University of Zurich - HS 2013

Computational Science I
Exercise notes: Matrices

Tobias Grubenmann

October 20, 2013

Exercise 1

The function `testSequences` compares two random sequences. One of them generated by the Halton algorithm with numbers 3 and 5, the other generated by the build-in random generator:

```
import pylab
import random

def getHaltonNumber(n, base):

    halton = 0
    exponent = 0

    while n > 0:
        r = n%base
        n = n/base

        exponent -= 1
        halton += r * pow(base, exponent)

    return halton

def testSequences():

    # plot Halton sequence
```

```
for i in range(512):
    pylab.plot(getHaltonNumber(i+1, 3), getHaltonNumber
        (i+1, 5), 'bo')

pylab.figure()

for i in range(512):
    pylab.plot(random.random(), random.random(), 'bo')
```

This generates the following output:

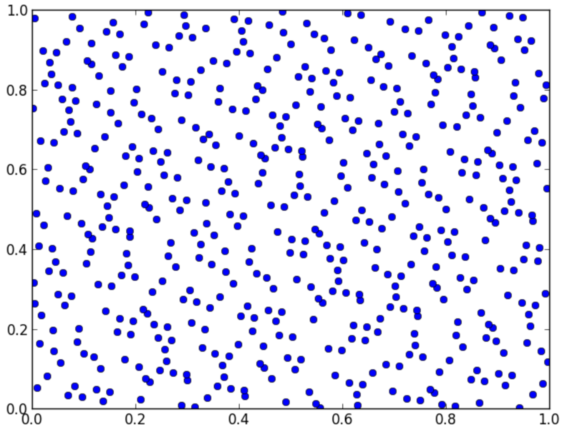

Figure 1: Random numbers generated by the Halton algorithmus with numbers 3 and 5

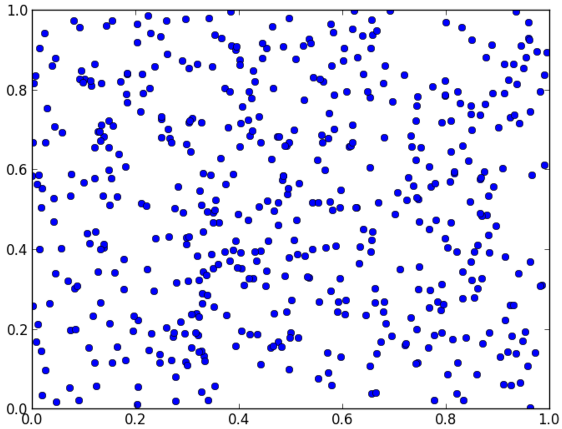

Figure 2: Random numbers generated by the built-in random generator

From the figures above we can see, that the quasi random numbers from the Halton algorithm have less clusters and seems to be more uniformly distributed.

Exercise 2

The function `simulateRandomWalks` simulates 10000 random walks with 50 steps and prints the distribution of the total distance as well as the distance from the origin:

```
import pylab
import random

def walkRandom(N):

    x = 0
    y = 0
    distance = 0

    for i in range(N):
        xSteps = random.random() - 0.5
        ySteps = random.random() - 0.5

        distance = sqrt(xSteps*xSteps + ySteps*ySteps)

        x += xSteps/distance
        y += ySteps/distance

    traveledDistance = sqrt(x*x + y*y)

    return (traveledDistance, x, y)

def simulateRandomWalks():

    distance = []
    xValues = []
    yValues = []

    for i in range(10000):

        distance += [walkRandom(50)[0]]
        xValues += [walkRandom(50)[1]]
        yValues += [walkRandom(50)[2]]

    subplot(311)
```

```
    hist(distance, bins=100, normed=True, histtype='step')
    subplot(312)
    hist(xValues, bins=100, normed=True, histtype='step')
    subplot(313)
    hist(yValues, bins=100, normed=True, histtype='step')
```

The plot of the distributions looks as follows:

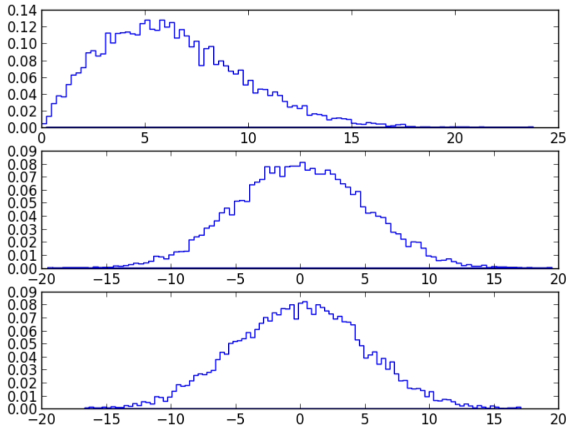

Figure 3: Distribution of total distance (above) and distance from the origin.

Exercise 3

I use the fact that:

$$\cos(x) = \frac{1}{2}(e^{ix} + e^{-ix})$$

With this, I can write:

$$\begin{aligned}\int_{-\pi}^{\pi} \cos^{100}(x)dx &= \int_{-\pi}^{\pi} (\frac{1}{2}(e^{ix} + e^{-ix}))^{100}dx = \frac{1}{2^{100}} \sum_{k=0}^{100} \binom{100}{k} \int_{-\pi}^{\pi} e^{ikx}(-1)^{100-k}e^{-i(100-k)x}dx \\ &= \frac{1}{2^{100}} \sum_{k=0}^{100} \binom{100}{k} (-1)^{100-k} \int_{-\pi}^{\pi} e^{ix(2k-100)}dx\end{aligned}$$

But if $k \neq 50$ we have $\int_{-\pi}^{\pi} e^{ix(2k-100)}dx = 0$ since we integrate just $N-2k$ times around the unit circle (in negative direction). Therefore:

$$\begin{aligned}\frac{1}{2^{100}}\sum_{k=0}^{100}\binom{100}{k}(-1)^{100-k}\int_{-\pi}^{\pi} e^{ix(2k-100)}dx &= \frac{1}{2^{100}}\binom{100}{50}\int_{-\pi}^{\pi} 1dx \\ &= \frac{1}{2^{100}}\binom{100}{50}2\pi\end{aligned}$$

Since we integrated over the domain $[-\pi, \pi]$, the average of $\cos^{100}(x)$ is just the integral divided by 2π which gives:

$$\frac{\binom{100}{50}}{2^{100}} \approx 0.0796$$

Exercise 4

The following script executes 100000 simulation where randomly D or M is reduced by 1 until either D or M is 0:

```
import pylab
import random

def run():

    D = 30
    M = 40

    maxFractDifferences = []

    for i in range(100000):

        currentD = D
        currentM = M

        maxFractDiff = 1.0*abs(currentD - currentM)/(D + M)

        while currentD > 0 and currentM > 0:

            # Choose randomly from D or M

            randomChoice = random.random()

            if randomChoice < 0.5:
                currentD -= 1
            else:
```

```
            currentM -= 1

        if 1.0*abs(currentD - currentM)/(D + M) >
           maxFractDiff:
            maxFractDiff = 1.0*abs(currentD - currentM)
               /(D + M)

    maxFractDifferences += [maxFractDiff]

# Print statistics from simulation
hist(maxFractDifferences, bins=20, histtype='step',
   normed=True, cumulative=True)

# Print KS statistics

v = sqrt(D*M/(D+M))

mu = v + 0.12 + 0.11/v

s = arange(0.01, 1, 0.01)

pksInv = [1 - 2 * reduce(lambda res, k : res + (-1)**k
   * exp(-2 *
    (k+1)**2 * mu**2 * t**2), range(100), 0) for t in s
       ]

pylab.plot(s, pksInv)
```

The output is a plot comparing the simulation with the Kolmogorov-Smirnov statistics:

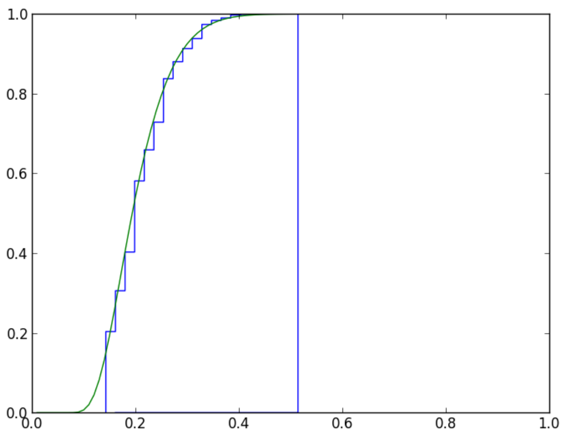

Figure 4: The Kolmogorov-Smirnov statistics and an approximation by a simulation