Problem set 1: Solutions

Workout 0.1

(a) Deterministic model: always produces the same results given the same initial conditions and parameters. Examples: ODEs, linear system of equations, earth surface via an integral.

Stochastic model: has a probabilistic nature and produces different results every time you solve it. Examples: Brownian motion describing particle movement, the SIR model by reactions, Stochastic differential equations (SDE)

(b) Deterministic method: always produces the same results given the same input data. Examples: Euler's method for ODEs, trapezoid and Simpson's rules for integration, Gaussian elimination for linear systems.

Stochastic method: produces different results every time, even if the input data stay the same. Example: Monte Carlo methods, SSA algorithm.

Workout 0.2 With $\boldsymbol{y} = (A, B, C)$ we have

$$\begin{array}{ll} A \xrightarrow{\alpha} B & \boldsymbol{v}_1 = [-1, 1, 0], \quad w_1 = \alpha y_1 \\ A + B \xrightarrow{\beta} 2B & \boldsymbol{v}_2 = [-1, 1, 0], \quad w_2 = \beta y_1 y_2 \\ C \xrightarrow{\gamma} A + B & \boldsymbol{v}_3 = [1, 1, -1], \quad w_3 = \gamma y_3 \end{array}$$

Workout 0.3

(a) The result would depend on the selection of function values $x_k \in \mathcal{U}(1, 3)$, as well as the value of N (i.e. the number of samples):

$$I \approx I_{app} = (3 - 1)\frac{1}{N}\sum_{k=1}^{N} x_k^2 \sin(x_k)$$

For example for $N = 1000$ we can write:

```
N = 1000
X = np.random.uniform(1,3,N)
I_app = 2*np.mean(X**2*np.sin(X))
```

(b) The absolute value of error is $|5.55342325 - I_{app}|$. One execution results in error 0.06851804. New execution gives a new error (randomness)!

(c) Since the approximate values of I are the *mean* of iid random variables, their distribution is *normal* according to the central limit theorem. The accuracy of approximation can be

measured by computing the *unbiased standard deviation* of function values and divide it by $\sqrt{N}$, where N is number of simulations.

(d) Since the order of convergence is $\mathcal{O}(1/\sqrt{N})$ or $e_N = c \cdot \frac{1}{\sqrt{N}}$ we have

$$\frac{e_{N_2}}{e_{N_1}} = \frac{\sqrt{N_1}}{\sqrt{N_2}} \Longrightarrow \frac{1}{4} = \frac{\sqrt{N_1}}{\sqrt{N_2}} \Longrightarrow N_2 = 16N_1$$

which means that to reduce the error by a factor of 4, we would need to increase the number of samples roughly by a factor of 16.

Workout 0.4

(a) At each step of the process, we take a random step from the previous position: $X(t+h) = X(t) + \sqrt{h}Z$, which means that every new state $X(t+h)$ only depends on the previous state $X(t)$ (plus "randomness"). In other words, the process is stochastic and the future is based solely on present and not past (memoryless), which makes it a Markov process.

(b) The random steps we take from $\mathcal{N}(0, h) \equiv \sqrt{h}\mathcal{N}(0, 1)$ are distributed with mean 0 and variance h (with h being the time step). So particles starting at x_0 will spread from there linearly in time, with the mean coordinate at x_0.

Workout 0.5

(a) Function `x = stock(x0, T)` is a stochastic process, and `x0` is a vector with 10 components. So, we run a Monte Carlo and at each step we call a random process generator:

```
N = 1000;  # or another big number
for i in range(N):
    x = stock(x0, T)  # one simulation
    v[i] = np.sum(x)
portfolio_value = np.mean(v)
std = np.std(v)
err = 1.96*std/np.sqrt(N)
```

(a) The error (the length of confidence interval) with for example 95% probability is calculated as $1.96\frac{s}{\sqrt{N}}$ where s is the sample standard deviation. By adding

```
s = np.std(v)
error = 1.96*s/np.sqrt(N)
```

to the above code, we can estimate the error with 95% probability.

Non-mandatory workouts

Workout 0.6

(a) To be a pdf, the area under f must be 1, so

$$\frac{1}{W}\int_0^{20}\frac{1}{1+x}dx = 1 \Longrightarrow W = \int_0^{20}\frac{1}{1+x}dx = \ln(1+x)\Big]_0^{20} = \ln 21$$

(b) The cdf $F(x)$ is computed as

$$F(x) = \int_0^x f(s)ds = \frac{1}{\ln 21}\int_0^x \frac{1}{1+s}ds = \frac{\ln(1+x)}{\ln 21}$$

To compute the inverse of F we write $x = F(y)$ to get $x \ln 21 = \ln(1+y)$ which gives $1 + y = e^{x \ln 21} = 21^x$ or $y = 21^x - 1 =: F^{-1}(x)$. To sample a random number from f we generate a uniform number $u \sim \mathcal{U}(0,1)$ and set $x = F^{-1}(u) = 21^u - 1$:

```
u = np.random.rand()
x = 21**u-1
```

Workout 0.7 We must first simulate the 4-sided dice by generating from the uniform discrete distribution with equal probabilities (1/4 for each side). Then we generate 10 numbers from this distribution and account the effect of each throw in all 10 tosses and compute the total wining. This process is repeated N times, and the final step to compute the probability of a negative wining involves counting the negative wining values of the accumulated results and dividing it by N:

```
N = 1000 # or another big number
total_win = np.empty(N)
for i in range(N):
    win = 0
    for i in range(10):
        side = RandDisct([1,2,3,4],[1/4,1/4,1/4,1/4])
        if side == 1 or side == 2:
           win += 1
        elif side == 3:
           win += 2
        else:
           win -= 1
    total_win[i] = win
pr_negative_win = np.size(np.where(total_win < 0))/N
```

Workout 0.8 The particle movement is modelled by a 3D Brownian motion, and the hitting time is computed by the product of time step h and number of steps until the boundary is hit. We choose a small time step $h = 0.005$ and large value $N = 1000$ (number of simulations). Finally we compute the mean to estimate the expected time. The error (the length of confidence interval) with 95% probability is calculated as $1.96\frac{s}{\sqrt{N}}$ where s is the sample standard deviation:

```
import numpy as np
def NextPos(X, h, dim):
    X = X + np.sqrt(h)*np.random.normal(0,1,dim)
    return X
h, N = 0.005, 1000
HitTime = np.empty(N)
for j in range(N):
    k = 0
    X = np.array([0,0,0])
    while True:
        X = NextPos(X, h, 3)
        if any(abs(X) >= 1):
            break
        k += 1
    HitTime[j] = h*k
HitTime_mean = np.mean(HitTime)
HitTime_std = np.std(HitTime)
err = 1.96*HitTime_std/np.sqrt(N)
print('Expected Hitting Time = ', HitTime_mean)
print('Error with 95% probability = ', err)
```

One execution gives the following outputs:

```
Expected Hitting Time = 0.485095
Error with 95% probability = 0.0193110
```