Scientific Computation Project 2

02216531

December 9, 2024

Part 1

1.

(a) We are given the system of n ODEs as

$$\begin{aligned}
\frac{\mathrm{d}x_0}{\mathrm{d}t} &= x_0\left(1 - x_0 - \gamma\left(x_{n-2} + x_1\right) + \mu x_{n-3}\right), \\
\frac{\mathrm{d}x_1}{\mathrm{d}t} &= x_1\left(1 - x_1 - \gamma\left(x_{n-1} + x_2\right)\right), \\
\frac{\mathrm{d}x_i}{\mathrm{d}t} &= x_i\left(1 - x_i - \gamma\left(x_{i-2} + x_{i+1}\right)\right), \quad i = 2, 3, \cdots, n-2, \\
\frac{\mathrm{d}x_{n-1}}{\mathrm{d}t} &= x_{n-1}\left(1 - x_{n-1} - \gamma\left(x_{n-3} + x_0\right)\right).
\end{aligned}$$

This is represented by the function `ODEs` in `part1q1a`.

To find a non-trivial equilibrium solution $\bar{\mathbf{x}}$ for the system, we set all derivatives to be zero and find the solution. This is done with `scipy.optimize.root`.

At the equilibrium point $\bar{\mathbf{x}}$, we apply a perturbation $\tilde{\mathbf{x}}$ so that $\mathbf{x} = \bar{\mathbf{x}} + \tilde{\mathbf{x}}$. Since $\tilde{\mathbf{x}}$ is much smaller than $\bar{\mathbf{x}}$, we neglect all higher order terms of $\tilde{\mathbf{x}}$ and linearise the resulting system to get

$$\frac{\mathrm{d}\tilde{\mathbf{x}}}{\mathrm{d}t} = \mathbf{J}\tilde{\mathbf{x}}, \tag{1}$$

where the matrix $\mathbf{J}$ is the Jacobian, with its (i, j)-th element $J_{i,j}$ defined as

$$J_{i,j} = \frac{\partial f_i}{\partial x_j},$$

where f_i is the i-th component of the RHS of the system.

We therefore have

$$\begin{aligned}
J_{0,0} &= 1 - 2x_0 - \gamma(x_{n-2} + x_1), \\
J_{0,n-2} = J_{0,1} &= -\gamma x_0, \quad J_{0,n-3} = \mu x_0, \\
J_{1,1} &= 1 - 2x_1 - \gamma(x_{n-1} + x_2), \\
J_{1,n-1} = J_{1,2} &= -\gamma x_1, \\
J_{i,i} &= 1 - 2x_i - \gamma(x_{i-2} + x_{i+1}), \\
J_{i,i-2} = J_{i,i+1} &= -\gamma x_i, \quad i = 2, 3, \cdots, n-2, \\
J_{n-1,n-1} &= 1 - 2x_{n-1} - \gamma(x_{n-3} + x_0), \\
J_{n-1,n-3} = J_{n-1,0} &= -\gamma x_{n-1}.
\end{aligned}$$

All other elements of $\mathbf{J}$ are zero.

We first use `np.linalg.eig` to find the eigenvalues and eigenvectors of $\mathbf{J}$ as λ_i and $\mathbf{v}_i$. Assuming that the initial perturbation $\tilde{\mathbf{x}}(0)$ is in the form of

$$\tilde{\mathbf{x}}(0) = \sum_i c_i \mathbf{v}_i,$$

with some coefficients c_i, the solution of (1) could be written as

$$\tilde{\mathbf{x}}(t) = \sum_i c_i \mathrm{e}^{\lambda_i t} \mathbf{v}_i.$$

Over time, the term with the largest eigenvalue will dominate, so we have

$$\tilde{\mathbf{x}}(t) \approx c_{\max} \mathrm{e}^{\lambda_{\max} t} \mathbf{v}_{\max}.$$

Hence, the perturbation energy at time t is given by

$$e(t) = \tilde{\mathbf{x}}(t)^T \tilde{\mathbf{x}}(t) \approx c_{\max}^2 \mathrm{e}^{2\lambda_{\max} t} \mathbf{v}_{\max}^T \mathbf{v}_{\max}.$$

Now we proceed to calculate $e(t = T)/e(t = 0)$ as

$$\frac{e(T)}{e(0)} \approx \frac{c_{\max}^2 \mathrm{e}^{2\lambda_{\max} T} \mathbf{v}_{\max}^T \mathbf{v}_{\max}}{c_{\max}^2 \mathbf{v}_{\max}^T \mathbf{v}_{\max}} = \mathrm{e}^{2\lambda_{\max} T}.$$

We can now proceed from here and let the code do the rest. For the initial perturbation vector, we simply used the largest eigenvector.

(b) We now proceed to the simulation part. We first substitute $n = 19, \gamma = 1.2, \mu = 2.5$ and $T = 50$ into `part1q1a` we just finished to compute the equilibrium solution $\bar{\mathbf{x}}$ and the maximum energy ratio r_a. We then run the simulation and get the solution $\mathbf{x}$. Since we have $\mathbf{x} = \bar{\mathbf{x}} + \tilde{\mathbf{x}}$, the perturbation energy $e(t) = \tilde{\mathbf{x}}(t)^T \tilde{\mathbf{x}}(t) = ||\tilde{\mathbf{x}}(t)||^2 = ||\bar{\mathbf{x}} - \mathbf{x}||^2$. The energy ratio r_{sim} can then be determined from here.

By running the code, we found that $r_a \approx 167.2$ and $r_{\text{sim}} \approx 10$. There is a large difference, but they are still roughly on the same order of magnitude, showing consistency. *However, this still indicates that there might be some errors in my code that potentially lead to the difference. Unfortunately, I tried my best and still could not detect them.*

(c) With $n = 19, \gamma = 2$ and $\mu = 0$, we loop over all values of T between 1 and 50. Within each loop, we calculate the solution $\mathbf{x}$ with `simulate1` and use `part1q1a` to compute the equilibrium solution $\bar{\mathbf{x}}$. From here, we can calculate the energy ratio just like what we did in `part1q1b`.

We plot the maximum energy ratio as a function of T and get the following graph.

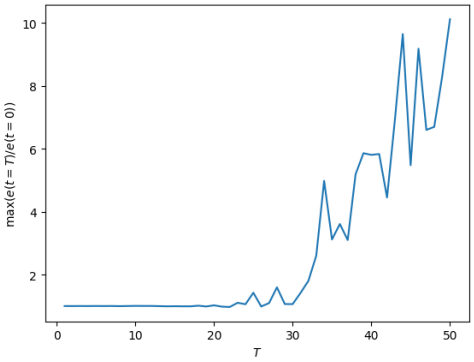

Figure 1: $\max(e(t = T)/e(t = 0))$ with respect to T.

We noticed that $\max(e(t = T)/e(t = 0))$ remains almost constant for lower values of T, around $0 < T \leq 20$. After a certain threshold ($T \approx 20$), the ratio begins to increase gradually. Beyond $T \approx 30$, the ratio starts to exhibit significant growth, showing sharp increases and fluctuations.

This probably indicates some non-linear or even chaotic behaviour associated with the model system as time goes by.

2.

We now set $\gamma = 1$ and $\mu = 0$. For the three cases $n = 9, 20, 59$, we run the simulation with final time $tf = 100, 250, 500$ respectively. We then discard the transient dynamics, which we shall set as the first 25%. For each case, we have created a time series plot and a contour plot respectively as shown below. We also take note of the mean μ and standard deviation σ of the results, which we shall later use to calculate the coefficient of variation (CV), defined as CV $= \frac{\mu}{\sigma}$. By comparing CV across different cases, we could examine and analyse the behaviour and stability of dynamics from a quantitative perspective.

We first present the relevant plots in each case. In the first case, we have $n = 9$, and we should expect some stable behaviour, likely in the form of simple sinusoidal oscillations. As shown in the plots, the solutions tend to be periodic and the components are synchronised, matching with our intuition. Chaotic dynamics is not present in this case.

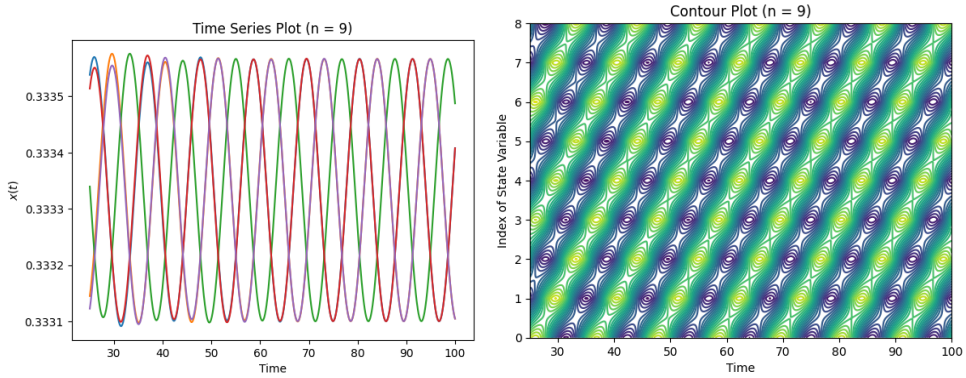

Figure 2: Time series plot and contour plot for $n = 9$.

In the second case, we have $n = 20$, and due to a larger number of ODEs in the system, the periodicity is lost and the solutions first stay stable, then evolve exponentially, and after a certain time the growth slows down. The time-dependence is shown even more obviously in the contour plot. Some chaotic behaviour is observed, but the solutions generally remain in control.

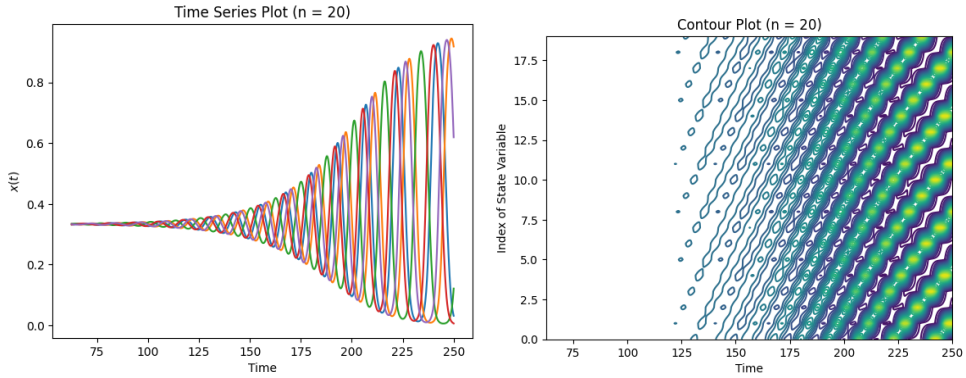

Figure 3: Time series plot and contour plot for $n = 20$.

In the third case, we have $n = 59$. With such a large system, we shall finally expect some chaos being present. Indeed as shown in the plots in the follwing page, the solutions quickly become wild and grow in a unpredictable fashion.

After some qualitative analysis with plots, we present some quantitative discussion based on CV.

```
Quantitative Discussion and Analysis:
n = 9: Coefficient of Variation = 0.0004944785125579439
- Dynamics appear stable and synchronised.
n = 20: Coefficient of Variation = 0.49988056775499284
- Dynamics exhibit oscillatory behaviour.
n = 59: Coefficient of Variation = 0.8775825860814006
- Dynamics become complex and chaotic.
```

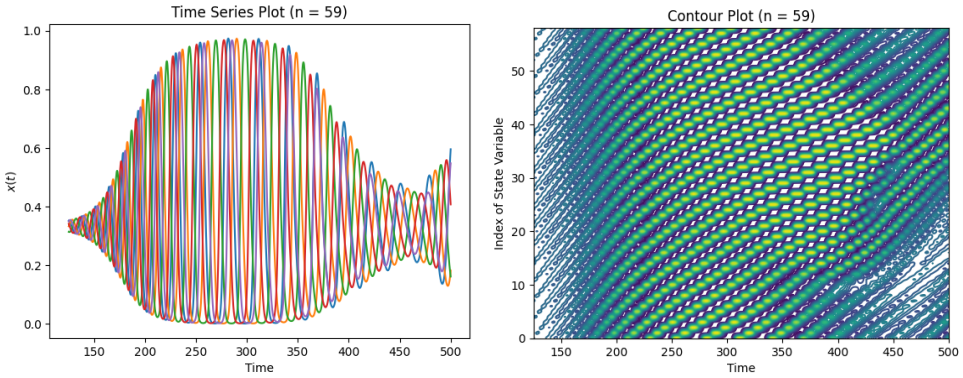

Figure 4: Time series plot and contour plot for $n = 59$.

Part 2

1.

In this question, an implicit finite difference method to compute the first and second derivatives of u is given. We start by writing out the equations.

For $j = 0$, we have

$$u'_{i,j} + 2u'_{i,j+1} - hu''_{i,j+1} = \frac{1}{h}\left[-3.5u_{i,j} + 4u_{i,j+1} - 0.5u_{i,j+2}\right], \tag{1}$$

$$-6u'_{i,j+1} + h\left(u''_{i,j} + 5u''_{i,j+1}\right) = \frac{1}{h}\left[9u_{i,j} - 12u_{i,j+1} + 3u_{i,j+2}\right]. \tag{2}$$

For $j = n - 1$, we have

$$u'_{i,j} + 2u'_{i,j-1} + hu''_{i,j-1} = -\frac{1}{h}\left[-3.5u_{i,j} + 4u_{i,j-1} - 0.5u_{i,j-2}\right], \tag{3}$$

$$-6u'_{i,j-1} - h\left(u''_{i,j} + 5u''_{i,j-1}\right) = -\frac{1}{h}\left[9u_{i,j} - 12u_{i,j-1} + 3u_{i,j-2}\right]. \tag{4}$$

And for $j = 1, 2, \cdots, n - 2$, we have

$$7u'_{i,j-1} + 16u'_{i,j} + 7u'_{i,j+1} + h\left(u''_{i,j-1} - u''_{i,j+1}\right) = \frac{15}{h}\left(u_{i,j+1} - u_{i,j-1}\right), \tag{5}$$

$$9\left(u'_{i,j+1} - u'_{i,j-1}\right) - h\left(u''_{i,j-1} - 8u''_{i,j} + u''_{i,j+1}\right) = \frac{24}{h}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right). \tag{6}$$

In these equations, we have $h = 1/(n-1)$ by definition.

To compute $u'_{i,j}$ and $u''_{i,j}$ at $i = 0, 1, \cdots, n-1$ for some j, we consider wrapping them into the vector $\mathbf{x}_j = \left(u'_{0,j}, u''_{0,j}, u'_{1,j}, u''_{1,j}, \cdots, u'_{n-1,j}, u''_{n-1,j}\right)^T$. We then solve the matrix equation $\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$ to find $\mathbf{x}_j$.

Based on the equations given, the matrix $\mathbf{A}$ will be in the form of a $2n \times 2n$ banded matrix, specifically a pentadiagonal one. We implement this as a `scipy.sparse.dia_matrix`. Rows with even index represent the first derivative and rows with odd index represent the second derivative. Hence, we will consider the even and odd cases separately.

For simplicity, from here, we will use D, D_1, D_2, D_{-1} and D_{-2} to refer to the main diagonal, the first and second upper diagonal, and the first and second lower diagonal, respectively. $D_1[1]$, for example, is the second element (index starts from 0) in the first upper diagonal, which is A_{23} in conventional matrix notation. We use the equations given to determine the values of these diagonals.

1. When the row index is even, equations (1), (3) and (5) apply. Specifically, the coefficients of $u'_{i,j}, u'_{i,j+1}, u''_{i,j+1}, u'_{i,j-1}$ and $u''_{i,j-1}$ correspond to $D[j], D_1[j], D_2[j], D_{-1}[j-1]$ and $D_{-2}[j-2]$ respectively.

   From equation (1), we have $D[0] = 1, D_1[0] = 2$ and $D_2[0] = -h$.

   From equation (3), we have $D[2n-2] = 1, D_{-1}[2n-3] = 2$ and $D_{-2}[2n-4] = h$.

   From equation (5), we have $D[j] = 16, D_1[j] = D_{-1}[j-1] = 7, D_2[j] = -h$ and $D_{-2}[j-2] = h$ for all $j = 2, 4, \cdots, 2n-4$.

2. When the row index is odd, equations (2), (4) and (6) apply. Specifically, the coefficients of $u''_{i,j}, u'_{i,j+1}, u''_{i,j+1}, u'_{i,j-1}$ and $u''_{i,j-1}$ correspond to $D[j], D_1[j], D_2[j], D_{-1}[j-1]$ and $D_{-2}[j-2]$ respectively.

   From equation (2), we have $D[1] = h, D_1[1] = -6$ and $D_2[1] = 5h$.

   From equation (4), we have $D[2n-1] = -h, D_{-1}[2n-2] = -6$ and $D_{-2}[2n-3] = -5h$.

   From equation (6), we have $D[j] = 8h, D_1[j] = 9, D_{-1}[j-1] = -9, D_2[j] = -h$ and $D_{-2}[j-2] = -h$ for all $j = 3, 5, \cdots, 2n-3$.

To illustrate, let's take $n = 3$ as an example. We have $h = 1/(n-1) = 0.5$.
Then we have

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & -0.5 & 0 & 0 & 0 \\ 0 & 0.5 & -6 & 2.5 & 0 & 0 \\ 0.5 & 7 & 16 & 7 & -0.5 & 0 \\ 0 & -0.5 & -9 & 4 & 9 & -0.5 \\ 0 & 0 & 0.5 & 2 & 1 & 0 \\ 0 & 0 & 0 & -2.5 & -6 & -0.5 \end{pmatrix}.$$

After $\mathbf{A}$ is determined, we turn our attention to $\mathbf{b}_j$. This is already defined as in `dualfd1`, so we just used this part of code. We then solve the system $\mathbf{A}\mathbf{x}_j = \mathbf{b}_j$ with `spsolve` from `scipy.sparse.linalg` and extract `df` and `d2f` from here.

2.

By inspection of `fd2`, we could see that it computes first and second derivative with explicit finite difference method, and we know that `part2q1` is using implicit method.

To compare the two methods, we should consider the effect of different n values. In `part2q2`, we used $n = 10, 20, \cdots, 1000$ and set $m := n$ since we are given that m is comparable to n. We used a simple function $\sin x$, and we only considered its first derivative, $\cos x$, to test the accuracy and efficiency. For the accuracy part, we took the error between calculated and theoretical values into account, and we neglected the boundary part. As for efficiency, we used the `time.time` method to determine the time taken by each method. We then created plots to visualise the results.

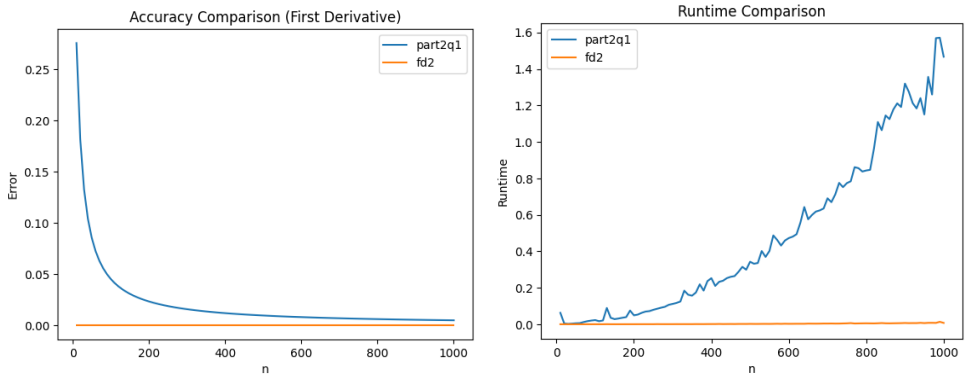

Figure 5: Accuracy and runtime comparison between `part2q1` and `fd2`.

From the plot, we can see that the error for `part2q1` is large when n is small but quickly decreases in a logarithmic fashion, indicating better stability. The runtime, however, increases roughly quadratic with n, suggesting a time complexity of around $\mathcal{O}(n^2)$. By checking out the implementation, we also see that the main loop consists of m times another loop of n times, giving $\mathcal{O}(nm)$ in total. On the other hand, from the implementation of `part2q1`, we could infer that the space complexity is $\mathcal{O}(n)$, since while the matrix is $2n \times 2n$, it is stored in a sparse format with diagonals only. In conclusion, implicit method is great in terms of both cost and efficiency for multiscale problems.

For `fd2` however, the error and runtime both seem to be constant and around 0. This probably indicates some errors in my idea and/or code; unfortunately, I could not detect them. By inspection of `fd2`, though, we could see that it does not utilise a loop anywhere, and the derivative is calculated in a direct way, so the time complexity should indeed be constant, $\mathcal{O}(1)$. I am not entirely sure if this is the correct approach to this part of the problem.