Coursework 1: MATH 3018/6141 - Numerical methods

Due: Thursday 21 November 2019

In this coursework you are to implement the numerical algorithms that are outlined and described below. The assessment is based on

- correct implementation of the algorithms – **4 marks**
- correct use of the algorithms in tests – **3 marks**
- figures to illustrate the tests – **2 marks**
- documentation of code – **3 marks**
- unit testing, robustness and error checking of code – **3 marks**.

The deadline is noon, Thursday 21 November 2017 (week 8). For late submissions there is a penalty of 10% of the total marks for the assignment per day after the assignment is due, for up to 5 days. No marks will be obtained for submissions that are later than 5 days.

Your work must be submitted electronically via Blackboard. Only the Python files needed to produce the output specified in the tasks below are required.

1 Stiff ODEs and transients

The ability to solve an Initial Value Problems (IVP) for $\mathbf{y}(x)$, of the form

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{1}$$

depends on the characteristic lengthscales involved in the problem. When the lengthscales differ by many orders of magnitude the system is called *stiff* and can be very difficult to solve numerically. In such stiff problems tiny errors in the large scale behaviour can swamp the correct transient, small scale behaviour, or vice versa. These difficulties can usually avoided by using *implicit* numerical solvers.

An example of such a stiff system is given by the ODE system

$$\frac{d}{dx}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} -1000y_1 \\ 1000y_1 - y_2 \end{pmatrix}, \qquad \mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \tag{2}$$

for which the solution is

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} e^{-1000x} \\ \frac{1000}{999}\left(e^{-x} - e^{-1000x}\right) \end{pmatrix}. \tag{3}$$

As shown by figure 1, initially y_2 jumps very rapidly from 0 to 1 before decaying very slowly. When such a stiff ODE is solved using a standard explicit algorithm it is often found that a very small step-length must be take in order to ensure stability of the result. In other words if too large step length is taken the result of the numerical method, based on the explicit algorithm, does not even come close to approximating the solution (because it is unstable). However the very small step length required by the explicit solver is a waste of computational time and so an implicit solver, which does not have these stability issues, is more appropriate.

Here we restrict our attention to equations of the form

$$\frac{d\mathbf{y}}{dx} = A\mathbf{y} + \mathbf{b}(x), \tag{4}$$

where A is an $n \times n$ matrix with constant coefficients and the vector $\mathbf{b}$ depends only on the independent variable x (and not on the solution $\mathbf{y}$). Since this equation is linear its discretisation, using an implicit approach, leads to an algorithm in which a linear system of equations (i.e. a matrix equation of the form $M\mathbf{u} = \mathbf{c}$) must be solved.

The example above in equation (2), is of the form

$$A = \begin{pmatrix} -a_1 & 0 \\ a_1 & -a_2 \end{pmatrix}, \qquad \mathbf{b} = \mathbf{0}. \tag{5}$$

With the initial data

$$\mathbf{y}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{6}$$

equation (5) has the solution

$$\mathbf{y} = \begin{pmatrix} e^{-a_1 x} \\ \frac{a_1}{a_1 - a_2}\left(e^{-a_2 x} - e^{-a_1 x}\right) \end{pmatrix}. \tag{7}$$

Stiff behaviour, characterised by an initial rapid transient followed by a slow decay, occurs for parameter values satisfying $a_1 \gg a_2 > 0$.

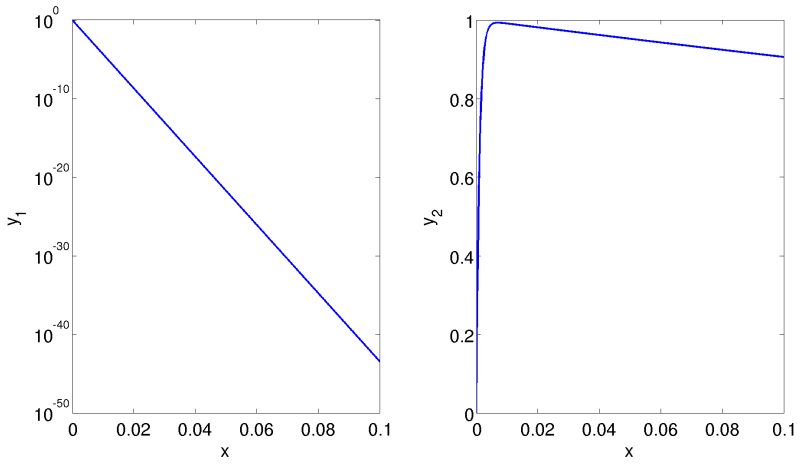

Figure 1: The solution for the system of equation (2) given by equation (3). Note the different scales required to show the rapid decay of y_1 and the slow decay of y_2. Note also the initial rapid transient behaviour of y_2.

2 Algorithms

2.1 Explicit algorithm

You are to implement the standard explict third order Runge-Kutta method, written RK3. Assume an evenly spaced grid with spacing h (so that $x_{n+1} = x_n + h$). Then, given that numerical approximation to $\mathbf{y}(x_n)$ is $\mathbf{y}_n$, the RK3 algorithm used to compute $\mathbf{y}_{n+1}$ (i.e the numerical approximation to $\mathbf{y}(x_{n+1})$) from the ODE (4) is

$$\mathbf{y}^{(1)} = \mathbf{y}_n + h\left[A\mathbf{y}_n + \mathbf{b}(x_n)\right], \tag{8a}$$

$$\mathbf{y}^{(2)} = \tfrac{3}{4}\mathbf{y}_n + \tfrac{1}{4}\mathbf{y}^{(1)} + \tfrac{1}{4}h\left[A\mathbf{y}^{(1)} + \mathbf{b}(x_n + h)\right], \tag{8b}$$

$$\mathbf{y}_{n+1} = \tfrac{1}{3}\mathbf{y}_n + \tfrac{2}{3}\mathbf{y}^{(2)} + \tfrac{2}{3}h\left[A\mathbf{y}^{(2)} + \mathbf{b}(x_n + h)\right]. \tag{8c}$$

where A is the matrix in (4).

2.2 Implicit algorithm

You are also to implement the optimal two stage third order accurate Diagonally Implicit Runge-Kutta method, written DIRK3. Once again assume an evenly spaced grid with spacing h (so that $x_{n+1} = x_n + h$). Then, given that numerical approximation to $\mathbf{y}(x_n)$ is $\mathbf{y}_n$, the DIRK3 algorithm used to compute $\mathbf{y}_{n+1}$ (i.e the numerical approxi-

mation to $\mathbf{y}(x_{n+1})$) from the ODE (4) is

$$[I - h\mu A]\,\mathbf{y}^{(1)} = \mathbf{y}_n + h\mu\mathbf{b}(x_n + h\mu), \tag{9a}$$

$$[I - h\mu A]\,\mathbf{y}^{(2)} = \mathbf{y}^{(1)} + h\nu\left[A\mathbf{y}^{(1)} + \mathbf{b}(x_n + h\mu)\right] + h\mu\mathbf{b}(x_n + h\nu + 2h\mu), \tag{9b}$$

$$\mathbf{y}_{n+1} = (1-\lambda)\mathbf{y}_n + \lambda\mathbf{y}^{(2)} + h\gamma\left[A\mathbf{y}^{(2)} + \mathbf{b}(x_n + h\nu + 2h\mu)\right], \tag{9c}$$

where I is the identity matrix, A is the matrix in (4) and the coefficients μ, ν, γ and λ are defined as

$$\mu = \tfrac{1}{2}\left(1 - \frac{1}{\sqrt{3}}\right), \tag{10a}$$

$$\nu = \tfrac{1}{2}\left(\sqrt{3} - 1\right), \tag{10b}$$

$$\gamma = \frac{3}{2\left(3+\sqrt{3}\right)}, \tag{10c}$$

$$\lambda = \frac{3\left(1+\sqrt{3}\right)}{2\left(3+\sqrt{3}\right)}. \tag{10d}$$

3 Task

1. Implement the explicit RK3 method given in equation (8) as a Python function

   `x, y = rk3(A, bvector, y0, interval, N)`

   The input arguments are the matrix `A` (as defined in (4)), a function `bvector`, the n-vector of initial data $\mathbf{y}_0 \equiv \mathbf{y}(x_0)$, a list `interval` giving the start and end values $[x_0, x_{\text{end}}]$ on which the solution is to be computed, and N the number of steps that the algorithm should take (so $h = (x_{\text{end}} - x_0)/N$). The function `bvector` should take the form

   `b = bvector(x)`

   where the input is the location x (which may vary inside the RK step), and the output is the vector $\mathbf{b}$ as defined in equation (4).

   The first output argument `x` is the locations x_j at which the solution is evaluated; this should be a real vector of length $N+1$ covering the required interval. The second output argument `y` should be the numerical solution approximated at the locations x_j, which will be an array of size $n \times (N+1)$.

   The input to the function should be carefully checked, and the function fully documented.

2. Implement the implicit DIRK3 algorithm given in equation (9) as a Python function

   `x, y = dirk3(A, bvector, y0, interval, N)`

The input and output arguments follow the same form as for the RK3 algorithm. Again the function should carefully check its input and should be fully documented. When solving the linear systems in equations (9a–9b) use the in-built `numpy` solvers.

3. Apply your RK3 and DIRK3 algorithms to the system of equations (5) with the explicit values $a_1 = 1000, a_2 = 1$ as in figure 1, using the initial data of equation (6) over the interval $x \in [0, 0.1]$. This will require defining a function to provide the (trivial) values of $\mathbf{b}$. Compute the solution for each N where $N = 40k$ with $k = 1, \ldots, 10$. Compute the 1-norm of the relative error in the component y_2 by comparing to the exact solution in equation (7) (for $x \neq 0$) as

$$\|\text{Error}\|_1 = h \sum_{j=2}^{N} \left| \frac{(y_2)_j - ((y_2)_{\text{exact}})_j}{((y_2)_{\text{exact}})_j} \right|. \tag{11}$$

Plot the error against h, using an appropriate scale. By fitting an appropriate curve to the data using `numpy.polyfit`, which should also be plotted, give evidence to show that the algorithm is converging at third order. In addition plot the solution computed with the highest resolution and the exact solution against x on appropriate scales (in one figure for each algorithm, but different variables in subplots).

4. Define a new system in which the matrix A is given by

$$A = \begin{pmatrix} -1 & 0 & 0 \\ -99 & -100 & 0 \\ -10\,098 & 9\,900 & -10\,000 \end{pmatrix} \tag{12}$$

and the vector function $\mathbf{b}$ by

$$\mathbf{b} = \begin{pmatrix} \cos(10x) - 10\sin(10x) \\ 199\cos(10x) - 10\sin(10x) \\ 208\cos(10x) + 10\,000\sin(10x) \end{pmatrix}. \tag{13}$$

With initial data

$$\mathbf{y}_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{14}$$

this system has the exact solution

$$\mathbf{y} = \begin{pmatrix} \cos(10x) - e^{-x} \\ \cos(10x) + e^{-x} - e^{-100x} \\ \sin(10x) + 2e^{-x} - e^{-100x} - e^{-10\,000x} \end{pmatrix}. \tag{15}$$

Apply both the RK3 and DIRK3 algorithms to this system over the interval $x \in [0, 1]$. Compute the solution for each N where $N = 200k$ with $k = 4, \ldots, 16$. Again plot the error against h, showing evidence of the convergence rate for DIRK3 only, and plot the solutions for both algorithms computed at the highest resolution. All components of $\mathbf{y}$ should be plotted against the exact solution, none on logarithmic scales. When computing the error, only the y_3 component should be used.

3.1 Summary and assessment criteria

You should submit the Python code electronically as noted above.

You are expected to submit all the Python code needed to produce the required output. Ideally there should be a top-level `Coursework1.py` script that, when run, produces all output. It is possible to complete the coursework by submitting a single file, but if you wish to use more files that is fine, provided all are submitted.

The script should produce 7 (seven) plots: for each system (the moderately stiff case in task 3 and the stiff case in task 4), and for each algorithm (RK3 and DIRK3), two plots are required. The first plot should show the behaviour of the errors against h and should be annotated to show the convergence rate where appropriate (this plot is not required for the RK3 algorithm in the stiff case). The second plot should show the exact and numerical solutions for each component on appropriate scales in individual subplots (e.g., as in figure 1). All plots should be suitably labelled and clear.

The primary assessment criteria will be the correct implementation of the RK3 and DIRK3 algorithms. The correctness of the algorithms must be clear, and must be demonstrated by completing the tests shown.

The secondary assessment criteria will be the clarity and robustness of your code. Your functions and scripts must be well documented with appropriate docstrings and internal comments describing inputs, outputs, what the function does and how it does it. The code should also be well structured to make it easy to follow. Input must be checked and sensible error messages given when problems are encountered. The clarity of the output (such as plots or results printed to the command window) is also important.

Code efficiency is not important unless the algorithm takes an exceptional amount of time to run.

For those interested in applications of these techniques to current problems, the review of Gottlieb et al. includes a broad range of examples, and the paper of Pareschi and Russo gives a specific example where DIRK methods are important.