs.

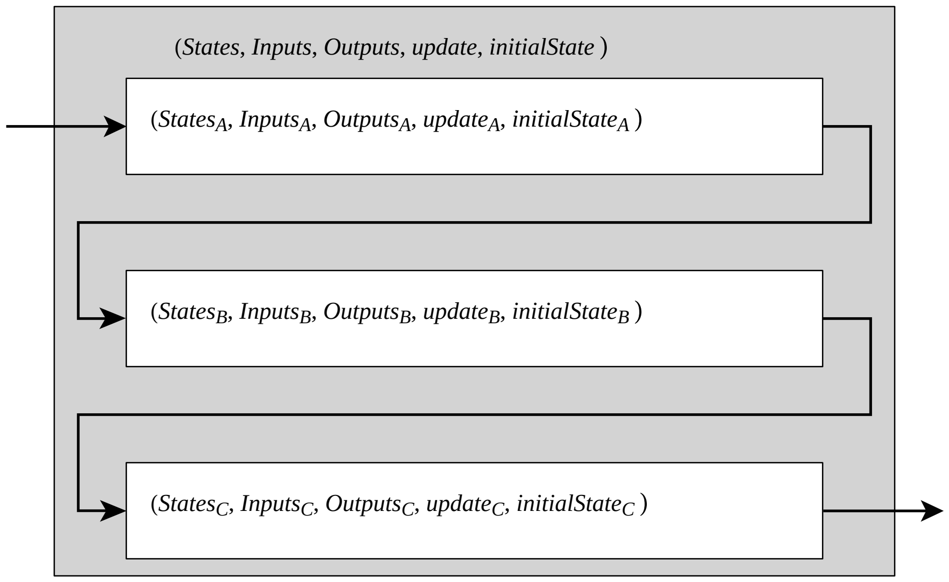

(*States*, *Inputs*, *Outputs*, *update*, *initialState*)
(*States*$_A$, *Inputs*$_A$, *Outputs*$_A$, *update*$_A$, *initialState*$_A$)
(*States*$_B$, *Inputs*$_B$, *Outputs*$_B$, *update*$_B$, *initialState*$_B$)
(*States*$_C$, *Inputs*$_C$, *Outputs*$_C$, *update*$_C$, *initialState*$_C$)

Figure 4.9: Cascade composition of three state machines. They can be composed in different ways into different, but bisimilar, state machines.

bisimilar state machine models (each simulates the other).

4.7 Feedback

In simple feedback systems, an output from a state machine is fed back to an input of the same state machine. In more complicated feedback systems, several state machines might be connected in a directed loop, where eventually the output of one affects its own input through some intervening state machines.

Feedback is a subtle form of composition under the synchronous model. In a synchronous composition, the output of a state machine is simultaneous with the input. Thus, the output of a machine in feedback composition depends on an input that depends on the output. Feedback, thus, involves self-referential logic.

Such self-referential logic is not all that unfamiliar. We see it all the time in systems of equations in mathematics. A simple and familiar problem is to find x such that

$$x = f(x) \tag{4.6}$$

for some function f. A solution to this equation, if it exists, is called a **fixed point** in mathematics. It is analogous to feedback because the "output" of f depends on its "input" and vice versa.

A more complicated fixed point problem in mathematics is to find x and y such that

$$x = f(y) \tag{4.7}$$

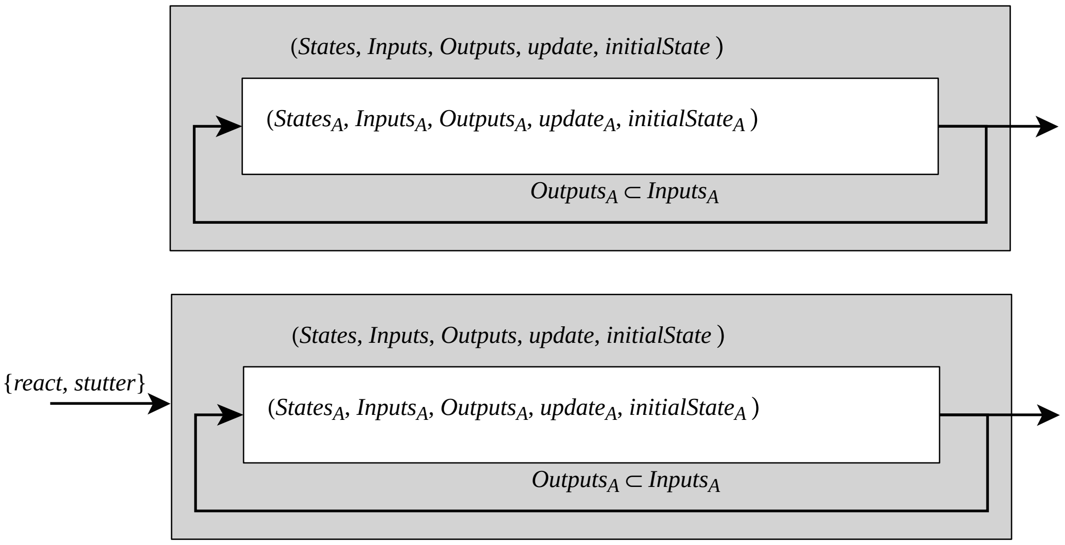

(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A ⊂ Inputs_A
{react, stutter}
(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A ⊂ Inputs_A

Figure 4.10: Feedback composition with no inputs.

and

$$y = g(x). \tag{4.8}$$

The analogous state machine problem will have two state machines with feedback.

Fixed-point equations like (4.6) may have no solution, a single solution, or multiple solutions. That is, they may have no fixed points, a unique fixed point, or multiple fixed points. When the fixed point is not unique, in the context of state machines, we will deem the feedback composition to be **ill-formed**. That is, we will refuse to evaluate it, and consider it defective. Fortunately, it is easy to construct **well-formed** feedback compositions, those with a unique fixed point, which prove surprisingly useful.

Some fixed-point problems are easy to solve. Consider for example the function $f\colon Nats \to Nats$ where $\forall x \in Nats,\quad f(x) = x^2$. In this case, (4.6) becomes $x = x^2$, which has two fixed points, namely $x = 0$ and $x = 1$. Unfortunately, state machines are rarely so accommodating. It will take somewhat more elaborate techniques.

4.7.1 Feedback composition with no inputs

Consider the upper state machine in figure 4.10, which has an output port that feeds back to an input port. We wish to construct a state machine model that hides the feedback, as suggested by the figure. The result would be a state machine with no input or ports, which does not fit our model well. We therefore artificially provide an input alphabet

$$Inputs = \{react, stutter\},$$

as suggested in the lower machine. We will interpret the input *react* as a command for the internal machine to react, and the input *stutter* as a command for the internal machine to stutter. The output

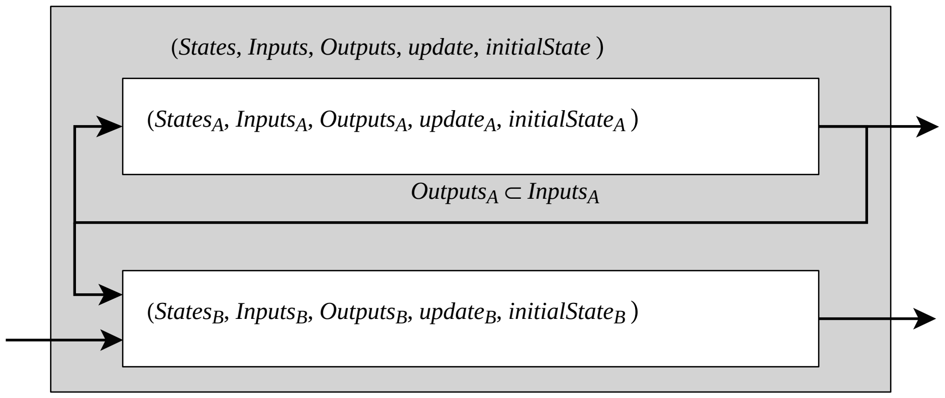

(States, Inputs, Outputs, update, initialState)
(States_A, Inputs_A, Outputs_A, update_A, initialState_A)
Outputs_A ⊂ Inputs_A
(States_B, Inputs_B, Outputs_B, update_B, initialState_B)

Figure 4.11: Feedback composition composed with another state machine.

alphabet is

$$\textit{Outputs} = \textit{Outputs}_A.$$

This example is a bit odd for a synchronous/reactive system because of the need for this artificial input alphabet. Recall that the reactions of a reactive system are driven by external inputs. Typically, such a system will be composed with others, as suggested in figure 4.11. Since that composition does have inputs, the subsystem with no inputs will react whenever the overall composition reacts. The interpretation there is clearer. When a stuttering element is provided to the composite, all components stutter. Otherwise, they react.

We consider the example in figure 4.10 first because, although it seems odd, the formulation of the composition is simplest. We will augment the model to allow inputs and outputs after this.

In figure 4.10, for the feedback connection to be possible, of course, we have to assume

$$\textit{Outputs}_A \subset \textit{Inputs}_A.$$

To determine the output, we need to solve the following equation. Given that the current state is $s \in \textit{States}_A$, find $s' \in \textit{States}_A$ and $b \in \textit{Outputs}_A$ such that

$$(s', b) = \textit{update}_A(s, b).$$

One solution that is always available is to stutter,

$$s' = s \text{ and } b = \textit{absent},$$

assuming that *absent* is the stuttering input for machine A. To find a non-stuttering reaction, it is sufficient, of course, to find b, and then compute s' by applying the *update* function. Define the function

$$\textit{output}_A \colon \textit{States}_A \times \textit{Inputs}_A \to \textit{Outputs}_A$$

This function gives the output as a function of the current state and the current input. Then we simply need to find a non-stuttering b that satisfies

$$b = \mathit{output}_A(s, b). \tag{4.9}$$

This is non-trivial because b appears on both sides. Except for the addition of s, which is a known constant when we go to solve this equations, this has exactly the form of (4.6). A solution, if it exists, is called a fixed point.

Finding this fixed point requires using the detailed definition of the state machine. We will do this first mathematically, and then show by example how the solution procedure can be surprisingly easily applied using a state transition diagram. The procedure will consist of first assuming that the value fed back (that is, b) is unknown, and then seeing whether the output can be determined even though the input is unknown. Often it can be. When it can be, then the output becomes determined, which then determines the input. When it cannot be, then we declare the feedback composition to be ill-formed.

Mathematically, this procedure involves augmenting the domain and range of the function *output*$_A$ to include an element that we will call *unknown*.[2] For the example in figure 4.10, we define the augmented sets

$$\begin{aligned} \mathit{Inputs}'_A &= \mathit{Inputs}_A \cup \{\mathit{unknown}\} \\ \mathit{Outputs}'_A &= \mathit{Outputs}_A \cup \{\mathit{unknown}\}. \end{aligned}$$

We then define a new function

$$\mathit{output}'_A : \mathit{States}_A \times \mathit{Inputs}'_A \to \mathit{Outputs}'_A. \tag{4.10}$$

This is just like the original function except that it can accept the value *unknown* in its second argument and may generate the value *unknown*.

We define the function *output*$'_A$ as follows. Given

$$s \in \mathit{States}_A \text{ and } a \in \mathit{Inputs}'_A,$$

there are two possible conditions:

1. $a \neq \mathit{unknown}$. In this case we define $\mathit{output}'_A(s, a) = \mathit{output}_A(s, a)$. This is possible because since a is not *unknown*, (s, a) is in the domain of *output*$_A$.
2. $a = \mathit{unknown}$. In this case, there are two possible outcomes.
   (a) There is a unique value $b \in \mathit{Outputs}_A$ such that

   $$\forall\, c \in \mathit{Inputs}_A \text{ where } c \neq \text{stuttering element}, \quad \mathit{output}_A(s, c) = b. \tag{4.11}$$

   That is, for all possible inputs, the output is always the same. Then we define

   $$\mathit{output}'_A(s, a) = \mathit{output}_A(s, b).$$

   In other words, the output depends only on the state, so we can determine the output even though the input is unknown.

[2] This element is often called **bottom** in the literature, for mathematical reasons that are beyond the scope of this text. In brief, bottom is the least element of a partial order. Note that *unknown* is different from *absent*.

(b) There is no such b, in which case we define

$$\mathit{output}'_A(s, a) = \mathit{unknown}.$$

In the last of these cases, we have found a fixed point of the function output'_A, namely *unknown*. This is not an acceptable answer for the function output_A (it is not in its domain), so we declare the state machine to be ill-formed. More specifically, we say that it has a **causality loop**, a self-referential logic that cannot be resolved.

A summary of the composite machine definition is:

$$\begin{aligned}
&\mathit{States} = \mathit{States}_A \\
&\mathit{Inputs} = \{\mathit{absent}\} \\
&\mathit{Outputs} = \mathit{Outputs}_A \\
&\mathit{initialState} = \mathit{initialState}_A \\
&\mathit{update}(s, x) = \begin{cases} \mathit{update}_A(s, b), \text{where } b \text{ satisfies (4.11)} & \text{if } x = \mathit{react} \\ (s, x) & \text{if } x = \mathit{stutter} \end{cases}
\end{aligned}$$

Notice that this is only valid if the state machine is well formed, i.e., there is a b that satisfies (4.11). If there is no such b, then the machine is not well formed.

Example 4.4: Consider the examples shown in figure 4.12. In all three cases, assume that the input and output alphabets of the component machines are

$$\mathit{Inputs}_A = \mathit{Outputs}_A = \{\mathit{true}, \mathit{false}, \mathit{absent}\}.$$

Let us apply the procedure to find b in (4.11) for each case, assuming the input $a' = \mathit{unknown}$.

Consider the top machine first, and assume the current state is the initial state, $s = 1$. Notice that there are two outgoing arcs, and that for a non-stuttering input, both produce $b = \mathit{false}$, so we can conclude that the output of the machine is *false*. Since the output is *false*, then the input is also *false*, so the state transition taken by the reaction goes from state 1 to state 2.

Suppose next that the current state is $s = 2$. Again, there are two outgoing arcs. Both outgoing arcs produce the output *true* for a non-stuttering input, so we can conclude that the output is *true*. Since the output is *true*, then so is the input, and the state transition taken goes from 2 to 1.

This machine, therefore, alternates states on each reaction, and produces the output sequence

$$(\mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}, \cdots)$$

given the input sequence

$$(\mathit{react}, \mathit{react}, \mathit{react}, \cdots).$$

The feedback composition is well-formed.

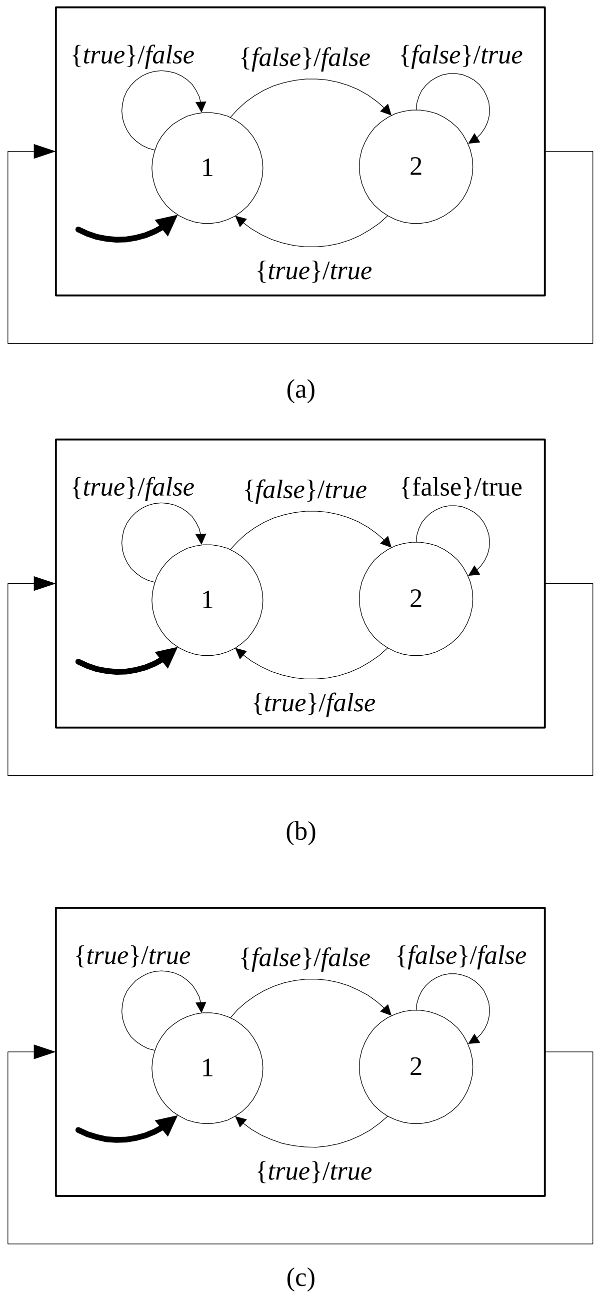


Figure 4.12: Three examples of feedback composition.

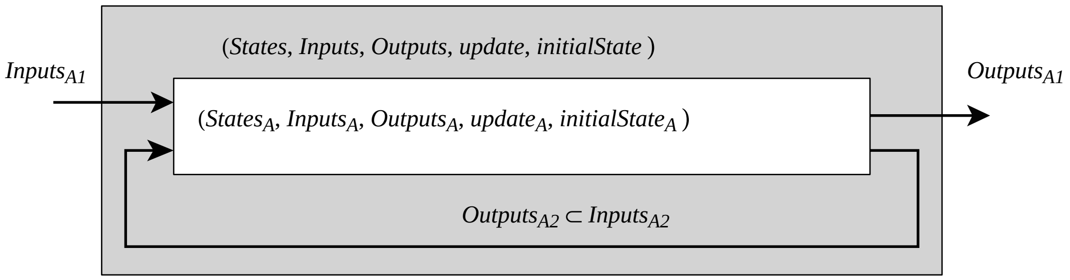

(States, Inputs, Outputs, update, initialState)
Inputs$_{A1}$
(States$_A$, Inputs$_A$, Outputs$_A$, update$_A$, initialState$_A$)
Outputs$_{A1}$
Outputs$_{A2}$ ⊂ Inputs$_{A2}$

Figure 4.13: Feedback composition of a state machine.

Consider by contrast the second machine in figure 4.12. If the initial state is 1 and the input is unknown, then there are two possible outgoing transitions, but they produce different output values. Thus, we cannot determine the output. The same is true of state 2. The feedback composition is not well-formed. In fact, by inspection, you can see that no output works. If the output is *true*, then a transition must be taken that produces the output *false*, so the output must be *false*. But if the output is *false*, then a transition is taken that produces the output *true*, so the output must be *true*. This is a contradiction.

Consider the third machine in figure 4.12. The same procedure fails to progress beyond *unknown* for this example, so this feedback composition is also ill-formed. However, the situation is not quite the same as in the middle example. By inspection, we can determine that if the output is *true*, then a transition will be taken that will produce *true* for a non-stuttering input. Thus, in this case, the output can be *true*! However, the output can also be *false*, since if it is, then a transition will be taken that produces the output *false*. Thus, the problem here is that there is more than one solution, not that there are none!

Our conclusion is that with machines like the second and third, you cannot connect them in a feedback composition as shown. The second and third machines are rejected equally vigorously by our procedure, even though the second has no solution and the third has more than one. We accept only solutions that where there is exactly one solution (below we will generalize this to nondeterministic machines, but we will still reject compositions as in the third example).

4.7.2 Feedback composition with inputs

Consider the state machine in figure 4.13, which has an output port that feeds back to an input port. We wish to construct a state machine model that hides the feedback, as suggested by the figure, and becomes a simple input/output state machine. This is similar to the example in figure 4.10, but now there is an additional input and an additional output. The approach is very similar, although the notation is more cumbersome, so we spare the reader the details.

Probing further: Least fixed point

The fact that we reject the second and third examples in figure 4.12 as being ill-formed may be objectionable to some readers. The third example, in particular, has more than one behavior. Why not model the composition as a nondeterministic state machine? There are very good mathematical reasons for rejecting it. Our basis for rejecting it is that we define the behavior of a composition to be the **least fixed point** (excluding the stuttering fixed point) of the augmented model (the one including *unknown*) when there is more than one fixed point. Our procedure, in fact, is guaranteed to find this least fixed point, even when large, multi-machine compositions are considered. Moreover, this least fixed point is guaranteed to be unique. The mathematical basis for these observations, which were first developed by Dana Scott around 1970, are deep and very robust.

The mathematical formulation constructs what is known as a **Scott topology**, which is an ordering of values where *unknown* is considered to be "less than" all other values, and all other values (including *absent*) are considered to be **incomparable** (neither less than nor greater than one another). This structure turns out to be a topological space with some very powerful properties. It is a special case of a class of mathematical structures known as **partial orders**. The $output'_A$ function of (4.10) is **monotonic nondecreasing** in this ordering, meaning in this simple case that it never yields *unknown* when the input is known. Monotonic functions in this topological space can be proven to have exactly one least fixed point, and moreover, that fixed point can always be found in finite time.

The robustness of this mathematical formulation means that it extends comfortably to much more complicated compositions, where there are multiple machines and multiple feedback paths. Any solution that embraces the third example in figure 4.12 is unlikely to be so robust. In that example, $output'_A$ has three fixed points, *unknown*, *true*, and *false*. Without the least fixed point principle, we have no basis for choosing among these. The reason for the three behaviors is a causality loop, where the causal relationships among values are circular.

In figure 4.12(b), the fixed point is unique, *unknown*. We reject this because it is not a very satisfying answer. It also indicates a causality loop.

A dense but complete and readable introduction to the mathematics of partial orders can be found in B. A. Davey and H. A. Priestly, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

Applying the procedure is much easier than defining it precisely in mathematical terms. The procedure is to assign the value *unknown* to the second (bottom) input of the A machine. If without knowing the value of that input we can determine the value of the second output of machine A, then the feedback composition is well-formed. We use that value of the second output as the value of the second input. We can now evaluate the output_A function to determine the output, and then evaluate *update* using that output as an input.

4.7.3 Feedback composition of multiple machines

Our examples so far involved only a single state machine and a feedback loop. Most interesting scenarios are more complicated, involving several state machines and several feedback loops. Our procedure for evaluating the state machines can be extended to such scenarios easily. The approach is simple. At each reaction, begin with all unspecified signals having value *unknown*. Then with what is known about the inputs, try each state machine to determine as much as possible about the outputs. You can try the state machines in any order. Given what you have learned about the outputs, then update what you know about the feedback inputs, and repeat the process, trying each state machine again. Repeat this process until all signal values are specified, or until you learn nothing more about the outputs. If any signals remain *unknown*, then you have a causality loop.

Example 4.5: Consider the example in figure 4.14. Here, two of the state machines from figure 4.12 have been connected in a feedback loop. One approach to getting the behavior of the composite would be to define a state machine for the serial connection of these two (in either order), and then use that state machine in the structure of figure 4.10. However, we can directly evaluate the behavior of the composite machine using a generalization of our procedure.

Assume all inputs to the composite are *react*, not *stutter*. We are interested in the state trajectory and output sequence of the composite machine. The state space is

$$\mathit{States} = \{1, 2\} \times \{1, 2\}$$

and the output space is

$$\mathit{Outputs} = \{\mathit{true}, \mathit{false}, \mathit{absent}\}, \{\mathit{true}, \mathit{false}, \mathit{absent}\}.$$

The input alphabet is

$$\mathit{Inputs} = \{\mathit{react}, \mathit{absent}\},$$

and the initial state is

$$\mathit{initialState} = (1, 1).$$

To apply our procedure, we first assume both outputs (and the inputs to the component state machines) are *unknown*. We then attempt to determine the outputs of the component state machines (in any order). Let us begin with the lower one. It is in state 1, and the input is *unknown*, so the output remains *unknown*, and we make no progress. Turning our attention to the upper one, its input is *unknown* and state is 1, but we can infer

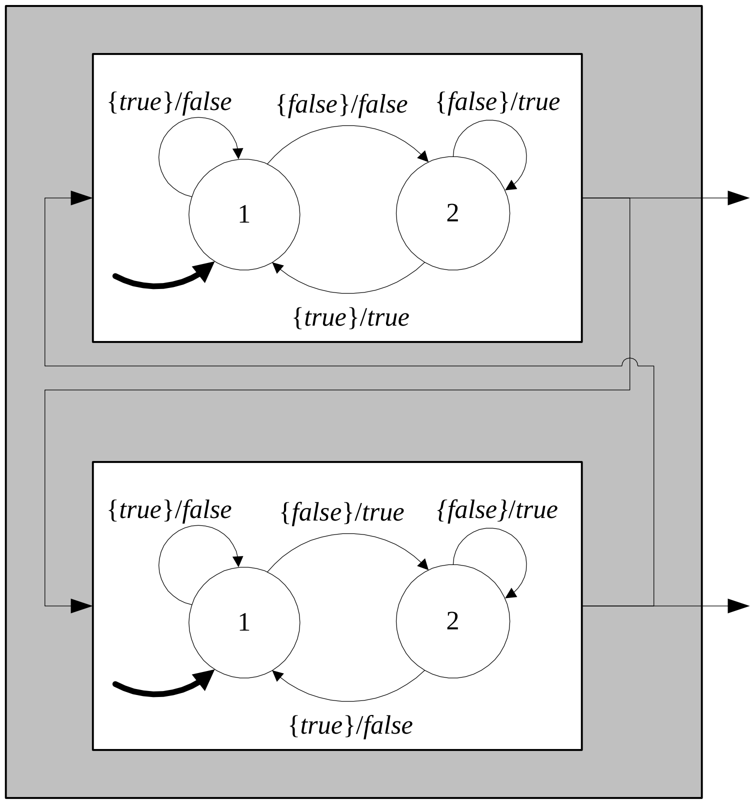


Figure 4.14: Feedback composition with two state machines.

that the output must be *false* because both outgoing transitions from state 1 produce *false*. Thus, we have made some progress on this round.

In the second round, we again turn attention to the lower machine. Its input is now *false*, from which we can infer that it will transition to state 2 and output a *true*. Turning attention to the upper machine, the input is now *true*, from which we infer that it will transition back to state 1 along the self loop (and we already know that it would output *false*). Now the state transition and output of the composite machine for the first reaction is known.

Continuing in this fashion, we can determine that the state trajectory will be

$$((1,1),(1,2),(1,2),(1,2),\cdots)$$

and the output sequence will be

$$((\mathit{false},\mathit{true}),(\mathit{false},\mathit{true}),\cdots).$$

The state machine gets stuck in state $(1,2)$.

This procedure can be applied in general to any composition of state machines. In summary, the procedure is to assign the value *unknown* to any unknown signals, and then to proceed through a series of rounds until a round yields no further information. Each round consists of examining each state machine (in any order) and, given what you know about its inputs, asserting what you can infer about its outputs.

Example 4.6: We wish to add more detail to the message recorder in figure 4.8. In particular, as shown in figure 4.15, we wish to model the fact that the message recorder stops recording when either it detects a dialtone or when a timeout period is reached. This is modeled by a two-state finite state machine, shown in figure 4.16.

The *MessageRecorder* and *AnsweringMachine* state machines form a feedback loop. Let us verify that there is no causality loop. First, note that in the *idle* state of the *MessageRecorder*, the upper output is known to be *absent* (see figure 4.16). Thus, only in the *recording* state is there any possibility of a causality loop. In that state, the output is not known unless the inputs are known. However, notice that the *recording* state is entered only when a *record* input is received. In figure 4.5, you can see that the *record* value is generated only when entering state *record message*. But in all arcs emerging from that state, the lower output of *AnsweringMachine* will always be absent; the input does not need to be known to know that. Continuing this reasoning by considering all possible state transitions from this point, we can convince ourselves that no causality loop emerges.

The sort of reasoning in this more complicated example is difficult and error prone for even moderate compositions of state machines. It is best automated. Compilers for synchronous languages do exactly this. Successfully compiling a program involves proving that causality loops cannot occur.

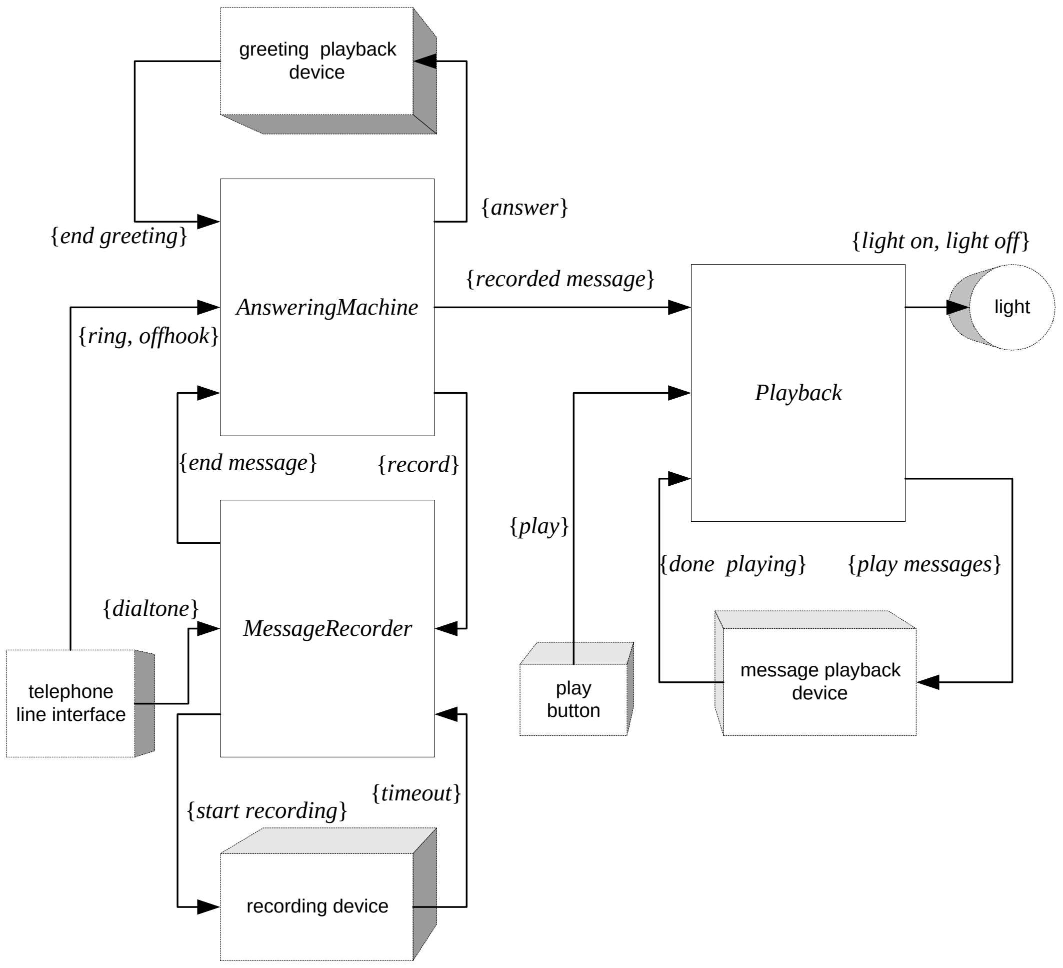


Figure 4.15: Answering machine composition with feedback. The *absent* elements are not shown (to reduce clutter)

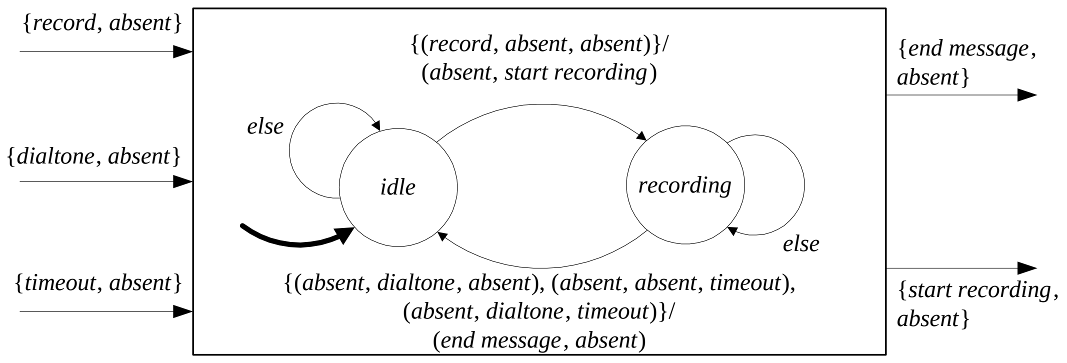

{record, absent}
{dialtone, absent}
{timeout, absent}
{(record, absent, absent)}/
(absent, start recording)
else
idle
recording
else
{(absent, dialtone, absent), (absent, absent, timeout),
(absent, dialtone, timeout)}/
(end message, absent)
{end message, absent}
{start recording, absent}

Figure 4.16: Message recorder subsystem of the answering system.

4.8 Nondeterministic machines

Nondeterministic state machines can be composed just as deterministic state machines are composed. In fact, since deterministic state machines are a special case, the two types of state machines can be mixed in a composition. Compositions without feedback are straightforward, and operate almost exactly as described above (see exercises 11 and 10). Compositions with feedback require a small modification to our evaluation process.

Recall that to evaluate the reaction of a feedback composition, we begin by setting to *unknown* any inputs that are not initially known. We then proceed through a series of rounds where in each round, we attempt to determine the outputs of the state machines in the composition given what we know about the inputs. After some number of rounds, no more information is gained. At this point, if the composition is well-formed, then all the inputs and outputs are known.

This process needs to be modified slightly for nondeterministic machines because in each reaction, a machine may have several possible outputs and several possible next states. For each machine, we define the sets $PossibleInputs \subset Inputs$, $PossibleNextStates \subset States$ and $PossibleNextOutputs \subset Outputs$. If the inputs to a particular machine in the composition are known completely, then *PossibleInputs* has exactly one element. If they are completely unknown, then *PossibleInputs* is empty.

The rounds proceed in a similar fashion to before. For each state machine in the composition, given what is known about the inputs, i.e. given *PossibleInputs*, determine what you can about the next state and outputs. This may result in elements being added to *PossibleNextStates* and *PossibleNextOutputs*. When a round results in no such added elements, the process has converged.

Exercises

In some of the following exercises you are asked to design state machines that carry out a given task. The design is simplified and elegant if the state space is properly chosen. Although the state space is not unique, there often is a natural choice. Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Define the composite state machine in figure 4.6 in terms of the component machines, as done for the simpler compositions in figures 4.2 and 4.1. Be sure to state any required assumptions.

2. **E** Define the composite state machine in figure 4.9 in terms of the component machines, as done for the simpler compositions in figures 4.2 and 4.1. Be sure to state any required assumptions. Give the definition in two different ways:

   (a) Directly form a product of the three state spaces.

(b) First compose the A and B state machines to get a new D state machine, and then compose D with C.

(c) Comment on the relationship between the models in part (a) and (b).

3. **T** Consider the state machine *UnitDelay* studied in part (a) of exercise 5 at the end of the previous chapter.

   (a) Construct a state machine model for a cascade composition of two such machines. Give the sets and functions model (it is easier than the state transition diagram or table).

   (b) Are all of the states in the state space of your model in part (a) reachable? If not, give an example of an unreachable state.

   (c) Give the state space (only) for cascade compositions of three and four unit delays. How many elements are there in each of these state spaces?

   (d) Give an expression for the size of the state space as function of the number N of cascaded delays in the cascade composition.

4. **C** Consider the parking meter example of the previous chapter, example 3.1, and the modulo N counter of exercise 4 at the end of the previous chapter. Use these two machines to model a citizen that parks at the meter when the machines start, and inserts 25 cents every 30 minutes, and a police officer who checks the meter every 45 minutes, and issues a ticket if the meter is expired. For simplicity, assume that the police office issues a new ticket each time he finds the meter expired, and that the citizen remains parked forever.

   You may construct the model at the block diagram level, as in figure 4.8, but describe in words any changes you need to make to the designs of the previous chapter. Give state transition diagrams for any additional state machines you need. How long does it take for the citizen to get the first parking ticket?

   Assume you have an eternal clock that emits an event *tick* every minute.

   Note that the output alphabet of the modulo N counter does not match the input alphabet of the parking meter. Neither does its input alphabet match the output alphabet of the parking meter. Thus, one or more intermediate state machines will be needed to translate these alphabets. You should fully specify these state machines (i.e., don't just give them at the block diagram level). **Hint:** These state machines, which perform an operation called **renaming**, only need one state.

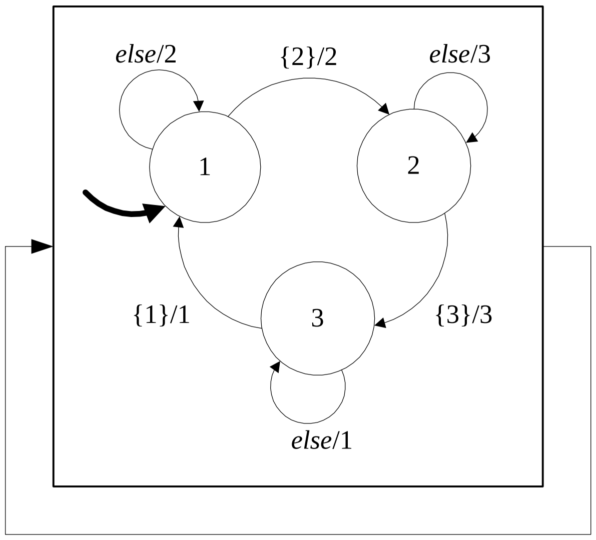
5. **C** A road has a pedestrian crossing with a traffic light. The light is normally green for vehicles, and the pedestrian is told to wait. However, if a pedestrian presses a button, the light turns yellow for 30 seconds and then red for 30 seconds. When it is red, the pedestrian is told "cross now." After the 30 seconds of red, the light turns back to green. If a pedestrian presses the button again while the light is red, then the red is extended to a full minute.

   Construct a composite model for this system that has at least two state machines, *TrafficLight* for the traffic light seen by the cars, and *WalkLight* for the walk light seen by the pedestrians. The state of machine should represent the state of the lights. For example, *TrafficLight* should have at least three states, one for green, one for yellow, and one for red. Each color may, however, have more than one state associated with it. For example, there may be more than one state in which the light is red. It is typical in modeling systems for the states of the model to represent states of the physical system.

Assume you have a timer available such that if you emit an output *start timer*, then 30 seconds later an input *timeout* will appear. It is sufficient to give the state transition graphs for the machines. State any assumptions you need to make.

6. **C** Recall the playback machine of figure 4.7 and the *CodeRecognizer* machine of Figure 3.4. Enclose *CodeRecognizer* in a block and compose it with the playback machine so that someone can play back the recorded messages only if she correctly enters the code 1100.

7. **E** Consider the following state machine in a feedback composition, where the input alphabet for the state machine is $\{1, 2, 3, \textit{absent}\}$:



Is it well-formed? If so, then find the outputs for the first 10 reactions.

8. **E** In this problem, we will explore the fact that a carefully defined delay in a feedback composition always makes the composition well-formed.

   (a) For an input and output alphabet

$$\textit{Inputs} = \textit{Outputs} = \{\textit{true}, \textit{false}, \textit{absent}\}$$

   design a state machine that outputs *false* on the first reaction, and then in subsequent reactions, outputs the value observed at the input in the previous reaction. This is similar to *UnitDelay* of problem 5 at the end of chapter 3, with the only difference being that it outputs an initial *false* instead of *absent*.

   (b) Compose the machine in figure 4.12 (b) with the delay from part (a) of this problem in a feedback loop (as in figure 4.14). Give an argument that the composition is well-formed. Then do the same for figure 4.12 (c) instead of (b).

9. **C** Construct a feedback state machine with the structure of figure4.10 that outputs the periodic sequence $a, b, c, a, b, c \cdots$.

10. **E** Modify figure 4.1 as necessary so that the machines in the side-by-side composition are both nondeterministic.

11. **E** Modify figure 4.2 as necessary so that the machines in the cascade composition are both nondeterministic.

12. **C,T** Data packets are to be reliably exchanged between two computers over communication links that may lose packets. The following protocol has been suggested. Suppose computer A is sending and B is receiving. Then A sends a packet and starts a timer. If B receives the packet it sends back an acknowledgment. (The packet or the acknowledgment or both may be lost.) If A does not receive the acknowledgment before the timer expires, it retransmits the packet. If the acknowledgment arrives before the timer expires, A sends the next packet.

    (a) Construct two state machines, one for A and one for B, that implement the protocol.

    (b) Construct a two-state nondeterministic machine to model the link from A to B, and another copy to model the link from B to A. Remember that the link may correctly deliver a packet or it may lose it.

    (c) Compose the four machines to model the entire system.

    (d) Suppose the link correctly delivers a packet, but after a delay that exceeds the timer setting. What will happen?

Chapter 5

Linear Systems

Recall that the **state** of a system is a summary of its past. It is what it is needs to remember about the past in order to react at the present and move into the future. In previous chapters, systems typically had a finite number of possible states. Many useful and interesting systems are not like that, however. They have an infinite number of states. The analytical approaches used to analyze finite-state systems, such as simulation and bisimulation, get much more difficult when the number of states is not finite.

In this chapter, we begin considering infinite-state systems. There are two key constraints that we impose. First, we require that the state space and input and output alphabets be numeric sets. That is, we must be able to do arithmetic on members of these sets. (Contrast this with the answering machine example, where the states are symbolic names, and no arithmetic makes sense.) Second, we require that the *update* function be linear. We will define what this means precisely. In exchange for these two constraints, we gain a very rich set of analytical methods for designing and understanding systems. In fact, the next five chapters are devoted to developing these methods.

In particular, we study state machines with

$$\begin{aligned} \mathit{States} &= \mathit{Reals}^N \\ \mathit{Inputs} &= \mathit{Reals}^M \\ \mathit{Outputs} &= \mathit{Reals}^K. \end{aligned} \tag{5.1}$$

Such state machines are shown schematically in figure 5.1. The inputs and outputs are in product form, as discussed for general state machines in section 4.4. The system, therefore, can be viewed as having M distinct inputs and K distinct outputs. Such a system is called a **multiple-input, multiple-output system**, or **MIMO** system. When $M = K = 1$, it is called a **single-input, single-output system**, or **SISO** system. The state is a tuple with N real numbers. The value of N is called the **dimension** of the system.

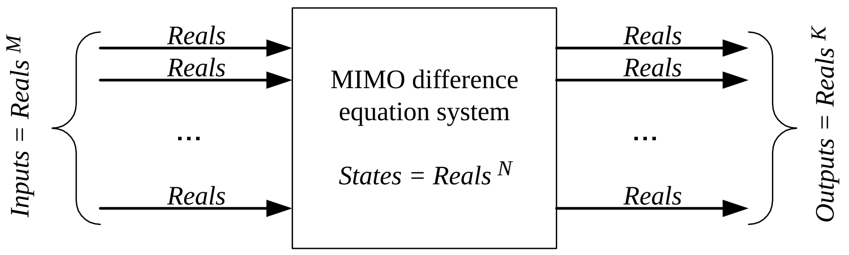


Figure 5.1: Block representing a multiple-input, multiple output difference equation system.

5.1 Operation of an infinite state machine

Recall that a deterministic state machine is a 5-tuple

$$M = (\textit{States}, \textit{Inputs}, \textit{Outputs}, \textit{update}, \textit{initialState}) \tag{5.2}$$

where *States* is the state space, *Inputs* is the input space, *Outputs* is the output space, $\textit{update}\colon \textit{States} \times \textit{Inputs} \to \textit{States} \times \textit{Outputs}$ is the update function, and *initialState* is the initial state.

In this chapter, the *update* function has the form

$$\textit{update}\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals}^N \times \textit{Reals}^K.$$

The result of evaluating this function is an N-tuple (the new state) and a K-tuple (the output). It will be useful in this chapter to break this function into two parts, one giving the new state and one giving the output,

$$\textit{update} = (\textit{nextState}, \textit{output})$$

where

$$\textit{nextState}\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals}^N$$

$$\textit{output}\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals}^K$$

such that

$$\forall\, s \in \textit{Reals}^N, x \in \textit{Reals}^M, \quad \textit{update}(s, x) = (\textit{nextState}(s, x), \textit{output}(s, x)).$$

These two functions separately give the state update and the output as a function of the current state and the input. Given an input sequence $x(0), x(1), \cdots$ of M-tuples in *Reals*^{M}, the system generates a state response $s(0), s(1), \cdots$ of N-tuples in *Reals*^{N} and an output sequence $y(0), y(1), \cdots$ of K-tuples in *Reals*^{K} as follows:

$$\begin{aligned} s(0) &= \textit{initialState}, \\ (s(n+1), y(n)) &= \textit{update}(s(n), x(n)), \quad n \geq 0 \end{aligned} \tag{5.3}$$

Basics: Arithmetic on tuples of real numbers

An element $s \in \textit{Reals}^N$ is an N**-tuple**. The components of $s \in \textit{Reals}^N$ are denoted $s = (s_1, \cdots, s_N)$; similarly $x = (x_1, \cdots, x_M) \in \textit{Reals}^M$, and $y = (y_1, \cdots, y_K) \in \textit{Reals}^K$. These components, s_i, x_i, y_i, are all real numbers, i.e. elements of *Reals*. A tuple is similar to an **array** in programming languages.

In general, a tuple can contain as elements members of any set. In this chapter, however, we are using a very special set, *Reals*. This set is numeric, and we can perform arithmetic on members of the set (addition, subtraction, multiplication, division). We wish to also be able to perform arithmetic on tuples composed of members of the set.

The sum of two tuples of the same dimension is defined to be the tuple of the element-wise sum. I.e. if $x \in \textit{Reals}^N$ and $w \in \textit{Reals}^N$, then

$$x + w = (x_1 + w_1, \cdots, x_N + w_N).$$

Subtraction is similarly defined. To perform more elaborate arithmetic operations, such as multiplication, we will need matrices and vectors. Tuples will not be sufficient.

The second equation can be rewritten as a separate **state update equation**,

$$\boxed{\forall\, n \in \textit{Ints},\ n \geq 0, \quad s(n+1) = \textit{nextState}(s(n), x(n))} \tag{5.4}$$

and an **output equation**,

$$\boxed{\forall\, n \in \textit{Ints},\ n \geq 0, \quad y(n) = \textit{output}(s(n), x(n)).} \tag{5.5}$$

These equations are collectively called a **state-space model** of the system, because instead of giving the output directly as a function of the input, the state is explicitly described. The equations suggest a detailed procedure for calculating the response of a system. We start with a given initial state $s(0) = \textit{initialState}$, and an input sequence $x(0), x(1), \cdots$. At step $n = 0$, we evaluate the right-hand side of (5.4) at the known values of $s(0), x(0)$ and we assign the result to $s(1)$. At step $n = 1$, we evaluate the right-hand side at the known values of $s(1), x(1)$ and we assign the result to $s(2)$. To proceed from step n to step $n+1$, we only need to remember $s(n)$ and know the new input $x(n)$. At each step n, we evaluate the output $y(n)$ using (5.5). This procedure is no different from that used for state machines in previous chapters. However, we will specialize the *nextState* and *output* functions so that they are linear, which will then lead to a powerful set of analytical tools.

5.1.1 Time

The index n in the equations above denotes the **step** number, the count of reactions, as with any state machine. For general state machines, it is rare to associate a fixed time interval with a step.

Basics: Functions yielding tuples

The ranges of the *nextState* and *output* functions are tuples. It will help to understand their role if we break them down further into an N-tuple and K-tuple of functions, one for each element of the result tuple. That is, we define the functions

$$\textit{nextState}_i\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals},\ i = 1, \cdots, N,$$

such that $\forall\ s \in \textit{Reals}^N, x \in \textit{Reals}^M$,

$$\textit{nextState}(s, x) = (\textit{nextState}_1(s, x), \cdots, \textit{nextState}_N(s, x)).$$

We write simply

$$\textit{nextState} = (\textit{nextState}_1, \cdots, \textit{nextState}_N).$$

The output function can be given similarly as

$$\textit{output} = (\textit{output}_1, \cdots, \textit{output}_K),$$

where

$$\textit{output}_i\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals},\ i = 1, \cdots, K.$$

Using these, the **state update equation** and **output equation** can be written as follows. For all $n \in \textit{Ints}$, $n \geq 0$,

$$\begin{aligned}
s_1(n+1) &= \textit{nextState}_1((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),\\
s_2(n+1) &= \textit{nextState}_2((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),\\
&\cdots\\
s_N(n+1) &= \textit{nextState}_N((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),
\end{aligned} \tag{5.6}$$

and

$$\begin{aligned}
y_1(n) &= \textit{output}_1((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),\\
y_2(n) &= \textit{output}_2((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),\\
&\cdots\\
y_K(n) &= \textit{output}_K((s_1(n), \cdots, s_N(n)), (x_1(n), \cdots, x_M(n))),
\end{aligned} \tag{5.7}$$

This system of equations shows the detailed structure of the operation of such a state machine.

So there is normally no simple relation between the step number and the real time at which the corresponding reaction occurs. For example, in the answering machine, if the initial state is *idle*, there may be an arbitrary amount of time before *ring* occurs and the state moves to *count1*.

Linear systems, however, usually evolve more smoothly in time, with a fixed time interval between updates. Suppose this interval is δ seconds. Then step n occurs at time $n\delta$ seconds, relative to time 0. Such systems are **discrete-time systems**, and the index n is called the **time index**.

The systems we consider in this chapter will be **time-invariant** systems, meaning that the *nextState* and *output* functions do not change with the time index n. For such systems, it is slightly peculiar to have time start at 0 and go to infinity. Why wouldn't time also go to minus infinity? We can easily augment the state machine model to dispense with the artifice of time starting at 0. All we do is assume that

$$\forall\, n \in \textit{Ints},\ n < 0, \quad s(n) = s(0).$$

That is, prior to time index 0, the system stutters. At time index 0, the system has already been stuttering forever. Thus, the time index $n = 0$ has the interpretation of being the time at which non-stuttering inputs begin arriving.

For linear systems, we will see that the stuttering input is a tuple of zeros, so

$$\forall\, n \in \textit{Ints},\ n < 0, \quad x(n) = 0.$$

For linear systems, we will see that $s(0) = 0$ and

$$\forall\, n \in \textit{Ints},\ n < 0, \quad y(n) = 0.$$

Thus, for **linear time-invariant (LTI)** systems given in this model, everything is zero prior to time index 0. The system is said to be **at rest**.

In previous chapters, the set of input signals to a state machine was

$$\textit{InputSignals} = [\textit{Nats}_0 \to \textit{Inputs}].$$

In this chapter, it will be

$$\boxed{\textit{InputSignals} = [\textit{Ints} \to \textit{Inputs}].}$$

We will simply assume that the inputs prior to time index 0 are all stuttering inputs. Correspondingly,

$$\boxed{\textit{OutputSignals} = [\textit{Ints} \to \textit{Outputs}].}$$

The state response, then, is a function

$$\boxed{s\colon \textit{Ints} \to \textit{States}}$$

where $s(n) = s(0)$ for all $n < 0$. With this device, the **state update equation** becomes

$$\boxed{\forall\, n \in \textit{Ints}, \quad s(n+1) = \textit{nextState}(s(n), x(n))} \tag{5.8}$$

and the **output equation** becomes,

$$\boxed{\forall\, n \in \textit{Ints}, \quad y(n) = \textit{output}(s(n), x(n)).} \tag{5.9}$$

That is, we dispense with the qualifier $n \geq 0$.

Basics: Linear functions

A function $f\colon Reals \to Reals$ is a **linear function** if $\forall\ x \in Reals$ and $w \in Reals$,

$$f(wx) = wf(x)$$

and $\forall\ x_1 \in Reals$ and $x_2 \in Reals$,

$$f(x_1 + x_2) = f(x_1) + f(x_2).$$

More compactly, f is linear if $\forall\ x_1, x_2 \in Reals$ and $w, u \in Reals$,

$$f(wx_1 + ux_2) = wf(x_1) + uf(x_2).$$

More generally, consider a function $f\colon X \to Y$, where X and Y are sets that support addition and multiplication by a real number. For example, X and Y could be tuples of reals instead of just reals. In fact, they could even be signal spaces, in which case f is the input/output function of a system. This function is linear if for all $x_1, x_2 \in X$, and $w, u \in Reals$,

$$\boxed{f(wx_1 + ux_2) = wf(x_1) + uf(x_2).} \tag{5.10}$$

The property (5.10) is called **superposition**. A function is linear if it satisfies the superposition property.

When the domain and range are both the set *Reals*, then a linear function has the form,

$$\forall\ x \in Reals, \quad f(x) = ax$$

for some constant a. The term "linear" comes from the fact that a plot of $f(x)$ vs. x is a straight line that passes through zero. If the line did not pass through zero, mathematicians would call this **affine** rather than linear. Note than an affine function would not satisfy the superposition property (we leave it as an exercise to show this).

Suppose $f\colon Reals \times Reals \to Reals$ is linear. Then it has the form

$$\forall\ s, x \in Reals, \quad f(s, x) = as + bx.$$

It is said to form a linear combination of its arguments. Similarly, a linear function of a tuple forms a linear combination of the elements of the tuple. And a linear function of a pair of tuples forms a linear combination of the elements of the tuples.

5.2 One-dimensional SISO systems

Consider a one-dimensional, single-input, single-output (SISO) system given by the following state-space model, $\forall n \in$ *Ints*,

$$\boxed{s(n+1) = as(n) + bx(n),} \tag{5.11}$$

$$\boxed{y(n) = cs(n) + dx(n),} \tag{5.12}$$

where a, b, c and d are fixed constants with values in the set *Reals*. The initial state is $s(0) =$ *initialState*. For this system, the state at a given time index is a real number, as are the input and the output. The *nextState* and *update* functions are

$$\begin{aligned} \textit{nextState}(s(n), x(n)) &= as(n) + bx(n) \\ \textit{output}(s(n), x(n)) &= cs(n) + dx(n). \end{aligned}$$

Both of these are linear functions (see box on page 132.

Let us consider an example where we construct a state-space model from another description of a system.

Example 5.1: In section 2.3.3 we considered a simple **moving average** example, where the output y is given in terms of the input x by

$$\forall\, n \in \textit{Ints}, \quad y(n) = (x(n) + x(n-1))/2. \tag{5.13}$$

This is not a state-space model. It gives the output directly in terms of the input. To construct a state-space model for it, we first need to decide what the state is. Usually, there are multiple answers, so a we face a choice. The state is a summary of the past. Examining (5.13), it is evident that we need to remember the previous input, $x(n-1)$, in order to produce an output $y(n)$ (of course, we also need the current input, $x(n)$, but that is not part of the past; that is the present). So we can define the state to be

$$\forall\, n \in \textit{Ints}, \quad s(n) = x(n-1).$$

Notice that, with this choice, the system is automatically initially at rest if we use the usual convention of assuming that the input is zero for negative time indices, $x(n) = 0,\ n < 0$. Given this choice of state, we need to choose a, b, c, and d so that (5.11) and (5.12) are equivalent to (5.13). Let us look first at (5.12), which reads

$$y(n) = cs(n) + dx(n).$$

Observing that $s(n) = x(n-1)$, can you determine c and d? From (5.13), it is obvious that $c = d = 1/2$.

Next, we determine a and b in

$$s(n+1) = as(n) + bx(n).$$

Since $s(n) = x(n-1)$, it follows that $s(n+1) = x(n)$, and this becomes

$$x(n) = ax(n-1) + bx(n),$$

from which we can see that $a = 0$ and $b = 1$.

Note that we could have chosen the state differently. For example,

$$\forall\, n \in \textit{Ints}, \quad s(n) = x(n-1)/2$$

would work fine. How would that change a, b, c, and d?

In the following example, we use a state-space model to calculate the output of a system given an input sequence.

Example 5.2: Consider a system in which the state $s(n)$ is your bank balance at the beginning of day n, and $x(n)$ is the amount you deposit or withdraw during day n. If $x(n) > 0$, it means that you are making a deposit of $x(n)$ dollars, and if $x(n) < 0$, it means that you are withdrawing $x(n)$ dollars. The output of the system at time index n is the bank balance on day n. Thus,[1]

$$\textit{States} = \textit{Inputs} = \textit{Outputs} = \textit{Reals}.$$

Suppose that the daily interest rate is r. Then your balance at the beginning of day $n+1$ is given by

$$\forall\, n \in \textit{Ints}, \quad s(n+1) = (1+r)s(n) + x(n). \tag{5.14}$$

The output of the system is your current balance,

$$\forall\, n \in \textit{Ints}, \quad y(n) = \textit{output}(s(n), x(n)) = s(n).$$

Comparing to (5.12), we see that the state update and output functions are linear, with $a = 1 + r$, $b = 1$, $c = 1$, and $d = 0$. The initial condition is *initialState*, your bank balance at the beginning of day 0. Suppose the daily interest rate is 0.01, or one percent.[2] Suppose that $\textit{initialState} = 100$, and you deposit $1,000$ dollars on day 0 and withdraw 30 every subsequent day for the next 30 days. What is your balance $s(31)$ on day 31?

You can compute $s(31)$ recursively from

$$\begin{aligned} s(0) &= 100, \\ s(1) &= 1.01s(0) + 1000, \\ &\cdots \\ s(n+1) &= 1.01s(n) - 30, \; n = 1, \cdots, 30 \end{aligned}$$

but this would be tedious. We can instead develop a formula that is a bit easier to use. We will do this for a more general problem.

[1] These sets are probably not, strictly speaking, equal to *Reals*, since deposits and withdrawals can only be a whole number of cents. Also, most likely, the bank will round your balance to the nearest cent. Thus, our model here is an approximation. Using *Reals* is a considerable simplification.

[2] This would only be reasonable in a country with hyperinflation.

Suppose we are given an input sequence $x(0), x(1), \cdots$. As in the previous example, if we repeatedly use (5.11) we obtain the first few terms of a sequence,

$$\begin{aligned}
s(0) &= \textit{initialState}, && (5.15)\\
s(1) &= as(0) + bx(0), \\
s(2) &= as(1) + bx(1) && (5.16)\\
&= a\{as(0) + bx(0)\} + bx(1) \\
&= a^2 s(0) + abx(0) + bx(1), \\
s(3) &= as(2) + bx(2) && (5.17)\\
&= a\{a^2 s(0) + abx(0) + bx(10)\} + bx(2) \\
&= a^3 s(0) + a^2 bx(0) + abx(1) + bx(2), \\
&\cdots && (5.18)
\end{aligned}$$

From this it is not difficult to guess the general pattern for the state response and the output sequence. The state response of (5.11) is given by

$$\boxed{s(n) = a^n \textit{initialState} + \sum_{m=0}^{n-1} a^{n-1-m} bx(m)} \qquad (5.19)$$

for all $n \geq 0$, and the output sequence of (5.12) is given by

$$\boxed{y(n) = ca^n \textit{initialState} + \{\sum_{m=0}^{n-1} ca^{n-1-m} bx(m) + dx(n)\}} \qquad (5.20)$$

for all $n \geq 0$. We use induction to show that these are correct. For $n = 0$, (5.19) gives $s(0) = a^0 \textit{initialState} = \textit{initialState}$ which matches (5.15), and hence is correct. [3]

Now suppose that the right-hand side of (5.19) gives the correct value of the response for some $n \geq 0$. We must show that it gives the correct value of the response for $n+1$. From (5.11) and using the hypothesis that (5.19) is the correct expression for $s(n)$, we get

$$\begin{aligned}
s(n+1) &= as(n) + bx(n) \\
&= a\{a^n \textit{initialState} + \sum_{m=0}^{n-1} a^{n-1-m} bx(m)\} + bx(n) \\
&= a^{n+1} \textit{initialState} + \sum_{m=0}^{n-1} a^{n-m} bx(m) + bx(n) \\
&= a^{n+1} \textit{initialState} + \sum_{m=0}^{n} a^{n-m} bx(m).
\end{aligned}$$

which is the expression on the right-hand side of (5.19) for $n+1$. It follows by induction that the response is indeed given by (5.19) for all $n \geq 0$. The fact that the output sequence is given by (5.20) follows immediately from (5.12) and (5.19).

[3] For any real number a, $a^0 = 1$ by definition of exponentiation.

5.2.1 Zero-state and zero-input response

The expressions (5.19) for the state response and (5.20) for the output are each the sum of two terms. The role of these two terms be better understood if we consider them in isolation.

If *initialState* $= 0$, then the first term vanishes, and only the second term is left. This second term is called the **zero-state response**. It gives the response of the system to an input sequence when the initial state is zero. For many applications, particularly when modeling a physical system, the zero-state response is what we are interested in. For a physical system, the initial state of zero reflects that the system is initially **at rest**, meaning that its initial state is zero. We will see that for a system to be linear, it must be initially at rest.

If the input sequence is zero, i.e. $0 = x(0) = x(1) = \cdots$, the second term vanishes, and only the first term is left. The first term is called the **zero-input response**. It gives the response of the system to some initial condition, with no input stimulus applied. Of course, if the system is initially at rest and the input is zero, then the state remains at zero.

So the right-hand side of both equations (5.19) and (5.20) are a sum of a zero-state response and a zero-input response. To make it clear which equation we are talking about, we use the following terminology:

zero-state state response	The state sequence $s(n)$ when the initial state is zero.
zero-input state response	The state sequence $s(n)$ when the input is zero.
zero-state output response	The output sequence $s(n)$ when the initial state is zero.
zero-input output response	The output sequence $s(n)$ when the input is zero.

Let us focus on the zero-state output response. Define the sequence

$$\boxed{h(n) = \begin{cases} 0, & \text{if } n < 0 \\ d, & \text{if } n = 0 \\ ca^{n-1}b, & \text{if } n \geq 1 \end{cases}.} \tag{5.21}$$

Then from (5.20) the zero-state output response can also be written as

$$\forall\, n \geq 0, \quad y(n) = \sum_{m=0}^{n} h(n-m)x(m). \tag{5.22}$$

Let $x(n) = 0$ for all $n < 0$, and, noting that $h(n) = 0$ for all $n < 0$, we can write this

$$\boxed{\forall\, n \in \textit{Ints}, \quad y(n) = \sum_{m=-\infty}^{\infty} h(n-m)x(m).} \tag{5.23}$$

A summation of this form is called a **convolution sum**. We say that y is the convolution of h and x, and write it using the shorthand

$$y = h * x.$$

The ‘*’ symbol represents convolution. By changing variables, defining $k = n - m$, it is easy to see that the convolution sum can also be written

$$\boxed{\forall\, n \in \textit{Ints}, \quad y(n) = \sum_{m=-\infty}^{\infty} h(k)x(n-k).} \tag{5.24}$$

That is, $h * x = x * h$. Convolution sums will be studied in much more detail in chapter 8.

Suppose the input x is given by $x = \delta$, where

$$\forall\, n \in \textit{Ints}, \quad \delta(n) = \begin{cases} 1, & \text{if } n = 0 \\ 0, & \text{if } n \neq 0 \end{cases}. \tag{5.25}$$

This function is called an **impulse**, or a **Kronecker delta function** (this function figures prominently in chapters 7 and 8). If we substitute this into the convolution sum we get

$$\forall\, n \in \textit{Ints}, \quad y(n) = h(n).$$

For this input, the output is simply given by the h function. For this reason, h is called the **impulse response** response, or more precisely, the **zero-state impulse response** of the system. Thus, if the system is initially at rest, its output is given by the convolution of the input and the impulse response.

Example 5.3: For our bank example, $a = 1 + r$, $b = 1$, $c = 1$, and $d = 0$ in (5.12). The impulse response of the bank system is given by (5.21),

$$h(n) = \begin{cases} 0, & \text{if } n \leq 0 \\ (1+r)^{n-1}, & \text{if } n \geq 1 \end{cases}.$$

This represents the balance of a bank account with daily interest rate r if an initial deposit of one dollar is put in on day 0, and no further deposits or withdrawals are made. Notice that since $1 + r > 1$, the balance continues to increase forever. In fact, such a system is called an **infinite impulse response** (**IIR**) system because the response to an impulse never completely dies out. This system is also said to be **unstable** because even though the input is always bounded, the output grows without bound.

Writing the output as a convolution, using (5.22), we see that

$$\forall\, n \geq 0, \quad y(n) = \sum_{m=0}^{n} (1+r)^{n-m-1} x(m).$$

This gives a relatively simple formula that we can use to calculate the bank balance on any given day (although it will be tedious for large n, and you will want to use a computer).

Basics: Matrices and vectors

An $M \times N$ **matrix** A is written as

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{bmatrix}.$$

The **dimension** of the matrix is said to be $M \times N$, where the number of rows is always given first, and the number of columns is given second. In general, the coefficients of the matrix are real or complex numbers, so they support all the standard arithmetic operations. We write the matrix more compactly as

$$A = [a_{i,j}, 1 \le i \le M, 1 \le j \le N]$$

or, even more simply as $A = [a_{i,j}]$ when the dimension of A is understood. The matrix entries $a_{i,j}$ are called the coefficients of the matrix.

A **vector** is a matrix with only one row or only one column. An N-dimensional **column vector** s is written as an $N \times 1$ matrix

$$s = \begin{bmatrix} s_1 \\ s_2 \\ \cdots \\ s_N \end{bmatrix}$$

An N-dimensional **row vector** z^T is written as a $1 \times N$ matrix

$$z^T = [z_1, z_2, \cdots, z_N]$$

The **transpose** of a $M \times N$ matrix $A = [a_{i,j}]$ is the $N \times M$ matrix $A^T = [a_{j,i}]$. Therefore, the transpose of an N-dimensional column vector s is the N-dimensional row vector s^T, and the transpose of a N-dimensional row vector z is the N-dimensional column vector z^T.

From now on, unless explicitly stated otherwise, all vectors denoted s, x, y, b, c etc. *without* the transpose notation are column vectors, and vectors denoted s^T, x^T, y^T, b^T, c^T *with* the transpose notation are row vectors.

A tuple of numeric values is often represented as a vector. A tuple, however, is neither a “row” nor a “column.” Thus, the representation as a vector carries the additional information that it is either a row or a column vector.

Basics: Matrix arithmetic

Two matrices (or vectors, since they are also matrices) can be added or subtracted provided that they have the same dimension. Just as with adding or subtracting tuples, the elements are added or subtracted. Thus if $A = [a_{i,j}]$ and $B = [b_{i,j}]$ and both have dimension $M \times N$, then

$$A + B = [a_{i,j} + b_{i,j}].$$

Under certain circumstances, matrices can also be multiplied. If A has dimension $M \times N$ and B has dimension $N \times P$, then the product AB is defined. The number of columns of A must match the number of rows of B. Suppose the matrices are given by

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{bmatrix} \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,P} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,P} \\ \cdots & \cdots & \cdots & \cdots \\ b_{N,1} & b_{N,2} & \cdots & b_{N,P} \end{bmatrix},$$

Then the i, j element of the product $C = AB$ is

$$c_{i,j} = \sum_{m=1}^{N} a_{i,m} b_{m,j}. \tag{5.26}$$

The product has dimension $M \times P$.

Of course, matrix multiplication also works if one of the matrices is a vector. If b is a column vector of dimension N, then $c = Ab$ as defined by (5.26) is a column vector of dimension M. If on the other hand b^T is a row vector of dimension M, then $c^T = b^T A$ as defined by (5.26) is a row vector of dimension N. By convention, we write x^T to indicate a row vector, and x to indicate a column vector. Also by convention, we (usually) use lower case variable names for vectors and upper case variable names for matrices.

Multiplying a matrix by a vector can be interpreted as applying a function to a tuple. The vector is the tuple and the matrix (together with the definition of matrix multiplication) defines the function. Thus, in introducing matrix multiplication into our systems, we are doing nothing new except introducing a more compact notation for defining a particular class of functions.

A matrix A is a **square matrix** if it has the same number of rows and columns. A square matrix may be multiplied by itself. Thus, A^n for some integer $n > 0$ is defined to be A multiplied by itself n times. A^0 is defined to be the **identity matrix**, which has ones along the diagonal and zeros everywhere else.

5.3 Multidimensional SISO systems

The previous section considered systems of the form of figure 5.1 where $M = K = N = 1$. Systems with larger dimension, $N > 1$, have much more interesting (and useful) behavior. In this section, we allow the dimension N to be larger, but keep the simplification that $M = K = 1$, so the system is still SISO.

Recall that a linear function forms a linear combination of its arguments (see box on page 132). The most convenient way to describe such functions in the multidimensional case is using matrices and vectors (see boxes on pages 138 and 139). The **state-space model** for an N-dimensional SISO system is

$$\boxed{s(n+1) = As(n) + bx(n)} \tag{5.27}$$

$$\boxed{y(n) = c^T s(n) + dx(n)} \tag{5.28}$$

This is identical to (5.11) and (5.12) except that A is an $N \times N$ matrix, and b, c are N-dimensional column vectors, so c^T is an N-dimensional row vector. As before, d is a scalar. It is conventional to write the matrix with a capital A rather than the lower-case a used in (5.11).

The *nextState* function is given by

$$\textit{nextState}(s(n), x(n)) = As(n) + bx(n).$$

The result of evaluating this function is an N-dimensional vector. The $N \times N$ matrix A defines the linear combination of the N elements of $s(n)$ that are used to calculate the N elements $s(n+1)$. The N-dimensional column vector b defines the weights used to include $x(n)$ in the linear combination for each element of $s(n+1)$.

The *output* function is

$$\textit{output}(s(n), x(n)) = c^T s(n) + dx(n).$$

The N-dimensional row vector c^T defines the linear combination of elements of $s(n)$ that used to calculate the output. The scalar d defines the weight used to include $x(n)$ in the linear combination.

Example 5.4: Above we constructed a state-space model for a length-two moving average. The general form of this is the M-point moving average, given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = \frac{1}{M} \sum_{k=0}^{M-1} x(n-k). \tag{5.29}$$

To be specific, let's suppose $M = 3$. Equation (5.29) becomes

$$\forall\, n \in \textit{Ints}, \quad y(n) = \frac{1}{3}(x(n) + x(n-1) + x(n-2)). \tag{5.30}$$

We can construct a state-space model for this in a manner similar to what we did for the length-two moving average. First, we need to decide what is the state. Recall that the state is the summary of the past. Equation (5.30) tells us that we need to remember

$x(n-1)$ and $x(n-2)$, the two past inputs. We could define these to be the state, collected as a column vector,

$$s(n) = \begin{bmatrix} x(n-1) \\ x(n-2) \end{bmatrix}.$$

(Notice that we could have equally well put the elements in the other order.)

Consider the output equation (5.28). We need to determine c^T and d. The vector c^T is a row vector with dimension $N = 2$, so we can fill in the blanks in the output equation below

$$y(n) = [\text{--}, \text{--}] \begin{bmatrix} x(n-1) \\ x(n-2) \end{bmatrix} + [\text{--}]x(n)$$

It is easy to see that each of the three blanks must be filled with $1/M = 1/3$ in order to get (5.30). Thus,

$$c = \begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}, \quad d = 1/3.$$

Consider next the output equation (5.27). We need to determine A and b. The matrix A is 2×2. The vector b is dimension 2 column vector. So we can fill in the blanks in the output equation below

$$s(n+1) = \begin{bmatrix} x(n) \\ x(n-1) \end{bmatrix} = \begin{bmatrix} \text{--} & \text{--} \\ \text{--} & \text{--} \end{bmatrix} \begin{bmatrix} x(n-1) \\ x(n-2) \end{bmatrix} + \begin{bmatrix} \text{--} \\ \text{--} \end{bmatrix} x(n).$$

From this, we can fill in the blanks, getting

$$A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The state response is given by an expression similar to (5.19), but involving matrices and vectors rather than just scalars,

$$\boxed{s(n) = A^n \textit{initialState} + \sum_{m=0}^{n-1} A^{n-1-m} b x(m)} \tag{5.31}$$

for all $n \geq 0$. The output sequence of (5.35) is given by

$$\boxed{y(n) = c^T A^n \textit{initialState} + \{\sum_{m=0}^{n-1} c^T A^{n-1-m} b x(m) + d x(n)\}} \tag{5.32}$$

for all $n \geq 0$.

The zero-state impulse response, in terms of the state-space model, is the sequence of real numbers $h(0), h(1), h(2), \cdots$

$$\boxed{h(n) = \begin{cases} 0, & \text{if } n < 0 \\ d, & \text{if } n = 0 \\ c^T A^{n-1} b, & \text{if } n \geq 1 \end{cases}} \tag{5.33}$$

This formula can be quite tedious to apply. It is usually easier to simply let $x = \delta$, the Kronecker delta function, and observe the output. The output will be the impulse response. The zero-state output response is given by convolution of this impulse response with the input, (5.24).

Example 5.5: We can find the impulse response h of the moving average system of (5.29) by letting $x = \delta$, where δ is given by (5.25). That is,

$$\forall\, n \in \textit{Ints}, \quad h(n) = \frac{1}{M}\sum_{k=0}^{M-1} \delta(n-k).$$

Now, $\delta(n-k) = 0$ except when $n = k$, at which point it equals one. Thus,

$$h(n) = \begin{cases} 0 & \text{if } n < 0 \\ 1/M & \text{if } 0 \leq n < M \\ 0 & \text{if } n \geq M \end{cases}$$

This function, therefore, is the impulse response of an M-point moving average system. This result could also have been obtained by comparing (5.29) to (5.24), the output as a convolution. Or it could have been obtained by constructing a state-space model for the general length-M moving average, and applying (5.33). However, this latter method would have proved the most tedious.

Notice that in the previous example, the impulse response is finite in extent (it starts at 0 and stops at $M-1$). For this reason, such a system is called a **finite impulse response system** or **FIR** system.

Example 5.6: The M-point moving average can be viewed as a special case of the more general FIR system given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = \sum_{k=0}^{M-1} h(k)x(n-k).$$

Letting $h(k) = 1/M$ for $0 \leq k < M$, we get the M-point moving average. Choosing other values for $h(k)$, however, we can get other responses (this will be explored in chapter 7).

A state-space model for the FIR system is constructed by again deciding on the state. A reasonable choice is the $M-1$ past samples of the input,

$$s(n) = [x(n-1), x(n-2), \cdots, x(n-M+1)]^T,$$

a column vector. The state-space model is then given by (5.27) and (5.28) with

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 0 \\ \cdots \\ 0 \\ 0 \end{bmatrix}, \; c = \begin{bmatrix} h(1) \\ h(2) \\ \cdots \\ h(M-2) \\ h(M-1) \end{bmatrix},$$

$$d = h(0)$$

Notice that the model that we found in example 5.4 has this form. The $(M-1)\times(M-1)$ matrix A has coefficients $a_{i+1,i} = 1$, while all other coefficients are zero. This is a rather special form of the A matrix, limited to FIR systems. The vector b has the first coefficient equal to 1, while all others are zero.

5.4 Multidimensional MIMO systems

In the above sections, the input and output were both scalars. A **multiple-input, multiple-output** (**MIMO**) system is only slightly more complicated. A state-space model for such a system is

$$\forall\, n \in \textit{Ints}, \quad s(n+1) = As(n) + Bx(n) \tag{5.34}$$
$$y(n) = Cs(n) + Dx(n) \tag{5.35}$$

where, $s(n) \in \textit{Reals}^N$, $x(n) \in \textit{Reals}^M$ and $y(n) \in \textit{Reals}^K$, for any integer n. A is an $N \times N$ (square) matrix, B is an $N \times M$ matrix, C is a $K \times N$ matrix, and D is a $K \times M$ matrix. Now that these are all matrices, it is conventional to write them with capital letters. Each matrix has fixed (given) coefficients, and the set of four matrices define the system.

Let *initialState* $\in \textit{Reals}^N$ be a given initial state. Let $x(0), x(1), x(2), \cdots$, be a sequence of inputs in *Reals*^{M} (each input is a vector of dimension M). The state response of (5.34) is given by

$$s(n) = A^n \textit{initialState} + \sum_{m=0}^{n-1} A^{n-1-m} Bx(m) \tag{5.36}$$

for all $n \geq 0$, and the output sequence of (5.35) is given by

$$y(n) = CA^n \textit{initialState} + \{\sum_{m=0}^{n-1} CA^{n-1-m} Bx(m) + Dx(n)\} \tag{5.37}$$

for all $n \geq 0$.

As before, the right-hand side of these equations are the sum of a **zero-input response** and a **zero-state response**. Consider the zero-state output response,

$$y(n) = \sum_{m=0}^{n-1} CA^{n-1-m} Bx(m) \tag{5.38}$$

for all $n \geq 0$. Define the sequence $h(0), h(1), h(2), \cdots$ of $K \times M$ matrices by

$$h(n) = \begin{cases} D, & \text{if } n = 0 \\ CA^{n-1}B, & \text{if } n \geq 1 \end{cases} \tag{5.39}$$

(Notice that this can no longer be called an impulse response, because this system cannot be given a simple impulse as an input. The input has to have dimension M). From (5.38) it follows that the zero-state output response is given by

$$y(n) = \sum_{m=0}^{n} h(n-m)x(m), \; n \geq 0 \tag{5.40}$$

This is once again a **convolution sum**.

5.5 Linear systems

In the systems that this chapter considers, the *nextState* and *output* functions are linear. Recall (see box on page 132) that a function $f\colon X \to Y$ is linear if (and only if) it satisfies the superposition property, i.e. for all $x_1, x_2 \in X$, and $w, u \in$ *Reals*,

$$\boxed{f(wx_1 + ux_2) = wf(x_1) + uf(x_2).}$$

What does it mean for a *system* to be linear? Recall that a system S is a function $S\colon X \to X$, where X is a signal space. For the systems in this chapter, $X = [\textit{Ints} \to \textit{Reals}]$. The function S is defined by (5.37), which gives $y = S(x)$, given x. So the answer is obvious. S is a **linear system** if S is a linear function.

Examining (5.37), or its simpler SISO versions, (5.20) or (5.32), it is easy to see that superposition is satisfied if *initialState* is zero. Hence, a system given by a state-space model that is initially at rest is a linear system. The superposition property turns out to be an extremely useful property, as we will discover in the next chapters.

5.6 Continuous-time state-space models

A continuous-time state-space model for a linear SISO system has the form

$$\dot{z}(t) = Az(t) + bv(t) \tag{5.41}$$

$$w(t) = cz(t) + dv(t) \tag{5.42}$$

where

- $z\colon \textit{Reals} \to \textit{Reals}^N$ gives the state response;
- $\dot{z}(t)$ is the derivative of z evaluated at $t \in \textit{Reals}_+$;
- $v\colon \textit{Reals} \to \textit{Reals}$ is the input signal; and
- $w\colon \textit{Reals} \to \textit{Reals}$ is the output signal.

As with the discrete-time SISO model, A is an $N \times N$ matrix, b and c are $N \times 1$ column vectors, and d is a scalar.

The major difference between this model and that of (5.27) and (5.28) is that instead of giving new state as a function of the input and the old state, (5.41) gives the derivative of the state. The derivative of a vector z is simply the vector consisting of the derivative of each element of the vector. A derivative, of course, gives the trend of the state at any particular time. Giving a trend makes more sense than giving a new state for a continuous-time system, because the state evolves continuously.

Probing further: Approximating continuous-time systems

Linear systems often arise as approximations of continuous-time systems that are described by **differential equations**. Those differential equations may describe the physics. A differential equation is of the form

$$\forall\, t \in \textit{Reals}, \quad \dot{z}(t) = g(z(t), v(t)). \tag{5.43}$$

Here $t \in \textit{Reals}$ stands for continuous time, $z : \textit{Reals} \to \textit{Reals}^N$ is the state response, and $v : \textit{Reals} \to \textit{Reals}^M$ is the input signal. That is, at any time t, $z(t)$ is the state and $v(t)$ is the input. The notation $\dot{z}$ stands for derivative of the state response z with respect to t, so $g\colon \textit{Reals}^N \times \textit{Reals}^M \to \textit{Reals}^N$ is a given function specifying the derivative of the state. Specifying the derivative of the state is similar to specifying a state update. Recall that a derivative is a normalized difference over an infinitesimally small interval. A continuous-time system can be thought of as one where the state updates occurs in intervals that are so small that the state evolves continuously rather than discretely.

In general, z is an N-tuple, $z = (z_1, \cdots, z_N)$, where $z_i\colon \textit{Reals}_+ \to \textit{Reals}$. The derivative of an N-tuple is simply the N-tuple of derivatives, $\dot{z} = (\dot{z}_1, \cdots, \dot{z}_N)$. We know from calculus that

$$\dot{z}(t) = \frac{dz}{dt} = \lim_{\delta \to 0} \frac{z(t+\delta) - z(t)}{\delta},$$

and so, if $\delta > 0$ is a small number, we can approximate this derivative by

$$\dot{z}(t) \approx \frac{z(t+\delta) - z(t)}{\delta}.$$

Using this for the derivative in the left-hand side of (5.43) we get

$$z(t+\delta) - z(t) = \delta g(z(t), v(t)). \tag{5.44}$$

Suppose we look at this equation at sample times $t = 0, \delta, 2\delta, \cdots$. Let us denote the value of the state response at the n-th sample time by $s(n) = z(n\delta)$, and the value of the input at this same sample time by $x(n) = v(n\delta)$. In terms of these variables, (5.44) becomes

$$s(n+1) - s(n) = \delta g(s(n), x(n))$$

which we can write as a state update equation,

$$s(n+1) = s(n) + \delta g(s(n), x(n)).$$

All of the methods that we have developed for discrete-time systems can also be developed for continuous-time systems. However, they get somewhat more challenging mathematically because the summations become integrals. We leave it to a more advanced text to explore this.

A continuous-time state-space model may be approximated by a discrete-time state-space model (see box). In fact, this approximation forms the basis for most computer simulations of continuous-time systems. Simulation of continuous-time systems is explored in labC.6.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires conceptualization. Many of the problems require using Matlab.

1. **E** Construct a SISO state-space model for a system whose input and output are related by

$$\forall\, n \in \mathit{Ints}, \quad y(n) = x(n-1) + x(n-2).$$

You may assume the system is initially at rest. It is sufficient to give the A matrix, vectors b and c, and scalar d of (5.27) and (5.28).

2. **E** The A matrix in (5.27) for a SISO system is

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Calculate the zero-input state response if

(a) the initial state is $[1, 0]^T$,
(b) the initial state is $[0, 1]^T$,
(c) the initial state is $[1, 1]^T$.

3. **E** Consider the one-dimensional state-space model, $\forall\, n \in$ *Ints*,

$$\begin{aligned} s(n+1) &= s(n) + x(n) \\ y(n) &= s(n) \end{aligned}$$

Suppose the initial state is $x(0) = a$ for some given constant a. Find another constant b such that if the first three inputs are $x(0) = x(1) = x(2) = b$, then y(3) = 0. Note: In general, problems of this type are concerned with **controllability**. The question is whether you can find an input (in this case constrained to be constant) such that some particular condition on the output is met. The input becomes a **control signal**.

4. **E** A SISO LTI system has the A matrix given by

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

and the b vector by $[0, 1]^T$. Suppose that $s(0) = [0, 0]^T$. Find the input sequence $x(0), x(1)$ so that the state at step 2 is $s(2) = [1, 2]^T$.

5. **E** Suppose the A matrix of a two-dimensional SISO system is

$$A = \sigma \begin{bmatrix} \cos(\pi/6) & \sin(\pi/6) \\ -\sin(\pi/6) & \cos(\pi/6) \end{bmatrix}.$$

   Suppose the initial state is $s(0) = [1, 0]^T$, and the input is zero. Sketch the zero-input state response for $n = 0, 1, \cdots, 12$ for the cases

   (a) $\sigma = 0$
   (b) $\sigma = 0.9$
   (c) $\sigma = 1.1$

6. **E** In this problem we consider the bank balance example further. As in the example, suppose *initialState* $= 100$, and you deposit $1,000$ dollars on day 0 and withdraw 30 dollars every subsequent day for the next 30 days.

   (a) Write a Matlab program to compute your bank balance $s(n), 0 \leq n \leq 31$, and plot the result.
   (b) Use formula (5.19) to calculate your bank balance at the beginning of day 31. The following identity may prove useful:

$$\sum_{m=0}^{N} a^m = \frac{1 - a^{N+1}}{1 - a}.$$

7. **E** Use Matlab to calculate and plot the impulse response of the system

$$\begin{aligned} s(n+1) &= as(n) + bx(n) \\ y(n) &= cs(n) \end{aligned}$$

   for the following cases:

   (a) $a = 1.1$
   (b) $a = 1.0$
   (c) $a = 0.9$
   (d) $a = -0.5$

   where in all cases, $b = c = 1$ and $d = 0$.

8. **E** Consider the 2-dimensional system with

$$A = \begin{bmatrix} \cos(\omega) & -\sin(\omega) \\ \sin(\omega) & \cos(\omega) \end{bmatrix}, \ b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \ c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, d = 0,$$

   (a) Show that for all $n = 0, 1, 2, \cdots$

$$A^n = \begin{bmatrix} \cos(n\omega) & -\sin(n\omega) \\ \sin(n\omega) & \cos(n\omega) \end{bmatrix},$$

Hint: use induction and the identities

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

(b) Find the zero-input state response and the zero-input response for the initial state

$$\textit{initialState} = [0,1]^T.$$

Hint: This system is called an **oscillator** because it oscillates without external stimulus.

(c) Find the zero-state impulse response.

9. **E** A SISO state-space model is expressed in the form

$$\begin{aligned} s(n+1) &= As(n) + bx(n) \\ y(n) &= c^T s(n) + dx(n) \end{aligned}$$

where A is a $N \times N$ matrix, b, c and $s(n)$ are $N \times 1$ column vectors, and $d, x(n), y(n)$ are scalars. Assume the input is x is the output is y.

(a) Write down the expression for the zero-state impulse response, i.e. what is the output when the initial state $x(0) = 0$, and $u(n) = \delta(n)$, the Kronecker delta function.

(b) Suppose

$$A = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad c^T = [0 \quad 1] \quad d = 1$$

Find the zero-state impulse response.

Hint: for all $n \geq 0$,

$$A^n = \begin{bmatrix} \cos(n\theta) & \sin(n\theta) \\ -\sin(n\theta) & \cos(n\theta) \end{bmatrix},$$

Chapter 6

Frequency Domain

We are generally interested in manipulating signals. We may wish to synthesize signals, as modems need to do in order to transmit a voice-like signal through the telephone channel. We may instead wish to analyze signals, as modems need to do in order to extract digital information from a received voice-like signal. In general, the field of **communications** is all about synthesizing signals with characteristics that match a channel, and then analyzing signals that have often been corrupted by the channel in order to extract the original information.

We may also wish to synthesize natural signals such as images or speech. The field of **computer graphics** puts much of its energy into synthesizing natural-looking images. **Image processing** includes **image understanding**, which involves analyzing images to determine their content. The field of **signal processing** includes analysis and synthesis of speech and music signals.

We may wish to control a physical process. The physical process is sensed (using temperature, pressure, position and speed sensors). The sensed signals are processed in order to estimate the internal state of the physical process. The physical process is controlled on the basis of the state estimate. Control system design includes the design of **state estimators** and **controllers**.

In order to analyze or synthesize signals, we need models of those signals. Since a signal is a function, a model of the signal is a description or a definition of the function. We will use two approaches. The first is a **declarative** (what is) approach. The second is an **imperative** (how to) approach. These two approaches are complementary. Depending on the situation, one approach is better than the other.

Signals are functions. This chapter in particular deals with signals where the domain is time (discrete or continuous). It introduces the concept of frequency-domain representation of these signals. The idea is that arbitrary signals can be described as sums of sinusoidal signals. This concept is first motivated by referring to psychoacoustics, how humans hear sounds. Sinusoidal signals have particular psychoacoustic significance. But the real justification for the frequency domain approach is much broader. It turns out to be particularly easy to understand the affect that LTI systems, discussed in the previous chapter, have on sinusoidal signals. A powerful set of analysis and design techniques then follow for arbitrary signals and the LTI systems that operate on them.

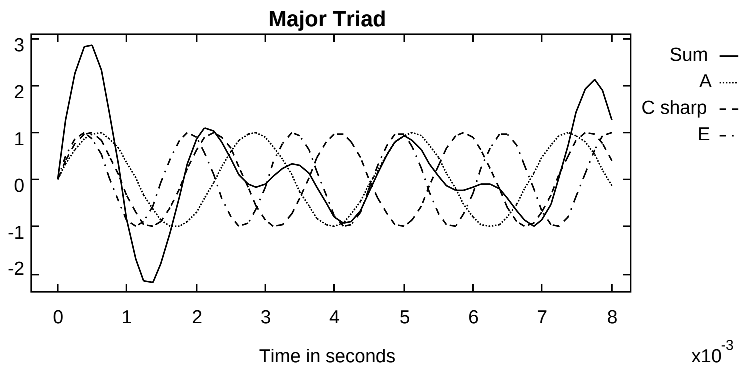


Figure 6.1: Graph of a major triad, showing its three sinusoidal components and their sum.

6.1 Frequency decomposition

For some signals, particularly natural signals like voice, music, and images, finding a concise and precise definition of the signal can be difficult. In such cases, we try to model signals as compositions of simpler signals that we can more easily model.

Psychoacoustics is the study of how humans hear sounds. Pure tones and their frequency turn out to be a very convenient way to describe sounds. Musical notes can be reasonably accurately modeled as combinations of relatively few pure tones (although subtle properties of musical sounds, such as the timbre of a sound, are harder to model accurately).

When studying sounds, it is reasonable on psychoacoustic grounds to decompose the sounds into sums of sinusoids. It turns out that the motivation for doing this extends well beyond psychoacoustics. Pure tones have very convenient mathematical properties that make it useful to model other types of signals as sums of sinusoids, even when there is no psychoacoustic basis for doing so. For example, there is no psychoacoustic reason for modeling radio signals as sums of sinusoids.

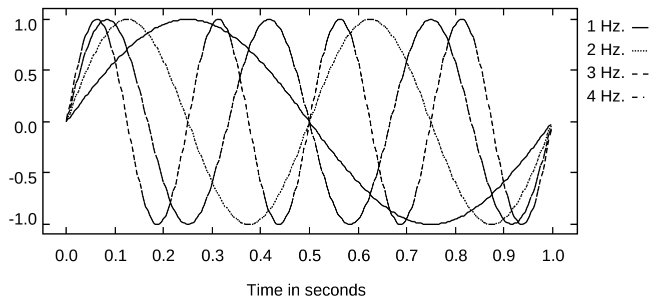
Consider the range of frequencies covering one **octave**, ranging from 440Hz to 880 Hz. “Octave” is the musical term for a factor of two in frequency. The frequencies 440 Hz and 880 Hz both correspond to the musical note A, but one octave apart. The next higher A in the musical scale would have the frequency 1760 Hz, twice 880 Hz. In the western musical scale, there are 12 notes in every octave. These notes are evenly distributed (geometrically), so the next note above A, which is B flat, has frequency $440 \times \sqrt[12]{2}$, where $\sqrt[12]{2} \approx 1.0595$. The next note above B flat, which is B, has frequency $440 \times \sqrt[12]{2} \times \sqrt[12]{2}$.

In table 6.1, the frequencies of the complete musical scale between middle A and A-880 are shown. Each frequency is $\beta = \sqrt[12]{2}$ times the frequency below it.

Frequencies that are harmonically related tend to sound good together. Figure6.1 shows the graph of a signal that is a major triad, a combination of the notes A, C♯ (C sharp), and E. By “combination” we mean “sum.” The A is a sinusoidal signal at 440 Hz. It is added to a C♯, which is a sinusoidal

Basics: Frequencies in Hertz and radians

A standard measure of frequency is **Hertz**, abbreviated **Hz**. It means **cycles per second**. Below is a plot of one second of four sine waves of different frequencies:



For example, the frequencies in Hertz of the musical note A on the piano keyboard are

$$f_1 = 55, f_2 = 110, f_3 = 220, f_4 = 440,$$
$$f_5 = 880, f_6 = 1760, f_7 = 3520, f_8 = 7040.$$

A sinusoidal waveform x with frequency $f_4 = 440$ can be defined by

$$\forall\, t \in \textit{Reals}, \quad x(t) = sin(440 \times 2\pi t).$$

The factor 2π in this expressions is a nuisance. An argument to a sine function has units of **radians**, so 2π has units of radians/cycle. Explicitly showing all the units (in square brackets), we have

$$440[\text{cycles/second}] \times 2\pi[\text{radians/cycle}]t[\text{seconds}] = (440 \times 2\pi t)[\text{radians}].$$

To avoid having to keep track of the factor 2π everywhere, it is common to use the alternative units for frequency, **radians per second**. The symbol ω is commonly used to denote frequencies in radians per second, while f is used for frequencies in Hertz. The relationship between Hertz and radians per second is simple,

$$\omega = 2\pi f,$$

as is easily confirmed by checking the units. Thus, in radians per second, the frequencies of the musical note A on the piano keyboard are

$$\omega_1 = 2\pi \times 55, \omega_2 = 2\pi \times 110, \omega_3 = 2\pi \times 220, \omega_4 = 2\pi \times 440,$$
$$\omega_5 = 2\pi \times 880, \omega_6 = 2\pi \times 1760, \omega_7 = 2\pi \times 3520, \omega_8 = 2\pi \times 7040.$$

Basics: Ranges of frequencies

An extremely wide range of frequencies is used by electrical engineers. The following abbreviations are common:

- Hz - hertz, cycles per second.
- kHz - kilohertz, thousands of cycles per second.
- MHz - megahertz, millions of cycles per second.
- GHz - gigahertz, billions of cycles per second.

Audible sounds signals are in the range of 20 Hz to 20 kHz. Sounds above this frequency are called ultrasonic. Electromagnetic waves range from less than one hertz (used speculatively in seismology for earthquake prediction) through visible light near 10^{15} Hz to cosmic ray radiation up to 10^{25} Hz.

A	880
A♭	831
G	784
F♯	740
F	698
E	659
D♯	622
D	587
C♯	554
C	523
B	494
B♭	466
A	440

Table 6.1: Frequencies of notes over one octave of the western musical scale.

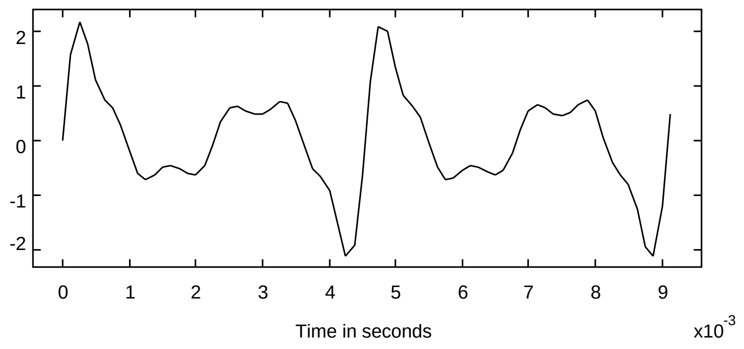


Figure 6.2: A sound waveform for an A-440 with more interesting timbre.

signal at 554 Hz. This sum is then added to an E, which is a sinusoidal signal at 659 Hz. Each of the components is also shown, so you can verify graphically that at each point in time, the value of the solid signal is equal to the sum of values of the dashed signals at that time.

The stimulus presented to the ear is the solid waveform. What you hear, however, assuming a small amount of musical training, is the three sinusoidal components, which the human ear interprets as musical notes. The human ear decomposes the stimulus into its sinusoidal components.

Example 6.1: The major triad signal can be written as a sum of sinusoids

$$s(t) = \sin(440 \times 2\pi t) + \sin(554 \times 2\pi t) + \sin(659 \times 2\pi t),$$

for all $t \in$ *Reals*. The human ear hears as distinct tones the frequencies of these sinusoidal components. Musical sounds such as chords can be characterized as sums of pure tones.

Purely sinusoidal signals, however, do not sound very good. Although they are recognizable as notes, they do not sound like any familiar musical instrument. Truly musical sounds are much more complex than a pure sinusoid. The characteristic sound of an instrument is its **timbre**, and as we shall see, some aspects of timbre can also be characterized as sums of sinusoids.

Timbre is due in part to the fact that musical instruments do not produce purely sinusoidal sounds. Instead, to a first approximation, they produce sounds that consist of a fundamental sinusoidal component and harmonics. The fundamental is at the frequency of the note being played, and the harmonics are at multiples of that frequency. Figure 6.2 shows a waveform for a sound that is heard as an A-220 but has a much more interesting timbre than a sinusoidal signal with frequency 220 Hz. In fact, this waveform is generating by adding together sinusoidal signals with frequencies 220 Hz, 440 Hz, 880 Hz, 1320 Hz, and higher multiples, with varying weights. The 220 Hz component is called the **fundamental**, while the others are called **harmonics**. The relative weights of the harmonics is a major part of what makes one musical instrument sound different from another.

Probing further: Circle of fifths

The western musical scale is based on our perception of frequency and the harmonic relationships between frequencies. The following frequencies all correspond to the note A:

110, 220, 440, 880, 1760, and 3520.

What about $440 \times 3 = 1320$? Notice that $1320/2 = 660$, which is almost exactly the E in table 6.1. Thus, 440×3 is the note E, one octave above the E above A-440. E and A are closely harmonically related, and to most people, they sound good together. It is because

$$440 \times 3 \approx 659 \times 2$$

The notes A, C♯, and E form a major triad. Where does the C♯ come from? Notice that

$$440 \times 5 \approx 554 \times 4.$$

Among all the harmonic relationships in the scale, A, C♯, and E have one of the simplest. This is the reason for their pleasing sound together.

For more arcane reasons, the interval between A and E, which is a frequency rise of approximately 3/2, is called a **fifth**. The note 3/2 (a fifth) above E has frequency 988, which is one octave above B-494. Another 3/2 above that is approximately F sharp (740 Hz). Continuing in this fashion, multiplying frequencies by 3/2, and then possibly dividing by two, you can approximately trace the twelve notes of the scale. This progression is called the **circle of fifths**. The notion of **key** and **scale** in music are based on this circle of fifths, as is the fact that there are 12 notes in the scale.

Table 6.1 is calculated by multiplying each frequency by $\sqrt[12]{2}$ to get the next higher frequency, not by using the circle of fifths. Indeed, the $\sqrt[12]{2}$ method applied twelve times yields a note that is *exactly* one octave higher than the starting point, while the circle of fifths only yields an approximation. The $\sqrt[12]{2}$ method yields the **well-tempered scale**. This scale was popularized by the composer J. S. Bach. It sounds much better than a scale based on the circle fifths when the range of notes spans more than one octave.

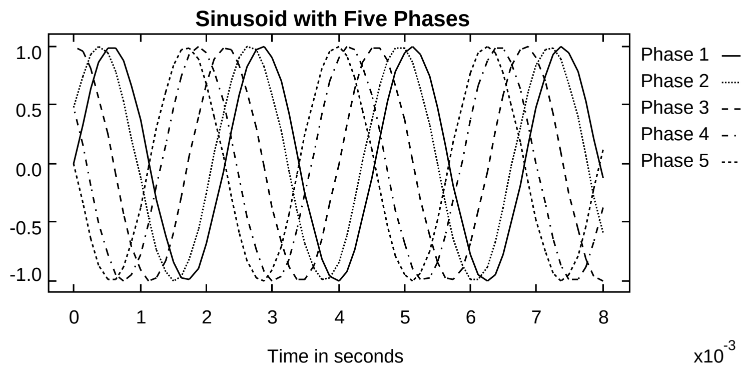


Figure 6.3: Five sinusoidal signals with different phases.

6.2 Phase

A sinusoidal sound has not just a frequency, but also a **phase**. The phase may be thought of as the relative starting point of the waveform. Figure 6.3 shows five sinusoidal signals with the same frequency but five different phases. These signals all represent the sound A-440, and all sound identical. For a simple sinusoidal signal, obviously, phase has no bearing on what the human ear hears.

Somewhat more surprising is that when two or more sinusoids are added together, the relative phase has a significant impact on the shape of the waveform, but no impact on the perceived sound. The human ear is relatively insensitive to the phase of sinusoidal components of a signal, even though the phase of those components can strongly affect the shape. If these waveforms represent something other than sound, like stock prices for example, the effect of phase could be quite significant. For a sinusoidal signal, the phase affects whether a particular point in time corresponds to a peak or a valley, for example. For stock prices, it makes a difference whether you sell at a high or a low.

There are certain circumstances in which the human ear is sensitive to phase. In particular, when two sinusoids of the same frequency combine, the relative phase has a big impact, since it affects the amplitude of the sum. For example, if the two sinusoids differ in phase by 180 degrees (π radians), then when they add, they exactly cancel, yielding a zero signal. The human brain can use the relative phase of a sound in the two ears to help spatially locate the origin of a sound. Also, audio systems with two speakers, which simulate spatially distributed sounds (“stereo”), can be significantly affected by the relative phase of the signal produced by the two speakers.

Phase is measured in either radians or degrees. An A-440 can be given by

$$g(t) = \sin(440 \times 2\pi t + \phi),$$

for all $t \in$ *Reals*, where $\phi \in$ *Reals* is the phase. Regardless of the value of ϕ, this signal is still an A-440. If $\phi = \pi/2$ then

$$g(t) = \cos(440 \times 2\pi t).$$

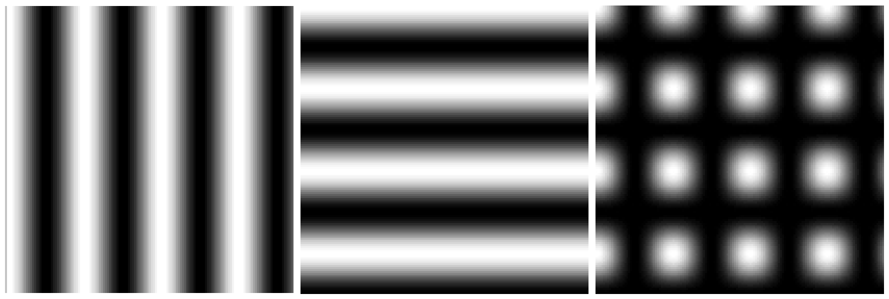


Figure 6.4: Images that are sinusoidal horizontally, vertically, and both.

6.3 Spatial frequency

Psychoacoustics provides a compelling motivation for decomposing audio signals as sums of sinusoids. In principal, images can also be similarly decomposed. However, the motivations in this case are more mathematical than perceptual.

Figure 6.4 shows three images that are sinusoidal. Specifically, the **intensity** of the image (the amount of white light that is reflected by the page) varies spatially according to a sinusoidal function. In the leftmost image, it varies horizontally only. There is no vertical variation in intensity. In the middle image, it varies vertically only. In the rightmost image, it varies in both dimensions.

The sinusoidal image has **spatial frequency** rather than temporal frequency. Its units are cycles per unit distance. The images in figure 6.4 have frequencies of roughly 2.5 cycles/inch. Recall that a grayscale picture is represented by a function

$$\mathit{Image} : \mathit{VerticalSpace} \times \mathit{HorizontalSpace} \to \mathit{Intensity}.$$

So an image that varies sinusoidally along the horizontal direction (with a spatial period of H inches) and is constant along the vertical direction is represented by

$$\forall x \in \mathit{VerticalSpace}\ \forall y \in \mathit{HorizontalSpace} \quad \mathit{Image}(x, y) = \sin(2\pi y/H).$$

Similarly, an image that varies sinusoidally along the vertical direction (with a spatial period of V inches) and is constant along the horizontal direction is represented by

$$\forall x \in \mathit{VerticalSpace}\ \forall y \in \mathit{HorizontalSpace} \quad \mathit{Image}(x, y) = \sin(2\pi x/V).$$

An image that varies sinusoidally along both directions is represented by

$$\forall x \in \mathit{VerticalSpace}\ \forall y \in \mathit{HorizontalSpace} \quad \mathit{Image}(x, y) = \sin(2\pi x/V) \times \sin(2\pi y/H).$$

These sinusoidal images have much less meaning than audio sinusoids, which we perceive as musical tones. Nonetheless, we will see that images can be described as sums of sinusoids, and that such description is sometimes useful.

6.4 Periodic and finite signals

When the domain is continuous or discrete time, we can define a **periodic signal**. Assuming the domain is *Reals*, a periodic signal x with period $p \in$ *Reals* is one where for all $t \in$ *Reals*

$$\boxed{x(t) = x(t+p).} \tag{6.1}$$

A signal with period p also has period $2p$, since

$$x(t) = x(t+p) = x(t+2p).$$

In fact, it has period Kp, for any positive integer K. Usually, we define the period to be the *smallest* p such that

$$\forall\, t \in \textit{Reals}, \quad x(t) = x(t+p).$$

Example 6.2: The sinusoidal signal x where for all $t \in$ *Reals*

$$x(t) = \sin(\omega_0 t)$$

is a periodic signal with period $2\pi/\omega_0$ since for all $t \in$ *Reals*

$$\sin(\omega_0(t + 2\pi/\omega_0)) = \sin(\omega_0 t).$$

A periodic signal is defined over an infinite interval. If the domain is instead a subset $[a,b] \subset$ *Reals*, for some finite a and b, then we call this a **finite signal**.

Example 6.3: The signal y where for all $t \in [0, 2\pi/\omega_0]$,

$$y(t) = \sin(\omega_0 t)$$

is a finite signal with duration $2\pi/\omega_0$. This interval spans exactly one cycle of the sine wave.

A finite signal with duration p can be used to define a periodic signal with period p. All that is needed is to periodically repeat the finite signal. Formally, given a finite signal $y\colon [a,b] \to$ *Reals*, we can define a signal $y'\colon$ *Reals* $\to$ *Reals* by

$$\forall\, t \in \textit{Reals}, \quad y'(t) = \begin{cases} y(t) & \text{if } t \in [a,b] \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

In other words, $y'(t)$ is simply $y(t)$ inside its domain, and zero elsewhere. Then the periodic signal can be given by

$$\boxed{x(t) = \sum_{m=-\infty}^{\infty} y'(t - mp)} \tag{6.3}$$

where $p = b - a$. This is called a **shift-and-add summation**, illustrated in figure 6.5. The periodic signal is a sum of versions of $y'(t)$ that have been shifted in time by multiples of p. Do not let the infinite sum intimidate: all but one of the terms of the summation are zero for any fixed t! Thus, a periodic signal can be defined in terms of a finite signal, which represents one period. Conversely, a finite signal can be defined in terms of a periodic signal (by taking one period).

We can check that x given by (6.3) is indeed periodic with period p,

$$\begin{aligned} x(t+p) &= \sum_{m=-\infty}^{\infty} y'(t+p-mp) = \sum_{m=-\infty}^{\infty} y'(t-(m-1)p) \\ &= \sum_{k=-\infty}^{\infty} y'(t-kp) = x(t), \end{aligned}$$

by using a change of variables, $k = m - 1$.

It is also important to note that the periodic signal x agrees with y in the finite domain $[a, b]$ of y, since

$$\begin{aligned} \forall t \in [a,b] \quad x(t) &= \sum_{m=-\infty}^{\infty} y'(t-mp) \\ &= y'(t) + \sum_{m \neq 0} y'(t-mp) \\ &= y(t), \end{aligned}$$

because, by (6.2), for $t \in [a, b]$, $y'(t) = y(t)$ and $y'(t - mp) = 0$ if $m \neq 0$.

We will see that any periodic signal, and hence any finite signal, can be described as a sum of sinusoidal signals. This result, known as the Fourier series, is one of the fundamental tools in electrical engineering.

6.5 Fourier series

A remarkable result, due to Joseph Fourier, 1768-1830, is that a periodic signal $x\colon \textit{Reals} \to \textit{Reals}$ with period $p \in \textit{Reals}$ can (usually) be described as a constant term plus a sum of sinusoids,

$$\boxed{x(t) = A_0 + \sum_{k=1}^{\infty} A_k \cos(k\omega_0 t + \phi_k)} \tag{6.4}$$

This representation of x is called its **Fourier series**. The Fourier series is widely used for signal analysis. Each term in the summation is a cosine with amplitude A_k and phase ϕ_k. The particular values of A_k and ϕ_k depend on x, of course. The frequency ω_0, which has units of radians per second (assuming the domain of x is in seconds), is called the **fundamental frequency**, and is related to the period p by

$$\boxed{\omega_0 = 2\pi/p.}$$

In other words, a signal with fundamental frequency ω_0 has period $p = 2\pi/\omega$. The constant term A_0 is sometimes called the **DC** term, where "DC" stands for **direct current**, a reference back to

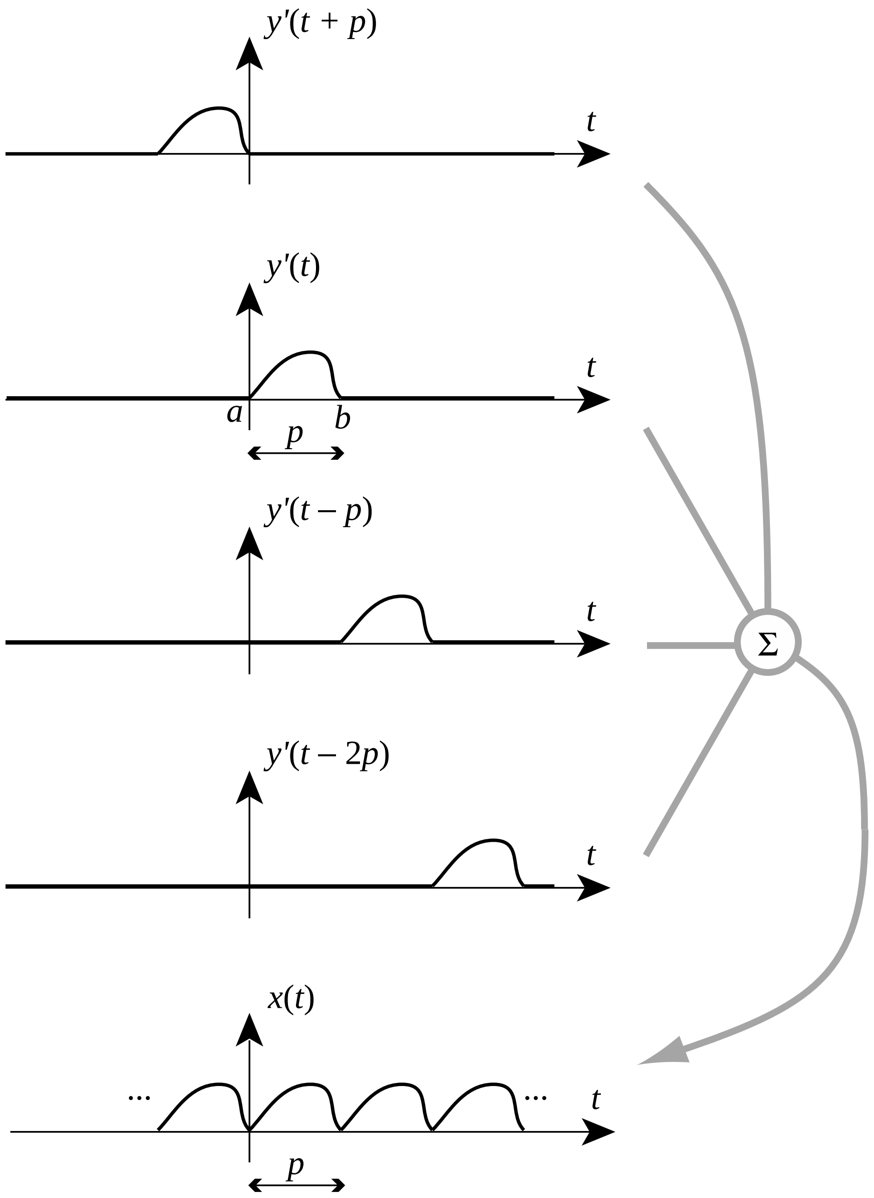


Figure 6.5: By repeating the finite signal y we can obtain a periodic signal x.

the early applications of this theory in electrical circuit analysis. The terms where $k \geq 2$ are called **harmonics**.

Equation (6.4) is often called the **Fourier series expansion** for x because it expands x in terms of its sinusoidal components.

If we had a facility for generating individual sinusoids, we could use the Fourier series representation (6.4) to synthesize any periodic signal. However, using the Fourier series expansion for synthesis of periodic signals is problematic because of the infinite summation. But for most practical signals, the coefficients A_k become very small (or even zero) for large k, so a finite summation can be used as an approximation. A **finite Fourier series approximation** with $K + 1$ terms has the form

$$\boxed{\tilde{x}(t) = A_0 + \sum_{k=1}^{K} A_k \cos(k\omega_0 t + \phi_k).} \tag{6.5}$$

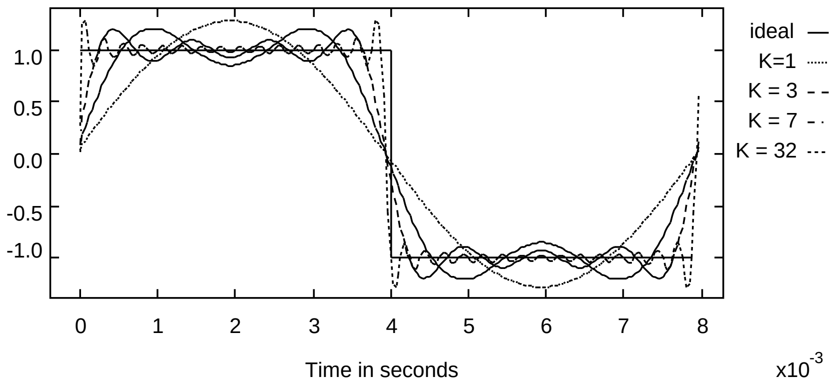
Even when an infinite summation is used, the expansion of a periodic waveform is not always exact. There are some technical mathematical conditions that x must satisfy for it to be exact (see box). Fortunately, these conditions are met almost always by practical, real-world time-domain signals.

Example 6.4: Figure 6.6 shows a square wave with period 8 msec and some finite Fourier series approximations to the square wave. Only one period of the square wave is shown. Notice in figure 6.6(a) that the $K = 1$ approximation consists only of the DC term (which is zero in this case) and a sinusoid with an amplitude slightly larger than that of the square wave. Its amplitude is depicted in figure 6.6(b) as the height of the largest bar. The horizontal position of the bar corresponds to the frequency of the sinusoid, 125 Hz, which is 1/(8 msec), the fundamental frequency. The $K = 3$ waveform is the sum of the $K = 1$ waveform and one additional sinusoid with frequency 375 Hz and amplitude equal to the height of the second largest bar in figure 6.6(b).

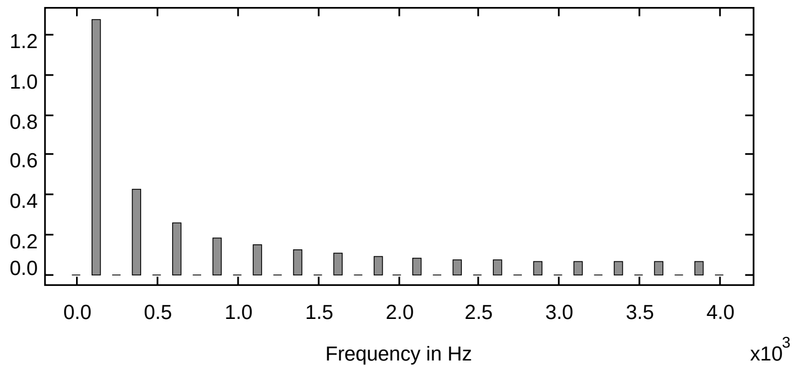
A plot like that in figure 6.6(b) is called a **frequency domain** representation of the square wave, because it depicts the square wave by the amplitude and frequency of its sinusoidal components. Actually, a complete frequency domain representation also needs to give the phase of each sinusoidal component.

Notice in figure 6.6(b) that all even terms of the Fourier series approximation have zero amplitude. Thus, for example, there is no component at 250 Hz. This is a consequence of the symmetry of the square wave, although it is beyond the scope of work now to explain exactly why.

Also notice that as the number of terms in the summation increases, the approximation more closely resembles a square wave, but the amount of its overshoot does not appear to decrease. This is known as **Gibb's phenomenon**. In fact, the maximum difference between the finite Fourier series approximation and the square wave does not converge to zero as the number of terms in the summation increases. In this sense, the square wave cannot be exactly described with a Fourier series (see box on page 164). Intuitively, the problem is due to the abrupt discontinuity in the square wave when it transitions between its high value and its low value.

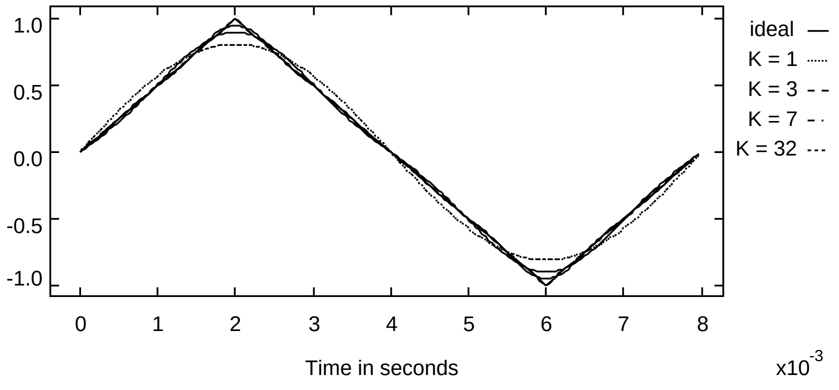


(a)

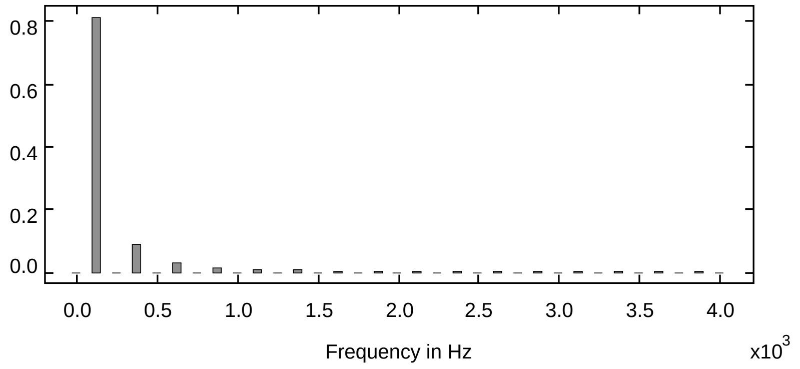


(b)

Figure 6.6: (a) One cycle of a square wave and some finite Fourier series approximations. (b) The relative amplitudes of the Fourier series terms for the square wave.



(a)



(b)

Figure 6.7: (a) One cycle of a triangle wave and some finite Fourier series approximations. (b) The relative amplitudes of the Fourier series terms for the triangle wave.

Example 6.5: Figure 6.7 shows some finite Fourier series approximations for a triangle wave. This waveform has no discontinuities, and therefore is much better behaved. Notice that its Fourier series components decrease in amplitude much more rapidly than those of the square wave. Moreover, the time-domain approximations appear to be much more accurate with fewer terms in the finite summation.

Many practical, real-world signals, such as audio signals, do not have discontinuities, and thus do not exhibit the convergence problems exhibited by the square wave. Other signals, however, such as images, are full of discontinuities. A (spatial) discontinuity in an image is simply an edge. Most images have edges. For such signals, we have to keep in mind that a Fourier series representation is always an approximation. It is, nonetheless, an extremely useful approximation.

Example 6.6: Consider an audio signal given by

$$s(t) = \sin(440 \times 2\pi t) + \sin(550 \times 2\pi t) + \sin(660 \times 2\pi t).$$

This is a major triad in a non-well-tempered scale. The first tone is A-440. The third is approximately E, with a frequency 3/2 that of A-440. The middle term is approximately C♯, with a frequency 5/4 that of A-440. It is these simple frequency relationships that result in a pleasant sound. We choose the non-well-tempered scale because it makes it much easier to construct a Fourier series expansion for this waveform. We leave the more difficult problem of finding the Fourier series coefficients for a well-tempered major triad to exercise 3.

To construct the Fourier series expansion, we can follow these steps:

1. Find p, the period. The period is the smallest number $p > 0$ such that $s(t) = s(t - p)$ for all t in the domain. To do this, note that

$$\sin(2\pi f t) = \sin(2\pi f (t - p))$$

   if fp is an integer. Thus, we want to find the smallest p such that $440p$, $550p$, and $660p$ are all integers. Equivalently, we want to find the largest fundamental frequency $f_0 = 1/p$ such that $440/f_0$, $550/f_0$, and $660/f_0$ are all integers. Such an f_0 is called the **greatest common divisor** of 440, 550, and 660. This can be computed using the `gcd` function in Matlab. In this case, however, we can do it in our heads, observing that $f_0 = 110$.
2. Find A_0, the constant term. By inspection, there is no constant component in $s(t)$, only sinusoidal components, so $A_0 = 0$.
3. Find A_1, the fundamental term. By inspection, there is no component at 110 Hz, so $A_1 = 0$. Since $A_1 = 0$, ϕ_1 is immaterial.
4. Find A_2, the first harmonic. By inspection, there is no component at 220 Hz, so $A_2 = 0$.
5. Find A_3. By inspection, there is no component at 330 Hz, so $A_3 = 0$.

Probing further: Convergence of the Fourier series

The Fourier series representation of a periodic signal x is a limit of a sequence of functions x_N for $N = 1, 2, \cdots$ where

$$\forall\, t \in \textit{Reals}, \quad x_N(t) = A_0 + \sum_{k=1}^{N} A_k \cos(k\omega_0 t + \phi_k).$$

Specifically, when the Fourier series representation exists, then for all $t \in$ *Reals*,

$$x(t) = \lim_{N\to\infty} x_N(t).$$

A particularly desirable form of convergence for this limit is **uniform convergence**, in which for each real number $\epsilon > 0$, there exists a positive integer M such that for all $t \in$ *Reals* and for all $N > M$,

$$|x(t) - x_N(t)| < \epsilon$$

A sufficient condition for uniform convergence is that the signal x is continuous and that its first derivative is piecewise continuous.

A square wave, for example, is not continuous, and hence does not satisfy this sufficient condition. Indeed, the Fourier series does not converge uniformly, as you can see in figure 6.6 by observing that the peak difference between $x(t)$ and $x_K(t)$ does not decrease to zero. A triangle wave, however, is continuous, and has a piecewise continuous first derivative. Thus, it does satisfy the sufficient condition. Its Fourier series approximation will therefore converge uniformly, as suggested in figure 6.7.

See for example R. G. Bartle, *The Elements of Real Analysis*, Second Edition, John Wiley & Sons, 1976, p. 117 (for uniform convergence) and p. 337 (for this sufficient condition).

6. Find A_4. There is a component at 440 Hz, $\sin(440 \times 2\pi t)$. We need to find A_4 and ϕ_4 such that

$$A_4 \cos(440 \times 2\pi t + \phi_4) = \sin(440 \times 2\pi t).$$

   By inspection, $\phi_4 = -\pi/2$ and $A_4 = 1$.
7. Similarly determine that $A_5 = A_6 = 1$, $\phi_5 = \phi_6 = -\pi/2$, and that all other terms are zero.

Putting this all together, the Fourier series expansion can be written

$$s(t) = \sum_{k=4}^{6} \cos(k\omega_0 t - \pi/2)$$

where $\omega_0 = 2\pi f_0 = 220\pi$.

Clearly the method used in the above example for determining the Fourier series coefficients is tedious and error prone, and will only work for simple signals. We will see much better techniques in Chapter 7.

6.5.1 Uniqueness of the Fourier series

Suppose x: *Reals* $\rightarrow$ *Reals* is a periodic function with period p. Then the Fourier series expansion is unique. In other words if it is both true that

$$x(t) = A_0 + \sum_{k=1}^{\infty} A_k \cos(k\omega_0 t + \phi_k)$$

and

$$x(t) = B_0 + \sum_{k=1}^{\infty} B_k \cos(k\omega_0 t + \theta_k),$$

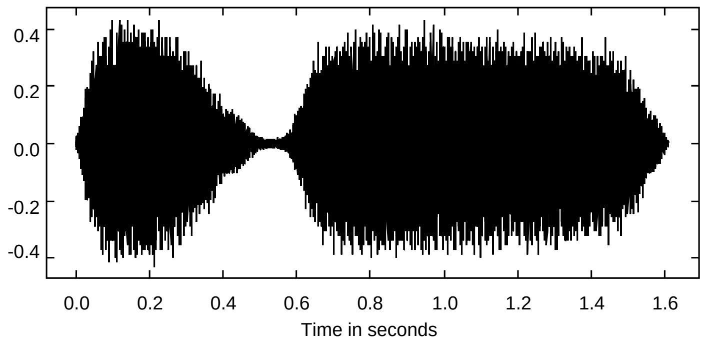
where $\omega_0 = 2\pi/p$, then it must also be true that

$$\forall\, k \geq 0, \quad A_k = B_k \text{ and } \phi_k \bmod 2\pi = \theta_k \bmod 2\pi.$$

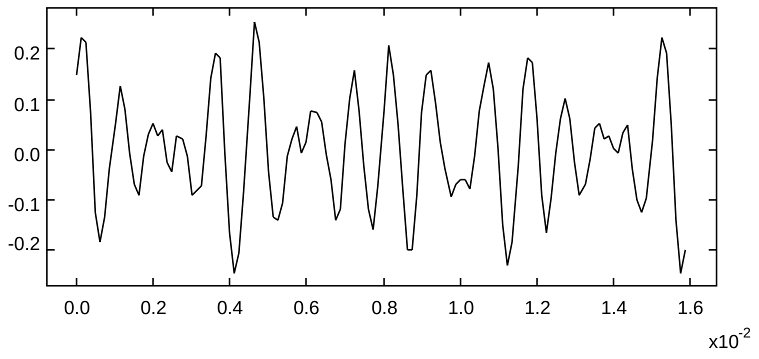
(The modulo operation is necessary because of the non-uniqueness of phase.) Thus, when we talk about *the* frequency content of a signal, we are talking about something that is unique and well defined. For a suggestion about how to prove this uniqueness, see problem9.

6.5.2 Periodic, finite, and aperiodic signals

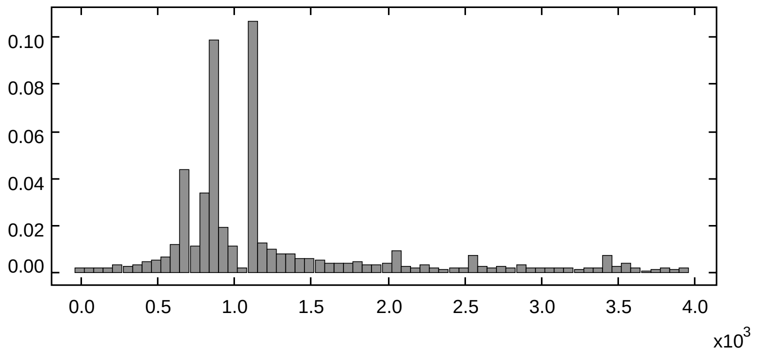
We have seen in section 6.4 that periodic signals and finite signals have much in common. One can be defined in terms of the other. Thus, a Fourier series can be used to describe a finite signal as well as a periodic one. The "period" is simply the extent of the finite signal. Thus, if the domain



(a)



(b)



(c)

Figure 6.8: (a) A 1.6 second train whistle. (b) A 16 msec segment of the train whistle. (c) The Fourier series coefficients for the 16 msec segment.

of the signal is $[a, b] \subset$ *Reals*, then $p = b - a$. The fundamental frequency, therefore, is just $\omega_0 = 2\pi/(b - a)$.

An aperiodic signal, like an audio signal, can be partitioned into finite segments, and a Fourier series can be constructed from each segment.

> **Example 6.7:** Consider the train whistle shown in figure 6.8(a). Figure 6.8(b) shows a segment of 16 msec. Notice that within this segment, the sound clearly has a somewhat periodic structure. It is not hard to envision how it could be described as sums of sinusoids. The magnitudes of the A_k Fourier series coefficients for this 16 msec segment are shown in figure 6.8(c). These are calculated on a computer using techniques we will discuss later, rather than being calculated by hand as in the previous example. Notice that there are three dominant frequency components that give the train whistle its tonality and timbre.

6.5.3 Fourier series approximations to images

Images are invariably finite signals. Given any image, it is possible to construct a periodic image by just tiling a plane with the image. Thus, there is again a close relationship between a periodic image and a finite one.

We have seen sinusoidal images (figure 6.4), so it follows that it ought to be possible to construct a Fourier series representation of an image. The only hard part is that images have a two-dimensional domain, and thus are finite in two distinct dimensions. The sinusoidal images in figure 6.4 have a vertical frequency, a horizontal frequency, or both.

Suppose that the domain of an image is $[a, b] \times [c, d] \subset$ *Reals* $\times$ *Reals*. Let $p_H = b - a$, and $p_V = d - c$ represent the horizontal and vertical "periods" for the equivalent periodic image. For constructing a Fourier series representation, we can define the horizontal and vertical fundamental frequencies as follows:

$$\omega_H = 2\pi/p_H$$
$$\omega_V = 2\pi/p_V$$

The Fourier series representation of *Image*: $[a, b] \times [c, d] \to Intensity$ is

$$\textit{Image}(x, y) = \sum_{k=0}^{\infty} \sum_{m=0}^{\infty} A_{k,m} \cos(k\omega_H x + \phi_k) \cos(m\omega_V y + \varphi_m)$$

For convenience, we have included the constant term $A_{0,0}$ in the summation, so we assume that $\phi_0 = \varphi_0 = 0$. (Recall that $\cos(0) = 1$).

6.6 Discrete-time signals

Consider signals of the form x: *Ints* $\to$ *Reals*, which are **discrete-time signals**. Discrete-time signals can be decomposed into sinusoidal components much like continuous-time signals. There are some minor subtleties, however.

Basics: Discrete-time frequencies

When the domain of a signal is *Ints*, then the units of frequency are cycles/sample. Consider for example the discrete-time signal given by

$$\forall n \in \textit{Ints}, x(n) = cos(2\pi f n).$$

Suppose this represents an audio signal that is sampled at 8000 samples/second. Then to convert f to Hertz, just watch the units:

$$f[cycles/sample] \times 8000[samples/second] = 8000f[cycles/second].$$

The frequency could have been equally well given in units of radians/sample, as in

$$x(n) = cos(\omega n).$$

for all $n \in \textit{Ints}$. To convert ω to Hertz,

$$\omega[radians/sample] \times 8000[samples/second] \times (1/2\pi)[cycles/radian]$$
$$= (8000w/2\pi)[cycles/second].$$

6.6.1 Periodicity

A discrete-time signal is **periodic** if there is a non-zero integer $p \in \textit{Ints}$ such that

$$\boxed{\forall\, n \in \textit{Ints}, \quad x(n+p) = x(n).}$$

Note that, somewhat counterintuitively, not all sinusoidal discrete-time signals are periodic. Consider

$$x(n) = cos(2\pi f n). \tag{6.6}$$

For this to be periodic, we must be able to find a non-zero integer p such that for all integers n,

$$x(n+p) = cos(2\pi f n + 2\pi f p) = cos(2\pi f n) = x(n).$$

This can be true only if $(2\pi f p)$ is an integer multiple of 2π. I.e., if there is some integer m such that

$$2\pi f p = 2\pi m.$$

Dividing both sides by $2\pi p$, we see that this signal is periodic only if we can find nonzero integers p and m such that

$$f = m/p.$$

In other words, f must be rational. Only if f is rational is this signal periodic.

Example 6.8: Consider a discrete-time sinusoid x given by

$$\forall\, n \in \textit{Ints}, \quad x(n) = \cos(4\pi n/5).$$

Putting this into the form of (6.6),

$$x(n) = \cos(2\pi(2/5)n),$$

we see that it has frequency $f = 2/5$ cycles/sample. If this were a continuous-time sinusoid, we could invert this to get the period. However, the inverse is not an integer, so it cannot possibly be the period. Noting that the inverse is 5/2 samples/cycle, we need to find the smallest multiple of this that is an integer. Multiply by 2, we get 5 samples/(2 cycles). So the period is $p = 5$ samples.

In general, for a discrete sinusoid with frequency f cycles/sample, the period is $p = K/f$, where $K > 0$ is the smallest integer such that K/f is an integer.

6.6.2 The discrete-time Fourier series

Assume we are given a periodic discrete-time signal x with period p. Just as with continuous-time signals, this signal can be described as a sum of sinusoids, called the **discrete-time Fourier series (DFS)** expansion,

$$x(n) = A_0 + \sum_{k=1}^{K} A_k \cos(k\omega_0 n + \phi_k) \tag{6.7}$$

where

$$K = \begin{cases} (p-1)/2 & \text{if } p \text{ is odd} \\ p/2 & \text{if } p \text{ is even} \end{cases}$$

Unlike the continuous-time case, the sum is finite. Intuitively, this is because discrete-time signals cannot represent frequencies above a certain value. We will examine this phenomenon in more detail in chapter 10, but for now, it proves extremely convenient. Mathematically, the above relation is much simpler than the continuous-time Fourier series. All computations are finite. There are a finite number of signal values in one period of the waveform. There are a finite number of terms in the Fourier series representation for each signal value. Unlike the continuous-time case, it is easy for computers to manage this representation. Given a finite set of values, $A_0, \cdots, A_K$, a computer can calculate $x(n)$. Moreover, the representation is exact for any periodic signal. No approximation is needed, and there is no question of convergence. In the continuous-time case, the Fourier series representation is accurate only for certain signals. For the discrete-time case, it is always accurate.

The DFS can be calculated efficiently on a computer using an algorithm called the **fast Fourier transform (FFT)**. All of the Fourier series examples that are plotted in this text were calculated using the FFT algorithm.

Exercises

Note: each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** In (6.1) we defined periodic for continuous-time signals.
   (a) Define **finite** and **periodic** for images.
   (b) Define **finite** and **periodic** for discrete-time signals, where the domain is *Ints*.

2. **E** Which of the following signals is periodic with a period greater than zero, and what is that period? All functions are of the form $x\colon \textit{Reals} \to \textit{Comps}$. The domain is time, measured in seconds, and so the period is in seconds.
   (a) $\forall\, t \in \textit{Reals}, \quad x(t) = 10\sin(2\pi t) + (10+2i)\cos(2\pi t)$
   (b) $\forall\, t \in \textit{Reals}, \quad x(t) = \sin(2\pi t) + \sin(\sqrt{2}\pi t)$
   (c) $\forall\, t \in \textit{Reals}, \quad x(t) = \sin(2\sqrt{2}\pi t) + \sin(\sqrt{2}\pi t)$

3. **T** Determine the fundamental frequency and the Fourier series coefficients for the well-tempered major triad,
$$s(t) = \sin(440 \times 2\pi t) + \sin(554 \times 2\pi t) + \sin(659 \times 2\pi t).$$

4. **E** Define $x\colon \textit{Reals} \to \textit{Reals}$
$$\forall\, t \in \textit{Reals}, \quad x(t) = 5\cos(\omega_0 t + \pi/2) + 5\cos(\omega_0 t - \pi/6) + 5\cos(\omega_0 t - 2\pi/3).$$
   Find A and ϕ so that
$$\forall\, t \in \textit{Reals}, \quad x(t) = A\cos(\omega_0 t + \phi).$$

5. **T** In this problem, we examine a practical application of the mathematical result in problem4. In particular, we consider multipath interference, a common problem with wireless systems where multiple paths from a transmitter to a receiver can result in destructive interference of a signal.

   When a transmitter sends a radio signal to a receiver, the received signal consists of the direct path plus several reflected paths. In figure 6.9, the transmitter is on a tower at the left of the figure, the receiver is the telephone in the foreground, and there are three paths: the direct path is l_0 meters long, the path reflected from the hill is l_1 meters long, and the path reflected from the building is l_2 meters long.

   Suppose the transmitted signal is a f Hz sinusoid $x\colon \textit{Reals} \to \textit{Reals}$,
$$\forall\, t \in \textit{Reals}, \quad x(t) = A\cos(2\pi f t)$$
   So the received signal is y such that $\forall\, t \in \textit{Reals}$,
$$y(t) = \alpha_0 A\cos(2\pi f(t - \frac{l_0}{c})) + \alpha_1 A\cos(2\pi f(t - \frac{l_1}{c})) + \alpha_2 A\cos(2\pi f(t - \frac{l_2}{c})). \quad (6.8)$$

Here, $0 \leq \alpha_i \leq 1$ are numbers that represent the attenuation (or reduction in signal amplitude) of the signal, and $c = 3 \times 10^8$ m/s is the speed of light.[1] Answer the following questions.

(a) Explain why the description of y given in (6.8) is a reasonable model of the received signal.

(b) What would be the description if instead of the 3 paths as shown in figure 6.9, there were 10 paths (one direct and 9 reflected).

(c) The signals received over the different paths cause different phase shifts, ϕ_i, so the signal y (with three paths) can also be written as

$$\forall\, t \in \textit{Reals}, \quad y(t) = \sum_{k=0}^{2} \alpha_k A \cos(2\pi f t - \phi_k)$$

What are the ϕ_k? Give an expression in terms of f, l_k, and c.

(d) Let $\Phi = \max\{\phi_1 - \phi_0, \phi_2 - \phi_0\}$ be the largest difference in the phase of the received signals and let $L = \max\{l_1 - l_0, l_2 - l_0\}$ be the maximum path length difference. What is the relationship between Φ, L, f?

(e) Suppose for simplicity that there is only one reflected path of distance l_1, i.e. take $\alpha_2 = 0$ in the expressions above. Then $\Phi = \phi_1 - \phi_0$. When $\Phi = \pi$, the reflected signal is said to *destroy* the direct signal. Explain why the term "destroy" is appropriate. (This phenomenon is called destructive interference.)

(f) In the context of mobile radio shown in the figure, typically $L \leq 500$m. For what values of f is $\Phi \leq \pi/10$? (Note that if $\Phi \leq \pi/10$ the signals will not interact destructively by much.)

(g) For the two-path case, drive an expression that relates the frequencies f that interfere destructively to the path length difference $L = l_1 - l_0$.

6. **T** The function $x\colon \textit{Reals} \to \textit{Reals}$ is given by its graph shown in figure 6.10. Note that $\forall\, t \notin [0,1],\ \ x(t) = 0$, and $x(0.4) = 1$. Define y by

$$\forall\, t \in \textit{Reals}, \quad y(t) = \sum_{k=-\infty}^{\infty} x(t - kp)$$

where p is the period.

(a) Prove that y is periodic with period p, i.e.

$$\forall\, t \in \textit{Reals}, y(t) = y(t+p).$$

(b) Plot y for $p = 1$.

(c) Plot y for $p = 2$.

(d) Plot y for $p = 0.8$.

(e) Plot y for $p = 0.5$.

[1] In reality, the reflections are more complicated than the model here.

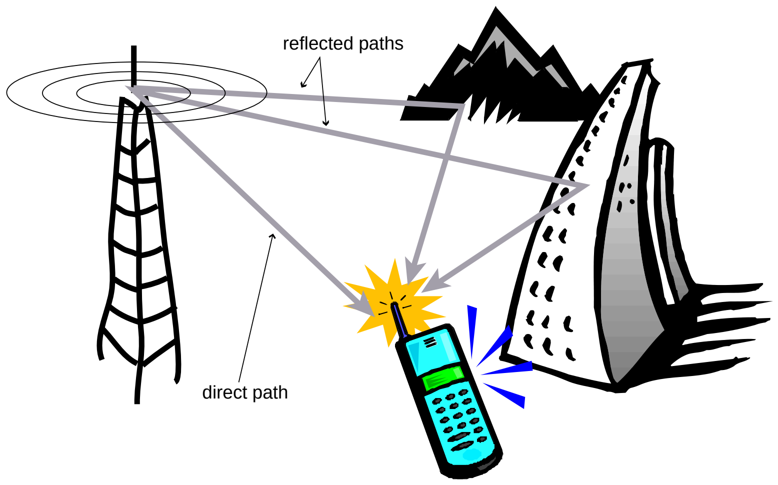


Figure 6.9: A direct and two reflected paths from transmitter to receiver.

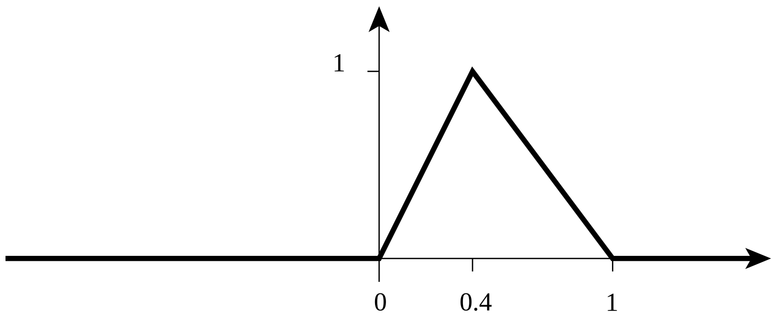


Figure 6.10: The graph of x.

(f) Suppose the function z is obtained by advancing x by 0.4, i.e.

$$\forall\, t \in \textit{Reals}, \quad z(t) = x(t + 0.4).$$

Define w by

$$\forall\, t \in \textit{Reals}, \quad w(t) = \sum_{k=-\infty}^{\infty} z(t - kp)$$

What is the relation between w and y. Use this relation to plot w for $p = 1$.

7. **T** Suppose $x\colon \textit{Reals} \to \textit{Reals}$ is a periodic signal with period p, i.e.

$$\forall\, t \in \textit{Reals}, \quad x(t) = x(t + p).$$

Let $f\colon \textit{Reals} \to \textit{Reals}$ be any function, and define the signal $y\colon \textit{Reals} \to \textit{Reals}$ by $y = f \circ x$, i.e.

$$\forall\, t \in \textit{Reals}, \quad y(t) = f(x(t)).$$

(a) Prove that y is periodic with period p.

(b) Suppose $\forall\, t \in \textit{Reals}, \quad x(t) = \sin(2\pi t)$. Suppose f is the sign function, $\forall\, a \in \textit{Reals}$,

$$f(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$$

Plot x and y.

(c) Suppose $\forall\, t \in \textit{Reals}, \quad x(t) = \sin(2\pi t)$. Suppose f is the square function, $\forall\, x \in \textit{Reals}$, $f(x) = x^2$. Plot y.

8. **C** Suppose the periodic square wave shown on the left in Figure 6.11 has the Fourier series representation

$$A_0 + \sum_{k=0}^{\infty} A_k \cos(2\pi kt/p + \phi_k)$$

Use this to obtain a Fourier series representation of the two-dimensional pattern of rectangles on the right. Note that the vertical and horizontal periods l, w are different.

9. **T** Suppose $A_k \in \textit{Comps}$, $\omega_k \in \textit{Reals}$, and $k = 1, 2$, such that

$$\forall\, t \in \textit{Reals}, \quad A_1 e^{i\omega_1 t} = A_2 e^{i\omega_2 t}. \tag{6.9}$$

Show that $A_1 = A_2$ and $\omega_1 = \omega_2$. **Hint:** Evaluate both sides of (6.9) at $t = 0$, and evaluate their derivatives at $t = 0$.

Discussion: This result shows that in order for two complex exponential signals to be equal, their frequencies, phases, and amplitudes must be equal. More interestingly, this result can be used to show that if a signal can be described as a sum of complex exponential signals, then that description is unique. There is no other sum of complex exponentials (one involving different frequencies, phases, or amplitudes) that will also describe the signal. In particular, the Fourier series representation of a periodic signal is unique, as stated above in theorem 6.5.1.

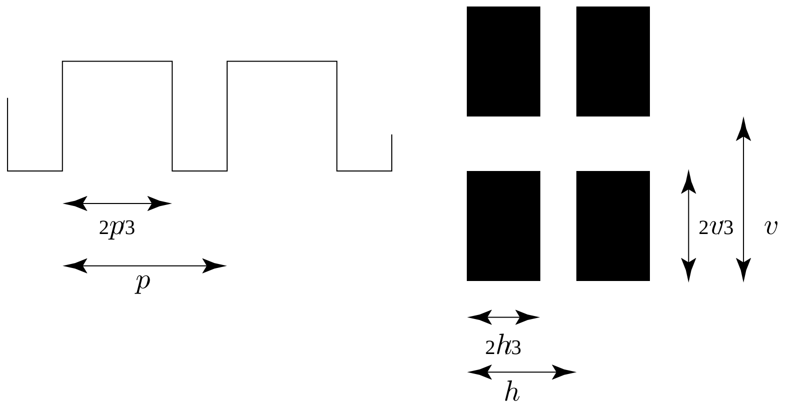


Figure 6.11: A periodic square wave (left) and a periodic pattern (right).

Chapter 7

Frequency Response

A class of systems that yield to sophisticated analysis techniques is the class of **linear time-invariant** (**LTI**) systems. LTI systems have a key property: given a sinusoidal input, the output is a sinusoidal signal with the same frequency, but possibly different amplitude and phase.

We can justify describing audio signals as sums of sinusoids on purely psychoacoustic grounds. However, because of this property of LTI systems, it is often convenient to describe any signal as sums of sinusoids, regardless of whether there is a psychoacoustic justification. The real value in this mathematical device is that by using the theory of LTI systems, we can design systems that operate more-or-less independently on the sinusoidal components of a signal. For example, abrupt changes in the signal value require higher frequency components. Thus, we can enhance or suppress these abrupt changes by enhancing or suppressing the higher frequency components. Such an operation is called **filtering** because it filters frequency components. We design systems by crafting their **frequency response**, their response to sinusoidal inputs. An audio equalizer, for example, is a filter that enhances or suppresses certain frequency components. Images can also be filtered. Enhancing the high frequency components will sharpen the image, whereas suppressing the high frequency components will blur the image.

State-space models described in previous chapters are precise and concise, but in a sense, not as powerful as a frequency response. For an LTI system, given a frequency response, you can assert a great deal about the relationship between an input signal and an output signal. Fewer assertions are practical in general with state-space models.

LTI systems, in fact, can also be described with state-space models, using **difference equations** and **differential equations**, as explored in chapter 5. But state-space models can also describe systems that are not LTI. Thus, state-space models are more general. It should come as no surprise that the price we pay for this increased generality is fewer analysis and design techniques. In this chapter, we explore the (very powerful) analysis and design techniques that apply to the special case of LTI systems.

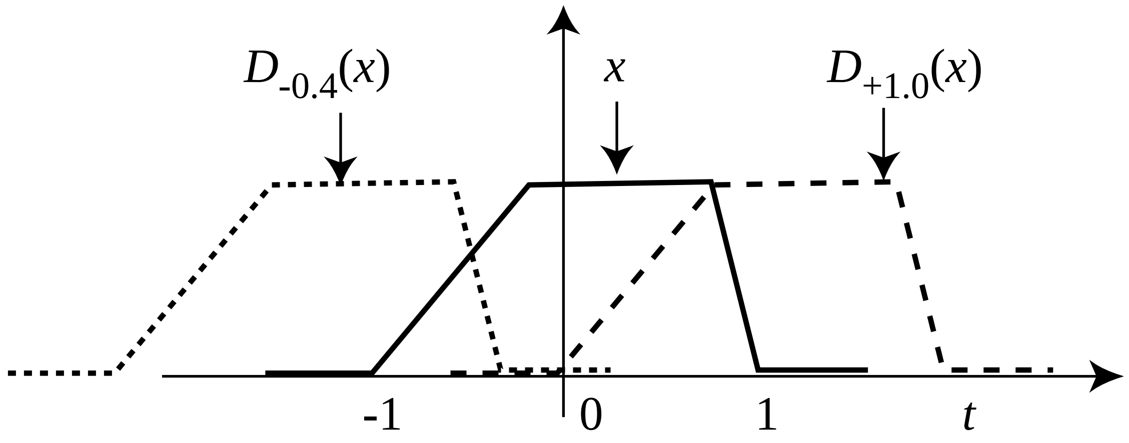


Figure 7.1: Illustration of the delay system D_τ. $D_{-0.4}(x)$ is the signal x to the left by 0.4, and $D_{+1.0}(x)$ is x moved to the right by 1.0.

7.1 LTI systems

LTI systems have received a great deal of intellectual attention for two reasons. First, they are relatively easy to understand. Their behavior is predictable, and can be fully characterized in fairly simple terms, based on the frequency domain representation of signals that we introduced in the previous chapter. Second, many physical systems can be reasonably approximated by them. Few physical systems perfectly fit the model, but many fit very well within a certain range of operation.

7.1.1 Time invariance

Consider the set of signals whose domain is *Reals*, interpreted as time. Such signals are **functions of time** or **time-domain signals**. This includes all audio signals, for example. For such signals, we can define a system called a **delay** operator D_τ so that if x is a function of time, then $D_\tau(x)$ is the new function of time given by

$$\forall\, t \in \textit{Reals}, \quad (D_\tau(x))(t) = x(t - \tau). \tag{7.1}$$

Positive values of τ result in positive delays, despite the subtraction in $x(t-\tau)$. Any delay results in a shifting left or right of the graph of a signal, as shown in figure7.1.

For sinusoidal signals time delay and phase changes are equivalent, except for the fact that phase is measured in radians (or degrees) rather than in time. In addition, a phase change of q is equivalent to a phase change of $q + n2\pi$ for any integer n. Phase applies to sinusoidal signals, whereas delay applies to any signal that is a function of time.

The space of all real-valued time-domain signals is denoted by $[\textit{Reals} \to \textit{Reals}]$. So the delay operator is a mapping from $[\textit{Reals} \to \textit{Reals}]$ to itself:

$$D_\tau : [\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}],$$

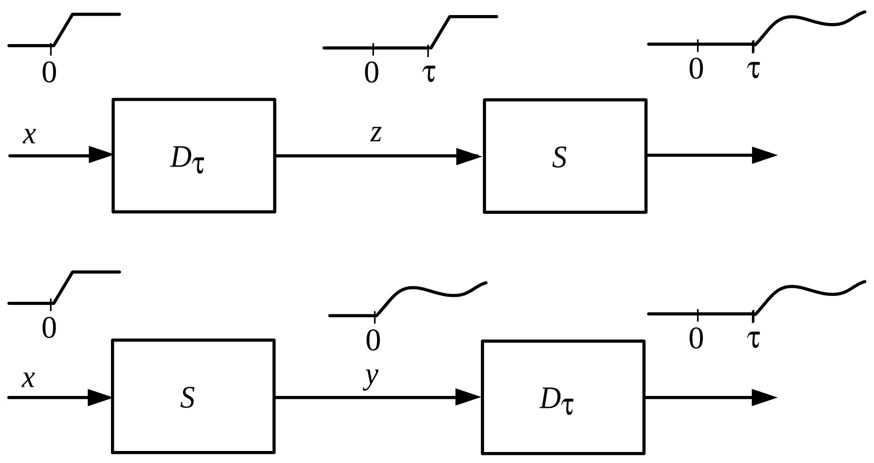


Figure 7.2: Time invariance implies that the top and bottom systems produce the same output signal for the same input signal.

where for each time-domain signal x, the time-domain signal $D_\tau(x)$ is given by (7.1).

Systems that map functions of time to functions of time are called **time-domain systems**. D_τ is a time-domain system. Audio systems are time-domain systems. Suppose that $S : [\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}]$ is a time-domain system. S is said to be **time invariant** if

$$S \circ D_\tau = D_\tau \circ S.$$

Figure 7.2 illustrates this equivalence, where the left hand side, $S \circ D_\tau$, is shown on top, and the right hand side, $D_\tau \circ S$, is shown on the bottom. It is very important to understand what this compact definition means. Since S and D_τ are both functions from $[\textit{Reals} \to \textit{Reals}]$ to $[\textit{Reals} \to \textit{Reals}]$, $S \circ D_\tau$ is also a function from $[\textit{Reals} \to \textit{Reals}]$ to $[\textit{Reals} \to \textit{Reals}]$. Its value at any x is given by $S(D_\tau(x))$. Similarly, $D_\tau \circ S$ is also a function from $[\textit{Reals} \to \textit{Reals}]$ to $[\textit{Reals} \to \textit{Reals}]$. Its value at x is $D_\tau(S(x))$.

S is **time-invariant** if for all x,

$$\boxed{S(D_\tau(x)) = D_\tau(S(x))}$$

That is, if we let $z = D_\tau(x)$ and $y = H(x)$, as in figure 7.2, then

$$\forall\, t \in \textit{Reals}, \quad (D_\tau(y))(t) = y(t - \tau) = (S(z))(t).$$

In pictures, time invariance implies that the upper and lower systems in figure 7.2 have identical behavior. A time-invariant system is one whose behavior (its response to inputs) does not change with time.

Time invariance is a mathematical fiction. No electronic system is time invariant in the strict sense. For one thing, such a system is turned on at some point in time. Clearly, its behavior before it is turned on is not the same as its behavior after it is turned on. Nevertheless, it proves to be a very convenient mathematical fiction, and it is a reasonable approximation for many systems if their behavior is constant over a relatively long period of time (relative to whatever phenomenon we are studying). For example, your audio amplifier is not a time-invariant system. Its behavior changes drastically when you turn it on or off, and changes less drastically when you raise or lower the volume. However, for the duration of a compact disc, if you leave the volume fixed, the system can be reasonably approximated as being time invariant.

Some systems have a similar property even though they operate on signals whose domain is not time. For example, the domain of an image is a region of a plane. The output of an image processing system may not depend significantly on where in the plane the input image is placed. Shifting the input image will only shift the output image by the same amount. This property which generalizes time invariance and holds for some image processing systems, is called **shift invariance** (see problem 3).

7.1.2 Linearity

Consider the set of signals whose range is *Reals* or *Comps*. Such signals are called **real-valued functions** or **complex-valued functions**. Since real-valued functions are a subset of complex-valued functions, we only need to talk about complex-valued functions. Suppose x is a complex-valued function and a is a complex constant. Then we can define a new complex-valued function ax such that for all t in the domain of x,

$$(ax)(t) = a(x(t)).$$

In other words, the new function, which we call ax, is simply scaled by the constant a.

Similarly, given two complex-valued functions x and y, we can define a new function $(x + y)$ such that for t in the domain of x (and domain of y),

$$(x + y)(t) = x(t) + y(t).$$

Consider the set of all systems that map complex-valued functions to complex-valued functions. Such systems are called **complex systems**. Suppose that S is a complex system. S is said to be **linear** if for all $a \in$ *Comps* and for all x and y that are complex-valued functions,

$$\boxed{S(ax) = aS(x)}$$

and

$$\boxed{S(x + y) = S(x) + S(y)}$$

The first of these says that if you scale the input, the output is scaled. The second one says that if the input is described as the sum of two component signals, then the output can be described as the sum of two signals that would result from the components alone.

In pictures, the first property says that the two systems in figure 7.3 are equivalent if S is linear. Here, the triangle represents the scaling operation. The second property says that the two systems figure 7.4 are equivalent.

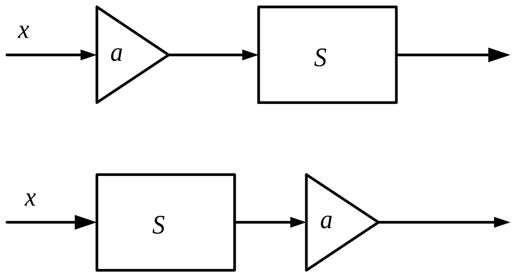


Figure 7.3: Linearity implies that these two systems are equivalent. The triangle is a system that scales a signal by some real constant a.

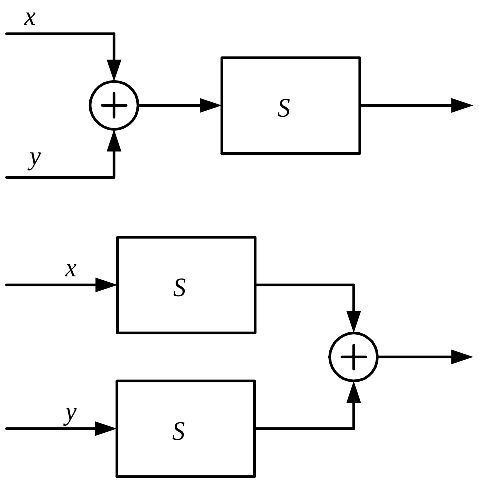


Figure 7.4: Linearity implies that these two systems are also equivalent.

Linearity is a mathematical fiction. No electronic system is linear in the strict sense. A system is designed to work with a range of input signals, and arbitrary scaling of the input does not translate into arbitrary scaling of the output. If you provide an input to your audio amplifier that is higher voltage than it is designed for, then it is not likely to just produce louder sounds. Its input circuits will get overloaded and signal distortion will result. Nonetheless, as a mathematical fiction, linearity is extremely convenient. It says that we can decompose the inputs to a system and study the effect of the system on the individual components.

7.1.3 Linearity and time-invariance

For time-domain systems, time-invariance is a useful (if fictional) property. For complex (or real) systems, linearity is a useful (if fictional) property. For complex (or real) time-domain systems, the combination of these properties is extremely useful. Linear time-invariant (LTI) systems turn out to have particularly simple behavior with sinusoidal inputs.

Given a sinusoid at the input, the output of the LTI system will be a sinusoid with the same frequency, but possibly with different phase and amplitude.

It then follows that

Given an input that is described as a sum of sinusoids of certain frequencies, the output can be described as a sum of sinusoids with the same frequencies, but with (possible) phase and amplitude changes at each frequency.

A straightforward way to show that LTI systems have these properties starts by considering **complex exponentials**. A complex exponential is a signal $x \in [\textit{Reals} \to \textit{Comps}]$ where

$$\boxed{\forall\, t \in \textit{Reals}, \quad x(t) = e^{i\omega t} = \cos(\omega t) + i\sin(\omega t).}$$

Complex exponential functions have an interesting property that will prove useful to us. Specifically,

$$\forall\, t \in \textit{Reals} \text{ and } \tau \in \textit{Reals}, \quad x(t-\tau) = e^{i\omega(t-\tau)} = e^{-i\omega\tau}e^{i\omega t}.$$

This follows from the multiplication property of exponentials,

$$e^{b+c} = e^b e^c.$$

Since $D_\tau(x)(t) = x(t-\tau)$, we have

$$D_\tau(x) = ax, \text{ where } a = e^{-i\omega\tau}. \tag{7.2}$$

In words, a delayed complex exponential is a scaled complex exponential, where the scaling constant is the complex number $a = e^{-i\omega\tau}$.

We will now show that if the input to an LTI system is $e^{i\omega t}$, then the output will be $H(\omega)e^{i\omega t}$, where $H(\omega)$ is a constant (not a function of time) that depends on the frequency ω of the complex exponential. In other words, the output is only a scaled version of the input.

When the output of a system is only a scaled version of the input, the input is called an **eigenfunction**, which comes from the German word for "same." The output is (almost) the same as the input.

Complex exponentials are eigenfunctions of LTI systems, as we will now show. This is the most important reason for the focus on complex exponentials in electrical engineering. This single property underlies much of the discipline of signal processing, and is used heavily in circuit analysis, communication systems, and control systems.

Given an LTI system $S : [\mathit{Reals} \to \mathit{Comps}] \to [\mathit{Reals} \to \mathit{Comps}]$, let x be an input signal where

$$\forall\, t \in \mathit{Reals}, \quad x(t) = e^{i\omega t}$$

Recall that time invariance implies that

$$S \circ D_\tau = D_\tau \circ S.$$

Thus

$$S(D_\tau(x)) = D_\tau(S(x)).$$

From (7.2),

$$S(D_\tau(x)) = S(ax)$$

where $a = e^{-i\omega\tau}$, and from linearity,

$$S(ax) = aS(x)$$

so

$$aS(x) = D_\tau(S(x)).$$

Let $y = S(x)$ be the corresponding output signal, so

$$ay = D_\tau(y).$$

In other words,

$$\forall\, t \in \mathit{Reals}, \quad e^{-i\omega\tau}y(t) = y(t-\tau).$$

In particular, this is true for $t = 0$, so

$$\forall\, \tau \in \mathit{Reals}, \quad y(-\tau) = e^{-i\omega\tau}y(0).$$

Changing variables, letting $t = -\tau$, we note that this implies that

$$\forall\, t \in \mathit{Reals}, \quad y(t) = e^{i\omega t}y(0).$$

Since $y(0)$ is a constant (it does not depend on t, although it does depend on ω), this establishes that the output is a complex exponential, just like the input, except that it is scaled by $y(0)$. Since

$y(0)$ in this case is a property of the system, and in general it depends on ω, we define the function $H\colon \textit{Reals} \to \textit{Comps}$ by

$$\boxed{\forall\, \omega \in \textit{Reals}, \quad H(\omega) = y(0) = (S(x))(0),} \tag{7.3}$$

where

$$\forall\, t \in \textit{Reals}, \quad x(t) = e^{i\omega t}.$$

That is, $H(\omega)$ is the output at time zero when the input is a complex exponential with frequency ω.

Using this notation, we write the output y as

$$\boxed{\forall\, t \in \textit{Reals}, \quad y(t) = H(\omega)e^{i\omega t}}$$

when the input is $e^{i\omega t}$. Note that $H(\omega)$ is a function of $\omega \in \textit{Reals}$, the frequency of the input complex exponential.

The function $H : \textit{Reals} \to \textit{Comps}$ is called the **frequency response**. It defines the response of the LTI system to a complex exponential input at any given frequency.

7.1.4 Discrete-time LTI systems

For discrete-time systems, the situation is similar. An N-sample **delay** is written D_N. A system $S\colon X \to Y$ is **time invariant** if for all $x \in X$ and for all $N \in \textit{Ints}$,

$$\boxed{S(D_N(x)) = D_N(S(x)).} \tag{7.4}$$

The system is **linear** if for all $x \in X$ and for all $a \in \textit{Reals}$,

$$\boxed{S(ax) = aS(x)}$$

and for all $x \in X$ and $x' \in X$

$$\boxed{S(x + x') = S(x) + S(x')}$$

By reasoning identical to that above, if the input is a **discrete complex exponential**,

$$\boxed{\forall\, x \in \textit{Ints}, \quad x(n) = e^{i\omega n}}$$

then the output is the same complex exponential scaled by a constant (a complex number that does not depend on time),

$$\boxed{\forall\, x \in \textit{Ints}, \quad y(n) = H(\omega)e^{i\omega n}}$$

H is once again called the frequency response, and since it is a function of ω, and is possibly complex valued, it has the form $H\colon \textit{Reals} \to \textit{Comps}$.

There is one key difference, however, between discrete-time systems and continuous-time systems. Since n is an integer, notice that

$$e^{i\omega n} = e^{i(\omega+2\pi)n} = e^{i(\omega+4\pi)n},$$

and so on. That is, a discrete complex exponential with frequency ω is identical to a discrete complex exponential with frequency $\omega + 2K\pi$, for any integer K. The frequency response, therefore, must be identical at these frequencies, since the inputs are identical. I.e.,

$$H(\omega) = H(\omega + 2K\pi)$$

for any integer K. That is, a discrete-time frequency response is periodic with period 2π.

7.2 Finding and using the frequency response

We have seen that if the input to an LTI system is a complex exponential signal $x \in [\mathit{Reals} \to \mathit{Comps}]$ where

$$\boxed{\forall\, t \in \mathit{Reals}, \quad x(t) = e^{i\omega t} = \cos(\omega t) + i \sin(\omega t).}$$

then the output is

$$\boxed{y(t) = H(\omega) e^{i\omega t}.} \tag{7.5}$$

where $H(\omega)$ is (possibly complex-valued) number that is a property of the system. $H(\omega)$ is called the frequency response at frequency ω. A similar

Example 7.1: Consider the delay system D_τ given by 7.1. Suppose the input to the delay is the complex exponential x given by

$$\forall\, t \in \mathit{Reals}, \quad x(t) = e^{i\omega t}.$$

Then the output y satisfies

$$\forall\, t \in \mathit{Reals}, \quad y(t) = e^{i\omega(t-\tau)} = e^{-i\omega\tau} e^{i\omega t}.$$

Comparing this to (7.5) we see that the frequency response of the delay is

$$H(\omega) = e^{-i\omega\tau}.$$

Example 7.2: Consider the discrete-time m-sample delay $D_N \colon [\mathit{Ints} \to \mathit{Reals}] \to [\mathit{Ints} \to \mathit{Reals}]$ such that for all $x \in [\mathit{Ints} \to \mathit{Reals}]$ and $n \in \mathit{Ints}$,

$$(D_N(x))(n) = x(n - N).$$

If the input to the delay is x and the output is y, then the delay is given by the difference equation

$$\forall\, n \in \mathit{Ints}, \quad y(n) = x(n - N). \tag{7.6}$$

This is an LTI system, so if the input is $x(n) = e^{i\omega n}$, then the output is $H(\omega)e^{i\omega n}$, where H is the frequency response. We can determine the frequency response using this fact by plugging this input and output into (7.6),

$$H(\omega) e^{i\omega n} = e^{i\omega(n-N)} = e^{i\omega n} e^{-i\omega N}.$$

Eliminating $e^{i\omega n}$ on both sides we get

$$H(\omega) = e^{-i\omega N}.$$

Basics: Sinusoids in terms of complex exponentials

Euler's formula states that

$$e^{i\theta} = \cos(\theta) + i\sin(\theta).$$

The complex conjugate is

$$e^{-i\theta} = \cos(\theta) - i\sin(\theta).$$

Summing these,

$$e^{i\theta} + e^{-i\theta} = 2cos(\theta)$$

or

$$\boxed{\cos(\theta) = (e^{i\theta} + e^{-i\theta})/2.}$$

Thus, for example,

$$\cos(\omega t) = 1/2(e^{i\omega t} + e^{-i\omega t}).$$

Similarly,

$$\boxed{\sin(\theta) = -i(e^{i\theta} - e^{-i\theta})/2.}$$

In appendix A we show that many useful trigonometric identities are easily derived from these simple relations.

The technique in the previous example can be used to find the frequency response of much more complicated systems. Simply replace the input x in a difference equation like (7.6) with $e^{i\omega n}$, and replace the output y with $H(\omega)e^{i\omega n}$, and then solve for $H(\omega)$.

Example 7.3: Consider a discrete-time, length two moving average, given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = (x(n) + x(n-1))/2,$$

where x is the input and y is the output. When the input is $e^{i\omega n}$, this becomes

$$H(\omega)e^{i\omega n} = (e^{i\omega n} + e^{i\omega(n-1)})/2.$$

Solving for $H(\omega)$, we find that the frequency response is

$$H(\omega) = (1 + e^{-i\omega})/2.$$

Complex exponentials as inputs are rather abstract. We have seen that with audio signals, sinusoidal signals are intrinsically significant because the human ear interprets the frequency of the sinusoid as its tone. Note that a real-valued sinusoidal signal can be given as a combination of exponential signals (see box),

$$\cos(\omega t) = (e^{i\omega t} + e^{-i\omega t})/2.$$

Tips and Tricks: Phasors

Consider a general continuous-time **sinusoidal signal**,

$$\forall\, t \in \textit{Reals}, \quad x(t) = A\cos(\omega t + \phi).$$

Here A is the **amplitude**, ϕ is the **phase**, and ω is the **frequency** of the sinewave. (We call this a sinewave, even though we are using cosine to describe it.) The units of ϕ are radians. The units of ω are radians per second, assuming t is in seconds. This can be written

$$x(t) = \textit{Re}\{Ae^{i\omega t + \phi}\} = \textit{Re}\{Ae^{i\phi}e^{\omega t}\} = \textit{Re}\{Xe^{i\omega t}\}$$

where $X = Ae^{i\phi}$ is called the **complex amplitude** or **phasor**. The representation

$$\boxed{x(t) = \textit{Re}\{Xe^{i\omega t}\}} \tag{7.7}$$

is called the phasor representation of x. It can be convenient.

Example 7.4: Consider summing two sinusoids with the same frequency,

$$x(t) = A_1\cos(\omega t + \phi_1) + A_2\cos(\omega t + \phi_2).$$

This is particularly easy using phasors, since

$$\begin{aligned} x(t) &= \textit{Re}\{(X_1 + X_2)e^{i\omega t}\} \\ &= |X_1 + X_2|\cos(\omega t + \angle(X_1 + X_2)) \end{aligned}$$

where $X_1 = A_1e^{i\phi_1}$ and $X_2 = A_2e^{i\phi_2}$. Thus, addition of the sinusoids reduces to addition of two complex numbers.

The exponential $Xe^{i\omega t}$ in (7.7) is complex valued. If we represent it in a two-dimensional plane as in figure 7.5, it will rotate in a counter-clockwise direction as t increases. The frequency of rotation will be ω radians per second. At time 0 it will be X, shown in gray. The real-valued sinewave $x(t)$ is the projection of $Xe^{i\omega t}$ on the real axis, namely

$$\textit{Re}\{Xe^{i\omega t}\} = |X|\cos(\omega t + \angle X).$$

The sum of two sinusoids with the same frequency is similarly depicted in figure 7.6. The two phasors, X_1 and X_2 are put head to tail and then rotated together. A similar method can be used to add sinusoids of different frequencies, but then the two vectors will rotate at different rates.

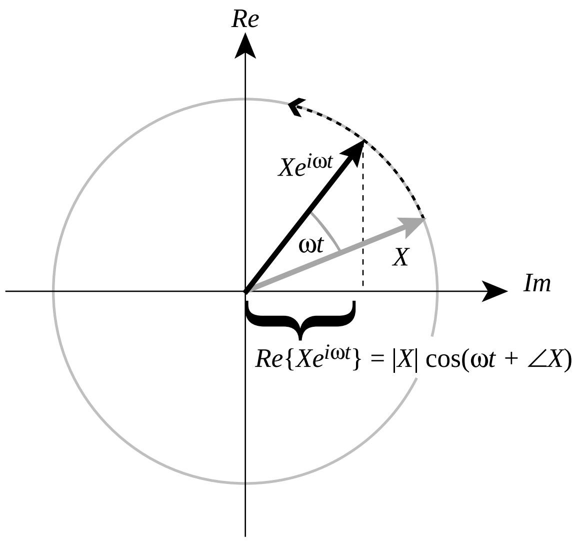


Figure 7.5: Phasor representation of a sinusoid.

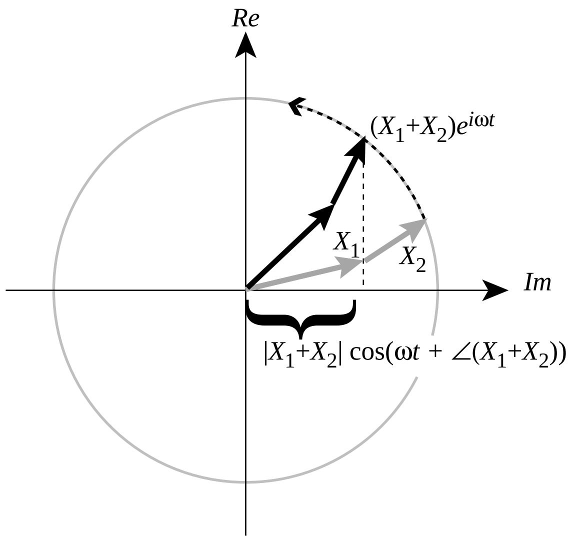


Figure 7.6: Phasor representation of the sum of two sinusoids with the same frequency.

Thus, if this is the input to an LTI system H, then the output will be

$$y(t) = (H(\omega)e^{i\omega t} + H(-\omega)e^{-i\omega t})/2.$$

Many (or most) LTI systems are not capable of producing complex-valued outputs when the input is real, so this $y(t)$ must be real. This implies that $H(\omega)e^{i\omega t}$ and $H(-\omega)e^{-i\omega t}$ must be complex conjugates of one another, which in turn implies that

$$H(\omega) = H^*(-\omega). \tag{7.8}$$

This property is called **conjugate symmetry**. The frequency response of a real system (one whose input and output signals are real-valued) is **conjugate symmetric**. Thus,

$$\forall\, t \in \textit{Reals}, \quad y(t) = \textit{Re}\{H(\omega)e^{i\omega t}\}.$$

If we write $H(\omega)$ in polar form,

$$H(\omega) = |H(\omega)|e^{i\angle H(\omega)},$$

then when the input is $\cos(\omega t)$, the output is

$$\boxed{\forall\, t \in \textit{Reals}, \quad y(t) = |H(\omega)|\cos(\omega t + \angle H(\omega)).}$$

Thus, $H(\omega)$ gives the **gain** $|H(\omega)|$ and **phase shift** $\angle H(\omega)$ that a sinusoidal input with frequency ω experiences. $|H(\omega)|$ is called the **magnitude response** of the system, and $\angle H(\omega)$ is called the **phase response**.

Example 7.5: The delay D_τ of example 7.1 has frequency response

$$H(\omega) = e^{-i\omega\tau}.$$

The magnitude response is

$$|H(\omega)| = 1.$$

Thus, any cosine input into a delay yields a cosine output with the same amplitude (obviously). A filter with a constant unity magnitude response is called an **allpass filter**, because it passes all frequencies equally. An N sample delay is a particularly simple form of an allpass filter.

The phase response is

$$\angle H(\omega) = -\omega\tau.$$

Thus, any cosine input yields a cosine output with phase shift $-\omega\tau$.

The discrete-time delay of example 7.2 is also an allpass filter. Its phase response is

$$\angle H(\omega) = -\omega N.$$

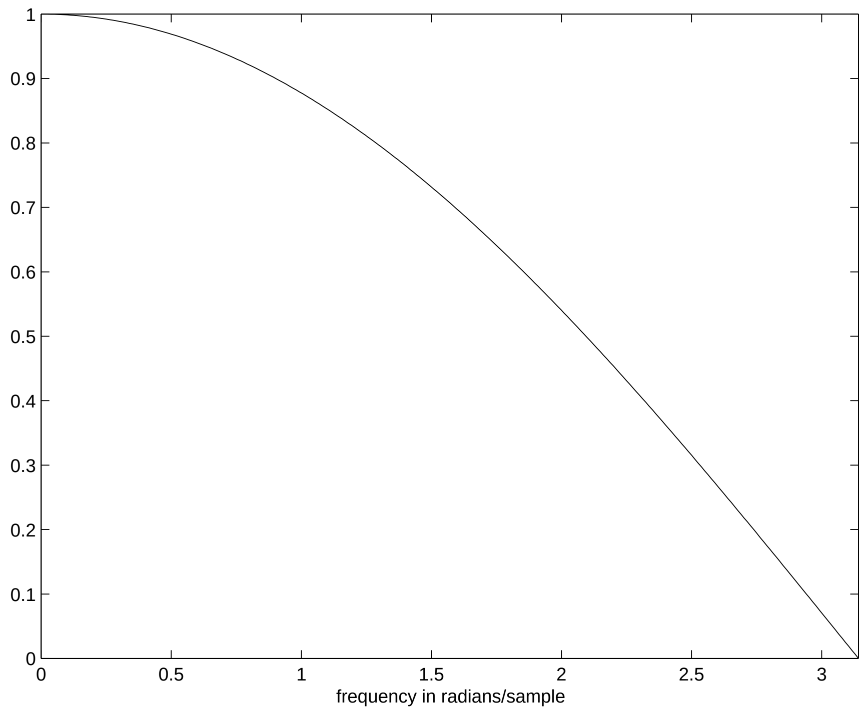


Figure 7.7: The magnitude response of a length-two moving average.

Example 7.6: The magnitude response of the length-two moving average considered in example 7.3 is

$$|H(\omega)| = |(1 + e^{-i\omega})/2|.$$

We can plot this using the following Matlab code, which (after adjusting the labels) results in figure 7.7.

```
omega = [0:pi/250:pi];
H = (1 + exp(-i*omega))/2;
plot(omega, abs(H));
```

Notice that at frequency zero (a cosine with zero frequency has constant value), the magnitude response is 1. That is, a constant signal gets through the filter without any reduction in amplitude. This is expected, since the average of two neighboring samples of a constant signal is simply the value of the constant signal. Notice that the magnitude response decreases as the frequency increases. Thus, higher frequency signals have their amplitudes reduced more by the filter than lower frequency signals. Such a filter is called a **lowpass** filter because it passes lower frequencies better than higher frequencies.

Often, we are given a frequency response, rather than some other description of an LTI system. The frequency response, in fact, tells us everything we need to know about the system. The next example begins the exploration of that idea.

Example 7.7: Suppose that the frequency response H of a discrete-time LTI system *Filter* is given by

$$\forall\, \omega \in \textit{Reals}, \quad H(\omega) = \cos(2\omega)$$

where ω has units of radians/sample. Suppose that the input signal $x\colon \textit{Ints} \to \textit{Reals}$ is such that for all $n \in \textit{Ints}$,

$$x(n) = \begin{cases} +1 & n \; even \\ -1 & n \; odd \end{cases}$$

We can determine the output. All we have to do is notice that the input can be written as

$$x(n) = \cos(\pi n).$$

Thus, the input is cosine. Hence, the output is

$$y(n) = |H(\pi)| \cos(\pi n + \angle H(\pi)) = \cos(\pi n) = x(n).$$

This input is passed unchanged by the system.

Suppose instead that the input is given by

$$x(n) = 5.$$

Once again, the input is a cosine, but this time with zero frequency,

$$x(n) = 5\cos(0n).$$

Hence the output is

$$y(n) = |H(0)| \cos(0n + \angle H(0)) = 5 = x(n).$$

This input is also passed unchanged by the system.

Suppose instead that the input is

$$x(n) = \cos(\pi n/2).$$

This input is given explicitly as a cosine, making our task easier. The output is

$$y(n) = |H(\pi/2)| \cos(\pi n/2 + \angle H(\pi/2)) = \cos(\pi n/2 + \pi) = -\cos(\pi n/2) = -x(n).$$

This input is inverted by the system.

Finally, suppose that the input is

$$x(n) = \cos(\pi n/4).$$

The output is

$$y(n) = |H(\pi/4)| \cos(\pi n/4 + \angle H(\pi/4)) = 0.$$

This input is removed by the system.

7.2.1 The Fourier series with complex exponentials

The Fourier series for a continuous-time, periodic signal $x : Reals \to Reals$ with period $p = 2\pi/\omega_0$, can be written as (see (6.4))

$$x(t) = A_0 + \sum_{k=1}^{\infty} A_k \cos(k\omega_0 t + \phi_k).$$

For reasons that we can now understand, the Fourier series is usually written in terms of complex exponentials rather than cosines. Since complex exponentials are eigenfunctions of LTI systems, this form of the Fourier series decomposes a signal into components that when processed by the system are only scaled.

Each term of the Fourier series expansion has the form

$$A_k \cos(k\omega_0 t + \phi_k)$$

which we can write (see box on page 184)

$$A_k \cos(k\omega_0 t + \phi_k) = A_k(e^{i(k\omega_0 t+\phi_k)} + e^{-i(k\omega_0 t+\phi_k)})/2.$$

So the Fourier series can also be written

$$x(t) = A_0 + \sum_{k=1}^{\infty} \frac{A_k}{2}(e^{i(k\omega_0 t+\phi_k)} + e^{-i(k\omega_0 t+\phi_k)}).$$

Observe that

$$e^{i(k\omega_0 t+\phi_k)} = e^{ik\omega_0 t}e^{i\phi_k},$$

and let

$$X_k = \begin{cases} A_0 & \text{if } k = 0 \\ 0.5A_k e^{i\phi_k} & \text{if } k > 0 \\ 0.5A_{-k}e^{-i\phi_{-k}} & \text{if } k < 0 \end{cases} \tag{7.9}$$

Then the Fourier series becomes

$$\boxed{x(t) = \sum_{k=-\infty}^{\infty} X_k e^{jk\omega_0 t}.} \tag{7.10}$$

This is the form in which one usually sees the Fourier series. Notice that the Fourier series coefficients are conjugate symmetric,

$$X_k = X^*_{-k}.$$

Of course, since (7.10) is an infinite sum, we need to worry about convergence (see box on page 164).

The **discrete-time Fourier series** (**DFS**) can be similarly written. If $x : Ints \to Reals$ is a periodic signal with period $p = 2\pi/\omega_0$, then we can write

$$\boxed{x(n) = \sum_{k=0}^{p-1} X_k e^{ik\omega_0 n}} \tag{7.11}$$

for suitably defined coefficients X_k. Relating the coefficients X_k to the coefficients A_k and ϕ_k is a bit more difficult in the discrete-time case than in the continuous-time case (see box).

There are two differences between (7.10) and (7.11). First, the sum in the discrete-time case is finite, making it manageable by computer. Second, it is exact for any periodic waveform. There are no mathematically tricky cases, and no approximation needed.

7.2.2 Examples

The Fourier series coefficients A_k of a square wave are shown in figure 6.6 in the previous chapter. The magnitudes of the corresponding coefficients X_k for the Fourier series expansion of (7.10) are shown in figure 7.8. Since each cosine is composed of two complex exponentials, there are twice as many coefficients.

Notice the symmetry in the figure. There are frequency components shown at both positive and negative frequencies. Notice also that the amplitude of the components is half that in figure 6.6, $|X_k| = |A_k|/2$. This is because there are now two components, one at negative frequencies and one at positive frequencies, that contribute.

7.3 Determining the Fourier series coefficients

We have seen that determining the Fourier series coefficients by directly attempting to determine the amplitude of individual frequency components can be difficult, even when the individual frequency components are known. Usually, however, they are not known. A general formula for computing the coefficients for a continuous-time periodic signal is given by

$$\boxed{X_m = \frac{1}{p}\int_0^p x(t)e^{-jm\omega_0 t}dt.} \tag{7.13}$$

The validity of this equation is demonstrated in the box on page 194.

The discrete-time case is somewhat simpler. A discrete-time periodic signal x with period $p \in$ *Ints* has Fourier series coefficients given by

$$\boxed{X_k = \frac{1}{p}\sum_{m=0}^{p-1} x(m)e^{-jmk\omega_0}.} \tag{7.14}$$

This can be shown by manipulations similar to those in the box on page 194. The practical importance in computing is much greater than that of the Fourier series for continuous-time signals. Since this sum is finite, the DFS coefficients can be easily computed precisely on a computer.

Probing further: Relating DFS coefficients

We have two discrete-time Fourier series expansions (see (7.11) and (6.7)),

$$x(n) = \sum_{k=0}^{p-1} X_k e^{ik\omega_0 n} \tag{7.12}$$

$$x(n) = A_0 + \sum_{k=1}^{K} A_k \cos(k\omega_0 n + \phi_k), \quad K = \begin{cases} (p-1)/2 & \text{if } p \text{ is odd} \\ p/2 & \text{if } p \text{ is even} \end{cases}$$

There is a relationship between the coefficients A_k, ϕ_k and X_k, but the relationship is more complicated than in the continuous-time case, given by (7.9). To develop that relationship, begin with the second of these expansions and write

$$x(n) = A_0 + \sum_{k=1}^{K} \frac{A_k}{2}(e^{i(k\omega_0 n+\phi_k)} + e^{-i(k\omega_0 n+\phi_k)}).$$

Note that since $\omega_0 = 2\pi/p$, then for all integers n, $e^{i\omega_0 pn} = 1$, so

$$e^{-i(k\omega_0 n+\phi_k)} = e^{-i(k\omega_0 n+\phi_k)} e^{i\omega_0 pn} = e^{i(\omega_0(p-k)n-\phi_k)}.$$

Thus,

$$\begin{aligned} x(n) &= A_0 + \sum_{k=1}^{K} \frac{A_k}{2} e^{i\phi_k} e^{ik\omega_0 n+} + \sum_{k=1}^{K} \frac{A_k}{2} e^{-i\phi_k} e^{i\omega_0(p-k)n} \\ &= A_0 + \sum_{k=1}^{K} \frac{A_k}{2} e^{i\phi_k} e^{ik\omega_0 n+} + \sum_{m=K}^{p-1} \frac{A_{p-m}}{2} e^{-i\phi_{p-m}} e^{i\omega_0 mn}, \end{aligned}$$

by change of variables. Comparing this against (7.12),

$$X_k = \begin{cases} A_0 & \text{if } k = 0 \\ A_k e^{i\phi_k}/2 & \text{if } k \in \{1, \cdots, K-1\} \\ A_k e^{i\phi_k}/2 + A_k e^{-i\phi_k}/2 = A_k \cos(\phi_k) & \text{if } k = K \\ A_{p-k} e^{-i\phi_{p-k}}/2 & \text{if } k \in \{K+1, \cdots, p-1\} \end{cases}$$

This relationship is more complicated than (7.9). Fortunately, it is rare that we need to use both forms of the DFS, so we can usually just pick one of the two forms and work only with its set of coefficients.

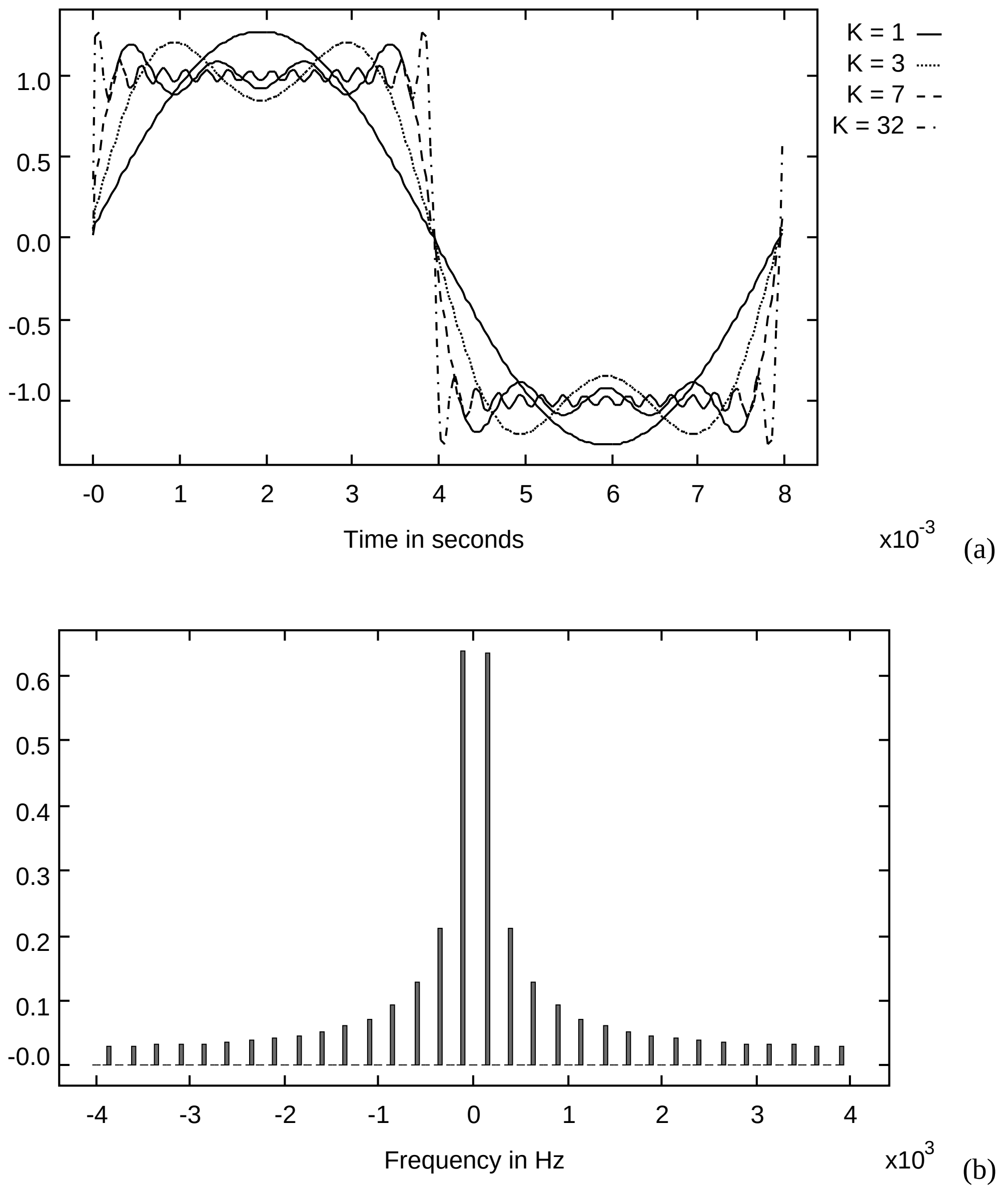


Figure 7.8: (a) One cycle of a square wave and some finite Fourier series approximations. The number of Fourier series terms that are included in the approximation is $2K + 1$, so K is the magnitude of the largest index the terms. (b) The magnitude of the complex Fourier series coefficients shown as a function of frequency.

Probing further: Formula for Fourier series coefficients

To see that (7.13) is valid, try substituting for $x(t)$ its Fourier series expansion,

$$x(t) = \sum_{k=-\infty}^{\infty} X_k e^{jk\omega_0 t}$$

to get

$$X_m = \frac{1}{p}\int_0^p \sum_{k=-\infty}^{\infty} X_k e^{jk\omega_0 t} e^{-jm\omega_0 t} dt.$$

Exchange the integral and summation (assuming this is valid, see box on page195) to get

$$X_m = \frac{1}{p}\sum_{k=-\infty}^{\infty} X_k \int_0^p e^{jk\omega_0 t} e^{-jm\omega_0 t} dt.$$

The exponentials can be combined to get

$$X_m = \frac{1}{p}\sum_{k=-\infty}^{\infty} X_k \int_0^p e^{j(k-m)\omega_0 t} dt.$$

In the summation, where k varies over all integers, there will be exactly one term of the summation where $k = m$. In that term, the integral evaluates to p. For the rest of the terms, $k \neq m$. Separating these two situations, we can write

$$X_m = X_m + \frac{1}{p}\sum_{k=-\infty, k\neq m}^{\infty} X_k \int\limits_0^p e^{j(k-m)\omega_0 t} dt,$$

where the first term X_m is the value of the term in the summation where $k = m$. For each remaining term of the summation, the integral evaluates to zero, thus establishing our result. To show that the integral evaluates to zero, let $n = k - m$, and note that $n \neq 0$. Then

$$\int_0^p e^{jn\omega_0 t} dt = \int_0^p \cos(n\omega_0 t) dt + j\int_0^p \sin(n\omega_0 t) dt$$

Since $\omega_0 = 2\pi/p$, these two integrals exactly span one or more complete cycles of the cosine or sine, and hence integrate to zero.

Probing further: Exchanging integrals and summations

The demonstration of the validity of the formula for the Fourier series coefficients in the box on page 194 relies on being able to exchange an integral and an infinite summation. The infinite summation can be given as a limit of a sequence of functions

$$x_N(t) = \sum_{k=-N}^{N} X_k e^{jk\omega_0 t}.$$

Thus, we wish to exchange the integral and limit in

$$X_m = \frac{1}{p}\int_0^p (\lim_{N\to\infty} x_N(t))dt.$$

A sufficient condition for being able to perform the exchange is that the limit converges uniformly in the interval [0, p]. A sufficient condition for uniform convergence is that x is continuous and that its first derivative is piecewise continuous.

See R. G. Bartle, *The Elements of Real Analysis*, Second Edition, John Wiley & Sons, 1976, p. 241.

7.3.1 Negative frequencies

The Fourier series expansion for a periodic signal $x(t)$ is

$$x(t) = \sum_{k=-\infty}^{\infty} X_k e^{jk\omega_0 t}$$

This includes Fourier series coefficients for the constant term (when $k = 0$, $e^{jk\omega_0 t} = 1$), plus the fundamental and harmonics $(k \geq 1)$. But it also includes terms that seem to correspond to negative frequencies. When $k \leq -1$, the frequency $k\omega_0$ is negative. These negative frequencies balance the positive ones so that the resulting sum is real-valued.

7.4 Frequency response and the fourier series

Recall that if the input to an LTI system S is a complex exponential signal $x \in [\textit{Reals} \to \textit{Comps}]$ where

$$\forall\, t \in \textit{Reals}, \quad x(t) = e^{i\omega_0 t} = \cos(\omega_0 t) + i\sin(\omega_0 t).$$

then the output for all t is

$$y(t) = H(\omega_0)e^{i\omega_0 t},$$

where the complex number $H(\omega_0)$ is is the frequency response of the system at frequency ω_0. It is equal to the output at time zero $y(0)$ when the input is $e^{i\omega_0 t}$. H itself is a function $H\colon \textit{Reals} \to$

Comps that in principle can be evaluated for any frequency $\omega \in$ *Reals*, including negative frequencies.

Recall further that if an input $x(t)$ to the system S is a periodic signal with period p, then it can be represented as a Fourier series,

$$x(t) = \sum_{k=-\infty}^{\infty} X_k e^{jk\omega_0 t}$$

By linearity and time invariance of S, the output $S(x)$ for this periodic input x, is

$$\boxed{y(t) = \sum_{k=-\infty}^{\infty} H(k\omega_0) X_k e^{jk\omega_0 t}}$$

Linearity tells us that if the input is decomposed into a sum of components, then the output can be decomposed into a sum of components where each component is the response of the system to a single input component. Linearity together with time invariance tells us that each component, which is a complex exponential, is simply scaled. Thus, the output is given by a Fourier series with coefficients $X_k H(k\omega_0)$.

This major result tells us:

- There are no frequency components in the output that were not in the input. The output consists of the same frequency components as the input, but with each component individually scaled.
- LTI systems can be used to enhance or suppress certain frequency components. Such operations are called **filtering**.
- The frequency response function characterizes which frequencies are enhanced or suppressed, and also what phase shifts might be imposed on individual components by the system.

7.5 Frequency response of composite systems

In section 2.1.5 we studied several ways of composing systems (using the block diagram syntax) to obtain more complex, composite systems. We will see in this section that when each block is an LTI system, the resulting composite system is also LTI. Moreover, we can easily obtain the frequency response of the composite system from the frequency response of each block. This provides a very important tool for the analysis and synthesis of composite LTI systems. This tool works equally well with discrete and continuous systems.

7.5.1 Cascade connection

Consider the composite system S obtained by the cascade connection of systems S_1 and S_2 in figure 7.9. Suppose S_1 and S_2 are LTI. We first show that $S = S_2 \circ S_1$ is LTI. To show that S is

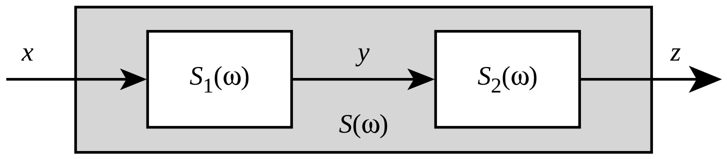


Figure 7.9: The cascade connection of the two LTI systems is the system $S = S_2 \circ S_1$. The frequency response is related by $\forall \omega, H(\omega) = H_1(\omega)H_2(\omega)$.

time-invariant, we must show that for any τ, $S \circ D_{tau} = D_\tau \circ S$. But,

$$\begin{aligned} S \circ D_\tau &= S_2 \circ S_1 \circ D_\tau \\ &= S_2 \circ D_\tau \circ S_1, \text{ since } S_1 \text{ is time-invariant} \\ &= D_\tau \circ S_2 \circ S_1, \text{ since } S_2 \text{ is time-invariant} \\ &= D_\tau \circ S, \end{aligned}$$

as required.

We now show that S is linear. Let x be any input signal and a any complex number. Then

$$\begin{aligned} S(ax) &= S_2 \circ S_1(ax) \\ &= S_2(aS_1(x)), \text{ since } S_1 \text{ is linear} \\ &= aG(S_1(x)), \text{ since } S_2 \text{ is linear} \\ &= aS(x). \end{aligned}$$

Lastly, if x and y are two input signals, then

$$\begin{aligned} S(x+y) &= S_2 \circ S_1(x+y) \\ &= S_2(S_1(x) + S_1(y)), \text{ since } S_1 \text{ is linear} \\ &= S_2(S_1(x)) + S_2(S_1(y)), \text{ since } S_2 \text{ is linear} \\ &= S(x) + S(y). \end{aligned}$$

This shows that S is linear.

We now compute the frequency response of S. Let $H_1(\omega), H_2(\omega), H(\omega)$ be the frequency response of these systems at the frequency ω. Consider the complex exponential input x:

$$\forall t \in \mathit{Reals}, \quad x(t) = e^{i\omega t}.$$

Then the signal $y = S_1(x)$ is a multiple of x, namely, $y = H_1(\omega)x$. In particular, y is a complex exponential input, and so $z = S_2(y)$ is given by

$$z = H_2(\omega)y = H_2(\omega) \times H_1(\omega)x.$$

But since $H(\omega)$ is the frequency response of S at the frequency ω, we also have

$$z = S(x) = H(\omega)x,$$

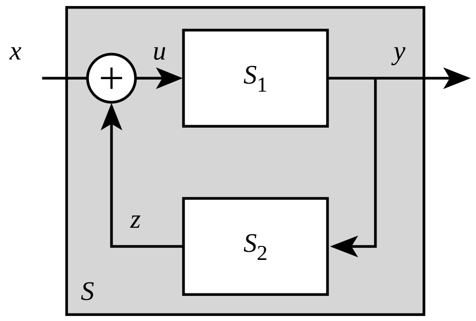


Figure 7.10: The feedback connection of the two LTI systems S_1, S_2 is S. The frequency response is related by $\forall \omega, H(\omega) = H_1(\omega)/[1 - H_1(\omega)H_2(\omega)]$.

and so we obtain

$$\boxed{\forall \omega \in \textit{Reals}, \quad H(\omega) = H_2(\omega)H_1(\omega).} \tag{7.15}$$

Exactly the same formula applies in the discrete-time case. This is a remarkable result. First, suppose that the cascade connection of figure 7.9 was reversed, i.e. consider the system $\tilde{S} = S_1 \circ S_2$. Then the frequency response of $\tilde{S}$ is

$$\tilde{H}(\omega) = H_1(\omega) \times H_2(\omega) = H_2(\omega) \times H_1(\omega) = H(\omega).$$

That is, $\tilde{S}$ and S have the same frequency response! This implies, in fact, that S and $\tilde{S}$ are equivalent: they give the same output for the same input. So in any cascade connection of LTI systems, the order in which the systems are composed does not matter.

7.5.2 Feedback connection

The feedback arrangement shown in figure 7.10 is fundamental to the design of control systems. The overall system S is called the **closed-loop** system. We first show that S is LTI if S_1 and S_2 are LTI, and we then calculate its frequency response. Suppose x is the input signal, and define the signals u, z and y as shown. The circle with the plus sign represents the relationship $u = x + z$. The signals are then related by,

$$\begin{aligned} y &= S_1(u) \\ &= S_1(x+z) \\ &= S_1(x) + S_1(z), \text{ since } S_1 \text{ is linear} \\ &= S_1(x) + S_1(S_2(y)), \end{aligned}$$

Note that this equation relates the input and output, but the output appears on both sides. We can rewrite this as

$$y - S_1(S_2(y)) = S_1(x). \tag{7.16}$$

Thus, given the input signal x, the output signal y is obtained by solving this equation. We will assume that for any signal x (7.16) has a unique solution y. Then, of course, $y = S(x)$. We can use methods similar to the ones we used for the cascade example to show that S is LTI (see box).

We now compute the frequency response $H(\omega)$ of S at frequency ω. Let the frequency response of S_1 be H_1, and of S_2 be H_2. Suppose the input signal is the complex exponential

$$\forall t \in \textit{Reals}, \quad x(t) = e^{i\omega t}.$$

For this input, we know that $S_1(x) = H_1(\omega)x$ and $S_2(x) = H_2(\omega)x$. Since S is LTI, we know that the output signal y is given by

$$y = H(\omega)x.$$

Using this relation for y in (7.16) we get

$$\begin{aligned} H(\omega)x - S_1(S_2(H(\omega)x)) &= H(\omega)[x - S_1(S_2(x))], \text{ since } S_2, S_1 \text{ are linear} \\ &= H(\omega)[x - H_1(\omega)H_2(\omega)x] \\ &= H(\omega)[1 - H_1(\omega)H_2(\omega)]x \\ &= S_1(x) \\ &= H_1(\omega)x, \text{ by (7.16)}, \end{aligned}$$

from which we get the frequency response of the feedback system,

$$\boxed{H(\omega) = \frac{H_1(\omega)}{1 - H_1(\omega)H_2(\omega)}} \quad (7.19)$$

This relation is at the foundation of linear feedback control design.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Find $A \in \textit{Comps}$ so that

$$\forall\, t \in Reals, \quad Ae^{i\omega t} + A^* e^{-i\omega t} = \cos(\omega t + \pi/4),$$

where A^* is the complex conjugate of A.

2. **E** Plot the function $s\colon \textit{Reals} \to \textit{Reals}$ given by

$$\forall\, x \in \textit{Reals}, \quad s(x) = \textit{Im}\{e^{(-x+i2\pi x)}\}.$$

You are free to choose a reasonable interval for your plot, but be sure it includes $x = 0$.

3. **E** Analogously to D_τ in (7.1) and D_N in example 7.2, define formally the following variants:

Probing further: Feedback systems are LTI

To show S in figure 7.10 is time-invariant we must show that for any τ,

$$S(D_\tau(x)) = D_\tau(S(x)) = D_\tau(y), \tag{7.17}$$

that is, we must show that $D_\tau(y)$ is the output of S when the input is $D_\tau(x)$. Now the left-hand side of (7.16) with y replaced by $D_\tau(y)$ is

$$\begin{aligned}
D_\tau(y) - S_1(S_2(D_\tau(y))) &= D_\tau(y) - D_\tau(S_1(S_2(y)), \\
&\quad \text{since } S_1 \text{ and } S_2 \text{ are time-invariant} \\
&= D_\tau(y - S_1(S_2(y))), \text{ since } D_\tau \text{ is linear}, \\
&= D_\tau(S_1(x)), \text{ by (7.16)} \\
&= S_1(D_\tau(x)), \text{ since } S_1 \text{ is time-invariant}
\end{aligned}$$

so that $D_\tau(y)$ is indeed the solution of (7.16) when the input is $D_\tau(x)$. This proves (7.17).

Linearity is shown in a similar manner. Let a be any complex number. To show that ay is the output when the input is ax we evaluate the left-hand side of (7.16) at ay,

$$\begin{aligned}
ay - S_1(S_2(ay)) &= ay - aS_1(S_2(y)), \text{ since } S_2 \text{ and } S_1 \text{ are linear} \\
&= a[y - S_1(S_2(y))] \\
&= aS_1(x), \text{ by (7.16)} \\
&= S_1(ax), \text{ since } S_1 \text{ is linear}
\end{aligned}$$

which shows that $S(ax) = aS(x)$.

Next suppose w is another input and $z = S(w)$ the corresponding output, i.e.

$$z - S_1(S_2(z)) = S_1(w). \tag{7.18}$$

We evaluate the left-hand side of (7.16) at $y + z$,

$$\begin{aligned}
(y+z) - S_1(S_2(y+z)) &= [y - S_1(S_2(y))] + [z - S_1(S_2(z)], \\
&\quad \text{since } S_2 \text{ and } S_1 \text{ are linear} \\
&= S_1(x) + S_1(w), \text{ by (7.16) and (7.18)} \\
&= S_1(x + w), \text{ since } S_1 \text{ is linear}
\end{aligned}$$

and so $S(x + w) = y + z = S(x) + S(z)$.

   (a) A shift operator $S_{v,h}$ that shifts an image v units vertically and h units horizontally, where $v \in$ *Reals* and $h \in$ *Reals*.
   (b) A shift operator $S_{m,n}$ that shifts a discrete image m units vertically and n units horizontally, where $m \in$ *Ints* and $n \in$ *Ints*.

4. **E** Consider a discrete-time system $D\colon [\textit{Ints} \to \textit{Reals}] \to [\textit{Ints} \to \textit{Reals}]$, where if $y = D(x)$ then
$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n-1).$$
   (a) Is D linear? Justify your answer.
   (b) Is D time-invariant? Justify your answer.

5. **E** Consider a continuous-time system *TimeScale*: $[\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}]$, where if $y = \textit{TimeScale}(x)$ then
$$\forall\, t \in \textit{Reals}, \quad y(t) = x(2t).$$
   (a) Is *TimeScale* linear? Justify your answer.
   (b) Is *TimeScale* time-invariant? Justify your answer.

6. **E** Suppose that the frequency response of a discrete-time LTI system S is given by
$$H(\omega) = |\sin(\omega)|$$
where ω has units of radians/sample. Suppose the input is the discrete-time signal x given by $\forall\, n \in \textit{Ints},\ x(n) = 1$. Give a *simple* expression for $y = S(x)$.

7. **T** Find the smallest positive integer n such that
$$\sum_{k=0}^{n} e^{i5k\pi/6} = 0.$$
Hint: Note that the term being summed is a periodic function of k. What is its period? What is the sum of a complex exponential over one period?

8. **T** Consider a continuous-time periodic signal x with fundamental frequency $\omega_0 = 1$ radian/second. Suppose that the Fourier series coefficients (see (6.4)) are
$$A_k = \begin{cases} 1 & k = 0,\ 1,\ or\ 2 \\ 0 & \text{otherwise} \end{cases}$$
and for all $k \in \textit{Nats}_0$, $\phi_k = 0$.

   (a) Find the Fourier series coefficients X_k for all $k \in$ *Ints* (see (7.10)).
   (b) Consider a continuous-time LTI system *Filter*: $[\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}]$, with frequency response
$$H(\omega) = \cos(\pi\omega/2).$$
   Find $y = \textit{Filter}(x)$. I.e., give a simple expression for $y(t)$ that is valid for all $t \in$ *Reals*.

(c) For y calculated in (b), find the fundamental frequency in radians per second. I.e., find the largest $\omega_0' > 0$ such that

$$\forall\, t \in \textit{Reals}, \quad y(t) = y(t + 2\pi/\omega_0')$$

9. **T** Consider a continuous-time LTI system S. Suppose that when the input x is given by

$$\forall t \in Reals, \quad x(t) = \begin{cases} 1, & \text{if } 0 \le t < 1 \\ 0, & \text{otherwise} \end{cases}$$

then the output $y = S(x)$ is given by

$$\forall t \in Reals, \quad y(t) = \begin{cases} 1, & \text{if } 0 \le t < 2 \\ 0, & \text{otherwise} \end{cases}$$

Give an expression and a sketch for the output of the same system if the input is

$$\forall t \in Reals, \quad x'(t) = \begin{cases} 1, & \text{if } 0 \le t < 1 \\ -1, & \text{if } 1 \le t < 2 \\ 0, & \text{otherwise} \end{cases}$$

10. **T** Suppose that the frequency response H of a discrete-time LTI system *Filter* is given by:

$$\forall\, \omega \in [-\pi, \pi], \quad H(\omega) = |\omega|.$$

where ω has units of radians/sample. Note that since a discrete-time frequency response is periodic with period 2π, this definition implicitly gives $H(\omega)$ for all $\omega \in$ *Reals*. Give simple expressions for the output y when the input signal x : *Ints* $\to$ *Reals* is such that $\forall\, n \in$ *Ints* each of the following is true:

(a) $x(n) = \cos(\pi n/2)$.

(b) $x(n) = 5$.

(c) $x(n) = \begin{cases} +1, & n \text{ even} \\ -1, & n \text{ odd} \end{cases}$

11. **T** Consider a continuous-time LTI system S. Suppose that when the input is given by

$$x(t) = \begin{cases} \sin(\pi t) & 0 \le t < 1 \\ 0 & \text{otherwise} \end{cases}$$

then the output $y = S(x)$ is given by

$$y(t) = \begin{cases} \sin(\pi t) & 0 \le t < 1 \\ \sin(\pi(t-1)) & 1 \le t < 2 \\ 0 & \text{otherwise} \end{cases}$$

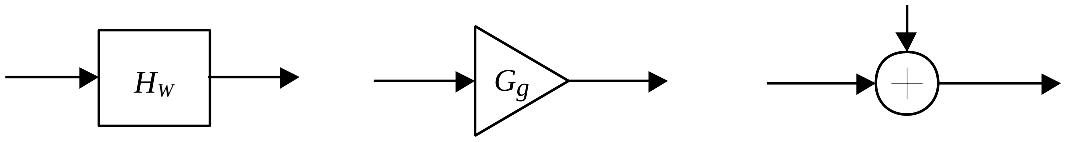
for all $t \in$ *Reals*.

(a) Carefully sketch these two signals.

(b) Give an expression and a sketch for the output of the same system if the input is

$$x(t) = \begin{cases} \sin(\pi t) & 0 \le t < 1 \\ -\sin(\pi(t-1)) & 1 \le t < 2 \\ 0 & \text{otherwise} \end{cases} .$$

12. **T** Suppose you are given the building blocks shown below for building block diagrams:



These blocks are defined as follows:

- An LTI system H_W: [*Reals* $\to$ *Reals*] $\to$ [*Reals* $\to$ *Reals*] that has a rectangular frequency response given by

$$\forall\, \omega \in \textit{Reals}, \quad H(\omega) = \begin{cases} 1 & -W < \omega < W \\ 0 & \text{otherwise} \end{cases}$$

where W is a parameter you can set.

- A gain block G_g: [*Reals* $\to$ *Reals*] $\to$ [*Reals* $\to$ *Reals*] where if $y = g(x)$, then

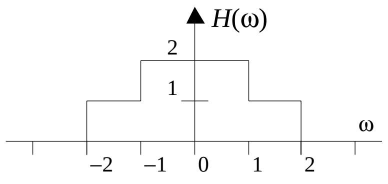
$$\forall\, t \in \textit{Reals}, \quad y(t) = gx(t)$$

where $g \in$ *Reals* is a parameter you can set.

- An adder, which can add two continuous-time signals. Specifically, *Add*: [*Reals* $\to$ *Reals*] $\times$ [*Reals* $\to$ *Reals*] $\to$ [*Reals* $\to$ *Reals*] such that if $y = \textit{Add}(x_1, x_2)$ then

$$\forall\, t \in \textit{Reals}, \quad y(t) = x_1(t) + x_2(t).$$

Use these building blocks to construct a system with the frequency response shown below:



13. **T** Let u be a discrete-time signal given by

$$\forall\, n \in \textit{Ints}, \quad u(n) = \begin{cases} 1 & 0 \le n \\ 0 & \text{otherwise} \end{cases}$$

This is the **unit step** signal, which we saw before in (2.15). Suppose that a discrete-time system H that is known to be LTI is such that if the input is u, the output is $y = H(u)$ given by

$$\forall\, n \in \textit{Ints}, \quad y(n) = nu(n).$$

This is called the **step response** of the system. Find a simple expression for the output $w = H(p)$ when the input is p given by

$$\forall\, n \in \textit{Ints}, \quad p(n) = \begin{cases} 2 & 0 \le n < 8 \\ 0 & \text{otherwise} \end{cases}$$

Sketch w.

14. **T** Suppose you are given the Fourier series coefficients $\cdots X_{-1}, X_0, X_1, X_2, \cdots$ for a periodic signal $x\colon \textit{Reals} \to \textit{Reals}$ with period p. Find the fundamental frequency and the Fourier series coefficients of the following signals in terms of those of x.

    (a) y such that $\forall\, t \in \textit{Reals}, \;\; y(t) = x(at)$, for some positive real number a.

    (b) w such that $\forall\, t \in \textit{Reals}, \;\; w(t) = x(t)e^{i\omega_0 t}$, where $\omega_0 = 2\pi/p$.

    (c) z such that $\forall\, t \in \textit{Reals}, \;\; z(t) = x(t)\cos(\omega_0 t)$, where $\omega_0 = 2\pi/p$.

15. Analogously to the box on page 194, show that the formula (7.14) for the discrete Fourier series coefficients is valid.

16. **C** Consider a system *Squarer*: $[\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}]$, where if $y = \textit{Squarer}(x)$ then

$$\forall\, t \in \textit{Reals}, \quad y(t) = x^2(t).$$

    (a) Show that this system is memoryless.

    (b) Show that this system is not linear.

    (c) Show that this system is time invariant.

    (d) Suppose that the input x is given by

$$\forall\, t \in \textit{Reals}, \quad x(t) = cos(\omega t),$$

    for some fixed ω. Show that the output y contains a component at frequency 2ω.

Chapter 8

Filtering

Linear time invariant systems have the property that if the input is described as a sum of sinusoids, then the output is a sum of sinusoids of the same frequency. Each sinusoidal component will typically be scaled differently, and each will be subjected to a phase change, but the output will not contain any sinusoidal components that are not also present in the input. For this reason, an LTI system is often called a **filter**. It can filter out frequency components of the input, and also enhance other components, but it cannot introduce components that are not already present in the input. It merely changes the relative amplitudes and phases of the frequency components that are present in the inputs.

LTI systems arise in two circumstances in an engineering context. First, they may be used as a model of a physical system. Many physical systems are accurately modeled as LTI systems. Second, they may present an ideal for an engineered system. For example, they may specify the behavior that an electronic system is expected to exhibit.

Consider for example an audio system. The human ear hears frequencies in the range of about 30 to 20,000 Hz, so a specification for a high fidelity audio system typically requires that the frequency response be constant (in magnitude) over this range. The human ear is relatively insensitive to phase, so the same specification may say nothing about the phase response (the argument, or angle of the frequency response). An audio system is free to filter out frequencies outside this range.

Consider an acoustic environment, a physical context such as a lecture hall where sounds are heard. The hall itself alters the sound. The sound heard by your ear is not identical to the sound created by the lecturer. The room introduces echoes, caused by reflections of the sound by the walls. These echoes tend to occur more for the lower frequency components in the sound than the higher frequency components because the walls and objects in the room tend to absorb higher frequency sounds better. Thus, the lower frequency sounds bounce around in the room, reinforcing each other, while the higher frequency sounds, which are more quickly absorbed, become relatively less pronounced. In an extreme circumstance, in a room where the walls a lined with shag carpeting, for example, the higher frequency sounds are absorbed so effectively that the sound gets muffled by the room.

The room can be modeled by an LTI system where the frequency response $H(\omega)$ is smaller in magnitude for large ω than for small ω. This is a simple form of **distortion** introduced by a **channel** (the room), which in this case carries a sound from its transmitter (the lecturer) to its receiver (the listener). This form of distortion is called **linear distortion**, a shorthand for linear, time-invariant distortion (the time invariance is left implicit).

A public address system in a lecture hall may compensate for the acoustics of the room by boosting the high frequency content in the sound. Such a compensator is called an **equalizer** because it corrects for distortion in the channel so that all frequencies are received equally well by the receiver.

In a communications system, a channel may be a physical medium, such as a pair of wires, that carries an electrical signal. That physical medium distorts the signal, and this distortion is often reasonably well approximated as linear and time invariant. An equalizer in the receiver compensates for this distortion. Unlike the audio example, however, such an equalizer often needs to compensate for the phase response, not just the magnitude response. Because the human ear is relatively insensitive to phase distortion, a public address system equalizer need not compensate for phase distortion. But the wire pair may be carrying a signal that is not an audio signal. It may be, for example, a modem signal.

Images may also be processed by LTI systems. Consider the three images shown in figure8.1. The top image is the original, undistorted image. The lower left image is blurred, as might result for example from unfocused optics. The lower right image is, in a sense, the opposite of the blurred image. Whereas the blurred image deemphasizes the patterns in the outfit, for example, the right image deemphasizes the regions of constant value, changing them all to a neutral gray.

For images, time is not the critical variable. Its role is replaced by two spatial variables, one in the horizontal direction and one in the vertical direction. Thus, instead of LTI, we might talk about an image processing system being a **linear, space-invariant** (**LSI**) system. The blurred image is constructed from the original by an LSI system that eliminates high (spatial) frequencies, passing unaltered the low frequencies. Such a system is called a **lowpass** filter. The lower right image is constructed from the original by an LSI system that eliminates low frequencies, passing unaltered the high frequencies. Such a system is called a **highpass** system. Both images were created using Adobe Photoshop, although the blurred image could have been just as easily created by a defocused lens.

8.1 Convolution

The frequency response of a system is a declarative description of the system. It tells us what it is, not how it works. It tells us, for example, that it is a lowpass filter, but it does not tell us whether it is a defocused lens or a computer program, much less telling us how the computer program works. In this chapter, we explore imperative descriptions of systems, and build up to detailed descriptions of software that can implement certain kinds of LTI (or LSI) systems. These imperative descriptions are based on **convolution**.

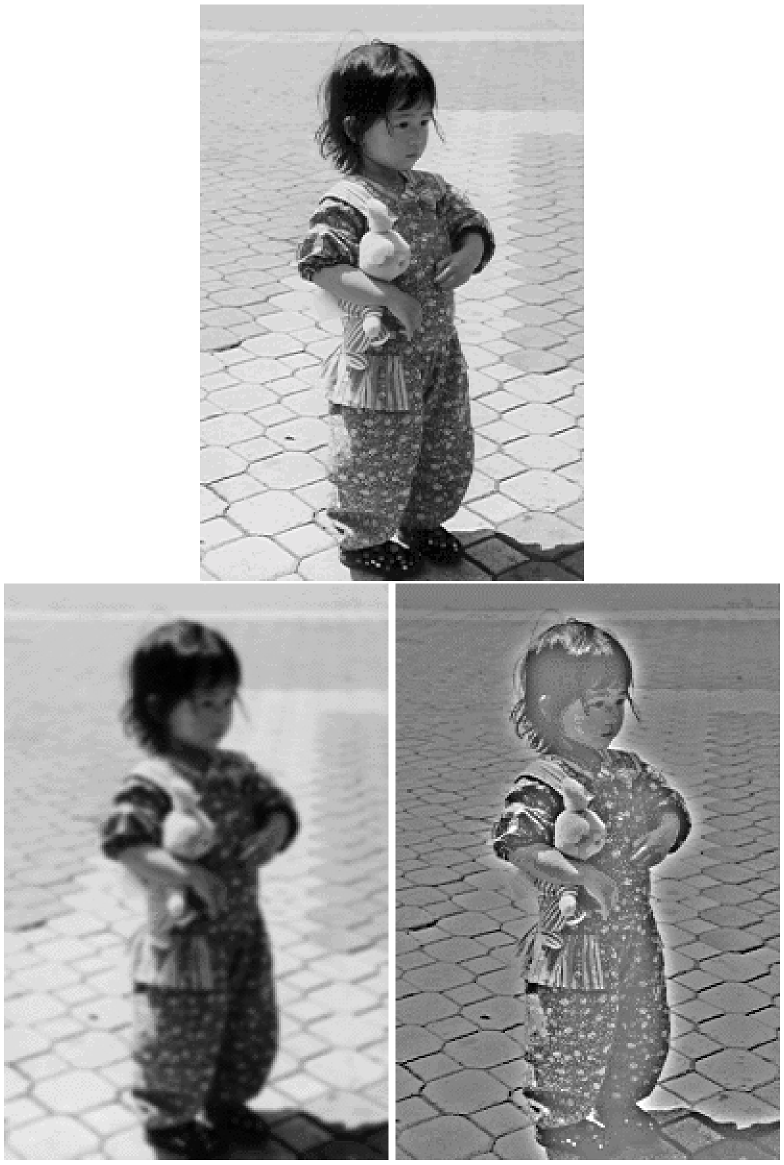


Figure 8.1: An image and two versions that have been distorted by a linear, space-invariant system.

8.1.1 Convolution sum and integral

For discrete-time signals the convolution operator is called the convolution sum, and for continuous-time signals it is called the convolution integral. We define these two operators now and note some important properties.

Let $x, y \in [\mathit{Ints} \to \mathit{Reals}]$ be two discrete-time signals. The **convolution sum** of x and y is the discrete-time signal, denoted $x * y$, given by

$$\boxed{\forall n \in \mathit{Ints}, \quad (x * y)(n) = \sum_{k=-\infty}^{\infty} x(k)y(n-k).} \tag{8.1}$$

We note two properties. First, the order in the convolution sum does not matter, $x * y = y * x$. Indeed, if in (8.1) we change the variables in the summation, letting $m = n - k$, we get

$$\begin{aligned}\forall n \in \mathit{Ints}, \quad (x * y)(n) &= \sum_{k=-\infty}^{\infty} x(k)y(n-k) \\ &= \sum_{m=-\infty}^{\infty} x(n-m)y(m).\end{aligned}$$

Thus,

$$\boxed{(x * y)(n) = (y * x)(n).} \tag{8.2}$$

This property is called **commutativity** of the convolution operator.[1]

Another property of convolution is **linearity**. That is, if x, y_1, y_2 are three signals and a_1 and a_2 are real numbers, then

$$\boxed{x * (a_1 y_1 + a_2 y_2) = a_1(x * y_1) + a_2(x * y_2),} \tag{8.3}$$

which may be checked directly by applying definition (8.1) to both sides.

We now use the convolution sum to define some LTI systems. Fix a discrete-time signal h, and define the system

$$S \colon [\mathit{Ints} \to \mathit{Reals}] \to [\mathit{Ints} \to \mathit{Reals}]$$

by

$$\forall\, x \in [\mathit{Ints} \to \mathit{Reals}], \quad S(x) = h * x.$$

Thus the output signal $y = S(x)$ corresponding to the input signal x is given by

$$\forall\, n \in \mathit{Ints}, \quad y(n) = \sum_{k=-\infty}^{\infty} h(k)x(n-k).$$

[1]Matrix multiplication is an example of an operator that is not commutative, while matrix addition is. Since the sum of two matrices M and N (of the same size) does not depend on the order, i.e., $M + N = N + M$, the matrix sum is a commutative operator. However, the product of two matrices depends on the order, i.e., it is not always true that $M \times N = N \times M$, so the matrix product is not commutative.

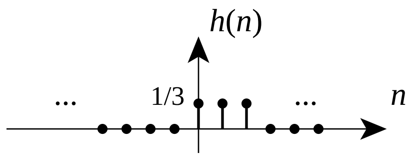


Figure 8.2: Signal in example 8.1.

We now show that S is LTI. The linearity of S follows immediately from the linearity property of the convolution. To show time-invariance, we must show that for any integer τ, and any input signal x,

$$D_\tau(h * x) = h * (D_\tau(x)),$$

where D_τ is a delay by τ. But this is easy to see, since for all n,

$$\begin{aligned}(D_\tau(h * x))(n) &= (h * x)(n - \tau) \\ &= \sum_{k=-\infty}^{\infty} h(k)x(n - \tau - k), \text{ by definition (8.1)} \\ &= \sum_{k=-\infty}^{\infty} h(k)z(n - k), \text{ where } z = D_\tau(x) \\ &= (h * z)(n).\end{aligned}$$

Thus every discrete-time signal h defines an LTI system via convolution. In the next section we will see the converse result, that every LTI system is defined by convolution with some signal h.

Example 8.1: Consider a discrete-time signal h defined by

$$\forall\, n \in \textit{Ints}, \quad h(n) = \begin{cases} 1/3 & \text{if } n \in \{0, 1, 2\} \\ 0 & \text{otherwise} \end{cases}$$

This is shown in figure 8.2. Let us define a system S as follows. If the input is x, then the output is

$$y = S(x) = h * x.$$

I.e,

$$\begin{aligned}\forall\, n \in \textit{Ints}, \quad y(n) &= \sum_{k=-\infty}^{\infty} h(k)x(n - k) \\ &= \sum_{k=0}^{2} (1/3)x(n - k) \\ &= (x(n) + x(n - 1) + x(n - 2))/3. \qquad (8.4)\end{aligned}$$

This system calculates the three-point moving average!

We now turn to the continuous time case. Let $x, y \in [\mathit{Reals} \to \mathit{Reals}]$ be two continuous-time signals. The **convolution integral** of x and y is the continuous-time signal, denoted $x * y$, given by

$$\boxed{\forall\, t \in \mathit{Reals}, \quad (x * y)(t) = \int\limits_{-\infty}^{\infty} x(\tau) y(t - \tau) d\tau.} \tag{8.5}$$

By a change of variable in the integral we can check that the convolution integral is commutative, i.e.,

$$\boxed{\forall\, t \in \mathit{Reals}, \quad (x * y)(t) = (y * x)(t),} \tag{8.6}$$

and it is linear; i.e. if x, y_1, y_2 are three continuous-time signals and a_1, a_2 are real numbers, then

$$\boxed{x * (a_1 y_1 + a_2 y_2) = a_1 (x * y_1) + a_2 (x * y_2).} \tag{8.7}$$

Again fix $h \in [\mathit{Reals} \to \mathit{Reals}]$, and define the system

$$S \colon [\mathit{Reals} \to \mathit{Reals}] \to [\mathit{Reals} \to \mathit{Reals}]$$

by

$$\forall\, x \in [\mathit{Reals} \to \mathit{Reals}], \quad S(x) = h * x.$$

Then in exactly the same way as for the discrete-time case, we can show that S is LTI.

Example 8.2: Consider a continuous-time signal h defined by

$$\forall\, t \in \mathit{Reals}, \quad h(t) = \begin{cases} 1/3 & \text{if } t \in [0, 3] \\ 0 & \text{otherwise} \end{cases}$$

This is shown in figure 8.3. Let us define a system S as follows. If the input is x, then the output is

$$y = S(x) = h * x.$$

I.e,

$$\begin{aligned} \forall\, t \in \mathit{Reals}, \quad y(t) &= \int\limits_{-\infty}^{\infty} h(\tau) x(t - \tau) d\tau \\ &= \frac{1}{3} \int\limits_{0}^{3} x(t - \tau) d\tau. \end{aligned} \tag{8.8}$$

This system is the length-three continuous-time moving average!

Note that we are using the same symbol '*' for the convolution sum and the convolution integral. The context will determine which operator is intended.

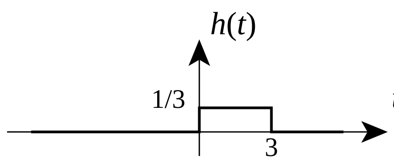


Figure 8.3: Signal in example 8.2.

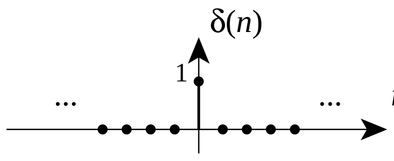


Figure 8.4: The Kronecker delta function, a discrete-time impulse.

8.1.2 Impulses

Intuitively, an impulse is a signal that is zero everywhere except at time zero. In the discrete-time case, the **Kronecker delta function**,

$$\delta\colon \mathit{Ints} \to \mathit{Reals}$$

is defined by

$$\boxed{\forall\, n \in \mathit{Ints}, \quad \delta(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}} \tag{8.9}$$

Its graph is shown in figure 8.4.

The continuous-time case, which is called the **Dirac delta function**, is mathematically much more difficult to work with. Like the Kronecker delta function, it is zero everywhere except at zero. But unlike the Kronecker delta function, its value is infinite at zero. We will not concentrate on its subtleties, but rather just introduce it and assert some results without fully demonstrating their validity. The Dirac delta function is defined to be

$$\delta\colon \mathit{Reals} \to \mathit{Reals}_{++}$$

where $\mathit{Reals}_{++} = \mathit{Reals} \cup \{\infty, -\infty\}$, and

$$\forall\, t \in \mathit{Reals} \text{ where } t \neq 0, \quad \delta(t) = 0$$

and where the following property is satisfied for any $\epsilon > 0$ in Reals_{++},

$$\int\limits_{-\epsilon}^{\epsilon} \delta(t)dt = 1$$

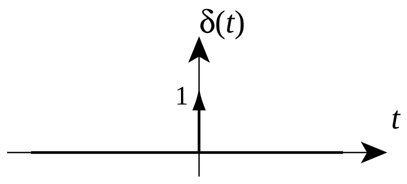


Figure 8.5: The Dirac delta function, a continuous-time impulse.

For the latter property to be satisfied, clearly no finite value at $t = 0$ would suffice. This is why the value must be infinite at $t = 0$. Notice that the Kronecker delta function has a similar property,

$$\sum_{n=-a}^{a} \delta(n) = 1$$

for any integer $a > 0$, but that in this case, the property is trivial. There is no mathematical subtlety.

The Dirac delta function is usually depicted as in figure 8.5. In any figure, of course, the infinite value at $t = 0$ cannot be shown directly. The figure suggests that infinite value with a vertical arrow. Next to the arrow is the **weight** of the delta function, '1' in this case. In general, a Dirac delta function can be multiplied by any real constant a. Of course, this does not change its value at $t = 0$, which is infinite, nor does it change its value at $t \neq 0$, which is zero. What it does change is its integral,

$$\int_{-\epsilon}^{\epsilon} a\delta(t)dt = a.$$

Thus, although the impulse is still infinitely narrow and infinitely high, the area under the impulse has been scaled by a.

8.1.3 Signals as sums of weighted delta functions

Any discrete-time signal x: *Ints* $\to$ *Reals* can be expressed as a sum of weighted Kronecker delta functions,

$$\boxed{\forall\, n \in \textit{Ints}, \quad x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k).} \tag{8.10}$$

The kth term in the sum is $x(k)\delta(n-k)$. This term, by itself, defines a signal that is zero everywhere except at $n = k$, where it has value $x(k)$. This signal is called a **weighted delta function** because it is a (time shifted) delta function with a specified weight. Thus, any discrete-time signal is a sum of weighted delta functions, much the way that the Fourier series describes a signal as a sum of weighted complex exponential functions.

Example 8.3: The signal h in example 8.1 can be written in terms of Kronecker delta functions,

$$\forall\, n \in \textit{Ints}, \quad h(n) = (\delta(n) + \delta(n-1) + \delta(n-2))/3.$$

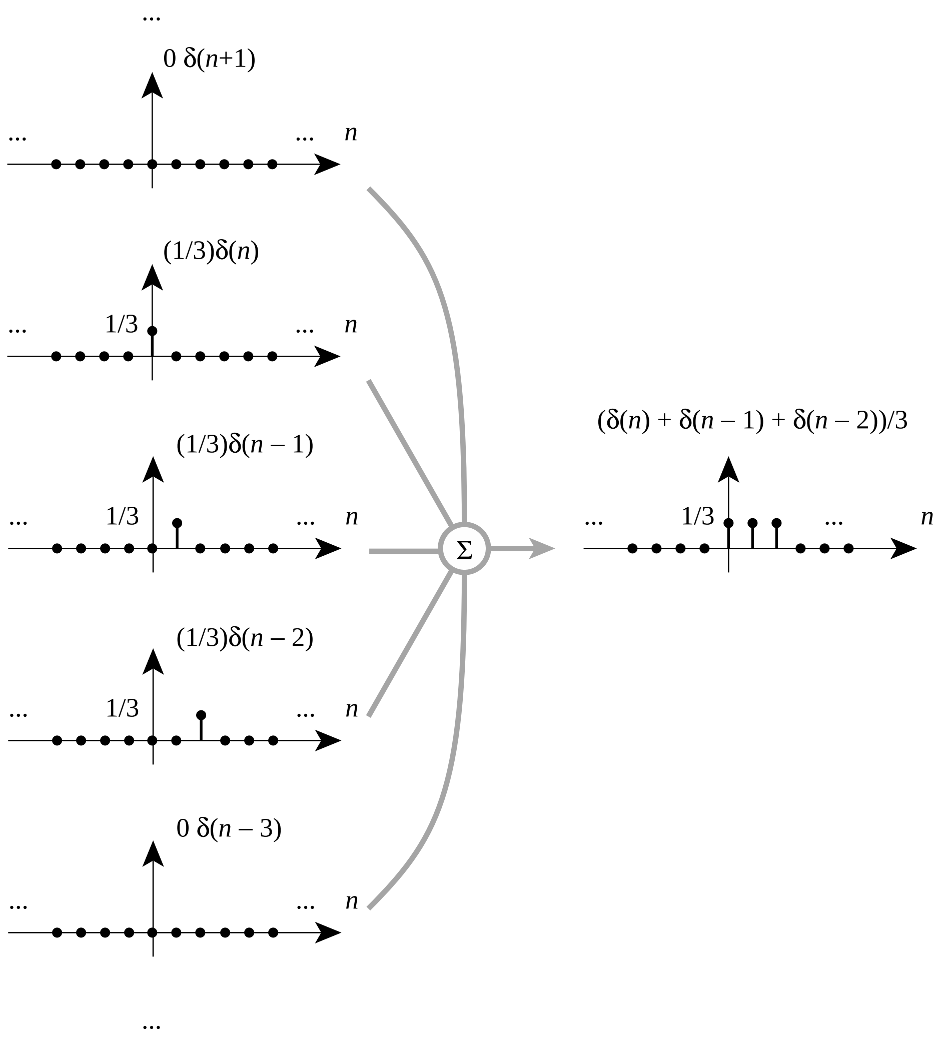


Figure 8.6: A discrete-time signal is a sum of weighted delta functions.

This has the form of (8.10), and is illustrated in figure 8.6. It is described as a sum of signals where each signal contains only a single weighted impulse.

Equation (8.10) is sometimes called the **sifting property** of the Kronecker delta function because it "sifts out" the value of a function x at some integer n. That is, the infinite sum reduces to a single number. This property can often be used to eliminate infinite summations in more complicated expressions.

The continuous-time version of this is similar, except that the sum becomes an integral (integration, after all, is just sum over a continuum). Given any signal $x\colon \mathit{Reals} \to \mathit{Reals}$,

$$\boxed{\forall\, t \in \mathit{Reals}, \quad x(t) = \int\limits_{-\infty}^{\infty} x(\tau)\delta(t-\tau)d\tau.} \tag{8.11}$$

Although this is mathematically much more subtle than the discrete-time case, it is very similar in structure. It describes a signal x as a sum (or more precisely, an integral) of weighted Dirac delta functions.

Example 8.4: The signal h in figure 8.3 and example 8.2 can be written as a sum (an integral, actually) of weighted Dirac delta functions,

$$\begin{aligned} \forall\, t \in \mathit{Reals}, \quad h(t) &= \int_{-\infty}^{\infty} h(\tau)\delta(t-\tau)d\tau \\ &= \int_{0}^{3} (1/3)\delta(t-\tau)d\tau. \end{aligned}$$

This has the form of (8.11).

Equation (8.11) is sometimes called the **sifting property** of the Dirac delta function, because it sifts out from the function x the value at a given time t. The sifting property can often be used to eliminate integrals, since it replaces an integral with a single value.

8.1.4 Impulse response and convolution

Consider a discrete-time LTI system $S\colon [\mathit{Ints} \to \mathit{Reals}] \to [\mathit{Ints} \to \mathit{Reals}]$. Define its **impulse response** h to be the output signal when the input signal is the Kronecker delta function (an impulse), $h = S(\delta)$, that is,

$$\forall n, \quad h(n) = (S(\delta))(n).$$

Now let x be any input signal, and let $y = S(x)$ be the corresponding output signal. In (8.10), x is given as sum of components, where each component is a weighted delta function. Since S is LTI, the output can be given as a sum of the responses to these components. Each component is a signal $x(k)\delta(n-k)$ for fixed k, and the response to this signal is $x(k)h(n-k)$. The response to a scaled and delayed impulse will the a scaled and delayed impulse response. Thus, the overall output is

$$\boxed{y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) = (x * h)(n) = (h * x)(n).}$$

Thus, the output of any discrete-time LTI system is given by the convolution of the input signal and the impulse response.

> **Example 8.5:** The three-point moving average system S of example 8.1 has impulse response
>
> $$\forall\, n \in \textit{Ints}, \quad h(n) = (\delta(n) + \delta(n-1) + \delta(n-2))/3.$$
>
> This can be determined from (8.4) by just letting the input be $x = \delta$. The impulse response, after all, is defined to be the output when the input is an impulse. The impulse response is shown in figure 8.2.

Consider now a continuous-time LTI system $S\colon [\textit{Reals} \to \textit{Reals}] \to [\textit{Reals} \to \textit{Reals}]$. Define its **impulse response** to be the output signal h when the input signal is the Dirac delta function, $h = S(\delta)$, i.e.,

$$\forall t \in \textit{Reals}, \quad h(t) = S(\delta)(t).$$

Now let x be any input signal and let $y = S(x)$, be the corresponding output signal. By the sifting property we can express x as the sum (integral) of weighted delta functions,

$$x(t) = \int_{-\infty}^{\infty} x(\tau)\delta(t-\tau)d\tau.$$

Since S is LTI, the output is a sum (integral) of the responses to each of the components (the integrand for fixed τ), or

$$y(t) = \int\limits_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau = (x * h)(t) = (h * x)(t). \qquad (8.12)$$

Thus,

> The output of any continuous-time LTI system is given by the convolution of the input signal and the impulse response.

> **Example 8.6:** The length-three continuous-time moving average system S of 8.2 has impulse response
>
> $$\forall\, t \in \textit{Reals}, \quad h(t) = \begin{cases} 1/3 & \text{if } t \in [0,3] \\ 0 & \text{otherwise} \end{cases}$$
>
> This can be determined from (8.8) by just letting the input be $x = \delta$ and then using the sifting property of the delta function. The impulse response, after all, is defined to be the output when the input is an impulse. The impulse response is shown in figure 8.3. In general, a moving average has an impulse response with this rectangular shape.

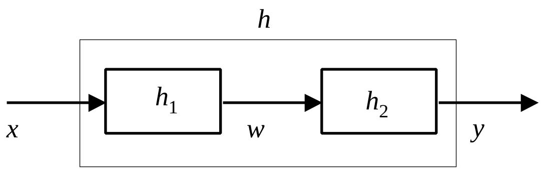


Figure 8.7: A cascade combination of two discrete-time LTI systems.

Example 8.7: Consider an N-step discrete-time delay system, as in example 7.2, where the output y is given in terms of the input x by the difference equation

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n - N). \tag{8.13}$$

The impulse response can be found by letting $x = \delta$, to get

$$h(n) = \delta(n - N).$$

The output can be given as a convolution of the input and the impulse response,

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n - N - k) = x(n - N),$$

using the sifting property. Of course, this agrees with (8.13).

Example 8.8: Consider an T second continuous-time delay system, where the output y is given in terms of the input x by the equation

$$\forall\, t \in \textit{Reals}, \quad y(t) = x(t - T). \tag{8.14}$$

The impulse response can be found by letting $x = \delta$, to get

$$h(t) = \delta(t - T).$$

The output can be given as a convolution of the input and the impulse response,

$$y(t) = \int_{-\infty}^{\infty} x(\tau)\delta(t - T - \tau)d\tau = x(t - T),$$

using the sifting property. Of course, this agrees with (8.14).

Example 8.9: Suppose that we have two LTI systems (discrete or continuous time) with impulse responses h_1 and h_2, and we connect them in a cascade structure as shown in figure 8.7. We can find the impulse response of the cascade composition by letting the input be an impulse, $x = \delta$. Then the output of the first system will be its impulse response, $w = h_1$. This provides the input to the second system, so its output will be $y = h_1 * h_2$. Thus, the overall impulse response is the convolution of the two impulse responses,

$$h = h_1 * h_2.$$

We also know from the previous chapter that the frequency responses relate in a very simple way, namely

$$H(\omega) = H_1(\omega)H_2(\omega).$$

We will find that, in general, convolution in the time domain is equivalent to multiplication in the frequency domain.

8.2 Frequency response and impulse response

If a discrete-time LTI system has impulse response h, then the output signal y corresponding to the input signal x is given by the convolution sum,

$$\forall n, \quad y(n) = \sum_{m=-\infty}^{\infty} h(m)x(n-m).$$

In particular, suppose the input is the complex exponential function

$$\forall n \in \textit{Ints}, \quad x(n) = e^{i\omega n},$$

for some real ω. Then the output signal is

$$y(n) = \sum_{m=-\infty}^{\infty} h(m)e^{i\omega(n-m)} = e^{i\omega n} \sum_{m=-\infty}^{\infty} h(m)e^{-i\omega m}.$$

Recall further that when the input is a complex exponential with frequency ω, then the output is given by

$$y(n) = H(\omega)e^{i\omega n}$$

where $H(\omega)$ is the frequency response. Comparing these two expressions for the output we see that the frequency response is related to the impulse response by

$$\boxed{\forall \omega \in \textit{Reals}, \quad H(\omega) = \sum_{m=-\infty}^{\infty} h(m)e^{-i\omega m}.} \tag{8.15}$$

This expression allows us, in principle, to calculate the frequency response from the impulse response. Equation (8.15) gives us a way to transform h, a time-domain function, into H, a frequency domain function. Equation (8.15) is called a **discrete-time Fourier transform** (**DTFT**). Equivalently, we say that H is the DTFT of h. So the frequency response of a discrete-time system is the DTFT of its impulse response.

Example 8.10: Consider the N-step delay from example 8.7. Its impulse response is

$$h(n) = \delta(n - N).$$

We can find the frequency response by calculating the DTFT,

$$\begin{aligned} H(\omega) &= \sum_{m=-\infty}^{\infty} h(m)e^{-i\omega m} \\ &= \sum_{m=-\infty}^{\infty} \delta(m-N)e^{-i\omega m} \\ &= e^{-i\omega N} \end{aligned}$$

where the last step follows from the sifting property. This same result was obtained more directly in example 7.2. Note that the magnitude response is particularly simple,

$$|H(\omega)| = 1.$$

This is intuitive. An N-step delay does not change the magnitude of any complex exponential input. It only shifts its phase.

Notice from (8.15) that

$$\boxed{\text{If } h \text{ is real-valued then } H^*(-\omega) = H(\omega)} \tag{8.16}$$

(just conjugate both sides of (8.15) and evaluate at $-\omega$). This property is called **conjugate symmetry**. It implies

$$|H(-\omega)| = |H(\omega)|. \tag{8.17}$$

This says that for any LTI system with a real-valued impulse response, a complex exponential with frequency ω experiences the same amplitude change as a complex exponential with frequency $-\omega$.

Notice further from (8.15) that

$$\boxed{\forall\, \omega \in \mathit{Reals}, \quad H(\omega + 2\pi) = H(\omega).} \tag{8.18}$$

I.e., the DTFT is periodic with period 2π. This says that a complex exponential with frequency ω experiences the same amplitude and phase change as a complex exponential with frequency $\omega + 2\pi$. This should not be surprising since the two complex exponentials are in fact identical,

$$e^{i(\omega+2\pi)n} = e^{i\omega n}e^{i2\pi n} = e^{i\omega n},$$

because $e^{i2\pi n} = 1$ for any integer n.

The continuous-time version proceeds in the same way. Let S be a continuous-time system with impulse response h. Then the output signal y corresponding to an input signal x is given by

$$\forall\, t \in \mathit{Reals}, \quad y(t) = \int_{-\infty}^{\infty} x(t-\tau)h(\tau)d\tau.$$

In particular, if the input signal is the complex exponential,

$$\forall\, t \in \textit{Reals}, \quad x(t) = e^{i\omega t},$$

then the output signal is

$$y(t) = \int_{-\infty}^{\infty} e^{i\omega(t-\tau)} h(\tau) d\tau = e^{i\omega t} \int_{-\infty}^{\infty} e^{-i\tau} h(\tau) d\tau.$$

The output is also given by $y(t) = H(\omega)e^{i\omega t}$ where $H(\omega)$ is the frequency response, and so we have

$$\boxed{H(\omega) = \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt.} \tag{8.19}$$

So, given its impulse response, we can calculate the frequency response of a continuous-time LTI system by evaluating the integral (8.19). Like the DTFT, this integral transforms a time-domain signal h into a frequency-domain signal H. It is called the **continuous-time Fourier transform** (**CTFT**), or more commonly, simply the **Fourier transform** (**FT**). Thus the frequency response H of a continuous-time LTI system is just the CTFT of its impulse response h.

Example 8.11: Consider the T second delay from example 8.8. Its impulse response is

$$h(t) = \delta(t - T).$$

We can find the frequency response by calculating the CTFT,

$$\begin{aligned} H(\omega) &= \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt \\ &= \int_{-\infty}^{\infty} \delta(t - T) e^{-i\omega t} dt \\ &= e^{-i\omega T} \end{aligned}$$

where the last step follows from the sifting property. Note that the magnitude response is particularly simple,

$$|H(\omega)| = 1.$$

This is intuitive. A T second delay does not change the magnitude of any complex exponential input. It only shifts its phase.

Notice from (8.19) that the CTFT is also conjugate symmetric if h is real,

$$H^*(-\omega) = H(\omega) \tag{8.20}$$

$$|H(-\omega)| = |H(\omega)|. \tag{8.21}$$

8.3 Causality

A system is **causal** if the output value at a particular time depends only on the input values at that time or before. For LTI systems, if we examine the convolution sum,

$$y(n) = \sum_{m=-\infty}^{\infty} h(m)x(n-m),$$

for a causal system it must be true that $h(m) = 0$ for all $m < 0$. Were this not true, there would be non-zero terms in the sum with $m < 0$, and those terms would involve a future sample of the input, $x(n-m)$. Conversely, if $h(m) = 0$ for all $m < 0$, then the system is causal since the sum for $y(n)$ will involve only previous input values, $x(n), x(n-1), x(n-2), \cdots$.

Causality is an important practical property of a system that receives its data in **real time** (physical time). Such systems cannot possibly look ahead in time, at least not until someone invents a time machine. However, there are many situations where causality is irrelevant. A system that processes stored data, such as digital audio off a compact disk or audio files in a computer, has no difficulty looking ahead in "time."

8.4 Finite impulse response (FIR) filters

Consider an LTI system S: [*Ints* → *Reals*] → [*Ints* → *Reals*] with impulse response h: *Ints* → *Reals* that has the properties

$$h(n) = 0 \text{ if } n < 0, \quad \text{and} \quad h(n) = 0 \text{ if } n \geq L,$$

where L is some positive integer. Such a system is called a **finite impulse response** (**FIR**) system because the interesting part (the non-zero part) of the impulse response is finite in extent. Because of that property, the convolution sum becomes a finite sum,

$$y(n) = \sum_{m=-\infty}^{\infty} x(n-m)h(m) = \sum_{m=0}^{L-1} x(n-m)h(m). \tag{8.22}$$

L is the length of the impulse response.

This sum, since it is finite, is convenient to work with. It can be used to define a procedure for computing the output of an FIR system given its input. This makes it easy to implement on a computer. We will see that it is also reasonably straightforward to implement certain **infinite impulse response** (**IIR**) filters on a computer.

A continuous-time finite impulse response could be defined, but in practice, continuous-time systems rarely have finite impulse responses, so there is not as much motivation for doing so. Moreover, even though the convolution integral acquires finite limits, this does not make it any more computable. Computing integrals on a computer is a difficult proposition whether the limits are finite or not.

Probing further: Causality

We can give a much more general and formal definition of causality that does not require a system to be LTI. Consider a system $S\colon [A \to B] \to [A \to B]$ that operates on signals of type $[A \to B]$. Assume A is an ordered set and B is any ordinary set. An ordered set is one with relations "$<$" and "$>$" where for any two elements a and b, one of the following assertions must be true:

$$a = b, \quad a > b, \text{ or } \quad a < b.$$

Examples of ordered sets are *Ints* and *Reals*.

An example of a set that is not an ordered set is *Ints* $\times$ *Ints* where we define the ordering relation "$<$" so that $(a, b) < (c, d)$ if $a < c$ and $b < c$, and we similarly define "$>$". Under these definitions, for the elements (1,2) and (2,1), none of the above assertions is true, so the set is not ordered.

However, we could define the ordering relation "$<$" so that $(a, b) < (c, d)$ if one of the following is true:

$$a < c \text{ or}$$

$$a = c, \text{ and } b < c.$$

The relation "$>$" could be similarly defined. Under these definitions, the set is ordered.

Define a function $\textit{Prefix}_t\colon [A \to B] \to [A_t \to B]$ that extracts a portion of a signal up to $t \in A$. Formally, $A_t \subset A$ such that $a \in A_t$ if $a \in A$ and $a \leq t$. Then for all $x \in [A \to B]$ and for all $t \in A_t$, $(\textit{Prefix}_t(x))(t) = x(t)$.

A system S is causal if for all $t \in A$ and for all $x, y \in [A \to B]$

$$\textit{Prefix}_t(x) = \textit{Prefix}_t(y) \Rightarrow \textit{Prefix}_t(S(x)) = \textit{Prefix}_t(S(y)).$$

The symbol "$\Rightarrow$" reads "implies." In words, the system is causal if when two input signals have the same prefix up to time t, it follows that the two corresponding output signals have the same prefix up to time t.

Example 8.12: We have seen in example 8.1 a 3-point moving average. An L-point moving average system with input x has output y given by

$$y(n) = \frac{1}{L}\sum_{m=0}^{L-1} x(n-m).$$

The output at index n is the average of the most recent L inputs. Such a filter is widely used on Wall Street to try to detect trends in stock market prices. We can study its properties. First, it is easy to see that it is an LTI system. In fact, it is an FIR system with impulse response

$$h(n) = \begin{cases} 1/L & \text{if } 0 \le n < L \\ 0 & \text{otherwise} \end{cases}$$

To see this, just let $x = \delta$. The frequency response is the DTFT of this impulse response,

$$\begin{aligned} H(\omega) &= \sum_{m=-\infty}^{\infty} h(m)e^{-i\omega m} \\ &= \frac{1}{L}\sum_{m=0}^{L-1} e^{-i\omega m}. \end{aligned}$$

With the help of the useful identity[2]

$$\boxed{\sum_{m=0}^{L-1} a^m = \frac{1-a^L}{1-a}} \qquad (8.23)$$

we can write the frequency response as

$$H(\omega) = \frac{1}{L}\left(\frac{1-e^{-i\omega L}}{1-e^{-i\omega}}\right)$$

where we have let $a = e^{-i\omega}$. We can plot the magnitude of the frequency response using Matlab as follows (for $L = 4$):[3]

```
L = 4;
omega = [-pi:pi/250:pi];
H = (1/L)*(1-exp(-i*omega*L))./(1-exp(-i*omega));
plot(omega, abs(H));
xlabel('frequency in radians/sample');
```

[2]You can verify the identity by multiplying both sides by $1 - a$.

[3]This issues a "divide by zero" warning, but yields a correct plot nonetheless. The magnitude of the frequency response at $\omega = 0$ is 1, as you can verify using L'Hopital's rule.

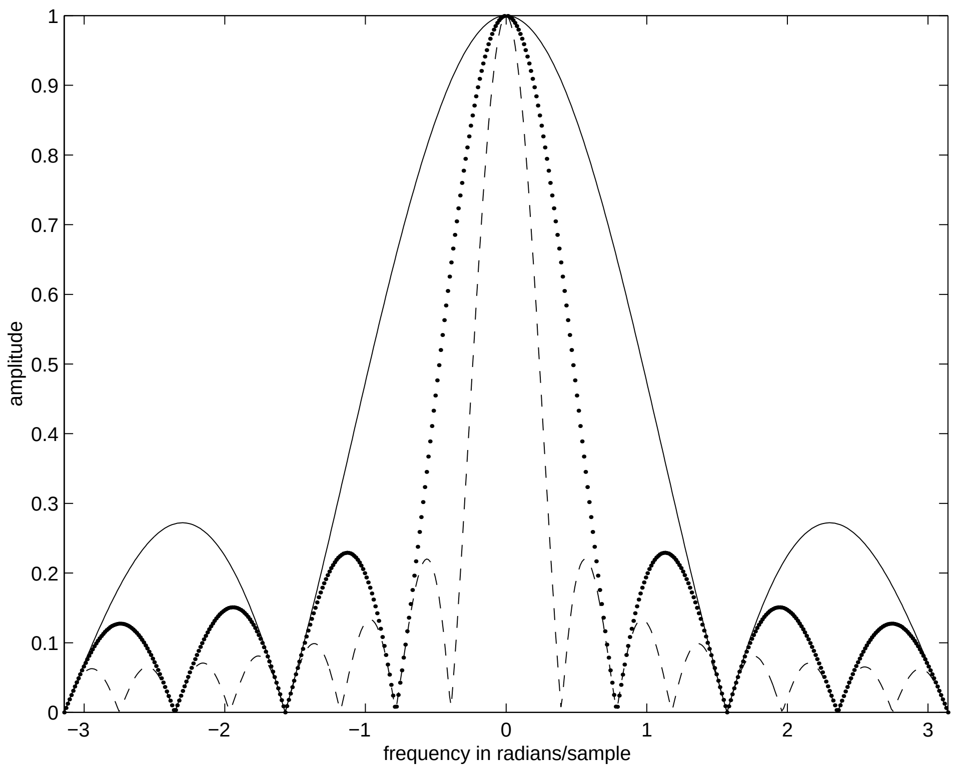


Figure 8.8: The magnitude response of the moving average filter with lengths $L = 4$ (solid line), 8 (dotted line), and 16 (dashed line).

The plot is shown in figure 8.8, together with plots for $L = 8$ and $L = 16$. Notice that the plot shows only the frequency range $-\pi < \omega < \pi$. Since the DTFT is periodic with period 2π, this plot simply repeats outside this range.

Notice that in all three cases shown in figure 8.8, the frequency response has a lowpass characteristic. A constant component (zero frequency) in the input passes through the filter unattenuated. Certain higher frequencies, such as $\pi/2$, are completely eliminated by the filter. However, if the intent was to design a lowpass filter, then we have not done very well. Some of the higher frequencies are attenuated only by a factor of about 1/10 (for the 16 point moving average) or 1/3 (for the four point moving average). We can do better than that with more intelligent filter design.

8.4.1 Design of FIR filters

The moving average system in example 8.12 exhibits a lowpass frequency response, but not a particularly good lowpass frequency response. A great deal of intellectual energy has historically gone into finding ways to choose the impulse response of an FIR filter. The subject is quite deep. Fortunately, much of the work that has been done is readily available in the form of easy-to-use software, so one does not need to be particularly knowledgeable about these techniques to be able to design good FIR filters.

Example 8.13: Matlab's filter design facilities in its DSP toolbox provide some well-studied algorithms for filter design. For example,

```
>> h = remez(7,[0,0.125,0.25,1],[1,1,0,0])

h =

  Columns 1 through 6

    0.0849    0.1712    0.1384    0.1912    0.1912    0.1384

  Columns 7 through 8

    0.1712    0.0849
```

This returns the impulse response of a length 8 FIR lowpass filter. The arguments to the `remez` function specify the filter by outlining approximately the desired frequency response. You can use Matlab's on-line help to get the details, but in brief, the arguments above define a **passband** (a region of frequency where the gain is roughly 1) and a **stopband** (a region of frequency where the gain is roughly 0). The first argument, `7` specifies that the length of the filter should be 8 (you will have to ask The MathWorks why this is off by one). The second argument, `[0,0.125,0.25,1]`, specifies that the first frequency band begins at 0 and extends to 0.125π radians/sample, and that the second band begins at 0.25π and extends to π radians/sample. (The π factor is left off.) The unspecified band, from 0.125π to 0.25π, is a "don't care" region, a **transition band** that allows for a gradual transition between the passband and the stopband.

The last argument, `[1,1,0,0]`, specifies that the first band should have a magnitude frequency response of approximately 1, and that the second band should have a magnitude frequency response of approximately 0.

The frequency response of this filter can be directly calculated and plotted using the following (rather brute force) Matlab code:

```
  omega = [-pi:pi/250:pi];
  H = h(1) + h(2)*exp(-i*omega) + h(3)*exp(-i*2*omega) + ...
      h(4)*exp(-i*3*omega) + h(5)*exp(-i*4*omega) + ...
      h(6)*exp(-i*5*omega) + h(7)*exp(-i*6*omega) + ...
      h(8)*exp(-i*7*omega);
  plot(omega, abs(H));
```

The result is shown in figure 8.9, where it is plotted together with the magnitude response of a moving average filter with the same length. Notice that the attenuation

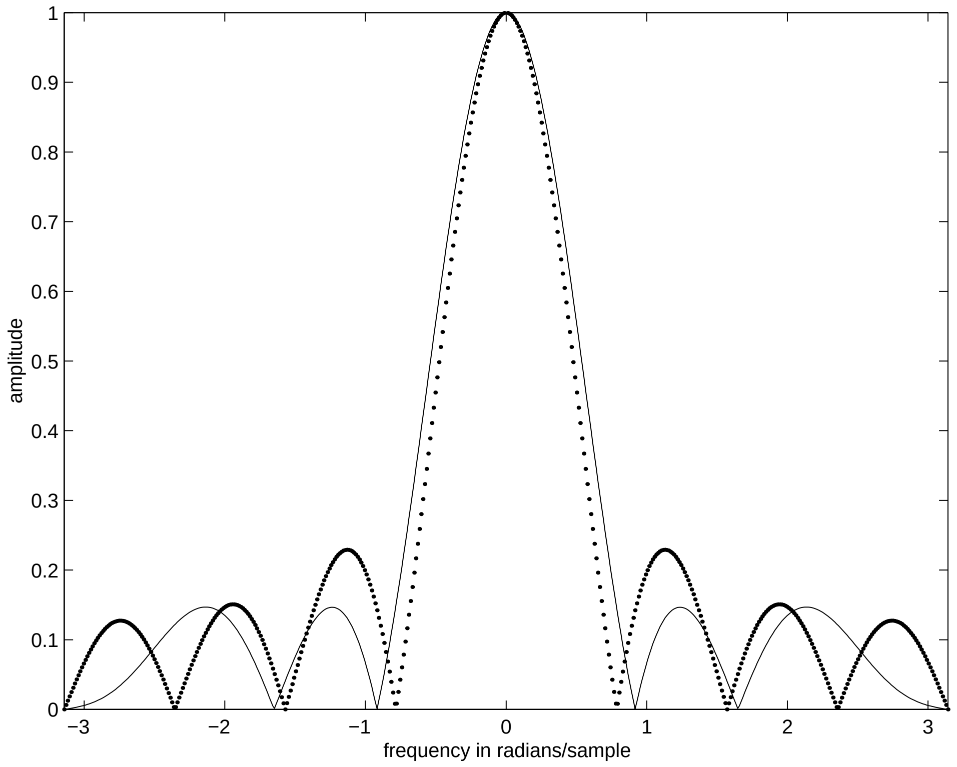


Figure 8.9: Magnitude response an FIR filter designed using the Parks-McClellan algorithm, compared to an ordinary moving average.

in the stopband is better for the filter designed using the `remez` function than for the moving average filter. Also notice that it is not zero, despite our request that it be zero, and that the passband gain is not one, despite our request that it be one.

The `remez` function used in this example applies an optimization algorithm called the Parks-McClellan algorithm, which is based on the Remez exchange algorithm (hence the name of the function). This algorithm ensures that the sidelobes (the humps in the stopband) are of equal size. This turns out to minimize their maximum size. The Parks-McClellan algorithm is widely used for designing FIR filters.

In this example, the `remez` function is unable to deliver a filter that meets our request. The gain in the stopband is not zero, and the gain in the passband is not one. Generally, this is the problem with filter design (and much of the rest of life); we cannot have what we want. We can get closer by using more resources (also as in much of life). In this case, that means designing a filter with a longer impulse response. Since the impulse response is longer, the filter will be more costly to implement. This is because the finite summation in (8.22) will have more terms.

Example 8.14: Consider for example the impulse response returned by

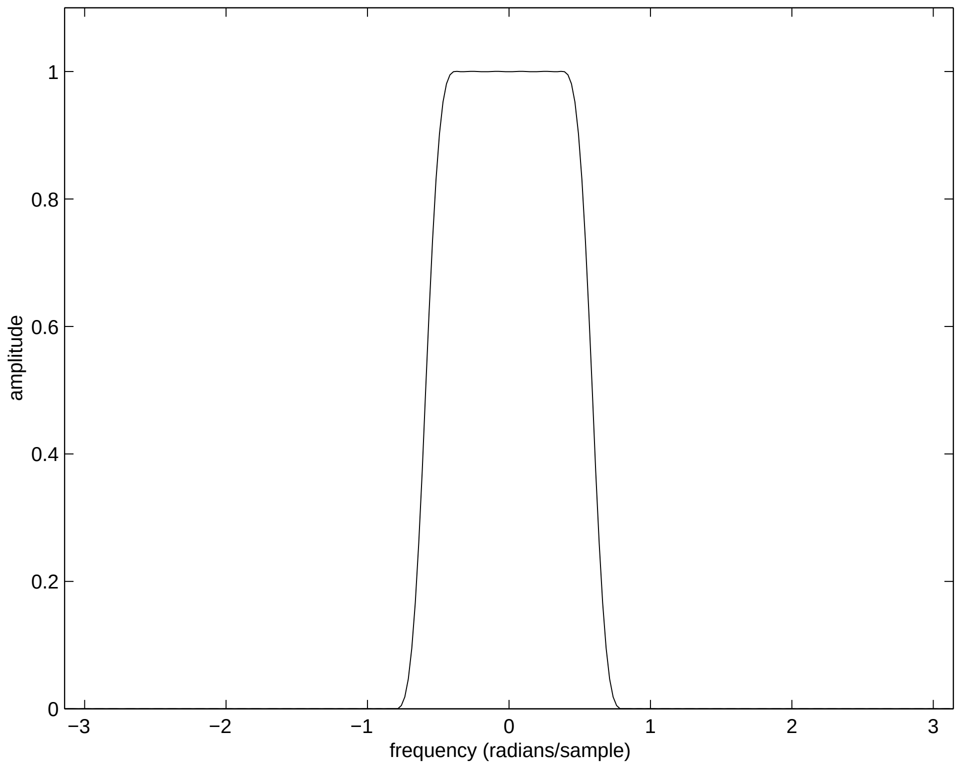


Figure 8.10: Magnitude frequency response of a 64-point lowpass FIR filter designed with the Parks-McClellan algorithm.

```
h = remez(63,[0,0.125,0.25,1],[1,1,0,0]);
```

This has length 64. The magnitude frequency response of this can be calculated using the following Matlab code (see lab C.10 for an explanation):

```
H = fft(h, 1024);
magnitude = abs([H(513:1024),H(1:512)]);
plot([-pi:pi/512:pi-pi/1024], magnitude);
```

This is plotted in figure 8.10.

The frequency response achieved in this example appears in figure 8.10 to exactly match our requirements. However, this plot is somewhat deceptive. It suggests that the magnitude frequency response in the stopband is in fact zero. However, it is not, except at certain discrete frequencies. If you zoom in sufficiently on the plot, you will see that. Instead of zooming such plots, engineers usually construct such plots using a logarithmic vertical axis, as explained in the next subsection.

8.4.2 Decibels

The amplitude response of a filter is the magnitude of the frequency response. It specifies the **gain** experienced by a complex exponential at that frequency. This gain is simply the ratio of the output amplitude to the input amplitude. Since it is the ratio of two amplitudes with same units, the gain itself is unitless.

It is customary in engineering to give gains in a logarithmic scale. Specifically, if the gain of a filter at frequency ω is $|H(\omega)|$, then we give instead

$$G(\omega) = 20\log_{10}(|H(\omega)|). \tag{8.24}$$

This has units of **decibels**, written **dB**. The multiplication by 20 and the use of base 10 logarithms is by convention (see box on page 229).

Example 8.15: The plot in figure 8.10 is redone using the following Matlab commands,

```
H = fft(h, 1024);
dB = 20*log10(abs([H(513:1024),H(1:512)]));
plot([-pi:pi/512:pi-pi/1024], dB);
```

After some adjustment of the axes, the yields the plot in figure 8.11. Notice that in the passband, the gain is 0 dB. This is because $log_{10}(1) = 0$. Zero decibels corresponds to a gain of unity. In the stopband, the gain is almost -70 dB, a very respectable attenuation. If this were an audio signal, then frequency components in the stopband would probably not be audible as long as there is some signal in the passband to mask them. They are 70 dB weaker than the components in the passband, which translates into a factor of 3162 smaller in amplitude, since

$$20\log_{10}(1/3162) \approx -70.$$

We can find the gain given decibels by solving (8.24) for $|H(\omega)|$ in terms of $G(\omega)$ to get

$$|H(\omega)| = 10^{G(\omega)/20}.$$

In this case, in the stopband

$$|H(\omega)| = 10^{-70/20} \approx 1/3162.$$

Notice that $20\log(0) = -\infty$. Thus, whenever the gain is zero, the gain in decibels is $-\infty$. The downward spikes in figure 8.11 occur at frequencies where the gain is zero, or $-\infty$ dB. The spikes do not necessarily precisely show this because the plot does not necessarily evaluate the gain at precisely the frequency that yields zero, hence the ragged bottoms.

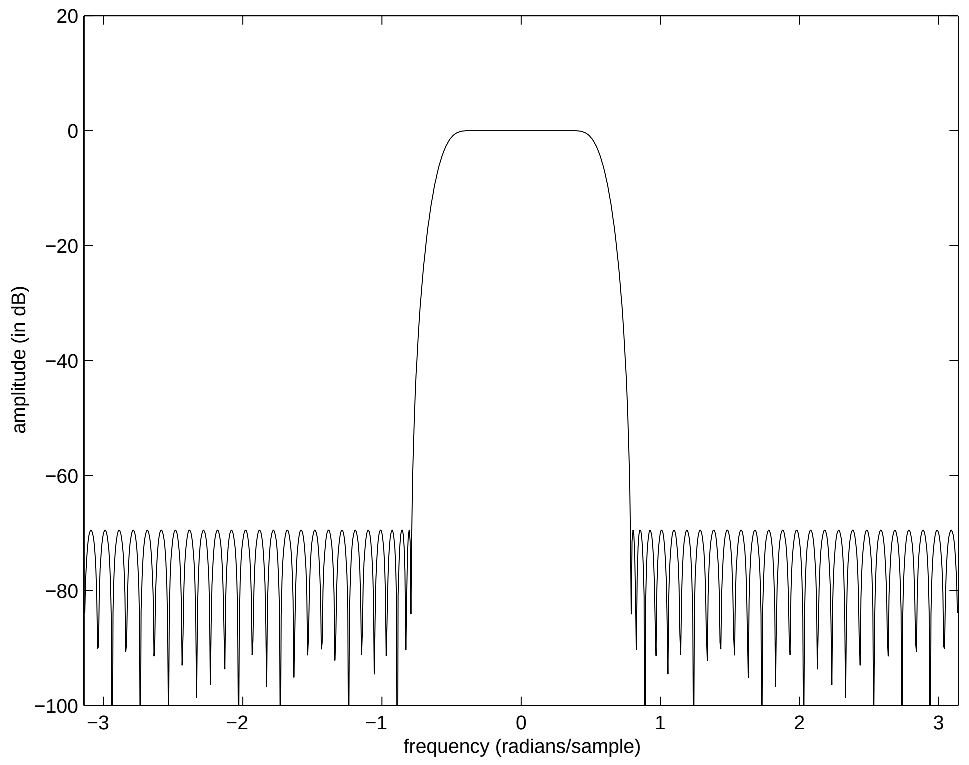


Figure 8.11: Magnitude frequency response in decibels of a 64-point low-pass FIR filter designed with the Parks-McClellan algorithm.

Probing further: Decibels

The term “decibel” is literally one tenth of a **bel**, which is named after Alexander Graham Bell. This unit of measure was originally developed by telephone engineers at Bell Telephone Labs to designate the ratio of the **power** of two signals.

Power is a measure of energy dissipation (work done) per unit time. It is measured in **watts** for electronic systems. One bel is defined to be a factor of 10 in power. Thus, a 1000 watt hair dryer dissipates 1 bel, or 10 dB, more power than a 100 watt light bulb. Let $p_1 = 1000$ watts be the power of the hair dryer and $p_2 = 100$ be the power of the light bulb. Then the ratio is

$$\log_{10}(p_1/p_2) = 1 \text{ bel}.$$

In decibels, this becomes

$$10\log_{10}(p_1/p_2) = 10 \text{ dB}.$$

Comparing against (8.24) we notice a discrepancy. There, the multiplying factor is 20, not 10. That is because the ratio in (8.24) is a ratio of amplitudes, not powers. In electronic circuits, if an amplitude represents the voltage across a resistor, then the power dissipated by the resistor is proportional to the *square* of the amplitude. Let a_1 and a_2 be two such amplitudes. Then the ratio of their powers is

$$10\log_{10}(a_1^2/a_2^2) = 20\log_{10}(a_1/a_2).$$

Hence the multiplying factor of 20 instead of 10.

A 3 dB power ratio amounts to a factor of 2 in power. In amplitudes, this is a ratio of $\sqrt{2}$. The edge of the passband of a bandpass filter is often defined to be the frequency at which the power drops to half, which is the frequency where the gain is 3 dB below the passband gain. The magnitude frequency response here is $1/\sqrt{2}$ times the passband gain.

In audio technology, decibels are used to measure sound pressure. You might hear statements like “a jet engine at 10 meters produces 120 dB of sound.” But decibels measure power ratios, not absolute power, so what does such a statement mean? By convention, sound pressure is measured relative to a defined reference of 20 micropascals, where a pascal is a pressure of 1 newton per square meter. For most people, this is approximately the threshold of hearing at 1 kHz. Thus, a sound at 0 dB is barely audible. A sound at 10 dB has 10 times the power. A sound at 100 dB has 10^{10} times the power. You would not be surprised to learn, therefore, that the jet engine cited above would probably make you deaf without ear protection.

8.5 Infinite impulse response (IIR) filters

The equation for an FIR filter (8.22) is a difference equation relating the output at index n to the inputs at indices $n - L + 1$ through n. A more general form of this difference equation includes more indices of the output,

$$y(n) = \sum_{m=0}^{L-1} x(n-m)b(m) + \sum_{m=1}^{M-1} y(n-m)a(m) \tag{8.25}$$

where L and M are positive integers, and $b(m)$ and $a(m)$ are **filter coefficients**, which are usually real valued. If all the a coefficients are zero, then this reduces to an FIR filter with impulse response $b = h$. However, if any a coefficient is non-zero, then the impulse response of this filter will never completely die out, since the output keeps getting recycled to affect future outputs. Such a filter is called an **infinite impulse response** or **IIR** filter, or sometimes a **recursive filter**.

Example 8.16: Consider a causal LTI system defined by the difference equation

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n) + 0.9y(n-1).$$

We can find the impulse response by letting $x = \delta$, the Kronecker delta function. Since the system is causal, we know that $h(n) = 0$ for $n < 0$ (see page 220). Thus,

$$\begin{array}{rcl} y(n) & = & 0 \quad \text{if } n < 0 \\ y(0) & = & 1 \\ y(1) & = & 0.9 \\ y(2) & = & 0.9^2 \\ y(3) & = & 0.9^3 \end{array}$$

Noticing the pattern, we conclude that

$$y(n) = (0.9)^n u(n),$$

where u is the **unit step**, seen before in (2.15),

$$u(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{8.26}$$

The output y and the magnitude of the frequency response (in dB) are plotted in figure 8.12. Notice that this filter has about 20 dB of gain at DC, dropping to about -6 dB at the Nyquist frequency.

8.5.1 Designing IIR filters

Designing IIR filters amounts to choosing the a and b coefficients for (8.25). As with FIR filters, how to choose these coefficients is a well-studied subject, and the results of this study are (for the

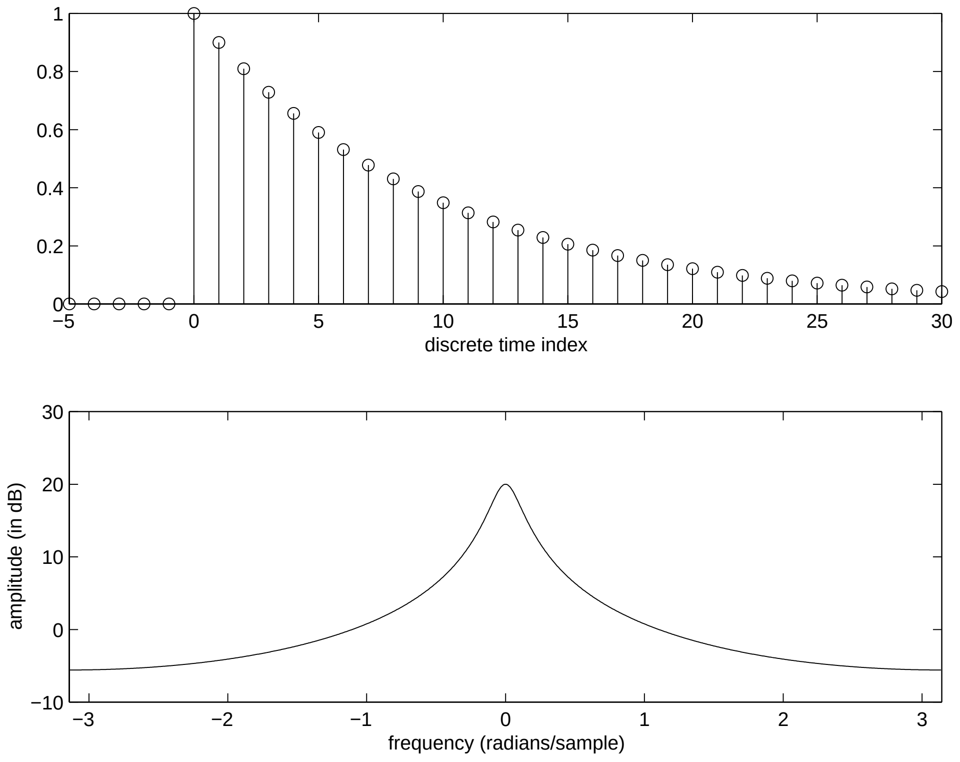


Figure 8.12: Impulse response (top) and magnitude frequency response in dB (bottom) of a simple causal IIR filter.

most part) available in easy-to-use software. There are four widely used methods for calculating these coefficients, resulting in four types of filters called **Butterworth**, **Chebyshev 1**, **Chebyshev 2**, and **elliptic** filters.

Example 8.17: We can design one of each of the four types of filters using the Matlab commands `butter`, `cheby1`, `cheby2`, `ellip`. The arguments for each of these specify either a **cutoff frequency**, which is the frequency at which the filter achieves -3 dB gain, or in the case of `cheby2`, the edge of the stopband. For example, to get lowpass filters with a gain of about 1 from 0 to $\pi/8$ radians/sample, and a stopband at higher frequencies, we can use the following commands:

```
N = 5;
Wn = 0.125;
[B1, A1] = butter(N, Wn);
[B2, A2] = cheby1(N,1,Wn);
[B3, A3] = cheby2(N,70,0.25);
[B4, A4] = ellip(N,1,70,Wn);
```

The returned values are vectors containing the a and b coefficients of (8.25). The magnitude frequency response of the resulting filters is shown in figure8.13. In that figure, we show only positive frequencies, since the magnitude frequency response is symmetric. We also only show the frequency range from 0 to π, since the frequency response is periodic with period 2π. Notice that the Butterworth filter has the most gradual **rolloff** from the passband to the stopband. Thus, for a given filter order, a Butterworth filter yields the weakest lowpass characteristic of the four. The elliptic filter has the sharpest rolloff. The Butterworth filter, on the other hand, has the smoothest characteristic. The Chebyshev 1 filter has **ripple** in the passband, the Chebyshev 2 filter has ripple in the stopband, and the elliptic filter has ripple in both.

In the above Matlab commands, `N` is the filter **order**, equal to L and M in (8.25). It is a constraint of these filter design techniques that $L = M$ in (8.25). `Wn` is the cutoff frequency, as a fraction of π radians/sample. A cutoff frequency of 0.125 means $0.125\pi = \pi/8$ radians/sample. The "1" in the `cheby1` command specifies the amount of passband ripple that we are willing to tolerate (in dB). The 70 in the `cheby2` and `ellip` commands specifies the amount of stopband attenuation that we want (in dB). Finally, the 0.25 in the `cheby2` line specifies the edge of the stopband.

8.6 Implementation of filters

We now have several ways to describe an LTI system (a filter). We can give a state-space description, a frequency response, an impulse response, or a difference equation such as (8.25) or (8.22). All are useful. The state-space description and the difference equations prove the most useful when constructing the filter.

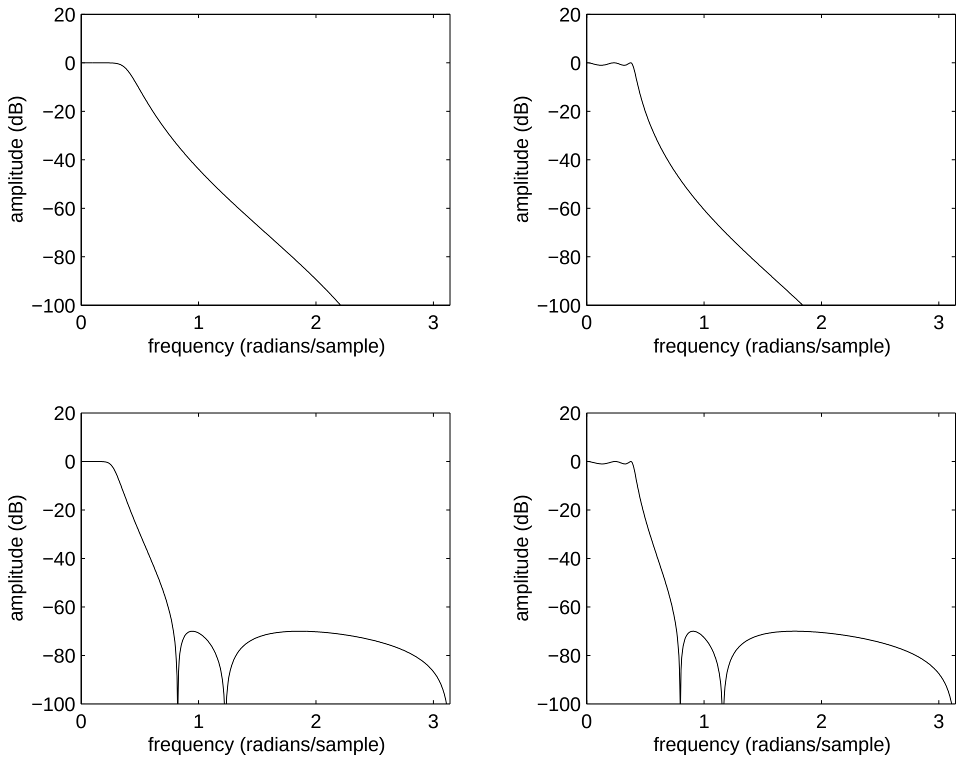


Figure 8.13: Four frequency responses for 5-th order IIR filters of type Butterworth (upper left), Chebyshev 1 (upper right), Chebyshev 2 (lower left), and Elliptic (lower right).

A realization of a filter in hardware or software is called an **implementation**. Do not confuse **filter design** with **filter implementation**. The term “filter design” is used in the community to refer to the choice of frequency response, impulse response, or coefficients for a difference equation, not to how the frequency response is implemented in hardware or software. In this section, we talk about implementation.

The output y of an FIR filter with impulse response h and input x is given by (8.22). The output of an IIR filter with coefficients a and b is given by (8.25). Each of these difference equations defines a procedure for calculating the output given the input. We discuss various ways of implementing these procedures.

8.6.1 Matlab implementation

If x is finite, and we can interpret it as an infinite signal that is zero outside the specified range, then we can compute the output of an FIR filter using Matlab’s `conv` function, and the output of an IIR filter using `filter`.

Example 8.18: Consider an FIR filter with an impulse response of length L. If `x` is a vector containing the P values of the input, and `h` is a vector containing the L values of the impulse response, then

```
y = conv(x, h);
```

yields a vector containing $L + P - 1$ values of the output.

For IIR examples using `filter`, see lab C.10. This strategy, of course, only works for finite input data, since Matlab requires that the input be available in a finite vector.

Discrete-time filters can be implemented using standard programming languages and using assembly languages on specialized processors (see boxes). These implementations do not have the limitation that the input be finite.

8.6.2 Signal flow graphs

We can describe the computations in a discrete-time filter using a block diagram with three visual elements, a unit delay, a multiplier, and an adder. In the convolution sum for an FIR filter,

$$y(n) = \sum_{m=0}^{L-1} h(m)x(n-m),$$

notice that at each n we need access to $x(n)$, $x(n-1)$, $x(n-2)$, $\cdots$, $x(n-L+1)$. We can maintain this set of values by cascading a set of **unit delay** elements to form a **delay line**, as shown in figure 8.14.

Probing further: Java implementation of an FIR filter

The following Java class implements an FIR filter:

```
1  class FIR {
2    private int length;
3    private double[] delayLine;
4    private double[] impResp;
5    private int count = 0;
6    FIR(double[] coefs) {
7      length = coefs.length;
8      impResp = coefs;
9      delayLine = new double[length];
10   }
11   double getOutputSample(double inputSample) {
12     delayLine[count] = inputSample;
13     double result = 0.0;
14     int index = count;
15     for (int i=0; i<length; i++) {
16       result += impResp[i] * delayLine[index--];
17       if (index < 0) index = length-1;
18     }
19     if (++count >= length) count = 0;
20     return result;
21   }
22 }
```

A class in Java (and in any object-oriented language) has both data members and methods. The methods are procedures that operate on the data members, and may or may not take arguments or return values. In this case, there are two procedures, "FIR" and "getOutputSample." The first, lines 6-10, is a constructor, which is a procedure that is called to create an FIR filter. It takes one argument, an array of double-precision floating-point numbers that specify the impulse response of the filter. The second, lines 11-22, takes a new input sample value as an argument and returns a new output sample. It also updates the delay line using a strategy called circular buffering. That is, the count member is used to keep track of where each new input sample should go. It gets incremented (line 19) each time the `getOutputSample()` method is called. When it exceeds the length of the buffer, it gets reset to zero. Thus, at all times, it contains the L most recently received input data samples. The most recent one is at index `count` in the buffer. The second most recent is at `count - 1`, or if that is negative, at `length - 1`. Line 17 makes sure that the variable `index` remains within the confines of the buffer as we iterate through the loop.

Probing further: Programmable DSP implementation of an FIR filter

The following section of code is the assembly language for a programmable DSP, which is a specialized microprocessor designed to implement signal processing functions efficiently in embedded systems (such as cellular telephones, digital cordless telephones, digital audio systems, etc.). This particular code is for the Motorola DSP56000 family of processors.

```
1     fir    movep           x:input,x:(r0)
2            clr a           x:(r0)-,x0      y:(r4)+,y0
3            rep m0
4            mac x0,y0,a     x:(r0)-,x0      y:(r4)+,y0
5            macr x0,y0,a    (r0)+
6            movep a,x:output
7            jmp fir
```

This processor has two memory banks called `x` and `y`. The code assumes that each input sample can be successively read from a memory location called `input`, and that the impulse response is stored in y memory beginning at an address stored in register r4. Moreover, it assumes that register r0 contains the address of a section of x memory to use for storing input samples (the delay line), and that this register has been set up to perform modulo addressing. Modulo addressing means that if it increments or decrements beyond the range of its buffer, then the address wraps around to the other end of the buffer. Finally, it assumes that the register m0 contains an integer specifying the number of samples in the impulse response minus one.

The key line (the one that does most of the work) is line 4. It follows a `rep` instruction, which causes that one line to be repeatedly executed the number of times specified by register m0. Line 4 multiplies the data in register x0 by that in y0 and adds the result to `a` (the accumulator). Such an operation is called a multiply and accumulate or mac operation. At the same time, it loads x0 with an input sample from the delay line, in x memory at a location given by r0. It also loads register y0 with a sample from the impulse response, stored in y memory at a location given by r4. At the same time, it increments r0 and decrements r4. This one-line instruction, which carries out several operations in parallel, is obviously highly tuned to FIR filtering. Indeed, FIR filtering is a major function of processors of this type.

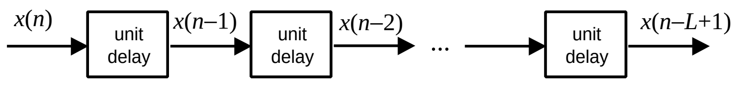


Figure 8.14: A delay line.

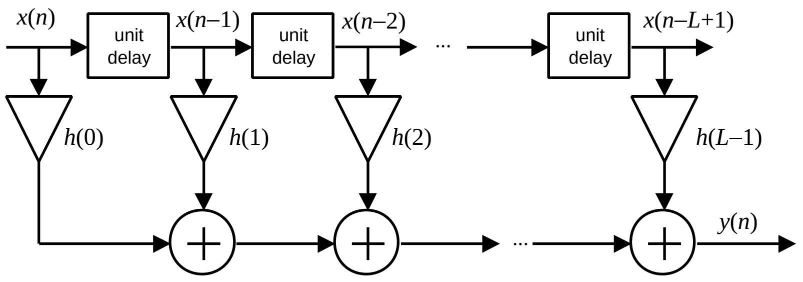


Figure 8.15: A tapped delay line realization of an FIR filter, described as a signal flow graph.

For each integer n, the output sample is the values in the delay line scaled by $h(0)$, $h(1)$, $\cdots$, $h(L-1)$. To obtain the values in the delay line, we simply tap the delay line, as shown in figure 8.15. The triangular boxes denote **multipliers** that multiply by a constant ($h(m)$, in this case). The circles with the plus signs denote **adders**. The structure in figure 8.15 is called a **tapped delay line** description of an FIR filter.

Diagrams of the type shown in figure 8.15 are called **signal flow graphs** because they describe computations by focusing on the flow of signals between operators. Signal flow graphs can be quite literal descriptions of digital hardware that implements a filter. But what they really describe is the computation, irrespective of whether the implementation is in hardware or software.

An IIR filter can also be described using a signal flow graph. Consider the difference equation in (8.25). A signal flow graph description of this equation is shown in figure 8.16. Notice that the left side is structurally equivalent to the tapped delay line, but stood on end. The right side represents the **feedback** in the filter, or the **recursion** that makes it an IIR filter rather than an FIR filter.

The filter structure in figure 8.16 can be simplified somewhat by observing that since the left and right sides are LTI systems, their order can be reversed without changing the output. Once the order is reversed, the two delay lines can be combined into one, as shown in figure 8.17. There are many other structures that can be used to realize IIR filters.

The relative advantages of one structure over another is a fairly deep topic, depending primarily on an understanding of the effects of finite precision arithmetic. It is beyond the scope of this text.

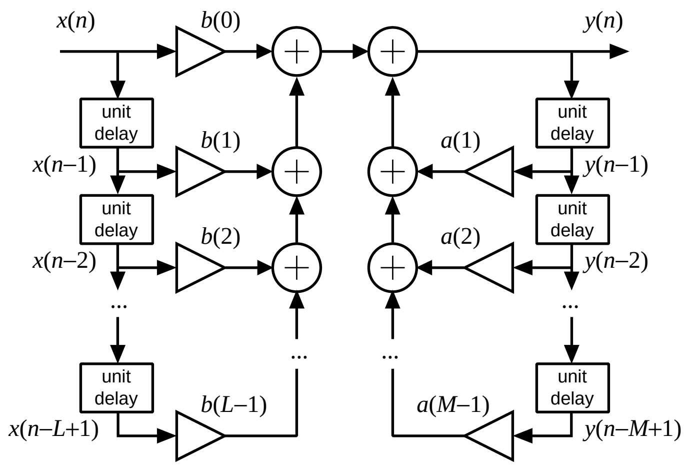


Figure 8.16: A signal flow graph describing an IIR filter. This is called a **direct form 1** filter structure.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Consider an LTI discrete-time system *Filter* with impulse response

$$h(n) = \sum_{k=0}^{7} \delta(n-k),$$

where δ is the Kronecker delta function.

(a) Sketch $h(n)$.

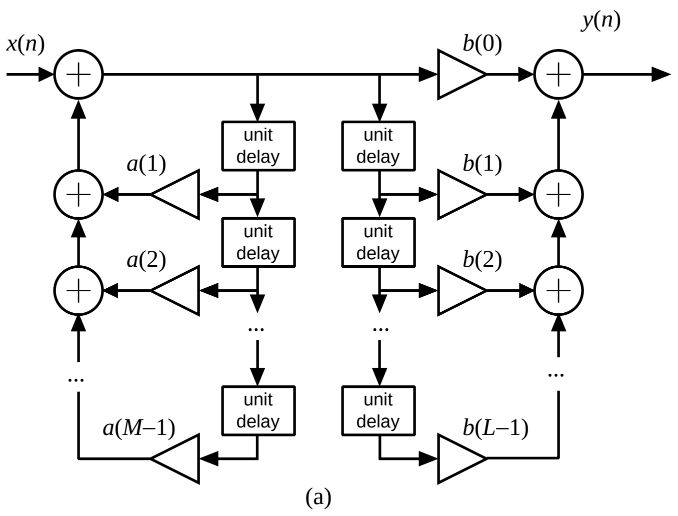
(b) Suppose the input signal x: *Ints* $\to$ *Reals* is such that $\forall\ n \in$ *Ints*, $x(n) = \cos(\omega n)$, where $\omega = \pi/4$ radians/sample. Give a simple expression for $y =$ *Filter*(x).

(c) Give the value of $H(\omega)$ for $\omega = \pi/4$, where H is the frequency response.

2. **E** Consider the continuous-time moving average system S, whose impulse response is shown in figure 8.3. Find its frequency response. The following fact from calculus may be useful:

$$\int_a^b e^{c\omega} c d\omega = e^{cb} - e^{ca}$$

for real a and b and complex c. Use Matlab to plot the magnitude of this frequency response over the range -5 Hz to 5 Hz. Note the symmetry of the magnitude response, as required by (8.21).



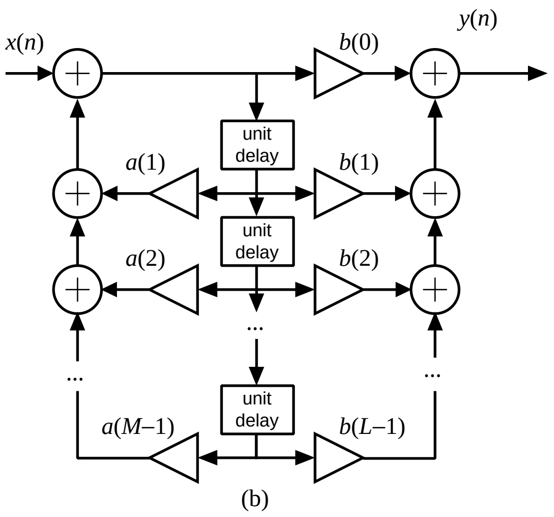


Figure 8.17: (a) A signal flow graph equivalent to that in figure 8.16, obtained by reversing the left and right components. (b) A signal flow graph equivalent to that in (a) obtained by merging the two delay lines into one. This is called a **direct form 2** filter structure.

3. **E** Consider a continuous-time LTI system with impulse response given by

$$\forall\, t \in \textit{Reals}, \quad h(t) = \delta(t-1) + \delta(t-2),$$

where δ is the Dirac delta function.

   (a) Find a simple equation relating the input x and output y of this system.
   (b) Find the frequency response of this system.
   (c) Use Matlab to plot the magnitude frequency response of this system in the range -5 to 5 Hz.

4. **E** Consider a discrete-time LTI system with impulse response h given by

$$\forall\, n \in \textit{Ints}, \quad h(n) = \delta(n) + 2\delta(n-1)$$

   (a) Plot the impulse response.
   (b) Find and sketch the output when the input is u, the unit step, given by (8.26).
   (c) Find and sketch the output when the input is a ramp,

$$r(n) = \begin{cases} n & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

   (d) Find the frequency response.
   (e) Show that the frequency response is periodic with period 2π.
   (f) Show that the frequency response is conjugate symmetric.
   (g) Give a simplified expression for the magnitude response.
   (h) Give a simplified expression for the phase response.
   (i) Suppose that the input x is given by

$$\forall\, n \in \textit{Ints}, \quad x(n) = \cos(\pi n/2 + \pi/6) + \sin(\pi n + \pi/3).$$

   Find the output y.

5. **E** Consider the sawtooth signal shown in figure 8.18. This is a periodic, continuous-time signal. Suppose it is filtered by an LTI system with frequency response

$$H(\omega) = \begin{cases} 1 & \text{if } |\omega| \leq 2.5 \text{ radians/second} \\ 0 & \text{otherwise} \end{cases}$$

What is the output?

6. **E** Suppose that the following difference equation relates the input x and output y of a discrete-time, causal LTI system S,

$$y(n) + \alpha y(n-1) = x(n) + x(n-1),$$

for some constant α.

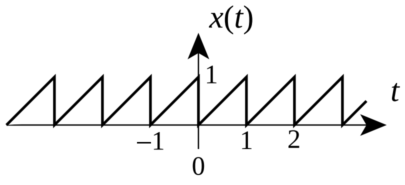


Figure 8.18: A sawtooth signal.

(a) Find the impulse response h.

(b) Find the frequency response H.

(c) Find a sinusoidal input with non-zero amplitude such that the output is zero.

(d) Use Matlab to create a plot of the magnitude of the frequency response, assuming $\alpha = -0.9$.

(e) Find a state-space description for this system (define the state s and find A, b, c^T, d).

(f) Suppose $\alpha = 1$. Find the impulse response and frequency response. Make sure your answer makes sense (check it against the original difference equation).

7. **T** Each of the statements below refers to a discrete-time system S with input x and output y. Determine whether the statement is true or false. The signal u below is the unit step, given by (8.26). The signal δ below is the Kronecker delta function.

   (a) Suppose you know that if x is a sinusoid then y is a sinusoid. Then you can conclude that S is LTI.

   (b) Suppose you know that S is LTI, and that if $x(n) = \cos(\pi n/2)$, then $y(n) = 2\cos(\pi n/2)$. Then you have enough information to determine the frequency response.

   (c) Suppose you know that S is LTI, and that if $x(n) = \delta(n)$, then
   $$y(n) = (0.9)^n u(n).$$
   Then you have enough information to determine the frequency response.

   (d) Suppose you know that S is LTI, and that if $x(n) = u(n)$, then $y(n) = (0.9)^n u(n)$. Then you have enough information to determine the frequency response.

   (e) Suppose you know that S is causal, and that input $x(n) = \delta(n)$ produces output $y(n) = \delta(n)+\delta(n-1)$, and input $x'(n) = \delta(n-2)$ produces output $y'(n) = 2\delta(n-2)+\delta(n-3)$. Then you can conclude that S is not LTI.

   (f) Suppose you know that S is causal, and that if $x(n) = \delta(n) + \delta(n-2)$ then $y(n) = \delta(n) + \delta(n-1) + 2\delta(n-2) + \delta(n-3)$. Then you can conclude that S is not LTI.

8. **T** Consider the continuous-time systems S_k given by, $\forall\, t \in$ *Reals*,
$$\begin{aligned}(S_1(x))(t) &= x(t-2),\\ (S_2(x))(t) &= x(t+2),\end{aligned}$$

$$\begin{aligned}(S_3(x))(t) &= x(t) - 2,\\ (S_4(x))(t) &= x(2-t),\\ (S_5(x))(t) &= x(2t),\\ (S_6(x))(t) &= t^2 x(t),\end{aligned}$$

(a) Which of these systems is linear?

(b) Which of these systems is time invariant?

(c) Which of these systems is causal?

9. **T** Consider an LTI discrete-time system *Filter* with impulse response

$$\forall\, n \in \textit{Ints}, \quad h(n) = \delta(n) + \delta(n-2),$$

where δ is the Kronecker delta function.

(a) Sketch h.

(b) Find the output when the input is u, the unit step, given by (8.26).

(c) Find the output when the input is a ramp,

$$r(n) = \begin{cases} n & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

(d) Suppose the input signal x is such that

$$\forall\, n \in \textit{Ints}, \quad x(n) = \cos(\omega n),$$

where $\omega = \pi/2$ radians/sample. Give a simple expression for $y = \textit{Filter}(x)$.

(e) Give an expression for $H(\omega)$ that is valid for all ω , where H is the frequency response.

(f) Sketch the magnitude of the frequency response. Can you explain your answer in part (b)?

(g) Is there any other frequency at which a sinusoidal input with a non-zero amplitude will yield an output that is zero?

10. **T** Consider an LTI discrete-time system *Filter* with impulse response

$$h(n) = \delta(n) - \delta(n-1)$$

where δ is the Kronecker delta function.

(a) Sketch $h(n)$.

(b) Suppose the input signal $x\colon \textit{Ints} \to \textit{Reals}$ is such that $\forall\, n \in \textit{Ints}$, $x(n) = 1$. Give a simple expression for $y = \textit{Filter}(x)$.

(c) Give an expression for $H(\omega)$ that is valid for all ω , where H is the frequency response.

(d) Sketch the magnitude of the frequency response. Can you explain your answer in part (b)?

11. **T** Consider a discrete-time LTI system with impulse response h given by

$$\forall\, n \in \textit{Ints}, \quad h(n) = \delta(n-1)/2 + \delta(n+1)/2$$

And consider the periodic discrete-time signal given by

$$\forall\, n \in \textit{Ints}, \quad x(n) = 2 + \sin(\pi n/2) + \cos(\pi n).$$

(a) Is the system causal?

(b) Find the frequency response of the system. Check that your answer is periodic with period 2π.

(c) For the given signal x, find the fundamental frequency ω_0 and the Fourier series coefficients X_k in the Fourier series expansion,

$$x(n) = \sum_{k=-\infty}^{\infty} X_k e^{i\omega_0 kn}.$$

Give the units of the fundamental frequency.

(d) Assuming the input to the system is x as given, find the output.

12. **T** Consider a continuous-time LTI system S. Suppose that when the input is the continuous-time **unit step**, given by

$$u(t) = \begin{cases} 1, & t \geq 0 \\ 0, & t < 0 \end{cases} \tag{8.27}$$

then the output $y = S(u)$ is given by

$$y(t) = \begin{cases} 1, & 0 \leq t \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

This output y is called the **step response** because it is the response to a unit step.

(a) Express y in terms of sums and differences of u and $D_1(u)$, where D_1 is the delay operator.

(b) Give a signal flow graph that produces this result $y = S(u)$ when the input is u. **Note**: We know that if two LTI systems have the same impulse response, then they are the same system. It is a fact, albeit a non-trivial one to demonstrate, that if two LTI systems have the same step response, then they are also the same system. Thus, your signal flow graph implements S.

(c) Use your signal flow graph to determine what the output y' of S is when the input is

$$x(t) = \begin{cases} 1, & 0 \leq t \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

Plot your answer.

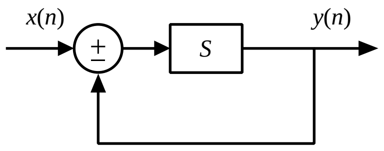


Figure 8.19: Feedback composition.

(d) What is the frequency response $H(\omega)$ of S?

13. **T** Suppose a discrete-time LTI system S has impulse response

$$h(n) = u(n)/2^n = \begin{cases} 1/2^n & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where u is the unit step function (8.26).

(a) What is the step response of this system? The step response is defined to be the output when the input is the unit step. **Hint**: The identity (8.23) might be helpful.

(b) What is the frequency response? Plot the magnitude and phase response (you may use Matlab, or do it by hand). **Hint**: The following variant of the identity (8.23) might be useful. If $|a| < 1$,

$$\sum_{m=0}^{\infty} a^m = \frac{1}{1-a}.$$

This follows immediately from (8.23) by letting L go to infinity.

(c) Suppose S is put in cascade with another identical system. What is the frequency response of the cascade composition?

(d) Suppose S is arranged in a feedback composition as shown in figure 8.19. What is the frequency response of the feedback composition?

Chapter 9

The Four Fourier Transforms

In Chapter 6 we saw that the Fourier series describes a periodic signal as a sum of complex exponentials. In Chapter 7 we saw that if the input to an LTI system is a sum of complex exponentials, then the frequency response of the LTI system describes its response to each of the component exponentials. Thus we can calculate the system response to any periodic input signal by combining the responses to the individual components.

The response of the LTI system to any input signal can also be obtained as the convolution of the input signal and the impulse response. The impulse response and the frequency response give us the same information about the system, but in different forms. The impulse response and the frequency response are closely related.

For discrete-time systems, the frequency response can be described as a sum of weighted complex exponentials (the DTFT), where the weights turn out to be the impulse response samples. We will see that the impulse response is, in fact, a Fourier series representation of the frequency response, with the roles of time and frequency reversed from the uses of the Fourier series that we have seen so far.

This reappearance of the Fourier series is not a coincidence. In this chapter, we explore this pattern by showing that the Fourier series is a special case of a family of representations of signals that are collectively called **Fourier transforms**. The Fourier series applies specifically to continuous-time, periodic signals. The discrete Fourier series applies to discrete-time, periodic signals. We complete the story with the **continuous-time Fourier transform** (**CTFT**), which applies to continuous-time signals that are not periodic, and the **discrete-time Fourier transform** (**DTFT**), which applies to discrete-time signals that are not periodic.

9.1 Notation

We define the following four sets of signals:

- *ContSignals* $= [Reals \to Comps]$. Since *Reals* is included in *Comps*, *ContSignals* includes continuous-time signals whose range is *Reals*, and so we won't need to consider these separately.

ContSignals includes continuous-time signals, but we are not insisting that the domain be interpreted as time. Indeed, sometimes the domain could be interpreted as space, if we are dealing with images. In this chapter we will see that it is useful sometimes to interpret the domain as frequency.

- $\mathit{DiscSignals} = [\mathit{Ints} \to \mathit{Comps}]$.

  This includes discrete-time signals whose domain is time or sample number, but again we are not insisting that the domain be interpreted as time.

- $\mathit{ContPeriodic}_p \subset \mathit{ContSignals}$.

  This set is defined to contain all continuous signals that are periodic with period p, where p is a real number.

- $\mathit{DiscPeriodic}_p \subset \mathit{DiscSignals}$.

  This set is defined to contain all discrete signals that are periodic with period p, where p is an integer.

Note that whenever we talk about periodic signals we could equally well talk about finite signals, where the finite signal consists of one cycle of the periodic signal.

9.2 The Fourier series (FS)

The **continuous-time Fourier series** of a periodic signal $x \in \mathit{ContPeriodic}_p$ is

$$\boxed{\forall\, t \in \mathit{Reals}, \quad x(t) = \sum_{k=-\infty}^{\infty} X_k e^{ik\omega_0 t},} \tag{9.1}$$

where $\omega_0 = 2\pi/p$ (radians/second). The Fourier series coefficients are given by

$$\boxed{\forall\, m \in \mathit{Ints}, \quad X_m = \frac{1}{p}\int_0^p x(t)e^{-im\omega_0 t}dt.} \tag{9.2}$$

Observe that the sequence of Fourier series coefficients given by (9.2) can be regarded as a signal $X \in \mathit{DiscSignals}$, where

$$\forall m \in \mathit{Ints}, \quad X(m) = X_m.$$

So we can define a system $\mathit{FourierSeries}_p$ with domain $\mathit{ContPeriodic}_p$ and range *DiscSignals* such that if the input is the periodic signal x, the output is its Fourier series coefficients, X. That is,

$$\mathit{FourierSeries}_p \colon \mathit{ContPeriodic}_p \to \mathit{DiscSignals}.$$

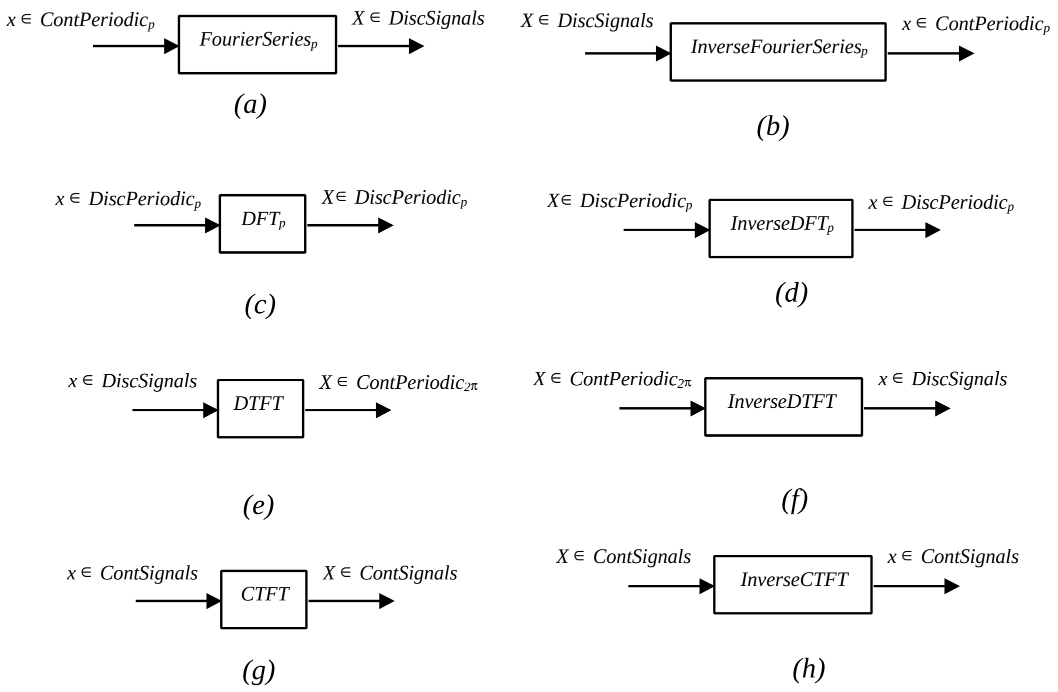


Figure 9.1: Fourier transforms as systems.

This system is the first of four forms of the Fourier transform, and is depicted graphically in figure 9.1(a). Its inverse is a system

$$\textit{InverseFourierSeries}_p \colon \textit{DiscSignals} \to \textit{ContPeriodic}_p,$$

depicted in figure 9.1(b).

The two systems, *FourierSeries*$_p$ and *InverseFourierSeries*$_p$, are **inverses** of each other, because (see box)

$$\forall x \in \textit{ContPeriodic}_p, \quad (\textit{InverseFourierSeries}_p \circ \textit{FourierSeries}_p)(x) = x \qquad (9.3)$$

$$\forall X \in \textit{DiscSignals}, \quad (\textit{FourierSeries}_p \circ \textit{InverseFourierSeries}_p)(X) = X \qquad (9.4)$$

9.3 The discrete Fourier transform (DFT)

The **discrete-time Fourier series** (**DFS**) expansion for $x \in \textit{DiscPeriodic}_p$ is (see (7.11))

$$\forall\, n \in \textit{Ints}, \quad x(n) = \sum_{k=0}^{p-1} X_k e^{ik\omega_0 n}, \qquad (9.5)$$

Probing further: Showing inverse relations

In this chapter, and in previous chapters, we have given formulas that describe time-domain functions in terms of frequency-domain functions, and vice versa. By convention, a formula that gives the frequency-domain function in terms of the time-domain function is called a **Fourier transform**, and the formula that gives the time-domain function in terms of the frequency-domain function is called an **inverse Fourier transform**. As shown in figure 9.1, these transforms can be viewed as systems that take as inputs signals in one domain and return signals in the other. We discuss four distinct Fourier transforms and their corresponding inverses. In each case, it is possible to show that the Fourier transform and its inverse are in fact inverses of one another, as stated in (9.3) and (9.4). We prove the second relation, (9.4), to illustrate how this is done. Similar proofs can be carried out for all four types of Fourier transforms, although sometimes these proofs require you to be adept at manipulating Dirac delta functions, which requires significant mathematical skill.

Let $X \in$ *DiscSignals* be the Fourier series for some $x \in$ *ContPeriodic*$_p$. That is, $x =$ *InverseFourierSeries*$_p(X)$. Let $Y =$ *FourierSeries*$_p(x)$. We now show that $Y = X$, i.e. $Y_m = X_m$ for all m.

$$
\begin{aligned}
Y_m &= \frac{1}{p}\int_0^p x(t)e^{-im\omega_0 t}dt \text{ by (9.2)} \\
&= \frac{1}{p}\int_0^p [\sum_{k=-\infty}^{\infty} X_k e^{ik\omega_0 t}]e^{-im\omega_0 t}dt, \text{ by (9.1)} \\
&= \sum_{k=-\infty}^{\infty} \frac{1}{p} X_k \int_0^p e^{i(k-m)\omega_0 t}dt \\
&= \frac{1}{p}X_m \int_0^p dt + \sum_{k\neq m} \frac{1}{p} X_k \int_0^p e^{i(k-m)\omega_0 t}dt \\
&= X_m,
\end{aligned}
$$

since for $k \neq m$,

$$\int_0^p e^{i(k-m)\omega_0 t}dt = 0.$$

where $\omega_0 = 2\pi/p$ (radians/sample). The Fourier series coefficients can be found using the formula

$$\forall\, k \in \textit{Ints}, \quad X_k = \frac{1}{p}\sum_{m=0}^{p-1} x(m)e^{-imk\omega_0}. \tag{9.6}$$

For historical reasons, the **discrete Fourier transform** (**DFT**) is the DFS with slightly different scaling. It is defined by

$$\boxed{\forall\, n \in \textit{Ints}, \quad x(n) = \tfrac{1}{p}\sum_{k=0}^{p-1} X'_k e^{ik\omega_0 n},} \tag{9.7}$$

$$\boxed{\forall\, k \in \textit{Ints}, \quad X'_k = \sum_{m=0}^{p-1} x(m)e^{-imk\omega_0}.} \tag{9.8}$$

Obviously, the DFT coefficients are related to the DFS coefficients by

$$X'_k = pX_k.$$

This scaling is somewhat unfortunate, since it means that the DFT coefficients do not have the same units as the signal x, but the scaling is firmly established in the literature, so we stick to it. We omit the prime when it is clear whether we are talking about the Fourier transform instead of the Fourier series.

Observe that $\textit{DFT}_p(x)$ is a discrete signal that is itself periodic with period p. To verify this, note that for any integer N and for all integers k,

$$\begin{aligned}
X'_{k+Np} &= \sum_{m=0}^{p-1} x(m)e^{-im(k+Np)\omega_0} \\
&= \sum_{m=0}^{p-1} x(m)e^{-imk\omega_0}e^{-imNp\omega_0} \\
&= \sum_{m=0}^{p-1} x(m)e^{-imk\omega_0}e^{-imN2\pi}, \text{ since } \omega_0 = 2\pi/p \\
&= \sum_{m=0}^{p-1} x(m)e^{-imk\omega_0} \\
&= X'_k.
\end{aligned}$$

Comparing (9.5) and (9.8) and observing that x is periodic with period p, we see that $x(-m)$ is the m-th Fourier series coefficient for the function X'! The DFT is rather special, therefore, in that both the time and frequency domain representations of a function are Fourier series, and both are periodic.

The DFT therefore is the function

$$\mathit{DFT}_p\colon \mathit{DiscPeriodic}_p \to \mathit{DiscPeriodic}_p,$$

given by (9.6). The inverse DFT is the function

$$\mathit{InverseDFT}_p\colon \mathit{DiscPeriodic}_p \to \mathit{DiscPeriodic}_p,$$

given by (9.5). As in (9.3), (9.4), DFT_p and $\mathit{InverseDFT}_p$ are inverses of each other. This can be verified using methods similar to those in the box on page 248. The DFT and its inverse are computed by systems that we can represent as shown in figure 9.1(c) and (d).

The DFT is the most useful form of the Fourier transform for computation, because the system DFT_p is easily implemented on a computer. Both summations (9.7) and (9.8) are finite. Moreover, there is an algorithm called the **fast Fourier transform** (**FFT**) that calculates these summations with far fewer arithmetic operations than the most direct method. Moreover, because the sums are finite, the DFT always exists. There are no mathematical problems with convergence.

9.4 The discrete-Time Fourier transform (DTFT)

We have shown that the frequency response of an LTI system is related to the impulse response by

$$\forall\, \omega \in \mathit{Reals}, \quad H(\omega) = \sum_{m=-\infty}^{\infty} h(m) e^{-i\omega m}.$$

H is called the **discrete-time Fourier transform** (**DTFT**) of h. For any $x \in \mathit{DiscSignals}$ (not just an impulse response), its DTFT is

$$\boxed{\forall\, \omega \in \mathit{Reals}, \quad X(\omega) = \sum_{m=-\infty}^{\infty} x(m) e^{-i\omega m}.} \tag{9.9}$$

Of course, this definition is only valid for those x and those ω where the sum converges (it is not trivial mathematically to characterize this).

Notice that the function X is periodic with period 2π. I.e., $X(\omega) = X(\omega + 2\pi N)$ for any integer N, because

$$e^{-i\omega t} = e^{-i(\omega + 2\pi N)t}$$

for any integer N. Thus, $X \in \mathit{ContPeriodic}_{2\pi}$.

Comparing (9.9) and (9.1), you can recognize that $x(-n)$ is the n-th Fourier series coefficient for the periodic function X. Thus, the DTFT is just a Fourier series, but with the role of time and frequency reversed! The time-domain function is a set of Fourier series coefficients for a periodic frequency domain function.

The DTFT (9.9) has similar structure to the DFT (9.8). In fact, the DTFT can be viewed as a generalization of the DFT to signals that are neither periodic nor finite. In other words, as p approaches infinity, ω_0 approaches zero, so instead of a discrete set of frequencies spaced by ω_0 we have a continuum.

The DTFT is a system

$$DTFT\colon DiscSignals \to ContPeriodic_{2\pi}$$

and its inverse is

$$InverseDTFT\colon ContPeriodic_{2\pi} \to DiscSignals.$$

The inverse is given by

$$\boxed{\forall\, n \in Ints, \quad x(n) = \frac{1}{2\pi}\int\limits_0^{2\pi} X(\omega)e^{i\omega n}d\omega.} \tag{9.10}$$

Notice that because X is periodic, this can equivalently be written as

$$\forall\, n \in Ints, \quad x(n) = \frac{1}{2\pi}\int\limits_{-\pi}^{\pi} X(\omega)e^{i\omega n}d\omega.$$

We integrate over one cycle of the periodic function, so it does not matter where we start. These are depicted graphically in figure 9.1(e) and (f).

DTFT and *InverseDTFT* are inverses of each other. This follows from the fact that *FourierSeries*$_p$ and *InverseFourierSeries*$_p$ are inverses of each other.

9.5 The continuous-time Fourier transform

The frequency response and impulse response of a continuous-time LTI system are related by the **continuous-time Fourier transform** (**CTFT**), more commonly called simply the **Fourier transform** (**FT**),

$$\boxed{\forall\, \omega \in Reals, \quad H(\omega) = \int\limits_{-\infty}^{\infty} h(t)e^{-i\omega t}dt.} \tag{9.11}$$

The CTFT can be defined for any function $h \in ContSignals$ where the integral exists. It need not be the impulse response of any system, and it need not be periodic or finite. The inverse relation is

$$\boxed{\forall\, t \in Reals, \quad h(t) = \frac{1}{2\pi}\int\limits_{-\infty}^{\infty} H(\omega)e^{i\omega t}d\omega.} \tag{9.12}$$

It is true but difficult to prove that the CTFT and the inverse CTFT are indeed inverses.

The CTFT can be viewed as a generalization of both the FS and DTFT where neither the frequency domain nor the time domain needs to be periodic. Alternatively, it can be viewed as the last remaining Fourier transform, where neither time nor frequency is discrete. Graphically, the CTFT is a system as shown in figure 9.1(g) and (h).

The four Fourier transforms are summarized in table 9.1.

	Aperiodic time Continuous frequency	Periodic time Discrete frequency
Aperiodic frequency Continuous time	*CTFT*: *ContSignals* $\to$ *ContSignals* $X(\omega) = \int\limits_{-\infty}^{\infty} x(t)e^{-i\omega t}dt$ *InverseCTFT*: *ContSignals* $\to$ *ContSignals* $x(t) = \frac{1}{2\pi}\int\limits_{-\infty}^{\infty} X(\omega)e^{i\omega t}d\omega$	*FourierSeries*$_p$: *ContPeriodic*$_p$ $\to$ *DiscSignals* $X_m = \frac{1}{p}\int\limits_{0}^{p} x(t)e^{-im\omega_0 t}dt$ *InverseFourierSeries*$_p$: *DiscSignals* $\to$ *ContPeriodic*$_p$ $x(t) = \sum\limits_{k=-\infty}^{\infty} X_k e^{ik\omega_0 t}$
Periodic frequency Discrete time	*DTFT*: *DiscSignals* $\to$ *ContPeriodic*$_{2\pi}$ $X(\omega) = \sum\limits_{n=-\infty}^{\infty} x(n)e^{-in\omega}$ *InverseDTFT*: *ContPeriodic*$_{2\pi}$ $\to$ *DiscSignals* $x(n) = \frac{1}{2\pi}\int\limits_{-\pi}^{\pi} X(\omega)e^{i\omega n}d\omega$	*DFT*$_p$: *DiscPeriodic*$_p$ $\to$ *DiscPeriodic*$_p$ $X_k = \sum\limits_{n=0}^{p-1} x(n)e^{-ink\omega_0}$ *InverseDFT*$_p$: *DiscPeriodic*$_p$ $\to$ *DiscPeriodic*$_p$ $x(n) = \frac{1}{p}\sum\limits_{k=0}^{p-1} X_k e^{ik\omega_0 n}$

Table 9.1: The four Fourier transforms summarized. The column and row titles tell you when to use the specified Fourier transform and its inverse. The first row applies to continuous-time signals, and the second column applies to periodic signals. Thus, if you have a continuous-time periodic signal, you should use the Fourier transform at the upper right, which is the Fourier series.

Probing further: Multiplying signals

We have seen that convolution in the time domain corresponds to multiplication in the frequency domain. It turns out that this relationship is symmetric, in that multiplication in the time domain corresponds to a peculiar form of convolution in the frequency domain. That is, given two discrete-time signals x and p with DTFTs X and P, if we multiply them in the time domain,

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n)p(n)$$

then in the frequency domain, $Y(\omega) = (X \circledast P)(\omega)$, where the symbol "$\circledast$" indicates circular convolution, defined by

$$\forall\, \omega \in \textit{Reals}, \quad (X \circledast P)(\omega) = \frac{1}{2\pi}\int_0^{2\pi} X(\Omega)P(\omega - \Omega)d\Omega.$$

To verify this, we can substitute into the above integral the definitions for the DTFTs $X(\omega)$ and $P(\omega)$ to get

$$\begin{aligned}
(X \circledast P)(\omega) &= \frac{1}{2\pi}\int_0^{2\pi}\left(\sum_{m=-\infty}^{\infty} x(m)e^{-i\Omega m}\right)\left(\sum_{k=-\infty}^{\infty} p(k)e^{-i(\omega-\Omega)k}\right)d\Omega \\
&= \sum_{k=-\infty}^{\infty} p(k)e^{-i\omega k}\sum_{m=-\infty}^{\infty} x(m)\frac{1}{2\pi}\int_0^{2\pi} e^{-i\Omega(m-k)}d\Omega \\
&= \sum_{k=-\infty}^{\infty} p(k)x(k)e^{-i\omega k},
\end{aligned}$$

where the last equality follows from the observation that the integral in the middle expression is zero except when $m = k$, when it has value one. Thus, $(X \circledast P)(\omega)$ is the DTFT of $y = xp$, as we claimed.

The continuous-time case is somewhat simpler. If we have $\forall\, t \in \textit{Reals}, \;\; y(t) = x(t)p(t)$ then in the frequency domain,

$$\forall\, \omega \in \textit{Reals}, \quad Y(\omega) = \frac{1}{2\pi}(X * P)(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\Omega)P(\omega - \Omega)d\Omega.$$

The "$*$" indicates ordinary convolution.

9.6 Relationship to convolution

Suppose a discrete-time LTI system has impulse response h and frequency response H. We have seen that if the input to this system is a complex exponential, $e^{i\omega n}$, then the output is $H(\omega)e^{i\omega n}$. Suppose the input is instead an arbitrary signal x with DTFT X. Using the inverse DTFT relation, we know that

$$\forall\, n \in \textit{Ints}, \quad x(n) = \frac{1}{2\pi}\int_0^{2\pi} X(\omega)e^{i\omega n}d\omega$$

View this as a summation of exponentials, each with weight $X(\omega)$. An integral, after all, is summation over a continuum. Each term in the summation is $X(\omega)e^{i\omega n}$. If this term were an input by itself, then the output would be $H(\omega)X(\omega)e^{i\omega n}$. Thus, by linearity, if the input is x, the output should be

$$\forall\, n \in \textit{Ints}, \quad y(n) = \frac{1}{2\pi}\int_0^{2\pi} H(\omega)X(\omega)e^{i\omega n}d\omega$$

Comparing to the inverse DTFT relation for $y(n)$, we see that

$$\boxed{\forall\, \omega \in \textit{Reals}, \quad Y(\omega) = H(\omega)X(\omega).} \tag{9.13}$$

This is the frequency-domain version of convolution

$$\forall\, n \in \textit{Ints}, \quad y(n) = (h * x)(n).$$

Thus, the frequency response of an LTI system multiplies the DTFT of the input. This is intuitive, since the frequency response gives the weight imposed by the system on each frequency component of the input.

Equation (9.13) applies equally well to continuous-time systems, but in that case, H is the CTFT of the impulse response, and X is the CTFT of the input.

9.7 Properties and examples

In this section, we give a number of useful properties of the various Fourier transforms, and a number of illustrative examples. The properties together with the examples can often be used to avoid solving integrals or summations to find the Fourier transform of some signal, or to find an inverse Fourier transform.

9.7.1 Conjugate symmetry

We have already shown (see (7.10)) that for real-valued signals, the Fourier series coefficients are conjugate symmetric,

$$X_k = X^*_{-k}.$$

In fact, all the Fourier transforms are conjugate symmetric if the time-domain function is real. We illustrate this with the CTFT. Suppose $x\colon \mathit{Reals} \to \mathit{Reals}$ is a real-valued signal, and let $X = \mathit{CTFT}(x)$. In general, $X(\omega)$ will be complex-valued. However, from (9.11), we have

$$\begin{aligned}
[X(-\omega)]^* &= \int_{-\infty}^{\infty} [x(t)e^{i\omega t}]^* dt \\
&= \int_{-\infty}^{\infty} x(t)e^{-i\omega t} dt \\
&= X(\omega),
\end{aligned}$$

Thus,

$$X(\omega) = X^*(-\omega),$$

i.e. for real-valued signals, $X(\omega)$ and $X(-\omega)$ are complex conjugates of one another.

We can show that, conversely, if the Fourier transform is conjugate symmetric, then the time-domain function is real. To do this, write the inverse CTFT

$$x(t) = \frac{1}{2\pi} \int\limits_{-\infty}^{\infty} X(\omega)e^{i\omega t} d\omega$$

and then conjugate both sides,

$$x^*(t) = \frac{1}{2\pi} \int\limits_{-\infty}^{\infty} X^*(\omega)e^{-i\omega t} d\omega.$$

By changing variables (replacing ω with $-\omega$) and using the conjugate symmetry of X, we can show that

$$x^*(t) = x(t),$$

which implies that $x(t)$ is real for all t.

The inverse Fourier transforms can be used to show that if a time-domain function is conjugate symmetric,

$$x(t) = x^*(-t),$$

then its Fourier transform is real. The same property applies to all four Fourier transforms.

In summary, if a function in one domain (time or frequency) is conjugate symmetric, then the function in the other domain (frequency or time) is real.

9.7.2 Time shifting

Given a continuous-time function x and its Fourier transform $X = \mathit{CTFT}(x)$, let y be defined by

$$\forall\, t \in \mathit{Reals}, \quad y(t) = x(t - \tau)$$

for some real constant τ. This is called **time shifting** or **delay**. We can find $Y = \mathit{CTFT}(y)$ in terms of X as follows,

$$\begin{aligned} Y(\omega) &= \int_{-\infty}^{\infty} y(t)e^{-i\omega t}dt \\ &= \int_{-\infty}^{\infty} x(t-\tau)e^{-i\omega t}dt \\ &= \int_{-\infty}^{\infty} x(t)e^{-i\omega(t+\tau)}dt \\ &= e^{-i\omega\tau}\int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt \\ &= e^{-i\omega\tau}X(\omega). \end{aligned}$$

Thus, in summary,

$$\boxed{y(t) = x(t-\tau) \;\Leftrightarrow\; Y(\omega) = e^{-i\omega\tau}X(\omega).} \tag{9.14}$$

The bidirectional arrow indicates that this relationship works both ways. If you know that $Y(\omega) = e^{-i\omega\tau}X(\omega)$, then you can conclude that $y(t) = x(t-\tau)$.

Example 9.1: One of the simplest Fourier transforms to compute is that of x when

$$x(t) = \delta(t),$$

where δ is the Dirac delta function. Plugging into the formula,

$$\begin{aligned} X(\omega) &= \int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt \\ &= \int_{-\infty}^{\infty} \delta(t)e^{-i\omega t}dt \\ &= e^{-i\omega 0} \\ &= 1, \end{aligned}$$

where we have used the sifting property of the delta function. The Fourier transform is a constant, 1.[1] This indicates that the Dirac delta function, interestingly, contains all frequencies in equal amounts.

[1]To see how the sifting property works, note that the integrand is zero everywhere except where $t = 0$, at which point the complex exponential evaluates to 1. Then the integral of the delta function itself has value one.

Moreover, this result is intuitive if one considers an LTI system with impulse response $h(t) = \delta(t)$. Such a system responds to an impulse by producing an impulse, which suggests that any input will be simply passed through unchanged. Indeed, its frequency response is $H(\omega) = 1$ for all $\omega \in$ *Reals*, so given an input x with Fourier transform X, the output y has Fourier transform

$$Y(\omega) = H(\omega)X(\omega) = X(\omega).$$

Since the output has the same Fourier transform as the input, the output is the same as the input.

Using (9.14), we can now find the Fourier transform of another signal

$$x(t) = \delta(t - \tau),$$

for some constant $\tau \in$ *Reals*. It is

$$X(\omega) = e^{-i\omega\tau}.$$

Note that, as required, this is conjugate symmetric. Moreover, it has magnitude 1. Its phase is $-\omega\tau$, a linear function of ω.

Again, we can gain some intuition by considering an LTI system with impulse response

$$h(t) = \delta(t - \tau). \tag{9.15}$$

Such a system introduces a fixed time delay of τ. Its frequency response is

$$H(\omega) = e^{-i\omega\tau}. \tag{9.16}$$

Since this has magnitude 1 for all ω, it tells us that all frequencies get through the delay system unattenuated, as expected. However, each frequency ω will experience a phase shift of $-\omega\tau$, corresponding to a time delay of τ.

Discrete-time signals are similar. Consider a discrete-time signal x with $X = \textit{DTFT}(x)$, and let y be defined by

$$\forall\, t \in \textit{Reals}, \quad y(n) = x(n - \tau)$$

for some integer constant τ. By similar methods, we can find that

$$\boxed{y(n) = x(n - \tau) \;\Leftrightarrow\; Y(\omega) = e^{-i\omega\tau}X(\omega).} \tag{9.17}$$

Example 9.2: Suppose we have a discrete-time signal x given by

$$x(n) = \delta(n),$$

where δ is the Kronecker delta function. It is easy to show from the DTFT definition that

$$X(\omega) = 1.$$

Using (9.17), we can now find the Fourier transform of another signal

$$x(n) = \delta(n - \tau),$$

for some constant $\tau \in$ *Ints*. It is

$$X(\omega) = e^{-i\omega\tau}.$$

Notice that if $\tau = 0$, this reduces to $X(\omega) = 1$, as expected.

9.7.3 Linearity

Consider three discrete-time signals x, x_1, x_2, related by

$$x(n) = ax_1(n) + bx_2(n).$$

Then it is easy to see from the definition of the DTFT that

$$X(\omega) = aX_1(\omega) + bX_2(\omega)$$

where $X = \mathit{DTFT}(x)$, $X_1 = \mathit{DTFT}(x_1)$, and $X_2 = \mathit{DTFT}(x_2)$.

The same linearity property applies to the CTFT,

$$x(t) = ax_1(t) + bx_2(t) \;\Leftrightarrow\; X(\omega) = aX_1(\omega) + bX_2(\omega)$$

Linearity of the Fourier transform is one of the most useful properties for avoiding evaluation of integrals and summations.

Example 9.3: Consider for example the discrete-time signal x given by

$$x(n) = \delta(n+1) + \delta(n-1),$$

where δ is the Kronecker delta function. Using linearity, we know that the DTFT of x is the sum of the DTFT of $\delta(n+1)$ and the DTFT of $\delta(n-1)$. From the previous example, we know those two DTFTs, so

$$X(\omega) = e^{i\omega} + e^{-i\omega}$$

because τ is -1 and 1, respectively. Using Euler's relation, we can simplify this to get

$$X(\omega) = 2\cos(\omega).$$

Interestingly, the DTFT of this example turns out to be real. This is because the time-domain function is conjugate symmetric (the conjugate of something real is itself). Moreover, since it is real in the time domain, the DTFT turns out to be conjugate symmetric.

Linearity can also be used to find inverse Fourier transforms.

> **Example 9.4:** Suppose you are told that a continuous-time signal has Fourier transform
>
> $$X(\omega) = \cos(\omega).$$
>
> How would you find the time-domain function? You could evaluate the inverse CTFT, but the integration that you would have to perform is quite difficult. Instead, use Euler's relation to write
>
> $$X(\omega) = (e^{i\omega} + e^{-i\omega})/2.$$
>
> Then use linearity. The inverse Fourier transform of this sum will be the sum of the inverse Fourier transforms of the terms. These we can recognize from (9.15) and (9.16), so
>
> $$x(t) = (\delta(t+1) + \delta(t-1))/2,$$
>
> where δ is the Dirac delta function.

9.7.4 Constant signals

We have seen that the Fourier transform of a delta function is a constant. With the symmetries that we have observed between time and frequency, it should come as no surprise that the Fourier transform of a constant is a delta function.

Consider first a continuous-time signal x given by

$$\forall\, t \in \textit{Reals}, \quad x(t) = K$$

for some real constant K. Its CTFT is

$$X(\omega) = K \int_{-\infty}^{\infty} e^{-i\omega t} dt,$$

which is not easy to evaluate. This integral is mathematically subtle. The answer is

$$\forall\, \omega \in \textit{Reals}, \quad X(\omega) = 2\pi K \delta(\omega),$$

where δ is the Dirac delta function. What this says is that a constant in the time domain is concentrated at zero frequency in the frequency domain (which should not be surprising). Except for the multiplying constant of 2π, we probably could have guessed this answer. We can verify this answer by evaluating the inverse CTFT,

$$\begin{aligned} x(t) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{i\omega t} d\omega \\ &= K \int_{-\infty}^{\infty} \delta(\omega) e^{i\omega t} d\omega \\ &= K, \end{aligned}$$

where the final step follows from the sifting property of the Dirac delta function. Thus, in summary,

$$\boxed{x(t) = K \;\Leftrightarrow\; X(\omega) = 2\pi K\delta(\omega)} \tag{9.18}$$

The discrete-time case is similar, but there is one subtlety because the DTFT is periodic. Let x be a discrete time signal where

$$\forall\, n \in \textit{Ints}, \quad x(n) = K$$

for some real constant K. Its DTFT is

$$\forall\, \omega \in [-\pi, \pi], \quad X(\omega) = 2\pi K\delta(\omega),$$

where δ is the Dirac delta function. This is easy to verify using the inverse DTFT. Again, this says is that a constant in the time domain is concentrated at zero frequency in the frequency domain (which should not be surprising). However, recall that a DTFT is periodic with period 2π, meaning that for all integers N,

$$X(\omega) = X(\omega + N2\pi).$$

Thus, in addition to a delta function at $\omega = 0$, there must be one at $\omega = 2\pi$, $\omega = -2\pi$, $\omega = 4\pi$, etc. This can be written using a shift-and-add summation,

$$\forall\, \omega \in \textit{Reals}, \quad X(\omega) = 2\pi K \sum_{k=-\infty}^{\infty} \delta(\omega - k2\pi).$$

Thus, in summary,

$$\boxed{x(n) = K \;\Leftrightarrow\; X(\omega) = 2\pi K \sum_{k=-\infty}^{\infty} \delta(\omega - k2\pi).} \tag{9.19}$$

9.7.5 Frequency shifting and modulation

Suppose that x is a continuous-time signal with CTFT X. Let y be another continuous-time signal defined by

$$y(t) = x(t)e^{i\omega_0 t}$$

for some real constant ω_0. The CTFT of y is easy to compute,

$$\begin{aligned} Y(\omega) &= \int_{-\infty}^{\infty} y(t)e^{-i\omega t}dt \\ &= \int_{-\infty}^{\infty} x(t)e^{i\omega_0 t}e^{-i\omega t}dt \\ &= \int_{-\infty}^{\infty} x(t)e^{-i(\omega-\omega_0)t}dt \\ &= X(\omega - \omega_0). \end{aligned}$$

Thus, the Fourier transform of y is the same as that of x, but shifted to the right by ω_0. In summary,

$$\boxed{y(t) = x(t)e^{i\omega_0 t} \;\Leftrightarrow\; Y(\omega) = X(\omega - \omega_0).} \qquad (9.20)$$

This result can be used determine the effect of multiplying a signal by a sinusoid, a process called **modulation** (see exercise 5 and lab C.10).

Example 9.5: Suppose

$$y(t) = x(t)\cos(\omega_0 t).$$

Use Euler's relation to rewrite this

$$y(t) = x(t)(e^{i\omega_0 t} + e^{-i\omega_0 t})/2.$$

Then use (9.20) to get the CTFT,

$$Y(\omega) = (X(\omega - \omega_0) + X(\omega + \omega_0))/2.$$

We can combine (9.20) with (9.18) to get the following facts:

$$\boxed{x(t) = e^{i\omega_0 t} \;\Leftrightarrow\; X(\omega) = 2\pi\delta(\omega - \omega_0).} \qquad (9.21)$$

This says that a complex exponential signal with frequency ω_0 is concentrated in the frequency domain at frequency ω_0, which should not be surprising. Similarly,

$$\boxed{x(t) = \cos(\omega_0 t) \;\Leftrightarrow\; X(\omega) = \pi(\delta(\omega - \omega_0) + \delta(\omega + \omega_0)).} \qquad (9.22)$$

We can get a similar set of results for discrete-time signals. We summarize the results here, and leave their verification to the reader (see exercise 3):

$$\boxed{y(n) = x(n)e^{i\omega_0 n} \;\Leftrightarrow\; Y(\omega) = X(\omega - \omega_0).} \qquad (9.23)$$

$$\boxed{y(n) = x(n)\cos(\omega_0 n) \;\Leftrightarrow\; Y(\omega) = (X(\omega - \omega_0) + X(\omega + \omega_0))/2.} \qquad (9.24)$$

$$\boxed{x(n) = e^{i\omega_0 n} \;\Leftrightarrow\; X(\omega) = 2\pi \sum_{k=-\infty}^{\infty} \delta(\omega - \omega_0 - k2\pi).} \qquad (9.25)$$

$$\boxed{x(n) = \cos(\omega_0 n) \;\Leftrightarrow\; X(\omega) = \pi \sum_{k=-\infty}^{\infty} (\delta(\omega - \omega_0 - k2\pi) + \delta(\omega + \omega_0 - k2\pi)).} \qquad (9.26)$$

Additional properties of Fourier transforms are explored in the exercises.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Show that if two discrete-time systems with frequency responses $H_1(\omega)$ and $H_2(\omega)$ are connected in cascade, that the DTFT of the output is given by $Y(\omega) = H_1(\omega)H_2(\omega)X(\omega)$, where $X(\omega)$ is the DTFT of the input.

2. **T** Consider a continuous-time signal x with Fourier transform X. Let y be such that
$$\forall\, t \in \textit{Reals}, \quad y(t) = x(at),$$
for some real number a. Show that its Fourier transform Y is such that
$$\forall\, \omega \in \textit{Reals}, \quad Y(\omega) = X(\omega/a).$$

3. **T** Consider a discrete-time signal x with Fourier transform X. For each of the new signals defined below, show that its Fourier transform is as shown.

   (a) If y is such that
   $$\forall\, n \in \textit{Ints}, \quad y(n) = \begin{cases} x(n/N) & \text{if } n \text{ is an integer multiple of } N \\ 0 & \text{otherwise} \end{cases}$$
   for some integer N, then its Fourier transform Y is such that
   $$\forall\, \omega \in \textit{Reals}, \quad Y(\omega) = X(\omega N).$$

   (b) If w is such that
   $$\forall\, n \in \textit{Ints}, \quad w(n) = x(n)e^{i\alpha n},$$
   for some real number α, then its Fourier transform W is such that
   $$\forall\, \omega \in \textit{Reals}, \quad W(\omega) = X(\omega - \alpha).$$

   (c) If z is such that
   $$\forall\, n \in \textit{Ints}, \quad z(n) = x(n)\cos(\alpha n),$$
   for some real number α, then its Fourier transform Z is such that
   $$Z(\omega) = (X(\omega - \alpha) + X(\omega + \alpha))/2.$$

4. **T** Consider the FIR system described by the following block diagram:

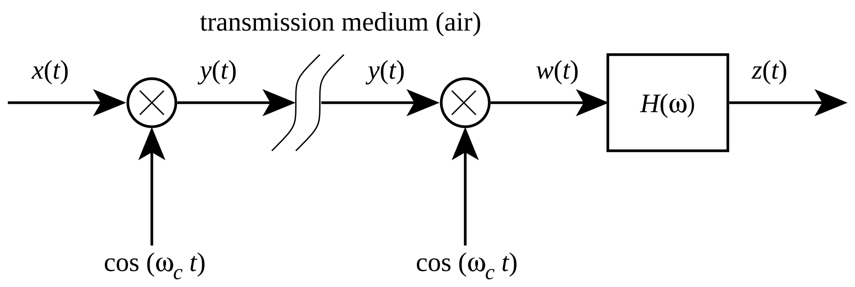
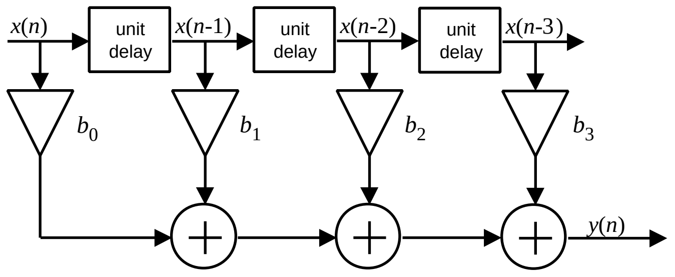


Figure 9.2: AM transmission. In the figure, attenuation and noise in the transmission medium are neglected, so the received signal is the same as the transmitted signal, y. The circular components multiply their input signals.



The notation here is the same as in figure 8.15. Suppose that this system has frequency response H. Define a new system with the identical structure as above, except that each unit delay is replaced by a double delay (two cascaded unit delays). Find the frequency response of that system in terms of H.

5. **T** This exercise discusses **amplitude modulation** or **AM**. AM is a technique that is used to convert low frequency signals into high frequency signals for transmission over a radio channel. Conversion of the high frequency signal back to a low frequency signal is called **demodulation**. The system structure is depicted in figure 9.2. The transmission medium (air, for radio signals) is approximated here as a medium that passes the signal y unaltered. Suppose your AM radio station is allowed to transmit signals at a carrier frequency of 740 kHz (this is the frequency of KCBS in San Francisco). Suppose you want to send the audio signal x : *Reals* $\rightarrow$ *Reals*. The AM signal that you would transmit is given by, for all $t \in$ *Reals*,

$$y(t) = x(t)\cos(\omega_c t),$$

where $\omega_c = 2\pi \times 740,000$ is the **carrier frequency** (in radians per second). Suppose $X(\omega)$ is the Fourier transform of an audio signal with magnitude as shown in figure9.3.

(a) Show that the Fourier transform Y of y in terms of X is

$$Y(\omega) = (X(\omega - \omega_c) + X(\omega + \omega_c))/2.$$

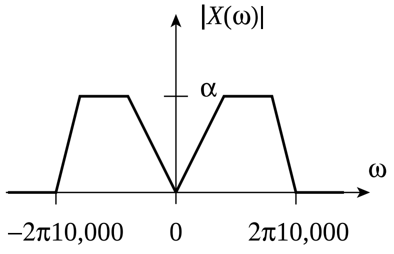


Figure 9.3: Magnitude of the Fourier transform of an example audio signal.

(b) Carefully sketch $|Y(\omega)|$ and note the important magnitudes and frequencies on your sketch.

Note that if $X(\omega) = 0$ for $|\omega| > 2\pi \times 10,000$, then $Y(\omega) = 0$ for $||\omega| - |\omega_c|| > 2\pi \times 10,000$. In words, if the signal x being modulated is bandlimited to less than 10 kHz, then the modulated signal is bandlimited to frequencies that are withing 10 kHz of the carrier frequency. Thus, an AM radio station only needs to occupy 20 kHz of the radio spectrum in order to transmit audio signals up to 10 kHz.

(c) At the receiver, the problem is to recover the audio signal x from y. One way is to demodulate by multiplying y by a sinewave at the carrier frequency to obtain the signal w, where

$$w(t) = y(t)\cos(\omega_c t).$$

What is the Fourier transform W of w in terms of X? Sketch $|W(\omega)|$ and note the important magnitudes and frequencies.

(d) After performing the demodulation of part (c), an AM receiver will filter the received signal through a low-pass filter with frequency response $H(\omega)$ such that $H(\omega) = 1$ for $|\omega| \leq 2\pi \times 10,000$ and $|H(\omega)| = 0$ for $|\omega| > 2\pi \times 20,000$. Let z be the filtered signal, as shown in figure 9.2. What is the Fourier transform Z of z? What is the relationship between z and x?

6. In the following parts, assume that x is a discrete-time signal given by

$$\forall\, n \in \textit{Ints}, \quad x(n) = \delta(n+1) + \delta(n) + \delta(n-1),$$

and that S is an LTI system with frequency response H given by

$$\forall\, \omega \in \textit{Reals}, \quad H(\omega) = e^{-i\omega}.$$

(a) Find $X = \textit{DTFT}(x)$ and make a well-labeled sketch for $\omega \in [-\pi, \pi]$ in radians/sample. Check that X is periodic with period 2π.

(b) Let $y = S(x)$. Find $Y = \textit{DTFT}(y)$.

(c) Find $y = S(x)$.

(d) Sketch x and y and comment on what the system S does.

7. Consider a causal discrete-time LTI system S with input x and output y such that

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n) + ay(n-1)$$

where $a \in \textit{Reals}$ is a given constant such that $|a| < 1$.

   (a) Find the impulse response h of S.

   (b) Find the frequency response of S by letting the input be $e^{i\omega n}$ and the output be $H(\omega)e^{i\omega n}$, and solving for $H(\omega)$.

   (c) Use your results in parts (a) and (b) and the fact that the DTFT of an impulse response is the frequency response to show that h given by

$$\forall\, n \in \textit{Ints}, \quad h(n) = a^n u(n)$$

   has the discrete-time Fourier transform $H = \textit{DTFT}(h)$ given by

$$H(\omega) = \frac{1}{1 - ae^{-i\omega}},$$

   where $u(n)$ is the unit step,

$$u(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

   (d) Use Matlab to plot $h(n)$ and $|H(\omega)|$ for $a = -0.9$ and $a = 0.9$. You may choose the interval of n for your plot of h, but you should plot $|H(\omega)|$ in the interval $\omega \in [-\pi, \pi]$. Discuss the differences between these plots for the two different values of a.

8. Suppose a discrete-time signal x has DTFT given by

$$X(\omega) = i \sin(K\omega)$$

for some positive integer K. Note that $X(\omega)$ is periodic with period 2π, as it must be to be a DTFT.

   (a) Determine from the symmetry properties of X whether the time-domain signal x is real.

   (b) Find x. **Hint**: Use Euler's relation and the linearity of the DTFT.

9. Consider a periodic continuous-time signal x with period p and Fourier series $X\colon \textit{Ints} \to \textit{Comps}$. Let y be another signal given by

$$y(t) = x(t - \tau)$$

for some real constant τ. Find the Fourier series coefficients of y in terms of those of X.

10. Consider the continuous-time signal given by

$$x(t) = \frac{\sin(\pi t/T)}{(\pi t/T)}.$$

Show that its CTFT is given by

$$X(\omega) = \begin{cases} T, & \text{if } |\omega| \le \pi/T \\ 0, & \text{if } |\omega| > \pi/T \end{cases}$$

The following fact from calculus may be useful:

$$\int_a^b e^{c\omega} c d\omega = e^{cb} - e^{ca}$$

for real a and b and complex c.

11. If x is a continuous-time signal with CTFT X, then we can define a new time-domain function y such that

$$\forall\, t \in \textit{Reals}, \quad y(t) = X(t).$$

That is, the new time domain function has the same shape as the frequency domain function X. Then the CTFT Y of y is given by

$$\forall\, \omega \in \textit{Reals}, \quad Y(\omega) = 2\pi x(-\omega).$$

That is, the frequency domain of the new function has the shape of the time domain of the old, but reversed and scaled by 2π. This property is called **duality** because it shows that time and frequency are interchangeable. Show that the property is true.

12. Use the results of exercises 10 and 11 to show that a continuous time signal x given by

$$x(t) = \begin{cases} \pi/a, & \text{if } |t| \le a \\ 0, & \text{if } |t| > a \end{cases}$$

where a is a positive real number, has CTFT X given by

$$X(\omega) = 2\pi \frac{\sin(a\omega)}{(a\omega)}.$$

Chapter 10

Sampling and Reconstruction

Digital hardware, including computers, take actions in discrete steps. So they can deal with discrete-time signals, but they cannot directly handle continuous-time signals that are prevalent in the physical world. This chapter is about the interface between these two worlds, one continuous, the other discrete. A discrete-time signal is constructed by **sampling** a continuous-time signal, and a continuous-time signal is **reconstructed** by **interpolating** a discrete-time signal.

10.1 Sampling

A sampler for complex-valued signals is a system

$$Sampler_T \colon [Reals \to Comps] \to [Ints \to Comps], \tag{10.1}$$

where T is the **sampling interval** (it has units of seconds/sample). The system is depicted in figure 10.1. The **sampling frequency** or **sample rate** is $f_s = 1/T$, in units of samples/second (or sometimes, Hertz), or $\omega_s = 2\pi/T$, in units radians/second. If $y = Sampler_T(x)$ then y is defined by

$$\forall\, n \in Ints, \quad y(n) = x(nT). \tag{10.2}$$

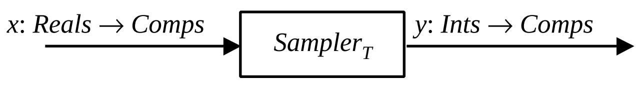


Figure 10.1: Sampler.

Basics: Units

Recall that frequency can be given with any of various units. The units of the f in (10.3) and (10.4) are Hertz, or cycles/second. In (10.3), it is sensible to give the frequency as $\omega = 2\pi f$, which has units of radians/second. The constant 2π has units of radians/cycle, so the units work out. Moreover, the time argument t has units of seconds, so the argument to the cosine function, $2\pi f t$, has units of radians, as expected.

In the discrete time case (10.4), it is sensible to give the frequency as $2\pi f T$, which has units of radians/sample. The timing interval T has units of seconds/sample, so again the units work out. Moreover, the integer n has units of samples, so again the argument to the cosine function, $2\pi f n T$, has units of radians, as expected.

In general, when discussing continuous-time signals and their sampled discrete-time signals, it is important to be careful and consistent in the units used, or considerable confusion can result. Many texts talk about **normalized frequency** when discussing discrete-time signals, by which they simply mean frequency in units of radians/sample. This is **normalized** in the sense that it does not depend on the sampling interval.

10.1.1 Sampling a sinusoid

Let $x\colon \mathit{Reals} \to \mathit{Reals}$ be the sinusoidal signal

$$\forall\, t \in \mathit{Reals}, \quad x(t) = \cos(2\pi f t), \tag{10.3}$$

where f is the frequency of the sinewave in Hertz. Let $y = \mathit{Sampler}_T(x)$. Then

$$\forall\, n \in \mathit{Ints}, \quad y(n) = \cos(2\pi f n T). \tag{10.4}$$

Although this looks similar to the continuous-time sinusoid, there is a fundamental difference. Because the index n is discrete, it turns out that the frequency f is indistinguishable from frequency $f + f_s$ when looking at the discrete-time signal. This phenomenon is called **aliasing**.

10.1.2 Aliasing

Consider another sinusoidal signal u,

$$u(t) = \cos(2\pi(f + N f_s)t),$$

where N is some integer and $f_s = 1/T$. If $N \neq 0$, then this signal is clearly different from x. Let

$$w = \mathit{Sampler}_T(u).$$

Then

$$w(n) = \cos(2\pi(f + Nf_s)nT) = \cos(2\pi fnT + 2\pi Nn) = \cos(2\pi fnT) = y(n),$$

because Nn is an integer. Thus, even though $u \neq x$, $\textit{Sampler}_T(u) = \textit{Sampler}_T(x)$. Thus, after being sampled, the signals x and u are indistinguishable. This phenomenon is called **aliasing**, presumably because it implies that any discrete-time sinusoidal signal has many continuous-time identities (its "identity" is presumably its frequency).

> **Example 10.1:** A typical sample rate for voice signals is $f_s = 8000$ Hz, so the sampling interval is $T = 0.125$ msec/sample. A continuous-time sinusoid with frequency 440 Hz, when sampled at this rate, is indistinguishable from a continuous-time sinusoid with frequency 8,440 Hz, when sampled at this same rate.

> **Example 10.2:** Compact discs are created by sampling audio signals at $f_s = 44,100$ Hz, so the sampling interval is about $T = 22.7$ μsec/sample. A continuous-time sinusoid with frequency 440 Hz, when sampled at this rate, is indistinguishable from a continuous-time sinusoid with frequency 44,540 Hz, when sampled at this same rate.

The frequency domain analysis of the previous chapters relied heavily on complex exponential signals. Recall that a cosine can be given as a sum of two complex exponentials, using Euler's relation,

$$\cos(2\pi ft) = 0.5(e^{i2\pi ft} + e^{-i2\pi ft}).$$

One of the complex exponentials is at frequency f, an the other is at frequency $-f$. Complex exponential exhibit the same aliasing behavior that we have illustrated for sinusoids.

Let $x\colon \textit{Reals} \to \textit{Comps}$ be

$$\forall\, t \in \textit{Reals}, \quad x(t) = e^{i2\pi ft}$$

where f is the frequency in Hertz. Let $y = \textit{Sampler}_T(x)$. Then for all n in *Ints*,

$$y(n) = e^{i2\pi fnT}$$

Consider another complex exponential signal u,

$$u(t) = e^{i2\pi(f+Nf_s)t}$$

where N is some integer. Let

$$w = \textit{Sampler}_T(u).$$

Then

$$w(n) = e^{i2\pi(f+Nf_s)nT} = e^{i2\pi fnT}e^{i2\pi Nf_snT} = e^{i2\pi fnT} = y(n),$$

because $e^{i2\pi Nf_snT} = 1$. Thus, as with sinusoids, when we sample a complex exponential signal with frequency f at sample rate f_s, it is indistinguishable from one at frequency $f + f_s$ (or $f + Nf_s$ for any integer N).

There is considerably more to this story. Mathematically, aliasing relates to the periodicity of the frequency domain representation (the DTFT) of a discrete-time signal. We will also see that the effects of aliasing on real-valued signals (like the cosine, but unlike the complex exponential) depend strongly on the conjugate symmetry of the DTFT as well.

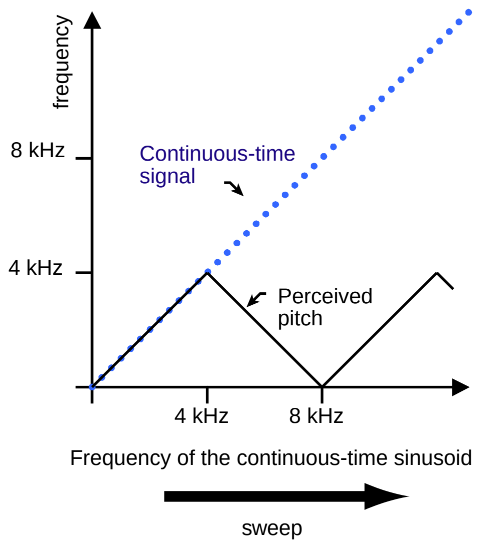


Figure 10.2: As the frequency of a continuous signal increases beyond the Nyquist frequency, the perceived pitch starts to drop.

10.1.3 Perceived pitch experiment

Consider the following experiment.[1] Generate a discrete-time audio signal with an 8 kHz sample rate according to the formula (10.4). Let the frequency f begin at 0 Hz and sweep upwards through 4 kHz to (at least) 8 kHz. Use the audio output of a computer to listen to the resulting sound. The result is illustrated in figure 10.2. As the frequency of the continuous-time sinusoid rises, so does the perceived pitch, until the frequency reaches 4 kHz. At that point, the perceived pitch begins to fall rather than rise, even as the frequency of the continuous-time sinusoid continues to rise. It will fall until the frequency reaches 8 kHz, at which point no sound is heard at all (the perceived pitch is 0 Hz). Then the perceived pitch begins to rise again.

That the perceived pitch rises from 0 after the frequency f rises above 8000 Hz is not surprising. We have already determined that in a discrete-time signal, a frequency of f is indistinguishable from a frequency $f + 8000$, assuming the sample rate is 8000 Hz. But why does the perceived pitch drop when f rises above 4 kHz?

The frequency 4 kHz, $f_s/2$, is called the **Nyquist frequency**, after Harry Nyquist, an engineer at Bell Labs who, in the 1920s and 1930s, laid much of the groundwork for digital transmission of information. The Nyquist frequency turns out to be a key threshold in the relationship between discrete-time and continuous-time signals, more important even than the sampling frequency. Intuitively, this is because if we sample a sinusoid with a frequency below the Nyquist frequency (below half the sampling frequency), then we take at least two samples per cycle of the sinusoid. It should be intuitively appealing that taking at least two samples per cycle of a sinusoid has some key

[1]This experiment can be performed at http://www.eecs.berkeley.edu/ẽal/eecs20/week13/aliasing.html. Similar experiments are carried out in lab C.11.

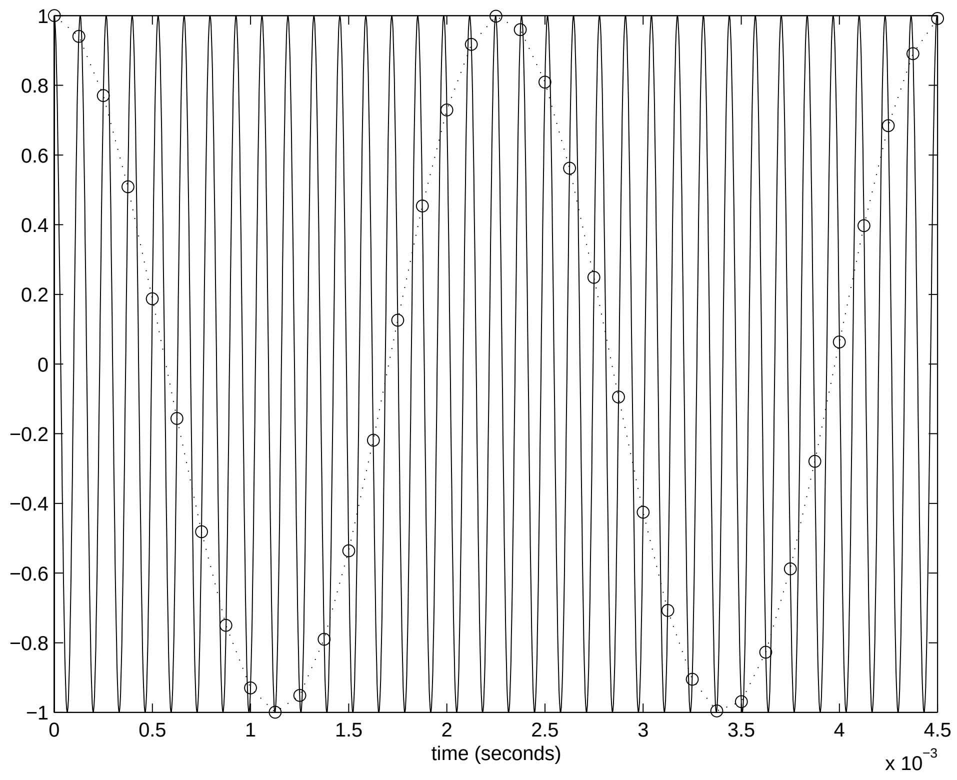


Figure 10.3: A sinusoid at 7.56 kHz and samples taken at 8 kHz.

significance. The two sample minimum allows the samples to capture the oscillatory nature of the sinusoid. Fewer than two samples would not do this. However, what happens when fewer than two samples are taken per cycle is not necessarily intuitive. It turns out that the sinusoid masquerades as one of another frequency.

Consider the situation when the frequency f of a continuous-time sinusoid is 7,560 Hz. Figure10.3 shows 4.5 msec of the continuous-time waveform, together with samples taken at 8 kHz. Notice that the samples trace out another sinusoid. We can determine the frequency of that sinusoid with the help of figure 10.2, which suggests that the perceived pitch will be $8000 - 7560 = 440$ Hz (the slope of the perceived pitch line is -1 in this region). Indeed, if we listen to the sampled sinusoid, it will be an A-440.

Recall that a cosine can be given as a sum of complex exponentials with frequencies that are negatives of one another. Recall further that a complex exponential with frequency f is indistinguishable from one with frequency $f + Nf_s$, for any integer N. A variant of figure 10.2 that leverages this representation is given in figure 10.4.

In figure 10.4, as we sweep the frequency of the continuous-time signal from 0 to 8 kHz, we move from left to right in the figure. The sinusoid consists not only of the rising frequency shown by the dotted line in figure 10.2, but also of a corresponding falling (negative) frequency as shown in figure

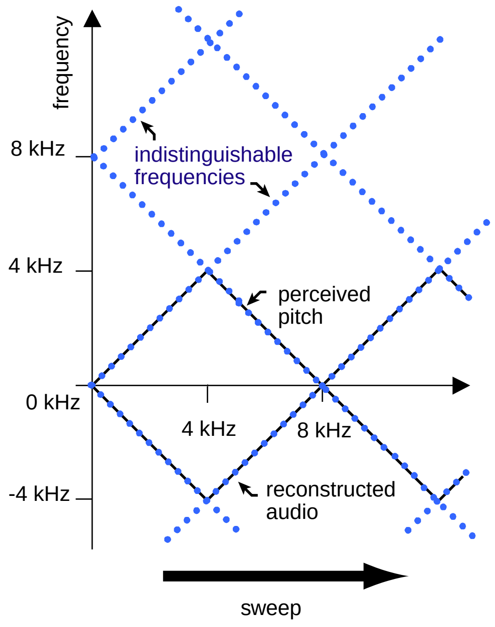


Figure 10.4: As the frequency of a continuous signal increases beyond the Nyquist frequency, the perceived pitch starts to drop because the frequency of the reconstructed continuous-time audio signal stays in the range $-f_s/2$ to $f_s/2$.

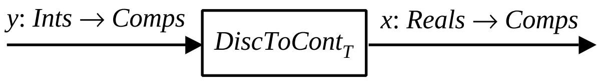


Figure 10.5: Discrete to continuous converter.

10.4. Moreover, these two frequencies are indistinguishable, after sampling, from frequencies that are 8 kHz higher or lower, also shown by dotted lines in figure 10.4.

When the discrete-time signal is converted to a continuous-time audio signal, the hardware performing this conversion can choose any matching pair of positive and negative frequencies. By far the most common choice is to select the matching pair with lowest frequency, shown in figure 10.4 by the solid lines behind dotted lines. These result in a sinusoid with frequency between 0 and the Nyquist frequency, $f_s/2$. This is why the perceived pitch falls after sweeping past $f_s/2 = 4$ kHz.

Recall that the frequency-domain representation (i.e. the DTFT) of a discrete-time signal is periodic with period 2π radians/sample. That is, if H is a DTFT, then

$$\forall\, \omega \in \textit{Reals}, \quad H(\omega) = H(\omega + 2\pi).$$

In radians per second, it is periodic with period $2\pi f_s$. In Hertz, it is periodic with period f_s, the sampling frequency. Thus, in figure 10.4, the dotted lines represent this periodicity. This periodicity is another way of stating that frequencies separated by f_s are indistinguishable.

10.1.4 Avoiding aliasing ambiguities

Figure 10.4 suggests that even though a discrete-time signal has ambiguous frequency content, it is possible to construct a uniquely defined continuous-time signal from the discrete-time waveform by choosing the one unique frequency for each component that is closest to zero. This will always result in a reconstructed signal that contains only frequencies between zero and the Nyquist frequency.

Correspondingly, this suggests that when sampling a continuous-time signal, if that signal contains only frequencies below the Nyquist frequency, then this reconstruction strategy will perfectly recover the signal. This is an intuitive statement of the **Nyquist-Shannon sampling theorem**.

Before probing this further, let us examine in more detail what we mean by reconstruction.

10.2 Reconstruction

Consider a system that constructs a continuous-time signal x from a discrete-time signal y,

$$\textit{DiscToCont}_T\colon \textit{DiscSignals} \to \textit{ContSignals}.$$

This is illustrated in figure 10.2. Systems that carry out such 'discrete-to-continuous' conversion can be realized in any number of ways. Some common examples are illustrated in figure 10.6, and defined below:

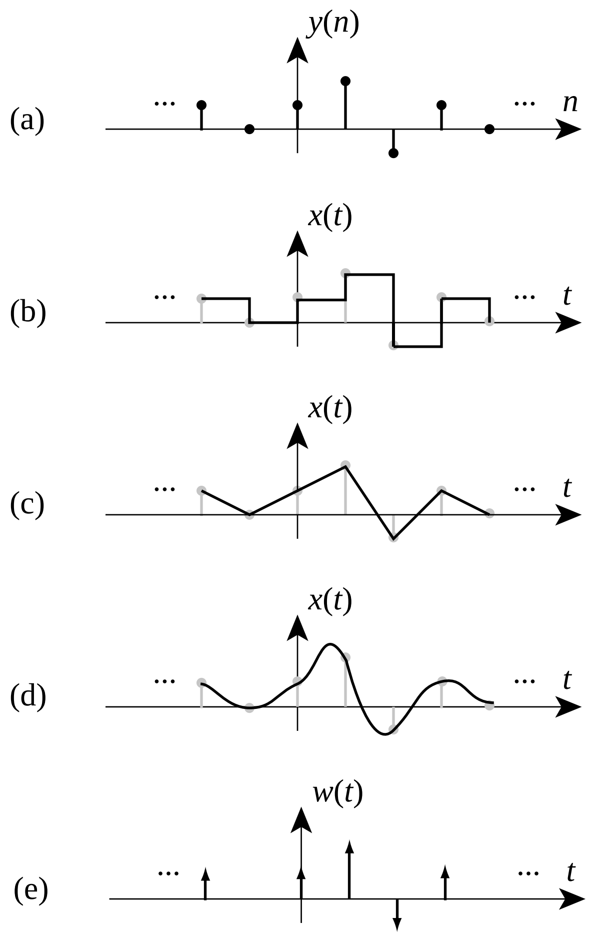


Figure 10.6: A discrete-time signal (a), a continuous-time reconstruction using zero-order hold (b), a reconstruction using linear interpolation (c), a reconstruction using ideal interpolation (d), and a reconstruction using weighted Dirac delta functions (e).

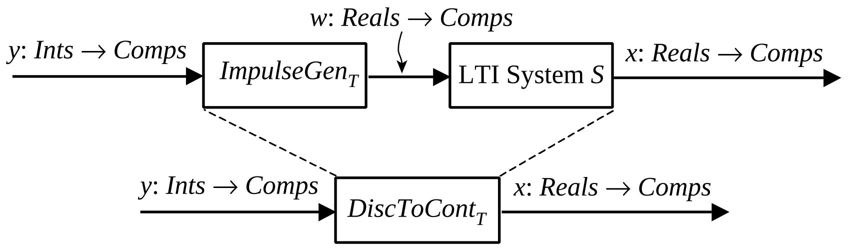


Figure 10.7: A model for reconstruction divides it into two stages.

- **zero-order hold**: This means simply that the value of the each sample $y(n)$ is held constant for duration T, so that $x(t) = y(n)$ for the time interval from $t = nT$ to $t = (n+1)T$, as illustrated in figure 10.6(b). Let this system be denoted

$$\textit{ZeroOrderHold}_T \colon \textit{DiscSignals} \to \textit{ContSignals}.$$

- **linear interpolation**: Intuitively, this means simply that we connect the dots with straight lines. Specifically, in the time interval from $t = nT$ to $t = (n+1)T$, $x(t)$ has values that vary along a straight line from $y(n)$ to $y(n+1)$, as illustrated in figure 10.6(c). Linear interpolation is sometimes called **first-order hold**. Let this system be denoted

$$\textit{LinearInterpolator}_T \colon \textit{DiscSignals} \to \textit{ContSignals}.$$

- **ideal interpolation**: It is not yet clear what this should mean, but intuitively, it should result in a smooth curve that passes through the samples, as illustrated in figure 10.6(d). We will give a precise meaning below. Let this system be denoted

$$\textit{IdealInterpolator}_T \colon \textit{DiscSignals} \to \textit{ContSignals}.$$

10.2.1 A model for reconstruction

A convenient mathematical model for reconstruction divides the reconstruction process into a cascade of two systems, as shown in figure 10.7. Thus

$$x = S(\textit{ImpulseGen}_T(y)),$$

where S is an LTI system to be determined. The first of these two subsystems,

$$\textit{ImpulseGen}_T \colon \textit{DiscSignals} \to \textit{ContSignals},$$

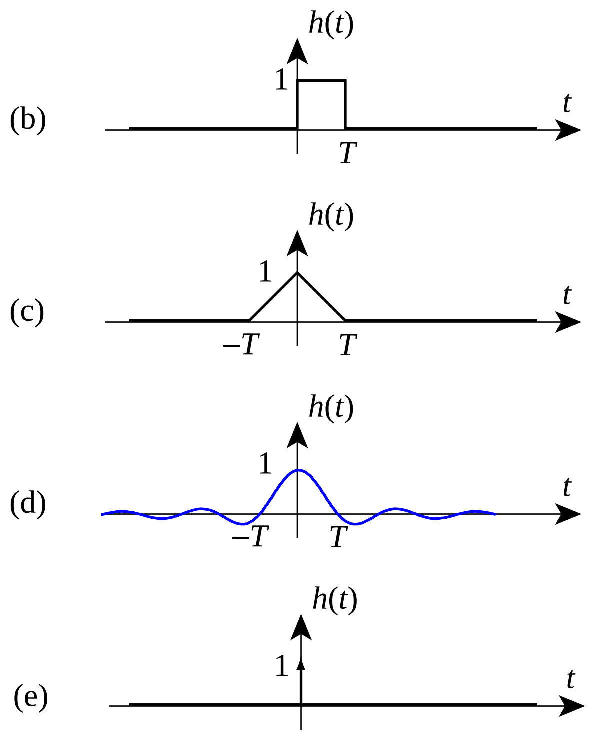


Figure 10.8: The impulse responses for the LTI system S in figure 10.7 that yield the interpolation methods in figure 10.6(b-e).

constructs a continuous-time signal, where for all $t \in$ *Reals*,

$$w(t) = \sum_{k=-\infty}^{\infty} y(k)\delta(t - kT).$$

This is a continuous-time signal that at each sampling instant kT produces a Dirac delta function with weight equal to the sample value, $y(k)$. This signal is illustrated in figure 10.6(e). It is a mathematical abstraction, since everyday engineering systems do not exhibit the singularity-like behavior of the Dirac delta function. Nonetheless, it is a useful mathematical abstraction.

The second system, S, is a continuous-time LTI filter with an impulse response that determines the interpolation method. The impulse responses that yield the interpolation methods in figure10.6(b-e) are shown in figure 10.8(b-e). If

$$h(t) = \begin{cases} 1 & 0 \le t < T \\ 0 & otherwise \end{cases}$$

then the interpolation method is zero-order hold. If

$$h(t) = \begin{cases} 1 + t/T & -T < t < 0 \\ 1 - t/T & 0 \le t < T \\ 0 & otherwise \end{cases}$$

then the interpolation method is linear. If the impulse response is

$$h(t) = \frac{\sin(\pi t/T)}{\pi t/T}$$

then the interpolation method is ideal. The above impulse response is called a **sinc function**, and it is characterized by having no frequency components above $f_s/2$ Hz, the Nyquist frequency. It is this characteristic that makes it ideal. It precisely performs the strategy illustrated in figure10.4, where among all indistinguishable frequencies we select the ones between $-f_s/2$ and $f_s/2$.

If we let Sinc_T denote the LTI system S when the impulse response is a sinc function, then

$$\mathit{IdealInterpolator}_T = \mathit{Sinc}_T \circ \mathit{ImpulseGen}_T.$$

In practice, ideal interpolation is difficult to accomplish. From the expression for the sinc function we can understand why. First, this impulse response is not causal. Second, it is infinite in extent. More importantly, its magnitude decreases rather slowly as t increases (proportional to $1/t$ only). Thus, truncating it at finite length leads to substantial errors.

If the impulse response of S is

$$h(t) = \delta(t),$$

where δ is the Dirac delta function, then the system S is a pass-through system, and the reconstruction consists of weighted delta functions.

10.3 The Nyquist-Shannon sampling theorem

We can now give a precise statement of the **Nyquist-Shannon sampling theorem**:

If x is a continuous-time signal with Fourier transform X and if $X(\omega)$ is zero outside the range $-\pi/T < \omega < \pi/T$, then

$$x = \mathit{IdealInterpolator}_T(\mathit{Sampler}_T(x)).$$

We can state this theorem slightly differently. Suppose x is a continuous-time signal with no frequency larger than some f_0. Then x can be recovered from its samples if $f_0 < f_s/2$, the Nyquist frequency.

A formal proof of this theorem involves some technical difficulties (it was first given by Claude Shannon of Bell Labs in the late 1940s). But we can get the idea from the following three-step argument. See figure 10.9.

Step 1. Let x be a continuous-time signal with Fourier transform X. At this point we do not require that $X(\omega)$ is zero outside the range $-\pi/T < \omega < \pi/T$. We sample x with sampling interval T to get the discrete-time signal

$$y = \mathit{Sampler}_T(x).$$

Probing further: Sampling

We can construct a mathematical model for sampling by using Dirac delta functions. Define a pulse stream by

$$p(t) = \sum_{k=-\infty}^{\infty} \delta(t - kT).$$

Consider a continuous-time signal x that we wish to sample with sampling period T. That is, we define $y(n) = x(nT)$. Construct first an intermediate continuous-time signal $w(t) = x(t)p(t)$. We can show that the CTFT of w is equal to the DTFT of y. This gives us a way to relate the CTFT of x to the DTFT of its samples y. Recall that multiplication in the time domain results in convolution in the frequency domain, so

$$W(\omega) = \frac{1}{2\pi} X(\omega) * P(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\Omega) P(\omega - \Omega) d\Omega.$$

It can be shown that the CTFT of $p(t)$ is

$$P(\omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\omega - k\frac{2\pi}{T}),$$

so

$$\begin{aligned} W(\omega) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\Omega) \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\omega - \Omega - k\frac{2\pi}{T}) d\Omega \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} \int_{-\infty}^{\infty} X(\Omega) \delta(\omega - \Omega - k\frac{2\pi}{T}) d\Omega \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} X(\omega - k\frac{2\pi}{T}) \end{aligned}$$

where the last equality follows from the sifting property (8.11). The next step is to show that

$$Y(\omega) = W(\omega/T).$$

We leave this as an exercise. From this, the basic Nyquist-Shannon result follows,

$$Y(\omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X\left(\frac{\omega - 2\pi k}{T}\right).$$

This relates the CTFT of the signal being sampled to the DTFT of the discrete-time result.

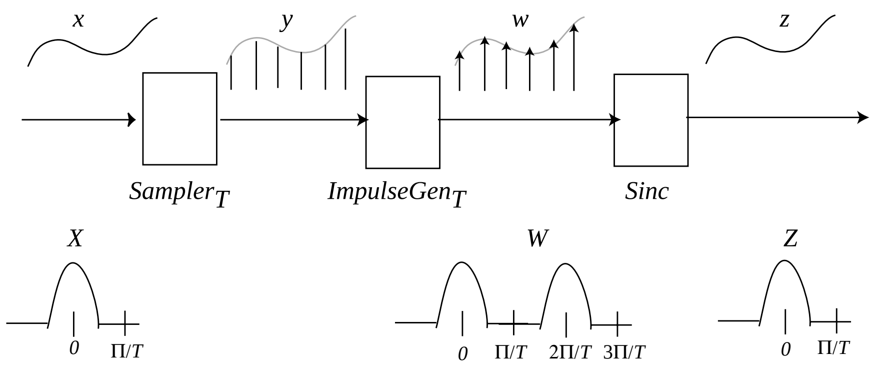


Figure 10.9: The different steps in the Nyquist-Shannon theorem.

It can be shown (see box on page 278) that the DTFT of y is related to the CTFT of x by

$$\boxed{Y(\omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X\left(\frac{\omega - 2\pi k}{T}\right).}$$

This important relation says that the DTFT Y of y is the sum of the CTFT X with copies of it shifted by multiples of $2\pi/T$. There are two cases to consider.

First, if $X(\omega) = 0$ outside the range $-\pi/T < \omega < \pi/T$, then the copies will not overlap, and in the range $-\pi < \omega < \pi$,

$$Y(\omega) = \frac{1}{T} X\left(\frac{\omega}{T}\right). \tag{10.5}$$

In this range of frequencies, Y has the same shape as X, scaled by $1/T$. This relationship between X and Y is illustrated in figure 10.3, where X is drawn with a triangular shape.

In the second case, illustrated on the top of figure 10.11, X does have non-zero frequency components higher than π/T. Notice that in the sampled signal, the frequencies in the vicinity of π are distorted by the overlapping of frequency components above and below π/T in the original signal. This distortion is called **aliasing distortion**.

We continue with the remaining steps, following the signals in figure 10.9.

Step 2. Let w be the signal produced by the impulse generator,

$$\forall\, t \in \textit{Reals}, \quad w(t) = \sum_{n=-\infty}^{\infty} y(n)\delta(t - nT).$$

(The Fourier Transform of w is $W(\omega) = Y(\omega T)$. See box on page 278.)

Let z be the output of the *IdealInterpolator*$_T$. Its Fourier transform is simply

$$\begin{aligned} Z(\omega) &= W(\omega)H(\omega) \\ &= Y(\omega T)H(\omega), \end{aligned}$$

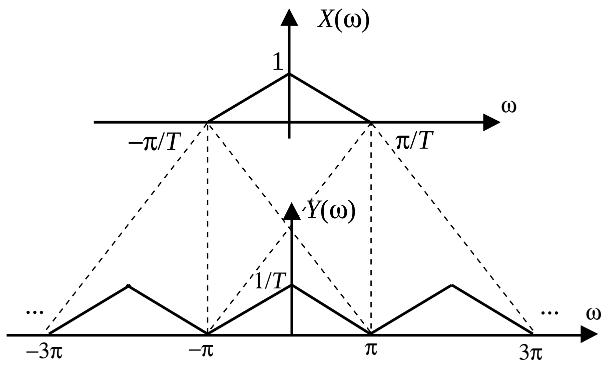


Figure 10.10: Relationship between the CTFT of a continuous-time signal and the DTFT of its discrete-time samples. The DTFT is the sum of the CTFT and its copies shifted by multiples of $2\pi T$, the sampling frequency in radians per second.

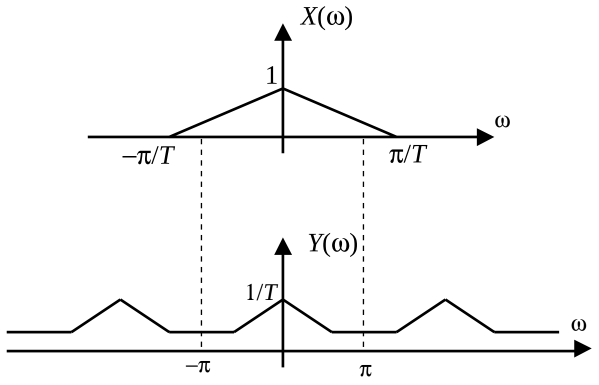


Figure 10.11: Relationship between the FT of a continuous-time signal and the DTFT of its discrete-time samples when the continuous-time signal has a broad enough bandwidth to introduce aliasing distortion.

where $H(\omega)$ is the frequency response of the reconstruction filter *Sinc*$_T$.

But as seen in exercise 10 of chapter 9,

$$H(\omega) = \begin{cases} T & -\pi/T < \omega < \pi/T \\ 0 & \text{otherwise} \end{cases} \tag{10.6}$$

Substituting for H and Y, we get

$$\begin{aligned} Z(\omega) &= \begin{cases} TY(\omega T) & -\pi/T < \omega < \pi/T \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} \sum\limits_{k=-\infty}^{\infty} X(\omega - 2\pi kT) & -\pi/T < \omega < \pi/T \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Step 3. If $X(\omega)$ is zero for $|\omega|$ larger than the Nyquist frequency π/T, then we conclude that

$$\forall\, \omega \in \textit{Reals}, \quad Z(\omega) = X(\omega).$$

That is, w is identical to x; this proves the Nyquist-Shannon result. This is the case illustrated in figures 10.3 and 10.9.

However, if $X(\omega)$ does have non-zero values for some $|\omega|$ larger than the Nyquist frequency, then z will be different from x, as illustrated in figure 10.11.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. Consider the continuous-time signal

$$x(t) = \cos(10\pi t) + \cos(20\pi t) + \cos(30\pi t).$$

   (a) Find the fundamental frequency. Give the units.

   (b) Find the Fourier series coefficients $A_0, A_1, \cdots$ and $\phi_1, \phi_2, \cdots$.

   (c) Let y be the result of sampling this signal with sampling frequency 10 Hz. Find the fundamental frequency for y, and give the units.

   (d) For the same y, find the discrete-time Fourier series coefficients, $A_0, A_1, \cdots$ and $\phi_1, \cdots$.

   (e) Find

$$w = \textit{IdealInterpolator}_T(\textit{Sampler}_T(x))$$

   for $T = 0.1$ seconds.

(f) Is there any aliasing distortion caused by sampling at 10 Hz? If there is, describe the aliasing distortion in words.

(g) Give the smallest sampling frequency that avoids aliasing distortion.

2. **E** Verify that *Sampler*$_T$ defined by (10.1) and (10.2) is linear but not time invariant.

3. **E** A real-valued sinusoidal signal with a negative frequency is always exactly equal to another sinusoid with positive frequency. Consider a real-valued sinusoid with a negative frequency -440 Hz,
$$y(n) = \cos(-2\pi 440 nT + \phi).$$
Find a positive frequency f and phase θ such that
$$y(n) = \cos(2\pi f nT + \theta).$$

4. **T** Consider a continuous-time signal x where for all $t \in$ *Reals*,
$$x(t) = \sum_{k=-\infty}^{\infty} r(t-k).$$
where
$$r(t) = \begin{cases} 1 & 0 \le t < 0.5 \\ 0 & otherwise \end{cases}.$$

   (a) Is $x(t)$ periodic? If so, what is the period?

   (b) Suppose that $T = 1$. Give a simple expression for $y =$ *Sampler*$_T(x)$.

   (c) Suppose that $T = 0.5$. Give a simple expression for $y =$ *Sampler*$_T(x)$ and $z =$ *IdealInterpolator*$_T$(*Sampler*$_T(x)$).

   (d) Find an upper bound for T (in seconds) such that $x =$ *IdealInterpolator*$_T$(*Sampler*$_T(x)$), or argue that no value of T makes this assertion true.

5. **T** Consider a continuous-time signal x with the following finite Fourier series expansion,
$$\forall\, t \in \textit{Reals}, \quad x(t) = \sum_{k=0}^{4} \cos(k\omega_0 t)$$
where $\omega_0 = \pi/4$ radians/second.

   (a) Give an upper bound on T (in seconds) such that $x =$ *IdealInterpolator*$_T$(*Sampler*$_T(x)$).

   (b) Suppose that $T = 4$ seconds. Give a simple expression for $y =$ *Sampler*$_T(x)$.

   (c) For the same $T = 4$ seconds, give a simple expression for
$$w = \textit{IdealInterpolator}_T(\textit{Sampler}_T(x)).$$

6. **T** Consider a continuous-time audio signal x with CTFT shown in figure 10.12. Note that it contains no frequencies beyond 10 kHz. Suppose it is sampled at 40 kHz to yield a signal that we will call x_{40}. Let X_{40} be the DTFT of x_{40}.

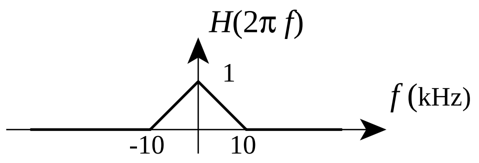


Figure 10.12: CTFT of an audio signal considered in exercise 6.

(a) Sketch $|X_{40}(\omega)|$ and carefully mark the magnitudes and frequencies.

(b) Suppose x is sampled at 20 kHz. Let x_{20} be the resulting sampled signal and X_{20} its DTFT. Sketch and compare x_{20} and x_{40}.

(c) Now suppose x is sampled at 15 kHz. Let x_{15} be the resulting sampled signal and X_{15} its DTFT. Sketch and compare X_{20} and X_{15}. Make sure that your sketch shows aliasing distortion.

7. **C** Consider two continuous-time sinusoidal signals given by

$$x_1(t) = \cos(\omega_1 t)$$

$$x_2(t) = \cos(\omega_2 t),$$

with frequencies ω_1 and ω_2 radians/second such that

$$0 \le \omega_1 \le \pi/T \quad \text{and} \quad 0 \le \omega_2 \le \pi/T.$$

Show that if $\omega_1 \neq \omega_2$ then

$$\textit{Sampler}_T(x_1) \neq \textit{Sampler}_T(x_2).$$

I.e., the two distinct sinusoids cannot be aliases of one another if they both have frequencies below the Nyquist frequency. **Hint**: Try evaluating the sampled signals at $n = 1$.

Appendix A

Sets and Functions

This appendix establishes the notation of sets as used in the book. It reviews the use of this mathematical language to describe sets in a variety of ways, to combine sets using the operations of union, intersection, and product, and to derive logical consequences. We also review how to formulate and understand predicates, and we define certain sets that occur frequently in the study of signals and systems. Finally, we review functions.

A.1 Sets

A **set** is a collection of **elements**. The set of **natural numbers**, for example, is the collection of all positive integers. This set is denoted (identified) by the name *Nats*,

$$\boxed{\mathit{Nats} = \{1, 2, 3, \cdots\}.} \tag{A.1}$$

In (A.1) the left hand side is the name of the set and the right hand side is an enumeration or list of all the elements of the set. We read (A.1) as '*Nats* is the set consisting of the numbers $1, 2, 3,$ and so on.' The ellipsis '$\cdots$' means 'and so on'. Because *Nats* is an infinite set we cannot enumerate all its elements, and so we have to use ellipsis. For a finite set, too, we may use ellipsis as a convenient shorthand as in

$$A = \{1, 2, 3, \cdots, 100\}, \tag{A.2}$$

which defines A to be the set consisting of the first 100 natural numbers. *Nats* and A are sets of numbers. The concept of sets is very general, as the examples below illustrate.

Students is the set of all students in this class, each element of which is referenced by a student's name:

$$\mathit{Students} = \{\text{John Brown}, \text{Jane Doe}, \text{Jie Xin Zhou}, \cdots\}.$$

USCities consists of all cities in the U.S., referenced by name:

$$\mathit{USCities} = \{\text{Albuquerque}, \text{San Francisco}, \text{New York}, \cdots\}.$$

BooksInLib comprises all books in the U.C. Berkeley library, referenced by a 4-tuple (first author's last name, book title, publisher, year of publication):

$$\begin{aligned} \textit{BooksInLib} \;=\; & \{(\text{Lee}, \textit{Digital Communication}, \text{Kluwer}, 1994), \\ & (\text{Walrand}, \textit{Communication Networks}, \text{MorganKaufmann}, 1996), \cdots\}. \end{aligned}$$

BookFiles consists of all LaTeX documents for this book, referenced by their file name:

$$\textit{BookFiles} = \{\text{sets.tex}, \text{functions.tex}, \cdots\}.$$

We usually use either italicized, capitalized names for sets, such as *Reals* and *Ints*, or single capital letters, such as A, B, X, Y.

An element of a set is also said to be a **member** of the set. The number 10 is a member of the set A defined in (A.2), but the number 110 is not a member of A. We express these two facts by the two expressions:

$$10 \in A, \quad 110 \notin A.$$

The symbol '$\in$' is read 'is a member of' or 'belongs to' and the symbol '$\notin$' is read 'is not a member of' or 'does not belong to.'

When we define a set by enumerating or listing all its elements, we enclose the list in curly braces $\{\cdots\}$. It is not always necessary to give the set a name. We can instead refer to it directly by enumerating its elements. Thus

$$\{1, 2, 3, 4, 5\}$$

is the set consisting of the numbers $1, 2, \cdots, 5$.

The order in which the elements of the set appear in the list is not (usually) significant. When the order is significant, the set is called an **ordered set**.

An element is either a member of a set or it is not. It cannot be a member more than once. So for example, $\{1, 2, 1\}$ is not a set.

Thus a set is defined by an unordered collection of its elements, without repetition. Two sets are equal if and only if every element of the first set is also an element of the second set, and every element of the second set is a member of the first set. So if $B = \{1, 2, 3, 4, 5\}$ and $C = \{5, 3, 4, 1, 2\}$, then it is correct to state that

$$B = C. \tag{A.3}$$

A.1.1 Assignment and assertion

Although the expressions (A.1) and (A.3) are both in the form of equations, the "=" in these two expressions have very different meanings. Expression (A.1) (as well as (A.2)) is an **assignment**: the set on the right-hand side is assigned to the name *Nats* on the left-hand side. Expression (A.3) is an **assertion**, which is an expression that can be true or false. In other words, an assertion is an expression that has a truth value. Thus (A.3) asserts that the two sides are equal. Since this is true, (A.3) is a true assertion. But the assertion

$$\textit{Nats} = A$$

is a false assertion. An assertion is true or false, while an assignment is a tautology (something that is trivially true because the definition makes it so). Some notation systems make a distinction between an assignment and an assertion by writing an assignment using the symbol ":=" instead of "=" as in

$$\mathit{Nats} := \{1, 2, 3, \cdots\}.$$

Other notation systems use "=" for assignments and '==' for assertions. We will not make these notational distinction, since it will be clear from the context whether an expression is an assignment or an assertion.[1]

Note that context is essential in order to disambiguate whether an expression like $\mathit{MyNumbers} = \{1, 3, 5\}$ is an assertion or an assignment. Thus, for example, in the context,

Define the set *MyNumbers* by $\mathit{MyNumbers} = \{1, 3, 5\}$,

The expression is clearly an assignment, as indicated by "Define the set $\cdots$". However, consider the following context:

If we define *MyNumbers* by $\mathit{MyNumbers} = \{1, 3, 5\}$, then $\mathit{MyNumbers} = \{3, 5, 1\}$.

The first "=" is an assignment, but in the second is an assertion.

A.1.2 Sets of sets

Anything can be an element of a set, so of course, sets can be elements of sets. Suppose, for example, that X is a set. Then we can construct the set of all subsets of X, which is written $P(X)$ and is called the **power set** of X. Notice that since $\emptyset \subset X$, then $\emptyset \in P(X)$.

A.1.3 Variables and predicates

We can refer to a particular element of a set by using its name, for example the element 55 in *Nats*, or Berkeley in *USCities*. We often need to refer to a general element in a set. We do this by using a **variable**. We usually use lower case letters for variable names, such as x, y, n, t. Thus $n \in \mathit{Nats}$ refers to any natural number. *Nats* is the **range** of the variable n. We may also use a character string for a variable name such as $\mathit{city} \in \mathit{USCities}$. We say that '$n$ is a variable over *Nats*' and '*city* is a variable over *USCities*'.

A variable can be assigned (or substituted by) any value in its range. Thus the assignment $n = 5$ assigns the value 5 to n, and $\mathit{city} =$ Berkeley assigns the value Berkeley to *city*.

Observe again that the use of "=" in an expression like $n = 5$ is ambiguous, because the expression could be an assignment or an assertion, depending on the context. Thus, for example, in the context,

[1] A symbol such as "=" which has more than one meaning depending on the context is said to be **overloaded**. C++ uses overloaded symbols, while Java does not (Java does support overloaded methods, however). People often have difficulty reading other people's code written in a language that permits overloading.

Probing further: Predicates in Matlab

In Matlab, “=” is always used as an assignment, while “==” is used to express an assertion. Thus the Matlab program,

```
n = 5,   m = 6,   k = 2,   m = k+n
```

returns

```
n = 5,   m = 6,   k = 2,   m =7
```

because the expression `m=k+n` assigns the value 7 to `m`. However, the Matlab program

```
n = 5,   m = 6,   k = 2,   m == k+n
```

returns

```
n = 5,   m = 6,   k = 2,   ans = 0
```

where `ans` = 0 signifies that the assertion `m == k+n` evaluates to false.

Let $n = 5, m = 6, k = 2$, then $m = k + n$

the first three “=” are assignments, but the last is an assertion, which happens to be false.

We can use variables to form expressions such as $n \leq 5$. Now, when we assign a particular value to n in this expression, the expression become an assertion that evaluates to true or false. (For instance, $n \leq 5$ evaluates to true for $n = 3$ and to false for $n = 6$.) An expression such as $x \leq 5$ which evaluates to true or false when we assign the variable a specific value is called a **predicate** (in x).[2] Predicates are widely used to define new sets from old ones, as in this example:

$$B = \{x \in \mathit{Nats} \mid x \leq 5\}$$

which reads “B is the set of all elements x in *Nats* such that (the symbol ‘|’ means ‘such that’) the predicate $x \leq 5$ is true.” More generally, we use the following prototype expression to define a new set *NewSet* from an old set *Set*

$$\boxed{\mathit{NewSet} = \{x \in \mathit{Set} \mid \mathit{Pred}(x)\}.} \tag{A.4}$$

[2] A specific value is said to **satisfy** a predicate if it evaluates to true, and **not to satisfy** the predicate if the predicate evaluates to false.

In this prototype expression for defining sets, $x \in \mathit{Set}$ means that x is a variable over the set *Set*, $\mathit{Pred}(x)$ is a predicate in x, and so *NewSet* is the set consisting of all elements x in *Set* for which $\mathit{Pred}(x)$ evaluates to true.

We illustrate the use of the prototype (A.4). In the examples below, note that the concept of predicate is very general. In essence, a predicate is a condition involving any attributes or properties of the elements of *Set*. Consider

$$\mathit{TallStudents} = \{\mathit{name} \in \mathit{Students} \mid \mathit{name} \text{ is more than 6 feet tall}\}.$$

Here the predicate is "*name* is more than 6 feet tall." If John Brown's height is 5' 10" and Jane Doe is 6'1" tall, the predicate evaluates to true for *name* = Jane Doe, and to false for *name* = John Brown, so Jane Doe $\in$ *TallStudents* and John Brown $\notin$ *TallStudents*. Consider

$$\mathit{NorthCities} = \{\mathit{city} \in \mathit{USCities} \mid \mathit{city} \text{ is located north of Washington DC}\}.$$

The predicate here evaluates to true for *city* = New York, and to false for *city* = Atlanta.

The variable name "x" used in (A.4) is not significant. The sets $\{x \in \mathit{Nats} \mid x \geq 5\}$ and $\{n \in \mathit{Nats} \mid n \geq 5\}$ are the *same* sets even though the variable names used in the predicates are different. This is because the predicates '$n \geq 5$' and '$x \geq 5$' both evaluate to true or both evaluate to false when the same value is assigned to n and x. We say that the variable name x used in (A.4) is a **dummy variable** since the meaning of the expression is unchanged if we substitute x with another variable name, say y. You are already familiar with the use of dummy variables in integrals. The two integrals below evaluate to the same number:

$$\int_0^1 x^2 dx = \int_0^1 y^2 dy.$$

A.1.4 Quantification over sets

Consider the set $A = \{1, 3, 4\}$. Suppose we want to make the assertion that every element of A is smaller than 7. We could do this by the three expressions

$$1 < 7, \quad 3 < 7, \quad 4 < 7,$$

which gets to be very clumsy if A has many more elements. So mathematicians have invented a shorthand. The idea is to be able to say that $x < 7$ for every value that the variable x takes in the set A. The precise expression is

$$\forall x \in A, \quad x < 7. \tag{A.5}$$

The symbol '$\forall$' reads 'for all', so the expression reads, "for all values of x in A, $x < 7$." The phrase $\forall x \in A$ is called **universal quantification**. Note that in the expression (A.5), x is again a dummy variable; the meaning of the expression is unchanged if we use another variable name. Note, also, that (A.5) is an assertion which, in this case, evaluates to true. However, the assertion

$$\forall x \in A, \quad x > 3$$

is false, since for at least one value of $x \in A$, namely $x = 1$, $x > 3$ is false.

Suppose we want to say that there is at least one element in A that is larger than 3. We can say this using **existential quantification** as in

$$\exists x \in A, \quad x > 3. \tag{A.6}$$

The symbol '$\exists$' reads 'there exists', so the expression (A.6) reads 'there exists a value of x in A, $x > 3$'. Once again, any other variable name could be used in place of x, and the meaning of the assertion is unchanged.

In general, the expression,

$$\boxed{\forall x \in \mathit{Set}, \quad \mathit{Pred}(x),} \tag{A.7}$$

is an assertion that evaluates to true if $\mathit{Pred}(x)$ evaluates to true for every value of $x \in A$, and

$$\boxed{\exists x \in A, \quad \mathit{Pred}(x),} \tag{A.8}$$

is an assertion that evaluates to true if $\mathit{Pred}(x)$ evaluates to true for at least one value of $x \in A$.

Conversely, the assertion (A.7) evaluates to false if $\mathit{Pred}(x)$ evaluates to false for at least one value of $x \in A$, and the assertion (A.8) evaluates to false if $\mathit{Pred}(x)$ evaluates to false for every value of $x \in A$.

We can use these two quantifiers to define sets using the prototype new set constructor (A.4). For example

$$\mathit{EvenNumbers} = \{n \in \mathit{Nats} \mid \exists k \in \mathit{Nats}, \ n = 2k\}$$

is the set of all even numbers, since the predicate in the variable n,

$$\exists k \in \mathit{Nats}, \quad n = 2k,$$

evaluates to true only if n is even.

A.1.5 Some useful sets

The following sets are frequently used in this text:

$\mathit{Nats} =$	$\{1, 2, \cdots\}$	natural numbers
$\mathit{Nats}_0 =$	$\{0, 1, 2, \cdots\}$	non-negative integers
$\mathit{Ints} =$	$\{\cdots, -2, -1, 0, 1, 2, \cdots\}$	integers
$\mathit{Ints}_+ =$	$\{0, 1, 2, \cdots\}$	non-negative integers, same as Nats_0
$\mathit{Reals} =$	$(-\infty, \infty)$	real numbers
$\mathit{Reals}_+ =$	$[0, \infty)$	non-negative real numbers
$\mathit{Comps} =$	$\{x + jy \mid x, y \in \mathit{Reals}\}$	complex numbers, $j = \sqrt{-1}$

If α, β are real numbers, then

$$\boxed{\begin{array}{rcl} [\alpha, \beta] & = & \{x \in \mathit{Reals} \mid \alpha \le x \le \beta\} \\ (\alpha, \beta) & = & \{x \in \mathit{Reals} \mid \alpha < x < \beta\} \\ (\alpha, \beta] & = & \{x \in \mathit{Reals} \mid \alpha < x \le \beta\} \\ (-\infty, \infty) & = & \mathit{Reals} \end{array}}$$

Note the meaning of the difference in notation: both end-points α and β are included in $[\alpha, \beta]$; we say that $[\alpha, \beta]$ is a **closed** interval; neither end-point is included in (α, β); we call this an **open** interval; the interval $(\alpha, \beta]$ is said to be half-open, half-closed. Whether an end-point is included in an interval or not is indicated by the use of $[,]$ or $(,)$.

Other useful sets are:

Bin $=$	$\{0, 1\}$, the binary values
Bin* $=$	$\{0, 1\}^*$, set of all finite binary strings
Char $=$	set of all alphanumeric characters
Char* $=$	set of all finite character strings

A.1.6 Set operations: union, intersection, complement

Let $A = \{1, 2, \cdots, 10\}$, and $B = \{1, \cdots, 5\}$. Clearly B is included in A, and A is included in *Nats*. We express these assertions as:

$$B \subset A, \quad A \subset \mathit{Nats},$$

which we read "B is contained by or included in A" and "A is contained by or included in *Nats*." For the sets above,

$$\mathit{Nats} \subset \mathit{Nats}_0 \subset \mathit{Ints} \subset \mathit{Reals} \subset \mathit{Comps}.$$

If A and B are sets, then $A \cap B$ is the set consisting of all elements that are in both A and B, and $A \cup B$ is the set consisting of all elements that are either in A or in B or in both A and B. $A \cap B$ is called the **intersection** of A and B, and $A \cup B$ is called the **union** of A and B. We can express these definitions using variables as:

$$\boxed{A \cap B = \{x \mid x \in A \wedge x \in B\}, \quad A \cup B = \{x \mid x \in A \vee x \in B\}}$$

where $\wedge$ is the notation for the logical **and** and $\vee$ is the symbol for the logical **or**. The predicate "$x \in A \wedge x \in B$" reads "x is a member of A and x is a member of B"; "$x \in A \vee x \in B$" reads "x is a member of A or x is a member of B." The logical and of two predicates is also called their **conjunction** and their logical or is also called their **disjunction**. The symbols $\wedge$ and $\vee$ are called **logical connectives**.

If A, X are sets and $A \subset X$, then $X \setminus A$ is the set consisting of all elements in X that are not in A. $X \setminus A$ is called the **complement** of A in X. When X is understood, we write A^c instead of $X \setminus A$.

We gain some intuitive understanding by depicting sets and set operations using pictures. Figure A.1 illustrates union, intersection, and complement.

A.1.7 Predicate operations

Given two predicates $P(x)$ and $Q(x)$ we can form their conjunction $P(x) \wedge Q(x)$, and their disjunction $P(x) \vee Q(x)$. These predicate operations correspond to the set operations of intersection

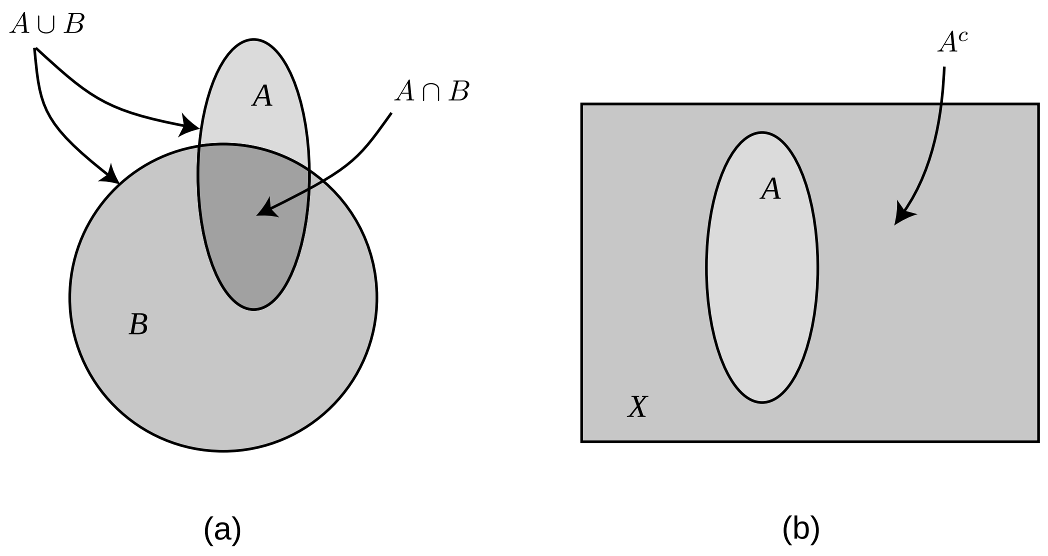


Figure A.1: (a) Union and intersection. (b) Set complement.

and union:

$$\boxed{\begin{array}{rcl}\{x \in X \mid P(x) \wedge Q(x)\} & = & \{x \in X \mid P(x)\} \cap \{x \in X \mid Q(x)\} \\ \{x \in X \mid P(x) \vee Q(x)\} & = & \{x \in X \mid P(x)\} \cup \{x \in X \mid Q(x)\}.\end{array}}$$

Note the visual similarity between $\wedge$ and $\cap$, and between $\vee$ and $\cup$.

The counterpart of the complement of a set is the **negation** of a predicate. We denote by $\neg \mathit{Pred}(x)$ the predicate that evaluates to false for any value of x for which $\mathit{Pred}(x)$ evaluates to true, and that evaluates to true for any value of x for which $\mathit{Pred}(x)$ evaluates to false. We read '$\neg \mathit{Pred}(x)$' as 'not $\mathit{Pred}(x)$' or the "negation of $\mathit{Pred}(x)$." For example,

$$\{n \in \mathit{Nats} \mid \neg(n < 5)\} = \{5, 6, 7, \cdots\},$$

since $\neg(n < 5)$ evaluates to true if and only if $n < 5$ evaluates to false, which happens if and only if n is larger than or equal to 5.

In general we have the following correspondence between predicate negation and set complements:

$$\{x \in X \mid \neg \mathit{Pred}(x)\} = \{x \in X \mid \mathit{Pred}(x)\}^c. \tag{A.9}$$

We can combine (A.1.7), (A.1.7), and (A.9) to obtain more complex identities. If $P(x)$ and $Q(x)$ are predicates, then

$$\boxed{\begin{array}{rcl}\{x \in X \mid \neg(P(x) \wedge Q(x))\} & = & \{x \in X \mid \neg P(x) \vee \neg Q(x)\}, \\ \{x \in X \mid \neg(P(x) \vee Q(x))\} & = & \{x \in X \mid \neg P(x) \wedge \neg Q(x)\}.\end{array}}$$

The counterparts of these identities for set operations are: if X and Y are sets, then

$$\boxed{\begin{array}{rcl}(X \cap Y)^c & = & X^c \cup Y^c, \\ (X \cup Y)^c & = & X^c \cap Y^c.\end{array}}$$

These identities are called **de Morgan's rules**.

A.1.8 Permutations and combinations

Given a set X with a finite number n of elements, we sometimes wish to construct a subset with a fixed number $m \leq n$ of elements. The number of such subsets is given by

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}, \tag{A.10}$$

where the exclamation point denotes the factorial function. The notation $\binom{n}{m}$ is read "n **choose** m". It gives the number of **combinations** of m elements from the set n.

A combination is a set, so order does not matter. Sometimes, however, order matters. Suppose for example that $X = \{a, b, c\}$. The number of subsets with two elements is

$$\binom{3}{2} = \frac{3!}{2!1!} = \frac{6}{2} = 3.$$

These subsets are $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$. Suppose instead that we wish to construct *ordered* subsets of X with two elements. In other words, we wish to consider $[a, b]$ to be distinct from $[b, a]$ (note the use of square brackets to avoid confusion with unordered sets). Such ordered subsets are called **permutations**. The number of m-element permutations of a set of size n is given by

$$\frac{n!}{(n-m)!}. \tag{A.11}$$

Notice that the number of permutations is a factor of $m!$ larger than the number of combinations in A.10. For example, the number of 2-element permutations of $X = \{a, b, c\}$ is six. They are $[a, b]$, $[a, c]$, $[b, c]$, $[b, a]$, $[c, a]$, and $[c, b]$.

A.1.9 Product sets

The **product** $X \times Y$ of two sets X and Y consists of all pairs of elements (x, y) with $x \in X$ and $y \in Y$, i.e.

$$X \times Y = \{(x, y) \mid x \in X, y \in Y\}.$$

The product of two sets may be visualized as in figure A.2. These pictures informal, to be used only to reinforce intuition. In figure A.2(a), the set $X = [0, 6]$ is represented by a horizontal line segment, and the set $Y = [1, 8]$ is represented by the vertical line segment. The product set

Basics: Tuples, strings, and sequences

Given N sets, $X_1, X_2, \cdots, X_N$, (which may be identical), an N**-tuple** is an ordered collection of one element from each set. It is written in parentheses, as in

$$(x_1, x_2, \cdots, x_N)$$

where

$$x_i \in X_i \text{ for each } i \in \{1, 2, \cdots, N\}.$$

The elements of an N-tuple are called its **components** or **coordinates**. Thus x_i is the i-th component or coordinate of $(x_1, \cdots, x_N)$. The order in which the components are given is important; it is part of the definition of the N-tuple. We can use a variable to refer to the entire N-tuple, as in $x = (x_1, \cdots, x_N)$.

Frequently the sets from which the tuple components are drawn are all identical, as in

$$(x_1, x_2, \cdots, x_N) \in X^N.$$

The above notation means simply that each component in the tuple is a member of the same set X. Of course, this means that a tuple may contain identical components. For example, if $X = \{a, b, c\}$ then (a, a) is a 2-tuple over X.

Recall that a permutation is ordered, like a tuple, but like a set and unlike a tuple, it does not allow duplicate elements. In other words, (a, a) is not a permutation of $\{a, b, c\}$. So a permutation is not the same as a tuple. Similarly, an ordered set does not allow duplicate elements, and thus is not the same as a tuple.

We define the set of finite **sequences** over X to be

$$\{(x_1, \cdots x_N) \mid x_i \in X, 1 \leq i \leq N, N \in \textit{Nats}_0\}.$$

where if $N = 0$ we call the sequence the **empty sequence**. This allows us to talk about tuples without specifying N. Finite sequences are also called **strings**, although by convention, strings are written differently, omitting the parentheses and commas, as in

$$x_1 x_2 \cdots x_N.$$

We may even wish to allow N to be infinite. We define the set of **infinite sequence** over a set X to be

$$\{(x_1, x_2, \cdots) \mid x_i \in X, i \in \textit{Nats}_0\}.$$

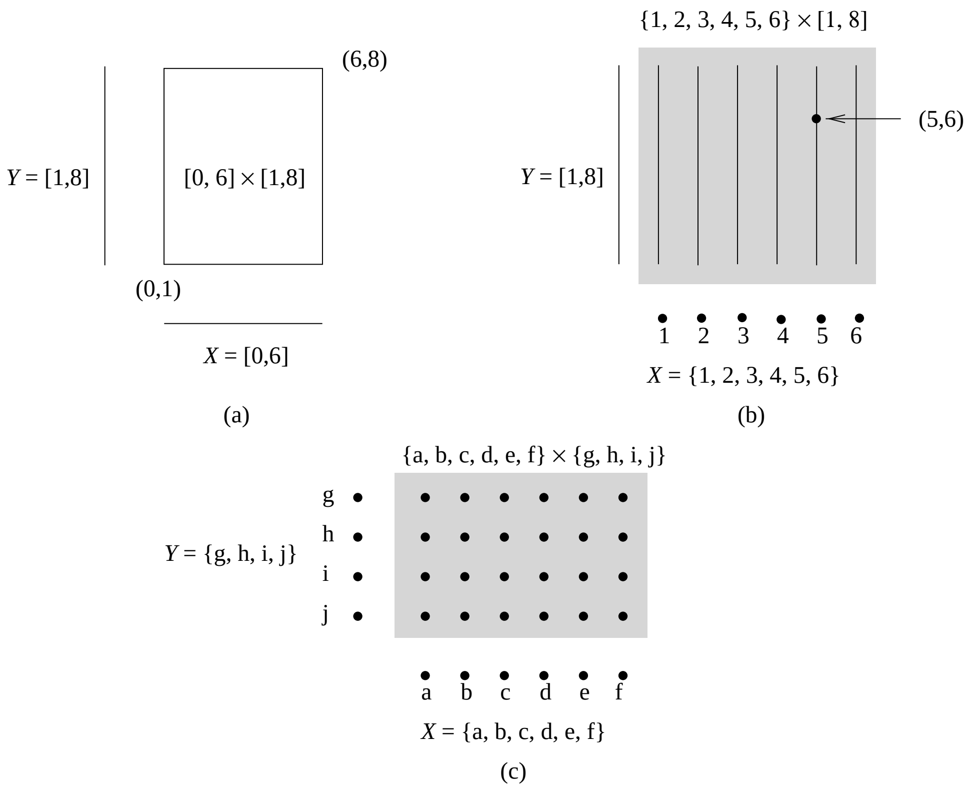


Figure A.2: Visualization of product sets. (a) The rectangle depicts the product set $[0, 6] \times [1, 8]$. (b) Together, the six vertical lines depict the product set $\{1, \cdots, 6\} \times [1, 8]$. (c) The array of dots depicts the set $\{a, b, c, d, e, f\} \times \{g, h, i, j\}$.

$X \times Y = [0,6] \times [1,8]$ is represented by the rectangle whose lower left corner is the pair $(0,1)$ and upper right corner is $(6,8)$.

In figure A.2(b), the discrete set $X = \{1,2,3,4,5,6\}$ is represented by six points, while and $Y = [1,8]$ is represented the same as before. The product set is depicted by six vertical line segments, one for each element of X. For example, the fifth segment from the left is the set $\{(5,y) \mid 1 \le y \le 8\}$. One point in that set is shown.

In figure A.2(c), the product of two discrete sets is shown as an array of points. Note that unless these are ordered sets, there is no significance to the the left-to-right or top-to-bottom order in which the points are shown. In all three cases in the figure, there is no significance to the choice to depict the first set in the product $X \times Y$ on the horizontal axis, and the second set on the vertical axis. We could have done it the other way around. Although there is no significance to which is depicted on the vertical axis and which on the horizontal, there *is* significance to the order of X and Y in $X \times Y$. The set $X \times Y$ is not the same as $Y \times X$ unless $X = Y$.

We generalize the product notation to three or more sets. Thus if X, Y, Z are sets, then $X \times Y \times Z$ is the set of all triples or 3-tuples,

$$X \times Y \times Z = \{(x,y,z) \mid x \in X, y \in Y, z \in Z\},$$

and if there are N sets, $X_1, X_2, \cdots, X_N$, their product is the set consisting of N-tuples,

$$\boxed{X_1 \times \cdots \times X_N = \{(x_1, \cdots, x_N) \mid x_i \in X_i, i = 1, \cdots, N\}.} \tag{A.12}$$

We can alternatively write (A.12) as

$$\boxed{\prod_{i=1}^{N} X_i.} \tag{A.13}$$

The large Π operator indicates a product of its arguments.

$X \times X$ is also written as X^2. The N-fold product of the same set X is also written as X^N. For example, *Reals*^{N} is the set of all N-tuples of real numbers, and *Comps*^{N} is the set of all N-tuples of complex numbers. In symbols,

$$\boxed{\begin{aligned} Reals^N &= \{x = (x_1, \cdots, x_N) \mid x_i \in Reals, i = 1, \cdots, N\}, \\ Comps^N &= \{z = (z_1, \cdots, z_N) \mid z_i \in Comps, i = 1, \cdots, N\}. \end{aligned}}$$

Predicates on product sets

A variable over $X \times Y$ is denoted by a pair (x,y), with x as the variable over X and y as the variable over Y. We can use predicates in x and y to define subsets of $X \times Y$.

Example 1.1: The set

$$\{(x,y) \in [0,1] \times [0,2] \mid x \le y\} \tag{A.14}$$

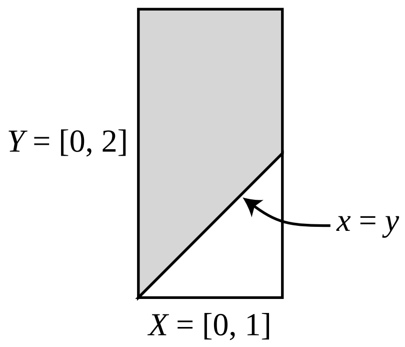


Figure A.3: The rectangle depicts the set $[0, 1] \times [0, 2]$, the shaded region depicts the set given by (A.14), and the unshaded triangle depicts the set given by (A.15).

can be depicted by the shaded region in Figure A.3. The unshaded triangle depicts the set

$$\{(x, y) \in [0, 1] \times [0, 2] \mid x \geq y\}. \tag{A.15}$$

Example 1.2: The solid line in Figure A.4(a) represents the set

$$\{(x, y) \in \mathit{Reals}^2 \mid x + y = 1\},$$

the shaded region depicts

$$\{(x, y) \in \mathit{Reals}^2 \mid x + y \geq 1\},$$

and the unshaded region (excluding the solid line) depicts

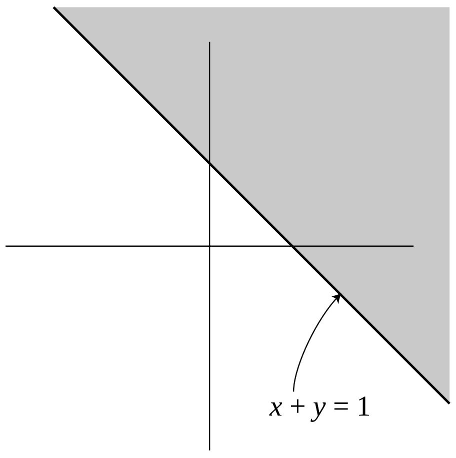
$$\{(x, y) \in \mathit{Reals}^2 \mid x + y < 1\}.$$

Similarly the shaded region in Figure A.4(b) depicts the set

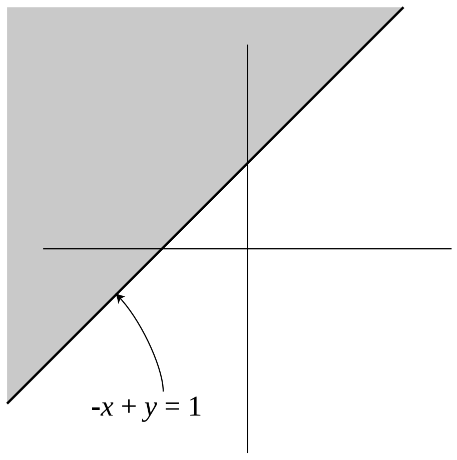
$$\{(x, y) \in \mathit{Reals}^2 \mid -x + y \geq 1\}.$$

The overlap region in Figure A.4 (c) depicts the intersection of the two shaded regions, and corresponds to the conjunction of two predicates:

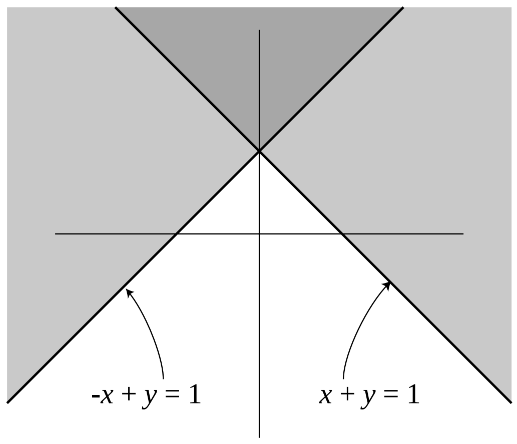
$$\{(x, y) \in \mathit{Reals}^2 \mid [x + y \geq 1] \wedge [-x + y \geq 1]\}.$$



(a)



(b)



(c)

Figure A.4: (a) The solid line depicts the subset of *Reals*^{2} satisfying the predicate $x + y = 1$. The shaded region satisfies $x + y \geq 1$. (b) The solid line satisfies $-x + y = 1$, and the shaded region satisfies $-x + y \geq 1$. (c) The overlap region satisfies $[x + y \geq 1] \wedge [-x + y \geq 1]$.

Example 1.3: The set *TallerThan* consists of all pairs of students $(\mathit{name}_1, \mathit{name}_2)$ such that name_1 is taller than name_2:

$$\mathit{TallerThan} = \{(\mathit{name}_1, \mathit{name}_2) \in \mathit{Students}^2 \mid \mathit{name}_1 \text{ is taller than } \mathit{name}_2\}.$$

NearbyCities consists of pairs of cities that are less than 50 miles apart:

$$\mathit{NearbyCities} = \{(\mathit{city}_1, \mathit{city}_2) \in \mathit{USCities}^2 \mid \mathit{distance}(\mathit{city}_1, \mathit{city}_2) \leq 50\}.$$

In the predicate above, $\mathit{distance}(\mathit{city}_1, \mathit{city}_2)$ is the distance in miles between city_1 and city_2.

A.1.10 Evaluating a predicate expression

We have evaluated predicate expressions several times in this appendix, each time relying on the reader's basic mathematical facility with expressions. We can develop systematic methods for evaluating expressions that rely less on intuition, and can therefore handle more complicated and intricate expressions.

We have introduced the following patterns of expressions:

- $A, B, \mathit{Nats}, \cdots$, names of sets
- $A = \{\text{list of elements}\}$
- $x \in A$, $x \notin A$, set membership
- $A = B$, $B \subset A$, and $A \supset B$, set inclusion
- $A \cap B$, $A \cup B$, $X \times Y$, $X \setminus A$, A^c, set operations
- $x, y, \cdots$, names of variables
- $P(x), Q(x), \cdots$, predicates in x
- $\forall x \in Set,\ P(x)$ and $\exists x \in Set,\ Q(x)$, assertions obtained by quantification
- $\mathit{NewSet} = \{x \in Set \mid \mathit{Pred}(x)\}$, set definition
- $P(x) \wedge Q(x)$, $P(x) \vee Q(x)$, $\neg(P(x))$, predicate operations

The patterns define the rules of grammar of the notation. An expression is **well-formed** if it conforms to these patterns. For example, if P, Q and R are predicates, then the syntax implies that

$$\neg[[\neg(P(x)) \vee Q(x)] \wedge [P(x) \vee [R(x) \wedge \neg(P(x))]]] \tag{A.16}$$

is also a predicate. Just as in the case of a computer language, you learn the syntax of mathematical expressions through practice.[3]

[3]The syntax of a language is the set of rules (or patterns) whereby words can be combined to form grammatical or well-formed sentences. Thus the syntax of the 'C' language is the set of rules that a sequence of characters (the code) must obey to be a C program. A C compiler checks whether the code conforms to the syntax. Of course, even if the code obeys the syntax, it may not be correct; i.e. it may not execute the correct computation.

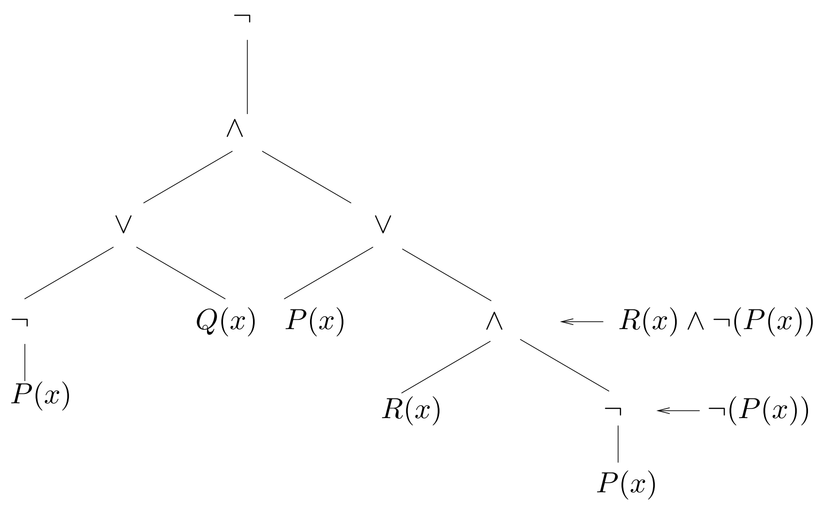


Figure A.5: Parse tree for the expression (A.16).

Parsing

To show that (A.16) is indeed a well-formed predicate, we must show that it can be constructed using the syntax. We do this by **parsing** the expression (A.16) with the help of matching brackets and parentheses. Parsing the expression will also enable us to evaluate it in a systematic way.

The result is the parse tree shown in figure A.5. The leaves of this tree (the bottom-most nodes) are labeled by the elementary predicates P, Q, R, and the other nodes of the tree are labeled by one of the predicate operations $\wedge, \vee, \neg$. Each of the intermediate nodes corresponds to a sub-predicate of (A.16). Two such sub-predicates are shown in the figure. The last leaf on the right is labeled $P(x)$. Its parent node is labeled $\neg$, and so that parent node corresponds to the sub-predicate $\neg(P(x))$. Its parent node is labeled $\wedge$, and it has another child node labeled $R(x)$, so this node corresponds to the sub-predicate $R(x) \vee \neg(P(x))$.

If we go up the tree in this way, we can see that the top of the tree (the root node) indeed corresponds to the predicate $(A.16)$. Since at each intermediate node, the sub-predicate is constructed using the syntax, the root node is a well-formed predicate.

Evaluating

Suppose we know whether the predicates $P(x), Q(x), R(x)$ evaluate to true or false for some value of x. Then we can use the parse tree to figure out whether the predicate (A.16) evaluates to true or false. To do this, we begin with the known truth values at the leaves of the parse tree, use the meaning of the predicate operations to figure out the truth values of the sub-predicates corresponding to the parents of the leaf nodes, and then the parents of those nodes, and work our way up the tree to figure out the truth value of the predicate (A.16) at the root node. In figure A.6 the parse tree is annotated with the truth values of each of the nodes. Since the root node is annotated 'false,' we

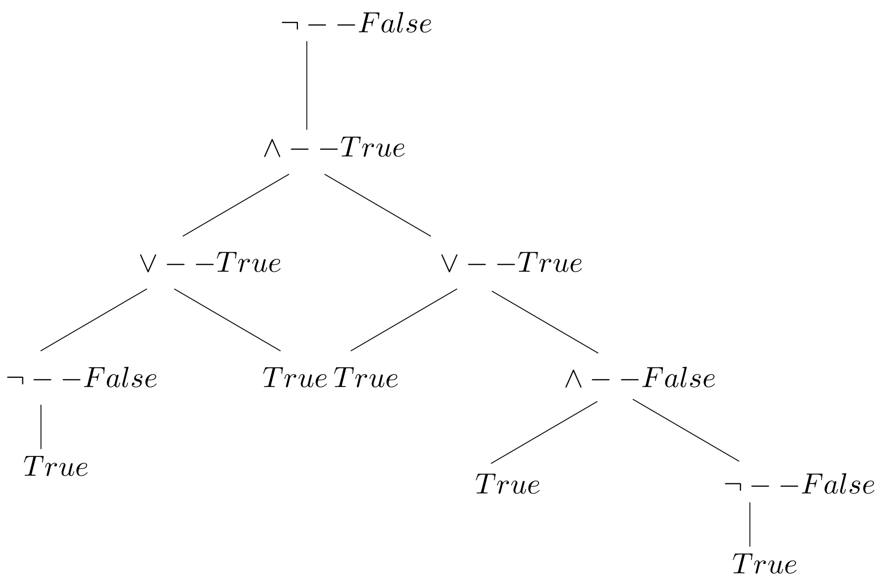


Figure A.6: Parse tree for the expression (A.16) annotated with the truth values of each of the nodes.

conclude that (A.16) evaluates to false.

Truth tables

The way in which the predicate operations transform the truth values is given in the following **truth table**:

$P(x)$	$Q(x)$	$\neg P(x)$	$P(x) \wedge Q(x)$	$P(x) \vee Q(x)$
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Consider a particular row of this table, say the first row. The first two entries specify that $P(x)$ is true and $Q(x)$ is true. The remaining three entries give the corresponding truth values for $\neg P(x)$, $P(x) \wedge Q(x)$ and $P(x) \vee Q(x)$, namely, $\neg P(x)$ is false, $P(x) \wedge Q(x)$ is true, and $P(x) \vee Q(x)$ is true. The four rows correspond to the four possible truth value assignments of $P(x)$ and $Q(x)$. This truth table can be used repeatedly to evaluate any well-formed expression given the truth value of $P(x)$ and $Q(x)$.

Thus, given the truth values of predicates $P_1(x), \ldots, P_n(x)$, the truth value of any well-formed expression involving these predicates can be obtained by a computer algorithm that constructs the parse tree and uses the truth table above. Such algorithms are used to evaluate logic circuits.

A.2 Functions

In the notation

$$f : X \to Y, \tag{A.17}$$

X and Y are sets, and f is the name of a function. The function is an assignment rule that assigns a value in Y to each element in X. If the element in X is x, then the value is written $f(x)$.

We read (A.17) as "f is (the name of) a function from X into (or to) Y." We also say that f maps X into Y. The set X is called the **domain** of f, written $X = \mathit{domain}(f)$, the set Y is called the **range** of f, written $Y = \mathit{range}(f)$.[4] When the domain and range are understood from the context, we write "f" by itself to represent the function or the map. If x is a variable over X, we also say "f is a function of x."

Example 1.4: Recall the set *Students* of all the students in this class. Each element of *Students* is represented by a student's name,

$$\mathit{Students} = \{\text{John Brown}, \text{Jane Doe}, \cdots\}.$$

We assign to each name in *Students* the student's marks in the final examination, a number between 0 and 100. This name-to-marks assignment is an example of a function. Just as we give names to sets (e.g. *Students*), we give names to functions. In this example the function might be named *Score*. When we evaluate the function *Score* at any name, we get the marks assigned to that name. We write this as

$$\mathit{Score}(\text{John Brown}) = 90, \ \mathit{Score}(\text{Jane Doe}) = 91.2, \cdots$$

Figure A.7 illustrates the function *Score*. Three things are involved in defining *Score*: the set *Students*, the set $[0, 100]$ of possible marks, and the assignment of marks to each name. In the figure this assignment is depicted by the arrows: the tail of the arrow points to a name and the head of the arrow points to the marks assigned to that name. We denote these three things by

$$\mathit{Score}\colon \mathit{Students} \to [0, 100]$$

which we read as "*Score* is a function from *Students* into $[0, 100]$." The domain of the function *Score* is *Students*, and the range of *Score* is $[0, 100]$.

It is easy to imagine other functions with the same domain *Students*. For example, *Height* assigns to each student his or her height measured in cm, *SSN* assigns students their social security number, and *Address* assigns students their address. The range of these functions is different. The range of *Height* might be defined to be $[0, 200]$ (200 cm is about 8 feet). Since a social security number is a 9-digit number, we can take $\{0, 1, \cdots, 9\}^9$ to be the range of *SSN*. And we can take the range of *Address* to be $Char^{100}$, assuming that an address can be expressed as a string of 100 characters, including blank spaces.

[4] In some mathematics texts, the set Y which we call the range is called the **codomain**, and $\mathit{range}(f)$ is defined to be the set of all values that f takes, i.e., $\mathit{range}(f) = \{f(x) \mid x \in X\}$. However, we will not use this terminology.

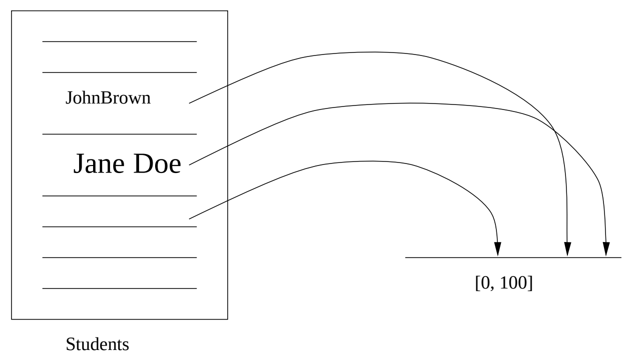


Figure A.7: Illustration of *Score*.

We usually use lower case letters for function names, such as f, g, h, or more descriptive names such as *Score*, *Voice*, *Video*, *SquareWave*, *AMSignal*.

Avoid a bad habit: It is important to distinguish between a function f and its value $f(x)$ at a particular point $x \in domain(f)$. The function f is a *rule* that assigns a value in $range(f)$ to each $x \in domain(f)$, whereas $f(x)$ is a *point* or element in $range(f)$. Unfortunately, too many books encourage the bad habit by using '$f(x)$' as a shorthand for 'f is a function of x.' If you keep the distinction between f and $f(x)$, it will be easier to avoid confusion when we study systems.

A.2.1 Defining functions

To define a function, you must give the domain, the range, and the rule that produces an element in the range given an element in the domain. There are many ways to do this, as explored in much more depth in chapter 2. Here we mention only two. The first is enumeration. That is, in tabular form or some other form, each possible value in the domain is associated with a value in the range. This method would be appropriate for the *Score* function, for example. Alternatively, functions can be mathematically defined by the prototype: define $f : X \to Y$,

$$\forall\, x \in X, \quad f(x) = \text{expression in } x.$$

The 'expression in x' may be specified by an algebraic expression, by giving the graph of f, by a table, or by a procedure.

A.2.2 Tuples and sequences as functions

An N-tuple $x = (x_1, \cdots, x_N) \in X^N$ can be viewed as a function

$$x \colon \{1, \cdots, N\} \to X.$$

For each integer in $\{1, \cdots, N\}$, it assigns a value in X. An infinite sequence y over the set Y can also be viewed as a function

$$y \colon \mathit{Nats} \to Y$$

or

$$y \colon \mathit{Nats}_0 \to Y,$$

depending on whether you wish to begin indexing the sequence at zero or one (unfortunately, both conventions are widely used). This view of sequences as functions is in fact our model for discrete-time signals and event traces, as developed in chapter 1.

A.2.3 Function properties

A function $f \colon X \to Y$ is **one-to-one** if

$$\forall\, x_1 \in X \text{ and } \forall\, x_2 \in X, \quad x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2).$$

Here the logical symbol '$\Rightarrow$' means 'implies' so the expression is read: if x_1, x_2 are two different elements in X, then $f(x_1), f(x_2)$ are different.

Example 1.5: The function *Cube*: *Reals* $\to$ *Reals* given by

$$\forall\, x \quad \mathit{Cube}(x) = x^3,$$

is one-to-one, because if $x_1 \neq x_2$, then $x_1^3 \neq x_2^3$. But *Square* is not one-to-one because, for example, $\mathit{Square}(1) = \mathit{Square}(-1)$.

A function $f \colon X \to Y$ is **onto** if

$$\forall\, y \in Y,\ \exists\, x \in X, \text{ such that } f(x) = y.$$

The symbol '$\exists$' is the existential quantifier, which means 'there exists' or 'for some'. So the expression above reads: 'For each y in Y, there exists x in X such that $f(x) = y$.'

Accordingly, f is onto if for every y in its range there is some x in its domain such that $y = f(x)$.

Example 1.6: The function *Cube*: *Reals* $\to$ *Reals* is one-to-one and onto, while *Square*: *Reals* $\to$ *Reals* is not onto. However, *Square*: *Reals* $\to$ Reals_+ *is* onto.

Probing further: Infinite sets

The size of a set A, denoted $|A|$, is the number of elements it contains. By counting, we immediately know that $\{1,2,3,4\}$ and $\{a,b,c,d\}$ have the same number of elements, whereas $\{1,2,3\}$ has fewer elements. But we cannot count infinite sets. It is more difficult to compare the number of elements in the following infinite sets:

$$A = \mathit{Nats} = \{1,2,3,4,\cdots\},\ B = \{2,3,4,5\cdots\},\ C = [0,1].$$

At first, we might say that A has one more element than B, since A includes B but has one additional element, $1 \in A$. In fact, these two sets have the same size.

The **cardinality** of a set is the number of elements in the set, but generalized to handle infinite sets. Comparing the cardinality of two sets is done by matching elements, using one-to-one functions. Consider two sets A and B, finite or infinite. We say that A has a smaller cardinality than B, written $|A| \leq |B|$, if there exists a one-to-one function mapping A into B. We say that A and B have the same cardinality, written $|A| = |B|$, if $|A| \leq |B|$ and $|B| \leq |A|$.

The cardinality of the infinite set $A = \mathit{Nats}$ is denoted $\aleph_0$, read "aleph zero" (aleph is the first letter of the Hebrew alphabet). It is quite easy to prove using the definition of cardinality that $n < \aleph_0$ for any finite number n.

We can now show that the cardinality of B is also $\aleph_0$. There is a one-to-one function $f\colon A \to B$, namely

$$\forall\, n \in A, \quad f(n) = n + 1,$$

so that $|A| \leq |B|$, and there is a one-to-one function $g\colon B \to A$, namely

$$\forall\, n \in B, \quad g(n) = n - 1,$$

so that $|B| \leq |A|$. A similar argument can be used to show that the set of even numbers and the set of odd numbers also have cardinality $\aleph_0$.

It is more difficult to show that the cardinality of $\mathit{Nats} \times \mathit{Nats}$ is also $\aleph_0$. To see this, we can define a one-to-one function $h\colon \mathit{Nats}^2 \to \mathit{Nats}$ as follows (see figure A.8).

$$h((1,1)) = 1,\ h((2,1)) = 2,\ h((2,2)) = 3,\ h((1,2)) = 4,\ h((1,3)) = 5,\ \cdots$$

Observe that since a rational number m/n can be identified with the pair $(m,n) \in \mathit{Nats}^2$, the argument above shows that the cardinality of the set of all rational numbers is also $\aleph_0$.

Probing further: Even bigger sets

We can show that the cardinality of $[0,1]$ is strictly larger than that of *Nats*; i.e. $|[0,1]| > |\mathit{Nats}|$ Since the function $f\colon \mathit{Nats} \to [0,1]$ defined by

$$\forall\, n \in \mathit{Nats}, \quad f(n) = 1/n$$

is one-to-one, we have $|\mathit{Nats}| \le |[0,1]|$. However, we can show that there is no one-to-one function in the other direction. If there were such a function $g\colon [0,1] \to \mathit{Nats}$, then it would be possible to enumerate all the elements of $[0,1]$ in an ordered list,

$$[0,1] = \{x^1, x^2, x^3, \cdots\}. \tag{A.18}$$

(The superscript here is not raising to a power, but just indexing.) We can show that this is not possible. If we express each element of $[0,1]$ by its decimal expansion (ignoring the element 1.0), this list looks like

$$\begin{array}{rcl} x^1 & = & 0.x^1_1 x^1_2 x^1_3 \cdots \\ x^2 & = & 0.x^2_1 x^2_2 x^2_3 \cdots \\ x^3 & = & 0.x^3_1 x^3_2 x^3_3 \cdots \\ & & \cdots \\ x^n & = & 0.x^n_1 x^n_2 x^n_3 \cdots \\ & & \cdots \end{array}$$

Construct any number $y \in [0,1]$ with the decimal expansion

$$y = 0.y_1 y_2 y_3 \cdots$$

such that for each i, $y_i \ne x^i_i$ where x^i_i is the i-th term in the decimal expansion of x^i. Clearly such a number exists and is in $[0,1]$. But then for every i, $y \ne x^i$, so that y cannot be in the list $\{x^1, x^2, x^3, \cdots\}$. Thus, the list (A.18) is not complete in that it does not include all the elements of $[0,1]$.

The cardinality of $[0,1]$ is denoted $\aleph_1$, and is strictly greater than $\aleph_0$. In this sense we can say that the continuum $[0,1]$ has more elements than the denumerable set *Nats*, even though both sets have infinite size.

The obvious question is whether there are cardinalities larger than $\aleph_1$. The answer is yes; in fact, there are sets of ever higher cardinality,

$$\aleph_0 < \aleph_1 < \aleph_2, \cdots$$

and sets with cardinality larger than all of these!

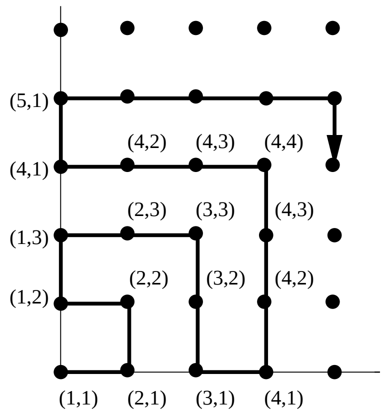


Figure A.8: A correspondence between *Nats*^{2} and *Nats*.

A.3 Summary

Sets are mathematical objects representing collections of elements. A variable is a representative for an element of a set. A predicate over a set is an expression involving a variable that evaluates to true or false when the variable is assigned a particular element of the set. Predicates are used to construct new sets from existing sets. If the variable in a predicate is quantified, the expression becomes an assertion.

Sets can be combined using the operations of union, intersection and complement. The corresponding operations on predicates are disjunction, conjunction and negation. New sets can also be obtained by the product of two sets.

There are precise rules that must be followed in using these operations. The collection of these rules is the syntax of sets and predicates. By parsing a complex expression, we can determine whether it is well-formed. The truth value of a predicate constructed from elementary predicates using predicate operations can be calculated using the parse tree and the truth table.

Functions are mathematical objects representing a relationship between two sets, the domain and the range of the function. We have introduced the following patterns of expressions for functions

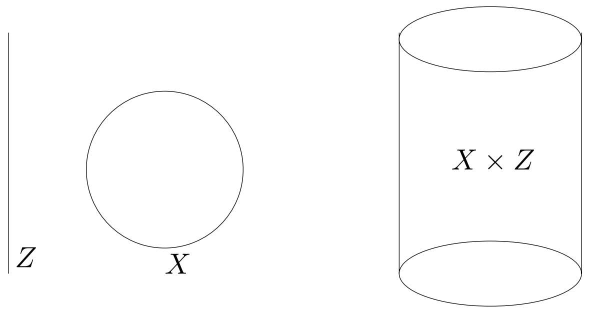
- $f, g, h, \mathit{Score}, \cdots$, names of functions,
- $f : X \to Y$, $X = \mathit{domain}(f), Y = \mathit{range}(f)$, a function from X to Y,

A function $f : X \to Y$ assigns to each value $x \in X$ a value $f(x) \in Y$.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** In the spirit of figure A.2, give a picture of the following sets:
   (a) $\{1, 2, 3\}$,
   (b) $[0, 1] \times \{0, 1\}$,
   (c) $[0, 1] \times [a, b]$.
   (d) $\{1, 2, 3\} \times \{a, b\}$,
   (e) $\{a, b\} \times [0, 1]$.
2. **E** How many elements are there in the sets (a) $\{1, \cdots, 6\}$, (b) $\{-2, -1, \cdots, 10\}$, and (c) $\{0, 1, 2\} \times \{2, 3\}$?
3. **T** Determine which of the following expressions is true and which is false:
   (a) $\forall n \in$ *Nats*, $\quad n > 1$,
   (b) $\exists n \in$ *Nats*, $\quad n < 10$,
   (c) If $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, then $\forall x \in A, \forall y \in B, \quad x \leq y$,
   (d) If $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, then $\forall x \in A, \exists y \in B, \quad x \leq y$,
   (e) If $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$, then $\exists x \in A, \forall y \in B, \quad x \leq y$.
4. **T** In the following figure, $X = \{(x, y) \mid x^2 + y^2 = 1\}$ is depicted as a 2-dimensional circle and $Z = [0, 2]$ is shown as a 1-dimensional line segment. Explain why it is reasonable to show the product set as a 3-dimensional cylinder.



5. **E** In the spirit of figure A.2, give a picture for the product set $\{M, Tu, W, Th, F\} \times [8.00, 17.00]$ and indicate on your drawing the lecture hours for this class.
6. **E** In the spirit of figure A.2, give a picture for the set $A = \{(x, y) \mid x \in [1, 2], y \in [1, 2]\}$ and the set $B = \{(x, x) \mid x \in [1, 2]\}$. Explain why the two sets are different.

7. **C** Give a precise expression for the predicate below so that *Triangle* is indeed a triangle:

$$\mathit{Triangle} = \{(x, y) \in \mathit{Reals}^2 \mid \mathit{Pred}(x, y)\}.$$

There are many ways of writing this predicate. One way is to express $\mathit{Pred}(x, y)$ as the conjunction of three linear inequality predicates. Hint: We used the conjunction of two linear inequality predicates in figure A.4.

8. **T** If X has m elements and Y has n elements, how many elements are there in $X \times Y$? If X_i has m_i elements, for $i = 1, \cdots, I$, for some constant I. How many elements are there in

$$\prod_{i=1}^{i=I} X_i = X_1 \times \cdots \times X_I?$$

9. **T** How many different 10-letter strings are there if each letter is drawn from the set $Alphabet$ consisting of the 26 lower case letters of the alphabet? How many such strings are there with *exactly* one occurrence of the letter a?

10. **T** Recall that a set cannot contain duplicate elements. Now suppose X contains 10 elements.
    - How many two-element combinations of elements from X are there?
    - How many two-element permutations are there?

11. **C** Construct predicates for use in the prototype (A.4) to define the following sets. Define them in terms of examples of sets introduced in this chapter.
    (a) The set of U.S. cities with a population exceeding one million.
    (b) The male students in the class.
    (c) The old books in the library.

12. **T** Which of the following expressions is well formed? For those that are well formed, state whether they are assertions. For those that are assertions, evaluate the assertion to true or false. For those that are not assertions, find an equivalent simpler expression.
    (a) $2 \in \{1, 3, 4\}$,
    (b) $3 \subset \{1, 3, 4\}$,
    (c) $\{3\} \subset \{1, 2, 3\}$,
    (d) $2 \cup \{1, 3, 4\}$,
    (e) $\{2\} \cup \{1, 3, 4\}$,
    (f) $[2.3, 3.4] = \{x \in \mathit{Reals} \mid 2.3 \leq x \leq 3.4\}$,
    (g) $\{x \in \mathit{Reals} \mid x > 3 \wedge x < 4\}$,
    (h) $[1, 2] \cap [3, 4] = \emptyset$.

13. **E** Define the following sets in terms of the sets named in section A.1.5
    (a) The set of all 10-letter passwords.

(b) The set of all 5×6 matrices of real numbers.

(c) The set of all complex numbers with magnitude at most 1.

(d) The set of all 2-dimensional vectors with magnitude exactly 1.

14. **E** Enumerate the set of all subsets (including the empty set) of $\{a, b, c\}$.

15. **T** Suppose a set X has n elements. Let $P(X)$ be the power set of X. How many elements are there in $P(X)$?

16. **T** Use Matlab to depict the following sets using the plot command:

    (a) $\{(t, x) \in \textit{Reals}^2 \mid x = \sin(t),\ \text{and } t \in \{0, \frac{1}{20}2\pi, \frac{2}{20}2\pi, \cdots, \frac{20}{20}2\pi\}\}$,

    (b) $\{(y, x) \in \textit{Reals}^2 \mid y = e^x,\ \text{and } x \in \{-1, -1 + \frac{1}{20}, -1 + \frac{2}{20}, \cdots, 1\}\}$,

    (c) $\{(y, x) \in \textit{Reals}^2 \mid y = e^{-x},\ \text{and } x \in \{-1, -1 + \frac{1}{20}, -1 + \frac{2}{20}, \cdots, 1\}\}$.

17. **T** Determine which of the following functions is onto, which is one-to-one, and which is neither, and give a short explanation for your answer.

    (a) $\textit{License}\colon \textit{CalVehicles} \to Char*$ given by $\forall\ \textit{vehicle} \in \textit{CalVehicles}$, $\textit{License}(\textit{vehicle})$ is the California license number of the vehicle

    (b) $f\colon \textit{Reals} \to [-2, 2]$, given by $\forall\ x \in \textit{Reals},\quad f(x) = 2\sin(x)$

    (c) $f\colon \textit{Reals} \to \textit{Reals}$, given by $\forall\ x \in \textit{Reals},\quad f(x) = 2\sin(x)$

    (d) $\textit{conj}\colon \textit{Comps} \to \textit{Comps}$, the complex conjugate function

    (e) $f\colon \textit{Comps} \to \textit{Reals}^2$, given by $\forall\ z \in \textit{Comps},\quad f(z) = (Re(z), Im(z))$, where $Re(z)$ is the real part of z and $Im(z)$ is the imaginary part of z.

    (f) $M\colon \textit{Reals}^2 \to \textit{Reals}^2,\ \forall\ (x_1, x_2) \in \textit{Reals}^2$,

    $$M(x_1, x_2) = (y_1, y_2)$$

    where
    $$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}.$$

    (g) $\textit{Zero} : \textit{Reals}^4 \to \textit{Reals}^4, \forall\ x \in \textit{Reals}^4,\ \ \textit{Zero}(x) = (0, 0, 0, 0)$

Appendix B

Complex Numbers

Complex numbers are used extensively in the modeling of signals and systems. There are two fundamental reasons for this. The first reason is that complex numbers provide a compact and elegant way to talk simultaneously about the phase and amplitude of sinusoidal signals. Complex numbers are therefore heavily used in Fourier analysis, which represents arbitrary signals in terms of sinusoidal signals. The second reason is that a large class of systems, called linear time-invariant (LTI) systems, treat signals that can be described as complex exponential functions in an especially simple way. They simply scale the signals.

These uses of complex numbers are developed in detail in the main body of this text. This appendix summarizes essential properties of complex numbers themselves. We review complex number arithmetic, how to manipulate complex exponentials, Euler's formula, the polar coordinate representation of complex numbers, and the phasor representation of sinewaves.

B.1 Imaginary numbers

The quadratic equation,

$$x^2 - 1 = 0,$$

has two solutions, $x = +1$ and $x = -1$. These solutions are said to be **roots** of the polynomial $x^2 - 1$. Thus, this polynomial has two roots, $+1$ and -1.

Consider an n-th degree polynomial of the form

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n. \tag{B.1}$$

The roots of this polynomial are defined to be the solutions to the polynomial equation

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n = 0. \tag{B.2}$$

The roots of a polynomial are related to a particularly useful factorization into first-degree polynomials. For example, we can factor the polynomial $x^2 - 1$ as

$$x^2 - 1 = (x-1)(x+1).$$

Notice the role of the roots, $+1$ and -1. In general, if (B.1) has roots $r_1, \cdots, r_n$, then we can factor the polynomial as follows

$$x^n + a_1 x^{n-1} + \cdots + a_{n-1}x + a_n = (x - r_1)(x - r_2)\cdots(x - r_n). \tag{B.3}$$

It is easy to see that if $x = r_i$ for any $i \in \{1, \cdots n\}$, then the polynomial evaluates to zero, so (B.2) is satisfied.

An interesting question that arises is whether (B.2) always has a solution for x. In other words, can we always find roots for a polynomial?

The equation

$$x^2 + 1 = 0 \tag{B.4}$$

has no solution for x in the set of real numbers. Thus, it would appear that not all polynomials have roots. However, a surprisingly simple and clever mathematical device changes the picture dramatically. With the introduction of **imaginary numbers**, mathematicians ensure that all polynomials have roots. Moreover, they ensure that any polynomial of degree n has exactly n factors as in (B.3). The n values $r_1, \cdots, r_n$ (some of which may be repeated) are the **roots** of the polynomial.

If we try by simple algebra to solve (B.4) we discover that we need to find x such that

$$x^2 = -1.$$

This suggests that

$$x = \sqrt{-1}.$$

But -1 does not normally have a square root.

The clever device is to define an imaginary number, usually written i or j, that is equal to $\sqrt{-1}$. By definition,[1]

$$i \times i = \sqrt{-1} \times \sqrt{-1} = -1.$$

This (imaginary) number, thus, is a solution of the equation $x^2 + 1 = 0$.

For any real number y, yi is an imaginary number. Thus, we can define the set of imaginary numbers as

$$\boxed{\textit{ImaginaryNumbers} = \{yi \mid y \in \textit{Reals}, \text{ and } i = \sqrt{-1}\}} \tag{B.5}$$

It is a profound result that this simple device is all we need to guarantee that every polynomial equation has a solution, and that every polynomial of degree n can be factored into n polynomials of degree one, as in (B.3).

B.2 Arithmetic of imaginary numbers

The sum of i and i is written $2i$. Sums and differences of imaginary numbers simplify like real numbers:

$$3i + 2i = 5i, \; 3i - 4i = -i.$$

[1] Here, the operator $\times$ is ordinary multiplication, not products of sets.

If $y_1 i$ and $y_2 i$ are two imaginary numbers, then

$$\boxed{y_1 i + y_2 i = (y_1 + y_2)i, \quad y_1 i - y_2 i = (y_1 - y_2)i.} \tag{B.6}$$

The product of a real number x and an imaginary number yi is

$$x \times yi = yi \times x = xyi.$$

To take the product of two imaginary numbers, we must remember that $i^2 = -1$, and so for any two imaginary numbers, $y_1 i$ and $y_2 i$, we have

$$y_1 i \times y_2 i = -y_1 \times y_2. \tag{B.7}$$

The result is a real number. We can use rule (B.7) repeatedly to multiply as many imaginary numbers as we wish. For example,

$$i \times i = -1, \; i^3 = i \times i^2 = -i, \; i^4 = 1.$$

The ratio of two imaginary numbers $y_1 i$ and $y_2 i$ is a real number

$$\frac{y_1 i}{y_2 i} = \frac{y_1}{y_2} \, .$$

B.3 Complex numbers

The sum of a real number x and an imaginary number yi is called a **complex number**. This sum does not simplify as do the sums of two reals numbers or two imaginary numbers, and it is written as $x + yi$, or equivalently, $x + iy$ or $x + jy$.

Examples of complex numbers are

$$2 + i, \; -3 - 2i, \; -\pi + \sqrt{2}i.$$

In general a complex number z is of the form

$$z = x + yi = x + y\sqrt{-1},$$

where x, y are real numbers. The **real part** of z, written $Re\{z\}$, is x. The **imaginary part** of z, written $Im\{z\}$, is y. *Notice that, confusingly, the imaginary part is a real number.* The imaginary part times i is an imaginary number. So

$$\boxed{z = Re\{z\} + Im\{z\}i.}$$

The set of complex numbers, therefore, is defined by

$$\boxed{Comps = \{x + yi \mid x \in Reals, y \in Reals, \text{ and } i = \sqrt{-1}\}.} \tag{B.8}$$

Every real number x is in *Comps*, because $x = x + 0i$; and every imaginary number yi is in *Comps*, because $yi = 0 + yi$.

Two complex numbers $z_1 = x_1 + y_1 i$ and $z_2 = x_2 + y_2 i$ are equal if and only if their real parts are equal and their imaginary parts are equal, i.e. $z_1 = z_2$ if and only if

$$Re\{z_1\} = Re\{z_2\}, \text{ and } Im\{z_1\} = Im\{z_2\}.$$

B.4 Arithmetic of complex numbers

In order to add two complex numbers, we separately add their real and imaginary parts,

$$(x_1 + y_1 i) + (x_2 + y_2 i) = (x_1 + x_2) + (y_1 + y_2)i.$$

The **complex conjugate** of $x + yi$ is defined to be $x - yi$. The complex conjugate of a complex number z is written z^*. Notice that

$$z + z^* = 2Re\{z\},\ z - z^* = 2Im\{z\}i.$$

Hence, the real and imaginary parts can be obtained using the complex conjugate,

$$Re\{z\} = \frac{z + z^*}{2}, \text{ and } Im\{z\} = \frac{z - z^*}{2i}\ .$$

The product of two complex numbers works as expected if you remember that $i^2 = -1$. So, for example,

$$(1 + 2i)(2 + 3i) = 2 + 3i + 4i + 6i^2 = 2 + 7i - 6 = -4 + 7i,$$

which seems strange, but follows mechanically from $i^2 = -1$. In general,

$$\boxed{(x_1 + y_1 i)(x_2 + y_2 i) = (x_1 x_2 - y_1 y_2) + (x_1 y_2 + x_2 y_1)i.} \tag{B.9}$$

If we multiply $z = x + yi$ by its complex conjugate z^* we get

$$zz^* = (x + yi)(x - yi) = x^2 + y^2,$$

which is a positive real number. Its positive square root is called the **modulus** or **magnitude** of z, and is written $|z|$,

$$|z| = \sqrt{zz^*} = \sqrt{x^2 + y^2}\ .$$

How to calculate the ratio of two complex numbers is less obvious, but it is equally mechanical. We convert the denominator into a real number by multiplying both numerator and denominator by the complex conjugate of the denominator,

$$\begin{aligned} \frac{2 + 3i}{1 + 2i} &= \frac{2 + 3i}{1 + 2i} \times \frac{1 - 2i}{1 - 2i} \\ &= \frac{(2 + 6) + (-4 + 3)i}{1 + 4} \\ &= \frac{8}{5} - \frac{1}{5}\, i. \end{aligned}$$

The general formula is

$$\frac{x_1 + y_1 i}{x_2 + y_2 i} = \frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2} + \frac{-x_1 y_2 + x_2 y_1}{x_2^2 + y_2^2} i\ . \tag{B.10}$$

In practice it is easier to calculate the ratio as in the example, rather than memorizing formula (B.10).

B.5 Exponentials

Certain functions of real numbers, like the exponential function, are defined by an infinite series. The exponential of a real number x, written e^x or $\exp(x)$, is

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

We also recall the infinite series expansion for $\cos$ and $\sin$:

$$\begin{aligned} \cos(\theta) &= 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4!} - \cdots \\ \sin(\theta) &= \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \cdots \end{aligned}$$

The exponential of a complex number z is written e^z or $\exp(z)$, and is defined in the same way as the exponential of a real number,

$$e^z = \sum_{k=0}^{\infty} \frac{z^k}{k!} = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots . \tag{B.11}$$

Note that $e^0 = 1$, as expected.

The exponential of an imaginary number $i\theta$ is very interesting,

$$\begin{aligned} e^{i\theta} &= 1 + (i\theta) + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \cdots \\ &= [1 - \frac{\theta^2}{2} + \frac{\theta^4}{4!} - \cdots] + i[\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \cdots] \\ &= \cos(\theta) + i\sin(\theta). \end{aligned}$$

This identity is known as **Euler's formula**:

$$\boxed{e^{i\theta} = \cos(\theta) + i\sin(\theta).} \tag{B.12}$$

Euler's formula is used heavily in this text in the analysis of linear time invariant systems. It allows sinusoidal functions to be given as sums or differences of exponential functions,

$$\boxed{\cos(\theta) = (e^{i\theta} + e^{-i\theta})/2} \tag{B.13}$$

and

$$\boxed{\sin(\theta) = (e^{i\theta} - e^{-i\theta})/(2i).} \tag{B.14}$$

This proves useful because exponential functions turn out to be simpler mathematically (despite being complex valued) than sinusoidal functions.

An important property of the exponential function is the **product formula**:

$$\boxed{e^{z_1+z_2} = e^{z_1}e^{z_2}.} \tag{B.15}$$

We can obtain many trigonometric identities by combining (B.12) and (B.15). For example, since

$$e^{i\theta}e^{-i\theta} = e^{i\theta - i\theta} = e^0 = 1,$$

and

$$e^{i\theta}e^{-i\theta} = [\cos(\theta) + i\sin(\theta)][\cos(\theta) - i\sin(\theta)] = \cos^2(\theta) + \sin^2(\theta),$$

so we have the identity

$$\boxed{\cos^2(\theta) + \sin^2(\theta) = 1.}$$

Here is another example. Using

$$e^{i(\alpha+\beta)} = e^{i\alpha}e^{i\beta}, \tag{B.16}$$

we get

$$\begin{aligned}\cos(\alpha+\beta) + i\sin(\alpha+\beta) &= [\cos(\alpha) + i\sin(i\alpha)][\cos(\beta) + i\sin(\beta)] \\ &= [\cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)] \\ &\quad +i[\sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)].\end{aligned}$$

Since the real part of the left side must equal the real part of the right side, we get the identity,

$$\boxed{\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta),}$$

Since the imaginary part of the left side must equal the imaginary part of the right side, we get the identity,

$$\boxed{\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta).}$$

It is much easier to remember (B.16) than to remember these identities.

B.6 Polar coordinates

The representation of a complex number as a sum of a real and an imaginary number, $z = x + iy$, is called its **Cartesian representation**.

Recall from trigonometry that if x, y, r are real numbers and $r^2 = x^2 + y^2$, then there is a unique number θ with $0 \le \theta < 2\pi$ such that

$$\cos(\theta) = \frac{x}{r}, \ \sin(\theta) = \frac{y}{r}.$$

That number is

$$\theta = \cos^{-1}(x/r) = \sin^{-1}(y/r) = \tan^{-1}(y/x).$$

We can therefore express any complex number $z = x + iy$ as

$$z = |z|(\frac{x}{|z|} + i\frac{y}{|z|}) = |z|(\cos\theta + i\sin\theta) = |z|e^{i\theta},$$

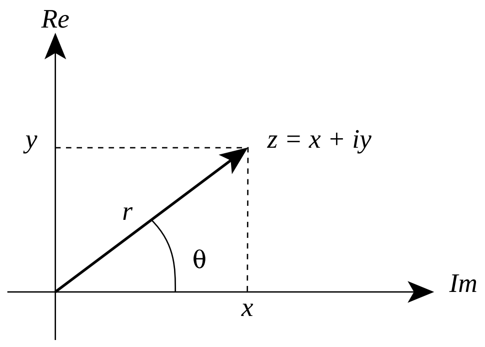


Figure B.1: A complex number z is represented in Cartesian coordinates as $z = x + iy$ and in polar coordinates as $z = re^{i\theta}$. The x-axis is called the **real axis**, the y axis is called the **imaginary axis**. The **angle** θ in radians is measured counter-clockwise from the real axis.

where $\theta = \tan^{-1}(y/x)$. The **angle** or **argument** θ is measured in radians, and it is written as $\arg(z)$ or $\angle z$. So we have the **polar representation** of any complex number z as

$$\boxed{z = x + iy = re^{i\theta}.} \tag{B.17}$$

The two representations are related by

$$r = |z| = \sqrt{x^2 + y^2}$$

and

$$\theta = \arg(z) = \tan^{-1}(y/x).$$

The values x and y are called the **Cartesian coordinates** of z, while r and θ are its **polar coordinates**. Note that r is real and $r \geq 0$.

Figure B.1 depicts the Cartesian and polar representations. Note that for any integer K,

$$re^{i(2K\pi+\theta)} = re^{i\theta}.$$

This is because

$$re^{i(2K\pi+\theta)} = re^{i2K\pi}e^{i\theta}$$

and

$$e^{i2K\pi} = \cos(2K\pi) + i\sin(2K\pi) = 1.$$

Thus, the polar coordinates (r, θ) and $(r, \theta + 2K\pi)$ for any integer K represent the same complex number. Thus, the polar representation is not unique; by convention, a unique polar representation can be obtained by requiring that the angle given by a value of θ satisfying $0 \leq \theta < 2\pi$ or $-\pi < \theta \leq \pi$, but we will rarely enforce this constraint.

Example 2.1: The polar representation of the number 1 is $1 = 1e^{i0}$. Notice that it is also true that $1 = 1e^{i2\pi}$, because the sine and cosine are periodic with period 2π. The polar representation of the number -1 is $-1 = 1e^{i\pi}$. Again, it is true that $-1 = 1e^{i3\pi}$, or, in fact, $-1 = 1e^{i\pi+K2\pi}$, for any integer K.

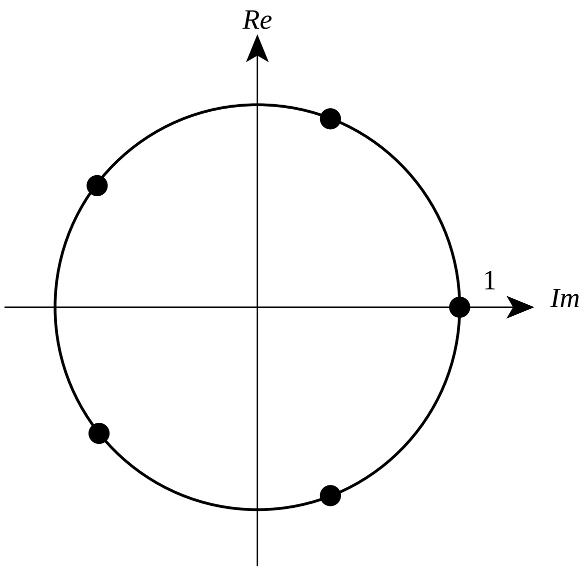


Figure B.2: The 5th roots of unity.

Products of complex numbers represented in polar coordinates are easy to compute. If $z_i = |r_i|e^{i\theta_i}$, then

$$z_1 z_2 = |r_1||r_2|e^{i(\theta_1+\theta_2)}.$$

Thus the magnitude of a product is a product of the magnitudes, and the angle of a product is the sum of the angles,

$$\boxed{|z_1 z_2| = |z_1||z_2|, \quad \angle(z_1 z_2) = \angle(z_1) + \angle(z_2).}$$

Example 2.2: We can use the polar representation to find the n distinct roots of the equation $z^n = 1$. Write $z = re^{i\theta}$, and $1 = 1e^{2k\pi}$, so

$$z^n = r^n e^{in\theta} = 1e^{i2k\pi},$$

which gives $r = 1$ and $\theta = 2k\pi/n$, $k = 0, 1, \cdots, n-1$. These are called the n **roots of unity**. Figure B.2 shows the 5 roots of unity.

Whereas it is easy to solve the polynomial equation $z^n = 1$, solving a general polynomial equation is difficult.

Theorem The polynomial equation

$$z^n + a_1 z^{n-1} + \cdots + a_{n-1} z + a_n = 0,$$

where $a_1, \cdots, a_n$ are complex constants, has exactly n factors of the form $(z - \alpha_i)$, where $\alpha_1, \cdots \alpha_n$ are called the n roots. In other words, we can always find the factorization,

$$z^n + a_1 z^{n-1} + \cdots + a_{n-1} z + a_n = \prod_{k=1}^{n} (z - \alpha_k) \, .$$

Some of the roots may be identical.

Note that although this theorem ensures the existence of this factorization, it does not suggest a way to find the roots. Indeed, finding the roots can be difficult. Fortunately, software for finding roots is readily available, for example using the Matlab `roots` function.

Exercises

Each problem is annotated with the letter **E, T, C** which stands for exercise, requires some thought, requires some conceptualization. Problems labeled **E** are usually mechanical, those labeled **T** require a plan of attack, those labeled **C** usually have more than one defensible answer.

1. **E** Simplify the following expressions:

   (a)
   $$\frac{3+4i}{5-6i} \times \frac{3+6i}{4-5i},$$

   (b)
   $$e^{2+\pi i}.$$

2. **E** Express the following in polar coordinates:
   $$2-2i,\ 2+2i, \frac{1}{2-2i}, \frac{1}{2+2i},\ 2i,\ -2i.$$

3. **E** Depict the following numbers graphically as in figureB.1:
   $$i1,\ -2,\ -3-i,\ -1-i.$$

4. **E** Find θ so that
   $$\mathit{Re}\{(1+i)e^{i\theta}\} = -1.$$

5. **E** Express the six distinct roots of unity, i.e. the six solutions to
   $$z^6 = 1$$
   in Cartesian and polar coordinates.

6. **T** Express the six roots of -1, i.e. the six solutions to
   $$z^6 = -1$$
   in Cartesian and polar coordinates. Depict these roots as in FigureB.2.

7. **T** Figure out i^n for all positive and negative integers n. (For a negative integer n, $z^{-n} = 1/z^n$.)

8. **T** Factor the polynomial $z^5 + 2$ as

$$z^5 + 2 = \prod_{k=1}^{5} (z - \alpha_k),$$

expressing the α_k in polar coordinates.

9. **C** How would you define $\sqrt{1+i}$? More generally, how would you define $\sqrt{z}$ for any complex number z?

10. **T** The logarithm of a complex number z is written $\log z$ or $\log(z)$. It can be defined as an infinite series, or as the inverse of the exponential, i.e. define $\log z = w$, if $e^w = z$. Using the latter definition, find the logarithm of the following complex numbers:

$$1,\ -1,\ i,\ -i,\ 1+i$$

More generally, if $z \neq 0$ is expressed in polar coordinates, what is $\log z$? For which complex numbers z is $\log z$ not defined?

11. **E** Use Matlab to answer the following questions. Let $z_1 = 2+3j$ and $z_2 = 4-2j$. **Hint**: Consult Matlab help on `i`, `j`, `exp`, `real`, `imag`, `abs`, `angle`, `conj`, and `complex`. Looking up "complex" in the help desk may also be helpful.

    (a) What is $z_1 + z_2$? What are the real and imaginary parts of the sum?

    (b) Express the sum in polar coordinates.

    (c) Draw by hand two rays in the complex plane, one from the origin to z_1 and the other from the origin to z_2. Now draw $z_1 + z_2$ and $z_1 - z_2$ on the same plane. Explain how you might systematically construct the sum and difference rays.

    (d) Draw two rays in the complex plane to $z_3 = -2 - 3j$ and $z_4 = 3 - 3j$. Now draw $z_3 \times z_4$ and z_3/z_4.

    (e) Consider $z_5 = 2e^{j\pi/6}$ and $z_6 = z_5^*$. Express z_6 in polar coordinates. What is $z_5 z_6$?

    (f) Draw the ray to $z_0 = 1 + 1j$. Now draw rays to $z_n = z_0 e^{jn\pi/4}$ for $n = 1, 2, 3, \ldots$. How many distinct z_n are there?

    (g) Find all the solutions of the equation $z^7 = 1$. Hint: Express z in polar coordinates, $z = re^{j\theta}$ and solve for r, θ.

12. **E** This problem explores how complex signals may be visualized and analyzed.

    (a) Use Matlab to plot the complex exponential function as follows:

    ```
    plot(exp((-2+10j)*[0:0.01:1]))
    ```

The result is a spiraling curve corresponding to the signal $f : [0,1] \to$ *Comps* where

$$\forall t \in [0,1] \quad f(t) = e^{(-2+10j)t}$$

. In the plot window, under the Tools menu item, use 'Axes properties' to turn on the grid. Print the plot and on it mark the points for which the function is purely imaginary. Is it evident what values of t yield purely imaginary $f(t)$?

(b) Find analytically the values of t that result in purely imaginary and purely real $f(t)$.

(c) Construct four plots, where the horizontal axis represents t and the vertical axis represents the real and imaginary parts of $f(t)$, and the magnitude and angle of $f(t)$. Give these as four subplots.

(d) Give the mathematical expressions for the four functions plotted above in part (c).

13. **T** Euler's formula is: for any real number θ,

$$e^{j\theta} = \cos\theta + j\sin\theta,$$

and the product formula is: for any complex numbers z_1, z_2,

$$e^{z_1+z_2} = e^{z_1}e^{z_2}.$$

The following problems show that these two formulas can be combined to obtain many useful identities.

(a) Express $\sin(2\theta)$ and $\cos(2\theta)$ as sums and products of $\sin\theta$ and $\cos\theta$. **Hint**: Write $e^{j2\theta} = e^{j\theta}e^{j\theta}$ (by the product formula) and then use Euler's formula.

(b) Express $\sin(3\theta)$ and $\cos(3\theta)$ also as sums and products of $\sin\theta$ and $\cos\theta$.

(c) The sum of several sinewaves of the *same* frequency ω but different phases is a sinewave of the same frequency, i.e. given A_k, ϕ_k, $k = 1, \ldots, n$, we can find A, ϕ so that

$$A\cos(\omega t + \phi) = \sum_{k=1}^{n} A_k \cos(\omega t + \phi_k)$$

Express A, ϕ in terms of $\{A_k, \phi_k\}$.

Appendix C

Laboratory Exercises

This appendix contains laboratory exercises based on Matlab and Simulink. The purpose of these exercises is to help reconcile the declarative (what is) and imperative (how to) points of view on signals and systems. The mathematical treatment that dominates in the main body of this text is declarative, in that it asserts properties of signals and studies the relationships between signals that are implied by systems. This appendix focuses on an imperative style, where signals and systems are constructed procedurally.

Matlab and Simulink, distributed by The MathWorks, Inc., are chosen as the basis for these exercises because they are widely used by practitioners in the field, and because they are capable of realizing interesting systems. Why use both Matlab and Simulink? Although they are integrated into a single package, Matlab and Simulink are two very different pieces of software with radically different approaches to modeling of signals and systems. Matlab is an imperative programming language, whereas Simulink is a block diagram language. In Matlab, one specifies the sequence of steps that construct a signal or operate on a signal to produce a new signal. In Simulink, one connects blocks that implement elementary systems to construct more interesting systems. The systems we construct are aggregates of simpler systems.

Matlab fundamentally operates on matrices and vectors. Simulink fundamentally operates on discrete and continuous-time signals. Discrete-time signals, of course, can be represented as vectors. Continuous-time signals, however, can only be approximated. Simulink, since it is a computer program, must of course approximate continuous-time signals as well by discretizing time. But that approximation is largely transparent, and the user (the model builder) can pretend that he or she is operating directly on continuous-time signals.

There is considerable value in becoming adept with these software packages. Matlab and Simulink are often used in practice for "quick-and-dirty" prototyping of concepts. In a matter of a few hours, very elaborate models can be constructed. This contrasts with the weeks or months that would often be required to build a hardware prototype to test the same concept.

Of course, a conventional programming language such as C++ or Java could also be used to construct prototypes of systems. However, these languages lack the rich libraries of built-in functions that Matlab and Simulink have. A task as conceptually simple as plotting a waveform can take weeks of programming in Java to accomplish well. Algorithms, such as the FFT or filtering algorithms, are also built in, saving considerable effort.

Matlab and Simulink both have capabilities that are much more sophisticated than anything covered in this text. This may be a bit intimidating at first ("what the heck is singular-value decomposition!?!?!?"). In fact, these tools are rich enough in functionality to keep you busy for an entire career in engineering. You will need to learn to ignore what you don't understand, and focus on building up your abilities gradually.

If you have no background in programming, these exercises will be difficult at first. Matlab, at its root, is a fairly conventionally programming language, and it requires a clear understanding of programming concepts such as variables and flow of control (for loops, while loops). As programming languages go, it is an especially easy one to learn, however. Its syntax (the way commands are written) is straightforward and close to that of the mathematical concepts that it emulates. Moreover, since it is an interpreted language (in contrast to a compiled language), you can easily experiment by just typing in commands at the console and seeing what happens. Be fearless! The worst that can happen is that you will have to start over again.

These labs assume the computer platform is Microsoft Windows, although any platform capable of running Matlab and Simulink will work, as long as it has full support for sound and images.

Mechanics of the labs

The labs are divided into two distinct sections, **in lab** and **independent**. The purpose of the in-lab section is to introduce concepts needed for later parts of the lab. Each in-lab section is designed to be completed during a scheduled lab time with an instructor present to clear up any confusing or unclear concepts. The in-lab section is completed by obtaining the signature of an instructor on a verification sheet.

The independent section begins where the in-lab section leaves off. It can be completed within the scheduled lab period, or may be completed on your own time. You should write a brief summary of your solution to the lab exercise and turn it in at the beginning of the next scheduled lab period. Your summary should clearly answer each question posed in the independent section of the lab.

The lab writeup should be kept simple. It must include the names of the members of the group (if the lab is done by a group), the time of the lab section, the name of the lab, and the date. It should then proceed to give clear answers to each of the questions posed by the lab. Matlab code should be provided in a fixed width font (Courier New, 10pt, for example) and plots should be clearly labeled and referenced in the writeup. Plots may be included directly in the flow of the analysis. If included on a separate page, two to eight plots should be placed on the same page, depending on the nature of the plots. You can copy Matlab plots into most word processors using the Copy Figure command in the Edit menu.

Here is an example of a response to a portion of a lab:

> 2. Simple Low Pass Filter
>
> Figure C.1 shows the data before (top) and after (bottom) the low pass filter. The low

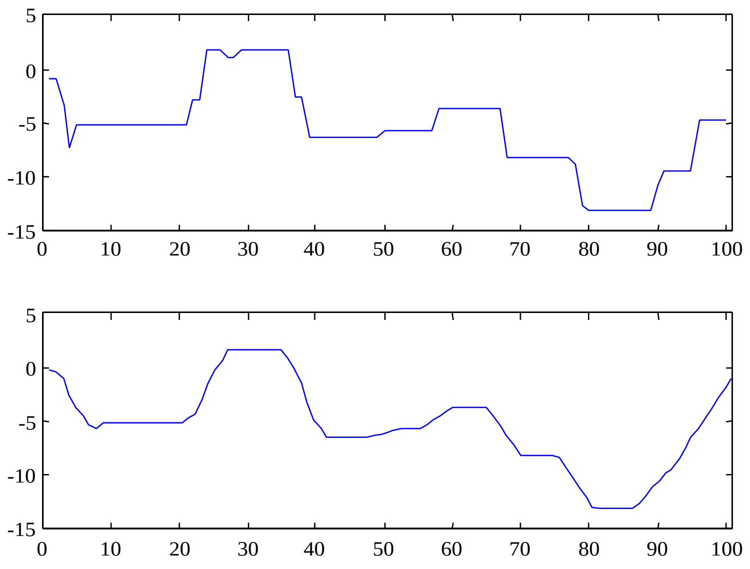


Figure C.1: Before and after LPF.

pass filter has the effect of smoothing sharp transitions in the original. For instance, notice the disappearance of the step from sample points 91 to 94. The MATLAB code used to generate the smoothed waveform v1 from the original waveform x1 is:

```
h5 = [1 1 1 1 1] / 5;
v1 = firfilt(x1, h5);
```

C.1 Arrays and sound

The purpose of this lab is to explore arrays in Matlab and to use them to construct sound signals. The lab is designed to help you become familiar with the fundamentals of Matlab. It is self contained, in the sense that no additional documentation for Matlab is needed. Instead, we rely on the on-line help facilities. Some people, however, much prefer to sit down with a tutorial text about a piece of software, rather than relying on on-line help. There are many excellent books that introduce Matlab. Check your local bookstore or The MathWorks' website (http://www.mathworks.com/).

Note that there is some potential confusion because Matlab uses the term "function" somewhat more loosely than we do when we refer to mathematical functions. Any Matlab command that takes arguments in parentheses is called a function. And most have a well-defined domain and range, and do, in fact, define a mapping from the domain to the range. These can be viewed formally as a (mathematical) functions. Some, however, such as `plot` and `sound` are a bit harder to view this way. The last exercise here explores this relationship.

C.1.1 In-lab section

To run Matlab simply double click on the Matlab icon on the desktop, or find the Matlab command in the start menu. This will open a Matlab command window, which displays a prompt "`>>`". You type commands at the prompt. Explore the built-in demos by typing `demo`.

Matlab provides an on-line help system accessible by using the `help` command. For example, to get information about the function `size`, enter the following:

```
>> help size
```

There also is a help desk (formatted in HTML for viewing from a web browser) with useful introductory material. It is accessed from the Help menu. If you have no prior experience with Matlab, see the topic "Getting Started" in the help desk. Spend some time with this. You can find in the help desk all the information you need to carry out the following exercises.

1. A variable in Matlab is an array. An array has dimensions $N \times M$, where N and M are in *Nats*. N is the number of **rows** and M is the number of **columns**. If $N = M = 1$, the variable is a **scalar**. If $N = 1$ and $M > 1$, then the variable is a **row vector**. If $N > 1$ and $M = 1$, then the variable is a **column vector**. If both N and M are greater than one, then the variable is a **matrix**, and if $N = M$ then the variable is a **square matrix**. The coefficients of an array are real or complex numbers.

   (a) Each of the following is an assignment of a value to a variable called `array`. For each, identify the dimensions of the array (M and N), and identify whether the variable is a scalar, row vector, column vector, or matrix.

   ```
   array = [1 2 3 4 5]
   array = [1:5]
   ```

```
array = 1:5
array = [1:1:5]
array = [1:-1:-5]
array = [1 2; 3 4]
array = [1; 2; 3; 4]
```

(b) Create a 2×3 matrix containing arbitrary data. Explore using the Matlab functions `zeros`, `ones`, `eye`, and `rand` to create the matrix. Find a way to use the square matrix `eye(2)` as part of your 2×3 matrix. Verify the sizes of your arrays using `size`.

(c) Use the Matlab commands `size` and `length` to determine the length of the arrays given by `1:0.3:10` and `1:1:-1`. Consider more generally the array constructor pattern

```
array = start : step : stop
```

where `start`, `stop`, and `step` are scalar variables or real numbers. How many elements are there in `array`? Give an expression in Matlab in terms of the variables `start`, `stop`, and `step`. That is, we should be able to do the following:

```
>> start = 1;
>> stop = 5;
>> step = 1;
>> array = start:step:stop;
```

and then evaluate your expression and have it equal `length(array)`. (Notice that the semicolons at the end of each command above suppress Matlab's response to each command.) Hint: to get a general expression, you will need something like the `floor` function. Verify your answer for the arrays `1:0.3:10` and `1:1:-1`.

2. Matlab can be used as a general-purpose programming language. Unlike a general-purpose programming language, however, it has special features for operating on arrays that make it especially convenient for modeling signals and systems.

(a) In this exercise, we will use Matlab to compute

$$\sum_{k=0}^{25} k.$$

Use a `for` loop (try `help for`) to specify each individual addition in the summation.

(b) Use the `sum` function to give a more compact, one-line specification of the sum in part (a). The difference between these two approaches illustrates the difference between using Matlab and using a more traditional programming language. The for loop is closer to the style one would use with C++ or Java. The `sum` function illustrates what Matlab does best: compact operations on entire arrays.

(c) In Matlab, any built-in function that operates on a scalar can also operate on an array. For example,

```
>> sin(pi/4)

ans =

    0.7071

>> sin([0 pi/4 pi/2 3*pi/4 pi])

ans =

         0    0.7071    1.0000    0.7071    0.0000
```

This feature is called **vectorization**. Use vectorization to construct a vector that tabulates the values of the sin function for the set $\{0, \pi/10, 2\pi/10, \cdots, \pi\}$. Use the colon notation explored in the previous exercise.

(d) Given two arrays `A` and `B` that have the same dimensions, Matlab can multiply the elements pointwise using the `.*` operator. For example,

```
>> [1 2 3 4].*[1 2 3 4]

ans =

     1     4     9    16
```

Use this pointwise multiply to tabulate the values of sin^2 for the set

$$\{0, \pi/10, 2\pi/10, \cdots, \pi\}.$$

3. A discrete-time signal may be approximated in Matlab by a vector (either a row or a column vector). In this exercise, you build a few such vectors and plot them.

(a) Create an array that is a row vector of length 36, with zeros everywhere except in the 18th position, which has value 1. (Hint: try `help zeros` to find a way to create a row vector with just zeros, and then assign the 18-th element of this vector the value one.) This array approximates a discrete-time **impulse**, which is a signal that is zero everywhere except at one sample point. We will use impulses to study linear systems. Plot the impulse signal, using both `plot` and `stem`.

(b) Sketch by hand the sine wave $x : [-1, 1] \to$ *Reals*, given by

$$\forall t \in [-1, 1], \quad x(t) = \sin(2\pi \times 5t + \pi/6).$$

In your sketch carefully plot the value at time 0. Assume the domain represents time in seconds. What is the frequency of this sine wave in Hertz and in radians per second, what is its period in seconds, and how many complete cycles are there in the interval $[-1, 1]$?

(c) Sample the function x from the previous part at 8 kHz, and using Matlab, plot the samples for the entire interval $[-1, 1]$. How many samples are there?

(d) Change the frequency of the sinewave from the previous section to 440 Hz and plot the signal for the interval $[-1, 1]$. Why is the plot hard to read? Plot the samples that lie in the interval $[0, 0.01]$ instead (this is a 10 msec interval).

(e) The Matlab function `sound` (see `help sound`) with syntax

```
sound(sampledSignal, frequency)
```

sends the one-dimensional array or vector `sampledSignal` to the audio card in your PC. The second argument specifies the sampling frequency in Hertz. The values in `sampledSignal` are assumed to be real numbers in the range $[-1.0, 1.0]$. Values outside this range are clipped to -1.0 or 1.0. Use this function to listen to the signal you created in the previous part. Listen to both a 10 msec interval and 2 second interval. Describe what you hear.

(f) Listen to

```
sound(0.5*sampledSignal,frequency)
```

and

```
sound(2*sampledSignal,frequency)
```

where `sampledSignal` is the signal you created in part (d) above. Explain in what way are these different from what you heard in the previous part. Listen to

```
sound(sampledSignal,frequency/2)
```

and

```
sound(sampledSignal,frequency*2)
```

Explain how these are different.

C.1.2 Independent section

1. Use Matlab to plot the following continuous-time functions $f\colon [-0.1, 0.1] \to$ *Reals*:

$$\begin{aligned}\forall t \in [-0.1, 0.1], \quad f(t) &= \sin(2\pi \times 100t)\\ \forall t \in [-0.1, 0.1], \quad f(t) &= \exp(-10t)\sin(2\pi \times 100t)\\ \forall t \in [-0.1, 0.1], \quad f(t) &= \exp(10t)\sin(2\pi \times 100t)\end{aligned}$$

The first of these a familiar sinusoidal signal. The second is a sinusoidal signal with a decaying exponential envelope. The third is a sinusoidal signal with a growing exponential envelope. Choose a sampling period so that the plots closely resemble the continuous-time functions. Explain your choice of the sampling period. Use `subplot` to plot all three functions in one tiled figure. Include the figure in your lab report.

2. Use Matlab to listen to a one-second sinusoidal waveform scaled by a decaying exponential given by

$$\forall t \in [0, 1], \quad f(t) = \exp(-5t)\sin(2\pi \times 440t).$$

Use a sample rate of 8 kHz. Describe how this sound is different from sinusoidal sounds that you listened to in the in-lab section.

3. Construct a sound signal that consists of a sequence of half-second sinusoids with exponentially decaying envelopes, as in the previous part, but with a sequence of frequencies: 494, 440, 392, 440, 494, 494, and 494. Listen to the sound. Can you identify the tune? In your lab report, give the Matlab commands that produce the sound. When the next lab meets, play the sound for your instructor.

4. This exercise explores the relationship between Matlab functions and mathematical functions.

   (a) The `sound` function in Matlab returns no value, as you can see from the following:

   ```
   >> x = sound(n)
   ??? Error using ==> sound
   Too many output arguments.
   ```

   Nonetheless, `sound` can be viewed as a function, with its range being the set of sounds. Read the help information on the `sound` function carefully and give a precise characterization of it as a mathematical function (define its domain and range). You may assume that the elements of Matlab vectors are members of the set *Doubles*, double-precision floating-point numbers, and you may, for simplicity, consider only the two-argument version of the function, and model only monophonic (not stereo) sound.

   (b) Give a similar characterization of the `soundsc` Matlab function, again considering only the two-argument version and monophonic sound.

   (c) Give a similar characterization of the `plot` Matlab function, considering the one argument version with a vector argument.

Instructor Verification Sheet for Lab C.1

Name: ______________________ Date: ______________________

1. Matlab arrays.

   Instructor verification: ______________________

2. Matlab programming.

   Instructor verification: ______________________

3. Discrete-time signals in Matlab.

   Instructor verification: ______________________

C.2 Images

The purpose of this lab to explore images and colormaps. You will create synthetic images and movies, and you will process a natural image by blurring it and by detecting its edges.

C.2.1 Images in Matlab

Figure C.2 shows a black and white image where the intensity of the image varies sinusoidally in the vertical direction. The top row of pixels in the image is white. As you move down the image, it gradually changes to black, and then back to white, completing one cycle. The image is 200×200 pixels so the vertical frequency is 1/200 cycles per pixel. The image rendered on the page is about 10×10 centimeters, so the vertical frequency is 0.1 cycles per centimeter. The image is constant horizontally (it has a horizontal frequency of 0 cycles per centimeter).

We begin this lab by constructing the Matlab commands that generate this image. To do this, you need to know a little about how Matlab represents images. In fact, Matlab is quite versatile with images, and we will only explore a portion of what it can do.

An image in Matlab can be represented as an array with two dimensions (a matrix) where each element of the matrix indexes a colormap. Consider for example the image constructed by the `image` command:

```
   >> v = [1:64];
   >> image(v);
```

This should create an image like that shown in figure C.3.

The image is 1 pixel high by 64 pixels wide (Matlab, by default, stretches the image to fit the standard rectangular graphic window, so the one pixel vertically is rendered as a very tall pixel.) You could use the `repmat` Matlab function to make an image taller than 1 pixel by just repeating this row some number of times.

The pixels each have value ranging from 1 to 64. These index the default colormap, which has length 64 and colors ranging from blue to red through the rainbow. To see the default colormap numerically, type

```
   >> map = colormap
```

To verify its size, type

```
>> size(map)

ans =

    64     3
```

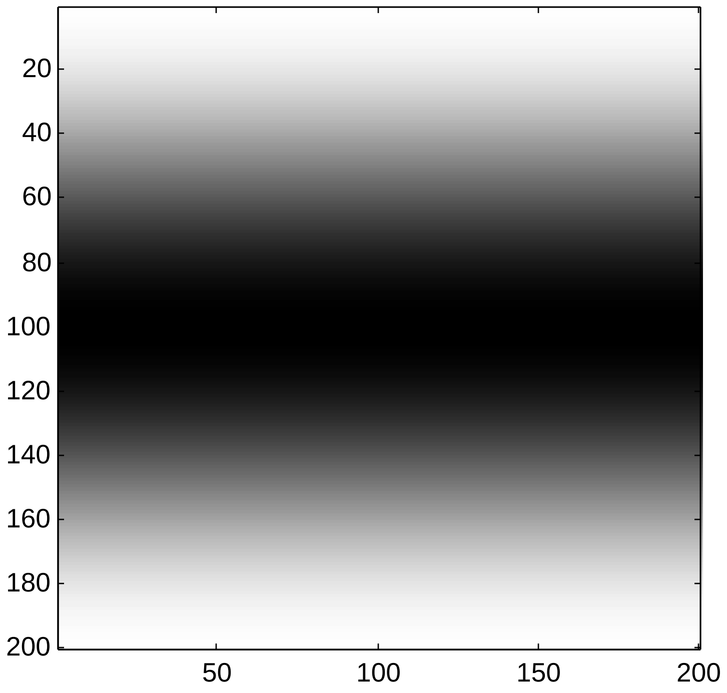


Figure C.2: An image where the intensity varies sinusoidally in the vertical direction.

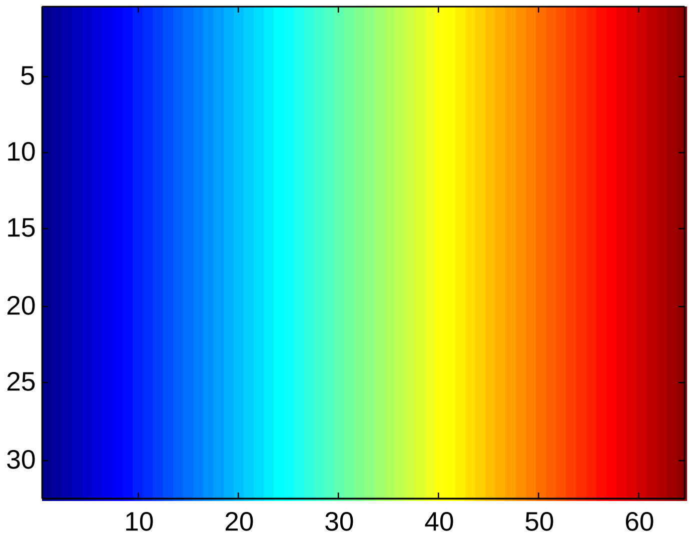


Figure C.3: An image of the default colormap.

Notice that it has 64 rows and three columns. Each row is one entry in the colormap. The three columns give the amounts of red, green, and blue respectively in the colormap. These amounts range from 0 (none of the color present) to 1.0 (the maximum amount of the color possible). Examine the colormap to convince yourself that it begins with blue and ends with red.

Change the colormap using the `colormap` command as follows:

```
>> map = gray(256);
>> colormap(map);
>> image([1:256]);
```

Examine the `map` variable to understand the resulting image. This is called a **grayscale colormap**.

C.2.2 In-lab section

1. What is the representation in a Matlab colormap for the color white? What about black?
2. Create a 200×200 pixel image like that shown in figure C.2. You will want the colormap set to `gray(256)`, as indicated above. Note that when you display this image using the `image` command, it probably will not be square. This is because of the (somewhat annoying) stretching that Matlab insists on doing to make the image fit the default graphics window. To disable the stretching and get a square image, issue the command `axis image`

   ```
   axis image
   ```

   As usual with Matlab, a brute-force way to create matrices is to use for loops, but there is almost always a more elegant (and faster) way that exploits Matlab's ability to operate on arrays all at once. Try to avoid using for loops to solve this and subsequent problems.
3. Change your image so that the sinusoidal variations are horizontal rather than vertical. Vary the frequency so that you get four cycles of the sinusoid instead of one. What is the frequency of this image?
4. An image can have both vertical and horizontal frequency content at the same time. Change your image so that the intensity at any point is the sum of a vertical and horizontal sinusoid. Be careful to stay with the numerical range that indexes the colormap.
5. Get the image file from

   ```
   http://www.eecs.berkeley.edu/˜eal/eecs20/images/helen.jpg
   ```

   Save it in some directory where you have write permission with the name “helen.jpg”. (**Note**: For reasons that only the engineers at Microsoft could possibly explain, Microsoft Internet Explorer does not allow you to save this file as a JPEG file, ’.jpg’. It only allows you to save the file as a bit map, ’.bmp’, which is already decoded. So we recommend using Netscape rather than IE.)

   In Matlab, change the current working directory to that directory using the `cd` command. Then use `imfinfo` to get information about the file, as follows:

```
>> imfinfo('helen.jpg')

ans =

           Filename: 'helen.jpg'
        FileModDate: '27-Jan-2000 10:48:16'
           FileSize: 18026
             Format: 'jpg'
      FormatVersion: ''
              Width: 200
             Height: 300
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
```

Make a note of the file size, which is given in bytes. Then use `imread` to read the image into Matlab and display it as follows:

```
>> helen = imread('helen.jpg');
>> image(helen);
>> axis image
```

Use the `whos` command to identify the size, in bytes, and the dimensions of the `helen` array. Can you infer from this what is meant by `'truecolor'` above? The file is stored in JPEG format, where JPEG, which stands for Joint Pictures Expert Group, is an image representation standard. The `imread` function in Matlab decodes JPEG images. What is the compression ratio achieved by the JPEG file format (the compression ratio is defined to be size of the uncompressed data in bytes divided by the size of the compressed data in bytes).

6. The `helen` array returned by `imread` has elements that are of type `uint8`, which means unsigned 8-bit integers. The possible values for such numbers are the integers from 0 to 255. The upper left pixel of the image can be accessed as follows:

```
>> pixel = helen(1,1,:)

pixel(:,:,1) =

   205


pixel(:,:,2) =

   205
```

```
pixel(:,:,3) =

   205

```

In this command, the final argument is ':' which means to Matlab, return all elements in the third dimension. The information about the result is:

```
>> whos pixel
  Name         Size         Bytes  Class

  pixel        1x1x3            3  uint8 array

Grand total is 3 elements using 3 bytes
```

Matlab provides the `squeeze` command to remove dimensions of length one:

```
>> rgb = squeeze(pixel)

rgb =

   205
   205
   205

```

Find the RGB values of the lower right pixel. By looking at the image, and correlating what you see with these RGB values, infer how white and black are represented in truecolor images.

Matlab can only do very limited operations arrays of this type.

C.2.3 Independent section

1. Construct a mathematical model for the Matlab `image` function as used in parts 3 and 4 of the in-lab section by giving its domain and its range. Notice that the colormap, although it is not passed to `image` as an argument, is in fact an argument. It is passed in the form of a **global variable**, the current colormap. Your mathematical model should show this as an explicit argument.

2. In Matlab, you can create a movie using the following template:

```
numFrames = 15;
m = moviein(numFrames);
for frame = 1:numFrames;
   ... create an image X ...
   image(X), axis image
   m(:,frame) = getframe;
end
movie(m)
```

The line with the `getframe` command grabs the current image and makes it a frame of the movie. Use this template to create a vertical sinusoidal image where the sine waves appear to be moving upwards, like waves in water viewed from above. Try `help movie` to learn about various ways to display this movie.

3. We can examine individually the contributions of red, green, and blue to the image by creating **color separations**. Matlab makes this very easy on truecolor images by providing its versatile array indexing mechanism. To extract the red portion of the `helen` image created above, we can simply do:

```
red = helen(:,:,1);
```

The result is a 300×200 array of unsigned 8-bit integers, as we can see from the following:

```
>> whos red
  Name      Size          Bytes  Class

  red     300x200         60000  uint8 array

Grand total is 60000 elements using 60000 bytes
```

(Note that, strangely, the `squeeze` command is not needed whenever the last dimension(s) collapse to size 1.) If we display this array, its value will be interepreted as indexes into the current color map:

```
image(red), axis image
```

If the current colormap is the default one, then the image will look very off indeed (and very colorful). Change the colormap to grayscale to get a more meaningful image:

```
map = gray(256);
colormap(map);
```

The resulting image gives the red portion of the image, albeit rendered in black and white. Construct a colormap to render it in red. Show the Matlab code that does this in your report (you need not show the image). Then give similar color separations for the green and blue portions. Again, showing the Matlab code is sufficient. **Hint**: Create a matrix to multiply pointwise by the `map` matrix above (using the `.*` operator) to zero out two of its three columns. The `zeros` and `ones` functions might be useful.

4. A moving average can be applied to an image, with the effect of blurring it. For simplicity, operate on a black and white image constructed from the above red color separation as follows:

```
>> bwImage = double(red);
>> image(bwImage), axis image
>> colormap(gray(256))
```

The first line converts the image to an array of doubles instead of unsigned 8-bit integers because Matlab cannot operate numerically on unsigned 8-bit integers. The remaining two lines simply display the image using a grayscale colormap.

Construct a new image where each pixel is the average of 25 pixels in the original image, where the 25 pixels lie in a 5×5 square. The new image will need to be slightly smaller than the original (figure out why). The result should be a blurred image because the moving average reduces the high frequency content of a signal, and sharp edges are high frequency phenomena.

5. A simple way to perform edge detection on a black-and-white image is to calculate the difference between a pixel and the pixel immediately above it and to the left of it. If either difference exceeds some threshold, we decide there is an edge at that position in the image. Perform this calculation on the image `bwImage` given in the previous part. To display with the edges, start with a white image the same size or slightly smaller than the original image. When you detect an edge at a pixel, replace the white pixel with a black one. The resulting image should resemble a line drawing of Helen. Experiment with various threshold values. **Hint**: To perform the threshold test, you will probably need the Matlab `if` command. Try `help if` and `help relop`.

   Note: Edge detection is often the first step in **image understanding**, which is the automatic interpretation of images. A common application of image understanding is **optical character recognition** or **OCR**, which is the transcription of printed documents into computer documents.

   The difference between pixels tends to emphasize high frequency content in the image and deemphasize low frequency content. This is why it is useful in detecting edges, which are high frequency content. This is obvious if we note that frequency in images refers to the rate of change of intensity over space. That rate is very fast at edges.

Instructor Verification Sheet for C.2

Name: ____________________ Date: ____________________

1. Representation in a colormap of white and black.

   Instructor verification: ____________________

2. Vertical sinusoidal image.

   Instructor verification: ____________________

3. Horizontal higher frequency image. Give the frequency.

   Instructor verification: ____________________

4. Horizontal and vertical sinusoidal image.

   Instructor verification: ____________________

5. Compression ratio.

   Instructor verification: ____________________

6. Representation in truecolor of white and black.

   Instructor verification: ____________________

C.3 State machines

State machines are sequential. They begin in a starting state, and react to a sequence of inputs by sequentially transitioning between states. Implementation of state machines in software is fairly straightforward. In this lab, we explore doing this systematically, and build up to an implementation that composes two state machines.

C.3.1 Background

Strings in Matlab

State machines operate on sequences of symbols from an alphabet. Sometimes, the alphabet is numeric, but more commonly, it is a set of arbitrary elements with names that suggest their meaning. For example, the input set for the answering machine in figure 3.1 is

$$Inputs = \{ring, offhook, end\ greeting, end\ message, absent\}.$$

Each element of the above set can be represented in Matlab as a string (try `help strings`). Strings are surrounded by single quotes. For example,

```
>> x = 'ring';
```

The string itself is an array of characters, so you can index individual characters, as in

```
>> x(1:3)

ans =

rin
```

You can join strings just as you join ordinary arrays,

```
>> y = 'the';
>> z = 'bell';
>> [x, y, z]

ans =

ringthebell
```

However, this is not necessarily what you want. You may want instead to construct an array of strings, where each element of the array is a string (rather than a character). Such a collection of strings can be represented in Matlab as a **cell array**,

```
>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
```

Notice the curly braces instead of the usual square braces. A cell array in Matlab is an array where the elements of the array are arbitrary Matlab objects (such as strings and arrays). Cell arrays are indexed like ordinary arrays, so

```
>> c(1)

ans =

    'ring'
```

Often, you wish to test a string to see whether it is equal to some string. You usually cannot compare strings or cells of a cell array using `==`, as illustrated here:

```
>> c = 'ring';
>> if (c == 'offhook') result = 1; end
??? Error using ==> ==
Array dimensions must match for binary array op.

>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
>> if (c(1) == 'ring') result = 1; end
??? Error using ==> ==
Function '==' not defined for variables of class 'cell'.
```

Strings should instead be compared using `strcmp` or `switch` (see the on-line help for these commands).

M-files

In Matlab, you can save programs in a file and execute them from the command line. The file is called an **m-file**, and has a name of the form *command*.m, where *command* is the name of the command that you enter on the command line to execute the program.

You can use any text editor to create and edit m-files, but the one built into Matlab is probably the most convenient. To invoke it, select “New” and “M-file” under the “File” menu.

To execute your program, Matlab needs to know where to find your file. The simplest way to handle this is to make the current directory in Matlab the same as the directory storing the m-file. For example, if you put your file in the directory

```
    D:\users\eal
```

then the following will make the file visible to Matlab

```
>> cd D:\users\eal
>> pwd

ans =

D:\users\eal
```

The `cd` command instructs Matlab to change the current working directory. The `pwd` command returns the current working directory (probably the mnemonic is *present* working directory).

You can instruct Matlab to search through some sequence of directories for your m-files, so that they do not have to all be in the same directory. See `help path`. For example, instead of changing the current directory, you could type

```
 path(path, 'D:\users\eal');
```

This command tells Matlab to search for functions wherever it was searching before (the first argument `path`) and also in the new directory.

Suppose you create a file called `hello.m` containing

```
   % HELLO - Say hello.
   disp('Hello');
```

The first line is a comment. The `disp` command displays its argument without displaying a variable name. On the command line, you can execute this

```
>> hello
Hello
```

Command names are not case sensitive, so `HELLO` is the same as `Hello` and `hello`. The comment in the file is used by Matlab to provide on-line help. Thus,

```
>> help hello

  HELLO - Say hello.
```

The M-file above is a program, not a function. There is no returned value. To define a function, use the `function` command in your m-file. For example, store the following in in a file `reverse.m`:

```
function result = reverse(argument)
% REVERSE - return the argument array reversed.
result = argument(length(argument):-1:1);
```

Then try:

```
>> reverse('hello')

ans =

olleh
```

The returned value is the string argument reversed.

A function can have any number of arguments and returned values. To define a function with two arguments, use the syntax

```
function [result1, result2] = myfunction(arg1, arg2)
```

and then assign values to `result1` and `result2` in the body of the file. To use such function, you must assign each of the return values to a variable as follows:

```
>> [r1, r2] = myfunction(a1, a2);
```

The names of the arguments and result variables are arbitrary.

C.3.2 In-lab section

1. Write a for loop that counts the number of occurrences of `'a'` in

   ```
   >> d = {'a' 'b' 'a' 'a' 'b'};
   ```

   Then define a function `count` that counts the number of occurrences of `'a'` in any argument. How many occurrences are there in the following two examples?

   ```
   >> x = ['a', 'b', 'c', 'a', 'aa'];
   >> y = {'a', 'b', 'c', 'a', 'aa'};
   >> count(x)

   ans =

        ??

   >> count(y)

   ans =

        ??
   ```

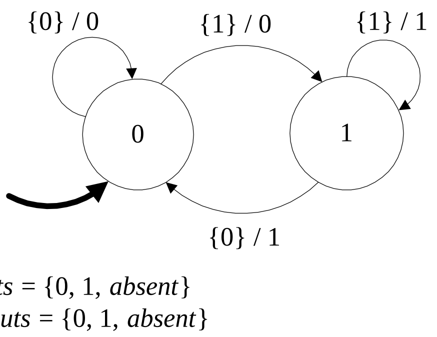


Figure C.4: A simple state machine.

Why are they different?

2. The `input` function can be used to interactively query the user for input. Write a program that repeatedly asks the user for a string and then uses your `count` function to report the number of occurrences of `'a'` in the string. Write the program so that if the user enters `quit` or `exit`, the program exits, and otherwise, it asks for another input. **Hint**: Try `help while` and `help break`.

3. Consider the state machine in figure C.4. Construct an m-file containing a definition of its *update* function. Then construct an m-file containing a program that queries the user for an input, then if the input is in the input alphabet of the machine, uses it to react, and then asks the user for another input. If the input is not in the input alphabet, the program should assume the input is *absent* and stutter. Be sure that your update function handles stuttering.

C.3.3 Independent section

1. Design a virtual pet,[1] in this case a cat, by constructing a state machine, writing an *update* function, and writing a program to repeatedly execute the function, as in (3) above. The cat should behave as follows:

   > It starts out *happy*. If you *pet* it, it *purrs*. If you *feed* it, it *throws up*. If *time passes*, it gets *hungry* and *rubs* against your legs. If you feed it when it is hungry, it purrs and gets happy. If you pet it when it is hungry, it *bites* you. If time passes when it is hungry, it *dies*.

   The italicized phrases in this description should be elements in either the state space or the input or output alphabets. Give the input and output alphabets and a state transition diagram. Define the *update* function in Matlab, and write a program to execute the state machine until the user types 'quit' or 'exit.'

[1]This problem is inspired by the Tamagotchi virtual pet made by Bandai in Japan. Tamagotchi which translates as "cute little egg," were extremely popular in the late 1990's, and had behavior considerably more complex than that described in this exercise.

2. Construct a state machine that provides inputs to your virtual cat so that the cat never dies. In particular, your state machine should generate *time passes* and *feed* outputs in such a way that the cat never reaches the *dies* state.

   Note that this state machine does not have particularly meaningful inputs. You can let the input alphabet be

   $$Inputs = \{1, absent\}$$

   where an input of 1 indicates that the machine should output a non-stuttering output, and an input of *absent* means it should output a stuttering output.

   Write a program where your feeder state machine is composed in cascade with your cat state machine, and verify (experimentally) that the cat does not die. Your state machine should allow time to pass (by producing an infinite number of 'time passes' outputs) but should otherwise be as simple as possible.

   Note that a major point of this exercise is to show that systematically constructed state machines can be very easily composed.

   The feeder state machine is called an **open-loop controller** because it controls the pet without observing the output of the pet. For most practical systems, it is not possible to design an open-loop controller. The next lab explores closed-loop controllers.

Instructor Verification Sheet for C.3

Name: ______________________ Date: ______________________

1. Count the number of occurrences of `'a'`. Understand the difference between a cell array and an array.

   Instructor verification: ______________________

2. Write a program with an infinite loop and user input.

   Instructor verification: ______________________

3. Construct and use *update* function.

   Instructor verification: ______________________

C.4 Control systems

This lab extends the previous one by introducing nondeterminism and feedback. In particular, you will modify the virtual pet that you constructed last time so that it behaves nondeterministically. The modification will make it impossible to keep the pet alive by driving it with another state machine in a cascade composition. You will instead have to use a feedback composition.

This scenario is typical of a control problem. The pet is a system to be controlled, with the objective of keeping it alive. You will construct a controller that observes the output of the virtual pet, and based on that output, constructs an appropriate input that will keep the pet alive. Since this controller observes the output of the pet, and provides input to the pet, it is called a **closed-loop controller**.

C.4.1 Background

Nondeterministic state machines have a *possibleUpdates* function rather than an *update* function. The *possibleUpdates* function returns a set of possible updates. You will construct this function to return a cell array, which was explored in the previous lab.

A software implementation of a nondeterministic state machine can randomly choose from among the results returned by *possibleUpdates*. It could conceptually flip coins to determine which result to choose each time. In software, the equivalent of coin flips is obtained through **pseudo-random number generators**. The Matlab function `rand` is just such a pseudo-random number generator. The way that it works is that each time you use it, it gives you a new number (try `help rand`).

For this lab, you will need to be able to use cell arrays in more sophisticated ways than in the previous lab. Recall that a cell array is like an ordinary array, except that the elements of the array can be arbitrary Matlab objects, including strings, arrays, or even cell arrays. A cell array can be constructed using curly braces instead of square brackets, as in

```
>> letters = {'a', 'b', 'c', 'd', 'e'};
>> whos letters
  Name           Size          Bytes  Class

  letters        1x5             470  cell array

Grand total is 10 elements using 470 bytes
```

The elements of the cell array can be accessed like elements of any other array, but there is one subtlety. If you access an element in the usual way, the result is a cell array, which might not be what you expect. For example,

```
>> x = letters(2)

x =
    'b'

>> whos x
  Name      Size         Bytes  Class

  x         1x1             94  cell array

Grand total is 2 elements using 94 bytes
```

To access the element as a string (or whatever the element happens to be), then use curly braces when indexing the array, as in

```
>> y = letters{2}

y =

b

>> whos y
  Name      Size         Bytes  Class

  y         1x1              2  char array

Grand total is 1 elements using 2 bytes
```

Notice that now the result is a character array rather than a 1×1 cell array.

You can also use curly braces to construct a cell array piece by piece. Here, for example, we construct and display a two-dimensional cell array of strings, and then access one of the elements as a string.

```
>> t{1,1} = 'upper left';
>> t{1,2} = 'upper right';
>> t{2,1} = 'lower left';
>> t{2,2} = 'lower right';
>> t

t =

    'upper left'    'upper right'
    'lower left'    'lower right'

>> t{2,1}

ans =
lower left
```

You can find out the size of a cell array in the usual way for arrays

```
>> [rows, cols] = size(t)

rows =

     2


cols =

     2
```

You can also extract an entire row or column from the cell array the same way you do it for ordinary arrays, using ':' in place of the index. For example, to get the first row, do

```
t(1,:)

ans =

    'upper left'    'upper right'
```

C.4.2 In-lab section

1. Construct a Matlab function `select` that, given a cell array with one row as an argument, returns a randomly chosen element of the cell array. Use your function to generate a random sequence of 10 letters from the cell array

   ```
   >> letters = {'a', 'b', 'c', 'd', 'e'};
   ```

   Hint: The Matlab function `floor` combined with `rand` might prove useful to get random indexes into the cell array.

2. Construct a Matlab function `chooserow` that, given a cell array with one or more rows, randomly chooses one of the rows and returns it as a cell array. Apply your function a few times to the '`t`' array that we constructed above.

3. A nondeterministic state machine has a *possibleUpdates* function rather than *updates*. This function returns a set of pairs, where each pair is a new state and an output.

A convenient Matlab implementation is a function that returns a two-dimensional cell array, with each of the possible updates on one row. As a first step towards this, modify your realization of the *update* function for the virtual cat of the previous lab so that it returns a 1×2 cell array with the next state and output. Also modify your program that runs the cat (without the driver) so that it uses your new function. Verify that the cat still works properly.

4. Now modify the cat's behavior so that if it is hungry and you feed it, it sometimes gets happy and purrs (as it did before), but it sometimes stays hungry and rubs against your legs. I.e., change your *update* function so that if the state is hungry and you feed the cat, then return a 2×2 cell array where the two rows specify the two possible next state, output pairs. Modify the program that runs the cat to use your `chooserow` function to choose from among the options.

5. Compose your driver machine from the previous lab with your nondeterministic cat, and verify that the driver no longer keeps the cat alive. In fact, no open-loop controller will be able to keep the cat alive and allow time to pass. In the independent section of this lab, you will construct a **closed-loop controller** that keeps the cat alive. It is a feedback composition of state machines.

C.4.3 Independent section

Design a deterministic state machine that you can put into a feedback composition with your non-deterministic cat so that the cat is kept alive and time passes. Give the state transition diagram for your state machine and write a Matlab function that implements its *update* function. Write a Matlab program that implements the feedback composition.

Note that your program that implements the feedback composition faces a challenging problem. When the program starts, neither the inputs to the controller machine nor the inputs to the cat machine are available. So neither machine can react. For your controller machine, you should define Matlab functions for both *update*, which requires a known input, and *output*, which does not. The *output* function, given the current state, returns the output that will be produced by the next reaction, if it is known, or *unknown* if it is not known. In the case of your controller, it should always be known, or the feedback composition will not be well formed.

Verify (by running your program) that the cat does not die.

Instructor Verification Sheet for C.4

Name: ____________________ Date: ____________________

1. Generated random sequence of letters using `'select'`.

   Instructor verification: ____________________

2. Applied `chooserow` to the `'t'` array.

   Instructor verification: ____________________

3. The cat still works with the update function returning a cell array.

   Instructor verification: ____________________

4. The nondeterministic sometimes stays hungry when fed.

   Instructor verification: ____________________

5. The nondeterministic cat dies under open-loop control.

   Instructor verification: ____________________

C.5 Difference equations

The purpose of this lab is to construct difference equation models of systems and to study their properties. In particular, we experimentally examine **stability** by constructing stable, unstable, and marginally stable systems. We will also introduce elementary complexity measures. The principal new Matlab skill required to develop these concepts is matrix operations.

C.5.1 In-lab section

1. Matlab is particularly good at matrix arithmetic. In this problem, we explore matrix multiplication (see sidebar on page 139).

   (a) Consider the 2×2 matrix
   $$M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$
   Without using Matlab, give M^n, for $n = 0, 1, 2, 3$. Recall that by mathematical convention, for any square matrix M, $M^0 = I$, the identity matrix, so in this case,
   $$M^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$
   Guess the general form of the matrix M^n. That is, give an expression for each of the elements of the matrix M^n.

   (b) Use Matlab to compute M^{25}. Was your guess correct? Calculate a few more values using Matlab until your guess is correct.

   (c) If your guess was correct, try to show it using induction. That is, first show that your guess for M^n is correct for some fixed n, like for example $n = 0$. Then assume your guess for M^n is correct is for some fixed n, and show that it is correct for M^{n+1}.

2. A vector is a matrix where either the number of rows is one (in the case of a **row vector**) or the number of columns is one (in the case of a **column vector**). Let
   $$b = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$
   be a column vector. We can equally well write this $b = [2, 3]^T$, where the superscript T indicates that the row vector $[2, 3]$ is transposed to make a column vector.

   (a) Create a column vector in Matlab equal to b above. Multiply it by M, given in the previous problem. Try forming both bM and Mb. Why does only one of these two work?

   (b) Create a row vector by transposing b. (Try `help transpose` or look up "transpose" in the help desk.) Multiply this transpose by M. Try both $b^T M$ and Mb^T. Why does only one of them work?

3. Consider a 2-dimensional difference equation system given by

$$A = \sigma \begin{bmatrix} \cos(\omega) & -\sin(\omega) \\ \sin(\omega) & \cos(\omega) \end{bmatrix}, \; b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \; c = \sigma \begin{bmatrix} -\cos(\omega) \\ \sin(\omega) \end{bmatrix}, d = 0,$$

   where $\omega, \sigma \in$ *Reals*. Note that this is similar to the systems studied in exercises 5 and 8 of chapter 5, with the differences being the multiplying constant σ and the c vector. Let $\omega = \pi/8$ and plot the first 100 samples of the zero-state impulse response for the following values of σ.

   (a) $\sigma = 1$.
   (b) $\sigma = 0.95$.
   (c) $\sigma = 1.05$.
   (d) For which values of σ is the result periodic? What is the period? The system producing the periodic output is called an **oscillator**.
   (e) You have constructed three distinct difference equation systems. One of these is a **stable system**, one is an **unstable system**, and one is a **marginally stable**. Which is which? You can infer the answer from the ordinary English-language meaning of the word "stable." What will happen if the unstable system is allowed to continue to run beyond the 100 samples you calculated?

C.5.2 Independent section

1. In lab C.1 you constructed a sound waveform f: *Reals* $\rightarrow$ *Reals* given by

$$\forall\, t \in [0, 1], \quad f(t) = \exp(-5t)\sin(2\pi \times 440t).$$

   You wrote a Matlab script that calculated samples of this waveform at a sample rate of 8 kHz. In this lab, we will construct the same waveform in a very different way, using difference equations.

   Construct a difference equation system with impulse response given by

$$\forall\, n \in \textit{Nats}_0, \quad h(n) = \exp(-5n/8000)\sin(2\pi \times 440n/8000).$$

   Give the matrix A, the vectors b, and c, and the scalar d of (5.27) and (5.28). Also give a Matlab program that produces the first 8000 samples of this impulse response and plays it as a sound. **Hint**: You will need to understand what you did in problem 3 of the in-lab section, and you may find it useful to use the results of exercise 8 in chapter 5.

2. For the system with the impulse response constructed in part 1, change the input so it consists of an impulse every 1/5 of a second. I.e., at an 8kH sample rate,

$$x(n) = \begin{cases} 1 & \text{if } n \text{ is a multiple of } 1600 \\ 0 & \text{otherwise} \end{cases}$$

   Write a Matlab script that plays two seconds of sound with this input. **NOTE**: This is a simplified model of a guitar string being repeatedly plucked. The model is justifiable on physical grounds, although it is a fairly drastic simplification.

3. Compare the complexity of the state machine model and the one you constructed in labC.1. In particular, assuming in each case that you generate one second of sound at an 8kHz sample rate, count the number of scalar multiplications and additions that must be done to construct the sound vector. In the realization in lab C.1, you used the built-in Matlab functions `exp` and `sin`. These functions are surprisingly expensive to compute, so count each evaluation of `exp` or `sin` on a scalar argument as 20 multiplications and 15 additions (they are actually typically more expensive even than this). You should find that the state machine realization is far less expensive by this measure. Do not count the cost of the Matlab `sound` function, which we can't easily determine.

Instructor Verification Sheet for C.5

Name: ______________________ Date: ______________________

1. Matrix multiplication in Matlab, and induction demonstration.

   Instructor verification: ______________________

2. Matrix-vector multiplication.

   Instructor verification: ______________________

3. Sinusoids with exponential envelopes; stability.

   Instructor verification: ______________________

C.6 Differential equations

The purpose of this lab is to experiment with models of continuous-time systems that are described as differential equations. The exercises aim to solidify state-space concepts while giving some experience with software that models continuous-time systems.

The lab uses Simulink, a companion to Matlab. The lab is self contained, in the sense that no additional documentation for Simulink is needed. Instead, we rely on the on-line help facilities. Be warned, however, that these are not as good for Simulink as for Matlab. The lab exercise will guide you, trying to steer clear of the more confusing parts of Simulink.

Simulink is a block-diagram modeling environment. As such, it has a more declarative flavor than Matlab, which is imperative. You do not specify exactly how signals are computed in Simulink. You simply connect together blocks that represent systems. These blocks declare a relationship between the input signal and the output signal.

Simulink excels at modeling continuous-time systems. Of course, continuous-time systems are not directly realizable on a computer, so Simulink must **simulate** the system. There is quite a bit of sophistication in how this is done. The fact that you do not specify how it is done underscores the observation that Simulink has a declarative flavor.

The simulation is carried out by a **solver**, which examines the block diagram you have specified and constructs an execution that simulates its behavior. As you read the documentation and interact with the software, you will see various references to the solver. In fact, Simulink provides a variety of solvers, and many of these have parameters you can control. Indeed, simulation of continuous-time systems is generally inexact, and some solvers work better on some models than others. The models that we will construct work well with the default solver, so we need not be concerned with this (considerable) complication.

Simulink can also model discrete-time systems, and (a bit clumsily) mixed discrete and continuous-time systems. We will emphasize the continuous-time modeling because this cannot be done (conveniently) in Matlab, and it is really the strong suit of Simulink.

C.6.1 Background

To run Simulink, start Matlab and type `simulink` at the command prompt. This will open the Simulink library browser. To explore Simulink demos, at the Matlab command prompt, type `demo`, and then find the Simulink item in the list that appears. To get an orientation about Simulink, open the help desk (using the Help menu), and find Simulink. Much of what is in the help desk will not be very useful to you. Find a section with a title “Building a Simple Model” or something similar and read that.

We will build models in state-space form, as in chapter 5, and as in the previous lab, but in continuous time. A continuous-time state-space model for a linear system has the form (see section 5.6)

$$\dot{z}(t) = Az(t) + bv(t) \tag{C.1}$$

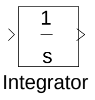
$$w(t) = cz(t) + dv(t) \tag{C.2}$$

where

- z: *Reals* $\to$ *Reals*^{N} gives the state response;
- $\dot{z}(t)$ is the derivative of z evaluated at $t \in$ *Reals*;
- v: *Reals* $\to$ *Reals* is the input signal; and
- w: *Reals* $\to$ *Reals* is the output signal.

The input and output are scalars, so the models are SISO , but the state is a vector of dimension N, which in general can be larger than one. The derivative of a vector z is simply the vector consisting of the derivative of each element of the vector.

The principle that we will follow in modeling such a system is to use an Integrator block, which looks like this in Simulink:



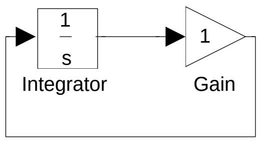
This block can be found in the library browser under "Simulink" and "Continuous." Create a new model by clicking on the blank-document icon at the upper left of the library browser, and drag an integrator into it. You should see the same icon as above.

If the input to the integrator is $\dot{z}$, then the output is z (just think about what happens when you integrate a derivative). Thus, the pattern we will follow is to provide as the input to this block a signal $\dot{z}$.

We begin with a one-dimensional system ($N = 1$) in order to get familiar with Simulink. Consider the scalar differential equation

$$\dot{z}(t) = az(t) \tag{C.3}$$

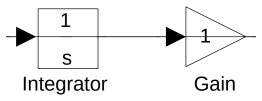
where $a \in$ *Reals* is a given scalar and z: *Reals* $\to$ *Reals* and $z(0)$ is some given initial state. We will set things up so that the input to the integrator is $\dot{z}$ and the output is z. To provide the input, however, we need the output, since $\dot{z}(t) = az(t)$. So we need to construct a feedback system that looks like this:



This model seems self-referential, and in fact it is, just as is (C.3).

Construct the above model. You can find the triangular "`Gain`" block in the library browser under "Simulink" and "Math." To connect the blocks, simply place the cursor on an output port and click and drag to an input port.

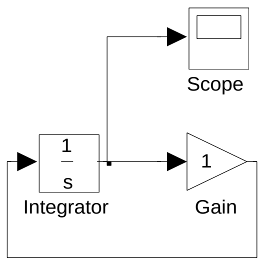
After constructing the feedback arc, you will likely see the following:



This is simply because Simulink is not very smart about routing your wires. You can stretch the feedback wire by clicking on it and dragging downwards so that it does not go over top of the blocks.

This model, of course, has no inputs, no initial state, and no outputs, so will not be very interesting to run it. You can set the initial state by double clicking on the integrator and filling in a value under "initial condition." Set the initial state to 1. Why is the initial state a property of the integrator? Because its output at time t is the state at time t. The "initial condition" parameter gives the output of the integrator when the model starts executing. Just like the feedback compositions of state machines in chapter 4, we need at least one block in the feedback loop whose output can be determined without knowing its input.

You will want to observe the output. To do this, find a block called "Scope" under "Simulink" and "Sinks" in the library browser, and drag it into your design. Connect it so that it displays the output of the integrator, as follows:



To make the connection, you need to hold the Control key while dragging from the output port of the integrator to the input port of the Scope. We are done with the basic construction of the model. Now we can experiment with it.

C.6.2 In-lab section

1. Set the gain of the gain block by double clicking on the triangular icon. Set it to -0.9. What value of a does this give you in the equation (C.3)?

2. Run the model for 10 time units (the default). To run the model, choose "Start" under the "Simulation" menu of the model window. To control the number of time units for the simulation, choose "Parameters" under the "Simulation" menu. To examine the result, double click on the Scope icon. Clicking on the binoculars icon in the scope window will result in a better display of the result.

3. Write down analytically the function z given by this model. You can guess its form by examining the simulation result. Verify that it satisfies (C.3) by differentiating.

4. Change the gain block to have value 0.9 instead of -0.9 and re-run the model. What happens? Is the system stable? (Stable means that if the input is bounded for all time, then the output is bounded for all time. In this case, clearly the input is bounded since it is zero.) Give an analytical formula for z for this model.

5. Experiment with values of the gain parameter. Determine over what range of values the system is stable.

C.6.3 Independent section

Continuous-time linear state-space models are reasonable for some musical instruments. In this exercise, we will simulate an idealized and a more realistic tuning fork, which is a particularly simple instrument to model. The model will be two-dimensional continuous-time state-space model.

Consider the state and output equations (C.1) and (C.2). Since the model is two dimensional, the state at each time is now a two-dimensional vector. The "initial condition" parameter of the Integrator block in Simulink can be given a vector. Set the initial value to the column vector

$$z(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{C.4}$$

The factor A must be a 2×2 matrix if the state is a two dimensional column vector. Unfortunately, the Gain block in Simulink cannot be given a matrix parameter. You must replace the Gain block with the MatrixGain block, also found in the "Math" library under "Simulink" in the library browser.

At first, we will assume there is no input, and we will examine the state response. Thus, we are only concerned at first with the simplified state equation

$$\dot{z}(t) = Az(t). \tag{C.5}$$

Recall that in chapter 2, equation (2.11) states that the displacement $x(t)$ at time t of a tine of the tuning fork satisfies the differential equation

$$\ddot{x}(t) = -\omega_0^2 x(t)$$

where ω_0 is constant that depends on the mass and stiffness of the tine, and and where $\ddot{x}(t)$ denotes the second derivative with respect to time of x (see box on page51). This does not have the form of (C.5). However, we can put it in that form using a simple trick. Let

$$z(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$$

and observe that

$$\dot{z}(t) = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix}.$$

Thus, we can write (C.5) as

$$\dot{z}(t) = \begin{bmatrix} \dot{x}(t) \\ \ddot{x}(t) \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$$

for suitably chosen constants $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$.

1. Find $a_{1,1}$, $a_{1,2}$, $a_{2,1}$, and $a_{2,2}$ for the tuning fork model.

2. Use Simulink to plot the state response of the tuning fork when the initial state is given by (C.4). You will have to pick a value of ω_0. Use Simulink to help you find a value of ω_0 so that the state completes one cycle in 10 time units. Each sample of the state response has two elements. These represent the displacement and speed, respectively, of the tuning fork tine in the model. The displacement is what directly translates into sound.

3. Change ω_0 so that the state has a frequency of 440 Hz, assuming the time units are seconds. Change the simulation parameters so that you run the model through 5 complete cycles.

4. Change the simulation parameters so that you run the model through 1 second. Use the Simulink To Workspace block to write the result to the workspace, and then use the Matlab soundsc function to listen to it. **Note**: You will need to set the sample time parameter of the To Workspace block to 1/8000. You will also need to specify that the save format should be a matrix. For your lab report, print your block diagram and annotate it with all the parameters that have values different from the defaults.

5. In practice, a tuning fork will not oscillate forever as the model does. We can add damping by modifying the matrix A. Try replacing the zero value of $a_{2,2}$ with -10. What happens to the sound? This is called **damping**. Experiment with different values for $a_{2,2}$. Describe how the different values affect the sound. Determine (experimentally) for what values of $a_{2,2}$ the system is stable.

6. A tuning fork is not much of a musical instrument. Its sound is too pure (spectrally). A guitar string, however, operates on similar principles as the tuning fork, but has a much more appealing sound.

   A tuning fork vibrates with only one mode. A guitar string, however, vibrates with multiple modes, as illustrated in figure C.5. Each of these vibrations produces a different frequency. The top one in the figure produces the lowest frequency, called the **fundamental**, which is typically the frequency of the note being played, such as 440 Hz for A-440. The next mode produces a component of the sound at twice that frequency, 880 Hz; this component is called the **first harmonic**. The third produces three times the frequency, 1320 Hz, and the fourth produces four times the fundamental, 1760 Hz; these components are the second and third harmonics.

   If the guitar string is undamped, and the fundamental frequency is f_0 Hz, then the the combined sound is a linear combination of the fundamental and the three (or more) harmonics. This can be written as a continuous-time function y where for all $t \in$ *Reals*,

$$y(t) = \sum_{k=0}^{N} c_k sin(2\pi f_k t)$$

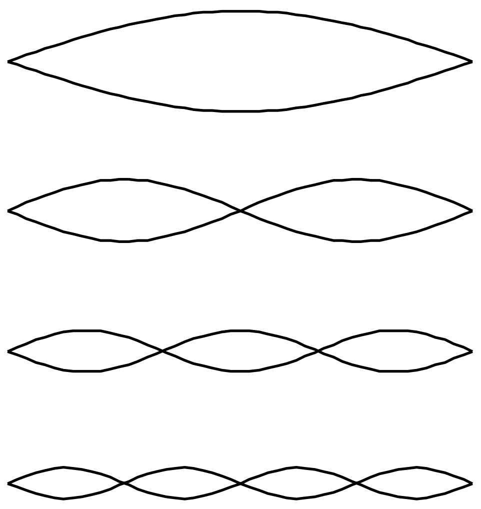


Figure C.5: Four modes of vibration of a guitar string.

where N is the number of harmonics and c_k gives the relative weights of these harmonics. The values of c_k will depend on the guitar construction and how it is played, and affect the **timbre** of the sound.

The model you have constructed above generates a damped sinusoid at 440 Hz. Create a Simulink model that produces a fundamental of 440 Hz plus three harmonics. Experiment with the amplitudes of the harmonics relative to the fundamental, as well as with the rates of decay of the four components. Note how the quality of the sound changes. Your report should include a printout of your model with the parameter values that you have chosen to get a sound like that of a plucked string.

Instructor Verification Sheet for C.6

Name: ______________________ Date: ______________________

1. Value of a.

   Instructor verification: ______________________

2. Plot of the state response.

   Instructor verification: ______________________

3. Formula for function z. Verified by differentiating.

   Instructor verification: ______________________

4. Formula for function z.

   Instructor verification: ______________________

5. Range of values for the gain over which the system is stable.

   Instructor verification: ______________________

C.7 Spectrum

The purpose of this lab is to learn to examine the frequency domain content of signals. Two methods will be used. The first method will be to plot the discrete Fourier series coefficients of finite signals. The second will be to plot the Fourier series coefficients of finite segments of time-varying signals, creating what is known as a **spectrogram**.

C.7.1 Background

A finite discrete-time signal with p samples has a discrete-time Fourier series expansion

$$x(n) = A_0 + \sum_{k=1}^{(p-1)/2} A_k \cos(k\omega_0 n + \phi_k) \tag{C.6}$$

for p odd and

$$x(n) = A_0 + \sum_{k=1}^{p/2} A_k \cos(k\omega_0 n + \phi_k) \tag{C.7}$$

for p even, where $\omega_0 = 2\pi/p$.

A finite signal can be considered to be one cycle of a periodic signal with fundamental frequency ω_0, in units of radians per sample, or $1/p$ in Hertz. In this lab, we will assume p is always even, and we will plot the magnitude of each of the frequency components, $|A_0|, \cdots, |A_{p/2}|$ for each of several signals, in order to gain intuition about the meaning of these coefficients.

Notice that each $|A_k|$ gives the amplitude of the sinusoidal component of the signal at frequency $k\omega_0 = k2\pi/p$, which has units of radians per sample. In order to interpret these coefficients, you will probably want to convert these units to Hertz. If the sampling frequency is f_s samples per second, then you can do the conversion as follows (see box on page151):

$$\frac{(k2\pi/p)[\text{radians/sample}]f_s[\text{samples/second}]}{2\pi[\text{radians/cycle}]} = kf_s/p[\text{cycles/second}]$$

Thus, each $|A_k|$ gives the amplitude of the sinusoidal component of the signal at frequency kf_s/p Hz.

Note that Matlab does not have any built-in function that directly computes the discrete Fourier series coefficients. However, it does have a realization of the fast Fourier transform, a function called `fft`, which can be used to construct the Fourier series coefficients. In particular, `fourierSeries` is a function that returns the DFS coefficients[2]:

```
function [magnitude, phase] = fourierSeries(x)
% FOURIERSERIES - Return the magnitude and phase of each
% sinusoidal component in the Fourier series expansion for
```

[2]This function can be found at http://www.eecs.berkeley.edu/ eal/eecs20/matlab/fourierSeries.m.

```
% the argument, which is interpreted as one cycle of a
% periodic signal.  The argument is assumed to have an
% even number p of samples. The first returned value is an
% array containing the amplitudes of the sinusoidal
% components in the Fourier series expansion, with
% frequencies 0, 1/p, 2/p, ... 1/2. The second returned
% value is an array of phases for the sinusoidal
% components. Both returned values are arrays with length
% (p/2)+1.
p = length(x);
f = fft(x)/p;
magnitude(1) = abs(f(1));
upper = p/2;
magnitude(2:upper) = 2*abs(f(2:upper));
magnitude(upper+1) = abs(f(upper+1));
phase(1) = angle(f(1));
phase(2:upper) = angle(f(2:upper));
phase(upper+1) = angle(f(upper+1));
```

In particular, if you have an array `x` with even length,

```
  [A, phi] = fourierSeries(x);
```

returns the DFS coefficients in a pair of vectors.

To plot the magnitudes of the Fourier series coefficients vs. frequency, you can simply say

```
  p = length(x);
  frequencies = [0:fs/p:fs/2];
  plot(frequencies, A);
  xlabel('frequency in Hertz');
  ylabel('amplitude');
```

where `fs` has been set to the sampling frequency (in samples per second). The line

```
  frequencies = [0:fs/p:fs/2];
```

bears further examination. It produces a vector with the same length as `A`, namely $1 + p/2$, where p is the length of the `x` vector. The elements of the vector are the frequencies in Hertz of each Fourier series component.

C.7.2 In-lab section

1. To get started, consider the signal generated by

```
t = [0:1/8000:1-1/8000];
x = sin(2*pi*800*t);
```

This is 8000 samples of an 800 Hz sinusoid sampled at 8 kHz. Listen to it. Use the `fourierSeries` function as described above to plot the magnitude of its discrete Fourier series coefficients. Explain the plot.

Consider the continuous-time sinusoid

$$x(t) = \sin(2\pi 800t).$$

The `x` vector calculated above is 8000 samples of this sinusoid taken at a sample rate of 8 kHz. Notice that the frequency of the sinusoid is the derivative of the argument to the sine function,

$$\omega = \frac{d}{dt} 2\pi 800t = 2\pi 800$$

in units of radians per second. This fact will be useful below when looking at more interesting signals.

2. With `t` as above, consider the more interesting waveform generated by

```
y = sin(2*pi*800*(t.*t));
```

This is called a **chirp**. Listen to it. Plot its Fourier series coefficients using the `fourierSeries` function as described above.

This chirp is 8 kHz samples of the continuous-time waveform

$$y(t) = \sin(2\pi 800t^2).$$

The **instantaneous frequency** of this waveform is defined to be the derivative of the argument to the sine function,

$$\omega(t) = \frac{d}{dt} 2\pi 800t^2 = 4\pi 800t.$$

For the given values `t` used to compute samples `y`, what is the range of instantaneous frequencies? Explain how this corresponds with the plot of the Fourier series coefficients, and how it corresponds with what you hear.

3. The Fourier series coefficients computed in part 2 describe the range of instantaneous frequencies of the chirp pretty well, but they do not describe the dynamics very well. Plot the Fourier series coefficients for the waveform given by

```
z = y(8000:-1:1);
```

Listen to this sound. Does it sound the same as `y`? Does its Fourier series plot look the same? Why?

4. The chirp signal has a dynamically varying frequency-domain structure. More precisely, there are certain properties of the signal that change slowly enough that our ears detect them as a change in the frequency structure of the signal rather than as part of the frequency structure (the timbre or tonal content). Recall that our ears do not hear sounds below about 30 Hz. Instead, the human brain hears changes below 30 Hz as variations in the nature of the sound rather than as frequency domain content. The Fourier series methods used above fail to reflect this psychoacoustic phenomenon.

   A simple fix is the **short-time Fourier series**. The chirp signals above have 8000 samples, lasting one second. But since we don't hear variations below 30 Hz as frequency content, it probably makes sense to reanalyze the chirp signal for frequency content 30 times in the one second. This can be done using the following function:[3]

```
function waterfallSpectrogram(s, fs, sizeofspectra, numofspectra)

% WATERFALLSPECTROGRAM - Display a 3-D plot of a spectrogram
% of the signal s.
%
% Arguments:
%   s - The signal.
%   fs - The sampling frequency (in samples per second).
%   sizeofspectra - The number of samples to use to calculate each
%          spectrum.
%   numofspectra - The number of spectra to calculate.

frequencies = [0:fs/sizeofspectra:fs/2];
offset = floor((length(s)-sizeofspectra)/numofspectra);
for i=0:(numofspectra-1)
    start = i*offset;
    [A, phi] = fourierSeries(s((1+start):(start+sizeofspectra)));
    magnitude(:,(i+1)) = A';
end
waterfall(frequencies, 0:(numofspectra-1), magnitude');
xlabel('frequency');
ylabel('time');
zlabel('magnitude');
```

   To invoke this function on the chirp, do

```
t = [0:1/8000:1-1/8000];
y = sin(2*pi*800*(t.*t));
waterfallSpectrogram(y, 8000, 400, 30);
```

   which yields the plot shown in figure C.6. That plot shows 30 distinct sets of Fourier series coefficients, each calculated using 400 of the 8000 available samples. Explain how this plot describes the sound you hear. Create a similar plot for the reverse chirp, signal `z` given in part 3.

[3]This code can be found at http://www.eecs.berkeley.edu/ eal/eecs20/matlab/waterfallSpectrogram.m.

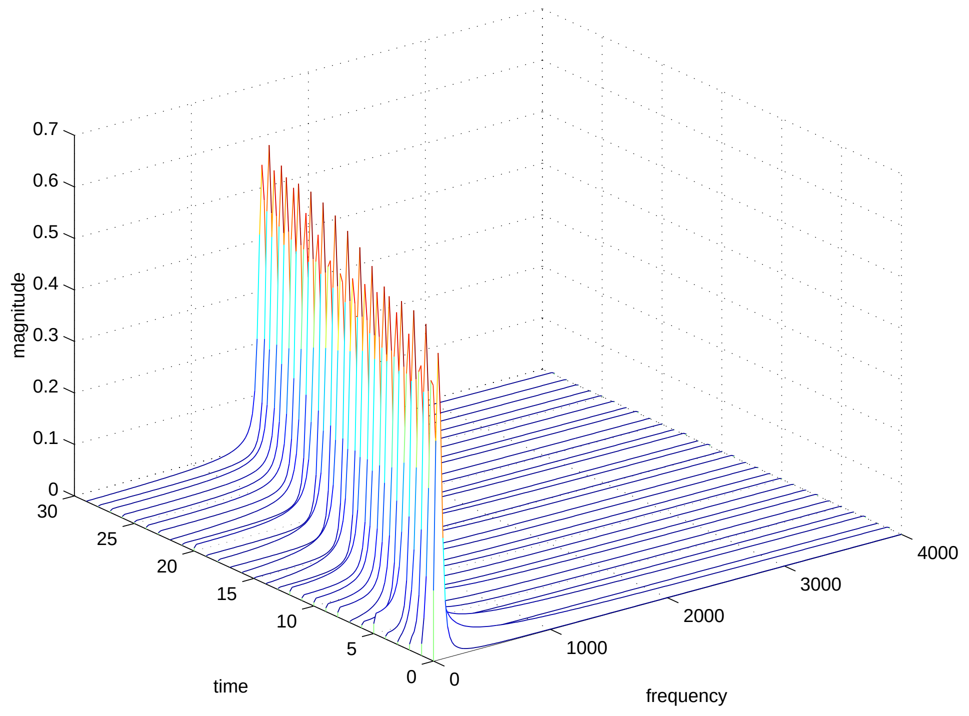


Figure C.6: Time varying discrete Fourier series analysis of a chirp.

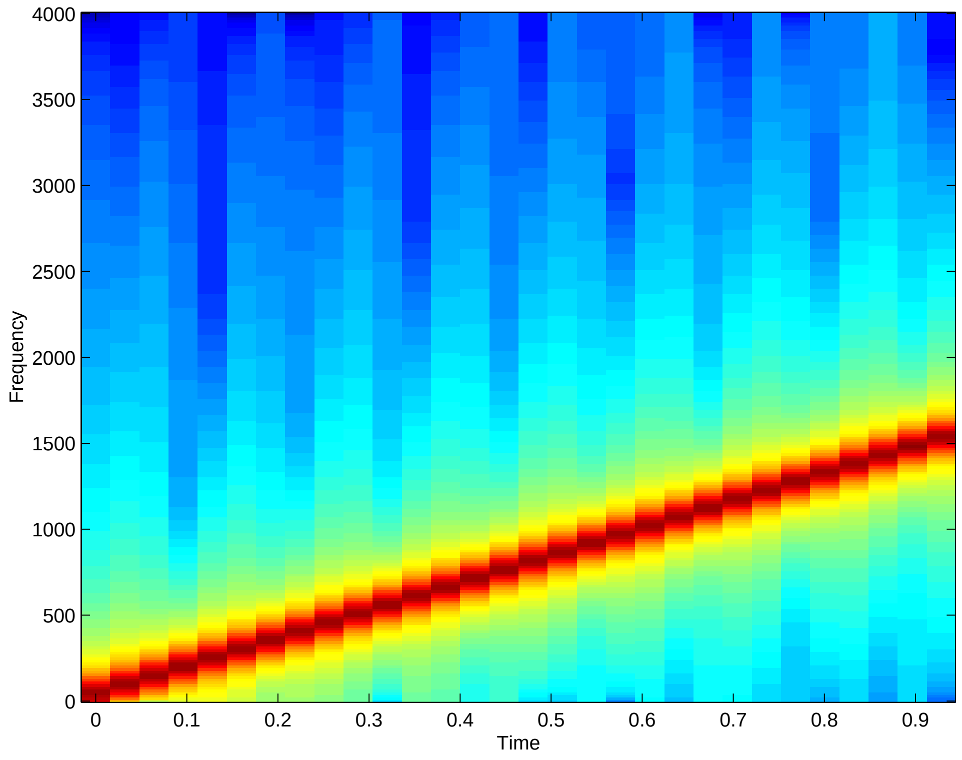


Figure C.7: Spectrogram of the chirp signal.

5. Figure C.6 is reasonably easy to interpret because of the relatively simple structure of the chirp signal. More interesting signals, however, become very hard to view this way. An alternative visualization of the frequency content of such signals is the **spectrogram**. A spectrogram is a plot like that in figure C.6, but looking straight down from above the plot. The height of each point is depicted by a color (or intensity, in a gray-scale image) rather than by height. You can generate a spectrogram of the chirp as follows:

   ```
   specgram(y,512,8000);
   ```

   This results in the image shown in figure C.7. There, the default colormap is used, which is `jet`. A rendition of this colormap is given in figure C.3. You could experiment with different colormaps for rendering this spectrogram by using the `colormap` command. A particularly useful one is `hot`, obtained by the command

   ```
   colormap(hot);
   ```

   Create a similar image for the reverse chirp, `z`, of part 3.

6. A number of audio files are available at

   ```
   http://www.eecs.berkeley.edu/~eal/eecs20/sounds
   ```

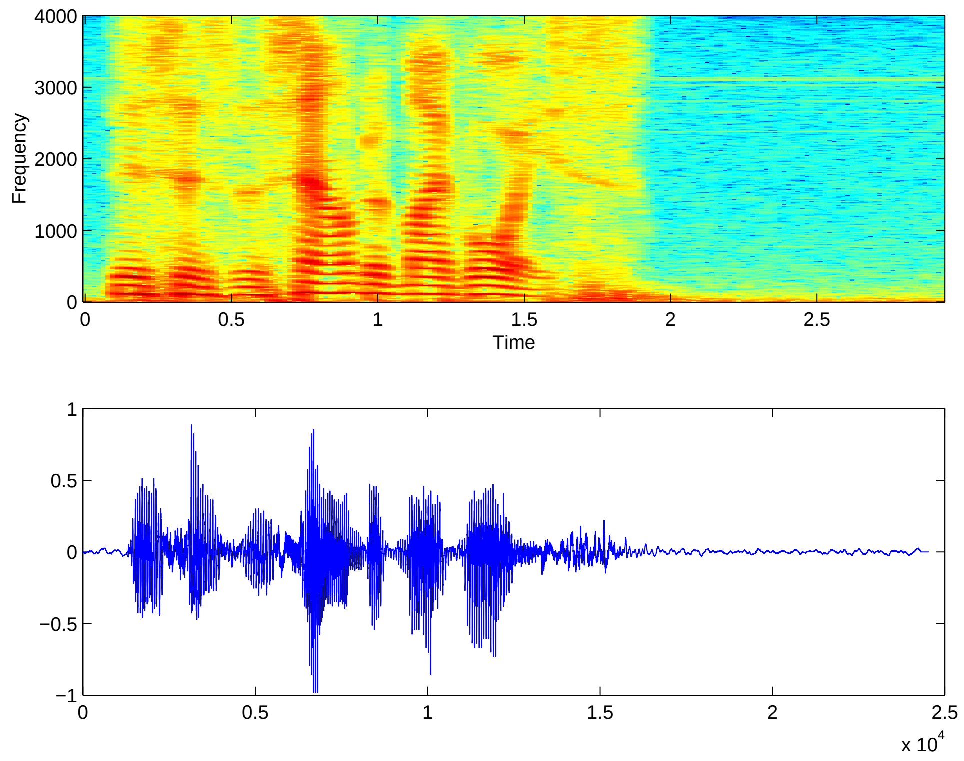


Figure C.8: Spectrogram and plot of a voice segment (one of the authors saying “this is the sound of my voice.”

In Netscape, you can save these to your local computer disk by placing the mouse on the file name, clicking with the right mouse button, and selecting ”Save Link As.” For example, if you save `voice.au` to your current working directory, then in Matlab you can do

```
y = auread('voice.au');
soundsc(y)
subplot(2,1,1); specgram(y,1024,8000,[],900)
subplot(2,1,2); plot(y)
```

to get the result shown in figure C.8. Use this technique to get similar results for other sound files in the same directory. Interpret the results.

C.7.3 Independent section

1. For the chirp signal as above,

```
y = sin(2*pi*800*(t.*t));
```

generate the discrete Fourier series coefficients using `fourierSeries` as explained in section C.7.1. The, write a Matlab function that uses (C.7) to reconstruct the original signal from the coefficients. Your Matlab function should begin as follows:

```
function x = reconstruct(magnitude, phase)
% RECONSTRUCT - Given a vector of magnitudes and a vector
% of phases, construct a signal that has these magnitudes
% and phases as its discrete Fourier series coefficients.
% The arguments are assumed to have odd length, p/2 + 1,
% and the returned vector will have length p.
```

Note that this function will require a large number of computations. If your computer is not up to the task, the construct the Fourier series coefficients for the first 1000 samples instead of all 8000, and reconstruct the original from those coefficients. To check that the reconstruction works, subtract your reconstructed signal from `y` and examine the difference. The difference will not be perfectly zero, but it should be very small compared to the original signal. Plot the difference signal.

2. In the remainder of this lab, we will study **beat signals**, which are combinations of sinusoidal signals with closely spaced frequencies. First, we need to develop some background.

   Use Euler's relation to show that

$$2\cos(\omega_c t)\cos(\omega_\Delta t) = \cos((\omega_c + \omega_\Delta)t) + \cos((\omega_c - \omega_\Delta)t).$$

   for any ω_c, ω_Δ, and t in *Reals*. **Hint**: See box on page 184.

   A consequence of this identity is that if two sinusoidal signals with different frequencies, ω_c and ω_Δ, are multiplied together, the result is the same as if two sinusoids with two other frequencies, $\omega_c + \omega_\Delta$ and $\omega_c - \omega_\Delta$, are added together.

3. Construct the sum of two cosine waves with frequencies of 790 and 810 Hz. Assume the sample rate is 8 kHz, and construct a vector in Matlab with 8000 samples. Listen to it. Describe what you hear. Plot the first 800 samples (1/10 second). Explain how the plot illustrates what you hear. Explain how the identity in part 2 explains the plot.

4. What is the period of the waveform in part 3? What is the fundamental frequency for its Fourier series expansion? Plot its discrete Fourier series coefficients (the magnitude only) using `fourierSeries`. Plot its spectrogram using `specgram`. Choose the parameters of `specgram` so that the warble is clearly visible. Which of these two plots best reflects perception?

Instructor Verification Sheet for C.7

Name: ______________________ Date: ______________________

1. Plot of the DFS coefficients of the sinusoid, with explanation.

   Instructor verification: ______________________

2. Plot of the DFS, plus range of instantaneous frequencies, plus correspondence with the sound.

   Instructor verification: ______________________

3. Plot of the DFS is the same, yet the sound is different. Explanation.

   Instructor verification: ______________________

4. Explain how figure C.6 describes the sound you hear. Plot the reverse chirp.

   Instructor verification: ______________________

5. Create and interpret a spectrogram for one other sound file, at least.

   Instructor verification: ______________________

C.8 Comb filters

The purpose of this lab is to use a kind of filter called a comb filter to deeply explore concepts of impulse response and frequency response.

The lab uses Simulink, like lab C.6. Unlike lab C.6, it will use Simulink for discrete-time processing. Be warned that discrete-time processing is not the best part of Simulink, so some operations will be awkward. Moreover, the blocks in the block libraries that support discrete-time processing are not well organized. It can be difficult to discover how to do something as simple as an N-sample delay or an impulse source. We will identify the blocks you will need.

The lab is self contained, in the sense that no additional documentation for Simulink is needed. As in lab C.6, be warned that the on-line documentation is not as good for Simulink as for Matlab. You will want to follow our instructions closely, or you are likely to discover very puzzling behavior.

C.8.1 Background

To run Simulink, start Matlab and type `simulink` at the command prompt. This will open the Simulink library browser. The library browser is a hierarchical listing of libraries with blocks. The names of the libraries are (usually) suggestive of the contents, although sometimes blocks are found in surprising places, and some of the libraries have meaningless names (such as “Simulink”).

Here, we explain some of the techniques you will need to implement the lab. You may wish to skim these now and return them when you need them.

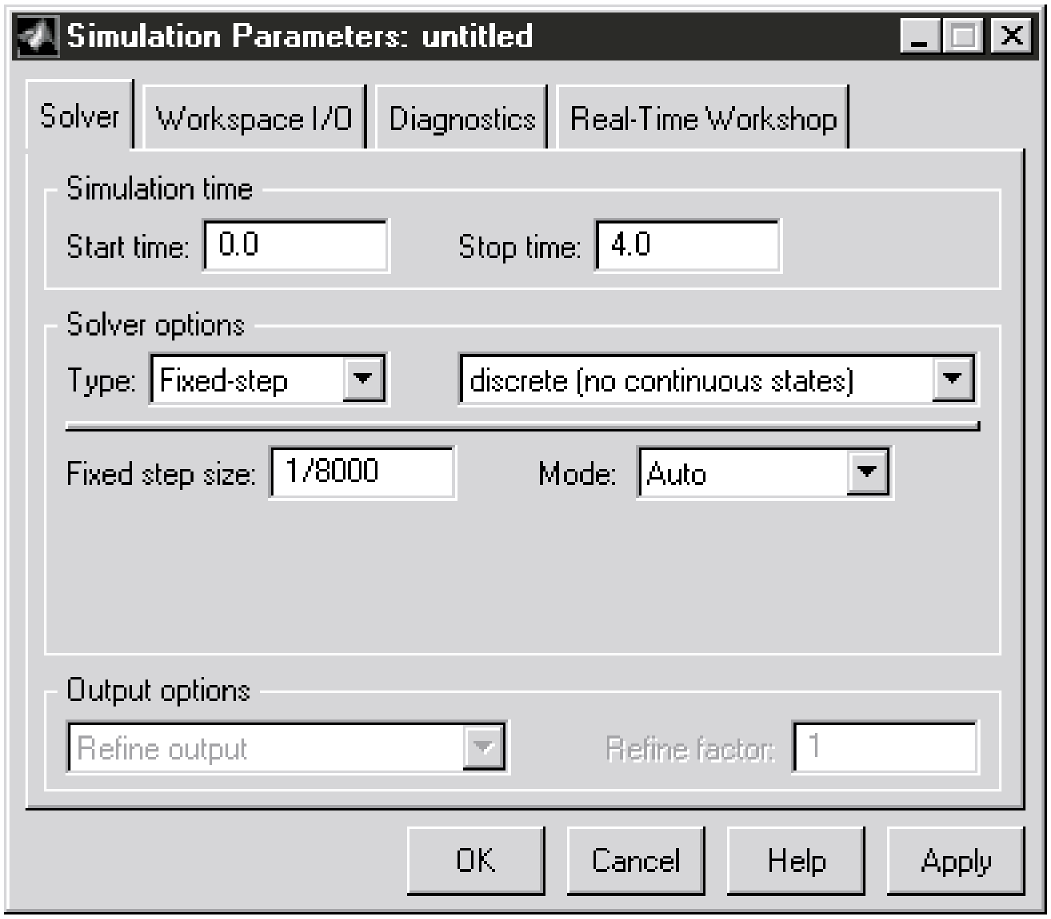
Simulation Parameters

First, since we will be processing audio signals with a sample rate of 8 kHz, you need to force Simulink to execute the model as a discrete-time model with sample rate 8 kHz (recall that Simulink excels at continuous-time modeling). Open a blank model by clicking on the document icon at the upper left of the library browser window. Find the Simulation menu in that window, and select Parameters. Set the parameters so that the window looks like what is shown in figureC.9. Specifically, set the stop time to 4.0 (seconds), the solver options to “Fixed-step” and “discrete (no continuous states),” and the fixed step size to 1/8000.

Reading and Writing Audio Signals

Surprisingly, Simulink is more limited and awkward than Matlab in its ability to read and write audio files. Consequently, the following will seem like more trouble than it is worth. Bear with us. Simulink only supports Microsoft wave files, which typically have the suffix “.wav”. You may obtain a suitable audio file for this lab at

```
http://www.eecs.berkeley.edu/˜eal/eecs20/sounds/voice.wav
```



Simulation time

Start time: 0.0 Stop time: 4.0

Solver options

Type: Fixed-step discrete (no continuous states)

Fixed step size: 1/8000 Mode: Auto

Output options

Refine output Refine factor: 1

Figure C.9: Simulation parameters for discrete-time audio processing in Simulink.

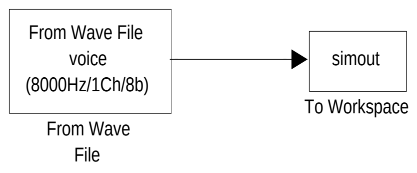


Figure C.10: Test model for Simulink audio.

In Netscape you can go to

```
http://www.eecs.berkeley.edu/˜eal/eecs20/sounds/
```

and then right click on the `voice.wav` filename to bring up a menu, and choose “Save Link As...” to save the file to your local disk. It is best to then, in the Matlab command window, to change the current working directory to the one in which you stored the file using the `cd` command. This will make it easier to use the file.

To make sure we can process audio signals, create the test model shown in figureC.10. To do this, in a new model window with the simulation parameters set as explained in “Simulation Parameters” on page 372, create an instance of the block called `From Wave File`. This block can be found in the library browser under `DSP Blockset` and `DSP Sources`. Set the parameters of that block to

```
File name: voice.wav
Samples per frame: 1
```

The first parameter assumes you have set the current working directory to the directory containing the `voice.wav` file. The second indicates to Simulink that it should produce audio samples one at a time, rather than collecting them into vectors to produce many at once.

Next, find the `To Workspace` block in the Simulink block library, under Sinks. Create an instance of that block in your model. Edit its parameters to change the “Save format” to “Matrix”. You can leave other parameters at their default values.

Connect the blocks as shown in figure C.10.

Assuming the simulation parameters have been set as explained in “Simulation Parameters” on page 372, you can now run the model by invoking the Start command under the Simulation menu. This will result in a new variable called `simout` appearing in the Matlab workspace. In the Matlab command window, do

```
soundsc(simout)
```

to listen to the voice signal.

Note that the `DSP Sinks` library has a block called `To Wave Device`, which in theory will produce audio directly to the audio device. In practice, however, it seems much easier to use the `To`

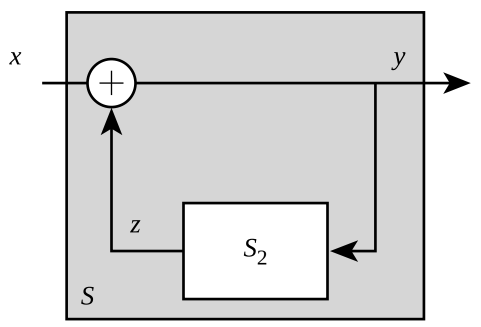


Figure C.11: Comb filter modeled as a feedback system.

`Workspace` block and the `soundsc` command. For one thing, `soundsc` scales the audio signal automatically. It also circumvents difficulties with real-time performance, platform dependence problems, and ideosyncrasies with buffering. However, if you wish to try the `To Wave Device` block, and can figure out how to get it to work, feel free to use it.

C.8.2 In-lab section

1. Consider the equation

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n) + \alpha y(n - N) \tag{C.8}$$

   for some real constant $\alpha < 1$ and integer constant $N > 0$. Assume the sample rate is 8 kHz. The input is $x(n)$ and the output is $y(n)$. The equation describes an LTI system where the output is delayed, scaled, and feb back. Such a system is called a **comb filter**, for reasons that will become apparent in this lab. The filter can be viewed as a feedback structure, as shown in figure C.11, where S_2 is a system with input y and output z. Give a similar equation describing S_2, relating y and z.

2. Implement in Simulink the comb filter from part (a). Provide as input the file `voice.wav` (see page 372). Send the output to the workspace, just like figure C.10, so that you can use `soundsc` to listen to the result. You will probably need the `Gain` and `Sum` blocks, which you can find in the Simlink, Math library. The delay in the feedback path can be implemented by the `Integer Delay` block, which you can find in the DSP Blockset, General DSP, Signal Operations library.

   Experiment with the values of N. Try $N = 2000$ and $N = 50$ and describe qualitatively the difference. With $N = 50$, the effect is called a sewer pipe effect. Why? Can you relate the physics of sound in a sewer pipe with our mathematical model? **Hint**: The speed of sound in air is approximately

$$331.5 + 0.6T\text{meters/second}$$

   where T is the temperature in degress celcius. Thus, at 20 degrees, sound travels at about 343.7 meters/second. A delay of $N = 50$ samples at an 8 kHz sample rate is equal to the time it takes sound to travel roughly 2 meters, twice the diameter of a 1 meter sewer pipe.

Experiment with the value of α. What happens when $\alpha = 0$? What happens when $\alpha = 1$? When $\alpha > 1$? You may wish to plot the output in addition to listening to it.

3. Modify your Simulink model so that its output is the first one second (the first 8001 samples) of the impulse response of the system defined by (C.8), with $\alpha = 0.99$ and $N = 40$.

   The simplest approach is to provide an impulse as an input. To do that, use the `Discrete Pulse Generator` block, found in the Simulink, Sources. This block can be (sort of) configured to generate a Kronecker delta function. Set its amplitude to 1, its period to something longer than the total number of samples (i.e. larger than 8001), its pulse width to 1, its phase delay to 0, and its sample time to 1/8000.

   You will also want to change the simulation parameters to execute your system for 1 second instead of 4.

   Listen to the impulse response. Plot it. Can you identify the tone that you hear? Is it a musical note? **Hint**: Over short intervals, a small fraction of a second, the impulse response is roughly periodic. What is its period?

4. In the next lab you will modify the comb filter to generate excellent musical sounds resembling plucked strings, such as guitars. As a first step towards that goal, we can make a much less mechanical sound than the impulse response by initializing the delay with random data. Modify your Simulink model so that the comb filter has no input, and instead of an input, the `Integer Delay` block is given random initial conditions. Use $\alpha = 0.99$ and $N = 40$, and change the parameters of the `Integer Delay` block so that its initial conditions are given by

   ```
   randn(1,40)
   ```

   The Matlab `randn` function returns a vector of random numbers (try `help randn` in the Matlab command window).

   Listen to the result. Compare it to the sound of the impulse response. It should be richer, and less mechanical, but should have the same tone. It is also louder (even though `soundsc` scales the sound).

C.8.3 Independent section

The comb filter is an LTI system. Figure C.11 is a special case of the feedback system considered in section 7.5.2, which is shown there to be LTI. Thus, if the input is

$$x(n) = e^{j\omega n}$$

then the output is

$$y(n) = H(\omega)e^{j\omega n}$$

where $H\colon \textit{Reals} \to \textit{Comps}$ is the frequency response. Find the frequency response of the comb filter. Plot the magnitude of the frequency response over the range 0 to 4 kHz using Matlab. Why is it called a comb filter? Explain the connection between the tone that you hear and the frequency response.

Instructor Verification Sheet for C.8

Name: ____________________ Date: ____________________

1. Found an equation for S_2, relating y and z.

   Instructor verification: ____________________

2. Constructed Simulink model and obtained both sewer pipe effect and echo effect.

   Instructor verification: ____________________

3. Constructed the impulse response and identified the tone.

   Instructor verification: ____________________

4. Created sound with random values in the feedback delay.

   Instructor verification: ____________________

C.9 Plucked string instrument

The purpose of this lab is to experiment with models of a plucked string instrument, using it to deeply explore concepts of impulse response, frequency response, and spectrograms. The methods discussed in this lab were invented by Karplus and Strong, and first reported in

> K. Karplus and A. Strong, "Digital Sythesis of Plucked-String and Drum Timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43-55, Summer 1983.

The lab uses Simulink, like lab C.8. It assumes you have understood that lab and the techniques it uses in detail.

C.9.1 Background

In the previous lab, you constructed in Simulink the feedback system shown in figureC.11, where S_2 was an N sample delay. In this lab, you will enhance the model by replacing S_2 with slightly more complicated filters. These filters will consist of the same N sample delay in cascade with two other filters, a **lowpass filter** and an **allpass filter**. The objective will be to get truly high-quality plucked string sounds.

Delays

Recall from example 7.2 that the N sample delay system has frequency response

$$H(\omega) = e^{-i\omega N}.$$

This same frequency response was obtained in example8.10 by calculating the DTFT of the impulse response. Note that the magnitude response is particularly simple,

$$|H(\omega)| = 1.$$

Recall that this is an **allpass filter**.

The phase response is

$$\angle H(\omega) = -\omega N.$$

The phase response is a linear function of frequency, ω, with slope $-N$. A filter with such a phase response is said to have **linear phase**. A delay is particularly simple form of a linear phase filter. Notice that the amount of delay is the negative of the derivative of the phase response,

$$\frac{d\angle H(\omega)}{d\omega} = -N.$$

This fact will be useful when we consider more complicated filters than this simple delay.

Allpass Filters

We will need a slightly more complicated allpass filter than the N sample delay. Consider a filter given by the following difference equation,

$$\forall\, n \in \textit{Ints}, \quad y(n) + ay(n-1) = ax(n) + x(n-1) \tag{C.9}$$

for some constant $0 < a \leq 1$. This defines an LTI system, so if the input is $x(n) = e^{i\omega n}$, then the output is $H(\omega)e^{i\omega n}$, where H is the frequency response. We can determine the frequency response using this fact by plugging this input and output into (C.9),

$$H(\omega)e^{i\omega n} + aH(\omega)e^{i\omega(n-1)} = ae^{i\omega n} + e^{i\omega(n-1)}.$$

This can be rewritten as

$$H(\omega)e^{i\omega n}(1 + ae^{-i\omega}) = e^{i\omega n}(a + e^{-i\omega}).$$

Eliminating $e^{i\omega n}$ on both sides we get

$$H(\omega)(1 + ae^{-i\omega}) = a + e^{-i\omega}.$$

Solving for $H(\omega)$ we get

$$H(\omega) = \frac{a + e^{-i\omega}}{1 + ae^{-i\omega}}. \tag{C.10}$$

We could immediately proceed to plotting the magnitude and phase response using Matlab, but instead, we will first manipulate this further to get some insight. Being slightly tricky, we will multiply top and bottom by $e^{i\omega/2}$ to get

$$H(\omega) = \frac{ae^{i\omega/2} + e^{-i\omega/2}}{e^{i\omega/2} + ae^{-i\omega/2}}.$$

Now notice that the top and bottom are complex conjugates of one another. I.e., let

$$b(\omega) = ae^{i\omega/2} + e^{-i\omega/2} \tag{C.11}$$

and note that

$$H(\omega) = \frac{b(\omega)}{b^*(\omega)}.$$

Since the numerator and denominator have the same magnitude,[4]

$$|H(\omega)| = 1.$$

The filter is allpass!

The phase response, however, is more complicated. Note that

$$\angle H(\omega) = \angle b(\omega) - \angle b^*(\omega).$$

[4] For any two complex numbers z and w, note that $|z/w| = |z|/|w|$ and $\angle(z/w) = \angle(z) - \angle(w)$.

But since for any complex number z, $\angle(z^*) = -\angle(z)$,

$$\angle H(\omega) = 2\angle b(\omega).$$

Thus, to find the phase response, we simply need to find $\angle b(\omega)$. Plugging Euler's relation into (C.11) we get

$$b(\omega) = (a+1)\cos(\omega/2) + i(a-1)\sin(\omega/2).$$

Since the angle of a complex number z is $\tan^{-1}(\mathit{Im}\{z\}/\mathit{Re}\{z\})$,

$$\angle H(\omega) = 2\tan^{-1}\left(\frac{(a-1)\sin(\omega/2)}{(a+1)\cos(\omega/2)}\right).$$

Since $\tan(w) = \sin(w)/\cos(w)$,

$$\angle H(\omega) = 2\tan^{-1}\left(\frac{a-1}{a+1}\tan(\omega/2)\right).$$

This form yields insight for small ω. In particular, when ω is small (compared to π),

$$\tan(\omega/2) \approx \omega/2,$$

so

$$\angle H(\omega) \approx 2\tan^{-1}\left(\frac{a-1}{a+1}\omega/2\right).$$

Since $0 < a \leq 1$, the argument to the arctangent is small if ω is small, so for low frequencies,

$$\angle H(\omega) \approx \frac{a-1}{a+1}\omega = -d\omega.$$

where d is defined by

$$d = -\frac{a-1}{a+1}. \tag{C.12}$$

Thus, at low frequencies, this allpass filter has linear phase with slope $-d$. At low frequencies, therefore, it is an allpass with linear phase, which means that behave exactly like a delay! However, unlike the N sample delay, the amount of delay is d, which depending on a can be any real number between 0 and 1. Thus, this allpass filter gives us a way to get fractional sample delays in a discrete time system, at least at low frequencies.

C.9.2 In-lab section

1. The lowpass filter we will use is a simple, length two moving average. If the input is x and the output is y, then the filter is given by the difference equation,

$$\forall\, n \in \mathit{Ints}, \quad y(n) = 0.5(x(n) + x(n-1)). \tag{C.13}$$

   Find an expression for the frequency response of the lowpass filter given by (C.13). Use Matlab to plot the magnitude and phase response over the frequency range 0 to π radians/sample. Is this a linear phase filter? If so, what is its delay?

2. In part 4 of the previous lab, you initialized a comb filter with random noise and produced a sound that reasonably well approximates a plucked string instrument, such as a guitar. We can improve the sound.

   Real instrument sounds have more dynamics in their frequency structure. That is, the spectrum of the sound within the first few milliseconds of plucking the string is different from the spectrum a second or so later. Physically, this is because the high frequency vibrations of the string die out more rapidly than the low frequency vibrations.

   We can approximate this effect by modifying the comb filter by inserting the lowpass filter given by (C.13) into the feedback loop. This can be accomplished by realizing the following difference equation:

$$\forall\, n \in \textit{Ints}, \quad y(n) = x(n) + 0.5\alpha(y(n-N) + y(n-N-1)).$$

   Modify your Simulink model you constructed in part 4 of the previous lab so that it uses a lowpass filter in the feedback loop, implementing this difference equation. Listen to the resulting sound, and compare it against the sound from the previous lab. Use $\alpha = 0.99$ and $N = 40$, as before. Can you hear the improvement?

3. In the last lab, you found that the tone of the sound generated by the comb filter had frequency $8000/N$, where N was the delay in the feedback loop, and 8000 was the sampling frequency. You used $N = 40$ to get a fundamental frequency of 200 Hz. Now, you have added an additional lowpass filter, which introduces additional delay in the feedback loop. You have determined that additional delay in part 1 above. What is the fundamental frequency now?

   The comb filter delay can only delay by an integer number of samples. The lowpass filter introduces a fixed delay. Consequently, there are only certain fundamental frequencies that are possible. In particular, assuming the sample rate is 8 kHz, is it possible to achieve a fundamental frequency of 440 Hz? This would be essential to have a reasonable guitar model, since we would certainly want to be able to play the note A-440 on the guitar. Determine the closest achievable frequency to 440 Hz. Is it close enough? In the independent section of this lab, you will show how to achieve a fundamental frequency very close to 440 Hz.

C.9.3 Independent section

1. Show analytically that the lowpass filter given by (C.13) has linear phase over the range of frequencies 0 to π radians/sample, and determine the slope. Verify that this agrees with the plot you constructed in the In-Lab section.

2. In part 2 of the in-lab section, you combined an N-sample delay with a lowpass filter in the feedback path of a comb filter. Calculate the frequency response of this version of the comb filter, and plot its magnitude using Matlab over the frequency range 0 to π. Compare it to the frequency response you calculated for the original comb filter in the previous lab. Find the fundamental frequency of the musical note from this plot and compare it to the answer that you gave in part 3 of the in-lab portion. **Hint**: The spectral peaks are very sharp, so you will need to calculate the magnitude frequency at many points in the range 0 to π to be sure to hit the peaks. We recommend calculating at least 2000 points.

3. The reason that the comb filter with a lowpass filter in the feedback loop yields a much better plucked string sound than the comb filter by itself is that it more accurately models the physical phenomenon that higher frequency vibrations on the string die out faster than lower frequency vibrations. Plot the spectrogram using `specgram` of the generated sound to demonstrate this phenomenon, and explain how your spectrogram demonstrates it.

4. Verify that the frequency response (C.10) of the allpass filter has constant magnitude and linear phase for low frequencies by plotting it using Matlab. Plot it for the following values of delay: $d = 0.1, 0.4, 0.7, and 1.0$. Plot it over the range of frequencies 0 to π radians/sample. Discuss how your plots support the conclusions about this filter. **Hint**: Use (C.12) to find a given d.

5. You determined in part 3 of the in-lab section that you could not get very close to A-440 with a comb filter with a lowpass filter in the feedback loop. The allpass filter given by (C.10), however, can be used to achieve delays that are a fraction of a sample period. Implement the allpass filter, modifying your Karplus-Strong plucked string model by putting it in the feedback loop. Set the parameters of the allpass filter and N to get an A-440. Show your Simulink diagram, and give the parameters of all the blocks.

Instructor Verification Sheet for C.9

Name: ______________________ Date: ______________________

1. Magnitude and phase of lowpass filter. Linear phase? Delay?

   Instructor verification: ______________________

2. Improved plucked string sound.

   Instructor verification: ______________________

3. Fundamental frequencies that are possible.

   Instructor verification: ______________________

C.10 Modulation and demodulation

The purpose of this lab is to learn to use frequency domain concepts in practical applications. The application selected here is **amplitude modulation** (**AM**), a widely used technique in communication systems, including of course AM radio, but also almost all digital communication systems, including digital cellular telephones, voiceband data modems, and wireless networking devices. A secondary purpose of this lab is to gain a working knowledge of the **fast Fourier transform** (**FFT**) algorithm, and an introductory working knowledge of **filter design**. Note that this lab requires the Signal Processing Toolbox of Matlab for filter design.

C.10.1 Background

Amplitude Modulation

The problem addressed by AM modulation is that we wish to convey a signal from one point in physical space to another through some **channel**. The channel has certain constraints, and in particular, can typically only pass frequencies within a certain range. An AM radio station, for example, might be constrained by government regulators to broadcast only radio signals in the range of 720 kHz to 760 kHz, a **bandwidth** of 40 kHz.

The problem, of course, is that the radio station has no interest in broadcasting signals that only contain frequencies in the range 720 kHz to 760 kHz. They are more likely to want to transmit a voice signal, for example, which contains frequencies in the range of about 50 Hz to about 10 kHz. AM modulation deals with this mismatch by **modulating** the voice signal so that it is shifted to the appropriate frequency range. A radio receiver, of course, must **demodulate** the signal, since 720 kHz is well above the hearing range of humans.

In this lab, we present a somewhat artificial scenario in order to maximize the experience. We will keep all frequencies that we work with within the audio range so that you can listen to any signal. Therefore, we will not modulate a signal up to 720 kHz (you would not be able to hear the result). Instead, we present the following scenario:

> Assume we have a signal that contains frequencies in the range of about 100 to 300 Hz, and we have a channel that can pass frequencies from 700 to 1300 Hz.[5] Our task will be to modulate the first signal so that it lies entirely within the channel **passband**, and then to demodulate to recover the original signal.

AM modulation is studied in detail in exercise 5 of chapter 9. In that problem, you showed that if

$$y(t) = x(t)\cos(\omega_c t),$$

then the CTFT is

$$Y(\omega) = X(\omega - \omega_c)/2 + X(\omega + \omega_c)/2.$$

[5]Since Fourier transforms of real signals are symmetric, the signal also contains frequencies in the range -100 to -300 Hz, and the channel passes frequencies in the range -700 to -1300 Hz.

In this lab, we will get a similar result experimentally, but working entirely with discrete-time signals, and staying entirely within the audio range so that we can hear (and not just plot) the results.

The FFT Algorithm

In order to understand AM modulation, we need to be able to calculate and examine Fourier transforms. We will do this numerically in this lab, unlike exercise 5 of chapter 9, where it is done analytically.

In lab C.7, we used a supplied function called `fourierSeries` to calculate the Fourier series coefficients A_k and ϕ_k for signals. In this lab, we will use the built-in function `fft`, which is used, in fact, by the `fourierSeries` function. Learning to use the FFT is extremely valuable; it is widely used all analytical fields that involve time series analysis, including not just all of engineering, but also the natural sciences and social sciences. The FFT is also widely abused by practitioners who do not really understand what it does.

The FFT algorithm operates most efficiently on finite signals whose lengths are a power of 2. Thus, in this lab, we will work with what might seem like a peculiar signal length, 8192. This is 2^{13}. At an 8 kHz sample rate, it corresponds to slightly more than one second of sound.

Recall that a periodic discrete-time signal with period p has a discrete-time Fourier series expansion

$$x(n) = A_0 + \sum_{k=1}^{(p-1)/2} A_k \cos(k\omega_0 n + \phi_k) \tag{C.14}$$

for p odd and

$$x(n) = A_0 + \sum_{k=1}^{p/2} A_k \cos(k\omega_0 n + \phi_k) \tag{C.15}$$

for p even, where $\omega_0 = 2\pi/p$, the fundamental frequency in cycles per sample. Recall further that we can alternatively write the Fourier series expansion in terms of complex exponentials,

$$x(n) = \sum_{k=0}^{p} X_k e^{ik\omega_0 n}. \tag{C.16}$$

Note that this sum covers one cycle of the periodic signal. If what we have is a finite signal instead of a periodic signal, then we can interpret the finite signal as being one cycle of a periodic signal.

In chapter 9, we describe four Fourier transforms. The only one of these that is computable on a computer is the **DFT**, or **discrete Fourier transform**. For a periodic discrete-time signal x with period p, we have the **inverse DFT**, which takes us from the frequency domain to the time domain,

$$\forall\, n \in \textit{Ints}, \quad x(n) = \frac{1}{p}\sum_{k=0}^{p-1} X'_k e^{ik\omega_0 n}, \tag{C.17}$$

and the **DFT**, which takes us from the time domain to the frequency domain,

$$\forall\, k \in \textit{Ints}, \quad X'_k = \sum_{m=0}^{p-1} x(m) e^{-imk\omega_0}. \tag{C.18}$$

Comparing (C.17) and (C.16), we see that the DFT yields coefficients that are just scaled versions of the Fourier series coefficients. This scaling is conventional.

In lab C.7 we calculated A_k and ϕ_k. In this lab, we will calculate X'_k. This can be done using (C.18). The FFT algorithm is simply a computationally efficient algorithm for calculating (C.18).

In Matlab, you will collect 8192 samples of a signal into a vector and then invoke the `fft` function. Invoke `help fft` to verify that this function is the right one to use. If your 8192 samples are in a vector `x`, then `fft(x)` will return a vector with 8192 complex number, representing $X_0, ..., X_{8191}$.

From (C.18) it is easy to verify that $X_k = X_{k+p}$ for all integers k (see part 1 of the in-lab section below). Thus, the DFT X is a periodic, discrete function with period p. If you have the vector `fft(x)`, representing $X_0, ..., X_{8191}$, you know all X_k. For example,

$$X_{-1} = X_{-1+8192} = X_{8191}$$

From C.17, you can see that each X_k scales a complex exponential component at frequency $k\omega_0 = k2\pi/p$, which has units of samples per second. In order to interpret the DFT coefficients X_k, you will probably want to convert the frequency units to Hertz. If the sampling frequency is f_s samples per second, then you can do the conversion as follows (see box on page151):

$$\frac{(k2\pi/p)[\text{radians/sample}]f_s[\text{samples/second}]}{2\pi[\text{radians/cycle}]} = kf_s/p[\text{cycles/second}] \tag{C.19}$$

Thus, each X_k gives the DFT value at frequency kf_s/p Hz. For our choices of numbers, $f_s = 8000$ and $p = 8192$, so X_k gives the DFT value at frequency $k \times 0.9766$ Hz.

Filtering in Matlab

The `filter` function can compute the output of an LTI system given by a difference equation of the form

$$a_1 y(n) = b_1 x(n) + b_2 x(n-1) + \cdots + b_N x(n-N+1) - a_2 y(n-1) - ... - a_M y(n-M+1). \tag{C.20}$$

To find the output y, first collect the (finite) signal x into a vector. Then collect the coefficients $a_1, \cdots, a_N$ into a vector `A` of length N, and the coefficients $b_1, \cdots, b_M$ into a vector `B` of length M. Then just do

```
y = filter(B, A, x);
```

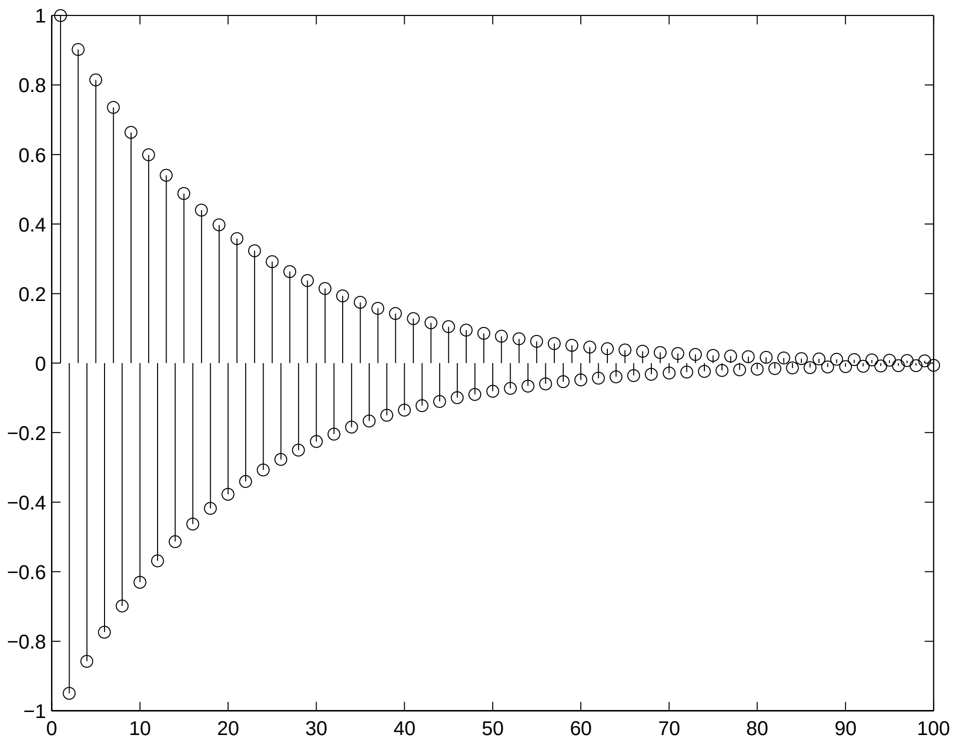


Figure C.12: Impulse response of a simple filter.

Example 3.1: Consider the difference equation

$$y(n) = x(n) - 0.95y(n-1).$$

We can find and plot the first 100 samples of the impulse response by letting the vector x be an impulse and using `filter` to calculate the output:

```
x = [1, zeros(1,99)];
y = filter([1], [1, 0.95], x);
stem(y);
```

which yields the plot shown in C.12. The natural question that arises next is how to decide on the values of B and A. This is addressed next.

Filter Design in Matlab

The signal processing toolbox of Matlab provides a set of useful functions that return filter coefficients A and B given a specification of a desired frequency response. For example, suppose we have a signal sampled at 8 kHz and we wish to design a filter that passes all frequency components below 1 kHz and removes all frequency components above that. The following Matlab command designs a filter that approximates this specification:

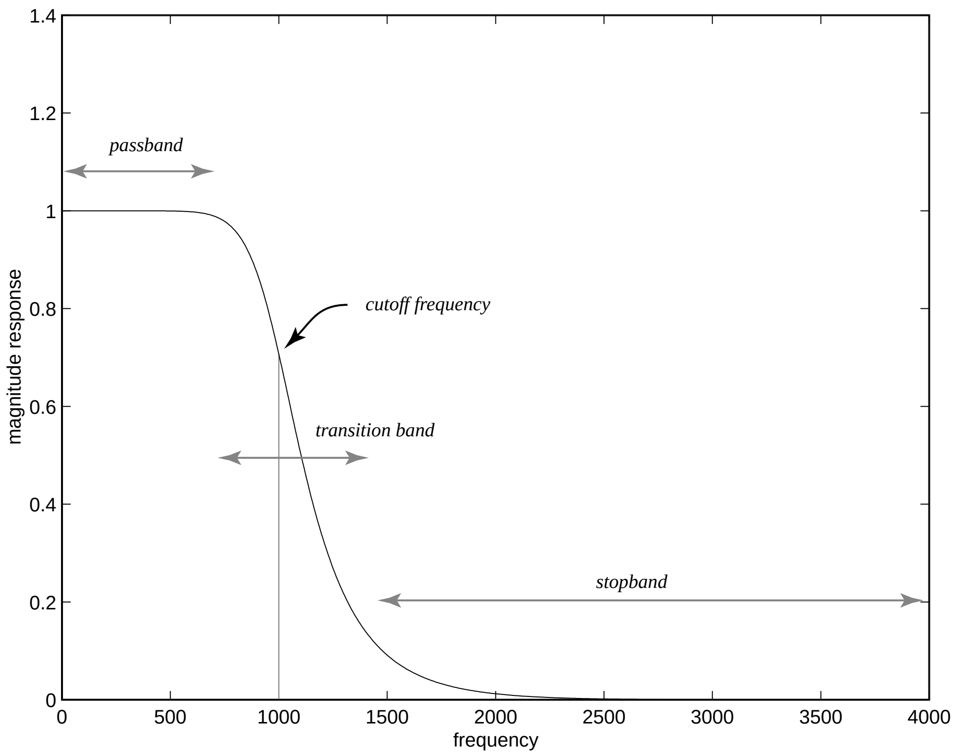


Figure C.13: Frequency response of a 10-th order Butterworth lowpass filter.

```
[B, A] = butter(10, 0.25);
```

The first argument, called the **filter order**, gives M and N in (C.20) (a constraint of the `butter` function is that $M = N$). The second argument gives the **cutoff frequency** of the filter as a fraction of half the sampling frequency. Thus, in the above, the cutoff frequency is $0.25 * (8000/2) = 1000$ Hertz. The cutoff frequency is by definition the point at which the magnitude response is $1/\sqrt{2}$. The returned arrays `B` and `A` are the arguments to supply to `filter` to calculate the filter output.

The frequency response of this filter can be plotted using the `freqz` function as follows:

```
[H,W] = freqz(B,A,512);
plot(W*(4000/pi), abs(H));
xlabel('frequency');
ylabel('magnitude response');
```

which yields the plot shown in figure C.13. (The argument 512 specifies how many samples of the continuous frequency response we wish to calculate.)

This frequency response bears further study. Notice that the response transitions gradually from the passband to the stopband. An abrupt transition is not implementable. The width of the transition

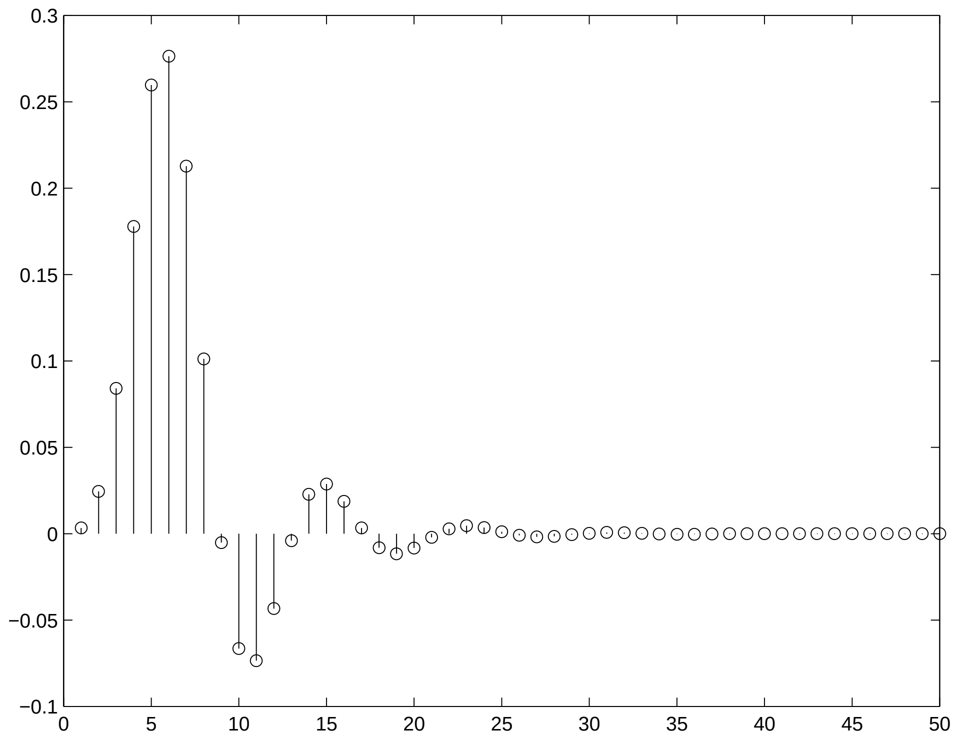


Figure C.14: Impulse response of a 10-th order Butterworth lowpass filter.

band can be reduced by using an order higher than 10 in the `butter` function, or by designing more sophisticated filters using the `cheby1`, `cheby2`, or `ellip` functions in the signal processing toolbox. The Butterworth filter returned by `butter`, however, will be adequate for our purposes in this lab.

Using a higher order to get a narrower transition band can be an expensive proposition. The function `filter` works by implementing the difference equation (C.20). As M and N get larger, each output sample $y(n)$ requires more computation.

The first 50 samples of the impulse response of the filter can be plotted using the following Matlab code:

```
x = [1, zeros(1,49)];
y = filter(B, A, x);
stem(y);
```

This yields the plot shown in figure C.14.

C.10.2 In-lab section

1. Use (C.18) to show that $X'_k = X'_{k+p}$ for all integers k. Also, show that the DFT is conjugate symmetric, i.e. $X'_k = (X'_{-k})^*$ for all integers k, assuming $x(n)$ is real for all integers n.

2. In part 2 of the in-lab portion of lab C.7, we studied a **chirp** signal. We will use a similar signal here, although it will vary over a narrower range of frequencies. Construct the signal `x` given by:

```
t = [0:1/8000:8191/8000];
x = sin(2*pi*100*t + 2*pi*100*(t.*t));
```

   This is a chirp that varies from about 100 Hz to about 300 Hz. Listen to it. Calculate its DFT using the `fft` function, and plot the magnitude of the DFT. Construct your plot in such a way that your horizontal axis is labeled in Hertz, not in the index k of X_k. The horizontal axis should vary in frequency from 0 to 8000 Hz.

3. Your plot from part 2 should show frequency components in the range 100 Hz to 300 Hz, but in addition, it shows frequency components in the range 7700 to 7900. These extra components are the potentially the most confusing aspect of the DFT, but in fact, they are completely predictable from the mathematical properties of the DFT.

   Recall that the DFT of a real signal is conjugate symmetric. Thus,

$$|X'_k| = |X'_{-k}|.$$

   Thus, if there are frequency components in the range 100 to 300 Hz, then there should also be frequency components with the same magnitude in the range -100 to -300 Hz. These do not show up on your plot simply because you have not plotted the frequency components at negative frequencies.

   Recall that the DFT is periodic with period p. That is, $X_k = X_{k+p}$ for all integers k. Recall from (C.19) that the $k-th$ DFT coefficient represents a frequency component at kf_s/p Hertz, where f_s is the sampling frequency, 8000 Hertz. Thus, a frequency component at some frequency f must be identical to a frequency component at $f + f_s$. Therefore, the components in the range -100 to -300 Hertz must be identical to the components in the range 7700 to 7900 Hertz! The image we are seeing in that latter range is a consequence of the conjugate symmetry and periodicity of the DFT!

   Since the DFT is periodic with period 8000 Hertz (when using units of Hertz), it possibly makes more sense to plot its values in the range -4000 to 4000 Hertz, rather than 0 to 8000 Hertz. This way, we can see the symmetry. Since the DFT gives the weights of complex exponential components, the symmetry is intuitive, because it takes two complex exponentials with frequencies that are negatives of one another to yield a real-valued sinusoid.

   Manipulate the result of the `fft` function to yield a plot of the DFT of the chirp where the horizontal axis is -4000 to 4000 Hertz. It is not essential to include both endpoints, at -4000 and at 4000 Hertz, since they are constrained to be identical anyway by periodicity.

C.10.3 Independent section

1. For the chirp signal as above, multiply it by a sine wave with frequency 1 kHz, and plot the magnitude of the DFT of the result over the frequency range -4000 to 4000 Hz. Verify that the resulting signal will get through the channel described in the scenario on page384. Listen to the modulated chirp. Does what you hear correspond with what you see in the DFT plot?

2. The modulated chirp signal constructed in the previous part can be demodulated by multiplying it again by a sine wave with frequency 1 kHz. Form that product, and plot the magnitude of the DFT of the result over the frequency range -4000 to 4000 Hz. How does the result differ from the original chip? Listen to the resulting signal. Would this be an acceptable demodulation by itself?

3. Use the `butter` function to design a filter that will process the result of the previous part so that it more closely resembles the original signal. You should be able to get very close with a modest filter order (say, 5). Filter the result of the previous part, listen to the result, and plot the magnitude of its DFT in the frequency range -4000 to 4000 Hz.

   The modulation and demodulation method you have just implemented is similar to what is used many communication systems. A number of practical problems have to be overcome in practice, however. For example, the receiver usually does not know the exact frequency and phase of the carrier signal, and hence it has to somehow infer this frequency and phase from the signal itself. One technique is to simply include the carrier in the modulated signal by adding it in. Instead of transmitting

   $$y(t) = x(t)\cos(\omega_c t),$$

   we can transmit

   $$y(t) = (1 + x(t))\cos(\omega_c t),$$

   in which case, the transmitted signal will contain the carrier itself. This can be used for demodulation. Another technique is to construct a **phase locked loop**, a clever device that extracts the carrier from the transmitted signal without it being explicitly present. This method is used in digital transmission schemes. The details, however, must be left to a more advanced text.

   In the scheme we have just implemented, the amplitude of a carrier wave is modulated to carry a signal. It turns out that we can also vary the phase of the carrier to carry additional information. Such **AM-PM** methods are used extensively in digital transmission. These methods make more efficient use of precious radio bandwidth than AM alone.

Instructor Verification Sheet for C.10

Name: ______________________ Date: ______________________

1. Verify periodicity and conjugate symmetry of the DFT.

   Instructor verification: ______________________

2. Plot of the magnitude of the DFT, correctly labeled, from 0 to 8000 Hz.

   Instructor verification: ______________________

3. Plot of the magnitude of the DFT, correctly labeled, from -4000 to 4000 Hz.

   Instructor verification: ______________________

C.11 Sampling and aliasing

The purpose of this lab is to study the relationship between discrete-time and continuous-time signals by examining sampling and aliasing. Of course, a computer cannot directly deal with continuous-time signals. So instead, we will construct discrete-time signals that are defined as samples of continuous-time signals, and then operate entirely on them, downsampling them to get new signals with lower sample rates, and upsampling them to get signals with higher sample rates.

Note that this lab has no independent part. Therefore, no lab writeup needs to be turned in. The instructor verification sheet is sufficient.

C.11.1 In-lab section

1. Recall from lab C.7 that a **chirp** signal given by

   $$x(t) = \sin(2\pi f t^2)$$

   has **instantaneous frequency**

   $$\frac{d}{dt} 2\pi f t^2 = 4\pi f t$$

   in radians per second, or

   $$2ft$$

   in Hertz. A sampled signal $y = \mathit{Sampler}_T(x)$ with sampling interval T will be

   $$y(n) = \sin(2\pi f (nT)^2).$$

   Create in Matlab a chirp sampled at 8 kHz that lasts 10 seconds and sweeps from frequency 0 to 12 kHz. Listen to the chirp. Explain the aliasing artifacts that you hear.

2. For the remainder of this lab, we will work with a particular chirp signal that has a convenient Fourier transform for visualizing and hearing aliasing effects. For efficient computation using the FFT algorithm, we will work with $8192 = 2^{13}$ samples, giving slightly more than 1 second of sound at an 8 kHz sample rate. You may wish to review lab C.10, which explains how to create well-labeled plots of the DFT of a finite signal using the FFT algorithm.

   The chirp signal we wish to create is given by

   $$\forall\, t \in [0, D], \quad x(t) = f(t) \sin(2\pi f t^2)$$

   where D is the duration of the signal and $f(t)$ is a time-varying amplitude given by

   $$f(t) = 1 - |t - D/2|/(D/2).$$

   This chirp starts with amplitude 0, rising linearly to peak at the midpoint of the duration, and then falls again back to zero, as shown in figure C.15. Thus, it has a triangular envelope.

   Generate such a chirp that lasts for 8192 samples at an 8 kHz sample rate and sweeps from a frequency of zero to 2500 Hz. Listen to the resulting sound. Are there any aliasing artifacts? Why or why not?

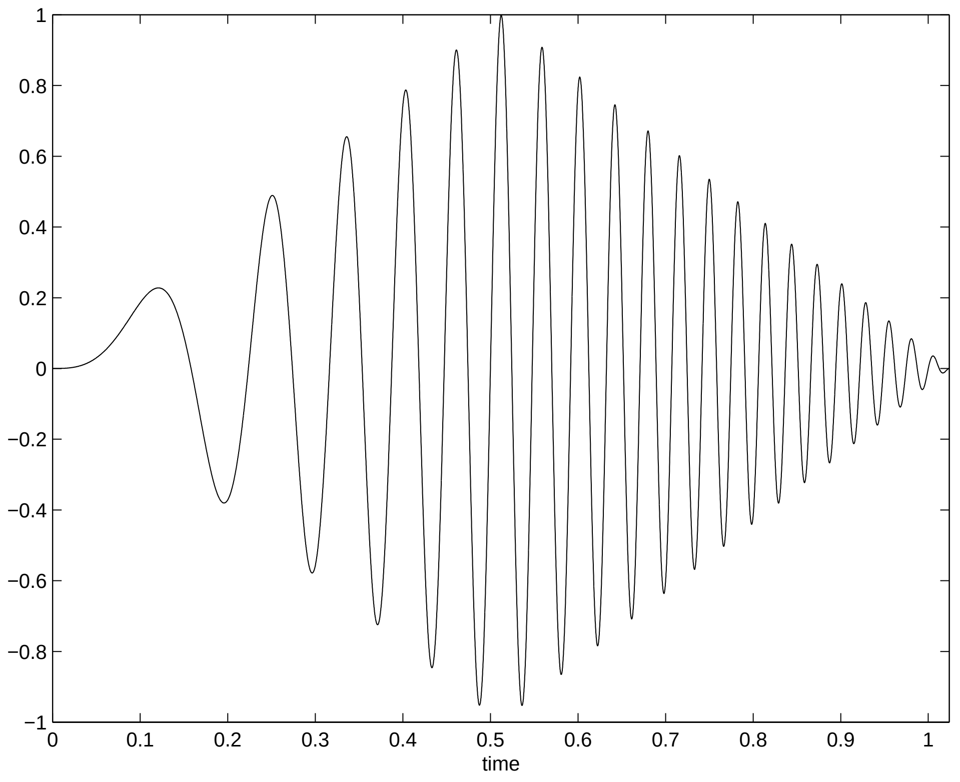


Figure C.15: Chirp signal with a triangular envelope.

3. Use the FFT algorithm, as done in lab C.10, to create a plot of the magnitude of the DFT of the chirp signal from the previous part over the frequency range -4 kHz to 4 kHz. Make sure the horizontal axis of your plot is labeled in Hertz. Does your plot look like a sensible frequency-domain representation of this chirp?

4. Modify your chirp so that it sweeps from 0 to 5 kHz. Listen to it. Do you hear aliasing artifacts? Plot the magnitude of the DFT over -4 kHz to 4 kHz. Can you see the aliasing artifacts in this plot? Explain why the plot has the shape that it does.

5. Return to the chirp that you generated in part 2, which sweeps from 0 to 2500 Hz. Create a new signal with sample rate 4 kHz by **downsampling** that chirp. That is, create a vector in Matlab that has half the length by selecting every second sample from the original chirp. I.e., if $y(n)$ is the original chirp, define w by

   $$w(n) = y(2n).$$

   Now plot the magnitude DFT of this signal.[6] Since the sampling rate is lower by a factor of 2, you should plot over the frequency interval -2 kHz to 2 kHz. Is there aliasing distortion? Why?

6. Return again to the chirp that you generated in part 2, which sweeps from 0 to 2500 Hz. Create a new signal with sample rate 16 kHz by **upsampling** that chirp. That is, create a vector in Matlab that has twice the length by inserting zero-valued samples between each pair of samples from the original chirp. I.e., if $y(n)$ is the original chirp, define z by

   $$z(n) = \begin{cases} y(n/2) & \text{if } n \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

   Now plot the magnitude DFT of this signal. Since the sampling rate is higher by a factor of 2, you should plot over the frequency interval -8 kHz to 8 kHz. Explain this plot. Listen to the sound by giving a sampling frequency argument to `soundsc` as follows:[7]

   ```
   soundsc(w, 16000);
   ```

   How does the sound correspond with the plot?

7. Design a Butterworth filter using the `butter` function in Matlab to get back a high quality rendition of the original chirp, but with a sample rate of 16000 Hz. Filter the signal with your filter and listen to it.

   Note that this technique is used in most compact disc players today. The signal on the CD is sampled at 44,100 Hz. The CD player first upsamples it by a factor of 4 or 8 to get a sample rate of 176,400 Hz or 352,800 Hz. The upsampling is accomplished by inserting

[6] Unfortunately, Matlab does not document what actually happens when you give a sampling frequency of 4000 to the `sound` or `soundsc` functions. On at least some computers, the sound that results from attempting to do this is difficult to interpret. Thus, we do not recommend attempting to listen to this downsampled chirp.

[7] The audio hardware on many computers directly supports a sample rate of 16 kHz, so at least on such computers, Matlab seems to correctly handle this command.

zero-valued samples. The resulting signal is then digitally filtered to get a high sample rate and accurate rendition of the audio. The higher sample rate signal is then converted to a continuous-time signal by a digital to analog converter that operates at the higher sample rate. This technique shifts most of the difficulty of rendering a high-quality audio signal to the discrete-time domain, where it can be done with digital circuits or microprocessors (such as programmable DSPs) rather than with analog circuits.

Instructor Verification Sheet for C.11

Name: ______________________________ Date: ______________________________

1. Explain the aliasing artifacts that you hear.

   Instructor verification: ______________________________

2. Generate chirp with triangular envelope.

   Instructor verification: ______________________________

3. Generate a frequency-domain plot of the chirp with a triangular envelope.

   Instructor verification: ______________________________

4. Show a frequency-domain plot with aliasing distortion and explain the distortion.

   Instructor verification: ______________________________

5. Show a frequency-domain plot with aliasing distortion and explain the distortion.

   Instructor verification: ______________________________

6. Show a frequency-domain plot with a double chirp and explain the sound.

   Instructor verification: ______________________________

7. Give a reasonable filter design.

   Instructor verification: ______________________________

Index