1. Assuming that each no. is stored as $0.abcd \times 10^x$, & two digits are stored after decimal. i.e. $0.ab \times 10^x$.

$$\left[\begin{array}{ccc|c} 0.4\times10^1 & 0.1\times10^1 & 0.2\times10^1 & 0.9\times10^1 \\ 0.2\times10^1 & 0.4\times10^1 & -0.1\times10^1 & -0.5\times10^1 \\ 0.1\times10^1 & 0.1\times10^1 & -0.3\times10^1 & -0.9\times10^1 \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.25 \\ - & - & - & \vdots \\ - & & & \end{array}\right]$$

$\downarrow \sim$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.2 \\ 2 & 4 & -1 & -5 \\ 1 & 1 & -3 & -9 \end{array}\right] \quad \begin{array}{l} (2.25 \sim 0.225\times10^1 \\ \sim 0.22\times10^1 \\ \sim 2.2) \end{array}$$

$\downarrow$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.2 \\ 0 & 3.5 & -2 & -9.4 \\ 0 & 0.75 & -3.5 & -11.2 \end{array}\right]$$

$\downarrow \sim$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.2 \\ 0 & 3.5 & -2 & -9.4 \\ 0 & 0.75 & -3.5 & -11 \end{array}\right] \quad \begin{array}{l} (11.2 \sim 0.112\times10^2 \\ \sim 0.11\times10^2 \\ \sim 11) \end{array}$$

$\downarrow$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.2 \\ 0 & 1 & -0.57 & -2.69 \\ 0 & 0.75 & -3.5 & -11 \end{array}\right]$$

$\downarrow \sim$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.2 \\ 0 & 1 & -0.57 & -2.7 \\ 0 & 0.75 & -3.5 & -11 \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & 0.25 & 0.5 & 2.2 \\ 0 & 1 & -0.57 & -2.7 \\ 0 & 0 & -3.1 & -9 \end{array}\right]$$

$$\therefore x_3 = \frac{9}{3.1} = 2.9 = 0.29 \times 10^1$$

$$x_2 = -2.7 + 0.57 \times 2.9 = -1.04 \approx -0.10 \times 10^1$$

$$x_1 = 2.2 - 0.5 \times 2.9 + 0.25 \times 1 = \cancel{0.50 \times 10^0}$$

$$= 1.00 = 0.10 \times 10^1$$

2.

(a) numpy.fft.fft (Python)

(b) numpy.linalg.qr (Python)

(c) numpy.random.lognormal (Python)

(d) ~~numpy~~ (gsl_odeiv2.h) with gsl_odeiv2_step_rk8pd (C)
or ~~np.sr~~ scipy.integrate.solve_ivp(method='DOP853')

(e) numpy.linald.svd (Python.

(f) ~~gsl_odeiv2.h~~ scipy.stats (Python), gsl_randist.h (C)

(g) (gsl_odeiv2.h) • gsl_odeiv2_control function.

(h) scikit monaco (Python)

$\left.\begin{array}{l}\text{gsl\_monte\_plain.h}\\ \text{gsl\_monte\_miser.h}\\ \text{gsl\_monte\_vegas.h}\end{array}\right\}$ C

(i) scipy.integrate.solve_bvp (Python)

(j) numpy.linalg.eig (Python.

3. $A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \cdots & & \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \cdots & & \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \cdots & & \\ 0 & 0 & a_{43} & a_{44} & a_{45} & 0 & \cdots & \\ & \cdots & & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1\,n} \\ 0 & 0 & & 0 & 0 & a_{nn-1} & a_{nn} \end{bmatrix} \begin{bmatrix} \\ \\ \\ \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$

Steps:

R_1/a_{11} → 3 ~~step~~ operations.

$-(R_1 \times a_{21}) + R_2$ → 3 'x' & 3 '-'.

∴ 9 operations followed by $\frac{R_2}{a_{22}'}$ → 3 → 12 operations for row 1 & 2

After these problem becomes:

$\begin{bmatrix} 1 & a_{12}' & 0 & 0 & \cdots \\ 0 & 1 & a_{23}' & \cdots & \\ & & \ddots & & \\ & & & & \end{bmatrix} \begin{bmatrix} b_1' \\ b_2' \\ \vdots \\ \end{bmatrix}$

Similarly for row 2 & 3 – 12 operations.

Thus $12 \times (n-1)$ such operations. After which matrix becomes:

$\begin{bmatrix} 1 & a_{12}' & 0 & 0 & \cdots \\ 0 & 1 & a_{23}' & 0 & \cdots \\ 0 & 0 & 1 & a_{34}' & \cdots \\ 0 & 0 & 0 & 1 & a_{45}' \cdots \\ \vdots & & & & \\ & & \cdots & 1 & a_{n-1,n}' \\ 0 & 0 & \cdots & 0 & a_n' \end{bmatrix} \begin{bmatrix} b_1' \\ b_2' \\ b_3' \\ b_4' \\ \vdots \\ b_{n-1}' \\ b_n' \end{bmatrix}$

Backward sub^n:

Then $x_n = b_n'/a_n'$

for each x_i,

$x_i = b_i' - a_{i-1,i}' \times x_{i-1}$

↓

2 operations.

Thus total no. of operation: $12(n-1) + 2(n-1) + 1$

$= 14(n-1)+1$ (roughly, there might be some error in counting but the order of magnitude of no. of operations is intact)

$= \boxed{O(n)}$

$12(n-1) \rightarrow$ row operations

$2(n-1) \rightarrow$ backward substitution.

4(c): $k_{max} = \pi\left(1-\frac{2}{1024}\right) \approx 3.1354$

$k_{min} = -\pi \approx -3.1416$

$x = 0, 1, 2, \ldots 1024$

$dx = 1.$

4(e) :

It was shown in class that Power spectrum of a data is the fourier transform of its correlation function. (time averaged).

Correlation function: $E[f_x(x_1) f_x(x_2)] = \begin{cases} R(x_1, x_2) & \text{if } |x_1|,|x_2| < X \\ 0, & \text{otherwise.} \end{cases}$

for unifor random distribution,

$f(x_1)$ & $f(x_2)$ are independent.

Thus $E[f_x(x_1) f_x(x_2)] = E(f(x_1)) E(f(x_2))$

$= 0.5 \times 0.5$ (f is uniform b/w $(0,1)$)

$= 0.25$

$=$ const

for any x_1, x_2.

Thus time-average is also independent of

$\xi = (x_1 - x_2)$

Now, fourier tramsform of constant function is a delta-function, which is what I get in Python.

Another way to see is that Power spectrum qualitatively gives the contribution of e^{ikx} to $f(x)$. But here since $f(x)$ is uniformly distributed only $k=0$ ~~distribute~~ contribute & hence power spectrum is a delta function.

5. Criteria to choose library:

i) Price & source code access: It might sound weird, but open-source libraries are always preferred above licensed ~~once~~ ones. They serve 2 advantages: They are free & they can be ~~modie~~ modified as per our requirement.

ii) Versatality: I would like to choose a library which work for quite general kind of problems. e.g. If I want a library for fourier tranform, I expect it to work for multidimensions. ~~I can~~

iii) Storage: The functions in library ~~is~~ generate various intermediate arrays & variables which need storage. A good library ~~have~~ require less storage for such arrays or variables.

iv) Speed: This is an obvious ~~chor~~ criteria. The faster the functions in library, the better.

v) Compatibility with language:- The library must be compatible with the language in which I do most of my coding work.

6. $\frac{d}{dx}(y_1 + 2y_2) = 32y_1 - 132y_1 + 66y_2 - 266y_2$

$= -100y_1 - 200y_2$

$= -100(y_1 + 2y_2)$

$\therefore y_1 + 2y_2 = e^{-100x}(y_1 + 2y_2)(0)$

$= e^{-100x}$

Thus $y_1 + 2y_2 \approx 0$ Since e^{-100x} rapidly goes to 0.

Thus $y_1 \approx -2y_2$, which is also observed in the obtained solution.

7. $x_{i+1} = (x_i \times a + c) \% m$

$a \to$ multiplier

$c \to$ increment

$m \to$ modulus.

Clearly all nos. lie bet^n 0 to $m-1$.

$x_0 \to$ seed.

i) Seed repeats:

e.g. $a = 17$
$c = 0$
$m = 32$
$x_0 = 1$.

Sequence: [1 17 33 49 65 81 97 113 1 17 ..]

Thus we see that the seed reappears.

ii) Seed does not ~~rep~~ reappear:
This can occur if we get $x=0$ somewhere along the chain, in that case the sequence gets stuck to c.
Or if some value other than seed appears again & hence seed cannot appear again.

1st kind of e.g. is $a=2$, $c=0$, $m=2^y$, $x_0=1$.
In that case $x_0=1$, $x_1 = (2\times 1)\% 2^y = 2$, $x_2 = 4$, $x_3 = 8$
..... $x_y = 0$, $x_{y+1} = 0$ ...,

Thus we never get 1 again. Though this is very bad generator.

Another e.g. is $a = even$
$c = even$
$m = even$

& $x_0 = odd$.

In that case $x_1 = (even \times odd + even) \bmod even$
$= even$

Thus every new no. will be even. Thus seed never appear again.