MASSIMO DI PIERRO

NUMERICAL ALGORITHMS IN NIM

APPLICATIONS IN PHYSICS, BIOLOGY, FINANCE

EXPERTS4SOLUTIONS

Massimo Di Pierro
Email: massimo.dipierro@gmail.com

Build Date: May 31, 2026

to my parents, who taught me everything that matters

Contents

1

Introduction

This is a port to the Nim language of a previous edition of this book from the same author. The original version of this book was in Python and was titled "Annotated Algorithms in Python". It was assembled from lectures given by the author over a period of 10 years at the School of Computing of DePaul University. The lectures cover multiple classes, including Analysis and Design of Algorithms, Scientific Computing, and Monte Carlo Simulations. These lectures teach the core knowledge required by any scientist interested in numerical algorithms and by students interested in computational finance.

We chose Nim because its syntax closely resembles that of Python, making the transition from the original edition natural for readers already familiar with Python, while at the same time Nim is as fast as the C language. This combination makes Nim a good choice for the development of scientific applications that does not compromise on performance. Over time the author has started to appreciate and love the syntax of Nim, its build speed, and its running speed.

The notes are not comprehensive, yet they try to identify and describe the most important concepts taught in those courses using a few common tools and unified notation.

In particular, these notes do not include proofs; instead, they provide

definitions and annotated code. The code is built in a modular way and is reused as much as possible throughout the book so that no step of the computations is left to the imagination. Each function defined in the code is accompanied by one or more examples of practical applications.

We take an interdisciplinary approach by providing examples in finance, physics, biology, and computer science. This is to emphasize that, although we often compartmentalize knowledge, there are very few ideas and methodologies that constitute the foundations of them all. Ultimately, this book is about problem solving using computers. The algorithms you will learn can be applied to different disciplines. Throughout history, it is not uncommon that an algorithm invented by a physicist would find application in, for example, biology or finance.

Almost all of the algorithms written in this book can be found in the `nlib` github repository:

`https://github.com/mdipierro/nlib-nim`

1.1 Main Ideas

Even if we cover many different algorithms and examples, there are a few central ideas in this book that we try to emphasize over and over.

The first idea is that we can simplify the solution of a problem by using an approximation and then systematically improve our approximation by iterating and computing corrections.

The divide-and-conquer methodology can be seen as an example of this approach. We do this with the insertion sort when we sort the first two numbers, then we sort the first three, then we sort the first four, and so on. We do it with merge sort when we sort each set of two numbers, then each set of four, then each set of eight, and so on. We do it with the Prim, Kruskal, and Dijkstra algorithms when we iterate over the nodes of a graph, and as we acquire knowledge about them, we use it to update the information about the shortest paths.

We use this approach in almost all our numerical algorithms because any

differentiable function can be approximated with a linear function:

$$f(x + \delta x) \simeq f(x) + f'(x)\delta x \tag{1.1}$$

We use this formula in the Newton method to solve nonlinear equations and optimization problems, in one or more dimensions.

We use the same approximation in the fix point method, which we use to solve equations like $f(x) = 0$; in the minimum residual and conjugate gradient methods; and to solve the Laplace equation in the last chapter of the book. In all these algorithms, we start with a random guess for the solution, and we iteratively find a better one until convergence.

The second idea of the book is that certain quantities are random, but even random numbers have patterns that we can capture using instruments like distributions and correlations. The presence of these patterns helps us model those systems that may have a random output (e.g., nuclear reactions, financial systems) and also helps us in computations. In fact, we can use random numbers to compute quantities that are not random (Monte Carlo methods). The most common approximation that we make in different parts of the book is that when a random variable x is localized at a point with a given uncertainty, δx, then its distribution is Gaussian. Thanks to the properties of Gaussian random numbers, we conclude the following:

- Using the linear approximation (our first big idea), if $z = f(x)$, the uncertainty in the output is

$$\delta z = f'(x)\delta x \tag{1.2}$$

- If we add two independent Gaussian random variables $z = x + y$, the uncertainty in the output is

$$\delta z = \sqrt{\delta x^2 + \delta y^2} \tag{1.3}$$

- If we add N independent and identically distributed Gaussian variables $z = \sum x_i$, the uncertainty in the output is

$$\delta z = \sqrt{N}\delta x \tag{1.4}$$

We use this over and over, for example, when relating the volatility over different time intervals (daily, yearly).

- If we compute an average of N independent and identically distributed Gaussian random variables, $z = 1/N \sum x_i$, the uncertainty in the average is

$$\delta z = \delta x / \sqrt{N} \tag{1.5}$$

  We use this to estimate the error on the average in a Monte Carlo computation. In that case, we write it as $d\mu = \sigma / \sqrt{N}$, and σ is the standard deviation of $\{x_i\}$.

The third idea is that the time it takes to run an iterative algorithm is proportional to the number of iterations. It is therefore our goal to minimize the number of iterations required to reach a target precision. We develop a language to compare algorithms based on their running time and classify algorithms into categories. This is useful to choose the best algorithm based on the problem at hand.

In the ultimate analysis, we can even try to understand ourselves as a parallel machine that models the input from the world by approximations. The brain is a graph that can be modeled by a neural network. The learning process is an ongoing optimization process in which the brain adjusts its synapses to produce better and better responses. The decision process mimics a search tree. We solve problems by searching for the most similar problems that we have encountered before, then we refine the solution. Our DNA is a code that evolved to efficiently compress the information necessary to grow us from a single cell into a complex being. We evolved according to evolutionary mechanisms that can be modeled using genetic algorithms. We can find our similarities with other organisms using the longest common subsequence algorithm. We can reconstruct our evolutionary tree using shortest-path algorithms and find out how we came to be.

1.2 About Nim

The programming language used in this book is Nim [1]. Nim has a high-level surface syntax that stays close to pseudo-code, which keeps the algorithms easy to read; at the same time it is statically typed and compiles to small native binaries, so the same code that documents an algorithm also runs efficiently. Nim's standard library, generics, first-class procedures, and lightweight macro system are sufficient for everything in this book without resorting to external dependencies.

The goal of the book is to explain the algorithms by building them from scratch. It is not our goal to teach the user about existing libraries that may be (and often are) faster than our implementation.

1.3 Book Structure

This book is divided into the following chapters:

- This introduction.
- An introduction to the Nim programming language. The introduction assumes the reader is not new to basic programming concepts such as conditionals, loops, and function calls, and teaches the syntax of Nim with particular focus on the standard-library modules that are important for scientific applications (`std/math`, `std/random`, `std/sequtils`, `std/tables`) and a few others.
- Chapter 3 is a short review of the general theory of algorithms with applications. There we review how to determine the running time of an algorithm from simple loops to more complex recursive algorithms. We review basic data structures used to store information such as lists, arrays, stacks, queues, trees, and graphs. We also review the classification of basic algorithms such as divide-and-conquer, dynamic programming, and greedy algorithms. In the examples, we peek into complex algorithms such as Shannon–Fano compression, a maze solver, a clustering algorithm, and a neural network.
- In chapter 4, we talk about traditional numerical algorithms, in particu-

lar, linear algebra, solvers, optimizers, integrators, and Fourier–Laplace transformations. We start by reviewing the concept of Taylor series and their convergence to understand approximations, sources of error, and convergence. We then use those concepts to build more complex algorithms by systematically improving their first-order (linear) approximation. Linear algebra serves us as a tool to approximate and implement functions of many variables.

- In chapter 5, we provide a review of probability and statistics and implement basic procedures to perform statistical analysis of random variables.
- In chapter 6, we discuss algorithms to generate random numbers from many distributions. Nim's `std/random` module provides high-quality uniform and Gaussian generators, and in subsequent chapters we use it, yet in this chapter we discuss in detail how pseudo random number generators work and their pitfalls.
- In chapter 7, we write about Monte Carlo simulations. This is a numerical technique that utilizes random numbers to solve otherwise deterministic problems. For example, in chapter 4, we talk about numerical integration in one dimension. Those algorithms can be extended to perform numerical integration in a few (two, three, sometimes four) dimensions, but they fail for very large numbers of dimensions. That is where Monte Carlo integration comes to our rescue, as it increasingly becomes the integration method of choice as the number of variables increases. We present applications of Monte Carlo simulations.
- Finally, in the appendix, we provide a compendium of useful formulas and definitions.

1.4 Book Software

We utilize the following software libraries developed by the author and available under an Open Source BSD License:

- `https://github.com/mdipierro/nlib-nim`

We also utilize the following third-party tools:

- `https://nim-lang.org` — the Nim compiler and standard library.
- `http://www.gnuplot.info/` — `gnuplot` is invoked from a small helper to render the figures in this book.

Acknowledgements

First and foremost many thanks to Andreas Rumpf who invented the beautiful Nim language and took the time to personally review this manuscript.

Thanks to the many people who helped review the original Python version of this book: Alan Etkins, Brian Fox, Dan Bowker, Ethan Sudman, Holly Monteith, Konstantinos Moutselos, Luca De Alfaro, Michael Gheith, Paula Mikrut, Sean Neilan, and John Plamondon. We also thank all the students of our classes for their useful comments and suggestions. Finally, we thank Wikipedia, from which we borrowed a few ideas and examples.

We also wish to thank Claude by Anthropic for significant help converting code from Python to Nim, catching technical inaccuracies, and clarifying explanations, and suggesting improvements throughout the text.

2

Overview of the Nim Language

2.1 About Nim

Nim [1] is a statically typed compiled systems-programming language with a high-level, indentation-based surface syntax. It produces small native binaries by compiling through C, C++, or JavaScript, while its standard library and metaprogramming facilities give it the productivity of a scripting language. Nim has a strong type system, automatic memory management (with optional manual control), generics, first-class procedures, exceptions, threads, channels, and a rich macro system.

Three properties of Nim shape the code in this book:

- Indentation defines blocks, but every declaration has a type. Type errors are caught at compile time.
- Identifiers are partially style-insensitive: after the first character, underscores and case differences are ignored, so `insertion_sort`, `insertionSort`, and `INSERTIONSORT` refer to the same identifier. The convention used here is `camelCase` for procedures and `PascalCase` for types.
- Procedures, methods, iterators, and operators are all separately overloadable. The $*$ suffix marks a name as exported from a module.

Nim is interoperable with other ecosystems: an external `db_sqlite` package binds it to SQLite [2] for persistence, and the C FFI gives access to

libraries such as BLAS or FFTW. For occasional needs that fall outside the standard library (e.g., plotting), we drive an external graphical toolkit through Nim's foreign-function bindings rather than re-implementing it.

You can find tutorials, the language manual, and the standard library reference on the official Nim website.

You may skip this chapter if you are already familiar with the Nim language; the rest of the book uses only the constructs introduced here.

2.2 Installing Nim

There are several ways to obtain a Nim toolchain: official binaries, the `choosenim` version manager, or your operating system's package manager. The approach we recommend, and the one used to develop and typeset this book, is the `Nix` package manager.

Nix is a package manager and build system, available on Linux and macOS, that takes a different approach from conventional tools such as `apt`, `brew`, or `pip`. Instead of mutating a shared system-wide set of installed packages, Nix builds each package in isolation from an explicit description of its inputs and stores the result under a unique path derived from those inputs. Because the inputs fully determine the output, builds are *reproducible*: the same description yields a byte-for-byte equivalent result on any machine, today or years from now. Different versions of the same tool can coexist without conflict, and nothing is installed globally unless you ask for it.

The practical consequence is that you can drop into a temporary shell that has exactly the tools you need, and nothing more, without touching the rest of your system. First install Nix itself with its official one-line installer:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Then, to get an environment with the Nim compiler, its package manager, and the `gnuplot` tool used for the figures in this book, run:

```
nix-shell -p nim nimble gnuplot
```

This downloads the requested tools (if they are not already in the local store) and opens a new shell in which `nim`, `nimble`, and `gnuplot` are on the `PATH`. You can confirm it works with:

```
nim --version
```

When you exit the shell, your system is left exactly as it was before. To make the environment persistent for a project, record the dependencies in a `shell.nix` file so that a bare `nix-shell` reconstructs the same environment for anyone who checks out the code.

2.3 The Nim REPL

While Nim is a compiled language, it ships with tooling that makes it possible to evaluate expressions interactively, in the same spirit as the Python REPL used in the original edition of this book. The interactive shell is convenient for exploring the language, trying out small expressions, inspecting the types inferred by the compiler, and quickly verifying the behavior of procedures from the standard library before committing them to a source file.

The most widely used interactive front-end is `inim`, an external tool that can be installed with Nim's package manager:

```
nimble install inim
```

Once installed, starting the REPL is as simple as typing:

```
inim
```

This opens a prompt where each line is compiled and executed on the fly:

```
nim> let x = 2 + 3
nim> echo x
5
nim> import std/math
nim> echo sqrt(2.0)
1.4142135623730951
```

Multi-line definitions (procedures, types, control-flow blocks) are supported by indentation, exactly as in a regular Nim source file. Variables and imports declared in the session remain available for the rest of the session, so the REPL behaves as an incremental sandbox.

For users who prefer not to install an extra tool, Nim itself provides a lightweight alternative through the `nim secret` command, which starts a small interpreter based on the compiler's VM. It supports a subset of the language sufficient for arithmetic, string manipulation, and most standard-library calls that do not rely on the C backend.

Throughout this book we will mostly write complete programs and compile them with `nim c -r`, but the REPL is a useful companion for experimenting with the snippets as you read.

2.4 Variables, types, and basic operators

2.4.1 `let`, `var`, and `const`

Every binding is introduced by one of three keywords:

```
let pi = 3.141592653589793   # immutable, value fixed at runtime
var counter = 0              # mutable
const Tau = 2.0 * pi         # compile-time constant
counter = counter + 1
```

A `let` binding cannot be reassigned; a `var` can. The compiler infers the type from the initializer; you can write it explicitly when there is no initializer or when you want to constrain the literal:

```
var n: int               # zero-initialized
var x: float = 0.5
let xs: seq[float] = @[]
```

The `@` symbol used in `@[]` is the sequence (heap-allocated, growable array) constructor. Written in front of an array literal it converts that fixed-size array into a `seq`; for example, `@[1, 2, 3]` is a `seq[int]` containing three elements, and `@[]` is an empty `seq` whose element type is taken from the surrounding context (here, `float`). Without the `@`, the literal `[1, 2, 3]` would denote a fixed-size `array[3, int]` instead. We will use the `@[...]` syntax extensively throughout the book whenever we need a dynamically sized list of values, much as one would use Python's `[...]` list literals.

2.4.2 Numbers, booleans, and characters

The fundamental numeric types are `int` (machine word, signed), `int8`, `int16`, `int32`, `int64`, the unsigned variants `uint8`–`uint64`, and the floating-point types `float` (a synonym for `float64`) and `float32`. `bool` has values `true` and `false`; `char` is an 8-bit byte.

```
let a = 7              # int
let b = 3.5            # float
let big: int64 = 1'i64 shl 40
let mask: uint32 = 0x9d2c5680'u32
echo a div 2           # integer division: 3
echo a mod 2           # remainder: 1
echo a / 2             # float division: 3.5
echo 2.0 ^ 10          # 1024.0 (power); import std/math for this operator
echo not true          # false
```

The bitwise operators are spelled `and`, `or`, `xor`, `not`, `shl`, `shr` (in Nim `and`/`or`/`not` cover both logical and bitwise; the meaning depends on the operand types). Type suffixes such as `'i64` and `'u32` fix the type of an integer literal.

2.4.3 Strings and stringification

Strings are mutable, length-prefixed byte sequences. Concatenation is the `&` operator; conversion to a string is the `$` prefix.

```
let name = "world"
let greeting = "Hello, " & name & "!"
echo greeting              # Hello, world!
echo $42 & " is a number"  # 42 is a number
echo greeting.len          # 13
echo greeting[0 ..< 5]     # "Hello"
echo greeting.toLowerAscii # std/strutils
```

For formatted floating-point output, the standard library provides `formatFloat` and `&""` string interpolation:

```
import std/strutils, std/strformat
let p = 3.14159
echo formatFloat(p, ffDecimal, 3)   # 3.142
echo &"pi rounded is {p:.3f}"       # pi rounded is 3.142
```

2.5 Composite types

2.5.1 Sequences and arrays

A `seq[T]` is a heap-allocated, growable, ordered list of `T`. An `array[N, T]` has a fixed length `N` known at compile time. Both are zero-indexed.

```
var xs: seq[int] = @[1, 2, 3]   # @[]: empty seq literal
xs.add 4                         # @[1, 2, 3, 4]
xs[0] = 10
echo xs.len                      # 4
echo xs[^1]                      # 4 (last element)
echo xs[1 .. 2]                  # @[2, 3] (slice)

var grid: array[3, float] = [0.0, 0.0, 0.0]
grid[2] = 1.0

var zeros = newSeq[float](100)               # 100 zeros
let ones = newSeqWith(100, 1.0)              # std/sequtils
let evens = (0 ..< 10).toSeq().mapIt(it * 2) # @[0, 2, 4, ...]
```

The `openArray[T]` type is used as a parameter type to accept either a `seq[T]`, an `array`, or a slice without copying.

```
proc sum(xs: openArray[float]): float =
  for x in xs: result += x

echo sum(@[1.0, 2.0, 3.0])
echo sum([10.0, 20.0])
```

2.5.2 Tuples

A `tuple` is a fixed-size record. Fields can be named or anonymous. Tuples support destructuring.

```
let point: (float, float) = (1.5, 2.5)
let (x, y) = point
echo x, " ", y                # 1.5 2.5

type
  Range = tuple[lo, hi: float]

let r: Range = (lo: 0.0, hi: 1.0)
echo r.hi - r.lo              # 1.0
```

A procedure can return multiple values simply by returning a tuple:

```
proc minmax(xs: openArray[float]): (float, float) =
  var lo = xs[0]
  var hi = xs[0]
  for v in xs:
```

```
    if v < lo: lo = v
    if v > hi: hi = v
  (lo, hi)

let (mn, mx) = minmax(@[3.0, 1.0, 4.0, 1.0, 5.0])
```

2.5.3 Tables (dictionaries)

The `Table[K, V]` container in `std/tables` provides a hash-table mapping. `initTable` creates an empty table; `toTable` converts a sequence of pairs.

```
import std/tables

var counts = initTable[string, int]()
counts["apples"] = 3
counts["oranges"] = 5
counts["apples"] += 1
echo counts["apples"]               # 4
echo "kiwi" in counts               # false
echo counts.getOrDefault("kiwi", 0) # 0

for key, value in counts:
  echo key, " -> ", value
```

For deterministic key order use `OrderedTable`; for thread-shared state use `SharedTable`.

2.5.4 Sets

For collections of unique values use `HashSet[T]` from `std/sets`.

```
import std/sets

var seen: HashSet[int]
seen.incl 5
seen.incl 7
seen.incl 5            # already there
echo 5 in seen         # true
echo seen.card         # 2

let alphabet = toHashSet("ABCDE")
echo 'C' in alphabet
```

The built-in `set[T]` (lowercase) is a fast bitset for ordinal types like `char` or small enums; `HashSet` works for any hashable type.

2.6 Control flow

2.6.1 `if`, `case`, `while`, `for`

`if/elif/else` works as in most languages; it is also an expression.

```
let x = 7
if x < 0:
  echo "negative"
elif x == 0:
  echo "zero"
else:
  echo "positive"

let parity = if x mod 2 == 0: "even" else: "odd"
```

`case` dispatches on a value (typically an ordinal):

```
let grade = 'B'
case grade
of 'A': echo "excellent"
of 'B', 'C': echo "good"
of 'D' .. 'F': echo "needs work"
else: echo "?"
```

`while` loops until its condition becomes false. `break` exits a loop; `continue` skips to the next iteration.

```
var n = 1
while n < 1000:
  n = n * 2
echo n            # 1024
```

`for` iterates over any iterator. Built-in iterators include integer ranges and any container's elements.

```
for i in 0 ..< 5: echo i        # 0 1 2 3 4 (exclusive upper)
for i in 1 .. 5: echo i         # 1 2 3 4 5 (inclusive)
for i in countdown(5, 1): echo i
for i in countup(0, 9, 2): echo i  # 0 2 4 6 8

let xs = @[10, 20, 30]
for v in xs: echo v             # values
for i, v in xs: echo i, " ", v  # index and value
```

2.7 Procedures

2.7.1 `proc`, default arguments, and `result`

A procedure is declared with `proc`. Each argument has a type. The return type follows a colon. The body uses indentation; `=` introduces it.

```
proc square(x: float): float =
  x * x                     # last expression is the return value

echo square(3.0)            # 9.0
```

Inside any procedure, the special variable `result` is automatically declared, initialized to its zero value, and returned at the end. This is useful when a result is built up incrementally. Depending on the Nim version and compiler flags, however, an explicit initialization may be enforced.

```
proc factorial(n: int): int =
  result = 1
  for i in 2 .. n:
    result *= i

echo factorial(6)           # 720
```

You can also use `return value` to exit early. Default arguments are written `name: type = default` and may be passed positionally or by name:

```
proc clamp(x: float, lo = 0.0, hi = 1.0): float =
  if x < lo: lo
  elif x > hi: hi
  else: x

echo clamp(2.0)                          # 1.0
echo clamp(-3.0, lo = -1.0, hi = 1.0)    # -1.0
```

A procedure that does not return a value omits the return-type annotation:

```
proc greet(name: string) =
  echo "Hello, ", name
```

2.7.2 The `discard` keyword

When a procedure returns a value, Nim does not let you call it as a statement and silently throw the result away: doing so is a compile-time error. This rule is intentional, and it catches the common mistake of forgetting to use a function's return value. When you really do want to ignore the result, you must say so explicitly with the `discard` keyword:

```
proc nextId(): int =
  result = 42

nextId()           # error: value of type 'int' has to be used
discard nextId()   # ok: result intentionally thrown away
```

The same keyword is also used as a no-op statement, useful when the syntax requires a body but you have nothing to do (for example, in an empty branch of an `if` or in a placeholder procedure):

```
proc todo() =
  discard          # placeholder body; does nothing

if x > 0:
  doSomething()
else:
  discard          # explicitly do nothing
```

We will see `discard` used throughout the book whenever a procedure is called purely for its side effects (e.g., advancing the state of a Markov chain or invoking an external command) and its return value is not needed.

2.7.3 Generics

Generic parameters go in square brackets after the procedure name. The compiler instantiates a separate copy for each set of types used. This is analogous to `template<class T>` in C++: `T` is a type placeholder, and the compiler generates a specialized version of the procedure for every concrete type at the call sites.

```
proc swap[T](a, b: var T) =
  let tmp = a
  a = b
  b = tmp

var i = 1
var j = 2
swap(i, j)

proc maxOf[T](xs: openArray[T]): T =
  result = xs[0]
  for x in xs:
    if x > result: result = x

```

```
echo maxOf(@[3, 1, 4, 1, 5, 9, 2, 6])
echo maxOf(@["apple", "kiwi", "pear"])
```

The `var T` marker on an argument means the procedure can mutate it; without `var`, arguments are passed by value and cannot be reassigned inside the procedure. However, we often use `ref` types for arguments, which are pointers, so they still allow for mutations, but not for rebindings.

2.7.4 First-class procedures and closures

A procedure type is written `proc(args): RetType`. Procedure values can be stored in variables, passed as arguments, and returned from other procedures. When a returned procedure captures a local variable, Nim creates a closure on the heap.

```
proc adder(n: int): proc(x: int): int =
  result = proc(x: int): int = x + n

let add3 = adder(3)
echo add3(10)               # 13

# A higher-order procedure:
proc apply(f: proc(x: float): float, xs: seq[float]): seq[float] =
  for x in xs: result.add f(x)

let squared = apply(proc(x: float): float = x * x, @[1.0, 2.0, 3.0])
echo squared                # @[1.0, 4.0, 9.0]
```

The `std/sugar` module offers a shorter `=>` syntax: `(x: float) => x * x`.

2.7.5 Iterators and `yield`

An `iterator` produces a sequence of values lazily. Each `yield` hands a value back to the `for` loop calling the iterator.

```
iterator countSquares(n: int): int =
  var i = 1
  while i * i <= n:
    yield i * i
    inc i

for sq in countSquares(50):
  echo sq                   # 1 4 9 16 25 36 49
```

Iterators are central to many algorithms in this book (e.g., the Metropolis

sampler in chapter 7).

2.7.6 Operator overloading

Operators are just procedures whose names are wrapped in backticks. You can overload them on user-defined types.

```
type Vec2 = object
  x, y: float

proc `+`(a, b: Vec2): Vec2 = Vec2(x: a.x + b.x, y: a.y + b.y)
proc `*`(s: float, v: Vec2): Vec2 = Vec2(x: s * v.x, y: s * v.y)
proc `$`(v: Vec2): string = "(" & $v.x & ", " & $v.y & ")"

let u = Vec2(x: 1.0, y: 2.0)
let v = Vec2(x: 3.0, y: 4.0)
echo u + v                  # (4.0, 6.0)
echo 0.5 * u                # (0.5, 1.0)
```

Unary operators are overloaded the same way — just declare a procedure that takes a single argument:

```
proc `-`(v: Vec2): Vec2 = Vec2(x: -v.x, y: -v.y)
echo -u                     # (-1.0, -2.0)
```

The indexing operators `` `[]` `` and `` `[]=` `` can also be overloaded, and they may take any number of indices of any type. This is the trick that chapter 4's `Matrix` type uses to expose two-dimensional element access with the natural `m[i, j]` syntax: the read form is implemented by `` `[]` `` (which receives the row and column as two separate arguments), and the write form `m[i, j] = value` is dispatched to `` `[]=` `` (which additionally receives the value to store):

```
type Grid = object
  rows, cols: int
  data: seq[float]

proc `[]`(g: Grid, i, j: int): float =
  g.data[i * g.cols + j]

proc `[]=`(g: var Grid, i, j: int, value: float) =
  g.data[i * g.cols + j] = value

var g = Grid(rows: 2, cols: 3, data: newSeq[float](6))
g[0, 1] = 4.2               # calls `[]=`(g, 0, 1, 4.2)
echo g[0, 1]                # calls `[]`(g, 0, 1) -> 4.2
```

In-place compound operators such as `+=`, `-=`, `*=`, and `/=` are likewise just procedures, and overloading them lets a user-defined type behave like a built-in numeric type:

```
proc `+=`(a: var Vec2, b: Vec2) =
  a.x += b.x
  a.y += b.y

var w = Vec2(x: 1.0, y: 2.0)
w += Vec2(x: 10.0, y: 20.0)
echo w                        # (11.0, 22.0)
```

The `Matrix` type in chapter 4 combines all of these: `` `[]` ``/`` `[]=` `` for (i, j) access, the binary arithmetic operators between matrices, between a matrix and a scalar, and a unary `` `-` `` for negation.

2.8 Object types and methods

2.8.1 `object` and `ref object`

An `object` groups named fields. By default, objects are value types: assignment and parameter passing copy the entire structure. A `ref object` is a reference type, allocated on the heap and managed by the runtime; assignment copies the pointer, not the contents.

```
type
  Point = object         # value type
    x, y: float
  Stamp = ref object     # reference type
    id: int
    label: string

let p = Point(x: 0.0, y: 1.0)        # value constructor
let s = Stamp(id: 1, label: "A")     # ref constructor (allocates)
echo p.y, " ", s.label
```

Most data structures in this book (`Matrix`, `DisjointSets`, `NeuralNetwork`, `MCEngine`, ...) are `ref objects` because we want them to be passed by reference and mutated in place. This is in contrast to modern Nim style, which prefers value types and move semantics instead of `ref objects`, but our style makes Nim easier to use for beginners, especially those coming from Python.

A common pattern is to provide a `newX` constructor procedure:

```
proc newPoint(x, y: float): Point =
  Point(x: x, y: y)
```

2.8.2 Methods and inheritance

For dynamic dispatch (selecting an implementation based on the runtime type of an argument) Nim uses `method` instead of `proc`. The base type must inherit from `RootObj`, and the base method is annotated with `{.base.}`.

```
type
  Shape = ref object of RootObj
  Circle = ref object of Shape
    radius: float
  Rectangle = ref object of Shape
    width, height: float

method area(s: Shape): float {.base.} =
  raise newException(CatchableError, "abstract")

method area(c: Circle): float = 3.141592653589793 * c.radius ^ 2
method area(r: Rectangle): float = r.width * r.height

let shapes: seq[Shape] = @[Shape(Circle(radius: 1.0)),
                           Shape(Rectangle(width: 2.0, height: 3.0))]
for s in shapes: echo s.area()
```

Chapter 7's `MCEngine` uses exactly this pattern: a base method `simulateOnce` declares the abstract behavior, and each subclass overrides it.

2.9 Modules and visibility

2.9.1 The export marker `*`

A `proc`, `type`, `var`, or field declared with a trailing `*` is exported from its module; without it, the name is private to the file. This is Nim's visibility marker; there is no separate `public` or `private` keyword.

```
# in nlib.nim
proc square*(x: float): float = x * x   # public
proc helper(x: float): float = x + 1    # private to nlib
```

A consumer imports the module by file name (no extension):

```
import nlib
```

```
echo square(3.0)
```

2.9.2 The standard library

The standard library lives under the `std/` prefix. The modules used in the rest of this book are:

`std/math`	`sin`, `cos`, `sqrt`, `exp`, `ln`, `pow`, `PI`
`std/random`	`rand`, `randomize`, `sample`, `gauss`, `shuffle`
`std/sequtils`	`mapIt`, `filterIt`, `toSeq`, `newSeqWith`
`std/algorithm`	`sort`, `reverse`, `binarySearch`
`std/tables`	`Table`, `initTable`, `OrderedTable`
`std/sets`	`HashSet`, `incl`, `excl`
`std/options`	`Option[T]`, `some`, `none`, `isSome`
`std/strutils`	`formatFloat`, `split`, `startsWith`
`std/strformat`	`&""` string interpolation
`std/heapqueue`	min-heap priority queue
`std/deques`	double-ended queue
`std/json`	JSON parsing and generation
`std/os`	file-system path manipulation, environment
`std/times`	`epochTime`, `now`

2.10 Exceptions

Errors are raised with `raise newException(ErrorType, "message")` and caught with `try/except/finally`.

```
proc safeDiv(a, b: float): float =
  if b == 0:
    raise newException(DivByZeroDefect, "division by zero")
  a / b

try:
  echo safeDiv(1.0, 0.0)
except DivByZeroDefect:
  echo "caught a division by zero"
except CatchableError as e:
  echo "other error: ", e.msg
finally:
  echo "always runs"
```

The most commonly used exception types in this book are `ArithmeticDefect` (numerical algorithms that fail to converge), `IOError`, `ValueError`, `IndexDefect`, and `CatchableError` as a catch-all for recoverable errors.

2.11 Files and persistence

2.11.1 Reading and writing files

```
writeFile("hello.txt", "Hello, world!\n")
let content = readFile("hello.txt")
echo content
```

2.11.2 A persistent dictionary backed by JSON

Several examples in later chapters use a dictionary that survives between program runs. The implementation below keeps the entire mapping as a JSON file on disk: it has no external dependencies and is enough for the memoization and cached-stock-data use cases.

Listing 2.1: in file: `nlib/persistent.nim`

```
import std/[json, os, tables]

type
  PersistentDictionary* = ref object
    path*: string
    cache*: Table[string, JsonNode]

proc load(p: PersistentDictionary) =
  if fileExists(p.path):
    let raw = readFile(p.path)
    if raw.len > 0:
      let root = parseJson(raw)
      for k, v in root:
        p.cache[k] = v

proc save(p: PersistentDictionary) =
  var root = newJObject()
  for k, v in p.cache:
    root[k] = v
  writeFile(p.path, $root)

proc newPersistentDictionary*(path: string): PersistentDictionary =
```

```
  result = PersistentDictionary(path: path,
                                cache: initTable[string, JsonNode]())
  load(result)

proc close*(p: PersistentDictionary) =
  save(p)

proc `[]=`*(p: PersistentDictionary, key: string, value: JsonNode) =
  p.cache[key] = value
  save(p)

proc `[]`*(p: PersistentDictionary, key: string): JsonNode =
  if key in p.cache: p.cache[key] else: nil

proc contains*(p: PersistentDictionary, key: string): bool =
  key in p.cache
```

Usage:

```
let storage = newPersistentDictionary("memo.json")
storage["pi"] = %3.141592653589793
echo storage["pi"].getFloat
```

The `%` prefix and `%*` macro from `std/json` convert any Nim value to a `JsonNode`; `.getFloat`, `.getStr`, `.getInt`, etc., extract the typed payload back. Together they give us a key-value store usable from any chapter that needs cross-run state (e.g., the memoization examples in chapter 3 and the stock-data examples in chapter 4). Swapping the backend for SQLite is a localized change once a maintained `db_sqlite` package is added to the project.

2.12 Plotting

Plotting is a peripheral concern for this book: the algorithms are independent of how their output is rendered. When a chapter produces a figure, we write the data to a CSV file and let an external plotting tool draw it. The default tool used here is `gnuplot`, which reads simple ASCII data and produces PNG/PDF/SVG output without any link-time dependency on the Nim program.

A small helper, used throughout the book, writes a sequence of (x, y) pairs to a file and invokes `gnuplot`:

Listing 2.2: in file: `nlib/plotting.nim`

```nim
import std/[strutils, osproc]

proc savePlot*(filename: string, xs, ys: openArray[float],
               title = "", xlab = "x", ylab = "y") =
  ## Writes (x, y) data to `filename.dat` and renders `filename` with
  ## gnuplot. The output format is determined by the file extension.
  let dataPath = filename & ".dat"
  var f = open(dataPath, fmWrite)
  for i in 0 ..< xs.len:
    f.writeLine xs[i].formatFloat(ffDecimal, 6) & " " &
                ys[i].formatFloat(ffDecimal, 6)
  f.close()
  discard execCmd("gnuplot -e \"" &
                  "set terminal pngcairo; " &
                  "set output '" & filename & "'; " &
                  "set title '" & title & "'; " &
                  "set xlabel '" & xlab & "'; " &
                  "set ylabel '" & ylab & "'; " &
                  "plot '" & dataPath & "' with lines\"")
```

Most plotting calls in the book funnel through `savePlot` or its variants for histograms, scatter plots, and surface plots. Replacing `gnuplot` with another back end (e.g., a native SVG writer or a C library bound through Nim's FFI) only requires re-implementing this single helper.

2.13 Putting it together

The remaining chapters build a small Nim package, `nlib`, organized as one umbrella module `nlib.nim` that re-exports a handful of focused submodules under `nlib/` (e.g. `nlib/sorting.nim`, `nlib/matrix.nim`, `nlib/randomgen.nim`, ...). Listings tagged

```
in file: nlib/⟨module⟩.nim
```

belong to the named submodule; other listings live in their own files (named in the caption) or are inline examples meant to be typed at the REPL or compiled into a small program. A consumer of the library only needs

```nim
import nlib
```

to pull in the whole API at once. To run an example, save it to a file and compile with

```
nim c -d:release --threads:on example.nim
./example
```

3
Theory of Algorithms

An algorithm is a step-by-step procedure for solving a problem and is typically developed before doing any programming. The word comes from *algorism*, from the mathematician al-Khwarizmi, and was used to refer to the rules of performing arithmetic using Hindu–Arabic numerals and the systematic solution of equations.

In fact, algorithms are independent of any programming language. Efficient algorithms can have a dramatic effect on our problem-solving capabilities.

The basic steps of algorithms are loops (`for`), conditionals (`if`), and function calls. Algorithms also make use of arithmetic expressions, logical expressions (`not`, `and`, `or`), and expressions that can be reduced to the other basic components.

The issues that concern us when developing and analyzing algorithms are the following:

1. Correctness: of the problem specification, of the proposed algorithm, and of its implementation in some programming language (we will not worry about the third one; program verification is another subject altogether)
2. Amount of work done: for example, running time of the algorithm in terms of the input size (independent of hardware and programming

language)

3. Amount of space used: here we mean the amount of extra space (system resources) beyond the size of the input (independent of hardware and programming language); we will say that an algorithm is *in place* if the amount of extra space is constant with respect to input size
4. Simplicity, clarity: unfortunately, the simplest is not always the best in other ways
5. Optimality: can we prove that it does as well as or better than any other algorithm?

3.1 Order of growth of algorithms

The *insertion sort* is a simple algorithm in which an array of elements is sorted in place, one entry at a time. It is not the fastest sorting algorithm, but it is simple and does not require extra memory other than the memory needed to store the input array.

The insertion sort works by iterating. Every iteration i of the insertion sort removes one element from the input data and inserts it into the correct position in the already-sorted subarray `A[j]` for $0 \leq j < i$. The algorithm iterates n times (where n is the total size of the input array) until no input elements remain to be sorted:

```
proc insertionSort[T](a: var seq[T]) =
  for i in 1 ..< a.len:
    var j = i
    while j > 0 and a[j] < a[j-1]:
      swap(a[j], a[j-1])
      dec j
```

Here is an example:

```
import std/random
randomize()
var a = newSeq[int](20)
for i in 0 ..< 20: a[i] = rand(100)
insertionSort(a)
echo a
# @[6, 8, 9, 17, 30, 31, 45, 48, 49, 56, 56, 57, 65, 66, 75, 75, 82, 89, 90, 99]
```

One important question is, how long does this algorithm take to run?

How does its running time scale with the input size?

Given any algorithm, we can define three characteristic functions:

- $T_{worst}(n)$: the running time in the worst case
- $T_{best}(n)$: the running time in the best case
- $T_{average}(n)$: the running time in the average case

The best case for an insertion sort is realized when the input is already sorted. In this case, the inner for loop exits (breaks) always at the first iteration, thus only the most outer loop is important, and this is proportional to n; therefore $T_{best}(n) \propto n$. The worst case for the insertion sort is realized when the input is sorted in reversed order. In this case, we can prove, and we do so subsequently, that $T_{worst}(n) \propto n^2$. For this algorithm, a statistical analysis shows that the worst case is also the average case.

Often we cannot determine exactly the running time function, but we may be able to set bounds to the running time.

We define the following sets:

- $O(g(n))$: the set of functions that grow no faster than $g(n)$ when $n \to \infty$
- $\Omega(g(n))$: the set of functions that grow no slower than $g(n)$ when $n \to \infty$
- $\Theta(g(n))$: the set of functions that grow at the same rate as $g(n)$ when $n \to \infty$
- $o(g(n))$: the set of functions that grow slower than $g(n)$ when $n \to \infty$
- $\omega(g(n))$: the set of functions that grow faster than $g(n)$ when $n \to \infty$

We can rewrite the preceding definitions in a more formal way:

$$O(g(n)) \equiv \{f(n) : \exists n_0, c_0,\ \forall n > n_0,\ 0 \leq f(n) < c_0 g(n)\} \quad (3.1)$$

$$\Omega(g(n)) \equiv \{f(n) : \exists n_0, c_0,\ \forall n > n_0,\ 0 \leq c_0 g(n) < f(n)\} \quad (3.2)$$

$$\Theta(g(n)) \equiv O(g(n)) \cap \Omega(g(n)) \quad (3.3)$$

$$o(g(n)) \equiv O(g(n)) - \Omega(g(n)) \quad (3.4)$$

$$\omega(g(n)) \equiv \Omega(g(n)) - O(g(n)) \quad (3.5)$$

We can also provide a practical rule to determine if a function f belongs to one of the previous sets defined by g.

Compute the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \quad (3.6)$$

and look up the result in the following table:

$$\begin{array}{lcl}
a \text{ is positive or zero} & \Longrightarrow & f(n) \in O(g(n)) \Leftrightarrow f \preceq g \\
a \text{ is positive or infinity} & \Longrightarrow & f(n) \in \Omega(g(n)) \Leftrightarrow f \succeq g \\
a \text{ is positive} & \Longrightarrow & f(n) \in \Theta(g(n)) \Leftrightarrow f \sim g \\
a \text{ is zero} & \Longrightarrow & f(n) \in o(g(n)) \Leftrightarrow f \prec g \\
a \text{ is infinity} & \Longrightarrow & f(n) \in \omega(g(n)) \Leftrightarrow f \succ g
\end{array} \quad (3.7)$$

Notice the preceding practical rule assumes the limits exist.

Here is an example:

Given $f(n) = n \log n + 3n$ and $g(n) = n^2$

$$\lim_{n\to\infty} \frac{n \log n + 3n}{n^2} \overset{l'Hopital}{\longrightarrow} \lim_{n\to\infty} \frac{1/n}{2} = 0 \quad (3.8)$$

we conclude that $n \log n + 3n$ is in $O(n^2)$.

Given an algorithm A that acts on input of size n, we say that the algorithm is $O(g(n))$ if its worst running time as a function of n is in $O(g(n))$. Similarly, we say that the algorithm is in $\Omega(g(n))$ if its best running time is in $\Omega(g(n))$. We also say that the algorithm is in $\Theta(g(n))$ if both its best running time and its worst running time are in $\Theta(g(n))$.

More formally, we can write the following:

$$T_{worst}(n) \in O(g(n)) \quad \Rightarrow \quad A \in O(g(n)) \tag{3.9}$$
$$T_{best}(n) \in \Omega(g(n)) \quad \Rightarrow \quad A \in \Omega(g(n)) \tag{3.10}$$
$$A \in O(g(n)) \text{ and } A \in \Omega(g(n)) \quad \Rightarrow \quad A \in \Theta(g(n)) \tag{3.11}$$
$$\tag{3.12}$$

We still have not solved the problem of computing the best, average, and worst running times.

3.1.1 Best and worst running times

The procedure for computing the worst and best running times is similar. It is simple in theory but difficult in practice because it requires an understanding of the algorithm's inner workings.

Consider the following algorithm, which finds the minimum of an array or list A:

```
proc findMinimum[T](a: openArray[T]): T =
  result = a[0]
  for element in a:
    if element < result:
      result = element
```

To compute the running time in the worst case, we assume that the maximum number of computations is performed. That happens when the if statements are always `True`. To compute the best running time, we assume that the minimum number of computations is performed. That happens when the if statement is always `False`. Under each of the two scenarios, we compute the running time by counting how many times the most nested operation is performed.

In the preceding algorithm, the most nested operation is the evaluation of the `if` statement, and that is executed for each element in `A`; for example, assuming `A` has n elements, the `if` statement will be executed n times.

Therefore both the best and worst running times are proportional to n, thus making this algorithm $O(n)$, $\Omega(n)$, and $\Theta(n)$.

More formally, we can observe that this algorithm performs the following operations:

- One assignment (line 2)
- Loops $n =$`len(A)` times (line 3)
- For each loop iteration, performs one comparison (line 4)
- Line 5 is executed only if the condition is true

Because there are no nested loops, the time to execute each loop iteration is about the same, and the running time is proportional to the number of loop iterations.

For a loop iteration that does not contain further loops, the time it takes to compute each iteration, its `running time`, is constant (therefore equal to 1). For algorithms that contain nested loops, we will have to evaluate nested sums.

Here is the simplest example:

```
1 proc loop0(n: int) =
2   for i in 0 ..< n:
3     echo i
```

which we can map into

$$T(n) = \sum_{i=0}^{i<n} 1 = n \in \Theta(n) \Rightarrow \texttt{loop0} \in \Theta(n) \tag{3.13}$$

Here is a similar example where we have a single loop (corresponding to a single sum) that loops n^2 times:

```
1 proc loop1(n: int) =
2   for i in 0 ..< n * n:
3     echo i
```

and here is the corresponding running time formula:

$$T(n) = \sum_{i=0}^{i<n^2} 1 = n^2 \in \Theta(n^2) \Rightarrow \texttt{loop1} \in \Theta(n^2) \tag{3.14}$$

The following provides an example of nested loops:

```
proc loop2(n: int) =
  for i in 0 ..< n:
    for j in 0 ..< n:
      echo i, " ", j
```

Here the time for the inner loop is directly determined by n and does not depend on the outer loop's counter; therefore

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<n} 1 = \sum_{i=0}^{i<n} n = n^2 + ... \in \Theta(n^2) \Rightarrow \texttt{loop2} \in \Theta(n^2) \tag{3.15}$$

This is not always the case. In the following code, the inner loop does depend on the value of the outer loop:

```
proc loop3(n: int) =
  for i in 0 ..< n:
    for j in 0 ..< i:
      echo i, " ", j
```

Therefore, when we write its running time in terms of a sum, care must be taken that the upper limit of the inner sum is the upper limit of the outer sum:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i} 1 = \sum_{i=0}^{i<n} i = \frac{1}{2} n(n-1) \in \Theta(n^2) \Rightarrow \texttt{loop3} \in \Theta(n^2) \tag{3.16}$$

The appendix of this book provides examples of typical sums that come up in these types of formulas and their solutions.

Here is one more example falling in the same category, although the inner loop depends quadratically on the index of the outer loop:

Example: loop4

```
proc loop4(n: int) =
  for i in 0 ..< n:
    for j in 0 ..< i * i:
      echo i, " ", j
```

Therefore the formula for the running time is more complicated:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{i<n} i^2 = \frac{1}{6} n(n-1)(2n-1) \in \Theta(n^3) \quad (3.17)$$

$$\Rightarrow \quad \texttt{loop4} \in \Theta(n^3) \quad (3.18)$$

If the algorithm does not contain nested loops, then we need to compute the running time of each loop and take the maximum:

Example: concatenateo

```
proc concatenate0(n: int) =
  for i in 0 ..< n * n:
    echo i
  for j in 0 ..< n * n * n:
    echo j
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate0} \in \Theta(n^3) \quad (3.19)$$

If there is an if statement, we need to compute the running time for each condition and pick the maximum when computing the worst running time, or the minimum for the best running time:

```
proc concatenate1(n: int, a: int) =
  if a < 0:
    for i in 0 ..< n * n:
      echo i
  else:
    for j in 0 ..< n * n * n:
      echo j
```

$$T_{worst}(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in O(n^3) \quad (3.20)$$

$$T_{best}(n) = \Theta(\min(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in \Omega(n^2) \quad (3.21)$$

This can be expressed more formally as follows:

$$O(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \quad (3.22)$$
$$\Theta(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \quad (3.23)$$
$$\Omega(f(n)) + \Theta(g(n)) = \Omega(f(n)) \text{ iff } f(n) \in \Omega(g(n)) \quad (3.24)$$

which we can apply as in the following example:

$$T(n) = [\underbrace{n^2 + n + 3}_{\Theta(n^2)} + \underbrace{e^n - \log n}_{\Theta(e^n)}] \in \Theta(e^n) \text{ because } n^2 \in O(e^n) \quad (3.25)$$

3.2 Recurrence relations

The *merge sort* [3] is another sorting algorithm. It is faster than the insertion sort. It was invented by John von Neumann, the physicist credited for inventing also modern computer architecture and game theory.

The merge sort works as follows.

If the input array has length 0 or 1, then it is already sorted, and the algorithm does not perform any other operation.

If the input array has a length greater than 1, it divides the array into two subsets of about half the size. Each subarray is sorted by applying the merge sort recursively (it calls itself!). It then merges the two subarrays back into one sorted array (this step is called *merge*).

Consider the following implementation of the merge sort:

```
proc merge[T](a: var seq[T], p, q, r: int) =
  var b: seq[T] = @[]
  var i = p
  var j = q
  while true:
    if a[i] <= a[j]:
      b.add a[i]; inc i
    else:
      b.add a[j]; inc j
    if i == q:
      while j < r: b.add a[j]; inc j
      break
```

```
    if j == r:
      while i < q: b.add a[i]; inc i
      break
  for k in 0 ..< b.len:
    a[p + k] = b[k]

proc mergesort*[T](a: var seq[T], p = 0, r = -1) =
  let r = if r < 0: a.len else: r
  if p < r - 1:
    let q = (p + r) div 2
    mergesort(a, p, q)
    mergesort(a, q, r)
    merge(a, p, q, r)
```

Because this algorithm calls itself *recursively*, it is more difficult to compute its running time.

Consider the `merge` function first. At each step, it increases either i or j, where i is always in between p and q and j is always in between q and r. This means that the running time of the merge is proportional to the total number of values they can span from p to r. This implies that

$$\text{merge} \in \Theta(r - p) \tag{3.26}$$

We cannot compute the running time of the `mergesort` function using the same direct analysis, but we can assume its running time is $T(n)$, where $n = r - p$ and n is the size of the input data to be sorted and also the difference between its two arguments p and r. We can express this running time in terms of its components:

- It calls itself twice on half of the input data, $2T(n/2)$
- It calls the merge once on the entire data, $\Theta(n)$

We can summarize this into

$$T(n) = 2T(n/2) + n \tag{3.27}$$

This is called a *recurrence relation*. We turned the problem of computing the running time of the algorithm into the problem of solving the recurrence relation. This is now a math problem.

Some recurrence relations can be difficult to solve, but most of them follow in one of these categories:

$$T(n) = aT(n-b) + \Theta(f(n)) \Rightarrow T \in \Theta(max(a^n, nf(n))) \tag{3.28}$$
$$T(n) = T(b) + T(n-b-a) + \Theta(f(n)) \Rightarrow T \in \Theta(nf(n)) \tag{3.29}$$
$$T(n) = aT(n/b) + \Theta(n^m) \wedge a < b^m \Rightarrow T \in \Theta(n^m) \tag{3.30}$$
$$T(n) = aT(n/b) + \Theta(n^m) \wedge a = b^m \Rightarrow T \in \Theta(n^m \log n) \tag{3.31}$$
$$T(n) = aT(n/b) + \Theta(n^m) \wedge a > b^m \Rightarrow T \in \Theta(n^{\log_b a}) \tag{3.32}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \wedge a < b^m \Rightarrow T \in \Theta(n^m \log^p n) \tag{3.33}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \wedge a = b^m \Rightarrow T \in \Theta(n^m \log^{p+1} n) \tag{3.34}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \wedge a > b^m \Rightarrow T \in \Theta(n^{\log_b a}) \tag{3.35}$$
$$T(n) = aT(n/b) + \Theta(q^n) \Rightarrow T \in \Theta(q^n) \tag{3.36}$$
$$T(n) = aT(n/a - b) + \Theta(f(n)) \Rightarrow T \in \Theta(f(n)\log(n)) \tag{3.37}$$

(they work for $m \geq 0$, $p \geq 0$, and $q > 1$).

These results are a practical simplification of a theorem known as the *master theorem* [4].

3.2.1 Reducible recurrence relations

Other recurrence relations do not immediately fit one of the preceding patterns, but often they can be reduced (transformed) to fit.

Consider the following recurrence relation:

$$T(n) = 2T(\sqrt{n}) + \log n \tag{3.38}$$

We can replace n with $e^k = n$ in eq. (3.38) and obtain

$$T(e^k) = 2T(e^{k/2}) + k \tag{3.39}$$

If we also replace $T(e^k)$ with $S(k) = T(e^k)$, we obtain

$$\underbrace{S(k)}_{T(e^k)} = 2\ \underbrace{S(k/2)}_{T(e^{k/2})} + k \tag{3.40}$$

so that we can now apply the master theorem to S. We obtain that $S(k) \in \Theta(k \log k)$. Once we have the order of growth of S, we can determine the order of growth of $T(n)$ by substitution:

$$T(n) = S(\log n) \in \Theta(\underbrace{\log n}_{k} \log \underbrace{\log n}_{k}) \tag{3.41}$$

Note that there are recurrence relations that cannot be solved with any of the methods described.

Here are some examples of recursive algorithms and their corresponding recurrence relations with solution:

```
proc factorial1(n: int): int =
  if n == 0: 1 else: n * factorial1(n-1)
```

$$T(n) = T(n-1) + 1 \Rightarrow T(n) \in \Theta(n) \Rightarrow \texttt{factorial1} \in \Theta(n) \tag{3.42}$$

```
proc recursive0(n: int): int =
  if n == 0: 1
  else:
    loop3(n)
    n * n * recursive0(n-1)
```

$$T(n) = T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(n^2) \Rightarrow \texttt{recursive0} \in \Theta(n^3) \tag{3.43}$$

```
proc recursive1(n: int): int =
  if n == 0: 1
  else:
    loop3(n)
    n * recursive1(n-1) * recursive1(n-1)
```

$$T(n) = 2T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(2^n) \Rightarrow \texttt{recursive1} \in \Theta(2^n) \tag{3.44}$$

```
proc recursive2(n: int): int =
  if n == 0: 1
  else:
    let a = factorial0(n)
    a * recursive2(n div 2) * recursive1(n div 2)
```

$$T(n) = 2T(n/2) + P_1(n) \Rightarrow T(n) \in \Theta(n \log n) \Rightarrow \texttt{recursive2} \in \Theta(n \log n) \tag{3.45}$$

One example of practical interest for us is the binary search below. It finds the location of the element in a sorted input array A:

```
proc binarySearch*[T](a: openArray[T], element: T): int =
  ## Returns the index of `element` in sorted `a`, or -1 if not found.
  var lo = 0
  var hi = a.len - 1
  while hi >= lo:
    let x = (lo + hi) div 2
    if a[x] < element: lo = x + 1
    elif a[x] > element: hi = x - 1
    else: return x
  return -1
```

Notice that this algorithm does not appear to be recursive, but in practice, it is because of the apparently infinite while loop. The content of the while loop runs in constant time and then loops again on a problem of half of the original size:

$$T(n) = T(n/2) + 1 \Rightarrow \texttt{binary\_search} \in \Theta(\log n) \tag{3.46}$$

The idea of the `binary_search` is used in the bisection method for solving nonlinear equations.

Do not confuse `T` notation with Θ notation:

Algorithm	Recurrence Relationship	Running time
Binary Search	$T(n) = T(\frac{n}{2}) + \Theta(1)$	$\Theta(log(n))$
Binary Tree Traversal	$T(n) = 2T(\frac{n}{2}) + \Theta(1)$	$\Theta(n)$
Optimal Sorted Matrix Search	$T(n) = 2T(\frac{n}{2}) + \Theta(log(n))$	$\Theta(n)$
Merge Sort	$T(n) = T(\frac{n}{2}) + \Theta(n)$	$\Theta(nlog(n))$

The theta notation can also be used to describe the memory used by an algorithm as a function of the input, T_{memory}, as well as its running time.

3.3 Types of algorithms

Divide-and-conquer is a method of designing algorithms that (informally) proceeds as follows: given an instance of the problem to be solved, split this into several, smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions to yield a solution for the original instance. This description raises the question, by what methods are the sub-instances to be independently solved? The answer to this question is central to the concept of the divide-and-conquer algorithm and is a key factor in gauging their efficiency. The solution is unique for each problem.

The merge sort algorithm of the previous section is an example of a divide-and-conquer algorithm. In the merge sort, we sort an array by dividing it into two arrays and recursively sorting (conquering) each of the smaller arrays.

Most divide-and-conquer algorithms are recursive, although this is not a requirement.

Dynamic programming is a paradigm that is most often applied in the construction of algorithms to solve a certain class of optimization problems, that is, problems that require the minimization or maximization of some measure. One disadvantage of using divide-and-conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly because identical sub-instances may arise. For example, if you are computing the path between two nodes in a graph, some portions of multiple paths will follow the same last few hops. Why compute the last few hops for every path when you would get the same result every time?

The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice. The method usually accomplishes this by maintaining a table of sub-instance results. We say that dynamic programming is a bottom-up technique in which the smallest sub-instances are explicitly solved first and the results of these are used to construct solutions to progressively larger sub-instances. In

contrast, we say that the divide-and-conquer is a top-down technique.

We can refactor the `mergesort` algorithm to eliminate recursion in the algorithm implementation, while keeping the logic of the algorithm unchanged. Here is a possible implementation:

```
proc mergesortNonrecursive*[T](a: var seq[T]) =
  let n = a.len
  var blocksize = 1
  while blocksize < n:
    var p = 0
    while p < n:
      let q = p + blocksize
      let r = min(q + blocksize, n)
      if r > q:
        merge(a, p, q, r)
      p += 2 * blocksize
    blocksize *= 2
```

Notice that this has the same running time as the original `mergesort` because, although it is not recursive, it performs the same operations:

$$T_{best} \in \Theta(n \log n) \tag{3.47}$$

$$T_{average} \in \Theta(n \log n) \tag{3.48}$$

$$T_{worst} \in \Theta(n \log n) \tag{3.49}$$

$$T_{memory} \in \Theta(1) \tag{3.50}$$

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. This "take what you can get now" strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the global optimum. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a suboptimal solution. If the best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer. Even for problems that can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a nontrivial process.

For example, computing change for a purchase in a store is a good case of a greedy algorithm. Assume you need to give change back for a purchase. You would have three choices:

- Give the smallest denomination repeatedly until the correct amount is returned
- Give a random denomination repeatedly until you reach the correct amount. If a random choice exceeds the total, then pick another denomination until the correct amount is returned
- Give the largest denomination less than the amount to return repeatedly until the correct amount is returned

In this case, the third choice is the correct one.

Other types of algorithms do not fit into any of the preceding categories. One is, for example, backtracking. Backtracking is not covered in this course.

3.3.1 Memoization

One case of a top-down approach that is very general and falls under the umbrella of dynamic programming is called *memoization*. Memoization consists of allowing users to write algorithms using a naive divide-and-conquer approach, but functions that may be called more than once are modified so that their output is cached, and if they are called again with the same initial state, instead of the algorithm running again, the output is retrieved from the cache and returned without any computations.

Consider, for example, Fibonacci numbers:

$$\begin{aligned} \text{Fib}(0) &= 0 && (3.51)\\ \text{Fib}(1) &= 1 && (3.52)\\ \text{Fib}(n) &= \text{Fib}(n-1) + \text{Fib}(n-2) \text{ for } n > 1 && (3.53) \end{aligned}$$

which we can implement using divide-and-conquer as follows:

```
proc fib(n: int): int =
  if n < 2: n else: fib(n-1) + fib(n-2)
```

The recurrence relation for this algorithm is $T(n) = T(n-1) + T(n-2) + 1$, and its solution can be proven to be exponential. This is because this algorithm calls itself more than necessary with the same input values and keeps solving the same subproblem over and over.

We can write a generic `memoize` higher-order procedure that takes a recursive function (which receives the memoized closure as its first argument so it can recurse) and returns a cached version:

Listing 3.1: in file: `nlib/memoize.nim`

```
import std/tables

proc memoize*[K, V](f: proc(self: proc(x: K): V, x: K): V):
                    proc(x: K): V =
  var storage = initTable[K, V]()
  proc memoized(x: K): V =
    if x in storage: return storage[x]
    result = f(memoized, x)
    storage[x] = result
  return memoized
```

and applying it to the recursive function as follows:

```
let fib = memoize[int, int](
  proc(self: proc(x: int): int, n: int): int =
    if n < 2: n else: self(n-1) + self(n-2))
```

which we can call as

```
echo fib(11)
# 89
```

The recursive function takes the memoized closure as its first parameter, so when it calls itself it goes through the cache.

This makes the algorithm run much faster. Its running time goes from exponential to linear. Notice that the preceding `memoize` decorator is very general and can be used to decorate any other function.

One more direct dynamic programming approach consists in removing the recursion:

```
proc fib(n: int): int =
  if n < 2: return n
```

```
  var a = 0
  var b = 1
  for i in 1 ..< n:
    (a, b) = (b, a + b)
  return b
```

This also makes the algorithm linear and $T(n) \in \Theta(n)$.

Notice that we easily modify the memoization algorithm to store the partial results in a shared space, for example, on disk using the `PersistentDictionary`:

Listing 3.2: in file: `nlib/memoize.nim`

```
proc memoizePersistent*[K, V](f: proc(self: proc(x: K): V, x: K): V,
                              path = "memoize.json"):
                              proc(x: K): V =
  let storage = newPersistentDictionary(path)
  proc memoized(x: K): V =
    let key = $x
    if key in storage:
      return storage[key]
    result = f(memoized, x)
    storage[key] = result
  return memoized
```

We can use it as we did before, but we can now start and stop the program, and as long as it has access to the "memoize.json" file, the cache will persist across runs.

3.4 Timing algorithms

The order of growth is a theoretical concept. In practice, we need to time algorithms to check if findings are correct and, more important, to determine the magnitude of the constants in the T functions.

For example, consider this:

```
proc f1(n: int): int =
  for x in 0 ..< n: result += g1(x)

proc f2(n: int): int =
  for x in 0 ..< n * n: result += g2(x)
```

Since `f1` is $\Theta(n)$ and `f2` is $\Theta(n^2)$, we may be led to conclude that the latter is slower. It may very well be that `g1` is 10^6 smaller than `g2` and therefore

$T_{f1}(n) = c_1 n$, $T_{f2}(n) = c_2 n^2$, but if $c_1 = 10^6 c_2$, then $T_{f1}(n) > T_{f2}(n)$ when $n < 10^6$.

To time procedures in Nim, we can use this simple helper:

Listing 3.3: in file: `nlib/timer.nim`

```
import std/times

proc timef*(f: proc(), ns = 1000, dt = 60.0): float =
  let t0 = epochTime()
  var t = t0
  var k = 1
  while k < ns:
    f()
    t = epochTime()
    if t - t0 > dt: break
    inc k
  result = (t - t0) / float(k)
```

This function calls and averages the running time of `f()` for the minimum between `ns=1000` iterations and `dt=60` seconds.

It is now easy, for example, to time the fib function without memoize,

```
proc fib(n: int): int =
  if n < 2: n else: fib(n-1) + fib(n-2)
for k in 15 .. 19:
  echo k, " ", timef(proc() = discard fib(k))
# 15 0.000315684575338
# 16 0.000576375363706
# 17 0.000936052104732
# 18 0.00135168084153
# 19 0.00217730337912
```

and with memoize,

```
let fib = memoize[int, int](
  proc(self: proc(x: int): int, n: int): int =
    if n < 2: n else: self(n-1) + self(n-2))
for k in 15 .. 19:
  echo k, " ", timef(proc() = discard fib(k))
# 15 4.24022311802e-06
# 16 4.02901146386e-06
# 17 4.21922128122e-06
# 18 4.02495429084e-06
# 19 3.73784963552e-06
```

The former shows an exponential behavior; the latter does not.

3.5 Data structures

3.5.1 Arrays

An array is a data structure in which a series of numbers are stored contiguously in memory. The time to access each number (to read or write it) is constant. The time to remove, append, or insert an element may require moving the entire array to a more spacious memory location, and therefore, in the worst case, the time is proportional to the size of the array.

Arrays are the appropriate containers when the number of elements does not change often and when elements have to be accessed in random order.

3.5.2 List

A list is a data structure in which data are not stored contiguously, and each element has knowledge of the location of the next element (and perhaps of the previous element, in a doubly linked list). This means that accessing any element for (read and write) requires finding the element and therefore looping. In the worst case, the time to find an element is proportional to the size of the list. Once an element has been found, any operation on the element, including read, write, delete, and insert, before or after can be done in constant time.

Lists are the appropriate choice when the number of elements can vary often and when their elements are usually accessed sequentially via iterations.

A word of caution about terminology: in Nim, the type called `seq[T]` is *not* a linked list. Despite the name, a `seq[T]` is a contiguous, growable buffer of `T` stored on the heap, with the same memory layout and asymptotic behavior as a C++ `std::vector` or a Python `list`. Random access is therefore $O(1)$, while inserting or removing in the middle is $O(n)$.

True singly and doubly linked lists, with the access characteristics described at the beginning of this subsection, are provided by the standard library module `std/lists` as `SinglyLinkedList[T]`, `DoublyLinkedList[T]`, and

their ring (circular) variants `SinglyLinkedRing[T]` and `DoublyLinkedRing[T]`.

Internally, these types do *not* keep their elements next to each other in memory. Each element is wrapped in a small heap-allocated node that holds the value plus one or two pointers. In a singly linked list, the node holds a `next` pointer to the following node; in a doubly linked list, it holds both a `next` and a `prev` pointer. The list itself only stores a reference to the *head* node (and, for doubly linked lists, the *tail*), and traversal proceeds by following the chain of pointers:

```
# Conceptual layout of a doubly linked list:
#
#   head -> [ prev=nil | "A" | next ] <-> [ prev | "B" | next ] <-> ... <- tail
#
# The nodes are scattered across the heap; only the
# pointers stitch them together into a linear sequence.
```

This layout has a clear cost model: prepending or appending an element is $O(1)$, and so is inserting or deleting at a node whose address is already known (for example, because we kept the reference returned during a previous traversal). On the other hand, accessing the k-th element requires walking k pointers from the head, which is $O(k)$, and traversal is generally less cache-friendly than walking a contiguous `seq`.

The `std/lists` API mirrors this structure. The following example uses a `DoublyLinkedList[string]`:

```
import std/lists

var lst = initDoublyLinkedList[string]()
lst.append "B"        # @["B"]
lst.append "C"        # @["B", "C"]
lst.prepend "A"       # @["A", "B", "C"]

for x in lst.items:   # iterate from head to tail
  echo x              # A, B, C

# Find a node and insert/delete using its reference:
let n = lst.find("B") # returns the node holding "B" (or nil)
if n != nil:
  lst.remove n        # O(1) given the node reference
```

The singly linked variant `SinglyLinkedList[T]` has the same surface API, except that it only supports forward traversal and that `prepend` is $O(1)$

while `append` is $O(n)$ (since reaching the last node requires walking the chain). The ring variants behave like their list counterparts, except that the last node's `next` pointer wraps around to the head, so iteration is cyclic.

Throughout this book we use `seq[T]` as our default growable container, and we reach for `std/lists` only on the rare occasions when the linked-list semantics (constant-time insertion or removal at a known node, or splicing two lists together without copying) really matter.

3.5.3 Stack

A stack data structure is a container, and it is usually implemented as a list. It has the property that the first thing you can take out is the last thing put in. This is commonly known as last-in, first-out, or LIFO. The method to insert or add data to the container is called *push*, and the method to extract data is called *pop*.

In Nim, we can implement push by appending an item at the end of a `seq` (the standard library provides `.add`), and we can implement pop by removing the last element of the `seq` and returning it (`.pop`).

A simple stack example is as follows:

```
var stk: seq[string] = @[]
stk.add "One"
stk.add "Two"
echo stk.pop()    # "Two"
stk.add "Three"
echo stk.pop()    # "Three"
echo stk.pop()    # "One"
```

3.5.4 Queue

A queue data structure is similar to a stack but, whereas the stack returns the most recent item added, a queue returns the oldest item in the list. This is commonly called first-in, first-out, or FIFO. Using a `seq` as a queue, we can insert each new element at the front:

```
var que: seq[string] = @[]
que.insert("One", 0)
que.insert("Two", 0)
echo que.pop()    # "One"
```

```
que.insert("Three", 0)
echo que.pop()    # "Two"
echo que.pop()    # "Three"
```

Inserting at the front of a `seq` requires shifting all later elements, which is inefficient. Nim's `std/deques` module provides a double-ended queue with O(1) push/pop on both ends. The same method `addLast` appends; `popLast` removes from the back (stack behavior), while `popFirst` removes from the front (queue behavior):

```
import std/deques
var que = initDeque[string]()
que.addLast "One"
que.addLast "Two"
echo que.popFirst()    # "One"
que.addLast "Three"
echo que.popFirst()    # "Two"
echo que.popFirst()    # "Three"
```

3.5.5 Sorting

In the previous sections, we have seen the *insertion sort* and the *merge sort*. Here we consider, as examples, other sorting algorithms: the *quicksort* [3], the *randomized quicksort*, and the *counting sort*:

```
proc partition[T](a: var seq[T], i, j: int): int =
  let x = a[i]
  var h = i
  for k in i+1 ..< j:
    if a[k] < x:
      inc h
      swap(a[h], a[k])
  swap(a[h], a[i])
  return h

proc quicksort*[T](a: var seq[T], p = 0, r = -1) =
  let r = if r < 0: a.len else: r
  if p < r - 1:
    let q = partition(a, p, r)
    quicksort(a, p, q)
    quicksort(a, q + 1, r)
```

The running time of the quicksort is given by

$$T_{best} \in \Theta(n \log n) \tag{3.54}$$
$$T_{average} \in \Theta(n \log n) \tag{3.55}$$
$$T_{worst} \in \Theta(n^2) \tag{3.56}$$
$$\tag{3.57}$$

The quicksort can also be randomized by picking the pivot, `A[r]`, at random:

```
proc quicksort*[T](a: var seq[T], p = 0, r = -1) =
  let r = if r < 0: a.len else: r
  if p < r - 1:
    let q0 = rand(p .. r - 1)
    swap(a[p], a[q0])
    let q = partition(a, p, r)
    quicksort(a, p, q)
    quicksort(a, q + 1, r)
```

In this case, the best and the worst running times do not change, but the average improves when the input is already almost sorted.

The *counting sort* algorithm is special because it only works for arrays of positive integers. This extra requirement allows it to run faster than other sorting algorithms, under some conditions. In fact, this algorithm is linear in the range span by the elements of the input array.

Here is a possible implementation:

```
proc countingsort*(a: var seq[int]) =
  if a.len == 0: return
  var lo = a[0]
  var hi = a[0]
  for v in a:
    if v < lo: lo = v
    if v > hi: hi = v
  if lo < 0:
    raise newException(ValueError, "countingsort requires non-negative ints")
  let k = hi + 1
  var c = newSeq[int](k)
  for v in a: inc c[v]
  var i = 0
  for j in 0 ..< k:
    while c[j] > 0:
      a[i] = j
```

```
17      dec c[j]
18      inc i
```

If we define $k = max(A) - min(A) + 1$ and $n = len(A)$, we see

$$T_{best} \in \Theta(k+n) \tag{3.58}$$
$$T_{average} \in \Theta(k+n) \tag{3.59}$$
$$T_{worst} \in \Theta(k+n) \tag{3.60}$$
$$T_{memory} \in \Theta(k) \tag{3.61}$$

Notice that here we have also computed T_{memory}, for example, the order of growth of memory (not of time) as a function of the input size. In fact, this algorithm differs from the previous ones because it requires a temporary array C.

3.6 Tree algorithms

3.6.1 Heapsort and priority queues

Consider a *complete binary tree* as the one in the following figure:

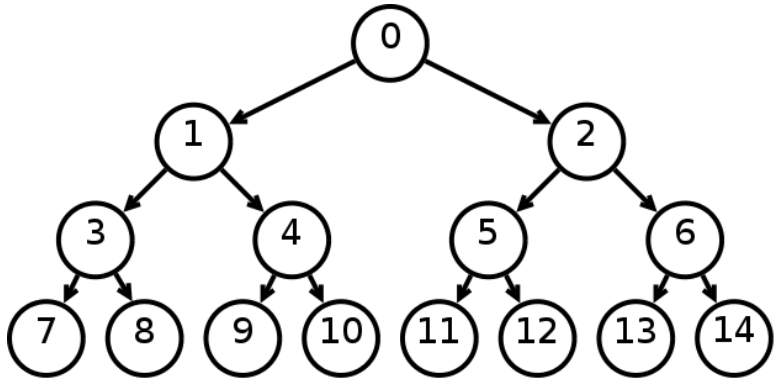

Figure 3.1: Example of a heap data structure. The number represents not the data in the heap but the numbering of the nodes.

It starts from the top node, called the *root*. Each node has zero, one, or two children. It is called complete because nodes have been added from top to bottom and left to right, filling available slots. We can think of each

level of the tree as a generation, where the older generation consists of one node, the next generation of two, the next of four, and so on. We can also number nodes from top to bottom and left to right, as in the image. This allows us to map the elements of a complete binary tree into the elements of an array.

We can implement a complete binary tree using a list, and the child–parent relations are given by the following formulas:

```
proc heapParent(i: int): int = (i - 1) div 2
proc heapLeftChild(i: int): int = 2 * i + 1
proc heapRightChild(i: int): int = 2 * i + 2
```

We can store data (e.g., numbers) in the nodes (or in the corresponding array). If the data are stored in such a way that the value at one node is always greater or equal than the value at its children, the array is called a *heap* and also a *priority queue*.

First of all, we need an algorithm to convert a list into a heap:

```
proc heapifyOne*[T](a: var seq[T], i: int, heapsize = -1) =
  let heapsize = if heapsize < 0: a.len else: heapsize
  let left = 2 * i + 1
  let right = 2 * i + 2
  var largest = i
  if left < heapsize and a[left] > a[largest]:
    largest = left
  if right < heapsize and a[right] > a[largest]:
    largest = right
  if largest != i:
    swap(a[i], a[largest])
    heapifyOne(a, largest, heapsize)

proc heapify*[T](a: var seq[T]) =
  for i in countdown(a.len div 2 - 1, 0):
    heapifyOne(a, i)
```

Now we can call `build_heap` on any array or list and turn it into a heap. Because the first element is by definition the smallest, we can use the heap to sort numbers in three steps:

- We turn the array into a heap
- We extract the largest element
- We apply recursion by sorting the remaining elements

Instead of using the preceding divide-and-conquer approach, it is better to use a dynamic programming approach. When we extract the largest element, we swap it with the last element of the array and make the heap one element shorter. The new, shorter heap does not need a full `build_heap` step because the only element out of order is the root node. We can fix this by a single call to `heapify`.

This is a possible implementation for the `heapsort` [5]:

```
proc heapsort*[T](a: var seq[T]) =
  heapify(a)
  for i in countdown(a.len - 1, 1):
    swap(a[0], a[i])
    heapifyOne(a, 0, i)
```

In the average and worst cases, it runs as fast as the quicksort, but in the best case, it is linear:

$$T_{best} \in \Theta(n) \tag{3.62}$$
$$T_{average} \in \Theta(n \log n) \tag{3.63}$$
$$T_{worst} \in \Theta(n \log n) \tag{3.64}$$
$$T_{memory} \in \Theta(1) \tag{3.65}$$

A heap can be used to implement a priority queue, for example, storage from which we can efficiently extract the largest element.

All we need is a function that allows extracting the root element from a heap (as we did in the `heapsort` and `heapify` of the remaining data) and a function to push a new value into the heap:

```
proc heapPop*[T](a: var seq[T]): T =
  if a.len < 1:
    raise newException(IndexDefect, "Heap Underflow")
  result = a[0]
  a[0] = a[^1]
  a.setLen(a.len - 1)
  if a.len > 0:
    heapifyOne(a, 0)

proc heapPush*[T](a: var seq[T], value: T) =
  a.add value
```

```
  var i = a.len - 1
  while i > 0:
    let j = heapParent(i)
    if a[j] < a[i]:
      swap(a[i], a[j])
      i = j
    else:
      break
```

The running times for `heap_pop` and `heap_push` are the same:

$$T_{best} \in \Theta(1) \tag{3.66}$$
$$T_{average} \in \Theta(\log n) \tag{3.67}$$
$$T_{worst} \in \Theta(\log n) \tag{3.68}$$
$$T_{memory} \in \Theta(1) \tag{3.69}$$

Here is an example:

```
let a = @[6, 2, 7, 9, 3]
var heap: seq[int] = @[]
for element in a: heapPush(heap, element)
while heap.len > 0: echo heapPop(heap)
# 9
# 7
# 6
# 3
# 2
```

Heaps find application in many numerical algorithms. Nim's standard library provides `std/heapqueue`, which offers similar functionality, except that `HeapQueue` is a min-heap (pops the smallest element) while we defined a max-heap above:

```
import std/heapqueue
let a = @[6, 2, 7, 9, 3]
var heap = initHeapQueue[int]()
for element in a: heap.push(element)
while heap.len > 0: echo heap.pop()
# 2
# 3
# 6
# 7
# 9
```

Notice that the order of `pop` results is reversed because `HeapQueue` is a min-

heap.

3.6.2 Binary search trees

A binary tree is a tree in which each node has at most two children (left and right). A binary tree is called a *binary search tree* if the value of a node is always greater than or equal to the value of its left child and less than or equal to the value of its right child.

A binary search tree is a kind of storage that can efficiently be used for searching if a particular value is in the storage. In fact, if the value for which we are looking is less than the value of the root node, we only have to search the left branch of the tree, and if the value is greater, we only have to search the right branch. Using divide-and-conquer, searching each branch of the tree is even simpler than searching the entire tree because it is also a tree, but smaller.

This means that we can search simply by traversing the tree from top to bottom along some path down the tree. We choose the path by moving down and turning left or right at each node, until we find the element for which we are looking or we find the end of the tree. We can search $T(d)$, where d is the depth of the tree. We will see later that it is possible to build binary trees where $d = \log n$.

To implement it, we need to have a class to represent a binary tree:

```
type
  BinarySearchTree*[K, V] = ref object
    key*: K
    value*: V
    hasKey*: bool
    left*, right*: BinarySearchTree[K, V]

proc newBinarySearchTree*[K, V](): BinarySearchTree[K, V] =
  BinarySearchTree[K, V]()

proc `[]=`*[K, V](t: BinarySearchTree[K, V], key: K, value: V) =
  if not t.hasKey:
    t.key = key; t.value = value; t.hasKey = true
  elif key == t.key:
    t.value = value
  elif key < t.key:
```

```
    if t.left.isNil: t.left = newBinarySearchTree[K, V]()
    t.left[key] = value
  else:
    if t.right.isNil: t.right = newBinarySearchTree[K, V]()
    t.right[key] = value

proc `[]`*[K, V](t: BinarySearchTree[K, V], key: K): V =
  if not t.hasKey: return default(V)
  if key == t.key: return t.value
  elif key < t.key and not t.left.isNil: return t.left[key]
  elif key > t.key and not t.right.isNil: return t.right[key]
  else: return default(V)

proc min*[K, V](t: BinarySearchTree[K, V]): (K, V) =
  var node = t
  while not node.left.isNil:
    node = node.left
  (node.key, node.value)

proc max*[K, V](t: BinarySearchTree[K, V]): (K, V) =
  var node = t
  while not node.right.isNil:
    node = node.right
  (node.key, node.value)
```

The binary tree can be used as follows:

```
let root = newBinarySearchTree[int, string]()
root[5] = "aaa"
root[3] = "bbb"
root[8] = "ccc"
echo root.left.key      # 3
echo root.left.value    # "bbb"
echo root[3]            # "bbb"
echo root.max()         # (8, "ccc")
```

3.6.3 Other types of trees

There are many other types of trees.

For example, AVL trees are binary search trees that are rebalanced after each insertion or deletion. They are rebalanced in such a way that for each node, the height of the left subtree minus the height of the right subtree is more or less the same. The rebalance operation can be done in $O(\log n)$.

For an AVL tree, the time for inserting or removing an element is given by

$$T_{best} \in \Theta(1) \tag{3.70}$$
$$T_{average} \in \Theta(\log n) \tag{3.71}$$
$$T_{worst} \in \Theta(\log n) \tag{3.72}$$
$$\tag{3.73}$$

Until now, we have considered binary trees (each node has two children and stores one value). We can generalize this to k trees, for which each node has k children and stores more than one value.

B-trees are a type of k-tree optimized to read and write large blocks of data. They are normally used to implement database indices and are designed to minimize the amount of data to move when the tree is rebalanced.

3.7 Graph algorithms

A *graph* G is a set of *vertices* V and a set of *links* (also called *edges*) connecting those vertices E. Each link connects one vertex to another.

As an example, you can think of a set of cities connected by roads. The cities are the vertices and the roads are the links.

A link may have attributes. In the case of a road, it could be the name of the road or its length.

In general, a link, indicated with the notation e_{ij}, connecting vertex i with vertex j is called a *directed link*. If the link has no direction $e_{ij} = e_{ji}$, it is called an undirected link. A graph that contains only undirected links is an *undirected graph*; otherwise, it is a *directed graph*.

In the road analogy, some roads can be "one way" (directed links) and some can be "two way" (undirected links).

A *walk* is an alternating sequence of vertices and links, with each link being incident to the vertices immediately preceding and succeeding it in the sequence. A *trail* is a walk with no repeated links.

A *path* is a walk with no repeated vertices. A walk is closed if the initial vertex is also the terminal vertex.

A *cycle* is a closed trail with at least one edge and with no repeated vertices, except that the initial vertex is also the terminal vertex.

A graph that contains no cycles is an *acyclic graph*. Any connected acyclic undirected graph is also a *tree*.

A *loop* is a one-link path connecting a vertex with itself.

A non null graph is *connected* if, for every pair of vertices, there is a walk whose ends are the given vertices. Let us write $i \tilde{} j$ if there is a path from i to j. Then $\tilde{}$ is an equivalence relation. The equivalence classes under $\tilde{}$ are the vertex sets of the connected components of G. A connected graph is therefore a graph with exactly one connected component.

A graph is called *complete* when every pair of vertices is connected by a link (or edge).

A *clique* of a graph is a subset of vertices in which every pair is an edge.

The *degree* of a vertex of a graph is the number of edges incident to it.

If i and j are vertices, the *distance* from i to j, written d_{ij}, is the minimum length of any path from i to j. In a connected undirected graph, the length of links induces a metric because for every two vertices, we can define their distance as the length of the shortest path connecting them.

The *eccentricity*, $e(i)$, of the vertex i is the maximum value of d_{ij}, where j is allowed to range over all of the vertices of the graph. This gives the largest shortest distance to any connected node in the graph.

The *subgraph* of G induced by a subset W of its vertices V ($W \subseteq V$) is the graph formed by the vertices in W and all edges whose two endpoints are in W.

The graph is the more complex of the data structures considered so far because it includes the tree as a particular case (yes, a tree is also a graph, but in general, a graph is not a tree), and the tree includes a list as a particular case (yes, a list is a tree in which every node has no more than

one child); therefore a list is also a particular case of a graph.

The graph is such a general data structure that it can be used to model the brain. Think of neurons as vertices and synapses as links connecting them. We push this analogy later by implementing a simple neural network simulator.

In what follows, we represent a graph in the following way, where links are edges:

```
let vertices = @["A", "B", "C", "D", "E"]
let links = @[(0, 1), (1, 2), (1, 3), (2, 5), (3, 4), (3, 2)]
let graph = (vertices, links)
```

Vertices are stored in a list or array and so are links. Each link is a tuple containing the ID of the source vertex, the ID of the target vertex, and perhaps optional parameters. Optional parameters are discussed later, but for now, they may include link details such as length, speed, reliability, or billing rate.

3.7.1 Breadth-first search

The breadth-first search [6] (BFS) is an algorithm designed to visit all vertices in a connected graph. In the cities analogy, we are looking for a travel strategy to make sure we visit every city reachable by roads, once and only once.

The algorithm begins at one vertex, the origin, and expands out, eventually visiting each node in the graph that is somehow connected to the origin vertex. Its main feature is that it explores the neighbors of the current vertex before moving on to explore remote vertices and their neighbors. It visits other vertices in the same order in which they are discovered.

The algorithm starts by building a table of neighbors so that for each vertex, it knows which other vertices it is connected to. It then maintains two lists, a list of blacknodes (defined as vertices that have been visited) and graynodes (defined as vertices that have been discovered because the algorithm has visited its neighbor). It returns a list of blacknodes in the order in which they have been visited.

Here is the algorithm:

Listing 3.4: in file: nlib/graph.nim

```
proc breadthFirstSearch*[V](
    graph: tuple[vertices: seq[V], links: seq[(int, int)]],
    start: int): seq[int] =
  var blacknodes: seq[int] = @[]
  var graynodes = @[start]
  var neighbors = newSeq[seq[int]](graph.vertices.len)
  for link in graph.links:
    neighbors[link[0]].add link[1]
  while graynodes.len > 0:
    let current = graynodes.pop()
    for neighbor in neighbors[current]:
      if neighbor notin blacknodes and neighbor notin graynodes:
        graynodes.insert(neighbor, 0)
    blacknodes.add current
  result = blacknodes
```

The BFS algorithm scales as follows:

$$T_{best} \in \Theta(n_E + n_V) \tag{3.74}$$

$$T_{average} \in \Theta(n_E + n_V) \tag{3.75}$$

$$T_{worst} \in \Theta(n_E + n_V) \tag{3.76}$$

$$T_{memory} \in \Theta(n) \tag{3.77}$$

3.7.2 Depth-first search

The depth-first search [7] (DFS) algorithm is very similar to the BFS, but it takes the opposite approach and explores as far as possible along each branch before backtracking.

In the cities analogy, if the BFS was exploring cities in the neighborhood before moving farther away, the DFS does the opposite and brings us first to distant places before visiting other nearby cities.

Here is a possible implementation:

Listing 3.5: in file: nlib/graph.nim

```
proc depthFirstSearch*[V](
    graph: tuple[vertices: seq[V], links: seq[(int, int)]],
    start: int): seq[int] =
  var blacknodes: seq[int] = @[]
```

```
5  var graynodes = @[start]
6  var neighbors = newSeq[seq[int]](graph.vertices.len)
7  for link in graph.links:
8    neighbors[link[0]].add link[1]
9  while graynodes.len > 0:
10   let current = graynodes.pop()
11   for neighbor in neighbors[current]:
12     if neighbor notin blacknodes and neighbor notin graynodes:
13       graynodes.add neighbor
14   blacknodes.add current
15 result = blacknodes
```

Notice that the BFS and the DFS differ by a single line (line 13 in the listings above), which determines whether graynodes is a queue (BFS) or a stack (DFS). When graynodes is a queue, the first vertex discovered is the first visited. When it is a stack, the last vertex discovered is the first visited.

The DFS algorithm goes as follows:

$$T_{best} \in \Theta(n_E + n_V) \tag{3.78}$$
$$T_{average} \in \Theta(n_E + n_V) \tag{3.79}$$
$$T_{worst} \in \Theta(n_E + n_V) \tag{3.80}$$
$$T_{memory} \in \Theta(1) \tag{3.81}$$

3.7.3 Disjoint sets

This is a data structure that can be used to store a set of sets and implements efficiently the join operation between sets. Each element of a set is identified by a representative element. The algorithm starts by placing each element in a set of its own, so there are n initial disjoint sets. Each is represented by itself. When two sets are joined, the representative element of the latter is made to point to the representative element of the former. The set of sets is stored as an array of integers. If at position i the array stores a negative number, this number is interpreted as being the representative element of its own set. If the number stored at position i is instead a nonnegative number j, it means that it belongs to a set that was joined with the set containing j.

Here is the implementation:

Listing 3.6: in file: `nlib/graph.nim`

```
type
  DisjointSets* = ref object
    sets*: seq[int]
    counter*: int

proc newDisjointSets*(n: int): DisjointSets =
  result = DisjointSets(sets: newSeq[int](n), counter: n)
  for i in 0 ..< n: result.sets[i] = -1

proc parent*(d: DisjointSets, i: int): int =
  var i = i
  while true:
    let j = d.sets[i]
    if j < 0: return i
    i = j

proc join*(d: DisjointSets, i, j: int): bool =
  let pi = d.parent(i)
  let pj = d.parent(j)
  if pi != pj:
    d.sets[pi] += d.sets[pj]
    d.sets[pj] = pi
    dec d.counter
    return true   # they have been joined
  return false    # they were already joined

proc joined*(d: DisjointSets, i, j: int): bool =
  d.parent(i) == d.parent(j)

proc len*(d: DisjointSets): int = d.counter
```

Notice that we added a member variable counter that is initialized to the number of disjoint sets and is decreased by one every time two sets are merged. This allows us to keep track of how many disjoint sets exist at each time. We also added a `len` procedure so that we can check the value of the counter on a `DisjointSet`.

As an example of application, here is a code that builds a n^d maze. It may be easier to picture it with $d = 2$, a two-dimensional maze. The algorithm works by assuming there is a wall connecting any couple of two adjacent cells. It labels the cells using an integer index. It puts all the cells into a `DisjointSets` data structure and then keeps tearing down walls at random. Two cells on the maze belong to the same set if they are connected, for example, if there is a path that connects them. At the beginning, each

cell is its own set because it is isolated by walls. Walls are torn down by being removed from the list `wall` if the wall was separating two disjoint sets of cells. Walls are torn down until all cells belong to the same set, for example, there is a path connecting any cell to any cell:

Listing 3.7: in file: `nlib/graph.nim`

```
proc makeMaze*(n, d: int): tuple[walls, tornDownWalls: seq[(int, int)]] =
  var walls: seq[(int, int)] = @[]
  let total = n ^ d
  for i in 0 ..< n * n:
    for j in 0 ..< d:
      if (i div (n ^ j)) mod n + 1 < n:
        walls.add (i, i + n ^ j)
  var tornDownWalls: seq[(int, int)] = @[]
  let ds = newDisjointSets(total)
  shuffle(walls)
  for wall in walls:
    if ds.join(wall[0], wall[1]):
      tornDownWalls.add wall
    if ds.len == 1:
      break
  var remaining: seq[(int, int)] = @[]
  for wall in walls:
    if wall notin tornDownWalls:
      remaining.add wall
  (remaining, tornDownWalls)
```

Here is an example of how to use it. This example also draws the walls and the border of the maze:

```
let (walls, tornDownWalls) = makeMaze(n = 20, d = 2)
```

The following figure shows a representation of a generated maze:

3.7.4 Minimum spanning tree: Kruskal

Given a connected graph with weighted links (links with a weight or length), a *minimum spanning tree* is a subset of that graph that connects all vertices of the original graph, and the sum of the link weights is minimal. This subgraph is also a tree because the condition of minimal weight implies that there is only one path connecting each couple of vertices.

One algorithm to build the minimal spanning tree of a graph is the Kruskal [8] algorithm. It works by placing all vertices in a `DisjointSets`

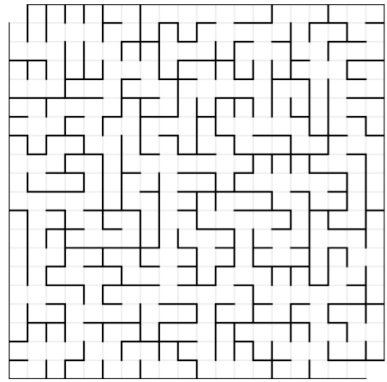

Figure 3.2: Example of a maze as generated using the `DisjointSets` algorithm.

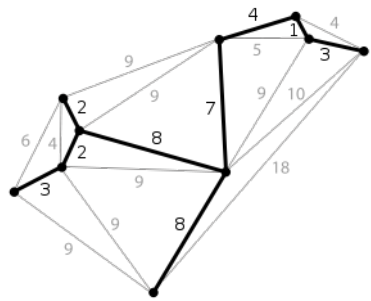

Figure 3.3: Example of a minimum spanning tree subgraph of a larger graph. The numbers on the links indicate their weight or length.

structure and looping over links in order of their weight. If the link connects two vertices belonging to different sets, the link is selected to be part of the minimum spanning tree, and the two sets are joined, else the link is ignored. The Kruskal algorithm assumes an undirected graph, for example, all links are bidirectional, and the weight of a link is the same in both directions:

Listing 3.8: in file: `nlib/graph.nim`

```
import std/algorithm

proc kruskal*[V](
    graph: tuple[vertices: seq[V],
                 links: seq[(int, int, float)]]
    ): seq[(int, int, float)] =
```

```
7   var links = graph.links
8   links.sort(proc(a, b: (int, int, float)): int = cmp(a[2], b[2]))
9   let s = newDisjointSets(graph.vertices.len)
10  for link in links:
11    let (source, dest, length) = link
12    if s.join(source, dest):
13      result.add (source, dest, length)
```

The `sort` procedure used above comes from `std/algorithm` (imported on the first line of the listing) and expects a comparator: a procedure that returns a negative integer, zero, or a positive integer to indicate that its first argument is less than, equal to, or greater than its second. The `cmp` procedure used inside the comparator is a built-in from Nim's `system` module (automatically imported in every program) and provides exactly that three-way comparison for any ordered type — here, the `float` weight stored in the third position of each link tuple.

The Kruskal algorithm goes as follows:

$$T_{worst} \in \Theta(n_E \log n_V) \tag{3.82}$$

$$T_{memory} \in \Theta(n_E) \tag{3.83}$$

We provide an example of application in the next subsection.

3.7.5 Minimum spanning tree: Prim

The Prim [9] algorithm solves the same problem as the Kruskal algorithm, but the Prim algorithm works on a directed graph. It works by placing all vertices in a minimum priority queue where the queue metric for each vertex is the length, or weighted value, of a link connecting the vertex to the closest known neighbor vertex. At each iteration, the algorithm pops a vertex from the priority queue, loops over its neighbors (adjacent links), and, if it finds that one of its neighbors is already in the queue and it is possible to connect it to the current vertex using a shorter link than the one connecting the neighbor to its current closest vertex, the neighbor information is then updated. The algorithm loops until there are no vertices in the priority queue.

The Prim algorithm also differs from the Kruskal algorithm because the

former needs a starting vertex, whereas the latter does not. The result when interpreted as a subgraph does not depend on the starting vertex:

Listing 3.9: in file: `nlib/graph.nim`

```
const PRIM_INFINITY* = 1e100

type
  PrimVertex* = ref object
    id*: int
    closest*: PrimVertex
    closestDist*: float
    neighbors*: seq[(int, float)]

proc newPrimVertex*(id: int, links: seq[(int, int, float)]): PrimVertex =
  result = PrimVertex(id: id, closest: nil, closestDist: PRIM_INFINITY)
  for link in links:
    if link[0] == id:
      result.neighbors.add (link[1], link[2])

proc `<`*(a, b: PrimVertex): bool = a.closestDist < b.closestDist

proc prim*[V](
    graph: tuple[vertices: seq[V], links: seq[(int, int, float)]],
    start: int): seq[(int, int, float)] =
  var p: seq[PrimVertex] = @[]
  for i in 0 ..< graph.vertices.len:
    p.add newPrimVertex(i, graph.links)
  var q: seq[PrimVertex] = @[]
  for i in 0 ..< graph.vertices.len:
    if i != start: q.add p[i]
  var vertex = p[start]
  while q.len > 0:
    for (neighborId, length) in vertex.neighbors:
      let neighbor = p[neighborId]
      if neighbor in q and length < neighbor.closestDist:
        neighbor.closest = vertex
        neighbor.closestDist = length
    q.sort(proc(a, b: PrimVertex): int = cmp(a.closestDist, b.closestDist))
    vertex = q[0]
    q.delete(0)
  for v in p:
    if v.id != start:
      result.add (v.id, v.closest.id, v.closestDist)
```

```
let vertices = toSeq(0 ..< 10)
var links: seq[(int, int, float)] = @[]
for i in vertices:
  for j in vertices:
```

```
    links.add (i, j, abs(sin(float(i + j + 1))))
let graph = (vertices, links)
let mst = prim(graph, 0)
for link in mst: echo link
# (1, 4, 0.279...)
# (2, 0, 0.141...)
# (3, 2, 0.279...)
# (4, 1, 0.279...)
# (5, 0, 0.279...)
# (6, 2, 0.412...)
# (7, 8, 0.287...)
# (8, 7, 0.287...)
# (9, 6, 0.287...)
```

The Prim algorithm, when using a priority queue for Q, goes as follows:

$$T_{worst} \in \Theta(n_E + n_V \log n_V) \tag{3.84}$$

$$T_{memory} \in \Theta(n_E) \tag{3.85}$$

One important application of the minimum spanning tree is in evolutionary biology. Consider, for example, the DNA for the genes that produce hemoglobin, a molecule responsible for the transport of oxygen in blood. This protein is present in every animal, and the gene is also present in the DNA of every known animal. Yet its DNA structure is a little different. One can select a pool of animals and, for each two of them, compute the similarity of the DNA of their hemoglobin genes using the `lcs` algorithm discussed later. One can then link each two animals by a metric that represents how similar the two animals are. We can then run the Prim or the Kruskal algorithm to find the minimum spanning tree. The tree represents the most likely evolutionary tree connecting those animal species. Actually, three genes are responsible for hemoglobin (*HBA1*, *HBA2*, and *HBB*). By performing the analysis on different genes and comparing the results, it is possible to establish a consistency check of the results. [10]

Similar studies are performed routinely in evolutionary biology. They can also be applied to viruses to understand how viruses evolved over time. [11]

3.7.6 Single-source shortest paths: Dijkstra

The Dijkstra [12] algorithm solves a similar problem to the Kruskal and Prim algorithms. Given a graph, it computes, for each vertex, the shortest path connecting the vertex to a starting (or source, or root) vertex. The collection of links on all the paths defines the *single-source shortest paths*.

It works, like Prim, by placing all vertices in a min priority queue where the queue metric for each vertex is the length of the path connecting the vertex to the source. At each iteration, the algorithm pops a vertex from the priority queue, loops over its neighbors (adjacent links), and, if it finds that one of its neighbors is already in the queue and it is possible to connect it to the current vertex using a link that makes the path to the source shorter, the neighbor information is updated. The algorithm loops until there are no more vertices in the priority queue.

The implementation of this algorithm is almost identical to the Prim algorithm, except for two lines:

Listing 3.10: in file: `nlib/graph.nim`

```
proc dijkstra*[V](
    graph: tuple[vertices: seq[V], links: seq[(int, int, float)]],
    start: int): seq[(int, int, float)] =
  var p: seq[PrimVertex] = @[]
  for i in 0 ..< graph.vertices.len:
    p.add newPrimVertex(i, graph.links)
  var q: seq[PrimVertex] = @[]
  for i in 0 ..< graph.vertices.len:
    if i != start: q.add p[i]
  var vertex = p[start]
  vertex.closestDist = 0
  while q.len > 0:
    for (neighborId, length) in vertex.neighbors:
      let neighbor = p[neighborId]
      let dist = length + vertex.closestDist
      if neighbor in q and dist < neighbor.closestDist:
        neighbor.closest = vertex
        neighbor.closestDist = dist
    q.sort(proc(a, b: PrimVertex): int = cmp(a.closestDist, b.closestDist))
    vertex = q[0]
    q.delete(0)
  for v in p:
    if v.id != start:
      result.add (v.id, v.closest.id, v.closestDist)
```

```
let vertices = toSeq(0 ..< 10)
var links: seq[(int, int, float)] = @[]
for i in vertices:
  for j in vertices:
    links.add (i, j, abs(sin(float(i + j + 1))))
let graph = (vertices, links)
let paths = dijkstra(graph, 0)
for link in paths: echo link
# (1, 2, 0.897...)
# (2, 0, 0.141...)
# (3, 2, 0.420...)
# (4, 2, 0.798...)
# (5, 0, 0.279...)
# (6, 2, 0.553...)
# (7, 2, 0.685...)
# (8, 0, 0.412...)
# (9, 0, 0.544...)
```

The Dijkstra algorithm goes as follows:

$$T_{worst} \in \Theta(n_E + n_V \log n_V) \tag{3.86}$$

$$T_{memory} \in \Theta(n_E) \tag{3.87}$$

An application of the Dijkstra is in solving a maze such as the one built when discussing disjoint sets. To use the Dijkstra algorithm, we need to generate a maze, take the links representing torn-down walls, and use them to build an undirected graph. This is done by symmetrizing the links (if i and j are connected, j and i are also connected) and adding to each link a length (1, because all links connect next-neighbor cells):

```
let (n, d) = (4, 2)
let (mazeWalls, tornDown) = makeMaze(n, d)
var symmetrizedLinks: seq[(int, int, float)] = @[]
for (i, j) in tornDown: symmetrizedLinks.add (i, j, 1.0)
for (i, j) in tornDown: symmetrizedLinks.add (j, i, 1.0)
let graph = (toSeq(0 ..< n * n), symmetrizedLinks)
let mst = dijkstra(graph, 0)
var paths = initTable[int, (int, float)]()
for (i, j, dd) in mst: paths[i] = (j, dd)
```

Given a maze cell `i`, `path[i]` gives us a tuple (j, d) where d is the number of steps for the shortest path to reach the origin (0) and j is the ID of the next cell along this path. The following figure shows a generated maze and a reconstructed path connecting an arbitrary cell to the origin:

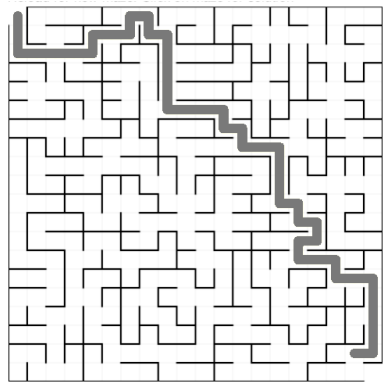

Figure 3.4: The result shows an application of the Dijkstra algorithm for the single source shortest path applied to solve a maze.

3.8 Greedy algorithms

3.8.1 Huffman encoding

The *Shannon–Fano encoding* [13][14] (also known as **minimal prefix code**) is a lossless data compression algorithm. In this encoding, each character in a string is mapped into a sequence of bits so characters that appear with less frequency are encoded with a longer sequence of bits, whereas characters that appear with more frequency are encoded with a shorter sequence.

The *Huffman encoding* [15] is an implementation of the Shannon–Fano encoding, but the sequence of bits into which each character is mapped is chosen such that the length of the compressed string is minimal. This choice is constructed in the following way. We associate a tree with each character in the string to compress. Each tree is a trivial tree containing only one node: the root node. We then associate with the root node the frequency of the character representing the tree. We then extract from the list of trees the two trees with rarest or lowest frequency: t1 and t2. We form a new tree, t3, we attach t1 and t2 to t3, and we associate a frequency with t3 equal to the sum of the frequencies of t1 and t2. We repeat this operation until the list of trees contains only one tree. At this point, we associate a sequence of bits with each node of the tree. Each bit corresponds

to one level on the tree. The more frequent characters end up being closer to the root and are encoded with a few bits, while rare characters are far from the root and encoded with more bits.

PKZIP, ARJ, ARC, JPEG, MPEG3 (mp3), MPEG4, and other compressed file formats all use the Huffman coding algorithm for compressing strings. Note that Huffman is a compression algorithm with no information loss. In the JPEG and MPEG compression algorithms, Huffman algorithms are combined with some form or cut of the Fourier spectrum (e.g., MP3 is an audio compression format in which frequencies below 2 KHz are dumped and not compressed because they are not audible). Therefore the JPEG and MPEG formats are referred to as compression with information loss.

Here is a possible implementation of Huffman encoding using the `std/heapqueue` module. For the priority-queue ordering to be correct we wrap each tree (the set of symbols it represents and their cumulative frequency) into a `HuffmanNode` record and overload the < operator on it.

Listing 3.11: in file: `nlib/compression.nim`

```
import std/heapqueue

type
  HuffmanTreeKind* = enum hkLeaf, hkNode
  HuffmanTree* = ref object
    frequency*: int
    case kind*: HuffmanTreeKind
    of hkLeaf: symbol*: char
    of hkNode: left*, right*: HuffmanTree

  HuffmanNode* = object
    frequency*: int
    tree*: HuffmanTree

proc `<`*(a, b: HuffmanNode): bool = a.frequency < b.frequency

proc inorderTreeWalk(t: HuffmanTree, key: string,
                     keys: var Table[char, string]) =
  case t.kind
  of hkLeaf: keys[t.symbol] = key
  of hkNode:
    inorderTreeWalk(t.left, key & "0", keys)
    inorderTreeWalk(t.right, key & "1", keys)

```

```nim
proc encodeHuffman*(input: string): (Table[char, string], string) =
  var symbols = initTable[char, int]()
  for s in input:
    symbols[s] = symbols.getOrDefault(s, 0) + 1
  var heap = initHeapQueue[HuffmanNode]()
  for k, f in symbols:
    heap.push(HuffmanNode(frequency: f,
              tree: HuffmanTree(kind: hkLeaf, frequency: f, symbol: k)))
  while heap.len > 1:
    let n1 = heap.pop()
    let n2 = heap.pop()
    let merged = HuffmanTree(kind: hkNode,
                             frequency: n1.frequency + n2.frequency,
                             left: n1.tree, right: n2.tree)
    heap.push(HuffmanNode(frequency: merged.frequency, tree: merged))
  var symbolMap = initTable[char, string]()
  inorderTreeWalk(heap[0].tree, "", symbolMap)
  var encoded = ""
  for s in input: encoded.add symbolMap[s]
  (symbolMap, encoded)

proc decodeHuffman*(keys: Table[char, string], encoded: string): string =
  var reversed = initTable[string, char]()
  for k, v in keys: reversed[v] = k
  var i = 0
  for j in 1 .. encoded.len:
    let chunk = encoded[i ..< j]
    if chunk in reversed:
      result.add reversed[chunk]
      i = j
```

We can use it as follows:

```nim
let input = "this is a nice day"
let (keys, encoded) = encodeHuffman(input)
echo encoded
# 00100101110011101100111011111000111100001111010000011110100
let decoded = decodeHuffman(keys, encoded)
echo decoded == input            # true
echo 1.0 * float(input.len) / (float(encoded.len) / 8.0)
# 2.44...
```

We managed to compress the original data by a factor 2.44.

We can ask how good is this compression factor. The maximum theoretical best compression factor is given by the Shannon *entropy*, defined as

$$E = -\sum_{u} w_i \log_2 w_i \tag{3.88}$$

where w_i is the relative frequency of each symbol. In our case, this is easy to compute as

```
import std/[math, sets]
let input = "this is a nice day"
var alphabet: HashSet[char]
for c in input: alphabet.incl c
var w: seq[float] = @[]
for c in alphabet:
  w.add float(input.count(c)) / float(input.len)
var entropy = 0.0
for wi in w: entropy -= wi * log2(wi)
echo entropy   # 3.23...
```

How could we have done better? Notice for example that the Huffman encoding does not take into account the order in which symbols appear. The original string contains the triple "is" twice, and we could have taken advantage of that pattern, but we did not.

Our choice of using characters as symbols is arbitrary. We could have used a couple of characters as symbols or triplets or any other subsequences of bytes of the original input. We could also have used symbols of different lengths for different parts of the input (we could have used a single symbol for "is"). A different choice would have given a different compression ratio, perhaps better, perhaps worse.

3.8.2 Longest common subsequence

Given two sequences of characters S_1 and S_2, this is the problem of determining the length of the longest common subsequence (LCS) that is a subsequence of both S_1 and S_2.

There are several applications for the LCS [16] algorithm:

- **Molecular biology**: DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub-molecules forming DNA. When biologists find a new sequence, they want to find similar sequences or ones that are close. One way of computing how similar two sequences are is to find the length of their LCS.

- **File comparison**: The Unix program `diff` is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a LCS of the lines of the two files and displays the set of lines that have changed. In this instance of the problem, we should think of each line of a file as being a single complicated character.
- **Spelling correction**: If some text contains a word, w, that is not in the dictionary, a "close" word (e.g., one with a small edit distance to w) may be suggested as a correction. Transposition errors are common in written text. A transposition can be treated as a deletion plus an insertion, but a simple variation on the algorithm can treat a transposition as a single point mutation.
- **Speech recognition**: Algorithms similar to the LCS are used in some speech recognition systems—find a close match between a new utterance and one in a library of classified utterances.

Let's start with some simple observations about the LCS problem. If we have two strings, say, "ATGGCACTACGAT" and "ATCGAGC, " we can represent a subsequence as a way of writing the two so that certain letters line up:

```
1    ATGGCACTACGAT
2    || | |   |
3    ATCG AG  C
```

From this we can observe the following simple fact: if the two strings start with the same letter, it's always safe to choose that starting letter as the first character of the subsequence. This is because, if you have some other subsequence, represented as a collection of lines as drawn here, you can "push" the leftmost line to the start of the two strings without causing any other crossings and get a representation of an equally long subsequence that does start this way.

Conversely, suppose that, like in the preceding example, the two first characters differ. Then it is not possible for both of them to be part of a common subsequence. There are three possible choices: remove the first letter from either one of the strings or remove the letter from both strings.

Finally, observe that once we've decided what to do with the first characters of the strings, the remaining subproblem is again a LCS problem on two shorter strings. Therefore we can solve it recursively. However, because we don't know which choice of the three to take, we will take them all and see which choice returns the best result.

Rather than finding the subsequence itself, it turns out to be more efficient to find the length of the longest subsequence. Then, in the case where the first characters differ, we can determine which subproblem gives the correct solution by solving both and taking the max of the resulting subsequence lengths. Once we turn this into a dynamic programming algorithm, we get the following:

Listing 3.12: in file: `nlib/text.nim`

```
proc lcs*(a, b: string): int =
  var previous = newSeq[int](b.len)
  var current = newSeq[int](b.len)
  for i, r in a:
    for j, c in b:
      var e: int
      if r == c:
        e = (if i * j > 0: previous[j-1] + 1 else: 1)
      else:
        let up = if i > 0: previous[j] else: 0
        let lf = if j > 0: current[j-1] else: 0
        e = max(up, lf)
      current[j] = e
    previous = current
  result = current[^1]
```

Here is an example:

```
let dna1 = "ATGCTTTAGAGGATGCGTAGATAGCTAAATAGCTCGCTAGA"
let dna2 = "GATAGGTACCACAATAATAAGGATAGCTCGCAAATCCTCGA"
echo lcs(dna1, dna2)   # 26
```

The algorithm can be shown to be $O(nm)$ (where m = len(a) and n = len(b)).

Another application of this algorithm is in the Unix `diff` utility. Here is a simple example to find the number of common lines between two files:

```
let a = readFile("file1.txt")
let b = readFile("file2.txt")
echo lcs(a, b)
```

3.8.3 Needleman–Wunsch

With some minor changes to the LCS algorithm, we obtain the Needleman–Wunsch algorithm [17], which solves the problem of *global sequence alignment*. The changes are that, instead of using only two alternating rows (`c` and `d` for storing the temporary results, we store all temporary results in an array `z`; when two matching symbols are found and they are not consecutive, we apply a penalty equal to p^m, where m is the distance between the two matches and is also the size of the gap in the matching subsequence:

Listing 3.13: in file: `nlib/text.nim`

```
proc needlemanWunsch*(a, b: string, p = 0.97): seq[seq[float]] =
  var z: seq[seq[float]] = @[]
  for i, r in a:
    var row: seq[float] = @[]
    for j, c in b:
      var e: float
      if r == c:
        e = if i * j > 0: z[i-1][j-1] + 1.0 else: 1.0
      else:
        let up = if i > 0: z[i-1][j] else: 0.0
        let lf = if j > 0: row[j-1] else: 0.0
        e = p * max(up, lf)
      row.add e
    z.add row
  result = z
```

This algorithm can be used to identify common subsequences of DNA between chromosomes (or in general common similar subsequences between any two strings of binary data). Here is an example in which we look for common genes in two randomly generated chromosomes:

```
let bases = "ATGC"
randomize()
proc randString(n: int): string =
  for _ in 0 ..< n: result.add bases[rand(bases.len - 1)]
var genes: seq[string] = @[]
for _ in 0 ..< 20: genes.add randString(10)
var chromosome1, chromosome2 = ""
for _ in 0 ..< 10:
  chromosome1.add genes[rand(genes.len - 1)]
  chromosome2.add genes[rand(genes.len - 1)]
let z = needlemanWunsch(chromosome1, chromosome2)
# `z` is a 2D scoring matrix; the helper `saveHeatmap` (defined
```

```
13 # alongside `savePlot` in nlib.nim) renders it to an image file.
```

The output of the algorithm is the following image:

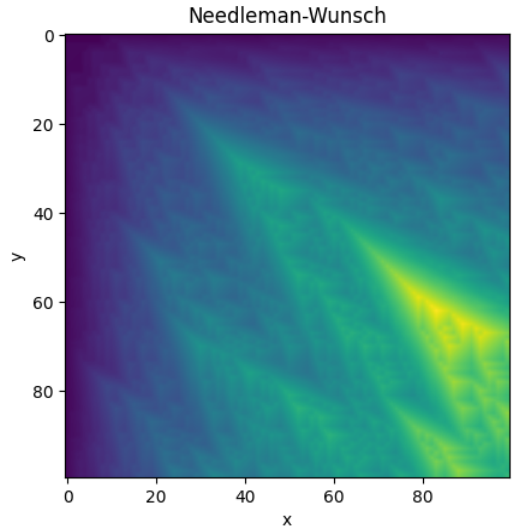

Figure 3.5: A Needleman and Wunsch plot sequence alignment. The arrow-like patterns indicate the point in the two sequences (represented by the X- and Y-coordinates) where the two sequences are more likely to align.

The arrow-like patterns in the figure correspond to locations where `chromosome1` (Y coordinate) and where `chromosome2` (X coordinate) have DNA in common. Those are the places where the sequences are more likely to be aligned for a more detailed comparison.

Optimized implementations of LCS and Needleman–Wunsch exist in many bioinformatics toolkits; in Nim, specialized packages wrap battle-tested C alignment tools through the foreign-function interface and expose comparable APIs.

3.8.4 Continuous Knapsack

Assume you want to fill your knapsack such that you will maximize the value of its contents [18]. However, you are limited by the volume your

knapsack can hold. In the continuous knapsack, the amount of each product can vary continuously. In the discrete one, each product has a finite size, and you either carry it or no.

The continuous knapsack problem can be formulated as the problem of maximizing

$$f(x) = a_0x_0 + a_1x_1 + ... + a_nx_n \tag{3.89}$$

given the constraint

$$b_0x_0 + b_1x_1 + ... + b_nx_n \leq c \tag{3.90}$$

where coefficients a_i, b_i, and c are provided and $x_i \in [0, 1]$ are to be determined.

Using financial terms, we can say that

- The set $\{x_0, x_1, ..., x_n\}$ forms a portfolio
- b_i is the cost of investment i
- c is the total investment capital available
- a_i is the expected return of investment for investment i
- $f(x)$ is the expected value of our portfolio $\{x_0, x_1, ..., x_n\}$

Here is the solving algorithm:

Listing 3.14: in file: `nlib/finance.nim`

```
proc continuumKnapsack*(a, b: seq[float], c: float):
                        (float, seq[(int, float)]) =
  var table: seq[(float, int)] = @[]
  for i in 0 ..< a.len: table.add (a[i] / b[i], i)
  table.sort(SortOrder.Descending)
  var f = 0.0
  var x: seq[(int, float)] = @[]
  var c = c
  for (y, i) in table:
    let quantity = min(c / b[i], 1.0)
    x.add (i, quantity)
    c -= b[i] * quantity
    f += a[i] * quantity
  (f, x)
```

This algorithm is dominated by the sort; therefore

$$T_{worst}(x) \in O(n \log n) \tag{3.91}$$

3.8.5 Discrete Knapsack

The discrete Knapsack problem is very similar to the continuous knapsack problem but $x_i \in \{0, 1\}$ (can only be 0 or 1).

Consider the jars of liquids replaced with baskets of objects, say, a basket each of gold bars, silver coins, copper beads, and Rolex watches. How many of each item do you take? The discrete knapsack problem does not consider "baskets of items" but rather all the items together. In this example, dump out all the baskets and you have individual objects to take. Which objects do you take, and which do you leave behind?

In this case, a greedy approach does not apply and the problem is, in general, NP complete. This concept is defined formally later but it means that there is no known algorithm that can solve this problem and that its order of growth is a polynomial. The best known algorithm has an exponential running time.

This kind of problem is unsolvable for large input.

If we assume that c and b_i are all multiples of a finite factor ε, then it is possible to solve the problem in $O(c/\varepsilon)$. Even when there is not a finite factor ε, we can always round c and b_i to some finite precision ε, and we can conclude that, for any finite precision ε, we can solve the problem in linear time. The algorithm that solves this problem follows a dynamic programming approach.

We can reformulate the problem in terms of a simple capital budgeting problem. We have to invest \$5M. We assume ε =\$1M. We are in contact with three investment firms. Each offers a number of investment opportunities characterized by an investment cost $c[i, j]$ and an expected return of investment $r[i, j]$. The index i labels the investment firm and the index j labels the different investment opportunities offered by the firm. We have to build a portfolio that maximizes the return of investment. We cannot select more than one investment for each firm, and we cannot select

fractions of investments.

Without loss of generality, we will assume that

$$c[i,j] \leq c[i,j+1] \text{ and } r[i,j] \leq r[i,j+1] \tag{3.92}$$

which means that investment opportunities for each firm are sorted according to their cost.

Consider the following explicit case:

	Firm	$i=0$	Firm	$i=1$	Firm	$i=2$
proposal	$c[0,j]$	$r[0,j]$	$c[1,j]$	$r[1,j]$	$c[2,j]$	$r[2,j]$
$j=0$	0	0	0	0	0	0
$j=1$	1	5	2	8	1	4
$j=2$	2	6	3	9	-	-
$j=3$	-	-	4	12	-	-

(Table 1)

(table values are always multiples of ε =\$1M).

Notice that we can label each possible portfolio by a triplet $\{j_0, j_1, j_2\}$.

A straightforward way to solve this is to try all possibilities and choose the best. In this case, there are only $3 \times 4 \times 2 = 24$ possible portfolios. Many of these are infeasible (e.g., portfolio $\{2,3,0\}$ costs \$6M and we cannot afford it). Other portfolios are feasible but very poor (like portfolio $\{0,0,1\}$, which is feasible but returns only \$4M).

Here are some disadvantages of total enumeration:

- For larger problems, the enumeration of all possible solutions may not be computationally feasible.
- Infeasible combinations may not be detectable a priori, leading to inefficiency.
- Information about previously investigated combinations is not used to eliminate inferior or infeasible combinations (unless we use memoization, but in this case the algorithm would grow polynomially in memory space).

We can, instead, use a dynamic programming approach.

We break the problem into three stages, and at each stage, we fill a table of optimal investments for each discrete amount of money. At each stage i, we only consider investments from firm i and the table during the previous stage.

So stage 0 represents the money allocated to firm 0, stage 1 the money to firm 1, and stage 2 the money to firm 2.

STAGE ZERO: we maximize the return of investment considering only offers from firm 0. We fill a table $f[0,k]$ with the maximum return of investment if we invest k million dollars in firm 0:

$$f[0,k] = \max_{j|c[0,j]<k} r[0,j] \tag{3.93}$$

k	$f[0,k]$
0	0
1	5
2^*	6^*
3	6
4	6
5	6

STAGE ONE: we maximize the return of investment considering offers from firm 1 and the prior table. We fill a table $f[1,k]$ with the maximum return of investment if we invest k million dollars in firm 0 and firm 1:

$$f[1,k] = \max_{j|c[1,j]<k} r[1,j] + f[0,k-c[0,j]] \tag{3.94}$$

k	$c[2,j]$	$f[0,k-c[0,j]]$	$f[1,k]$
0	0	0	0
1	0	1	5
2	2	0	8
3	2	1	9
4	3	1	13
5^*	4^*	1^*	18^*

STAGE TWO: we maximize the return of investment considering offers from firm 2 and the preceding table. We fill a table $f[2,k]$ with the maxi-

mum return of investment if we invest k million dollars in firm 0, firm 1, and firm 2:

$$f[2,k] = \max_{j|c[2,j]<k} r[2,j] + f[1,k-c[1,j]] \tag{3.95}$$

k	$c[2,j]$	$f[1,k-c[1,j]]$	$f[2,k]$
0	0	0	0
1	0	1	5
2	2	0	8
3	2	1	9
4	1	3	13
5^*	2^*	3^*	18^*

The maximum return of investment with \$5M is therefore \$18M. It can be achieved by investing \$2M in firm 2 and \$3M in firms 0 and 1. The optimal choice is marked with a star in each table. Note that to determine how much money has to be allocated to maximize the return of investment requires storing past tables to be able to look up the solution to subproblems.

We can generalize eq.(3.94) and eq.(3.95) for any number of investment firms (decision stages):

$$f[i,k] = \max_{j|c[i,j]<k} r[i,j] + f[i-1,k-c[i-1,j]] \tag{3.96}$$

3.9 Artificial intelligence and machine learning

3.9.1 Clustering algorithms

There are many algorithms available to cluster data [19]. They are all based on empirical principles because the clusters themselves are defined by the algorithm used to identify them. Normally we distinguish three categories:

- *Hierarchical clustering*: These algorithms start by considering each point a cluster of its own. At each iteration, the two clusters closest to each

other are joined together, forming a larger cluster. Hierarchical clustering algorithms differ from each other about the rule used to determine the distance between clusters. The algorithm returns a tree representing the clusters that are joined, called a *dendrogram*.

- *Centroid-based clustering*: These algorithms require that each point be represented by a vector and each cluster also be represented by a vector (centroid of the cluster). With each iteration, a better estimation for the centroids is given. An example of centroid-based clustering is *k-means* clustering. These algorithms require an a priori knowledge of the number of clusters and return the position of the centroids as well as the set of points belonging to each cluster.
- *Distribution-based clustering*: These algorithms are based on statistics (more than the other two categories). They assume the points are generated from a distribution (which must be known a priori) and determine the parameters of the distribution. It provides clustering because the distribution may be a sum of more than one localized distribution (each being a cluster).

Both *k*-means and distribution-based clustering assume an a priori knowledge about the data that often defies the purpose of using clustering: learn something we do not know about the data using an empirical algorithm. They also require that the points be represented by vectors in a Euclidean space, which is not always the case. Consider the case of clustering DNA sequences or financial time series. Technically the latter can be presented as vectors, but their dimensionality can be very large, thus making the algorithms impractical.

Hierarchical clustering only requires the notion of a distance between points, for some of the points.

The following algorithm is a hierarchical clustering algorithm with the following characteristics:

- Individual points do not need to be vectors (although they can be).
- Points may have a weight used to determine their relative importance in identifying the characteristics of the cluster (think of clustering finan-

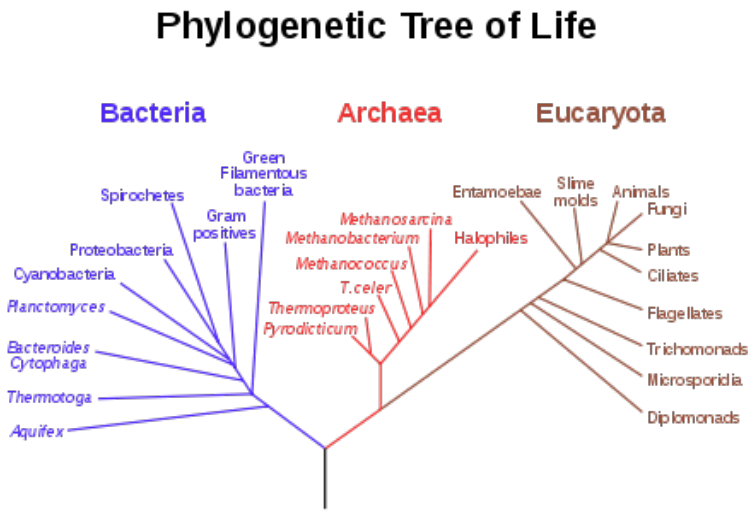

Figure 3.6: Example of a dendrogram.

cial assets based on the time series of their returns; the weight could the average traded volume).

- The distance between points is computed by a metric function provided by the user. The metric can return `None` if there is no known connection between two points.
- The algorithm can be used to build the entire *dendrogram* , or it can stop for a given value of k, a target number of clusters.
- For points that are vectors and a given k, the result is similar to the result of the k-means clustering.

The algorithm works like any other hierarchical clustering algorithm. At the beginning, all-to-all distances are computed and stored in a list `d`. Each point is its own cluster. At each iteration, the two clusters closer together are merged to form one bigger cluster. The distance between each other cluster and the merged cluster is computed by performing a weighted average of the distances between the other cluster and the two merged clusters. The weight factors are provided as input. This is equivalent to what the k-means algorithm does by computing the position of a centroid

based on the vectors of the member points.

The algorithm `self.q` implements disjoint sets representing the set of clusters. The algorithm `self.q` is a dictionary. If `self.q[i]` is a list, then `i` is its own cluster, and the list contains the IDs of the member points. If `self.q[i]` is an integer, then cluster `i` is no longer its own cluster as it was merged to the cluster represented by the integer.

At each point in time, each cluster is represented by one element, which can be found recursively by `self.parent(i)`. This function returns the ID of the cluster containing element `i` and returns a list of IDs of all points in the same cluster:

Listing 3.15: in file: `nlib/cluster.nim`

```
type
  ClusterEntry = object
    case isList: bool
    of true: members: seq[int]
    of false: parentId: int

  Cluster*[T] = ref object
    points*: seq[T]
    metric*: proc(a, b: T): float
    k*: int
    w*: seq[float]
    q*: Table[int, ClusterEntry]
    d*: seq[(float, int, int)]
    dd*: seq[(float, int)]
    r*: float
    v*: seq[seq[int]]

proc newCluster*[T](points: seq[T],
                    metric: proc(a, b: T): float,
                    weights: seq[float] = @[]): Cluster[T] =
  result = Cluster[T](points: points, metric: metric, k: points.len)
  result.w = if weights.len > 0: weights
             else: newSeqWith(points.len, 1.0)
  for i in 0 ..< points.len:
    result.q[i] = ClusterEntry(isList: true, members: @[i])
  for i in 0 ..< points.len:
    for j in i + 1 ..< points.len:
      let m = metric(points[i], points[j])
      if not m.isNaN:
        result.d.add (m, i, j)
  result.d.sort()
```

```
32
33 proc parent*[T](c: Cluster[T], i: int): (int, seq[int]) =
34   var i = i
35   while c.q[i].isList == false:
36     i = c.q[i].parentId
37   (i, c.q[i].members)
38
39 proc step*[T](c: Cluster[T]): (float, seq[seq[int]]) =
40   if c.k > 1:
41     let head = c.d[0]
42     c.r = head[0]
43     var i = head[1]
44     var j = head[2]
45     c.d.delete(0)
46     let (pi, x) = c.parent(i)
47     let (pj, y) = c.parent(j)
48     var merged = x & y
49     c.q[pi] = ClusterEntry(isList: true, members: merged)
50     c.q[pj] = ClusterEntry(isList: false, parentId: pi)
51     dec c.k
52     var newD: seq[(float, int, int)] = @[]
53     var oldD = initTable[int, (float, float)]()
54     for (r, h, k) in c.d:
55       if h in [pi, pj]:
56         let (a, b) = oldD.getOrDefault(k, (0.0, 0.0))
57         oldD[k] = (a + c.w[k] * r, b + c.w[k])
58       elif k in [pi, pj]:
59         let (a, b) = oldD.getOrDefault(h, (0.0, 0.0))
60         oldD[h] = (a + c.w[h] * r, b + c.w[h])
61       else:
62         newD.add (r, h, k)
63     for k, ab in tables.pairs(oldD):
64       newD.add (ab[0] / ab[1], pi, k)
65     newD.sort()
66     c.d = newD
67     c.w[pi] = c.w[pi] + c.w[pj]
68     var v: seq[seq[int]] = @[]
69     for _, e in tables.pairs(c.q):
70       if e.isList: v.add e.members
71     c.v = v
72     c.dd.add (c.r, c.v.len)
73   (c.r, c.v)
74
75 proc find*[T](c: Cluster[T], k: int): (float, seq[seq[int]]) =
76   while c.k > k:
77     discard c.step()
78   (c.r, c.v)
```

Given a set of points, we can determine the most likely number of clusters

representing the data, and we can make a plot of the number of clusters versus distance and look for a plateau in the plot. In correspondence with the plateau, we can read from the y-coordinate the number of clusters. This is done by the function `cluster` in the preceding algorithm, which returns the average distance between clusters and a list of clusters.

For example:

```
proc metric(a, b: seq[float]): float =
  for i in 0 ..< a.len: result += (a[i] - b[i]) ^ 2
  result = sqrt(result)

randomize()
var points: seq[seq[float]] = @[]
for i in 0 ..< 200:
  var p = newSeq[float](10)
  for j in 0 ..< 10:
    p[j] = gauss(float(i mod 5), 0.3)
  points.add p
let c = newCluster[seq[float]](points, metric)
let (r, clusters) = c.find(1)
# Render the dendrogram using `savePlot` on (distance, count).
```

With our sample data, we obtain the following plot ("clustering1.png"):

and the location where the curve bends corresponds to five clusters. Although our points live in 10 dimensions, we can try to project them into two dimensions and see the five clusters ("clustering2.png"):

3.9.2 Neural network

An artificial *neural network* is an electrical circuit (usually simulated in software) that mimics the functionality of the neurons in the animal (and human) brain [20]. It is usually employed in pattern recognition. The network consists of a set of simulated neurons, connected by links (synapses). Some links connect the neurons with each other, some connect the neurons with the input and some with the output. Neurons are usually organized in the layers with one *input layer* of neurons connected only with the input and the next layer. Another one, the *output layer*, comprises neurons connected only with the output and previous layers, or many *hidden layers* of neurons connected only with other neurons. Each neuron is characterized by input links and output links. Each output of a neuron is a

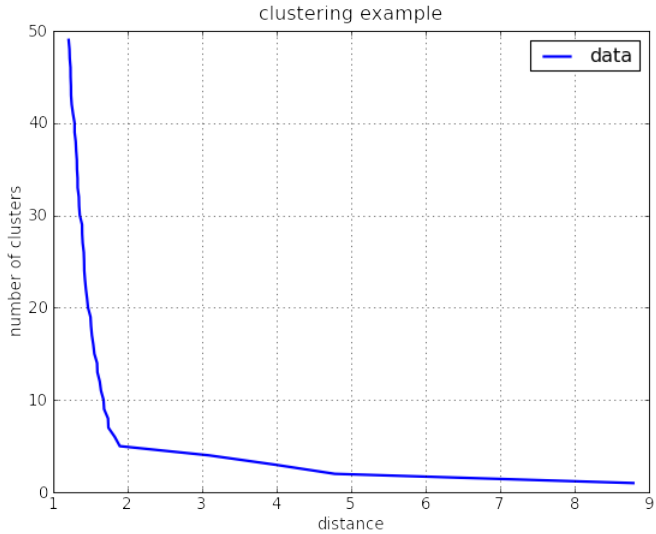

Figure 3.7: Number of clusters found as a function of the distance cutoff.

function of its inputs. The exact shape of that function depends on the network and on parameters that can be adjusted. Usually this function is chosen to be a monotonic increasing function on the sum of the inputs, where both the inputs and the outputs take values in the [0, 1] range. The inputs can be thought as electrical signals reaching the neuron. The output is the electrical signal emitted by the neuron. Each neuron is defined by a set of parameters a which determined the relative weight of the input signals. A common choice for this characteristic function is:

$$\text{output}_{ij} = \tanh(\sum_k a_{ijk}\text{input}_{ik}) \tag{3.97}$$

where i labels the neuron, j labels the output, k labels the input, and a_{ijk} are characteristic parameters describing the neurons.

The network is trained by providing an input and adjusting the characteristics a_{ijk} of each neuron k to produce the expected output. The network is trained iteratively until its parameters converge (if they converge), and

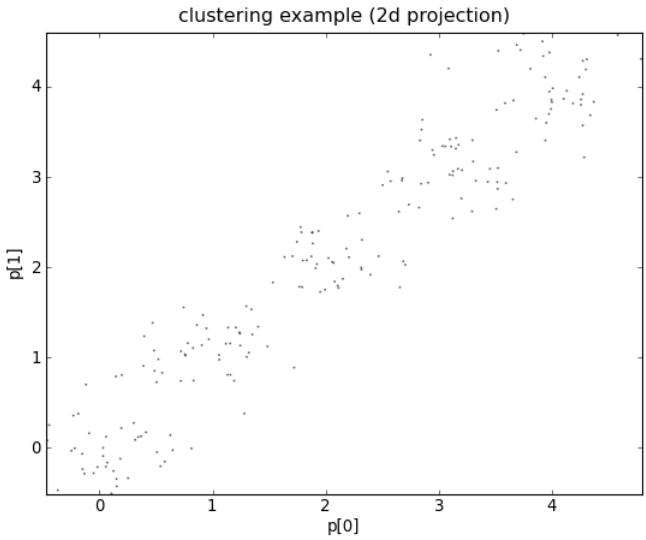

Figure 3.8: Visual representation of the clusters where the points coordinates are projected in 2D.

then it is ready to make predictions. We say the network has learned from the training data set.

Listing 3.16: in file: `nlib/neural.nim`

```
type
  NeuralNetwork* = ref object
    ni*, nh*, no*: int
    ai*, ah*, ao*: seq[float]
    wi*, wo*: Matrix       # weights
    ci*, co*: Matrix       # last weight change (momentum)

proc nnRand(a, b: float): float = (b - a) * rand(1.0) + a
proc sigmoid(x: float): float = tanh(x)
proc dsigmoid(y: float): float = 1.0 - y * y

proc newNeuralNetwork*(ni, nh, no: int): NeuralNetwork =
  result = NeuralNetwork(
    ni: ni + 1,    # +1 for bias node
    nh: nh,
    no: no,
    ai: newSeqWith(ni + 1, 1.0),
    ah: newSeqWith(nh, 1.0),
```

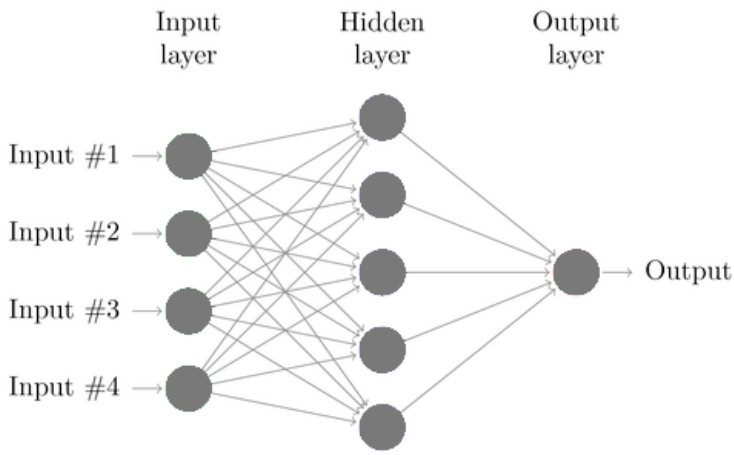

Figure 3.9: Example of a minimalist neural network.

```
    ao: newSeqWith(no, 1.0),
    wi: newMatrix(ni + 1, nh,
                  proc(r, c: int): float = nnRand(-0.2, 0.2)),
    wo: newMatrix(nh, no,
                  proc(r, c: int): float = nnRand(-2.0, 2.0)),
    ci: newMatrix(ni + 1, nh),
    co: newMatrix(nh, no))

proc update*(n: NeuralNetwork, inputs: seq[float]): seq[float] =
  if inputs.len != n.ni - 1:
    raise newException(ValueError, "wrong number of inputs")
  for i in 0 ..< n.ni - 1:
    n.ai[i] = inputs[i]
  for j in 0 ..< n.nh:
    var s = 0.0
    for i in 0 ..< n.ni: s += n.ai[i] * n.wi[i, j]
    n.ah[j] = sigmoid(s)
  for k in 0 ..< n.no:
    var s = 0.0
    for j in 0 ..< n.nh: s += n.ah[j] * n.wo[j, k]
    n.ao[k] = sigmoid(s)
  result = n.ao

proc backPropagate*(n: NeuralNetwork, targets: seq[float],
                    N, M: float): float =
  if targets.len != n.no:
    raise newException(ValueError, "wrong number of target values")
  var outputDeltas = newSeq[float](n.no)
```

```
  for k in 0 ..< n.no:
    let error = targets[k] - n.ao[k]
    outputDeltas[k] = dsigmoid(n.ao[k]) * error
  var hiddenDeltas = newSeq[float](n.nh)
  for j in 0 ..< n.nh:
    var error = 0.0
    for k in 0 ..< n.no: error += outputDeltas[k] * n.wo[j, k]
    hiddenDeltas[j] = dsigmoid(n.ah[j]) * error
  for j in 0 ..< n.nh:
    for k in 0 ..< n.no:
      let change = outputDeltas[k] * n.ah[j]
      n.wo[j, k] = n.wo[j, k] + N * change + M * n.co[j, k]
      n.co[j, k] = change
  for i in 0 ..< n.ni:
    for j in 0 ..< n.nh:
      let change = hiddenDeltas[j] * n.ai[i]
      n.wi[i, j] = n.wi[i, j] + N * change + M * n.ci[i, j]
      n.ci[i, j] = change
  var error = 0.0
  for k in 0 ..< targets.len:
    error += 0.5 * (targets[k] - n.ao[k]) ^ 2
  return error

proc test*(n: NeuralNetwork, patterns: seq[(seq[float], seq[float])]) =
  for p in patterns:
    echo p[0], " -> ", n.update(p[0])

proc weights*(n: NeuralNetwork) =
  echo "Input weights:"
  for i in 0 ..< n.ni:
    var row: seq[float]
    for j in 0 ..< n.nh: row.add n.wi[i, j]
    echo row
  echo ""
  echo "Output weights:"
  for j in 0 ..< n.nh:
    var row: seq[float]
    for k in 0 ..< n.no: row.add n.wo[j, k]
    echo row

proc train*(n: NeuralNetwork,
            patterns: seq[(seq[float], seq[float])],
            iterations = 1000, N = 0.5, M = 0.1,
            check = false) =
  for i in 0 ..< iterations:
    var error = 0.0
    for p in patterns:
      discard n.update(p[0])
      error += n.backPropagate(p[1], N, M)
```

```
    if check and i mod 100 == 0:
      echo "error ", error
```

In the following example, we teach the network the XOR function, and we create a network with two inputs, two intermediate neurons, and one output. We train it and check what it learned:

```
let pat = @[(@[0.0, 0.0], @[0.0]),
            (@[0.0, 1.0], @[1.0]),
            (@[1.0, 0.0], @[1.0]),
            (@[1.0, 1.0], @[0.0])]
let n = newNeuralNetwork(2, 2, 1)
n.train(pat)
n.test(pat)
# @[0.0, 0.0] -> @[0.00...]
# @[0.0, 1.0] -> @[0.98...]
# @[1.0, 0.0] -> @[0.98...]
# @[1.0, 1.0] -> @[-0.00...]
```

Now, we use our neural network to learn patterns in stock prices and predict the next day return. We then check what it has learned, comparing the sign of the prediction with the sign of the actual return for the same days used to train the network:

Listing 3.17: in file: `test.nim`

```
let storage = newPersistentDictionary("sp100.json")
var v: seq[float] = @[]
for day in storage["AAPL/2011"][1 .. ^1]:
  v.add day["arithmetic_return"].getFloat * 300.0
var pat: seq[(seq[float], seq[float])] = @[]
for i in 0 ..< v.len - 5:
  pat.add (v[i ..< i + 5], @[v[i + 5]])
let n = newNeuralNetwork(5, 5, 1)
n.train(pat)
var predictions: seq[seq[float]] = @[]
for item in pat: predictions.add n.update(item[0])
var hits = 0.0
for i, e in predictions:
  if e[0] * v[i + 5] > 0: hits += 1.0
let successRate = hits / float(pat.len)
```

The learning process depends on the random number generator; therefore, sometimes, for this small training data set, the network succeeds in predicting the sign of the next day arithmetic return of the stock with more than 50% probability, and sometimes it does not. We leave it to the reader to study the significance of this result by using a different subset

of the data for the training of the network and for testing its success rate.

3.9.3 Genetic algorithms

Here we consider a simple example of genetic algorithms [21].

We have a population of chromosomes in which each chromosome is just a data structure, in our example, a string of random "ATGC" characters.

We also have a metric to measure the fitness of each chromosome.

At each iteration, only the top-ranking chromosomes in the population survive. The top 10 mate with each other, and their offspring constitute the population for the next iteration. When two members of the population mate, the newborn member of the population has a new DNA sequence, half of which comes from the father and half from the mother, with two randomly mutated DNA bases.

The algorithm stops when we reach a maximum number of generations or we find a chromosome of the population with maximum fitness.

In the following example, the fitness is measured by the similarity between a chromosome and a random target chromosome. The population evolves to approximate better and better that one random target chromosome:

Listing 3.18: in file: `genetic.nim`

```
import std/[random, algorithm, sequtils]

const
  ALPHABET = "ATGC"
  SIZE = 32
  MUTATIONS = 2

type
  Chromosome* = ref object
    dna*: string

proc randomChar(): char = ALPHABET[rand(ALPHABET.len - 1)]

proc newChromosome*(): Chromosome =
  result = Chromosome(dna: newString(SIZE))
  for i in 0 ..< SIZE: result.dna[i] = randomChar()
```

```
17
18 proc newChromosome*(father, mother: Chromosome): Chromosome =
19   result = Chromosome(
20     dna: father.dna[0 ..< SIZE div 2] & mother.dna[SIZE div 2 .. ^1])
21   for _ in 0 ..< MUTATIONS:
22     result.dna[rand(SIZE - 1)] = randomChar()
23
24 proc fitness*(c, target: Chromosome): int =
25   for i, ch in c.dna:
26     if ch == target.dna[i]: inc result
27
28 proc top*(population: seq[Chromosome], target: Chromosome,
29           n = 10): seq[Chromosome] =
30   var table: seq[(int, Chromosome)] = @[]
31   for chromo in population:
32     table.add (chromo.fitness(target), chromo)
33   table.sort(proc(a, b: (int, Chromosome)): int = cmp(b[0], a[0]))
34   for i in 0 ..< min(n, table.len):
35     result.add table[i][1]
36
37 proc oneof*(population: seq[Chromosome]): Chromosome =
38   population[rand(population.len - 1)]
39
40 proc main() =
41   const GENERATIONS = 10000
42   const OFFSPRING = 20
43   const SEEDS = 20
44   let target = newChromosome()
45
46   var population: seq[Chromosome] = @[]
47   for _ in 0 ..< SEEDS: population.add newChromosome()
48   for i in 0 ..< GENERATIONS:
49     echo "\n\nGENERATION: ", i
50     echo 0, " ", target.dna
51     let fittest = top(population, target)
52     for chromosome in fittest: echo i, " ", chromosome.dna
53     var best = 0
54     for chromo in fittest: best = max(best, chromo.fitness(target))
55     if best == SIZE:
56       echo "SOLUTION FOUND"
57       break
58     population = @[]
59     for _ in 0 ..< OFFSPRING:
60       population.add newChromosome(oneof(fittest), oneof(fittest))
61
62 when isMainModule:
63   randomize()
64   main()
```

Notice that this algorithm can easily be modified to accommodate other fitness metrics and DNA that consists of a data structure other than a sequence of "ATGC" symbols. The only trickery is finding a proper mating algorithm that preserves some of the fitness features of the parents in the DNA of their offspring. If this does not happen, each next generation loses the fitness properties gained by its parents, thus causing the algorithm not to converge. In our case, it works because if the parents are "close" to the target, then half of the DNA of each parent is also close to the corresponding half of the target DNA. Therefore the DNA of the offspring is as fit as the average of their parents. On top of this, the two random mutations allow the algorithm to further explore the space of all possible DNA sequences.

3.10 Long and infinite loops

3.10.1 P, NP, and NPC

We say a problem is in P if it can be solved in polynomial time: $T_{worst} \in O(n^{\alpha})$ for some α.

We say a problem is in NP if an input string can be verified to be a solution in polynomial time: $T_{worst} \in O(n^{\alpha})$ for some α.

We say a problem is in co-NP if an input string can be verified not to be a solution in polynomial time: $T_{worst} \in O(n^{\alpha})$ for some α.

We say a problem is in NPH (NP Hard) if it is harder than any other problem in NP.

We say a problem is in NPC (NP Complete) if it is in NP and in NPH. Consequences:

$$\text{if } \exists x \mid x \in NPC \text{ and } x \in P \Rightarrow \forall y \in NP, y \in P \tag{3.98}$$

There are a number of open problems about the relations among these sets. Is the set co-NP equivalent to NP? Or perhaps is the intersection between co-NP and NP equal to P? Are NP and NPC the same set? These questions are very important in computer science because if, for example,

NP turns out to be the same set as NPC, it means that it must be possible to find algorithms that solve in polynomial time problems that currently do not have a polynomial time solution. Conversely, if one could prove that NP is not equivalent to NPC, we would know that a polynomial time solution to NPC problems does not exist [22].

3.10.2 Cantor's argument

Cantor proved that the real numbers in any interval (e.g., in $[0,1)$) are more than the integer numbers, therefore real numbers are uncountable [23]. The proof proceeds as follows:

1. Consider the real numbers in the interval $[0,1)$ not including 1.
2. Assume that these real numbers are countable. Therefore it is possible to associate each of them to an integer

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.xxxxxxxxxxx... \\ 2 & \longleftrightarrow & 0.xxxxxxxxxxx... \\ 3 & \longleftrightarrow & 0.xxxxxxxxxxx... \\ 4 & \longleftrightarrow & 0.xxxxxxxxxxx... \\ 5 & \longleftrightarrow & 0.xxxxxxxxxxx... \\ ... & ... & ... \end{array} \tag{3.99}$$

   (here x represent a decimal digit of a real numbers)

3. Now construct a number $\alpha = 0.yyyyyyyy....$ where the first decimal digit differs from the first decimal digit of the first real number of table 3.99, the second decimal digit differs from the second decimal digit of the second real number of table 3.99, and so on and so on for all the infinite decimal digits:

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.\bar{x}xxxxxxxxxx... \\ 2 & \longleftrightarrow & 0.x\bar{x}xxxxxxxxx... \\ 3 & \longleftrightarrow & 0.xx\bar{x}xxxxxxxx... \\ 4 & \longleftrightarrow & 0.xxx\bar{x}xxxxxxx... \\ 5 & \longleftrightarrow & 0.xxxx\bar{x}xxxxxx... \\ ... & ... & ... \end{array} \tag{3.100}$$

4. The new number α is a real number, and by construction, it is not in the table. In fact, it differs with each item by at least one decimal digit. Therefore the existence of α disproves the assumption that all real numbers in the interval $[0, 1)$ are listed in the table.

There is a very practical consequence of this argument. In fact, in chapter 2, we have seen the distinction between type `float` and class `Decimal`. We have seen about pitfalls of `float` and how `Decimal` can represent floating point numbers with arbitrary precision (assuming we have the memory to do so). Cantor's argument tells us there are numbers that cannot even be represented as `Decimal` because they would require an infinite amount of storage; π and e are examples of these numbers.

3.10.3 Gödel's theorem

Gödel used a similar diagonal argument to prove that there are as many problems (or theorems) as real numbers and as many algorithms (or proofs) as natural numbers [23]. Because there is more of the former than the latter, it follows that there are problems for which there is no corresponding solving algorithm. Another interpretation of Gödel's theorem is that, in any formal language, for example, mathematics, there are theorems that cannot be proved.

Another consequence of Gödel's theorem is the following: it is impossible to write a computer program to test if a given algorithm stops or enters into an infinite loop.

Consider the following code:

```
import std/sets

proc next(i: int) =
  var i = i
  while toHashSet($($(i * i))).card > 2:
    i += 2
  echo i

next(81621)
```

This code check searches for a number equal or greater than 81621 whose square is comprised of only two digits. Nobody knows whether such

number exists, therefore nobody knows if this code stops.

Although one day this problem may be solved, there are many other problems that are still unsolved; actually, there are an infinite number of them.

4
Numerical Algorithms

4.1 Well-posed and stable problems

Numerical algorithms deal mostly with well-posed and stable problems.

A problem is well posed if

- The solution exists and is unique
- The solution has a continuous dependence on input data (a small change in the input causes a small change in the output)

Most physical problems are well posed, except at *critical points*, where any infinitesimal variation in one of the input parameters of the system can cause a large change in the output and therefore in the behavior of the system. This is called *chaos*.

Consider the case of dropping a ball on a triangular-shaped mountain. Let the input of the problem be the horizontal position where the drop occurs and the output the horizontal position of the ball after a fixed amount of time. Almost anywhere the ball is dropped, it will roll down the mountain following deterministic and classical laws of physics, thus the position is calculable and a continuous function of the input position. This is true everywhere, except when the ball is dropped on top of the peak of the mountain. In this case, a minor infinitesimal variation to the right or to the left can make the ball roll to the right or to the left,

respectively. Therefore this is not a well posed problem.

A problem is said to be *stable* if the solution is not just continuous but also weakly sensitive to input data. This means that the change of the output (in percent) is smaller than the change in the input (in percent).

Numerical algorithms work best with stable problems.

We can quantify this as follows. Let x be an input and y be the output of a function:

$$y = f(x) \tag{4.1}$$

We define the condition number of f in x as

$$\text{cond}(f, x) \equiv \frac{|dy/y|}{|dx/x|} = |xf'(x)/f(x)| \tag{4.2}$$

(the latter equality only holds if f is differentiable in x).

A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned.

We say that a problem characterized by a function f is well conditioned in a domain D if the condition number is less than 1 for every input in the domain. We also say that a problem is stable if it is well conditioned.

In this book, we are mostly concerned with stable (well-conditioned) problems. If a problem is well-conditioned for all input in a domain, it is also stable.

4.2 Approximations and error analysis

Consider a physical quantity, for example, the length of a nail. Given one nail, we can measure its length by choosing a measuring instrument. Whatever instrument we choose, we will be able to measure the length of the nail within the resolution of the instrument. For example, with a tape measure with a resolution of 1 mm, we will only be able to determine the length of the nail within 1 mm of resolution. Repeated measurements performed at different times, by different people, using different instruments

may bring different results. We can choose a more precise instrument, but it would not change the fact that different measures will bring different values compatible with the resolution of the instrument. Eventually one will have to face the fact that there may not be such a thing as the length of a nail. For example, the length varies with the temperature and the details of how the measurement is performed. In fact, a nail (as everything else) is made out of atoms, which are made of protons, neutrons, and electrons, which determine an electromagnetic cloud that fluctuates in space and time and depends on the surrounding objects and interacts with the instrument of measure. The length of the nail is the result of a measure.

For each measure there is a result, but the results of multiple measurements are not identical. The results of many measurements performed with the same resolution can be summarized in a distribution of results. This distribution will have a mean $\bar{x}$ and a standard deviation δx, which we call uncertainty. From now on, unless otherwise specified, we assume that the distribution of results is Gaussian so that $\bar{x}$ can be interpreted as the mean and δx as the standard deviation.

Now let us consider a system that, given an input x, produces the output y; x and y are physical quantities that we can measure, although only with a finite resolution. We can model the system with a function f such that $y = f(x)$ and, in general, f is not known.

We have to make various approximations:

- We can replace the "true" value for the input with our best estimate, $\bar{x}$, and its associated uncertainty, δx.
- We can replace the "true" value for the output with our best estimate, $\bar{y}$, and its associated uncertainty, δy.
- Even if we know there is a "true" function f describing the system, our implementation for the function is always an approximation, $\bar{f}$. In fact, we may not have a single approximation but a series of approximations of increasing precision, f_n, which become more and more accurate (usually) as n increases. If we are lucky, up to precision errors,

as n increases, our approximations will become closer and closer to f, but this will take an infinite amount of time. We have to stop at some finite n.

With the preceding definition, we can define the following types of errors:

- **Data error**: the difference between x and $\bar{x}$.
- **Computational error**: the difference between $\bar{f}(\bar{x})$ and y. Computational error includes two parts systematic error and statistical error.
- **Statistical error**: due to the fact that, often, the computation of $\bar{f}(x) = \lim_{n\to\infty} f_n(x)$ is too computationally expensive and we must approximate $\bar{f}(x)$ with $f_n(x)$. This error can be estimated and controlled.
- **Systematic error**: due to the fact that $\bar{f}(x) = \lim_{n\to\infty} f_n(x) \neq f(x)$. This is for two reasons: modeling errors (we do not know $f(x)$) and rounding errors (we do not implement $f(x)$ with arbitrary precision arithmetics).
- **Total error**: defined as the computational error + the propagated data error and in a formula:

$$\delta y = |f(\bar{x}) - f_n(\bar{x})| + |f_n'(\bar{x})|\delta x \tag{4.3}$$

  The first term is the computational error (we use f_n instead of the true f), and the second term is the propagated data error (δx, the uncertainty in x, propagates through f_n).

4.2.1 Error propagation

When a variable x has a finite Gaussian uncertainty δx, how does the uncertainty propagate through a function f? Assuming the uncertainty is small, we can always expand using a Taylor series:

$$y + \delta y = f(x + \delta x) = f(x) + f'(x)\delta x + O(\delta x^2) \tag{4.4}$$

And because we interpret δy as the width of the distribution y, it should be positive:

$$\delta y = |f'(x)|\delta x \tag{4.5}$$

We have used this formula before for the propagated data error. For functions of two variables $z = f(x, y)$ and assuming the uncertainties in x and y are independent,

$$\delta z = \sqrt{\left|\frac{\partial f(x,y)}{\partial x}\right|^2 \delta x^2 + \left|\frac{\partial f(x,y)}{\partial y}\right|^2 \delta y^2} \tag{4.6}$$

which for simple arithmetic operations reduces to

$$\begin{array}{ll} z = x + y & \delta z = \sqrt{\delta x^2 + \delta y^2} \\ z = x - y & \delta z = \sqrt{\delta x^2 + \delta y^2} \\ z = x * y & \delta z = |x * y|\sqrt{(\delta x/x)^2 + (\delta y/y)^2} \\ z = x/y & \delta z = |x/y|\sqrt{(\delta x/x)^2 + (\delta y/y)^2} \end{array}$$

Notice that when $z = x - y$ approaches zero, the uncertainty in z is larger than the uncertainty in x and y and can overwhelm the result. Also notice that if $z = x/y$ and y is small compared to x, then the uncertainty in z can be large. Bottom line: try to avoid differences between numbers that are in proximity of each other and try to avoid dividing by small numbers.

4.3 Standard strategies

Here are some strategies that are normally employed in numerical algorithms:

- Approximate a continuous system with a discrete system
- Replace integrals with sums
- Replace derivatives with finite differences
- Replace nonlinear with linear + corrections
- Transform a problem into a different one
- Approach the true result by iterations

Here are some examples of each of the strategies.

4.3.1 Approximate continuous with discrete

Consider a ball in a one-dimensional box of size L, and let x be the position of the ball in the box. Instead of treating x as a continuous variable, we can assume a finite resolution of $h = L/n$ (where h is the minimum distance we can distinguish without instruments and n is the maximum number of distinct discrete points we can discriminate), and set $x \equiv hi$, where i is an integer in between 0 and n; $x = 0$ when $i = 0$ and $x = L$ when $i = n$.

4.3.2 Replace derivatives with finite differences

Computing $\mathrm{d}f(x)/\mathrm{d}x$ analytically is only possible when the function f is expressed in simple analytical terms. Computing it analytically is not possible when $f(x)$ is itself implemented as a numerical algorithm. Here is an example:

```
proc f(x: float): float =
  var s = 1.0
  var t = 1.0
  for i in 1 ..< 10:
    (s, t) = (s + t, t * x / float(i))
  return s
```

What is the derivative of $f(x)$?

The most common ways to define a derivative are the right derivative

$$\frac{\mathrm{d}f^+(x)}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{4.7}$$

the left derivative

$$\frac{\mathrm{d}f^-(x)}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x) - f(x-h)}{h} \tag{4.8}$$

and the average of the two

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{1}{2}\left(\frac{\mathrm{d}f^+(x)}{\mathrm{d}x} + \frac{\mathrm{d}f^-(x)}{\mathrm{d}x}\right) = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{4.9}$$

If the function is differentiable in x, then, by definition of "differentiable," the left and right definitions are equal, and the three prior definitions

are equivalent. We can pick one or the other, and the difference will be a systematic error.

If the limit exists, then it means that

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{f(x+h) - f(x-h)}{2h} + O(h) \tag{4.10}$$

where $O(h)$ indicates a correction that, at most, is proportional to h.

The three definitions are equivalent for functions that are differentiable in x, and the latter is preferable because it is more symmetric.

Notice that even more definitions are possible as long as they agree in the limit $h \to 0$. Definitions that converge faster as h goes to zero are referred to as "improvement."

We can easily implement the concept of a numerical derivative in code by creating a *functional* D that takes a function f and returns the function $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ (a functional is a function that returns another function):

Listing 4.1: in file: `nlib/calculus.nim`

```
proc D*(f: proc(x: float): float, h = 1e-6): proc(x: float): float =
  result = proc(x: float): float = (f(x + h) - f(x - h)) / 2.0 / h
```

We can do the same with the second derivative:

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = \frac{f(x+h) - 2f(x) - f(x-h)}{h^2} + O(h) \tag{4.11}$$

Listing 4.2: in file: `nlib/calculus.nim`

```
proc DD*(f: proc(x: float): float, h = 1e-6): proc(x: float): float =
  result = proc(x: float): float =
    (f(x + h) - 2.0 * f(x) + f(x - h)) / (h * h)
```

Here is an example:

```
proc f(x: float): float = x * x - 5.0 * x
echo f(0.0)            # 0.0
let f1 = D(f)          # first derivative
echo f1(0.0)           # -5.0
let f2 = DD(f)         # second derivative
echo f2(0.0)           # 2.00000...
let f2b = D(f1)        # second derivative (composing)
```

```
echo f2b(0.0)          # 1.99999...
```

Notice how composing the first derivative twice or computing the second derivative directly yields a similar result.

We could easily derive formulas for higher-order derivatives and implement them, but they are rarely needed.

4.3.3 Replace nonlinear with linear

Suppose we are interested in the values of $f(x) = \sin(x)$ for values of x between 0 and 0.1:

```
import std/math
for i in 0 .. 10:
  let x = 0.01 * float(i)
  echo x, " ", sin(x), " ",
       formatFloat(abs(x - sin(x)) / sin(x) * 100, ffDecimal, 2)
# 0.01 0.009999833... 0.00
# 0.02 0.019998666... 0.01
# 0.03 0.029995500... 0.02
# 0.04 0.039989334... 0.03
# 0.05 0.049979169... 0.04
# 0.06 0.059964006... 0.06
# 0.07 0.069942847... 0.08
# 0.08 0.079914693... 0.11
# 0.09 0.089878549... 0.14
# 0.1 0.0998334166... 0.17
```

Here the first column is the value of x, the second column is the corresponding $\sin(x)$, and the third column is the relative difference (in percent) between x and $\sin(x)$. The difference is always less than 20%; therefore, if we are happy with this precision, then we can replace $\sin(x)$ with x.

This works because any function $f(x)$ can be expanded using a Taylor series. The first order of the Taylor expansion is linear. For values of x sufficiently close to the expansion point, the function can therefore be approximated with its Taylor expansion.

Expanding on the previous example, consider the following code:

```
import std/math
for i in 0 .. 10:
  let x = 0.01 * float(i)
```

```
  let s = x - x * x * x / 6.0
  echo x, " ", sin(x), " ", s, " ",
       formatFloat(abs(s - sin(x)) / sin(x) * 100, ffDecimal, 6)
# 0.01 0.009999833... 0.009999... 0.000000
# 0.02 0.019998666... 0.019998... 0.000000
# 0.03 0.029995500... 0.029995... 0.000001
# 0.04 0.039989334... 0.039989... 0.000002
# 0.05 0.049979169... 0.049979... 0.000005
# 0.06 0.059964006... 0.059964... 0.000011
0.07 0.069942847... 0.069942... 0.000020
0.08 0.079914693... 0.079914... 0.000034
0.09 0.089878549... 0.089878... 0.000055
0.1 0.0998334166... 0.099833... 0.000083
```

Notice that the third column $s = x - x^3/6$ is very close to $\sin(x)$. In fact, the difference is less than one part in 10, 000 (fourth column). Therefore, for $x \in [-1, +1]$, it is possible to replace the $sin(x)$ function with the $x - x^3/6$ polynomial. Here we just went one step further in the Taylor expansion, replacing the first order with the third order. The error committed in this approximation is very small.

4.3.4 Transform a problem into a different one

Continuing with the previous example, the polynomial approximation for the sin function works when x is smaller than 1 but fails when x is greater than or equal to 1. In this case, we can use the following relations to reduce the computation of $\sin(x)$ for large x to $\sin(x)$ for $0 < x < 1$. In particular, we can use

$$\sin(x) = -\sin(-x) \text{when} x < 0 \tag{4.12}$$

to reduce the domain to $x \in [0, \infty]$. We can then use

$$\sin(x) = \sin(x - 2k\pi) \qquad k \in \mathbb{N} \tag{4.13}$$

to reduce the domain to $x \in [0, 2\pi)$

$$\sin(x) = -\sin(2\pi - x) \tag{4.14}$$

to reduce the domain to $x \in [0, \pi)$

$$\sin(x) = \sin(\pi - x) \tag{4.15}$$

to reduce the domain to $x \in [0, \pi/2)$, and

$$\sin(x) = \sqrt{1 - \sin(\pi/2 - x)^2} \tag{4.16}$$

to reduce the domain to $x \in [0, \pi/4)$, where the latter is a subset of $[0, 1)$.

4.3.5 Approximate the true result via iteration

The approximations $\sin(x) \simeq x$ and $\sin(x) \simeq x - x^3/6$ came from linearizing the function $\sin(x)$ and adding a correction to the previous approximation, respectively. In general, we can iterate the process of finding corrections and approximating the true result.

Here is an example of a general iterative algorithm:

```
result=guess
loop:
    compute correction
    result=result+correction
    if result sufficiently close to true result:
        return result
```

For the sin function:

```
proc mysin(x: float): float =
  var s = 0.0
  var t = x
  var i = 3
  while i < 10:
    s += t
    t = -t * x * x / float(i) / float(i - 1)
    i += 2
  return s
```

Where do these formulas come from? How do we decide how many iterations we need? We address these problems in the next section.

4.3.6 Taylor series

A function $f(x) : \mathbb{R} \to \mathbb{R}$ is said to be a *real analytical* in $\bar{x}$ if it is continuous in $x = \bar{x}$ and all its derivatives exist and are continuous in $x = \bar{x}$.

When this is the case, the function can be locally approximated with a local power series:

$$f(x) = f(\bar{x}) + f^{(0)}(\bar{x})(x - \bar{x}) + ... + \frac{f^{(k)}(\bar{x})}{n!}(x - \bar{x})^k + R_k \tag{4.17}$$

The remainder R_k can be proven to be (Taylor's theorem):

$$R_k = \frac{f^{(k+1)}(\xi)}{(k+1)!}(x - \bar{x})^{k+1} \tag{4.18}$$

where ξ is a point in between x and $\bar{x}$. Therefore, if $f^{(k+1)}$ exists and is limited within a neighborhood $D = \{x \text{ for } |x - \bar{x}| < \epsilon\}$, then

$$|R_k| < \left| max_{x \in D} f^{(k+1)} \right| |(x - \bar{x})^{k+1}| \tag{4.19}$$

If we stop the Taylor expansion at a finite value of k, the preceding formula gives us the statistical error part of the computational error.

Some Taylor series are very easy to compute:

Exponential for $\bar{x} = 0$:

$$\begin{aligned} f(x) &= e^x && (4.20)\\ f^{(1)}(x) &= e^x && (4.21)\\ ... &= ... && (4.22)\\ f^{(k)}(x) &= e^x && (4.23)\\ e^x &= 1 + x + \frac{1}{2}x^2 + ... + \frac{1}{k!}x^k + ... && (4.24) \end{aligned}$$

Sin for $\bar{x} = 0$:

$$
\begin{aligned}
f(x) &= sin(x) && (4.25)\\
f^{(1)}(x) &= cos(x) && (4.26)\\
f^{(2)}(x) &= -sin(x) && (4.27)\\
f^{(3)}(x) &= -cos(x) && (4.28)\\
... &= ... && (4.29)\\
sin(x) &= x - \frac{1}{3!}x^3 + ... + \frac{(-1)^n}{(2k+1)!}x^{(2k+1)} + ... && (4.30)
\end{aligned}
$$

We can show the effects of the various terms:

```
var X: seq[float] = @[]
for i in 0 ..< 200: X.add 0.03 * float(i)

savePlot("images/sin.png", X, X.mapIt(sin(it)),
         title = "sin(x) approximations")
# Plot Taylor approximations on the same set of axes; in nlib.nim,
# `savePlot` accepts an optional list of additional curves.
```

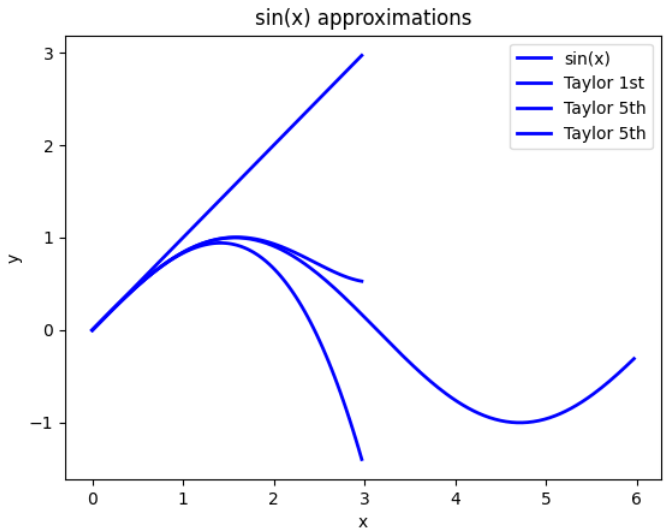

Figure 4.1: The figure shows the sin function and its approximation using the Taylor expansion around $x = 0$ at different orders.

Notice that we can very well expand in Taylor around any other point, for example, $\bar{x} = \pi/2$, and we get

$$sin(x) = 1 - \frac{1}{2}(x - \frac{\pi}{2})^2 + ... + \frac{(-1)^n}{(2k)!}(x - \frac{\pi}{2})^{(2k)} + ... \tag{4.31}$$

and a plot would show:

```
let a = PI / 2.0
var X: seq[float] = @[]
for i in 0 ..< 200: X.add 0.03 * float(i)
savePlot("images/sin2.png", X, X.mapIt(sin(it)),
         title = "sin(x) approximations")
# overlay Taylor expansions around `a` (orders 2, 4, 6) using
# the multi-curve form of `savePlot`.
```

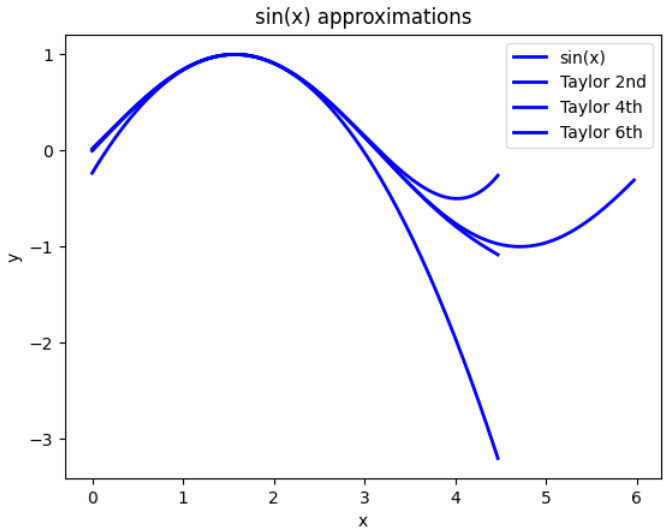

Figure 4.2: The figure shows the sin function and its approximation using the Taylor expansion around $x = \pi/2$ at different orders.

Similarly we can expand the cos function around $\bar{x} = 0$. Not accidentally, we would get the same coefficients as the Taylor expansion of the sin function around $\bar{x} = \pi/2$. In fact, $\sin(x) = \cos(x - \pi/2)$:

$$
\begin{aligned}
f(x) &= cos(x) && (4.32)\\
f^{(1)}(x) &= -sin(x) && (4.33)\\
f^{(2)}(x) &= -cos(x) && (4.34)\\
f^{(3)}(x) &= sin(x) && (4.35)\\
... &\quad ... && (4.36)\\
cos(x) &= 1 - \frac{1}{2}x^2 + ... + \frac{(-1)^n}{(2k)!}x^{(2k)} + ... && (4.37)
\end{aligned}
$$

With a simple replacement, it is easy to prove that

$$e^{ix} = \cos(x) + i\sin(x) \tag{4.38}$$

which will be useful when we talk about Fourier and Laplace transforms.

Now let's consider the kth term in Taylor expansion of e^x. It can be rearranged as a function of the previous $(k-1) - th$ term:

$$T_k(x) = \frac{1}{k!}x^n = \frac{x}{k}\frac{1}{(k-1)!}x^{k-1} = \frac{x}{k}T_{k-1}(x) \tag{4.39}$$

For $x < 0$, the terms in the sign have alternating sign and are decreasing in magnitude; therefore, for $x < 0$, $R_k < T_{k+1}(1)$. This allows for an easy implementation of the Taylor expansion and its stopping condition:

Listing 4.3: in file: `nlib/taylor.nim`

```
proc myexp*(x: float, precision = 1e-6, maxSteps = 40): float =
  if x == 0:
    return 1.0
  if x > 0:
    return 1.0 / myexp(-x, precision, maxSteps)
  var t = 1.0
  var s = 1.0
  for k in 1 ..< maxSteps:
    t = t * x / float(k)
    s = s + t
    if abs(t) < precision: return s
  raise newException(ArithmeticDefect, "no convergence")
```

This code presents all the features of many of the algorithms we see later in the chapter:

- It deals with the special case $e^0 = 1$.
- It reduces difficult problems to easier problems (exponential of a positive number to the exponential of a negative number via $e^x = 1/e^{-x}$).
- It approximates the "true" solution by iterations.
- The max number of iterations is limited.
- There is a stopping condition.
- It detects failure to converge.

Here is a test of its convergence:

```
for i in 0 ..< 10:
  let x = 0.1 * float(i)
  doAssert abs(myexp(x) - exp(x)) < 1e-4
```

We can do the same for the sin function:

$$T_k(x) = -\frac{x^2}{(2k)(2k+1)} T_{k-1}(x) \tag{4.40}$$

In this case, the residue is always limited by

$$|R_k| < |x^{2k+1}| \tag{4.41}$$

because the derivatives of sin are always sin and cos and their image is always between $[-1, 1]$.

Also notice that the stopping condition is only true when $0 \leq x < 1$. Therefore, for other values of x, we must use trigonometric relations to reduce the problem to a domain where the Taylor series converges.

Hence we write:

Listing 4.4: in file: `nlib/taylor.nim`

```
proc mysin*(x: float, precision = 1e-6, maxSteps = 40): float =
  if x == 0:
    return 0
  if x < 0:
    return -mysin(-x, precision, maxSteps)
  if x > 2.0 * PI:
    return mysin(x mod (2.0 * PI), precision, maxSteps)
  if x > PI:
```

```
    return -mysin(2.0 * PI - x, precision, maxSteps)
  if x > PI / 2:
    return mysin(PI - x, precision, maxSteps)
  if x > PI / 4:
    return sqrt(1.0 - mysin(PI / 2 - x, precision, maxSteps) ^ 2)
  var t = x
  var s = x
  for k in 1 ..< maxSteps:
    t = t * (-1.0) * x * x / float(2 * k) / float(2 * k + 1)
    s = s + t
    let r = x ^ (2 * k + 1)
    if r < precision: return s
  raise newException(ArithmeticDefect, "no convergence")
```

Here we test it:

```
for i in 0 ..< 10:
  let x = 0.1 * float(i)
  doAssert abs(mysin(x) - sin(x)) < 1e-4
```

Finally, we can do the same for the cos function:

Listing 4.5: in file: `nlib/taylor.nim`

```
proc mycos*(x: float, precision = 1e-6, maxSteps = 40): float =
  if x == 0:
    return 1.0
  if x < 0:
    return mycos(-x, precision, maxSteps)
  if x > 2.0 * PI:
    return mycos(x mod (2.0 * PI), precision, maxSteps)
  if x > PI:
    return mycos(2.0 * PI - x, precision, maxSteps)
  if x > PI / 2:
    return -mycos(PI - x, precision, maxSteps)
  if x > PI / 4:
    return sqrt(1.0 - mycos(PI / 2 - x, precision, maxSteps) ^ 2)
  var t = 1.0
  var s = 1.0
  for k in 1 ..< maxSteps:
    t = t * (-1.0) * x * x / float(2 * k) / float(2 * k - 1)
    s = s + t
    let r = x ^ (2 * k)
    if r < precision: return s
  raise newException(ArithmeticDefect, "no convergence")
```

Here is a test of convergence:

```
for i in 0 ..< 10:
  let x = 0.1 * float(i)
  doAssert abs(mycos(x) - cos(x)) < 1e-4
```

4.3.7 Stopping Conditions

To implement a stopping condition, we have two options. We can look at the absolute error, defined as

$$[\text{absolute error}] = [\text{approximate value}] - [\text{true value}] \tag{4.42}$$

or we can look at the relative error

$$[\text{relative error}] = [\text{absolute error}]/[\text{true value}] \tag{4.43}$$

or better, we can consider both. Here is an example of pseudo-code:

```
result = guess
loop:
    compute correction
    result = result+correction
    compute remainder
    if |remainder| < target_absolute_precision return result
    if |remainder| < target_relative_precision * |result| return result
```

In the code, we use the computed `result` as an estimate of the [true value] and, occasionally, the computed `correction` is an estimate of the [absolute error]. The target absolute precision is an input value that we use as an upper limit for the absolute error. The target relative precision is an input value we use as an upper limit for the relative error. When absolute error falls below the target absolute precision or the relative error falls below the target relative precision, we stop looping and assume the result is sufficiently precise:

```
proc genericLoopingFunction(guess: float, ap, rp: float, ns: int): float =
  result = guess
  for k in 0 ..< ns:
    let correction = ...   # compute correction
    result = result + correction
    let remainder = ...    # compute remainder
    if norm(remainder) < max(ap, norm(result) * rp): return result
  raise newException(ArithmeticDefect, "no convergence")
```

In the preceding code,

- `ap` is the target absolute precision.

- `rp` is the target relative precision.
- `ns` is the maximum number of steps.

From now on, we will adopt this naming convention.

Consider, for example, a financial algorithm that outputs a dollar amount. If it converges to a number very close to 1 or 0, the concept of relative precision loses significance for a result equal to zero, and the algorithm never detects convergence. In this case, setting an absolute precision of $1 or 1c is the right thing to do. Conversely, if the algorithm converges to a very large dollar amount, setting a precision of $1 or 1c may be a too strong requirement, and the algorithm will take too long to converge. In this case, setting a relative precision of 1% or 0.1% is the correct thing to do.

Because in general we do not know in advance the output of the algorithm, we should use both stopping conditions. We should also detect which of the two conditions causes the algorithm to stop looping and return, so that we can estimate the uncertainty in the result.

4.4 Linear algebra

In this section, we consider the following algorithms:

- Arithmetic operation among matrices
- Gauss–Jordan elimination for computing the inverse of a matrix A
- Cholesky decomposition for factorizing a symmetric positive definite matrix A into LL^t, where L is a lower triangular matrix
- The Jacobi algorithms for finding eigenvalues
- Fitting algorithms based on linear least squares

We will provide examples of applications.

4.4.1 Linear systems

In mathematics, a system described by a function f is linear if it is additive:

$$f(x+y) = f(x) + f(y) \tag{4.44}$$

and if it is homogeneous,

$$f(\alpha x) = \alpha f(x) \tag{4.45}$$

In simpler words, we can say that the output is proportional to the input.

As discussed in the introduction to this chapter, one of the simplest techniques for approximating any unknown system consists of approximating it with a linear system (and this approximation will be correct for some system and not for others).

When we try to model a new system, approximating the system with a linear system is often the first step in describing it in a quantitative way, even if it may turn out that this is not a good approximation.

This is the same as approximating the function f describing the system with the first-order Taylor expansions $f(x+h) - f(x) = f'(x)h$.

For a multidimensional system with input $\mathbf{x}$ (now a vector) and output $\mathbf{y}$ (also a vector, not necessarily of the same size as $\mathbf{x}$), we can still approximate $\mathbf{y} = f(\mathbf{x})$ with $f(\mathbf{y}+\mathbf{h}) - \mathbf{y} \simeq A\mathbf{h}$, yet we need to clarify what this latter equation means.

Given

$$\mathbf{x} \equiv \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix} \qquad \mathbf{y} \equiv \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_{m-1} \end{pmatrix} \tag{4.46}$$

$$A \equiv \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0,n-1} \\ a_{10} & a_{11} & \dots & a_{1,n-1} \\ \dots & \dots & \dots & \dots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,n-1} \end{pmatrix} \tag{4.47}$$

the following equation means

$$\mathbf{y} = f(\mathbf{x}) \simeq A\mathbf{x} \tag{4.48}$$

which means

$$\begin{aligned}
y_0 &= f_0((x) &&\simeq a_{00}x_0 + a_{01}x_1 + ... + a_{0,n-1}x_{n-1} && (4.49)\\
y_1 &= f_1((x) &&\simeq a_{10}x_0 + a_{11}x_1 + ... + a_{1,n-1}x_{n-1} && (4.50)\\
y_2 &= f_2((x) &&\simeq a_{20}x_0 + a_{21}x_1 + ... + a_{2,n-1}x_{n-1} && (4.51)\\
... &= &&... && (4.52)\\
y_{m-1} &= f_{m-1}((x) &&\simeq a_{m-1,0}x_0 + a_{m-1,1}x_1 + ...a_{m-1,n-1}x_{n-1} && (4.53)
\end{aligned}$$

which says that every output variable y_j is approximated with a function proportional to each of the input variables x_i.

A system is linear if the $\simeq$ relations turn out to be exact and can be replaced by $=$ symbols.

As a corollary of the basic properties of a linear system discussed earlier, linear systems have one nice additional property. If we combine two linear systems $y = Ax$ and $z = By$, the combined system is also a linear system $z = (BA)x$.

Elementary algebra is defined as a set of numbers (e.g., real numbers) endowed with the ordinary four elementary operations ($+$, $-$, $\times$, $/$).

Abstract algebra is a generalization of the concept of elementary algebra to other sets of objects (not necessarily numbers) by definition operations among them such as addition and multiplication.

Linear algebra is the extension of elementary algebra to matrices (and vectors, which can be seen as special types of matrices) by defining the four elementary operations among them.

We will implement them in code using Nim. A matrix can naively be stored as a sequence of sequences:

```
1 let a = @[@[1, 2, 3], @[4, 5, 6], @[7, 8, 9]]
```

But such an object (a list of lists) does not have the mathematical properties we want, so we have to define them.

First, we define a class representing a matrix:

Listing 4.6: in file: `nlib/matrix.nim`

```
type
  Matrix* = ref object
    nrows*, ncols*: int
    data*: seq[float]    # row-major

proc newMatrix*(rows: int, cols = 1, fill = 0.0): Matrix =
  ## A `rows x cols` matrix filled with `fill`.
  Matrix(nrows: rows, ncols: cols,
         data: newSeqWith(rows * cols, fill))

proc newMatrix*(rows, cols: int, fill: proc(r, c: int): float): Matrix =
  ## A `rows x cols` matrix where element (r, c) is `fill(r, c)`.
  result = Matrix(nrows: rows, ncols: cols,
                  data: newSeq[float](rows * cols))
  for r in 0 ..< rows:
    for c in 0 ..< cols:
      result.data[r * cols + c] = fill(r, c)

proc newMatrix*(rows: seq[seq[float]]): Matrix =
  ## Build from a list of rows.
  let n = rows.len
  let m = rows[0].len
  result = newMatrix(n, m)
  for r in 0 ..< n:
    for c in 0 ..< m:
      result.data[r * m + c] = rows[r][c]

proc newMatrix*(values: seq[float]): Matrix =
  ## Build a column vector from a flat list.
  newMatrix(values.len, 1, proc(r, c: int): float = values[r])
```

Notice that the constructor takes the number of rows and columns (cols) of the matrix but also a `fill` value, which can be used to initialize the matrix elements and defaults to zero. It can be callable in case we need to initialize the matrix with row, col dependent values.

Now we define a getter method, a setter method, and a string representation for the matrix elements:

Listing 4.7: in file: `nlib/matrix.nim`

```
proc `[]`*(a: Matrix, i, j: int): float =
  ## `x = a[i, j]`
  a.data[i * a.ncols + j]
```

```
4
5  proc `[]=`*(a: Matrix, i, j: int, value: float) =
6    ## `a[i, j] = value`
7    a.data[i * a.ncols + j] = value
8
9  proc tolist*(a: Matrix): seq[seq[float]] =
10   for r in 0 ..< a.nrows:
11     var row = newSeq[float](a.ncols)
12     for c in 0 ..< a.ncols: row[c] = a[r, c]
13     result.add row
14
15 proc `$`*(a: Matrix): string = $a.tolist()
16
17 proc flatten*(a: Matrix): seq[float] = a.data
18
19 proc reshape*(a: Matrix, n, m: int): Matrix =
20   if n * m != a.nrows * a.ncols:
21     raise newException(ValueError, "Impossible reshape")
22   let flat = a.data
23   newMatrix(n, m, proc(r, c: int): float = flat[r * m + c])
24
25 proc swapRows*(a: Matrix, i, j: int) =
26   for c in 0 ..< a.ncols:
27     swap(a.data[i * a.ncols + c], a.data[j * a.ncols + c])
```

We also define some convenience functions for constructing the identity matrix (given its size) and a diagonal matrix (given the diagonal elements).

Listing 4.8: in file: `nlib/matrix.nim`

```
1  proc identity*(rows = 1, e = 1.0): Matrix =
2    newMatrix(rows, rows,
3      proc(r, c: int): float = (if r == c: e else: 0.0))
4
5  proc diagonal*(d: seq[float]): Matrix =
6    newMatrix(d.len, d.len,
7      proc(r, c: int): float = (if r == c: d[r] else: 0.0))
```

Now we are ready to define arithmetic operations among matrices. We start with addition and subtraction:

Listing 4.9: in file: `nlib/matrix.nim`

```
1  proc `+`*(a, b: Matrix): Matrix =
2    ## Element-wise addition. Dimensions must match.
3    if a.nrows != b.nrows or a.ncols != b.ncols:
4      raise newException(ArithmeticDefect, "incompatible dimensions")
5    result = newMatrix(a.nrows, a.ncols)
```

```
  for i in 0 ..< a.data.len:
    result.data[i] = a.data[i] + b.data[i]

proc `+`*(a: Matrix, x: float): Matrix =
  ## `a + x`: scalar `x` is treated as `x * I` for square `a`,
  ## or as a constant column for column vectors.
  if a.nrows == a.ncols:
    return a + identity(a.nrows, x)
  if a.nrows == 1 or a.ncols == 1:
    return a + newMatrix(a.nrows, a.ncols, x)
  raise newException(ArithmeticDefect, "incompatible dimensions")

proc `+`*(x: float, a: Matrix): Matrix = a + x

proc `-`*(a: Matrix): Matrix =
  newMatrix(a.nrows, a.ncols,
    proc(r, c: int): float = -a[r, c])

proc `-`*(a, b: Matrix): Matrix = a + (-b)

proc `-`*(a: Matrix, x: float): Matrix = a + (-x)

proc `-`*(x: float, a: Matrix): Matrix = (-a) + x
```

With the preceding definitions, we can add matrices to matrices, subtract matrices from matrices, but also add and subtract scalars to and from matrices and vectors (scalars are interpreted as diagonal matrices when added to square matrices and as constant vectors when added to vectors).

Here are some examples:

```
let A = newMatrix(@[@[1.0, 2.0], @[3.0, 4.0]])
echo A + A          # @[@[2.0, 4.0], @[6.0, 8.0]]
echo A + 2.0        # @[@[3.0, 2.0], @[3.0, 6.0]]
echo A - 1.0        # @[@[0.0, 2.0], @[3.0, 3.0]]
echo -A             # @[@[-1.0, -2.0], @[-3.0, -4.0]]
echo 5.0 - A        # @[@[4.0, -2.0], @[-3.0, 1.0]]
let b = newMatrix(@[1.0, 2.0, 3.0])
echo b + 2.0        # @[@[3.0], @[4.0], @[5.0]]
```

For complex matrices we would parameterize `Matrix` on a numeric type `T` (e.g., `Matrix[Complex]`). For brevity, the implementation in this book uses `float` only; extending it to a generic element type is a straightforward exercise.

Now we implement multiplication. We are interested in three types of multiplication: multiplication of a scalar by a matrix, multiplication of a

matrix by a matrix, and scalar product between two vectors:

Listing 4.10: in file: `nlib/matrix.nim`

```
proc `*`*(x: float, a: Matrix): Matrix =
  ## Scalar times matrix.
  result = newMatrix(a.nrows, a.ncols)
  for i in 0 ..< a.data.len:
    result.data[i] = x * a.data[i]

proc `*`*(a: Matrix, x: float): Matrix = x * a

proc `*`*(a, b: Matrix): Matrix =
  ## Matrix product (or scalar product for two same-length column vectors).
  if a.ncols == 1 and b.ncols == 1 and a.nrows == b.nrows:
    var s = 0.0
    for r in 0 ..< a.nrows: s += a[r, 0] * b[r, 0]
    result = newMatrix(1, 1, s)
    return result
  if a.ncols != b.nrows:
    raise newException(ArithmeticDefect, "Incompatible dimension")
  result = newMatrix(a.nrows, b.ncols)
  for r in 0 ..< a.nrows:
    for c in 0 ..< b.ncols:
      var s = 0.0
      for k in 0 ..< a.ncols:
        s += a[r, k] * b[k, c]
      result[r, c] = s
```

This allows us the following operations:

```
let A = newMatrix(@[@[1.0, 2.0], @[3.0, 4.0]])
echo 2.0 * A        # scalar * matrix    @[@[2.0, 4.0], @[6.0, 8.0]]
echo A * A          # matrix * matrix    @[@[7.0, 10.0], @[15.0, 22.0]]
let b = newMatrix(@[1.0, 2.0, 3.0])
echo b * b          # scalar product (returns 1x1 matrix containing 14)
```

4.4.2 Examples of linear transformations

In this section, we try to provide an intuitive understanding of two-dimensional linear transformations.

In the following code, we consider an image (a set of points) containing a circle and two orthogonal axes. We then apply the following linear transformations to it:

- A_1, which scales the X-axis

- A_2, which scales the Y-axis
- S, which scales both axes
- B_1, which scales the X-axis and then rotates (R) the image 0.5 rad
- B_2, which is neither a scaling nor a rotation; as it can be seen from the image, it does not preserve angles

```
var points: seq[(float, float)] = @[]
for t in 0 ..< 200:
  points.add (cos(0.0628 * float(t)), sin(0.0628 * float(t)))
for t in 0 ..< 50: points.add (0.02 * float(t), 0.0)
for t in 0 ..< 50: points.add (0.0, 0.02 * float(t))

proc plotXY(filename: string, points: seq[(float, float)]) =
  savePlot(filename, points.mapIt(it[0]), points.mapIt(it[1]),
           title = "Linear Transformation")

plotXY("la1.png", points)
proc f(A: Matrix, points: seq[(float, float)], filename: string) =
  var data: seq[(float, float)] = @[]
  for (x, y) in points:
    data.add (A[0, 0] * x + A[0, 1] * y, A[1, 0] * x + A[1, 1] * y)
  plotXY(filename, points & data)

let A1 = newMatrix(@[@[0.2, 0.0], @[0.0, 1.0]]); f(A1, points, "la2.png")
let A2 = newMatrix(@[@[1.0, 0.0], @[0.0, 0.2]]); f(A2, points, "la3.png")
let S  = newMatrix(@[@[0.3, 0.0], @[0.0, 0.3]]); f(S, points, "la4.png")
let s = sin(0.5)
let c = cos(0.5)
let R  = newMatrix(@[@[c, -s], @[s, c]])
let B1 = R * A1;                                    f(B1, points, "la5.png")
let B2 = newMatrix(@[@[0.2, 0.4], @[0.5, 0.3]]);  f(B2, points, "la6.png")
```

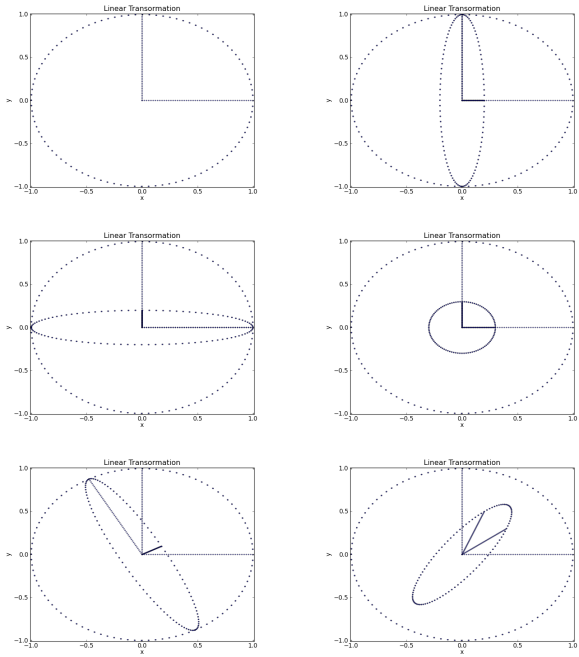

Figure 4.3: Example of the effect of different linear transformations on the same set of points. From left to right, top to bottom, they show stretching along both the X- and Y-axes, scaling across both axes, a rotation, and a generic transformation that does not preserve angles.

4.4.3 Matrix inversion and the Gauss–Jordan algorithm

Implementing the inverse of the multiplication (division) is a more challenging task.

We define A^{-1}, the inverse of the square matrix A, as that matrix such that for every vector b, $A(x) = \mathbf{b}$ implies $(x) = A^{-1}\mathbf{b}$. The Gauss–Jordan algorithm computes A^{-1} given A.

To implement it, we must first understand how it works. Consider the following equation:

$$A\mathbf{x} = \mathbf{b} \tag{4.54}$$

We can rewrite it as:

$$A\mathbf{x} = B\mathbf{b} \tag{4.55}$$

where $B = 1$, the identity matrix. This equation remains true if we multiply both terms by a nonsingular matrix S_0:

$$S_0 A\mathbf{x} = S_0 B\mathbf{b} \tag{4.56}$$

The trick of the Gauss–Jordan elimination consists in finding a series of matrices S_0, S_1, ..., S_{n-1} so that

$$S_{n-1}...S_1 S_0 A\mathbf{x} = S_{n-1}...S_1 S_0 B\mathbf{b} = \mathbf{x} \tag{4.57}$$

Because the preceding expression must be true for every b and because x is the solution of $Ax = b$, by definition, $S_{n-1} \dots S_1 S_0 B \equiv A^{-1}$.

The Gauss-Jordan algorithm works exactly this way: given A, it computes A^{-1}:

Listing 4.11: in file: `nlib/matrix.nim`

```
proc inv*(a0: Matrix, x = 1.0): Matrix =
  ## Returns x * a^-1 using Gauss-Jordan elimination.
  let n = a0.ncols
  if a0.nrows != n:
    raise newException(ArithmeticDefect, "matrix not squared")
  let a = newMatrix(a0.tolist())   # copy
  let b = identity(n, x)
  for c in 0 ..< n:
    for r in c + 1 ..< n:
      if abs(a[r, c]) > abs(a[c, c]):
        a.swapRows(r, c)
        b.swapRows(r, c)
    let p = a[c, c]
    for k in 0 ..< n:
      a[c, k] = a[c, k] / p
      b[c, k] = b[c, k] / p
    for r in 0 ..< n:
      if r == c: continue
      let pr = a[r, c]
      for k in 0 ..< n:
        a[r, k] = a[r, k] - a[c, k] * pr
        b[r, k] = b[r, k] - b[c, k] * pr
  result = b

proc `/`*(x: float, a: Matrix): Matrix = inv(a, x)
```

```
proc `/`*(a: Matrix, x: float): Matrix = (1.0 / x) * a
proc `/`*(a, b: Matrix): Matrix = a * (1.0 / b)
```

Here is an example, and we will see many more applications later:

```
let A = newMatrix(@[@[1.0, 2.0], @[4.0, 9.0]])
echo 1.0 / A    # @[@[9.0, -2.0], @[-4.0, 1.0]]
echo A / A      # @[@[1.0, 0.0], @[0.0, 1.0]]
echo A / 2.0    # @[@[0.5, 1.0], @[2.0, 4.5]]
```

4.4.4 Transposing a matrix

Another operation that we will need is transposition:

Listing 4.12: in file: `nlib/matrix.nim`

```
proc T*(a: Matrix): Matrix =
  ## Transpose of `a`.
  newMatrix(a.ncols, a.nrows,
    proc(r, c: int): float = a[c, r])
```

Notice the new matrix is defined with the number of rows and columns switched from matrix A. In Nim, a no-argument procedure can be called as either `a.T()` or `a.T` thanks to the uniform-call-syntax rule. This can be used as follows:

```
let A = newMatrix(@[@[1.0, 2.0], @[3.0, 4.0]])
echo A.T()      # @[@[1.0, 3.0], @[2.0, 4.0]]
```

For later use, we define two functions to check whether a matrix is symmetrical or zero within a given precision.

Another typical transformation for matrices of complex numbers is the Hermitian operation, which is a transposition combined with complex conjugation of the elements. Since our `Matrix` type stores `float` values, the Hermitian coincides with the transpose; for a complex `Matrix[Complex]` the same template would apply `conj` element-wise.

In later algorithms we will need to check whether a matrix is symmetrical (or almost symmetrical given precision) or zero (or almost zero):

Listing 4.13: in file: `nlib/matrix.nim`

```
proc isAlmostSymmetric*(a: Matrix, ap = 1e-6, rp = 1e-4): bool =
  if a.nrows != a.ncols: return false
  for r in 0 ..< a.nrows:
    for c in 0 ..< r:
```

```
      let delta = abs(a[r, c] - a[c, r])
      if delta > ap and delta > max(abs(a[r, c]), abs(a[c, r])) * rp:
        return false
  return true

proc isAlmostZero*(a: Matrix, ap = 1e-6, rp = 1e-4): bool =
  for r in 0 ..< a.nrows:
    for c in 0 ..< a.ncols:
      let delta = abs(a[r, c] - a[c, r])
      if delta > ap and delta > max(abs(a[r, c]), abs(a[c, r])) * rp:
        return false
  return true
```

4.4.5 Solving systems of linear equations

Linear algebra is fundamental for solving systems of linear equations such as the following:

$$x_0 + 2x_1 + 2x_2 = \quad 3 \tag{4.58}$$
$$4x_0 + 4x_1 + 2x_2 = \quad 6 \tag{4.59}$$
$$4x_0 + 6x_1 + 4x_2 = \quad 10 \tag{4.60}$$

This can be rewritten using the equivalent linear algebra notation:

$$A\mathrm{x} = \mathrm{b} \tag{4.61}$$

where

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{pmatrix} \quad \textit{and} \quad b = \begin{pmatrix} 3 \\ 6 \\ 10 \end{pmatrix} \tag{4.62}$$

The solution of the equation can now be written as

$$\mathrm{x} = A^{-1}\mathrm{b} \tag{4.63}$$

We can easily solve the system with our library:

```
let A = newMatrix(@[@[1.0, 2.0, 2.0], @[4.0, 4.0, 2.0], @[4.0, 6.0, 4.0]])
let b = newMatrix(@[3.0, 6.0, 10.0])
let x = (1.0 / A) * b
echo x          # @[@[-1.0], @[3.0], @[-1.0]]
```

Notice that b is a column vector and therefore

```
let b = newMatrix(@[3.0, 6.0, 10.0])
```

but not

```
let b = newMatrix(@[@[3.0, 6.0, 10.0]])  # wrong: row vector
```

We can also obtain a column vector by performing a transposition of a row vector:

```
let b = newMatrix(@[@[3.0, 6.0, 10.0]]).T()
```

4.4.6 Norm and condition number again

By norm of a vector, we often refer to the 2-norm defined using the Pythagoras theorem:

$$||x||_2 = \sqrt{\sum_i x_i^2} \tag{4.64}$$

For a vector, we can define the p-norm as a generalization of the 2-norm:

$$||x||_p \equiv \left(\sum_i abs(x_i)^p \right)^{\frac{1}{p}} \tag{4.65}$$

We can extend the notation of a norm to any function that maps a vector into a vector, as follows:

$$||f||_p \equiv \max_x ||f(x)||_p / ||x||_p \tag{4.66}$$

An immediate application is to functions implemented as linear transformations:

$$||A||_p \equiv \max_x ||Ax||_p / ||x||_p \tag{4.67}$$

This can be difficult to compute in the general case, but it reduces to a simple formula for the 1-norm:

$$||A||_1 \equiv \max_j \sum_i abs(A_{ij}) \tag{4.68}$$

The 2-norm is difficult to compute for a matrix, but the 1-norm provides an approximation. It is computed by adding up the magnitude of the elements per each column and finding the maximum sum.

This allows us to define a generic function to compute the norm of lists, matrices/vectors, and scalars:

Listing 4.14: in file: `nlib/linalg.nim`

```
proc norm*(x: float, p = 1): float = abs(x)

proc norm*(xs: seq[float], p = 1): float =
  var s = 0.0
  for x in xs: s += abs(x).pow(float(p))
  result = s.pow(1.0 / float(p))

proc norm*(a: Matrix, p = 1): float =
  if a.nrows == 1 or a.ncols == 1:
    var s = 0.0
    for r in 0 ..< a.nrows:
      for c in 0 ..< a.ncols:
        s += abs(a[r, c]).pow(float(p))
    return s.pow(1.0 / float(p))
  if p == 1:
    var best = 0.0
    for c in 0 ..< a.ncols:
      var s = 0.0
      for r in 0 ..< a.nrows: s += abs(a[r, c])
      if s > best: best = s
    return best
  raise newException(ValueError, "norm not implemented for p != 1 on matrix")
```

Now we can implement a function that computes the condition number for ordinary functions as well as for linear transformations represented by a matrix:

Listing 4.15: in file: `nlib/linalg.nim`

```
proc conditionNumber*(f: proc(x: float): float, x: float,
                      h = 1e-6): float =
  D(f, h)(x) * x / f(x)

proc conditionNumber*(a: Matrix): float =
  norm(a) * norm(1.0 / a)
```

Here are some examples:

```
proc f(x: float): float = x * x - 5.0 * x
echo formatFloat(conditionNumber(f, 1.0), ffDecimal, 2)
```

```
# 0.75
let A = newMatrix(@[@[1.0, 2.0], @[3.0, 4.0]])
echo formatFloat(conditionNumber(A), ffDecimal, 2)
# 21.0
```

Having the norm for matrices also allows us to extend the definition of convergence of a Taylor series to a series of matrices:

Listing 4.16: in file: `nlib/linalg.nim`

```
proc exp*(a: Matrix, ap = 1e-6, rp = 1e-4, ns = 40): Matrix =
  ## Matrix exponential by Taylor series.
  var t = identity(a.ncols)
  var s = identity(a.ncols)
  for k in 1 ..< ns:
    t = t * a * (1.0 / float(k))
    s = s + t
    if norm(t) < max(ap, norm(s) * rp):
      return s
  raise newException(ArithmeticDefect, "no convergence")
```

```
let A = newMatrix(@[@[1.0, 2.0], @[3.0, 4.0]])
echo exp(A)
# @[@[51.96..., 74.73...], @[112.10..., 164.07...]]
```

4.4.7 Cholesky factorization

A matrix is said to be positive definite if $x^t Ax > 0$ for every $x \neq 0$.

If a matrix is symmetric and positive definite, then there exists a lower triangular matrix L such that $A = LL^t$. A lower triangular matrix is a matrix that has zeros above the diagonal elements.

The Cholesky algorithm takes a matrix A as input and returns the matrix L:

Listing 4.17: in file: `nlib/linalg.nim`

```
proc cholesky*(a: Matrix): Matrix =
  ## Cholesky decomposition: returns lower-triangular `L`
  ## such that `L * L.T == a`.
  if not isAlmostSymmetric(a):
    raise newException(ArithmeticDefect, "not symmetric")
  let l = newMatrix(a.tolist())   # copy
  for k in 0 ..< l.ncols:
    if l[k, k] <= 0:
      raise newException(ArithmeticDefect, "not positive definite")
    let p = sqrt(l[k, k])
```

```
    l[k, k] = p
    for i in k + 1 ..< l.nrows:
      l[i, k] = l[i, k] / p
    for j in k + 1 ..< l.nrows:
      let pj = l[j, k]
      for i in k + 1 ..< l.nrows:
        l[i, j] = l[i, j] - pj * l[i, k]
  for i in 0 ..< l.nrows:
    for j in i + 1 ..< l.ncols:
      l[i, j] = 0
  result = l
```

Here we provide an example and a check that indeed $A = LL^t$:

```
let A = newMatrix(@[@[4.0, 2.0, 1.0],
                    @[2.0, 9.0, 3.0],
                    @[1.0, 3.0, 16.0]])
let L = cholesky(A)
echo isAlmostZero(A - L * L.T())   # true
```

The Cholesky algorithm fails if and only if the input matrix is not symmetric or not positive definite, therefore it can be used to check whether a symmetric matrix is positive definite.

Consider for example a generic covariance matrix A. It is supposed to be positive definite, but sometimes it is not, because it is computed incorrectly by taking different subsets of the data to compute A_{ij}, A_{jk}, and A_{ik}. The Cholesky algorithm provides an algorithm to check whether a matrix is positive definite:

Listing 4.18: in file: `nlib/linalg.nim`

```
proc isPositiveDefinite*(a: Matrix): bool =
  if not isAlmostSymmetric(a):
    return false
  try:
    discard cholesky(a)
    return true
  except Defect:        # cholesky raises ArithmeticDefect on indefinites
    return false
  except CatchableError:
    return false
```

Another application of the Cholesky is in generating vectors $\mathbf{x}$ with probability distribution

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}\mathbf{x}^t A^{-1}\mathbf{x}\right) \tag{4.69}$$

where A is a symmetric and positive definite matrix. In fact, if $A = LL^t$, then

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}(L^{-1}\mathbf{x})^t(L^{-1}\mathbf{x})\right) \tag{4.70}$$

and with a change of variable $\mathbf{u} = \mathbf{L^{-1}x}$, we obtain

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}\mathbf{u}^t\mathbf{u}\right) \tag{4.71}$$

and

$$p(\mathbf{x}) \propto e^{-\frac{1}{2}u_0^2}e^{-\frac{1}{2}u_1^2}e^{-\frac{1}{2}u_2^2}... \tag{4.72}$$

Therefore the u_i components are Gaussian random variables.

In summary, given a covariance matrix A, we can generate random vectors x or random numbers with the same covariance simply by doing

Listing 4.19: in file: `nlib/finance.nim`

```
iterator randomList*(a: Matrix): seq[float] =
  ## Yields successive random vectors with covariance matrix `a`.
  let l = cholesky(a)
  while true:
    var u = newMatrix(l.nrows, 1)
    for c in 0 ..< l.nrows: u[c, 0] = gauss(0.0, 1.0)
    yield (l * u).flatten()
```

Here is an example of how to use it:

```
let A = newMatrix(@[@[1.0, 0.1], @[0.2, 3.0]])
for v in randomList(A):
  echo v
```

The `randomList` is a Nim iterator. The `yield` keyword is comparable to `return`. For more information on iterators, consult a Nim language guide.

4.4.8 Modern portfolio theory

Modern portfolio theory [24] is an investment approach that tries to maximize return given a fixed risk. Many different metrics have been proposed. One of them is the *Sharpe ratio*.

For a stock or a portfolio with an average return r and risk σ, the Sharpe ratio is defined as

$$\text{Sharpe}(r, \sigma) \equiv (r - \bar{r})/\sigma \tag{4.73}$$

Here $\bar{r}$ is the current risk-free investment rate. Usually the risk is measured as the standard deviation of its daily (or monthly or yearly) return.

Consider the stock price p_{it} of stock i at time t and its arithmetic daily return $r_{it} = (p_{i,t+1} - p_{it})/p_{it}$ given a risk-free interest equal to $\bar{r}$.

For each stock, we can compute the average return and the risk (the standard deviation, i.e., the square root of the variance of daily returns) and display it in a risk-return plot as we did in chapter 2.

We can try to build arbitrary portfolios by investing in multiple stocks at the same time. Modern portfolio theory states that there is a maximum Sharpe ratio and there is one portfolio that corresponds to it. It is called the tangency portfolio.

A portfolio is identified by fractions of \$1 invested in each stock in the portfolio. Our goal is to determine the tangent portfolio.

If we assume that daily returns for the stocks are Gaussian, then the solving algorithm is simple.

All we need is to compute the average return for each stock, defined as

$$r_i = 1/T \sum_t r_{it} \tag{4.74}$$

and the covariance matrix

$$A_{ij} = \frac{1}{T} \sum_t (r_{it} - r_i)(r_{jt} - r_j) \tag{4.75}$$

Modern portfolio theory tells us that the tangent portfolio is given by

$$\mathbf{x} = A^{-1}(\mathbf{r} - \bar{r}\mathbf{1}) \tag{4.76}$$

The inputs of the formula are the covariance matrix (A), a vector or arithmetic returns for the assets in the portfolio (r), the risk free rate ($\bar{r}$). The output is a vector (x) whose elements are the percentages to be invested

in each asset to obtain a tangency portfolio. Notice that some elements of x can be negative and this corresponds to short position (sell, not buy, the asset).

Here is the algorithm:

Listing 4.20: in file: `nlib/finance.nim`

```
proc markowitz*(mu, a: Matrix, rFree: float):
                (seq[float], float, float) =
  ## Markowitz tangency portfolio. Returns (weights, return, risk).
  var x = (1.0 / a) * (mu - rFree)
  var s = 0.0
  for r in 0 ..< x.nrows: s += x[r, 0]
  x = x / s
  var portfolio: seq[float] = @[]
  for r in 0 ..< x.nrows: portfolio.add x[r, 0]
  let ret = (mu.T * x)[0, 0]
  let risk = sqrt((x.T * (a * x))[0, 0])
  (portfolio, ret, risk)
```

Here is an example. We consider three assets (0, 1, 2) with the following covariance matrix:

```
let cov = newMatrix(@[@[0.04, 0.006, 0.02],
                      @[0.006, 0.09, 0.06],
                      @[0.02, 0.06, 0.16]])
```

and the following expected returns (arithmetic returns, not log returns: arithmetic returns are additive across the assets of a portfolio at a fixed time, so the portfolio return is the weighted average of the asset returns; log returns do not have this property, which is why MPT is naturally formulated in terms of arithmetic returns):

```
let mu = newMatrix(@[0.10, 0.12, 0.15])
```

Given the risk-free interest rate

```
let rFree = 0.05
```

we compute the tangent portfolio (highest Sharpe ratio), its return, and its risk with one function call:

```
let (x, ret, risk) = markowitz(mu, cov, rFree)
echo x
# @[0.5566343042071198, 0.27508090614886727, 0.16828478964401297]
echo ret, " ", risk
# 0.113915857605 0.186747095412
echo (ret - rFree) / risk
```

```
# 0.34225891152
for r in 0 ..< 3:
  echo (mu[r, 0] - rFree) / sqrt(cov[r, r])
# 0.25
# 0.233333333333
# 0.25
```

Investing 55% in asset 0, 27% in asset 1, and 16% in asset 2, the resulting portfolio has an expected return of 11.39% and a risk of 18.67%, which corresponds to a Sharpe ratio of 0.34, much higher than 0.25, 0.23, and 0.23 for the individual assets.

Notice that the tangency portfolio is not the only one with the highest Sharpe ratio (return for unit of risk). In fact, any linear combination of the tangency portfolio with a risk-free asset (putting money in the bank) has the same Sharpe ratio. For any target risk, one can find a linear combination of the risk-free asset and the tangent portfolio that has a better Sharpe ratio than any other possible portfolio comprising the same assets.

If we call α the fraction of the money to invest in the tangency portfolio and $1 - \alpha$ the fraction to keep in the bank at the risk free rate, the resulting combined portfolio has return:

$$\alpha \mathbf{x} \cdot \mathbf{r} + (1 - \alpha)\bar{r} \tag{4.77}$$

and risk

$$\alpha \sqrt{\mathbf{x}^t A \mathbf{x}} \tag{4.78}$$

We can determine α by deciding how much risk we are willing to take, and these formulas tell us the optimal portfolio for that amount of risk.

4.4.9 Linear least squares, χ^2

Consider a set of data points $(x_J, y_j) = (t_j, o_j \pm do_j)$. We want to fit them with a linear combination of linear independent functions f_i so that

$$
\begin{aligned}
c_0 f_0(t_0) + c_1 f_1(t_0) + c_2 f_2(t_0) + ... &= e_0 \simeq o_0 \pm do_0 && (4.79)\\
c_0 f_0(t_1) + c_1 f_1(t_1) + c_2 f_2(t_1) + ... &= e_1 \simeq o_1 \pm do_1 && (4.80)\\
c_0 f_0(t_2) + c_1 f_1(t_2) + c_2 f_2(t_2) + ... &= e_2 \simeq o_2 \pm do_2 && (4.81)\\
... &= ... && (4.82)
\end{aligned}
$$

We want to find the $\{c_i\}$ that minimizes the sum of the squared distances between the actual "observed" data o_j and the predicted "expected" data e_j, in units of do_j. This metric is called χ^2 in general [25]. An algorithm that minimizes the χ^2 and is linear in the c_i coefficients (our case here) is called *linear least squares* or *linear regression*.

$$\chi^2 = \sum_j \left| \frac{e_j - o_j}{do_j} \right|^2 \tag{4.83}$$

If we define the matrix A and B as

$$A = \begin{pmatrix} \frac{f_0(t_0)}{do_0} & \frac{f_1(t_0)}{do_0} & \frac{f_2(t_0)}{do_0} & ... \\ \frac{f_0(t_1)}{do_1} & \frac{f_1(t_1)}{do_1} & \frac{f_2(t_1)}{do_1} & ... \\ \frac{f_0(t_2)}{do_2} & \frac{f_1(t_2)}{do_2} & \frac{f_2(t_2)}{do_2} & ... \\ ... & ... & ... & ... \end{pmatrix} \qquad b = \begin{pmatrix} \frac{o_0}{do_0} \\ \frac{o_1}{do_1} \\ \frac{o_2}{do_2} \\ ... \end{pmatrix} \tag{4.84}$$

then the problem is reduced to

$$
\begin{aligned}
\min_{\mathbf{c}} \chi^2 &= \min_{\mathbf{c}} |A\mathbf{c} - \mathbf{b}|^2 && (4.85)\\
&= \min_{\mathbf{c}} (A\mathbf{c} - \mathbf{b})^t (A\mathbf{c} - \mathbf{b}) && (4.86)\\
&= \min_{\mathbf{c}} (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{b}^t A\mathbf{c} + \mathbf{b}^t\mathbf{b}) && (4.87)
\end{aligned}
$$

This is the same as solving the following equation:

$$
\begin{aligned}
\nabla_c (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{c}^t A^t\mathbf{b} + \mathbf{b}^t\mathbf{b}) &= 0 && (4.88)\\
A^t A\mathbf{c} - A^t\mathbf{b} &= 0 && (4.89)
\end{aligned}
$$

Its solution is

$$\mathbf{c} = (A^t A)^{-1}(A^t \mathbf{b}) \tag{4.90}$$

The following algorithm implements a fitting function based on the preceding procedure. It takes as input a list of functions f_i and a list of points $p_j = (t_j, o_j, do_j)$ and returns three objects—a list with the c coefficients, the value of χ^2 for the best fit, and the fitting function:

Listing 4.21: in file: `nlib/fitting.nim`

```
type
  FitFunc* = proc(x: float): float

proc fitLeastSquares*(points: seq[(float, float, float)],
                      f: seq[FitFunc]):
                      (seq[float], float, proc(x: float): float) =
  ## Linear least-squares fit of `points = (x, y, dy)` to
  ## `\sum_j c_j f[j](x)`. Returns (coefficients, chi2, fit function).
  let n = points.len
  let m = f.len
  let a = newMatrix(n, m)
  let b = newMatrix(n, 1)
  for i in 0 ..< n:
    let weight = 1.0 / points[i][2]
    b[i, 0] = weight * points[i][1]
    for j in 0 ..< m:
      a[i, j] = weight * f[j](points[i][0])
  let c = (1.0 / (a.T * a)) * (a.T * b)
  let chi = a * c - b
  let chi2Val = norm(chi, 2) ^ 2
  let cs = c.flatten()
  let fittingF = proc(x: float): float =
    var s = 0.0
    for i in 0 ..< f.len: s += f[i](x) * cs[i]
    s
  (cs, chi2Val, fittingF)

# examples of fitting functions
proc polynomial*(n: int): seq[FitFunc] =
  proc monomial(power: int): FitFunc =
    # Wrap closure creation in a separate proc so each closure
    # captures its own `power` parameter (rather than sharing a
    # single mutable loop variable).
    result = proc(x: float): float = x ^ power
  for p in 0 .. n:
```

```
    result.add monomial(p)

let CONSTANT*  = polynomial(0)
let LINEAR*    = polynomial(1)
let QUADRATIC* = polynomial(2)
let CUBIC*     = polynomial(3)
let QUARTIC*   = polynomial(4)
```

As an example, we can use it to perform a polynomial fit: given a set of points, we want to find the coefficients of a polynomial that best approximate those points.

In other words, we want to find the c_i such that, given t_j and o_j,

$$c_0 + c_1 t_0^1 + c_2 t_0^2 + ... = e_0 \simeq o_0 \pm do_0 \quad (4.91)$$
$$c_0 + c_1 t_1^1 + c_2 t_1^2 + ... = e_1 \simeq o_1 \pm do_1 \quad (4.92)$$
$$c_0 + c_1 t_2^1 + c_2 t_2^2 + ... = e_2 \simeq o_2 \pm do_2 \quad (4.93)$$
$$... \quad ... \quad (4.94)$$
$$(4.95)$$

Here is how we can generate some random points and solve the problem for a polynomial of degree 2 (or quadratic fit):

```
var points: seq[(float, float, float)] = @[]
for k in 0 ..< 100:
  let kf = float(k)
  points.add (kf, 5.0 + 0.8 * kf + 0.3 * kf * kf + sin(kf), 2.0)
let (a, chi2, fittingF) = fitLeastSquares(points, QUADRATIC)
for p in points[^10 .. ^1]:
  echo p[0], " ",
       formatFloat(p[1], ffDecimal, 2), " ",
       formatFloat(fittingF(p[0]), ffDecimal, 2)
# 90 2507.89 2506.98
# ...

# Render the first 10 observations with their error bars and the
# fitted polynomial through them.
savePlot("images/polynomialfit.png",
         points[0 ..< 10].mapIt(it[0]),
         points[0 ..< 10].mapIt(fittingF(it[0])),
         title = "polynomial fit", xlab = "t", ylab = "e(t), o(t)")
```

Fig. 4.4.9 is a plot of the first 10 points compared with the best fit:

We can also define a $\chi^2_{dof} = \chi^2/(N-1)$ where N is the number of c

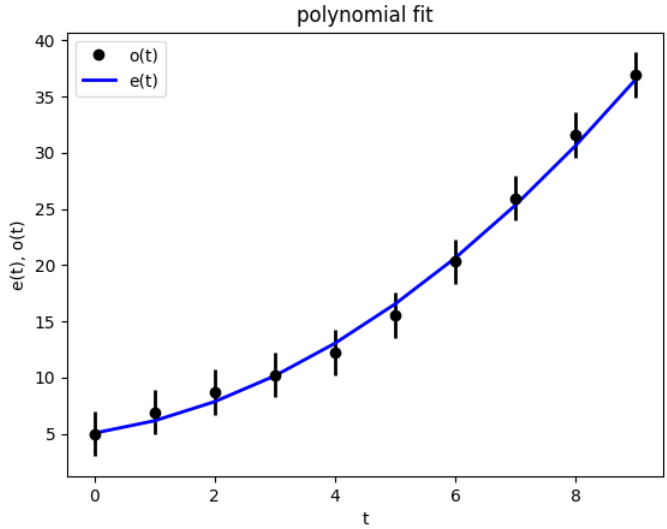

Figure 4.4: Random data with their error bars and the polynomial best fit.

parameters determined by the fit. A value of $\chi^2_{dof} \simeq 1$ indicates a good fit. In general, the smaller χ^2_{dof}, the better the fit. A large value of χ^2_{dof} is a symptom of poor modeling (the assumptions of the fit are wrong), whereas a value χ^2_{dof} much smaller than 1 is a symptom of an overestimate of the errors do_j (or perhaps manufactured data).

4.4.10 Trading and technical analysis

In finance, *technical analysis* is an empirical discipline that consists of forecasting the direction of prices through the study of patterns in historical data (in particular, price and volume). As an example, we implement a simple strategy that consists of the following steps:

- We fit the adjusted closing price for the previous seven days and use our fitting function to predict the adjusted close for the next day.
- If we have cash and predict the price will go up, we buy the stock.
- If we hold the stock and predict the price will go down, we sell the

stock.

Listing 4.22: in file: nlib/finance.nim

```
type
  Trader* = ref object

proc model*(t: Trader, window: seq[float]): float =
  ## Fit the last few days quadratically and extrapolate tomorrow's price.
  var points: seq[(float, float, float)] = @[]
  for i, v in window:
    points.add (float(i), v, 1.0)
  let (_, _, fittingF) = fitLeastSquares(points, QUADRATIC)
  fittingF(float(points.len))

proc strategy*(t: Trader, history: seq[float], ndays = 7): string =
  if history.len < ndays: return ""
  let todayClose = history[^1]
  let tomorrowPrediction = t.model(history[^ndays .. ^1])
  if tomorrowPrediction > todayClose: "buy" else: "sell"

proc simulate*(t: Trader, data: seq[float],
               cash = 1000.0, shares = 0.0,
               dailyRate = 0.03 / 360.0): float =
  var cash = cash
  var shares = shares
  for tIdx in 0 ..< data.len:
    let suggestion = t.strategy(data[0 ..< tIdx])
    let todayClose = data[max(tIdx - 1, 0)]
    if cash > 0 and suggestion == "buy":
      let sharesBought = float(int(cash / todayClose))
      shares += sharesBought
      cash -= sharesBought * todayClose
    elif shares > 0 and suggestion == "sell":
      cash += shares * todayClose
      shares = 0.0
    cash *= exp(dailyRate)
  cash + shares * data[^1]
```

Now we back test the strategy using fake financial data. We can generate fake financial data with this function:

Listing 4.23: in file: nlib/finance.nim

```
proc fakeStockPrices*(startPrice = 100.0, averageReturn = 0.05,
                      volatility = 0.30, days = 100): seq[float] =
  let dailyVolatility = volatility / sqrt(250.0)
  let dailyReturn = averageReturn / 250.0
  var s: seq[float] = @[]
  for _ in 0 ..< days - 1: s.add gauss(0.0, dailyVolatility)
```

```
  var sumS = 0.0
  for v in s: sumS += v
  let mu = dailyReturn - sumS / float(days - 1)
  var v = @[startPrice]
  for item in s:
    v.add v[^1] * exp(mu + item)
  result = v
```

```
let initialCash = 1000.0
let data = fakeStockPrices(averageReturn = 0.05)
let finalCash = Trader().simulate(data, cash = initialCash)
let naiveProfit = initialCash * (exp(0.05) - 1.0)
let strategyProfit = finalCash - initialCash
```

If `strategy_profit` > `naive_profit` then our strategy outperforms the naive buy and hold strategy.

Of course, we can always engineer a strategy based on historical data that will outperform holding the stock, but *past performance is never a guarantee of future performance*.

According to the definition from investopedia.com, "technical analysts believe that the historical performance of stocks and markets is an indication of future performance."

The efficacy of both technical and fundamental analysis is disputed by the efficient-market hypothesis, which states that stock market prices are essentially unpredictable [26].

It is easy to extend the previous class to implement other strategies and back test them.

4.4.11 Eigenvalues and the Jacobi algorithm

Given a matrix A, an eigenvector is defined as a vector $\mathbf{x}$ such that $A\mathbf{x}$ is proportional to $\mathbf{x}$. The proportionality factor is called an eigenvalue, e. One matrix may have many eigenvectors $\mathbf{x}_i$ and associated eigenvalues e_i:

$$A\mathbf{x}_i = e_i\mathbf{x}_i \tag{4.96}$$

For example:

$$A = \begin{pmatrix} 1 & -2 \\ 1 & 4 \end{pmatrix} \qquad and \qquad x_i = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \tag{4.97}$$

$$\begin{pmatrix} 1 & -2 \\ 1 & 4 \end{pmatrix} * \begin{pmatrix} -1 \\ 1 \end{pmatrix} \qquad = \qquad 3 * \begin{pmatrix} -1 \\ 1 \end{pmatrix} \tag{4.98}$$

In this case, x_i is an eigenvector and the corresponding eigenvalue is $e = 3$.

Some eigenvalues may be zero ($e_i = 0$), which means the matrix A is singular. A matrix is singular if it maps a nonzero vector into zero.

Given a square matrix A, if the space generated by the linear independent eigenvalues has the same dimensionality as the number of rows (or columns) of A, then its eigenvalues are real and the matrix can we written as

$$A = UDU^t \tag{4.99}$$

where D is a diagonal matrix with eigenvalues on the diagonal $D_{ii} = e_i$ and U is a matrix whose column i is the $\mathbf{x}_i$ eigenvalue.

The following algorithm is called the Jacobi algorithm. It takes as input a symmetric matrix A and returns the matrix U and a list of corresponding eigenvalues e, sorted from smallest to largest:

Listing 4.24: in file: `nlib/linalg.nim`

```
proc jacobiEigenvalues*(a: Matrix): (Matrix, seq[float]) =
  ## Returns (U, e) so that `a == U * diagonal(e) * U.T`.
  ## The i-th column of U is the eigenvector for eigenvalue `e[i]`.
  proc maxind(m: Matrix, k: int): int =
    var j = k + 1
    for i in k + 2 ..< m.ncols:
      if abs(m[k, i]) > abs(m[k, j]): j = i
    j
  let n = a.nrows
  if n != a.ncols:
    raise newException(ArithmeticDefect, "matrix not squared")
  let s = newMatrix(a.tolist())
  let eMat = identity(n)
```

```
  var state = n
  var ind = newSeq[int](n)
  var e = newSeq[float](n)
  var changed = newSeq[bool](n)
  for k in 0 ..< n:
    ind[k] = maxind(s, k)
    e[k] = s[k, k]
    changed[k] = true
  while state > 0:
    var m = 0
    for k in 1 ..< n - 1:
      if abs(s[k, ind[k]]) > abs(s[m, ind[m]]): m = k
    let k = m
    let h = ind[m]
    let p = s[k, h]
    var y = (e[h] - e[k]) / 2.0
    var t = abs(y) + sqrt(p * p + y * y)
    var sv = sqrt(p * p + t * t)
    var c = t / sv
    var ss = p / sv
    t = p * p / t
    if y < 0:
      ss = -ss
      t = -t
    s[k, h] = 0
    y = e[k]
    e[k] = y - t
    if changed[k] and y == e[k]:
      changed[k] = false; dec state
    elif (not changed[k]) and y != e[k]:
      changed[k] = true; inc state
    y = e[h]
    e[h] = y + t
    if changed[h] and y == e[h]:
      changed[h] = false; dec state
    elif (not changed[h]) and y != e[h]:
      changed[h] = true; inc state
    for i in 0 ..< k:
      let a1 = c * s[i, k] - ss * s[i, h]
      let a2 = ss * s[i, k] + c * s[i, h]
      s[i, k] = a1; s[i, h] = a2
    for i in k + 1 ..< h:
      let a1 = c * s[k, i] - ss * s[i, h]
      let a2 = ss * s[k, i] + c * s[i, h]
      s[k, i] = a1; s[i, h] = a2
    for i in h + 1 ..< n:
      let a1 = c * s[k, i] - ss * s[h, i]
      let a2 = ss * s[k, i] + c * s[h, i]
      s[k, i] = a1; s[h, i] = a2
```

```
    for i in 0 ..< n:
      let a1 = c * eMat[k, i] - ss * eMat[h, i]
      let a2 = ss * eMat[k, i] + c * eMat[h, i]
      eMat[k, i] = a1; eMat[h, i] = a2
    ind[k] = maxind(s, k)
    ind[h] = maxind(s, h)
  # sort vectors
  for i in 1 ..< n:
    var j = i
    while j > 0 and e[j - 1] > e[j]:
      swap(e[j], e[j - 1])
      eMat.swapRows(j, j - 1)
      dec j
  # normalize vectors
  let u = newMatrix(n, n)
  for i in 0 ..< n:
    var s2 = 0.0
    for j in 0 ..< n: s2 += eMat[i, j] ^ 2
    let nrm = sqrt(s2)
    for j in 0 ..< n: u[j, i] = eMat[i, j] / nrm
  (u, e)
```

Here is an example that shows, for a particular case, the relation between the input, A, of the output of the U, e of the Jacobi algorithm:

```
randomize()
let A = newMatrix(4, 4)
for r in 0 ..< A.nrows:
  for c in r ..< A.ncols:
    A[r, c] = gauss(10.0, 10.0)
    A[c, r] = A[r, c]
let (U, e) = jacobiEigenvalues(A)
echo isAlmostZero(U * diagonal(e) * U.T() - A)   # true
```

Eigenvalues can be used to filter noise out of data and find hidden dependencies in data. Following are some examples.

4.4.12 Principal component analysis

One important application of the Jacobi algorithm is for principal component analysis (PCA). This is a mathematical procedure that converts a set of observations of possibly correlated vectors into a set of uncorrelated vectors called *principal components*.

Here we consider, as an example, the time series of the adjusted arithmetic returns for the S&P100 stocks that we downloaded and stored in chapter

2.

Each time series is a vector. We know they are not independent because there are correlations. Our goal is to model each time series as a vector plus noise where the vector is the same for all series. We also want to find that vector that has maximal superposition with the individual time series, the principal component.

First, we compute the correlation matrix for all the stocks. This is a non-trivial task because we have to make sure that we only consider those days when all stocks were traded.

Listing 4.25: in file: `nlib/linalg.nim`

```
proc computeCorrelationMatrix*(v: seq[seq[float]]): Matrix =
  ## Pearson correlation matrix of `v` (rows are series).
  let m = v.len
  let n = v[0].len
  var mus = newSeq[float](m)
  var vars = newSeq[float](m)
  for i in 0 ..< m:
    var s = 0.0
    for k in 0 ..< n: s += v[i][k]
    mus[i] = s / float(n)
    var s2 = 0.0
    for k in 0 ..< n: s2 += v[i][k] ^ 2
    vars[i] = s2 / float(n) - mus[i] ^ 2
  result = newMatrix(m, m,
    proc(i, j: int): float =
      var s = 0.0
      for k in 1 ..< n: s += v[i][k] * v[j][k]
      (s / float(n) - mus[i] * mus[j]) / sqrt(vars[i] * vars[j]))
```

We use the preceding function to compute the correlation and pass it as input to the Jacobi algorithm and plot the output eigenvalues:

```
var stocks: seq[seq[float]] = @[]
for _ in 0 ..< 5: stocks.add fakeStockPrices()
let corr = computeCorrelationMatrix(stocks)
let (U, e) = jacobiEigenvalues(corr)
savePlot("images/correlations.png",
         (0 ..< e.len).toSeq().mapIt(float(it)),
         e, title = "Fake Stock Correlations",
         xlab = "i", ylab = "e[i]")
```

The image shows that one eigenvalue, the last one, is much larger than the others. It tells us that the data series have something in common. In

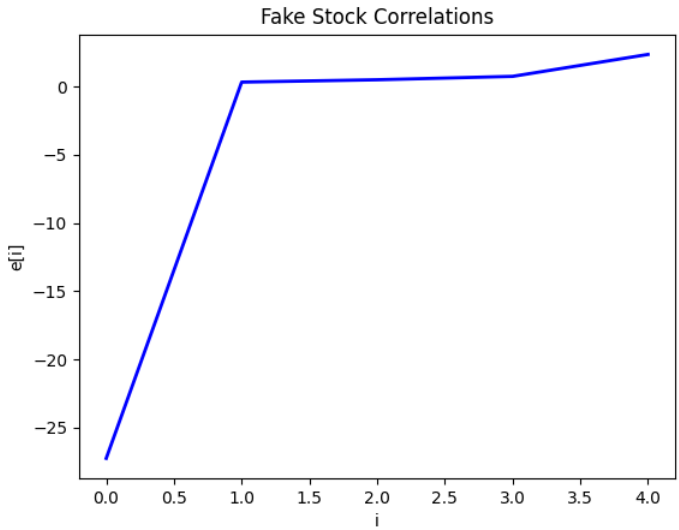

Figure 4.5: Eigenvalues of the correlation matrix for 5 fake stocks, sorted by their magnitude.

fact, the arithmetic returns for stock j at time t can be written as

$$r_{it} = \beta_i p_t + \alpha_{it} \tag{4.100}$$

where p is the principal component given by

$$p_t = \sum_i U_{n-1,j} r_{jt} \tag{4.101}$$

$$\beta_i = \sum_t r_{it} p_t \tag{4.102}$$

$$\alpha_{it} = r_{it} - \beta_i p_t \tag{4.103}$$

Here $\mathbf{p}$ is the vector of adjusted arithmetic returns that better correlates with the returns of the individual assets and therefore best represents the market. The β_i coefficient tells us how much $\mathbf{r}_i$ overlaps with $\mathbf{p}$; α, at first approximation, measures leftover noise.

4.5 Sparse matrix inversion

Sometimes we have to invert matrices that are very large, and the Gauss-Jordan algorithm fails. Yet, if the matrix is sparse, in the sense that most of its elements are zeros, then two algorithms come to our rescue: the *minimum residual* and the *biconjugate gradient* (for which we consider a variant called the *stabilized bi-conjugate gradient*).

We will also assume that the matrix to be inverted is given in some implicit algorithmic as $\mathbf{y} = f(\mathbf{x})$ because this is always the case for sparse matrices. There is no point to storing all its elements because most of them are zero.

4.5.1 Minimum residual

Given a linear operator f, the Krylov space spanned by a vector x is defined as

$$K(f, y, i) = \{y, f(y), f(f(y)), f(f(f(y))), (f^i)(y)\} \tag{4.104}$$

The *minimum residual* [27] algorithm works by solving $x = f^{-1}(y)$ iteratively. At each iteration, it computes a new orthogonal basis vector q_i for the Krylov space $K(f, y, i)$ and computes the coefficients α_i that project x_i into component i of the Krylov space:

$$x_i = y + \alpha_1 q_1 + \alpha_2 q_2 + ... + \alpha_i q_i \in K(f, y, i+1) \tag{4.105}$$

which minimizes the norm of the residue defined as:

$$r = f(x_i) - y \tag{4.106}$$

Therefore $\lim_{i\to\infty} f(x_i) = y$. If a solution to the original problem exists, ignoring precision issues, the minimum residual converges to it, and the residue decreases at each iteration.

Notice that in the following code, x and y are exchanged because we adopt the convention that y is the output and x is the input:

Listing 4.26: in file: `nlib/linalg.nim`

```
proc invertMinimumResidual*(f: proc(x: Matrix): Matrix,
                            x: Matrix,
                            ap = 1e-4, rp = 1e-4, ns = 200): Matrix =
  ## Minimum residual iterative solver for `f(x) = y` (sparse linear ops).
  var y = newMatrix(x.tolist())
  var r = x - f(x)
  for k in 0 ..< ns:
    let q = f(r)
    let alpha = (q.T * r)[0, 0] / (q.T * q)[0, 0]
    y = y + alpha * r
    r = r - alpha * q
    let residue = sqrt((r.T * r)[0, 0] / float(r.nrows))
    if residue < max(ap, norm(y) * rp): return y
  raise newException(ArithmeticDefect, "no convergence")
```

4.5.2 Stabilized biconjugate gradient

The stabilized biconjugate gradient [28] method is also based on constructing a Krylov subspace and minimizing the same residue as in the minimum residual algorithm, yet it is faster than the minimum residual and has a smoother convergence than other conjugate gradient methods:

Listing 4.27: in file: `nlib/linalg.nim`

```
proc invertBicgstab*(f: proc(x: Matrix): Matrix,
                     x: Matrix,
                     ap = 1e-4, rp = 1e-4, ns = 200): Matrix =
  ## Stabilized bi-conjugate gradient iterative solver.
  var y = newMatrix(x.tolist())
  var r = x - f(x)
  let q = newMatrix(r.tolist())
  var p = newMatrix(r.nrows, 1)
  var s = newMatrix(r.nrows, 1)
  var rhoOld = 1.0
  var alpha = 1.0
  var omega = 1.0
  for k in 0 ..< ns:
    let rho = (q.T * r)[0, 0]
    let beta = (rho / rhoOld) * (alpha / omega)
    rhoOld = rho
    p = beta * p + r - (beta * omega) * s
    s = f(p)
    alpha = rho / (q.T * s)[0, 0]
    r = r - alpha * s
    let tt = f(r)
    omega = (tt.T * r)[0, 0] / (tt.T * tt)[0, 0]
    y = y + omega * r + alpha * p
```

```
    let residue = sqrt((r.T * r)[0, 0] / float(r.nrows))
    if residue < max(ap, norm(y) * rp): return y
    r = r - omega * tt
  raise newException(ArithmeticDefect, "no convergence")
```

Notice that the minimum residual and the stabilized biconjugate gradient, if they converge, converge to the same value.

As an example, consider the following. We take a picture using a camera, but we take the picture out of focus. The image is represented by a set of m^2 pixels. The defocusing operation can be modeled as a first approximation with a linear operator acting on the "true" image, x, and turning it into an "out of focus" image, y. We can store the pixels in a one-dimensional vector (both for x and y) as opposed to a matrix by mapping the pixel (r, c) into vector component i using the relation $(r, c) = (i/m, i\%m)$.

Hence we can write

$$\mathbf{y} = A\mathbf{x} \tag{4.107}$$

Here the linear operator A represents the effects of the lens, which transforms one set of pixels into another.

We can model the lens as a sequence of β smearing operators:

$$A = S^{\beta} \tag{4.108}$$

where a smearing operator is a next neighbor interaction among pixels:

$$S_{ij} = (1 - \alpha/4)\delta_{i,j} + \alpha\delta_{i,j\pm1} + \alpha\delta_{i,j\pm m} \tag{4.109}$$

Here α and β are smearing coefficients. When $\alpha = 0$ or $\beta = 0$, the lens has no effect, and $A = I$. The value of α controls how much the value of light at point i is averaged with the value at its four neighbor points: left $(j - 1)$, right $(j + 1)$, top $(j + m)$, and bottom $(j - m)$. The coefficient β determines the width of the smearing radius. The larger the values of β and α, the more out of focus is the original image.

In the following code, we generate an image x and filter it through a lens operator `smear`, obtaining a smeared image y. We then use the sparse matrix inverter to reconstruct the original image x given the smeared image y. We use the `color2d` plotting function to represent the images:

```
let m = 30
let x = newMatrix(m * m, 1,
  proc(r, c: int): float =
    if r div m in [10, 20] or r mod m in [10, 20]: 1.0 else: 0.0)

proc smear(x: Matrix): Matrix =
  let alpha = 0.4
  let beta = 8
  var x = x
  for _ in 0 ..< beta:
    let y = newMatrix(x.nrows, 1)
    for r in 0 ..< m:
      for c in 0 ..< m:
        y[r * m + c, 0] = (1.0 - alpha / 4.0) * x[r * m + c, 0]
        if c < m - 1: y[r * m + c, 0] += alpha * x[r * m + c + 1, 0]
        if c > 0:     y[r * m + c, 0] += alpha * x[r * m + c - 1, 0]
        if r < m - 1: y[r * m + c, 0] += alpha * x[r * m + c + m, 0]
        if r > 0:     y[r * m + c, 0] += alpha * x[r * m + c - m, 0]
    x = y
  result = x

let y = smear(x)
let z = invertMinimumResidual(smear, y, ns = 1000)
saveHeatmap("images/defocused.png", y.reshape(m, m).tolist(),
            title = "Defocused image")
saveHeatmap("images/refocused.png", z.reshape(m, m).tolist(),
            title = "refocus image")
```

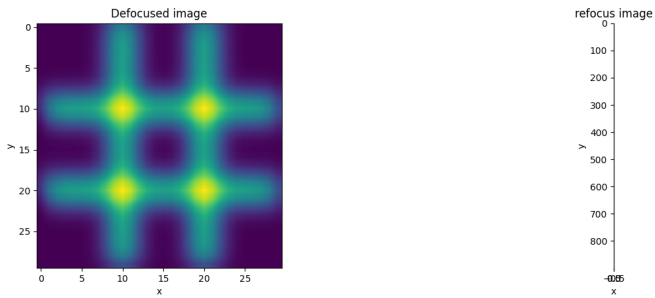

Figure 4.6: An out-of-focus image (left) and the original image (image) computed from the out-of-focus one, using sparse matrix inversion.

When the Hubble telescope was first put into orbit, its mirror was not installed properly and caused the telescope to take pictures out of focus. Until the defect was physically corrected, scientists were able to fix the images using a similar algorithm.

4.6 Solvers for nonlinear equations

In this chapter, we are concerned with the problem of solving in x the equation of one variable:

$$f(x) = 0 \tag{4.110}$$

4.6.1 Fixed-point method

It is always possible to reformulate $f(x) = 0$ as $g(x) = x$ using, for example, one of the following definitions:

- $g(x) = f(x)/c + x$ for some constant c
- $g(x) = f(x)/q(x) + x$ for some $q(x) > 0$ at the solution of $f(x) = 0$

We start at x_0, an arbitrary point in the domain, and close to the solution we seek. We compute

$$\begin{aligned} x_1 &= g(x_0) && (4.111)\\ x_2 &= g(x_1) && (4.112)\\ x_3 &= g(x_2) && (4.113)\\ ... &= ... && (4.114) \end{aligned}$$

We can compute the distance between x_i and x as

$$\begin{aligned} |x_i - x| &= |g(x_{i-1}) - g(x)| && (4.115)\\ &= |g(x) + g''(\xi)(x_{i-1} - x) - g(x)| && (4.116)\\ &= |g''(\xi)||x_{i-1} - x| && (4.117) \end{aligned}$$

where we use *de l"Hopital rule* and ξ is a point in between x and x_{i-1}.

If the magnitude of the first derivative of g, $|g''|$, is less than 1 in a neighborhood of x, and if x_0 is in such a neighborhood, then

$$|x_i - x| = |g''(\xi)||x_{i-1} - x| < |x_{i-1} - x| \tag{4.118}$$

The x_i series will get closer and closer to the solution x.

Here is the process implemented into an algorithm:

Listing 4.28: in file: `nlib/solvers.nim`

```
proc solveFixedPoint*(f: proc(x: float): float, x0: float,
                      ap = 1e-6, rp = 1e-4, ns = 100): float =
  let g = proc(x: float): float = f(x) + x
  let Dg = D(g)
  var x = x0
  for k in 0 ..< ns:
    if abs(Dg(x)) >= 1:
      raise newException(ArithmeticDefect, "error D(g)(x)>=1")
    let xOld = x
    x = g(x)
    if k > 2 and norm(xOld - x) < max(ap, norm(x) * rp):
      return x
  raise newException(ArithmeticDefect, "no convergence")
```

And here is an example:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0) / 10.0
echo formatFloat(solveFixedPoint(f, 1.0, rp = 0.0), ffDecimal, 4)
# 2.0000
```

4.6.2 Bisection method

The goal of the bisection [29] method is to solve $f(x) = 0$ when the function is continuous and it is known to change sign in between $x = a$ and $x = b$. The bisection method is the continuous equivalent of the binary search algorithm seen in chapter 3. The algorithm iteratively finds the middle point of the domain $x = (b + a)/2$, evaluates the function there, and decides whether the solution is on the left or the right, thus reducing the size of the domain from (a, b) to (a, x) or (x, b), respectively:

Listing 4.29: in file: `nlib/solvers.nim`

```
proc solveBisection*(f: proc(x: float): float, a0, b0: float,
                     ap = 1e-6, rp = 1e-4, ns = 100): float =
```

```
  var a = a0
  var b = b0
  var fa = f(a)
  var fb = f(b)
  if fa == 0: return a
  if fb == 0: return b
  if fa * fb > 0:
    raise newException(ArithmeticDefect,
                       "f(a) and f(b) must have opposite sign")
  for k in 0 ..< ns:
    let x = (a + b) / 2.0
    let fx = f(x)
    if fx == 0 or norm(b - a) < max(ap, norm(x) * rp):
      return x
    if fx * fa < 0:
      b = x; fb = fx
    else:
      a = x; fa = fx
  raise newException(ArithmeticDefect, "no convergence")
```

Here is how to use it:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(solveBisection(f, 1.0, 3.0), ffDecimal, 4)
# 2.0000
```

4.6.3 Newton method

The Newton [30] algorithm also solves $f(x) = 0$. It is faster (on average) than the bisection method because it makes the additional assumption that the function is also differentiable. This algorithm starts from an arbitrary point x_0 and approximates the function at that point with its first-order Taylor expansion

$$f(x) \simeq f(x_0) + f'(x_0)(x - x_0) \tag{4.119}$$

and solves it exactly:

$$f(x) = 0 \to x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4.120}$$

thus finding a new and better estimate for the solution. The algorithm iterates the preceding equation, and when it converges, it approximates the exact solution better and better:

Listing 4.30: in file: nlib/solvers.nim

```
proc solveNewton*(f: proc(x: float): float, x0: float,
                  ap = 1e-6, rp = 1e-4, ns = 20): float =
  var x = x0
  for k in 0 ..< ns:
    let fx = f(x)
    let Dfx = D(f)(x)
    if norm(Dfx) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    x = x - fx / Dfx
    if k > 2 and norm(x - xOld) < max(ap, norm(x) * rp):
      return x
  raise newException(ArithmeticDefect, "no convergence")
```

The algorithm is guaranteed to converge if $|f'(x)| > 1$ in some neighborhood of the solution and if the starting point is in this neighborhood. It may also converge if this condition is not true. It is likely to fail when $|f'(x)| \simeq 0$ is in the neighborhood of the solution or the starting point because the terms `fx/Dfx` would become very large.

Here is an example:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(solveNewton(f, 1.0), ffDecimal, 4)
# 2.0000
```

4.6.4 Secant method

The secant method is very similar to the Newton method, except that $f'(x)$ is replaced by a numerical estimate computed using the current point x and the previous point visited by the algorithm:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{4.121}$$

$$x_{i+i} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.122}$$

As the algorithm approaches the exact solution, the numerical derivative becomes a better and better approximation for the derivative:

Listing 4.31: in file: nlib/solvers.nim

```
proc solveSecant*(f: proc(x: float): float, x0: float,
                  ap = 1e-6, rp = 1e-4, ns = 20): float =
```

```
  var x = x0
  var fx = f(x)
  var Dfx = D(f)(x)
  for k in 0 ..< ns:
    if norm(Dfx) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    let fxOld = fx
    x = x - fx / Dfx
    if k > 2 and norm(x - xOld) < max(ap, norm(x) * rp):
      return x
    fx = f(x)
    Dfx = (fx - fxOld) / (x - xOld)
  raise newException(ArithmeticDefect, "no convergence")
```

Here is an example:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(solveSecant(f, 1.0), ffDecimal, 4)
# 2.0000
```

4.7 Optimization in one dimension

While a solver is an algorithm that finds x such that $f(x) = 0$, an optimization algorithm is one that finds the maximum or minimum of the function $f(x)$. If the function is differentiable, this is achieved by solving $f'(x) = 0$.

For this reason, if the function is differentiable twice, we can simply rename all previous solvers and replace $f(x)$ with $f'(x)$ and $f'(x)$ with $f''(x)$.

4.7.1 Bisection method

Listing 4.32: in file: `nlib/solvers.nim`

```
proc optimizeBisection*(f: proc(x: float): float, a, b: float,
                        ap = 1e-6, rp = 1e-4, ns = 100): float =
  solveBisection(D(f), a, b, ap, rp, ns)
```

Here is an example:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(optimizeBisection(f, 2.0, 5.0), ffDecimal, 4)
# 3.5000
```

4.7.2 Newton method

Listing 4.33: in file: nlib/solvers.nim

```
proc optimizeNewton*(f: proc(x: float): float, x0: float,
                     ap = 1e-6, rp = 1e-4, ns = 20): float =
  var x = x0
  let f1 = D(f)
  let f2 = DD(f)
  for k in 0 ..< ns:
    let fx = f1(x)
    let Dfx = f2(x)
    if Dfx == 0: return x
    if norm(Dfx) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    x = x - fx / Dfx
    if norm(x - xOld) < max(ap, norm(x) * rp): return x
  raise newException(ArithmeticDefect, "no convergence")
```

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(optimizeNewton(f, 3.0), ffDecimal, 3)
# 3.500
```

4.7.3 Secant method

As in the Newton case, the secant method can also be used to find extrema, by replacing f with f':

Listing 4.34: in file: nlib/solvers.nim

```
proc optimizeSecant*(f: proc(x: float): float, x0: float,
                     ap = 1e-6, rp = 1e-4, ns = 100): float =
  var x = x0
  let f1 = D(f)
  let f2 = DD(f)
  var fx = f1(x)
  var Dfx = f2(x)
  for k in 0 ..< ns:
    if fx == 0: return x
    if norm(Dfx) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    let fxOld = fx
    x = x - fx / Dfx
    if norm(x - xOld) < max(ap, norm(x) * rp): return x
    fx = f1(x)
    Dfx = (fx - fxOld) / (x - xOld)
  raise newException(ArithmeticDefect, "no convergence")
```

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(optimizeSecant(f, 3.0), ffDecimal, 3)
# 3.500
```

4.7.4 Golden section search

If the function we want to optimize is continuous but not differentiable, then the previous algorithms do not work. In this case, there is one algorithm that comes to our rescue, the golden section [31] search. It is similar to the bisection method, with one caveat; in the bisection method, at each point, we need to know if a function changes sign in between two points, therefore two points are all we need. If instead we are looking for a max or min, we need to know if the function is concave or convex in between those two points. This requires one extra point in between the two. So while the bisection method only needs one point in between $[a, b]$, the golden search needs two points, x_1 and x_2, in between $[a, b]$, and from them it can determine whether the extreme is in $[a, x_2]$ or in $[x_1, b]$. This is also represented pictorially in fig. 4.7.4. The two points are chosen in an optimal way so that at the next iteration, one of the two points can be recycled by leaving the ratio between $x_1 - a$ and $b - x_2$ fixed and equal to 1:

Listing 4.35: in file: `nlib/solvers.nim`

```
proc optimizeGoldenSearch*(f: proc(x: float): float, a0, b0: float,
                           ap = 1e-6, rp = 1e-4, ns = 100): float =
  var a = a0
  var b = b0
  let tau = (sqrt(5.0) - 1.0) / 2.0
  var x1 = a + (1.0 - tau) * (b - a)
  var x2 = a + tau * (b - a)
  var fa = f(a); var f1 = f(x1); var f2 = f(x2); var fb = f(b)
  for k in 0 ..< ns:
    if f1 > f2:
      a = x1; fa = f1; x1 = x2; f1 = f2
      x2 = a + tau * (b - a); f2 = f(x2)
    else:
      b = x2; fb = f2; x2 = x1; f2 = f1
      x1 = a + (1.0 - tau) * (b - a); f1 = f(x1)
    if k > 2 and norm(b - a) < max(ap, norm(b) * rp): return b
  raise newException(ArithmeticDefect, "no convergence")
```

Here is an example:

```
proc f(x: float): float = (x - 2.0) * (x - 5.0)
echo formatFloat(optimizeGoldenSearch(f, 2.0, 5.0), ffDecimal, 3)
# 3.500
```

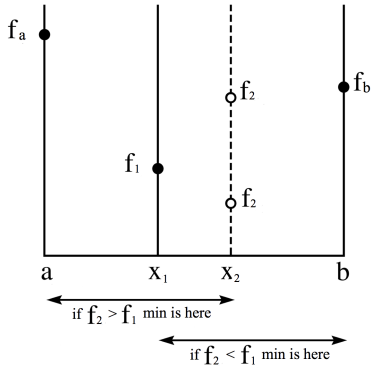

Figure 4.7: Pictorial representation of the golden search method. If the function is concave ($f''(x) > 0$), then knowledge of the function in 4 points (a, x_1, x_2, b) permits us to determine whether a minimum is between $[a, x_2]$ or between $[x_1, b]$.

.

4.8 Functions of many variables

To be able to work with functions of many variables, we need to introduce the concept of the partial derivative:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + \mathbf{h}_i) - f(\mathbf{x} - \mathbf{h}_i)}{2h} \tag{4.123}$$

where $\mathbf{h}_i$ is a vector with components all equal to zero but $h_i = h > 0$.

We can implement it as follows:

Listing 4.36: in file: `nlib/calculus.nim`

```
proc partial*(f: proc(x: seq[float]): float, i: int, h = 1e-4):
              proc(x: seq[float]): float =
  result = proc(x: seq[float]): float =
    var x = x
    x[i] += h
    let fPlus = f(x)
    x[i] -= 2.0 * h
    let fMinus = f(x)
```

```
    (fPlus - fMinus) / (2.0 * h)

proc partial*(f: proc(x: seq[float]): seq[float], i: int, h = 1e-4):
              proc(x: seq[float]): seq[float] =
  result = proc(x: seq[float]): seq[float] =
    var x = x
    x[i] += h
    let fPlus = f(x)
    x[i] -= 2.0 * h
    let fMinus = f(x)
    result = newSeq[float](fPlus.len)
    for k in 0 ..< fPlus.len:
      result[k] = (fPlus[k] - fMinus[k]) / (2.0 * h)
```

Similarly to `D(f)`, we have implemented it in such a way that `partial(f, i)` returns a function that can be evaluated at any point x. Also notice that the function f may return a scalar, a matrix, a list, or a tuple. The `if` condition allows the function to deal with the difference between two lists or tuples.

Here is an example:

```
proc f(x: seq[float]): float = 2.0*x[0] + 3.0*x[1] + 5.0*x[1]*x[2]
let df0 = partial(f, 0)
let df1 = partial(f, 1)
let df2 = partial(f, 2)
let x = @[1.0, 1.0, 1.0]
echo (formatFloat(df0(x), ffDecimal, 4),
      formatFloat(df1(x), ffDecimal, 4),
      formatFloat(df2(x), ffDecimal, 4))
# ("2.0000", "8.0000", "5.0000")
```

4.8.1 Jacobian, gradient, and Hessian

A generic function $f(x_0, x_1, x_2, ...)$ of multiple variables $\mathbf{x} = (x_0, x_1, x_2, ..)$ can be expanded in Taylor series to the second order as

$$f(x_0, x_1, x_2, ...) = f(\bar{x}_0, \bar{x}_1, \bar{x}_2, ...) + \tag{4.124}$$

$$\sum_i \frac{\partial f(\bar{\mathbf{x}})}{\partial x_i}(x_i - \bar{x}_i) + \tag{4.125}$$

$$\sum_{ij} \frac{1}{2}\frac{\partial^2 f}{\partial x_i \partial x_j}(\bar{\mathbf{x}})(x_i - \bar{x}_i)(x_j - \bar{x}_j) + ... \tag{4.126}$$

We can rewrite the above expression in terms of the vector $\mathbf{x}$ as follows:

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^t H_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + ... \tag{4.127}$$

where we introduce the gradient vector

$$\nabla_f(x) \equiv \begin{pmatrix} \partial f(x)/\partial x_0 \\ \partial f(x)/\partial x_1 \\ \partial f(x)/\partial x_2 \\ ... \end{pmatrix} \tag{4.128}$$

and the Hessian matrix

$$H_f(x) \equiv \begin{pmatrix} \partial^2 f(x)/\partial x_0 \partial x_0 & \partial^2 f(x)/\partial x_0 \partial x_1 & \partial^2 f(x)/\partial x_0 \partial x_2 & ... \\ \partial^2 f(x)/\partial x_1 \partial x_0 & \partial^2 f(x)/\partial x_1 \partial x_1 & \partial^2 f(x)/\partial x_1 \partial x_2 & ... \\ \partial^2 f(x)/\partial x_2 \partial x_0 & \partial^2 f(x)/\partial x_2 \partial x_1 & \partial^2 f(x)/\partial x_2 \partial x_2 & ... \\ ... & ... & ... & ... \end{pmatrix} \tag{4.129}$$

Given the definition of partial, we can compute the gradient and the Hessian using the two functions

Listing 4.37: in file: `nlib/calculus.nim`

```
proc gradient*(f: proc(x: seq[float]): float,
               x: seq[float], h = 1e-4): Matrix =
  newMatrix(x.len, 1,
    proc(r, c: int): float = partial(f, r, h)(x))

proc hessian*(f: proc(x: seq[float]): float,
              x: seq[float], h = 1e-4): Matrix =
  newMatrix(x.len, x.len,
    proc(r, c: int): float = partial(partial(f, r, h), c, h)(x))
```

Here is an example:

```
proc f(x: seq[float]): float = 2.0*x[0] + 3.0*x[1] + 5.0*x[1]*x[2]
echo gradient(f, @[1.0, 1.0, 1.0])
# @[@[1.999999...], @[7.999999...], @[4.999999...]]
echo hessian(f, @[1.0, 1.0, 1.0])
# @[@[0.0, 0.0, 0.0], @[0.0, 0.0, 5.000000...], @[0.0, 5.000000..., 0.0]]
```

When dealing with functions returning multiple values like

$$f(\mathbf{x}) = (f_0(\mathbf{x}), f_1(\mathbf{x}), f_2(\mathbf{x}), ...) \tag{4.130}$$

we need to Taylor expand each component:

$$f(\mathbf{x}) = \begin{pmatrix} f_0(\mathbf{x}) \\ f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ ... \end{pmatrix} = \begin{pmatrix} f_0(\bar{\mathbf{x}}) + \nabla_{f_0}(\mathbf{x} - \bar{\mathbf{x}}) + ... \\ f_1(\bar{\mathbf{x}}) + \nabla_{f_1}(\mathbf{x} - \bar{\mathbf{x}}) + ... \\ f_2(\bar{\mathbf{x}}) + \nabla_{f_2}(\mathbf{x} - \bar{\mathbf{x}}) + ... \\ ... \end{pmatrix} \tag{4.131}$$

which we can rewrite as

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + J_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + ... \tag{4.132}$$

where J_f is called Jacobian and is defined as

$$J_f \equiv \begin{pmatrix} \partial f_0(x)/\partial x_0 & \partial f_0(x)/\partial x_1 & \partial f_0(x)/\partial x_2 & ... \\ \partial f_1(x)/\partial x_0 & \partial f_1(x)/\partial x_1 & \partial f_1(x)/\partial x_2 & ... \\ \partial f_2(x)/\partial x_0 & \partial f_2(x)/\partial x_1 & \partial f_2(x)/\partial x_2 & ... \\ ... & ... & ... & ... \end{pmatrix} \tag{4.133}$$

which we can implement as follows:

Listing 4.38: in file: `nlib/calculus.nim`

```
proc jacobian*(f: proc(x: seq[float]): seq[float],
               x: seq[float], h = 1e-4): Matrix =
  var partials: seq[seq[float]] = @[]
  for c in 0 ..< x.len:
    partials.add partial(f, c, h)(x)
  newMatrix(partials[0].len, x.len,
    proc(r, c: int): float = partials[c][r])
```

Here is an example:

```
proc f(x: seq[float]): seq[float] =
  @[2.0*x[0] + 3.0*x[1] + 5.0*x[1]*x[2], 2.0*x[0]]
echo jacobian(f, @[1.0, 1.0, 1.0])
# @[@[1.9999999..., 7.999999..., 4.9999999...], @[1.9999999..., 0.0, 0.0]]
```

4.8.2 Newton method (solver)

We can now solve eq. 4.132 iteratively as we did for the one-dimensional Newton solver with only one change—the first derivative of f is replaced by the Jacobian:

Listing 4.39: in file: `nlib/solvers.nim`

```
proc solveNewtonMulti*(f: proc(x: seq[float]): seq[float],
                       x0: seq[float],
                       ap = 1e-6, rp = 1e-4, ns = 20): seq[float] =
  ## Multidimensional Newton solver. Finds `x` such that `f(x) == 0`.
  var x = newMatrix(x0)
  for k in 0 ..< ns:
    let fx = newMatrix(f(x.flatten()))
    let J = jacobian(f, x.flatten())
    if norm(J) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    x = x - (1.0 / J) * fx
    if k > 2 and norm(x - xOld) < max(ap, norm(x) * rp):
      return x.flatten()
  raise newException(ArithmeticDefect, "no convergence")
```

Here is an example:

```
proc f(x: seq[float]): seq[float] =
  @[x[0] + x[1], x[0] + x[1] ^ 2 - 2.0]
echo solveNewtonMulti(f, @[0.0, 0.0])
# @[1.0..., -1.0...]
```

4.8.3 Newton method (optimize)

As for the one-dimensional case, we can approximate $f(\mathbf{x})$ with its Taylor expansion at the first order,

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^t H_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) \tag{4.134}$$

set its derivative to zero, and solve it, thus obtaining

$$\mathbf{x} = \bar{\mathbf{x}} - H_f^{-1}\nabla_f \tag{4.135}$$

which constitutes the core of the multidimensional Newton optimizer:

Listing 4.40: in file: `nlib/solvers.nim`

```
proc optimizeNewtonMulti*(f: proc(x: seq[float]): float,
                          x0: seq[float],
                          ap = 1e-6, rp = 1e-4, ns = 20): seq[float] =
  ## Multidimensional Newton optimizer. Finds extremum of `f`.
  var x = newMatrix(x0)
  for k in 0 ..< ns:
    let grad = gradient(f, x.flatten())
    let H = hessian(f, x.flatten())
    if norm(H) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    let xOld = x
    x = x - (1.0 / H) * grad
    if k > 2 and norm(x - xOld) < max(ap, norm(x) * rp):
      return x.flatten()
  raise newException(ArithmeticDefect, "no convergence")
```

```
proc f(x: seq[float]): float = (x[0] - 2.0) ^ 2 + (x[1] - 3.0) ^ 2
echo optimizeNewtonMulti(f, @[0.0, 0.0])
# @[2.0, 3.0]
```

4.8.4 Improved Newton method (optimize)

We can further improve the Newton multidimensional optimizer by using the following technique. At each step, if the next guess does not reduce the value of f, we revert to the previous point, and we perform a one-dimensional Newton optimization along the direction of the gradient. This method greatly increases the stability of the multidimensional Newton optimizer:

Listing 4.41: in file: `nlib/solvers.nim`

```
proc optimizeNewtonMultiImproved*(
    f: proc(x: seq[float]): float, x0: seq[float],
    ap = 1e-6, rp = 1e-4, ns = 20, hStart = 10.0): seq[float] =
  ## Newton optimizer with line-search fallback to steepest descent.
  var x = newMatrix(x0)
  var fx = f(x.flatten())
  var h = hStart
  for k in 0 ..< ns:
    let grad = gradient(f, x.flatten())
    let H = hessian(f, x.flatten())
    if norm(H) < ap:
      raise newException(ArithmeticDefect, "unstable solution")
    var fxOld = fx
    var xOld = x
    x = x - (1.0 / H) * grad
    fx = f(x.flatten())
```

```
    while fx > fxOld:
      fx = fxOld; x = xOld
      let normGrad = norm(grad)
      xOld = x
      x = x - grad * (h / normGrad)
      fxOld = fx
      fx = f(x.flatten())
      h = h / 2.0
    h = norm(x - xOld) * 2.0
    if k > 2 and h / 2.0 < max(ap, norm(x) * rp):
      return x.flatten()
  raise newException(ArithmeticDefect, "no convergence")
```

4.9 Nonlinear fitting

Finally, we have all the ingredients to implement a very generic fitting function that will work for linear and nonlinear least squares.

Here we consider a generic experiment or simulated experiment that generates points of the form $(x_i, y_i \pm \delta y_i)$. Our goal is to minimize the χ^2 defined as

$$\chi^2(\mathbf{a}, \mathbf{b}) = \sum_i \left| \frac{y_i - f(x_i, \mathbf{a}, \mathbf{b})}{\delta y_i} \right|^2 \tag{4.136}$$

where the function f is known but depends on unknown parameters $\mathbf{a} = (a_0, a_1, ...)$ and $\mathbf{b} = (b_0, b_1, ...)$. In terms of these parameters, the function f can be written as follows:

$$f(x, \mathbf{a}, \mathbf{b}) = \sum_j a_j f_j(x, \mathbf{b}) \tag{4.137}$$

Here is an example:

$$f(x, \mathbf{a}, \mathbf{b}) = a_0 e^{-b_0 x} + a_1 e^{-b_1 x} + a_2 e^{-b_2 x} + ... \tag{4.138}$$

The goal of our algorithm is to efficiently determine the parameters $\mathbf{a}$ and $\mathbf{b}$ that minimize the χ^2.

We proceed by defining the following two quantities:

$$\mathbf{z} = \begin{pmatrix} y_0 \ / \ \delta y_0 \\ y_1 \ / \ \delta y_1 \\ y_2 \ / \ \delta y_2 \\ \dots \end{pmatrix} \tag{4.139}$$

and

$$A(\mathbf{b}) = \begin{pmatrix} f_0(x_0,\mathbf{b})/\delta y_0 & f_1(x_0,\mathbf{b})/\delta y_0 & f_2(x_0,\mathbf{b})/\delta y_0 & \dots \\ f_0(x_1,\mathbf{b})/\delta y_1 & f_1(x_1,\mathbf{b})/\delta y_1 & f_2(x_1,\mathbf{b})/\delta y_1 & \dots \\ f_0(x_2,\mathbf{b})/\delta y_2 & f_1(x_2,\mathbf{b})/\delta y_2 & f_2(x_2,\mathbf{b})/\delta y_2 & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} \tag{4.140}$$

In terms of A and z, the χ^2 can be rewritten as

$$\chi^2(\mathbf{a},\mathbf{b}) = |A(\mathbf{b})\mathbf{a} - \mathbf{z}|^2 \tag{4.141}$$

We can minimize this function in a using the linear least squares algorithm, exactly:

$$\mathbf{a}(\mathbf{b}) = (A(\mathbf{b})A(\mathbf{b})^t)^{-1}(A(\mathbf{b})^t z) \tag{4.142}$$

We define a function that returns the minimum χ^2 for a fixed input $\mathbf{b}$:

$$g(\mathbf{b}) = \min_{\mathbf{a}} \chi^2(\mathbf{a},\mathbf{b}) = \chi^2(\mathbf{a}(\mathbf{b}),\mathbf{b}) = |A(\mathbf{b})\mathbf{a}(\mathbf{b}) - \mathbf{z}|^2 \tag{4.143}$$

Therefore we have reduced the original problem to a simple problem by reducing the number of unknown parameters from $N_a + N_b$ to N_b.

The following code takes as input the `data` as a list of $(x_i, y_i, \delta y_i)$, a list of functions (or a single function), and a guess for the $\mathbf{b}$ values. If the `fs` argument is not a list but a single function, then there is no $\mathbf{a}$ to compute, and the function proceeds by minimizing the χ^2 using the improved Newton optimizer (the one-dimensional or the improved multidimensional

one, as appropriate). If the argument **b** is missing, then the fitting parameters are all linear, and the algorithm reverts to regular linear least squares. Otherwise, run the more complex algorithm described earlier:

Listing 4.42: in file: `nlib/fitting.nim`

```
type
  NonlinearFit* = proc(b: seq[float], x: float): float

proc fit*(data: seq[(float, float, float)],
          fs: seq[NonlinearFit],
          b0: seq[float],
          ap = 1e-6, rp = 1e-4, ns = 200,
          constraint: proc(b: seq[float]): float = nil
          ): (seq[float], float) =
  ## Generic linear/nonlinear chi^2 fit. Returns (parameters, chi2).
  let na = fs.len
  proc core(b: seq[float]): (seq[float], float) =
    let A = newMatrix(data.len, na,
      proc(r, c: int): float = fs[c](b, data[r][0]) / data[r][2])
    let z = newMatrix(data.len, 1,
      proc(r, c: int): float = data[r][1] / data[r][2])
    let a = (1.0 / (A.T * A)) * (A.T * z)
    let chi2Val = norm(A * a - z) ^ 2
    (a.flatten(), chi2Val)
  proc g(b: seq[float]): float =
    let (_, chi2v) = core(b)
    var s = chi2v
    if constraint != nil: s += constraint(b)
    s
  let b = optimizeNewtonMultiImproved(g, b0, ap, rp, ns)
  let (a, chi2v) = core(b)
  (a & b, chi2v)
```

Here is an example:

```
var data: seq[(float, float, float)] = @[]
for i in 1 ..< 10:
  let fi = float(i)
  data.add (fi, fi + 2.0 * fi^2 + 300.0 / (fi + 10.0), 2.0)
let fs: seq[NonlinearFit] = @[
  proc(b: seq[float], x: float): float = x,
  proc(b: seq[float], x: float): float = x * x,
  proc(b: seq[float], x: float): float = 1.0 / (x + b[0]),
]
let (ab, chi2) = fit(data, fs, @[5.0])
echo ab, " ", chi2
# @[0.999..., 2.000..., 300.000..., 10.000...] ...
```

In the preceding implementation, we added a somewhat mysterious argument `constraint`. This is a function of **b**, and its output gets added to the value of χ^2, which we are minimizing. By choosing the appropriate function, we can set constraints about the expected values b. These constraints represent a priori knowledge about the parameters, that is, knowledge that does not come from the data being fitted.

For example, if we know that b_i must be close to some $\bar{b}_i$ with some uncertainty δb_i, then we can use

```
proc constraint(b, barB, deltaB: seq[float]): float =
  for i in 0 ..< b.len:
    result += ((b[i] - barB[i]) / deltaB[i]) ^ 2
```

and pass the preceding function as a constraint. From a practical effect, this stabilizes our fit. From a theoretical point of view, the $\bar{b}_i$ are the *priors* of Bayesian statistics.

4.10 Integration

Consider the integral of $f(x)$ for x in domain $[a, b]$, which we normally represent as

$$I = \int_a^b f(x)dx \tag{4.144}$$

and which measures the area under the curve $y = f(x)$ delimited on the left by $x = a$ and on the right by $x = b$.

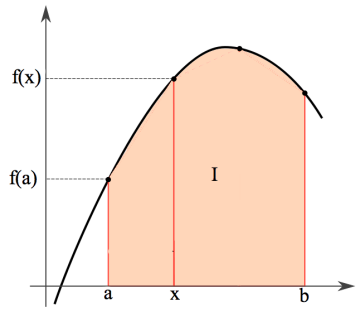

Figure 4.8: Visual representation of the concept of an integral as the area under a curve.

As we did in the previous subsection, we can approximate the possible values taken by x as discrete values $x \equiv hi$, where $h = (b-a)/n$. At those values, the function f evaluates to $f_i \equiv f(hi)$. Thus the integral can be approximated as a sum of trapezoids:

$$I_n \simeq \sum_{i=0}^{i<n} \frac{h}{2}(f_i + f_{i+1}) \tag{4.145}$$

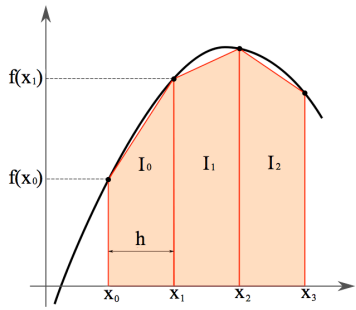

Figure 4.9: Visual representation of the trapezoid method for numerical integration.

If a function is discontinuous only in a finite number of points in the domain $[a, b]$, then the following limit exists:

$$\lim_{n\to\infty} I_n \to I \tag{4.146}$$

We can implement the naive integration as a function of N as follows:

Listing 4.43: in file: `nlib/integration.nim`

```
proc integrateNaive*(f: proc(x: float): float,
                     a, b: float, n = 20): float =
  ## Trapezoidal-rule integration of `f` over `[a, b]` with `n` slices.
  let h = (b - a) / float(n)
  result = h / 2.0 * (f(a) + f(b))
  for i in 1 ..< n:
    result += h * f(a + h * float(i))
```

And here we implement the limit by doubling the number of points until convergence is achieved:

Listing 4.44: in file: `nlib/integration.nim`

```
proc integrate*(f: proc(x: float): float, a, b: float,
                ap = 1e-4, rp = 1e-4, ns = 20): float =
  ## Iteratively-refined trapezoidal integration.
  var I = integrateNaive(f, a, b, 1)
  for k in 1 ..< ns:
    let IOld = I
    I = integrateNaive(f, a, b, 2 ^ k)
    if k > 2 and norm(I - IOld) < max(ap, norm(I) * rp):
      return I
  raise newException(ArithmeticDefect, "no convergence")
```

We can test the convergence as follows:

```
echo integrateNaive(sin, 0.0, 3.0, n = 2)   # 1.6020...
echo integrateNaive(sin, 0.0, 3.0, n = 4)   # 1.8958...
echo integrateNaive(sin, 0.0, 3.0, n = 8)   # 1.9666...
echo integrate(sin, 0.0, 3.0)               # 1.9899...
echo 1.0 - cos(3.0)                         # 1.9899...
```

4.10.1 Quadrature

In the previous integration, we divided the domain $[a, b]$ into subdomains, and we computed the area under the curve f in each subdomain by approximating it with a trapezoid; for example, we approximated the function in between x_i and x_{i+1} with a straight line. We can do better by approximating the function with a polynomial of arbitrary degree n and then compute the area in the subdomain by explicitly integrating the polynomial.

This is the basic idea of quadrature. For a subdomain delimited by $(0, 1)$, we can impose

$$\int_0^1 1 \mathrm{d}x \quad = h \quad = \sum_i c_i (i/n)^0 \tag{4.147}$$

$$\int_0^1 x \mathrm{d}x \quad = h^2/2 \quad = \sum_i c_i (i/n)^1 \tag{4.148}$$

$$\ldots \quad \ldots \quad \ldots \tag{4.149}$$

$$\int_0^1 x^{n-1} \mathrm{d}x \quad = h^n/n \quad = \sum_i c_i (i/n)^2 \tag{4.150}$$

where c_i are coefficients to be determined:

Listing 4.45: in file: nlib/integration.nim

```
type
  QuadratureIntegrator* = ref object
    w*: Matrix

proc newQuadratureIntegrator*(order = 4): QuadratureIntegrator =
  let h = 1.0 / float(order - 1)
  let A = newMatrix(order, order,
    proc(r, c: int): float = (float(c) * h) ^ r)
  let s = newMatrix(order, 1,
    proc(r, c: int): float = 1.0 / float(r + 1))
  result = QuadratureIntegrator(w: (1.0 / A) * s)

proc integrate*(q: QuadratureIntegrator,
                f: proc(x: float): float, a, b: float): float =
  let order = q.w.nrows
  let h = (b - a) / float(order - 1)
  for i in 0 ..< order:
    result += q.w[i, 0] * f(a + float(i) * h)
  result *= (b - a)

proc integrateQuadratureNaive*(f: proc(x: float): float,
                               a, b: float, n = 20, order = 4): float =
  let h = (b - a) / float(n)
  let q = newQuadratureIntegrator(order = order)
  for i in 0 ..< n:
    result += q.integrate(f, a + float(i) * h,
                          a + float(i) * h + h)
```

Here is an example of usage:

```
echo integrateQuadratureNaive(sin, 0.0, 3.0, n = 2, order = 2) # 1.602...
echo integrateQuadratureNaive(sin, 0.0, 3.0, n = 2, order = 3) # 1.993...
echo integrateQuadratureNaive(sin, 0.0, 3.0, n = 2, order = 4) # 1.991...
```

4.11 Fourier transforms

A function with a domain over a finite interval $[a, b]$ can be approximated with a vector. For example, consider a function $f(x)$ with domain $[0, T]$. We can sample the function at points $x_k = a + (b - a)k/N$ and represent the discretized function with a vector

$$\mathbf{u}_f \equiv \{cf(x_0), cf(x_1), cf(x_2), ...cf(x_N)\} \tag{4.151}$$

where c is an arbitrary constant that we choose to be $c = \sqrt{(b-a)/N}$. This choice simplifies our later algebra. Summarizing, we define

$$u_{fk} \equiv \sqrt{\frac{b-a}{N}} f(x_k) \tag{4.152}$$

Given any two functions, we can define their scalar product as the limit of $N \to \infty$ of the scalar product between their corresponding vectors:

$$f \cdot g \equiv \lim_{N\to\infty} \mathbf{u}_f \cdot \mathbf{u}_g = \lim_{N\to\infty} \frac{b-a}{N} \sum_k f(x_k) g(x_k) \tag{4.153}$$

Using the definition of integral, it can be proven that, in the limit $N \to \infty$, this is equivalent to

$$f \cdot g = \int_a^b f(x) g(x) dx \tag{4.154}$$

This is because we have chosen c such that c^2 is the width of a rectangle in the Riemann integration.

From now on, we will omit the f subscript in $\mathbf{u}$ and simply use different letters for vectors representing different sampled functions ($\mathbf{u}$, $\mathbf{v}$, $\mathbf{b}$, etc.).

Because we are interested in numerical algorithms, we will keep N finite and work with the sum instead of the integral.

Given a fixed N, we can always find N vectors $\mathbf{b}_0, \mathbf{b}_1 ... \mathbf{b}_{N_1}$ that are linearly independent, normalized, and orthogonal, that is,

$$\mathbf{b}_i \cdot \mathbf{b}_j = \sum_k b_{ik} b_{ik} = \delta_{ij} \tag{4.155}$$

Here b_{jk} is the k component of vector $\mathbf{b}_j$ and δ_{ij} is the Kronecker delta defined as 0 when $i \neq j$ and 1 when $i == j$.

Any set of vectors $\{\mathbf{b}_j\}$ meeting the preceding condition is called an orthonormal basis. Any other vector $\mathbf{u}$ can be represented by its projections over the basis vectors:

$$u_i = \sum_i v_j b_{ji} \tag{4.156}$$

where v_j is the projection of u along $\mathbf{b}_j$, which can be computed as

$$v_j = \sum_i u_i b_{ji} \tag{4.157}$$

In fact, by direct substitution, we obtain

$$\begin{aligned} v_j &= \sum_k u_k b_{jk} && (4.158)\\ &= \sum_k (\sum_i v_i b_{ik}) b_{jk} && (4.159)\\ &= \sum_i v_i (\sum_k b_{ik} b_{jk}) && (4.160)\\ &= \sum_i v_i \delta_{ij} && (4.161)\\ &= v_j && (4.162) \end{aligned}$$

In other words, once we have a basis of vectors, the vector $\mathbf{u}$ can be represented in terms of the vector $\mathbf{v}$ of v_j coefficients and, conversely, $\mathbf{v}$ can be computed from $\mathbf{u}$; $\mathbf{u}$ and $\mathbf{v}$ contain the same information.

The transformation from $\mathbf{u}$ to $\mathbf{v}$, and vice versa, is a linear transformation. We call T^+ the transformation from $\mathbf{u}$ to $\mathbf{v}$ and T^- its inverse:

$$\mathbf{v} = T^+(\mathbf{u}) \qquad \mathbf{u} = T^-(\mathbf{v}) \tag{4.163}$$

From the definition, and without attempting any optimization, we can implement these operators as follows:

```
proc transform(u: seq[float], b: seq[seq[float]]): seq[float] =
  for bi in b:
    var s = 0.0
    for k in 0 ..< u.len: s += u[k] * bi[k]
    result.add s

proc antitransform(v: seq[float], b: seq[seq[float]]): seq[float] =
  result = newSeq[float](v.len)
  for k in 0 ..< v.len:
    var s = 0.0
    for i, bi in b: s += v[i] * bi[k]
    result[k] = s
```

Here is an example of usage:

```
proc makeBasis(N: int): seq[seq[float]] =
  for j in 0 ..< N:
    var row = newSeq[float](N)
    row[j] = 1.0
    result.add row
let b = makeBasis(4)
echo b
# @[@[1.0, 0.0, 0.0, 0.0], @[0.0, 1.0, 0.0, 0.0], ...]
let u = @[1.0, 2.0, 3.0, 4.0]
let v = transform(u, b)
echo antitransform(v, b)
# @[1.0, 2.0, 3.0, 4.0]
```

Of course, this example is trivial because of the choice of basis which makes v the same as u. Yet our argument works for any basis $\mathbf{b}_i$. In particular, we can make the following choice:

$$b_{ji} = \frac{1}{\sqrt{2\pi}} e^{2\pi I i j / N} \tag{4.164}$$

where I is the imaginary unit. With this choice, the T^+ and T^- functions become

$$v_j \underset{FT^+}{=} N^{-\frac{1}{2}} \sum_i u_i e^{2\pi I i j / N} \tag{4.165}$$

$$u_i \underset{FT^-}{=} N^{-\frac{1}{2}} \sum_j v_j e^{-2\pi I i j / N} \tag{4.166}$$

and they take the names of Fourier transform and anti-transform [32], respectively; we can implement them as follows:

```
import std/complex

proc fourier(u: seq[Complex64], sign = 1): seq[Complex64] =
  let N = u.len
  let coeff = 1.0 / sqrt(float(N))
  let omega = complex(0.0, 2.0 * PI * float(sign) / float(N))
  result = newSeq[Complex64](N)
  for j in 0 ..< N:
    var s = complex(0.0, 0.0)
    for i in 0 ..< N:
      s = s + complex(coeff, 0.0) * u[i] *
              exp(omega * complex(float(i * j), 0.0))
    result[j] = s

proc antiFourier(v: seq[Complex64]): seq[Complex64] = fourier(v, sign = -1)
```

Here `complex(0.0, 1.0)` represents the imaginary unit I in Nim's `std/complex` module, and `exp` on a `Complex64` is the complex exponential.

Notice how the transformation works even when **u** is a vector of complex numbers.

Something special happens when u is real:

$$Re(v_j) = +Re(v_{N-j-1}) \tag{4.167}$$

$$Im(v_j) = -Im(v_{N-j-1}) \tag{4.168}$$

We can speed up the code even more using recursion and by observing that if N is a power of 2, then

$$v_j = N^{-\frac{1}{2}} \sum_i u_{2i} e^{2\pi I(2i)j/N} + \tag{4.169}$$

$$N^{-\frac{1}{2}} \sum_i u_{2i+1} e^{2\pi I(2i+1)j/N} \tag{4.170}$$

$$= 2^{-\frac{1}{2}} (v_j^{even} + e^{2\pi j/N} v_j^{even}) \tag{4.171}$$

where v_j^{even} is the Fourier transform of the even terms and v_j^{odd} is the Fourier transform of the odd terms.

The preceding recursive expression can be implemented using dynamic programming, thus obtaining

```
import std/complex

proc fastFourier(u: seq[Complex64], sign = 1): seq[Complex64] =
  let N = u.len
  let sqrtN = sqrt(float(N))
  result = newSeq[Complex64](N)
  for i in 0 ..< N:
    result[i] = u[i] / complex(sqrtN, 0.0)
  var k = N div 2
  while k > 0:
    let omega = exp(complex(0.0, 2.0 * PI * float(sign * k) / float(N)))
    for i in 0 ..< N:
      let j = i xor k
      if i < k:
        let ik = i div k
        let jk = j div k
        let vi = result[i]
        let vj = result[j]
        result[i] = vi + pow(omega, complex(float(ik), 0.0)) * vj
        result[j] = vi + pow(omega, complex(float(jk), 0.0)) * vj
    k = k div 2

proc fastAntiFourier(v: seq[Complex64]): seq[Complex64] =
  fastFourier(v, sign = -1)
```

This implementation of the Fourier transform is equivalent to the previous one in the sense that it produces the same result (up to numerical issues), but it is faster as it runs in $\Theta(N \log_2 N)$ versus $\Theta(N^2)$ of the naive implementation. Here `i ^ j` is a binary operator, specifically a XOR. For each binary digit of i, it returns a flipped bit if the corresponding bit in j is 1. For example:

```
i  : 10010010101110
j  : 00010001000010
i^j: 10000011001110
```

4.12 Differential equations

In this section, we deal specifically with differential equations of the following form:

$$a_0 f(x) + a_1 f'(x) + a_2 f''(x) + ... = g(x) \tag{4.172}$$

where $f(x)$ is an unknown function to be determined; f', f'', and so on, are its derivatives; a_i are known input coefficients; and $g(x)$ is a known input function:

$$f''(x) - 4f'(x) + f(x) = sin(x) \tag{4.173}$$

In this case, $a_2(x) = 1$, $a_1(x) = -4$, $a_0(x) = 1$, and $g(x) = \sin(x)$.

This can be solved using Fourier transforms by observing that if the Fourier transform of $f(x)$ is $\tilde{f}(y)$, then the Fourier transform of $f'(x)$ is $iy\tilde{f}(y)$.

Hence, if we Fourier transform both the left and right side of

$$\sum_k a_k f^{(k)}(x) = g(x) \tag{4.174}$$

we obtain

$$(\sum_k a_k (iy)^k)\tilde{f}(y) = \tilde{g}(y) \tag{4.175}$$

therefore $f(x)$ is the anti-Fourier transform of

$$\tilde{f}(y) = \frac{\tilde{g}(y)}{(\sum_k a_k (iy)^k)} \tag{4.176}$$

In one equation, the solution of eq. 4.172 is

$$f(x) = T^{-}(T^{+}(g)/(\sum_{k} a_k (iy)^k)) \tag{4.177}$$

This is fine and useful when the Fourier transformations are easy to compute.

A more practical numerical solution is the following. We define

$$y_i(x) \equiv f^{(i)}(x) \tag{4.178}$$

and we rewrite the differential equation as

$$\begin{aligned}
y''_0 &= y_1 && (4.179)\\
y''_1 &= y_2 && (4.180)\\
y''_2 &= y_3 && (4.181)\\
\ldots &\quad \ldots && (4.182)\\
y''_{N-1} &= y_N = (g(x) - \sum_{k<N} a_k y_k(x))/a_N(x) && (4.183)
\end{aligned}$$

or equivalently

$$\mathbf{y}'' = F(\mathbf{y}) \tag{4.184}$$

where

$$F(\mathbf{y}) = \mathbf{y} + \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \ldots \\ (g(x) - \sum_{k<N} a_k(x) y_k(x))/a_N(x) \end{pmatrix} \tag{4.185}$$

The naive solution is due to Euler:

$$\mathbf{y}(x+h) = \mathbf{y}(x) + hF(\mathbf{y}, x) \tag{4.186}$$

The solution is found by iterating the latest equation. Here h is an arbitrary discretization step. Euler's method works even if the a_k coefficients depend on x.

Although the Euler integrator works in theory, its systematic error adds up and does not disappear in the limit $h \to 0$. More accurate integrators are the Runge–Kutta and the Adams–Bashforth. In the fourth-order Runge–Kutta, the *classical Runge–Kutta method*, we also solve the differential equation by iteration, except that eq. 4.186 is replaced with

$$\mathbf{y}(x+h) = \mathbf{y}(x) + h/6(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{4.187}$$

where

$$\mathbf{k}_1 = F(\mathbf{y}, x) \tag{4.188}$$
$$\mathbf{k}_2 = F(\mathbf{y} + hk_1/2, x + h/2) \tag{4.189}$$
$$\mathbf{k}_3 = F(\mathbf{y} + hk_2/2, x + h/2) \tag{4.190}$$
$$\mathbf{k}_4 = F(\mathbf{y} + hk_3, x + h) \tag{4.191}$$

5
Probability and Statistics

5.1 Probability

Probability derives from the Latin *probare* (to prove or to test). The word probably means roughly "likely to occur" in the case of possible future occurrences or "likely to be true" in the case of inferences from evidence. See also probability theory.

What mathematicians call probability is the mathematical theory we use to describe and quantify uncertainty. In a larger context, the word *probability* is used with other concerns in mind. Uncertainty can be due to our ignorance, deliberate mixing or shuffling, or due to the essential randomness of Nature. In any case, we measure the uncertainty of events on a scale from zero (impossible events) to one (certain events or no uncertainty).

There are three standard ways to define probability:

- (frequentist) Given an experiment and a set of possible outcomes S, the probability of an event $A \subset S$ is computed by repeating the experiment N times, counting how many times the event A is realized, N_A, then taking the limit
$$\text{Prob}(A) \equiv \lim_{N \to \infty} \frac{N_A}{N} \tag{5.1}$$
This definition actually requires that one performs an experiment, if

not an infinite, then a number of times.

- (a priori) Given an experiment and a set of possible outcomes S with cardinality $c(S)$, the probability of an event $A \subset S$ is defined as

$$\text{Prob}(A) \equiv \frac{c(A)}{c(S)} \tag{5.2}$$

  This definition is ambiguous because it assumes that each "atomic" event $x \in S$ has the same a priori probability and therefore the definition itself is circular. Nevertheless we use this definition in many practical circumstances. What is the probability that when rolling a dice we will get an even number? The space of possible outcomes is $S = \{1,2,3,4,5,6\}$ and $A = \{2,4,6\}$ therefore $\text{Prob}(A) = c(A)/c(S) = 3/6 = 1/2$. This analysis works for an ideal die and ignores the fact that a real dice may be biased. The former definition takes into account this possibility, whereas the latter does not.

- (axiomatic definition) Given an experiment and a set of possible outcomes S, the probability of an event $A \subset S$ is a number $\text{Prob}(A) \in [0,1]$ that satisfies the following conditions: $\text{Prob}(S) = 1$; $\text{Prob}(A_1 \cup A_2) = \text{Prob}(A_1) + \text{Prob}(A_2)$ if $A_1 \cap A_2 = 0$.

In some sense, probability theory is a physical theory because it applies to the physical world (this is a nontrivial fact). While the axiomatic definition provides the mathematical foundation, the a priori definition provides a method to make predictions based on combinatorics. Finally the *frequentist* definition provides an experimental technique to confront our predictions with experiment (is our dice a perfect dice, or is it biased?).

We will differentiate between an "atomic" event defined as an event that can be realized by a single possible outcome of our experiment and a general event defined as a subset of the space of all possible outcomes. In the case of a dice, each possible number (from 1 to 6) is an event and is also an atomic event. The event of getting an even number is an event but not an atomic event because it can be realized in three possible ways.

The axiomatic definition makes it easy to prove theorems, for example,

If $S = A \cup A^c$ *and* $A \cap A^c = 0$ *then* $Prob(A) = 1 - Prob(A^c)$

Nim's standard library includes the `std/random` module, which can generate random numbers; we can use it to perform some experiments. Let's simulate a dice with six possible outcomes. We can use the frequentist definition:

```
import std/random
randomize()
let S = @[1, 2, 3, 4, 5, 6]

proc Prob[T](A, S: seq[T], N = 1000): float =
  var hits = 0
  for _ in 0 ..< N:
    if S[rand(S.len - 1)] in A: inc hits
  hits / N

echo Prob(@[6], S)        # 0.166
echo Prob(@[1, 2], S)     # 0.308
```

Here `Prob(A)` computes the probability that the event is set `A` using `N=1000` simulated experiments. The `random.choice` function picks one of the choices at random with equal probability.

We can compute the same quantity using the a priori definition:

```
proc Prob[T](A, S: seq[T]): float = float(A.len) / float(S.len)
echo Prob(@[6], S)        # 0.16666666666666666
echo Prob(@[1, 2], S)     # 0.3333333333333333
```

As stated before, the latter is more precise because it produces results for an "ideal" dice while the frequentist's approach produces results for a real dice (in our case, a simulated dice).

5.1.1 Conditional probability and independence

We define $\text{Prob}(A|B)$ as the probability of event A given event B, and we write

$$\text{Prob}(A|B) \equiv \frac{\text{Prob}(AB)}{\text{Prob}(B)} \tag{5.3}$$

where $\text{Prob}(AB)$ is the probability that A and B both occur and $\text{Prob}(B)$ is the probability that B occurs. Note that if $\text{Prob}(A|B) = \text{Prob}(A)$, then we say that A and B are independent. From eq.(5.3) we conclude $\text{Prob}(AB) = \text{Prob}(A)\text{Prob}(B)$; therefore the probability that two independent events occur is the product of the probability that each individual event occurs.

We can experiment with conditional probability programmatically. Let's consider two dice, X and Y. The space of all possible outcomes is given by $S^2 = S \times S$. And we are interested in the probability of the second die giving a 6 given that the first dice is also a 6:

```
proc cross[T](u, v: seq[T]): seq[(T, T)] =
  for i in u:
    for j in v:
      result.add (i, j)

proc probConditional[T](A, B, S: seq[T]): float =
  Prob(cross(A, B), cross(S, S)) / Prob(B, S)

echo probConditional(@[6], @[6], S)   # 0.16666666666666666
```

Because we are only considering cases in which the second die is 6, we will pretend that when the second die is 1 through 5 didn't occur. Not surprisingly, we find that `Prob_conditional([6], [6], S)` produces the same result as `Prob([6], S)` because the two dice are independent.

In fact, we say that two sets of events A and B are independent if and only if $\mathrm{P}(A|B) = \mathrm{P}(A)$.

5.1.2 Discrete random variables

If S is in the space of all possible outcomes of an experiment and we associate an integer number X to each element of S, we say that X is a *discrete random variable*. If X is a discrete variable, we define $p(x)$, the *probability mass function* or *distribution*, as the probability that $X = x$:

$$p(x) \equiv \mathrm{Prob}(X = x) \tag{5.4}$$

We also define the *expectation value* of any function of a discrete random variable $f(X)$ as

$$E[f(X)] \equiv \sum_i f(x_i)p(x_i) \tag{5.5}$$

where i loops all possible variables x_i of the random variable X.

For example, if X is the random variable associated with the outcome of

rolling a dice, $p(x) = 1/6$ if $x = 1, 2, 3, 4, 5$ or 6 and $p(x) = 0$ otherwise:

$$E[X] = \sum_i x_i p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} x_i \frac{1}{6} = 3.5 \tag{5.6}$$

and

$$E[(X - 3.5)^2] = \sum_i (x_i - 3.5)^2 p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} (x_i - 3.5)^2 \frac{1}{6} = 2.9167 \tag{5.7}$$

We call $E[X]$ the *mean* of X and usually denote it with μ_X. We call $E[(X - \mu_X)^2]$ the *variance* of X and denote it with σ_X^2. Note that

$$\sigma_X = E[X^2] - E[X]^2 \tag{5.8}$$

For discrete random variables, we can implement these definitions as follows:

Listing 5.1: in file: `nlib/stats.nim`

```
proc E*(f: proc(x: float): float, S: seq[float]): float =
  if S.len == 0: return 0.0
  for x in S: result += f(x)
  result /= float(S.len)

proc mean*(X: seq[float]): float =
  E(proc(x: float): float = x, X)

proc variance*(X: seq[float]): float =
  let mu = mean(X)
  E(proc(x: float): float = (x - mu) ^ 2, X)

proc sd*(X: seq[float]): float = sqrt(variance(X))
```

As an example, let's consider a simple bet on a dice. We roll the dice once and win $20 if the dice returns 6; we lose $5 otherwise:

```
let S = @[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
echo mean(S)            # 3.5
echo sd(S)              # 1.707...
proc payoff(x: float): float = (if x == 6.0: 20.0 else: -5.0)
echo E(payoff, S)       # -0.83333...
```

The average expected payoff is $-0.83...$, which means that on average, we should expect to lose 83 cents at this game.

5.1.3 Continuous random variables

If S is the space of all possible outcomes of an experiment and we associate a real number X with each element of S, we say that X is a *continuous random variable*. We also define a *cumulative distribution function* $F(x)$ as the probability that $X \leq x$:

$$F(x) \equiv \text{Prob}(X \leq x) \tag{5.9}$$

If S is a continuous set and X is a continuous random variable, then we define a *probability density* or *distribution* $p(x)$ as

$$p(x) \equiv \frac{dF(x)}{dx} \tag{5.10}$$

and the probability that X falls into an interval $[a, b]$ can be computed as

$$\text{Prob}(a \leq X \leq b) = \int_a^b p(x)dx \tag{5.11}$$

We also define the *expectation value* of any function of a random variable $f(X)$ as

$$E[f(X)] = \int_{-\infty}^{\infty} f(x)p(x)dx \tag{5.12}$$

For example, if X is a uniform random variable (probability density $p(x)$ equal to 1 if $x \in [0, 1]$, equal to 0 otherwise)

$$E[X] = \int_{-\infty}^{\infty} xp(x)dx = \int_0^1 xdx = \frac{1}{2} \tag{5.13}$$

and

$$E[(X - \frac{1}{2})^2] = \int_{-\infty}^{\infty} (x - \frac{1}{2})^2 p(x)dx = \int_0^1 (x^2 - x + \frac{1}{4})dx = \frac{1}{12} \tag{5.14}$$

We call $E[X]$ the **mean** of X and usually denote it with μ_X. We call $E[(X - \mu_X)^2]$ the **variance** of X and denote it with σ_X^2. Note that

$$\sigma_X^2 = E[X^2] - E[X]^2 \tag{5.15}$$

By definition,

$$F(\infty) \equiv \text{Prob}(X \leq \infty) = 1 \tag{5.16}$$

therefore

$$\text{Prob}(-\infty \leq X \leq \infty) = \int_{-\infty}^{\infty} p(x)dx = 1 \tag{5.17}$$

The distribution p is always normalized to 1.

Moreover,

$$\begin{aligned} E[aX+b] &= \int_{-\infty}^{\infty} (ax+b)p(x)dx & (5.18)\\ &= a\int_{-\infty}^{\infty} xp(x)dx + b\int_{-\infty}^{\infty} p(x)dx & (5.19)\\ &= aE[X] + b & (5.20) \end{aligned}$$

therefore $E[X]$ is a linear operator.

One important consequence of all these formulas is that if we have a function f and a domain $[a,b]$, we can compute its integral by choosing p to be a uniform distribution with values exclusively between a and b:

$$E[f] = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx \tag{5.21}$$

We can also compute the same integral by using the definition of expectation value for a discrete distribution:

$$E[f] = \sum_{x_i} f(x_i)p(x_i) = \frac{1}{N}\sum_{x_i} f(x_i) \tag{5.22}$$

where x_i are N random points drawn from the uniform distribution p defined earlier. In fact, in the large N limit,

$$\lim_{N\to\infty} \frac{1}{N}\sum_{x_i} f(x_i) = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx \tag{5.23}$$

We can verify the preceding relation numerically for a special case:

Listing 5.2: in file: `nlib/montecarlo.nim`

```
proc integrateMc*(f: proc(x: float): float, a, b: float,
                  N = 1000): float =
  for _ in 0 ..< N: result += f(rand(b - a) + a)
  result = result / float(N) * (b - a)

echo integrateMc(sin, 0.0, PI, N = 10000)  # 2.00...
```

This is the simplest case of Monte Carlo integration, which is the subject of a following chapter.

5.1.4 Covariance and correlations

Given two random variables, X and Y, we define the covariance (cov) and the correlation (corr) between them as

$$\begin{aligned} \text{cov}(X,Y) &\equiv E[(X-\mu_X)(Y-\mu_Y)] = E[XY] - E[X]E[Y] \quad (5.24) \\ \text{corr}(X,Y) &\equiv \text{cov}(X,Y)/(\sigma_X\sigma_Y) \quad (5.25) \end{aligned}$$

Applying the definitions:

$$\begin{aligned} E[XY] &= \int\int xyp(x,y)dxdy \quad (5.26) \\ &= \int\int xyp(x)p(y)dxdy \quad (5.27) \\ &= \left[\int xp(x)dx\right]\left[\int yp(y)dy\right] \quad (5.28) \\ &= E[X]E[Y] \quad (5.29) \end{aligned}$$

therefore

$$\text{cov}(X,Y) = E[XY] - E[X]E[Y] = 0 \quad (5.30)$$

Therefore

$$\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y + 2\text{cov}(X,Y) \quad (5.31)$$

and if X and Y are independent, then $\text{cov}(X,Y) = \text{corr}(X,Y) = 0$.

Notice that the reverse is not true. Even if the correlation and the covariance are zero, X and Y may be dependent.

Moreover,

$$\begin{aligned} \text{cov}(X,Y) &= E[(X-\mu_X)(Y-\mu_Y)] && (5.32)\\ &= E[(X-\mu_X)(\pm X \mp \mu_X)] && (5.33)\\ &= \pm E[(X-\mu_X)(X-\mu_X)] && (5.34)\\ &= \pm\sigma_X^2 && (5.35) \end{aligned}$$

Therefore, if X and Y are completely correlated or anti-correlated ($Y = \pm X$), then $\text{cov}(X,Y) = \pm\sigma_X^2$ and $\text{corr}(X,Y) = \pm 1$. Notice that the correlation lies always in the range $[-1,1]$.

Finally, notice that for uncorrelated random variables X_i,

$$\begin{aligned} E[\sum_i a_i X_i] &= \sum_i a_i E[X_i] && (5.36)\\ E[(\sum_i X_i)^2] &= \sum_i E[X_i^2] && (5.37) \end{aligned}$$

We can define covariance and correlation for discrete distributions:

Listing 5.3: in file: `nlib/stats.nim`

```
proc covariance*(X, Y: seq[float]): float =
  for i in 0 ..< X.len: result += X[i] * Y[i]
  result = result / float(X.len) - mean(X) * mean(Y)

proc correlation*(X, Y: seq[float]): float =
  covariance(X, Y) / sd(X) / sd(Y)
```

5.1.5 Strong law of large numbers

If $X_1, X_2, ...X_n$ are a sequence of independent and identically distributed random variables with $E[X_i] = \mu$ and finite variance, then

$$\lim_{n\to\infty} \frac{X_1 + X_2 + ... + X_n}{n} = \mu \tag{5.38}$$

This theorem means that "the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed." The name of this law is due to Poisson [33].

5.1.6 Central limit theorem

This is one of the most important theorems concerning distributions [34]: if $X_1, X_2, ...X_n$ are a sequence of random variables with finite means, μ_i, and finite variance, σ_i^2, then

$$Y = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} X_i \tag{5.39}$$

follows a Gaussian distribution with mean and variance:

$$\mu = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \mu_i \tag{5.40}$$

$$\sigma^2 = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \sigma_i^2 \tag{5.41}$$

We can numerically verify this for the simple case in which X_i are uniform random variables with mean equal to 0:

```
proc addedUniform(n: int): float =
  for _ in 0 ..< n: result += rand(2.0) - 1.0
  result /= float(n)

proc makeSet(n: int, m = 10000): seq[float] =
  for _ in 0 ..< m: result.add addedUniform(n)

for (n, name) in [(1, "central1.png"), (2, "central3.png"),
                  (4, "central4.png"), (8, "central8.png")]:
  saveHistogram("images/" & name, makeSet(n),
                title = "Central Limit Theorem (N=" & $n & ")",
                xlab = "y", ylab = "p(y)")
```

This theorem is of fundamental importance for stochastic calculus. Notice that the theorem does not apply when the X_i follow distributions that do not have a finite mean or a finite variance.

Distributions that do not follow the central limit are called *Levy distributions*. They are characterized by fat tails for the form

$$p(x) \underset{x\to\infty}{\sim} \frac{1}{|x|^{1+\alpha}}, \qquad 0 < \alpha < 2 \tag{5.42}$$

An example is the Pareto distribution.

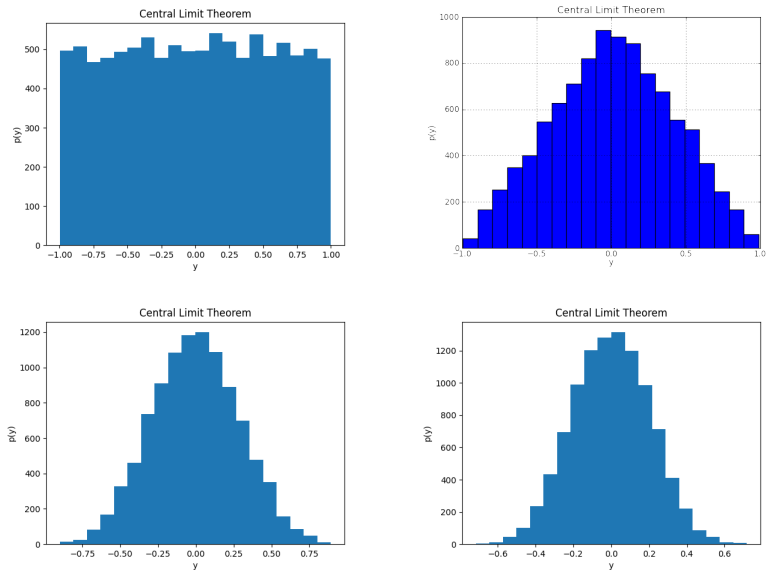

Figure 5.1: Example of distributions for sums of 1, 2, 4, and 8 uniform random variables. The more random variables are added, the better the result approximates a Gaussian distribution.

5.1.7 Error in the mean

One consequence of the Central Limit Theorem is a useful formula for evaluating the error in the mean. Let's consider the case of N repeated experiments with outcomes X_i. Let's also assume that each X_i is supposed to be equal to an unknown value μ, but in practice, $X_i = \mu + \varepsilon$, where ε is a random variable with Gaussian distribution centered at zero. One could estimate μ by $\mu = E[X] = \frac{1}{N}\Sigma_i X_i$. In this case, statistical error in the mean is given by

$$\delta\mu = \sqrt{\frac{\sigma^2}{N}} \tag{5.43}$$

where $\sigma^2 = E[(X-\mu)^2] = \frac{1}{N}\Sigma_i (X_i - \mu)^2$.

5.2 Combinatorics and discrete random variables

Often, to compute the probability of discrete random variables, one has to confront the problem of calculating the number of possible finite outcomes of an experiment. Often this problem is solved by combinatorics.

5.2.1 Different plugs in different sockets

If we have n different plugs and m different sockets; in how many ways can we place the plugs in the sockets?

- Case 1, $n \geq m$. All sockets will be filled. We consider the first socket, and we can select any of the n plugs (n combinations). We consider the second socket, and we can select any of the remaining $n-1$ plugs ($n-1$ combinations), and so on, until we are left with no free sockets and $n-m$ unused plugs; therefore there are

$$n!/(n-m)! = n(n-1)(n-2)...(n-m+1) \tag{5.44}$$

  combinations.

- Case 2, $n \leq m$. All plugs have to be used. We consider the first plug, and we can select any of the m sockets (m combinations). We consider the second plug, and we can select any of the remaining $m-1$ sockets ($m-1$ combinations), and so on, until we are left with no spare plugs and $m-n$ free sockets; therefore there are

$$m!/(m-n)! = m(m-1)(m-2)...(m-n+1) \tag{5.45}$$

  combinations. Note that if $m = n$ then case 1 and case 2 agree, as expected.

5.2.2 Equivalent plugs in different sockets

If we have n equivalent plugs and m different sockets, in how many ways can we place the plugs in the sockets?

- Case 1, $n \geq m$. All sockets will be filled. We cannot distinguish one combination from the other because all plugs are the same. There is only one combination.

- Case 2, $n \leq m$. All plugs have to be used but not all sockets. There are $m!/(m-n)!$ ways to fill the sockets with different plugs, and there are $n!$ ways to arrange the plugs within the same filled sockets. Therefore there are
$$\binom{m}{n} = \frac{m!}{(m-n)!n!} \tag{5.46}$$
ways to place n equivalent plugs into m different sockets. Note that if $m = n$
$$\binom{n}{n} = \frac{n!}{(n-n)!n!} = 1 \tag{5.47}$$
in agreement with case 1.

Here is another example. A club has 20 members and has to elect a president, a vice president, a secretary, and a treasurer. In how many different ways can they select the four officeholders? Think of each office as a socket and each person as a plug; therefore the number combination is $20!/(20-4)! \simeq 1.2 \times 10^5$.

5.2.3 Colored cards

We have 52 cards, 26 black and 26 red. We shuffle the cards and pick three.

- What is the probability that they are all red?
$$\text{Prob}(3red) = \frac{26}{52} \times \frac{25}{51} \times \frac{24}{50} = \frac{2}{17} \tag{5.48}$$
- What is the probability that they are all black?
$$\text{Prob}(3black) = \text{Prob}(3red) = \frac{2}{17} \tag{5.49}$$
- What is the probability that they are not all black or all red?
$$\begin{aligned} \text{Prob}(mixture) &= 1 - \text{Prob}(3red \cup 3black) && (5.50)\\ &= 1 - \text{Prob}(3red) - \text{Prob}(3black) && (5.51)\\ &= 1 - 2\frac{2}{17} && (5.52)\\ &= \frac{13}{17} && (5.53) \end{aligned}$$

Here is an example of how we can simulate the deck of cards to compute an answer to the last question:

```
proc makeDeck(): seq[string] =
  for i in 0 ..< 26:
    result.add "red"
    result.add "black"

proc makeShuffledDeck(): seq[string] =
  result = makeDeck()
  shuffle(result)

proc pickThreeCards(): seq[string] = makeShuffledDeck()[0 ..< 3]

proc simulateCards(n = 1000): float =
  var counter = 0
  for _ in 0 ..< n:
    let c = pickThreeCards()
    if not (c[0] == c[1] and c[1] == c[2]):
      counter += 1
  result = counter / n

echo simulateCards()
```

5.2.4 Gambler's fallacy

The typical error in computing probabilities is mixing a priori probability with information about past events. This error is called the *gambler's fallacy* [35]. For example, we consider the preceding problem. We see the first two cards, and they are both red. What is the probability that the third one is also red?

- **Wrong answer**: The probability that they are all red is $\text{Prob}(3red) = 2/17$; therefore the probability that the third one is also $2/17$.
- **Correct answer**: Because we know that the first two cards are red, the third card must belong to a set of (26 black cards + 24 red cards); therefore the probability that it is red is

$$\text{Prob}(red) = \frac{24}{24+26} = \frac{12}{25} \tag{5.54}$$

6

Random Numbers and Distributions

In the previous chapters, we have seen how using the `std/random` module, we can generate uniform random numbers. This module can also generate random numbers following other distributions. The point of this chapter is to understand how random numbers are generated.

6.1 Randomness, determinism, chaos and order

Before we proceed further, there are four important concepts that should be defined because of their implications:

- **Randomness** is the characteristic of a process whose outcome is unpredictable (e.g., at the moment I am writing this sentence, I cannot predict the exact time and date when you will be reading it).
- **Determinism** is the characteristic of a process whose outcome can be predicted from the initial conditions of the system (e.g., if I throw a ball from a known position, at a known velocity and in a known direction, I can predict—calculate—its entire future trajectory).
- **Chaos** is the emergence of randomness from order [36] (e.g., if I am on the top of a hill and I throw the ball in a vertical direction, I cannot predict on which side of the hill it is going to end up). Even if the equations that describe a phenomenon are known and are determinis-

tic, it may happen that a small variation in the initial conditions causes a large difference in the possible deterministic evolution of the system. Therefore the outcome of a process may depend on a tiny variation of the initial parameters. These variations may not be measurable in practice, thus making the process unpredictable and chaotic. Chaos is generally regarded as a characteristic of some differential equations.

- **Order** is the opposite of chaos. It is the emergence of regular and reproducible patterns from a process that, in itself, may be random or chaotic (e.g., if I keep throwing my ball in a vertical direction from the top of a hill and I record the final location of the ball, I eventually find a regular pattern, a probability distribution associated with my experiment, which depends on the direction of the wind, the shape of the hill, my bias in throwing the ball, etc.).

These four concepts are closely related, and they do not necessarily come in opposite pairs as one would expect.

A deterministic process may cause chaos. We can use chaos to generate randomness (we will see examples when covering random number generation). We can study randomness and extract its ordered properties (probability distributions), and we can use randomness to solve deterministic problems (Monte Carlo) such as computing integrals and simulating a system.

6.2 Real randomness

Note that randomness does not necessarily come from chaos. Randomness exists in nature [37][38]. For example, a radioactive atom "decays" into a different atom at some random point in time. For example, an atom of carbon 14 decays into nitrogen 14 by emitting an electron and a neutrino

$$ {}^{14}_{6}C \longrightarrow {}^{14}_{7}N + e^{-} + \overline{\nu}_e \tag{6.1} $$

at some random time t; t is unpredictable. It can be proven that the randomness in the nuclear decay time is not due to any underlying deterministic process. In fact, constituents of matter are described by quantum

physics, and randomness is a fundamental characteristic of quantum systems. Randomness is not a consequence of our ignorance.

This is not usually the case for macroscopic systems. Typically the randomness we observe in some macroscopic systems is not always a consequence of microscopic randomness. Rather, order and determinism emerge from the microscopic randomness, while chaos originates from the complexity of the system.

Because randomness exists in nature, we can use it to produce random numbers with any desired distribution. In particular, we want to use the randomness in the decay time of radioactive atoms to produce random numbers with uniform distribution. We assemble a system consisting of many atoms, and we record the time when we observe atoms decay:

$$t_0, t_1, t_2, t_3, t_4, t_5, ... \tag{6.2}$$

One could study the probability distribution of the t_i and find that it follows an exponential probability distribution like

$$\text{Prob}(t_i = t) = \lambda e^{-\lambda t} \tag{6.3}$$

where $t_0 = 1/\lambda$ is the decay time characteristic of the particular type of atom. One characteristic of this distribution is that it is a memoryless process: t_i does not depend on t_{i-1} and therefore the probability that $t_i > t_{i-1}$ is the same as the probability that $t_i < t_{i-1}$.

6.2.1 Memoryless to Bernoulli distribution

Given the sequence $\{t_i\}$ with exponential distribution, we can build a random sequence of zeros and ones (Bernoulli distribution) by applying the following formula, known as the Von Neumann procedure [39]:

$$x_i = \begin{cases} 1 & \text{if } t_i > t_{i-1} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

Note that the procedure can be applied to map any random sequence into a Bernoulli sequence even if the numbers in the original sequence do not follow an exponential distribution, as long as t_i is independent of t_j for any $j < i$ (memoryless distribution).

6.2.2 Bernoulli to uniform distribution

To map a Bernoulli distribution into a uniform distribution, we need to determine the precision (resolution in number of bits) of the numbers we wish to generate. In this example, we will assume 8 bits.

We can think of each number as a point in a [0, 1) segment. We generate the uniform number by making a number of choices: we break the segment in two and, according to the value of the binary digit (0 or 1), we select the first part or the second part and repeat the process on the subsegment. Because at each stage we break the segment into two parts of equal length and we select one or the other with the same probability, the final distribution of the selected point is uniform. As an example, we consider the Bernoulli sequence

$$01011110110101010111011010 \tag{6.5}$$

and we perform the following steps:

- break the sequence into chunks of 8 bits

$$01011110\text{-}11010101\text{-}01110110\text{-}..... \tag{6.6}$$

- map each chunk $a_0a_1a_2a_3a_4a_5a_6a_7$ into $x = \sum_{k=0}^{k<8} a_k/2^{k+1}$ thus obtaining:

$$0.3671875 - 0.83203125 - 0.4609375 - ... \tag{6.7}$$

A uniform random number generator is usually the first step toward building any other random number generator.

Other physical processes can be used to generate real random numbers using a similar process. Some microprocessors can generate random num-

bers from random temperature fluctuations. An unpredictable source of randomness is called an *entropy source*.

6.3 Entropy generators

The Linux/Unix operating system provides its own entropy source accessible via `/dev/urandom`. From Nim, we read it as a regular file using `std/streams`.

Here we define a class that can access this entropy source and use it to generate uniform random numbers. It follows the same process outlined for the radioactive days:

```
import std/[os, streams]

type URANDOM* = ref object

proc newURANDOM*(data = ""): URANDOM =
  result = URANDOM()
  if data.len > 0:
    let s = newFileStream("/dev/urandom", fmWrite)
    if not s.isNil:
      s.write(data); s.close()

proc random*(u: URANDOM): float =
  const n = 16
  var bytes = newString(n)
  let s = newFileStream("/dev/urandom", fmRead)
  doAssert s.readData(addr bytes[0], n) == n
  s.close()
  var randomInteger = 0'u64
  for k in 0 ..< n:
    randomInteger += uint64(byte(bytes[k])) * (256'u64 ^ k)
  result = float(randomInteger) / (256.0 ^ n)
```

Notice how the constructor allows us to further randomize the data by contributing input to the entropy source. Also notice how the `random()` method reads 16 bytes from the stream (using `os.urandom()`), converts each into 8-bit integers, combines them into a 128-bit integer, and then converts it to a float by dividing by 256^{16}.

6.4 Pseudo-randomness

In many cases we do not have a physical device to generate random numbers, and we require a software solution. Software is deterministic, the outcome is reproducible, therefore it cannot be used to generate randomness, but it can generate pseudo-randomness. The outputs of pseudo random number generators are not random, yet they may be considered random for practical purposes. John von Neumann observed in 1951 that "anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." (For attempts to generate "truly random" numbers, see the article on hardware random number generators.) Nevertheless, pseudo random numbers are a critical part of modern computing, from cryptography to the Monte Carlo method for simulating physical systems.

Pseudo random numbers are relatively easy to generate with software, and they provide a practical alternative to random numbers. For some applications, this is adequate.

6.4.1 Linear congruential generator

Here is probably the simplest possible pseudo random number generator:

$$x_i = (ax_{i-1} + c) \mathrm{mod} m \tag{6.8}$$

$$y_i = x_i / m \tag{6.9}$$

With the choice $a = 65539$, $c = 0$, and $m = 2^{31}$, this generator is called RANDU. It is of historical importance because it is implemented in the C `rand()` function. The RANDU generator is particularly fast because the modulus can be implemented using the finite 32-bit precision.

Here is a possible implementation for $c = 0$:

Listing 6.1: in file: `nlib/randomgen.nim`

```
type
  MCG* = ref object
    x*, a*, m*: int
```

```
4
5  proc newMCG*(seed: int, a = 66539, m = 1 shl 31): MCG =
6    MCG(x: seed, a: a, m: m)
7
8  proc next*(g: MCG): int =
9    g.x = (g.a * g.x) mod g.m
10   g.x
11
12 proc random*(g: MCG): float =
13   float(g.next()) / float(g.m)
```

which we can test with

```
1  let randu = newMCG(seed = 1071914055)
2  for i in 0 ..< 10: echo randu.random()
```

The output numbers "look" random but are not truly random. Running the same code with the same seed generates the same output. Notice the following:

- PRNGs are typically implemented as a recursive expression that, given x_{i-1}, produces x_i.
- PRNGs have to start from an initial value, x_0, called the *seed*. A typical choice is to set the seed equal to the number of seconds from the conventional date and time "Thu Jan 01 01:00:00 1970." This is not always a good choice.
- PRNGs are periodic. They generate numbers in a finite set and then they repeat themselves. It is desirable to have this set as large as possible.
- PRNGs depend on some parameters (e.g., a and m). Some parameter choices lead to trivial random number generators. In general, some choices are better than others, and a few are optimal. In particular, the values of a and m determine the period of the random number generator. An optimal choice is the one with the longest period.

For a linear congruential generator, because of the `mod` operation, the period is always less than or equal to m. When c is nonzero, the period is equal to m only if c and m are relatively prime, $a - 1$ is divisible by all prime factors of m, and $a - 1$ is a multiple of 4 when m is a multiple of 4.

In the case of RANDU, the period is $m/4$. A better choice is using $a = 7^5$

and $m = 2^{31} - 1$ (known as the Mersenne prime number) because it can be proven that m is in fact the period of the generator:

$$x_i = (7^5 x_{i-1}) \mathrm{mod} (2^{31} - 1) \tag{6.10}$$

Here are some examples of MCG used by various systems:

Source	m	a	c
Numerical Recipes	2^{32}	1664525	1013904223
glibc (used by GCC)	2^{32}	1103515245	12345
Apple CarbonLib	$2^{31} - 1$	16807	0
java.util.Random	2^{48}	25214903917	11

When c is set to zero, a linear congruential generator is also called a multiplicative congruential generator.

Modern programming-language standard libraries no longer rely on linear congruential generators, because LCGs — even with carefully tuned parameters — exhibit hyperplane patterns in higher dimensions and have relatively short periods. The PRNGs shipped with current language runtimes use different recurrences (XOR/rotate/shift over a small word-aligned state, twisted feedback shift registers, counter-based PRFs, etc.) that pass much stronger empirical tests:

Source	Algorithm	State (bits)	Period
Python `random`	Mersenne Twister	19 937	$2^{19937} - 1$
C++ `std::mt19937`	Mersenne Twister	19 937	$2^{19937} - 1$
NumPy ($\geq$ 1.17)	PCG64	128	2^{128}
Nim `std/random`	xoroshiro128+	128	$2^{128} - 1$

Among these, **xoroshiro128+** (the default of Nim's standard library `std/random` module) is the one we recommend for the Monte Carlo simulations later in this book whenever raw speed matters. It updates a 128-bit state with a handful of XOR, rotate, and shift instructions per draw — noticeably faster than MT19937 or PCG64 on modern hardware — and its period of $2^{128} - 1$ is more than sufficient for any practical simulation. Cryptography is the one context in which we do *not* use it: like the LCGs above, xoroshiro128+ is a statistical generator, not a cryptographically secure one (see the discussion of CSPRNGs later in this section).

6.4.2 Defects of PRNGs

The non-randomness of pseudo random number generators manifests itself in at least two different ways:

- The sequence of generated numbers is periodic, therefore only a finite set of numbers can come out of the generator, and many of the numbers will never be generated. This is not a major problem if the period is much larger (some order of magnitude) than the number of random numbers needed in the Monte Carlo computation.
- The sequence of generated numbers presents bias in the form of "patterns." Sometimes these patterns are evident, sometimes they are not evident. Patterns exist because the pseudo random numbers are not random but are generated using a recursive formula. The existence of these patterns may introduce a bias in Monte Carlo computations that use the generator. This is a nasty problem, and the implications depend on the specific case.

An example of pattern/bias is discussed in ref. [41] and can be seen in fig. 6.4.2.

6.4.3 Multiplicative recursive generator

Another modification of the multiplicative congruential generator is the following:

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + ... + a_k x_{i-k}) \mathrm{mod} m \tag{6.11}$$

The advantage of this generator is that if m is prime, the period of this type of generator can be as big as $m^k - 1$. This is much larger than a simple multiplicative congruential generator.

An example is $a_1 = 107374182$, $a_2 = a_3 = a_4 = 0$, $a_5 = 104480$, and $m = 2^{31} - 1$, where the period is

$$(2^{31} - 1)^5 - 1 \simeq 4.56 \times 10^{46} \tag{6.12}$$

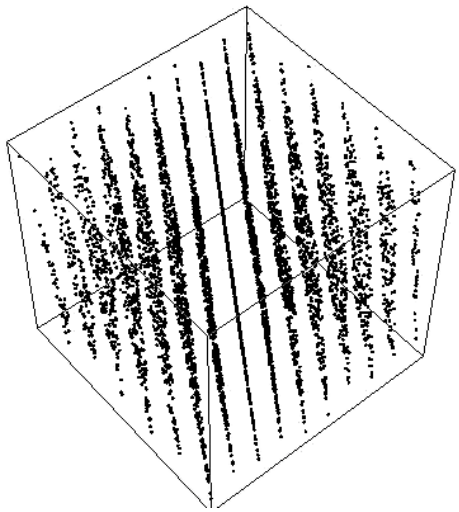

Figure 6.1: In this plot, each three consecutive random numbers (from RANDU) are interpreted as (x, y, z) coordinates of a random point. The image clearly shows the points are not distributed at random. Image from ref. [41].

6.4.4 Lagged Fibonacci generator

$$x_i = (x_{i-j} + x_{i-k}) \mathrm{mod} m \tag{6.13}$$

This is similar to the multiplicative recursive generator earlier. If m is prime and $j \neq k$, the period can be as large as $m^k - 1$.

6.4.5 Marsaglia's add-with-carry generator

$$x_i = (x_{i-j} + x_{i-k} + c_i) \mathrm{mod} m \tag{6.14}$$

where $c_1 = 0$ and $c_i = 1$ if $(x_{i-1-j} + x_{i-1-k} + c_{i-1}) < m$, 0

6.4.6 Marsaglia's subtract-and-borrow generator

$$x_i = (x_{i-j} - x_{i-k} - c_i) \mathrm{mod} m \tag{6.15}$$

where $k > j > 0$, $c_1 = 0$, and $c_i = 1$ if $(x_{i-1-j} - x_{i-1-k} - c_{i-1}) < 0$, 0 otherwise.

6.4.7 Lüscher's generator

The Marsaglia's subtract-and-borrow is a very popular generator, but it is known to have some problems. For example, if we construct vector

$$v_i = (x_i, x_{i+1}, ..., x_{i+k}) \tag{6.16}$$

and the coordinates of the point v_i are numbers closer to each other then the coordinates of the point v_{i+k} are also close to each other. This indicates that there is an unwanted correlation between the points $x_i, x_{i+1}, ..., x_{i+k}$. Lüscher observed [40] that the Marsaglia's subtract-and-borrow is equivalent to a chaotic discrete dynamic system, and the preceding correlation dies off for points that distance themselves more than k. Therefore he proposed to modify the generator as follows: instead of taking all x_i numbers, read k successive elements of the sequence, discard $p - k$ numbers, read k numbers, and so on. The number p has to be chosen to be larger than k. When $p = k$, the original Marsaglia generator is recovered.

6.4.8 Knuth's polynomial congruential generator

$$x_i = (ax_{i-1}^2 + bx_{i-1} + c) \mathrm{mod} m \tag{6.17}$$

This generator takes the form of a more complex function. It makes it harder to guess one number in the sequence from the following numbers; therefore it finds applications in cryptography.

Another example is the Blum, Blum, and Shub generator:

$$x_i = x_{i-1}^2 \mathrm{mod} m \tag{6.18}$$

6.4.9 PRNGs in cryptography

Random numbers find many applications in cryptography. For example, consider the problem of generating a random password, a digital signature, or random encryption keys for the Diffie–Hellmann and the RSA encryption schemes.

A cryptographically secure pseudo random number generator (CSPRNG) is a pseudo random number generator (PRNG) with properties that make it suitable for use in cryptography.

In addition to the normal requirements for a PRNG (that its output should pass all statistical tests for randomness) a CSPRNG must have two additional properties:

- It should be difficult to predict the output of the CSPRNG, wholly or partially, from examining previous outputs.
- It should be difficult to extract all or part of the internal state of the CSPRNG from examining its output.

Most PRNGs are not suitable for use as CSPRNGs. They must appear random in statistical tests, but they are not designed to resist determined mathematical reverse engineering.

CSPRNGs are designed explicitly to resist reverse engineering. There are a number of examples of CSPRNGs. Blum, Blum, and Shub has the strongest security proofs, though it is slow.

Many pseudo random number generators have the form

$$x_i = f(x_{i-1}, x_{i-2}, ..., x_{i-k}) \tag{6.19}$$

For example, the next random number depends on the past k numbers. Requirements for CSPRNGs used in cryptography are that

- Given $x_{i-1}, x_{i-2}, ..., x_{i-k}$, x_i can be computed in polynomial time, while
- Given $x_i, x_{i-2}, ..., x_{i-k}$, x_{i-1} must not be computable in polynomial time.

The first requirement means that the PRNG must be fast. The second requirement means that if a malicious agent discovers a random number used as a key, he or she cannot easily compute all previous keys generated using the same PRNG.

6.4.10 Inverse congruential generator

$$x_i = (ax_{i-1}^{-1} + c) \mathrm{mod} m \tag{6.20}$$

where x_{i-1}^{-1} is the multiplicative inverse of x_{i-1} modulo m, for example, $x_{i-1}x_{i-1}^{-1} = 1 \mathrm{mod} m$.

6.4.11 Mersenne twister

One of the best PRNG algorithms (because of its long period, uniform distribution, and speed) is the Mersenne twister, which produces a 53-bit random number, and it has a period of $2^{19937} - 1$ (this number is 6002 digits long!). Nim's `std/random` module is built on top of a Mersenne-twister-style generator (specifically Xoshiro/Mersenne, depending on the implementation). Although discussing the inner working of this algorithm is beyond the scope of these notes, we provide a pure-Nim implementation of the Mersenne twister:

Listing 6.2: in file: `nlib/randomgen.nim`

```
type
  MersenneTwister* = ref object
    w*: array[625, uint32]
    wi*: int

proc newMersenneTwister*(seed: uint32 = 4357'u32): MersenneTwister =
  result = MersenneTwister()
  result.w[0] = seed
  for i in 1 ..< 625:
    result.w[i] = 69069'u32 * result.w[i-1]
  result.wi = 624

proc random*(g: MersenneTwister): float =
  const
    N = 624
    M = 397
    U = 0x80000000'u32
    L = 0x7fffffff'u32
  let K = [0'u32, 0x9908b0df'u32]
  var y: uint32 = 0
  if g.wi >= N:
    var kk = 0
    while kk < N - M:
      y = (g.w[kk] and U) or (g.w[kk + 1] and L)
      g.w[kk] = g.w[kk + M] xor (y shr 1) xor K[y and 1'u32]
      inc kk
    while kk < N - 1:
      y = (g.w[kk] and U) or (g.w[kk + 1] and L)
      g.w[kk] = g.w[kk + (M - N)] xor (y shr 1) xor K[y and 1'u32]
```

```
      inc kk
    y = (g.w[N - 1] and U) or (g.w[0] and L)
    g.w[N - 1] = g.w[M - 1] xor (y shr 1) xor K[y and 1'u32]
    g.wi = 0
  y = g.w[g.wi]; inc g.wi
  y = y xor (y shr 11)
  y = y xor ((y shl 7) and 0x9d2c5680'u32)
  y = y xor ((y shl 15) and 0xefc60000'u32)
  y = y xor (y shr 18)
  result = float(y) / float(0xffffffff'u32)
```

In the above code, numbers starting with `0x` are represented in hexadecimal notation, and the `'u32` suffix fixes their type as unsigned 32-bit integers. The keywords `and`, `or`, `xor`, `shl`, and `shr` are Nim's bitwise operators (when applied to integer types): `and` is binary AND, `xor` is binary exclusive OR, `shl` shifts all bits to the left, and `shr` shifts all bits to the right. The Nim manual gives the full details.

6.5 Parallel generators and independent sequences

It is often necessary to generate many independent sequences.

For example, you may want to generate streams or random numbers in parallel using multiple machines and processes, and you need to ensure that the streams do not overlap.

A common mistake is to generate the sequences using the same generator with different seeds. This is not a safe procedure because it is not obvious if the seed used to generate one sequence belongs to the sequence generated by the other seed. The two sequences of random numbers are not independent, but they are merely shifted in respect to each other.

For example, here are two RANDU sequences generated with different

but dependent seeds:

seed	1071931562	50554362	
y_0	0.252659081481	0.867315522395	
y_1	0.0235412092879	0.992022250779	
y_2	0.867315522395	0.146293803118	(6.21)
y_3	0.992022250779	0.949562561698	
y_4	0.146293803118	0.380731142126	
y_5	...	...	

Note that the second sequence is the same as the first but shifted by two lines.

Three standard techniques for generating independent sequences are non-overlapping blocks, leapfrogging, and Lehmer trees.

6.5.1 Non-overlapping blocks

Let's consider one sequence of pseudo random numbers:

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{6.22}$$

One can break it into subsequences of k numbers:

$$x_0, x_1, ..., x_{k-1} \tag{6.23}$$

$$x_k, x_{k+1}, ..., x_{2k-1} \tag{6.24}$$

$$x_{2k}, x_{2k+1}, ..., x_{3k-1} \tag{6.25}$$

$$... \tag{6.26}$$

If the original sequence is created with a multiplicative congruential generator

$$x_i = ax_{i-1} \mathrm{mod} m \tag{6.27}$$

the subsequences can be generated independently because

$$x_{nk-1} = a^{nk-1} x_0 \mathrm{mod} m \tag{6.28}$$

if the seed of the arbitrary sequence is $x_{nk}, x_{nk+1}, ..., x_{nk-1}$. This is particularly convenient for parallel computers where one computer generates the seeds for the subsequences and the processing nodes, independently, generated the subsequences.

6.5.2 Leapfrogging

Another and probably more popular technique is leapfrogging. Let's consider one sequence of pseudo random numbers:

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{6.29}$$

One can break it into subsequences of k numbers:

$$x_0, x_k, x_{2k}, x_{3k}, ... \tag{6.30}$$

$$x_1, x_{1+k}, x_{1+2k}, x_{1+3k}, ... \tag{6.31}$$

$$x_2, x_{2+k}, x_{2+2k}, x_{2+3k}, ... \tag{6.32}$$

$$... \tag{6.33}$$

The seeds $x_1, x_2, ..x_{k-1}$ are generated from x_0, and the independent sequences can be generated independently using the formula

$$x_{i+k} = a^k x_i \text{mod} m \tag{6.34}$$

Therefore leapfrogging is a viable technique for parallel random number generators.

Here is an example of a usage of `leapfrog`:

Listing 6.3: in file: `nlib/randomgen.nim`

```
proc leapfrog*(mcg: MCG, k: int): seq[MCG] =
  # Compute (mcg.a^k) mod mcg.m incrementally to avoid integer
  # overflow: for typical seeds, mcg.a^k overflows int64 long before
  # `k` reaches a useful number of streams.
  var a = 1
  for _ in 0 ..< k:
    a = (a * mcg.a) mod mcg.m
  for i in 0 ..< k:
    result.add newMCG(mcg.next(), a, mcg.m)
```

Here is an example of usage:

```
let generators = leapfrog(newMCG(m), 3)
for k in 0 ..< 3:
  for i in 0 ..< 5:
    let x = generators[k].random()
    echo k, "\t", i, "\t", x
```

The Mersenne twister algorithm implemented in `os.random` has leapfrogging built in. In fact, the module includes a `random.jumpahead(n)` method that allows us to efficiently skip n numbers.

6.5.3 Lehmer trees

Lehmer trees are binary trees, generated recursively, where each node contains a random number. We start from the root containing the seed, x_0, and we append two children containing, respectively,

$$x_i^L = (a_L x_{i-1} + c_L) \mathrm{mod} m \tag{6.35}$$

$$x_i^R = (a_R x_{i-1} + c_R) \mathrm{mod} m \tag{6.36}$$

then, recursively, append nodes to the children.

6.6 Generating random numbers from a given distribution

In this section and the next, we provide examples of distributions other than uniform and algorithms to generate numbers using these distributions. The general strategy consists of finding ways to map uniform random numbers into numbers following a different distribution. There are two general techniques for mapping uniform into nonuniform random numbers:

- accept–reject (applies to both discrete and continuous distributions)
- inversion methods (applies to continuous distributions only)

Consider the problem of generating a random number x from a given distribution $p(x)$. The *accept–reject method* consists of generating x using a different distribution, $g(x)$, and a uniform random number, u, between 0, 1. If $u < p(x)/Mg(x)$ (M is the max of $p(x)/g(x)$), then x is the desired random number following distribution $p(x)$. If not, try another number.

To visualize why this works, imagine graphing the distribution p of the random variable x onto a large rectangular board and throwing darts at it, the coordinates of the dart being (x, u). Assume that the darts are uniformly distributed around the board. Now take off (reject) all of the

darts that are outside the curve. The remaining darts will be distributed uniformly within the curve, and the x-positions of these darts will be distributed according to the random variable's density. This is because there is the most room for the darts to land where the curve is highest and thus the probability density is greatest.

The g distribution is nothing but a shape so that all darts we throw are below it. There are two particular cases. In one case, $g = p$, we only throw darts below the p that we want; therefore we accept them all. This is the most efficient case, but it is not of practical interest because it means the accept–reject is not doing anything, as we already know now to generate numbers according to p. The other case is $g(x) = constant$. This means we generate the x uniformly before the accept–reject. This is equivalent to throwing the darts everywhere on the square board, without even trying to be below the curve p.

The inversion method instead is more efficient but requires some math. It states that if $F(x)$ is a cumulative distribution function and u is a uniform random number between 0 and 1, then $x = F^{-1}(u)$ is a random number with distribution $p(x) = f'(x)$. For those distributions where F can be expressed in analytical terms and inverted, the inversion method is the best way to generate random numbers. An example is the exponential distribution.

We will create a new class `RandomSource` that includes methods to generate the random number.

6.6.1 Uniform distribution

The uniform distributions are simple probability distributions which, in the discrete case, can be characterized by saying that all possible values are equally probable. In the continuous case, one says that all intervals of the same length are equally probable.

There are two types of uniform distribution: discrete and continuous.

Here we consider the discrete case as we implement it into a `randint` method:

Listing 6.4: in file: nlib/randomgen.nim

```
type
  RandomSource* = ref object
    generator*: proc(): float    # any uniform `[0, 1)` source

proc newRandomSource*(generator: proc(): float = nil): RandomSource =
  let g = if generator.isNil: (proc(): float = rand(1.0)) else: generator
  RandomSource(generator: g)

proc random*(r: RandomSource): float = r.generator()

proc randint*(r: RandomSource, a, b: int): int =
  a + int(float(b - a + 1) * r.random())
```

Notice that the random `RandomSource` constructor expects a generator such as `MCG`, `MersenneTwister`, or simply `random` (default value). The `random()` method is a proxy method for the equivalent method of the underlying generator object.

We can use `randint` to generate a random choice from a finite set when each option has the same probability:

Listing 6.5: in file: nlib/randomgen.nim

```
proc choice*[T](r: RandomSource, S: openArray[T]): T =
  S[r.randint(0, S.len - 1)]
```

6.6.2 Bernoulli distribution

The Bernoulli distribution, named after Swiss scientist James Bernoulli, is a discrete probability distribution that takes value 1 with probability of success p and value 0 with probability of failure $q = 1 - p$:

$$p(k) \equiv \left\{ \begin{array}{ll} p & \text{if } k = 1 \\ 1 - p & \text{if } k = 0 \\ 0 & \text{if not } k \in \{0, 1\} \end{array} \right\} \tag{6.37}$$

A Bernoulli random variable has an expected value of p and variance of pq.

We implement it by adding a corresponding method to the `RandomSource` class:

Listing 6.6: in file: nlib/randomgen.nim

```
proc bernoulli*(r: RandomSource, p: float): int =
  if r.random() < p: 1 else: 0
```

6.6.3 Biased dice and table lookup

A generalization of the Bernoulli distribution is a distribution in which we have a finite set of choices, each with an associated probability. The table can be a list of tuples `(value, probability)` or a dictionary of `value:probability`:

Listing 6.7: in file: nlib/randomgen.nim

```
proc lookup*[K](r: RandomSource, table: seq[(K, float)],
                epsilon = 1e-6): K =
  var u = r.random()
  for (key, p) in table:
    if u < p + epsilon: return key
    u = u - p
  raise newException(ArithmeticDefect, "invalid probability")
```

Let's say we want a random number generator that can only produce the outcome 0, 1 or 2 with known probabilities:

$$\text{Prob}(X = 0) = 0.50 \tag{6.38}$$

$$\text{Prob}(X = 1) = 0.23 \tag{6.39}$$

$$\text{Prob}(X = 2) = 0.27 \tag{6.40}$$

Because the probability of the possible outcomes are rational numbers (fractions), we can proceed as follows:

```
proc testLookup(nevents = 100,
                table = @[(0, 0.50), (1, 0.23), (2, 0.27)]) =
  let g = newRandomSource()
  var f = newSeq[int](table.len)
  for _ in 0 ..< nevents:
    let p = g.lookup(table)
    write(stdout, $p, " ")
    f[p] += 1
  echo ""
  for i in 0 ..< table.len:
    let freq = f[i] / nevents
    echo "frequency[", i, "]=", freq
```

which produces the following output:

```
0 1 2 0 0 0 2 2 2 2 2 0 0 0 2 1 1 2 0 0 2 1 2 0 1
0 0 0 0 0 0 0 0 0 0 1 2 2 0 0 1 2 2 0 0 1 0 0 1 0
0 0 0 0 0 2 2 0 2 0 2 0 0 0 0 2 1 2 0 2 0 2 0 0 0
0 0 0 2 2 0 0 0 0 2 1 1 0 2 0 0 0 0 0 1 0 1 0 0 0
frequency[0]=0.600000
frequency[1]=0.140000
frequency[2]=0.260000
```

Eventually, by repeating the experiment many more times, the frequencies of 0, 1 and 2 will approach the input probabilities.

Given the output frequencies, what is the probability that they are compatible with the input frequency? The answer to this question is given by the χ^2 and its distribution. We discuss this later in the chapter.

In some sense, we can think of the table lookup as an application of the linear search. We start with a segment of length 1, and we break it into smaller contiguous intervals of length $\text{Prob}(X = 0), \text{Prob}(X = 1), ..., \text{Prob}(x = n - 1)$ so that $\sum \text{Prob}(X = i) = 1$. We then generate a random point on the initial segment, and we ask in which of the n intervals it falls. The table lookup method linearly searches the interval.

This technique is $\Theta(n)$, where n is the number of outcomes of the computation. Therefore it becomes impractical if the number of cases is large. In this case, we adopt one of the two possible techniques: the Fishman–Yarberry method or the accept–reject method.

6.6.4 Fishman–Yarberry method

The Fishman–Yarberry [42] (F-Y) method is an improvement over the naive table lookup that runs in $O(\lceil \log_2 n \rceil)$. As the naive table lookup is an application of the linear search, the F-Y is an application of the binary search.

Let's assume that $n = 2^t$ is an exact power of 2. If this is not the case, we can always reduce to this case by adding new values to the lookup table corresponding to 0 probability. The basic data structure behind the F-Y method is an array of arrays a_{ij} built according to the following rules:

- $\forall j \geq 0, a_{0j} = \text{Prob}(X = x_j)$

- $\forall j \geq 0$ and $i > 0$, $a_{ij} = a_{i-1,2j} + a_{i-1,2j+1}$

Note that $0 \leq i < t$ and $\forall i \geq 0$, $0 \leq j < 2^{t-i}$, where $t = \log_2 n$. The array of arrays a can be represented as follows:

$$a_{ij} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \ldots & a_{0,n-1} \\ \ldots & \ldots & \ldots & \ldots & \\ a_{t-2,0} & a_{t-2,1} & a_{t-2,2} & a_{t-2,3} & \\ a_{t-1,0} & a_{t-1,1} & & & \end{pmatrix} \tag{6.41}$$

In other words, we can say that

- a_{ij} represents the probability

$$\text{Prob}(X = x_j) \tag{6.42}$$

- a_{1j} represents the probability

$$\text{Prob}(X = x_{2j} or X = x_{2j+1}) \tag{6.43}$$

- a_{4j} represents the probability

$$\text{Prob}(X = x_{4j} \text{ or } X = x_{4j+1} \text{ or } X = x_{4j+2} \text{ or } X = x_{4j+3}) \tag{6.44}$$

- a_{ij} represents the probability

$$\text{Prob}(X \in \{x_k | 2^i j \leq k < 2^i (j+1)\}) \tag{6.45}$$

This algorithm works like the binary search, and at each step, it confronts the uniform random number u with a_{ij} and decides if u falls in the range $\{x_k | 2^i j \leq k < 2^i (j+1)\}$ or in the complementary range $\{x_k | 2^i (j+1) \leq k < 2^i (j+2)\}$ and decreases i.

Here is the algorithm implemented as a class member function. The constructor of the class creates an array a once and for all. The method discrete_map maps a uniform random number u into the desired discrete integer:

```
type
  FishmanYarberry*[K] = ref object
    table*: seq[(K, float)]
    t*: int
    a*: seq[seq[float]]

proc newFishmanYarberry*[K](table0: seq[(K, float)]):
                           FishmanYarberry[K] =
  ## Pad table to a power-of-two length and precompute the F-Y tree.
  var table = table0
  var n = table.len
  while (n and (n - 1)) != 0:        # not a power of two
    table.add (default(K), 0.0)
    n = table.len
  let t = fastLog2(n)
  var a: seq[seq[float]] = @[]
  for i in 0 ..< t:
    var row: seq[float] = @[]
    if i == 0:
      for j in 0 ..< n: row.add table[j][1]
    else:
      let prev = a[i - 1]
      for j in 0 ..< (n shr i):
        row.add prev[2 * j] + prev[2 * j + 1]
    a.add row
  FishmanYarberry[K](table: table, t: t, a: a)

proc discreteMap*[K](fy: FishmanYarberry[K], u: float): K =
  var i = fy.t - 1
  var j = 0
  var b = 0.0
  var u = u
  while i > 0:
    if u > b + fy.a[i][j]:
      b += fy.a[i][j]
      j = 2 * j + 2
    else:
      j = 2 * j
    dec i
  if u > b + fy.a[i][j]:
    j += 1
  fy.table[j][0]
```

6.6.5 Binomial distribution

The binomial distribution is a discrete probability distribution that describes the number of successes in a sequence of n independent experi-

ments, each of which yields success with probability p. Such a success–failure experiment is also called a Bernoulli experiment.

A typical example is the following: 7% of the population are left-handed. You pick 500 people randomly. How likely is it that you get 30 or more left-handed? The number of left-handed you pick is a random variable X that follows a binomial distribution with $n = 500$ and $p = 0.07$. We are interested in the probability $\text{Prob}(X = 30)$.

In general, if the random variable X follows the binomial distribution with parameters n and p, the probability of getting exactly k successes is given by

$$p(k) = \text{Prob}(X = k) \equiv \binom{n}{k} p^k (1-p)^{n-k} \tag{6.46}$$

for $k = 0, 1, 2, ..., n$.

The formula can be understood as follows: we want k successes (p^k) and $n - k$ failures ($(1-p)^{n-k}$). However, the k successes can occur anywhere among the n trials, and there are $\binom{n}{k}$ different ways of distributing k successes in a sequence of n trials.

The mean is $\mu_X = np$, and the variance is $\sigma_X^2 = np(1-p)$.

If X and Y are independent binomial variables, then $X + Y$ is again a binomial variable; its distribution is

$$p(k) = \text{Prob}(X = k) = \binom{n_X + n_Y}{k} p^k (1-p)^{n-k} \tag{6.47}$$

We can generate random numbers following binomial distribution using a table lookup with table

$$\text{table}[k] = \text{Prob}(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{6.48}$$

For large n, it may be convenient to avoid storing the table and use the formula directly to compute its elements on a need-to-know basis. Moreover, because the table is accessed sequentially by the table lookup algorithm,

one may just notice that the current recursive relation holds:

$$\text{Prob}(X = 0) = (1-p)^n \tag{6.49}$$

$$\text{Prob}(X = k+1) = \frac{n}{k+1}\frac{p}{1-p}\text{Prob}(X = k) \tag{6.50}$$

This allows for a very efficient implementation:

Listing 6.8: in file: `nlib/randomgen.nim`

```
proc binomial*(r: RandomSource, n: int, p: float,
               epsilon = 1e-6): int =
  var u = r.random()
  var q = pow(1.0 - p, float(n))
  for k in 0 .. n:
    if u < q + epsilon: return k
    u = u - q
    q = q * float(n - k) / float(k + 1) * p / (1.0 - p)
  raise newException(ArithmeticDefect, "invalid probability")
```

6.6.6 Negative binomial distribution

In probability theory, the negative binomial distribution is the probability distribution of the number of trials n needed to get a fixed (nonrandom) number of successes k in a Bernoulli process. If the random variable X is the number of trials needed to get r successes in a series of trials where each trial has probability of success p, then X follows the negative binomial distribution with parameters r and p:

$$p(n) = \text{Prob}(X = n) = \binom{n-1}{k-1} p^k (1-p)^{n-k} \tag{6.51}$$

Here is an example:

John, a kid, is required to sell candy bars in his neighborhood to raise money for a field trip. There are thirty homes in his neighborhood, and he is told not to return home until he has sold five candy bars. So the boy goes door to door, selling candy bars. At each home he visits, he has a 0.4 probability of selling one candy bar and a 0.6 probability of selling nothing.

- What's the probability of selling the last candy bar at the nth house?

$$p(n) = \binom{n-1}{4} 0.4^5 0.6^{n-5} \tag{6.52}$$

- What's the probability that he finishes on the tenth house?

$$p(10) = \binom{9}{4} 0.4^5 0.6^5 = 0.10 \tag{6.53}$$

- What's the probability that he finishes on or before reaching the eighth house? Answer: To finish on or before the eighth house, he must finish at the fifth, sixth, seventh, or eighth house. Sum those probabilities:

$$\sum_{i=5,6,7,8} p(i) = 0.1737 \tag{6.54}$$

- What's the probability that he exhausts all houses in the neighborhood without selling the five candy bars?

$$1 - \sum_{i=5,..,30} p(i) = 0.0015 \tag{6.55}$$

As with the binomial distribution, we can find an efficient recursive formula for the negative binomial distribution:

$$\text{Prob}(X = k) = p^k \tag{6.56}$$

$$\text{Prob}(X = n+1) = \frac{n}{n-k+1}(1-p)\text{Prob}(X = n) \tag{6.57}$$

This allows for a very efficient implementation:

Listing 6.9: in file: `nlib/randomgen.nim`

```
proc negativeBinomial*(r: RandomSource, k: int, p: float,
                       epsilon = 1e-6): int =
  var u = r.random()
  var n = k
  var q = pow(p, float(k))
  while true:
    if u < q + epsilon: return n
    u = u - q
    q = q * float(n) / float(n - k + 1) * (1.0 - p)
    n += 1
```

Notice once again that, unlike the binomial case, here k is fixed, not n, and the random variable has a minimum value of k but no upper bound.

6.6.7 Poisson distribution

The Poisson distribution is a discrete probability distribution discovered by Siméon-Denis Poisson. It describes a random variable X that counts, among other things, the number of discrete occurrences (sometimes called *arrivals*) that take place during a time interval of given length. The probability that there are exactly x occurrences (x being a natural number including 0, $k = 0, 1, 2, ...$) is

$$p(k) = \text{Prob}(X = k) = e^{-\lambda} \frac{\lambda^k}{k!} \tag{6.58}$$

The Poisson distribution arises in connection with Poisson processes. It applies to various phenomena of discrete nature (i.e., those that may happen 0, 1, 2, 3, ..., times during a given period of time or in a given area) whenever the probability of the phenomenon happening is constant in time or space. The Poisson distribution differs from the other distributions considered in this chapter because it is different than zero for any natural number k rather than for a finite set of k values.

Examples include the following:

- The number of unstable nuclei that decayed within a given period of time in a piece of radioactive substance.
- The number of cars that pass through a certain point on a road during a given period of time.
- The number of spelling mistakes a secretary makes while typing a single page.
- The number of phone calls you get per day.
- The number of times your web server is accessed per minute.
- The number of roadkill you find per mile of road.
- The number of mutations in a given stretch of DNA after a certain

amount of radiation.

- The number of pine trees per square mile of mixed forest.
- The number of stars in a given volume of space.

The limit of the binomial distribution with parameters n and $p = \lambda/n$, for n approaching infinity, is the Poisson distribution:

$$\frac{n!}{(n-k)!k!}\left(\frac{\lambda}{n}\right)^k\left(1-\frac{\lambda}{n}\right)^{n-k} \simeq e^{-\lambda}\frac{\lambda^k}{k!} + O(\frac{1}{n}) \tag{6.59}$$

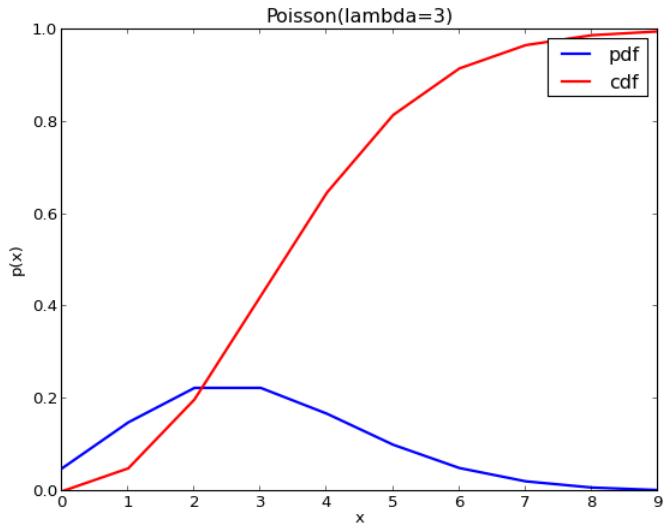

Figure 6.2: Example of Poisson distribution.

Intuitively, the meaning of λ is the following:

Let's consider a unitary time interval T and divide it into n subintervals of the same size. Let p_n be the probability of one success occurring in a single subinterval. For T fixed when $n \to \infty$, $p_n \to 0$ but the limit

$$\lim_{n\to\infty} pn \tag{6.60}$$

is finite. This limit is λ.

We can use the same technique adopted for the binomial distribution and observe that for Poisson,

$$\text{Prob}(X = 0) = e^{-\lambda} \tag{6.61}$$

$$\text{Prob}(X = k+1) = \frac{\lambda}{k+1}\text{Prob}(X = k) \tag{6.62}$$

therefore the preceding algorithm can be modified into

Listing 6.10: in file: `nlib/randomgen.nim`

```
proc poisson*(r: RandomSource, lamb: float, epsilon = 1e-6): int =
  var u = r.random()
  var q = exp(-lamb)
  var k = 0
  while true:
    if u < q + epsilon: return k
    u = u - q
    q = q * lamb / float(k + 1)
    k += 1
```

Note how this algorithm may take an arbitrary amount of time to generate a Poisson distributed random number, but eventually it stops. If u is very close to 1, it is possible that errors due to finite machine precision cause the algorithm to enter into an infinite loop. The $+\varepsilon$ term can be used to correct this unwanted behavior by choosing ε relatively small compared with the precision required in the computation, but larger than machine precision.

6.7 Probability distributions for continuous random variables

6.7.1 Uniform in range

A typical problem is generating random integers in a given range $[a, b]$, including the extreme. We can map uniform random numbers $y_i \in (0, 1)$ into integers by using the formula

$$h_i = a + \lfloor (b - a + 1)y_i \rfloor \tag{6.63}$$

Listing 6.11: in file: `nlib/randomgen.nim`

```
proc uniform*(r: RandomSource, a, b: float): float =
  a + (b - a) * r.random()
```

6.7.2 Exponential distribution

The exponential distribution is used to model Poisson processes, which are situations in which an object initially in state A can change to state B with constant probability per unit time λ. The time at which the state actually changes is described by an exponential random variable with parameter λ. Therefore the integral from 0 to T over $p(t)$ is the probability that the object is in state B at time T.

The probability mass function is given by

$$p(x) = \lambda e^{-\lambda x} \tag{6.64}$$

The exponential distribution may be viewed as a continuous counterpart of the geometric distribution, which describes the number of Bernoulli trials necessary for a discrete process to change state. In contrast, the exponential distribution describes the time for a continuous process to change state.

Examples of variables that are approximately exponentially distributed are as follows:

- the time until you have your next car accident
- the time until you get your next phone call
- the distance between mutations on a DNA strand
- the distance between roadkill

An important property of the exponential distribution is that it is memoryless: the chance that an event will occur s seconds from now does not depend on the past. In particular, it does not depend on how much time we have been waiting already. In a formula we can write this condition as

$$\text{Prob}(X > s + t | X > t) = \text{Prob}(X > s) \tag{6.65}$$

Any process satisfying the preceding condition is a Poisson process. The number of events per time unit is given by the Poisson distribution, and the time interval between consecutive events is described by the exponential distribution.

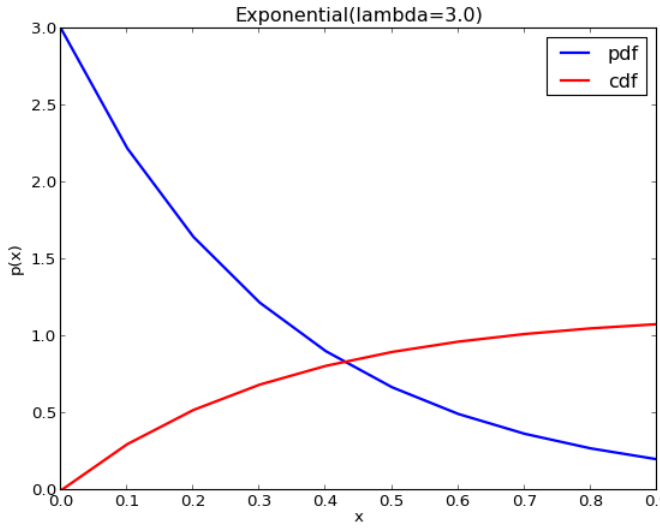

Figure 6.3: Example of exponential distribution.

The exponential distribution can be generated using the inversion method. The scope is to determine a function $x = f(u)$ that maps a uniformly distributed variable u into a continuous random variable x with probability mass function $p(x) = \lambda e^{-\lambda x}$.

According to the inversion method, we proceed by computing F:

$$F(x) = \int_0^x p(y)dy = 1 - e^{-\lambda x} \tag{6.66}$$

and we then invert $u = F(x)$, thus obtaining

$$x = -\frac{1}{\lambda} \log(1 - u) \tag{6.67}$$

Now notice that if u is uniform, $1 - u$ is also uniform; therefore we can

further simplify:

$$x = -\frac{1}{\lambda} \log u \tag{6.68}$$

We implement as follows:

Listing 6.12: in file: `nlib/randomgen.nim`

```
proc exponential*(r: RandomSource, lamb: float): float =
  -ln(r.random()) / lamb
```

This is an important distribution. Nim's `std/random` does not ship with a built-in exponential generator, but the inversion-based one above (`-ln(rand(1.0)) / lamb`) is a one-liner.

6.7.3 Normal/Gaussian distribution

The normal distribution (also known as *Gaussian distribution*) is an extremely important probability distribution considered in statistics. Here is the probability mass function:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{6.69}$$

where $E[X] = \mu$ and $E[(x-\mu)^2] = \sigma^2$.

The standard normal distribution is the normal distribution with a mean of 0 and a standard deviation of 1. Because the graph of its probability density resembles a bell, it is often called the *bell curve*.

The Gaussian distribution has two important properties:

- The average of many independent random variables with finite mean and finite variance tends to be a Gaussian distribution.
- The sum of two independent Gaussian random variables with means μ_1 and μ_2 and variances σ_1^2 and σ_2^2 is also a Gaussian random variable with mean $\mu = \mu_1 + \mu_2$ and variance $\sigma^2 = \sigma_1^2 + \sigma_2^2$.

There is no way to map a uniform random number into a Gaussian number but there is an algorithm to generate two independent Gaussian random numbers (y_1 and y_2) using two independent uniform random numbers (x_1 and x_2):

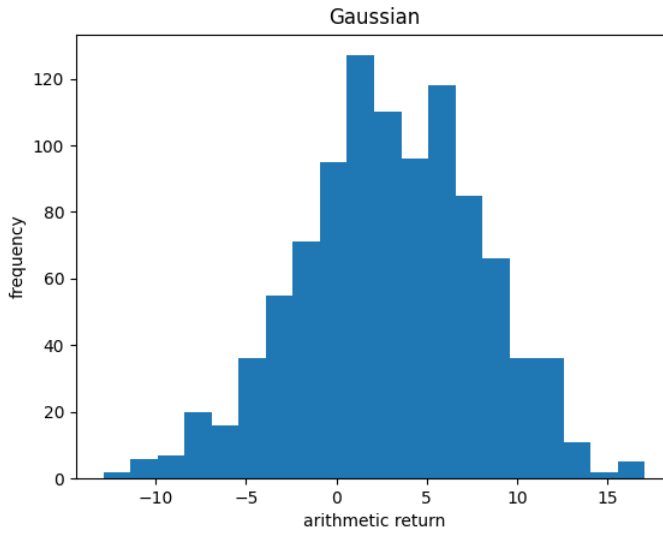

Figure 6.4: Example of Gaussian distribution.

- computing $v_1 = 2x_1 - 1, v_2 = 2x_2 - 1$ and $s = v_1^2 + v_2^2$
- if $s > 1$ start again
- $y_1 = v_1\sqrt{(-2/s)\log s}$ and $y_2 = v_2\sqrt{(-2/s)\log s}$

Listing 6.13: in file: `nlib/randomgen.nim`

```
type
  GaussRandomSource* = ref object
    base*: RandomSource
    other*: Option[float]   # cached second value from Marsaglia pair

proc newGaussRandomSource*(base: RandomSource): GaussRandomSource =
  GaussRandomSource(base: base, other: none(float))

proc gauss*(g: GaussRandomSource, mu = 0.0, sigma = 1.0): float =
  ## Marsaglia polar method: produces two Gaussians per call;
  ## one is cached for the next invocation.
  var thisVal: float
  if g.other.isSome:
    thisVal = g.other.get()
    g.other = none(float)
  else:
```

```
    var v1, v2, r: float
    while true:
      v1 = g.base.uniform(-1.0, 1.0)
      v2 = g.base.uniform(-1.0, 1.0)
      r = v1 * v1 + v2 * v2
      if r < 1: break
    thisVal = sqrt(-2.0 * ln(r) / r) * v1
    g.other = some(sqrt(-2.0 * ln(r) / r) * v2)
  mu + sigma * thisVal
```

Note how the first time the method next is called, it generates two Gaussian numbers (*this* and *other*), stores *other*, and returns *this*. Every other time, the method next is called if *other* is stored, and it returns a number; otherwise it recomputes *this* and *other* again.

To map a random Gaussian number y with mean 0 and standard deviation 1 into another Gaussian number y' with mean μ and standard deviation σ,

$$y' = \mu + y\sigma \tag{6.70}$$

We used this relation in the last line of the code.

This is also an important distribution; Nim's `std/random` provides it directly:

```
import std/random
echo gauss(mu, sigma)
```

Given a Gaussian random variable with mean μ and standard deviation σ, it is often useful to know how many standard deviations a correspond to a confidence c defined as

$$c = \int_{\mu-a\sigma}^{\mu+a\sigma} p(x)\mathrm{d}x \tag{6.71}$$

The following algorithm generates a table of a versus c given μ and σ:

Listing 6.14: in file: `nlib/randomgen.nim`

```
const CONFIDENCE* = [
  (0.68,    1.0),
  (0.80,    1.281551565545),
  (0.90,    1.644853626951),
  (0.95,    1.959963984540),
  (0.98,    2.326347874041),
```

```
  (0.99,    2.575829303549),
  (0.995,   2.807033768344),
  (0.998,   3.090232306168),
  (0.999,   3.290526731492),
  (0.9999,  3.890591886413),
  (0.99999, 4.417173413469)
]

proc confidenceIntervals*(mu, sigma: float): seq[(float, float, float)] =
  ## Returns (confidence, lower bound, upper bound) for a Gaussian.
  for (a, b) in CONFIDENCE:
    result.add (a, mu - b * sigma, mu + b * sigma)
```

6.7.4 Pareto distribution

The Pareto distribution, named after the economist Vilfredo Pareto, is a power law probability distribution that coincides with social, scientific, geophysical, actuarial, and many other types of observable phenomena. Outside the field of economics, it is sometimes referred to as the Bradford distribution. Its cumulative distribution function is

$$F(x) \equiv \text{Prob}(X < x) = 1 - \left(\frac{x_m}{x}\right)^{\alpha} \tag{6.72}$$

It can be implemented as follows using the inversion method:

Listing 6.15: in file: `nlib/randomgen.nim`

```
proc pareto*(r: RandomSource, alpha, xm: float): float =
  let u = r.random()
  xm * pow(1.0 - u, -1.0 / alpha)
```

When `RandomSource` is not available, the same closed-form expression `xm * pow(1.0 - rand(1.0), -1.0 / alpha)` can be used directly.

The Pareto distribution is an example of Levy distribution. The Central Limit theorem does not apply to it.

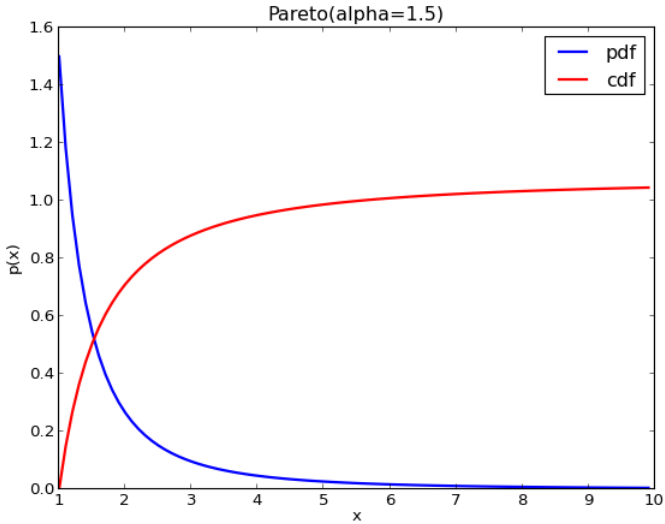

Figure 6.5: Example of Pareto distribution.

6.7.5 In and on a circle

We can generate a random point (x, y) uniformly distributed on a circle by generating a random angle.

$$x = \cos(2\pi u) \tag{6.73}$$
$$y = \sin(2\pi u) \tag{6.74}$$

Listing 6.16: in file: `nlib/randomgen.nim`

```
proc pointOnCircle*(r: RandomSource, radius = 1.0): (float, float) =
  let angle = 2.0 * PI * r.random()
  (radius * cos(angle), radius * sin(angle))
```

We can generate a random point uniformly distributed inside a circle by generating, independently, the x and y coordinates of points inside a square and rejecting those outside the circle:

Listing 6.17: in file: `nlib/randomgen.nim`

```
proc pointInCircle*(r: RandomSource, radius = 1.0): (float, float) =
  while true:
```

```
    let x = r.uniform(-radius, radius)
    let y = r.uniform(-radius, radius)
    if x * x + y * y < radius * radius:
      return (x, y)
```

6.7.6 In and on a sphere

A random point (x, y, z) uniformly distributed on a sphere of radius 1 is obtained by generating three uniform random numbers u_1, u_2, u_3,; compute $v_i = 2u_i - 1$, and if $v_1^2 + v_2^2 + v_3^2 \leq 1$,

$$x = v_1 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.75}$$

$$y = v_2 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.76}$$

$$z = v_3 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.77}$$

else start again.

Listing 6.18: in file: `nlib/randomgen.nim`

```
proc pointInSphere*(r: RandomSource, radius = 1.0):
                    (float, float, float) =
  while true:
    let x = r.uniform(-radius, radius)
    let y = r.uniform(-radius, radius)
    let z = r.uniform(-radius, radius)
    if x * x + y * y + z * z < radius * radius:
      return (x, y, z)

proc pointOnSphere*(r: RandomSource, radius = 1.0):
                    (float, float, float) =
  let (x, y, z) = r.pointInSphere(radius)
  let nrm = sqrt(x * x + y * y + z * z)
  (x / nrm, y / nrm, z / nrm)
```

6.8 Resampling

So far we always generated random numbers by modeling the random variable (e.g., uniform, or exponential, or Pareto) and using an algorithm to generate possible values of the random variables.

We now introduce a different methodology, which we will need later when talking about the bootstrap method. If we have a population S

of equally distributed events and we need to generate an event from the same distribution as the population, we can simply draw a random element from the population. In Nim, `std/random` provides the `sample` procedure:

```
import std/random
randomize()
let S = @[1, 2, 3, 4, 5, 6]
echo sample(S)
```

We can therefore generate a sample of random events by repeating this procedure. This is called *resampling* [43]:

Listing 6.19: in file: `nlib/randomgen.nim`

```
proc resample*[T](S: openArray[T], size = -1): seq[T] =
  let n = if size < 0: S.len else: size
  for _ in 0 ..< n: result.add sample(S)
```

6.9 Binning

Binning is the process of dividing a space of possible events into partitions and counting how many events fall into each partition. We can bin the numbers generated by a pseudo random number generator and measure the distribution of the random numbers.

Let's consider the following program:

```
import std/times

proc bin(generator: RandomSource, nevents: int,
         a, b: float, nbins: int): seq[float] =
  result = newSeq[float](nbins)
  for _ in 0 ..< nevents:
    let x = generator.random()
    if x >= a and x <= b:
      let k = int((x - a) / (b - a) * float(nbins))
      result[k] += 1.0
  for i in 0 ..< nbins:
    result[i] /= float(nevents)

proc testBin(nevents = 1000, nbins = 10) =
  let g = newRandomSource(
    proc(): float = newMCG(int(epochTime())).random())
  let bins = bin(g, nevents, 0.0, 1.0, nbins)
  for i, b in bins:
```

```
    echo i, " ", b

testBin()
```

It produces the following output:

```
i frequency[i]
0 0.101
1 0.117
2 0.092
3 0.091
4 0.091
5 0.122
6 0.096
7 0.102
8 0.090
9 0.098
```

Note that

- all bins have the same size 1/nbins;
- the size of the bins is normalized, and the sum of the values is 1
- the distribution of the events into bins approaches the distribution of the numbers generated by the random number generator

As an experiment, we can do the same binning on a larger number of events,

```
>>> test_bin(100000)
```

which produces the following output:

```
i frequency[i]
0 0.09926
1 0.09772
2 0.10061
3 0.09894
4 0.10097
5 0.09997
6 0.10056
7 0.09976
8 0.10201
9 0.10020
```

Note that these frequencies differ from 0.1 for less than 3%, whereas some of the preceding numbers differ from 0.11 for more than 20%.

7
Monte Carlo Simulations

7.1 Introduction

Monte Carlo methods are a class of algorithms that rely on repeated random sampling to compute their results, which are otherwise deterministic.

7.1.1 Computing π

The standard way to compute π is by applying the definition: π is the length of a semicircle with a radius equal to 1. From the definition, one can derive an exact formula:

$$\pi = 4 \arctan 1 \tag{7.1}$$

The arctan has the following Taylor series expansion:[1]:

$$\arctan x = \sum_{i=0} (-1)^i \frac{x^{2i+1}}{2i+1} \tag{7.8}$$

[1] Taylor expansion:

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + ... + \frac{1}{i!}f^{(i)}(0)x^2 + ... \tag{7.2}$$

and one can approximate π to arbitrary precision by computing the sum

$$\pi = \sum_{i=0} (-1)^i \frac{4}{2i+1} \tag{7.9}$$

We can use the program

```
proc piTaylor(n: int) =
  var pi = 0.0
  for i in 0 ..< n:
    pi = pi + 4.0 / float(2 * i + 1) * pow(-1.0, float(i))
    echo i, " ", pi

piTaylor(1000)
```

which produces the following output:

```
0 4.0
1 2.66666...
2 3.46666...
3 2.89523...
4 3.33968...
...
999 3.14..
```

A better formula is due to Plouffe,

$$\pi = \sum_{i=0} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right) \tag{7.10}$$

which we can implement as follows: we can use the program

```
# Nim does not ship with arbitrary-precision decimals in the stdlib;
# we use plain `float` here. For the high-precision form a `BigFloat`
# package such as `bignum` would replace the `float` type below.
```

and if $f(x) = \arctan x$ then:

$$f'(x) = \frac{d \arctan x}{dx} = \frac{1}{1+x^2} \rightarrow f'(0) = 1 \tag{7.3}$$

$$f''(x) = \frac{d^2 \arctan x}{d^2 x} = \frac{d}{dx} \frac{1}{1+x^2} = -\frac{2x}{(1+x^2)^2} \tag{7.4}$$

$$... \tag{7.5}$$

$$f^{(2i+1)}(x) = (-1)^i \frac{(2i)!}{(1+x^2)^{2i+1}} + x... \rightarrow f^{(2i+1)}(0) = (-1)(2i)! \tag{7.6}$$

$$f^{(2i)}(x) = x... \rightarrow f^{(2i)}(0) = 0 \tag{7.7}$$

```
4
5  proc piPlouffe(n: int): float =
6    var pi = 0.0
7    let a = 4.0
8    let b = 2.0
9    let c = 1.0
10   let d = 1.0 / 16.0
11   for i in 0 ..< n:
12     let i8 = float(8 * i)
13     pi += pow(d, float(i)) * (a/(i8+1.0) - b/(i8+4.0) -
14                               c/(i8+5.0) - c/(i8+6.0))
15   return pi
16
17 echo piPlouffe(1000)
```

The preceding formula works and converges very fast and already in 100 iterations produces

$$\pi = 3.14159265358979323846264 33... \tag{7.11}$$

There is a different approach based on the fact that π is also the area of a circle of radius 1. We can draw a square or area containing a quarter of a circle of radius 1. We can randomly generate points (x, y) with uniform distribution inside the square and check if the points fall inside the circle. The ratio between the number of points that fall in the circle over the total number of points is proportional to the area of the quarter of a circle ($\pi/4$) divided by the area of the square (1).

Here is a program that implements this strategy:

```
1  import std/random
2  randomize()
3
4  proc piMc(n: int) =
5    var counter = 0
6    for i in 0 ..< n:
7      let x = rand(1.0)
8      let y = rand(1.0)
9      if x * x + y * y < 1.0:
10       inc counter
11     let pi = 4.0 * float(counter) / float(i + 1)
12     echo i, " ", pi
13
14 piMc(1000)
```

The output of the algorithm is shown in fig. 7.1.1.

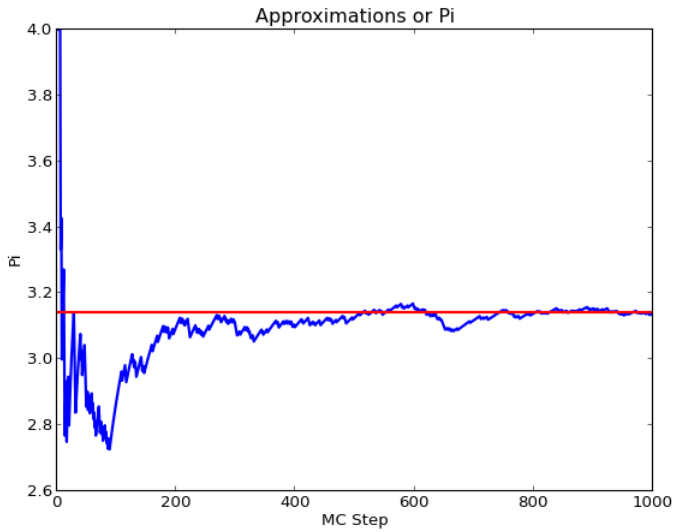

Figure 7.1: Convergence of **pi_mc**.

The convergence rate in this case is very slow, and this algorithm is of no practical value, but the methodology is sound, and for some problems, this method is the only one feasible.

Let's summarize what we have done: we have formulated our problem (compute π) as the problem of computing an area (the area of a quarter of a circle), and we have computed the area using random numbers. This is a particular example of a more general technique known as a Monte Carlo integration. In fact, the computation of an area is equivalent to the problem of computing an integral.

Sometimes the formula is not known, or it is too complex to compute reliably, hence a Monte Carlo solution becomes preferable.

7.1.2 Simulating an online merchant

Let's consider an online merchant. A website is visited many times a day. From the logfile of the web application, we determine that the average number of visitors in a day is 976, the number of visitors is Gaussian

distributed, and the standard deviation is 352. We also observe that each visitor has a 5% probability of purchasing an item if the item is in stock and a 2% probability to buy an item if the item is not in stock.

The merchant sells only one type of item that costs $100 per unit. The merchant maintains N items in stock. The merchant pays $30 a day to store each unit item in stock. What is the optimal N to maximize the average daily income of the merchant?

This problem cannot easily be formulated analytically or reduced to the computation of an integral, but it can easily be simulated.

In particular, we simulate many days and, for each day i, we start with N items in stock, and we loop over each simulated visitor. If the visitor finds an item in stock, he buys it with a 5% probability (producing an income of $70), whereas if the item is not in stock, he buys it with 2% probability (producing an income of $100). At the end of each day, we pay $30 for each item remaining in stock.

Here is a program that takes N (the number of items in stock) and d (the number of simulated days) and computes the average daily income:

```
proc simulateOnce(N: int): float =
  var profit = 0.0
  let loss = 30.0 * float(N)
  var instock = N
  let visitors = int(gauss(976.0, 352.0))
  for _ in 0 ..< visitors:
    if instock > 0:
      if rand(1.0) < 0.05:
        instock -= 1
        profit += 100.0
    else:
      if rand(1.0) < 0.02:
        profit += 100.0
  profit - loss

proc simulateMany(N: int, ap = 1.0, rp = 0.01, ns = 1000): float =
  var s = 0.0
  var muOld = 0.0
  for k in 1 ..< ns:
    s += simulateOnce(N)
    let mu = s / float(k)
    if k > 10 and mu - muOld < max(ap, rp * mu):
```

```
      return mu
    muOld = mu
  raise newException(ArithmeticDefect, "no convergence")
```

By looping over different N (items in stock), we can compute the average daily income as a function of N:

```
for N in countup(0, 99, 10):
  echo N, " ", simulateMany(N, ap = 100.0)
```

The program produces the following output:

```
n income
0 1955
10 2220
20 2529
30 2736
40 2838
50 2975
60 2944
70 2711
80 2327
90 2178
```

From this we deduce that the optimal number of items to carry in stock is about 50. We could increase the resolution and precision of the simulation by increasing the number of simulated days and reducing the step of the amount of items in stock.

Note that the statement `gauss(976, 352)` generates a random floating point number with a Gaussian distribution centered at 976 and standard deviation equal to 352, whereas the statement

```
if random()<0.05:
```

ensures that the subsequent block is executed with a probability of 5%.

The basic ingredients of every Monte Carlo simulation are here: (1) a function that simulates the system once and uses random variables to model unknown quantities; (2) a function that repeats the simulation many times to compute an average.

Any Monte Carlo solver comprises the following parts:

- A generator of random numbers (such as we have discussed in the previous chapter)

- A function that uses the random number generator and can simulate the system once (we will call x the result of each simulate once)
- A function that calls the preceding simulation repeatedly and averages the results until they converge $\mu = \frac{1}{N}\sum x_i$
- A function to estimate the accuracy of the result and determine when to stop the simulation, $\delta\mu <$ precision

7.2 Error analysis and the bootstrap method

The result of any MC computation is an average:

$$\mu = \frac{1}{N}\sum x_i \tag{7.12}$$

The error on this average can be estimated using the formula

$$\delta\mu = \frac{\sigma}{\sqrt{N}} = \sqrt{\frac{1}{N}\left(\frac{1}{N}\sum x_i^2 - \mu^2\right)} \tag{7.13}$$

This formula assumes the distribution of the x_i is Gaussian. Using this formula, we can compute a 68% confidence level for the MC computation of π, shown in fig. 7.2.

The purpose of the bootstrap [44] algorithm is computing the error in an average $\mu = (1/N)\sum x_i$ without making the assumption that the x_i are Gaussian.

The first step of the bootstrap methodology consists of computing the average not only on the initial sample $\{x_i\}$ but also on many data samples obtained by resampling the original data. If the number of elements N of the original sample were infinity, the average on each other sample would be the same. Because N is finite, each of these means produces slightly different results:

$$\mu_k = \frac{1}{N}\sum x_i^{[k]} \tag{7.14}$$

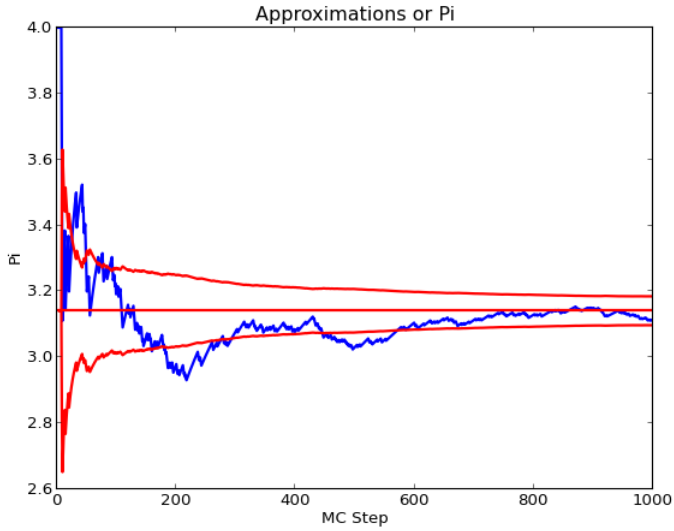

Figure 7.2: Convergence of π.

where $x_i^{[k]}$ is the ith element of resample k and μ_k is the average of that resample.

The second step of the bootstrap methodology consists of sorting the μ_k and finding two values μ^- and μ^+ that with a given percentage of the means follows in between those two values. The given percentage is the confidence level, and we set it to 68%.

Here is the complete algorithm:

Listing 7.1: in file: `nlib/montecarlo.nim`

```
proc bootstrap*(x: seq[float], confidence = 0.68, nsamples = 100):
                (float, float, float) =
  ## Bootstrap confidence interval. Returns (lower, mean, upper).
  var means: seq[float] = @[]
  for _ in 0 ..< nsamples: means.add mean(resample(x))
  means.sort()
  let leftTail = int(((1.0 - confidence) / 2.0) * float(nsamples))
  let rightTail = nsamples - 1 - leftTail
  (means[leftTail], mean(x), means[rightTail])
```

Here is an example of usage:

```
var S: seq[float] = @[]
for _ in 0 ..< 100: S.add gauss(2.0, 1.0)
echo bootstrap(S)
# (1.7767055865879007, 1.8968778392283303, 2.003420362236985)
```

In this example, the output consists of μ^-, μ, and μ^+.

Because S contains 100 random Gaussian numbers, with average 2 and standard deviation 1, we expect μ to be close to 2. We get 1.89. The bootstrap tells us that with 68% probability, the true average of these numbers is indeed between 1.77 and 2.00. The uncertainty $(2.00 - 1.77)/2 = 0.12$ is compatible with $\sigma/\sqrt{100} = 1/10 = 0.10$.

7.3 A general purpose Monte Carlo engine

We can now combine everything we have seen so far into a generic program that can be used to perform the most generic Monte Carlo computation/simulation:

Listing 7.2: in file: `nlib/montecarlo.nim`

```
type
  MCEngine* = ref object of RootObj
    results*: seq[float]
    convergence*: bool

method simulateOnce*(e: MCEngine): float {.base.} =
  raise newException(CatchableError, "simulateOnce not implemented")

proc simulateMany*(e: MCEngine, ap = 0.1, rp = 0.1, ns = 1000):
                  (float, float, float) =
  e.results = @[]
  var s1 = 0.0
  var s2 = 0.0
  e.convergence = false
  for k in 1 ..< ns:
    let x = e.simulateOnce()
    e.results.add x
    s1 += x
    s2 += x * x
    let mu = s1 / float(k)
    let variance = s2 / float(k) - mu * mu
    let dmu = sqrt(variance / float(k))
    if k > 10 and abs(dmu) < max(ap, abs(mu) * rp):
      e.convergence = true
```

```
25        break
26    e.results.sort()
27    bootstrap(e.results)
```

The preceding class has two methods:

- `simulate_once` is not implemented because the class is designed to be subclassed, and the method is supposed to be implemented for each specific computation.
- `simulate_many` is the part that stays the same; it calls `simulate_once` repeatedly, computes average and error analysis, checks convergence, and computes bootstrap error for the result.

It is also useful to have a function, which we call **var** (aka *value at risk* [45]), which computes a numerical value so that the output of a given percentage of the simulations falls below that value:

Listing 7.3: in file: `nlib/montecarlo.nim`

```
proc valueAtRisk*(e: MCEngine, confidence = 95): float =
  let index = int(0.01 * float(e.results.len) * float(confidence) + 0.999)
  if e.results.len - index < 5:
    raise newException(ArithmeticDefect, "not enough data, not reliable")
  e.results[index]
```

Now, as a first example, we can recompute π using this class:

```
type PiSimulator = ref object of MCEngine
method simulateOnce(e: PiSimulator): float =
  let x = rand(1.0); let y = rand(1.0)
  if x * x + y * y < 1.0: 4.0 else: 0.0

let s = PiSimulator()
echo s.simulateMany()
# (2.1818181818181817, 2.909090909090909, 3.6363636363636362)
```

Our engine finds that the value of π with 68% confidence level is between 2.18 and 3.63, with the most likely value of 2.90. Of course, this is incorrect, because it generates too few samples, but the bounds are correct, and that is what matters.

7.3.1 Value at risk

Let's consider a business subject to random losses, for example, a large bank subject to theft from employees. Here we will make the following

reasonable assumptions (which have been verified with data):

- There is no correlation between individual events.
- There is no correlation between the time when a loss event occurs and the amount of the loss.
- The time interval between losses is given by the exponential distribution (this is a Poisson process).
- The distribution of the loss amount is a Pareto distribution (there is a fat tail for large losses).
- The average number of losses is 10 per day.
- The minimum recorded loss is \$5000. The average loss is \$15, 000.

Our goal is to simulate one year of losses and to determine

- The average total yearly loss
- How much to save to make sure that in 95% of the simulated scenarios, the losses can be covered without going broke

From these assumptions, we determine that the $\lambda = 10$ for the exponential distribution and $x_m = 3000$ for the Pareto distribution. The mean of the Pareto distribution is $\alpha x_m/(\alpha - 1) = 15,000$, from which we determine that $\alpha = 1.5$.

We can answer the first questions (the average total loss) simply multiplying the average number of losses per year, 52×5, by the number of losses in one day, 10, and by the average individual loss, \$15, 000, thus obtaining

$$[\text{average yearly loss}] = \$39,000,000 \tag{7.15}$$

To answer the second question, we would need to study the width of the distribution. The problem is that, for $\alpha = 1.5$, the standard deviation of the Pareto distribution is infinity, and analytical methods do not apply. We can do it using a Monte Carlo simulation:

Listing 7.4: in file: `risk.nim`

```
1 import nlib
```

```
2
3  type
4    RiskEngine = ref object of MCEngine
5      lamb*, xm*, alpha*: float
6
7  proc newRiskEngine(lamb, xm, alpha: float): RiskEngine =
8    RiskEngine(lamb: lamb, xm: xm, alpha: alpha)
9
10 method simulateOnce(e: RiskEngine): float =
11   var totalLoss = 0.0
12   var t = 0.0
13   while t < 260:
14     let dt = -ln(rand(1.0)) / e.lamb
15     let amount = e.xm * pow(1.0 - rand(1.0), -1.0 / e.alpha)
16     t += dt
17     totalLoss += amount
18   totalLoss
19
20 proc main() =
21   randomize()
22   let s = newRiskEngine(lamb = 10.0, xm = 5000.0, alpha = 1.5)
23   echo s.simulateMany(rp = 1e-4, ns = 1000)
24   echo s.valueAtRisk(95)
25
26 main()
```

This produces the following output:

```
(38740147.179054834, 38896608.25084647, 39057683.35621854)
45705881.8776
```

The output of `simulate_many` should be compatible with the true result (defined as the result after an infinite number of iterations and at infinite precision) within the estimated statistical error.

The output of the `var` function answers our second questions: We have to save $45, 705, 881 to make sure that in 95% of cases our losses are covered by the savings.

7.3.2 Network reliability

Let's consider a network represented by a set of n_{nodes} nodes and n_{links} bidirectional links. Information packets travel on the network. They can originate at any node (`start`) and be addressed to any other node (`stop`). Each link of the network has a probability p of transmitting the packet

(success) and a probability $(1 - p)$ of dropping the packet (failure). The probability p is in general different for each link of the network.

We want to compute the probability that a packet starting in `start` finds a successful path to reach `stop`. A path is successful if, for a given simulation, all links in the path succeed in carrying the packet.

The key trick in solving this problem is in finding the proper representation for the network. Since we are not requiring to determine the exact path but only proof of existence, we use the concept of equivalence classes.

We say that two nodes are in the same equivalence class if and only if there is a successful path that connects the two nodes.

The optimal data structure to implement equivalence classes is `DisjSets`, discussed in chapter 3.

To simulate the system, we create a class `Network` that extends `MCEngine`. It has a `simulate_once` method that tries to send a packet from `start` to `stop` and simulates the network once. During the simulation each link of the network may be up or down with given probability. If there is a path connecting the `start` node to the `stop` node in which all links of the network are up, then the packet transfer succeeds. We use the DisjointSets to represent sets of nodes connected together. If there is a link up connecting a node from a set to a node in another set, then the two sets are joined. If, in the end, the `start` and `stop` nodes are found to belong to the same set, then there is a path and `simulate_once` returns 1, otherwise it returns 0.

Listing 7.5: in file: `network.nim`

```
import nlib, std/random

type
  NetworkReliability = ref object of MCEngine
    nNodes*, start*, stop*: int
    links*: seq[(int, int, float)]

proc newNetworkReliability(nNodes, start, stop: int): NetworkReliability =
  NetworkReliability(nNodes: nNodes, start: start, stop: stop)

proc addLink(s: NetworkReliability, i, j: int,
             failureProbability: float) =
  s.links.add (i, j, failureProbability)
```

```
14
15 method simulateOnce(s: NetworkReliability): float =
16   let nodes = newDisjointSets(s.nNodes)
17   for (i, j, pf) in s.links:
18     if rand(1.0) > pf:
19       discard nodes.join(i, j)
20   if nodes.joined(s.start, s.stop): 1.0 else: 0.0
21
22 proc main() =
23   randomize()
24   let s = newNetworkReliability(100, start = 0, stop = 1)
25   for _ in 0 ..< 300:
26     s.addLink(rand(99), rand(99), rand(1.0))
27   echo s.simulateMany()
28
29 main()
```

7.3.3 Critical mass

Here we consider the simulation of a chain reaction in a fissile material, for example, the uranium in a nuclear reactor [46]. We assume a material is in a spherical shape of known radius. At each point there is a probability of a nuclear fission, which we model as the emission of two neutrons. Each of the two neutrons travels and hits an atom, thus causing another fission. The two neutrons are emitted in random opposite directions and travel a distance given by the exponential distribution. The new fissions may occur inside material itself or outside. If outside, they are ignored. If the number of fission events inside the material grows exponentially with time, we have a self-sustained chain reaction; otherwise, we do not.

Fig. 7.3.3 provides a representation of the process.

Here is a possible implementation of the process. We store each event that happens inside the material in a queue (`events`). For each simulation, the queue starts with one event, and from it we generate two more, and so on. If the new events happen inside the material, we place the new events back in the queue. If the size of the queue shrinks to zero, then we are subcritical. If the size of the queue grows exponentially, we have a self-sustained chain reaction. We detect this by measuring the size of the queue and whether it exceeds a threshold (which we arbitrarily set to 200). The average free flight distance for a neutron in uranium is 1.91 cm.

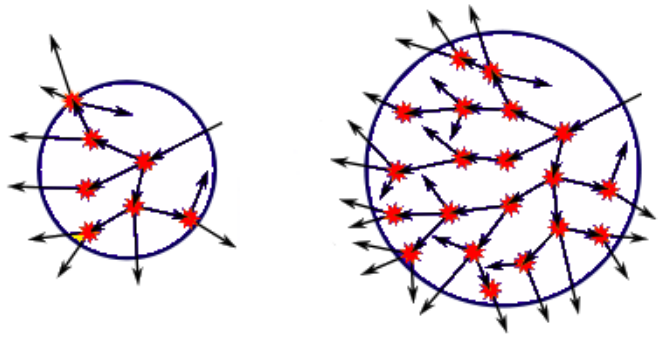

Figure 7.3: Example of chain reaction within a fissile material. If the mass is small, most of the decay products escape (left, sub-criticality), whereas if the mass exceeds a certain critical mass, there is a self-sustained chain reaction (right).

We use this number in our simulation. Given the radius of the material, `simulate_once` returns 1.0 if it detects a chain reaction and 0.0 if it does not. The output of `simulate_many` is the probability of a chain reaction:

Listing 7.6: in file: `nuclear.nim`

```
import nlib, std/random, std/math

type
  NuclearReactor = ref object of MCEngine
    radius*, density*: float
    threshold*: int

proc newNuclearReactor(radius: float, meanFreePath = 1.91,
                       threshold = 200): NuclearReactor =
  NuclearReactor(radius: radius, density: 1.0 / meanFreePath,
                 threshold: threshold)

proc pointOnSphereLocal(): (float, float, float) =
  while true:
    let x = rand(1.0); let y = rand(1.0); let z = rand(1.0)
    let d = sqrt(x*x + y*y + z*z)
    if d < 1: return (x/d, y/d, z/d)

method simulateOnce(s: NuclearReactor): float =
  var events: seq[(float, float, float)] = @[(0.0, 0.0, 0.0)]
  while events.len > 0:
    let p = events.pop()
    let (vx, vy, vz) = pointOnSphereLocal()
```

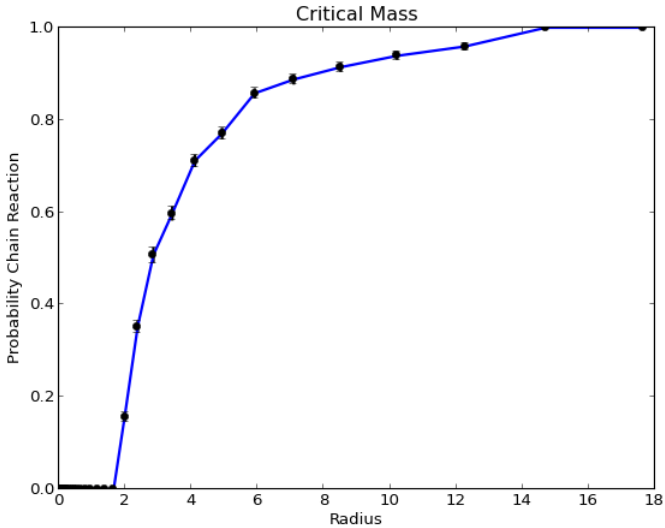

Figure 7.4: Probability of chain reaction in uranium.

```
    let d1 = -ln(rand(1.0)) / s.density
    let d2 = -ln(rand(1.0)) / s.density
    let p1 = (p[0] + vx*d1, p[1] + vy*d1, p[2] + vz*d1)
    let p2 = (p[0] - vx*d2, p[1] - vy*d2, p[2] - vz*d2)
    let r2 = s.radius * s.radius
    if p1[0]^2 + p1[1]^2 + p1[2]^2 < r2:
      events.add p1
    if p2[0]^2 + p2[1]^2 + p2[2]^2 < r2:
      events.add p2
    if events.len > s.threshold:
      return 1.0
  return 0.0

proc main() =
  randomize()
  var data: seq[(float, float, float)] = @[]
  var radius = 0.01
  while radius < 21.0:
    let s = newNuclearReactor(radius)
    let r = s.simulateMany(ap = 0.01, rp = 0.01, ns = 1000)
    data.add (radius, r[1], (r[2] - r[0]) / 2.0)
    radius *= 1.2
  saveErrorbar("nuclear.png",
               data.mapIt(it[0]), data.mapIt(it[1]), data.mapIt(it[2]),
```

```
48                 title = "Critical Mass",
49                 xlab = "Radius",
50                 ylab = "Probability of chain reaction")
51
52 main()
```

Fig. 7.3.3 shows the output of the program, the probability of a chain reaction as function of the size of the uranium mass. We find a critical radius between 2 cm and 10 cm, which corresponds to a critical mass between 0.5 kg and 60 kg. The official number is 15 kg for uranium 233 and 60 kg for uranium 235. The lesson to learn here is that it is not safe to accumulate too much fissile material together. This simulation can be easily tweaked to determine the thickness of a container required to shield a radioactive material.

7.4 Monte Carlo integration

7.4.1 One-dimensional Monte Carlo integration

Let's consider a one-dimensional integral

$$I = \int_a^b f(x)dx \tag{7.16}$$

Let's now determine two functions $g(x)$ and $p(x)$ such that

$$p(x) = 0 \text{ for } x \in [-\infty, a] \cup [n, \infty] \tag{7.17}$$

and

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \tag{7.18}$$

and

$$g(x) = f(x)/p(x) \tag{7.19}$$

We can interpret $p(x)$ as a probability mass function and

$$E[g(X)] = \int_{-\infty}^{+\infty} g(x)p(x)dx = \int_a^b f(x)dx = I \tag{7.20}$$

Therefore we can compute the integral by computing the expectation value of the function $g(X)$, where X is a random variable with a distribution (probability mass function) $p(x)$ different from zero in $[a, b]$ generated.

An obvious, although not in general an optimal choice, is

$$p(x) \equiv \left\{ \begin{array}{ll} 1/(b-a) & \text{if } x \in [a,b] \\ 0 & \text{otherwise} \end{array} \right\} \qquad (7.21)$$
$$g(x) \equiv (b-a)f(x)$$

so that X is just a uniform random variable in $[a, b]$. Therefore

$$I = E\left[g(X)\right] = \frac{1}{N}\sum_{i=0}^{i<N} g(x_i) \qquad (7.22)$$

This means that the integral can be evaluated by generating N random points x_i with uniform distribution in the domain, evaluating the integrand (the function f) on each point, averaging the results, and multiplying the average by the size of the domain $(b - a)$.

Naively, the error on the result can be estimated by computing the variance

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{i<N} \left[g(x_i) - \langle g \rangle\right]^2 \qquad (7.23)$$

with

$$\langle g \rangle = \frac{1}{N}\sum_{i=0}^{i<N} g(x_i) \qquad (7.24)$$

and the error on the result is given by:

$$\delta I = \sqrt{\frac{\sigma^2}{N}} \qquad (7.25)$$

The larger the set of sample points N, the lower the variance and the error. The larger N, the better $E[g(X)]$ approximates the correct result I.

Here is a Nim program:

Listing 7.7: in file: `integrate.nim`

```
import nlib, std/random, std/math

type
  MCIntegrator = ref object of MCEngine
    f*: proc(x: float): float
    a*, b*: float
```

```
7
8  proc newMCIntegrator(f: proc(x: float): float, a, b: float): MCIntegrator =
9    MCIntegrator(f: f, a: a, b: b)
10
11 method simulateOnce(s: MCIntegrator): float =
12   let x = s.a + (s.b - s.a) * rand(1.0)
13   (s.b - s.a) * s.f(x)
14
15 proc main() =
16   randomize()
17   let s = newMCIntegrator(proc(x: float): float = sin(x), 0.0, 1.0)
18   echo s.simulateMany()
19
20 main()
```

This technique is very general and can be extended to almost any integral assuming the integrand is smooth enough on the integration domain.

The choice (7.21) is not always optimal because the integrand may be very small in some regions of the integration domain and very large in other regions. Clearly some regions contribute more than others to the average, and one would like to generate points with a probability mass function that is as close as possible to the original integrand. Therefore we should choose a $p(x)$ according to the following conditions:

- $p(x)$ is very similar and proportional to $f(x)$
- given $F(x) = \int_{-\infty}^{x} p(x)dx$, $F^{-1}(x)$ can be computed analytically.

Any choice for $p(x)$ that makes the integration algorithm converge faster with less calls to `simulate_once` is called a *variance reduction technique*.

7.4.2 Two-dimensional Monte Carlo integration

The technique described earlier can easily be extended to two-dimensional integrals:

$$I = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 \tag{7.26}$$

where $\mathfrak{D}$ is some two-dimensional domain. We determine two functions $g(x_0, x_1)$ and $p_0(x_0), p_1(x_1)$ such that

$$p_0(x_0) = 0 \text{ or } p_1(x_1) = 0 \text{ for } x \notin \mathfrak{D} \tag{7.27}$$

and

$$\int p_0(x_0)p_1(x_1)dx_0dx_1 = 1 \tag{7.28}$$

and

$$g(x_0, x_1) = \frac{f(x_0, x_1)}{p_0(x_0)p_1(x_1)} \tag{7.29}$$

We can interpret $p(x_0, x_1)$ as a probability mass function for two independent random variables X_0 and X_1 and

$$E[g(X_0, X_1)] = \int g(x_0, x_1)p_0(x_0)p_1(x_1)dx = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 = I \tag{7.30}$$

Therefore

$$I = E[g(X_0, X_1)] = \frac{1}{N}\sum_{i=0}^{i<N} g(x_{i0}, x_{i1}) \tag{7.31}$$

7.4.3 *n*-dimensional Monte Carlo integration

The technique described earlier can also be extended to n-dimensional integrals

$$I = \int_{\mathfrak{D}} f(x_0, ..., x_{n-1})dx_0...dx_{n-1} \tag{7.32}$$

where $\mathfrak{D}$ is some n-dimensional domain identified by a function $domain(x_0, ..., x_{n-1})$ equal to 1 if $\mathbf{x} = (x_0, ..., x_{n-1})$ is in the domain, 0 otherwise. We determine two functions $g(x_0, ..., x_{n-1})$ and $p(x_0, ..., x_{n-1})$ such that

$$p(x_0, ..., x_{n-1}) = 0 \text{ for } x \notin \mathfrak{D} \tag{7.33}$$

and

$$\int p(x_0, ..., x_{n-1})dx_0...dx_{n-1} = 1 \tag{7.34}$$

and

$$g(x_0, ..., x_{n-1}) = f(x_0, ..., x_{n-1}) / p(x_0, ..., x_{n-1}) \tag{7.35}$$

We can interpret $p(x_0, ..., x_{n-1})$ as a probability mass function for n independent random variables $X_0...X_{n-1}$ and

$$E[g(X_0, ..., X_{n-1})] = \int g(x_0, ..., x_{n-1})p(x_0, ..., x_{n-1})dx \tag{7.36}$$

$$= \int_{\mathfrak{D}} f(x_0, ..., x_{n-1})dx_0...dx_{n-1} = I \tag{7.37}$$

Therefore

$$I = E[g(X_0, .., X_{n-1})] = \frac{1}{N} \sum_{i=0}^{i<N} g(\mathbf{x}_i) \tag{7.38}$$

where for every point $\mathbf{x}_i$ is a tuple $(x_{i0}, x_{i1}, ..., x_{i,n-1})$.

As an example, we consider the integral

$$I = \int_0^1 dx_0 \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 sin(x_0 + x_1 + x_2 + x_3) \tag{7.39}$$

Here is the code:

```
type
  MCIntegrator4D = ref object of MCEngine
    f*: proc(x: array[4, float]): float

method simulateOnce(s: MCIntegrator4D): float =
  let volume = 1.0
  var x: array[4, float]
  while true:
    for d in 0 ..< 4: x[d] = rand(1.0)
    var s2 = 0.0
    for d in 0 ..< 4: s2 += x[d] * x[d]
    if s2 < 1.0: break
  volume * s.f(x)

let s = MCIntegrator4D(
  f: proc(x: array[4, float]): float = sin(x[0] + x[1] + x[2] + x[3]))
echo s.simulateMany()
```

7.5 Stochastic, Markov, Wiener, and processes

A *stochastic process* [47] is a random function, for example, a function that maps a variable n with domain D into X_n, where X_n is a random variable with domain R. In practical applications, the domain D over which the function is defined can be a time interval (and the stochastic is called a *time series*) or a region of space (and the stochastic process is called a *random field*). Familiar examples of time series include *random walks* [48]; stock market and exchange rate fluctuations; signals such as speech, audio, and video; or medical data such as a patient's EKG, EEG, blood pressure, or temperature. Examples of random fields include static im-

ages, random topographies (landscapes), or composition variations of an inhomogeneous material.

Let's consider a grasshopper moving on a straight line, and let X_n be the position of the grasshopper at time $t = n\Delta_t$. Let's also assume that at time 0, $X_0 = 0$. The position of the grasshopper at each future ($t > 0$) time is unknown. Therefore it is a random variable.

We can model the movements of the grasshopper as follows:

$$X_{n+1} = X_n + \mu + \varepsilon_n \Delta_x \tag{7.40}$$

where Δ_x is a fixed step and ε_n is a random variable whose distribution depends on the model; μ is a constant drift term (think of wind pushing the grasshopper in one direction). It is clear that X_{n+1} only depends on X_n and ε_n; therefore the probability distribution of X_{n+1} only depends on X_n and the probability distribution of ε_n, but it does not depend on the past history of the grasshopper's movements at times $t < n\Delta_t$. We can write the statement by saying that

$$\text{Prob}(X_{n+1} = x | \{X_i\} \text{ for } i \leq n) = \text{Prob}(X_{n+1} = x | X_n) \tag{7.41}$$

A process in which the probability distribution of its future state only depends on the present state and not on the past is called a *Markov process* [49].

To complete our model, we need to make additional assumptions about the probability distribution of ε_n. We consider the two following cases:

- ε_n is a random variable with a Bernoulli distribution ($\varepsilon_n = +1$ with probability p and $\varepsilon_n = -1$ with probability $1 - p$).
- ε_n is a random variable with a normal (Gaussian) distribution with probability mass function $p(\varepsilon) = e^{-\varepsilon^2/2}$. Notice that the previous case (Bernoulli) is equivalent to this case (Gaussian) over long time intervals because the sum of many independent Bernoulli variables approaches a Gaussian distribution.

A continuous time stochastic process (when ε_n is a continuous random number) is called a *Wiener process* [50].

The specific case when ε_n is a Gaussian random variable is called an *Ito process* [51]. An Ito process is also a Wiener process.

7.5.1 Discrete random walk (Bernoulli process)

Here we assume a discrete random walk: ε_n equal to $+1$ with probability p and equal to -1 with probability $1-p$. We consider discrete time intervals of equal length Δ_t; at each time step, if $\varepsilon_n = +1$, the grasshopper moves forward one unit (Δ_x) with probability p, and if $\varepsilon_n = -1$, he moves backward one unit ($-\Delta_x$) with probability $1-p$.

For a total n steps, the probability of moving n_+ steps in a positive direction and $n_- = n - n_+$ in a negative direction is given by

$$\frac{n!}{n_+!(n-n_+)!} p^{n_+} (1-p)^{n-n_+} \tag{7.42}$$

The probability of going from $a = 0$ to $b = k\Delta_x > 0$ in a time $t = n\Delta_t > 0$ corresponds to the case when

$$n = n_+ + n_i \tag{7.43}$$

$$k = n_+ - n_- \tag{7.44}$$

that solved in n_+ gives $n_+ = (n+k)/2$, and therefore the probability of going from 0 to k in time $t = n\Delta_t$ is given by

$$\text{Prob}(n,k) = \frac{n!}{((n+k)/2)!((n-k)/2)!} p^{(n+k)/2} (1-p)^{(n-k)/2} \tag{7.45}$$

Note that $n+k$ has to be even, otherwise it is not possible for the grasshopper to reach $k\Delta_x$ in exactly n steps.

For large n, the following distribution in k/n tends to a Gaussian distribution.

7.5.2 Random walk: Ito process

Let's assume an Ito process for our random walk: ε_n is normally (Gaussian) distributed. We consider discrete time intervals of equal length

Δ_t, at each time step if $\varepsilon_n = \varepsilon$ with probability density function $p(\varepsilon) = \frac{1}{\sqrt{2\pi}} e^{-\varepsilon^2/2}$. It turns out that eq.(7.40) gives

$$X_n = n\mu + \Delta_x \sum_{i=0}^{i<n} \varepsilon_i \tag{7.46}$$

Therefore the location of the random walker at time $t = n\Delta_t$ is given by the sum of n normal (Gaussian) random variables:

$$p(X_n) = \frac{1}{\sqrt{2\pi n \Delta_x^2}} e^{-(X_n - n\mu)^2/(2n\Delta_x^2)} \tag{7.47}$$

Notice how the mean and the variance of X_n are both proportional to n, whereas the standard deviation is proportional to $\sqrt{n}$.

$$\begin{aligned}
\text{Prob}(a \;\leq\; X_n \leq b) &= \frac{1}{\sqrt{2\pi n \Delta_x^2}} \int_a^b e^{-(X_n - n\mu)^2/(2n\Delta_x^2)} dx && (7.48)\\
&= \frac{1}{\sqrt{2\pi}} \int_{(a-n\mu)/(\sqrt{n}\Delta_x)}^{(b-n\mu)/(\sqrt{n}\Delta_x)} e^{-x^2/2} dx && (7.49)\\
&= \frac{1}{2}\left[\text{erf}\left(\frac{b - n\mu}{\sqrt{2n}\,\Delta_x} \right) - \text{erf}\left(\frac{a - n\mu}{\sqrt{2n}\,\Delta_x} \right) \right] && (7.50)
\end{aligned}$$

where we have used the standard identity $\int_\alpha^\beta e^{-x^2/2}\, dx/\sqrt{2\pi} = \frac{1}{2}[\text{erf}(\beta/\sqrt{2}) - \text{erf}(\alpha/\sqrt{2})]$ relating the standard normal CDF to the error function.

7.6 Option pricing

A European call option is a contract that depends on an asset S. The contract gives the buyer of the contract the right (the option) to buy S at a fixed price A some time in the future, even if the actual price S may be different. The actual current price of the asset is called the *spot price*. The buyer of the option hopes that the price of the asset, S_t, will exceed A, so that he will be able to buy it at a discount, sell it at market price, and make a profit. The seller of the option hopes this does not happen, so he earns the full sale price. For the buyer of the option, the worst case

scenario is not to be able to recover the price paid for the option, but there is no best case because, hypothetically, he can make an arbitrarily large profit. For the seller, it is the opposite. He has an unlimited liability.

In practice, a call option allows a buyer to sell risk (the risk of the price of S going up) to the seller. He pays a price for it, the cost of the option. This is a form of insurance. There are two types of people who trade options: those who are willing to pay to get rid of risk (because they need the underlying asset and want it at a guaranteed price) and those who simply speculate (buy risk and sell insurance). On average, speculators make money because, if they sell many options, risk averages out, and they collect the premiums (the cost of the options).

The European option has a term or expiration, τ. It can only be exercised at expiration. The amount A is called the *strike price*.

The value at expiration of a European call option is

$$\max(S_\tau - A, 0) \tag{7.51}$$

Its present value is therefore

$$\max(S_\tau - A, 0)e^{-r\tau} \tag{7.52}$$

where r is the risk-free interest rate. This value corresponds to how much we would have to borrow today from a bank so that we can repay the bank at time τ with the profit from the option.

All our knowledge about the future spot price $x = S_\tau$ of the underlying asset can be summarized into a probability mass function $p_\tau(x)$. Under the assumption that $p_\tau(x)$ is known to both the buyer and the seller of the option, it has to be that the averaged net present value of the option is zero for any of the two parties to want to enter into the contract. Therefore

$$C_{call} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(x - A, 0) p_\tau(x) dx \tag{7.53}$$

Similarly, we can perform the same computations for a put option. A put option gives the buyer the option to sell the asset on a given day at a fixed

price. This is an insurance against the price going down instead of going up. The value of this option at expiration is

$$\max(A - S_\tau, 0) \tag{7.54}$$

and its pricing formula is

$$C_{put} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(A - x, 0) p_\tau(x) dx \tag{7.55}$$

Also notice that $C_{call} - C_{put} = S_0 - Ae^{-r\tau}$. This relation is called the *call-put parity*.

Our goal is to model $p_\tau(x)$, the distribution of possible prices for the underlying asset at expiration of the option, and compute the preceding integrals using Monte Carlo.

7.6.1 Pricing European options: Binomial tree

To price an option, we need to know $p_\tau(S_\tau)$. This means we need to know something about the future behavior of the price S_τ of the underlying asset S (a stock, an index, or something else). In absence of other information (crystal ball or illegal insider's information), one may try to gather information from a statistical analysis of the past historic data combined with a model of how the price S_τ evolves as a function of time. The most typical model is the binomial model, which is a Wiener process. We assume that the time evolution of the price of the asset X is a stochastic process similar to a random walk. We divide time into intervals of size Δ_t, and we assume that in each time interval $\tau = n\Delta_t$, the variation in the asset price is

$$S_{n+1} = S_n u \text{ with probability } p \tag{7.56}$$

$$S_{n+1} = S_n d \text{ with probability } 1 - p \tag{7.57}$$

where $u > 1$ and $0 < d < 1$ are measures for historic data. It follows that for $\tau = n\Delta_t$, the probability that the spot price of the asset at expiration is $S_0 u^i d^{n-i}$ is given by

$$\text{Prob}(S_\tau = S_0 u^i d^{n-i}) = \binom{n}{i} p^i (1-p)^{n-i} \tag{7.58}$$

and therefore (the binomial probabilities already sum to one, so no extra normalization is needed)

$$C_{call} = e^{-r\tau} \sum_{i=0}^{n} \binom{n}{i} p^i (1-p)^{n-i} \max(S_0 u^i d^{n-i} - A, 0) \tag{7.59}$$

and

$$C_{put} = e^{-r\tau} \sum_{i=0}^{n} \binom{n}{i} p^i (1-p)^{n-i} \max(A - S_0 u^i d^{n-i}, 0) \tag{7.60}$$

The parameters of this model are u, d and p, and they must be determined from historical data. For example,

$$p = \frac{e^{r\Delta_t} - d}{u - d} \tag{7.61}$$

$$u = e^{\sigma\sqrt{\Delta_t}} \tag{7.62}$$

$$d = e^{-\sigma\sqrt{\Delta_t}} \tag{7.63}$$

where Δ_t is the length of the time interval, r is the risk-free rate, and σ is the volatility of the asset, that is, the standard deviation of the log returns.

Here is the code to simulate an asset price using a binomial tree:

```
proc binomialSimulation(S0, u, d, p: float, n: int): seq[float] =
  var S = S0
  for _ in 0 ..< n:
    result.add S
    if rand(1.0) < p: S = u * S
    else:             S = d * S
```

The function takes the present spot value, S_0, of the asset, the values of u, d and p, and the number of simulation steps and returns a list containing the simulated evolution of the stock price. Note that because of the exact formulas, eqs.(7.59) and (7.60), one does not need to perform a simulation unless the underlying asset is a stock that pays dividends or we want to include some other variable in the model.

This method works fine for European call options, but the method is not easy to generalize to other options, when it depends on the path of the asset (e.g., the asset is a stock that pays dividends). Moreover, to increase precision, one has to decrease Δ_t or redo the computation from the beginning.

The Monte Carlo method that we see next is slower in the simple cases but is more general and therefore more powerful.

7.6.2 Pricing European options: Monte Carlo

Here we adopt the Black–Scholes model assumptions. We assume that the time evolution of the price of the asset X is a stochastic process similar to a random walk [52]. We divide time into intervals of size Δ_t, and we assume that in each time interval $t = n\Delta_t$, the log return is a Gaussian random variable:

$$\log \frac{S_{n+1}}{S_n} = \text{gauss}(\mu\Delta_t, \sigma\sqrt{\Delta_t}) \tag{7.64}$$

There are three parameters in the preceding equation:

- Δ_t is the time step we use in our discretization. Δ_t is not a physical parameter; it has nothing to do with the asset. It has to do with the precision of our computation. Let's assume that $\Delta_t = 1$ day.
- μ is a drift term, and it represents the expected rate of return of the asset over a time scale of one year. For risk-neutral option pricing the relevant drift in the *log-return* formulation is not the risk-free rate r itself, but $r - \sigma^2/2$ (the Itô convexity correction): with this choice the discounted spot $e^{-rt}S_t$ is a martingale, which is the no-arbitrage condition behind the Black–Scholes price. Setting $\mu = r$ in the equation above instead of $\mu = r - \sigma^2/2$ overprices the option by a factor that grows with $\sigma^2\tau$.
- σ is called volatility, and it represents the typical size of stochastic fluctuations of the log-price over a time interval of one year.

Notice that this model is equivalent to the previous binomial model for large time intervals, in the same sense as the binomial distribution for

large values of n approximates the Gaussian distribution. For large T, converge to the same result.

Notice how our assumption that log-return is Gaussian is different and not compatible with Markowitz's assumption of modern portfolio theory (the arithmetic return is Gaussian). In fact, log returns and arithmetic returns cannot both be Gaussian. It is therefore incorrect to optimize a portfolio using MPT when the portfolio includes options priced using Black–Scholes. The price of an individual asset cannot be negative, therefore its arithmetic return cannot be negative and it cannot be Gaussian. Conversely, a portfolio that includes both short and long positions (the holder is the buyer and seller of options) can have negative value. A change of sign in a portfolio is not compatible with the Gaussian log-return assumption.

If we are pricing a European call option, we are only interested in S_T and not in S_t for $0 < t < T$; therefore we can choose $\Delta_t = T$. In this case, we obtain

$$S_T = S_0 exp(r_T) \tag{7.65}$$

and

$$p(r_T) \propto \exp\left(-\frac{(r_T - \mu T)^2}{2\sigma^2 T}\right) \tag{7.66}$$

This allows us to write the following:

Listing 7.8: in file: `options.nim`

```
import nlib, std/random, std/math

type
  EuropeanCallOptionPricer = ref object of MCEngine
    spotPrice*, mu*, sigma*, strike*, riskFreeRate*: float
    timeToExpiration*: int

proc presentValue(p: EuropeanCallOptionPricer, payoff: float): float =
  let dailyReturn = p.riskFreeRate / 250.0
  payoff * exp(-dailyReturn * float(p.timeToExpiration))

```

```
method simulateOnce(p: EuropeanCallOptionPricer): float =
  let T = float(p.timeToExpiration)
  let S = p.spotPrice
  let rT = gauss(p.mu * T, p.sigma * sqrt(T))
  let ST = S * exp(rT)
  let payoff = max(ST - p.strike, 0.0)
  p.presentValue(payoff)

proc main() =
  randomize()
  let pricer = EuropeanCallOptionPricer(
    spotPrice: 100.0,
    mu: 0.12 / 250.0,
    sigma: 0.30 / sqrt(250.0),
    strike: 110.0,
    timeToExpiration: 90,
    riskFreeRate: 0.05)
  echo pricer.simulateMany(ap = 0.01, rp = 0.01)

main()
```

7.6.3 Pricing any option with Monte Carlo

An option is a contract, and one can write a contract with many different clauses. Each of them can be implemented into an algorithm. Yet we can group them into three different categories:

- Non-path-dependent: They depend on the price of the underlying asset at expiration but not on the intermediate prices of the asset (path).
- Weakly path-dependent: They depend on the price of the underlying asset and events that may happen to the price before expiration, but they do not depend on when the events exactly happen.
- Strongly path-dependent: They depend on the details of the time variation of price of the underlying asset before expiration.

Because non-path-dependent options do not depend on details, it is often possible to find approximate analytical formulas for pricing the option. For weakly path-dependent options, usually the binomial tree approach of the previous section is a preferable approach. The Monte Carlo approach applies to the general case, for example, that of strongly path-dependent options.

We will use our `MCEngine` to implement a generic option pricer.

First we need to recognize the following:

- The value of an option at expiration is defined by a payoff function $f(x)$ of the spot price of the asset at the expiration date. The fact that a call option has payoff $f(x) = \max(x - A, 0)$ is a convention that defined the European call option. A different type of option will have a different payoff function $f(x)$.
- The more accurately we model the underlying asset, the more accurate will be the computed value of the option. Some options are more sensitive than others to our modeling details.

Note one never models the option. One only models the underlying asset. The option payoff is given. We only choose the most efficient algorithm based on the model and the option:

Listing 7.9: in file: `options.nim`

```
import nlib, std/random, std/math

type
  GenericOptionPricer = ref object of MCEngine
    spotPrice*, mu*, sigma*, strike*, riskFreeRate*: float
    timeToExpiration*: int
    payoff*: proc(p: GenericOptionPricer, path: seq[float]): float

proc model(p: GenericOptionPricer, dt = 1.0): float =
  gauss(p.mu * dt, p.sigma * sqrt(dt))

proc presentValue(p: GenericOptionPricer, payoff: float): float =
  let dailyReturn = p.riskFreeRate / 250.0
  payoff * exp(-dailyReturn * float(p.timeToExpiration))

method simulateOnce(p: GenericOptionPricer): float =
  var S = p.spotPrice
  var path = @[S]
  for _ in 0 ..< p.timeToExpiration:
    let r = p.model()
    S = S * exp(r)
    path.add S
  p.presentValue(p.payoff(p, path))

proc payoffEuropeanCall(p: GenericOptionPricer, path: seq[float]): float =
  max(path[^1] - p.strike, 0.0)
```

```
27
28 proc payoffEuropeanPut(p: GenericOptionPricer, path: seq[float]): float =
29   max(p.strike - path[^1], 0.0)
30
31 proc payoffExoticCall(p: GenericOptionPricer, path: seq[float]): float =
32   let last5 = path[^5 .. ^1]
33   var s = 0.0
34   for v in last5: s += v
35   max(s / float(last5.len) - p.strike, 0.0)
36
37 proc main() =
38   randomize()
39   let pricer = GenericOptionPricer(
40     spotPrice: 100.0,
41     mu: 0.12 / 250.0,
42     sigma: 0.30 / sqrt(250.0),
43     strike: 110.0,
44     timeToExpiration: 90,
45     riskFreeRate: 0.05,
46     payoff: payoffEuropeanCall)
47   echo pricer.simulateMany(ap = 0.01, rp = 0.01)
48
49 main()
```

This code allows us to price any option simply by changing the payoff function.

One can also change the model for the underlying using different assumptions. For example, a possible choice is that of including a model for market crashes, and on random days, separated by intervals given by the exponential distribution, assume a negative jump that follows the Pareto distribution (similar to the losses in our previous risk model). Of course, a change of the model requires a recalibration of the parameters.

7.7 Markov chain Monte Carlo (MCMC) and Metropolis

Until this point, all our simulations were based on independent random variables. This means that we were able to generate each random number independently of the others because all the random variables were uncorrelated. There are cases when we have the following problem.

We have to generate $\mathbf{x} = x_0, x_1, ..., x_{n-1}$, where $x_0, x_1, ..., x_{n-1}$ are n corre-

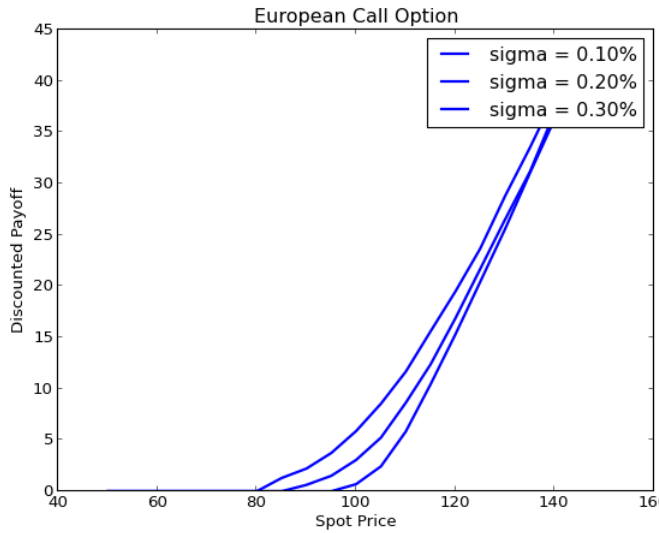

Figure 7.5: Price for a European call option for different spot prices and different values of σ.

lated random variables whose probability mass function

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{7.67}$$

cannot be factored, as in $p(x_0, x_1, ..., x_{n-1}) = p(x_0)p(x_1)...p(x_{n-1})$. Consider for example the simple case of generating two random numbers x_0 and x_1 both in $[0, 1]$ with probability mass function $p(x_0, x_1) = 6(x_0 - x_1)^2$ (note that $\int_0^1 \int_0^1 6p(x_0, x_1)dx_0dx_1 = 1$, as it should be).

In the case where each of the x_i has a Gaussian distribution and the only dependence between x_i and x_j is their correlation, the solution was already examined in a previous section about the Cholesky algorithm. Here we examine the most general case.

The Metropolis algorithm provides a general and simpler solution to this problem. It is not always the most efficient, but more sophisticated algorithms are nothing but refinements and extensions of its simple idea.

Let's formulate the problem once more: we want to generate $\mathbf{x} =$

$x_0, x_1, ..., x_{n-1}$ where $x_0, x_1, ..., x_{n-1}$ are n correlated random variables whose probability mass function is given by

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{7.68}$$

The procedure works as follows:

1 Start with a set of independent random numbers $\mathbf{x}^{(0)} = (x_0^{(0)}, x_1^{(0)}, ..., x_{n-1}^{(0)})$ in the domain.

2 Generate another set of independent random numbers $\mathbf{x}^{(i+1)} = (x_0^{(i+1)}, x_1^{(i+1)}, ..., x_{n-1}^{(i+1)})$ in the domain. This can be done by an arbitrary random function $Q(\mathbf{x}^{(i)})$. The only requirement for this function Q is that the probability of moving from a current point x to a new point y be the same as that of moving from a current point y to a new point x.

3 Generate a uniform random number z.

4 If $p(\mathbf{x}^{(i+1)})/p(\mathbf{x}^{(i)}) < z$, then $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}$.

5 Go back to step 2.

The set of random numbers $\mathbf{x}^{(i)}$ generated in this way for large values of i will have a probability mass function given by $p(\mathbf{x})$.

Here is a possible implementation in Nim using an iterator:

```
iterator metropolis(p: proc(x: seq[float]): float,
                    q: proc(x: seq[float]): seq[float],
                    x0: seq[float]): seq[float] =
  var x = x0
  while true:
    let xOld = x
    x = q(x)
    if p(x) / p(xOld) < rand(1.0):
      x = xOld
    yield x

proc P(x: seq[float]): float = 6.0 * (x[0] - x[1]) ^ 2
proc Q(x: seq[float]): seq[float] = @[rand(1.0), rand(1.0)]

var i = 0
for x in metropolis(P, Q, @[0.0, 0.0]):
  echo x
```

```
  inc i
  if i == 100: break
```

In this example, `Q` is the function that generates random points in the domain (in the example, $[0,1] \times [0,1]$), and `P` is an example probability $p(x) = 6(x_0 - x_1)^2$. We used a Nim `iterator` with the `yield` keyword instead of a `proc` with `return`: each `yield` hands a value back to the `for` loop, so values are generated lazily, one at a time, on demand.

Notice that the Metropolis algorithm can generate (and will generate) repeated values. This is because the next random vector x is highly correlated with the previous vector. For this reason, it is often necessary to de-correlate metropolis values by skipping some of them:

```
iterator metropolisDecorrelate(p: proc(x: seq[float]): float,
                               q: proc(x: seq[float]): seq[float],
                               x0: seq[float], ds = 100): seq[float] =
  var k = 0
  for x in metropolis(p, q, x0):
    inc k
    if k mod ds == ds - 1:
      yield x
```

The value of `ds` must be fixed empirically. The value of `ds` which is large enough to make the next vector independent from the previous one is called *decorrelation length*. This generator works as the previous one. For example:

```
var i = 0
for x in metropolisDecorrelate(P, Q, @[0.0, 0.0]):
  echo x
  inc i
  if i == 100: break
```

7.7.1 The Ising model

The Ising model, proposed by Wilhelm Lenz and analyzed in one dimension by his student Ernst Ising in 1925, is one of the most studied models of statistical mechanics. It captures, in a deceptively simple form, the essence of a phase transition: a competition between an ordering interaction — spins want to align with their neighbors — and thermal disorder, which tends to randomize them. Despite its simplicity, the model exhibits a sharp, genuine phase transition in two and higher dimensions and has

become a paradigm for the study of critical phenomena, magnetism, alloys, lattice gases, and even spin-glass-like models of neural networks.

The model is defined on a regular lattice (here a three-dimensional simple cubic lattice, but the same algorithm works in any dimension) where each site i carries a binary spin $s_i \in \{+1, -1\}$. The total energy of a configuration $\{s_i\}$ is

$$E(\{s_i\}) = -J \sum_{\langle i,j \rangle} s_i s_j - h \sum_i s_i, \tag{7.69}$$

where $J > 0$ is the coupling strength between nearest-neighbor pairs (the notation $\langle i, j \rangle$ means each unordered nearest-neighbor pair is counted once), h is an external magnetic field, and the first sum runs over the z nearest neighbors of each site ($z = 6$ for the simple cubic lattice, $z = 4$ for the square lattice, $z = 2$ for the chain). In what follows we set $J = 1$ and the Boltzmann constant $k_B = 1$, so temperatures and fields are expressed in units of J/k_B and J, respectively.

The aligned configuration with all s_i pointing in the direction of h minimizes both terms, so at zero temperature the system is fully ordered. As we feed energy in by raising the temperature T, thermal fluctuations randomly flip spins, and at high enough T the orientations become essentially independent. The probability of finding the system in a given configuration s in thermal equilibrium is the Boltzmann distribution

$$p(s) = \frac{1}{Z} \exp\left(-\frac{E(s)}{k_B T}\right), \qquad Z = \sum_s \exp\left(-\frac{E(s)}{k_B T}\right), \tag{7.70}$$

where Z is the partition function (which we will not need to compute explicitly).

A central result is that there exists a critical temperature T_c, depending on the lattice dimension, separating an ordered phase ($T < T_c$, nonzero spontaneous magnetization in the thermodynamic limit even at $h = 0$) from a disordered phase ($T > T_c$, zero magnetization at $h = 0$). In one dimension $T_c = 0$: thermal fluctuations destroy any long-range order at any finite temperature. In two dimensions Onsager solved the model exactly in 1944 and obtained $T_c = 2/\ln(1 + \sqrt{2}) \approx 2.269$ on the square

lattice. In three dimensions no closed-form solution is known, and high-precision numerical estimates on the simple cubic lattice give $T_c \approx 4.5115$. The simulation below is in three dimensions, so the crossover from order to disorder visible in fig. 7.7.1 should be looked for around this value.

We sample configurations from $p(s)$ using the Metropolis algorithm of the previous subsection. Two features of the Boltzmann distribution make the algorithm particularly cheap. First, the ratio

$$\frac{p(s')}{p(s)} = \exp\left(-\frac{E(s') - E(s)}{k_B T}\right) = \exp\left(-\frac{\Delta E}{k_B T}\right) \tag{7.71}$$

depends only on the energy difference ΔE, so the partition function Z cancels out. Second, the proposal Q that the Metropolis algorithm requires is free for us to choose. We pick the simplest possible local move: flip one randomly chosen spin $s_i \to -s_i$. With this choice ΔE involves only the chosen site and its neighbors, and a short calculation gives

$$\Delta E = 2 s_i \left(h + J \sum_{j \in \mathrm{nbrs}(i)} s_j \right), \tag{7.72}$$

which evaluates in constant time, independent of the lattice size. The Metropolis acceptance condition $\xi < \exp(-\Delta E / k_B T)$, with ξ uniformly distributed in $(0, 1)$, is therefore equivalent to $-\Delta E > k_B T \ln \xi$.

Two more practical points before we list the code. We use *periodic boundary conditions*: lattice site (x, y, z) is identified with $(x \pm n, y, z)$, $(x, y \pm n, z)$, and $(x, y, z \pm n)$, so the lattice has the topology of a 3-torus and there are no surface effects to bias the bulk behavior. (In the code, the modular arithmetic `((x mod n) + n) mod n` implements this wrap-around correctly even for negative indices, which is necessary because Nim's `mod` can return negative values.) And, before averaging any observable, the chain must be allowed to *equilibrate* from its initial state for some number of sweeps; the early samples are correlated with the initial condition and should be discarded.

Here is the code for a three-dimensional spin system:

Listing 7.10: in file: `ising.nim`

```
import nlib, std/random, std/math, std/sequtils

type
  Ising* = ref object
    n*: int
    s*: seq[seq[seq[int]]]
    magnetization*: int

proc newIsing*(n: int): Ising =
  result = Ising(n: n, magnetization: n * n * n)
  result.s = newSeqWith(n, newSeqWith(n, newSeqWith(n, 1)))

proc `[]`*(ising: Ising, x, y, z: int): int =
  let n = ising.n
  ising.s[((x mod n) + n) mod n]
              [((y mod n) + n) mod n]
              [((z mod n) + n) mod n]

proc `[]=`*(ising: Ising, x, y, z, value: int) =
  let n = ising.n
  ising.s[((x mod n) + n) mod n]
              [((y mod n) + n) mod n]
              [((z mod n) + n) mod n] = value

proc step*(ising: Ising, t, h: float): int =
  let n = ising.n
  let x = rand(n - 1); let y = rand(n - 1); let z = rand(n - 1)
  let nbrs = [(x-1, y, z), (x+1, y, z),
              (x, y-1, z), (x, y+1, z),
              (x, y, z-1), (x, y, z+1)]
  var sumNbrs = 0
  for (xn, yn, zn) in nbrs: sumNbrs += ising[xn, yn, zn]
  let dE = -2.0 * float(ising[x, y, z]) * (h + float(sumNbrs))
  if dE > t * ln(rand(1.0)):
    ising[x, y, z] = -ising[x, y, z]
    ising.magnetization += 2 * ising[x, y, z]
  ising.magnetization

proc simulate*(steps = 100): Table[int, seq[(float, float, float)]] =
  let ising = newIsing(10)
  for h in 0 .. 10:
    var row: seq[(float, float, float)] = @[]
    for t in 1 .. 10:
      var m: seq[float] = @[]
      for _ in 0 ..< steps:
        m.add float(ising.step(float(t), float(h)))
      row.add (float(t), mean(m), sd(m))
    result[h] = row

```

```
proc main*(name = "ising.png") =
  randomize()
  let data = simulate(steps = 10000)
  saveErrorbarSeries(name, data,
                     xlab = "temperature",
                     ylab = "magnetization")

main()
```

A note on the code: the variable named `dE` actually stores $-\Delta E$ rather than ΔE, because that is the quantity that appears directly in the acceptance test `dE > t * ln(rand(1.0))` (which is the rearrangement of $\xi < \exp(-\Delta E / k_B T)$ derived above). After accepting a flip, the magnetization is updated incrementally with `magnetization += 2 * ising[x, y, z]`: by the time this line runs the spin has already been flipped, so the increment $2 s_i^{\text{new}} = -2 s_i^{\text{old}}$ is exactly the change in $\sum_i s_i$. Updating the magnetization incrementally avoids re-summing the entire lattice on every accepted move and is essential to keep each Metropolis step $O(1)$.

Beyond the magnetization, two related observables are routinely measured in equilibrium and provide much sharper diagnostics of the phase transition than $\langle M \rangle$ alone:

- the magnetic susceptibility $\chi = \dfrac{1}{k_B T} \left(\langle M^2 \rangle - \langle M \rangle^2 \right)$, which diverges at T_c in the thermodynamic limit and develops a sharp finite-size peak that is the standard tool for locating the transition;
- the specific heat per spin $C = \dfrac{1}{N k_B T^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right)$, which also peaks at T_c.

Because of the $\mathbb{Z}_2$ symmetry $s_i \to -s_i$ at $h = 0$, an infinite-time average gives $\langle M \rangle = 0$ exactly even in the ordered phase: the simulation will spend long periods in the $+M$ basin and long periods in the $-M$ basin and average to zero. To diagnose the ordered phase one therefore measures $\langle |M| \rangle$ rather than $\langle M \rangle$.

Fig. 7.7.1 shows how the spins tend to align in the direction of the external magnetic field, but the larger the temperature (left to right), the more random they are, and the average magnetization tends to zero. The higher the external magnetic field (bottom to top curves), the longer it takes for

the transition from order (aligned spins) to chaos (random spins).

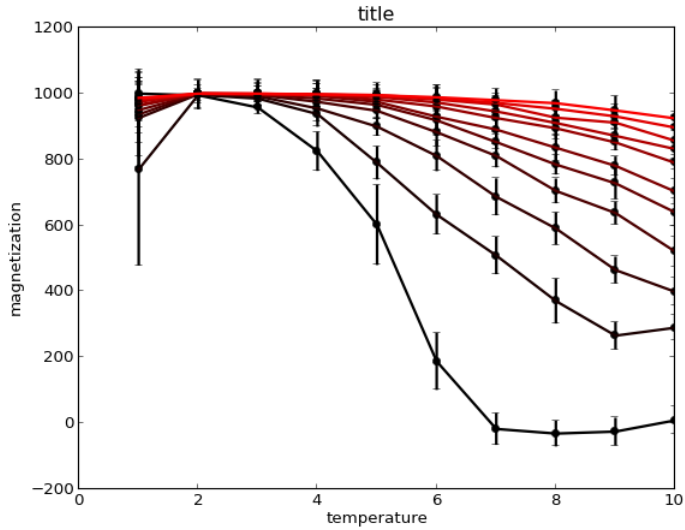

Figure 7.6: Average magnetization as a function of the temperature for a spin system.

Fig. 7.7.1 shows the two-dimensional section of some random three-dimensional states for different values of the temperature. One can clearly see that the lower the temperature, the more the spins are aligned, and the higher the temperature, the more random they are.

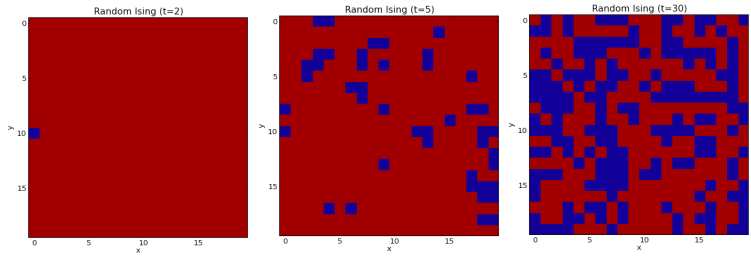

Figure 7.7: Random Ising states (2D section of 3D) for different temperatures.

A practical limitation of the local single-spin-flip Metropolis algorithm shown above is *critical slowing down*: as T approaches T_c, the autocorrelation time τ of any observable grows as $\tau \sim L^z$ with a dynamical exponent

$z \approx 2$, so doubling the linear lattice size L multiplies the cost of a thermalized run by roughly four. The reason is that near criticality the system develops large correlated domains; flipping a single spin deep inside such a domain almost always raises the energy and is therefore rejected. *Cluster algorithms* — the Swendsen–Wang (1987) and Wolff (1989) algorithms — bypass this problem by building entire correlated clusters of like-aligned spins and flipping them as a single update, bringing z down to nearly zero. They are the methods of choice for high-precision Ising simulations near the critical point and are recommended further reading for the reader interested in numerical statistical mechanics.

7.8 Simulated annealing

Simulated annealing is an application of Monte Carlo to solve optimization problems. It is best understood within the context of the Ising model. When the temperature is lowered, the system tends toward the state of minimum energy. At high temperature, the system fluctuates randomly and moves in the space of all possible states. This behavior is not specific to the Ising model. Hence, for any system for which we can define an energy, we can find its minimum energy state, by starting in a random state and slowly lowering the temperature as we evolve the simulation. The system will find a minimum. There may be more than one minimum, and one may need to repeat the procedure multiple times from different initial random states and compare the solutions. This process takes the name of annealing in analogy with the industrial process for removing impurities from metals: heat, cool slowly, repeat.

We can apply this process to any system for which we want to minimize a function $f(x)$ of multiple variables. We just have to think of x as the state s and of f as the energy E. This analogy is purely semantic because the quantity we want to minimize is not necessarily an energy in the physical sense.

Simulated annealing does not assume the function is differentiable or continuous in its variables.

7.8.1 Protein folding

In the following we apply simulated annealing to the problem of folding of a protein. A protein is a linear chain of amino acids, synthesized one residue at a time on the ribosome. As soon as the chain emerges into the aqueous environment of the cell it folds, typically within microseconds to seconds, into a specific three-dimensional shape (its *native state*) that determines its biological function. The driving force is largely the hydrophobic effect: some amino acids are hydrophobic (their side chains repel water), some are hydrophilic (they prefer contact with water), and the chain rearranges itself so that hydrophobic residues are buried in the protein's interior while hydrophilic ones face the solvent. Equivalently, the native state is the configuration that minimizes the area of hydrophobic surface exposed to water [53]. This is represented graphically in fig. 7.8.1.

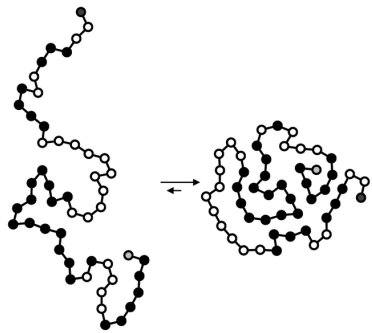

Figure 7.8: Schematic example of protein folding. The white circles are hydrophilic amino-acids. The black ones are hydrophobic.

Here we assume only two types of amino-acids (H for hydrophobic and P for hydrophilic), and we assume each amino acid is a cube, that all cubes have the same size, and that each two consecutive amino-acids are connected at a face. These assumptions greatly simplify the problem because they limit the possible solid angles to six possible values (0: up, 1: down, 2: right, 3: left, 4: front, 5: back). Our goal is arranging the cubes to minimize the number of faces of hydrophobic cubes that are exposed to water:

Listing 7.11: in file: folding.nim

```
import nlib, std/random, std/math, std/tables

type
  Cell = (int, int, int)
  Protein* = ref object
    aminoacids*: string
    angles*: seq[int]
    folding*: Table[Cell, char]
    energy*: int

proc move(angle: int, c: Cell): Cell =
  case angle
  of 0: (c[0]+1, c[1], c[2])
  of 1: (c[0]-1, c[1], c[2])
  of 2: (c[0], c[1]+1, c[2])
  of 3: (c[0], c[1]-1, c[2])
  of 4: (c[0], c[1], c[2]+1)
  of 5: (c[0], c[1], c[2]-1)
  else: c

proc computeFolding(p: Protein, angles: seq[int]): Table[Cell, char] =
  var c: Cell = (0, 0, 0)
  var k = 0
  result[c] = p.aminoacids[k]
  for angle in angles:
    inc k
    let nxt = move(angle, c)
    if nxt in result: return initTable[Cell, char]()
    result[nxt] = p.aminoacids[k]
    c = nxt

proc computeEnergy(p: Protein, folding: Table[Cell, char]): int =
  for c, aa in folding:
    if aa == 'H':
      for face in 0 ..< 6:
        if not (move(face, c) in folding):
          inc result

proc newProtein*(aminoacids: string): Protein =
  result = Protein(
    aminoacids: aminoacids,
    angles: newSeq[int](aminoacids.len - 1))
  result.folding = result.computeFolding(result.angles)
  result.energy = result.computeEnergy(result.folding)

proc fold*(p: Protein, t: float): int =
  var newAngles: seq[int]
```

```
  var newFolding: Table[Cell, char]
  while true:
    newAngles = p.angles
    let n = rand(1 .. p.aminoacids.len - 2)
    newAngles[n] = rand(0 .. 5)
    newFolding = p.computeFolding(newAngles)
    if newFolding.len > 0: break
  let newEnergy = p.computeEnergy(newFolding)
  if float(p.energy - newEnergy) > t * ln(rand(1.0)):
    p.angles = newAngles
    p.folding = newFolding
    p.energy = newEnergy
  p.energy

proc main() =
  randomize()
  var aas = ""
  for _ in 0 ..< 20:
    aas.add (if rand(1.0) < 0.5: 'H' else: 'P')
  let protein = newProtein(aas)
  var t = 10.0
  while t > 1e-5:
    discard protein.fold(t)
    echo protein.energy, " ", protein.angles
    t *= 0.99

main()
```

This is the well-known *HP lattice model*, introduced by Ken Dill in 1985 [53]: the protein is reduced to a self-avoiding walk on a (here cubic) lattice in which each residue is classified as hydrophobic (H) or polar (P), and the energy counts only hydrophobic–solvent contacts. Despite its drastic simplification, it captures the cooperative collapse driven by the hydrophobic effect and is a standard playground for studying folding algorithms.

The `move` procedure dispatches on the solid-angle code `0..5` and maps a cell `(x, y, z)` to the coordinates of the neighboring cube in that direction.

The annealing procedure is performed in the `main` function. The `fold` procedure is one Metropolis step. The purpose of the `while` loop inside `fold` is to draw a valid trial folding for the accept–reject step. Some trials are invalid because they are not physical: they would require two amino-acids to occupy the same lattice cell (a self-avoiding-walk violation). When this

happens, `computeFolding` returns an empty `Table`, and the loop tries again with a different randomly perturbed angle.

8

Appendices

8.1 Appendix A: Math Review and Notation

8.1.1 Symbols

∞	infinity
$\wedge$	and
$\vee$	or
$\cap$	intersection
$\cup$	union
$\in$	element or In
$\forall$	for each
$\exists$	exists
$\Rightarrow$	implies
:	such that
iff	if and only if

(8.1)

8.1.2 Set theory

Important sets

$\mathbf{0}$	empty set
$\mathbb{N}$	natural numbers {0, 1, 2, 3, ...}
$\mathbb{N}^+$	positive natural numbers {1, 2, 3, ...}
$\mathbb{Z}$	all integers {..., -3, -2, -1, 0, 1, 2, 3, ...}
$\mathbb{R}$	all real numbers
$\mathbb{R}^+$	positive real numbers (not including 0)
$\mathbb{R}^0$	positive numbers including 0

(8.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**

$$\mathcal{A} \cap \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{8.3}$$

- **Union**

$$\mathcal{A} \cup \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{8.4}$$

- **Difference**

$$\mathcal{A} - \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{8.5}$$

Set laws

- Empty set laws

$$\mathcal{A} \cup \mathbf{0} = \mathcal{A} \tag{8.6}$$
$$\mathcal{A} \cap \mathbf{0} = \mathbf{0} \tag{8.7}$$

- Idempotency laws

$$\mathcal{A} \cup \mathcal{A} = \mathcal{A} \tag{8.8}$$
$$\mathcal{A} \cap \mathcal{A} = \mathcal{A} \tag{8.9}$$

- Commutative laws

$$\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A} \tag{8.10}$$
$$\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A} \tag{8.11}$$

- Associative laws

$$\mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \quad (8.12)$$
$$\mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C} \quad (8.13)$$

- Distributive laws

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}) \quad (8.14)$$
$$\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \quad (8.15)$$

- Absorption laws

$$\mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) = \mathcal{A} \quad (8.16)$$
$$\mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) = \mathcal{A} \quad (8.17)$$

- DeMorgan laws

$$\mathcal{A} - (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C}) \quad (8.18)$$
$$\mathcal{A} - (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C}) \quad (8.19)$$

More set definitions

- $\mathcal{A}$ is a **subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$
- $\mathcal{A}$ is a **proper subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$ and $\exists x \in \mathcal{B}, x \notin \mathcal{A}$
- $P = \{S_i, i = 1, ..., N\}$ (a set of sets S_i) is a **partition** of $\mathcal{A}$ iff $S_1 \cup S_2 \cup ... \cup S_N = \mathcal{A}$ and $\forall i, j, S_i \cap S_j = \mathbf{0}$
- The number of elements in a set $\mathcal{A}$ is called the **cardinality** of set $\mathcal{A}$.
- cardinality($\mathbb{N}$)=countable infinite (∞)
- cardinality($\mathbb{R}$)=uncountable infinite (∞) !!!

Relations

- A **Cartesian Product** is defined as

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \quad (8.20)$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ is a subset of their Cartesian product.
- A binary relation is **transitive** if aRb and bRc implies aRc
- A binary relation is **symmetric** if aRb implies bRa
- A binary relation is **reflexive** if aRa if always true for each a.

Examples:

- $a < b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a > b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a = b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, symmetric and reflexive)
- $a \leq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- $a \geq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- A relation R that is transitive, symmetric and reflexive is called an **equivalence relation** and is often indicated with the notation $a \sim b$.

An equivalence relation is the same as a partition.

Functions

- A **function** between two sets $\mathcal{A}$ and $\mathcal{B}$ is a binary relation on $\mathcal{A} \times \mathcal{B}$ and is usually indicated with the notation $f : \mathcal{A} \longmapsto \mathcal{B}$
- The set $\mathcal{A}$ is called **domain** of the function.
- The set $\mathcal{B}$ is called **codomain** of the function.
- A function **maps** each element $x \in \mathcal{A}$ into an element $f(x) = y \in \mathcal{B}$
- The **image** of a function $f : \mathcal{A} \longmapsto \mathcal{B}$ is the set $\mathcal{B}' = \{y \in \mathcal{B} : \exists x \in \mathcal{A}, f(x) = y\} \subseteq \mathcal{B}$
- If $\mathcal{B}'$ is $\mathcal{B}$ then a function is said to be **surjective**.
- If for each x and x' in $\mathcal{A}$ where $x \neq x'$ implies that $f(x) \neq f(x')$ (e.g., if not two different elements of $\mathcal{A}$ are mapped into different element in $\mathcal{B}$) the function is said to be a **bijection**.

- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is **invertible** if it exists a function $g : \mathcal{B} \longmapsto \mathcal{A}$ such that for each $x \in \mathcal{A}, g(f(x)) = x$ and $y \in \mathcal{B}, f(g(y)) = y$. The function g is indicated with f^{-1}.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is a surjection and a bijection iff f is an invertible function.

Examples:

- $f(n) \equiv n \text{mod} 2$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection nor a bijection.
- $f(n) \equiv n \text{mod} 2$ with domain $\mathbb{N}$ and codomain $\{0, 1\}$ is a surjection but not a bijection
- $f(x) \equiv 2x$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection but is a bijection (in fact it is not invertible on odd numbers)
- $f(x) \equiv 2x$ with domain $\mathbb{R}$ and codomain $\mathbb{R}$ is not a surjection and is a bijection (in fact it is invertible)
-

8.1.3 Logarithms

If $x = a^y$ with $a > 0$, then $y = \log_a x$ with domain $x \in (0, \infty)$ and codomain $y = (-\infty, \infty)$. If the base a is not indicated, the natural log $a = e = 2.7183...$ is assumed.

Properties of logarithms:

$$\log_a x = \frac{\log x}{\log a} \tag{8.21}$$

$$\log xy = (\log x) + (\log y) \tag{8.22}$$

$$\log \frac{x}{y} = (\log x) - (\log y) \tag{8.23}$$

$$\log x^n = n \log x \tag{8.24}$$

8.1.4 Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \equiv f(0) + f(1) + ... + f(n-1) \tag{8.25}$$

Properties

- **Linearity I**

$$\sum_{i=0}^{i\leq n} f(i) = \sum_{i=0}^{i<n} f(i) + f(n) \tag{8.26}$$

$$\sum_{i=a}^{i\leq b} f(i) = \sum_{i=0}^{i\leq b} f(i) - \sum_{i=0}^{i<a} f(i) \tag{8.27}$$

- **Linearity II**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \tag{8.28}$$

Proof:

$$\begin{aligned}
\sum_{i=0}^{i<n} af(i) + bg(i) &= (af(0) + bg(0)) + ... + (af(n-1) + bg(n-1)) \\
&= af(0) + ... + af(n-1) + bg(0) + ... + bg(n-1) \\
&= a\left(f(0) + ... + f(n-1)\right) + b\left(g(0) + ... + g(n-1)\right) \\
&= a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right)
\end{aligned} \tag{8.29}$$

Examples:

$$\sum_{i=0}^{i<n} c = cn \text{ for any constant } c \tag{8.30}$$

$$\sum_{i=0}^{i<n} i = \frac{1}{2}n(n-1) \tag{8.31}$$

$$\sum_{i=0}^{i<n} i^2 = \frac{1}{6}n(n-1)(2n-1) \tag{8.32}$$

$$\sum_{i=0}^{i<n} i^3 = \frac{1}{4}n^2(n-1)^2 \tag{8.33}$$

$$\sum_{i=0}^{i<n} x^i = \frac{x^n-1}{x-1} \text{ (geometric sum)} \tag{8.34}$$

$$\sum_{i=0}^{i<n} \frac{1}{i(i+1)} = 1 - \frac{1}{n} \text{ (telescopic sum)} \tag{8.35}$$

8.1.5 Limits ($n \to \infty$)

In this section we will only deal with limits ($n \to \infty$) of positive functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = ? \tag{8.36}$$

First compute limits of the numerator and denominator separately:

$$\lim_{n\to\infty} f(n) = a \tag{8.37}$$

$$\lim_{n\to\infty} g(n) = b \tag{8.38}$$

- If $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{a}{b} \tag{8.39}$$

- If $a \in \mathbb{R}$ and $b = \infty$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{8.40}$$

- If ($a \in \mathbb{R}^+$ and $b = 0$) or ($a = \infty$ and $b \in \mathbb{R}$)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{8.41}$$

- If ($a = 0$ and $b = 0$) or ($a = \infty$ and $b = \infty$) use de l"Hopital rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{8.42}$$

  and start again!

- Else ... the limit does not exist (typically oscillating functions or non-analytic functions).

For any $a \in \mathbb{R}$ or $a = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \Rightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1/a \tag{8.43}$$

Table of derivatives

$f(x)$	$f'(x)$
c	0
ax^n	anx^{n-1}
$\log x$	$\frac{1}{x}$
e^x	e^x
a^x	$a^x \log a$
$x^n \log x, n > 0$	$x^{n-1}(n \log x + 1)$

(8.44)

Practical rules to compute derivatives

$$\frac{d}{dx}\left(f(x)+g(x)\right) = f'(x)+g'(x) \tag{8.45}$$

$$\frac{d}{dx}\left(f(x)-g(x)\right) = f'(x)-g'(x) \tag{8.46}$$

$$\frac{d}{dx}\left(f(x)g(x)\right) = f'(x)g(x)+f(x)g'(x) \tag{8.47}$$

$$\frac{d}{dx}\left(\frac{1}{f(x)}\right) = -\frac{f'(x)}{f(x)^2} \tag{8.48}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)}{g(x)}-\frac{f(x)g'(x)}{g(x)^2} \tag{8.49}$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x) \tag{8.50}$$

Index

Bibliography

[1] Andreas Rumpf and contributors, "Nim Programming Language". https://nim-lang.org

[2] http://www.sqlite.org/

[3] Donald Knuth, "The Art of Computer Programming, Volume 3", Addison-Wesley, (1997). ISBN 0-201-89685-0

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill (2001). ISBN 0-262-03293-7

[5] J.W.J. Williams, J. W. J. "Algorithm 232 - Heapsort", Communications of the ACM 7 (6) (1964) pp 347–348

[6] E. F. Moore, "The shortest path through a maze", in Proceedings of the International Symposium on the Theory of Switching, Harvard University Press (1959) pp 285–292

[7] Charles Pierre Trémaux (1859–1882) École Polytechnique of Paris (1876). re-published in the Annals academic, March 2011 – ISSN: 0980-6032

[8] Joseph Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", in Proceedings of the American Mathematical Society, Vol.7, N.1 (1956) pp 48–50

[9] R. C. Prim, "Shortest connection networks and some generalizations" in Bell System Technical Journal, 36 (1957) pp 1389–1401

[10] M. Farach-Colton *et al.*, "Mathematical Support for Molecular Biology", DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (1999) Volume 47. ISBN:0-8218-0826-5

[11] B. Korber *et al.*, "Timing the Ancestor of the HIV-1 Pandemic Strains", Science (9 Jun 2000) Vol.288 no.5472.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269–271 (1959). DOI:10.1007/BF01386390

[13] C. E. Shannon, "A Mathematical Theory of Communication". Bell System Technical Journal 27 (1948) pp 379–423

[14] R. M. Fano, "The transmission of information", Technical Report No. 65 MIT (1949)

[15] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., (1952) pp 1098–1102

[16] Bergroth and H. Hakonen and T. Raita, "A Survey of Longest Common Subsequence Algorithms". SPIRE (IEEE Computer Society) 39–48 (2000). DOI:10.1109/SPIRE.2000.878178. ISBN:0-7695-0746-8

[17] Saul Needleman and Christian Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3) (1970) pp 443–53. DOI:10.1016/0022-2836(70)90057-4

[18] Tobias Dantzig, "Numbers: The Language of Science", 1930.

[19] V. Estivill-Castro, "Why so many clustering algorithms", ACM SIGKDD Explorations Newsletter 4 (2002) pp 65. DOI:10.1145/568574.568575

[20] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", Proc. Natl. Acad. Sci. USA Vol. 79 (1982) pp 2554-2558

[21] Nils Aall Barricelli, Nils Aall, "Symbiogenetic evolution processes realized by artificial methods", Methodos (1957) pp 143–182

[22] Michael Garey and David Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", San Francisco: W. H. Freeman and Company (1979). ISBN:0-7167-1045-5

[23] Douglas R. Hofstadter, "Gödel, Escher, Bach: An Eternal Golden Braid", Basic Books (1979). ISBN:0-465-02656-7

[24] Harry Markowitz, "Foundations of portfolio theory", The Journal of Finance 46.2 (2012) pp 469-477

[25] P. E. Greenwood and M. S. Nikulin, "A guide to chi-squared testing". Wiley, New York (1996). ISBN:0-471-55779-X

[26] Andrew Lo and Jasmina Hasanhodzic, "The Evolution of Technical Analysis: Financial Prediction from Babylonian Tablets to Bloomberg Terminals", Bloomberg Press (2010). ISBN:1576603490

[27] Y. Saad and M.H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", SIAM J. Sci. Stat. Comput. 7 (1986). DOI:10.1137/0907058

[28] H. A. Van der Vorst, "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems". SIAM J. Sci. and Stat. Comput. 13 (2) (1992) pp 631–644. DOI:10.1137/0913035

[29] Richard Burden and Douglas Faires, "2.1 The Bisection Algorithm", Numerical Analysis (3rd ed.), PWS Publishers (1985). ISBN:0-87150-857-5

[30] Michiel Hazewinkel, "Newton method", Encyclopedia of Mathematics, Springer (2001). ISBN:978-1-55608-010-4

[31] Mordecai Avriel and Douglas Wilde, "Optimality proof for the symmetric Fibonacci search technique", Fibonacci Quarterly 4 (1966) pp 265–269 MR:0208812

[32] Loukas Grafakos, "Classical and Modern Fourier Analysis", Prentice-Hall (2004). ISBN:0-13-035399-X

[33] S.D. Poisson, "Probabilité des jugements en matière criminelle et en matière civile, précédées des règles générales du calcul des probabilitiés", Bachelier (1837)

[34] A. W. Van der Vaart, "Asymptotic statistics", Cambridge University Press (1998). ISBN:978-0-521-49603-2

[35] Jonah Lehrer, "How We Decide", Houghton Mifflin Harcourt (2009). ISBN:978-0-618-62011-1

[36] Edward N. Lorenz, "Deterministic non-periodic flow". Journal of the Atmospheric Sciences 20 (2) (1963) pp 130–141. DOI:10.1175/1520-0469

[37] Ian Hacking, "19th-century Cracks in the Concept of Determinism", Journal of the History of Ideas, 44 (3) (1983) pp 455-475 JSTOR:2709176

[38] F. Cannizzaro, G. Greco, S. Rizzo, E. Sinagra, "Results of the measurements carried out in order to verify the validity of the poisson-exponential distribution in radioactive decay events". The International Journal of Applied Radiation and Isotopes 29 (11) (1978) pp 649. DOI:10.1016/0020-708X(78)90101-1

[39] Yuval Perez, "Iterating Von Neumann's Procedure for Extracting Random Bits". The Annals of Statistics 20 (1) (1992) pp 590–597

[40] Martin Luescher, "A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations", Comput. Phys. Commun. 79 (1994) pp 100-110

[41] `http://demonstrations.wolfram.com/PoorStatisticalQualitiesForTheRANDURandomNumberGenerator`

[42] G. S. Fishman, "Grouping observations in digital simulation", Management Science 24 (1978) pp 510-521

[43] P. Good, "Introduction to Statistics Through Resampling Methods and R/S-PLUS", Wiley (2005). ISBN:0-471-71575-1

[44] J. Shao and D. Tu, "The Jackknife and Bootstrap", Springer-Verlag (1995)

[45] Paul Wilmott, "Paul Wilmott Introduces Quantitative Finance", Wiley 92005). ISBN:978-0-470-31958-1

[46] http://www.fas.org/sgp/othergov/doe/lanl/lib-www/la-pubs/00326407.pdf

[47] S. M. Ross, "Stochastic Processes", Wiley (1995). ISBN:978-0-471-12062-9

[48] Révész Pal, "Random walk in random and non random environments", World Scientific (1990)

[49] A.A. Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". reprinted in Appendix B of R. Howard. "Dynamic Probabilistic Systems", Vol.1, John Wiley and Sons (1971)

[50] W. Vervaat, "A relation between Brownian bridge and Brownian excursion". Ann. Prob. 7 (1) (1979) pp 143–149 JSTOR:2242845

[51] Kiyoshi Ito, "On stochastic differential equations", Memoirs, American Mathematical Society 4, 1–51 (1951)

[52] Steven Shreve, "Stochastic Calculus for Finance II: Continuous Time Models", Springer (2008) pp 114. ISBN:978-0-387-40101-0

[53] Sorin Istrail and Fumei Lam, "Combinatorial Algorithms for Protein Folding in Lattice Models: A Survey of Mathematical Results" (2009)