MASSIMO DI PIERRO

COMPUTATIONS IN PYTHON

EXPERTS4SOLUTIONS

Massimo Di Pierro
School of Computing
DePaul University
243 S Wabash Ave
Chicago, IL 60604 (USA)
Email: massimo.dipierro@gmail.com

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-0000 Build Date: April 3, 2012

to my family

Contents

Preface

1

Introduction

Scientific Computing is "the study of Algorithms for the problems of *continuous mathematics* (as distinguished from *discrete mathematics*)" (from Wikipedia).

Scientific Computing can also be defined as "The study of approximation techniques for solving mathematical problems, taking into account the extent of possible errors" (from Answers.com)

These two definitions are equivalent because computers have a finite memory and can only represent continuous quantities via approximations.

Applications of Scientific Computing include physics, biology, finance, etc.

Most algorithms for Scientific Computing can be grouped in the following categories:

- Linear Algebra
- Solvers of non-linear equations
- Optimization (minimization and maximization)
- Fitting
- Numerical Integration and Differentiation
- Differential Equations

- Fourier and Laplace transform
- Statistical and Stochastic methods

We will deal with most of them in this chapter but we will deal with Statistical Stochastic methods in the next chapter.

Acknowledgements

Michael Gheith, Ethan Sudman

2

Overview of the Python Language

2.1 About Python

Python is a general-purpose high-level programming language. Its design philosophy emphasizes programmer productivity and code readability. It has a minimalist core syntax with very few basic commands and simple semantics. It also has a large and comprehensive standard library, including an Application Programming Interface (API) to many of the underlying operating system (OS) functions. Python provides built-in objects such as linked lists (`list`), tuples (`tuple`), hash tables (`dict`), arbitrarily long integers (`long`), complex numbers, and arbitrary precision decimal numbers.

Python supports multiple programming paradigms, including object-oriented (`class`), imperative (`def`), and functional (`lambda`) programming. Python has a dynamic type system and automatic memory management using reference counting (similar to Perl, Ruby, and Scheme).

Python was first released by Guido van Rossum in 1991. The language has an open, community-based development model managed by the non-profit Python Software Foundation. There are many interpreters and compilers that implement the Python language, including one in Java (Jython), one built on .Net (IronPython) and one built in Python itself (PyPy). In this brief review, we refer to the reference C implementation created by Guido.

You can find many tutorials, the official documentation, and library references of the language on the official Python website. [1]

For additional Python references, we can recommend the books in ref. [7] and ref. [8].

You may skip this chapter if you are already familiar with the Python language.

2.1.1 *Python vs Java and C++ syntax*

equation

	Java/C++	**Python**
assignment	$a = b$;	$a = b$
comparison	if $(a == b)$	if $a == b$:
loops	for$(a = 0; a < n; a++)$	for a in $range(0, n)$:
block	indentation	indentation
function	function $float f$(float a) {	def $f(a)$:
function call	$f(a)$	$f(a)$
arrays/lists	$a[i]$	$a[i]$
member	a.member	a.member
nothing	$null$ / $void*$	$None$

equation

As in Java, variables which are primitive types (bool, int, float) are passed by copy but more complex types, unlike C++, are passed by reference.

2.1.2 *help, dir*

The Python language provides two commands to obtain documentation about objects defined in the current scope, whether the object is built-in or user-defined.

We can ask for `help` about an object, for example "1":

```
>>> help(1)
Help on int object:

class int(object)
```

```
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base.  It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
...
```

and, since "1" is an integer, we get a description about the `int` class and all its methods. Here the output has been truncated because it is very long and detailed.

Similarly, we can obtain a list of methods of the object "1" with the command `dir`:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
'__floordiv__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__',
'__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__',
'__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__',
'__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__str__', '__sub__', '__truediv__', '__xor__']
```

2.2 *Python Types*

Python is a dynamically typed language, meaning that variables do not have a type and therefore do not have to be declared. Values, on the other hand, do have a type. You can query a variable for the type of value it contains:

```
>>> a = 3
>>> print(type(a))
<type 'int'>
>>> a = 3.14
```

```
>>> print(type(a))
<type 'float'>
>>> a = 'hello python'
>>> print(type(a))
<type 'str'>
```

Python also includes, natively, data structures such as lists and dictionaries.

2.2.1 `int` *and* `long`

There are two types representing integer numbers, they are `int` and `long`. The difference is that `int` corresponds to the microprocessor 32 bites integer representation and can only hold signed integers in range $[-2^{31}, +2^{31}]$ while the `long` type can hold arbitrary integers. It is important that Python automatically converts one into the other as necessary and you can mix and match the two types in computations. Here is an example:

```
>>> a = 1024
>>> type(a)
<type 'int'>
>>> b = a**128
>>> print(b)
2081586438932879816385048065472817107723052449453340961063822470080721611934672059602447888346464836968484322790856201558276713249664692981627981321135464152584825901877844069154636669932316710094591884109537962242338735429509695773392500276887652058346469777062232165707683317005651120933244966378183760369413644440628104205339687097746591605775610173947237380142944142111140633745817
>>> print(type(b))
<type 'long'>
```

Computers represent 32 bit integer numbers by converting them to base 2. The conversion works in the following way:

```
def int2binary(n, nbits=32):
    if n<0:
        return [1 if bit==0 else 0 for bit in int2binary(-n-1,nbits)]
    bits = [0]*nbits
    for i in range(nbits):
        n, bits[i] = divmod(n,2)
    if n: raise OverflowError
    return bits
```

The case $n < 2$ is called *two's complement* and is defined as the value obtained by subtracting the number from the largest power of two (2^{32} for 32 bits).

Just by looking at the most significative bit one can determine the sign of the binary number (1 for negative and 0 for zero or positive).

2.2.2 `float` *and* `decimal`

There are two ways to represent decimal numbers in Python. Using the native double precision (64bits) representation, `float`, or using the `decimal` module.

Most numerical problems are dealt with simply using `float`:

```
>>> pi = 3.141592653589793
>>> two_pi = 2.0 * pi
```

The `cmath` module, described later, contains many math functions that you can use with the `float` type.

Floating point numbers are internally represented as follows:

$$x = \pm m 2^e \tag{2.1}$$

where x is the number, m is called mantissa and it is zero or a number in range [1,2), e is called exponent. The sign, m and e can be computed using the following algorithm which also writes their representation in binary:

```
def float2binary(x,nm=52,ne=11):
    if x==0:
        return 0, [0]*nm, [0]*ne
    sign,mantissa, exponent = (1 if x<0 else 0),abs(x),0
    while abs(mantissa)>=2:
        mantissa,exponent = 0.5*mantissa,exponent+1
    while 0<abs(mantissa)<1:
        mantissa,exponent = 2.0*mantissa,exponent-1
    return sign,int2binary(int(2**nm*(mantissa-1)),nm),int2binary(exponent,ne)
```

Because the mantissa is stored in a fixed number of bits (11 for a 64 bits floating point number) then exponents smaller than -1022 and larger than 1023 cannot be represented. An arithmetic operation that returns a number smaller than $2^{-1022} \simeq 10^{-308}$ cannot be represented and results in an Underflow error. An operation that returns a number larger than $2^{1023} \simeq 10^{308}$ also cannot be represented and results in an Overflow error.

Here is an example of Overflow:

```
>>> a = 10.0**200
>>> a*a
inf
```

And here is an example of Underflow:

```
>>> a = 10.0**-200
>>> a*a
0.0
```

Another problem with finite precision arithmetic is the loss of precision for computation. Consider the case of the difference between two numbers with very different orders of magnitude. In order to compute the difference the CPU reduces them to the same exponent (the largest of the two) and then computes the difference in the two mantissas. If two numbers differ for a factor 2^k than the mantissa of the smallest number, in binary, needs to be shifted by k position thus resulting in a loss of information because the k least significant bits in the mantissa are ignored. If the difference between the two numbers is greater than a factor 2^{52} all bits in the mantissa of the smallest number are ignored and the smallest number becomes completely invisible.

Below is a practical example that produces a wrong result:

```
>>> a = 1.0
>>> b = 2.0**53
>>> a+b-b
0.0
```

This precision issue is always present but not always obvious. It may consist of a small discrepancy between the true value and the computed value. This difference may increase during the computation, in particular in iterative algorithms, and may be sizable in the result of a complex algorithm.

Python also has a module for decimal floating point arithmetic which allows decimal numbers to be represented exactly. The class Decimal incorporates a notion of significant places (unlike hardware based binary floating point, the decimal module has a user alterable precision):

```
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 28 # set precision
```

```
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
```

Decimal numbers can be used almost everywhere in place of floating point number arithmetics but they are slower and should be used only where arbitrary precision arithmetics is required. It does not suffer from the overflow, underflow, and precision issues described above.

```
>>> from decimal import Decimal
>>> a = Decimal(10.0)**300
>>> a*a
Decimal('1.00000000000000000000000000E+600')
```

2.2.3 complex

Python has native support for complex numbers. The imaginary unit is represented by the character `j`.

```
>>> c = 1+2j
>>> print(c)
(1+2j)
>>> print(c.real)
1.0
>>> print(c.imag)
2.0
>>> print(abs(c))
2.2360679775
```

The real and imaginary parts of a complex number are stored as 64 bits floating point numbers.

Normal arithmetic operations are supported. The `cmath` module contains trigonometric and other functions for complex numbers.

2.2.4 str

Python supports the use of two different types of strings: ASCII strings and Unicode strings. ASCII strings are delimited by '...', "..." or by '..' or """...""". Triple quotes delimit multiline strings. Unicode strings start with a `u` followed by the string containing Unicode characters. A Unicode string

can be converted into an ASCII string by choosing an encoding (for example UTF8):

```
>>> a = 'this is an ASCII string'
>>> b = u'This is a Unicode string'
>>> a = b.encode('utf8')
```

After executing these three commands, the resulting `a` is an ASCII string storing UTF8 encoded characters.

It is also possible to write variables into strings in various ways:

```
>>> print('number is ' + str(3))
number is 3
>>> print('number is %s' % (3))
number is 3
>>> print('number is %(number)s' % dict(number=3))
number is 3
```

The final notation is more explicit and less error prone, and is to be preferred.

Many Python objects, for example numbers, can be serialized into strings using `str` or `repr`. These two commands are very similar but produce slightly different output. For example:

```
>>> for i in [3, 'hello']:
...     print(str(i), repr(i))
3 3
hello 'hello'
```

For user-defined classes, `str` and `repr` can be defined/redefined using the special operators `__str__` and `__repr__`. These are briefly described later in this chapter. For more information on the topic, refer to the official Python documentation [9].

Another important characteristic of a Python string is that it is an iterable object, similar to a list:

```
>>> for i in 'hello':
...     print(i)
h
e
l
l
o
```

2.2.5 `list` *and* `array`

The main methods of a Python list are append, insert, and delete:

```
>>> a = [1, 2, 3]
>>> print(type(a))
<type 'list'>
>>> a.append(8)
>>> a.insert(2, 7)
>>> del a[0]
>>> print(a)
[2, 7, 3, 8]
>>> print(len(a))
4
```

Lists can be sliced:

```
>>> a= [2, 7, 3, 8]
>>> print(a[:3])
[2, 7, 3]
>>> print(a[1:])
[7, 3, 8]
>>> print(a[-2:])
[3, 8]
```

and concatenated:

```
>>> a = [2, 7, 3, 8]
>>> a = [2, 3]
>>> b = [5, 6]
>>> print(a + b)
[2, 3, 5, 6]
```

A list is iterable; you can loop over it:

```
>>> a = [1, 2, 3]
>>> for i in a:
...     print(i)
1
2
3
```

The elements of a list do not have to be of the same type; they can be any type of Python object.

There is a very common situation for which a *list comprehension* can be used. Consider the following code:

```
>>> a = [1,2,3,4,5]
>>> b = []
>>> for x in a:
...     if x % 2 == 0:
...         b.append(x * 3)
>>> b
[6, 12]
```

This code clearly processes a list of items, selects and modifies a subset of the input list, and creates a new result list. This code can be entirely replaced with the following list comprehension:

```
>>> a = [1,2,3,4,5]
>>> b = [x * 3 for x in a if x % 2 == 0]
>>> b
[6, 12]
```

Python has a module called `array` which defines an array object that provides an efficient array implementation:

```
>>> from array import array
>>> a = array('d',[1,2,3,4,5])
array('d',[1.0, 2.0, 3.0, 4.0, 5.0])
```

An array object can be used in the same way as a list but its elements must all be of the same type, specified by the first argument of the constructor ("d" for double, "l" for signed long, "f" for float, and "c" for character). When not iterating over them, accessing array elements is much faster than accessing list elements because in a list they are stored close together in a predictable position in memory. Accessing `a[i]` for a list requires looping and locating the *i*th element while for an array the computer can jump to the requested element without looping or searching.

2.2.6 `tuple`

A tuple is similar to a list, but its size and elements are immutable. If a tuple element is an object, the object attributes are mutable. A tuple is defined by elements separated by a comma and optionally delimited by round brackets:

```
>>> a = 1, 2, 3
>>> a = (1, 2, 3)
```

The round brakets are required for a tuple of zero elements such as

```
>>> a = () # this is an empty tuple
```

A comma is required even for a tuple of one element:

```
>>> a = (1)  # not a tuple
>>> a = (1,) # this is a tuple of one element
```

So while this works for a list:

```
>>> a = [1, 2, 3]
>>> a[1] = 5
>>> print(a)
[1, 5, 3]
```

the element assignment does not work for a tuple:

```
>>> a = (1, 2, 3)
>>> print(a[1])
2
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

A tuple, like a list, is an iterable object. Notice that a tuple consisting of a single element must include a trailing comma, as shown below:

```
>>> a = (1)
>>> print(type(a))
<type 'int'>
>>> a = (1,)
>>> print(type(a))
<type 'tuple'>
```

Tuples are very useful for efficient packing of objects because of their immutability. The brackets are often optional:

```
>>> a = 2, 3, 'hello'
>>> x, y, z = a
>>> print(x)
2
>>> print(z)
hello
```

2.2.7 dict

A Python dict-ionary is a hash table that maps a key object to a value object. For example:

```
>>> a = {'k':'v', 'k2':3}
>>> print(a['k'])
v
>>> print(a['k2'])
3
>>> 'k' in a
True
>>> 'v' in a
False
```

Keys can be of any hashable type (int, string, or any object whose class implements the __hash__ method). Values can be of any type. Different keys and values in the same dictionary do not have to be of the same type. If the keys are alphanumeric characters, a dictionary can also be declared with the alternative syntax:

```
>>> a = dict(k='v', h2=3)
>>> print(a['k'])
v
>>> print(a)
{'h2': 3, 'k': 'v'}
```

Useful methods are has_key, keys, values and items:

```
>>> a = dict(k='v', k2=3)
>>> print(a.keys())
['k2', 'k']
>>> print(a.values())
['v', 3]
>>> print(a.items())
[('k', 'v'), ('k2', 3)]
```

The items method produces a list of tuples, each containing a key and its associated value.

Dictionary elements and list elements can be deleted with the command del:

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> print(a)
[1, 3]
```

```
>>> a = dict(k='v', h2=3)
>>> del a['h2']
>>> print(a)
{'k': 'v'}
```

Internally, Python uses the `hash` operator to convert objects into integers, and uses that integer to determine where to store the value.

```
>>> hash("hello world")
-1500746465
```

2.2.8 set

A set is something in between a list and dictionary. It represents a non-ordered list of elements that cannot be repeated. Internally it is implemented as a hash-table, similar to a set of keys in a dictionary. A set is created using the `set` constructor. Its argument can be a list, a tuple, or an iterator:

```
>>> s = set([1,2,3,4,5])
>>> s = set((1,2,3,4,5))
>>> s = set(i for i in range(1,6))
>>> print(s)
set([1, 2, 3, 4, 5])
```

Instead of `append`, we can add elements to a set using the `add` method:

```
>>> s = set()
>>> s.add(2)
>>> s.add(3)
>>> s.add(2)
>>> print(s)
set([2, 3])
```

Notice that we cannot add the same element twice (2 in the example). The `set` object supports normal set operations like union, intersection, and difference:

```
>>> a = set([1,2,3])
>>> b = set([2,3,4])
>>> print(a.union(b))
set([1, 2, 3, 4])
>>> print(a.intersection(b))
set([2, 3])
>>> print(a.difference(b))
set([1])
```

to check for membership:

```
>>> 2 in a
True
```

2.3 *Python control flow statements*

Python uses indentation to delimit blocks of code. A block starts with a line ending with colon, and continues for all lines that have a similar or higher indentation as the next line. For example:

```
>>> i = 0
>>> while i < 3:
...     print(i)
...     i = i + 1
0
1
2
```

It is common to use four spaces for each level of indentation. It is a good policy not to mix tabs with spaces, which can result in (invisible) confusion.

2.3.1 `for...in`

In Python, you can loop over iterable objects:

```
>>> a = [0, 1, 'hello', 'python']
>>> for i in a:
...     print(i)
0
1
hello
python
```

A very important keyword is `xrange`. It generates an iterable range instead of generating the entire list of elements.

```
>>> for i in xrange(0, 4):
...     print(i)
0
1
2
3
```

This is equivalent to the C/C++/C#/Java syntax:

```
for(int i=0; i<4; i=i+1) { print(i); })
```

Another useful command is `enumerate`, which counts while looping:

```
>>> a = [0, 1, 'hello', 'python']
>>> for i, j in enumerate(a):
...     print(i, j)
0 0
1 1
2 hello
3 python
```

There is also a keyword `range(a, b, c)` that returns a list of integers starting with the value `a`, incrementing by `c`, and ending with the last value smaller than `b`, `a` defaults to 0 and `c` defaults to 1. `xrange` is similar but does not actually generate the list, only an iterator over the list; thus it is better for looping.

You can jump out of a loop using `break`

```
>>> for i in [1, 2, 3]:
...      print(i)
...      break
1
```

You can jump to the next loop iteration without executing the entire code block with `continue`

```
>>> for i in [1, 2, 3]:
...      print(i)
...      continue
...      print('test')
1
2
3
```

Python also supports list comprehensions and you can build lists using using the following syntax:

```
>>> a = [i*i for i in [0, 1, 2, 3]:
>>> print(a)
[0, 1, 4, 9]
```

Sometimes you may need a counter to "count" the elements of a list while looping:

```
>>> a = [e*(i+1) for (i,e) in ['a','b','c','d']]
>>> print(a)
['a', 'bb', 'ccc', 'dddd']
```

2.3.2 while

The `while` loop in Python works much as it does in many other programming languages, by looping an indefinite number of times and testing a condition before each iteration. If the condition is `False`, the loop ends.

```
>>> i = 0
>>> while i < 10:
...     i = i + 1
>>> print(i)
10
```

There is no `loop...until` construct in Python.

2.3.3 if...elif...else

The use of conditionals in Python is intuitive:

```
>>> for i in range(3):
...     if i == 0:
...         print('zero')
...     elif i == 1:
...         print('one')
...     else:
...         print('other')
zero
one
other
```

`elif` means "else if". Both `elif` and `else` clauses are optional. There can be more than one `elif` but only one `else` statement. Complex conditions can be created using the `not`, `and` and `or` logical operators.

```
>>> for i in range(3):
...     if i == 0 or (i == 1 and i + 1 == 2):
...         print('0 or 1')
```

2.3.4 `try...except...else...finally`

Python can throw - pardon, raise - Exceptions:

```
>>> try:
...     a = 1 / 0
... except Exception, e:
...     print('oops: %s' % e)
... else:
...     print('no problem here')
... finally:
...     print('done')
oops: integer division or modulo by zero
done
```

If the exception is raised, it is caught by the `except` clause, and while the `else` clause is not. If an exception is raised, the `except` clause is executed, otherwise the `else` one is executed. The `finally` clause is always executed.

There can be multiple `except` clauses for different possible exceptions:

```
>>> try:
...     raise SyntaxError
... except ValueError:
...     print('value error')
... except SyntaxError:
...     print('syntax error')
syntax error
```

The `else` and `finally` clauses are optional.

Here is a list of built-in Python exceptions

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- GeneratorExit
      +-- StopIteration
      +-- StandardError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
```

```
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |    |    +-- UnicodeError
      |    |         +-- UnicodeDecodeError
      |    |         +-- UnicodeEncodeError
      |    |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
```

For a detailed description of each of them, refer to the official Python documentation.

Any object can be raised as an exception, but it is good practice to raise objects that extend one of the built-in exception classes.

2.3.5 `def...return`

Functions are declared using `def`. Here is a typical Python function:

```
>>> def f(a, b):
...     return a + b
```

```
>>> print(f(4, 2))
6
```

There is no need (or way) to specify argument or the return type(s). In this example, a function `f` is defined that can take two arguments.

Functions are the first code syntax feature described in this chapter to introduce the concept of *scope*, or *namespace*. In the above example, the identifiers `a` and `b` are undefined outside of the scope of function `f`:

```
>>> def f(a):
...     return a + 1
>>> print(f(1))
2
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
```

Identifiers defined outside of function scope are accessible within the function; observe how the identifier `a` is handled in the following code:

```
>>> a = 1
>>> def f(b):
...     return a + b
>>> print(f(1))
2
>>> a = 2
>>> print(f(1) # new value of a is used)
3
>>> a = 1 # reset a
>>> def g(b):
...     a = 2 # creates a new local a
...     return a + b
>>> print(g(2))
4
>>> print(a # global a is unchanged)
1
```

If `a` is modified, subsequent function calls will use the new value of the global `a` because the function definition binds the storage location of the identifier `a`, not the value of `a` itself at the time of function declaration; however, if `a` is assigned-to inside function `g`, the global `a` is unaffected because the new local `a` hides the global value. The external-scope reference can be used in

the creation of *closures*:

```
>>> def f(x):
...     def g(y):
...         return x * y
...     return g
>>> doubler = f(2) # doubler is a new function
>>> tripler = f(3) # tripler is a new function
>>> quadrupler = f(4) # quadrupler is a new function
>>> print(doubler(5))
10
>>> print(tripler(5))
15
>>> print(quadrupler(5))
20
```

Function `f` creates new functions; and note that the scope of the name `g` is entirely internal to `f`. Closures are extremely powerful.

Function arguments can have default values, and can return multiple results:

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(5)
>>> print(x)
7
>>> print(y)
3
```

Function arguments can be passed explicitly by name, and this means that the order of arguments specified in the caller can be different than the order of arguments with which the function was defined:

```
>>> def f(a, b=2):
...     return a + b, a - b
>>> x, y = f(b=5, a=2)
>>> print(x)
7
>>> print(y)
-3
```

Functions can also take a runtime-variable number of arguments:

```
>>> def f(*a, **b):
...     return a, b
>>> x, y = f(3, 'hello', c=4, test='world')
>>> print(x)
(3, 'hello')
```

```
>>> print(y)
{'test': 'world', 'c': 4}
```

Here arguments not passed by name (3, 'hello') are stored in the tuple `a`, and arguments passed by name (`c` and `test`) are stored in the dictionary `b`.

In the opposite case, a list or tuple can be passed to a function that requires individual positional arguments by unpacking them:

```
>>> def f(a, b):
...     return a + b
>>> c = (1, 2)
>>> print(f(*c))
3
```

and a dictionary can be unpacked to deliver keyword arguments:

```
>>> def f(a, b):
...     return a + b
>>> c = {'a':1, 'b':2}
>>> print(f(**c))
3
```

2.3.6 `lambda`

`lambda` provides a way to define a short unnamed function:

```
>>> a = lambda b: b + 2
>>> print(a(3))
5
```

The expression "`lambda` [a]:[b]" literally reads as "a function with arguments [a] that returns [b]". The `lambda` expression is itself unnamed, but the function acquires a name by being assigned to identifier `a`. The scoping rules for `def` apply to `lambda` equally, and in fact the code above, with respect to `a`, is identical to the function declaration using `def`:

```
>>> def a(b):
...     return b + 2
>>> print(a(3))
5
```

The only benefit of `lambda` is brevity; however, brevity can be very convenient in certain situations. Consider a function called `map` that applies a function to all items in a list, creating a new list:

```
>>> a = [1, 7, 2, 5, 4, 8]
>>> map(lambda x: x + 2, a)
[3, 9, 4, 7, 6, 10]
```

This code would have doubled in size had `def` been used instead of `lambda`. The main drawback of `lambda` is that (in the Python implementation) the syntax allows only for a single expression; however, for longer functions, `def` can be used and the extra cost of providing a function name decreases as the length of the function grows. Just like `def`, `lambda` can be used to *curry* functions: new functions can be created by wrapping existing functions such that the new function carries a different set of arguments:

```
>>> def f(a, b): return a + b
>>> g = lambda a: f(a, 3)
>>> g(2)
5
```

Python functions, created with either `def` *or* `lambda` *allow re-factoring of existing functions in terms of a different set of arguments.*

2.4 Classes

Because Python is dynamically typed, Python classes and objects may seem odd. In fact, you do not need to define the member variables (attributes) when declaring a class, and different instances of the same class can have different attributes. Attributes are generally associated with the instance, not the class (except when declared as "class attributes", which is the same as "static member variables" in C++/Java).

Here is an example:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.myvariable = 3
>>> print(myinstance.myvariable)
3
```

Notice that `pass` is a do-nothing command. In this case it is used to define a class `MyClass` that contains nothing. `MyClass()` calls the constructor of the class (in this case the default constructor) and returns an object, an instance of the class. The `(object)` in the class definition indicates that our class extends the built-in `object` class. This is not required, but it is good practice.

Here is a more complex class:

```
>>> class Complex(object):
...     z = 2
...     def __init__(self, real=0.0, imag=0.0):
...         self.real, self.imag = real, imag
...     def magnitude(self):
...         return (self.real**2 + self.imag**2)**0.5
...     def __add__(self,other):
...         return Complex(self.real+other.real,self.imag+other.imag)
>>> a = Complex(1,3)
>>> b = Complex(2,1)
>>> c = a + b
>>> print(c.magnitude())
5
```

Functions declared inside the class are methods. Some methods have special reserved names. For example, `__init__` is the constructor. In the example we created a class to store the `real` and the `imag` part of a complex number. The constructor takes these two variables and stores them into `self` (not a keyword but a variable that plays the same role as `this` in Java and `(*this)` in C++. (This syntax is necessary to avoid ambiguity when declaring nested classes, such as a class that is local to a method inside another class, something the Python allows but Java and C++ do not).

The `self` variable is defined by the first argument of each method. They all must have it but they can use another variable name. Even if we use another name, the first argument of a method always refers to the object calling the method. It plays the same role as the `this` keyword in the Java and the C++ languages.

`__add__` is also a special method (all special methods start and end in double underscore) and it overloads the `+` operator between `self` and `other`. In the example, `a+b` is equivalent to a call to `a.__add__(b)` and the `__add__` method receives `self=a` and `other=b`.

All variables are local variables of the method except variables declared outside methods which are called *class variables*, equivalent to C++ *static member variables* that hold the same value for all instances of the class.

2.4.1 *Special methods and operator overloading*

Class attributes, methods, and operators starting with a double underscore are usually intended to be private (i.e. to be used internally but not exposed outside the class) although this is a convention that is not enforced by the interpreter.

Some of them are reserved keywords and have a special meaning.

Here, as an example, are three of them:

- `__len__`
- `__getitem__`
- `__setitem__`

They can be used, for example, to create a container object that acts like a list:

```
>>> class MyList(object):
>>>     def __init__(self, *a): self.a = list(a)
>>>     def __len__(self): return len(self.a)
>>>     def __getitem__(self, key): return self.a[key]
>>>     def __setitem__(self, key, value): self.a[key] = value
>>> b = MyList(3, 4, 5)
>>> print(b[1])
4
>>> b.a[1] = 7
>>> print(b.a)
[3, 7, 5]
```

Other special operators include `__getattr__` and `__setattr__`, which define the get and set attributes for the class, and `__add__`, `__sub__`, `__mul__`, `__div__` which overload arithmetic operators. For the use of these operators we refer the reader to the chapter on linear algebra where they will be used to implement algebra for matrices.

2.4.2 *Financial Transaction*

As one more example of a class, we will implement a class that represents a financial transaction. We can think of a simple transaction as a single money transfer of quantity `a` that occurs at a given time t. We adopt the convention that a positive `amount` represents money flowing in and a negative value represents money flowing out.

The Net Present Value (computed at time t_0) for a transaction occurring at time t days from now of amount a is defined as

$$\text{NPV}(d, a) = ae^{(t-d)r} \tag{2.2}$$

where r is the daily risk free interest rate. If t and d are measured in days, r has to be the daily risk free return. Here we will assume it defaults to $r = 005/365$ (5% annually).

Here is a possible implementation of the Transaction as a class and CashFlow as a class.

```
from datetime import date
from math import exp
today = date.today()
r_free = 0.05/365

class FinancialTransaction(object):
    def __init__(self,t,a,description=''):
        self.t= t
        self.a = a
        self.description = description
    def npv(self, t0=today, r=r_free):
        return self.a*exp(r*(t0-self.t).days)
    def __str__(self):
        return '%.2f dollars in %i days (%s)' % \
            (self.a, self.t, self.description)

class CashFlow(object):
    def __init__(self):
        self.transactions = []
    def add(self,transaction):
        self.transactions.append(transaction)
    def npv(self, t0, r=r_free):
        return sum(x.npv(t0,r) for x in self.transactions)
    def __str__(self):
        return '\n'.join(str(x) for x in self.transactions)
```

Here we assume t and t_0 are `datetime.date` objects that store a date. The `date` constructor takes the year, the month, and the day separated by a comma. The expression `(t0-t).days` computes the distance in days between t_0 and t.

What is the net present value at the beginning of 2012 for a fixed rate bond that pays one coupon of $1000 the 20th of each month for the following 24 month)?

```
>>> fixed_rate_bond = CashFlow()
>>> today = date(2012,1,1)
>>> for year in range(2012,2014):
...     for month in range(1,13):
...         coupon = FinancialTransaction(date(year,month,20),1000)
...         fixed_rate_bond.add(coupon)
>>> print(round(fixed_rate_bond.npv(today,r=0.05/365),0))
22826
```

This means the cost for this bond should be $22826.

2.5 *File input/output*

In Python you can open and write in a file with:

```
>>> file = open('myfile.txt', 'w')
>>> file.write('hello world')
>>> file.close()
```

Similarly, you can read back from the file with:

```
>>> file = open('myfile.txt', 'r')
>>> print(file.read())
hello world
```

Alternatively, you can read in binary mode with "rb", write in binary mode with "wb", and open the file in append mode "a", using standard C notation.

The `read` command takes an optional argument, which is the number of bytes. You can also jump to any location in a file using `seek`.

You can read back from the file with `read`

```
>>> print(file.seek(6))
>>> print(file.read())
world
```

and you can close the file with:

```
>>> file.close()
```

In the standard distribution of Python, which is known as CPython, variables are reference-counted, including those holding file handles, so CPython knows that when the reference count of an open file handle decreases to zero, the file may be closed and the variable disposed. However, in other implementations of Python such as PyPy, garbage collection is used instead of reference counting, and this means that it is possible that there may accumulate too many open file handles at one time, resulting in an error before the gc has a chance to close and dispose of them all. Therefore it is best to explicitly close file handles when they are no longer needed.

2.6 *Module* `import`

The real power of Python is in its library modules. They provide a large and consistent set of Application Programming Interfaces (APIs) to many system libraries (often in a way independent of the operating system).

For example, if you need to use a random number generator, you can do:

```
>>> import random
>>> print(random.randint(0, 9))
5
```

This prints a random integer between 0 and 9 (including 9), 5 in the example. The function `randint` is defined in the module `random`. It is also possible to import an object from a module into the current namespace:

```
>>> from random import randint
>>> print(randint(0, 9))
```

or import all objects from a module into the current namespace:

```
>>> from random import *
>>> print(randint(0, 9))
```

or import everything in a newly defined namespace:

```
>>> import random as myrand
>>> print(myrand.randint(0, 9))
```

In the rest of this book, we will mainly use objects defined in modules `math`, `cmath`, `os`, `sys`, `datetime`, `time` and `cPickle`. We will also use the `random` module but we will describe it in a later chapter.

In the following subsections we consider those modules that are most useful.

2.6.1 `math` *and* `cmath`

[SAY MORE]

2.6.2 `os`

This module provides an interface to the operating system API. For example:

```
>>> import os
>>> os.chdir('..')
>>> os.unlink('filename_to_be_deleted')
```

> *Some of the* `os` *functions, such as* `chdir`*, are not thread-safe, i.e. they should not be used in a multi-threaded environment.*

`os.path.join` is very useful; it allows the concatenation of paths in an OS-independent way:

```
>>> import os
>>> a = os.path.join('path', 'sub_path')
>>> print(a)
path/sub_path
```

System environment variables can be accessed via:

```
>>> print(os.environ)
```

which is a read-only dictionary.

2.6.3 `sys`

The `sys` module contains many variables and functions, but the one we use the most is `sys.path`. It contains a list of paths where Python searches for

modules. When we try to import a module, Python looks for it in all the folders listed in `sys.path`. If you install additional modules in some location and want Python to find them, you need to append the path to that location to `sys.path`.

```
>>> import sys
>>> sys.path.append('path/to/my/modules')
```

2.6.4 `datetime`

The use of the datetime module is best illustrated by some examples:

```
>>> import datetime
>>> print(datetime.datetime.today())
2008-07-04 14:03:90
>>> print(datetime.date.today())
2008-07-04
```

Occasionally you may need to time-stamp data based on the UTC time as opposed to local time. In this case you can use the following function:

```
>>> import datetime
>>> print(datetime.datetime.utcnow())
2008-07-04 14:03:90
```

The `datetime` module contains various classes: date, `datetime`, time and `timedelta`. The difference between two date or two datetime or two time objects is a `timedelta`:

```
>>> a = datetime.datetime(2008, 1, 1, 20, 30)
>>> b = datetime.datetime(2008, 1, 2, 20, 30)
>>> c = b - a
>>> print(c.days)
1
```

We can also parse dates and datetimes from strings for example:

```
>>> s = '2011-12-31'
>>> a = datetime.datetime.strptime(s,'%Y-%m-%d')  #modified
>>> print(s.year, s.day, s.month)
2011 31 12  #modified
```

Notice that "%Y" matches the 4-digits year, "%m" matches the month as a number (1-12), "%d" matches the day (1-31), "%H" matches the hour, "%M"

matches the minute, and "%S" matches the second. Check the documentation for more options.

2.6.5 `time`

The time module differs from `date` and `datetime` because it represents time as seconds from the epoch (beginning of 1970).

```
>>> import time
>>> t = time.time()
1215138737.571
```

Refer to the Python documentation for conversion functions between time in seconds and time as a `datetime`.

2.6.6 `urllib` *and* `json`

The `urllib` is a module to download data from a URL. For example we can download a web page using the following code:

```
>>> page = urllib.urlopen('http://www.google.com/')
>>> html = page.read()
```

Usually data posted online can be retrieved using the same mechanism. The main problem may be parsing the data (converting from the representation used to post it to a proper Python representation).

Below create a simple helper class that can download data from Yahoo Finance and convert each stock's historical data into a list of dictionaries. Each list element corresponds to a trading day of history of the stock and each dictionary stores the data relative to that trading day (date, open, close, volume, adjusted close, arithmetic_return, log_return, etc.):

Listing 2.1: in file: `numeric.py`

```
class YStock:
    """
    Class that downloads and stores data from Yahoo Finance
    Examples:
    >>> google = YStock('GOOG')
```

```
    >>> current = google.current()
    >>> price = current['price']
    >>> market_cap = current['market_cap']
    >>> h = google.historical()
    >>> last_adjusted_close = h[-1]['adjusted_close']
    >>> last_log_return = h[-1]['log_return']
    """
    URL_CURRENT = 'http://finance.yahoo.com/d/quotes.csv?s=%(symbol)s&f=%(columns)s
        '
    URL_HISTORICAL = 'http://ichart.yahoo.com/table.csv?s=%(s)s&a=%(a)s&b=%(b)s&c
        =%(c)s&d=%(d)s&e=%(e)s&f=%(f)s'
    def __init__(self,symbol):
        self.symbol = symbol.upper()

    def current(self):
        import urllib
        FIELDS = (('price', 'l1'),
                  ('change', 'c1'),
                  ('volume', 'v'),
                  ('average_daily_volume', 'a2'),
                  ('stock_exchange', 'x'),
                  ('market_cap', 'j1'),
                  ('book_value', 'b4'),
                  ('ebitda', 'j4'),
                  ('dividend_per_share', 'd'),
                  ('dividend_yield', 'y'),
                  ('earnings_per_share', 'e'),
                  ('52_week_high', 'k'),
                  ('52_week_low', 'j'),
                  ('50_days_moving_average', 'm3'),
                  ('200_days_moving_average', 'm4'),
                  ('price_earnings_ratio', 'r'),
                  ('price_earnings_growth_ratio', 'r5'),
                  ('price_sales_ratio', 'p5'),
                  ('price_book_ratio', 'p6'),
                  ('short_ratio', 's7'))
        columns = ''.join([row[1] for row in FIELDS])
        url = self.URL_CURRENT % dict(symbol=self.symbol, columns=columns)
        raw_data = urllib.urlopen(url).read().strip().strip('"').split(',')
        current = dict()
        for i,row in enumerate(FIELDS):
            try:
                current[row[0]] = float(raw_data[i])
            except:
                current[row[0]] = raw_data[i]
        return current

    def historical(self,start=None, stop=None):
        import datetime, time, urllib, math
```

```
        start =  start or datetime.date(1900,1,1)
        stop = stop or datetime.date.today()
        url = self.URL_HISTORICAL % dict(
            s=self.symbol,
            a=start.month-1,b=start.day,c=start.year,
            d=stop.month-1,e=stop.day,f=stop.year)
        # Date,Open,High,Low,Close,Volume,Adj Close
        lines = urllib.urlopen(url).readlines()
        raw_data = [row.split(',') for row in lines[1:]]
        previous_adjusted_close = 0
        series = []
        raw_data.reverse()
        for row in raw_data:
            adjusted_close = float(row[6])
            if previous_adjusted_close:
                arithmetic_return = adjusted_close/previous_adjusted_close-1.0

                log_return = math.log(adjusted_close/previous_adjusted_close)
            else:
                arithmetic_return = log_return = None
            previous_adjusted_close = adjusted_close
            series.append(dict(
               date = datetime.datetime.strptime(row[0],'%Y-%m-%d'),
               open = float(row[1]),
               high = float(row[2]),
               low = float(row[3]),
               close = float(row[4]),
               volume = float(row[5]),
               adjusted_close = adjusted_close,
               arithmetic_return = arithmetic_return,
               log_return = log_return))
        return series

    @staticmethod
    def download(symbol='goog',what='adjusted_close',start=None,stop=None):
        return [d[what] for d in YStock(symbol).historical(start,stop)]
```

Many web services return data in JSON format. JSON is slowly replacing XML as favorite protocol for data stransfer on the web. It is lighter, simpler to use and more human readbale. JSON is a serialized Javascript. the JSON data can be converted to a Python object using a library called `simplejson`. Once you have installed **simplejson** you can use it to convert Python objects into JSON objects and vice-versa:

```
>>> import simplejson
>>> a = [1,2,3]
>>> b = simplejson.dumps(a)
```

```
>>> print(type(b))
<type 'str'>
>>> c = simplejson.loads(b)
>>> a == c
True
```

`simplejson`'s `loads` and `dumps` method work very much as `cPickle`'s methods but they serialize the objects into a string using JSON instead of the pickle protocol.

2.6.7 `cPickle`

This is a very powerful module. It provides functions that can serialize almost any Python object, including self-referential objects. For example, let's build a weird object:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.x = 'something'
>>> a = [1 ,2, {'hello':'world'}, [3, 4, [myinstance]]]
```

and now:

```
>>> import cPickle
>>> b = cPickle.dumps(a)
>>> c = cPickle.loads(b)
```

In this example, `b` is a string representation of `a`, and `c` is a copy of `a` generated by de-serializing `b`. cPickle can also serialize to and de-serialize from a file:

```
>>> cPickle.dump(a, open('myfile.pickle', 'wb'))
>>> c = cPickle.load(open('myfile.pickle', 'rb'))
```

2.6.8 `sqlite` *database and persistence*

The Python dictionary type is very useful but it lacks persistence because it is stored in RAM (it is lost if a program ends) and cannot be shared by more than one process running concurrently. Moreover it is not transaction safe. This means that it is not possible to group operations together so that they succeed or fail as one.

Think for example of using the dictionary to store a bank account. The key is the account number and the value is a list of transactions. We want the dictionary to be safely stored on file. We want it to be accessible by multiple processes/applications. We want transaction safety: it should not be possible for an application to fail during a money transfer resulting in the disappearance of money.

Python provides a module called `shelve` with the same interface as `dict` which is stored on disk instead of RAM. One problem with this module is that the file is not locked when accessed. If two processes try to access it concurrently, the data becomes corrupted. This module also does not provide transactional safety.

The proper alternative consists of using a database. There are two types of databases: relational databases (which normally use SQL syntax) and non relational databases (often referred to as NoSQL). Key-value persistent storage databases usually follow under the latter category. Relational databases excel at storing structured data (in the form of tables), establishing relations between rows of those tables, and searches involving multiple tables linked by references. NoSQL databases excel at storing and retrieving schema-less data, replication of data (redundancy for fail safety).

Python comes with an embedded SQL database called SQLite. All data in the database are stored in one single file. It supports the SQL query language and transactional safety. It is very fast and allows concurrent read (from multiple processes) although not concurrent write (the file is locked when a process is writing to the file until the transaction is committed).

Installing and using any of these database systems is beyond the scope of this book and not necessary for our purposes. In particular we are not concerned with relations, data replications, and speed.

As an exercise we are going to implement a new Python class called `PersistentDictionary` which exposes an interface similar to a `dict` but uses the SQLite database for storage. The database file is created if it does not exist. `PersistentDictionary` will use a single table (also called `persistence` to store rows containing a key (`pkey`) and a value (`pvalue`).

For later conveience we will also add a method that can generate a UUID key. A UUID is a random string that is long enough and most likely unique. This means that two calls to the same function will return different value and the probability that the two values is the same is negligible. Python includes a library to generate UUID strings based on a common industry standard. We use the function uuid4 which also use the time and the ip of the machine to generate the UUID. This means the UUID is unlikely to have conflicts with (be equal to) another UUID generated on other machines. The `uuid` method will be useful to generate random unique keys.

We will also add a method that allows us to search for keys in the database using GLOB patterns (in a GLOB patter "*" represents a generic wildcard and "?" is a single character wildcard).

Here is the code:

Listing 2.2: in file: `numeric.py`

```
import os
import uuid
import sqlite3
import cPickle as pickle

class PersistentDictionary(object):
    """
    A sqlite based key,value storage.
    The value can be any pickable object.
    Similar interface to Python dict
    Supports the GLOB syntax in methods keys(),items(), __delitem__()

    Usage Example:
    >>> p = PersistentDictionary(path='test.sqlite')
    >>> key = 'test/' + p.uuid()
    >>> p[key] = {'a': 1, 'b': 2}
    >>> print(p[key])
    {'a': 1, 'b': 2}
    >>> print(len(p.keys('test/*')))
    1
    >>> del p[key]
    """

    CREATE_TABLE = "CREATE TABLE persistence (pkey, pvalue)"
    SELECT_KEYS = "SELECT pkey FROM persistence WHERE pkey GLOB ?"
    SELECT_VALUE = "SELECT pvalue FROM persistence WHERE pkey GLOB ?"
    INSERT_KEY_VALUE = "INSERT INTO persistence(pkey, pvalue) VALUES (?,?)"
```

```
    DELETE_KEY_VALUE = "DELETE FROM persistence WHERE pkey LIKE ?"
    SELECT_KEY_VALUE = "SELECT pkey,pvalue FROM persistence WHERE pkey GLOB ?"

    def __init__(self,
                 path='persistence.sqlite',
                 autocommit=True):
        self.path = path
        self.autocommit = autocommit
        create_table = not os.path.exists(path)
        self.connection  = sqlite3.connect(path)
        self.connection.text_factory = str # do not use unicode
        self.cursor = self.connection.cursor()
        if create_table:
            self.cursor.execute(self.CREATE_TABLE)
            self.connection.commit()

    def uuid(self):
        return str(uuid.uuid4())

    def keys(self,pattern='*'):
        "returns a list of keys filtered by a pattern, * is the wildcard"
        self.cursor.execute(self.SELECT_KEYS,(pattern,))
        return [row[0] for row in self.cursor.fetchall()]

    def __contains__(self,key):
        return True if self[key] else False

    def __iter__(self):
        for key in self:
            yield key

    def __setitem__(self,key,value):
        if value is None:
            del self[key]
            return
        self.cursor.execute(self.INSERT_KEY_VALUE,
                            (key, pickle.dumps(value)))
        if self.autocommit: self.connection.commit()

    def __getitem__(self,key):
        self.cursor.execute(self.SELECT_VALUE, (key,))
        row = self.cursor.fetchone()
        return pickle.loads(row[0]) if row else None

    def __delitem__(self,pattern):
        self.cursor.execute(self.DELETE_KEY_VALUE, (pattern,))
        if self.autocommit: self.connection.commit()

    def items(self,pattern='*'):
```

```
77          self.cursor.execute(self.SELECT_KEY_VALUE, (pattern,))
78          return [(row[0],pickle.loads(row[1])) \
79                      for row in self.cursor.fetchall()]
```

This code now allows us to do the following:

- Create a persistent dictonary:

```
1 >>> p = PersistentDictionary(path='storage.sqlite',autocommit=False)
```

- Store data in it

```
1 >>> p['some/key'] = 'some value'
```

  where "some/key" must be a string and "some value" can be any Python pickable object.

- Generate a UUID to be used as the key:

```
1 >>> key = p.uuid()
2 >>> p[key] = 'some other value'
```

- Retrieve the data

```
1 >>> data = p['some/key']
```

- Loop over keys

```
1 >>> for key in p: print(key, p[key])
```

- List all keys

```
1 >>> keys = p.keys()
```

- List all keys matching a pattern

```
1 >>> keys = p.keys('some/*')
```

- List all key-value pairs matching a pattern

```
1 >>> for key,value in p.items('some/*'): print(key, value)
```

- Delete keys matching a pattern:

```
1 >>> del p['some/*']
```

We will now use our persistence storage to download 2011 financial data from the SP100 stocks. This will allow us to later perform various analysis tasks on these stocks:

Listing 2.3: in file: `numeric.py`

```
>>> SP100 = ['AA', 'AAPL', 'ABT', 'AEP', 'ALL', 'AMGN', 'AMZN', 'AVP', 'AXP', 'BA',
      'BAC',
... 'BAX', 'BHI', 'BK', 'BMY', 'BRK.B', 'CAT', 'C', 'CL', 'CMCSA', 'COF', 'COP', '
     COST',
... 'CPB', 'CSCO', 'CVS', 'CVX', 'DD', 'DELL', 'DIS', 'DOW', 'DVN', 'EMC', 'ETR', '
     EXC',
... 'F', 'FCX', 'FDX', 'GD', 'GE', 'GILD', 'GOOG', 'GS', 'HAL', 'HD', 'HNZ', 'HON',
      'HPQ',
... 'IBM', 'INTC', 'JNJ', 'JPM', 'KFT', 'KO', 'LMT', 'LOW', 'MA', 'MCD', 'MDT', '
     MET',
... 'MMM', 'MO', 'MON', 'MRK', 'MS', 'MSFT', 'NKE', 'NOV', 'NSC', 'NWSA', 'NYX', '
     ORCL',
... 'OXY', 'PEP', 'PFE', 'PG', 'PM', 'QCOM', 'RF', 'RTN', 'S', 'SLB', 'SLE', 'SO',
     'T',
... 'TGT', 'TWX', 'TXN', 'UNH', 'UPS', 'USB', 'UTX', 'VZ', 'WAG', 'WFC', 'WMB', '
     WMT',
... 'WY', 'XOM', 'XRX']
>>> from datetime import date
>>> storage = PersistentDictionary('sp100.sqlite')
>>> for symbol in SP100:
...     key = symbol+'/2011'
...     if not key in storage:
...         storage[key] = YStock(symbol).historical(start=date(2011,1,1),
...                                                  stop=date(2011,12,31))
```

Notice that while storing one item may be slower than storing the item in its own files, accessing the file system becomes lower and slower the more files there are. Storing data in database, long term, is a winning strategy as it scales much better and it is easier to search for data. Which type of database is most appropriate depends on the type of data and the the type of queries we need to perform on the data.

2.6.9 `matplotlib`

`matplotlib` is the *de facto* standard plotting library for Python. It is one of the best and more versatile plotting libraries out there. It has a two modes of operation. On mode of operation, called `pylab`, follows a matlab-like syntax.

The other mode follows a more Python-style syntax. Here we shall use this latter:

In matplotlib we need to distinguish the following objects:

- `Figure`: a blank grid which can contain pairs of XY axes
- `Axes`: a pair of XY axes that may contain multiple superimposed plots
- `Plot`: a representation for a dataset like a line plot or a scatter plot
- `Canvas`: a binary representation of a figure with everything that it contains.

In matplotlib canvas can be visualized in a window or serialized into an image file. Here we will take the latter approach and will create two helper functions that take data and configuration parameters, and output PNG images.

We start by importing matplotlib and other required libraries:

Listing 2.4: in file: `numeric.py`

```
import math
import cmath
import random
import os
import tempfile

os.environ['MPLCONfigureDIR'] = tempfile.mkdtemp()
try:
    from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
    from matplotlib.figure import Figure
    from matplotlib.patches import Ellipse
except ImportError:
    print('warning: matplotlib not available')
```

Now we define a single helper that can plot lines, points with error bars, histograms and scatterplots on one single canvas:

Listing 2.5: in file: `numeric.py`

```
def draw(title='title',xlab='x',ylab='y',filename='tmp.png',
         linesets=None, pointsets=None, histsets=None, ellisets=None,
         xrange=None, yrange=None):
    figure = Figure(frameon=False)
    figure.set_facecolor('white')
    axes = figure.add_subplot(111)
```

```
    axes.grid(True)
    if title: axes.set_title(title)
    if xlab: axes.set_xlabel(xlab)
    if ylab: axes.set_ylabel(ylab)
    if xrange: axes.set_xlim(xrange)
    if yrange: axes.set_ylim(yrange)
    legend = [],[]

    for histset in histsets or []:
        data = histset['data']
        bins = histset.get('bins',20)
        color = histset.get('color','blue')
        q = axes.hist(data,bins, color=color)
        if 'legend' in histset:
            legend[0].append(q[0])
            legend[1].append(histset['legend'])

    for lineset in linesets or []:
        data = lineset['data']
        color = lineset.get('color','black')
        linestyle = lineset.get('style','-')
        linewidth = lineset.get('width',2)
        x = [p[0] for p in data]
        y = [p[1] for p in data]
        q = axes.plot(x, y, linestyle=linestyle,
                      linewidth=linewidth, color=color)
        if 'legend' in lineset:
            legend[0].append(q[0])
            legend[1].append(lineset['legend'])

    for pointset in pointsets or []:
        data = pointset['data']
        color = pointset.get('color','black')
        marker = pointset.get('marker','o')
        linewidth = pointset.get('width',2)
        x = [p[0] for p in data]
        y = [p[1] for p in data]
        yerr = [p[2] for p in data]
        q = axes.errorbar(x, y, yerr=yerr, fmt=marker,
                          linewidth=linewidth, color=color)
        if 'legend' in pointset:
            legend[0].append(q[0])
            legend[1].append(pointset['legend'])


    for elliset in ellisets or []:
        data = elliset['data']
        color = elliset.get('color','blue')
        for point in data:
```

```
            x, y = point[:2]
            dx = point[2] if len(point)>2 else 0.01
            dy = point[3] if len(point)>3 else dx
            ellipse = Ellipse(xy=(x,y),width=dx,height=dy)
            axes.add_artist(ellipse)
            ellipse.set_clip_box(axes.bbox)
            ellipse.set_alpha(0.5)
            ellipse.set_facecolor(color)

    if legend[0]: axes.legend(*legend)
    canvas = FigureCanvas(figure)
    canvas.print_png(open(filename,'wb'))
```

Notice we only make one set of `axes`. The `111` in `figure.add_subplot(111)` indicates thet we want a grid of 1×1 axes and we ask for the 1st one of them (the only one).

The `linesets` parameter is a list of dictionaries. Each dictionary must have a "data" key corresponding to a list of (x, y) values. Each dictionary is rendered by a line connecting the points. It can have a "label", a "color", a "style" and a "width".

The `pointsets` parameter is a list of dictionaries. Each dictionary must have a "data" key corresponding to a list of $(x, y, \delta y)$ values. Each dictionary is rendered by a set of circles with error bars. It can optionally have a "label", a "color", and a "marker" (symbol to replace the circle).

The `histsets` parameter is a list of dictionaries. Each dictionary must have a "data" key corresponding to a list of x values. Each dictionary is rendered by histogram. Each dictionary can optionally have a "label" and a "color".

The `ellisets` parameter is also a list of dictionaries. Each dictionary must have a "data" key corresponding to a list of $(x, y, \delta x, \delta y)$ values. Each dictionary is rendered by a set of ellipses. It can optionally have a "color".

We chose to draw all these types of plots with one single function because it is common to superimpose fitting lines to histograms, points and scatterplots.

As an example we can plot the adjusted closing price for AAPL:

Listing 2.6: in file: `numeric.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> appl = storage['AAPL/2011']
```

```
>>> points = [(x,y['adjusted_close']) for (x,y) in enumerate(appl)]
>>> draw(title='Apple Stock (2011)',xlab='trading day',ylab='adjusted close',
...      linesets = [{'label':'AAPL','data':points}],filename='images/aapl2011.png'
    )
```

Figure 2.1: Example of line plot. Adjusted closing price for the APPL stock in 2011 (source Yahoo Finance)

Here is an example here of a histogram of daily arithmetic returns for the AAPL stock in 2011:

Listing 2.7: in file: `numeric.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> appl = storage['AAPL/2011'][1:] # skip 1st day
>>> points = [day['arithmetic_return'] for day in appl]
>>> draw(title='Apple Stock (2011)',xlab='arithmetic return', ylab='frequency',
...      histsets = [{'data':points}],filename='images/aapl2011hist.png')
```

Here is a scatterplot for random data points:

Listing 2.8: in file: `numeric.py`

```
>>> from random import gauss
>>> points = [(gauss(0,1),gauss(0,1),gauss(0,0.2),gauss(0,0.2)) for i in range(30)]
>>> draw(title='example scatter plot', xrange=(-2,2), yrange=(-2,2),
```

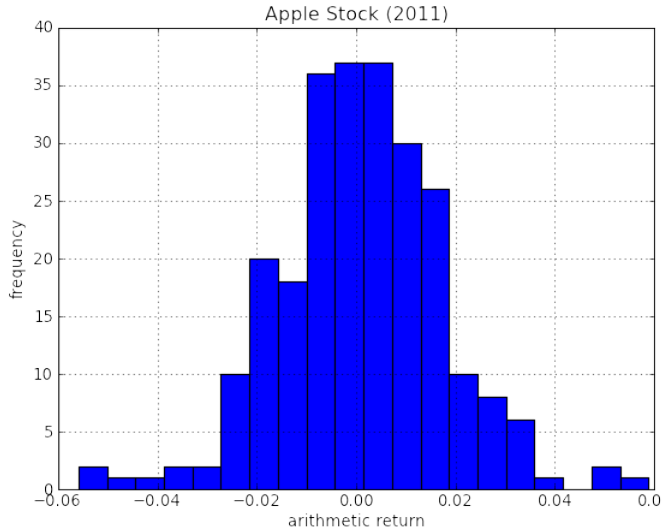

Figure 2.2: Example of histogram plot. Distribution of daily arithmetic returns for the APPL stock in 2011 (source Yahoo Finance)

```
...      ellisets = [{'data':points}],filename='images/scatter.png')
```

here is a scatter plot showing the return and variance of the S&P100 stocks:

Listing 2.9: in file: `numeric.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> points = []
>>> for key in storage.keys('*/2011'):
...     v = [day['log_return'] for day in storage[key][1:]]
...     ret = sum(v)/len(v)
...     var = sum(x**2 for x in v)/len(v) - ret**2
...     points.append((var*math.sqrt(len(v)),ret*len(v),0.0002,0.02))
>>> draw(title='S&P100 (2011)',xlab='risk',ylab='return',
...      ellisets = [{'data':points}],filename='images/sp100rr.png',
...      xrange = (min(p[0] for p in points),max(p[0] for p in points)),
...      yrange = (min(p[1] for p in points),max(p[1] for p in points)))
```

Notice that we have multiplied the daily log return for the number of days in one year to obtain the annual return. Similarly we have multiplied the daily volatility by the square root of the number of days in one year to obtain the

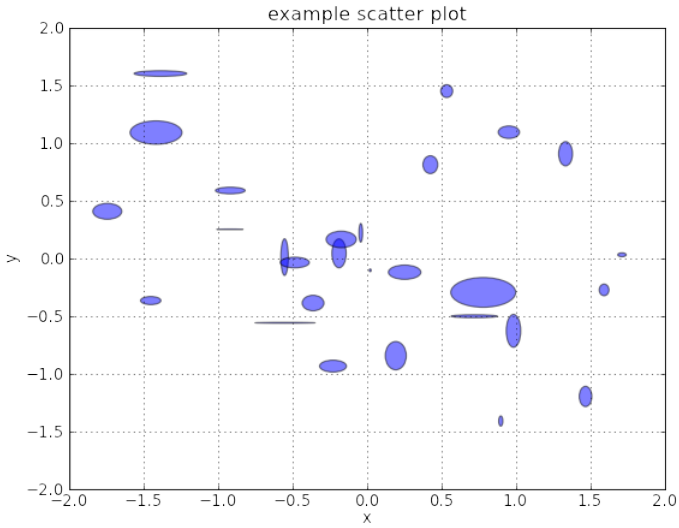

Figure 2.3: Example of scatter plot using some random points.

annual volatility (risk). The reason for this procedure will be explained in a later chapter.

The helper below takes a 2D grid (a list of lists) of scalar values, interpolates them and produce a 2D color plot:

Listing 2.10: in file: `numeric.py`

```
def color2d(title='title',xlab='x',ylab='y',
            data=[[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7]],
            filename = 'tmp.png'):
    figure=Figure()
    figure.set_facecolor('white')
    axes=figure.add_subplot(111)
    if title: axes.set_title(title)
    if xlab: axes.set_xlabel(xlab)
    if ylab: axes.set_ylabel(ylab)
    image=axes.imshow(data)
    image.set_interpolation('bilinear')
    canvas = FigureCanvas(figure)
    canvas.print_png(open(filename,'wb'))
```

Here is an example:

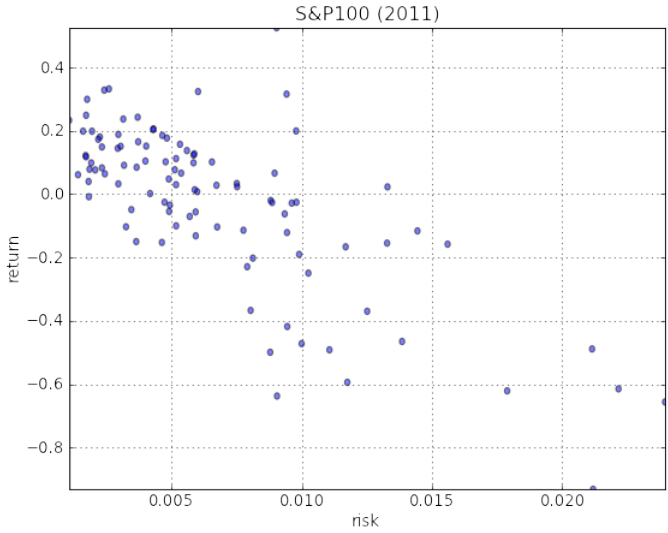

Figure 2.4: Example of scatter plot. Risk-return plot for the S&P100 stocks in 2011 (source Yahoo Finance)

Listing 2.11: in file: `numeric.py`

```
>>> def f(x,y): return (x-1)**2+(y-2)**2
>>> points = [[f(0.1*i-3,0.1*j-3) for i in range(61)] for j in range(61)]
>>> color2d(title='example 2d function',
...       data = points,filename='images/color2d.png')
```

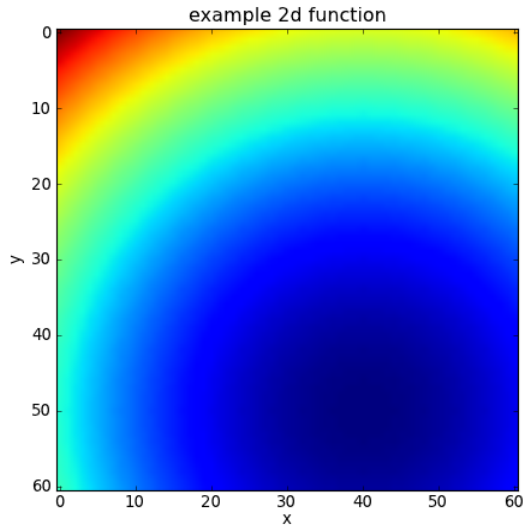

Figure 2.5: Example of 2d color plot using for $f(x, y) = (x - 1)^2 + (y - 2)^2$

3
Theory of Algorithms

An algorithm is a step-by-step procedure for solving a problem, and is typically developed before doing any programming. In fact, it is independent of any programming language. Efficient algorithms can have a dramatic effect on our problem-solving capabilities. The issues that will concern us when developing and analyzing algorithms are:

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: i.e., 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 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 the best of any 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 to store the input array.

The insertion sort works by iterations. Every iteration i of insertion sort removes one element from the input data and inserts it into the correct position in the already-sorted sub-array **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.

```
def insertion_sort(A):
    for i in range(1,len(A)):
        for j in range(i,0,-1):
            if A[j]<A[j-1]:
                A[j], A[j-1] = A[j-1], A[j]
            else: break
```

Here is an example:

```
>>> import random
>>> a=[random.randint(0,100) for k in range(20)]
>>> insertion_sort(a)
>>> print(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

In best case for the insertion sort is realized when the input is already sorted. In this case the inner for loop exists (break) always at the first iteration, thus only the most outer loop is important and it 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 prove and we will do it below that $T_{worst}(n) \propto n^2$. For this algorithm a statistical analysis shows that worst case is also the average case.

Often we cannot determine exactely the running time function but we may still want to set bounds to the running time.

We can 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 above definitions in a more formal way:

$$O(g(n)) \stackrel{def}{=} \{f(n) : \exists n_0, c_0, \ \forall n > n_0, \ 0 \leq f(n) < c_0 g(n)\} \tag{3.1}$$

$$\Omega(g(n)) \stackrel{def}{=} \{f(n) : \exists n_0, c_0, \ \forall n > n_0, \ 0 \leq c_0 g(n) < f(n)\} \tag{3.2}$$

$$\Theta(g(n)) \stackrel{def}{=} O(g(n)) \cap \Omega(g(n)) \tag{3.3}$$

$$o(g(n)) \stackrel{def}{=} O(g(n)) - \Omega(g(n)) \tag{3.4}$$

$$\omega(g(n)) \stackrel{def}{=} \Omega(g(n)) - O(g(n)) \tag{3.5}$$

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

Compute the limit

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

And look up the result in the following table:

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

Notice the above practical rule assumes the limits exist and that may not be the case. Moreover the inverse is not true. For example:

$$f(n) \in \Theta(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is positive} \tag{3.8}$$

Here are some more useful rules:

- Any polynomial $T(n) = P_m(n)$ of degree m is $T(n) \in \Theta(n^m), \in O(n^m)$
- $T_1(n) \in O(f(n))$ and $T_2 \in O(g(n)) \Rightarrow T_1(n) + T_2(n) \in O(\max(f(n), g(n)))$
- $T_1(n) \in O(f(n))$ and $T_2 \in O(g(n)) \Rightarrow T_1(n)T_2(n) \in O(f(n)g(n))$

Here are some examples:

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

$$\lim_{n\to\infty} \frac{n \log n + 3n}{n^2} \overset{l'Hopital}{\longrightarrow} \lim_{n\to\infty} \frac{1/n}{2} = 0 \tag{3.9}$$

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

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

More formally we can write:

$$T_{worst}(n) \in O(g(n)) \quad \Rightarrow \quad A \in O(g(n)) \tag{3.10}$$

$$T_{best}(n) \in \Omega(g(n)) \quad \Rightarrow \quad A \in \Omega(g(n)) \tag{3.11}$$

$$A \in O(g(n)) \text{and} A \in \Omega(g(n)) \quad \Rightarrow \quad A \in \Theta(g(n)) \tag{3.12}$$

$$\tag{3.13}$$

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

3.1.1 Best and worst running time

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

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

```
def find_minimum(A):
    minimum = None
    for element in A:
        if minimum is None or element<minimum:
            minimum = element
    return minimum
```

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

In the above algorithm the most nested operation is the evaluation of the `if` statement and that is executed for each element in `A`; .i.e. n times.

Therefore both the best and worst running time 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 02)
- 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.

We can add up the the time it takes to compute each loop iteration. For a loop iteration that does not contain further loops, hence we can assume the running time for one iteration 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 def loop0(n):
2     for i in range(0,n):
3         print(i)
```

Which we can map into:

$$T(n) = \sum_{i=0}^{i<n} 1 = n \in \Theta(n) \Rightarrow \texttt{loop0} \in \Theta(n) \tag{3.14}$$

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

```
1 def loop1(n):
2     for i in range(0,n*n):
3         print(i)
```

and here is the corresponding running time formula:

$$T(n) = \sum_{i=0}^{i<n^2} 1 = n^2 \in \Theta(n^2) \Rightarrow \texttt{loop1} \in \Theta(n^2) \tag{3.15}$$

The example below provides an example of nested loops.

```
def loop2(n):
    for i in range(0,n):
        for j in range(0,n):
            print(i,j)
```

Here the time for the inner loop does not depend on the value of the value of the counter of the outer loop therefore:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<n} 1 = \sum_{i=0}^{i<n} n = n^2 + ... \in \Theta(n^2) \Rightarrow \texttt{loop2} \in \Theta(n^2) \tag{3.16}$$

This is not always the case. In the code below the inner loop depends on value of the outer loop:

```
def loop3(n):
    for i in range(0,n):
        for j in range(0,i):
            print(i,j)
```

Therefore, when we write its running time in terms of a sum we have to be careful that the upper limit of the inner sum must be the upper limit of the outer sum:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i} 1 = \sum_{i=0}^{i<n} i = \frac{1}{2} n(n-1) \in \Theta(n^2) \Rightarrow \texttt{loop3} \in \Theta(n^2) \tag{3.17}$$

The appendix of this book provides examples of typical sums that come up in these type 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

```
def loop4(n):
    for i in range(0,n):
        for j in range(0,i*i):
            print(i,j)
```

therefore the formula for the running time is more complicated:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{i<n} i^2 = \frac{1}{6} n(n-1)(2n-1) \in \Theta(n^3) \quad (3.18)$$

$$\Rightarrow \quad \texttt{loop4} \in \Theta(n^3) \quad (3.19)$$

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

Example: concatenateo

```
def concatenate0(n):
    for i in range(n*n):
        print(i)
    for j in range(n*n*n):
        print(j)
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate0} \in \Theta(n^3) \quad (3.20)$$

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:

```
def concatenate1(n):
    if a<0:
        for i in range(n*n):
            print(i)
    else:
        for j in range(n*n*n):
            print(j)
```

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

$$T_{best}(n) = \Theta(\min(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in \Omega(n^2) \quad (3.22)$$

This can be expressed more formally as follows:

$$O(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \tag{3.23}$$
$$\Theta(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \tag{3.24}$$
$$\Omega(f(n)) + \Theta(g(n)) = \Omega(f(n)) \text{ iff } f(n) \in \Omega(g(n)) \tag{3.25}$$

Which we can apply as in the following example:

$$T(n) = [\underbrace{n^2 + n + 3}_{\Theta(n^2)} + \underbrace{e^n - \log n}_{\Theta(e^n)}] \in \Theta(e^n) \text{ because } n^2 \in O(e^n) \tag{3.26}$$

3.2 *Recurrence relations*

The *merge sort* is another sorting algorithm. It is faster than the insertion sort. It was invented by John von Neumann, the physicist credited for inventing also the 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 sub-array 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 Python implementation of the merge sort:

```
def mergesort(A, p=0, r=None):
    if r is None: r = len(A)
    if p<r-1:
        q = int((p+r)/2)
        mergesort(A,p,q)
        mergesort(A,q,r)
        merge(A,p,q,r)

def merge(A,p,q,r):
    B,i,j = [],p,q
    while True:
```

```
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==q:
            while j<r:
                B.append(A[j])
                j=j+1
            break
        if j==r:
            while i<q:
                B.append(A[i])
                i=i+1
            break
    A[p:r]=B
```

Since this algorithm calls itself *recursively*, it is more difficult to compute its running time. In this section we will explain how to do it.

Consider the `merge` algorithm first. At each step it increases 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:

$$\text{merge} \in \Theta(r-p) \tag{3.27}$$

We cannot compute the running time of the `mergesort` using the same direct analysis but we can assume its running time is $T(n)$ where $n = r - p$ 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

$$T(n) = 2T(n/2) + n \tag{3.28}$$

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 the categories below:

$$T(n) = aT(n-b) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(max(a^n, nf(n))) \quad (3.29)$$

$$T(n) = T(b) + T(n-b-a) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(nf(n)) \quad (3.30)$$

$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m) \quad (3.31)$$

$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log n) \quad (3.32)$$

$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \quad (3.33)$$

$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m \log^p n) \quad (3.34)$$

$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log^{p+1}(n)) \quad (3.35)$$

$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \quad (3.36)$$

$$T(n) = aT(n/b) + \Theta(q^n) \Rightarrow T(n) \in \Theta(q^n) \quad (3.37)$$

$$T(n) = aT(n/a-b) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(f(n)\log(n)) \quad (3.38)$$

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

These results are a practical simplificaiton of a theorem known as *Master Theorem* [**?**]

3.2.1 *Reducible Recurrence Relations*

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

Consider the following recurrence relation:

$$T(n) = 2T(\sqrt{n}) + \log n \quad (3.39)$$

we can replace n with $e^k = n$ in eq.(3.39) and obtain:

$$T(e^k) = 2T(e^{k/2}) + k \quad (3.40)$$

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

$$\underbrace{S(k)}_{T(e^k)} = 2\,\underbrace{S(k/2)}_{T(e^{k/2})} + k \tag{3.41}$$

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:

$$T(n) = S(\log n) \in \Theta(\underbrace{\log n}_{k} \log \underbrace{\log n}_{k}) \tag{3.42}$$

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

Here are some examples:

```
def factorial1(n):
    if n==0:
        return 1
    else:
        return n*factorial1(n-1)
```

$$T(n) = T(n-1) + 1 \Rightarrow T(n) \in \Theta(n) \Rightarrow \texttt{factorial1} \in \Theta(n) \tag{3.43}$$

```
def recursive0(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*n*recursive0(n-1)
```

$$T(n) = T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(n^2) \Rightarrow \texttt{recursive0} \in \Theta(n^3) \tag{3.44}$$

```
def recursive1(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*recursive1(n-1)*recursive1(n-1)
```

$$T(n) = 2T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(2^n) \Rightarrow \texttt{recursive1} \in \Theta(2^n) \quad (3.45)$$

```
def recursive2(n):
    if n==0:
        return 1
    else:
        a=factorial0(n)
        return a*recursive2(n/2)*recursive1(n/2)
```

$$T(n) = 2T(n/2) + P_1(n) \Rightarrow T(n) \in \Theta(n \log n) \Rightarrow \texttt{recursive2} \in \Theta(n \log n) \quad (3.46)$$

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

```
def binary_search(A,element):
    a,b = 0, len(A)-1
    while b>=a:
        x = int((a+b)/2)
        if A[x]<element:
            a = x+1
        elif A[x]>element:
            b = x-1
        else:
            return x
    return None
```

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:

$$T(n) = T(n/2) + 1 \Rightarrow \texttt{binary\_search} \in \Theta(\log n) \quad (3.47)$$

The idea of the `binary_search` is used in the bisection method for solving non-linear equations.

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 so as 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 depends on the 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 and recursvely sorting (conquering) each of the smaller arrays.

Most Divide and conquer algorithms are also 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 which 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 since identical sub-instances may arise. 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 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:

```
1 def mergesort_nonrecursive(A):
```

```
    blocksize, n = 1, len(A)
    while blocksize<n:
        for p in range(0, n, 2*blocksize):
            q = p+blocksize
            r = min(q+blocksize, n)
            if r>q:
                Merge(A,p,q,r)
        blocksize = 2*blocksize
```

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

$$T_{best} \in \Theta(n \log n) \tag{3.48}$$
$$T_{average} \in \Theta(n \log n) \tag{3.49}$$
$$T_{worst} \in \Theta(n \log n) \tag{3.50}$$
$$T_{memory} \in \Theta(1) \tag{3.51}$$

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 which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.

There are other types of algorithms that do not follow in any of the above categories. One is, for example, backtracking. Backtracking is not covered in this course.

3.3.1 *Memoization*

One case of Top-Down approach which 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, instead of running again, the output is retrieved from the cache.

Consider for example Fibonacci numbers:

$$\text{Fib}(0) = 0 \tag{3.52}$$
$$\text{Fib}(1) = 1 \tag{3.53}$$
$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \text{ for } n > 1 \tag{3.54}$$

which we can implement using Divide and Conquer as follows:

```
def fib(n):
    return n if n<2 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 imput values and keep solving the same subproblem over and over.

In Python can implement memoization using the following decorator:

Listing 3.1: in file: `numeric.py`

```
class memoize(object):
    def __init__ (self, f):
        self.f = f
        self.storage = {}
    def __call__ (self, *args, **kwargs):
        key = str((self.f.__name__, args, kwargs))
        try:
            value = self.storage[key]
        except KeyError:
            value = self.f(*args, **kwargs)
            self.storage[key] = value
        return value
```

and simply decorating the recursive function as follows:

Listing 3.2: in file: `numeric.py`

```
@memoize
def fib(n):
    return n if n<2 else fib(n-1)+fib(n-2)
```

which we can call as:

Listing 3.3: in file: `numeric.py`

```
>>> print(fib(11))
89
```

A decorator is a Python function that takes a function and returns a callable object (or a function) to replace the one passed as input. In the previous example we are using the `@memoize` decorator to replace the `fib` function with the `__call__` argument of the `memoize` class.

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

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

```
def fib(n):
    if n < 2: return n
    a, b = 0, 1
    for i in range(1,n):
        a, b = b, a+b
    return b
```

This also makes 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 3.4: in file: `numeric.py`

```
class memoize_persistent(object):
    STORAGE = 'memoize.sqlite'
    def __init__ (self, f):
        self.f = f
```

```
5           self.storage = PersistentDictionary(memoize_persistent.STORAGE)
6       def __call__ (self, *args, **kwargs):
7           key = str((self.f.__name__, args, kwargs))
8           try:
9               value = self.storage[key]
10          except KeyError:
11              value = self.f(*args, **kwargs)
12              self.storage[key] = value
13          return value
```

We can use it as we did before but we can now start and stop the program or run concurrent parallel programs and, as long as they have access to the "memoize.sqlite" file, they will share the cache.

3.4 *List*

3.4.1 *List*

3.4.2 *Stack*

A stack data structure is a container and it is usually implemented as a list. It has the properly that the first thing you can take out is the last thing you put into you. The methods to insert data in the container is called *push* and the method to extract data is called *pop*.

In python we can implement push by appending an item at the end of a list (Python has already a method for this called `.append`) and we can implement pop by removing the last element of a list and returning it (Python has a method for this called `.pop`).

FILL HERE

3.4.3 *Queue*

3.4.4 *Sorting*

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

```
def quicksort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=partition(A,p,r)
        quicksort(A,p,q)
        quicksort(A,q+1,r)

def partition(A,i,j):
    x=A[i]
    h=i
    for k in range(i+1,j):
        if A[k]<x:
            h=h+1
            A[h],A[k] = A[k],A[h]
    A[h],A[i] = A[i],A[h]
    return h
```

The running time of the quicksort is given by:

$$T_{best} \in \Theta(n \log n) \tag{3.55}$$
$$T_{average} \in \Theta(n \log n) \tag{3.56}$$
$$T_{worst} \in \Theta(n^2) \tag{3.57}$$
$$\tag{3.58}$$

The *counting sort* algorithm is special because it only works for arrays of 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:

```
def countingsort(A):
    if min(A)<0:
        raise '_counting_sort List Unbound'
```

```
    i, n, k = 0, len(A), max(A)+1
    C = [0]*k
    for j in range(n):
        C[A[j]] = C[A[j]]+1
    for j in range(k):
        while C[j]>0:
           (A[i], C[j], i) = (j, C[j]-1, i+1)
```

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

$$T_{best} \in \Theta(k+n) \tag{3.59}$$
$$T_{average} \in \Theta(k+n) \tag{3.60}$$
$$T_{worst} \in \Theta(k+n) \tag{3.61}$$
$$T_{memory} \in \Theta(k) \tag{3.62}$$

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

3.5 Tree Algorithms

3.5.1 Heapsort and priority queues

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

It starts from one top node called *root*. Each node has zero, one or two children. It is called complete because nodes have been added from top to bottom, left two 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 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:

```
def heap_parent(i):
    return int((i-1)/2)

```

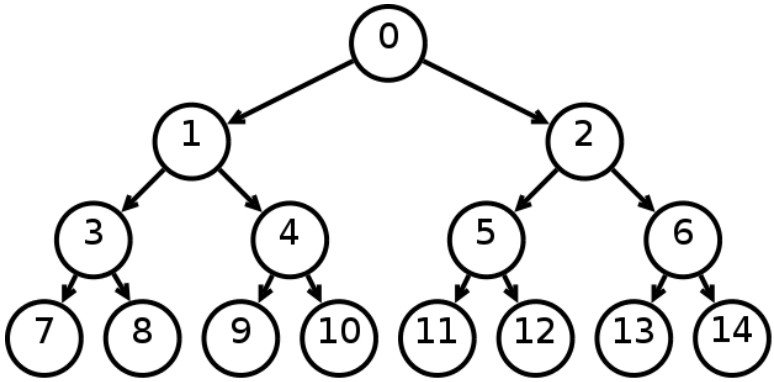

Figure 3.1: Example of a heap data structure. The number represent not the data in the heap, but the numbering of the nodes.

```
def heap_left_child(i):
    return 2*i+1

def heap_right_child(i):
    return 2*i+2
```

We can store data (for example numbers) in the nodes (or in the corresponding array). If the data is stored in such a way that the value at one node is always greater or equal then 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:

```
def heapify(A):
    for i in range(int(len(A)/2)-1,-1,-1):
        heapify_one(A,i)

def heapify_one(A,i,heapsize=None):
    if heapsize is None:
        heapsize = len(A)
    left = 2*i+1
    right = 2*i+2
    if left<heapsize and A[left]>A[i]:
        largest = left
    else:
        largest = i
    if right<heapsize and A[right]>A[largest]:
        largest = right
    if largest!=i:
```

```
17          (A[i], A[largest]) = (A[largest], A[i])
18          heapify_one(A,largest,heapsize)
```

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 above 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 it the root node. We can fix this by a single call to `heapify`.

This is a possible implementation for the `heapsort`:

```
1  def heapsort(A):
2      heapify(A)
3      n = len(A)
4      for i in range(n-1,0,-1):
5          (A[0],A[i]) = (A[i],A[0])
6          heapify_one(A,0,i)
```

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

$$T_{best} \in \Theta(n) \tag{3.63}$$
$$T_{average} \in \Theta(n \log n) \tag{3.64}$$
$$T_{worst} \in \Theta(n \log n) \tag{3.65}$$
$$T_{memory} \in \Theta(1) \tag{3.66}$$

A heap can be used to implemented as a priority queue. I.e. a 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` the remaining data) and a function to push a new value into the heap:

```
def heap_pop(A):
    if len(A)<1:
        raise RuntimeErro, 'Heap Underflow'
    largest = A[0]
    A[0] = A[len(A)-1]
    del A[len(A)-1]
    heapify_one(A,0)
    return largest

def heap_push(A,value):
    A.append(value)
    i = len(A)-1
    while i>0:
        j = heap_parent(i)
        if A[j]<A[i]:
            (A[i],A[j],i) = (A[j],A[i],j)
        else:
            break
```

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

$$T_{best} \in \Theta(1) \tag{3.67}$$
$$T_{average} \in \Theta(\log n) \tag{3.68}$$
$$T_{worst} \in \Theta(\log n) \tag{3.69}$$
$$T_{memory} \in \Theta(1) \tag{3.70}$$

Here is an example:

```
>>> a = [6,2,7,9,3]
>>> heap = []
>>> for element in a: heap_push(heap,element)
>>> while heap: print(heap_pop(heap))
9
7
6
3
2
```

Heaps find application in many numerical algorithms. In fact there is even a built-in Python module for them called `heapq` which provides similar functionality to the functions defined here, except that we defined a max heap (pops the max element) while `heapq` is a min heap (pops the minimum):

```
>>> from heapq import heappop, heappush
>>> a = [6,2,7,9,3]
>>> heap = []
>>> for element in a: heappush(heap,element)
>>> while heap: print(heappop(heap))
9
7
6
3
2
```

Notice the `heappop` instead of `heap_pop` and `heappush` instead of `heap_push`.

3.5.2 *Binary search trees*

A binary tree is a tree where 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 of equal to the value of its left child and less 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 we are looking for is less then 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 by simply 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 we are looking for or we find the end of the tree. This means 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 $d = \log n$.

In order to implment it we need to have a class to represent a binary tree:

```
class BinarySearchTree(object):
    def __init__(self):
        self.left = self.right = None
        self.key = self.value =  None
    def __setitem__(self,key,value):
        if self.key == None:
            self.key, self.value = key, value
        elif key == self.key:
            self.value = value
        elif key < self.key:
            if self.left:
                self.left.insert(key,value)
            else:
                self.left = BinarySearchTree(key,value)
        else:
            if self.right:
                self.right.insert(key,value)
            else:
                self.right = BinarySearchTree(key,value)
    def __getitem__(self,key):
        if self.key == None:
            retur None
        elif key == self.key:
            return self.value
        elif key<self.key and self.left:
            return self.left[key]
        elif key>self.key and self.right:
            return self.right[key]
        else:
            return None
    def min(self):
        node = self
        while node.left:
            node = self.left
        return node.key, node.value
    def max(self):
        node = self
        while node.right:
            node = self.right
        return node.key, node.value
```

which allows to create a binary tree and use a binary tree as follows:

```
>>> root = BinarySearchTree()
>>> root[5] = 'aaa'
>>> root[3] = 'bbb'
>>> root[8] = 'ccc'
```

```
>>> print(root.left.key)
3
>>> print(root.left.value)
bbb
>>> print(root[3])
bbb
>>> print(root.max())
8 ccc
```

Notice the case of empty tree is treated as an exception `key = None`.

3.5.3 *AVL trees*

AVL trees are binary search trees that are rebalanced after each insertion/deletion. They are rebalanced in such a way that for each node the height of the left subtree minus 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:

$$T_{best} \in \Theta(1) \tag{3.71}$$
$$T_{average} \in \Theta(\log n) \tag{3.72}$$
$$T_{worst} \in \Theta(\log n) \tag{3.73}$$
$$\tag{3.74}$$

3.5.4 *k-trees, B-trees and Red-black trees*

Until now we have considered binary trees (each node has 2 children and stores 1 value). We can generalize this to *k*trees where each node has k children and store more than one value. B-trees are normally used to implement databases and that is beyond the scope of this book.

3.6 Graph algorithms

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

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*.

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 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 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 an undirected graph, this induces a metric.

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.

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.

In what follows we will represent a graph in the following way:

```
>>> vertices = ['A','B','C','D','E']
>>> links = [(0,1),(1,2),(1,3),(2,5),(3,4),(3,2)]
>>> 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.

3.6.1 *Breadth first search*

The breadth-first search, also indicated with BFS, is an algorithm designed to visit all vertices in a connected graph. The algorithm begins at one vertex and then moves on. Its main feature is that it is explores the neighbors of the current vertex before moving on to explore remote vertices and their neighbors. It visit 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 a 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 3.5: in file: `numeric.py`

```
def breadth_first_search(graph,start):
    vertices, link = graph
    blacknodes = []
    graynodes = [start]
    neighbors = [[] for vertex in vertices]
```

```
    for link in links:
        neighbors[link[0]].append(link[1])
    while graynodes:
        current = graynodes.pop()
        for neighbor in neighbors[current]:
            if not neighbor in blacknodes+graynodes:
                graynodes.insert(0,neighbor)
        blacknodes.append(current)
    return blacknodes
```

The BFS algorithm scales as follows:

$$T_{best} \in \Theta(n_E + n_V) \tag{3.75}$$
$$T_{average} \in \Theta(n_E + n_V) \tag{3.76}$$
$$T_{worst} \in \Theta(n_E + n_V) \tag{3.77}$$
$$\tag{3.78}$$

3.6.2 *Depth first search*

This algorithm is very similar to the BFS but it takes the opposite approach and it explores as far as possible along each branch before backtracking.

Here is a possible implementation:

Listing 3.6: in file: `numeric.py`

```
def depth_first_search(graph,start):
    vertices, link = graph
    blacknodes = []
    graynodes = [start]
    neighbors = [[] for vertex in vertices]
    for link in links:
        neighbors[link[0]].append(link[1])
    while graynodes:
        current = graynodes.pop()
        for neighbor in neighbors[current]:
            if not neighbor in blacknodes+graynodes:
                graynodes.append(neighbor)
        blacknodes.append(current)
    return blacknodes
```

Notice that the BFS and the DFS differ for a single line which determines whether graynodes is a queue (BSF) 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 like:

$$T_{best} \in \Theta(n_E + n_V) \tag{3.79}$$
$$T_{average} \in \Theta(n_E + n_V) \tag{3.80}$$
$$T_{worst} \in \Theta(n_E + n_V) \tag{3.81}$$
$$T_{memory} \in \Theta(1) \tag{3.82}$$

3.6.3 *Disjoint Sets*

This is a data structure that can be used to store sets of sets and implements efficiently the join operation between tests. Each set of sets can be identified by a representative element. The algorithm starts by assuming as same disjoint sets as the initial number of elements n. Each is represented by itself. When two sets are joined the representative element of the latter is made 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 non-negative number j it means that it belongs to a set that was joined with the set containing j.

Here is the implementation:

Listing 3.7: in file: `numeric.py`

```
class DisjointSets(object):
    def __init__(self,n):
        self.sets = [-1]*n
        self.counter = n
    def parent(self,i):
        while True:
            j = self.sets[i]
            if j<0:
                return i
            i = j
    def join(self,i,j):
        i,j = self.parent(i),self.parent(j)
        if i!=j:
            self.sets[i] += self.sets[j]
```

```
            self.sets[j] = i
            self.counter-=1
            return True # they have been joined
        return False    # they were already joined
    def __len__(self):
        return self.counter
```

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

As an example of application, here is a code that build a n^d maze. It may 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, i.e. if there is a path that connects them. At the benning, 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, i.e. there is a path connecting any cell to any cell.

```
def make_maze(n,d):
    walls = [(i,i+n**j) for i in xrange(n**2) for j in xrange(d) if (i/n**j)%n+1<n]
    torn_down_walls = []
    ds = DisjointSets(n**d)
    random.shuffle(walls)
    for i,wall in enumerate(walls):
        if ds.join(wall[0],wall[1]):
            torn_down_walls.append(wall)
        if len(ds)==1:
            break
    walls = [wall for wall in walls if not wall in torn_down_walls]
    return walls, torn_down_walls
```

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

```
>>> walls, torn_down_walls = make_maze(n=20,d=2)
```

The figure below shows a representation of a generated maze.

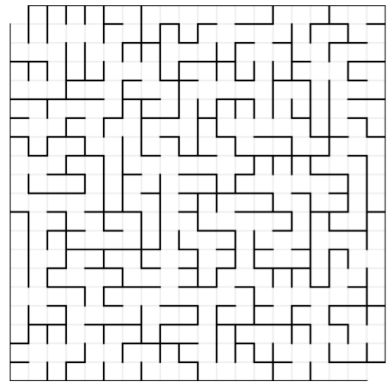

Figure 3.2: Example of a maze as generated using the `DisjointSets` algorithm.

3.6.4 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 there is only one path connecting each couple of vertices.

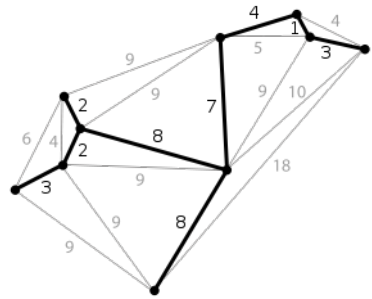

Figure 3.3: Example of a minimum spanning tree subgraph of a larger graph. The number on the links indicate their weight or length.

One algorithm to buld the minimal spanning tree of a graph is the Kruskal algorithm. It works by placing all vertices in a `DisjointSets` sturucture

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 mimimum spanning tree and the two sets are joined, else the links is ignored. The Kruskal algorithm assumes an undirected graph, i.e. all links are bi-directional and the weight of a link is the same in both directions.

Listing 3.8: in file: `numeric.py`

```
def Kruskal(graph):
    vertices, links = graph
    A = []
    S = DisjointSets(len(vertices))
    links.sort(cmp=lambda a,b: cmp(a[2],b[2]))
    for source,dest,length in links:
        if S.join(source,dest):
            A.append((source,dest,length))
    return A
```

The Kruskal algorithm goes like:

$$T_{worst} \in \Theta(n_E \log n_V) \tag{3.83}$$

$$T_{memory} \in \Theta(n_E) \tag{3.84}$$

We will provide an example of application in the next subsection.

3.6.5 Minimum spenning tree: Prim

The Prim algorithm solves the same problem as the Kruskal algorithm but does it also for a directed graph. It works by placing all vertices in a min priority queue were the queue metric for each vertex is the length of 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 here are vertices in the priority queue.

The Prim algorithm also differs from the Kruskal because the former needs

a starting vertex while the latter does not. The result when interpreted as a subgraph does not depend on the start vertex.

Listing 3.9: in file: `numeric.py`

```
class PrimVertex(object):
    INFINITY = 1e100
    def __init__(self,id,links):
        self.id = id
        self.closest = None
        self.closest_dist = PrimVertex.INFINITY
        self.neighbors = [link[1:] for link in links if link[0]==id]
    def __cmp__(self,other):
        return cmp(self.closest_dist, other.closest_dist)

def Prim(graph, start):
    from heapq import heappush, heappop, heapify
    vertices, links = graph
    P = [PrimVertex(i,links) for i in vertices]
    Q = [P[i] for i in vertices if not i==start]
    vertex = P[start]
    while Q:
        for neighbor_id,length in vertex.neighbors:
            neighbor = P[neighbor_id]
            if neighbor in Q and length<neighbor.closest_dist:
                 neighbor.closest = vertex
                 neighbor.closest_dist = length
        heapify(Q)
        vertex = heappop(Q)
    return [(v.id,v.closest.id,v.closest_dist) for v in P if not v.id==start]
```

```
>>> vertices = range(10)
>>> links = [(i,j,abs(math.sin(i+j+1))) for i in vertices for j in vertices]
>>> graph = [vertices,links]
>>> link = Prim(graph,0)
>>> for link in links: print(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 like:

$$T_{worst} \in \Theta(n_E + n_V \log n_V) \tag{3.85}$$

$$T_{memory} \in \Theta(n_E) \tag{3.86}$$

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 there are three genes 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 virus evolved in time. [11]

3.6.6 Single source shortest paths: Dijkstra

The Dijktra algorthm solves a similar problem to the Kruskal and Prim algorithms but not quite the same. Given a graph it computes, for each vertex, the shortest path connecting the vertex to a start (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 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 here are vertices in the priority queue.

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

Listing 3.10: in file: `numeric.py`

```
def Dijkstra(graph, start):
    from heapq import heappush, heappop, heapify
    vertices, links = graph
    P = [PrimVertex(i,links) for i in vertices]
    Q = [P[i] for i in vertices if not i==start]
    vertex = P[start]
    vertex.closest_dist = 0
    while Q:
        for neighbor_id,length in vertex.neighbors:
            neighbor = P[neighbor_id]
            dist = length+vertex.closest_dist
            if neighbor in Q and dist<neighbor.closest_dist:
                 neighbor.closest = vertex
                 neighbor.closest_dist = dist
        heapify(Q)
        vertex = heappop(Q)
    return [(v.id,v.closest.id,v.closest_dist) for v in P if not v.id==start]
```

Listing 3.11: in file: `numeric.py`

```
>>> vertices = range(10)
>>> links = [(i,j,abs(math.sin(i+j+1))) for i in vertices for j in vertices]
>>> graph = [vertices,links]
>>> links = Dijkstra(graph,0)
>>> for link in links: print(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 like:

$$T_{worst} \in \Theta(n_E + n_V \log n_V) \tag{3.87}$$

$$T_{memory} \in \Theta(n_E) \tag{3.88}$$

An application of the Dijkstra is in solving a maze such as the one built when discussing disjoint sets. In order 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 each link a length (1 because all links connect next neighbor cells).

```
>>> n,d = 4, 2
>>> walls, links = make_maze(n,d)
>>> symmetrized_links = [(i,j,1) for (i,j) in links]+[(j,i,1) for (i,j) in links]
>>> graph = [range(n*n),symmetrized_links]
>>> links = Dijkstra(graph,0)
>>> paths = dict((i,(j,d)) for (i,j,d) in links)
```

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 figure below shows a generated maze and the a reconstructed path connecting an arbitrary cell to the origin.

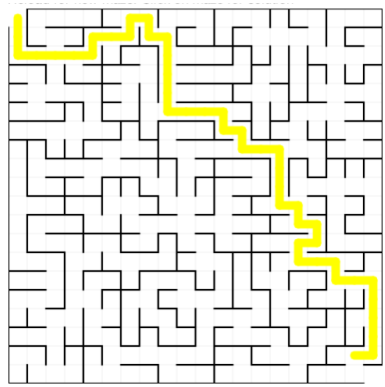

Figure 3.4: The result shows an application of the Dijkstra algorithm for the single source shortest path applied to solve a maze.

An interactive version of this algorithm implemented in Python with a Javascript interface can be found at `http://web2py.com/mazes`.

3.7 More on Greedy Algorithms

3.7.1 Huffman encoding

The *Shannon-Fano encoding* (also known as **minimal prefix code**) is 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 in a longer sequence of bits while characters that appear with more frequency are encoded with a shorter sequence.

The *Huffman Encoding* is an implementation of the Shannon-Fano encoding but the sequence of bits into which each character is mapped into is choose so that the length of the compressed string is minimal. This choice is constructed in the following way. We associate a tree to each character in the string to compress. Each tree is a trivial tree containing only one node: the root node. We then associate to the root node the frequency of the character representing the tree. We then extract from the list of trees the two trees with lowest frequency: t1 and t2. We form a new tree t3, we attach t1 and t2 to t3 and we associate a frequency to t3 equal to the sum of the frequencies of t1 and t2. We repeat this operation until the list of trees containing only one tree. At this point we associate a sequence of bits to 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 few bits, while rare characters are far from the root and encoded with more bits.

PKZIP, ARJ, ARC, JPEG, MPEG3 (mp3), MPEG4 and a other programs and 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 if combined with some form of cut of the Fourier spectrum (for example MP3 is an audio compression format where frequencies below 2KHz 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 the Huffman-Encoding:

Listing 3.12: in file: numeric.py

```
def encode_huffman(input):
    from heapq import heappush, heappop

    def inorder_tree_walk(t, key, keys):
        (f,ab) = t
        if isinstance(ab,tuple):
            inorder_tree_walk(ab[0],key+'0',keys)
            inorder_tree_walk(ab[1],key+'1',keys)
        else:
            keys[ab] = key

    symbols = {}
    for symbol in input:
        symbols[symbol] = symbols.get(symbol,0)+1
    heap = []
    for (k,f) in symbols.items():
        heappush(heap,(f,k))
    while len(heap)>1:
        (f1,k1) = heappop(heap)
        (f2,k2) = heappop(heap)
        heappush(heap,(f1+f2,((f1,k1),(f2,k2))))
    symbol_map = {}
    inorder_tree_walk(heap[0],'',symbol_map)
    encoded = ''.join(symbol_map[symbol] for symbol in input)
    return symbol_map, encoded

def decode_huffman(keys, encoded):
    reversed_map = dict((v,k) for (k,v) in keys.items())
    i, output = 0, []
    for j in range(1,len(encoded)+1):
        if encoded[i:j] in reversed_map:
           output.append(reversed_map[encoded[i:j]])
           i=j
    return ''.join(output)
```

We can use it as follows:

Listing 3.13: in file: numeric.py

```
>>> input = 'this is a nice day'
>>> keys, encoded = encode_huffman(input)
>>> print(encoded)
10111001110010001100100011110010101100110100000011111111110
>>> decoded = decode_huffman(keys,encoded)
>>> print(decoded == input)
True
>>> print(1.0*len(input)/(len(encoded)/8))
2.57...
```

We managed to compress the original data by a factor 2.57.

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

$$E = -\sum_u w_i \log_2 w_i \tag{3.89}$$

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

Listing 3.14: in file: `numeric.py`

```
>>> from math import log
>>> input = 'this is a nice day'
>>> w = [1.0*input.count(c)/len(input) for c in set(input)]
>>> E = -sum(wi*log(wi,2) for wi in w)
>>> print(E)
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 no.

Our choice of using characters as symbols is arbitrary. We cold have used 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 differs 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.

3.7.2 *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 sub-sequence, LCS, that is a sub-sequence of both S_1 and S_2.

There are several applications for the Longest Common Subsequence 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 typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.
- **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 longest common subsequence 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 a text contains a word, w, that is not in the dictionary, a "close" word (i.e. 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 above, 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.

On the other hand, suppose that, like the example above, the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed.

Finally, observe that once we've decided what to do with the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. Therefore we can solve it recursively.

Rather than finding the subsequence itself, it turns out to be more efficient to find directly 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 programing algorithm we get the following:

Listing 3.15: in file: `numeric.py`

```
def lcs(a, b):
    previous = [0]*len(a)
    for i,r in enumerate(a):
        current = []
        for j,c in enumerate(b):
            if r==c:
                e = previous[j-1]+1 if i*j>0 else 1
            else:
                e = max(previous[j] if i>0 else 0,
                        current[-1] if j>0 else 0)
            current.append(e)
        previous=current
    return current[-1]
```

Here is an example:

Listing 3.16: in file: `numeric.py`

```
>>> dna1 = 'ATGCTTTAGAGGATGCGTAGATAGCTAAATAGCTCGCTAGA'
>>> dna2 = 'GATAGGTACCACAATAATAAGGATAGCTCGCAAATCCTCGA'
>>> print(lcs(dna1,dna2))
26
```

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

3.7.3 *Needleman-Wunsch*

With some minor changes to the LCS algorithm we obtain the Needleman and Wunsch algorithm [12] which solves the problem of *glbal sequence alignment*. The changes are: instead of using only two alternating rows (`c` and `d` for storing the temporary results, we store all temporary results into an array `z`; when find two matching symbols between the two strings, 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 3.17: in file: `numeric.py`

```
def needleman_wunsch(a,b,p=0.97):
    z=[]
    for i,r in enumerate(a):
        z.append([])
        for j,c in enumerate(b):
            if r==c:
                e = z[i-1][j-1]+1 if i*j>0 else 1
            else:
                e = p*max(z[i-1][j] if i>0 else 0,
                          z[i][j-1] if j>0 else 0)
            z[-1].append(e)
    return 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:

Listing 3.18: in file: `numeric.py`

```
>>> bases = 'ATGC'
>>> from random import choice
>>> genes = [''.join(choice(bases) for k in range(10)) for i in range(20)]
>>> chromosome1 = ''.join(choice(genes) for i in range(10))
>>> chromosome2 = ''.join(choice(genes) for i in range(10))
>>> z = needleman_wunsch(chromosome1, chromosome2)
>>> color2d(title='Needleman-Wunsch', data=z,
...         filename='images/needleman.png')
```

The output of the algorithm is the following image:

The arrow-like patterns in the figure correspond to locations where `chromosome1` (Y coordinate) and where `chromosome2` (X coordinate) have DNA

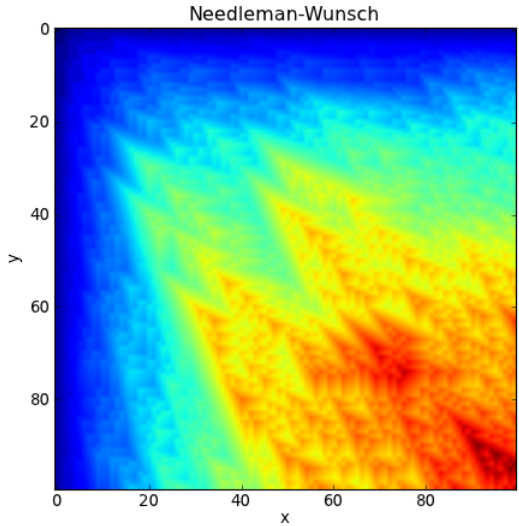

Figure 3.5: A Needleman and Wunsch plot sequence alignment. The arrow-like patters indicate the point in the two sequences (represented by the X and Y coordinates) where the two sequences are more likely to align.

in common. Those are the places where the sequences are more likely to be aligned for a more detailed comparison.

3.7.4 *Continuum Knapsack*

The continuum Knapsack problem can be formulated as the problem of maximizing:

$$f(x) = a_0x_0 + a_1x_1 + ... + a_nx_n \tag{3.90}$$

given the contraint

$$b_0x_0 + b_1x_1 + ... + b_nx_n \leq c \tag{3.91}$$

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 3.19: in file: `numeric.py`

```
def continuum_knapsack(a,b,c):
    table = [(a[i]/b[i],i) for i in range(len(a))]
    table.sort()
    table.reverse()
    f=0.0
    for (y,i) in table:
        quantity = min(c/b[i],1)
        x.append((i,quantity))
        c = c-b[i]*quantity
        f = f+a[i]*quantity
    return (f,x)
```

This algorithm is dominated by the sort therefore

$$T_{worst}(x) \in O(n \log n) \tag{3.92}$$

3.7.5 *Discrete Knapsack*

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

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 it cannot be solved in polynomial time.

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 re-formulate the problem in terms of simple capital budgeting problem. We have to invest \$5M. We assume ε =\$1M. We are in contact with 3 investment firms. Each of them 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

$$c[i,j] \leq c[i,j+1] \text{ and } r[i,j] \leq r[i,j+1] \tag{3.93}$$

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

Let's 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 multiple 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 (for instance, 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 not 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 on 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 on firm 0:

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

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

(3.95)

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

$$f[1,k] = \max_{j|c[1,j]<k} r[1,j] + f[0,k-c[0,j]] \tag{3.96}$$

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^*

(3.97)

STAGE THREE: we maximize the return of investment considering offers from firm 2 and the above table. We fill a table $f[2,k]$ with the maximum return of investment if we invest k million dollars on firm 0, firm 1 and firm 2:

$$f[2,k] = \max_{j|c[2,j]<k} r[2,j] + f[1,k-c[1,j]] \tag{3.98}$$

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^*

(3.99)

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

We can generalize eq.(3.96) and eq.(3.98) for any number of investment firms (decision stages):

$$f[i,k] = \max_{j|c[i,j]<k} r[i,j] + f[i-1,k-c[i-1,j]] \tag{3.100}$$

3.8 Long and infinite loops

3.8.1 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:

$$\text{if } \exists x \mid x \in NPC \text{ and } x \in P \Rightarrow \forall y \in NP, y \in P \tag{3.101}$$

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 which solve in polynomial time, may algorithms that currently do not have a polynomial time solution. Conversely is one could prove that NP is not equivalent to NPC, we would know that a polynomial time solution to NPC problems does not exist.

3.8.2 Cantor's argument

Cantor proved that the real numbers in any interval (for example in $[0,1)$) are more then the integer numbers, therefore real numbers are uncountable. The proof proceeds as follow:

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

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 2 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 3 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 4 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 5 & \longleftrightarrow & 0.xxxxxxxxxx... \\ ... & ... & ... \end{array} \tag{3.102}$$

   (here the x represent a decimal digits 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 3.102, the second decimal digit differs from the second decimal digit of the second real number of table 3.102 and so on and on for all the infinite decimal digits:

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.\overline{x}xxxxxxxxx... \\ 2 & \longleftrightarrow & 0.x\overline{x}xxxxxxxx... \\ 3 & \longleftrightarrow & 0.xx\overline{x}xxxxxxx... \\ 4 & \longleftrightarrow & 0.xxx\overline{x}xxxxxx... \\ 5 & \longleftrightarrow & 0.xxxx\overline{x}xxxxx... \\ ... & ... & ... \end{array} \tag{3.103}$$

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 for 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.

3.8.3 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. Since 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 the 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:

```
def verify(i):
    s=str(i*i)
    d=[]
    for c in s:
        if not c in d:
            d.append(c)
    if len(d)==2:
        return 1
    else:
        return 0

def next(i):
    i=long(i)
    while 1:
        i=i+2
        if verify(i):
            print(i, i*i)
            break

next(81619)
```

Nobody knows whether this code stops or not.

While 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.

A problem is 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:

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

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

We say that a f is well-conditioned in a domain D if $\text{cond}(f, x) < 1$ for every x in D. Conversely we say a problem is ill-conditioned if $\text{cond}(f, x) > 1$

A problem is stable if well-conditioned.

In this book we are mostly concerned with stable (well-conditioned) problems.

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, a tape measure with a resolution of 1mm we will only be able to determine the length of the nail within 1mm 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. 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 interment 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.

Now, let us consider a system that given an input x produces the output y. x and y are physical quantity 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.

There are various approximations we have to make:

- We must replace the "true" value for the input with our best estimate, $\bar{x}$, and its associated uncertainty δx.
- We must replace the "true" value for the output with our best estimate, $\bar{y}$, and its associated uncertainty δy.
- Even if we know the "true" function f describing the system, our implementation for the function will constitute an approximation, $\bar{f}$. As we have seen before we may not have a single approximation for f but a series of approximations f_n which become more accurate as n increases.

With the above 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: is due to the fact that often the computation of $\bar{f}(x) = \lim_{n\to\infty} f_n(x)$ is toocomputationally expensive and we must approximate $\bar{f}(x)$ with $f_n(x)$. This error can be estimated and controlled.

Systematic Error: is 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: is 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 $z = x - y$ is very close to zero then the uncertainty in z is still larger than in x and y. Also notice that if $z = x/y$ and y is small compared to x than 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.

4.2.2 buckingham

Buckingham is a Python library that implements error propagation and unit conversion. It defines a single class called `Number` and a number object has value, an uncertainty and a dimensionality (for example length, volume, mass, etc.). Here is an example:

```
>>> from buckingham import *
>>> globals().update(allunits())
>>> L = (4 + pm(0.5)) * meter
>>> v = 5 * meter/second
>>> t = L/v
>>> print(t

    )
(8.00 +/- 1.00)/10
>>> print(time.convert('kilometer/hour'))
>>> print(time.convert('lightyear/week'))
```

Notice how adding an an uncertainty to a numeric value with `+ pm(...)` or adding units to a numeric value (integer or floating point) transforms the number into a `Number` object. A `Number` object behaves like a floating point but propagates its uncertainty and its units. Internally all units are converted to the International System unless an explicit conversion is specified.

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 non-linear with linear + corrections
- Transform a problem into a different one
- Approach the true result by iterations

Here are some example 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 with out instruments and n is 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

```
def f(x):
    (s,t) = (1.0,1.0)
    for i in range(1,10): (s, t) = (s+t, t*x/i)
    return s
```

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

The most common ways to define a derivative are:

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

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

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

$$= \lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2h} \tag{4.10}$$

If the function is differentiable in x then the three definitions are equivalent.

If the limit exists then it means that:

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

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

up to corrections of order h. The three definitions are equivalent for functions that are differentiable in x and the latter is preferable because it is more symmetric.

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 $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$:

Listing 4.1: in file: `numeric.py`

```
def D(f,h=1e-6): # first derivative of f
    return lambda x,f=f,h=h: (f(x+h)-f(x-h))/2/h
```

We can do the same with the second derivative:

$$\frac{\mathrm{d}^2f(x)}{\mathrm{d}x^2} = \frac{f(x+h)-2f(x)-f(x-h)}{h^2}+O(h) \tag{4.12}$$

Listing 4.2: in file: `numeric.py`

```
def DD(f,h=1e-6): # second derivative of f
    return lambda x,f=f,h=h: (f(x+h)-2.0*f(x)+f(x-h))/(h*h)
```

Here is an example:

Listing 4.3: in file: `numeric.py`

```
>>> def f(x): return x*x-5.0*x
>>> print(f(0))
0.0
>>> f1 = D(f) # first derivative
>>> print(f1(0))
-5.0
>>> f2 = DD(f) # second derivative
>>> print(f2(0))
2.00000...
>>> f2 = D(f1) # second derivative
>>> print(f2(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 non-linear with linear*

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

```
>>> from math import sin
>>> points = [0.01*i for i in range(0,11)]
>>> for x in points:
...     print(x, sin(x), "%.2f" % (abs(x-sin(x))/sin(x)*100))
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 1st order of the Taylor expansion is linear. For values 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:

```
>>> from math import sin
>>> points = [0.01*i for i in range(0,11)]
>>> for x in points:
...     s = x - x*x*x/6
...     print(x, math.sin(x), s, ``%.6f'' % (abs(s-sin(x))/(sin(x))*100))
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 10000 (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 1st order with the 3rd 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 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

$$\sin(x) = -\sin(-x) \tag{4.13}$$

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

$$\sin(x) = \sin(x - 2k\pi) \qquad k \in \mathbb{N} \tag{4.14}$$

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

$$\sin(x) = -\sin(2\pi - x) \tag{4.15}$$

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

$$\sin(x) = \sin(\pi - x) \tag{4.16}$$

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

$$\sin(x) = \sqrt{1 - \sin(\pi/2 - x)^2} \tag{4.17}$$

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*

$\sin(x) \simeq x$ and $\sin(x) \simeq x - x^3/6$ came from respectively linearizing the function $\sin(x)$ and adding a correction to the previous approximation. 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:

```
def mysin(x):
    (s,t) = (0.0,x)
    for i in range(3,10,2): (s, t) = (s+t, -t*x*x/i/(i-1))
    return s
```

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

4.3.6 *Taylor Series*

A function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ 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:

$$f(x) = f(\bar{x}) + f^{(0)}(\bar{x})(x - \bar{x}) + ... + \frac{f^{(k)}(\bar{x})}{n!}(x - \bar{x})^k + R_k \tag{4.18}$$

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

$$R_k = \frac{f^{(k+1)}(\xi)}{(k+1)!}(x - \bar{x})^{k+1} \tag{4.19}$$

were 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

$$|R_k| < \left| max_{x \in D} f^{(k+1)} \right| |(x - \bar{x})^{k+1}| \tag{4.20}$$

If we stop the Taylor expansion at a finite value of k, the above formula gives us the Statistical Error part of the Computational Error.

Some Taylor series are very easy to compute:

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

$$
\begin{aligned}
f(x) &= e^x && (4.21)\\
f^{(1)}(x) &= e^x && (4.22)\\
... &= ... && (4.23)\\
f^{(k)}(x) &= e^x && (4.24)\\
e^x &= 1 + x + \frac{1}{2}x + ... + \frac{1}{k!}x^k + ... && (4.25)
\end{aligned}
$$

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

$$
\begin{aligned}
f(x) &= sin(x) && (4.26)\\
f^{(1)}(x) &= cos(x) && (4.27)\\
f^{(2)}(x) &= -sin(x) && (4.28)\\
f^{(3)}(x) &= -cos(x) && (4.29)\\
... &= ... && (4.30)\\
sin(x) &= x - \frac{1}{3!}x^3 + ... + \frac{(-1)^n}{(2k+1)!}x^{(2k+1)} + ... && (4.31)
\end{aligned}
$$

We can show the effects of the various terms:

Listing 4.4: in file: `numeric.py`

```
>>> X = [0.03*i for i in xrange(200)]
>>> Y = {'label':'sin(x)','data':[(x,math.sin(x)) for x in X]}
>>> Y1 = {'label':'Taylor 1st','data':[(x,x) for x in X[:100]]}
>>> Y2 = {'label':'Taylor 3rd','data':[(x,x-x**3/6) for x in X[:100]]}
>>> Y3 = {'label':'Taylor 5th','data':[(x,x-x**3/6+x**5/120) for x in X[:100]]}
>>> draw(title='sin(x) approximations',filename='images/sin.png',linesets=[Y,Y1,Y2,
    Y3])
```

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

$$sin(x) = 1 - \frac{1}{2}(x - \frac{\pi}{2})^2 + ... + \frac{(-1)^n}{(2k)!}(x - \frac{\pi}{2})^{(2k)} + ... \quad (4.32)$$

and a plot would show:

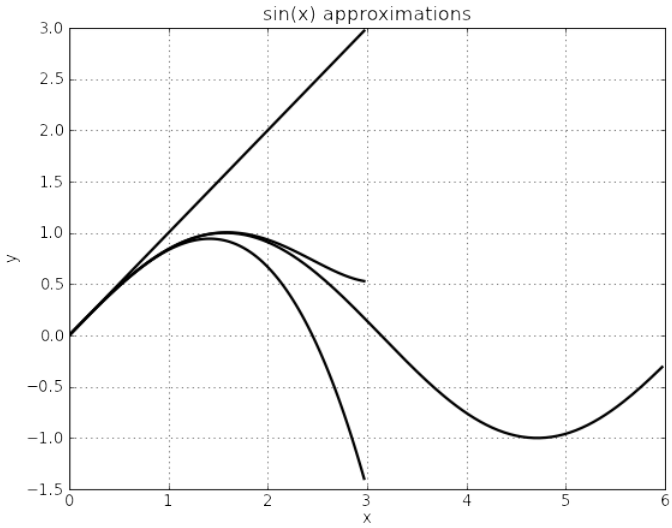

Figure 4.1: The figure shows the sin function and its approximation using the Taylor expansion around $x = 0$ at different orders.

Listing 4.5: in file: `numeric.py`

```
>>> a = math.pi/2
>>> X = [0.03*i for i in xrange(200)]
>>> Y = {'label':'sin(x)','data':[(x,math.sin(x)) for x in X]}
>>> Y1 = {'label':'Taylor 2nd','data':[(x,1-(x-a)**2/2) for x in X[:150]]}
>>> Y2 = {'label':'Taylor 4th','data':[(x,1-(x-a)**2/2+(x-a)**4/24) for x in X
     [:150]]}
>>> Y3 = {'label':'Taylor 6th','data':[(x,1-(x-a)**2/2+(x-a)**4/24-(x-a)**6/720)
     for x in X[:150]]}
>>> draw(title='sin(x) approximations',filename='images/sin2.png',linesets=[Y,Y1,Y2
     ,Y2])
```

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, $\sin(x) = \cos(x - \pi/2)$:

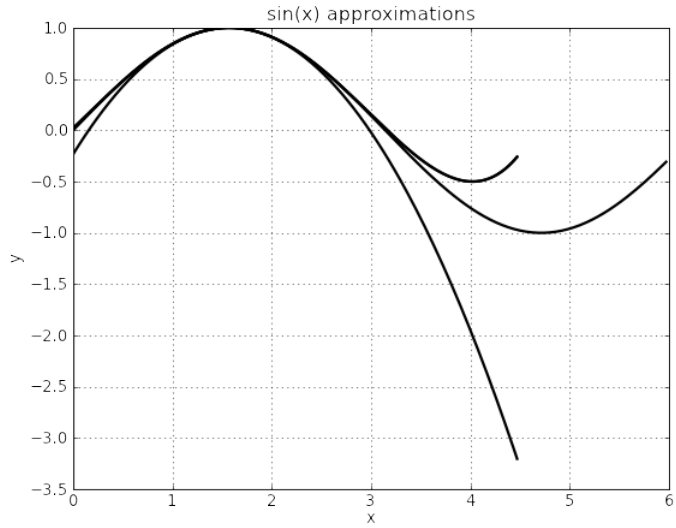

Figure 4.2: The figure shows the sin function and its approximation using the Taylor expansion around $x = \pi/2$ at different orders.

$$
\begin{aligned}
f(x) &= cos(x) && (4.33)\\
f^{(1)}(x) &= -sin(x) && (4.34)\\
f^{(2)}(x) &= cos(x) && (4.35)\\
f^{(3)}(x) &= sin(x) && (4.36)\\
... &= ... && (4.37)\\
cos(x) &= 1 - \frac{1}{2}x^2 + ... + \frac{(-1)^n}{(2k)!}x^{(2k)} + ... && (4.38)
\end{aligned}
$$

With a simple replacement it is easy to prove that

$$e^{ix} = \cos(x) + i\sin(x) \tag{4.39}$$

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, for example. It can

be re-arranged as a function of the previous $(k-1)-th$ term:

$$T_k(x) = \frac{1}{k!}x^n = \frac{x}{k}\frac{1}{(k-1)!}x^{k-1} = \frac{x}{k}T_{k-1}(x) \tag{4.40}$$

Moreover All terms are positive and monotonically increasing. Therefore, $x < 0$, $R_n < T_{k+1}(1)$. This allows for an easy implementation of the Taylor expansion of the stopping condition:

Listing 4.6: in file: `numeric.py`

```
def myexp(x,precision=1e-6,max_steps=40):
    if x==0:
       return 1.0
    elif x>0:
       return 1.0/myexp(-x,precision,max_steps)
    else:
       t = s = 1.0 # first term
       for k in range(1,max_steps):
           t = t*x/k   # next term
           s = s + t   # add next term
           if abs(t)<precision: return s
       raise ArithmeticError, '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.
- Detects failure to converge.

Here is a test of its convergence:

Listing 4.7: in file: `numeric.py`

```
>>> for i in range(10):
...     x= 0.1*i
...     assert abs(myexp(x) - math.exp(x)) < 1e-4
```

We can do the same for the sin function.

$$T_k(x) = -\frac{x^2}{(2k)(2k+1)}T_{k-1}(x) \tag{4.41}$$

In this case the residue is always limited by

$$|R_k| < |x^{2k+1}| \tag{4.42}$$

because the derivatives of sin are always sin and cos and their image is always between [-1,1]. Hence we write:

Listing 4.8: in file: `numeric.py`

```
def mysin(x,precision=1e-6,max_steps=40):
    pi = math.pi
    if x==0:
       return 0
    elif x<0:
       return -mysin(-x)
    elif x>2.0*pi:
       return mysin(x % (2.0*pi))
    elif x>pi:
       return -mysin(2.0*pi - x)
    elif x>pi/2:
       return mysin(pi-x)
    elif x>pi/4:
       return sqrt(1.0-mysin(pi/2-x)**2)
    else:
       t = s = x                    # first term
       for k in range(1,max_steps):
           t = t*(-1.0)*x*x/(2*k)/(2*k+1)   # next term
           s = s + t                # add next term
           r = x**(2*k+1)           # estimate residue
           if r<precision: return s  # stopping condition
       raise ArithmeticError, 'no convergence'
```

Notice that for the stopping condition to be true x must be less than one. We therefore added many constraints to use trigonometric properties to reduce the problem to $x < \pi/4 < 1$.

Here we test it:

Listing 4.9: in file: `numeric.py`

```
>>> for i in range(10):
...     x= 0.1*i
...     assert abs(mysin(x) - math.sin(x)) < 1e-4
```

Finally we can do the same for the cos function:

Listing 4.10: in file: `numeric.py`

```
def mycos(x,precision=1e-6,max_steps=40):
    pi = math.pi
    if x==0:
       return 1.0
    elif x<0:
       return mycos(-x)
    elif x>2.0*pi:
       return mycos(x % (2.0*pi))
    elif x>pi:
       return mycos(2.0*pi - x)
    elif x>pi/2:
       return -mycos(pi-x)
    elif x>pi/4:
       return sqrt(1.0-mycos(pi/2-x)**2)
    else:
       t = s = 1                    # first term
       for k in range(1,max_steps):
           t = t*(-1.0)*x*x/(2*k)/(2*k-1)   # next term
           s = s + t                # add next term
           r = x**(2*k)             # estimate residue
           if r<precision: return s  # stopping condition
       raise ArithmeticError, 'no convergence'
```

Here is a test of convergence:

Listing 4.11: in file: `numeric.py`

```
>>> for i in range(10):
...     x = 0.1*i
...     assert abs(mycos(x) - math.cos(x)) < 1e-4
```

4.3.7 *Stopping Conditions*

In order to implement a stopping condition we have two options. We can look at the absolute error defined as:

$$[\text{absolute error}] = [\text{approximate value}] - [\text{true value}] \tag{4.43}$$

or we can look at the relative error:

$$[\text{relative error}] = [\text{absolute error}]/[\text{true value}] \tag{4.44}$$

or better we can consider both. Here is an example of pseudocode:

```
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
```

or in more compact and Pythonic notation:

```
def generic_looping_function(guess):
    result = guess
    for k in range(ns):
        correction = ...
        result = result+correction
        remainder = ...
        if norm(remainder) < max(ap,norm(resut)*rp): return result
    raise ArithmeticError, "no convergence"
```

In the code above:

- `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 zero or zero, 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 correct thing to do.

Since in general we do not known 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 will 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 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 mathemtics a system described by a function f is linear if it is additive:

$$f(x+y) = f(x) + f(y) \tag{4.45}$$

and if it is homogeneous:

$$f(\alpha x) = \alpha f(x) \tag{4.46}$$

In simpler words we can say that the output is proportional to the input.

As discussed in the introduction to this chapter one and 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 study a new unknown system, approximating the system with a

linear system is often the first step for 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 above means.

Given

$$\mathbf{x} \equiv \begin{pmatrix} x_0 \\ x_1 \\ ... \\ y_{n-1} \end{pmatrix} \qquad \mathbf{y} \equiv \begin{pmatrix} y_0 \\ y_1 \\ ... \\ y_{m-1} \end{pmatrix} \tag{4.47}$$

$$A \equiv \begin{pmatrix} a_{00} & a_{01} & ... & a_{0,n-1} \\ a_{10} & a_{11} & ... & a_{1,n-1} \\ ... & ... & ... & ... \\ a_{m-1,0} & a_{m-1,1} & ... & a_{m-1,n-1} \end{pmatrix} \tag{4.48}$$

Than the following equation means:

$$\mathbf{y} = f(\mathbf{x}) \simeq A\mathbf{x} \tag{4.49}$$

means

$$y_0 = f_0((x) \simeq a_{00}x_0 + a_{01}x_1 + ... + a_{0,n-1}x_{n-1} \tag{4.50}$$

$$y_1 = f_1((x) \simeq a_{10}x_0 + a_{11}x_1 + ... + a_{1,n-1}x_{n-1} \tag{4.51}$$

$$y_2 = f_2((x) \simeq a_{20}x_0 + a_{21}x_1 + ... + a_{2,n-1}x_{n-1} \tag{4.52}$$

$$... = ... \tag{4.53}$$

$$y_{m-1} = f_{m-1}((x) \simeq a_{m-1,0}x_0 + a_{m-1,1}x_1 + ...a_{m-1,n-1}x_{n-1} \tag{4.54}$$

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 above turn to be exact and can be replaced by $=$ symbols.

As a corollary of the basic properties of a linear system discussed above, linear systems have one nice additional property. If we combine two linear systems $y = Ax$ and $z = By$, the new system is also a linear system $z = (BA)z$.

Elementary Algebra is defined as a set of numbers (real numbers for example) 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 Python. In Python we can implement a matrix as a list of lists as follows:

```
>>> A = [[1,2,3],[4,5,6],[7,8,9]]
```

But such an object (list of list) does not have the mathematical properties we want so we have to define them.

First, we define a class representing a matrix:

Listing 4.12: in file: `numeric.py`

```
class Matrix(object):
    def __init__(self,rows=1,cols=1,fill=0.0):
        """
        Constructor a zero matrix
        Parameters
        - rows: the integer number of rows
        - cols: the integer number of columns
        - fill: the value or callable to be used to fill the matrix
        """
        self.rows = rows
        self.cols = cols
        if callable(fill):
            self.values = [fill(r,c) for r in xrange(rows) for c in xrange(cols)]
        else:
            self.values = [fill for r in xrange(rows) for c in xrange(cols)]
```

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.

The actual matrix elements are stored as a list or array into the `data` member variable. If `optimize=True` the data is stored into an `array` of double precision Floating Point numbers ("d"). This optimization will prevent you from building matrices of complex numbers or matrices of arbitrary precision decimal numbers.

Now we define a getter method, a setter method, and a string representation for the matrix elements:

Listing 4.13: in file: `numeric.py`

```
    def __getitem__(A,(i,j)):
        return A.values[i*A.cols+j]

    def __setitem__(A,(i,j),value):
        A.values[i*A.cols+j] = value

    def row(A,i):
        return Matrix(A.cols,1,fill=lambda r,c: A[i,c])

    def col(A,i):
        return Matrix(A.rows,1,fill=lambda r,c: A[r,i])

    def as_list(A):
        return [[A[i,j] for j in xrange(A.cols)] for i in xrange(A.rows)]

    def __str__(A):
        return str(A.as_list())
```

We also define two factory methods to create an identity matrix and a diagonal matrix (given a list of diagonal elements):

Listing 4.14: in file: `numeric.py`

```
    @staticmethod
    def identity(rows=1,one=1.0,fill=0.0):
        """
        Constuctor a diagonal matrix
        Parameters
        - rows: the integer number of rows (also number of columns)
        - fill: the value to be used to fill the matrix
        - one: the value in the diagonal
```

```
        """
        M = Matrix(rows,rows,fill)
        for i in xrange(rows): M[i,i] = one
        return M

    @staticmethod
    def diagonal(d):
        M = Matrix(len(d),len(d))
        for i,e in enumerate(d): M[i,i] = e
        return M
```

Now we define a static factory method to create a matrix from a list of lists (static because it is a class method, not an object method, it does not act on an existing matrix, it creates a new one):

Listing 4.15: in file: `numeric.py`

```
    @staticmethod
    def from_list(v):
        "builds a matrix from a list of lists"
        return Matrix(len(v),len(v[0]),fill=lambda r,c: v[r][c])
```

Now we are ready to define arithmetic operations among matrices. We start with addition and subtraction:

Listing 4.16: in file: `numeric.py`

```
    def __add__(A,B):
        """
        Adds A and B element by element, A and B must have the same size
        Example
        >>> A = Matrix.from_list([[4,3.0], [2,1.0]])
        >>> B = Matrix.from_list([[1,2.0], [3,4.0]])
        >>> C = A + B
        >>> print(C)
        [[5, 5.0], [5, 5.0]]
        """
        n, m = A.rows, A.cols
        if not isinstance(B,Matrix):
            if n==m:
                B = Matrix.identity(n,B)
            elif n==1 or m==1:
                B = Matrix(n,m,fill=B)
        if B.rows!=n or B.cols!=m:
            raise ArithmeticError, "Incompatible dimensions"
        C = Matrix(n,m)
        for r in xrange(n):
```

```
            for c in xrange(m):
                C[r,c] = A[r,c]+B[r,c]
        return C

    def __sub__(A,B):
        """
        Adds A and B element by element, A and B must have the same size
        Example
        >>> A = Matrix.from_list([[4.0,3.0], [2.0,1.0]])
        >>> B = Matrix.from_list([[1.0,2.0], [3.0,4.0]])
        >>> C = A - B
        >>> print(C)
        [[3.0, 1.0], [-1.0, -3.0]]
        """
        n, m = A.rows, A.cols
        if not isinstance(B,Matrix):
            if n==m:
                B = Matrix.identity(n,B)
            elif n==1 or m==1:
                B = Matrix(n,m,fill=B)
        if B.rows!=n or B.cols!=m:
            raise ArithmeticError, "Incompatible dimensions"
        C = Matrix(n,m)
        for r in xrange(n):
            for c in xrange(m):
                C[r,c] = A[r,c]-B[r,c]
        return C
    def __radd__(A,B): #B+A
        return A+B
    def __rsub__(A,B): #B-A
        return (-A)+B
    def __neg__(A):
        return Matrix(A.rows,A.cols,fill=lambda r,c:-A[r,c])
```

With the above definitions we can add matrices to matrices, subtract matrices from matrices but also add/subtract scalars to/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:

Listing 4.17: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1.0,2.0],[3.0,4.0]])
>>> print(A + A)      # calls A.__add__(A)
[[2.0, 4.0], [6.0, 8.0]]
>>> print(A + 2)      # calls A.__add__(2)
[[3.0, 2.0], [3.0, 6.0]]
```

```
>>> print(A - 1)      # calls A.__add__(1)
[[0.0, 2.0], [3.0, 3.0]]
>>> print(-A)         # calls A.__neg__()
[[-1.0, -2.0], [-3.0, -4.0]]
>>> print(5 - A)      # calls A.__rsub__(5)
[[4.0, -2.0], [-3.0, 1.0]]
>>> b = Matrix.from_list([[1.0],[2.0],[3.0]])
>>> print(b + 2)      # calls b.__add__(2)
[[3.0], [4.0], [5.0]]
```

The class Matrix works with complex numbers as well:

Listing 4.18: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1,2],[3,4]])
>>> print(A + 1j)
[[(1+1j), 2.0], [3.0, (4+1j)]]
```

Now we to implement multiplication. We are interested in three types of multiplications, multiplication of a scalar by a Matrix (`__rmul__`), multiplication of a Matrix by a Matrix (`__mul__`) and scalar product between two vectors (also handled by `__mul__`):

Listing 4.19: in file: `numeric.py`

```
    def __rmul__(A,x):
        "multiplies a number of matrix A by a scalar number x"
        import copy
        M = copy.deepcopy(A)
        for r in xrange(M.rows):
            for c in xrange(M.cols):
                 M[r,c] *= x
        return M

    def __mul__(A,B):
        "multiplies a number of matrix A by another matrix B"
        if isinstance(B,(list,tuple)):
            return (A*Matrix(len(B),1,fill=lambda r,c:B[r])).rows
        elif not isinstance(B,Matrix):
            return B*A
        elif A.cols == 1 and B.cols==1 and A.rows == B.rows:
            # try a scalar product ;-)
            return sum(A[r,0]*B[r,0] for r in xrange(A.rows))
        elif A.cols!=B.rows:
            raise ArithmeticError, "incompatible dimensions"
        M = Matrix(A.rows,B.cols)
        for r in xrange(A.rows):
```

```
            for c in xrange(B.cols):
                for k in xrange(A.cols):
                    M[r,c] += A[r,k]*B[k,c]
        return M
```

This allows us the following operations:

Listing 4.20: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1.0,2.0],[3.0,4.0]])
>>> print(2*A)       # scalar * matrix
[[2.0, 4.0], [6.0, 8.0]]
>>> print(A*A)       # matrix * matrix
[[7.0, 10.0], [15.0, 22.0]]
>>> b = Matrix.from_list([[1],[2],[3]])
>>> print(b*b)       # scalar product
14
```

4.4.2 *Examples of linear transformations*

In this section we want to provide an understanding of what a linear transformation in 2D is about (and things are similar in more than 2D).

In the code below we consider an image (a set of points) containing a circle and two orthogonal axis. 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 of 0.5 radiants.
- B_2 which is not a scaling, nor a rotation. As it can be seen from the image, it does not preserve angles.

Listing 4.21: in file: `numeric.py`

```
>>> points = [(math.cos(0.0628*t),math.sin(0.0628*t)) for t in range(200)]
>>> points += [(0.02*t,0) for t in range(50)]
>>> points += [(0,0.02*t) for t in range(50)]
>>> draw(title='Linear Transformation',xlab='x',ylab='y',filename = 'la1.png',
...      ellisets = [{'data':points}], xrange=(-1,1), yrange=(-1,1))
```

```
>>> def f(A,points,filename):
...       data = [(A[0,0]*x+A[0,1]*y,A[1,0]*x+A[1,1]*y) for (x,y) in points]
...       draw(title='Linear Transformation',xlab='x',ylab='y',filename=filename,
...            ellisets = [{'data':points},{'data':data}])
>>> A1 = Matrix.from_list([[0.2,0],[0,1]])
>>> f(A1, points, 'la2.png')
>>> A2 = Matrix.from_list([[1,0],[0,0.2]])
>>> f(A2, points, 'la3.png')
>>> S = Matrix.from_list([[0.3,0],[0,0.3]])
>>> f(S, points, 'la4.png')
>>> s, c = math.sin(0.5), math.cos(0.5)
>>> R = Matrix.from_list([[c,-s],[s,c]])
>>> B1 = R*A1
>>> f(B1, points, 'la5.png')
>>> B2 = Matrix.from_list([[0.2,0.4],[0.5,0.3]])
>>> f(B2, points, 'la6.png')
```

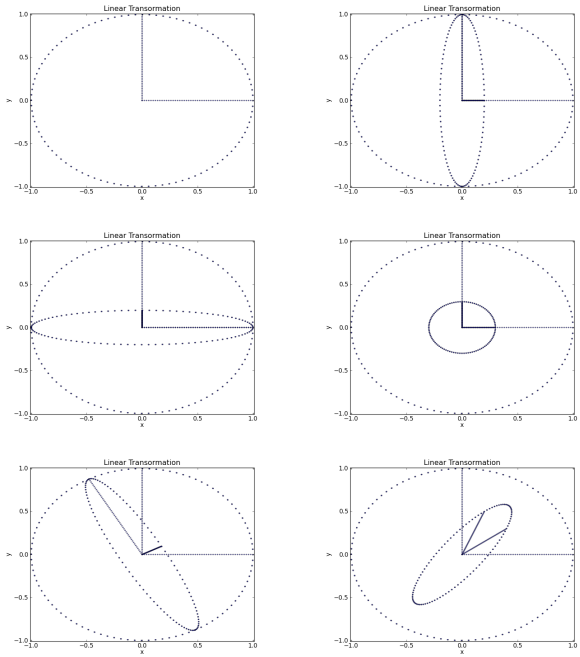

Figure 4.3: Example of effect of different linear transformation on the same set of points. From left-right, top-bottom they show streching along both the X and Y axes, scaling across both axes, a rotation, and a generic transformation which does not preserves angles.

4.4.3 *Matrix inversion and Guass-Jordan algorithm*

Implementing the inverse of the multiplication (i.e. 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.

In order to implement it we must first understand how it works. Consider the following equation:

$$A\mathbf{x} = \mathbf{b} \tag{4.55}$$

we can rewrite it as:

$$A\mathbf{x} = B\mathbf{b} \tag{4.56}$$

where $B = 1$ is the identity matrix. This equation remains true if we multiply both terms by a non singular matrix S_0:

$$S_0 A\mathbf{x} = S_0 B\mathbf{b} \tag{4.57}$$

The trick of the Guass-Jordan elimintation consists of finding a series of matrices S_0, S_1, ...S_{n-1} so that

$$S_{n-1}...S_1 S_0 A\mathbf{x} = S_{n-1}...S_1 S_0 B\mathbf{b} = \mathbf{x} \tag{4.58}$$

Because the above expression must be true for every b and because x is the solution of $Ax = b$, then by definition $S_{n-1}...S_1 S_0 B \equiv A^{-1}$.

Here is an algorithm that, given A, computes A^{-1}

Listing 4.22: in file: `numeric.py`

```
    def __rdiv__(A,x):
        """Computes x/A using Gauss-Jordan elimination where x is a scalar"""
        import copy
        n = A.cols
        if A.rows != n:
           raise ArithmeticError, "matrix not squared"
        indexes = range(n)
        A = copy.deepcopy(A)
        B = Matrix.identity(n,x)
        for c in indexes:
            for r in xrange(c+1,n):
                if abs(A[r,c])>abs(A[c,c]):
                    A.swap_rows(r,c)
                    B.swap_rows(r,c)
            p = 0.0 + A[c,c] # trick to make sure it is not integer
            for k in indexes:
                A[c,k] = A[c,k]/p
                B[c,k] = B[c,k]/p
            for r in range(0,c)+range(c+1,n):
                p = 0.0 + A[r,c] # trick to make sure it is not integer
                for k in indexes:
                    A[r,k] -= A[c,k]*p
                    B[r,k] -= B[c,k]*p
            # if DEBUG: print(A, B)
        return B

```

```
    def __div__(A,B):
        if isinstance(B,Matrix):
            return A*(1.0/B) # matrix/marix
        else:
            return (1.0/B)*A # matrix/scalar

    def swap_rows(A,i,j):
        for c in xrange(A.cols):
            A[i,c],A[j,c] = A[j,c],A[i,c]
```

Here is an example and we will see many more applications later.

Listing 4.23: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1,2],[4,9]])
>>> print(1/A)
[[9.0, -2.0], [-4.0, 1.0]]
>>> print(A/A)
[[1.0, 0.0], [0.0, 1.0]]
>>> print(A/2)
[[0.5, 1.0], [2.0, 4.5]]
```

4.4.4 *Transposing a matrix*

Another operation that we will need is transposition:

Listing 4.24: in file: `numeric.py`

```
    @property
    def t(A):
        """Transposed of A"""
        return Matrix(A.cols,A.rows, fill=lambda r,c: A[c,r])
```

which we can use as follows:

Listing 4.25: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1,2],[3,4]])
>>> print(A.t)
[[1, 3], [2, 4]]
```

For later use, we define two functions to check whether a matrix is symmetrical or zero within a given precision.

Listing 4.26: in file: `numeric.py`

```
def is_almost_symmetric(A, ap=1e-6, rp=1e-4):
    if A.rows != A.cols: return False
    for r in xrange(A.rows):
        for c in xrange(r):
            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

def is_almost_zero(A, ap=1e-6, rp=1e-4):
    for r in xrange(A.rows):
        for c in xrange(A.cols):
            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 also fundamental for solving systems of linear equations such as the following:

$$x_0 + 2x_1 + 2x_2 = \quad 3 \tag{4.59}$$
$$4x_0 + 4x_1 + 2x_2 = \quad 6 \tag{4.60}$$
$$4x_0 + 6x_1 + 4x_2 = \quad 10 \tag{4.61}$$

This can be rewritten using the equivalent linear algebra notation:

$$A\mathrm{x} = \mathrm{b} \tag{4.62}$$

where

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 4 & 4 & 2 \\ 4 & 6 & 4 \end{pmatrix} \quad \textit{and} \quad b = \begin{pmatrix} 3 \\ 6 \\ 10 \end{pmatrix} \tag{4.63}$$

The solution of the equaton can now be written as:

$$\mathrm{x} = A^{-1}\mathrm{b} \tag{4.64}$$

We can easily solve the system with our Python library:

Listing 4.27: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1,2,2],[4,4,2],[4,6,4]])
>>> b = Matrix.from_list([[3],[6],[10]])
>>> x = (1/A)*b
>>> print(x)
[[-1.0], [3.0], [-1.0]]
```

Notice that b is a column vector and therefore

```
>>> b = Matrix.from_list([[3],[6],[10]])
```

but not

```
>>> b = Matrix.from_list([[3,6,10]]) # wrong
```

We can also obtain a column vector by performing a transposition of a row vector:

```
>>> b = Matrix.from_list([[3,6,10]]).t
```

4.4.6 *Norm and condition number again*

By norm of a vector we often refer to the 2-norm defined using Pythagora's Theorem:

$$||x||_2 = \sqrt{\sum_i x_i^2} \tag{4.65}$$

For a vector we can define the p-norm as a generalization of the 2 norm:

$$||x||_p \equiv \left(\sum_i abs(x_i)^p\right)^{\frac{1}{p}} \tag{4.66}$$

We can extend the notation of a norm to any function that maps a vector into a vector as follows:

$$||f||_p \equiv \max_x ||f(x)||_p / ||x||_p \tag{4.67}$$

An immediate application is to fuctions implemented as linear transformations:

$$||A||_p \equiv \max_x ||Ax||_p / ||x||_p \tag{4.68}$$

This can be difficult to compute in the general case but it reduces to a simple formula for the 1-norm:

$$||A||_1 \equiv \max_j \sum_i abs(A_{ij}) \tag{4.69}$$

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 4.28: in file: `numeric.py`

```
def norm(A,p=1):
    if isinstance(A,(list,tuple)):
        return sum(abs(x)**p for x in A)**(1.0/p)
    elif isinstance(A,Matrix):
        if A.rows==1 or A.cols==1:
            return sum(norm(A[r,c])**p \
               for r in xrange(A.rows) \
               for c in xrange(A.cols))**(1.0/p)
        elif p==1:
            return max([sum(norm(A[r,c]) \
               for r in xrange(A.rows)) \
               for c in xrange(A.cols)])
        else:
            raise NotImplementedError
    else:
        return abs(A)
```

This allows us to extend the concept of a condition number, defined in the first chapter as functions of multiple variables, expanded in Taylor series:

$$\mathbf{y} = f(\mathbf{x}) \simeq f((x)_0) + J_f(\mathbf{x} - \mathbf{x}_0) + O(\mathbf{x} - \mathbf{x}_0)^2 \tag{4.70}$$

where $\mathbf{x}_0$ is the point around which the Taylor expansion is performed, $f(\mathbf{x}_0)$ is the 0-th order in the Taylor expansion and $J_{f,ij} = \partial f_i(\mathbf{x})/\partial x_j|_{\mathbf{x}_0}$ is the Jacobian of the function f, also the 1-th order in the Taylor expansion.

Applying the definition of condition number we obtain:

$$\text{condition number of } f \text{ in } x_0 \equiv ||J_f||_p * ||J_f^{-1}||_p \tag{4.71}$$

which allows us to compute the condition number for any function of multiple variables. This is also called the condition number of the matrix J representing a linear transformation.

Now we can implement a general function to compute the condition number of a function f even if the transformation is represented by a linear transformation $f \propto J$:

Listing 4.29: in file: `numeric.py`

```
def condition_number(f,x=None,h=1e-6):
    if callable(f) and not x is None:
        return D(f,h)(x)*x/f(x)
    elif isinstance(f,Matrix): # if is the Matrix J
        return norm(f)*norm(1/f)
    else:
        raise NotImplementedError
```

Here are some examples:

Listing 4.30: in file: `numeric.py`

```
>>> def f(x): return x*x-5.0*x
>>> print(condition_number(f,1))
0.74999...
>>> A = Matrix.from_list([[1,2],[3,4]])
>>> print(condition_number(A))
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 4.31: in file: `numeric.py`

```
def exp(x,ap=1e-6,rp=1e-4,ns=40):
    if isinstance(x,Matrix):
```

```
        t = s = Matrix.identity(x.cols)
        for k in range(1,ns):
            t = t*x/k   # next term
            s = s + t   # add next term
            if norm(t)<max(ap,norm(s)*rp): return s
        raise ArithmeticError, 'no convergence'
    elif type(x)==type(1j):
        return cmath.exp(x)
    else:
        return math.exp(x)
```

Listing 4.32: in file: `numeric.py`

```
>>> A = Matrix.from_list([[1,2],[3,4]])
>>> print(exp(A))
[[51.96..., 74.73...], [112.10..., 164.07...]]
```

4.4.7 *Cholesky factorization*

A matrix is said to be positive definitive if $x^t Ax > 0$ for every $x \neq 0$.

If a matrix is symmetric and positive definitive then there exists a triangular matrix L such that $A = LL^t$.

The Cholesky algorithm takes a matrix A as input and returns the matrix L.

Listing 4.33: in file: `numeric.py`

```
def Cholesky(A):
    import copy, math
    if not is_almost_symmetric(A):
        raise ArithmeticError, 'not symmetric'
    L = copy.deepcopy(A)
    for k in xrange(L.cols):
        if L[k,k]<=0:
            raise ArithmeticError, 'not positive definitive'
        p = L[k,k] = math.sqrt(L[k,k])
        for i in xrange(k+1,L.rows):
            L[i,k] /= p
        for j in xrange(k+1,L.rows):
            p=float(L[j,k])
            for i in xrange(k+1,L.rows):
                L[i,j] -= p*L[i,k]
    for  i in xrange(L.rows):
        for j in range(i+1,L.cols):
            L[i,j]=0
    return L
```

Here we provide an example and a check that indeed $A = LL^t$:

Listing 4.34: in file: `numeric.py`

```
>>> A = Matrix.from_list([[4,2,1],[2,9,3],[1,3,16]])
>>> L = Cholesky(A)
>>> print(is_almost_zero(A - L*L.t))
True
```

The Cholesky algorithm fails if and only if the input matrix is not symmetric or not positive definitive, therefore it can be used to check whether a symmetric matrix is positive definite.

Consider for example a generic covariance matrix C. It is supposed to be positive definitive but sometimes it is not because it is computed incorrectly by taking different subsets of the data to compute C_{ij}, C_{jk} and C_{ik}. The Cholesky algorithm provides an algorithm to check whether a matrix is positive definite:

Listing 4.35: in file: `numeric.py`

```
def is_positive_definite(A):
    if not is_almost_symmetric(A):
        return False
    try:
        Cholesky(A)
        return True
    except RuntimeError:
        return False
```

Another application of the Cholesky is in generating vectors $\mathbf{x}$ with probability distribution

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}\mathbf{x}^t C^{-1}\mathbf{x}\right) \tag{4.72}$$

where C is a symmetric and positive definite matrix. In fact if $C = LL^t$ then:

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}(L^{-1}\mathbf{x})^t(L^{-1}\mathbf{x})\right) \tag{4.73}$$

and with a change of variable $\mathbf{u} = \mathbf{L}^{-1}\mathbf{x}$ we obtain:

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}\mathbf{u}^t\mathbf{u}\right) \tag{4.74}$$

and

$$p(\mathbf{x}) \propto e^{-\frac{1}{2}u_0^2} e^{-\frac{1}{2}u_1^2} e^{-\frac{1}{2}u_2^2} \ldots \tag{4.75}$$

Therefore the u_i components are Gaussian random variables.

In summary, given a covariance matrix C we can generate random vectors x or random numbers with the same covariance simply by doing:

```
class CovarianceGenerator(object):
    def __init__(self,C):
        self.L = Cholesky(C)
    def generate(self):
        import random
        u = Matrix(self.L.rows,1,fill=lambda r,c:random.gauss(0,1))
        x = self.L*u
        return x
```

We will use this algorithm later in the chapter on Monte Carlo simulations.

4.4.8 *Modern Portfolio Theory*

Modern portfolio theory [13] 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

$$\text{Sharpe}(r, \sigma) \equiv (r - \bar{r})/\sigma \tag{4.76}$$

Here $\bar{r}$ is the current risk free investement 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 average risk (variance of daily returns) and display it in a risk-return plot as we did in chapter 2.

We can try to building arbitrary portfolios by investing in multiple stocks at the same time. Modern portfolio theory states that there is a maximum Sharpe ratio we can achieve and there is only one portfolio that can achieve 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:

$$r_i = 1/T \sum_t r_{it} \tag{4.77}$$

and the covarance matrix:

$$A_{ij} = \frac{1}{T} \sum_t (r_{it} - r_i)(r_{jt} - r_j) \tag{4.78}$$

Modern Portfolio Theory tells use that the tangent portfolio is given by:

$$\mathbf{x} = A^{-1}(\mathbf{r} - \bar{r}\mathbf{1}) \tag{4.79}$$

Here is the algorithm:

Listing 4.36: in file: `numeric.py`

```
def Markowitz(mu, A, r_free):
    """Assess Markowitz risk/return.
    Example:
    >>> cov = Matrix.from_list([[0.04, 0.006,0.02],
    ...                         [0.006,0.09, 0.06],
    ...                         [0.02, 0.06, 0.16]])
    >>> mu = Matrix.from_list([[0.10],[0.12],[0.15]])
    >>> r_free = 0.05
    >>> x, ret, risk = Markowitz(mu, cov, r_free)
    >>> print(x)
    [0.556634..., 0.275080..., 0.1682847...]
    >>> print(ret, risk)
    0.113915... 0.186747...
    """
    x = Matrix(A.rows, 1)
    x = (1/A)*(mu - r_free)
    x = x/sum(x[r,0] for r in range(x.rows))
    portfolio = [x[r,0] for r in range(x.rows)]
    portfolio_return = mu*x
    portfolio_risk = sqrt(x*(A*x))
    return portfolio, portfolio_return, portfolio_risk
```

Here is an example. We consider three assets (0,1,2) with the following covariance matrix,

```
>>> cov = Matrix.from_list([[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, because the former are additive while the latter are not):

```
>>> mu = Matrix.from_list([[0.10],[0.12],[0.15]])
```

Given the following risk free interest rate:

```
>>> r_free = 0.05
```

We compute the tangent portfolio (highest Sharpe ratio), its return and its risk with one function call:

```
>>> x, ret, risk = Markowitz(mu, cov, r_free)
>>> print(x)
[0.5566343042071198, 0.27508090614886727, 0.16828478964401297]
>>> print(ret, risk)
0.113915857605 0.186747095412
>>> print((ret-r_free).risk)
0.34225891152
>>> for r in range(3): print((mu[r,0]-r_free)/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 ration of 0.34, much higher than 0.25, 0.23, and 0.23 for the individual assets.

Notice that in general the tangent portfolio is not the one we want to invest in. In fact given a fixed amount of money to invest we can invest a faction α in the tangent portfolio and leave the remaining $1-\alpha$ fraction in the bank at the risk free rate $\bar{r}$. This choice will give an overall return equal to:

$$\alpha \mathbf{x} \cdot \mathbf{r} + (1-\alpha)\bar{r} \tag{4.80}$$

and an overall risk:

$$\sqrt{\alpha \mathbf{x}^t A \mathbf{x}} \tag{4.81}$$

which we can adjust to our subjective preferences by choosing α. Whatever α we choose this strategy gives us the same Sharpe ratio as the tangent portfolio.

(4.82)

4.4.9 *Linear Least Squares and 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

$$\begin{aligned}
c_0 f_0(t_0) + c_1 f_1(t_0) + c_2 f_2(t_0) + ... &= e_0 \simeq o_0 && (4.83)\\
c_0 f_0(t_1) + c_1 f_1(t_1) + c_2 f_2(t_1) + ... &= e_1 \simeq o_1 && (4.84)\\
c_0 f_0(t_2) + c_1 f_1(t_2) + c_2 f_2(t_2) + ... &= e_2 \simeq o_2 && (4.85)\\
... &= ... && (4.86)
\end{aligned}$$

we want to find the $\{c_i\}$ which 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 and *least squares* when the $do_j = 1$.

$$\chi^2 = \sum_j \left| \frac{e_j - o_j}{do_j} \right|^2 \tag{4.87}$$

If we define the matrix A as

$$A = \begin{pmatrix} \frac{f_0(t_0)}{do_0} & \frac{f_1(t_0)}{do_0} & \frac{f_2(t_0)}{do_0} & \cdots \\ \frac{f_0(t_1)}{do_1} & \frac{f_1(t_1)}{do_1} & \frac{f_2(t_1)}{do_1} & \cdots \\ \frac{f_0(t_2)}{do_2} & \frac{f_1(t_2)}{do_2} & \frac{f_2(t_2)}{do_2} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix} \qquad b = \begin{matrix} \frac{o_0}{do_0} \\ \frac{o_1}{do_1} \\ \frac{o_2}{do_2} \\ \cdots \end{matrix} \tag{4.88}$$

Then the problem is reduced to

$$\chi^2 = \min_{\mathbf{c}} |A\mathbf{c} - \mathbf{b}|^2 \tag{4.89}$$

$$= \min_{\mathbf{c}} (A\mathbf{c} - \mathbf{b})^t (A\mathbf{c} - \mathbf{b}) \tag{4.90}$$

$$= \min_{\mathbf{c}} (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{b}^t A\mathbf{c} + \mathbf{b}^t\mathbf{b}) \tag{4.91}$$

This is the same as solving the following equation:

$$\nabla_c (\mathbf{c}^t A^t A\mathbf{x} - 2\mathbf{c}^t A^t\mathbf{b} + \mathbf{b}^t\mathbf{b}) = 0 \tag{4.92}$$

$$A^t A\mathbf{c} - A^t\mathbf{b} = 0 \tag{4.93}$$

Its solution is:

$$\mathbf{c} = (A^t A)^{-1}(A^t\mathbf{b}) \tag{4.94}$$

The algorithm below implements a fitting function based on the above 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, the fitting function.

Listing 4.37: in file: `numeric.py`

```
def fit_least_squares(points, f):
    """
    Computes c_j for best linear fit of y[i] \pm dy[i] = fitting_f(x[i])
    where fitting_f(x[i]) is \sum_j c_j f[j](x[i])

```

```
    parameters:
    - a list of fitting functions
    - a list with points (x,y,dy)

    returns:
    - column vector with fitting coefficients
    - the chi2 for the fit
    - the fitting function as a lambda x: ....
    """
    def eval_fitting_function(f,c,x):
        if len(f)==1: return c*f[0](x)
        else: return sum(func(x)*c[i,0] for i,func in enumerate(f))
    A = Matrix(len(points),len(f))
    b = Matrix(len(points))
    for i in range(A.rows):
        weight = 1.0/points[i][2] if len(points[i])>2 else 1.0
        b[i,0] = weight*float(points[i][1])
        for j in range(A.cols):
            A[i,j] = weight*f[j](float(points[i][0]))
    c = (1.0/(A.t*A))*(A.t*b)
    chi = A*c-b
    chi2 = norm(chi,2)**2
    fitting_f = lambda x, c=c, f=f, q=eval_fitting_function: q(f,c,x)
    return c.values, chi2, fitting_f

# examples of fitting functions
def POLYNOMIAL(n):
    return [(lambda x, p=p: x**p) for p in range(n+1)]
CONSTANT  = POLYNOMIAL(0)
LINEAR    = POLYNOMIAL(1)
QUADRATIC = POLYNOMIAL(2)
CUBIC     = POLYNOMIAL(3)
QUARTIC   = POLYNOMIAL(4)
```

As an example, we can use it to perform a polynomial fit:

Given a set of points we want to the find the coefficients of a polynomial that best approximates those points.

In other words, we want to find the c_i such that, given t_j and o_j,

$$c_0 + c_1 t_0^1 + c_2 t_0^2 + ... = e_0 \simeq o_0 \pm do_0 \quad (4.95)$$
$$c_0 + c_1 t_1^1 + c_2 t_1^2 + ... = e_1 \simeq o_1 \pm do_1 \quad (4.96)$$
$$c_0 + c_1 t_2^1 + c_2 t_2^2 + ... = e_2 \simeq o_2 \pm do_2 \quad (4.97)$$
$$... \quad ... \quad (4.98)$$
$$(4.99)$$

Here is how we can generate some random points and solve the problem for a polynomial of degree 2 (or quadratic fit):

Listing 4.38: in file: `numeric.py`

```
>>> points = [(k,5+0.8*k+0.3*k*k+math.sin(k),2) for k in range(100)]
>>> a,chi2,fitting_f = fit_least_squares(points,QUADRATIC)
>>> for p in points[-10:]:
...     print(p[0], round(p[1],2), round(fitting_f(p[0]),2))
90 2507.89 2506.98
91 2562.21 2562.08
92 2617.02 2617.78
93 2673.15 2674.08
94 2730.75 2730.98
95 2789.18 2788.48
96 2847.58 2846.58
97 2905.68 2905.28
98 2964.03 2964.58
99 3023.5 3024.48
>>> draw(title='polynomial fit',xlab='t',ylab='e(t),o(t)',
...      pointsets=[{'label':'o(t)','data':points[:10]}],
...      linesets=[{'label':'e(t)','data':[(p[0],fitting_f(p[0])) for p in points
    [:10]]}],
...      filename = 'images/polynomialfit.png')
```

And here is a plot of the first 10 points compared with the best fit:

4.4.10 *Trading and technical analysis*

In finance, *technical analysis* is an empirical discipline which 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:

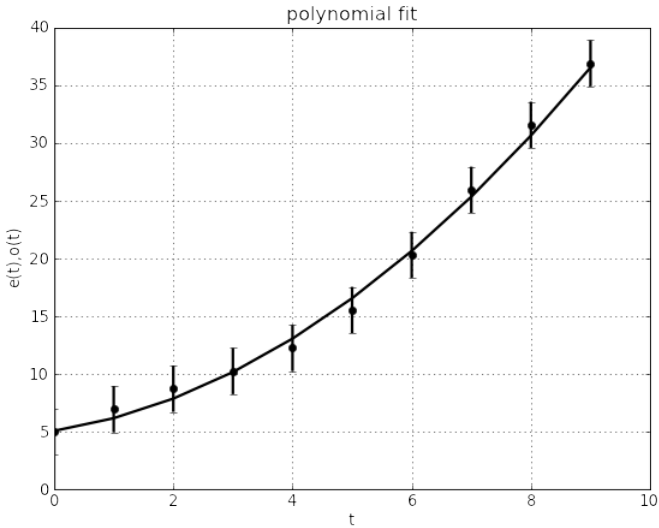

Figure 4.4: Random data with their erro bars and the polynomial best fit.

- 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 4.39: in file: `numeric.py`

```
class Trader:
    def model(self,window):
        "the forecasting model"
        # we fit last few days quadratically
        points = [(x,y) for (x,y) in enumerate(window)]
        a,chi2,fitting_f = fit_least_squares(points,QUADRATIC)
        # and we extrapolate tomorrow's price
        price_tomorrow = fitting_f(len(points))
        return price_tomorrow

    def strategy(self,window):
        "the trading strategy"
        price_today = window[-1]
        price_tomorrow = self.model(window)
        if price_tomorrow>price_today:
```

```
            return 'buy'
        else:
            return 'sell'

    def simulate(self,data,cash=1000.0,shares=0.0,days=7,daily_rate=0.03/360):
        "find fitting parameters that optimize the trading strategy"
        for t in range(days,len(data)):
            window =  data[t-days:t]
            today_close = window[-1]
            suggestion = self.strategy(window)
            # and we buy or sell based on our strategy
            if cash>0 and suggestion=='buy':
                # we keep track of finances
                shares_bought = int(cash/today_close)
                shares += shares_bought
                cash -= shares_bought*today_close
            elif shares>0 and suggestion=='sell':
                cash += shares*today_close
                shares = 0.0
            # we assume money in the bank also gains an interest
            cash*=math.exp(daily_rate)
        # we return the net worth
        return cash+shares*data[-1]
```

Now we back test the strategy using financial data for AAPL for the year 2011:

Listing 4.40: in file: `numeric.py`

```
>>> from datetime import date
>>> data = YStock.download('aapl','adjusted_close',
...        start=date(2011,1,1),stop=date(2011,12,31))
>>> print(Trader().simulate(data,cash=1000.0))
1133.2463...
>>> print(1000.0*math.exp(0.03))
1030.4545...
>>> print(1000.0*data[-1]/data[0])
1228.8739...
```

Our strategy did considerably better than the risk free return of 3%, but not as well as investing and holding AAPL shares over the same period.

Of course we can always engineer a strategy based on historical data that will outperform holding the stock, but *that 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 are indications 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 [15].

It is easy to extend the previous class to implement other strategies and backtest them.

4.4.11 *Eigenvalues and 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:

$$A\mathbf{x}_i = e_i\mathbf{x}_i \tag{4.100}$$

Some may be repeated (appear more than once) and they are called degenerate. Some may be zero ($e_i = 0$) which means the matrix A is singular. A matrix is singular if it maps a non-zero 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 cols) of A then its eigenvalues are real and the matrix can we written as:

$$A = UDU^t \tag{4.101}$$

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 the smallest to the largest.

Listing 4.41: in file: `numeric.py`

```
1 def sqrt(x):
```

```
    try:
        return math.sqrt(x)
    except ValueError:
        return cmath.sqrt(x)

def Jacobi_eigenvalues(A,checkpoint=False):
    """Returns U end e so that A=U*Matrix.diagonal(e)*transposed(U)
       where i-column of U contains the eigenvector corresponding to
       the eigenvalue e[i] of A.

       from http://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm
    """
    def maxind(M,k):
        j=k+1
        for i in xrange(k+2,M.cols):
            if abs(M[k,i])>abs(M[k,j]):
               j=i
        return j
    n = A.rows
    if n!=A.cols:
        raise ArithmeticError, 'matrix not squared'
    indexes = xrange(n)
    S = Matrix(n,n, fill=lambda r,c: float(A[r,c]))
    E = Matrix.identity(n)
    state = n
    ind = [maxind(S,k) for k in indexes]
    e = [S[k,k] for k in indexes]
    changed = [True for k in indexes]
    iteration = 0
    while state:
        if checkpoint: checkpoint('rotating vectors (%i) ...' % iteration)
        m=0
        for k in xrange(1,n-1):
            if abs(S[k,ind[k]])>abs(S[m,ind[m]]): m=k
            pass
        k,h = m,ind[m]
        p = S[k,h]
        y = (e[h]-e[k])/2
        t = abs(y)+sqrt(p*p+y*y)
        s = sqrt(p*p+t*t)
        c = t/s
        s = p/s
        t = p*p/t
        if y<0: s,t = -s,-t
        S[k,h] = 0
        y = e[k]
        e[k] = y-t
        if changed[k] and y==e[k]:
            changed[k],state = False,state-1
```

```
        elif (not changed[k]) and y!=e[k]:
            changed[k],state = True,state+1
        y = e[h]
        e[h] = y+t
        if changed[h] and y==e[h]:
            changed[h],state = False,state-1
        elif (not changed[h]) and y!=e[h]:
            changed[h],state = True,state+1
        for i in xrange(k):
            S[i,k],S[i,h] = c*S[i,k]-s*S[i,h],s*S[i,k]+c*S[i,h]
        for i in xrange(k+1,h):
            S[k,i],S[i,h] = c*S[k,i]-s*S[i,h],s*S[k,i]+c*S[i,h]
        for i in xrange(h+1,n):
            S[k,i],S[h,i] = c*S[k,i]-s*S[h,i],s*S[k,i]+c*S[h,i]
        for i in indexes:
            E[k,i],E[h,i] = c*E[k,i]-s*E[h,i],s*E[k,i]+c*E[h,i]
        ind[k],ind[h]=maxind(S,k),maxind(S,h)
        iteration+=1
    # sort vectors
    for i in xrange(1,n):
        j=i
        while j>0 and e[j-1]>e[j]:
            e[j],e[j-1] = e[j-1],e[j]
            E.swap_rows(j,j-1)
            j-=1
    # normalize vectors
    U = Matrix(n,n)
    for i in indexes:
        norm = sqrt(sum(E[i,j]**2 for j in indexes))
        for j in indexes: U[j,i] = E[i,j]/norm
    return U,e
```

Here is an example which shows, for a particular case, the relation between the input, A of the output of the U, e of the Jacobi algorithm.

Listing 4.42: in file: `numeric.py`

```
>>> import random
>>> A = Matrix(4,4)
>>> for r in range(A.rows):
...     for c in range(r,A.cols):
...         A[r,c] = A[c,r] = random.gauss(10,10)
>>> U,e = Jacobi_eigenvalues(A)
>>> print(is_almost_zero(U*Matrix.diagonal(e)*U.t-A))
True
```

Eigenvalues can be used to filter noise out of data and find hidden dependencies in data. Below 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 which we have downloaded and stored in chapter 2.

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 4.43: in file: `numeric.py`

```
def compute_correlation(stocks, key='arithmetic_return'):
    "The input must be a list of YStock(...).historical() data"
    # find trading days common to all stocks
    days = set()
    nstocks = len(stocks)
    iter_stocks = xrange(nstocks)
    for stock in stocks:
         if not days: days=set(x['date'] for x in stock)
         else: days=days.intersection(set(x['date'] for x in stock))
    n = len(days)
    v = []
    # filter out data for the other days
    for stock in stocks:
        v.append([x[key] for x in stock if x['date'] in days])
    # compute mean returns (skip first day, data not reliable)
    mus = [sum(v[i][k] for k in range(1,n))/n for i in iter_stocks]
    # fill in the covariance matrix
    var = [sum(v[i][k]**2 for k in range(1,n))/n - mus[i]**2 for i in iter_stocks]
    corr = Matrix(nstocks,nstocks,fill=lambda i,j: \
             (sum(v[i][k]*v[j][k] for k in range(1,n))/n - mus[i]*mus[j])/ \
             math.sqrt(var[i]*var[j]))
    return corr
```

We use the above function to compute the correlation and pass it as input to the Jacobi algorithm and plot the output eigenvalues:

Listing 4.44: in file: `numeric.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> symbols = storage.keys('*/2011')[:20]
>>> stocks = [storage[symbol] for symbol in symbols]
>>> corr = compute_correlation(stocks)
>>> U,e = Jacobi_eigenvalues(corr)
>>> draw(title='SP100 eigenvalues',xlab='i',ylab='e[i]',filename='images/sp100eigen
    .png',
...      linesets=[{'data':[(i,ei) for i,ei, in enumerate(e)]}])
```

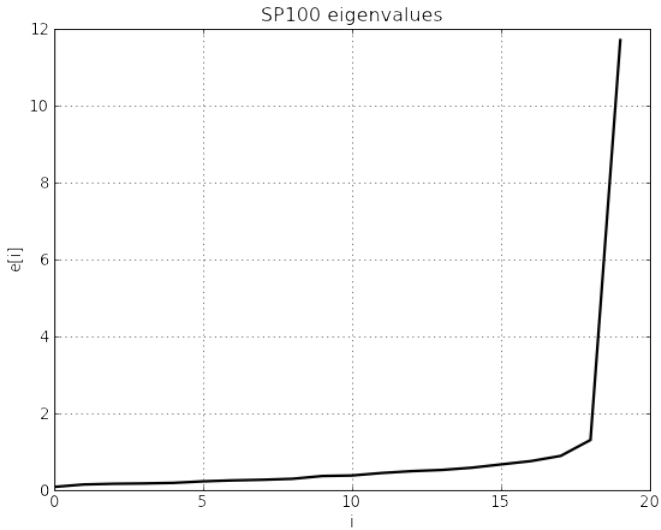

Figure 4.5: Eigenvalues of the correlation matrix for 20 of the S&P100 stocks, sorted by their magnitude.

The image shows that one eigenvalue, the last one, is much larger than the others. It tells us that the data serieses have something in common and the arithmetic retuns for stock j can be written as

$$\mathbf{r}_i = \beta_i \mathbf{p} + \alpha_i \tag{4.102}$$

where $\mathbf{p}$ is the principal component given by

$$\mathbf{p} = \sum_i U_{n-1,j} \mathbf{r}_j \tag{4.103}$$

and

$$\beta_i = \sum_{i,j} U_{n-1,j} \mathbf{r}_i^t \mathbf{r}_j \tag{4.104}$$

$\mathbf{p}$ is a vector of adjusted arithmetic returns that represents the market better than any other vector, while the α_i are independent from $\mathbf{p}$ and can be thought of as noise. The β_i coefficient tells us how much $\mathbf{r}_i$ overlaps with $\mathbf{p}$.

4.5 Sparse matrix inversion

A sparse matrix is a matrix with few off-diaginal element different from zero. This happens very frequently in science, engineering and finance when solving differential equations. Often we have to invert matrices which are very large and the exact Gauss-Jordan algorithms would be impractical but we know they are sparse. In this case two algorithms come to our rescue the *minimum residue* and the *bi-conjugate gradient* (for which we consider a variant called *stabilized bi-conjugate gradient*):

We will also assume that the matrix to be inverted is given in some implicit algorithmic as $\mathbf{y} = f(\mathbf{x})$ since this is always the case for sparse matrices. There is no point to storing all its elements since most of them are zero.

4.5.1 Minimum residue

Given a linear operator f, the Krylov space spanned by a vector x is defined as

$$K(f, y, i) = \{y, f(y), f(f(y)), f(f(f(y))), (f^i)(y)\} \tag{4.105}$$

The *minimum residue* algorithm works by solving $x = f^{-1}(y)$ iteratively. At each iteration it computes a new orthgonal basis vector q_i for the Krylov space $K(f, x, i)$, and computes α_i

$$x_i = y + \alpha_1 q_1 + \alpha_2 q_2 + ... + \alpha_i q_i \in K(f, y, i+1) \tag{4.106}$$

which minimizes the norm of the residue defined as:

$$r = f(x_i) - y \tag{4.107}$$

Therefore $\lim_{i\to\infty} f(x_i) = y$. If a solution to the original problem exists, ignoring precision issues, the minimum residue converges to it and the residue decreses at each iteration.

Notice that in the code below x and y are exchanged because we adopt the convention that y is the output and x is the input.

Listing 4.45: in file: `numeric.py`

```
def invert_minimum_residue(f,x,ap=1e-4,rp=1e-4,ns=200):
    import copy
    y = copy.copy(x)
    r = x-1.0*f(x)
    for k in xrange(ns):
        q = f(r)
        alpha = (q*r)/(q*q)
        y = y + alpha*r
        r = r - alpha*q
        residue = sqrt((r*r)/r.rows)
        if residue<max(ap,norm(y)*rp): return y
    raise ArithmeticError, 'no convergence'
```

4.5.2 *Stabilized bi-conjugate gradient*

The stabilized biconjugate gradient method is also based on constructing a Krylov subspace and minimizing the same residue as in the minimum residue algorithm, yet it is faster than the minimum residue and has a smoother converegence than other conjugate gradient methods.

Listing 4.46: in file: `numeric.py`

```
def invert_bicgstab(f,x,ap=1e-4,rp=1e-4,ns=200):
    import copy
    y = copy.copy(x)
    r = x - 1.0*f(x)
    q = r
    p = 0.0
    s = 0.0
    rho_old = alpha = omega = 1.0
    for k in xrange(ns):
```

```
        rho = q*r
        beta = (rho/rho_old)*(alpha/omega)
        rho_old = rho
        p = beta*p + r - (beta*omega)*s
        s = f(p)
        alpha = rho/(q*s)
        r = r - alpha*s
        t = f(r)
        omega = (t*r)/(t*t)
        y = y + omega*r + alpha*p
        residue=sqrt((r*r)/r.rows)
        if residue<max(ap,norm(y)*rp): return y
    raise ArithmeticError, 'no convergence'
```

Notice that the minimum residue and the stabilized biconjugate gradient, if they converge, they 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 pixels. The de-focusing 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. If we store the pixels in a 1D vector (both for x and y) as opposed to a matrix, we can write:

$$\mathbf{y} = A\mathbf{x} \tag{4.108}$$

If we assume the image is $m \times m$ squared, then the point x_i corresponds to the pixel of coordinates $(r, c) = (i/m, i\%m)$. The exact shape of A depends on the details of the lens. For a simple lens, we can write:

$$A = S^{\beta} \tag{4.109}$$

and

$$S_{ij} = (1 - \alpha/4)\delta_{i,j} + \alpha\delta_{i,j\pm1} + \alpha\delta_{i,j\pm m} \tag{4.110}$$

where 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 code below 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:

Listing 4.47: in file: `numeric.py`

```
>>> m = 30
>>> x = Matrix(m*m,1,fill=lambda r,c:(r//m in(10,20) or r%m in(10,20)) and 1. or
    0.)
>>> def smear(x):
...     alpha, beta = 0.4, 8
...     for k in range(beta):
...        y = Matrix(x.rows,1)
...        for r in range(m):
...            for c in range(m):
...                y[r*m+c,0] = (1.0-alpha/4)*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 c>0:   y[r*m+c,0] += alpha * x[r*m+c-m,0]
...        x = y
...     return y
>>> y = smear(x)
>>> z = invert_minimum_residue(smear,y,ns=1000)
>>> y.cols = y.rows = z.rows = z.cols = m
>>> color2d(title="Defocused image", data = y.as_list(),
...         filename='images/defocused.png')
>>> color2d(title="refocus image", data = z.as_list(),
...         filename='images/refocused.png')
```

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.

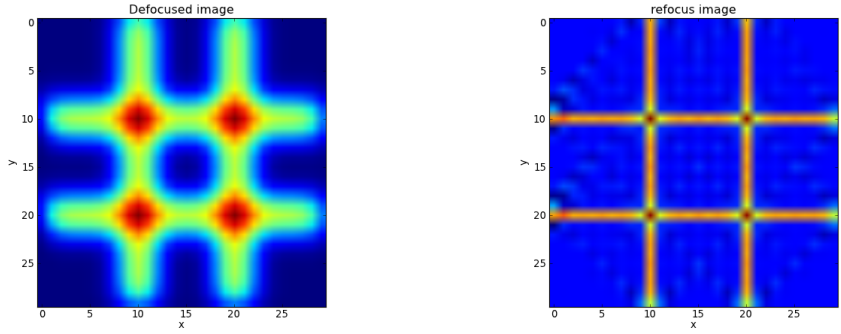

Figure 4.6: An out of focus image (left) and the original image (image) computed from the out of focus one, using sparse matric inversion.

4.6 *Solvers for non-linear equations*

In this chapter we are concerned with the problem of solving in x the equation of one variable:

$$f(x) = 0 \tag{4.111}$$

4.6.1 *Fixed-point method*

It is always possible to reformulate $f(x) = 0$ as $g(x) = x$ using 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:

$$x_1 = g(x_0) \tag{4.112}$$
$$x_2 = g(x_1) \tag{4.113}$$
$$x_3 = g(x_2) \tag{4.114}$$
$$\ldots = \ldots \tag{4.115}$$

We can compute the distance between x_i and x as:

$$\begin{aligned} |x_i - x| &= |g(x_{i-1}) - g(x)| && (4.116)\\ &= |g(x) + g'(\xi)(x_{i-1} - x) - g(x)| && (4.117)\\ &= |g'(\xi)||x_{i-1} - x| && (4.118) \end{aligned}$$

where we use *de l'Hopital rule* and ξ is a point in between x and x_{i-1}.

If the magnitude of first derivative of g, $|g'|$, is less then one in a neighborhood of x and if x_0 is in such neighborhood then:

$$|x_i - x| = |g'(\xi)||x_{i-1} - x| < |x_{i-1} - x| \quad (4.119)$$

the x_i series will get closer and closer to the solution x.

Here is the process implemented into an algorithm:

Listing 4.48: in file: `numeric.py`

```
def solve_fixed_point(f, x, ap=1e-6, rp=1e-4, ns=100):
    def g(x): return f(x)+x # f(x)=0 <=> g(x)=x
    Dg = D(g)
    for k in xrange(ns):
        if abs(Dg(x)) >= 1:
            raise ArithmeticError, 'error D(g)(x)>=1'
        (x_old, x) = (x, g(x))
        if k>2 and norm(x_old-x)<max(ap,norm(x)*rp):
            return x
    raise ArithmeticError, 'no convergence'
```

And here is an example:

Listing 4.49: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)/10
>>> print(round(solve_fixed_point(f,1.0,rp=0),4))
2.0
```

4.6.2 *Bisection method*

The goal of the bisection 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 4.50: in file: `numeric.py`

```
def solve_bisection(f, a, b, ap=1e-6, rp=1e-4, ns=100):
    fa, fb = f(a), f(b)
    if fa == 0: return a
    if fb == 0: return b
    if fa*fb > 0:
        raise ArithmeticError, 'f(a) and f(b) must have opposite sign'
    for k in xrange(ns):
        x = (a+b)/2
        fx = f(x)
        if fx==0 or norm(b-a)<max(ap,norm(x)*rp): return x
        elif fx * fa < 0: (b,fb) = (x, fx)
        else: (a,fa) = (x, fx)
    raise ArithmeticError, 'no convergence'
```

Here is how to use it:

Listing 4.51: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(solve_bisection(f,1.0,3.0),4))
2.0
```

4.6.3 *Newton method*

The Newton algorithm also solves $f(x) = 0$. It is faster (in average) than the bisection method because it make 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 1st order Taylor expansion:

$$f(x) \simeq f(x_0) + f'(x_0)(x - x_0) \tag{4.120}$$

and solves it exactly:

$$f(x) = 0 \to x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4.121}$$

thus finding a new and better estimate for the solution. The algorithm iterates the above equation and, when it converges, it approximates the exact solution better and better. The algorithm is only guaranteed to converge if $|f'(x)|$ is limited without a neighborhood of the solution and if the initial starting point is within this neighborhood.

Listing 4.52: in file: `numeric.py`

```
def solve_newton(f, x, ap=1e-6, rp=1e-4, ns=20):
    x = float(x) # make sure it is not int
    for k in xrange(ns):
        (fx, Dfx) = (f(x), D(f)(x))
        if norm(Dfx) < ap:
            raise ArithmeticError, 'unstable solution'
        (x_old, x) = (x, x-fx/Dfx)
        if k>2 and norm(x-x_old)<max(ap,norm(x)*rp): return x
    raise ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.53: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(solve_newton(f,1.0),4))
2.0
```

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:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{4.122}$$

$$x_{i+i} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.123}$$

As the algorithm approaches the exact solution, the numerical derivative becomes a better and better approximation for the derivative.

Listing 4.54: in file: `numeric.py`

```
def solve_secant(f, x, ap=1e-6, rp=1e-4, ns=20):
    x = float(x) # make sure it is not int
    (fx, Dfx) = (f(x), D(f)(x))
    for k in xrange(ns):
        if norm(Dfx) < ap:
            raise ArithmeticError, 'unstable solution'
        (x_old, fx_old,x) = (x, fx, x-fx/Dfx)
        if k>2 and norm(x-x_old)<max(ap,norm(x)*rp): return x
        fx = f(x)
        Dfx = (fx-fx_old)/(x-x_old)
    raise ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.55: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(solve_secant(f,1.0),4))
2.0
```

4.6.5 *Newton Stabilized*

The Newton stabilized is a combination of the Newton algorithm and the bisection algorithm. It requires a domain (a_0, b_0) as input that brackets the function and changes sign in between a_0 and b_0. In the first iteration it uses the bisection algorithm to find the middle point and then uses the Newton algorithm to find a better estimate for the solution x. At each iteration it reduces the domain (a_i, b_i) to $(a_{i+1}, b_{i+1}) = (a_i, x)$ or $(a_{i+1}, b_{i+1}) = (x, b_i)$ depending on the sign of $f(x)$.

The Newton algorithm is unstable if $|f'(x)|$ is large. When this happens the algorithm shoots x out of the domain (a_i, b_i). In this case the stabilized newton ignores the newly computed value of x outside of the domain and performs a bisection step, moving x back in the domain.

Listing 4.56: in file: `numeric.py`

```
def solve_newton_stabilized(f, a, b, ap=1e-6, rp=1e-4, ns=20):
    fa, fb = f(a), f(b)
    if fa == 0: return a
    if fb == 0: return b
    if fa*fb > 0:
        raise ArithmeticError, 'f(a) and f(b) must have opposite sign'
```

```
    x = (a+b)/2
    (fx, Dfx) = (f(x), D(f)(x))
    for k in xrange(ns):
        x_old, fx_old = x, fx
        if norm(Dfx)>ap: x = x - fx/Dfx
        if x==x_old or x<a or x>b: x = (a+b)/2
        fx = f(x)
        if fx==0 or norm(x-x_old)<max(ap,norm(x)*rp): return x
        Dfx = (fx-fx_old)/(x-x_old)
        if fx * fa < 0: (b,fb) = (x, fx)
        else: (a,fa) = (x, fx)
    raise ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.57: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(solve_newton_stabilized(f,1.0,3.0),4))
2.0
```

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 apply simply rename all previous solvers and replace $f(x)$ with $f'(x)$ and $f'(x)$ with $f''(x)$.

4.7.1 *Bisection method*

Listing 4.58: in file: `numeric.py`

```
def optimize_bisection(f, a, b, ap=1e-6, rp=1e-4, ns=100):
    Dfa, Dfb = D(f)(a), D(f)(b)
    if Dfa == 0: return a
    if Dfb == 0: return b
    if Dfa*Dfb > 0:
        raise ArithmeticError, 'D(f)(a) and D(f)(b) must have opposite sign'
    for k in xrange(ns):
        x = (a+b)/2
```

```
        Dfx = D(f)(x)
        if Dfx==0 or norm(b-a)<max(ap,norm(x)*rp): return x
        elif Dfx * Dfa < 0: (b,Dfb) = (x, Dfx)
        else: (a,Dfa) = (x, Dfx)
    raise ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.59: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(optimize_bisection(f,2.0,5.0),4))
3.5
```

4.7.2 *Newton method*

Listing 4.60: in file: `numeric.py`

```
def optimize_newton(f, x, ap=1e-6, rp=1e-4, ns=20):
    x = float(x) # make sure it is not int
    for k in xrange(ns):
        (Dfx, DDfx) = (D(f)(x), DD(f)(x))
        if Dfx==0: return x
        if norm(DDfx) < ap:
            raise ArithmeticError, 'unstable solution'
        (x_old, x) = (x, x-Dfx/DDfx)
        if norm(x-x_old)<max(ap,norm(x)*rp): return x
    raise ArithmeticError, 'no convergence'
```

Listing 4.61: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(optimize_newton(f,3.0),3))
3.5
```

4.7.3 *Secant method*

Listing 4.62: in file: `numeric.py`

```
def optimize_secant(f, x, ap=1e-6, rp=1e-4, ns=100):
    x = float(x) # make sure it is not int
    (fx, Dfx, DDfx) = (f(x), D(f)(x), DD(f)(x))
    for k in xrange(ns):
        if Dfx==0: return x
        if norm(DDfx) < ap:
            raise ArithmeticError, 'unstable solution'
        (x_old, Dfx_old, x) = (x, Dfx, x-Dfx/DDfx)
```

```
        if norm(x-x_old)<max(ap,norm(x)*rp): return x
        fx = f(x)
        Dfx = D(f)(x)
        DDfx = (Dfx - Dfx_old)/(x-x_old)
    raise ArithmeticError, 'no convergence'
```

Listing 4.63: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(optimize_secant(f,3.0),3))
3.5
```

4.7.4 *Newton stabilized*

Listing 4.64: in file: `numeric.py`

```
def optimize_newton_stabilized(f, a, b, ap=1e-6, rp=1e-4, ns=20):
    Dfa, Dfb = D(f)(a), D(f)(b)
    if Dfa == 0: return a
    if Dfb == 0: return b
    if Dfa*Dfb > 0:
        raise ArithmeticError, 'D(f)(a) and D(f)(b) must have opposite sign'
    x = (a+b)/2
    (fx, Dfx, DDfx) = (f(x), D(f)(x), DD(f)(x))
    for k in xrange(ns):
        if Dfx==0: return x
        x_old, fx_old, Dfx_old = x, fx, Dfx
        if norm(DDfx)>ap: x = x - Dfx/DDfx
        if x==x_old or x<a or x>b: x = (a+b)/2
        if norm(x-x_old)<max(ap,norm(x)*rp): return x
        fx = f(x)
        Dfx = (fx-fx_old)/(x-x_old)
        DDfx = (Dfx-Dfx_old)/(x-x_old)
        if Dfx * Dfa < 0: (b,Dfb) = (x, Dfx)
        else: (a,Dfa) = (x, Dfx)
    raise ArithmeticError, 'no convergence'
```

Listing 4.65: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(optimize_newton_stabilized(f,2.0,5.0),3))
3.5
```

4.7.5 *Golden-section search*

If the function we want to optimize is continuous but not differentiable then the previous algorithms do not work (none of them). In this case there is one

algorithm that comes to our rescue, the Golden-section 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_i, b_i), the golden search needs two points, x_1 and x_2 in between (a_i, b_i) and from them it can determine whether the extreme is in (a_i, x_2) or in (x_1, b_i). The two points are chosen in an optimal way so that at each iteration, one of the two can be recycled, by leaving the ratio between (a_i, x_1) and (x_2, b_i) fixed and equal to 1, at each iteration.

Listing 4.66: in file: `numeric.py`

```
def optimize_golden_search(f, a, b, ap=1e-6, rp=1e-4, ns=100):
    a,b=float(a),float(b)
    tau = (sqrt(5.0)-1.0)/2.0
    x1, x2 = a+(1.0-tau)*(b-a), a+tau*(b-a)
    fa, f1, f2, fb = f(a), f(x1), f(x2), f(b)
    for k in xrange(ns):
        if f1 > f2:
            a, fa, x1, f1 = x1, f1, x2, f2
            x2 = a+tau*(b-a)
            f2 = f(x2)
        else:
            b, fb, x2, f2 = x2, f2, x1, 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 ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.67: in file: `numeric.py`

```
>>> def f(x): return (x-2)*(x-5)
>>> print(round(optimize_golden_search(f,2.0,5.0),3))
3.5
```

4.8 Functions of many variables

In order to be able to work with functions of many variables we need to introduce the concept of partial derivative:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h\to 0} \frac{f(\mathbf{x}+\mathbf{h}_i) - f(\mathbf{x}-\mathbf{h}_i)}{2h} \tag{4.124}$$

where $\mathbf{h}_i$ is a vector with components all equal to zero but $h_i = h$.

We can implement it as follows:

Listing 4.68: in file: `numeric.py`

```
def partial(f,i,h=1e-4):
    def df(x,f=f,i=i,h=h):
        u = f([e+(h if i==j else 0) for j,e in enumerate(x)])
        w = f([e-(h if i==j else 0) for j,e in enumerate(x)])
        try:
            return (u-w)/2/h
        except TypeError:
            return [(u[i]-w[i])/2/h for i in range(len(u))]
    return df
```

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 `except` condition allows the function to deal with the difference between two lists or tuples.

Here is an example:

Listing 4.69: in file: `numeric.py`

```
>>> def f(x): return 2.0*x[0]+3.0*x[1]+5.0*x[1]*x[2]
>>> df0 = partial(f,0)
>>> df1 = partial(f,1)
>>> df2 = partial(f,2)
>>> x = (1,1,1)
>>> print(round(df0(x),4), round(df1(x),4), round(df2(x),4))
2.0 8.0 5.0
```

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 2nd order as:

$$\begin{aligned} f(x_0, x_1, x_2, ...) &= f(\bar{x}_0, \bar{x}_1, \bar{x}_2, ...) + && (4.125)\\ &\sum_i \frac{\partial f(\bar{\mathbf{x}})}{\partial x_i}(x_i - \bar{x}_i) + && (4.126)\\ &\sum_{ij} \frac{1}{2}\frac{\partial^2 f}{\partial x_i \partial x_j}(\bar{\mathbf{x}})(x_i - \bar{x}_i)(x_j - \bar{x}_j) + ... && (4.127) \end{aligned}$$

We can rewrite the Jacobian in terms of the vector $\mathbf{x}$ as follows

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}})^t H_f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}) + ... \tag{4.128}$$

where we introduce the gradient vector

$$\nabla_f(x) \equiv \begin{pmatrix} \partial f(x)/\partial x_0 \\ \partial f(x)/\partial x_1 \\ \partial f(x)/\partial x_2 \\ ... \end{pmatrix} \tag{4.129}$$

and the Hessian matrix:

$$H_f(x) \equiv \begin{pmatrix} \partial^2 f(x)/\partial x_0 \partial x_0 & \partial^2 f(x)/\partial x_0 \partial x_1 & \partial^2 f(x)/\partial x_0 \partial x_2 & ... \\ \partial^2 f(x)/\partial x_1 \partial x_0 & \partial^2 f(x)/\partial x_1 \partial x_1 & \partial^2 f(x)/\partial x_1 \partial x_2 & ... \\ \partial^2 f(x)/\partial x_2 \partial x_0 & \partial^2 f(x)/\partial x_2 \partial x_1 & \partial^2 f(x)/\partial x_2 \partial x_2 & ... \\ ... & ... & ... & ... \end{pmatrix} \tag{4.130}$$

Given the definition of partial we can compute the gradient and the Hessian using the two functions below:

Listing 4.70: in file: `numeric.py`

```
def gradient(f, x, h=1e-4):
    return Matrix(len(x),fill=lambda r,c: partial(f,r,h)(x))

def hessian(f, x, h=1e-4):
    return Matrix(len(x),len(x),fill=lambda r,c: partial(partial(f,r,h),c,h)(x))
```

Here is an example:

Listing 4.71: in file: `numeric.py`

```
>>> def f(x): return 2.0*x[0]+3.0*x[1]+5.0*x[1]*x[2]
>>> print(gradient(f, x=(1,1,1)))
[[1.999999...], [7.999999...], [4.999999...]]
>>> print(hessian(f, x=(1,1,1)))
[[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

$$f(\mathbf{x}) = (f_0(\mathbf{x}), f_1(\mathbf{x}), f_2(\mathbf{x}), ...) \tag{4.131}$$

we need to Taylor expand each component:

$$f(\mathbf{x}) = \begin{pmatrix} f_0(\mathbf{x}) \\ f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ ... \end{pmatrix} = \begin{pmatrix} f_0(\bar{\mathbf{x}}) + \nabla_{f_0}(\mathbf{x}-\bar{\mathbf{x}}) + ... \\ f_1(\bar{\mathbf{x}}) + \nabla_{f_1}(\mathbf{x}-\bar{\mathbf{x}}) + ... \\ f_2(\bar{\mathbf{x}}) + \nabla_{f_2}(\mathbf{x}-\bar{\mathbf{x}}) + ... \\ ... \end{pmatrix} \tag{4.132}$$

which we can rewrite as:

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + J_f(\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}}) + ... \tag{4.133}$$

where J_f is called Jacobian and defined as:

$$J_f \equiv \begin{pmatrix} \partial f_0(x)/\partial x_0 & \partial f_0(x)/\partial x_1 & \partial f_0(x)/\partial x_2 & ... \\ \partial f_1(x)/\partial x_0 & \partial f_1(x)/\partial x_1 & \partial f_1(x)/\partial x_2 & ... \\ \partial f_2(x)/\partial x_0 & \partial f_2(x)/\partial x_1 & \partial f_2(x)/\partial x_2 & ... \\ ... & ... & ... & ... \end{pmatrix} \tag{4.134}$$

which we can implement as follows:

Listing 4.72: in file: numeric.py

```
def jacobian(f, x, h=1e-4):
    partials = [partial(f,c,h)(x) for c in xrange(len(x))]
    return Matrix(len(partials[0]),len(x),fill=lambda r,c: partials[c][r])
```

Here is an example:

Listing 4.73: in file: numeric.py

```
>>> def f(x): return (2.0*x[0]+3.0*x[1]+5.0*x[1]*x[2], 2.0*x[0])
>>> print(jacobian(f, x=(1,1,1)))
[[1.9999999..., 7.999999..., 4.9999999...], [1.9999999..., 0.0, 0.0]]
```

4.8.2 *Newton method (solver)*

As for the 1D case we can approximate $f(\mathbf{x})$ with its Taylor expansion at the first order:

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla_f(\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}}) + \frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})^t H_f(\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}}) \tag{4.135}$$

set its derivative to zero and solve it, thus obtaining:

$$\mathbf{x} = \bar{\mathbf{x}} - H_f^{-1}\nabla_f \tag{4.136}$$

which constitutes the core of the multidimensional Newton optimizer:

Listing 4.74: in file: numeric.py

```
def solve_newton_multi(f, x, ap=1e-6, rp=1e-4, ns=20):
    """
    Computes the root of a multidimensional function f near point x.

    Parameters
    f is a function that takes a list and returns a scalar
    x is a list

    Returns x, solution of f(x)=0, as a list
    """
    x = Matrix.from_list([x]).t
    fx = Matrix.from_list([f(x.values)]).t
    for k in xrange(ns):
        (fx.values, J) = (f(x.values), jacobian(f,x.values))
        if norm(J) < ap:
            raise ArithmeticError, 'unstable solution'
```

```
        (x_old, x) = (x, x-(1.0/J)*fx)
        if k>2 and norm(x-x_old)<max(ap,norm(x)*rp): return x.values
    raise ArithmeticError, 'no convergence'
```

Here is an example:

Listing 4.75: in file: numeric.py

```
>>> def f(x): return (x[0]+x[1], x[0]+x[1]**2-2)
>>> print(solve_newton_multi(f, x=(0,0)))
[1.0..., -1.0...]
```

4.8.3 Newton method (optimize)

Listing 4.76: in file: numeric.py

```
def optimize_newton_multi(f, x, ap=1e-6, rp=1e-4, ns=20):
    """
    Finds the extreme of multidimensional function f near point x.

    Parameters
    f is a function that takes a list and returns a scalar
    x is a list

    Returns x, which maximizes of minimizes f(x)=0, as a list
    """
    x = Matrix.from_list([x]).t
    for k in xrange(ns):
        (grad,H) = (gradient(f,x.values), hessian(f,x.values))
        if norm(H) < ap:
            raise ArithmeticError, 'unstable solution'
        (x_old, x) = (x, x-(1.0/H)*grad)
        if k>2 and norm(x-x_old)<max(ap,norm(x)*rp): return x.values
    raise ArithmeticError, 'no convergence'
```

Listing 4.77: in file: numeric.py

```
>>> def f(x): return (x[0]-2)**2+(x[1]-3)**2
>>> print(optimize_newton_multi(f, x=(0,0)))
[2.0, 3.0]
```

4.8.4 Improved Newton method (optimize)

We can further improve the Newton multidimensial 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 perfom a one dimensional newton optimization along the direction of the gradient. This method greatly increases the stability of the multidimensional Newton optimizer.

Listing 4.78: in file: `numeric.py`

```
def optimize_newton_multi_imporved(f, x, ap=1e-6, rp=1e-4, ns=20, h=10.0):
    """
    Finds the extreme of multidimensional function f near point x.

    Parameters
    f is a function that takes a list and returns a scalar
    x is a list

    Returns x, which maximizes of minimizes f(x)=0, as a list
    """
    x = Matrix.from_list([x]).t
    fx = f(x.values)
    for k in xrange(ns):
        (grad,H) = (gradient(f,x.values), hessian(f,x.values))
        if norm(H) < ap:
            raise ArithmeticError, 'unstable solution'
        (fx_old, x_old, x) = (fx, x, x-(1.0/H)*grad)
        fx = f(x.values)
        while fx>fx_old: # revert to steepest descent
            (fx, x) = (fx_old, x_old)
            norm_grad = norm(grad)
            (x_old, x) = (x, x - grad/norm_grad*h)
            (fx_old, fx) = (fx, f(x.values))
            h = h/2
        h = norm(x-x_old)*2
        if k>2 and h/2<max(ap,norm(x)*rp): return x.values
    raise ArithmeticError, 'no convergence'
```

4.9 *Non-linear fitting*

Finally we have all the ingredients to implement a very generic fitting function that will work linear and non-linear least squares.

Here we consider a generic experiment or simulated experiment which generates points of the form $(x_i, y_i \pm \delta y_i)$. Our goal is to minimize the χ^2 defined as

$$\chi^2(\mathbf{a}, \mathbf{b}) = \sum_i \left| \frac{y_i - f(x_i, \mathbf{a}, \mathbf{b})}{\delta y_i} \right|^2 \tag{4.137}$$

where the function f is known but depends on unknown parameters $\mathbf{a} = (a_0, a_1, ...)$ and $\mathbf{b} = (a_0, a_1, ...)$. In terms of these parameters the function f can be written as follows:

$$f(x, \mathbf{a}, \mathbf{b}) = \sum_j a_j f_j(x, \mathbf{b}) \tag{4.138}$$

Here is an example:

$$f(x, \mathbf{a}, \mathbf{b}) = a_0 e^{-b_0 x} + a_1 e^{-b_1 x} + a_2 e^{-b_2 x} + ... \tag{4.139}$$

The goal of our algorithm is to efficiently determine the parameters $\mathbf{a}$ and $\mathbf{b}$ which minimize the χ^2.

We proceed by defining the following two quantities:

$$\mathbf{z} = \begin{pmatrix} y_0 \ / \ \delta y_0 \\ y_1 \ / \ \delta y_1 \\ y_2 \ / \ \delta y_2 \\ ... \end{pmatrix} \tag{4.140}$$

and

$$A(\mathbf{b}) = \begin{pmatrix} f_0(x_0, \mathbf{b})/\delta y_0 & f_1(x_0, \mathbf{b})/\delta y_0 & f_2(x_0, \mathbf{b})/\delta y_0 & ... \\ f_0(x_1, \mathbf{b})/\delta y_1 & f_1(x_1, \mathbf{b})/\delta y_1 & f_2(x_1, \mathbf{b})/\delta y_1 & ... \\ f_0(x_2, \mathbf{b})/\delta y_2 & f_1(x_2, \mathbf{b})/\delta y_2 & f_2(x_2, \mathbf{b})/\delta y_2 & ... \\ ... & ... & ... & ... \end{pmatrix} \tag{4.141}$$

In terms of A and z the χ^2 can be rewritten as

$$\chi^2(\mathbf{a}, \mathbf{b}) = |A(\mathbf{b})\mathbf{a} - \mathbf{z}|^2 \tag{4.142}$$

We can minimize this function in a using the linear least squares algorithm, exactly:

$$\mathbf{a}(\mathbf{b}) = (A(\mathbf{b})A(\mathbf{b})^t)^{-1}(A(\mathbf{b})^t z) \tag{4.143}$$

We define a function which returns the minimum χ^2 for a fixed input $\mathbf{b}$:

$$g(\mathbf{b}) = \min_{\mathbf{a}} \chi^2(\mathbf{a}, \mathbf{b}) = \chi^2(\mathbf{a}(\mathbf{b}), \mathbf{b}) = |A(\mathbf{b})\mathbf{a}(\mathbf{b}) - \mathbf{z}|^2 \tag{4.144}$$

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 code below 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, than 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 $\mathbf{b}$ is missing then the fitting parameters are all linear and the algorithm reverts to regular linear least squares. Otherwise if run the more complex algorithm described above:

Listing 4.79: in file: `numeric.py`

```
def fit(data, fs, b=None, ap=1e-6, rp=1e-4, ns=200, constraint=None):
    if not isinstance(fs,(list,tuple)):
        def g(b, data=data, f=fs, constraint=constraint):
            chi2 = sum(((y-f(b,x))/dy)**2 for (x,y,dy) in data)
            if constraint: chi2+=constraint(b)
            return chi2
        if isinstance(b,(list,tuple)):
            b = optimize_newton_multi_imporved(g,b,ap,rp,ns)
        else:
            b = optimize_newton(g,b,ap,rp,ns)
        return b, g(b,data,constraint=None)
    elif not b:
        a, chi2, ff = fit_least_squares(data, fs)
        return a, chi2
    else:
        na = len(fs)
        def core(b,data=data,fs=fs):
            A = Matrix.from_list([[fs[k](b,x)/dy for k in xrange(na)] \
                                  for (x,y,dy) in data])
            z = Matrix.from_list([[y/dy] for (x,y,dy) in data])
            a = (1/(A.t*A))*(A.t*z)
            chi2 = norm(A*a-z)**2
            return a.values, chi2
        def g(b,data=data,fs=fs,constraint=constraint):
            a, chi2 = core(b, data, fs)
            if constraint:
```

```
                chi += constraint(b)
            return chi2
        b = optimize_newton_multi_imporved(g,b,ap,rp,ns)
        a, chi2 = core(b,data,fs)
        return a+b,chi2
```

Here is an example:

```
>>> data = [(i, i+2.0*i**2+300.0/(i+10), 2.0) for i in range(1,10)]
>>> fs = [(lambda b,x: x), (lambda b,x: x*x), (lambda b,x: 1.0/(x+b[0]))]
>>> ab, chi2 = fit(data,fs,[5])
>>> print(ab, chi2)
[0.999..., 2.000..., 300.000..., 10.000...] ...
```

Notice that the algotihms above take an extra argument, called a *constraint*. It is a function of the **b** parameters that is added to the computed χ^2. It allows us to perform Bayesian fits by taking into consideration *a priori* values and uncertainty for the **b** parameters. For example if we know that:

$$b_i \simeq \bar{b}_i \pm \delta b_i \tag{4.145}$$

we can define:

```
def constraint(b, bar_b, delta_b):
    return sum(((b[i]-bar_b[i])/delta_b[i])**2 for i in range(len(b)))
```

and pass the above function as a constraint. The effect of this constraint is that of adding an artificial contribution to the χ^2 when any of the b_i parameters departs from its *a priori* estimate, in units of the uncertainty about this estimate. This forces the fitter to stay close to expected values for the **b** parameters.

4.10 Integration

Consider the integral of $f(x)$ for x in domain $[a, b]$ which we normally represent as

$$I = \int_a^b f(x)dx \tag{4.146}$$

and which measures the area under the curve $y = f(x)$ delimited on the left by $x = a$ and on the right by $x = b$.

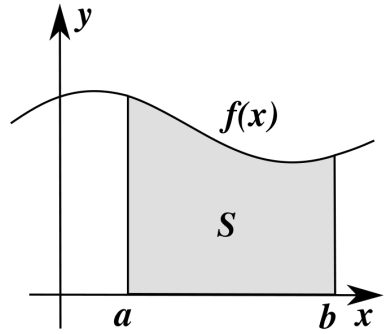

Figure 4.7: Visual representation of the concept of integral as area under a curve.

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:

$$I_n \simeq \sum_{i=0}^{i<n} \frac{h}{2}(f_i + f_{i+1}) \tag{4.147}$$

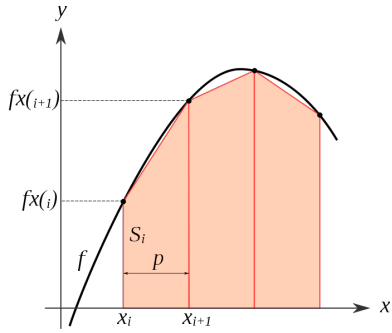

Figure 4.8: Visual representation of the trapeziod method for numerical integration.

If a function is discontinuous only in a finite number of points in the domain

$[a, b]$ then the following limit exists:

$$\lim_{n\to\infty} I_n \to I \tag{4.148}$$

We can implement the naive integration as function of N as follows:

Listing 4.80: in file: `numeric.py`

```
def integrate_naive(f, a, b, n=20):
    """
    Integrates function, f, from a to b using the trapezoidal rule
    >>> from math import sin
    >>> integrate(sin, 0, 2)
    1.416118...
    """
    a,b= float(a),float(b)
    h = (b-a)/n
    return h/2*(f(a)+f(b))+h*sum(f(a+h*i) for i in range(1,n))
```

And here we implement the limit by doubling the number of points until convergence is achieved:

Listing 4.81: in file: `numeric.py`

```
def integrate(f, a, b, ap=1e-4, rp=1e-4, ns=20):
    """
    Integrates function, f, from a to b using the trapezoidal rule
    converges to precision
    """
    I = integrate_naive(f,a,b,1)
    for k in range(1,ns):
        I_old, I = I, integrate_naive(f,a,b,2**k)
        if k>2 and norm(I-I_old)<max(ap,norm(I)*rp): return I
    raise ArithmeticError, 'no convergence'
```

We can test the convergence as follows:

Listing 4.82: in file: `numeric.py`

```
>>> from math import sin, cos
>>> print(integrate_naive(sin,0,3,n=2))
1.6020...
>>> print(integrate_naive(sin,0,3,n=4))
1.8958...
>>> print(integrate_naive(sin,0,3,n=8))
1.9666...
>>> print(integrate(sin,0,3))
```

```
1.9899...
>>> print(1.0-cos(3))
1.9899...
```

4.10.1 Quadrature

In the previous integration we have divided the domain $[a, b]$ into sub-domains and we have computed the area under the curve f in each sub-domain by approximating it with a trapezoid; i.e. we have 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 sub-domain by explicitly integrating the polynomial.

This is the basic idea of quadrature. For a sub-domain delimited by $(0, h)$ we can impose:

$$\int_0^h 1\mathrm{d}x \quad = h \quad = \sum_i c_i (hi/n)^0 \tag{4.149}$$

$$\int_0^h x\mathrm{d}x \quad = h^2/2 \quad = \sum_i c_i (hi/n)^1 \tag{4.150}$$

$$\ldots \quad \ldots \quad \ldots \tag{4.151}$$

$$\int_0^h x^{n-1}\mathrm{d}x \quad = h^n/n \quad = \sum_i c_i (hi/n)^2 \tag{4.152}$$

where c_i are coefficients to be determined.

Listing 4.83: in file: `numeric.py`

```
class QuadratureIntegrator:
    """
    Calculates the integral of the function f from points a to b
    using n Vandermonde weights and numerical quadrature.
    """
    def __init__(self,delta,order=4):
        h = float(delta)/(order-1)
        A = Matrix(order, order, fill = lambda r,c: (c*h)**r)
        s = Matrix(order, 1, fill = lambda r,c: (delta**(r+1))/(r+1))
        w = (1/A)*s
        self.packed = (h, order, w)
    def integrate(self,f,a):
        (h, order, w) = self.packed
```

```
        return sum(w[i,0]*f(a+i*h) for i in range(order))

def integrate_quadrature_naive(f,a,b,n=20,order=4):
    a,b = float(a),float(b)
    h = (b-a)/n
    q = QuadratureIntegrator((b-a)/n,order=order)
    return sum(q.integrate(f,a+i*h) for i in range(n))
```

Listing 4.84: in file: `numeric.py`

```
>>> from math import sin
>>> print(integrate_quadrature_naive(sin,0,3,n=2,order=2))
1.60208248595
>>> print(integrate_quadrature_naive(sin,0,3,n=2,order=3))
1.99373945223
>>> print(integrate_quadrature_naive(sin,0,3,n=2,order=4))
1.99164529955
```

4.10.2 *Differential equations*

WORK IN PROGRESS

4.11 *Artificial Intelligence and Machine Learning*

4.11.1 *Clustering Algorithms*

There are many algorithms available to cluster data. They are all based on empirical principles. 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 closer 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 which are joined, called *dendrogram*.
- *Centroid based clustering*. These algorithms require that each point be represented by a vector and each cluster is also represented by a vector (centroid of the cluster). Each each iteration a better estimation for the centroids is given. *k-means* clustering is an example of centroid based clustering. These algorithm require an a-priori knowledge of the number of clusters and return the position of the centroids as well 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 mush be known a priori) and determine the parameters of the distribution. It provides clustering because the distribution may be a sub of more than one localized distributions (each being a cluster).

Both k-means and distribution based clustering assume an a-priori knowledge about the data which often defies the purpose of using clustering: learn something we do now know about the data using an empirical

algorithm. They also require that the points be represented by vectors in an 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 inpractical.

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

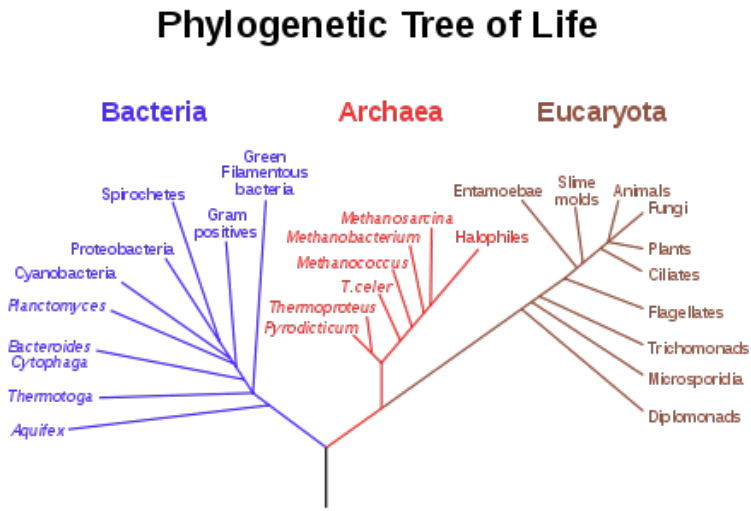

Figure 4.9: Example of a dendrogram.

The algorithm below 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 financial 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` is 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 which are vectors, and a given k, its result is almost identical 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 into 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 distanced 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.

In the algorithm `self.q` implements disjoints sets representing the set of clusters. `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 4.85: in file: `numeric.py`

```
class Cluster(object):
    def __init__(self,points,metric,weights=None):
        self.points, self.metric = points, metric
        self.k = len(points)
        self.w = weights or [1.0]*self.k
        self.q = dict((i,[i]) for i,e in enumerate(points))
        self.d = []
        for i in xrange(self.k):
            for j in xrange(i+1,self.k):
                m = metric(points[i],points[j])
                if not m is None:
                    self.d.append((m,i,j))
        self.d.sort()
        self.dd = []
```

```
    def parent(self,i):
        while isinstance(i,int): (parent, i) = (i, self.q[i])
        return parent, i
    def step(self):
        if self.k>1:
            # find new clusters to join
            (self.r,i,j),self.d = self.d[0],self.d[1:]
            # join them
            i,x = self.parent(i) # find members of cluster i
            j,y = self.parent(j) # find members if cluster j
            x += y               # join members
            self.q[j] = i        # make j cluster point to i
            self.k -= 1          # decrease cluster count
            # update all distances to new joined cluster
            new_d = [] # links not related to joined clusters
            old_d = {} # old links related to joined clusters
            for (r,h,k) in self.d:
                if h in (i,j):
                    a,b = old_d.get(k,(0.0,0.0))
                    old_d[k] = a+self.w[k]*r,b+self.w[k]
                elif k in (i,j):
                    a,b = old_d.get(h,(0.0,0.0))
                    old_d[h] = a+self.w[h]*r,b+self.w[h]
                else:
                    new_d.append((r,h,k))
            new_d += [(a/b,i,k) for k,(a,b) in old_d.items()]
            new_d.sort()
            self.d = new_d
            # update weight of new cluster
            self.w[i] = self.w[i]+self.w[j]
            # get new list of cluster memebrs
            self.v = [s for s in self.q.values() if isinstance(s,list)]
            self.dd.append((self.r,len(self.v)))
        return self.r, self.v

    def find(self,k):
        # if necessary start again
        if self.k<k: self.__init__(self.points,self.metric)
        # step until we get k clusters
        while self.k>k: self.step()
        # return list of cluster members
        return self.r, self.v
```

Given a set of points we can determine the most likely number of clusters representing the data we can make a plot of number of clusters vs distance and look for a a plateau in the plot. In correspondence of the plateau we can read from the y coordinate the number of clusters. This is done by the

function `cluster` in the above algorithm which returns the average distance between clusters and a list of clusters.

For example:

Listing 4.86: in file: `numeric.py`

```
>>> def metric(a,b):
...     return math.sqrt(sum((x-b[i])**2 for i,x in enumerate(a)))
>>> points = [[random.gauss(i % 5,0.3) for j in range(10)] for i in range(200)]
>>> c = Cluster(points,metric)
>>> r, clusters = c.find(1) # cluster all points until one cluster only
>>> draw(title='clustering example',xlab='distance',ylab='number of clusters',
...      linesets = [{'data':c.dd[150:]}], filename = 'clustering1.png')
>>> draw(title='clustering example (2d projection)',xlab='p[0]',ylab='p[1]',
...      ellisets = [{'data':[p[:2] for p in points]}],filename = 'clustering2.png'
    )
```

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

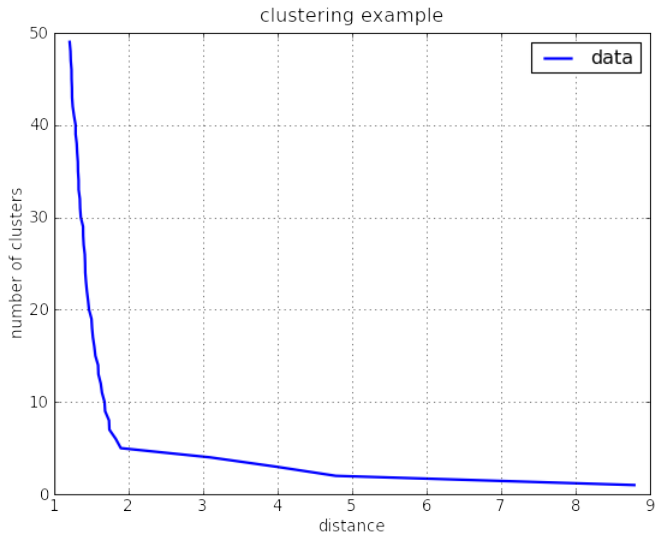

Figure 4.10: Number of clusters as function of the distance cut-off.

and, in fact, the last knee corresponds to 5 clusters. Although out points live in 10 dimensions we can try project them into 2 dimensions, and see the 5 clusters ("clustering2.png"):

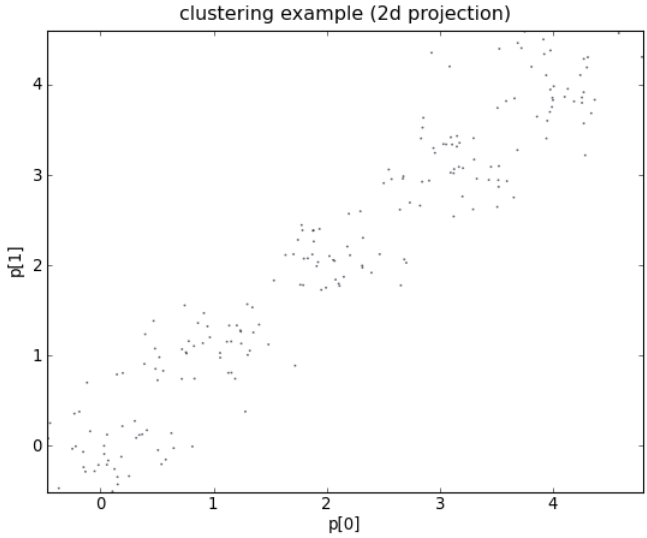

Figure 4.11: Data projected in 2 dimenions.

4.11.2 *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) brains. It is usually employed in pattern recognition. The network consists of of a set of simulated neurons, connected by links (synapses). Some linkes 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, one *output layer* of neurons connected only with the output and previous layers, one 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 funciton of its inputs. The exact shape f that funciton depends on the network and on parameters which can adjusted. A

common choice is when the output is

$$\text{output}_{ij} = \tanh(\sum_k a_{ijk}\text{input}_{ij}) \tag{4.153}$$

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

The network is trained by providing an input and adjusting the characteristics of each neuron in order to produce the expected output. The network is trained iteratively until its parameters converge (if they converge) and then it is ready to make predictions. We say the network has learned from the training data set.

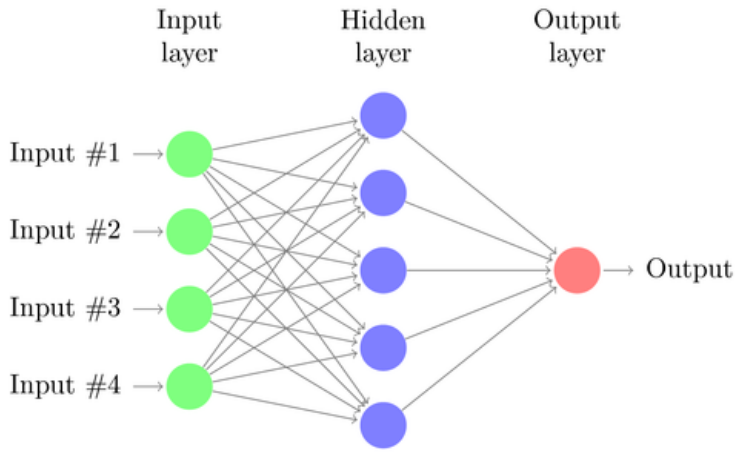

Figure 4.12: Example of a minimalist neural network.

Listing 4.87: in file: `numeric.py`

```
class NeuralNetwork:
    """
    Back-Propagation Neural Networks
    Placed in the public domain.
    Original author: Neil Schemenauer <nas@arctrix.com>
    Modified by: Massimo Di Pierro
    Read more: http://www.ibm.com/developerworks/library/l-neural/
    """

```

```
10      @staticmethod
11      def rand(a, b):
12          """ calculate a random number where:  a <= rand < b """
13          return (b-a)*random.random() + a
14
15      @staticmethod
16      def sigmoid(x):
17          """ our sigmoid function, tanh is a little nicer than the standard 1/(1+e^-
                x) """
18          return math.tanh(x)
19
20      @staticmethod
21      def dsigmoid(y):
22          """ # derivative of our sigmoid function, in terms of the output (i.e. y) "
                ""
23          return 1.0 - y**2
24
25      def __init__(self, ni, nh, no):
26          # number of input, hidden, and output nodes
27          self.ni = ni + 1 # +1 for bias node
28          self.nh = nh
29          self.no = no
30
31          # activations for nodes
32          self.ai = [1.0]*self.ni
33          self.ah = [1.0]*self.nh
34          self.ao = [1.0]*self.no
35
36          # create weights
37          self.wi = Matrix(self.ni, self.nh, fill=lambda r,c: self.rand(-0.2, 0.2))
38          self.wo = Matrix(self.nh, self.no, fill=lambda r,c: self.rand(-2.0, 2.0))
39
40          # last change in weights for momentum
41          self.ci = Matrix(self.ni, self.nh)
42          self.co = Matrix(self.nh, self.no)
43
44      def update(self, inputs):
45          if len(inputs) != self.ni-1:
46              raise ValueError, 'wrong number of inputs'
47
48          # input activations
49          for i in range(self.ni-1):
50              self.ai[i] = inputs[i]
51
52          # hidden activations
53          for j in range(self.nh):
54              s = sum(self.ai[i] * self.wi[i,j] for i in range(self.ni))
55              self.ah[j] = self.sigmoid(s)
56
```

```
        # output activations
        for k in range(self.no):
            s = sum(self.ah[j] * self.wo[j,k] for j in range(self.nh))
            self.ao[k] = self.sigmoid(s)
        return self.ao[:]

    def back_propagate(self, targets, N, M):
        if len(targets) != self.no:
            raise ValueError, 'wrong number of target values'

        # calculate error terms for output
        output_deltas = [0.0] * self.no
        for k in range(self.no):
            error = targets[k]-self.ao[k]
            output_deltas[k] = self.dsigmoid(self.ao[k]) * error

        # calculate error terms for hidden
        hidden_deltas = [0.0] * self.nh
        for j in range(self.nh):
            error = sum(output_deltas[k]*self.wo[j,k] for k in range(self.no))
            hidden_deltas[j] = self.dsigmoid(self.ah[j]) * error

        # update output weights
        for j in range(self.nh):
            for k in range(self.no):
                change = output_deltas[k]*self.ah[j]
                self.wo[j,k] = self.wo[j,k] + N*change + M*self.co[j,k]
                self.co[j,k] = change
                #print(N*change, M*self.co[j,k])

        # update input weights
        for i in range(self.ni):
            for j in range(self.nh):
                change = hidden_deltas[j]*self.ai[i]
                self.wi[i,j] = self.wi[i,j] + N*change + M*self.ci[i,j]
                self.ci[i,j] = change

        # calculate error
        error = sum(0.5*(targets[k]-self.ao[k])**2 for k in range(len(targets)))
        return error

    def test(self, patterns):
        for p in patterns:
            print(p[0], '->', self.update(p[0]))

    def weights(self):
        print('Input weights:')
        for i in range(self.ni):
            print(self.wi[i])
```

```
        print
        print('Output weights:')
        for j in range(self.nh):
            print(self.wo[j])

    def train(self, patterns, iterations=1000, N=0.5, M=0.1, check=False):
        # N: learning rate
        # M: momentum factor
        for i in xrange(iterations):
            error = 0.0
            for p in patterns:
                inputs = p[0]
                targets = p[1]
                self.update(inputs)
                error = error + self.back_propagate(targets, N, M)
            if check and i % 100 == 0:
                print('error %-14f' % error)
```

In the following example we teach the network the XOR function, we create a network with 2 inputs, 2 intermediate neurons and 1 output. We train it and check what it learned:

Listing 4.88: in file: `numeric.py`

```
>>> pat = [[[0,0], [0]], [[0,1], [1]], [[1,0], [1]], [[1,1], [0]]]
>>> n = NeuralNetwork(2, 2, 1)
>>> n.train(pat)
>>> n.test(pat)
[0, 0] -> [0.00...]
[0, 1] -> [0.98...]
[1, 0] -> [0.98...]
[1, 1] -> [-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 4.89: in file: `test.py`

```
>>> storage = PersistentDictionary('sp100.sqlite')
>>> v = [day['arithmetic_return']*300 for day in storage['AAPL/2011'][1:]]
>>> pat = [[v[i:i+5],[v[i+5]]] for i in range(len(v)-5)]
>>> n = NeuralNetwork(5, 5, 1)
>>> n.train(pat)
>>> predictions = [n.update(item[0]) for item in pat]
>>> success_rate = sum(1.0 for i,e in enumerate(predictions)
```

```
...                     if e[0]*v[i+5]>0)/len(pat)
```

The learning process depends on the random number generator therefore some times, 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

4.11.3 Genetic Algorithms

Here we consider a simple example of genetic programming, mostly to explain the idea since this specific problem is better solved using other techniques.

We have a population of chromosomes where 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.

We loop at each iteration only the top ranking chromosomes in the population survive. The top 10 mate with each other and their offspring constitutes the population for the next iteration. When two members of the population mate, the new born 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 basis.

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

In the example below 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.

```
from random import randint, choice

class Chromosome:
    alphabet = 'ATGC'
    size = 32
    mutations = 2
    def __init__(self,father=None,mother=None):
        if not father or not mother:
            self.dna = [choice(self.alphabet) for i in range(self.size)]
```

```
        else:
            self.dna = father.dna[:self.size/2]+mother.dna[self.size/2:]
            for mutation in range(self.mutations):
                self.dna[randint(0,self.size-1)] = choice(self.alphabet)
    def fitness(self,target):
        return sum(1 for i,c in enumerate(self.dna) if c==target.dna[i])

def top(population,target,n=10):
    table = [(chromo.fitness(target), chromo) for chromo in population]
    table.sort(reverse = True)
    return [row[1] for row in table][:n]

def oneof(population):
    return population[randint(0, len(population)-1)]

def main():
    GENERATIONS = 10000
    OFFSPRING = 20
    SEEDS = 20
    TARGET = Chromosome()

    population = [Chromosome() for i in range(SEEDS)]
    for i in range(GENERATIONS):
        print('\n\nGENERATION:',i)
        print(0, TARGET.dna)
        fittest = top(population,TARGET)
        for chromosome in fittest: print(i,chromosome.dna)
        if max(chromo.fitness(TARGET) for chromo in fittest)==Chromosome.size:
            print('SOLUTION FOUND')
            break
        population = [Chromosome(father=oneof(fittest),mother=oneof(fittest)) \
                      for i in range(OFFSPRING)]

if __name__=='__main__': main()
```

Notice that this algorithm can easily be modified to accommodate other fitness metrics and a DNA which consists of a data structure other than a sequence of "ATGC" symbols. The only trickery is finding a proper mating algorithm which preserves some of the fitness features of the parents into the DNA of their offspring. If this does not happen, each next generation loses the fitness properties gained by their parents this 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

algorithm to further explore the space of all possible DNA sequences. This algorithm is very slow to converge because it does not take any advantage of the knowledge of the location of the DNA bases that actually agree with the target.

5
Functional Analysis

WORK IN PROGRESS

6

Probability and Statistics

6.1 Probability

Probability derives from the Latin *probare* (to prove, or to test). The word probable 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
$$\text{Prob}(A) \stackrel{def}{=} \lim_{N \to \infty} \frac{N_A}{N} \tag{6.1}$$
This definition actually requires that one perform a large experiment, if not an infinite, 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

$$\text{Prob}(A) \stackrel{def}{=} \frac{c(A)}{c(S)} \tag{6.2}$$

  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 that fact that a real dice may be biased. The former definition takes into account this possibility while 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 3 possible ways, therefore it is represented by a subset of S.

The axiomatic definition makes it easy to prove theorems, for example:
Theorem 6.1.1. *If $S = A \cup A^c$ and $A \cap A^c = 0$ then Prob$(A) = 1 - $ Prob(A^c)*

Python has a module called `random` which can generate random nummbers and we can use to perform some experiments. Let's simulate a dice with 6 possible outcomes. We can use the frequentist definition:

Listing 6.1: in file: `numeric.py`

```
>>> import random
>>> S = [1,2,3,4,5,6]
>>> def Prob(A, S, N=1000):
...     return float(sum(random.choice(S) in A for i in range(N)))/N
>>> Prob([6],S)
0.166
>>> Prob([1,2],S)
0.308
```

Here `Prob(A)` compute 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.

Listing 6.2: in file: `numeric.py`

```
>>> def Prob(A, S): return float(len(A))/len(S)
>>> Prob([6],S)
0.16666666666666666
>>> Prob([1,2],S)
0.3333333333333333
```

As stated before the latter is more precice because it produces results for an "ideal" dice while the frequentist approach produces results for a real dice (in our case a simulated dice).

6.1.1 *Conditional probability and independence*

We define $\text{Prob}(A|B)$ as the probability of event A given event B and we write

$$\text{Prob}(A|B) \stackrel{def}{=} \frac{\text{Prob}(AB)}{\text{Prob}(B)} \tag{6.3}$$

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 $\text{Prob}(A|B) = \text{Prob}(A)$ is independent of B then we say that A and B are uncorrelated and from eq.(6.3) we conclude

$\text{Prob}(AB) = \text{Prob}(A)\text{Prob}(B)$ therefore the probability that two uncorrelated events occur is the product of the probability that each individual event occurs.

We can experiment with conditional probability using python. Let's consider two dices, X and Y. The space of all possible outcomes is given by $S^2 = S \times S$. And we are interested in the probabily of the second dice giving a 6 given that the first dice is also a 6.

Listing 6.3: in file: `numeric.py`

```
>>> def cross(u,v): return [(i,j) for i in u for j in v]
>>> def Prob_conditional(A, B, S): return Prob(cross(A,B),cross(S,S))/Prob(B,S)
>>> Prob_conditional([6],[6],S)
0.16666666666666666
```

Not surprising we find that `Prob_conditional([6],[6],S)` produces the same result as `Prob([6],S)` because the two dices are independent.

In fact we say that two sets of events A and B are independent if and only if $\text{P}(A|B) = \text{P}(A)$.

6.1.2 *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 $X = x$

$$p(x) \stackrel{def}{=} \text{Prob}(X = x) \tag{6.4}$$

We also define the *expectation value* of any function of a discrete random variable $f(X)$ as

$$E[f(X)] \stackrel{def}{=} \sum_i f(x_i)p(x_i) \tag{6.5}$$

where i loops all possible variables x_i of the random variable X.

Example 6.1.1. X is the random variable associated to the outcome of rolling

a dice, $p(x) = 1/6$ if $x = 1, 2, 3, 4, 5$ or 6 and $p(x) = 0$ otherwise,

$$E[X] = \sum_i x_i p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} x_i \frac{1}{6} = 3.5 \tag{6.6}$$

and

$$E[(X-3.5)^2] = \sum_i (x_i - 3.5)^2 p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} (x_i - 3.5)^2 \frac{1}{6} = 2.9167 \tag{6.7}$$

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

$$\sigma_X = E[X^2] - E[X]^2 \tag{6.8}$$

For discerete random variables we can implement these definitions as follows:

Listing 6.4: in file: `numeric.py`

```
def E(f,S): return float(sum(f(x) for x in S))/len(S)
def mean(X): return E(lambda x:x, X)
def variance(X): return E(lambda x:x**2, X) - E(lambda x:x, X)**2
def sd(X): return sqrt(variance(X))
```

which we can test with a simulated experiment:

Listing 6.5: in file: `numeric.py`

```
>>> S = [random.random()+random.random() for i in range(100)]
>>> print(mean(S))
1.000...
>>> print(sd(S))
0.4...
```

As another 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.

Listing 6.6: in file: `numeric.py`

```
>>> S = [1,2,3,4,5,6]
>>> def payoff(x): return 20.0 if x==6 else -5.0
>>> print(E(payoff,S))
-0.83333...
```

The average expected payoff is $-0.83...$ which means that in average we should expect to lose at this game.

6.1.3 Continuous random variables

If S is the space of all possible outcomes of an experiment and we associate a real number X to 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 $X \leq x$

$$F(x) \stackrel{def}{=} \text{Prob}(X \leq x) \tag{6.9}$$

If S is a continuous set and X is a continuous random variable then we define a *probability density* or *distribution* $f(x)$ as

$$p(x) \stackrel{def}{=} \frac{dF(x)}{dx} \tag{6.10}$$

and the probability that X falls into an interval $[a, b]$ can be computed as

$$\text{Prob}(a \leq X \leq b) = \int_a^b p(x)dx \tag{6.11}$$

We also define the *expectation value* of any function of a random variable $f(X)$ as

$$E[f(X)] = \int_{-\infty}^{\infty} f(x)p(x)dx \tag{6.12}$$

Example 6.1.2. X is a uniform random variable (probability density $p(x)$ equal to 1 if $x \in [0, 1]$, equal to o otherwise)

$$E[X] = \int_{-\infty}^{\infty} xp(x)dx = \int_0^1 xdx = \frac{1}{2} \tag{6.13}$$

and

$$E[(X - \frac{1}{2})^2] = \int_{-\infty}^{\infty} (x - \frac{1}{2})^2 p(x)dx = \int_0^1 (x^2 - x + \frac{1}{4})dx = \frac{1}{12} \tag{6.14}$$

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

$$\sigma_X^2 = E[X^2] - E[X]^2 \tag{6.15}$$

Theorem 6.1.2. $\int_{-\infty}^{\infty} p(x)dx = 1$.

Proof: By definition

$$F(\infty) \stackrel{def}{=} \text{Prob}(X \leq \infty) = 1 \tag{6.16}$$

therefore

$$\text{Prob}(-\infty \leq X \leq \infty) = \int_{-\infty}^{\infty} p(x)dx = 1 \tag{6.17}$$

Theorem 6.1.3. *$E[X]$ is a linear operator.*

Proof:

$$\begin{aligned} E[aX+b] &= \int_{-\infty}^{\infty} (ax+b)p(x)dx && (6.18)\\ &= a\int_{-\infty}^{\infty} xp(x)dx + b\int_{-\infty}^{\infty} p(x)dx && (6.19)\\ &= aE[X] + b && (6.20) \end{aligned}$$

One important consequence of all these formulas is that if we have a function f and a domain $[a,b]$, we can compute it is integral by choosing p to be a uniform distribution with values exclusively between a and b.

$$E[f] = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx \tag{6.21}$$

We can also compute the same integral by using the definition of expectation value for a discerete distribution:

$$E[f] = \sum_{x_i} f(x_i)p(x_i) = \frac{1}{N}\sum_{x_i} f(x_i) \tag{6.22}$$

where x_i are N random points drawn from the uniform distribution p defined above. In fact, in the large N limit:

$$\lim_{N\to\infty} \frac{1}{N}\sum_{x_i} f(x_i) = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx \tag{6.23}$$

We can verify the above relation numerically for a special case:

Listing 6.7: in file: `numeric.py`

```
>>> from math import sin
>>> def integrate_mc(f,a,b,N=1000):
...     return sum(f(random.uniform(a,b)) for i in range(N))/N*(b-a)
>>> print(integrate_mc(sin,0,pi,N=10000))
2.000....
```

This is the simplest case of Monte Carlo integration, which is the subject of a following chapter.

6.1.4 *Multiple random variables, covariance and correlations*

Given two random variables, X and Y, we define the covariance (cov) and the correlation (corr) between them as

$$\text{cov}(X,Y) \stackrel{def}{=} E[(X-\mu_X)(Y-\mu_Y)] = E[XY] - E[X]E[Y] \quad (6.24)$$

$$\text{corr}(X,Y) \stackrel{def}{=} \text{cov}(X,Y)/(\sigma_X\sigma_Y) \quad (6.25)$$

Theorem 6.1.4. $\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y + 2cov(X,Y)$
Theorem 6.1.5. *If X and Y are independent then* $cov(X,Y) = corr(X,Y) = 0$.

Proof:

$$\begin{aligned} E[XY] &= \int\int xyp(x,y)dxdy & (6.26)\\ &= \int\int xyp(x)p(y)dxdy & (6.27)\\ &= \left[\int xp(x)dx\right]\left[\int yp(y)dy\right] & (6.28)\\ &= E[X]E[Y] & (6.29)\end{aligned}$$

therefore

$$\text{cov}(X,Y) = E[XY] - E[X]E[Y] = 0 \quad (6.30)$$

Notice that the reverse is not true. Even if the correlation and the covariance are zero, X and Y may be dependent.
Theorem 6.1.6. *If X and Y are completely correlated or anticorrelated* ($Y = \pm X$) *then* $cov(X,Y) = \pm\sigma^2_X$ *and* $corr(X,Y) = \pm 1$.

Proof:

$$\begin{aligned} \mathrm{cov}(X,Y) &= E[(X-\mu_X)(Y-\mu_Y)] && (6.31)\\ &= E[(X-\mu_X)(\pm X \mp \mu_X)] && (6.32)\\ &= \pm E[(X-\mu_X)(X-\mu_X)] && (6.33)\\ &= \pm\sigma_X^2 && (6.34) \end{aligned}$$

Notice that the correlation lies always in the range $[-1,1]$.

Theorem 6.1.7. *For uncorrelated random variables X_i*

$$\begin{aligned} E[\sum_i a_i X_i] &= \sum_i a_i E[X_i] && (6.35)\\ E[(\sum_i X_i)^2] &= \sum_i E[X_i^2] && (6.36) \end{aligned}$$

We can define covariance and correlation for discerete distributions:

Listing 6.8: in file: `numeric.py`

```
def cov(X,Y):
    return sum(X[i]*Y[i] for i in range(len(X)))/len(X) - mean(X)*mean(Y)
def corr(X,Y):
    return cov(X,Y)/sd(X)/sd(Y)
```

Here is an example:

Listing 6.9: in file: `numeric.py`

```
>>> X = []
>>> Y = []
>>> for i in range(1000):
...     u = random.random()
...     X.append(u+random.random())
...     Y.append(u+random.random())
>>> print(mean(X))
0.989780352018
>>> print(sd(X))
0.413861115381
>>> print(mean(Y))
1.00551523013
>>> print(sd(Y))
0.404909628555
>>> print(cov(X,Y))
0.0802804358268
>>> print(corr(X,Y))
0.479067813484
```

6.1.5 Weak Law of large numbers

Theorem 6.1.8. *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, for any $\varepsilon > 0$*

$$\lim_{n\to\infty} P\left(\left|\frac{X_1 + X_2 + ... + X_n}{n} - \mu\right| > \varepsilon\right) = 0 \tag{6.37}$$

In other words, this theorem says that "for any nonzero margin ε specified, no matter how small, with a sufficiently large sample there will be a very high probability that the average of the observations will be close to the expected value, that is, within the margin."

6.1.6 Strong Law of large numbers

Theorem 6.1.9. *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*

$$\lim_{n\to\infty} \frac{X_1 + X_2 + ... + X_n}{n} = \mu \tag{6.38}$$

This is a more strict theorem than the previous one and it can rephrased as "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."

6.1.7 Central Limit Theorem

This is one of the most important theorems concerning distributions. Is states that

Theorem 6.1.10. *If $X_1, X_2, ...X_n$ are a sequence of random variables with finite means, μ_i and finite variance, σ_i^2 than*

$$Y = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} X_i \tag{6.39}$$

Follows a Gaussian distribution with mean and variance:

$$\mu = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \mu_i \tag{6.40}$$

$$\sigma^2 = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \sigma_i^2 \tag{6.41}$$

We can numerically verify this for a the simple case where X_i are uniform random variables with mean equal 0.

Listing 6.10: in file: `numeric.py`

```
>>> def added_uniform(n): return sum([random.uniform(-1,1) for i in range(n)])/n
>>> def make_set(n,m=10000): return [added_uniform(n) for j in range(m)]
>>> draw(title='Central Limit Theorem',xlab='y',ylab='p(y)',
...      histsets = [{'label':'N=1','data':make_set(1)}],
...      filename='images/central1.png')
>>> draw(title='Central Limit Theorem',xlab='y',ylab='p(y)',
...      histsets = [{'label':'N=2','data':make_set(2)}],
...      filename='images/central2.png')
>>> draw(title='Central Limit Theorem',xlab='y',ylab='p(y)',
...      histsets = [{'label':'N=4','data':make_set(4)}],
...      filename='images/central4.png')
>>> draw(title='Central Limit Theorem',xlab='y',ylab='p(y)',
...      histsets = [{'label':'N=5','data':make_set(8)}],
...      filename='images/central8.png')
```

This theoream 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:

$$p(x) \underset{x\to\infty}{\sim} \frac{1}{|x|^{1+\alpha}}, \qquad 0 < \alpha < 2 \tag{6.42}$$

6.1.8 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

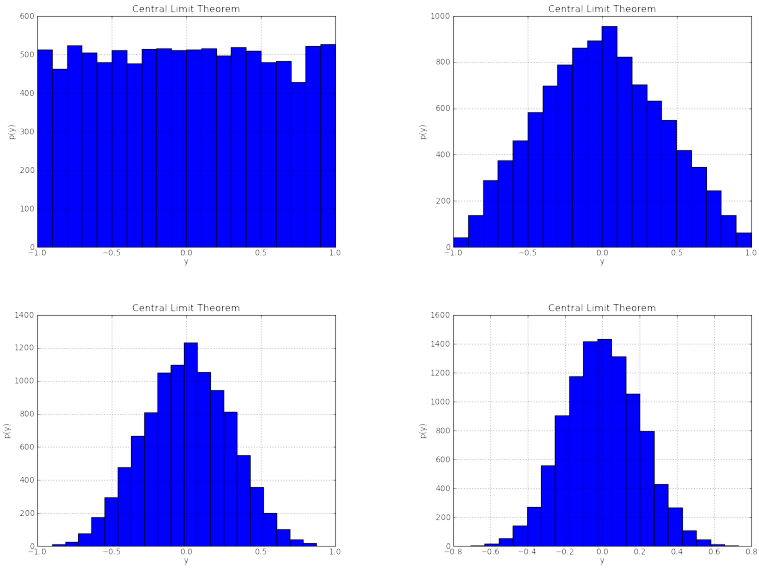

Figure 6.1: ...

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

$$\delta\mu = \sqrt{\frac{\sigma^2}{N-1}} \tag{6.43}$$

where $\sigma^2 = E[(X-\mu)^2] = \frac{1}{N}\Sigma_i (X_i - \mu)^2$.

6.2 *Combinatorics and discrete random variables*

Often, in order 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.

6.2.1 *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), etc. until we are left with no free sockets and $n-m$ unused plugs, therefore there are

$$n!/(n-m)! = n(n-1)(n-2)...(n-m+1) \tag{6.44}$$

  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), etc. until we are left with no spare plugs and $m-n$ free sockets, therefore there are

$$m!/(m-n)! = m(m-1)(m-2)...(m-n+1) \tag{6.45}$$

  combinations. Note that if $m = n$ than case 1 and case 2 agree as expected.

6.2.2 *Equivalent plugs in different sockets*

If we have n equivalient 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

$$\binom{m}{n} = \frac{m!}{(m-n)!n!} \tag{6.46}$$

ways to place n equivalent plugs into m different sockets. Note that if $m = n$

$$\binom{n}{n} = \frac{n!}{(n-n)!n!} = 1 \tag{6.47}$$

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 office holders? Think of each office as a socket and each person as a plug therefore the number combinations is $20!/(20-4)! \simeq 1.2 \times 10^5$

6.2.3 *Colored Cards*

We have 48 cards, 24 black and 24 red. We shuffle the cards and pick three.

- What is the probability that they are all red?

$$\text{Prob}(3red) = \frac{24}{48} \times \frac{23}{47} \times \frac{22}{46} = \frac{11}{94} \tag{6.48}$$

- What is the probability that they are all black?

$$\text{Prob}(3black) = \text{Prob}(3red) = \frac{11}{94} \tag{6.49}$$

- What is the probability that they are not all black nor all red?

$$\begin{aligned} \text{Prob}(mixture) &= 1 - \text{Prob}(3red \cup 3black) && (6.50)\\ &= 1 - \text{Prob}(3red) - \text{Prob}(3black) && (6.51)\\ &= 1 - 2\frac{11}{94} && (6.52)\\ &= \frac{36}{47} && (6.53) \end{aligned}$$

Here is an example of how we can simulate the deck of cards using Python to compute an answer to the last questions:

Listing 6.11: in file: `tests.py`

```
>>> def make_deck(): return [color for i in range(24) for color in ('red','black')]
>>> def make_shuffled_deck(): return random.shuffle(make_deck())
>>> def pick_three_cards(): return make_shuffled_deck()[:3]
>>> def simulate_cards(n=1000):
...     counter = 0
...     for k in range(n):
...         c = pick_three_cards()
...         if not (c[0]==c[1] and c[1]==c[2]): counter += 1
...     return float(counter)/n
>>> print(simulate_cards())
```

6.2.4 *Typical error*

The typical error in computing probabilities is mixing a priori probability with information about past events. For example we consider the problem above. We see the first two cards and they are all red. What is the probability that the third one is also red?

- **Wrong answer**: The probability that they are all red is $\text{Prob}(3red) = 11/94$ therefore the probability that the third one is red is $1/8$.
- **Correct answer**: Since we know that the first two cards are red then the third card must belong to a set of (24 black cards + 22 red cards) therefore the probability that it is red is

$$\text{Prob}(red) = \frac{22}{22+24} = \frac{11}{23} \tag{6.54}$$

7
Random Numbers and Distributions

We have seen how using the Python `random` module we can generate uniform random numbers. It can also generate random numbers following other distributions. Here we want to understand how random numbers are generated.

7.1 Randomness, Determinism, Chaos and Order

Before we proceed further there are four important concepts that we need to define because of their implications:

- **Randomness** is the characteristic of a process whose outcome is unpredictable (i.e., 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 (i.e., 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 (i.e., if I am on the top of a hill and I throw the ball in 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 deterministic it may happen

that a small variation in the initial conditions causes a big difference in the possible deterministic evolution of the system. Therefore the outcome of a process may depend on a tiny variation of initial parameters and 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 (i.e., 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.

7.2 *Real Randomness*

Note that randomness does not necessarily come from chaos. Randomness exists in nature. i.e. a radioactive atom "decays" into a different atom at some random time. i.e. an atom of Carbon 14 decays into Nitrogen 14 by emitting an electron and a neutrino

$$ {}^{14}_{6}C \longrightarrow {}^{14}_{7}N + e^{-} + \overline{\nu}_e \tag{7.1} $$

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 matter, in its most elementary known form,

it is described by quantum physics and randomness is a fundamental characteristics of quantum systems, not a consequence of our ability to describe them in a deterministic way.

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 but, rather, order and determinism emerge from the microscopic randomness while chaos originates from the complexity of the system.

Since 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 times when we observe a atom decays:

$$t_0, t_1, t_2, t_3, t_4, t_5, ... \tag{7.2}$$

One could study the probability distribution of the t_i and would find that they follow an exponential probability distribution like

$$\text{Prob}(t_i = t) = \lambda e^{-\lambda t} \tag{7.3}$$

where $t_0 = 1/\lambda$ is the decay time characteristic of the particular type of atom. One characteristics 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 $t_i < t_{i-1}$.

7.2.1 *Memoryless to Bernoulli distribution*

Given the sequence $\{t_i\}$ with exponential distribution we can build a random sequence of 0s and 1s (Bernoulli distribution) by applying the following formula

$$x_i = \begin{cases} 1 & \text{if } t_i > t_{i-1} \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

Note that the above 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 on t_j for any $j < i$ (memoryless distribution).

7.2.2 *Bernoulli to uniform distribution*

In order to map a Bernoulli distribution into a uniform distribution we need to determine the precision (resolution in # of bits) of the numbers we wish to generate. In this example we will assume 5 bits. In practical applications we want to use the IEEE-754 standard representation.

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 we repeat the process on the sub-segment. Since 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

$$01011110110101010101111010 1 \tag{7.5}$$

and we perform the following steps:

- break the sequence into chunks of 5 bits

$$01011\text{-}11011\text{-}01010\text{-}10101\text{-}11101\text{-}..... \tag{7.6}$$

- map each chunk $a_1a_2a_3a_4a_5$ into $x = \sum_{k=1}^{5} a_k 2^{-k}$

$$0.\,25195\text{-}0.\,75195\text{-}0.25000\text{-}0.\,25195\text{-}0.87500\text{-}... \tag{7.7}$$

A uniform random number generator is usually the first step towards building any other random number generator.

Other physical processes can be used to generate real random numbers using a similar process. Some mircoprocessors can generate random numbers from random temperature fluctuations. An unpredictable source of randomness is called an *entropy source*.

7.3 Entropy Generators

The Linux/Unix operating system provides its own entropy source accessible via "/dev/urandom". Once can read random characters from this data source and the can be trated as random. This data source is available in Python via `os.urandom()`.

Here we define a class that can access this entropy source and uses it to generate uniform random numbers using the same process outlined for the the radioactive days:

```
class URANDOM(object):
    def __init__(self, data=None):
        if data: open('/dev/urandom','wb').write(str(data))
    def random(self):
        import os
        n = 16
        random_bytes =  os.urandom(n)
        random_integer = sum(ord(random_bytes[k])*256**k for k in range(n))
        random_float = float(random_integer)/256**n
```

Notice how the constuctor allows to further randomize the data source by writing into it any data specified by the user. Also notice how the `random()` method reads 16 bytes from the stream (using `os.urandom()`, converts each into a 8-bits integers, combines them into a 128-bits integer and then converts it to a float by diving by 256^16.

While the numbers generated in this way are indeed random, they are not really uniformly distributed because there may be a bias in the urandom renerator.

7.4 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 reproducable, therefore it cannot be used to generate randomness but it can generate pseudo-randomness. The outputs of pseudorandom 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, pseudorandom 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 good enough.

7.4.1 Linear congruential generator

Here is probably the simples possible pseudo random number generator:

$$x_i = (ax_{i-1} + c) \mathrm{mod} m \tag{7.8}$$

$$y_i = x_i / m \tag{7.9}$$

With the choice $a = 65539$, $c = 0$ and $m = 2^{31}$ this generator is called RANDU and it is of historical implementation 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:

Listing 7.1: in file: `numeric.py`

```
class LCG(object):
    def __init__(self,seed,a=66539,c=0,m=2**31):
        self.x = seed
        self.a, self.c, self.m = a, c, m
    def next(self):
        self.x = (self.a*self.x + self.c) % self.m
        return self.x
    def random(self):
        return float(self.next())/self.m
```

which we can test with:

```
>>> randu = LCG(seed=1071914055)
>>> for i in range(10): print randu.random()
...
```

The output numbers "look" random but are not really random. Running the same code with the same seed generates the same output. Notice that:

- 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 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 depend on some parameters (for example a and m). Some choices of the parameters lead to trivial random number generators. In general some choices are better then others and a few are optimal. In particular the value 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 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 a 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 Marsenne prime number) because it can be proven that m is in fact the period of the generator:

$$x_i = (7^5 x_{i-1}) \mathrm{mod} (2^{31} - 1) \tag{7.10}$$

Here are some examples of LCG 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.

7.4.2 *PRNGs in cryptography*

Random numbers find many applications in cryptography. Think for example of the problem of generating a random password or 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, namely that its output should pass all statistical tests for randomness, a CSPRNG must have two extra properties:

- it should be difficult to predict the output of the CSPRNG, wholly or partially, from examining previous outputs, and in particular
- 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, as whilst they appear random to statistical tests, 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 Shub has the strongest security proofs, though it is slow. Most stream ciphers work by generating a pseudorandom stream of bits that are XORed with the message; this stream can be used as a good CSPRNG (thought not always: see RC4 cipher). A secure block cipher can also be converted into a CSPRNG by running it in counter mode. This is done by choosing an arbitrary key and encrypting a zero, then encrypting a 1, then encrypting a 2, etc. The counter can also be started at an arbitrary number other than zero. Obviously, the period will be 2n for an n-bit block cipher. Finally, there are PRNGs that have been designed to be cryptographically secure. One example is ISAAC (based on a variant of the RC4 cipher) which is fast and has an expected cycle length of 28295 and for which no successful attack in a reasonable time has yet been found.

Many pseudo-random generators have the form

$$x_i = f(x_{i-1}, x_{i-2}, ..., x_{i-k}) \tag{7.11}$$

i.e., 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 have to be fast. The second requirement means that if a malicious agent discovers a random number used as a key, he/she cannot easily compute all previous keys generated using the same PRNG.

7.4.3 *Multiplicative recursive generator*

Another modification of the multiplicative congruential generator is the following:

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + ... + a_k x_{i-k}) \mathrm{mod} m \tag{7.12}$$

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

$$(2^{31} - 1)^5 - 1 \simeq 4.56 \times 10^{46} \tag{7.13}$$

7.4.4 *Lagged Fibonacci generator*

$$x_i = (x_{i-j} + x_{i-k}) \mathrm{mod} m \tag{7.14}$$

This is similar to the multiplicative recursive generator above. If m is prime and $j \neq k$ the period can be as large as $m^k - 1$.

7.4.5 *Marsaglia's add-with-carry generator*

$$x_i = (x_{i-j} + x_{i-k} + c_i) \mathrm{mod} m \tag{7.15}$$

where $c_1 = 0$ and $c_i = 1$ if $(x_{i-1-j} + x_{i-1-k} + c_{i-1}) < m$, 0

7.4.6 *Marsaglia's subtract-and-borrow generator*

$$x_i = (x_{i-j} - x_{i-k} - c_i) \mathrm{mod} m \tag{7.16}$$

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.

7.4.7 *Luescher'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

$$v_i = (x_i, x_{i+1}, ..., x_{i+k}) \tag{7.17}$$

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 $x_i, x_{i+1}, ..., x_{i+k}$. Luescher observed that the Marsaglia's subtract-and-borrow is equivalent to a chaotic discrete dynamical system and the above 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.

7.4.8 *Knuth's polynomial congruential generator*

$$x_i = (ax_{i-1}^2 + bx_{i-1} + c) \mathrm{mod} m \tag{7.18}$$

This generator makes the form of a function less simple and therefore the PRNG less obvious and therefore it finds applications in cryptography. One example is the Blum, Blum and Shub generator:

$$x_i = x_{i-1}^2 \mathrm{mod} m \tag{7.19}$$

7.4.9 *Inverse congruential generator*

$$x_i = (a x_{i-1}^{-1} + c) \mathrm{mod} m \tag{7.20}$$

where x_{i-1}^{-1} is the multiplicative inverse of x_{i-1} modulo m, i.e. $x_{i-1} x_{i-1}^{-1} = 1 \mathrm{mod} m$.

7.4.10 *Defects of PRNGs*

The non-randomness of pseudo-random number generators manifest 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 patters 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.

As en example of what we mean by patters we present here two scatter plots. In the plot the coordinates of each point (x, y) are given by subsequent random numbers (z_i, z_{i+1}). In the plot on the left, the z_i are real random

numbers generated using a chaotic solid state device while, in the plot on the right, the numbers are generated with RANDU.

[ADD FIGURE HERE]

7.4.11 *Marsenne Twister*

One of the best algorithms is the Marsenne Twister which produces 53-bits random number and it has a period of $2^{19937} - 1$ (this number is 6002 digits long!). The Python `random` module uses the Marsenne Twister. Although discussing the inner working of this algorithm is beyond the scobe of this notes we here provide a pure python implementation of the Marsenne Twister:

Listing 7.2: in file: `numeric.py`

```
class MarsenneTwister(object):
    """
    based on:
    Knuth 1981, The Art of Computer Programming
    Vol. 2 (2nd Ed.), pp102]
    """
    def __init__(self,seed=4357):
        self.w = []   # the array for the state vector
        self.w.append(seed & 0xffffffffL)
        for i in xrange(1, 625):
            self.w.append((69069 * self.w[i-1]) & 0xffffffffL)
        self.wi = i
    def random(self):
        w = self.w
        wi = self.wi
        N, M, U, L = 624, 397, 0x80000000L, 0x7fffffffL
        K = [0x0L, 0x9908b0dfL]
        y = 0
        if wi >= N:
            for kk in xrange((N-M) + 1):
                y = (w[kk]&U)|(w[kk+1]&L)
                w[kk] = w[kk+M] ^ (y >> 1) ^ K[y & 0x1]

            for kk in xrange(kk, N):
                y = (w[kk]&U)|(w[kk+1]&L)
                w[kk] = w[kk+(M-N)] ^ (y >> 1) ^ K[y & 0x1]
            y = (w[N-1]&U)|(w[0]&L)
            w[N-1] = w[M-1] ^ (y >> 1) ^ K[y & 0x1]
        wi = 0
```

```
        y = w[wi]
        wi += 1
        y ^= (y >> 11)
        y ^= (y << 7) & 0x9d2c5680L
        y ^= (y << 15) & 0xefc60000L
        y ^= (y >> 18)
        return (float(y)/0xffffffffL )
```

7.5 *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/processes and you need to make sure that the streams do not overlap.

A common and wrong choice is that of generating 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. If this happens to be the case the two sequences of random numbers are not independent but just shifted. For example here are two RANDU sequence 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
y_3	0.992022250779	0.949562561698
y_4	0.146293803118	0.380731142126
y_5	...	...

(7.21)

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.

7.5.1 *Non-overlapping blocks*

Let's consider one sequence of pseudo random numbers

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{7.22}$$

One can break it into sub-sequences of k numbers

$$x_0, x_1, ..., x_{k-1} \tag{7.23}$$

$$x_k, x_{k+1}, ..., x_{2k-1} \tag{7.24}$$

$$x_{2k}, x_{2k+1}, ..., x_{3k-1} \tag{7.25}$$

$$... \tag{7.26}$$

If the original sequence is created with a multiplicative congruential generator

$$x_i = ax_{i-1} \text{mod} m \tag{7.27}$$

the sub-sequences can be generated independently because

$$x_{nk-1} = a^{nk-1} x_0 \text{mod} m \tag{7.28}$$

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 sub-sequences.

7.5.2 *Leapfrogging*

Another and probably more popular technique is leapfrogging. Let's consider one sequence of pseudo random numbers

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{7.29}$$

One can break it into sub-sequences of k numbers

$$x_0, x_k, x_{2k}, x_{3k}, ... \tag{7.30}$$

$$x_1, x_{1+k}, x_{1+2k}, x_{1+3k}, ... \tag{7.31}$$

$$x_2, x_{2+k}, x_{2+2k}, x_{2+3k}, ... \tag{7.32}$$

$$... \tag{7.33}$$

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

$$x_{i+k} = a^k x_i \mathrm{mod} m \tag{7.34}$$

therefore leapfrogging is also a viable technique for parallel random number generators.

Here we define a Multiplictive Congruential Generator with a `leapfrog(k)` method which returns k distinct and independent generators of the same time:

```
class MCG(LCG):
    def __init__(self,seed,a=7**5,m=2**31-1):
        LCG.__init__(self,seed,a=a,c=c,m=m)
    def leapfrog(self,k):
        a, m = self.a, self.m
        for i in range(k):
            self.next()
            generators.append(MCG(self.x,a=(a**k % m),m))
        return generators
```

Here is an example of usage:

```
>>> generators=MCG(1).leapfrog(3)
>>> for k in range(3):
...     for i in range(5):
...        x=generators[k].random()
...        print(k,'\t',i,'\t',x)
```

The Marsenne Twister algorithm implemented in `os.random` has leapforgging built-in. In fact the module include a `random.jumpahead(n)` method that allows to efficiently skip n numbers.

7.5.3 *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 respectiveley

$$x_i^L = (a_L x_{i-1} + c_L) \mathrm{mod} m \tag{7.35}$$

$$x_i^R = (a_R x_{i-1} + c_R) \mathrm{mod} m \tag{7.36}$$

then, recursively, append nodes to the children.

7.6 *Generating random number form given distribution*

In this section and the next we will provide examples of distrubutions other then uniform and algorithms to generate numbers using those 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 non-uniform random numbers:

- accept-reject (applies to both disecrete and continuous distributions)
- inversion methods (applies to continuous distributions only)

Consider the problem of generating a 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 coorinates 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 curve is highest and thus the probability density is greatest. The g distibution 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 thorw darts below the p that we want therefore we accept them all. This is the mst efficient case but it is not of pratical interest because it means the accept-reject is not doing anything, as we already know now to generate numbers accoring to p. The other case is $g(x) = constant$. This means we generate the x uniformly before the accept-reject. This is equivalent to thrwing the darts everywhere on the board, without even trying to be below the curve p. This

is now the algorithm is often utilized.

The inverion method instead is more efficient but requires some math. It states that if $F(x)$ is a cumulative distribution funciton 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 distribution where F can be expressed in analytical terms and inversted, the inversion method is the best way to generate random numbers. An example is the exponential distribution.

[SAY MORE]

We will create a new class `RandomSource` which includes methods to generate the random number.

7.6.1 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 discerete case as we implement it into a `randint` method:

Listing 7.3: in file: `numeric.py`

```
class RandomSource(object):
    def __init__(self,generator=None):
        if not generator:
            import random as generator
        self.generator = generator
    def random(self):
        return self.generator.random()
    def randint(self,a,b):
        return int(a+(b-a+1)*self.random())
```

Notice that the random `RadomSource` constructor expects a generator suck as `LCG`, `MCG`, `MarsenneTwister`, 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 7.4: in file: `numeric.py`

```
    def choice(self,S):
        return S[self.randint(0,len(S)-1)]
```

7.6.2 *Bernoulli distribution*

The Bernoulli distribution, named after Swiss scientist James Bernoulli, is a discrete probability distribution, which takes value 1 with success probability p and value 0 with failure probability $q = 1 - p$.

$$p(k) \stackrel{def}{=} \left\{ \begin{array}{ll} p & \text{if } k = 1 \\ 1-p & \text{if } k = 0 \\ 0 & \text{if not } k \in \{0,1\} \end{array} \right\} \tag{7.37}$$

A Bernoulli random variable has expected value of p, and variance of pq.

We implement it by adding a corresponding method to the `RandomSource` class

Listing 7.5: in file: `numeric.py`

```
    def bernoulli(self,p):
        return 1 if self.random()<p else 0
```

7.6.3 *Biased Dice and Table Lookup*

A generalization of the Bernoulli distribution is a distributon 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 7.6: in file: `numeric.py`

```
    def lookup(self,table, epsilon=1e-6):
        if isinstance(table,dict): table = table.items()
        u = self.random()
        for key,p in table:
            if u<p+epsilon:
                return key
```

```
            u = u - p
        raise ArithmeticError, "invalid probability"
```

Let's say we want a random number generator that can only produce the outcome 0,1, or 2 with known probabilities:

$$\text{Prob}(X = 0) = 0.50 \tag{7.38}$$

$$\text{Prob}(X = 1) = 0.23 \tag{7.39}$$

$$\text{Prob}(X = 2) = 0.27 \tag{7.40}$$

since the probability of the possible outcomes are rational numbers (fractions) we can proceed as follows:

```
>>> def test_lookup(nevents=100,table=[(0,0.50),(1,0.23),(2,0.27)]):
...     g = RandomSource()
...     f=[0,0,0]
...     for k in range(nevents):
...         p=g.lookup(table)
...         print(p,)
...         f[p]=f[p]+1
...     print
...     for i in range(len(table)):
...         f[i]=float(f[i])/nevents
...         print('frequency[%i]=%f' % (i,f[i]))
```

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 requencies, 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 will discuss this later in the chapter.

In some sense we can think of the table look-up as an application of the linear search. We start with a segment of length 1 and we break 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 generated a random point on the initial segment and we ask in which of the n intervals it falls. The table look-up 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.

7.6.4 Fishman-Yarberry method

The Fishman Yarberry (F-Y) method is an improvement over the naive table look-up that runs in $O(\lceil \log_2 n \rceil)$. As the naive table look-up 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 look-up 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:

$$a_{ij} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & ... & a_{0,n-1} \\ ... & ... & ... & ... & \\ a_{t-2,0} & a_{t-2,1} & a_{t-2,2} & a_{t-2,3} & \\ a_{t-1,0} & a_{t-1,1} & & & \end{pmatrix} \tag{7.41}$$

In other words we can say that:

- a_{ij} represents the probability

$$\text{Prob}(X = x_j) \tag{7.42}$$

- a_{1j} represents the probability

$$\text{Prob}(X = x_{2j} or X = x_{2j+1}) \tag{7.43}$$

- a_{4j} represents the probability

$$\text{Prob}(X = x_{4j} \text{ or } X = x_{4j+1} \text{ or } X = x_{4j+2} \text{ or } X = x_{4j+3}) \tag{7.44}$$

- a_{ij} represents the probability

$$\text{Prob}(X \in \{x_k | 2^i j \leq k < 2^i (j+1)\}) \tag{7.45}$$

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 the array a once for all. The method discrete_map maps a uniform random number u into the desired discerete integer.

```
def log2(x): return log(x)/log(2)

class FishmanYarberry1993:
    def __init__(self,table=[[0,0.2], [1,0.5], [2,0.3]]):
        t=log2(len(table))
        while t!=int(t):
            table.append([0,0.0])
            t=log2(len(table))
        t=int(t)
        a=[]
        for i in range(t):
            a.append([])
            if i==0:
                for j in range(2**t):
                    a[i].append(table[j,1])
            else:
                for j in range(2**(t-i)):
                    a[i].append(a[i-1][2*j]+a[i-1][2*j+1])
        self.table=table
        self.t=t
        self.a=a
```

```
22
23      def discrete_map(self, u):
24          i=int(self.t)-1
25          j=0
26          b=0
27          while i>0:
28              if u>b+self.a[i][j]:
29                  b=b+self.a[i][j]
30                  j=2*j+2
31              else:
32                  j=2*j
33              i=i-1
34          if u>b+self.a[i][j]:
35              j=j+1
36          return self.table[j][0]
```

7.6.5 *Binomial distribution*

The binomial distribution is a discrete probability distribution which describes the number of successes in a sequence of n independent experiments, 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: 5% of the population are HIV-positive. You pick 500 people randomly. How likely is it that you get 30 or more HIV-positives? The number of HIV-positives you pick is a random variable X which follows a binomial distribution with $n = 500$ and $p = 0.05$. 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

$$p(k) = \text{Prob}(X = k) \stackrel{def}{=} \binom{n}{k} p^k (1-p)^{n-k} \tag{7.46}$$

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

$$p(k) = \text{Prob}(X = k) = \binom{n_X + n_Y}{k} p^k (1-p)^{n-k} \tag{7.47}$$

We can generate random numbers following binomial distribution using a table lookup with table:

$$\text{table}[k] = \text{Prob}(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{7.48}$$

For large n it may be convenient not avoid storing the table and use the formula directly to compute its elements on a need-to-know basis. Moreover since the table is accessed sequentially by the table look-up algorithm one may just notice that the current recursive relation holds:

$$\text{Prob}(X = 0) = (1-p)^n \tag{7.49}$$

$$\text{Prob}(X = k+1) = \frac{n}{k+1}\frac{p}{1-p}\text{Prob}(X = k) \tag{7.50}$$

This allows for a very efficient implementation:

Listing 7.7: in file: `numeric.py`

```
    def binomial(self,n,p,epsilon=1e-6):
        u = self.random()
        q = (1.0-p)**n
        for k in range(n+1):
            if u<q+epsilon:
                return k
            else:
                u = u - q
            q = q*(n-k)/(k+1)*p/(1.0-p)
        raise ArithmeticError, "invalid probability"
```

7.6.6 *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 (i.e., non-random)

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 success probability p, then X follows the negative binomial distribution with parameters r and p.

$$p(n) = \text{Prob}(X = n) = \binom{n-1}{k-1} p^k (1-p)^{n-k} \tag{7.51}$$

The implementation is trivial:

Listing 7.8: in file: `numeric.py`

```
    def negative_binomial(self,n,p,epsilon=1e-6):
        return self.binomial(n-1,p-1,epsilon)
```

Here is a fun example attributed to Dr. Diane Evans, a math professor at Rose-Hulman Institute of Technology:

Johnny, a sixth grader at Honey Creek Middle School in Terre Haute, Indiana, is required to sell candy bars in his neighborhood to raise money for the 6th grade field trip. There are thirty homes in his neighborhood, and his father has told him 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 mass function for selling the last candy bar at the nth house?

$$p(n) = \binom{n-1}{4} 0.4^5 0.6^{n-5} \tag{7.52}$$

- What's the probability that he finishes on the tenth house?

$$p(10) = \binom{9}{4} 0.4^5 0.6^5 = 0.10 \tag{7.53}$$

- 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:

$$\sum_{i=5,6,7,8} p(i) = 0.1737 \tag{7.54}$$

- What's the probability that he exhausts all houses in the neighborhood, gives up, and then goes to live on the streets?

$$1 - \sum_{i=5,..,30} p(i) = 0.0015 \tag{7.55}$$

7.6.7 *Poisson distribution*

The Poisson distribution is a discrete probability distribution (discovered by Siméon-Denis Poisson (1781-1840) and published, together with his probability theory, in 1838 in his work Recherches sur la probabilité des jugements en matières criminelles et matière civile) describing certain random variables X that count, among other things, a 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:

$$p(k) = \text{Prob}(X = k) = e^{-\lambda} \frac{\lambda^k}{k!} \tag{7.56}$$

The Poisson distribution arises in connection with Poisson processes. It applies to various phenomena of discrete nature (that is, 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 becase it is different than zero for any natural number k, rather then for a finite set of k values.

Examples include:

- 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 number of soldiers killed by horse-kicks each year in each corps in the Prussian cavalry (an example made famous by a book of Ladislaus Josephovich Bortkiewicz (1868-1931)).
- The number of bombs falling on each square mile of London during a German air raid in the early part of the Second World War.

The binomial distribution with parameters n and $p = \lambda/n$, i.e., the probability distribution of the number of successes in n trials, with probability λ/n of success on each trial, approaches the Poisson distribution with expected value λ as n approaches infinity

$$\frac{n!}{(n-k)!k!}\left(\frac{\lambda}{n}\right)^k\left(1-\frac{\lambda}{n}\right)^{n-k} \simeq e^{-\lambda}\frac{\lambda^k}{k!} + O(\frac{1}{n}) \tag{7.57}$$

Intuitively the meaning of λ is the following: Let's consider a unitary time interval T and divide it into n sub-intervals of the same size. Let p_n be the probability of one success occurring in a single sub-interval. For T fixed when $n \to \infty$, $p_n \to 0$ but the limit

$$\lim_{n\to\infty} pn \tag{7.58}$$

is finite. This limit is λ.

We can use the same technique adopted for the Binomial distribution and observe that for Poisson

$$\text{Prob}(X = 0) = e^{-\lambda} \tag{7.59}$$

$$\text{Prob}(X = k+1) = \frac{\lambda}{k+1}\text{Prob}(X = k) \tag{7.60}$$

therefore the above algorithm can be modified into:

Listing 7.9: in file: `numeric.py`

```
    def poisson(self,lamb,epsilon=1e-6):
        u = self.random()
        q = exp(-lamb)
        k=0
        while True:
            if u<q+epsilon:
                return k
            else:
                u = u - q
            q = q*lamb/(k+1)
            k = k+1
        raise ArithmeticError, "invalid probability"
```

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 behaviour by chosing ε very small compared with the precision required in the computation but larger than machine precision.

7.7 Probability distributions for continuous random variables

7.7.1 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

$$h_i = a + \lfloor (b - a + 1) y_i \rfloor \tag{7.61}$$

Listing 7.10: in file: `numeric.py`

```
    def uniform(self,a,b):
        return a+(b-a)*self.random()
```

7.7.2 *Exponential distribution*

Probability mass function is given by:

$$p(x) = \lambda e^{-\lambda x} \tag{7.62}$$

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 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:

- 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. This means that if a random variable X is exponentially distributed, its conditional probability obeys

$$\text{Prob}(X > s + t | X > t) = \text{Prob}(X > s) \tag{7.63}$$

In other words, the chance that the state change is going to happen in the next s seconds is unaffected by the amount of time that has already elapsed.

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:

$$F(x) = \int_0^x p(y)dy = 1 - e^{-\lambda x} \tag{7.64}$$

and we then invert $u = F(x)$ thus obtaining:

$$x = -\frac{1}{\lambda}\log(1-u) \tag{7.65}$$

Now notice that if u is uniform, then $1 - u$ is also uniform therefore we can further simplify:

$$x = -\frac{1}{\lambda}\log u \tag{7.66}$$

This we implement as follows:

Listing 7.11: in file: `numeric.py`

```
    def exponential(self,lamb):
        return -log(self.random())/lamb
```

7.7.3 *Normal/Gaussian distribution*

The normal distribution is an extremely important probability distribution considered in statistics. Among people whose field is not primarily probability theory or statistics (notably in physics) it is often called the Gaussian distribution. Here is the probability mass function:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7.67}$$

where $E[X] = \mu$ and $E[(x-\mu)^2] = \sigma^2$

The standard normal distribution is the normal distribution with a mean of zero and a standard deviation of one. Because the graph of its probability density resembles a bell, it is often called the bell curve.

The Gaussian distribution has two important properties:

- Becasue of the central Limit the average of many independent random variables with finite mean and finite variance tends to a Gaussies distributon.
- 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 a also a Gaussian random variable with mean $\mu = \mu_1 + \mu_2$ and variance $\sigma^2 = \sigma_1^2 + \sigma_2^2$.

A consequence of these properties if very important for random walks.

The Gaussian distribution is a more difficult example of those seen above therefore we do not give a proof here. The complication is that there is no way to map one single uniform random variable into a Gaussian distributed random variable but, there are ways to map two uniform independent random variables (x_1 and x_2) into two independent Gaussian distributed variables (y_1 and y_2). This is achieved by:

- 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 7.12: in file: `numeric.py`

```
    def gauss(self,mu=0.0,sigma=1.0):
        if hasattr(self,'other') and self.other:
            this, other = self.other, None
        else:
            while True:
                v1 = self.random(-1,1)
                v2 = self.random(-1,1)
                r = v1*v1+v2*v2
                if r<1: break
            this = sqrt(-2.0*log(r)/r)*v1
            self.other = sqrt(-2.0*log(r)/r)*v1
        return mu+sigma*this
```

Note how the first time the method next is called it generated two Gaussian numbers (*this* and *other*), stores *other* and returns *this*. Every other time the method next is called if *other* is stored it, returns it, otherwise it recomputes *this* and *other* again.

To map a random Gaussian number y with mean 0 and standard devaition 1

into another Gaussian number y' with mean μ and standard deviation σ:

$$y' = \mu + y\sigma \tag{7.68}$$

We used thie relation in the last line of the code.

7.7.4 *Pareto Distribution*

The Pareto distribution, named after the Italian 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:

$$F(x) \stackrel{def}{=} \text{Prob}(X < x) = 1 - \left(\frac{x_m}{x}\right)^{\alpha} \tag{7.69}$$

It can be implemented as follows using the inversion method:

Listing 7.13: in file: `numeric.py`

```
    def pareto(self,alpha,xm):
        u = self.random()
        return xm*(1.0-u)**(-1.0/alpha)
```

Let me define a hypervolume V as a continuum and connected subset of N-dimensional space $\mathbb{R}^N$. With this definition an interval $[a, b]$ is a upervolume of dimension 1; the area of a circle is an hypervolume of dimension 2; the volume of a cylinder is an hypervolume of dimension 3; etc.

7.7.5 *On a circle*

A random point (x, y) uniformly distributed on a circle is obtained by generating a uniform random number u and

$$x = \cos(2\pi u) \tag{7.70}$$
$$y = \sin(2\pi u) \tag{7.71}$$

Listing 7.14: in file: numeric.py

```
    def point_on_circle(self, radius=1.0):
        angle = 2.0*pi*self.random()
        return radius*math.cos(angle), radius*math.sin(angle)
```

Listing 7.15: in file: numeric.py

```
    def point_in_circle(self,radius=1.0):
        while True:
            x = self.uniform(-radius,radius)
            y = self.uniform(-radius,radius)
            if x*x+y*y < radius*radius:
                return x,y
```

Listing 7.16: in file: numeric.py

```
    def point_in_sphere(self,radius=1.0):
        while True:
            x = self.uniform(-radius,radius)
            y = self.uniform(-radius,radius)
            z = self.uniform(-radius,radius)
            if x*x+y*y*z*z < radius*radius:
                return x,y,z
```

7.7.6 *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 $v_1^2 + v_2^2 + v_3^2 \leq 1$

$$x = v_1 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{7.72}$$

$$y = v_2 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{7.73}$$

$$z = v_3 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{7.74}$$

else start again.

Listing 7.17: in file: numeric.py

```
    def point_on_sphere(self, radius=1.0):
        x,y,z = self.point_in_sphere(radius)
        norm = math.sqrt(x*x+y*y+z*z)
        return x/norm,y/norm,z/norm
```

7.8 Resampling

7.8.1 Binning

Binning is the process of dividing a space of possible events into partitions and count how many events fall into each partition. We can bin the numbers generated by a pseudo-random generator and measure the distribution of the random numbers.

Let's consider the following program:

```
def bin(generator,nevents,a,b,nbins):
    # create empty bins
    bins=[]
    for k in range(nbins):
        bins.append(0)
    # fill the bins
    for i in range(nevents):
        x=generator.uniform()
        if x>=a and x<=b:
            k=int((x-a)/(b-a)*nbins)
            bins[k]=bins[k]+1
    # normalize bins
    for i in range(nbins):
        bins[i]=float(bins[i])/nevents
    return bins

def test_bin(nevents=1000,nbins=10):
    bins=bin(MCG(time()),nevents,0,1,nbins)
    for i in range(len(bins)):
        print(i, bins[i])

>>> test_bin()
```

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;
- size of the bins is normalized, the sum of the values are 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 frequences differ from 0.1 for less than 3% while some of the preceding numbers differ from 0.11 for more than 20%.

We can think of the generation of a single random number as the simulation of the simplest possible event. The more events we simulate the more we learn about the system, in this case the random generator itself and the distribution/frequencies of numbers that come out.

7.8.2 Chi-square

The most commonly used measure of difference between observed relative frequencies f_i and hypothesized probabilities π_i is the chi-squared. For n

bins the chi-square is defined as

$$\chi^2 = \sum_{i=0}^{i<n} \frac{(f_i - \pi_i)^2}{\pi_i} \tag{7.75}$$

where n_{events} is the total number of events. The lower the chi-square, the better the experimental frequencies approximate the hypothesized probabilities. Here is a sample code:

```
def chi_square(nevents, frequencies, pis):
    sum=0
    for i in range(len(frequencies)):
        sum=sum+((frequencies[i]-pis[i])**2)/pis[i]
    return nevents*sum
```

We now compute the χ^2 for the two sets of frequencies shown above which correspond to $n_{events} = 1000$ and $n_{events} = 100000$ respectively. In both cases the expected distribution is uniform therefore $\pi_i = 0.1$. We obtain:

$$\chi^2_{(1000)} = 11.24 \tag{7.76}$$

$$\chi^2_{(100000)} = 0.126348 \tag{7.77}$$

If we repeat the experiment more and more times, and the random generator is not biased, the chi-squared tend to zero. Our experiment has two parameters: the number of simulated events, n_{events}, and the number of bins, n. For a real uniform random number generator

$$\forall n \forall \varepsilon \exists \overline{n} \mid \forall n_{events} > \overline{n}, \chi^2(n_{events}, n) < \varepsilon \tag{7.78}$$

i.e. for any given number of beans, n, we can reduce the chi-square arbitrarily (less then any given ε) by increasing the number of simulated events ($n_{events} > \overline{n}$ and $\exists \overline{n}$). This statement is not true if we use a pseudo-random number generator. In fact, since a PRNG is periodic there is only a finite number of possible different simulated events.

Other tests can de built to test the goodness of a PRNG. Two standard test suits are Marsaglia's DIEHARD (1985) and the NIST (2000).

7.9 *Randomness and distributions*

The technique of binning allows us to "measure" the distribution of numbers generated by a PRNG. The chi-squared test allows us to compare a measure distribution (relative frequency) with the expected distribution. We now want to see how to use a uniform PRNG to implement generators that realize different distributions.

8

Monte Carlo Simulations

Monte Carlo is the use of random numbers in computation. Let's consider two examples:

8.1 Introduction

8.1.1 Computing π

The standard way to compute π is by applying the definition: π is the length of a semicircle of radius equal to 1. From the definition one can derive an exact formula

$$\pi = 4 \arctan 1 \tag{8.1}$$

The arctan has the following Taylor series expansion[1]:

$$\arctan x = \sum_{i=0} (-1)^i \frac{x^{2i+1}}{2i+1} \tag{8.8}$$

[1] Taylor expansion:

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + ... + \frac{1}{i!}f^{(i)}(0)x^2 + ... \tag{8.2}$$

and one can approximate π to arbitrary precision by computing the sum

$$\pi = \sum_{i=0}(-1)^i \frac{4}{2i+1} \tag{8.9}$$

We can use the following program:

```
def pi_Taylor(n):
    pi=0
    for i in range(n):
        pi=pi+4.0/(2*i+1)*(-1)**i
        print(i,pi)

>>> pi_Taylor(1000)
```

which produces the following output:

```
0 4.0
1 2.66666...
2 3.46666...
3 2.89523...
4 3.33968...
...
999 3.14..
```

The above formula works but converges very slowly to the exact value of

$$\pi = 3.14159265358979323846264338... \tag{8.10}$$

There is a different approach based on the fact that π is also the area of a circle of radius one. We can draw a square or area one containing a quarter of a circle of radius one. We can randomly generate points (x, y) with uniform

and if $f(x) = \arctan x$ then:

$$f'(x) = \frac{d \arctan x}{dx} = \frac{1}{1+x^2} \to f'(0) = 1 \tag{8.3}$$

$$f''(x) = \frac{d^2 \arctan x}{d^2 x} = \frac{d}{dx}\frac{1}{1+x^2} = -\frac{2x}{(1+x^2)^2} \tag{8.4}$$

$$... \tag{8.5}$$

$$f^{(2i+1)}(x) = (-1)^i \frac{(2i)!}{(1+x^2)^{2i+1}} + x... \to f^{(2i+1)}(0) = (-1)(2i)! \tag{8.6}$$

$$f^{(2i)}(x) = x... \to f^{(2i)}(0) = 0 \tag{8.7}$$

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:

```
from random import *

def pi_mc(n):
    pi=0
    counter=0
    for i in range(n):
        x=random()
        y=random()
        if x**2 + y**2 < 1:
            counter=counter+1
        pi=4.0*float(counter)/(i+1)
        print(i,pi)

>>> pi_mc(1000)
```

which produces the following output:

```
0 4.0
1 2.0
2 2.666666...
3 2.0
4 1.6
5 2.0
6 2.285714...
7 2.5
8 2.666666...
9 2.4
...
998 3.1410...
999 3.1437...
```

There are problems when using this technique, it is the only viable way since an exact formula is not known or it is too complex. Note that the function `random()` produces a random floating point number with uniform distribution in the interval (0,1).

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

$$\pi = 4 \int_0^1 \sqrt{1 - x^2} dx \tag{8.11}$$

The problem of computing numerical integrals can often be solved using Monte Carlo.

8.1.2 Simulating an on-line merchant

Let's consider an on-line merchant. A web site is visited many times a day. We assume that the average number of visitors in a day is 976, the number of visitors is Gaussian distributed and the standard deviation is 352. 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. This information is available from analysis of past data.

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 in analytical terms or reduced to the computation of an integral but it can easily be simulated.

In particular we simulate `days` days, for each day i we put N items in stock and we loop over a gaussian number of simulated visitors. For each visitor j, if he/she finds an item in stock they buy it with a 5% probability producing an income of $70, if the item is not in stock he/she buys it with 2% probability producing an income of $100. At the end of the each day we pay $30 for each item remained 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:

```
1 def simulate_once(N):
2     profit = 0
3     loss = 30*N
```

```
    instock = N
    for j in range(int(gauss(976,352))):
        if instock>0:
            if random()<0.05:
                instock=instock-1
                profit = profit + 100
        else:
            if random()<0.02:
                profit = profit + 100
    return profit-loss

def simulate_many(N,ap=1,rp=0.01,ns=1000):
    s = 0.0
    for k in range(1,ns):
        x = simulate_once(N)
        s += x
        mu = s/k
        if k>10 and mu-mu_old<max(ap,rp*mu):
            return mu
        else:
            mu_old = mu
    raise ArithmeticError, "no convergence"
```

by looping over different N (items in stock) we can compute the average daily income as function of N:

```
>>> for N in range(0,100,10):
>>>     print(N,simulate_many(N,ap=100))
```

which produces the following table

```
n income
0 1955
10 2220
20 2529
30 2736
40 2838
50 2975
60 2944
70 2711
80 2327
90 2178
```

From which 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 itmes 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; while the statement

```
if random()<0.05:
```

ensures that the subsequent block is executed with a probability of 5%.

8.2 General Purpose Monte Carlo Engine

Any Monte Carlo solver comprises of the following parts:

- A generator of random numbers (such as as have discussed in the previous chapter)
- A function that uses the random number generator and can simulate the system once
- A function that calls the above simulation repeatedly and averages the results until they converge
- A function to estimate the accuracy of the result and determine when to stop the simulations.

The code below implements those methods and some more:

Listing 8.1: in file: `numeric.py`

```
def resample(x,size=None):
    return [x[random.randint(0,len(x)-1)] for i in range(size or len(x))]

def bootstrap(x, confidence=0.68, nsamples=100):
    """Computes the bootstrap errors of the input list."""
    def mean(S): return float(sum(x for x in S))/len(S)
    means=[mean(resample(x)) for k in range(nsamples)]
    means.sort()
    left_tail = int(((1.0-confidence)/2)*nsamples)
    right_tail = nsamples-1-left_tail
    return means[left_tail], mean(x), means[right_tail]

def confidence_intervals(mu,sigma):
    """Computes the normal confidence intervals"""
    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)
        ]
    return [(a,mu-b*sigma,mu+b*sigma) for (a,b) in CONFIDENCE]

class MCEngine:
    """
    Monte Carlo Engine parent class.
    Runs a simulation many times and computes average and error in average.
    Must be extended to provide the simulate_once method
    """
    def simulate_once(self):
        raise NotImplementedError

    def simulate_many(self, ap=0.1, rp=0.1, ns=1000):
        self.results = []
        s1=s2=0.0
        self.convergence=False
        for k in range(1,ns):
            x = self.simulate_once()
            self.results.append(x)
            s1 += x
            s2 += x*x
            mu = float(s1)/k
            variance = float(s2)/k-mu*mu
            dmu = sqrt(variance)/k
            if k>10:
                if abs(dmu)<max(ap,ams(mu)*rp):
                    self.converence = True
                    break
        self.confidence_intervals = confidence_intervals(mu,dmu)
        return bootstrap(self.results)

    def var(self, confidence=0.68):
        self.results.sort()
        left_tail = (1.0-confidence)/2
        right_tail = 1.0-left_tail
        min_index = int(left_tail*len(self.results))
        max_index = int(right_tail*len(self.results))
        if min_index<10:
            raise ArithmeticError, 'not enough data, not reliable'
```

```
66          return self.results[min_index], self.results[max_index]
```

8.3 Example: Network Reliability

Let's consider a network represented by a set of n_{nodes} nodes and n_{links} bidirectional links. TCP 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 is trasmitting 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 implement a Network emulator that, given a description of the network (specified by the number of nodes and by a set of links), the probability of success for each link, computes 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 do not require to determine the exact path but only if a path exists the proper data structure is one that implements equivalence classes.

We can say that two nodes follow in the same equivalence class if and only if there is a successful path that connects the two nodes.

The optimal data structure is to implemenet equivalent classes called disjoint sets (DisjSets in short). The DisjSet class has a single member varable, an array (`sets`) with as many elements as the nodes in the network. Each array cell corresponds to a node of the network.

We adopt the convention that if `sets[`i`]` is negative then node i is the representative element of the set of nodes connected to node number i, and $-$`sets[`i`]` is the number of nodes in the set of nodes connected to node number i. By default we assume that all nodes are disconnected therefore for each i, `sets[`i`]`$= -1$. We also adopt the convention that if `sets[`i`]`≥ 0 then node i is not a representative element and it is connected by a path to node $i' =$`sets[`i`]`. We can imagine a tree like structure for each set of

connected nodes. Each node i has a parent (node $i' =$`sets[`i`]`≥ 0) or is the root node and by definition the represenative element of the set (`sets[`i`]`< 0).

Our class DisjSets needs a method, `rep(`i`)`, that given a node i finds out and returns the representative element of the set of nodes connected to i. This is achieved by ascending the tree (while `sets[`i`]`≥ 0 then $i =$`sets[`i`]` else return i).

If we `connect` a link between node i and j there are two possibilitites:

- The representative node of the set containing i and the represenattive node of the set containing j are the same then i and j are already connected and the new link only adds redundancy to the system then we can ignore it.
- The representative node (i') of the set containing i and the represenattive node (j') of the set containing j are different therefore the existence of the new link tells us that we should merge the two sets. This is achieved by saying that the set i' now contains also the nodes in j' (`sets[`i'`]=sets[`i'`]+sets[`j'`]`) and that the representative element of the set containing j and j' is no longer j' but it is i' (`sets[`j'`]=`i').

Class DisjSet also needs a method `is_path(`i,j`)` that checks if i and j are connected by looking if they have the same `rep(`i`)==rep(`j`)`.

Listing 8.2: in file: `numeric.py`

```
class NetworkReliability(MCEngine):
    def __init__(self,n_nodes,start,stop):
        self.links = []
        self.b_nodes = n_nodes
        self.start = start
        self.stop = stop
    def add_link(self,i,j,failure_probability):
        self.links.append((i,j,failure_probability))
    def simulate_once(self):
        nodes = DisjoinSets(self.n_nodes)
        for i,j,pf in self.links:
            if random.random()>pf:
                nodes.join(i,j)
        return nodes.joined(i,j)
```

In order to simulate the system we need a class Network with two methods

- `SimulateOnce` that tries to send a packet from `start` to `stop` and simulates

the network once. The simulation may succeed (packet arrives to destination) or fail (packet is dropped). Each simulation consists of creating a DisjSets representing all disconnected nodes in the network; looping over all links and accepting each link with a probability equal to the probability of success for that link (p); if a link is accepted than there is a successful connection between the sets containing the endpoints of the link. If after looping over all links there is a path consisting of accepted (successful) links between `start` and `stop`, `SimulateOnce` return 1, otherwise 0.

- `SimulateMany` that simulates many trasmissions (N) of the packet and counts the rate of success. For large numbers of simulations ($N \to \infty$) the rate of success gives the probability of success of sending a packet from `start` to `stop`.

8.4 Example: Nuclear Reactor Simulation

Listing 8.3: in file: `numeric.py`

```
class NuclearReactor(MCEngine):
    def __init__(self,radius,density):
        self.random = RandomSource()
        self.radius = radius
        self.density = density
    def simulate_once(self):
        p = self.random.point_in_shpere(self.radius)
        events = [p]
        while events:
            event = events.pop()
            v = self.random.point_on_sphere()
            d1 = self.random.exponential(self.density)
            d2 = self.random.exponential(self.density)
            p1 = (p[0]+v[0]*d1,p[1]+v[1]*d1,p[2]+v[2]*d1)
            p2 = (p[0]-v[0]*d2,p[1]-v[1]*d2,p[2]-v[2]*d2)
            if p1[0]**2+p1[1]**2+p1[2]**2<self.radius:
                events.append(p1)
            if p2[0]**2+p2[1]**2+p2[2]**2<self.radius:
                events.append(p2)
            if len(events)>1000:
                return 1.0
        return 0.0
```

8.5 Monte Carlo Integration

8.5.1 1d Monte Carlo integration

Let's consider a one dimensional integral

$$I = \int_a^b f(x)dx \tag{8.12}$$

Let's now determine two functions $g(x)$ and $p(x)$ such that

$$p(x) = 0 \text{ for } x \in [-\infty, a] \cup [n, \infty] \tag{8.13}$$

and

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \tag{8.14}$$

and

$$g(x) = f(x)/p(x) \tag{8.15}$$

We can interpret $p(x)$ as a probability mass function and

$$E[g(X)] = \int_{-\infty}^{+\infty} g(x)p(x)dx = \int_a^b f(x)dx = I \tag{8.16}$$

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, altough not in general an optimal choice, is

$$\begin{aligned} p(x) &\stackrel{def}{=} \left\{ \begin{array}{ll} 1/(b-a) & \text{if } x \in [a,b] \\ 0 & \text{otherwise} \end{array} \right\} \\ g(x) &\stackrel{def}{=} (b-a)f(x) \end{aligned} \tag{8.17}$$

so that X is just a uniform random variable in $[a, b]$ and using eq.(**??**):

$$I = E\left[f(X)\right] = (b-a)\frac{1}{N}\sum_{i=0}^{i<N} f(x_i) \tag{8.18}$$

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

$$\sigma^2 = (b-a)^2 \frac{1}{N} \sum_{i=0}^{i<N} [f(x_i) - \langle f \rangle]^2 \tag{8.19}$$

with

$$\langle f \rangle = \frac{1}{N} \sum_{i=0}^{i<N} f(x_i) \tag{8.20}$$

and the error on the result is

$$\delta I = \sqrt{\frac{\sigma^2}{N-1}} \tag{8.21}$$

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 program in Python:

```
def mc_integrate_1d(f,a,b,n,generator):
    sum1=0.0
    sum2=0.0
    for i in range(n):
        x=a+(b-a)*generator.uniform()
        y=f(x)
        sum1=sum1+y
        sum2=sum2+y*y
    I=(b-a)*sum1/n
    variance=((b-a)**2)*sum2/n-I*I
    dI=sqrt(variance/(n-1))
    return I,dI

def test_f(x):
    return sin(x)

>>> mc_integrate_1d(test_f,0,1,10000,MCG(time()))
```

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 (8.17) 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 one 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.

8.5.2 2-d Monte Carlo integration

The technique described above can easily be extended to 2d integrals

$$I = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 \tag{8.22}$$

where $\mathfrak{D}$ is some 2-dimensional domain. We determine two functions $g(x_0, x_1)$ and $p_0(x_0), p_1(x_1)$ such that

$$p_0(x_0) = 0 \text{ or } p_1(x_1) = 0 \text{ for } x \notin \mathfrak{D} \tag{8.23}$$

and

$$\int p_0(x_0)p_1(x_1)dx_0dx_1 = 1 \tag{8.24}$$

and

$$g(x_0, x_1) = \frac{f(x_0, x_1)}{p_0(x_0)p_1(x_1)} \tag{8.25}$$

We can interpret $p(x_0, x_1)$ as a probability mass function for two independent random variables X_0 and X_1 and

$$E[g(X_0, X_1)] = \int g(x_0, x_1)p_0(x_0)p_1(x_1)dx = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 = I \tag{8.26}$$

Therefore

$$I = E[g(X_0, X_1)] = Area(\mathfrak{D})\frac{1}{N}\sum_{i=0}^{i<N} f(x_{i0}, x_{i1}) \tag{8.27}$$

8.5.3 nD Monte Carlo integration

The technique described above can also be extended to n-d integrals

$$I = \int_{\mathfrak{D}} f(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} \tag{8.28}$$

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

$$p(x_0, ..., x_{n-1}) = 0 \text{ for } x \notin \mathfrak{D} \tag{8.29}$$

and

$$\int p(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} = 1 \tag{8.30}$$

and

$$g(x_0, ..., x_{n-1}) = f(x_0, ..., x_{n-1}) / p(x_0, ..., x_{n-1}) \tag{8.31}$$

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

$$\begin{aligned} E[g(X_0, ..., X_{n-1})] &= \int g(x_0, ..., x_{n-1}) p(x_0, ..., x_{n-1}) dx & (8.32) \\ &= \int_{\mathfrak{D}} f(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} = I & (8.33) \end{aligned}$$

Therefore

$$I = E[g(X_0, .., X_{n-1})] = Volume(\mathfrak{D}) \frac{1}{N} \sum_{i=0}^{i<N} f(\mathbf{x}_i) \tag{8.34}$$

where for every point $\mathbf{x}_i$ is a tuple $(x_{i0}, x_{i1}, ..., x_{i,n-1})$.

Here is the Python code:

```
def mc_integrate_nd(f,box,domain,volume,n,generator):
    sum1=0.0
    sum2=0.0
    for i in range(n):
        x=random_point_in_domain(box,domain,generator)
        y=f(x)
        sum1=sum1+y
        sum2=sum2+y*y
    I=volume*sum1/n
```

```
    variance=(volume**2)*sum2/n-I*I
    dI=sqrt(variance/(n-1))
    return I,dI

def test_f_4d(x):
    return sin(x[0]+x[1]+x[2]+x[3])

def test_domain_4d(x):
    if x[0]**2+x[1]**2+x[2]**4+x[3]**2<1: return 1
    return 0

def identity(x):
    return 1

def test_mc_integrate_nd():
    g=MCG(time()
    N=10000
    box=[[0,1],[0,1],[0,1],[0,1]]
    volume=mc_integrate_nd(identity,box,test_domain_4d,1,N,g))
    print('volume=',volume)
    I=mc_integrate_nd(test_f_4d,box,test_domain_4d,volume[0],N,g)
    print('integral=',I)

>>> test_mc_integrate_nd()
```

8.6 Stochastic, Markov, Wiener and Ito Processes

A *Stochastic process* is a random function, i.e. a function that maps a variable i with domain D into X_i where X_i 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*, stock market and exchange rate fluctuations, signals such as speech, audio and video; medical data such as a patient's EKG, EEG, blood pressure or temperature. Examples of random fields include static images, random topographies (landscapes), or composition variations of an inhomogeneous material.

Let's consider a drunk man moving on a straight line and let S_n be the position of the man at time $t = n\Delta_t$. Let's also assume that at time o $S_0 = 0$. The position of the man at each future ($t > 0$) time is unknown. Therefore it

is a random variable.

We can model the movements of the man as follow:

$$S_{n+1} = S_n + \varepsilon_n \Delta_x \tag{8.35}$$

where Δ_x is a fixed step and ε_n is a random variable whose distribution depends on the model. It is clear that S_{n+1} only depends on S_n and ε_n therefore the probability distribution of S_{n+1} only depends on S_n and the probability distribution of ε_n but it does not depend on the past history of the man's movements at times $t < n\Delta_t$. We can write the statement by saying that

$$\text{Prob}(S_{n+1} = x | \{S_i\} \text{ for } i \leq n) = \text{Prob}(S_{n+1} = x | S_n) \tag{8.36}$$

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*.

To complete our model we need to make additional assumptions about the probability distribution of ε_n.

A continuous time stochastic process (when ε_n is a continuous random number) is called a *Wiener process*.

The specific case when ε_n is a Gaussian random variable, is called an *Ito Process*. And Ito process is also a Weiver process.

- ε_n is a random variable with a Bernoulli distribution ($\varepsilon_n = +1$ with probability p and $\varepsilon_n = -1$ with probability $1 - p$). This assumption makes the Markov process a *Wiener process*.
- ε_n is a random variable with a normal (Gaussian) distribution (with probability mass function $p(\varepsilon) = e^{-\varepsilon^2/2}$). This assumption makes the Markov process an *Ito process*[2].

[2]Sometimes a Ito process is also called a Wiener process because for a Wiener process with $p = 0.5$, according to the definition above, $S_{n+k} - S_n$ approaches a Gaussian distribution in n for large k.

8.6.1 Discrete Random Walk

Here we assume discerete random walk: ε_n equal to $+1$ with probability p and equals 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 man moves forward of one unit (Δ_x) with probability p and if $\varepsilon_n = -1$ he moves backward of 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

$$\frac{n!}{n_+!(n-n_+)!} p^{n_+}(1-p)^{n-n_+} \tag{8.37}$$

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

$$n = n_+ + n_i \tag{8.38}$$
$$k = n_+ - n_- \tag{8.39}$$

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

$$\text{Prob}(n,k) = \frac{n!}{((n+k)/2)!((n-k)/2)!} p^{(n+k)/2}(1-p)^{(n-k)/2} \tag{8.40}$$

Note that $n+k$ has to be even, otherwise it is not possible for the drunk man to reach $k\Delta_x$ in exactly n steps.

8.6.2 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 mass function $p(\varepsilon) = e^{-\varepsilon^2/2}$. It turns out that eq.(8.35) gives

$$S_n = \Delta_x \sum_{i=0}^{i<n} \varepsilon_i \tag{8.41}$$

Therefore the location of the random walker at time $t = n\Delta_t$ is given by the sum of n normal (Gaussian) random variables. It turns out (but we are not prooving it) that

$$p(S_n) = \frac{1}{\sqrt{2\pi n\Delta_x^2}} e^{-x^2/(2n\Delta_x^2)} \tag{8.42}$$

where the variance is $k\Delta_x^2$. Therefore the probability that the random walk is in $[a, b]$ in n steps is given by

$$\begin{aligned} \text{Prob}(a \leq S_n \leq b) &= \frac{1}{\sqrt{2\pi n\Delta_x^2}} \int_a^b e^{-x^2/(2n\Delta_x^2)} dx && (8.43)\\ &= \frac{1}{\sqrt{2\pi}} \int_{a/(\sqrt{n}\Delta_x)}^{b/(\sqrt{n}\Delta_x)} e^{-x^2/2} dx && (8.44)\\ &= \text{erf}(\frac{b}{\sqrt{n}\Delta_x}) - \text{erf}(\frac{a}{\sqrt{n}\Delta_x}) && (8.45) \end{aligned}$$

8.7 Financial Applications

8.7.1 Simple Transaction

The simplest financial operation is a *transaction*. We receive a payment of a fixed amount A at fixed time τ from `abc.com`. From now on we will assume that payment amounts are expressed in dollars (positive if an income, negative otherwise) and time is measured in years or fractions of a year (days/365). The entity we perform the transaction with is only relevant for accounting purposes, not for financial purposes therefore it is irrelevant.

```
class Transaction:
    def __init__(self,args):
        self.time=args[0]
        self.amount=args[1]
```

8.7.2 Net Present Value

Given two possible financial operations such as

- We receive $1000 next year (`Transaction(1,1000)`) or
- We receive $2000 in 2 years (`Transaction(2,2000)`)

(an egg today or a chicken tomorrow). Which is better? This is simple in theory and difficult in practice. We can go to a bank and ask, given the bank's current fixed interest rate, how much can we borrow today so that our balance can be paid in full in one year with $1000? and, how much can we borrow today so that our balance can be paid in full in two years with $2000? The answer to each question is the *Net Present Value* (NPV) of the Transaction.. There are different ways to compute the Net Present Value. Different formulas depend on how the Bank charges interests on our account (*daily compounding, monthly compounding, yearly compounding, etc.*). For the purposes of these notes different formulas are equivalent up to a redefinition of the Bank *interest rate* therefore, from now on, we will adopt the formula for "*continuous compounding*" and will assume the Bank's yearly interest rate r is defined from this formula.

$$\text{NPV} = Ae^{-rt} \tag{8.46}$$

or in Python:

```
def PresentValueOfTransaction(transaction,r):
    return transaction.amount*exp(-r*transaction.time)
```

Firms that operate in the financial sector do not go to the local Bank to get a loan but borrow from the State (Treasury bills), from other companies (in the form of Bonds) or from other Financial Institutions (at the Londor InterBank Offer Rate or LIBOR). The interest rate depends on the available source of borrowing. From now we will assume $r = 0.05$ (5%).

8.7.3 Other deterministic financial operations

Any other deterministic financial operation can be described as a set (or a list) of simple transactions. Let's consider the following financial operation

time	0	2	3	4
amount	$100	-$50	-$40	-$30

(8.47)

that is we get $100 now and we give back $50 after 2 years, $40 after 3 years and $30 after 4 years. What is its NPV? It's present value is given by the sum of the present values of each individual transaction

$$\begin{aligned} \text{NPV} &= \sum_i A_i e^{-rt_i} && (8.48)\\ &= 100 - 50e^{0.05\cdot 2} - 40e^{0.05\cdot 3} - 30e^{0.05\cdot 4} && (8.49)\\ &= -4.23211255114 && (8.50) \end{aligned}$$

The fact that the present value is negative means that we are better off borrowing the $100 from a bank at the rate r (5% in the example). This can be implemented as follows:

```
def NetPresentValue(list_transactions,r):
    sum=0
    for item in list_transactions:
        sum=sum+PresentValueOfTransaction(Transaction(item),r)
    return sum

print(NetPresentValue([(0,100),(2,-50),(3,-40),(4,-30)],0.05))
```

That prints:

```
-4.23211255114
```

We are now able to compare any two deterministic financial operations.

8.7.4 Non-deterministic financial operations

Some financial operations are non deterministic because we are not 100% sure of the date or of the amount of a transaction. For example a car manufacturer knows that in one year from now it has to buy 10,000 tons of steel but it does not know how much that steel is going to cost. Steel is publicly traded therefore its price changes with time. The car manufacturer has some choices:

- Enter in a forward contract. i.e. sign an agreement to buy the steel from a seller at a fixed (agreed upon) future time for a fixed (agreed upon) price. This comes at a fixed cost (agreed upon) by the two parties.

- Enter in a future contract. A future contract is the same as a forward contract but the contract itself is publicly traded. The cost of the contract varies with time and the two parties (buyer and seller) do not know each other. The Future Exchange provides mechanisms to guarantee that contracts are honored.
- Buy an insurance on the cost of steel. If the cost of steel exceeds a certain amount the insurance will pay the difference.
- Buy an option. This is like an insurance but it is publicly traded therefore the cost of the insurance changes with time and, as with the futures, buyer and seller do not know each other.

8.7.5 Futures

Futures are called derivatives because they depend (derive) on an underlying asset X (for example the price of steel).

Since futures themselves can be sold they are more flexible than forward contracts and one would expect that futures cost more than equivalent forward contracts. If interest rate are non-stochastic, that is if they vary at a deterministic known rate in the future, future and forward prices are the same (for similar contracts) and this is true in practice for short term future and forward contracts.

In our example we considered steel but one can buy and sell futures on almost any *asset* that is publicly traded (including the value of an index such as the NASDAQ). If one buys a future on an asset X, one agrees to buy X at the *expiration date* of the future at a fixed price (called *delivery price*) or to sell the futures contract on the market before the expiration date. If one sells a future on X one agrees to sell (and therefore to own) X at the expiration date of the futures contract at a fixed price (delivery price) or to buy back the future before the expiration date.

Let's consider a future on an underlying asset X (where X could be for example a Stock). The current price of en elementary unit of X is called spot price and we will indicate it with S. We will also use A to indicate the

delivery price and τ to indicate the expiration date. Even if S changes with time, at any given time S, A and τ are known therefore the future contract is a financial operation and we can compute its net present value

$$\text{NPV} = S - Ae^{-rt} \tag{8.51}$$

In pratice A is made time dependent and adjusted so that the NPV is 0.

The Present Value of the future is the cost of buying the future. Pricing a future is relatively easy. Here is the Python code:

```
def FutureCost(asset_price,delivery_price,expiration_time,rate):
    return asset_price-delivery_price*exp(-rate*expiration_time)
```

Using a future (or a forward contract) as insurance is common but not always optimal. In the case of the company that has to buy steel, the company may not know the exact date it is going to need the steel. In a forward or future contract the expiration date is fixed. In the case of a futures contract one needs the asset before the expiration date one can always sell the futures contract at that time, but the value of the futures contract will be unknown and this introduces an unwanted risk. Moreover the underlying asset of the future may not be exactly the same type of asset the company need.

Options provide a more sophisticated and flexible way to buy insurance.

The entity that buys a future or other derivative is said to have a *long position*, the entity that sells a future or other derivative is said to have a *short position*.

8.7.6 Options

The simplest option is a European[3] call option. It is a contract that gives the buyer of the contract the right to buy an underlying asset X at a fixed price (called the *strike price*, A) at a fixed date (*expiration date, T*). We will use the notation τ to indicate $T - t_{today}$ that is the time to expiration. From now on we will speak almost exclusively of European options therefore we will omit the "European" identifier. A call option, similarly to a future contract,

[3]European does not mean that they are traded only in Europe. European and American options identify types of options not physical locations. The names have historical reasons.

is publicly traded, its price varies with time and depends on the price of the underlying asset. A call option differs from a future contract because it does not say the buyer of the option has to buy the underlying asset, it only says the buyer of the option has the option (the right) to buy the underlying asset at its strike price at the expiration date of the option. This means that if the price of the underlying asset goes up the buyer of the option may want to *exercise* the option (use the insurance) but if the price goes down he/she can benefit from a lower price. Because of this added flexibility options are more expensive than futures. It is intuitive that the price of an option depend on two parameters: the expected future behavior of the cost of the underlying asset (up or down) and the amount of fluctuations of the price of the underlying asset.

The most common publicly traded options are the following:

European call	the right to buy X at fixed price A at fixed date τ
European put	the right to sell X at fixed price A at fixed date τ
American call	the right to buy X at fixed price A before or at fixed date τ
American put	the right to sell X at fixed price A before or at fixed date τ

(8.52)

Each option is a contract that can be bought or sold. I can sell an European call option; I can buy an American put option; etc. There are 8 combinations. An American option has to cost more than the corresponding European option since it offers more flexibility.

An option that differs from the 8 cases described above is called *exotic*. An exotic option is nothing else than a contract that does not match any of the 8 prototypes known as standard options.

One can sell a call option on X not having X (this is called naked call) as long as one buys X by the expiration date so that he can sell it at the agreed upon price. One can promise to sell something that does not own but one cannot sell something that does not own.

The entity that buys an option is said to have a *long position*, the entity that sells the option is said to have a *short position*.

8.7.7 *European options*

European options are easy to price and we consider them first.

Let's consider a call option on an underlying asset X. Let's consider the moment (day) of the expiration of the option. Let's assume that, at expiration, cost of the underlying asset (the spot price) is S_τ. If S_τ is greater then the strike price of the option A, it is convenient to exercise the option. Whether or not we actually buy the underlying asset is irrelevant. What matters is the fact that we have the right to buy something at A when the market price is S_τ. The value of our contract is therefore $S_\tau - A$. If instead the cost of the underlying asset is lower then the strike price of the option $S_\tau < A$ then there is no reason to exercise the option and the option has no value. Therefore we say that the value at expiration of a European call is

$$\max(S_\tau - A, 0) \tag{8.53}$$

A buyer of the option makes a profit of $\max(S_\tau - A, 0)$ while the seller of the option makes a loss of $-\max(S_\tau - A, 0) = \min(A - S_\tau, 0)$. Let's now go back into the past to the moment when we buy a call option at a price C_{call}. The present value of the financial operation we are entering in is given by

$$\text{PresentValue}_{long-call} = -C_{call} + \max(S_\tau - A, 0)e^{-r\tau} \tag{8.54}$$

while from the point of view of the seller of the same option

$$\text{PresentValue}_{short-call} = +C_{call} - \max(S_\tau - A, 0)e^{-r\tau} \tag{8.55}$$

Where C, A and τ are known at the moment we buy or sell the option while S_τ, the price of the underlying asset at the expiration date, is unknown.

The present value of buying a call option that cost \$3, has a strike price of \$50 and expires in 1 year and is a function of $x = S_\tau$

Conversely the present value of a selling the same call option is

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:

$$C_{call} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(x - A, 0) p_\tau(x) dx \tag{8.56}$$

Similarly we can perform the same computations for a put option. If the spot price of the asset at expiration, S_τ, is lower then the strike price of a put option than the buyer of the option finds it convenient to buy the asset and exercise the option thus making a profit of $A - S_\tau$. Otherwise the option has no value. So we find that

$$\text{PresentValue}_{long-put} = -C_{put} + \max(A - S_\tau, 0) e^{-r\tau} \tag{8.57}$$

$$\text{PresentValue}_{short-put} = +C_{put} - \max(A - S_\tau, 0) e^{-r\tau} \tag{8.58}$$

They can be visually represented as

Conversely the present value of a selling the same put option is

The cost of the put option can be estimated by

$$C_{put} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(A - S_\tau, 0) p_\tau(S_\tau) dS_\tau \tag{8.59}$$

Note that the formulas are the similar but if we own a call option we make a profit when the price of the underlying asset exceeds the strike of the option, while if we own a put option we make a profit when the price of the underlying asset falls below the strike price of the option. In both cases the profit is proportional to the difference between the strike price and the spot price. If instead we sold an option we are in the opposite situation. We made a profit from the sale but we will have a loss if the option is exercised.

The questions that remain open are:

- How do we determine $p_\tau(S_\tau)$?
- How do we compute efficiently the above integrals?

8.7.8 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 X (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 on how the price S_τ evolves as 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 in time intervals of size Δ_t and we assume that in each time interval $\tau = n\Delta_t$ the variation in the asset price is

$$S_{n+1} = S_n u \text{ with probability } p \tag{8.60}$$

$$S_{n+1} = S_n d \text{ with probability } 1-p \tag{8.61}$$

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 $S_u u^i d^{n-i}$ is given by

$$\text{Prob}(S_\tau = S_u u^i d^{n-i}) = \binom{n}{i} p^i (1-p)^{n-i} \tag{8.62}$$

and therefore

$$C_{call} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i\leq n} \binom{n}{i} p^i (1-p)^{n-i} \max(S_u u^i d^{n-i} - A, 0) \tag{8.63}$$

and

$$C_{put} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i\leq n} \binom{n}{i} p^i (1-p)^{n-i} \max(A - S_u u^i d^{n-i}, 0) \tag{8.64}$$

Question: The parameters of this model are u, d and p. How would you measure them?

This works fine in this simple cases but the method is not easy to generalize to cases when the value of the option depends on the history of the asset (for example the asset is a stock that pays dividends). Moreover in order to increase precision one has to decrease Δ_t or redo the computation from the beginning.

Here is a Python code to simulate an asset price using a Binomial tree:

```
def BinomialSimulation(S0,u,d,p,n):
    data=[]
    S=S0
    for i in range(n):
        data.append(S)
        if uniform()<p:
            S=u*S
        else:
            S=d*S
    return data
```

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.(8.63) and (8.64) 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.

The Monte Carlo method that we see next is slower in the simple cases but is more general and therefore more powerful.

8.7.9 Pricing European Options - MC

We now follow the Black-Scholes model (without ever writing the Black-Scholes equation). 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 in time intervals of size Δ_t and we assume that in each time interval $t = n\Delta_t$ the variation in the asset price is

$$S_{n+1} = S_n \left[1 + \mu\Delta_t + \sigma\varepsilon_n\sqrt{\Delta_t}\right] \tag{8.65}$$

where $\forall n, \varepsilon_n$ can be a Bernoulli $(+1, -1)$ distributed random variable (Wiener process) or Gaussian distributed random variable with mean 0 and variance

1 (Ito Process). The choice depends on the model. From now on we will assume an Ito Process. There are three parameters in the above 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.
- σ is called volatility and it represents the amount of stochastic fluctuations of the asset over a time interval of one year.

The reason for $\sqrt{\Delta_t}$ is quite subtle and its origin requires stochastic calculus (beyond the scope of these notes). It makes sure that the interpretation of the parameters of the equation do not change when we change the time step Δ_t, i.e. if we measured μ and σ for a given Δ_t, we do not have to re-measure them when we change Δ_t.

Let's consider a particular case:

If $\sigma = 0$ eq.(8.65) becomes $S_{n+1} = S_n(1 + \mu\Delta_t)$ whose solution is

$$S_n = S_0(1 + \mu\Delta_t)^n \tag{8.66}$$

if we consider continuous time $t = n\Delta_t$

$$S(t) = S_0(1 + \mu\Delta_t)^{t/\Delta_t} \tag{8.67}$$

and in the limit $\Delta_t \to 0$

$$S(t) = S_0 e^{\mu t} \tag{8.68}$$

where we recognize the formula eq.(8.46) with $\mu = -r$. Therefore in absence of stochastic fluctuations the price of the asset would grow exponentially at an interest rate μ.

In general

$$\text{Mean of } \frac{\Delta S}{S} = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{n<N} \frac{S_{n+1} - S_n}{S_n} = \mu\Delta_t \tag{8.69}$$

$$\text{Variance of } \frac{\Delta S}{S} = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{n<N} \left[\frac{S_{n+1} - S_n}{S_n} - \mu\Delta_t\right]^2 = \sigma^2\Delta_t \tag{8.70}$$

This formula gives us a way to measure μ and σ from historic data.

Therefore one can compute the cost C of an option:

$$C = e^{-r\tau} \int_{-\infty}^{+\infty} f(x) p_\tau(x) dx \tag{8.71}$$

with $f(x) = \max(x - A, 0)$ for a call and $f(x) = \max(A - x, 0)$ for a put by using eq.(**??**):

$$\int_{-\infty}^{+\infty} f(x) p(x) dx \simeq \frac{1}{N} \sum_{i=0}^{i<N} f(x_i) \tag{8.72}$$

Therefore

$$C = e^{-r\tau} \frac{1}{N} \sum_{i=0}^{i<N} f(x_i) \tag{8.73}$$

where x_i ($i = 0...N$) are predictions for the spot price of the call option at expiration given from our model.

8.7.10 Pricing Any Options

There are few things to recognize:

- The value of an option at expiration is only identified by a 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 people agreed upon.
- We generated sets of spot prices for the asset at expiration date $\{x_i\}$ using a model for the stochastic temporal evolution of the asset, eq.(8.65). We can use another model and eq.(8.73) would still be valid.

Python code:

```
def MeasureMeanFromHistoric(historic, delta_t):
    '''''' This measures mu ''''''
    sum=0
    for i in range(len(historic)-1):
        sum=sum+(historic[i+1]-historic[i])/historic[i]
    return sum/(len(historic)-1)/delta_t

def MeasureVolatilityFromHistoric(historic, delta_t):
    '''''' This measures sigma ''''''
    sum=0
```

```
    for i in range(len(historic)-1):
        sum=sum+((historic[i+1]-historic[i])/historic[i])**2
    variance=(sum/(len(historic)-1)-(ComputeMean(historic)*delta_t)**2)
    volatility=sqrt(variance/delta_t)
    return volatility

def SimulateAsset(S0,mu,sigma,tau,delta_t,g):
    S=S0
    nsteps=int(tau/delta_t)
    for i in range(nsteps):
        S=S*(1+mu*delta_t+sigma*g.gaussian()*sqrt(delta_t))
    return S


def PriceOption(option,S0,mu,sigma,tau,delta_t,r,ap,g):
    sum=0.0
    sum2=0.0
    i=0
    while 1:
        i=i+1
        x_i=SimulateAsset(S0,mu,sigma,tau,delta_t,g)
        value_at_expiration=option(x_i)
        sum=sum+value_at_expiration
        sum2=sum2+value_at_expiration**2
        if i>100 and sqrt((sum2/i-(sum/i)**2)/i)<ap: break
    return exp(-r*tau)*sum/i

def LongCallOption(x,A=50):
    # Strike price of the option is $50
    return max(x-A,0)

def ShortCallOption(x,A=50):
    # Strike price of the option is $50
    return -max(x-A,0)

def LongPutOption(x,A=50):
    # Strike price of the option is $50
    return max(A-x,0)

def ShortPutOption(x,A=50):
    # Strike price of the option is $50
    return -max(A-x,0)

def StangleOption(x):
    # this represnts a portflio of two options called a strangle
    return LongCallOption(x,A=60)+ShortPutOption(x,A=40)

def TabulatePriceOption(option, minS, maxS, stepS,
                        mu, sigma, tau, delta_t, r, ap, g):
```

```
    print('asset price\toption price')
    for S in range(minS,maxS+stepS,stepS):
        print(S,'\n',PriceOption(option,S,mu,sigma,tau,delta_t,r,ap,g))
```

Note that:

- `option` is the function that gives the value of the option at expiration as a function of the stop price of the asset, X
- `minS` is the minimum current price of the asset to consider
- `maxS` is the maximum current price of the asset to consider
- `stepS` is the step to be used to compute S in range(minS,maxS,stepS)
- μ is the expected yearly rate of return from the asset measured from historic data using MeasureMeanFromHistoric
- σ is the yearly volatility of the asset measured from historic data using MeasureVolatilityFromHistoric
- τ is the time distance between now and the expiration date in years or [days]/[trading days]
- Δ_t is the time step of the simulation (for example $\Delta_t = 1.0/253$ indicates a daily simulation considering 252 trading days in one year rather than 365).
- r is the expected yearly rate of return from the Bank or other risk free source of funding.
- `ap` is the required absolute precision of the simulation (for example $0.01=1cent)
- `g` is the random generator to be used (such as the `MCG()`)

If we run

```
for days in [0,10,20,50]:
    TabulatePriceOption(LongCallOption,
                        minS=30,maxS=70,stepS=2,
                        mu=0.12,sigma=0.5,tau=days/253,
                        delta_t=1.0/253,r=0.05,ap=0.05,g=MCG())
```

which can be plotted as:

8.8 Markov Chain Monte Carlo (MCMC)

Until this point all out simulations were based on independent random variables. This means that we were able to generate each random number independently on 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 correlated random variables whose probability mass function

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{8.74}$$

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).

How do we generate such correlated random numbers?

8.9 Quadratic (not covered in class)

This technique only applies to the case when the $p(\mathbf{x})$ has the form

$$p(\mathbf{x}) \simeq e^{-\sum_{ij} \mathbf{A}_{ij} x_i x_j - \sum_i b_i x_i} \tag{8.75}$$

where A is a symmetric matrix with positive eigenvalues and b is a 1d array (a vector).

We can reduce this problem to that of finding another matrix $\mathbf{K}$ such that $\mathbf{K}^t\mathbf{A}\mathbf{K} = \mathbf{D}$ is a diagonal matrix. This is a linear algebra problem and we omit details here.

Once K is found one can define $\mathbf{y} = \mathbf{K}^{-1}\mathbf{x}$ and from eq.(8.75) and it follows that

$$p(\mathbf{y}) = p(y_0)p(y_1)...p(y_{n-1}) \tag{8.76}$$

where

$$p(y_i) \simeq e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \tag{8.77}$$

and

$$\sigma_i^2 = \frac{1}{2D_{ii}} \tag{8.78}$$

$$\mu_i = \sqrt{2}\sigma_i \sum_j b_j K_{ji}^{-1} \tag{8.79}$$

Therefore $\mathbf{x} = x_0, x_1, ..., x_{n-1}$ can be generated as follow:

1. Find **K** and **D**
2. Generate n independent Gaussian random numbers y_i with mean and variance given respectively by μ_i and σ_i.
3. Map $\mathbf{y} = (y_0, y_1, ..., \dot{y}_{n-1})$ into $\mathbf{x} = (x_0, x_1, ..., x_{n-1})$ by $\mathbf{x} = \mathbf{Ky}$.

x generated this way will have the required probability mass function.

8.10 Metropolis

The exact technique described above works only when the probability mass function has the form of eq.(8.75), moreover determining the matrix **K** is not always easy. The Metropolis algorithm provides a more general and simpler solution, although a slower one for the particular case described above.

Lets' 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 given by

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{8.80}$$

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 somehow 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

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 $p(\mathbf{x})$.

Example in Python:

```
class Metropolis:
    def __init__(self,generator):
        self.generator=generator

    def step(self,p,q):
        x_old=self.x
        x=q(self.generator)
        if p(x)/p(x_old)<self.generator.uniform():
            x=x_old
        self.x=x

def exampleP(x):
    return 6.0*(x[0]-x[1])**2

def exampleQ(generator):
    x0=generator.uniform()
    x1=generator.uniform()
    return [x0, x1]

def test_Metropolis():
    m=Metropolis(MCG())
    m.x=exampleQ(m.generator)
    for i in range(100):
        m.step(exampleP,exampleQ)
        print(m.x)

>>> test_Metropolis()
```

8.10.1 *The ising model*

```
class Ising:
    def __init__(self, n):
        s=[]
        for i in range(n):
            s.append([])
            for j in range(n):
                s[i].append([])
```

```
                for k in range(n):
                    s[i][j].append(0)
                    s[i][j][k]=+1
        self.s=s
        self.n=n

    def E(self):
        n=self.n
        s=self.s
        sum=0.0
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    sum=sum+s[i][j][k]*s[(i+1) % n][j][k]
                    sum=sum+s[i][j][k]*s[i][(j+1) % n][k]
                    sum=sum+s[i][j][k]*s[i][j][(k+1)%n]
        return sum

    def H(self):
        n=self.n
        s=self.s
        sum=0.0
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    sum=sum+s[i][j][k]
        return sum/n**3

    def probability(self,t):
        return exp(-self.E()/t)

    def randomize(self,generator):
        n=self.n
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    if generator.random()<0.5:
                        self.s[i][j][k]=+1
                    else:
                        self.s[i][j][k]=-1
```

8.11 Example: Metropolis Integration

We can use the Metropolis algorithm to compute nD integrals. For example

$$\int_0^1 \int_0^1 \sin(x_0 x_1)(x_0 - x_1)^2 dx_0 dx_1 \tag{8.81}$$

$$= \int_0^1 \int_0^1 \frac{1}{6} \sin(x_0 x_1) p(x_0, x_1) dx_0 dx_1 \tag{8.82}$$

$$= \frac{1}{N} \sum_{i=k}^{i<n+N} \frac{1}{6} \sin(x_0^{(i)} x_1^{(i)}) \tag{8.83}$$

Here is the program:

```
def test__metropolis_integral(k=100,n=10000):
    m=Metropolis(MCG())
    m.x=exampleQ(m.generator)
    # termalize
    for i in range(n):
        m.step(exampleP,exampleQ)
    # average
    sum=0.0
    for i in range(n):
        m.step(exampleP,exampleQ)
        sum=sum+1.0/6.0*sin(m.x[0]*m.x[1])
        print(sum/(i+1))

>>> test__metropolis_integral()
```

and here is the output:

```
0.0406617428277
0.0411113642801
0.0410627602329
0.0408950122005
0.0408211666869
...
0.0277582169669
0.0277445586923
0.0277309176305
```

The output slowly converges to the exact answer: 0.02 7229...

8.12 Example: average of random permutations

We are given random variables $x_0, x_1, ..., x_{n-1}$, each of them can have any value $0, 1, ..., n-1$ but they have to all be different. We can say that $x_0, x_1, ..., x_{n-1}$ form a permutation of $(0, 1, ..., n-1)$. We want to compute the average of the expression

$$\frac{1}{n}\left(|x_0 - x_1| + |x_0 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|\right) \tag{8.84}$$

over all permutations. Clearly problem of evaluating all permutations is exponential with n. The best approach is to generate sample random permutations using the Metropolis algorithm. In this particular case all possible permutations have the same probability therefore $p(\mathbf{x}^{(i+1)})/p(\mathbf{x}^{(i)})$ in step #4 is always one and steps #3 and #4 become unnecessary. In order to generate random permutations $\mathbf{x}^{(i+1)}$ one may proceed by swapping two random elements in $\mathbf{x}^{(i)}$. Here is the code:

```
def test__permutation_distance(n=10,generator=MCG()):
    x=range(n)
    sum=0.0
    counter=0
    while 1:
        counter=counter+1
        i=generator.randint(0,n-1)
        j=generator.randint(0,n-1)
        if i!=j:
            # swap i and j
            x[i],x[j] = x[j],x[i]
            sum2=0.0;
            for k in range(n+1):
                sum2=sum2+abs(x[i % n]-x[j % n])
            sum=sum+sum2/n
            print(sum/counter)


>>> test__permutation_distance(10)
```

Notice that the program that generates all combinations runs in $\Theta(n!)$ while the program above converges quite fast to the right answer.

8.13 Metropolis on Locality

Lets' consider a probability mass function of the form

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) = e^{\sum_{jk} A_{ij}(x_j)x_k} \tag{8.85}$$

where $A_{jk}(x_j)$ is some function of x_j. The running time to evaluate the above function and therefore to evaluate Metropolis step #4 is $\Theta(n^2)$ because of the two loops involved in $\sum_{ij}$. It is possible to choose step #2 so that step #4 becomes faster. In fact the Metropolis does not give a unique prescription on how to choose $\mathbf{x}^{(i+1)}$ given $\mathbf{x}^{(i)}$ in step #2. We can decide to replace #2 with the following step:

2 Generate a random integer k in $[0, n)$ and a random float δ and let

$$\forall j \neq k, x_j^{(i+1)} = x_j^{(i+1)} \text{ and } x_k^{(i+1)} = \delta \tag{8.86}$$

With this choice and a desired probability mass function given by eq.(8.85) step #4 becomes

4 If $\exp((\delta - x_k^{(i)}) \sum_{jk} A_{jk}(x_j^{(i)})) < z$ then $x_k^{(i+1)} = x_k^{(i)}$

Therefore if the desired probability mass function has the form of eq.(8.85) one can change only one random variable for each Metropolis step thus making the Metropolis step run faster. Apparently there is no obvious advantage from this procedure but it can make the sets $\mathbf{x}^{(i)}$ converge faster to the desired distribution.

8.14 Simulated Annealing

8.14.1 Protein Folding

```
class Protein:
    def __init__(self,amino):
        self.amino=amino

    def next_atom(self,p,move):
        if move==1:
```

```
            p=(p[0]+1,p[1],p[2])
        elif move==2:
            p=(p[0]-1,p[1],p[2])
        elif move==3:
            p=(p[0],p[1]+1,p[2])
        elif move==4:
            p=(p[0],p[1]-1,p[2])
        elif move==5:
            p=(p[0],p[1],p[2]+1)
        elif move==6:
            p=(p[0],p[1],p[2]-1)
        return p

    def energy(self):
        amino=self.amino
        folding=self.folding
        n=len(amino)
        energy=0.0
        db={}
        p=(0,0,0)
        for i in range(len(amino)):
            if db.has_key(p):
                return None
            else:
                db[p]=amino[i]
            if amino[i]=='H':
                r=(p[0]+1,p[1],p[2])
                if db.has_key(r):
                    if db[r]=='H':
                        energy=energy-1
                r=(p[0]-1,p[1],p[2])
                if db.has_key(r):
                    if db[r]=='H':
                        energy=energy-1
                r=(p[0],p[1]+1,p[2])
                if db.has_key(r):
                    if db[r]=='H':
                        energy=energy-1
                r=(p[0],p[1]-1,p[2])
                if db.has_key(r):
                    if db[r]=='H':
                        energy=energy-1
                r=(p[0],p[1],p[2]+1)
                if db.has_key(r):
                    if db[r]=='H':
                        energy=energy-1
                r=(p[0],p[1],p[2]-1)
                if db.has_key(r):
                    if db[r]=='H':
```

```
                        energy=energy-1
                if i>0 and amino[i-1]=='H':
                    energy=energy+1
            if i==n-1:
                break
            else:
                p=self.next_atom(p,folding[i])
        return energy
    def folding(self, nsteps=300):
        amino=self.amino
        self.folding=folding=[]
        n=len(amino)
        counter=1
        for i in range(n-1):
            folding.append(1)
        old_energy=self.energy()
        sum_energy=old_energy
        min_energy=old_energy
        min_folding=folding
        while true:
            i=randint(1,n-2)
            j=randint(1,6)
            k=folding[i]
            folding[i]=j
            energy=self.energy()
            if energy != None:
                if exp(-energy)/exp(-old_energy)>random():
                    old_energy=energy
                else:
                    folding[i]=k
                    energy=old_energy
                print counter, sum_energy/counter, min_energy

                sum_energy=sum_energy+energy
                counter=counter+1

                if energy<min_energy:
                    min_energy=energy
                    min_folding=copy(folding)
            else:
                folding[i]=k
            if counter==nsteps:
                break
        return sum_energy/counter, min_energy, min_folding

def test_folding(amino=''HHPHPHHPHPHPPPPHHPHPHHPHPPHHHHP''):
    protein=Protein(amino)
    avg,min,folding=protein.folding(100)
    protein.printVRML(``folding.wrl'')
```

9
Parallel Algorithms

9.1 Parallel Architectures

Consider a program that performs a computation:

```
y = f(x)
z = g(x)
```

The function $g(x)$ does not depend on the result of the function $f(x)$ and therefore the two functions could be computed independently and in parallel.

Often large problems can be divided into smaller computational problems, which can be solved concurrently ("in parallel") using different processing units (nodes, CPUs, Cores). This is called Parallel Computing. Algorithms designed to work in parallel are called Parallel Algorithms.

Programs can be parallelized at many levels: bit-level, instruction level, data, and task parallelism. Bit level parallelism is usually implemented in hardware. Instruction level parallelism is also implemented in hardware in modern multi pipeline CPUs. Data parallelism is usually referred as SIMD and we will say more about it later. Task parallelism is also referred as MIMD.

Historically parallelism was found applications in high-performance computing but today it is employed in many devices including common cell phones. The reason is heat dissipation. It is getting harder and harder to

improve speed by increasing CPU frequency because there is a physical limit to how much we can cool the CPU. So the recent trend is keeping frequency constant and increasing the number of processing units on the same chip.

Parallel architectures are classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Optimizing an algorithm to run on a parallel architacture is not an easy task and it depends on the type of parallelism and details of the architecture.

In this chapter, we will learn how to classify architectures, compute running times of parallel algorithms and measure their perfomance and scaling.

We will learn how to write parallel programs using standard programming patterns as building blocks.

For some parts of this chapter, we will use a simulator called `PSim` and it is written in Python. Its performances will only scale on multicore machines but it will allow us to emulate various network topologies.

9.1.1 Flynn taxonomy

The classification of parallel computer architectures is known as Flynn's taxonomy and due to Michael J. Flynn in 1966.

Flynn identified the following architectures (text from Wikipedia):

- **Single Instruction, Single Data stream (SISD)**

  A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single Instruction Stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single Data Stream (DS) i.e. one operation at a time Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently

manufactured PCs have multiple processors) or old mainframes.

- **Single Instruction, Multiple Data streams (SIMD)** A computer which exploits multiple data streams against a single instruction stream to perform operations that may be naturally parallelized (e.g. an array processor or GPU).
- **Multiple Instruction, Single Data stream (MISD)**

  Multiple instructions operate on a single data stream. Uncommon architecture that is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.
- **Multiple Instruction, Multiple Data streams (MIMD)** Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space (using threads) or a distributed memory space (using a message passing protocol such as MPI).

MIMD can be further subdivided into:

- **Single Program, Multiple Data (SPMD)** Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data. Also referred to as "Single Process, multiple data" - the use of this terminology for SPMD is erroneous and should be avoided, SPMD is a parallel execution model and assumes multiple cooperating processes executing a program. SPMD is the most common style of parallel programming.
- **Multiple Program Multiple Data (MPMD)** Multiple autonomous processors simultaneously operating at least 2 independent programs. Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "Manager/Worker" strategy), which runs one program that farms out data to all the other nodes which all run a second program. Those other nodes then return their results directly to the manager. An example of this would be the Sony PlayStation 3 game console, with its SPU/PPU processor architecture.

Google's Map-Reduce also falls under this category.

An embarrassingly parallel workload (or embarrassingly parallel problem) is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks.

The Manager/Worker node strategy, when workers do not need to communicate with each other is an example of "Embarrassingly Parallel" problem.

9.1.2 Network Topologies

In the MIMD case multiple copies of the same problem runs concurrently (on different data subsets and branching differently thus performing different instructions) on different processing units and they exchange information using a network. How fast they can communicate depends on the network characteristics identified by the network topology and the latency and bandwith of the individual links of the network.

Normally we classify network topologies based on the following taxonomy:

- **Completely Connected** Each node is connected by a directed link to each other node.
- **Bus Topology** All nodes are connected to the same single cable. Each computer can therefore communicate with each other computer using one and the same bus cable. The limitation of this approach is that the communication bandwidth is limited by the bandwidth of the cable. Most bus networks only allow two machines to communicate with each other at one time (with the exception of one to many broadcast messages). While two machines communicate the others are stuck waiting. The bus topology is the most inexpensive but also slow and constitutes a single point of failure.
- **Switch Topology (Star topology)** In local area networks with a switch topology, each computer is connected via a direct link to a central device, usually a switch and it resembles a star. Two computers can communicate

using two links (to the switch and from the switch). The central point of failure is the switch. The switch is usually intelligent and can re-route the messages from any computer to any other computer. If the switch has sufficient bandwidth it can allow multiple computers to talk to each other at the same time. For example for 10Gbit/s links and an 80Gbit/s switch, 8 computers can talk to each other (in pairs) at the same time.

- **Mesh Topology** In a mesh topology computers are assembled into an array (1D,2D,etc.) and each computer is connected via a direct link to the computers immediately close (left, right, above below, etc.). Next neighbour communication is very fast because it involves a single link therefore low latency. For two computers not physically close to communicate it is necessary to reroute messages. The latency is proprtional to the distance in links between the computers. Some meshes do not support this kind of rerouting because the extra logic, even if unused, may be cause for extra latency. Meshes are ideal for solving numerical problems such as solving differential equations because they can be naturally mapped into this kind of topology.

- **Torus Topology** Very similar to a mesh topology (1D, 2D, 3D, etc), except that the networks wraps around the edges. So in one dimenation for example node i is connected to $(i+1)\%p$ where p is total number of nodes. A 1D torus is called a Ring Network.

- **Tree Network** The tree topology looks like a tree where the computer may be associated to every tree node or every leaf only. The tree links are the communication link. For a binary tree, each computer only talks to its parent and its two children nodes. One node is special because it has no parent and it is the root node. Tree networks are ideal for global operations such as broadcasting and for sharing IO devices such as disks. If the IO device is connected to the root node, every other computer can communicate with it using only $\log p$ links (where p is the number of computers connected). Moreover, each subset of a tree network is also a tree network. This makes it easy to distribute subtasks to different subsets of the same architecture.

- **Hypercube** This network assumes 2^d nodes and each node corresponds

to a vertex of a hypercube. Nodes are connected by direct links, which correspond to the edges of the hypercube. Its importance is more academical than practical although some ideas from hypercube networks are implemented in some algorithms.

If we identify each node on the network with a unique integer number called its rank, we write explicit code to determine if two nodes i and j are connected for each network topology:

Listing 9.1: in file: `psim.py`

```
import os, string, cPickle, time, math

def BUS(i,j):
    return True

def SWITCH(i,j):
    return True

def MESH1(p):
    return lambda i,j,p=p: (i-j)**2==1

def TORUS1(p):
    return lambda i,j,p=p: (i-j+p)%p==1 or (j-i+p)%p==1

def MESH2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q)**2,(i/q-j/q)**2) in [(1,0),(0,1)]

def TORUS2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q+q)%q,(i/q-j/q+q)%q) in [(0,1),(1,0)] or \
                           ((j%q-i%q+q)%q,(j/q-i/q+q)%q) in [(0,1),(1,0)]
def TREE(i,j):
    return i==int((j-1)/2) or j==int((i-1)/2)
```

9.1.3 *Network Characteristics*

- **Number of links**
- **Diameter**: The max distance between any two nodes measured as minimum number of links connecting them. Smaller diameter means smaller latency.

- **Bisection Width**: The minimum number of links one has to cut to turn the network into two disjoint networks. Higher bisection width means higher reliability of the network.
- **Arc Connectivity**: The number of different paths connecting any two nodes. Higher connectivity means higher bandwith and higher reliability.

Network	Links	Diameter	Width	Connectivity
completely connected	1		$p^2/4$	$p(p-1)/2$
switch	$p(p-1)$	2	*n.d.*	1
1D mesh	$p-1$	$p-1$	1	1
2D mesh	$2(p^{\frac{1}{2}}-1)p^{\frac{1}{2}}$	$2(p^{\frac{1}{2}}-1)$	$p^{\frac{1}{2}}$	$p^{\frac{1}{2}}$
3D mesh	$3(p^{\frac{1}{3}}-1)p^{\frac{2}{3}}$	$3(p^{\frac{1}{3}}-1)$	$p^{\frac{2}{3}}$	$p^{\frac{2}{3}}$
1D torus	p	$p/2$	2	2
2D torus	$2p$	$2p^{\frac{1}{2}}$	$2p^{\frac{1}{2}}$	$2p^{\frac{1}{2}}$
3D torus	$3p$	$3/2p^{\frac{1}{3}}$	$2p^{\frac{1}{3}}$	$2p^{\frac{2}{3}}$
hypercube	$p/2\log p$	$\log p$	$p/2$	$\log 2$
tree	$p-1$	$\log p$	1	1

[FILL THIS, IT IS IN THE BOOK]

9.2 *Actual Architectures*

Most actual supercomputers implement a variety of taxonomies and topologies simultaneously. A modern supercomputer has many nodes, each node has many CPUs, each CPU has many cores, and each core implements SIMD instructions. Each core has it own cache, each CPU has its own cache, and each node has its own memory shared by all threads running on that one node. Nodes communicate with each other using multiple networks (typically a multidimensional mesh for point to point communications and a tree network for global communication and general disk IO).

This makes writeing parallel programs very difficult. Parallel programs must be optimized for each specific archicture.

9.3 Basic definitions

9.3.1 Latency and bandwidth

The time it takes for a message of size m (in bytes) to travel over a wire can be broken into two components: a fixed overall time that does not depend on the size of the message, called latency (and indicated with t_s); a time proportional to the message size, called inverse bandwitdh (and indicated with t_s).

Think of a pipe of length L and section s and you want to pump m litres of water thought the pipe at velocity v. From the moment you start pumping, it takes L/v seconds before the water starts arriving at the other end of the pipe. From that moment it will take m/sv for all the water to arrive at destination. In this analogy L/v is the latency t_s, sv is the bandwidth, and $t_w = 1/sv$.

The total time to send the message (or the water) is:

$$T(m) = t_s + t_w m \tag{9.1}$$

Consider a numerical algorithm with a running time $\Theta(f(n))$ but requires an input of size n and an produces output of size also n. When conputing the actual running time we should consider the input/output performed by the CPU. If we assume the CPU does not need to perform any IO while doing the computation (because the data fits in cache) then total running time is

$$T = f(n)t_a + (t_s + t_w n) + (t_s + t_w n) \tag{9.2}$$

where t_a is the time to perform one elementary instruction. Because typically $t_s >> t_w >> t_a$ we now see that we have a problem: even very fast algorithms can be limited by the time required to feed the data to the processing units. The faster the running time of the algorithm ignoring communication, the more communications adds overhead (in percent).

9.3.2 *Speedup*

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \tag{9.3}$$

where T_1 is the time it takes to run the algorithm on an imput of size n on 1 processing unit (Node for example) and T_p is the time it takes to run the same algorithm on the same input using p nodes in parallel.

9.3.3 *Efficiency*

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_1(n)}{pT_p(n)} \tag{9.4}$$

Notice that in case of perfect parallelization (impossible) $T_p = T_1/p$ and therefore $E_p(n) = 1$.

9.3.4 *Admahl's Law*

Consider an algorithm that can be parallized but one faction α of its total sequential running time αT_1 cannot be parallelized. that means that $T_p = \alpha T_1 + (1-\alpha)T_1/p$ this yields:

$$S_p = \frac{1}{\alpha + (1-\alpha)/p} < \frac{1}{\alpha} \tag{9.5}$$

and

$$E_p = \frac{1}{p\alpha + (1-\alpha)} < \frac{1}{1-\alpha} \tag{9.6}$$

Therefore both the speedup and the efficiency are theoretically limited.

This raises an issue: how do we choose p given n?

9.3.5 *Isoefficiency*

Given a value of efficiency that we choose as target $E = \bar{E}$ we can ask what is the maximum size of a problem we can solve given p nodes? The answer can be found by solving n the following equation:

$$E_p(n) = \bar{E} \tag{9.7}$$

For example given an algorithm with

$$T_p = n^2 t_a + 2p(t_s + t_w n/p) \tag{9.8}$$

when obtain:

$$E_p = \frac{1}{p + 2p^2(t_s + t_w n/p)/(n^2 t_a)} = \bar{E} \tag{9.9}$$

Which solved in n yields:

$$n(p) = ... \tag{9.10}$$

We can see the function is monotonically increasing, the larger p, the larger n, but it also has a horizontal asyntote $maxn = ...$

9.3.6 *Cost*

The cost of a computation is equal to the time it takes to run on each node, multiplied by the number of nodes involved in the computation:

$$C_p(n) = pT_p(n) \tag{9.11}$$

Notice that also

$$C_p(n) = pT_p(n) = p\alpha T_1(n) + (1 - alpha)T_1(n) \tag{9.12}$$

therefore

$$\frac{\mathrm{d}C_p(n)}{\mathrm{d}p} = \alpha T_1(n) > 0 \tag{9.13}$$

This means that for a fixed problem size n the more an algorithm is parallelized, the more it costs to run it (because it gets less and less efficienct).

So why do people run parallel algorithms? Consider weather forecasts. To predict tomorrow's weather, it would take months on a PC (if possible at all) and hours on a supercomputer. We need the forecast before tomorrow therefore we have no choice. Our cost formula as defined above does not take into account the added value of getting a result sooner rather than later.

9.3.7 *Cost-optimality*

With the disclaimer above we define cost optimality as the choice of p (as function of n) which make the cost scale proportional to $T_1(n)$:

$$pT_p(n) \propto T_1(n) \tag{9.14}$$

Or in other words:

$$p'T_p(n) + p\left(\frac{\partial T_p(n)}{\partial n} + p'\frac{\partial T_p(n)}{\partial p}\right) = \frac{\mathrm{d}T_1(n)}{\mathrm{d}n} \tag{9.15}$$

where $p' = \frac{\mathrm{d}p}{\mathrm{d}n}$

In general this is a non-trivial differential equation to solve but because we only need the order of growth of p we solve it by empirically.

9.4 *Message passing fundamentals*

Consider the following program:

```
def f():
    import os
    if os.fork(): print(True)
    else: print(False)
f()
```

The output of the current program is

```
True
False
```

The function `fork` creates a copy of the current process (a child). The parent process returns the id of the child process and the child process returns 0. Therefore the `if` condition is both true and false, just on different processes.

A real parallel MIMD program consists of many copies of the same program although the copies are not actually created by the fork function. They are created by some startup script which submits the program to the various nodes. In the case of MPI the script is called `mpirun`.

In order to be able to test parallel programs we will use the following program which emulates a parallel MIMD architecture by using fork. The program below also implements a message passing protocol between the processes using pipes connecting them. The program can simulate multiple architectures by using the topologies defined above:

Listing 9.2: in file: `psim.py`

```
class PSim(object):
    def log(self,message):
        """
        logs the message into self._logfile
        """
        if self.logfile!=None:
            self.logfile.write(message)

    def __init__(self,p,topology=SWITCH,logfilename=None):
        """
        forks p-1 processes and creates p*p
        """
        self.logfile = logfilename and open(logfile,'w')
        self.topology = topology
        self.log("START: creating %i parallel processes\n" % p)
        self.nprocs = p
        self.pipes = {}
        for i in range(p):
            for j in range(p):
                self.pipes[i,j] = os.pipe()
        self.rank = 0
        for i in range(1,p):
            if not os.fork():
                self.rank = i
```

```
                break
        self.log("START: done.\n")

    def _send(self,j,data):
        """
        sends data to process #j
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: send(%i,...) failed!\n" % (self.rank,j))
            raise Exception
        self.log("process %i: send(%i,%s) starting...\n" % \
                 (self.rank,j,repr(data)))
        s = cPickle.dumps(data)
        os.write(self.pipes[self.rank,j][1], string.zfill(str(len(s)),10))
        os.write(self.pipes[self.rank,j][1], s)
        self.log("process %i: send(%i,%s) success.\n" % \
                 (self.rank,j,repr(data)))

    def send(self,j,data):
        if not self.topology(self.rank,j):
            raise RuntimeError, 'topology violation'
        self._send(j,data)

    def _recv(self,j):
        """
        returns the data recvd from process #j
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: recv(%i) failed!\n" % (self.rank,j))
            raise RuntimeError
        self.log("process %i: recv(%i) starting...\n" % (self.rank,j))
        try:
            size=int(os.read(self.pipes[j,self.rank][0],10))
            s=os.read(self.pipes[j,self.rank][0],size)
        except Exception, e:
            self.log("process %i: COMMUNICATION ERROR!!!\n" % (self.rank))
            raise e
        data=cPickle.loads(s)
        self.log("process %i: recv(%i) done.\n" % (self.rank,j))
        return data

    def recv(self,j):
        if not self.topology(self.rank,j):
            raise RuntimeError, 'topology violation'
        return self._recv(j)
```

An instance of a class `PSim` represents what is called *communicator*. That is an object that can be used to determine total number of parallel

processes, the own rank, send messages to other processes as well as receive messages from them.

`send` and `recv` represent the simplest type of communication pattern, a point to point communication.

More interesting patterns are global communication patterns implemeneted on top of send and recv:

9.4.1 *Broadcast*

The simplest type of broadcast is the one-2-all which consist of one process (source) sending a message (value) to every other process. A more complex braodcast is when each process broadcast a message simultaneously of each node receives the list of values concatenated in a list and ordered by the rank of the sender:

Listing 9.3: in file: `psim.py`

```
    def one2all_broadcast(self, source, value):
        self.log("process %i: BEGIN one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        if self.rank==source:
            for i in range(0, self.nprocs):
                if i!=source:
                    self._send(i,value)
        else:
            value=self._recv(source)
        self.log("process %i: END one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        return value

    def all2all_broadcast(self, value):
        self.log("process %i: BEGIN all2all_broadcast(%s)\n" % \
                 (self.rank, repr(value)))
        vector=self.all2one_collect(0,value)
        vector=self.one2all_broadcast(0,vector)
        self.log("process %i: END all2all_broadcast(%s)\n" % \
                 (self.rank, repr(value)))
        return vector
```

We have implemented the all-to-all broadcast using a trick. We send collected all values at node with rank 0 (via a function collect) and then

we did a one-to-all broadcast of the entire list from node 0. In general the implementation depends on the topology of the available network.

9.4.2 *Scatter and collect*

The all-to-one collect pattern works as follows. Every process sends a value to process `destination` which receives the values in a list ordered according to the rank of the senders:

Listing 9.4: in file: `psim.py`

```
    def one2all_scatter(self,source,data):
        self.log('process %i: BEGIN all2one_scatter(%i,%s)\n' % \
                 (self.rank,source,repr(data)))
        if self.rank==source:
             h, remainder = divmod(len(data),self.nprocs)
             if remainder: h+=1
             for i in range(self.nprocs):
                 self._send(i,data[i*h:i*h+h])
        vector = self._recv(source)
        self.log('process %i: END all2one_scatter(%i,%s)\n' % \
                 (self.rank,source,repr(data)))
        return vector

    def all2one_collect(self,destination,data):
        self.log("process %i: BEGIN all2one_collect(%i,%s)\n" % \
                 (self.rank,destination,repr(data)))
        self._send(destination,data)
        if self.rank==destination:
            vector = [self._recv(i) for i in range(self.nprocs)]
        else:
            vector = []
        self.log("process %i: END all2one_collect(%i,%s)\n" % \
                 (self.rank,destination,repr(data)))
        return vector
```

9.4.3 *Reduce*

The all-to-one reduce pattern is very similar to the collect except that the `destination` does not receive the entire list of values but some aggregated information about the values. The aggregation must be performed using a commutative binary function $f(x,y) = f(y,x)$. This guarantees that the

reduction from the values to down in any order and thus be optimized for different network topologies.

The all-to-all reduce is similar to reduce but every process will get the result of the reduction, not just one `destination` node. This may be achieved by an all-to-one reduce followed by a one-to-all broadcast.

Listing 9.5: in file: `psim.py`

```
    def all2one_reduce(self,destination,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        self._send(destination,value)
        if self.rank==destination:
            result = reduce(op,[self._recv(i) for i in range(self.nprocs)])
        else:
            result = None
        self.log("process %i: END all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result

    def all2all_reduce(self,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        result=self.all2one_reduce(0,value,op)
        result=self.one2all_broadcast(0,result)
        self.log("process %i: END all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result
```

And here are some examples of reduce operation

Listing 9.6: in file: `psim.py`

```
        @staticmethod
        def sum(x,y): return x+y
        @staticmethod
        def mul(x,y): return x*y
        @staticmethod
        def max(x,y): return max(x,y)
        @staticmethod
        def min(x,y): return min(x,y)
```

9.4.4 *Barrier*

Another global communication pattern is the barrier. It forces all process when they reach the barrier to stop and wait until all the other processes have reached the barrier. Think of runners gathering at the start line of a race; when all the runners are there, the race can start.

Here we implement it using a simple all-to-all broadcast.

Listing 9.7: in file: `psim.py`

```
    def barrier(self):
        self.log("process %i: BEGIN barrier()\n" % (self.rank))
        self.all2all_broadcast(0)
        self.log("process %i: END barrier()\n" % (self.rank))
        return
```

The use of `barrier` is usually a sympton of bad code because it forces parallel processes to wait for other processes without data actually being transferred.

9.5 *Examples of MIMD programs*

Listing 9.8: in file: `psim.py`

```
def test():
    comm=PSim(5,SWITCH)
    if comm.rank==0: print('start test')
    a=sum(comm.all2all_broadcast(comm.rank))
    comm.barrier()
    b=comm.all2all_reduce(comm.rank)
    if a!=10 or a!=b:
        print('from process', comm.rank)
        raise Exception
    if comm.rank==0: print('test passed')

if __name__=='__main__': test()
```

Listing 9.9: in file: `psim.py`

```
def scalar_product_test1(n,p):
    import random
    from psim import PSim
```

```
    comm = PSim(p)
    h = n/p
    if comm.rank==0:
        a = [random.random() for i in range(n)]
        b = [random.random() for i in range(n)]
        for k in range(1,p):
            comm.send(k, a[k*h:k*h+h])
            comm.send(k, b[k*h:k*h+h])
    else:
        a = comm.recv(0)
        b = comm.recv(0)
    scalar = sum(a[i]*b[i] for i in range(h))
    if comm.rank == 0:
        for k in range(1,p):
            scalar += comm.recv(k)
        print(scalar)
    else:
        comm.send(0,scalar)
```

Listing 9.10: in file: psim.py

```
def scalar_product_test2(n,p):
    import random
    from psim import PSim
    comm = PSim(p)
    a = b = None
    if comm.rank==0:
        a = [random.random() for i in range(n)]
        b = [random.random() for i in range(n)]
    a = comm.one2all_scatter(0,a)
    b = comm.one2all_scatter(0,b)

    scalar = sum(a[i]*b[i] for i in range(len(a)))

    scalar = comm.all2one_reduce(0,scalar)
    if comm.rank == 0:
        print(scalar)
```

Listing 9.11: in file: psim.py

```
def mergesort(A, p=0, r=None):
    if r is None: r = len(A)
    if p<r-1:
        q = int((p+r)/2)
        mergesort(A,p,q)
        mergesort(A,q,r)
        merge(A,p,q,r)

def merge(A,p,q,r):
```

```
    B,i,j = [],p,q
    while True:
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==q:
            while j<r:
                B.append(A[j])
                j=j+1
            break
        if j==r:
            while i<q:
                B.append(A[i])
                i=i+1
            break
    A[p:r]=B

def mergesort_test(n,p):
    import random
    from psim import PSim
    comm = PSim(p)
    if comm.rank==0:
        data = [random.random() for i in range(n)]
        comm.send(1, data[n/2:])
        mergesort(data,0,n/2)
        data[n/2:] = comm.recv(1)
        merge(data,0,n/2,n)
        print(data)
    else:
        data = comm.recv(0)
        mergesort(data)
        comm.send(0,data)
```

9.6 *mpi4py*

The Psim simulator does not provide any actual speedup unless you have multiple cores or processors for the forked processes to run on. A better approach would be to use `mpi4py` which is a Python interface to MPI.

You can download `mpi4py` from `http://mpi4py.scipy.org/`

Here is an example:

```
from mpi4py import MPI
```

```
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5
6  if rank == 0:
7     data = {'a': 7, 'b': 3.14}
8     comm.send(data, dest=1, tag=11)
9  elif rank == 1:
10    data = comm.recv(source=0, tag=11)
```

The `comm` object of class `MPI.COMM_WORLD` plays a similar role as the `PSim` object of previous section. The MPI `send` and `recv` functions are very similar to the `PSim` equivalent ones except that they require details about the type of the data beying transferred, and a communication tag. The tag allows node A to send node B multiple messages and B can receive them out of order using the tag as identifier.

9.7 *multiprocessing and threading*

SAY MORE XXXX

9.8 *Master-worker and Map-reduce*

XXXX REWRITE BELOW

Map-Reduce is a framework for processing highly distributable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes use the same hardware) or a grid (if the nodes use different hardware). Computational processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). "Map" step (implemented below in a function `mapfn`): The master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node. "Reduce" step (implemented below in a function

`reducefn`): The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve. MapReduce allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the others, all maps can be performed in parallel – though in practice it is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase - provided all outputs of the map operation that share the same key are presented to the same reducer at the same time. While this process can often appear inefficient compared to algorithms that are more sequential, MapReduce can be applied to significantly larger datasets than "commodity" servers can handle – a large server farm can use MapReduce to sort a petabyte of data in only a few hours. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data is still available.

The `mapfn` and `reducefn` functions of MapReduce are both defined with respect to data structured in (key, value) pairs. `mapfn` takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$\text{mapfn}(k1, v1) \rightarrow (k2, v2) \tag{9.16}$$

The `mapfn` function is applied in parallel to every item in the input dataset. This produces a list of (k2,v2) pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys. The `reducefn` function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$\text{reducefn}(k2, [\text{list of } v2]) \rightarrow (k2, v3) \tag{9.17}$$

Each `reducefn` call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list. Thus

the MapReduce framework transforms a list of (key, value) pairs into a list of values. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines all the values returned by map. It is necessary but not sufficient to have implementations of the map and reduce abstractions in order to implement MapReduce. Distributed implementations of MapReduce require a means of connecting the processes performing the `mapfn` and `reducefn` phases. This may be a distributed file system. Other options are possible, such as direct streaming from mappers to reducers, or for the mapping processors to serve up their results to reducers that query them.

Here is a non-parallel implementation:

```
def mapreduce(mapper,reducer,data):
    """
    >>> def mapfn(x): return x%2, 1
    >>> def reducefn(key,values): return len(values)
    >>> data = range(100)
    >>> print(mapreduce(mapfn,reducefn,data))
    {0: 50, 1: 50}
    """
    partials = {}
    results = {}
    for item in data:
        key,value = mapper(item)
        if not key in partials:
            partials[key]=[value]
        else:
            partials[key].append(value)
    for key,values in partials.items():
        results[key] = reducer(key,values)
    return results
```

And here is an example which we can use to find how many random DNA strings contain the subsequence 'ACCA':

```
>>> from random import choice
>>> strings = [''.join(choice('ATGC') for i in range(10))
...            for j in range(100)]
>>> def mapfn(string): return ('ACCA' in string, 1)
>>> def reducefn(check, values): return len(values)
>>> print(mapreduce(mapfn,reducefn,strings))
{False: ..., True: ...}
```

The important thing about the above code is that there are two loops in Map-Reduce. Each loops consists of executing tasks (map tasks, and reduce tasks) which are independent from each other (all the maps are independent, all the reduce are independent, but the reduce depend on the maps). Because they are independent they can be executed in parallel and by different processes.

The code about is the source of a library called `mincemeat` with implement parallel MapReduce. The workers connect and authenticate to the server using a password and the use request tasks to executed. The server accepts connections and assigns the map tasks and the reduce tasks to the workers.

The communication if performed using asynchronous sockets which means neither workers nor the master is ever in a wait state. The code is event based and communication only happens when a socket connecting the master to a worker is ready for a write (task assignment) or a read (task completed).

The code is also failsafe because if a worker closes the connection prematurely, the task is re-assigned to another worker.

`mincemeat` uses the python libraries `asyncore` and `asynchat` to implement the communication patterns, for which we refer to the Python documentation.

In the code a Worker is called a Client and we decided not to change it.

```
#!/usr/bin/env python

############################################################################
# Copyright (c) 2010 Michael Fairley
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
```

```
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
##############################################################################

import asynchat
import asyncore
import cPickle as pickle
import hashlib
import hmac
import logging
import marshal
import optparse
import os
import random
import socket
import sys
import types

VERSION = 0.0
DEFAULT_PORT = 11235

class Protocol(asynchat.async_chat):
    def __init__(self, conn=None):
        if conn:
            asynchat.async_chat.__init__(self, conn)
        else:
            asynchat.async_chat.__init__(self)
        self.set_terminator("\n")
        self.buffer = []
        self.auth = None
        self.mid_command = False

    def collect_incoming_data(self, data):
        self.buffer.append(data)

    def send_command(self, command, data=None):
        if not ":" in command:
            command += ":"
        if data:
            pdata = pickle.dumps(data)
            command += str(len(pdata))
            logging.debug( "<- %s" % command)
            self.push(command + "\n" + pdata)
        else:
```

```
65            logging.debug( "<- %s" % command)
66            self.push(command + "\n")
67
68        def found_terminator(self):
69            if not self.auth == "Done":
70                command, data = (''.join(self.buffer).split(":",1))
71                self.process_unauthed_command(command, data)
72            elif not self.mid_command:
73                logging.debug("-> %s" % ''.join(self.buffer))
74                command, length = (''.join(self.buffer)).split(":", 1)
75                if command == "challenge":
76                    self.process_command(command, length)
77                elif length:
78                    self.set_terminator(int(length))
79                    self.mid_command = command
80                else:
81                    self.process_command(command)
82            else: # Read the data segment from the previous command
83                if not self.auth == "Done":
84                    logging.fatal("Recieved pickled data from unauthed source")
85                    sys.exit(1)
86                data = pickle.loads(''.join(self.buffer))
87                self.set_terminator("\n")
88                command = self.mid_command
89                self.mid_command = None
90                self.process_command(command, data)
91            self.buffer = []
92
93        def send_challenge(self):
94            self.auth = os.urandom(20).encode("hex")
95            self.send_command(":".join(["challenge", self.auth]))
96
97        def respond_to_challenge(self, command, data):
98            mac = hmac.new(self.password, data, hashlib.sha1)
99            self.send_command(":".join(["auth", mac.digest().encode("hex")]))
100           self.post_auth_init()
101
102       def verify_auth(self, command, data):
103           mac = hmac.new(self.password, self.auth, hashlib.sha1)
104           if data == mac.digest().encode("hex"):
105               self.auth = "Done"
106               logging.info("Authenticated other end")
107           else:
108               self.handle_close()
109
110       def process_command(self, command, data=None):
111           commands = {
112               'challenge': self.respond_to_challenge,
113               'disconnect': lambda x, y: self.handle_close(),
```

```
            }

        if command in commands:
            commands[command](command, data)
        else:
            logging.critical("Unknown command received: %s" % (command,))
            self.handle_close()

    def process_unauthed_command(self, command, data=None):
        commands = {
            'challenge': self.respond_to_challenge,
            'auth': self.verify_auth,
            'disconnect': lambda x, y: self.handle_close(),
            }

        if command in commands:
            commands[command](command, data)
        else:
            logging.critical("Unknown unauthed command received: %s" \
                             % (command,))
            self.handle_close()

class Client(Protocol):
    def __init__(self):
        Protocol.__init__(self)
        self.mapfn = self.reducefn = self.collectfn = None

    def conn(self, server, port):
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect((server, port))
        asyncore.loop()

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def set_mapfn(self, command, mapfn):
        self.mapfn = types.FunctionType(marshal.loads(mapfn),
                                        globals(), 'mapfn')

    def set_collectfn(self, command, collectfn):
        self.collectfn = types.FunctionType(marshal.loads(collectfn),
                                            globals(), 'collectfn')

    def set_reducefn(self, command, reducefn):
        self.reducefn = types.FunctionType(marshal.loads(reducefn),
                                           globals(), 'reducefn')
```

```
163
164     def call_mapfn(self, command, data):
165         logging.info("Mapping %s" % str(data[0]))
166         results = {}
167         for k, v in self.mapfn(data[0], data[1]):
168             if k not in results:
169                 results[k] = []
170             results[k].append(v)
171         if self.collectfn:
172             for k in results:
173                 results[k] = [self.collectfn(k, results[k])]
174         self.send_command('mapdone', (data[0], results))
175
176     def call_reducefn(self, command, data):
177         logging.info("Reducing %s" % str(data[0]))
178         results = self.reducefn(data[0], data[1])
179         self.send_command('reducedone', (data[0], results))
180
181     def process_command(self, command, data=None):
182         commands = {
183             'mapfn': self.set_mapfn,
184             'collectfn': self.set_collectfn,
185             'reducefn': self.set_reducefn,
186             'map': self.call_mapfn,
187             'reduce': self.call_reducefn,
188             }
189
190         if command in commands:
191             commands[command](command, data)
192         else:
193             Protocol.process_command(self, command, data)
194
195     def post_auth_init(self):
196         if not self.auth:
197             self.send_challenge()
198
199
200 class Server(asyncore.dispatcher, object):
201     def __init__(self):
202         asyncore.dispatcher.__init__(self)
203         self.mapfn = None
204         self.reducefn = None
205         self.collectfn = None
206         self.datasource = None
207         self.password = None
208
209     def run_server(self, password="", port=DEFAULT_PORT):
210         self.password = password
211         self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        self.bind(("", port))
        self.listen(1)
        try:
            asyncore.loop()
        except:
            self.close_all()
            raise

        return self.taskmanager.results

    def handle_accept(self):
        conn, addr = self.accept()
        sc = ServerChannel(conn, self)
        sc.password = self.password

    def handle_close(self):
        self.close()

    def set_datasource(self, ds):
        self._datasource = ds
        self.taskmanager = TaskManager(self._datasource, self)

    def get_datasource(self):
        return self._datasource

    datasource = property(get_datasource, set_datasource)


class ServerChannel(Protocol):
    def __init__(self, conn, server):
        Protocol.__init__(self, conn)
        self.server = server

        self.start_auth()

    def handle_close(self):
        logging.info("Client disconnected")
        self.close()

    def start_auth(self):
        self.send_challenge()

    def start_new_task(self):
        command, data = self.server.taskmanager.next_task(self)
        if command == None:
            return
        self.send_command(command, data)

    def map_done(self, command, data):
```

```
        self.server.taskmanager.map_done(data)
        self.start_new_task()

    def reduce_done(self, command, data):
        self.server.taskmanager.reduce_done(data)
        self.start_new_task()

    def process_command(self, command, data=None):
        commands = {
            'mapdone': self.map_done,
            'reducedone': self.reduce_done,
            }

        if command in commands:
            commands[command](command, data)
        else:
            Protocol.process_command(self, command, data)

    def post_auth_init(self):
        if self.server.mapfn:
            self.send_command('mapfn',
                marshal.dumps(self.server.mapfn.func_code))
        if self.server.reducefn:
            self.send_command('reducefn',
                marshal.dumps(self.server.reducefn.func_code))
        if self.server.collectfn:
            self.send_command('collectfn',
                marshal.dumps(self.server.collectfn.func_code))
        self.start_new_task()

class TaskManager:
    START = 0
    MAPPING = 1
    REDUCING = 2
    FINISHED = 3

    def __init__(self, datasource, server):
        self.datasource = datasource
        self.server = server
        self.state = TaskManager.START

    def next_task(self, channel):
        if self.state == TaskManager.START:
            self.map_iter = iter(self.datasource)
            self.working_maps = {}
            self.map_results = {}
            #self.waiting_for_maps = []
            self.state = TaskManager.MAPPING
        if self.state == TaskManager.MAPPING:
```

```
            try:
                map_key = self.map_iter.next()
                map_item = map_key, self.datasource[map_key]
                self.working_maps[map_item[0]] = map_item[1]
                return ('map', map_item)
            except StopIteration:
                if len(self.working_maps) > 0:
                    key = random.choice(self.working_maps.keys())
                    return ('map', (key, self.working_maps[key]))
                self.state = TaskManager.REDUCING
                self.reduce_iter = self.map_results.iteritems()
                self.working_reduces = {}
                self.results = {}
        if self.state == TaskManager.REDUCING:
            try:
                reduce_item = self.reduce_iter.next()
                self.working_reduces[reduce_item[0]] = reduce_item[1]
                return ('reduce', reduce_item)
            except StopIteration:
                if len(self.working_reduces) > 0:
                    key = random.choice(self.working_reduces.keys())
                    return ('reduce', (key, self.working_reduces[key]))
                self.state = TaskManager.FINISHED
        if self.state == TaskManager.FINISHED:
            self.server.handle_close()
            return ('disconnect', None)

    def map_done(self, data):
        # Don't use the results if they've already been counted
        if not data[0] in self.working_maps:
            return

        for (key, values) in data[1].iteritems():
            if key not in self.map_results:
                self.map_results[key] = []
            self.map_results[key].extend(values)
        del self.working_maps[data[0]]

    def reduce_done(self, data):
        # Don't use the results if they've already been counted
        if not data[0] in self.working_reduces:
            return

        self.results[data[0]] = data[1]
        del self.working_reduces[data[0]]

def run_client():
    parser = optparse.OptionParser(
        usage="%prog [options]",
```

```
        version="%%prog %s"%VERSION)
    parser.add_option("-p", "--password", dest="password",
                      default="", help="password")
    parser.add_option("-P", "--port", dest="port",
                      type="int", default=DEFAULT_PORT, help="port")
    parser.add_option("-v", "--verbose",
                      dest="verbose", action="store_true")
    parser.add_option("-V", "--loud", dest="loud", action="store_true")

    (options, args) = parser.parse_args()

    if options.verbose:
        logging.basicConfig(level=logging.INFO)
    if options.loud:
        logging.basicConfig(level=logging.DEBUG)

    client = Client()
    client.password = options.password
    client.conn(args[0], options.port)

if __name__ == '__main__':
    run_client()
```

```
#!/usr/bin/env python
import mincemeat
from random import choice

strings = [''.join(choice('ATGC') for i in range(10)) for j in range(100)]
def mapfn(k1, string): yield ('ACCA' in string, 1)
def reducefn(k2, values): return len(values)

s = mincemeat.Server()
s.mapfn = mapfn
s.reducefn = reducefn
s.datasource = dict(enumerate(strings))
results = s.run_server(password='changeme')
print(results)
```

Notice that in mincemeat the datasource is not list of a dictionary of key:value where the values are the ones to be processes. The key is also passed to the `mapfn` function as first argument. Moreover the `mapfn` function can return more than one value using `yield`. This syntactical notation makes `minemeat` more flexible.

Execute this script on the server:

```
> python mincemeat_example.py
```

Run mincemeat.py as a worker on a client:

```
> python mincemeat.py -p changeme [server address]
```

You can run more than one worker, although for such example the server will terminate almost immediately.

9.9 *pyOpenCL*

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL devices), plus APIs that are used to define and then control the platforms. OpenCL provides parallel computing using task-based and data-based parallelism. It has been adopted by Intel, AMD, Nvidia, and ARM. OpenCL gives any application access to the graphics processing unit for non-graphical computing. Thus, OpenCL extends the power of the Graphics Processing Unit beyond graphics (general-purpose computing on graphics processing units). Academic researchers have investigated automatically compiling OpenCL programs into application-specific processors running on FPGAs, and commercial FPGA vendors are developing tools to translate OpenCL to run on their FPGA devices.

pyOpenCL gives you easy, Pythonic access to the OpenCL parallel computation API.

Object cleanup is tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. Completeness. PyOpenCL puts the full power of OpenCL's API at your disposal, if you wish.

pyOpenCL provides Automatic Error Checking. All errors are automatically translated into Python exceptions. Speed. PyOpenCL's base layer is written in C++ for efficiency reasons.

You can download `pyopencl` here:

`http://mathema.tician.de/software/pyopencl`

Here is an example:

```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf)

print(la.norm(a_plus_b - (a+b)))
```

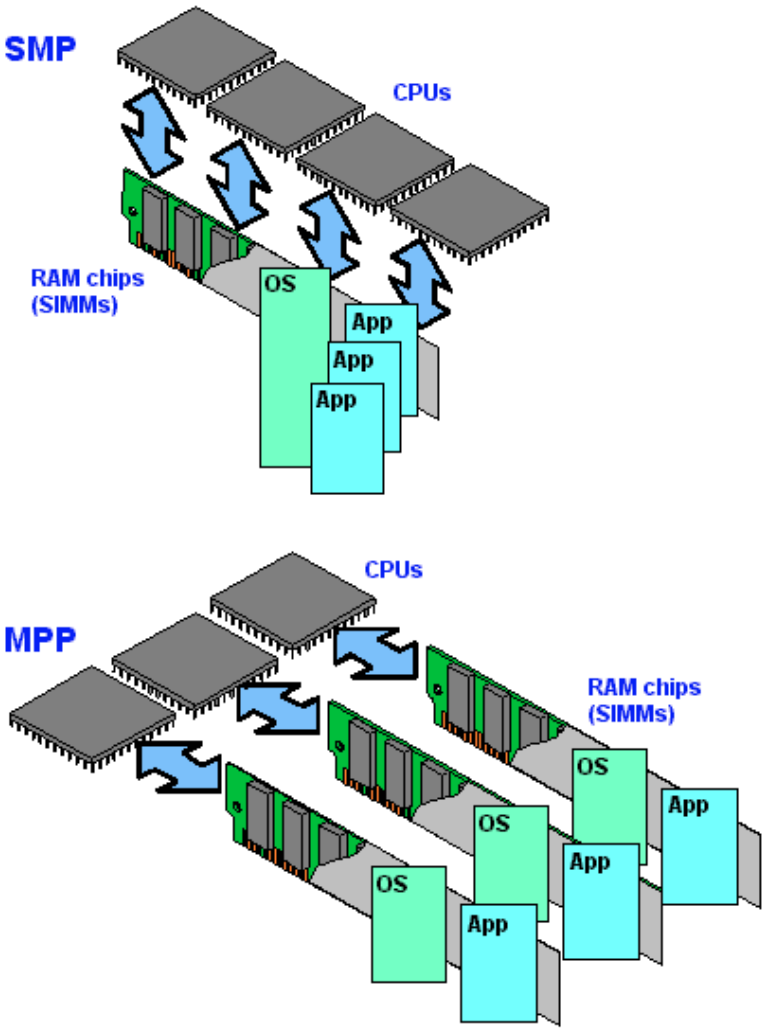

Figure 9.1: SMP (shared memory) vs MPP (distributed memory) architectures.

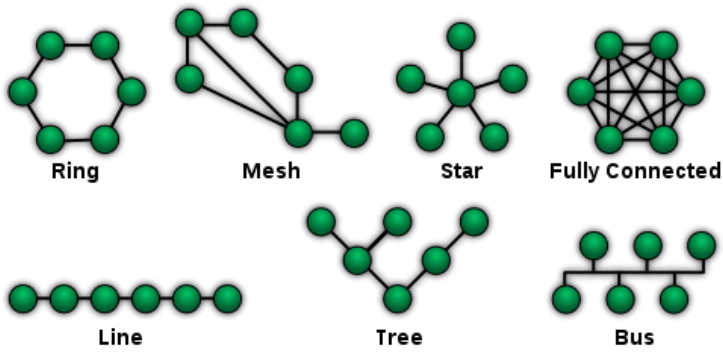

Figure 9.2: Examples of network topologies

10

Appendices

10.1 Appendix A: Math review

10.1.1 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

(10.1)

10.1.2 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)
$\mathbb{R}^*$	positive numbers including 0

(10.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**

$$\mathcal{A} \cap \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{10.3}$$

- **Union**

$$\mathcal{A} \cup \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{10.4}$$

- **Difference**

$$\mathcal{A} - \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{10.5}$$

Set laws

- Empty set laws

$$\mathcal{A} \cup \mathbf{0} = \mathcal{A} \tag{10.6}$$
$$\mathcal{A} \cap \mathbf{0} = \mathbf{0} \tag{10.7}$$

- Idempotency laws

$$\mathcal{A} \cup \mathcal{A} = \mathcal{A} \tag{10.8}$$
$$\mathcal{A} \cap \mathcal{A} = \mathcal{A} \tag{10.9}$$

- Commutative laws

$$\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A} \tag{10.10}$$
$$\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A} \tag{10.11}$$

- Associative laws

$$\mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \tag{10.12}$$
$$\mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C} \tag{10.13}$$

- Distributive laws

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}) \tag{10.14}$$
$$\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \tag{10.15}$$

- Absorption laws

$$\mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) = \mathcal{A} \tag{10.16}$$
$$\mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) = \mathcal{A} \tag{10.17}$$

- DeMorgan laws

$$\mathcal{A} - (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C}) \tag{10.18}$$
$$\mathcal{A} - (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C}) \tag{10.19}$$

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

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \tag{10.20}$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ if a subset of their Cartesian product.
- A binary relation is **transitive** is 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 $f(x) \neq f(x')$ (i.e. if not two different elements of $\mathcal{A}$ are mapped into the same 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:

- $f(n) \stackrel{def}{=} n\text{mod}2$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection nor a bijection.
- $f(n) \stackrel{def}{=} n\text{mod}2$ with domain $\mathbb{N}$ and codomain $\{0, 1\}$ is a surjection but not a bijection
- $f(x) \stackrel{def}{=} 2x$ 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)
- $f(x) \stackrel{def}{=} 2x$ with domain $\mathbb{R}$ and codomain $\mathbb{R}$ is not a surjection and is a bijection (in fact it is invertible)
-

10.1.3 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:

$$\log_a x = \frac{\log x}{\log a} \tag{10.21}$$
$$\log xy = (\log x) + (\log y) \tag{10.22}$$
$$\log \frac{x}{y} = (\log x) - (\log y) \tag{10.23}$$
$$\log x^n = n \log x \tag{10.24}$$

10.1.4 Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \stackrel{def}{=} f(0) + f(1) + ... + f(n-1) \tag{10.25}$$

Properties

- **Linearity I**

$$\sum_{i=0}^{i\leq n} f(i) = \sum_{i=0}^{i<n} f(i) + f(n) \tag{10.26}$$
$$\sum_{i=a}^{i\leq b} f(i) = \sum_{i=0}^{i\leq b} f(i) - \sum_{i=0}^{i<a} f(i) \tag{10.27}$$

- **Linearity II**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \tag{10.28}$$

Proof:

$$\begin{aligned}
\sum_{i=0}^{i<n} af(i) + bg(i) &= (af(0) + bg(0)) + ... + (af(n-1) + bg(n-1)) \\
&= af(0) + ... + af(n-1) + bg(0) + ... + bg(n-1) \\
&= a\left(f(0) + ... + f(n-1)\right) + b\left(g(0) + ... + g(n-1)\right) \\
&= a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \qquad (10.29)
\end{aligned}$$

Example 10.1.1.

$$\sum_{i=0}^{i<n} c = cn \text{ for any constant } c \qquad (10.30)$$

$$\sum_{i=0}^{i<n} i = \frac{1}{2}n(n-1) \qquad (10.31)$$

$$\sum_{i=0}^{i<n} i^2 = \frac{1}{6}n(n-1)(2n-1) \qquad (10.32)$$

$$\sum_{i=0}^{i<n} i^3 = \frac{1}{4}n^2(n-1)^2 \qquad (10.33)$$

$$\sum_{i=0}^{i<n} x^i = \frac{x^n - 1}{x - 1} \text{ (geometric sum)} \qquad (10.34)$$

$$\sum_{i=0}^{i<n} \frac{1}{i(i+1)} = 1 - \frac{1}{n} \text{ (telescopic sum)} \qquad (10.35)$$

10.1.5 Limits ($n \to \infty$)

In these section we will only deal with limits ($n \to \infty$) of positive functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = ? \qquad (10.36)$$

First compute limits of numerator and denominator separately

$$\lim_{n\to\infty} f(n) = a \tag{10.37}$$
$$\lim_{n\to\infty} g(n) = b \tag{10.38}$$

- If $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{a}{b} \tag{10.39}$$

- If $a \in \mathbb{R}$ and $b = \infty$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{10.40}$$

- If ($a \in \mathbb{R}^+$ and $b = 0$) or ($a = \infty$ and $b \in \mathbb{R}$)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{10.41}$$

- If ($a = 0$ and $b = 0$) or ($a = \infty$ and $b = \infty$) use de l'Hopital rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{10.42}$$

  and start again!

- Else ... the limit does not exist (typically oscillating functions or non-analytic functions).

For any $a \in \mathbb{R}$ or $a = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \Rightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1/a \tag{10.43}$$

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	$a^x \log a$
$x^n \log x, n > 0$	$x^{n-1}(n \log x + 1)$

(10.44)

Practical rules to compute derivatives

$$\frac{d}{dx}\left(f(x) + g(x)\right) = f'(x) + g'(x) \tag{10.45}$$

$$\frac{d}{dx}\left(f(x) - g(x)\right) = f'(x) - g'(x) \tag{10.46}$$

$$\frac{d}{dx}\left(f(x)g(x)\right) = f'(x)g(x) + f(x)g'(x) \tag{10.47}$$

$$\frac{d}{dx}\left(\frac{1}{f(x)}\right) = -\frac{f'(x)}{f(x)^2} \tag{10.48}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2} \tag{10.49}$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x) \tag{10.50}$$

Index

Bibliography

[1] `http://www.python.org`

[2] `http://www.sqlite.org/`

[3] `http://numpy.scipy.org/`

[4] `http://www.scipy.org/`

[5] `http://matplotlib.sourceforge.net/`

[6] `http://www.python.org/dev/peps/pep-0008/`

[7] `http://www.network-theory.co.uk/docs/pytut/`

[8] `http://oreilly.com/catalog/9780596158071`

[9] `http://www.python.org/doc/`

[10] M. Farach-Colton *et al.*, "Mathematical Support for Molecular Biology", DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (1999) Volume 47 ISBN-10: 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] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (1970)

[13] H. M. Markowitz "Portfolio Selection". The Journal of Finance 7 (1952)

[14] http://remembersaurus.com/mincemeatpy/

[15] Andrew Lo and Jasmina Hasanhodzic, The Evolution of Technical Analysis: Financial Prediction from Babylonian Tablets to Bloomberg Terminals. Bloomberg Press (2010). ISBN 1576603490