CSC521 Class Notes - Draft

Massimo Di Pierro

Abstract

Attention: these notes may contain errors. They will be replaced on a weekly basis as errors are corrected and new material is added. Version 0.3

Contents

Chapter 1

Notation

1.1 Java/C++ vs Python:

	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
function	void F(int 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*

(1.1)

Chapter 2

Introduction

Monte Carlo is the use of random numbers in computation. Let's consider two examples:

2.1 Computation of 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

$$\pi = 4\arctan 1 \tag{2.1}$$

The arctan has the following Taylor series expansion[1]:

[1]Taylor expansion:

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + ... + \frac{1}{i!}f^{(i)}(0)x^2 + ... \tag{2.2}$$

and if $f(x) = \arctan x$ then:

$$\begin{aligned}
f'(x) &= \frac{d\arctan x}{dx} = \frac{1}{1+x^2} \rightarrow f'(0) = 1 \\
f''(x) &= \frac{d^2\arctan x}{d^2x} = \frac{d}{dx}\frac{1}{1+x^2} = -\frac{2x}{(1+x^2)^2} \\
&\;\;... \\
f^{(2i+1)}(x) &= (-1)^i\frac{(2i)!}{(1+x^2)^{2i+1}} + x... \rightarrow f^{(2i+1)}(0) = (-1)(2i)! \\
f^{(2i)}(x) &= x... \rightarrow f^{(2i)}(0) = 0
\end{aligned}$$

$$\arctan x = \sum_{i=0} (-1)^i \frac{x^{2i+1}}{2i+1} \tag{2.3}$$

and one can approximate π to arbitrary precision by computing the sum

$$\pi = \sum_{i=0} (-1)^i \frac{4}{2i+1} \tag{2.4}$$

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)
```

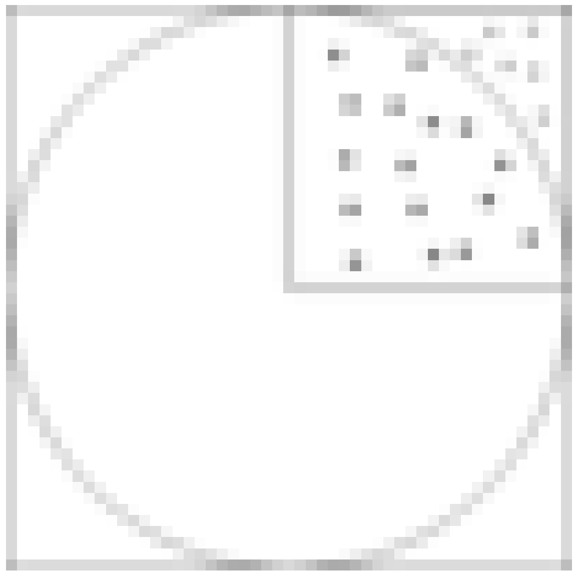
which produces the following output:

```
0 4.0
1 2.66666666667
2 3.46666666667
3 2.89523809524
4 3.33968253968
5 2.97604617605
6 3.28373848374
7 3.01707181707
8 3.25236593472
9 3.04183961893
...
98 3.14259365434
999 3.14059265384
```

The above formula works but converges very slowly to the exact value of

$$\pi = 3.14159265358979323846264 33... \tag{2.5}$$

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 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 2.66666666667
3 2.0
4 1.6
5 2.0
6 2.28571428571
7 2.5
8 2.66666666667
9 2.4
...
998 3.14102564103
999 3.14376996805
```

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

$$\pi = 4\int_0^1 \sqrt{1-x^2}dx \tag{2.6}$$

The problem of computing numerical integrals can often be solved using Monte Carlo.

2.2 Simulation of 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:

```
def daily_income(N,days=1000):
    income=0
    for i in range(days):
        instock=N
        for j in range(int(gauss(976,352))):
            if instock>0:
                if random()<0.05:
                    instock=instock-1
                    income=income+70
            else:
                if random()<0.02:
                    income=income+100
        income=income-30*instock
    return income/days
```

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,daily_income(n)
```

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%.

The goals of this class are:

- How to go from a formulation of a problem to the corresponding Monte Carlo algorithm
- How do we estimate the statistical error in out prediction so that we can adjust our simulation parameters accordingly.

2.3 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.

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

$$ {}^{14}_{6}C \longrightarrow {}^{14}_{7}N + e^{-} + \overline{\nu}_e \tag{2.7}$$

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 and because of the complexity of typical macroscopic systems chaos emerges from the deterministic system.

Chapter 3

Review of Probability and Statistics

3.1 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
$$P(A) \overset{def}{=} \lim_{N\to\infty} \frac{N_A}{N} \tag{3.1}$$
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

$$P(A) \overset{def}{=} \frac{c(A)}{c(S)} \tag{3.2}$$

  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 die 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 $P(A) = c(A)/c(S) = 3/6 = 1/2$. This analysis works for an ideal die and ignores that fact that a real die 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 $P(A) \in [0, 1]$ that satisfies the following conditions: $P(S) = 1$; $P(A_1 \cup A_2) = P(A_1) + P(A_2)$ 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 die a perfect die 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 die 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 1 *If* $S = A \cup A^c$ *and* $A \cap A^c = 0$ *then* $P(A) = 1 - P(A^c)$

3.1.1 Conditional probability and independence

We define $P(A|B)$ as the probability of event A given event B and we write

$$P(A|B) \overset{def}{=} \frac{P(AB)}{P(B)} \tag{3.3}$$

where $P(AB)$ is the probability that A and B both occur and $P(B)$ is the probability that B occurs. Note that if $P(A|B) = P(A)$ is independent of B then we say that A and B are uncorrelated and from eq.(3.3) we conclude $P(AB) = P(A)P(B)$ therefore the probability that two uncorrelated events occur is the product of the probability that each individual event occurs.

3.1.2 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 $X = x$

$$p(x) = P(X = x) \tag{3.4}$$

We also define the **expectation value** of any function of a discrete random variable $f(X)$ as

$$E[f(X)] = \sum_i f(x_i)p(x_i) \tag{3.5}$$

where i loops all possible variables x_i of the random variable X.

Example 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,*

$$E[X] = \sum_i x_i p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} x_i \frac{1}{6} = 3.5 \tag{3.6}$$

and

$$E[(X-3.5)^2] = \sum_i (x_i-3.5)^2 p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} (x_i-3.5)^2 \frac{1}{6} = 2.9167 \tag{3.7}$$

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

$$\sigma_X = E[X^2] - E[X]^2 \tag{3.8}$$

3.1.3 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 $X \leq x$

$$F(x) \overset{def}{=} P(X \leq x) \tag{3.9}$$

If S is a continuous set and X is a continuous random variable then we define a **probability density or distribution** $f(x)$ as

$$p(x) \overset{def}{=} \frac{dF(x)}{dx} \tag{3.10}$$

and the probability that X falls into an interval $[a, b]$ can be computed as

$$P(a \leq X \leq b) = \int_a^b p(x)dx \tag{3.11}$$

We also define the **expectation value** of any function of a random variable $f(X)$ as

$$E[f(X)] = \int_{-\infty}^{\infty} f(x)p(x)dx \tag{3.12}$$

Example 2 *X is a uniform random variable (probability density $p(x)$ equal to 1 if $x \in [0, 1]$, equal to 0 otherwise)*

$$E[X] = \int_{-\infty}^{\infty} xp(x)dx = \int_0^1 xdx = \frac{1}{2} \tag{3.13}$$

and

$$E[(X - \frac{1}{2})^2] = \int_{-\infty}^{\infty} (x - \frac{1}{2})^2 p(x)dx = \int_0^1 (x^2 - x + \frac{1}{4})dx = \frac{1}{12} \tag{3.14}$$

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

$$\sigma_X^2 = E[X^2] - E[X]^2 \tag{3.15}$$

Theorem 2 $\int_{-\infty}^{\infty} p(x)dx = 1$.

Proof: By definition

$$F(\infty) \overset{def}{=} P(X \le \infty) = 1 \tag{3.16}$$

therefore

$$P(-\infty \le X \le \infty) = \int_{-\infty}^{\infty} p(x)dx = 1 \tag{3.17}$$

Theorem 3 *$E[X]$ is a linear operator.*

Proof:

$$\begin{aligned} E[aX+b] &= \int_{-\infty}^{\infty} (ax+b)p(x)dx \\ &= a\int_{-\infty}^{\infty} xp(x)dx + b\int_{-\infty}^{\infty} p(x)dx \\ &= aE[X]+b \end{aligned}$$

3.1.4 Multiple random variables, covariance and correlations

Given two random variables, X and Y, we define the covariance between them as

$$\sigma_{XY}^2 = E[(X-\mu_X)(Y-\mu_Y)] = E[XY] - E[X]E[Y] \tag{3.18}$$

Theorem 4 $\sigma_{X+Y}^2 = \sigma_X^2 + \sigma_Y^2 + 2\sigma_{XY}^2$

Theorem 5 *If X and Y are independent then $\sigma_{XY}^2 = 0$.*

Proof:

$$\begin{aligned} E[XY] &= \int\int xyp(x,y)dxdy \\ &= \int\int xyp(x)p(y)dxdy \\ &= \left[\int xp(x)dx\right]\left[\int yp(y)dy\right] \\ &= E[X]E[Y] \end{aligned}$$

therefore

$$\sigma_{XY}^2 = E[XY] - E[X]E[Y] = 0 \tag{3.19}$$

Theorem 6 *If X and Y are completely correlated or anticorrelated ($Y = \pm X$) then $\sigma_{XY}^2 = \pm\sigma_X^2$*

Proof:

$$\begin{aligned}
\sigma_{XY}^2 &= E[(X-\mu_X)(Y-\mu_Y)] \\
&= E[(X-\mu_X)(\pm X \mp \mu_X)] \\
&= \pm E[(X-\mu_X)(X-\mu_X)] \\
&= \pm\sigma_X^2
\end{aligned}$$

In general

$$c(X,Y) = \frac{\sigma_{XY}^2}{\sqrt{\sigma_X^2\sigma_Y^2}} \tag{3.20}$$

lies in the range $[-1,1]$ and, because of the two theorems above $c(X,Y)=0$ implies X and Y are uncorrelated; $=+1$ implies X and Y are perfectly correlated; $=-1$ implies X and Y are perfectly anti-correlated.

Theorem 7 *For uncorrelated random variables X_i*

$$\begin{aligned}
E[\sum_i a_i X_i] &= \sum_i a_i E[X_i] \\
E[(\sum_i X_i)^2] &= \sum_i E[X_i^2]
\end{aligned}$$

3.1.5 Measuring correlations from experiments

Let's assume that we have the ability to repeat an experiment (real or simulated) N times where N is large, and each time we obtain a different result X_i. X_i are independent random variables with some unkown probability mass function $p(x_i)$ where p only depends on the experiment and it is therefore independent on i. Each time we perform the experiment we perform a measureament $f(X_i)$ and we want to determine the expectation value $E[f(X)]$. If $p(X)$ is not known a priori we have to assume that each experiment has equal probability therefore $p(X) = \frac{1}{N}\sum_{i=0}^{i<N}\delta(X-X_i)$

From the definition of expectation value we obtain in this case

$$E[f(X)] = \int_{-\infty}^{+\infty} f(x)p(x)dx = \frac{1}{N}\sum_{i=0}^{i<N} f(X_i) \tag{3.21}$$

This is not a different definition of expectation value of f than the ones above. It is a consequence of those definitions in the case $p(x)$ is an unknown, but we have samples X_i drawn from a probability distribution $p(x)$.

Example 3 *One performs an experiment $N = 10$ times and obtains the following results:*

```
i   X_i
0   4.660
1   −0.017
2   4.406
3   1.621
4   0.076
5   1.672
6   6.766
7   1.401
8   4.955
9   0.581
```

The mean and variance of these numbers are given by

$$\begin{aligned}\mu &= E[X_i] = \frac{1}{N}\sum_{i=0}^{i<N} X_i = 2.612 \\ \sigma^2 &= E[(X_i-\mu)^2] = \frac{1}{N}\sum_{i=0}^{i<N}(X_i-\mu)^2 = 5.097\end{aligned}$$

3.1.6 Weak Law of large numbers

Theorem 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 $\varepsilon > 0$*

$$\lim_{n\to\infty} P\left(\left|\frac{X_1 + X_2 + ... + X_n}{n} - \mu\right| > \varepsilon\right) = 0 \tag{3.22}$$

3.1.7 Strong Law of large numbers

Theorem 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*

$$\lim_{n\to\infty} \frac{X_1 + X_2 + ... + X_n}{n} = \mu \tag{3.23}$$

(proof in attached paper).

3.2 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.

3.2.1 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

$$n!/(n-m)! = n(n-1)(n-2)...(n-m+1) \tag{3.24}$$

  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

$$m!/(m-n)! = m(m-1)(m-2)...(m-n+1) \tag{3.25}$$

  combinations. Note that if $m = n$ than case 1 and case 2 agree as expected.

3.2.2 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

$$\binom{m}{n} = \frac{m!}{(m-n)!n!} \tag{3.26}$$

  ways to place n equivalent plugs into m different sockets. Note that if $m = n$

$$\binom{n}{n} = \frac{n!}{(n-n)!n!} = 1 \tag{3.27}$$

  in agreement with case 1.

3.2.3 Other examples

- 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 $20!/(20-4)! \simeq 1.2 \times 10^5$

- etc. etc.

3.2.4 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?

$$P(3red) = \frac{24}{48} \times \frac{23}{47} \times \frac{22}{46} = \frac{11}{94} \tag{3.28}$$

- What is the probability that they are all black?

$$P(3black) = P(3red) = \frac{11}{94} \tag{3.29}$$

- What is the probability that they are not all black nor all red?

$$\begin{aligned}
P(mixture) &= 1 - P(3red \cup 3black) \\
&= 1 - P(3red) - P(3black) \\
&= 1 - 2\frac{11}{94} \\
&= \frac{36}{47}
\end{aligned}$$

3.2.5 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 $P(3red) = 11/94$ therefore the probability that the third one is red is 1/8.
- **Wrong answer**: The probability that they are a mixture is $P(mixture) = 36/47$ therefore the probability that the third one is red is $1 - 36/47 = 11/47$
- **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

$$P(red) = \frac{22}{22+24} = \frac{11}{23} \tag{3.30}$$

3.3 Probability distributions for discrete random variables

3.3.1 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.

3.3.2 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 $q = 1 - p$.

$$p(k) = \left\{ \begin{array}{ll} p^k(1-p)^{1-k} & \text{if } k \in \{0,1\} \\ 0 & \text{otherwise} \end{array} \right\} \tag{3.31}$$

A Bernoulli random variable has expected value of p, and variance of pq.

3.3.3 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 $P(X = 30)$.

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

$$p(k) = P(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{3.32}$$

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

$$p(k) = P(X = k) = \binom{n_X + n_Y}{k} p^k (1-p)^{n-k} \tag{3.33}$$

3.3.4 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.

$$p(n) = P(X = n) = \binom{n-1}{k-1} p^k (1-p)^{n-k} \tag{3.34}$$

Example 4 *(after a problem by 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?
$$p(n) = \binom{n-1}{4} 0.4^5 0.6^{n-5} \tag{3.35}$$

- What's the probability that he finishes on the tenth house?

$$p(10) = \binom{9}{4} 0.4^5 0.6^5 = 0.10 \tag{3.36}$$

- 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:

$$\sum_{i=5,6,7,8} p(i) = 0.1737 \tag{3.37}$$

- What's the probability that he exhausts all houses in the neighborhood, gives up, and then goes to live on the streets?

$$1 - \sum_{i=5,..,30} p(i) = 0.0015 \tag{3.38}$$

3.3.5 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:

$$p(k) = P(X = k) = e^{-\lambda} \frac{\lambda^k}{k!} \tag{3.39}$$

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

$$\frac{n!}{(n-k)!k!}\left(\frac{\lambda}{n}\right)^k\left(1-\frac{\lambda}{n}\right)^{n-k} \simeq e^{-\lambda}\frac{\lambda^k}{k!} + O(\frac{1}{n}) \tag{3.40}$$

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

$$\lim_{n\to\infty} pn \tag{3.41}$$

is finite. This limit is λ.

3.3.6 Boltzmann distribution

The Maxwell-Boltzmann distribution is an important relationship that finds many applications in physics and chemistry. It forms the basis of the kinetic theory of gases, which accurately explains many fundamental gas properties, including pressure and diffusion. The Maxwell-Boltzmann distribution also finds important applications in electron transport and other phenomena.

The Maxwell-Boltzmann probability mass function can be expressed as the probability of finding a system in a state x where

$$p(x) = P(X = x) = \frac{e^{-\frac{E(x)}{kT}}}{\sum_y e^{-\frac{E(y)}{kT}}} \tag{3.42}$$

where $E(x)$ is the energy of the system when in state x, k is the Boltzmann constant and T is the temperature of the system.

3.4 Probability distributions for continuous random variables

3.4.1 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.

3.4.2 Exponential distribution

Probability mass function is given by:

$$p(x) = \lambda e^{-\lambda x} \tag{3.43}$$

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

$$P(X > s + t | X > t) = P(X > s) \tag{3.44}$$

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.

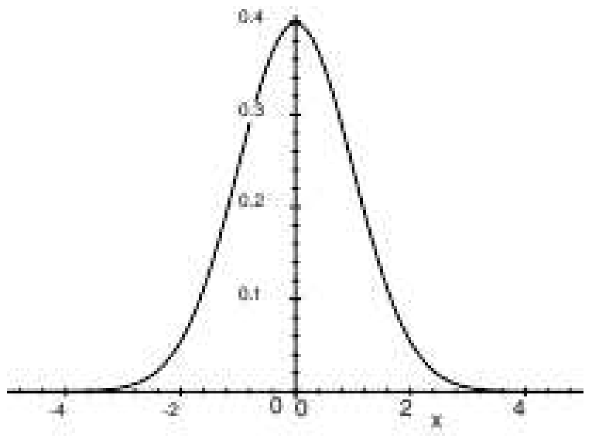
3.4.3 Normal (Gaussian) distribution

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:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.45}$$

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.



3.4.4 Error in the mean

Let's consider the case of N repeated experiments with outcomes X_i. Let's also assume that each X_i is supposed to be equal to an unknown value μ but in practice $X_i = \mu + \varepsilon$ where ε is a random variable with gaussian distribution centered at zero. One could estimate μ by $\mu = E[X] = \frac{1}{N}\Sigma_i X_i$. What would be the error on this mean? The answer is

$$\delta\mu = \sqrt{\frac{\sigma^2}{N-1}} \tag{3.46}$$

where $\sigma^2 = E[(X-\mu)^2] = \frac{1}{N}\Sigma_i (X_i - \mu)^2$.

Chapter 4

Random Numbers

4.1 Real Randomness

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 billions of billions of atoms and we record the time when we observe an atom decay:

$$t_0, t_1, t_2, t_3, t_4, t_5, ... \tag{4.1}$$

One could study the probability distribution of these t and would find that they follow an exponential probability distribution like

$$P(t) = \lambda e^{-\lambda t} \tag{4.2}$$

where $t_0 = 1/\lambda$ is the decay time characteristic of the particular type of atom. In order to map the numbers t_i into numbers x_i with a uniform distribution we proceed as follows:

- map the sequence $\{t_i\}$ into a sequence of 0s and 1s with the Bernoulli distribution (the probability of 0 is 50% and the probability of 1 is 50% and the probability of each element in the sequence does not depend on the probability of the preceding element in the sequence).
- We break the sequence of 0s and 1s into blocks of binary digit and we map each block into the [0,1) interval.

Once a sequence of uniform random numbers is built one can map it to any other distribution.

4.1.1 Memoryless to Bernoulli distribution

Given the sequence $\{t_i\}$ with exponential distribution we can build a Bernoulli distribution by applying the following formula

$$x_i = \begin{cases} 1 & \text{if } t_{2i+1} > t_{2i} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

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).

4.1.2 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

$$0101111011010101010111110101 \tag{4.4}$$

and we perform the following steps:

- break the sequence into chunks of 5 bits

$$01011\text{-}11011\text{-}01010\text{-}10101\text{-}11101\text{-}..... \tag{4.5}$$

- map each chunk $a_1a_2a_3a_4a_5$ into $x = \sum_{k=1}^{5} a_k 2^{-k}$

$$0.25195\text{-}0.75195\text{-}0.25000\text{-}0.25195\text{-}0.87500\text{-}... \tag{4.6}$$

A uniform random number generator is usually the first step towards building any other random number generator.

Other physical processes can be used for this purpose. For example Toshiba produces a device, called Random Master, that can be plugged into a PCI board and generate uniform random numbers from the thermal fluctuations in a semiconductor.

4.2 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—they only approximate some of the properties of random numbers. 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.

4.2.1 Multiplicative congruential generator

This is perhaps the simplest random number generator:

$$\begin{aligned} x_i &= a x_{i-1} \mathrm{mod} m \\ y_i &= x_i / m \end{aligned}$$

For example $a = 65539$ and $m = 2^{31}$ is a typical choice and the random numbers y_i are uniformly distributed in $(0, 1)$. This generator is called RANDU and it is the generator used in Visual Basic and the C rand() function. The RANDU generator is particularly fast because the modulus is implemented using the finite precision (32bits) of modern processors (IEEE-754)

Here is a possible implementation:

```
from time import *

def random_mcg(x,a,m):
    return a*x % m

def test_random_mcg(a=66539,m=2**31):
    x=int(time())
    print 'seed=',x
    for i in range(10):
    x=random_mcg(x,a,m)
    print float(x)/m

>>> test_random_mcg()
```

which produces the following output:

```
seed= 1071914055
0.718363384251
0.817840396892
0.441771923099
0.290067966562
0.764460491482
0.976151249837
0.976763075683
0.0752172055654
0.660435552243
0.28565846337
```

These 10 numbers "look" random. Note 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 example:

```
>>> test_mcg(7**5,5)
```

produces the following output:

```
seed= 1071913474
0.6
0.2
0.4
0.8
0.6
0.2
0.4
0.8
0.6
0.2
```

Note that the generator is producing only four numbers (0.6, 0.2, 0.4 and 0.8) and the sequence is periodic. This is a characteristic of all PRNGs. The generated sequence is periodic! The period depends on a and m.

If the period is larger then the number of random numbers needed in computation the pseudo-random generator is probably good enough. We will later explore the relation between a, m and the period.

Theorem 10 *For a multiplicative congruential generator if m is a power of 2 and $a = \pm 3 mod 8$ than the period of the PRNG is $m/4$.*

4.2.2 Modular arithmetic and period of PRNGs

We say that

$$a = b \text{mod} m \tag{4.7}$$

if and only if $a - b = nm$ where n is an integer.

The modulus function has the following properties:

- $a = b \bmod m \Rightarrow b = a \bmod m$ (symmetric)
- $a = a \bmod m$ for any a (reflexive)
- $a = b \bmod m$ and $b = c \bmod m \Rightarrow a = c \bmod m$ (transitive)

Therefore mod operation implies an equivalence relation. As en example we can say that the set

$$2, 7, 12, 17, 22, ..., 2 + 5i, ... \tag{4.8}$$

forms an equivalence class under mod5 meaning that for each couple a and b in the sequence

$$a = b \bmod 5 \tag{4.9}$$

The modulus function has two more properties that will come handy later:

$$\begin{aligned}(a+b)\bmod m &= ((a \bmod m) + (b \bmod m)) \bmod m \\ (ab)\bmod m &= ((a \bmod m)(b \bmod m)) \bmod m\end{aligned}$$

If p is a prime number and n is an integer we define the **totient function** as

$$\phi(p^n) = p^{n-1}(p-1) \tag{4.10}$$

For any other number a,

$$\phi(a) = \phi(p_0^{n_0} p_1^{n_1} p_2^{n_2} ...) = \phi(p_0^{n_0})\phi(p_1^{n_1})\phi(p_2^{n_2})... \tag{4.11}$$

where p_i are the prime factors of p and n_i their corresponding powers.

For example:

$$\begin{aligned}\phi(14175) &= \phi(3^4 5^2 7^1) = \phi(3^4)\phi(5^2)\phi(7^1) \\ &= \left[3^3(3-1)\right]\left[5^1(5-1)\right]\left[7^0(7-1)\right] = 6480\end{aligned}$$

Here is a possible coding (although not very efficient) of the totient function:

```
def totient(a):
    t=1
    for p in [2]+range(3,int(sqrt(a)+1,2):
        n=0
        while a%p==0:
            a=a/p
            n=n+1
        if n>0:
            print p,n
            t=t*(p**(n-1))*(p-1)
    if a!=1:
        p,n=a,1
        t=t*(p**(n-1))*(p-1)
    return t
```

The totient function is important to set a bound on the period of a PRNG. In particular the Euler-Fermat theorem states that given a PRNG

$$x_i = ax_{i-1}\text{mod}m \tag{4.12}$$

if a and m are relatively prime (have no common factor) the period cannot be greater than $\phi(m)$. A choice for a so that the period is exactly $\phi(m)$ is called a **primitive root** of m. For a given m there are $\phi(m-1)$ primitive roots. It turns out that m has primitive roots only if $m = 2^{n_0}p^{n_1}$ where n_0 is 0 or 1 and $n_1 > 0$. This property is important to set up limits on the period of a random number generator.

Let's go back to the RANDU generator. It has $m = 2^{31}$ therefore m is not of the form $2^{n_0}p^{n_1}$ and therefore m has no primitive roots and the period of the generator is certainly less than $\phi(2^{31}) = 1073741824$ and this number is already small considering that typical Monte Carlo computations may require more than 1 billion random numbers.

A better choice is using $m = 2^{31} - 1$ known as the Marsenne prime number. In fact $\phi(m) = 2147483646$ and $a = 7^5$ is a typical multiplier that is known to be a primitive root of the Marsenne prime. From now on this will be our preferred generator:

$$x_i = (7^5 x_{i-1})\text{mod}(2^{31} - 1) \tag{4.13}$$

Another typical choice is $m = 2^{61}-1$ that is also a Marsenne prime whose period is for a non primitive root multiplier is $\phi(m) = 2305843009213693951$. Better!

4.2.3 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

$$x_i = f(x_{i-1}, x_{i-2}, ..., x_{i-k}) \tag{4.14}$$

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.

4.2.4 Linear congruential generator

A modification of the multiplicative congruential generator:

$$x_i = (ax_{i-1} + c)\mathrm{mod}m \tag{4.15}$$

4.2.5 Multiplicative recursive generator

Another modification of the multiplicative congruential generator:

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + ... + a_k x_{i-k})\mathrm{mod}m \tag{4.16}$$

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

$$(2^{31} - 1)^5 - 1 \simeq 4.56 \times 10^{46} \tag{4.17}$$

4.2.6 Lagged Fibonacci generator

$$x_i = (x_{i-j} + x_{i-k})\mathrm{mod}m \tag{4.18}$$

This is similar to the multiplicative recursive generator above. If m is prime and $j \neq k$ the period can be as large as $m^k - 1$.

4.2.7 Marsaglia's add-with-carry generator

$$x_i = (x_{i-j} + x_{i-k} + c_i)\mathrm{mod}m \tag{4.19}$$

where $c_1 = 0$ and $c_i = 1$ if $(x_{i-1-j} + x_{i-1-k} + c_{i-1}) < m$, 0

4.2.8 Marsaglia's subtract-and-borrow generator

$$x_i = (x_{i-j} - x_{i-k} - c_i)\text{mod}m \tag{4.20}$$

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.

4.2.9 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

$$v_i = (x_i, x_{i+1}, ..., x_{i+k}) \tag{4.21}$$

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.

4.2.10 Knuth's polynomial congruential generator

$$x_i = (ax_{i-1}^2 + bx_{i-1} + c)\text{mod}m \tag{4.22}$$

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:

$$x_i = x_{i-1}^2\text{mod}m \tag{4.23}$$

4.2.11 Inverse congruential generator

$$x_i = (ax_{i-1}^{-1} + c)\text{mod}m \tag{4.24}$$

where x_{i-1}^{-1} is the multiplicative inverse of x_{i-1} modulo m, i.e. $x_{i-1}x_{i-1}^{-1} = 1\text{mod}m$.

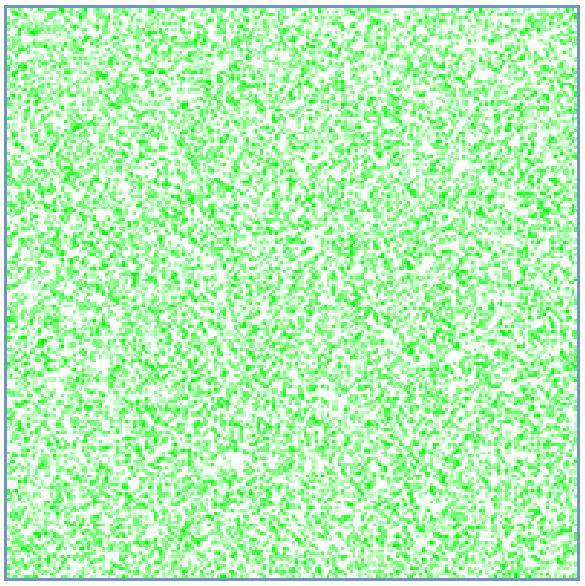
4.2.12 Defects of PRNGs

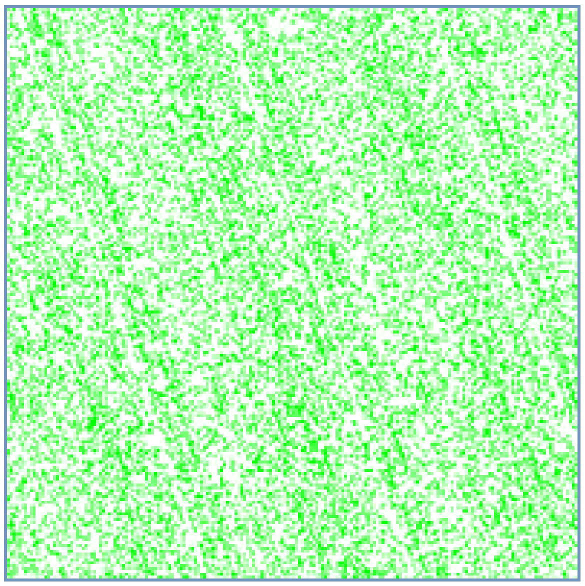
The non-randomness of pseudo-random number generators manifest itself in at least two different ways:

- 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 "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 (the rand() function of C and Visual Basic)





The second plot clearly shows stripes and clusters of points that do not appear on the left. The numbers generated by RANDU are not random and it shows. Do not use RANDU!

4.2.13 Combining generators

In order to increase "randomness" it may be a good idea to combine random number generators. For example

$$\begin{aligned}
x_i^0 &= a^0 x_{i-1}^0 \mathrm{mod} m^0 \text{ where } a^0 = 170, m^0 = 30323 \\
x_i^1 &= a^1 x_{i-1}^1 \mathrm{mod} m^1 \text{ where } a^1 = 171, m^1 = 30269 \\
x_i^2 &= a^2 x_{i-1}^2 \mathrm{mod} m^2 \text{ where } a^2 = 172, m^2 = 30307
\end{aligned}$$

can be combined to produce

$$y_i = \left[\frac{x_i^0}{m^0} + \frac{x_i^1}{m^1} + \frac{x_i^2}{m^2}\right] \mathrm{mod} 1 \tag{4.25}$$

which is known as the Wichman and Hill generator.

The idea of combining generators can be used to increase the period of the generator but its does not necessarily reduce the emergence of patterns in the numbers and actually different generators may conspire and introduce undesirable effects.

One obvious thing to lookup is the following. Consider these two combined generators:

$$A_i = \left[0.2\frac{x_i^0}{m^0} + 0.3\frac{x_i^1}{m^1} + 0.4\frac{x_i^2}{m^2}\right] \mathrm{mod} 1 \tag{4.26}$$

and

$$B_i = \frac{\left[\frac{x_i^0}{m^0} + \frac{x_i^1}{m^1} + \frac{x_i^2}{m^2}\right] \mathrm{mod} 2}{2} \tag{4.27}$$

The first of them combines three numbers uniformly distributed in $(0, 0.2), (0, 0.3)$ and $(0, 0.4)$ therefore it cannot generate numbers larger than 0.9. The random sequence A_i is uniform in $(0, 0.9)$. The second generator has a more subtle problem. It combines three numbers uniformly distributed in $(0, 1)$ therefore, before the mod, it produces random numbers uniformly distributed in $(0, 3)$. Therefore after taking the mod2 and the division by 2 numbers $B_i \in (0, 0.5)$ are more likely to appear that numbers in $B_i \in (0.5, 1)$ because for each $B \in (0, 0.5)$

$$\frac{\overline{x} \mathrm{mod} 2}{2} = B \Rightarrow \overline{x} = B \text{ or } \overline{x} = 2 + B \tag{4.28}$$

(there are two ways to realize it) while for each $B \in (0.5, 1)$

$$\frac{\overline{x}\text{mod}2}{2} = B \Rightarrow \overline{x} = B \tag{4.29}$$

(there is one way to realize it).

4.3 Implementation issues

4.3.1 Arbitrary precision modular arithmetic

If we try to compute $ab\text{mod}m$ when m is less but not equal to machine precision and ab is larger than machine precision we have a problem and we need arbitrary precision arithmetic. In our examples we use the Python language which supports arbitrary precision arithmetic. Languages such as Java and C++ do not support arbitrary precision arithmetic and additional libraries are required to perform arithmetic operations with large integers. Here is how they work.

In order to compute $ab\text{mod}m$ we rewrite a and b into a different base n and write

$$a = \sum_i a_i n^i,\ b = \sum_j b_j n^j \tag{4.30}$$

therefore

$$c = \sum_k c_k n^k = ab = \sum_k \sum_{i,j} a_i b_j n^k\ \delta_{i+j,k} \tag{4.31}$$

and by using properties of the modulus function

$$ab\text{mod}m = \left[\sum_k \sum_{i,j} (a_i b_j \text{mod}m)(n^k \text{mod}m)\ \delta_{i+j,k}\right] \text{mod}m \tag{4.32}$$

If we choose a base n less but close than half machine precision, `sizeof(int)` for unsigned integers, then for any i, j the product $a_i b_j$ fits into a an integer and the computation is well defined. Note that the term $d_k = (n^k \text{mod}m)$ can be computed recursively and stored:

$$\begin{aligned} d_0 &= 1 \\ d_i &= n d_{i-1} \text{mod}m \end{aligned}$$

Here is a possible implementation of arbitrary precision mod

```
def libreak(a,bits):
    size=2**bits
    chunks=[]
    while a!=0:
        chunks.append(a % size)
        a=a/size
    return chunks

def mulmod(a,b,m,bits=8):
    ac=libreak(a,bits)
    bc=libreak(b,bits)
    d=[1]
    for k in range(1,len(ac)+len(bc)):
        d.append(d[k-1]*(2**bits) % m)
    c=0
    for i in range(len(ac)):
        for j in range(len(bc)):
            k=i+j
            c=c+(ac[i]*bc[j] % m)*d[k]
            c=c % m
    return c

>>> print mulmod(2**32-2,2**32-3, 2**32-1)
```

which computes

$$(2^{32}-2)(2^{32}-3)\mathrm{mod}(2^{32}-1)=2 \tag{4.33}$$

Note that Python supports natively arbitrary precision integer arithmetic therefore the above functions are not necessary. They are necessary in order to code the PRNGs in other languages.

Analogously it is difficult to compute

$$c=a^n \mathrm{mod} m \tag{4.34}$$

because a^n may not fit in machine precision. This can be done recursively:

- if n is even, $c=(aa\mathrm{mod}m)^{n/2}\mathrm{mod}m$
- if n is odd, $c=((a^{n-1}\mathrm{mod}m)a)\mathrm{mod}m$

that we can implement recursively as follows:

```
def powmod(a,n,m,bits=8):
    if n==0:
        return 1
    elif n==1:
        return a
    if n%2==0:
        return powmod(mulmod(a,a,m,bits),n/2,m,bits)
    else:
        return mulmod(powmod(a,n-1,m,bits),a,m,bits)

>>> print powmod(9,124,78)
```

This computes

$$9^{124}\text{mod}78 = 9 \tag{4.35}$$

4.3.2 Encapsulation

It is convenient to implement each random number generator as a class rather than as a function. We require that the class implements a method `float uniform()` that returns a uniform floating point random number in the range $(0,1)$. In this way we will be able to write applications that are independent of the choice of random number generator.

Here is a possible class implementation of the Marsenne generator

$$x_i = (7^5 x_{i-1})\text{mod}(2^{31}-1) \tag{4.36}$$

```
class MCG:
    def __init__(self,seed,a=7**5,m=2**31-1): #constructor
        self.a=a # Python: self.a; Java: this.a; C++: this->a
        self.m=m
        self.x=seed
    def next(self):
        self.x=mulmod(self.a,self.x,self.m)
    def uniform(self):
        self.next()
        return float(self.x)/self.m
    def random(self):
        return self.uniform()

def testMCG():
    marsenne=MCG(int(time()))
    for i in range(10):
        print marsenne.uniform()
```

4.4 Parallel generators and independent sequences

It is often necessary to generate many independent sequences. Consider for example our original problem of computing π. It may be a good idea to perform the computation in parallel and have different CPUs generate different random points. In this case it is imperative that the sequences are independent otherwise correlations between the sequences will make the additional CPUs statistically useless.

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	...	...

(4.37)

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.

4.4.1 Non-overlapping blocks

Let's consider one sequence of pseudo random numbers

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{4.38}$$

One can break it into sub-sequences of k numbers

$$\begin{aligned}
&x_0, x_1, ..., x_{k-1}\\
&x_k, x_{k+1}, ..., x_{2k-1}\\
&x_{2k}, x_{2k+1}, ..., x_{3k-1}\\
&...
\end{aligned}$$

If the original sequence is created with a multiplicative congruential generator

$$x_i = ax_{i-1}\text{mod}m \tag{4.39}$$

the sub-sequences can be generated independently because

$$x_{nk-1} = a^{nk-1}x_0\text{mod}m \tag{4.40}$$

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.

4.4.2 Leapfrogging

Let's consider one sequence of pseudo random numbers

$$x_0, x_1, ..., x_k, x_{k+1}, ..., x_{2k}, x_{2k+1}, ..., x_{3k}, x_{3k+1}, ..., \tag{4.41}$$

One can break it into sub-sequences of k numbers

$$\begin{aligned}
&x_0, x_k, x_{2k}, x_{3k}, ...\\
&x_1, x_{1+k}, x_{1+2k}, x_{1+3k}, ...\\
&x_2, x_{2+k}, x_{2+2k}, x_{2+3k}, ...\\
&...
\end{aligned}$$

The seeds $x_1, x_2, ..x_{k-1}$ are generated from x_0 and the independent sequences can be generated independently using the formula

$$x_{i+k} = a^k x_i\text{mod}m \tag{4.42}$$

therefore leapfrogging is also a viable technique for parallel random number generators.

```
class MCG:
    # ... see class definition above
    def leapfrog(self,k):
        a=self.a
        m=self.m
        generators=range(k)
        for i in range(k):
            self.next()
            generators[i]=MCG(self.x,powmod(a,k,m),m)
        return generators

def test_leapfrog():
    generators=MCG().leapfrog(3)
    for k in range(3):
        for i in range(5):
            x=generators[k].uniform()
            print k,'\t',i,'\t',x

>>> testleapfrog()
```

4.4.3 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

$$\begin{aligned} x_i^L &= (a_L x_{i-1} + c_L)\mathrm{mod} m \\ x_i^R &= (a_R x_{i-1} + c_R)\mathrm{mod} m \end{aligned}$$

then, recursively, append nodes to the children.

4.5 Analysis

4.5.1 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 then 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.

4.5.2 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

$$\chi^2 = \sum_{i=0}^{i<n} \frac{(f_i - \pi_i)^2}{\pi_i} \tag{4.43}$$

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:

$$\begin{aligned} \chi^2_{(1000)} &= 11.24 \\ \chi^2_{(100000)} &= 0.126348 \end{aligned}$$

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

$$\forall n \forall \varepsilon \exists \overline{n} \mid \forall n_{events} > \overline{n}, \chi^2(n_{events}, n) < \varepsilon \tag{4.44}$$

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).

Chapter 5

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.

5.1 Generate PRNG from a discrete distribution

5.1.1 Uniform random integers 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

$$h_i = a + \lfloor (b - a + 1) y_i \rfloor \tag{5.1}$$

We will call the function that return random integers `randint(a,b)`

```
class MCG:
    # see class definition above
    def randint(self,a,b):
        return a+int((b-a+1)*self.uniform())
```

5.1.2 Table lookup

Let's say we want a random number generator that can only produce the outcome 0,1, or 2 with probabilities:

$$\begin{aligned} P(X &= 0) = 0.50 \\ P(X &= 1) = 0.23 \\ P(X &= 2) = 0.27 \end{aligned}$$

since the probability of the possible outcomes are rational numbers (fractions) we can proceed as follows:

```
class MCG:
    # see class definition above
    def table_lookup(self, table):
        u=self.uniform()
        for item,probability in table:
            if u<probability:
                return item
            else:
                u=u-probability
        return item #program should not arrive here if sum table[1]

def test_discrete_map(nevents=100,table=[[0,0.50],[1,0.23],[2,0.27]]):
    generator=MCG(time())
    f=[0,0,0]
    for k in range(nevents):
        p=generator.table_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 desired probabilities.

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 $P(X = 0), P(X = 1), ..., P(x = n - 1)$ so that $\sum P(X = i) = 1$. 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.

5.1.3 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:

- $\forall j \geq 0, a_{0j} = P(X = x_j)$
- $\forall j \geq 0$ 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:

$$a_{ij} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & ... & a_{0,n-1} \\ ... & ... & ... & ... & \\ a_{t-2,0} & a_{t-2,1} & a_{t-2,2} & a_{t-2,3} & \\ a_{t-1,0} & a_{t-1,1} & & & \end{pmatrix} \tag{5.2}$$

In other words we can say that:

- a_{ij} represents the probability

$$P(X = x_j) \tag{5.3}$$

- a_{1j} represents the probability

$$P(X = x_{2j} or X = x_{2j+1}) \tag{5.4}$$

- a_{4j} represents the probability

$$P(X = x_{4j} \text{ or } X = x_{4j+1} \text{ or } X = x_{4j+2} \text{ or } X = x_{4j+3}) \tag{5.5}$$

- a_{ij} represents the probability

$$P(X \in \{x_k | 2^i j \leq k < 2^i(j+1)\}) \tag{5.6}$$

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

    def discrete_map(self, u):
        i=int(self.t)-1
        j=0
        b=0
        while i>0:
            if u>b+self.a[i][j]:
                b=b+self.a[i][j]
                j=2*j+2
            else:
                j=2*j
            i=i-1
        if u>b+self.a[i][j]:
            j=j+1
        return self.table[j][0]
```

5.1.4 Accept-reject method

The accept-reject method is based on the simple statement that if u is a uniform random number and x is a given number in range $[0, 1]$, than the block

```
if u<x:
    # block
```

is executed with probability $(100x)\%$. Here is a function that draws from a discerete probability distribution using accept-reject.

```
class MCG:
    # see class definition above
    def discrete_accept_reject(self, table):
        while 1:
            i=self.randint(0,len(table)-1)
            u=self.uniform()
            if u<table[i]: return i

def test_discrete_ar(nevents=100,table=[0.50,0.23,0.27]):
    generator=MCG(time())
    f=[0,0,0]
    for k in range(nevents):
        p=generator.discrete_accept_reject(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])

>>> test_discrete_ar()
```

produces the following output:

```
1 2 0 0 1 2 1 0 0 2 0 2 2 1 1 1 1 0 2 1 0 2 0 1 1
2 2 1 0 1 1 0 2 0 2 2 1 0 0 0 0 2 0 2 1 0 0 0 0 0
0 1 0 2 2 2 1 0 1 2 0 2 0 2 0 1 0 0 1 0 0 2 0 2 2
0 2 0 0 2 1 0 1 0 2 2 2 1 2 0 0 0 0 0 0 1 0 2 2 0
frequency[0]=0.450000
frequency[1]=0.240000
frequency[2]=0.310000
```

5.1.5 Drawing from a binomial distribution

As a particular case of what has been seen so far is a method to generate an integer number according to the binomial distribution. One way would be to generate a table according to

$$\text{table}[k] = P(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{5.7}$$

For large n it may be convenient not to store the table and use the formula directly. Moreover since the table is accessed sequentially by the table look-up algorithm one may just note that

$$\begin{aligned} P(X &= 0) = (1-p)^n \\ P(X &= k+1) = \frac{n}{k+1}\frac{p}{1-p}P(X=k) \end{aligned}$$

and rewrite the table look-up algorithm optimizing for the binomial distribution as

```python
class MCG:
    # see class definition above
    def binomial(self,n,p,precision=0.000001):
        u=self.uniform()
        probability=(1.0-p)**n
        for k in range(n+1):
            if u<probability+precision:
                return k
            else:
                u=u-probability
            probability=probability*(n-k)/(k+1)*p/(1.0-p)
        return n
```

The last line is there only to deal with precision issues.

5.1.6 Drawing from a Poisson distribution

Another particular case is the Poisson distribution, which is a limit case of the binomial distribution. In the Poisson distribution it is not possible to build a table of possible values because the random variable X can, in principle, take any integer value k and therefore the table would be infinite. On the other side we can use the same technique adopted for the Binomial distribution and observe that for Poisson

$$\begin{aligned} P(X &= 0) = e^{-\lambda} \\ P(X &= k+1) = \frac{\lambda}{k+1} P(X=k) \end{aligned}$$

therefore the above algorithm can be modified into

```python
class MCG:
    # see class definition above
    def binomial(self,lamb,precision=0.000001):
        u=self.uniform()
        probability=(1.0-p)**n
        k=0
        while 1:
            if u<probability+precision:
                return k
            else:
                u=u-probability
            probability=probability*lamb/(k+1)
            k=k+1
        return n
```

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.

5.2 Generate PRNG from a continuous distribution

Drawing from a given continuous distribution can be done in three ways:

- Accept-reject method
- Inversion method
- *ad-hoc* combined techniques

The accept-reject technique is very general but also very slow in most practical cases. The inversion methods requires an analytical computation that is only viable in some particular cases. Ad-hoc techniques can be used to circumvent the problems of the other two techniques.

5.2.1 Accept-reject method

This works very much as in the continuous case with one difference. A continuous distribution function is not necessarily bound by 1, as in the case of a discrete distribution. Let us assume that $p(x)$, the probability mass function of x, is bound by c

$$\forall x, p(x) \leq c \tag{5.8}$$

therefore we can generate an $x \in [a, b]$ with probability mass function $p(x)$ by generating a couple of uniform continuous random numbers x and y and returning x if $y \leq p(x)/c$. If the condition is not true the procedure is repeated. Here is the code that takes a probability mass function $p(x)$ and returns x.

```
class MCG:
    # see class definition above
    def discrete_accept_reject(self,p,a,b,c):
        while 1:
            x=a+(b-a)*self.uniform()
            y=c*self.uniform()
            if y<p(x): return x
```

5.2.2 Inversion method

The inversion method is based on the idea of finding a function $Y = f(X)$ so that if X is uniformly distributed in $[0, 1]$, Y has the desired probability mass function $p(y)$ with support in some subset of $[-\infty, \infty]$. In mathematical terms the cumulative distribution function is

$$F(y) = P[Y \leq y] = \int_{-\infty}^{y} p(y)dy = P[X < x(y)] = x(y) \tag{5.9}$$

therefore

$$y = f(x) = F^{-1}(x) \tag{5.10}$$

In English we can say that $f(x)$ is the inverse of the cumulative distribution function of Y.

Example 5 *Determine the function $Y = f(X)$ that maps a uniformly distributed variable X into a continuous random variable Y with probability mass function $p(y) = \lambda e^{-\lambda y}$ for $y \in [0, \infty]$*

Solution:

- Determine $F(y)$ from the definition:

$$F(y) = \int_{0}^{y} p(y)dy = 1 - e^{-\lambda y} \tag{5.11}$$

- Invert $x = F(y)$

$$y = -\frac{1}{\lambda}\log(1-x) \tag{5.12}$$

And here is the Python code that generates a y with the desired probability mass function $p(y) = \lambda e^{-\lambda y}$. The algorithm is implemented as a method of class MCG

```
class MCG:
    # ... see class definition above
    def exponential(self,lamb):
        return -log(1.0-self.uniform())/lamb
```

Note that for $u = 0$ this function returns 0 and when u approaches 1 this function approaches ∞ therefore it correctly maps $[0, 1]$ into $[0, \infty]$.

Example 6 *Determine the function $Y = f(X)$ that maps a uniformly distributed variable X into a continuous random variable Y with probability mass function $p(y) = \frac{3}{a^3}y^2$ for $y \in [0, a]$.*

Solution:

- Determine $F(y)$ from the definition:

$$F(y) = \int_0^y p(y)dy = \frac{y^3}{a^3} \tag{5.13}$$

- Invert $x = F(y)$

$$y = ax^{1/3} \tag{5.14}$$

Example 7 *Determine the funcion $Y = f(X)$ that maps a uniformly distributed variable X into a continuum random variable Y with probability mass function $p(y) = \frac{1}{2}\cos(y)$ for $x \in [-\pi/2, \pi/2]$.*

Solution:

- Determine $F(y)$ from the definition:

$$F(y) = \int_{-\pi/2}^y p(y)dy = \frac{1}{2}\sin(y) + \frac{1}{2} \tag{5.15}$$

- Invert $x = F(y)$

$$y = \arcsin(2x - 1) \tag{5.16}$$

5.2.3 Drawing from a Gaussian distribution

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 $y_2 = v_2\sqrt{(-2/s)\log s}$

Implemented in the method gaussian() of the class MCG:

```
class MCG:
    # ... see class definition above
    def gaussian(self):
        try:
            if self.other==None: raise Exception
            this=self.other
            self.other=None
            return this
        except:
            while 1:
                v1=2*self.uniform()-1
                v2=2*self.uniform()-1
                s=v1**2+v2**2
                if s<=1: break
            this=sqrt(-2*log(s)/s)*v1
            self.other=sqrt(-2*log(s)/s)*v2
            return 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.

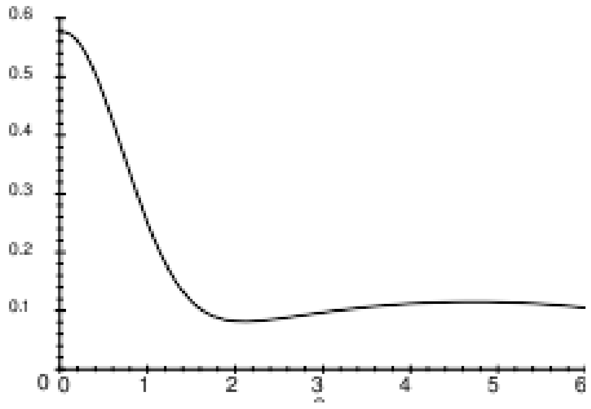
To map a random Gaussian number y with mean 0 and standard devaition 1 into another Gaussian number y' with mean μ and standard deviation σ:

$$y' = \mu + y\sigma \tag{5.17}$$

5.2.4 *ad-hoc* methods

Consider a probability mass function that looks like this:

$$\begin{aligned} p(x) &= c(5e^{-x^2} + \sin\frac{x}{3}) \text{ for } x \in [0,6] \\ c &= 0.11521 \end{aligned}$$



This function cannot be integrated analytically therefore the inversion method is out of question. We could use the accept-reject in the box $[0,6] \times [0,5c]$ but many of the accept reject steps would be wasted. At this point any useful observation about the function could be useful for example:

For $x < 2$, $p(x)$ is almost a gaussian and is bound by 0.6. For $x \geq 2$, $p(x)$ is almost $\sin(x)$ and it is bound by 0.11. We can therefore proceed in the following way:

We break the domain $[0,6]$ arbitrarily into $[0,2] \cup [2,6]$ and observe that our random variable fall into $[0,2]$ with probability equal to

$$P(0 \leq x \leq 2) = \int_0^2 c\left(5e^{-x^2} + \sin\frac{x}{3}\right) dx = 0.58213 \tag{5.18}$$

while the probability that the random variable falls into $[2,6]$ is

$$P(2 \leq x \leq 6) = \int_2^6 c\left(5e^{-x^2} + \sin\frac{x}{3}\right) dx = 0.41785 \tag{5.19}$$

and we can therefore choose one or the other segment using a table look-up then if x falls into $[0,2]$ we can use an accept reject in $[0,2] \times [0,5c]$ and if x falls in $[2,6]$ we can use an accept reject in $[2,6] \times [0,0.11]$.

5.3 Random points

5.3.1 In a hypervolume

Let me define a hypervolume V as a continuum and connected subset of N-dimensional space 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.

We can implement an impervolume as a function that takes the N coordinates $\mathbf{x} = (x_0, x_1, ...x_{n-1})$ of a point in the space R^N and returns 1 if the point belongs to the hypervolume, 0 otherwise.

Let's consider limited hypervolumes, i.e. hypervolumes that do not extend indefinitely in any direction in space. If this is the case then we can find a set of N pair a_i, b_i so that the function that defines the hypervolume differs from zero only if the coordinates of a point $\mathbf{x} = (x_0, x_1, ...x_{n-1})$ satisly $a_i \leq x_i \leq b_i$. In English: we can project the hypervolume on any axis i and the projection is bound by an interval $[a_i, b_i]$.

In order to generate a random point $\mathbf{x} = (x_0, x_1, ...x_{n-1})$ uniformly distributed inside a hypervolume we have to proceed in the following way:

- $\forall i$ identify a_i and b_i
- $\forall i$ generate a random number $x_i = a + (b - a)u_i$ where u_i is a uniformly distributed random number
- Accept the point if it falls inside the hypervolume, otherwise start again from previous step.

Here is an example of a program that generates a random point inside a circle:

```
def random_point_in_domain(box, S, generator):
    d=len(box)
    while 1:
        p=[]
        for i in range(d):
            p.append(generator.uniform()*
                     (box[i][1]-box[i][0])+box[i][0])
        if S(p)==1:
            return p

def example_circle(p):
    if p[0]**2+p[1]**2<200*2:
        return 1
    else:
        return 0

def example01():
    g=MCG(time())
    canvas=MyCanvas(200,200)
    for i in range(1000):
        p=random_point_in_domain([[0,200],[0,200]],
            example_circle,g)
        canvas.mark_point(p[0], p[1], 'red')
```

5.3.2 On a circle

A random point (x, y) uniformly distributed on a circle is obtained by generating a uniform random number u and

$$\begin{aligned} x &= \cos(2\pi u) \\ y &= \sin(2\pi u) \end{aligned}$$

5.3.3 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 $v_1^2 + v_2^2 + v_3^2 \leq 1$

$$\begin{aligned} x &= v_1/\sqrt{v_1^2 + v_2^2 + v_3^2} \\ y &= v_2/\sqrt{v_1^2 + v_2^2 + v_3^2} \\ z &= v_3/\sqrt{v_1^2 + v_2^2 + v_3^2} \end{aligned}$$

else start again.

Chapter 6

Simulations

6.1 General considerations

All Monte Carlo simulations start by building a model of the system to be simulated and identifying two key aspects:

- How randomness influences the time evolution of the system
- Which quantities we need to measure on the simulated system

When these questions have been answered the simulation is typically encoded in four parts

- SimulateOnce: the algorithm that computes one possible history of the system. The unknown events that affect the evolution of the system are generated at random using the PRNG
- Measure: that given a possible history of the system (such as the one coming from the simulation above) performs a measure of the desired quantity.
- SimulateMany: an algorithm that calls SimulateOnce in a loop and generates many possible histories for the system and, on each of them, measures (Measure) the interesting quantity. SimulateMany then returns the average of the different measurements.
- EstimateError: an algorithm that from the many possible histories used to compute the average measure quantity is able to determine the error on such average.

In practice, in many cases the Measure is coded into SimulateMany and returned by it. To EstimateError we dedicate an entire chapter. EstimateError can be used to implement a stopping condition in SimulateMany so that the latter generates as many history as required to reach a given target precision in the averaged measured quantity. In the following simple examples we will assume that the results of each individual measurements are Gaussian distributed around their average and, therefore, the error on the average is given by eq.(3.46). In the Chapter on Markov Chain Monte Carlo we will see when we use correlated random numbers this assumption is not true and a more sophisticated technique to esitamte the error is required.

6.2 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 simulator 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`)`.

Here is the class in Python:

```
class DisjSets:
    def __init__(self,n):
        self.sets=range(n)
        for i in range(n):
            self.sets[i]=-1

    def rep(self,i):
        while self.sets[i]>=0:
            i=self.sets[i]
        return i

    def connect(self,i,j):
        i1=self.rep(i)
        j1=self.rep(j)
        if i1!=j1:
            self.sets[i1]=self.sets[i1]+self.sets[j1]
            self.sets[j1]=i1

    def is_path(self,i,j):
        return self.rep(i)==self.rep(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 end-points 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`.

Here is the Python code:

```
class Network:
    def __init__(self,n_nodes,links ,generator):
        self.n_nodes=n_nodes
        self.links=links
        self.generator=generator
    def simulate_once(self,start ,stop):
        meta_net=DisjSets(self.n_nodes)
        for i,j,p in self.links:
            if self.generator.uniform()<p:
                meta_net.connect(i,j)
        if meta_net.is_path(start ,stop):
            return 1
        return 0
    def simulate_many(self,start,stop,n):
        sum=0.0
        for i in range(n):
            sum=sum+self.simulate_once(start,stop)
        return sum/n

def test_Network():
    n_nodes=3
    links=[(0,1,0.5), (1,2,0.5)]
    generator=MCG()
    net=Network(n_nodes,links,generator)
    print net.simulate_many(0,2,n=1000)
```

6.3 Example: Nuclear reactor

By a nuclear reactor we refer to a solid object of a given shape consisting of radioactive material. [FILL HERE]

```
class Queue:
    def __init__(self):
        self.queue=[]

    def enqueue(self,object):
        self.queue.append(deepcopy(object))

    def dequeue(self):
        if len(self.queue)==0:
            return None
        node=self.queue[0]
        del self.queue[0]
        return node


class Point:
    def __init__(self,x=0,y=0,z=0):
        self.x=x
        self.y=y
        self.z=z

def domainSphere(p,r):
    # for example a sphere of radius one centered at the origin
    if sqrt(p.x**2+p.y**2+p.z**2)<r: return 1
    return 0

def randomPointInDomain(domain,r,g):
    p=Point()
    while true:
        p.x=r*(2.0*g.uniform()-1)
        p.y=r*(2.0*g.uniform()-1)
        p.z=r*(2.0*g.uniform()-1)
        if domain(p,r):
            return p

def randomDecayVector(lamb,g):
    p=Point()
    while 1:
        x=2.0*g.uniform()-1
        y=2.0*g.uniform()-1
        z=2.0*g.uniform()-1
        s2=x*x+y*y+z*z
        if s2<1:
            dist=g.exponential(lamb)/sqrt(s2)
            p.x=dist*x
            p.y=dist*y
            p.z=dist*z
            return p

class NuclearReactor:
    def __init__(self, domain, r, lamb, maxhits, generator):
        self.domain=domain
        self.r=r
        self.lamb=lamb
        self.maxhits=maxhits
        self.generator=generator
    def simulate_once(self):
        self.history=[]
        q=Queue()
        p=randomPointInDomain(self.domain,self.r,self.generator)
        while p:
            # first decay particle
            v=randomDecayVector(self.lamb,self.generator)
            u=Point(p.x+v.x,p.y+v.y,p.z+v.z)
            if self.domain(u,self.r):
                q.enqueue(u)
            # for plotting in vrml
            self.history.append(deepcopy([p,u]))
            # second decay particle
```

```
                v=randomDecayVector(self.lamb,self.generator)
                u=Point(p.x+v.x,p.y+v.y,p.z+v.z)
                if self.domain(u,self.r):
                    q.enqueue(u)
                # for plotting in vrml
                self.history.append(deepcopy([p,u]))
                # if size of queue increases exponentially
                # we have a chain reaction
                if len(q.queue)>self.maxhits:
                    return 1
                p=q.dequeue()
            # if size of queue went to zero
            # we have not chain reaction
            return 0

    def simulate_many(self,nevents):
        sum=0.0
        for i in range(nevents):
            sum=sum+self.simulate_once()
        return sum/nevents

def test_NuclearReactor():
    g=MCG()
    for r in range(5,15):
        reactor=NuclearReactor(domainSphere,r,lamb=0.1,
                               maxhits=100,generator=g)
        print r, reactor.simulate_many(100)

test_NuclearReactor()
```

Chapter 7

Monte Carlo Integration

7.1 1d Monte Carlo integration

Let's consider a one dimensional integral

$$I = \int_a^b f(x)dx \tag{7.1}$$

Let's now determine two functions $g(x)$ and $p(x)$ such that

$$p(x) = 0 \text{ for } x \in [-\infty, a] \cup [n, \infty] \tag{7.2}$$

and

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \tag{7.3}$$

and

$$g(x) = f(x)/p(x) \tag{7.4}$$

We can interpret $p(x)$ as a probability mass function and

$$E[g(X)] = \int_{-\infty}^{+\infty} g(x)p(x)dx = \int_a^b f(x)dx = I \tag{7.5}$$

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

$$p(x) \overset{def}{=} \left\{ \begin{array}{ll} 1/(b-a) & \text{if } x \in [a,b] \\ 0 & \text{otherwise} \end{array} \right\} \tag{7.6}$$
$$g(x) \overset{def}{=} (b-a)f(x)$$

so that X is just a uniform random variable in $[a,b]$ and using eq.(3.21):

$$I = E\left[f(X)\right] = (b-a)\frac{1}{N}\sum_{i=0}^{i<N} f(x_i) \tag{7.7}$$

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

$$\sigma^2 = (b-a)^2\frac{1}{N}\sum_{i=0}^{i<N} \left[f(x_i) - \langle f\rangle\right]^2 \tag{7.8}$$

with

$$\langle f\rangle = \frac{1}{N}\sum_{i=0}^{i<N} f(x_i) \tag{7.9}$$

and the error on the result is

$$\delta I = \sqrt{\frac{\sigma^2}{N-1}} \tag{7.10}$$

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 (7.6) 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.

7.2 2-d Monte Carlo integration

The technique described above can easily be extended to 2d integrals

$$I = \int_D f(x_0, x_1) dx_0 dx_1 \tag{7.11}$$

where 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

$$p_0(x_0) = 0 \text{ or } p_1(x_1) = 0 \text{ for } x \notin D \tag{7.12}$$

and

$$\int p_0(x_0) p_1(x_1) dx_0 dx_1 = 1 \tag{7.13}$$

and

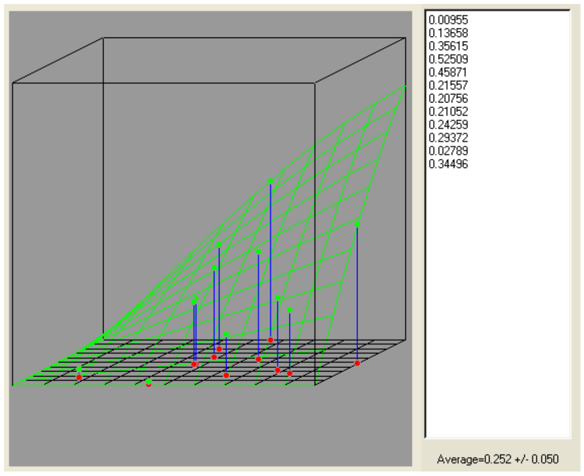
$$g(x_0, x_1) = \frac{f(x_0, x_1)}{p_0(x_0) p_1(x_1)} \tag{7.14}$$

We can interpret $p(x_0, x_1)$ as a probability mass function for two independent random variables X_0 and X_1 and

$$E[g(X_0, X_1)] = \int g(x_0, x_1) p_0(x_0) p_1(x_1) dx = \int_D f(x_0, x_1) dx_0 dx_1 = I \quad (7.15)$$

Therefore

$$I = E[g(X_0, X_1)] = Area(D) \frac{1}{N} \sum_{i=0}^{i<N} f(x_{i0}, x_{i1}) \quad (7.16)$$



```
0.00955
0.13658
0.35615
0.52509
0.45871
0.21557
0.20756
0.21052
0.24259
0.29372
0.02789
0.34496

Average=0.252 +/- 0.050
```

7.3 nD Monte Carlo integration

The technique described above can also be extended to n-d integrals

$$I = \int_D f(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} \quad (7.17)$$

where 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

$$p(x_0, ..., x_{n-1}) = 0 \text{ for } x \notin D \quad (7.18)$$

and

$$\int p(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} = 1 \quad (7.19)$$

and

$$g(x_0, ..., x_{n-1}) = f(x_0, ..., x_{n-1})/p(x_0, ..., x_{n-1}) \tag{7.20}$$

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

$$\begin{aligned} E[g(X_0, ..., X_{n-1})] &= \int g(x_0, ..., x_{n-1}) p(x_0, ..., x_{n-1}) dx \\ &= \int_D f(x_0, ..., x_{n-1}) dx_0 ... dx_{n-1} = I \end{aligned}$$

Therefore

$$I = E[g(X_0, .., X_{n-1})] = Volume(D) \frac{1}{N} \sum_{i=0}^{i<N} f(\mathbf{x}_i) \tag{7.21}$$

where for every point $\mathbf{x}_i$ is a tuple $(x_{i0}, x_{i1}, ..., x_{i,n-1})$.

Here is the Python code:

```python
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()
```

Chapter 8

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.

8.1 Random Walk

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 0 $S_0 = 0$. 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:

$$S_{n+1} = S_n + \varepsilon_n \Delta_x \tag{8.1}$$

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

$$P(S_{n+1} = x | \{S_i\} \text{ for } i \leq n) = P(S_{n+1} = x | S_n) \tag{8.2}$$

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. Typical models:

- 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*[1].

8.1.1 Random Walk - Wiener process

Let's assume a Wiener process for our 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

$$\frac{n!}{n_+!(n-n_+)!} p^{n_+} (1-p)^{n-n_+} \tag{8.3}$$

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

$$\begin{aligned} n &= n_+ + n_i \\ k &= n_+ - n_- \end{aligned}$$

[1]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.

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

$$P(n,k) = \frac{n!}{((n+k)/2)!((n-k)/2)!} p^{(n+k)/2}(1-p)^{(n-k)/2} \tag{8.4}$$

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.

8.1.2 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.(8.1) gives

$$S_n = \Delta_x \sum_{i=0}^{i<n} \varepsilon_i \tag{8.5}$$

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

$$p(S_n) = \frac{1}{\sqrt{2\pi n \Delta_x^2}} e^{-x^2/(2n\Delta_x^2)} \tag{8.6}$$

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

$$\begin{aligned} P(a \;\; \leq \;\; S_n \leq b) &= \frac{1}{\sqrt{2\pi n \Delta_x^2}} \int_a^b e^{-x^2/(2n\Delta_x^2)} dx \\ &= \frac{1}{\sqrt{2\pi}} \int_{a/(\sqrt{n}\Delta_x)}^{b/(\sqrt{n}\Delta_x)} e^{-x^2/2} dx \\ &= \operatorname{erf}(\frac{b}{\sqrt{n}\Delta_x}) - \operatorname{erf}(\frac{a}{\sqrt{n}\Delta_x}) \end{aligned}$$

8.2 Example: Financial Applications

Before we can introduce options and option pricing we have to introduce some basic financial concepts.

8.2.1 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.

```python
class Transaction:
    def __init__(self,args):
        self.time=args[0]
        self.amount=args[1]
```

8.2.2 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.

$$\text{NPV} = Ae^{-rt} \tag{8.7}$$

or in Python:

```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%).

8.2.3 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

(8.8)

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

$$\begin{aligned}\text{NPV} &= \sum_i A_i e^{-rt_i} \\ &= 100 - 50e^{0.05\cdot 2} - 40e^{0.05\cdot 3} - 30e^{0.05\cdot 4} \\ &= -4.23211255114\end{aligned}$$

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:

```python
def NetPresentValue(listTransactions,r):
    sum=0
    for item in listTransactions:
        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.

8.2.4 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.

8.2.5 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

$$\text{NPV} = S - Ae^{-rt} \tag{8.9}$$

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:

```python
def FutureCost(assetPrice ,deliveryPrice ,expirationTime ,rate):
    return assetPrice-deliveryPrice*exp(-rate*expirationTime)
```

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.*

8.2.6 Options

The simplest option is a European[2] 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, 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

(8.10)

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

[2]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.

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*.

8.2.7 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

$$\max(S_\tau - A, 0) \tag{8.11}$$

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

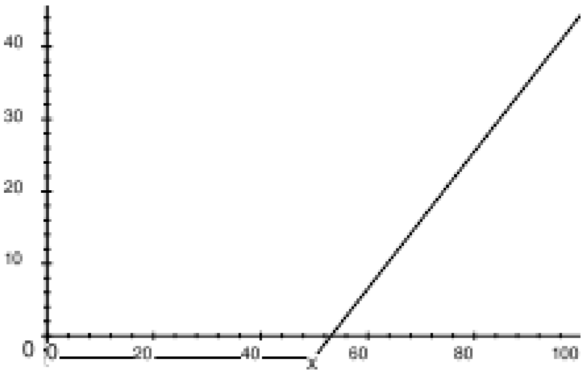
$$\text{PresentValue}_{long-call} = -C_{call} + \max(S_\tau - A, 0)e^{-r\tau} \tag{8.12}$$

while from the point of view of the seller of the same option

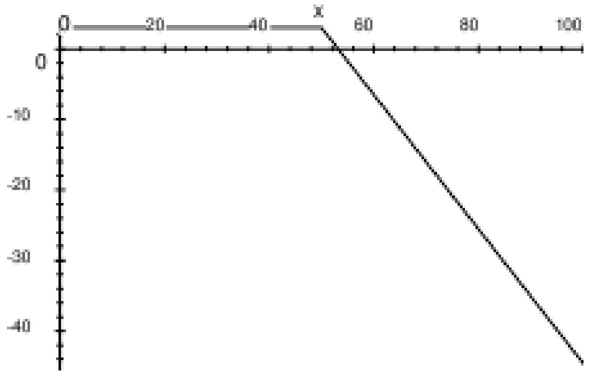
$$\text{PresentValue}_{short-call} = +C_{call} - \max(S_\tau - A, 0)e^{-r\tau} \tag{8.13}$$

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:

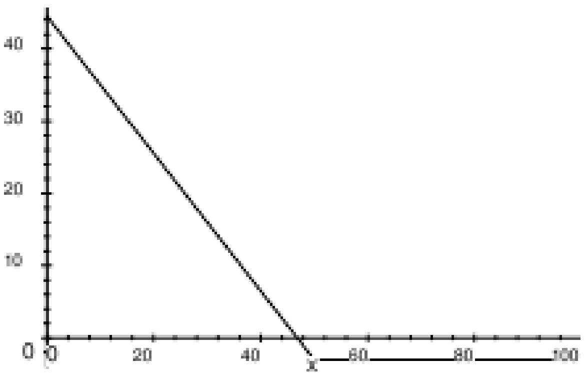
$$C_{call} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(x - A, 0) p_\tau(x) dx \tag{8.14}$$

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 then 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

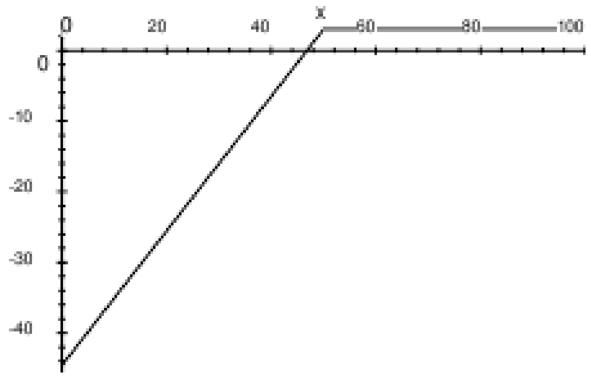
$$\text{PresentValue}_{long-put} = -C_{put} + \max(A - S_\tau, 0) e^{-r\tau} \tag{8.15}$$

$$\text{PresentValue}_{short-put} = +C_{put} - \max(A - S_\tau, 0)e^{-r\tau} \qquad (8.16)$$

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

$$C_{put} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(A - S_\tau, 0) p_\tau(S_\tau) dS_\tau \qquad (8.17)$$

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?

8.2.8 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

$$
\begin{aligned}
S_{n+1} &= S_n u \text{ with probability } p && (8.18)\\
S_{n+1} &= S_n d \text{ with probability } 1-p && (8.19)
\end{aligned}
$$

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

$$P(S_\tau = S_u u^i d^{n-i}) = \binom{n}{i} p^i (1-p)^{n-i} \tag{8.20}$$

and therefore

$$C_{call} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i \le n} \binom{n}{i} p^i (1-p)^{n-i} \max(S_u u^i d^{n-i} - A, 0) \tag{8.21}$$

and

$$C_{put} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i \le n} \binom{n}{i} p^i (1-p)^{n-i} \max(A - S_u u^i d^{n-i}, 0) \tag{8.22}$$

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.(8.21) and (8.22) 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.

8.2.9 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

$$S_{n+1} = S_n \left[1 + \mu\Delta_t + \sigma\varepsilon_n\sqrt{\Delta_t}\right] \tag{8.23}$$

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.(8.23) becomes $S_{n+1} = S_n(1 + \mu\Delta_t)$ whose solution is

$$S_n = S_0(1 + \mu\Delta_t)^n \tag{8.24}$$

if we consider continuous time $t = n\Delta_t$

$$S(t) = S_0(1 + \mu\Delta_t)^{t/\Delta_t} \tag{8.25}$$

and in the limit $\Delta_t \to 0$

$$S(t) = S_0 e^{\mu t} \tag{8.26}$$

where we recognize the formula eq.(8.7) with $\mu = -r$. Therefore in absence of stochastic fluctuations the price of the asset would grow exponentially at an interest rate μ.

In general

$$\begin{aligned}
\text{Mean of } \frac{\Delta S}{S} &= \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{n<N} \frac{S_{n+1} - S_n}{S_n} = \mu\Delta_t \\
\text{Variance of } \frac{\Delta S}{S} &= \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{n<N} \left[\frac{S_{n+1} - S_n}{S_n} - \mu\Delta_t\right]^2 = \sigma^2\Delta_t
\end{aligned}$$

This formula gives us a way to measure μ and σ from historic data. Therefore one can compute the cost C of an option:

$$C = e^{-r\tau} \int_{-\infty}^{+\infty} f(x) p_\tau(x) dx \tag{8.27}$$

with $f(x) = \max(x - A, 0)$ for a call and $f(x) = \max(A - x, 0)$ for a put by using eq.(3.21):

$$\int_{-\infty}^{+\infty} f(x) p(x) dx \simeq \frac{1}{N} \sum_{i=0}^{i<N} f(x_i) \tag{8.28}$$

Therefore

$$C = e^{-r\tau} \frac{1}{N} \sum_{i=0}^{i<N} f(x_i) \tag{8.29}$$

where x_i ($i = 0...N$) are predictions for the spot price of the call option at expiration given from our model.

8.2.10 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.(8.23). We can use another model and eq.(8.29) 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)
```

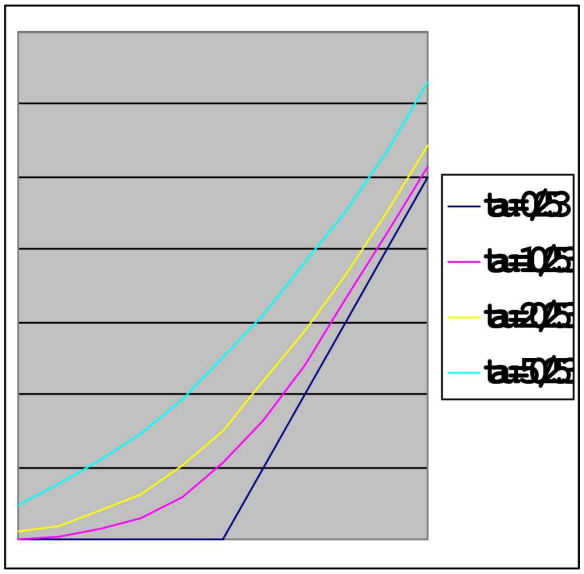
Note that:

- `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:



Chapter 9

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

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{9.1}$$

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?

9.1 Quadratic (not covered in class)

This technique only applies to the case when the $p(\mathbf{x})$ has the form

$$p(\mathbf{x}) \simeq e^{-\sum_{ij} \mathbf{A}_{ij} x_i x_j - \sum_i b_i x_i} \tag{9.2}$$

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.(9.2) and it follows that

$$p(\mathbf{y}) = p(y_0)p(y_1)...p(y_{n-1}) \tag{9.3}$$

where

$$p(y_i) \simeq e^{-\frac{(x-\mu_i)^2}{2\sigma_i^2}} \tag{9.4}$$

and

$$\begin{aligned}\sigma_i^2 &= \frac{1}{2D_{ii}} \\ \mu_i &= \sqrt{2}\sigma_i \sum_j b_j K^{-1}_{ji}\end{aligned}$$

Therefore $\mathbf{x} = x_0, x_1, ..., x_{n-1}$ can be generated as follow:

1. Find $\mathbf{K}$ and $\mathbf{D}$
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 $\mathbf{x} = \mathbf{K}\mathbf{y}$.

$\mathbf{x}$ generated this way will have the required probability mass function.

9.2 Metropolis

The exact technique described above works only when the probability mass function has the form of eq.(9.2), moreover determining the matrix $\mathbf{K}$ 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

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) \tag{9.5}$$

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:

```python
class Metropolis:
    def __init__(self,generator):
        self.generator=generator

    def step(self,p,q):
        old_x=self.x
        x=q(self.generator)
        if p(x)/p(old_x)<self.generator.uniform():
            x=old_x
        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()
```

9.3 Example: Metropolis Integration

We can use the Metropolis algorithm to compute nD integrals. For example

$$\begin{aligned}
& \int_0^1 \int_0^1 \sin(x_0 x_1)(x_0 - x_1)^2 dx_0 dx_1 \\
= & \int_0^1 \int_0^1 \frac{1}{6} \sin(x_0 x_1) p(x_0, x_1) dx_0 dx_1 \\
= & \frac{1}{N} \sum_{i=k}^{i<n+N} \frac{1}{6} \sin(x_0^{(i)} x_1^{(i)})
\end{aligned}$$

Here is the program:

```
def test_MetropolisIntegral(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_MetropolisIntegral()
```

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...

9.4 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

$$\frac{1}{n}\left(|x_0 - x_1| + |x_0 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|\right) \tag{9.6}$$

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_PermutationDistance(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_PermutationDistance(10)
```

Notice that the program that generates all combinations runs in $\Theta(n!)$ while the program above converges quite fast to the right answer.

9.5 Metropolis on Locality

Lets' consider a probability mass function of the form

$$p(\mathbf{x}) = p(x_0, x_1, ..., x_{n-1}) = e^{\sum_{jk} A_{ij}(x_j) x_k} \tag{9.7}$$

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

$$\forall j \neq k, x_j^{(i+1)} = x_j^{(i+1)} \text{ and } x_k^{(i+1)} = \delta \tag{9.8}$$

With this choice and a desired probability mass function given by eq.(9.7) 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.(9.7) 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.

9.6 Example: Ising model

Chapter 10

Jackknife and Bootstrap errors

10.1 Statistical errors in MCMC

10.2 Jackknife

10.3 Bootstrap

Chapter 11

Simulated Annealing

11.1 Minimization problems

11.2 Example: Protein folding

Chapter 12

Appendices

12.1 Appendix A: Math review

12.1.1 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

(12.1)

12.1.2 Set Theory Review

Important Sets

0	empty set
N	natural numbers {0,1,2,3,...}
N^+	positive natural numbers {1,2,3,...}
Z	all integers {...,-3,-2,-1,0,1,2,3,...}
R	all real numbers
R^+	positive real numbers (not including 0)
R^*	positive numbers including 0

(12.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**
$$\mathcal{A} \cap \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{12.3}$$
- **Union**
$$\mathcal{A} \cup \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{12.4}$$
- **Difference**
$$\mathcal{A} - \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{12.5}$$

Corresponding Python functions:

```
def Intersection(A,B):
    C=[]
    for element in A:
        if element in B:
            C=C+[element]
    return C

def Union(A,B):
    C=[]
    for element in A+B:
        if not element in C:
            C=C+[element]
    return C

def Difference(A,B):
    C=[]
    for element in A:
        if not element in B:
            C=C+[element]
    return C
```

Set laws

- Empty set laws

$$\mathcal{A} \cup \mathbf{0} = \mathcal{A}$$
$$\mathcal{A} \cap \mathbf{0} = \mathbf{0}$$

- Idempotency laws

$$\mathcal{A} \cup \mathcal{A} = \mathcal{A}$$
$$\mathcal{A} \cap \mathcal{A} = \mathcal{A}$$

- Commutative laws

$$\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A}$$
$$\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A}$$

- Associative laws

$$\mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C}$$
$$\mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C}$$

- Distributive laws

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C})$$
$$\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C})$$

- Absorption laws

$$\mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) = \mathcal{A}$$
$$\mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) = \mathcal{A}$$

- DeMorgan laws

$$\mathcal{A} - (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C})$$
$$\mathcal{A} - (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C})$$

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(N)=countable infinite (∞)
- cardinality(R)=uncountable infinite (∞) !!!

Relations

- A **Cartesian Product** is defined as

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \tag{12.6}$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ if a subset of their Cartesian product.
- A binary relation is **transitive** if aRb and bRc implies aRc
- A binary relation is **symmetric** if aRb implies bRa
- A binary relation is **reflexive** if aRa if always true for each a.

Example 8 *$a < b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)*

Example 9 *$a > b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)*

Example 10 *$a = b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, symmetric and reflexive)*

Example 11 *$a \leq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)*

Example 12 *$a \geq b$ 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 $a \sim b$.

Theorem 11 *An equivalence relation is the same as a partition.*

Corresponding Python functions:

```
def CartesianProduct(A,B):
    C=[]
    for a in A:
        for b in B:
            C=C+[(a,b)]
    return C
```

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 there 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.

Example 13 *$f(n) \overset{def}{=} n mod 2$ with domain N and codomain N is not a surjection nor a bijection.*

Example 14 *$f(n) \overset{def}{=} n mod 2$ with domain N and codomain $\{0, 1\}$ is a surjection but not a bijection*

Example 15 *$f(x) \overset{def}{=} 2x$ with domain N and codomain N is not a surjection but is a bijection (in fact it is not invertible on odd numbers)*

Example 16 *$f(x) \overset{def}{=} 2x$ with domain R and codomain R is not a surjection and is a bijection (in fact it is invertible)*

12.1.3 Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \overset{def}{=} f(0) + f(1) + ... + f(n-1) \tag{12.7}$$

Corresponding Python functions:

```
def Sum(f, min_i, max_i):
    a=0
    for i in range(min_i, max_i):
        a=a+f(i)
    return a
```

Properties

- **Linearity**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \tag{12.8}$$

Proof:

$$\begin{aligned}
\sum_{i=0}^{i<n} af(i)+bg(i) &= (af(0)+bg(0))+(af(1)+bg(1))+...+(af(n-1)+bg(n-1)) \\
&= af(0)+af(1)+...+af(n-1)+bg(0)+bg(1)+...bg(n-1) \\
&= a\left(f(0)+f(1)+...+f(n-1)\right)+b\left(g(0)+g(1)+...+g(n-1)\right) \\
&= a\left(\sum_{i=0}^{i<n} f(i)\right)+b\left(\sum_{i=0}^{i<n} g(i)\right) \qquad (12.9)
\end{aligned}$$

Example 17

$$\sum_{i=0}^{i<n} c = cn$$

$$\sum_{i=0}^{i<n} i = \frac{1}{2}n(n-1)$$

$$\sum_{i=0}^{i<n} i^2 = \frac{1}{6}n(n-1)(2n-1)$$

$$\sum_{i=0}^{i<n} i^3 = \frac{1}{4}n^2(n-1)^2$$

$$\sum_{i=0}^{i<n} x^i = \frac{x^n-1}{x-1} \textit{ (geometric sum)}$$

$$\sum_{i=0}^{i<n} \frac{1}{i(i+1)} = 1-\frac{1}{n} \textit{ (telescopic sum)}$$

12.2 Appendix B

12.2.1 Preliminaries

From the Taylor expansion of the exponential function

$$e^\alpha = \sum_{n=0}^{\infty} \frac{\alpha^n}{n!} \qquad (12.10)$$

one can prove that

$$e^{i\alpha} = \cos(\alpha) + i\sin(\alpha) \tag{12.11}$$

Proof.

$$\begin{aligned} e^{i\alpha} &= \sum_{n=0}^{\infty} \frac{(i\alpha)^n}{n!} \qquad (12.12) \\ &= \underbrace{\sum_{k=0}^{\infty}(-1)^k \frac{(\alpha)^{2k}}{(2k)!}}_{\text{even terms}} + i \underbrace{\sum_{k=0}^{\infty}(-1)^k \frac{(\alpha)^{2k+1}}{(2k+1)!}}_{\text{odd terms}} \\ &= \cos(\alpha) + i\sin(\alpha) \end{aligned}$$

Let's now define a θ function,

$$\theta(x) \overset{def}{=} \left\{ \begin{array}{ll} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{array} \right\} \tag{12.13}$$

and a δ_ε function

$$\delta_\varepsilon(x) \overset{def}{=} \frac{1}{2\varepsilon}\left[\theta(x+\varepsilon) - \theta(x+\varepsilon)\right] \tag{12.14}$$

that has the following properties:

$$\delta_\varepsilon(x) = \left\{ \begin{array}{ll} 0 & \text{if } x > +\varepsilon \\ \frac{1}{2\varepsilon} & \text{if } -\varepsilon \leq x \leq +\varepsilon \\ 0 & \text{if } x < -\varepsilon \end{array} \right\} \tag{12.15}$$

$$\int_{-\infty}^{+\infty} \delta_\varepsilon(x)dx = \int_{-\varepsilon}^{+\varepsilon} \delta_\varepsilon(x)dx = \frac{1}{2\varepsilon}\int_{-\varepsilon}^{+\varepsilon} dx = 1 \tag{12.16}$$

Let's also define a Dirac δ function as the limit of δ_ε for ε that goes to 0

$$\delta(x) \overset{def}{=} \lim_{\varepsilon \to 0} \delta_\varepsilon(x) \tag{12.17}$$

The δ function is zero for every value of the integrand and infinity when the integrand is zero

$$\delta(x) = \left\{ \begin{array}{ll} 0 & \text{if } x > 0 \\ \infty & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{array} \right\} \tag{12.18}$$

Moreover the δ function has the property that

$$\int_{-\infty}^{+\infty} f(x)\delta(x-x_0)dx = f(x_0) \tag{12.19}$$

Proof.

$$\begin{aligned}
\int_{-\infty}^{+\infty} f(x)\delta(x-x_0)dx &= \lim_{\varepsilon\to 0}\int_{-\infty}^{+\infty} f(x)\delta_\varepsilon(x-x_0)dx \qquad (12.20)\\
\text{(substitute)} &= \lim_{\varepsilon\to 0}\frac{1}{2\varepsilon}\int_{x_0-\varepsilon}^{x_0+\varepsilon} f(x)dx\\
\text{(Taylor expand)} &= \lim_{\varepsilon\to 0}\frac{1}{2\varepsilon}\int_{x_0-\varepsilon}^{x_0+\varepsilon}\left[\sum_{n=0}^{\infty}\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n\right]dx\\
\text{(order change)} &= \lim_{\varepsilon\to 0}\frac{1}{2\varepsilon}\sum_{n=0}^{\infty}\frac{f^{(n)}(x_0)}{n!}\int_{x_0-\varepsilon}^{x_0+\varepsilon}(x-x_0)^n dx\\
\text{(integrate)} &= \lim_{\varepsilon\to 0}\frac{1}{2\varepsilon}\sum_{n=0}^{\infty}\frac{f^{(n)}(x_0)}{n!}\frac{\varepsilon^{n+1}-(-\varepsilon)^{n+1}}{n+1}\\
\text{(substitute } n \text{ with } 2k) &= \lim_{\varepsilon\to 0}\sum_{k=0}^{\infty}\frac{f^{(2k)}(x_0)}{(2k+1)!}\varepsilon^{2k}\\
&= \lim_{\varepsilon\to 0}\left[f(x_0)+O(\varepsilon)\right]\\
&= f(x_0)
\end{aligned}$$

One can prove, but we will not, that

$$\int_{-\infty}^{+\infty} e^{ixy}dy = 2\pi\delta(x) \tag{12.21}$$

12.2.2 Laplace Transforms

The Laplace Transform is a map between a function f and a function $\overline{f}$, where f and $\overline{f}$ are complex functions. The map is invertible. The inverse map is called anti Laplace Transform.

$$\begin{aligned} f &: R \to C \quad (12.22) \\ &\Downarrow \text{ L.T.} \\ \overline{f} &: R \to C \\ &\Downarrow \text{ anti L.T.} \\ \overline{\overline{f}} &: R \to C \text{ and } \overline{\overline{f}} \equiv f \end{aligned}$$

The Laplace Transform is defined as

$$\overline{f}(y) \overset{def}{=} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x) e^{ixy} dx \quad (12.23)$$

The anti Laplace Transform is defined as

$$\overline{\overline{f}}(x') \overset{def}{=} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \overline{f}(y) e^{-ix'y} dy \quad (12.24)$$

If one replaces $\overline{f}$ from eq.(12.23) in eq.(12.24) $\overline{\overline{f}}$ comes out to be the same as the original function f. This proves that eq.(12.23) and eq.(12.24) are one the inverse transformation of the other.

Proof.

$$\begin{aligned} \overline{\overline{f}}(x') &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} \left[\int_{-\infty}^{+\infty} f(x) e^{i(x-x')y} dx \right] dy \quad (12.25) \\ \text{(substitute)} &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x) e^{i(x-x')y} dy dx \\ \text{(use eq.(12.21))} &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} f(x) 2\pi \delta(x - x') dx \\ \text{(use eq.(12.19))} &= f(x') \end{aligned}$$

12.2.3 Discrete Laplace Transform

In many cases of practical interest the function f is a discrete function that is represented as an array of complex numbers. The map between the integer index of the array and the complex number stored at the location is the function f. In this case instead of the notation $f(x)$ we use the notation f_j where j is the array index and f_j is the value of the jth array element. Let n be the size of the array.

Under these conditions the Laplace Transform becomes a Discrete Laplace Transform

$$\begin{aligned} f &: \{0, ..., n-1\} \rightarrow C \\ &\Downarrow \text{ D.L.T.} \\ \overline{f} &: \{0, ..., n-1\} \rightarrow C \\ &\Downarrow \text{ anti-D.L.T.} \\ \overline{\overline{f}} &: \{0, ..., n-1\} \rightarrow C \text{ and } \overline{\overline{f}} \equiv f \end{aligned} \tag{12.26}$$

One can go from the continuous to the discrete transform by performing the following changes of variables in eq.(12.23) and eq.(12.24)

$$\begin{aligned} f(x) &\rightarrow f_j \text{ for } 0 \leq j < N \\ \overline{f}(y) &\rightarrow g_k \text{ for } 0 \leq k < N \\ x &\rightarrow \sqrt{\frac{2\pi}{n}} j \\ y &\rightarrow \sqrt{\frac{2\pi}{n}} k \end{aligned} \tag{12.27}$$

thus obtaining

$$g_k \overset{def}{=} \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j e^{\frac{2\pi ijk}{n}} \tag{12.28}$$

$$f_j \overset{def}{=} \frac{1}{\sqrt{n}} \sum_{k=0}^{k<n} g_k e^{-\frac{2\pi ijk}{n}} \tag{12.29}$$

The Discrete Laplace Transform and the Discrete Anti Laplace Transform are implemented in the following C++ code.

```
typedef complex<float> mdp_complex;
const float Pi=4.0*atan((float)1.0);
// input:  f[], n
// output: g[]
void LT(mdp_complex f[], mdp_complex g[], int n) {
  mdp_complex phase;
  for(int k=0; k<n; k++) {
    g[k]=0;
    for(int j=0; j<n; j++) {
      phase=mdp_complex(0,2.0*Pi*j*k/n); // (real, imag)
      g[k]+=f[j]*exp(phase);
    }
    g[k]/=sqrt((float) n);
  }
}

// input:  g[], n
// output: f[]
void anti_LT(mdp_complex f[], mdp_complex g[], int n) {
  mdp_complex phase;
  for(int j=0; j<n; j++) {
    f[j]=0;
    for(int k=0; k<n; k++) {
      phase=mdp_complex(0,-2.0*Pi*j*k/n); // (real, imag)
      f[j]+=g[k]*exp(phase);
    }
    f[j]/=sqrt((float) n);
  }
}
```

12.2.4 Fourier Transfroms

If the array f_i contains only real numbers ($\mathrm{Im} f_j = 0$ for each j) then

$$\begin{aligned} g_k &= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j e^{\frac{2\pi i j k}{n}} \qquad (12.30) \\ &= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \cos\left(\frac{2\pi j k}{n}\right) + i \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \sin\left(\frac{2\pi j k}{n}\right) \end{aligned}$$

and one can prove that

$$\begin{aligned} \mathrm{Re} g_{n-k} &= \mathrm{Re} g_k \qquad (12.31) \\ \mathrm{Im} g_{n-k} &= -\mathrm{Im} g_k \end{aligned}$$

Proof.

$$\begin{aligned}
\mathrm{Re} g_{n-k} &= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \cos\left(\frac{2\pi j(n-k)}{n}\right) \\
&= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \cos\left(2\pi j - \frac{2\pi jk}{n}\right) \\
&= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \cos\left(\frac{2\pi jk}{n}\right) = \mathrm{Re} g_k
\end{aligned}$$

$$\begin{aligned}
\mathrm{Im} g_{N-k} &= \frac{1}{\sqrt{N}} \sum_{j=0}^{j<N} f_j \sin\left(\frac{2\pi j(N-k)}{N}\right) \\
&= \frac{1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \sin\left(2\pi j - \frac{2\pi jk}{n}\right) \\
&= \frac{-1}{\sqrt{n}} \sum_{j=0}^{j<n} f_j \sin\left(\frac{2\pi jk}{n}\right) = -\mathrm{Im} g_k
\end{aligned}$$

If f_k are real numbers (no complex part) the Laplace transform takes the name of Fourier trasform and because of eqs.(12.31) there are relations between the elements of its transformed functions. In particular elements g_k with $k > n/2$ are redundant.

If this is the case one typically redefines the Fourier transform as a map between an array f of real values and size n and two arrays of real values and size $n/2$

$$\begin{aligned}
a_k &\overset{def}{=} \mathrm{Re} g_k \\
b_k &\overset{def}{=} \mathrm{Im} g_k
\end{aligned}$$

$$
\begin{array}{rcl}
f & : & \{0,...,n-1\} \to R \\
 & \Downarrow & \text{D.F.T.} \\
a & : & \{0,...,n/2\} \to R \\
b & : & \{1,...,n/2\} \to R \\
 & \Downarrow & \text{anti-D.F.T.} \\
\overline{\overline{f}} & : & \{0,...,n-1\} \to R \text{ and } \overline{\overline{f}} \equiv f
\end{array}
$$

Here is a C++ implementation of the Fourier Transform based on the Laplace Transform implemented above. Note that this is not a fast implementation and many optimizations are possible (FFT).

```

void FT(float f[], float a[], float b[], int n) {
  // n has to be even here
  int k;
  mdp_complex* fc=new mdp_complex[n];
  mdp_complex* gc=new mdp_complex[n];
  for(k=0; k<n; k++) fc[k]=mdp_complex(f[k],0);

  LT(fc,gc,n);

  for(k=0; k<n/2+1; k++) {
    a[k]=real(gc[k]);
    b[k]=imag(gc[k]);
  }
  delete[] gc;
  delete[] fc;
}

void anti_FT(float a[], float b[], float f[], int n) {
  // n has to be even here
  int k;
  mdp_complex* fc=new mdp_complex[n];
  mdp_complex* gc=new mdp_complex[n];
  for(k=0; k<n/2; k++) gc[k]=mdp_complex(a[k],b[k]);
  for(k=n/2; k<n; k++) gc[k]=mdp_complex(a[n-k],-b[n-k]);

  anti_LT(fc,gc,n);

  for(k=0; k<n; k++) f[k]=real(fc[k]);
  delete[] gc;
  delete[] fc;
}
```

12.2.5 Examples and Applications

Example 1

Let's consider a simple array f_j $(0 \leq j < n = 100)$ containing real numbers that are known to be generated from $f_j = A \sin \omega j$ but A and ω are unkown. The problem is determining A and ω. The f_j are plotted below

its DFT consists of a_k (identically zero because no *cos*) and b_k

The location of the peak $k = 7$ corresponds to the frequency $\omega = \frac{2\pi k}{n} = 0.43982$

The height of the peak $h = 5$ corresponds to the amplitude $A = \frac{2h}{\sqrt{n}} = 1.0$

Example 2

Let's now consider a new array f_i

In this case the origin of the data is not evident at all. The DFT gives the following a_k and b_k

from which we deduce that our f_j "contains"

- $A\cos(\omega j)$ with $A = \frac{2 \cdot 2.5}{\sqrt{n}} = 0.5$ and $\omega = \frac{2\pi \cdot 23}{n} = 1.4451$
- $A\sin(\omega j)$ with $A = \frac{2 \cdot 5}{\sqrt{n}} = 1.0$ and $\omega = \frac{2\pi \cdot 7}{n} = 0.43982$
- $A\sin(\omega j)$ with $A = \frac{2 \cdot 3.5}{\sqrt{n}} = 0.7$ and $\omega = \frac{2\pi \cdot 13}{n} = 0.81681$

Therefore

$$f_j = 0.5 \cos \frac{46\pi j}{n} + 1 \sin \frac{14\pi j}{n} + 0.7 \sin \frac{26\pi j}{n} \tag{12.32}$$

Example 3

Let's now consider another array f_i

In this case the DFT looks like

From which we deduce that

$$f_j = 1\cos\frac{26\pi j}{n} + 1\sin\frac{10\pi j}{n} + 1\sin\frac{14\pi j}{n} + ... \tag{12.33}$$

and the ... represents the contribution of all smaller peaks. Cutting the contribution of peaks smaller than 1 (in modulus) and performing an Anti-DFT one gets exactly

$$\overline{\overline{f}}_j = 1\cos\frac{26\pi j}{n} + 1\sin\frac{10\pi j}{n} + 1\sin\frac{14\pi j}{n} \tag{12.34}$$

that looks like

Which is not much different than the original f_j.

This example shows how Fourier transformtions can be used to extract some type of information from a data series. If the data series includes an oscillating contribution (sine or cosine), the DFT extracts that contribution.

For this reason DFT can be used to "clean up" a signal from noise, assuming that the signal has the form of a linear combination of sines and cosines. This is typically the case for radio waves and audible sound waves.

The DFT can also be used to implement some data compression. In fact by cutting small elements in the FT (according to some criteria that has to be decided on a case by case basis) one can reduce the amount of storage.

For example in the case of MP3, a soundwave is recorded (f_j) and Fourier transformed in a_k and b_k. The MP3 encoder therefore keeps (stores) only those components of the FT that have audible frequences. The MP3 encoder adds an extra level of compression by Huffman encoding the FT coefficients. The MP3 decoder performs Huffman decoding and computes the anti-FT, thus producing a new sound wave that is very similar to the original one.