Ch.8.1-8.3: Random numbers and Monte Carlo simulation

Joakim Sundnes[1,2] Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Nov 15, 2017

Plan for this week

- Wednesday November 15:
  - Exer E.21, E.22
  - Random numbers and games
  - Exer 8.1, 8.5, (8.16)
- Friday November 17:
  - Vector ODEs (Systems of ODEs)
  - A class hierarchy of ODE solvers
  - Disease modeling (final project)

Random numbers are used to simulate uncertain events

Deterministic problems

- Some problems in science and technology are desrcribed by "exact" mathematics, leading to "precise" results
- Example: throwing a ball up in the air ($y(t) = v_0 t - \frac{1}{2}gt^2$)

Stochastic problems

- Some problems appear physically uncertain
- Examples: rolling a die, molecular motion, games
- Use *random numbers* to mimic the uncertainty of the experiment.

Random numbers are used to simulate uncertain events

Deterministic problems

- Some problems in science and technology are desrcribed by "exact" mathematics, leading to "precise" results
- Example: throwing a ball up in the air ($y(t) = v_0 t - \frac{1}{2}gt^2$)

Stochastic problems

- Some problems appear physically uncertain
- Examples: rolling a die, molecular motion, games
- Use *random numbers* to mimic the uncertainty of the experiment.

Random numbers are used to simulate uncertain events

Deterministic problems

- Some problems in science and technology are desrcribed by "exact" mathematics, leading to "precise" results
- Example: throwing a ball up in the air ($y(t) = v_0 t - \frac{1}{2} g t^2$)

Stochastic problems

- Some problems appear physically uncertain
- Examples: rolling a die, molecular motion, games
- Use *random numbers* to mimic the uncertainty of the experiment.

Drawing random numbers

Python has a `random` module for drawing random numbers.
`random.random()` draws random numbers in $[0, 1)$:

```
>>> import random
>>> random.random()
0.8155046885338104
>>> random.random()
0.4491326809029852
>>> random.random()
0.88320653116367454
```

Notice

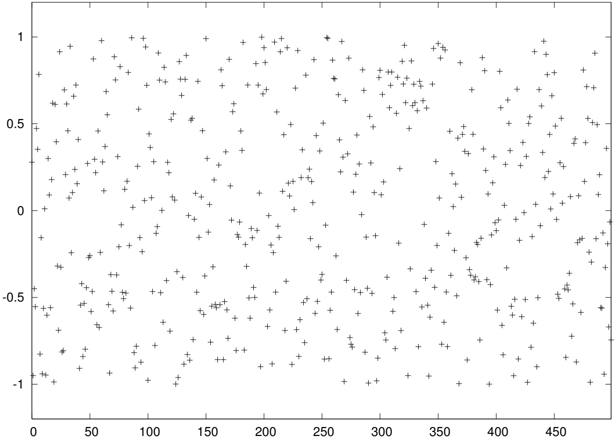
The sequence of random numbers is produced by a deterministic algorithm - the numbers just appear random.

Distribution of random numbers

- `random.random()` generates random numbers that are *uniformly distributed* in the interval $[0, 1)$
- `random.uniform(a, b)` generates random numbers uniformly distributed in $[a, b)$
- "Uniformly distributed" means that if we generate a large set of numbers, no part of $[a, b)$ gets more numbers than others

Distribution of random numbers visualized

```
N = 500  # no of samples
x = range(N)
y = [random.uniform(-1,1) for i in x]
import matplotlib.pyplot as plt
plt.plot(x, y, '+')
plt.show()
```



Vectorized drawing of random numbers

- `random.random()` generates one number at a time
- `numpy` has a `random` module that efficiently generates a (large) number of random numbers at a time

```
from numpy import random
r = random.random()              # one no between 0 and 1
r = random.random(size=10000)    # array with 10000 numbers
r = random.uniform(-1, 10)       # one no between -1 and 10
r = random.uniform(-1, 10, size=10000)  # array
```

- Vectorized drawing is important for speeding up programs!
- Possible problem: two `random` modules, one Python "built-in" and one in `numpy` (`np`)
- Convention: use `random` (Python) and `np.random`

```
random.uniform(-1, 1)                 # scalar number
import numpy as np
np.random.uniform(-1, 1, 100000)      # vectorized
```

Drawing integers

- Quite often we want to draw an integer from $[a, b]$ and not a real number
- Python's `random` module and `numpy.random` have functions for drawing uniformly distributed integers:

```
import random
r = random.randint(a, b)  # a, a+1, ..., b

import numpy as np
r = np.random.randint(a, b+1, N)          # b+1 is not included
r = np.random.random_integers(a, b, N)    # b is included
```

Example: Rolling a die

Problem

- Any no of eyes, 1-6, is equally probable when you roll a die
- What is the chance of getting a 6?

Solution by Monte Carlo simulation:

Rolling a die is the same as drawing integers in $[1, 6]$.

```python
import random
N = 10000
eyes = [random.randint(1, 6) for i in range(N)]
M = 0  # counter for successes: how many times we get 6 eyes
for outcome in eyes:
    if outcome == 6:
        M += 1
print('Got six %d times out of %d' % (M, N))
print('Probability:', float(M)/N)
```

Probability: `M/N` (exact: $1/6$)

Properties of Monte Carlo simulation

What is the probability that a certain event A happens?

Simulate N events and count how many times M the event A happens. The probability of the event A is then M/N (as $N \to \infty$).

- Not very useful for simple cases (like rolling a single die)
- Extremely useful for complex cases, where analytical solutions are hard or impossible to find
- Requires large N for accurate results (10^3-10^6 depending on application)

Example: Rolling a die; vectorized version

```
import sys, numpy as np
N = int(sys.argv[1])
eyes = np.random.randint(1, 7, N)
success = eyes == 6          # True/False array
M = np.sum(success)        # treats True as 1, False as 0
print('Got six %d times out of %d' % (M, N))
print('Probability:', float(M)/N)
```

Important!

Use sum from numpy and not Python's built-in sum function! (The latter is slow, often making a vectorized version slower than the scalar version.)

How accurate and fast is Monte Carlo simulation?

Programs:

- `single_die.py`: loop version
- `single_die_vec.py`: vectorized version

```
Terminal> time python single_die.py 100
Probability: 0.12
real        0m0.042s

Terminal> time python single_die.py 1000
Probability: 0.16
real        0m0.047s

Terminal> time python single_die.py 10000
Probability: 0.1636
real        0m0.058s

Terminal> time python single_die.py 1000000
Probability: 0.16696
real        0m1.348s

Terminal> time python single_die_vec.py 1000000
Probability: 0.167253
real        0m0.231s
```

Debugging programs with random numbers requires fixing the seed of the random sequence

- Debugging programs with random numbers is difficult because the numbers produced vary each time we run the program
- For debugging it is important that a new run reproduces the sequence of random numbers in the last run
- This is possible by fixing the *seed* of the `random` module: `random.seed(121)` (`int` argument)

```
>>> import random
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
>>> ['%.2f' % random.random() for i in range(7)]
['0.31', '0.61', '0.61', '0.58', '0.16', '0.43', '0.39']

>>> random.seed(2)    # repeat the random sequence
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
```

By default, the seed is based on the current time

Summary of Monte Carlo simulation

- The idea of MC simulation is very simple:
  - Repeat the experiment N times (i.e. a `for`-loop)
  - Count number of successes M
  - Probability of success is $p = M/N$
- Use the `random` or `numpy.random` modules for drawing random numbers