Computational Physics
final Exam

Name: Rounak Chatterjee
Dept: DNAP
mail: rexrounak97@gmail.com
rounak.chatterjee@tifr.res.in

1> Gauss Elimination with a computer of only two floating point digit. Hence when ever there are more number of digits than permitted by the computer, we will round it off via standard rounding off rules.

Given $Ax = b$, where

$$A = \begin{pmatrix} 4 & 1 & 2 \\ 2 & 4 & -1 \\ 1 & 1 & -3 \end{pmatrix}, \quad b = \begin{pmatrix} 9 \\ -5 \\ -9 \end{pmatrix}$$

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

We can write the rectangular joint Matrix $(A|b)$ and apply row reduction followed by back substitution to solve the equation.

$$\therefore (A|b) = \left(\begin{array}{ccc|c} 4 & 1 & 2 & 9 \\ 2 & 4 & -1 & -5 \\ 1 & 1 & -3 & -9 \end{array}\right)$$

$R_1 \longrightarrow R_1/4$

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.30 \\ 2 & 4 & -1 & -5 \\ 1 & 1 & -3 & -9 \end{array}\right)$$

$R_2 \to R_2 - 2R_1$, $R_3 \to R_3 - R_1$

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.30 \\ 0 & 3.5 & -2.0 & -9.5 \\ 0 & 0.75 & -3.5 & -11.3 \end{array}\right)$$

$R_2 \rightarrow R_2 / 3.50$

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.3 \\ 0 & 1 & -0.57 & -2.7 \\ 0 & 0.25 & 3.5 & -11.3 \end{array}\right)$$

$R_3 \rightarrow R_3 - 0.75 R_1$.

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.3 \\ 0 & 1 & -0.57 & -2.7 \\ 0.75 & -3.5 & & -11. \end{array}\right)$$

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.3 \\ 0 & 1 & -0.57 & -2.7 \\ 0 & 0 & -3.1 & -9.2 \end{array}\right)$$

$R_3 \rightarrow R_3 / -3.1$

$$= \left(\begin{array}{ccc|c} 1 & 0.25 & 0.50 & 2.3 \\ 0 & 1 & -0.57 & -2.7 \\ 0 & 0 & 1 & 3.0 \end{array}\right)$$

Hence we do back substitution as;

$x_3 = 3.0$.

$x_2 - 0.57 x_3 = -2.7$

$\Rightarrow x_2 = -2.7 + 0.57 \times 3.0$

$= -0.99$

$x_1 + 0.25x_2 + 0.5x_3 = 2.3$.

$\Rightarrow x_4 = 2.3 - 0.25(-0.99) - 0.5(3.0)$.

$\Rightarrow x_4 = 1.0$.

$\therefore (x_4, x_3, x_3)^T = (1.0, -0.99, 3.0)^T$

(Ans).

This is the solution to the problem.

2) Packages to Use.

Since I prefer using Python the most, I have made an attempt to quote the relevant python libraries useful for the following usages.

<a> For Fourier Transform the best package is numpy.fft, it can compute

1D → numpy.fft.fft & ifft

2D → numpy.fft.fft2 & ifft2

or nD → numpy.fft.fftn & ifftn.

<b> Computing QR decomposition

numpy.linalg.qr(Matrix)

is a function that can compute QR decomposition

<c> Million lognormal distributed numbers

numpy.random.lognormal(mean=0.0, sigma=1.0, size = 1e6)

d) Solving intial value problem using explicit Runge Kutta 8th order algorithm:

scipy.integrate.DOP853()

We can also use.

scipy.integrate.solve_ivp()

but we must give in parameter method = "DOP853" to make it solve using explicit RK8.

e) The Singular value decomposition can be done by numpy method.

numpy.linalg.svd(matrix)

f). To sample from a 548-dimensional PDF, we can do it by first sampling it from a standard normal and then using that distribution, we can sample other PDFs, the function is

numpy.random.Generator.multivariate_normal().

where we can input a 546 dimension mean to sample 546 dimensional random points.

g) Even though there is no explicit description of the fact that the ~~solve~~ following mode uses Adaptive step size, but experience says that it does.

The module is:

scipy.integrate.solve_ivp().

Until dense = True is not set, the module computes an ODE using adaptive step control.

h) The package that can do this very easily is called

~~SciKit~~ SciKit-monaco.

The name of the function is:

Skmonaco.mcquad()

This can do a definite integral using uniform sampling based monte carlo on an n-dimensional function.

i) To solve a coupled system of Boundary value problems, the package that we can use is:

scipy.integrate.solve_bvp().

j) For a general 10x10 matrix, complex where eigen values ~~are~~ to and vectors are to be computed, we use the function.

numpy.linalg.eig().

3) If A is a bi diagonal Matrix and we're to solve the equation

$$Ax = b, \quad x = (x_1 \cdots x_n)^T \text{ and } b = (b_1 \cdots b_n)^T$$

Now to figure out the complexity, we must see that when we're executing a Gauss elimination procedure, then How many steps we're actually executing per ~~pari~~ step in the n-step loop.

Now $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & 0 & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \cdots \\ 0 & a_{32} & a_{33} & a_{34} & \cdots 0 \cdots \\ & \ddots & \ddots & \ddots & 0 \\ 0 & & \ddots & \ddots & \ddots \\ & & & a_{n\,n-1} & a_{nn} \end{bmatrix}$.

So, if we were to solve Gauss elimination on this the conjoined Matrix will be written as;

$$\left[\begin{array}{cccccc|c} a_{11} & a_{12} & \cdots & 0 & \cdots & & b_1 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \cdots & & \\ 0 & a_{32} & a_{33} & a_{34} & \cdots & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & & a_{nn-1} & a_{nn} & & b_n \end{array}\right]$$

step 1: $R_1 \to \frac{R_1}{a_{11}}$, $R_2 \to R_2 - a_{21} \cdot R_1$.

which will yield ..

$$\left[\begin{array}{cccc|c} 1 & a_{12}/a_{11} & \cdots & 0 \cdots & b_1/a_{11} \\ 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11}} & a_{23} & 0 \cdots & b_2 - b_1/a_{11} \\ & \ddots & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & \end{array}\right]$$

$$= \left[\begin{array}{cccc|c} 1 & a_{12}' & \cdots & & b_1' \\ 0 & a_{22}' & a_{23}' & \cdots 0 & b_2' \\ 0 & a_{32} & a_{33} & a_{34} & b_3 \\ & & & & \\ & 1 & & & \end{array}\right]$$

now if we take the next block and apply same steps as:

$$R_2 \longrightarrow \frac{R_2}{a_{22}'} \qquad R_3 \rightarrow R_3 - a_{32} R_2$$

we will again get.

$$\left[\begin{array}{cccc|c} 1 & a_{12}' & 0 \cdots & \cdots & b_1' \\ & 1 & a_{23}' & \cdots & b_2'' \\ & 0 & a_{33}' & a_{34} & b_3' \\ & & \vdots & & \vdots \\ & & & & \end{array}\right]$$

Thus if we design the computer code / Algorithm to only consider the computation ~~of dividing~~ on non-zero elements, then we can easily follow that per ~~step to~~ step we only need a fixed number of computations, for example for the $i = 1$ step we need 3 divisions ~~and~~ 3. multiplications and only ~~two~~ three subtraction (because the element above a_{23} it's zero so no computation needed).

In this way we need only 9 computation for $(n-1)$ steps and only 3 computation for the last step which is just ~~div~~ three divisions to make the last diagonal element unity, so for these steps we need steps $= 9(n-1) + 3$

$= 9n - 6$ steps.

finaly for back substitution we will have.
the last ~~sol~~ step as.

$$x_n = b_n'$$

while other ~~steps~~ will have form

$$x_i + a_{i,i+1}' x_{i+1} = b_i' \quad i = 1, \ldots n-1.$$

$$\Rightarrow x_i = b_i' - a_{i+1}' x_{i+1}.$$

which is two computations per ~~step~~ step

So total computations

$$= 2(n-1)$$

plus total no. of steps for tridiagonal matrix is:

$$3n-6 + 2n-2 = 6 \; || \; n-8.$$

If we write this in the O-notation

Then $O(19n-8) \approx O(n)$.

This shows that solution of a Tridiagonal Matrix is just an $O(n)$ computation provided we design ~~a~~ computer algorithm to neglect any calculation that includes ~~zero elements~~ the elements that don't fall in the tridiagonal matrix set.

∴ Full $Ax = b$ solve $\Rightarrow O(n)$ process.

4>e> To Justify the above result, we will refer to Wiener Khinchin Theorem. This Theorem says that the power spectral density of a data distribution is equal to its averaged two point correlation function, we can write this as.

$$P(k) = \int_{-\infty}^{\infty} d\xi \left[\lim_{x \to \infty} \frac{1}{2x} \int_{-x}^{x'} dx_1 \, R(x_1, x_1 + \xi) \right] \exp(-ik\xi)$$

where $R(x_1, x_1 + \xi)$

$$= E\left[f_x(x_1) \, f_x(x_1 + \xi) \right]$$

where $f_x(x)$ is a restricted function given as:

$$f_x(x) = \begin{cases} f(x), & x \leq x \\ 0, & x > x \end{cases}$$

This is done to make function follow Dirichlet criterion.

Now for our case $f(x) = x$ and the process is isotropic.

Then we can write.

$$\lim_{x \to \infty} \frac{1}{2x} \int_{-x}^{x} dx_1 \, R(x_1, x_1 + \xi) = R(\xi)$$

and since $R(\xi)$ is compiled on a function $f(x) = x$ over an uniform distribution will quite appear.

Net $R(\xi) = 1$.

Hence we will have:

$$P(k) = \int_{-\infty}^{\infty} dx \, R(\xi) \, e^{-ik\xi} \, d\xi$$

$$P(k) = 2\pi \, \delta(k)$$

Thus this immediately tells us that the power spectral density must be a delta function; and higher the number of samples, the power spectral density will be a dirac delta function, centered around $k = 0$, which justifies the result of the plot.

5) Even though there are no hard and fast criteria to choose from while deciding to use a software, there are a few points that I can come up with to categorize the softwares.

(i) We must be well aware of the capibilities of the software and how much it is relevant to the kind of work we're seeking to undertake. Many softwares specialize in different categories and we must make sure to look for these specialities and consider them. For example if our scientific work needs image processing, MATLAB is a very powerful software with well established libraries that can do good image processing.

(ii) The flexibility of the package is a very important factor to look into. Flexibility in the sense that how much compatible it is to handle

various kinds of data, the usage of its various modules, their ease of use and relevance etc; A well documented and flexible library makes the scientific work easier making us invest lesser time in hardcore computation issues and maximizing our time investment on the Physics at hand. One of the best example is Python, which due to its very flexible code structure is very easy to use and operate.

(iii) Precision of out put → Some times scientific work require very precise data handelling capabilities and large data handelling capabilities also. We must examine whether the software handles precision, error propagation and mitigation upto an acceptable level. Also we must check whether the random number generators are random enough. All these go into deciding in the choice of the software. In recent years most software handle these very well.

(iv) One major point is the consideration of cost. Like every useful thing, the judgement boils down to the cost of the software also and more importantly the value for money.

(v) Finally the major concerns come under the tag of speed and compatibility. Mostly a well debugged library made available for most common environments are desirable.

And also the fact that it has good enough algorithms that are optimized to get most effective speed of computation.

67. Justification of Solution.

Computationally we have obtained y_1 and y_2 but How do we deduce it is correct?

If we look closely at the equations

$$y_1' = 32y_1 + 66y_2 + \tfrac{2}{3}x + \tfrac{2}{3} \longrightarrow (1)$$

$$y_2' = -66y_1 - 133y_2 - \tfrac{1}{3}x - \tfrac{1}{3} \longrightarrow (2)$$

multiplying 2 with eqn: (1) and adding with eqn (2) yields:

$$2y_1' + y_2' = -2y_1 - y_2 + (x+1)$$

$$= (2y_1 + y_2)' + (2y_1 + y_2) = (x+1)$$

let us denote: $y = 2y_1 + y_2$

$\Rightarrow y' + y = (x+1)$

This is a linear ODE in y

where $y(0) = 2y_1(0) + y_2(0)$

$= \tfrac{2}{3} + \tfrac{1}{3} = 1.$

using this we can find.

$$y(x) = \exp(-x) + x.$$

Thus what we can do is plot.
the sum of $2y_1(x) + y_2(x)$ vs x computed numerically and compare with $y(x) = \exp(-x) + x$

A graph doing this thing is done in the code and one can very well observe that they match exactly telling us that the solution obtained is justified.

7) We know that a Linear congruential Pseudo Random number generator is given by the form

x_0 => seed , then

$$x_{i+1} = (a x_i + c) \bmod m$$

where x_0 = seed a = multiplier

c = increment and m = modulus

This is a common method to generate ~~Random~~ numbers provided the constants are chosen ~~this~~ well enough.

(a) For a chosen generator of type.

$a = 2$ $c = 1$ $m = 5$ $x_0 = 1$

$x_0 = 1$

$x_1 = (2 \cdot 1 + 1) \bmod 5 = 3 \bmod 5 = 3$.

$x_2 = (2 \cdot 3 + 1) \bmod 5 = 7 \bmod 5 = 2$

$x_3 = (2 \cdot 2 + 1) \bmod 5 = 5 \bmod 5 = 0$

$x_4 = (2 \cdot 0 + 1) \bmod 5 = 1 \bmod 5 = 1 = x_0$.

Thus the sequence will go as:

$\{1, 3, 2, 0, 1, \ldots\}$

This is an example of a congruential generator that repeates its seed after some time.

(b) One of the Best Congruential ~~random~~ generator used to produce ~~pseudo~~ pseudo random numbers have the following parameters

$x_0 = 1$, $a = 1103515245$. } → one way to think!
$c = 12345$, $m = 2^{31}$. }

This is a very powerful pseudo random no. of generator using Congruential generator where the seed $x_0 = 1$ hardly ever returns.

To depict its power we have written a ~~pby th~~ python code that runs it for a sufficiently large number of iterations, to see if $x_0 = 1$ was ever obtained back. Another way of thinking is that the Congruent generator is locked in a constant loop such that it can never get out of it. In that case it will never come back.

say we choose.

$x_0 = 3$ $a = 2$ $c = 0$ $m = 6$

$\therefore x_0 = 3.$

$x_1 = 2 \cdot 3 \bmod 6 = 6 \bmod 6 = 0.$

$x_2 = 2 \cdot 0 \bmod 6 = 0 \bmod 6 = 0$

The sequence goes as:

$$\{3, 0, 0, \ldots\}$$

Thus the cycle is locked into a single cycle and it literally can never come back to its seed value $x_0 = 3$.