Martin P. King

Computational Methods with MATLAB Examples and Exercises

A notebook of basic computational methods in example m-files, MATLAB exercises, and author's solutions

Table of Contents

Foreword....2
Exercise 1. 2D Laplace Equation (an elliptic PDE)....3
Exercise 2. 2D Poisson Equation (an elliptic PDE)....7
Exercise 3. 1D Time-dependent Heat Conduction Equation (a parabolic PDE)....11
Exercise 4. 2D Time-dependent Heat Conduction Equation (a parabolic PDE)....14
Exercise 5. ODE – Numerical order of accuracy....23
Exercise 6. ODE – Numerical order of accuracy and stability....27
Exercise 7. ODE – Lorenz Equations....32
Exercise 8. ODE – Eigenvalues and eigenvectors....38
Exercise 9. ODE – Boundary value problems....41
Exercise 10. Finite Element Method....44
Exercise 11. Interpolation....45
Exercise 12. Filters....46

Foreword

The materials covered in this notebook are computational methods commonly taught to second or third year undergraduates (in the UK and Australia systems) in engineering, physical sciences, and applied mathematics. Students normally have learned programming, simple linear algebra and numerical methods before they take a course of this type. A course like this is often an essential component of certain accredited engineering degrees and is also a foundational preparation before advancing to more complex or modern methods.

This is not a textbook and it does not provides much explanation of the methods, theories, and equations. Rather, it is a collection of computational lab exercises and many example codes, all in MATLAB. Example solutions are also provided immediately after the exercises. It may also be useful for looking up ‘classical’ computational methods to solve simple physical problems. The contents are based on courses for mechanical engineering students I taught some years ago.

The exercises involve modifying codes that are provided or writing your own codes, as well as perform some investigations on questions given in the exercises. Some information and tips are also given to help you complete the exercises. Whether you have taken a similar course or not, it is likely that you need to refer to a textbook or search for relevant information on the Web. I provide a list of selected references in *Bibliography*.

Of course you are encouraged to complete the exercises before looking at my suggested solutions. Plan your methods first, then implement them in codes, fix bugs, check the solutions, and then look for ways to further improve your codes.

Most of the methods I touch on are highly standard, widely taught and shared. I have also learned certain fancier MATLAB techniques from people who generously share their codes on the Web. I have not recorded the numerous and varied sources I learned from. I do not intentionally copy anyone’s works verbatim, but for the reasons I just mentioned, I do not claim originality in what are presented.

 You are welcome to send me any comment on martin.p.king@bath.edu. Enjoy!

Martin P. King

Bergen, Norway, 2022.

Exercise 1. 2D Laplace Equation (an elliptic PDE)

2D Laplace Equation (an elliptic PDE):

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0$$

Methods:

a) Inverse Laplacian

DT=B → T=**D**\B

Example codes: `laplace2d_fdm_direct_eg1.m`, `laplace2d_fdm_direct_eg2.m`

b) Iterative methods

- Jacobi, example code: `laplace2d_fdm_jacobi.m`
- Gauss-Seidel with SOR (Liebmann) (this exercise)

Exercise

This problem involves finding the steady solutions for the heat equation:

$$\frac{\partial T}{\partial t}=\frac{\partial T^2}{\partial x^2}+\frac{\partial T^2}{\partial y^2}.$$

In other words, by setting the time derivative to 0, you are required to solve:

$$\frac{\partial T^2}{\partial x^2}+\frac{\partial T^2}{\partial y^2}=0.$$

This is also referred to as the Laplace equation (if the RHS is non-zero; then it is called the Poisson equation).

The computational domain is a two-dimensional square plate measuring 2 units long on each side. At one edge T=6 and at all the other three edges T=0 are maintained. Initially, T=0 in the whole plate away from the edge.

a) Solve this problem by the iterative method of Gauss-Seidel with successive over-relaxation (SOR).

Write a MATLAB code to perform the computation. Your script should not need to be longer than 100 lines including the compulsory comments, although no marks will be deducted for exceeding this length.

b) Consider the following questions in your investigation: What is the value for T in the exact centre of the plate at steady state? How do you know if your answer is physically reasonable? What is the convergence criterion you use? What is the grid size you choose and why is that your choice? What is the SOR parameter (α), you choose, and why?

N.B. You may find referring to `laplace2d_fdm_jacobi.m`; `laplace2d_fdm_direct_eg1.m`; and `laplace2d_fdm_direct_eg2.m` helpful.

Useful formulas

The PDE here can be discretised as:

$$\frac{u_{i-1,j}-2u_{i,j}+u_{i+1,j}}{\Delta x^2}+\frac{u_{i,j-1}-2u_{i,j}+u_{i,j+1}}{\Delta y^2}=0.$$

If $\Delta x^2=\Delta y^2$, then

$$u_{i,j}=\frac{1}{4}\left(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}\right).$$

If $u_{i,j}$ is the current Gauss-Seidel estimate at the point (i,j) and $u_{i,j}^{old}$ is the previous estimate, the following 'updating' procedure may be implemented:

$$u_{i,j}^{new}=(1+\alpha)u_{i,j}-\alpha u_{i,j}^{old},$$

where $-1<\alpha<1$. If $\alpha=0$, then this is just the Gauss-Seidel iteration. If $\alpha<0$, then this is an under-relaxtion; if $\alpha>0$, then this is an over-relaxation.

Solutions

I realised that with minimal changes, I can immediately solve the PDE using `laplace2d_fdm_direct_eg1.m`. I set xmax=2, ymax=2, Thot=6 and found that the temperature at the centre of the plate at steady state is 1.5. I set dx=0.05 and rerun and obtain the same answer. My first instinct is that 1.5 is the exact answer. However, I am not yet solving this problem using Gauss-Seidel with SOR.

Since I know the Jacobi method is also an iterative method, I modify `laplace2d_fdm_jacobi.m` to obtain the following code (see `laplace2d_fdm_gssor.m`):

`laplace2d_fdm_gssor.m`

```
% sample solution for Lab 8, Q 1.
% by martin king, 31 Aug 2008
%------------------------------
% solves 2D Laplace Equation by SOR Gauss-Seidel iteration.
% 0=u_xx+u_yy in domain [0,2]x[0,2]
% bc's: one side u=6, 3 sides u=0.
% ic: u=0.

clear all; close all;
format long;


xmin=0.; xmax=2.0; ymin=0.; ymax=2.0;  % defining domain
```

```
dx=0.05; dy=0.05;  % grid spacing
x=[xmin:dx:xmax]; y=[ymin:dy:ymax];  % defining mesh
[X,Y]=meshgrid(x,y);
[nx,ny]=size(X);  % setting number of grid points

Thot=6.0; Tcold=0.;  % boundary values
wallhot=find(X==xmin);  % grid point indices for hot wall
wallcold=find(X==xmax|Y==ymin|Y==ymax);  % grid point indices for cold
wall

u=zeros(nx,ny);
u(wallcold)=Tcold; u(wallhot)=Thot;  %boundary conditions
uold=u;
alpha=0.9; %-1<alpha<1; negative for under-relaxation, positive for
over-relaxation

err=1;  %so that first iteration must run
iter=0;
while err>=0.5e-10
    for ix=2:nx-1
      for iy=2:ny-1
          u(iy,ix)=(1+alpha)*0.25*(u(iy,ix-1)+u(iy,ix+1)+u(iy-1,ix)
+u(iy+1,ix))-alpha*u(iy,ix);
      end
    end
err=norm(u-uold);
uold=u;
iter=iter+1
end

%
%plotting
        mesh(X,Y,u), axis([0 2 0 2 0. 6]), view(44,38);
        xlabel x, ylabel y, zlabel T
        colormap(1e-6*[1 1 1]);
        xmid=ceil(nx/2);ymid=ceil(ny/2);
        title(['u(0,0)=',num2str(u(xmid,ymid))]);
        umid=u(xmid,ymid)
%
```

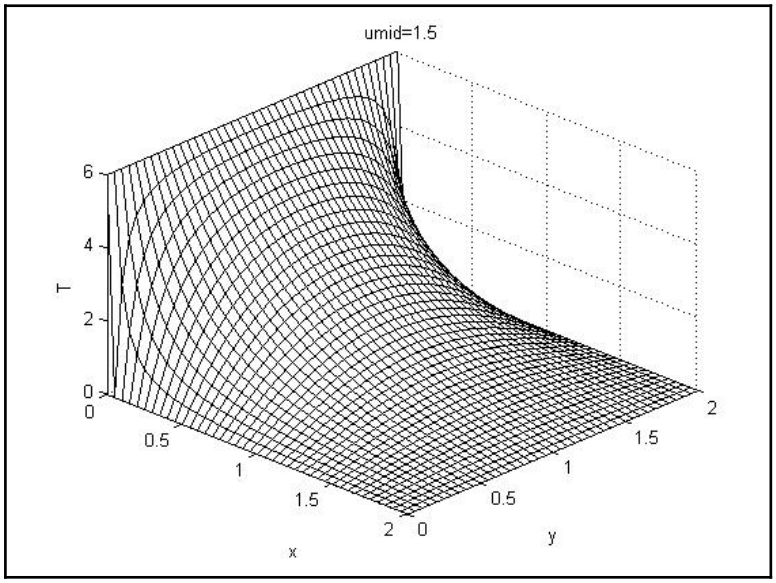
The two codes are essentially the same except what are inside the while loop. You can see that `u(iy,ix)` is immediately updated, and that I also use SOR.

I have set the stopping criterion as 0.5e-10, therefore my approximations have reached the 10th decimal place (although this doesn't mean that my final approximation is correct to the 10th decimal place when compared with the analytical answer). I increased α from 0.2 to more than 0.9 and found that at around 0.9, the smallest number of iterations are required to satisfy the convergence criterion. I chose dx=dy=0.05, as this seems to be a good compromise between accuracy and number of iterations to reach convergence. The suitable values for α and the grid sizes may be problem-dependent.

The answer I obtained with the above code is: 1.500000000002050. Besides that the direct method also gives me an answer which is very close to this, I noticed that this is also the average temperature of the 4 edges, i.e. (6+0+0+0)/4. A few experiments suggest to me that as long as I have a square plate of any side length with any uniform temperature on an edge, this kind of prediction is always true! This is a consequence of diffusion, which can be thought of as a Brownian motion problem. A point in the middle of the plate is equi-distanced from all sides. Therefore, the concentration of the species

diffused from all edges is the average concentration at the middle of the plate at steady state. Note also, that this must be correct because the concentration at any point is the average concentrations of the surrounding points.

Plotted figure:



Exercise 2. 2D Poisson Equation (an elliptic PDE)

Exercise

The governing equation for two-dimensional steady heat conduction with a heat source is

$$\frac{\partial T^2}{\partial x^2}+\frac{\partial T^2}{\partial y^2}+f(x,y)=0.$$

Here, we consider a heat source with a Gaussian-shaped spatial distribution

$$f(x,y)=\frac{A}{\sigma}\exp\left[\frac{-1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)\right]$$

applied at the centre of a square plate measuring 2 units long on each side. The temperatures T=5; T=3; T=2, T=0 are maintained at the 4 sides respectively. For $\sigma=0.2$, find the value of A necessary to maintain T=3.0 at the centre of the plate.

a) You are required to solve this problem by the Gauss-Seidel iterative method with successive over-relaxation (SOR). Write a MATLAB code to perform the computation.

b) Include also in your discussion on the following: What is the convergence criterion used? What is the grid size you choose and why? What is the SOR parameter (α) you choose and why?

Solutions

First I discretise the PDE with FDM.

$$\frac{T_{i-1,j}-2T_{i,j}+T_{i+1,j}}{\Delta x^2}+\frac{T_{i,j-1}-2T_{i,j}+T_{i,j+1}}{\Delta y^2}+f(x,y)=0.$$

If we use

$$\Delta x=\Delta y,$$

the above discretised equation can be rearranged to

$$T_{i,j}=0.25\left[T_{i-1,j}+T_{i+1,j}+T_{i,j-1}+T_{i,j+1}+f(x,y)\Delta x^2\right].$$

8

Even though the question asks us to do Gauss-Seidel with SOR, I want to solve this first in Jacobi method by modifying the provided m-file `laplace2d_fdm_jacobi.m` from the last exercise because this is a simpler task and will provide an answer to guide my further work. The resulted code I used is called `poisson2d_fdm_jacobi.m`.

By trial-and-error, I found that (using dx=dy=0.05)

A=1.539

will give T at the middle of the plate to be

T=3.000005008124929.

The relative error is about 1.7e-4% which is very small.

I then modified the m-file to perform Gauss-Seidel iteration (without SOR). The resulted m-file is named `poisson2d_fdm_gs.m`. Using the same value of A as above, I obtained

T=3.000005008528098.

The results are slightly different but have errors of same orders of magnitude. (Note the main changes from lines 40 to 48).

Next, I add the successive-over-relaxation part. The resulted m-file is called `poisson2d_fdm_gssor.m` (script shown below). Note lines 35 and 46. I also added `niter` in lines 38 and 51 to record the number of iterations required for convergence. From this, I found that if I used `alpha=0.9`, the calculation needed the least number of iterations.

For the Gauss-Seidel SOR method, I obtained

T=3.000005008931867

by using A=1.5329.

Note that the question has not specified how the 4 boundary conditions are arranged. I think there are 3 combinations. From

$$\frac{4!}{4\times 2}=3.$$

There is 4! because there are 4 different b.c.'s one for each side; there is the division by 4 because the square has rotational symmetry of in 90 degrees; there is the division by 2 because the square has mirror symmetry.

I have also coded in the three combinations of b.c's arrangements (see lines 19 to 26). Interestingly, as far as finding an A so that T in the middle is 3 is concerned, the answer is unaffected by which boundary case is used.

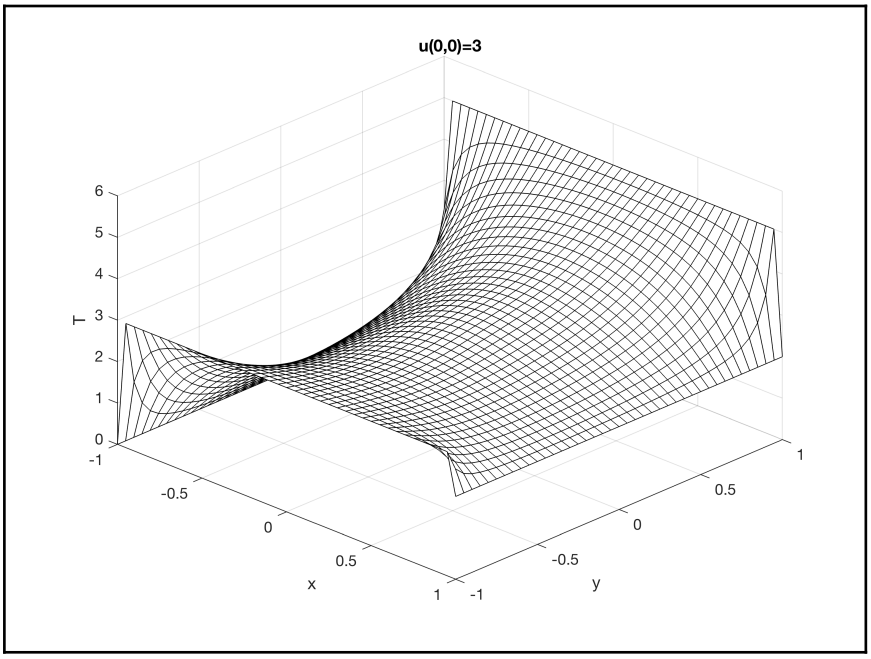
You should also notice that for f(x)=0, T in the middle is the average of the boundary values, i.e. T=(0+5+3+2)/4=2.5. If I set A=0., I find that this is indeed the result from the numerical iteration.

poisson2d_fdm_gssor.m

```
% sample solution m-file
% by martin king, 30 Sep 2009
%-----------------------------
% solves 2D Poisson Equation by Gauss-Seidel iteration with SOR.
% 0=u_xx+u_yy+f(x,y) in domain [0,2]x[0,2]
% bc's: u=5, u=3, u=2, u=0
% ic: u=0.

clear all; close all; clc;
format long;

xmin=-1.; xmax=1.; ymin=-1.; ymax=1.;  % defining domain
dx=0.05; dy=0.05;  % grid spacing
x=[xmin:dx:xmax]; y=[ymin:dy:ymax];  % defining mesh
[X,Y]=meshgrid(x,y);
[nx,ny]=size(X);  % setting number of grid points

T5=5.0; T3=3.0; T2=2.0; T0=0.0;  % boundary values
switch 3
    case 1
        wall5=find(Y==ymax); wall3=find(X==xmax); wall2=find(Y==ymin);
wall0=find(X==xmin);
    case 2
        wall5=find(X==xmax); wall3=find(Y==ymin); wall2=find(Y==ymax);
wall0=find(X==xmin);
    case 3
        wall5=find(Y==ymax); wall3=find(Y==ymin); wall2=find(X==xmax);
wall0=find(X==xmin);
end

u=zeros(nx,ny);
u(wall5)=T5; u(wall3)=T3; u(wall2)=T2; u(wall0)=T0;  % boundary
conditions
uprev=u;

f=@(x,y,A) A/0.2*exp(-0.5*(x^2+y^2)/0.2^2); % defining a function

A=1.5329;
alpha=0.9;

err=1;  %so that 1st iteration must run
niter=0; %iteration counter
while err>=0.5e-10
%update=del2(u)+u;
%the above line does the same job as these nested loops:
for ix=2:nx-1
    for iy=2:ny-1
     u(ix,iy)=0.25*(u(ix,iy+1)+u(ix,iy-1)+u(ix-1,iy)+u(ix+1,iy)+ ...
         A/0.2*exp(-0.5*(x(ix)^2+y(iy)^2)/0.2^2)*dx^2);
     u(ix,iy)=(1+alpha)*u(ix,iy)-alpha*uprev(ix,iy);
    end
end
err=norm(u(2:end-1,2:end-1)-uprev(2:end-1,2:end-1));  %calculating
aprox error
uprev(2:end-1,2:end-1)=u(2:end-1,2:end-1);  %updating internal points
niter=niter+1
end
```

```
%
%plotting
        mesh(X,Y,u), axis([xmin xmax ymin ymax 0 6]), view(44,38);
        xlabel x, ylabel y, zlabel T
        colormap(1e-6*[1 1 1]);
        xmid=ceil(nx/2);ymid=ceil(ny/2);
        title(['u(0,0)=',num2str(u(xmid,ymid))]);
        umid=u(xmid,ymid)
%
```

Plotted figure for Case 3:



Exercise 3. 1D Time-dependent Heat Conduction Equation (a parabolic PDE)

1D Time-dependent Heat Conduction Equation (a parabolic PDE):

$$\frac{\partial T}{\partial t}=\frac{\partial^2 T}{\partial x^2}$$

This is more generally known as the diffusion equation.

Methods:

a) Explicit time-marching and 2nd-order central finite differencing in space.
Codes: `heat1D_fdm_explicita.m`, `heat1D_fdm_explicitb.m`.

b) Implicit time-marching and 2nd-order central finite differencing in space.
Code: `heat1D_fdm_implicit.m`.

c) Crank-Nicolson with the THETA schme. Code `heat1D_fdm_CN.m` and explanation below.

This chapter has no exercise. I provide four example m-files to solve this equation, and explanations for `heat1D_fdm_CN.m` below.

Explanation for `heat1D_fdm_CN.m`

This short note explains the solution strategy for the m-file. The aim is to solve the time-dependent 1D heat equation

$$\frac{\partial u}{\partial t}=\frac{\partial^2 u}{\partial x^2}.$$

We use the Crank-Nicolson method with the second-order central finite differencing. The THETA scheme is implemented too. After rearranging the discretised equation, we obtain the following (with $C=\Delta t/\Delta x^2$):

$$-C(1+\theta)T_{i+1}^{l+1}+(2+2C(1+\theta))T_i^{l+1}-C(1+\theta)T_{i-1}^{l+1}=\ldots$$
$$C(1-\theta)T_{i+1}^{l}+(2-2C(1-\theta))T_i^{l}+C(1-\theta)T_{i-1}^{l}$$

Up to line 31:

This part sets up the computational domain, values for necessary parameters, boundary conditions, timestepping parameters. Together with the comments, they should be self-explanatory.

Lines 33 and 34:

In line 33, I create an 'operator' matrix for the LHS, and also one for the RHS for the discretised equation above.

Lines 36 to 46:

Here is the for loop which does the timestepping. I first need to create the RHS by applying `TR=R*TR'` in line 39. Then in lines 40 to 43, for `TR(1)` and `TR(end)`, I add the boundary terms from both the LHS and RHS in the discretised equation above. Now I should have a complete RHS vector. I find T of the internal points in the next timestep by doing an inverse Laplacian line 45.

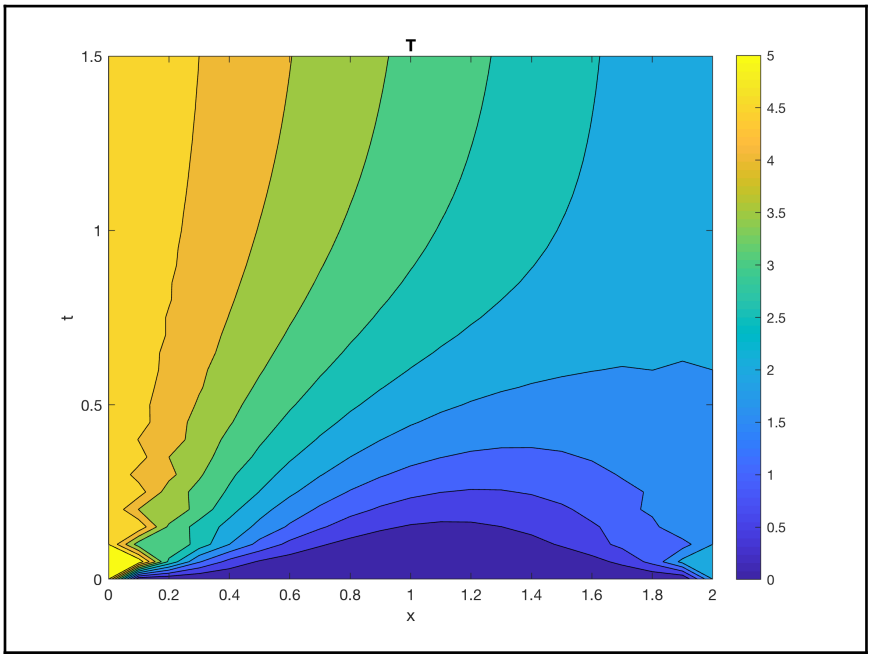
The default value for theta is set to 0 in line 15. Change it to -1 and then 0, and see what happen. The default C=5. For an explicit method to work, recall what is the stability condition you learned from the lectures.

heat1D_fdm_CN.m

```
%example m-file.
%finite differencing method for the 1D heat equation.
%solving dT/dt=d^2T/dx^2.
%crank-nicolson with 2nd order central differencing in space.
%with theta scheme.
%see also explanatory note.
%theta=1: implicit.
%theta=-1: explicit.
%theta=0: crank-nicolson.
%----------------------------------------
%

clear all, close all;

theta=0.;  %see above

%defining the spatial domain
xmin=0; xmax=2; dx=0.1; x=xmin:dx:xmax; nx=length(x);

%where C=dt/dx^2
C=5; dt=C*dx.^2;

Thot=5.0; Tcold=2.0;  %Dirichlet boundary values
%grid point indices for hot and cold ends:
hotend=find(x==xmin); coldend=find(x==xmax);

T=zeros(1,nx); T(hotend)=Thot; T(coldend)=Tcold; Tnew=T;  %initial T;
and applying b.c. too
tlast=1.5;  %final time
ntsteps=ceil(tlast/dt);  %number of timesteps
```

```
dt=tlast/ntsteps;  %re-correct dt
C=dt./dx.^2;  %re-correct C

L=toeplitz([2+2*C*(1+theta) -C*(1+theta) zeros(1,nx-4)]);  %operator
for LHS
R=toeplitz([2-2*C*(1-theta) C*(1-theta) zeros(1,nx-4)]);  %operator for
RHS

for istep=1:ntsteps  %time stepping
    Tstore(istep,:)=T;  %storing T to Tstore
    TR=T(2:end-1);  %setting rhs vector; only the internal points
    TR=R*TR';  %applying the RHS operator
    TR(1)=TR(1)+C*(1+theta)*T(1);  %add the b.c.'s for the implicit
(LHS) terms
    TR(end)=TR(end)+C*(1+theta)*T(end);
    TR(1)=TR(1)+C*(1-theta)*T(1);  %add the b.c.'s for the explicit
(RHS) terms
    TR(end)=TR(end)+C*(1-theta)*T(end);

    T(2:end-1)=L\TR;  %implicit euler's
end
Tstore(ntsteps+1,:)=T;  %T at tlast

%for plotting
t=[0:dt:tlast];
[xx,tt]=meshgrid(x,t);
contourf(xx,tt,Tstore),colorbar;
xlabel('x'),ylabel('t');
title('T')
```

Plotted figure:



Exercise 4. 2D Time-dependent Heat Conduction Equation (a parabolic PDE)

2D Time-dependent Heat Conduction Equation (a parabolic PDE):

$$\frac{\partial T}{\partial t} = a\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

Methods:

a) Implicit (Backward) Euler time-marching with 2nd-order central finite differencing in space. Code `diffusion2d_fdm_implicit.m` and explanatory note below.

b) Explicit (Forward) Euler time-marching with 2nd-order central finite differencing in space (this exercise).

c) Crank-Nicolson method. Code: Not available; but once you know how to do *a)* and *b)* above, this should be straightforward too.

This chapter is longer than the others. I first provide explanation for an m-file solving the equation with the implicit Euler time-marching method. And then an exercise to program a code for explicit Euler time-marching is given, followed by suggested solutions.

Explanation for diffusion2d_fdm_implicit.m

This short note explains the solution strategy for the above m-file. The aim of the m-file is to solve the 2D diffusion equation

$$\frac{\partial u}{\partial t} = a\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right).$$

The method used is the Second-Order Central Differencing for the spatial derivatives with Implicit (Backward) Euler timestepping (SOCDBE).

I aim to use MATLAB functions to simplify the task whenever possible, instead of doing the coding explicitly. For example, I don't create matrices and vectors manually using for loops, etc. The rules I give myself are that I don't use the MATLAB toolboxes and I have only one self-contained m-file.

Up to line 35:

This part sets up the computational domain, values for necessary parameters, boundary conditions, timestepping parameters. Together with the comments, they should be self-explanatory.

Lines 37 to 41:

The above PDE can be discretised for SOCDBE and then rearranged to become:

$$u_{i,j}^{(n+1)} - a(dt)\left[\frac{u_{i+1,j}^{(n+1)} - 2u_{i,j}^{(n+1)} + u_{i-1,j}^{(n+1)}}{\Delta x^2} + \frac{u_{i,j+1}^{(n+1)} - 2u_{i,j}^{(n+1)} + u_{i,j-1}^{(n+1)}}{\Delta y^2}\right] = u_{i,j}^{n}$$

Where (n+1) denotes values of *u* (unknown) for the next timestep, while n denotes the most current *u*'s. Lines 38 and 40 make a 2D Laplacian matrix (operating only on the internal points, since we don't need to solve *u* at the boundaries) which essentially does the second term in the LHS above. In line 41, we complete making an 'operating' matrix for the LHS by adding the first term on the LHS. I call this the **B** matrix. Note that `B` is a sparse matrix for MATLAB. You can visualise it by `spy(B)`.

For each timestep we need to solve for the vector **u** in **Bu=r**. We will do this in MATLAB by `u=B\r`. But first I need to make the **r** vector.

Lines 43 to 53:

Before I worry about the RHS in the discretised equation above, I need to think about terms which contain the boundary points in the second term in LHS. In MATLAB, `4*del2(u,dx,dy)` is a useful way for computing the 2D Laplacian if u is known. Now `uinit` is a matrix containing the known *u*'s on the boundaries and 0's in all the internal terms. So I obtain `rhsb` in line 45 in such a way. The meaningful boundary terms for the Laplacian of the internal points are extracted in line 47. In lines 49 and 51, I reshape `rhsb` into a vector (first step in making **r**) and change its sign because I need to move these terms to the RHS. The **r** vector also contains the current *u'*s (RHS of the discretised equation above). So I add them in line 53.

Lines 54, etc:

Here is the real action! At each timestep, I solve `B\rhs`. I store the result to `utemp`. This is the solution for internal points of `u` in the next timestep. The **B** operator matrix will not change, but **r** will need to be updated with the current internal *u's* for use in the next timestep, so I do what is done in line 56 (the same is done in line 53). The for loop does the timestepping, advancing the solution to the next timestep from `rhs`. The rest of the code contains some plotting facilities.

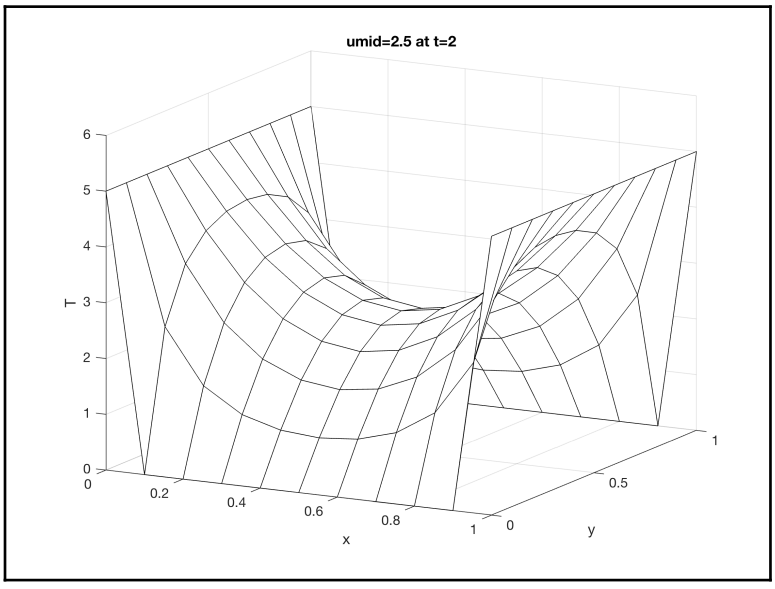
N.B. Here the MATLAB functions `toeplitz`, `kron`, `del2` and `reshape` make our life easier. You may not have such conveniences if you use other programming languages. I am especially proud of how I construct the boundary terms in the Laplacian in line 45.

`diffusion2d_fdm_implicit.m`

```
% example m-file
% the note king_2008_diffusion2d_fdm_implicit.pdf accompanies this
mfile
% martin king, martin.king@eng.monash.edu.my, 4 Sept 2008
%-----------------------------
% solves 2D Diffusion Equation by Backward/Implicit Euler timestepping.
% du/dt=a(u_xx+u_yy) in domain [0,1]x[0,1]
% bc's: one side x=0, x=1: u=5. y=0, y=1: u=0.
% ic: u=0.

clear all; close all;
format long;

a=1.; %the coefficient of diffusion

xmin=0.; xmax=1.0; ymin=0.; ymax=1.0;  % defining domain
dx=0.1; dy=0.1;  % grid spacing; this code will only work for dx=dy
x=[xmin:dx:xmax]; y=[ymin:dy:ymax];  % defining mesh
[X,Y]=meshgrid(x,y);  %X,Y are only used for plotting later
[nx,ny]=size(X);  % setting number of grid points

%where C=dt/dx^2, dt is timestep size
C=1.0; dt=C*dx.^2;

Thot=5.0; Tcold=0.;  % boundary values
wallhot=find(X==xmin|X==xmax);  % grid point indices for hot wall
wallcold=find(Y==ymin|Y==ymax);  % grid point indices for cold wall

u=zeros(nx,ny);  %declaring u
u(wallcold)=Tcold;u(wallhot)=Thot;  %boundary conditions
uinit=u;  %store initial u

tlast=2.;  %final time
ntsteps=ceil(tlast/dt);  %number of timesteps
dt=tlast/ntsteps;  %re-correct dt
C=dt./dx.^2;  %re-correct C

I = speye(nx-2); II=speye((nx-2)^2);% identity matrices
D = -a*dt*dx^(-2)*toeplitz([-2 1 zeros(1,nx-4)]);  % 1D Laplacian, see
formula for Backward Euler
% 2D Laplacian; difficult to explain, but this is the MATLAB way to
make this:
L = kron(I,D) + kron(D,I);
B = II+L;

%forming the rhs (here, it's the 2D Laplacian of the b.c.'s) by taking
DEL^2,
%internal points must first be zero for this to work
rhsb=-a*dt*4*del2(uinit,dx,dy);
%the border of internal points in rhsb nicely forms the 2D Laplacian of
b.c's
rhsb=rhsb(2:end-1,2:end-1);
%reshape rhs into a vector
rhsb=reshape(rhsb,(nx-2)*(ny-2),1);
%don't forget we have to change the sign, when we move terms from LHS
to RHS
```

```
rhsb=-1.*rhsb;

rhs=rhsb+reshape(uinit(2:end-1,2:end-1),(nx-2)*(ny-2),1);
for istep=1:ntsteps  %time stepping
    utemp=B\rhs;  %all the above was done just to do this!
    rhs=utemp+rhsb;
%
% 'animation'
    if(mod(istep,2)==0.)
        figure(1)
        utemp=reshape(utemp,nx-2,ny-2);  %reshaping into a matrix
        u(2:end-1,2:end-1)=utemp;  %update internal points
        surf(X,Y,u); shading interp; view(28,16); drawnow;
        xlabel x, ylabel y, zlabel T;
    end
%
end


%
%plotting the last solution

        figure(2)
        mesh(X,Y,u), axis([xmin xmax ymin ymax 0. 6]), view(28,16);
        xlabel x, ylabel y, zlabel T;
        colormap(1e-6*[1 1 1]);
        xmid=ceil(nx/2);ymid=ceil(ny/2);
        title(['umid=',num2str(u(xmid,ymid)),' at
t=',num2str(istep*dt)]);
        umid=u(xmid,ymid)
%
```

Plotted figure:



Exercise

This problem involves finding the solutions for the heat equation:

$$\frac{\partial T}{\partial t}=\frac{\partial T^2}{\partial x^2}+\frac{\partial T^2}{\partial y^2}.$$

It is the time-dependent version of the problem from the earlier 2D Laplace Equation exercise.

Again, the computational domain is a two-dimensional square plate with edges measuring 2 units long. At one edge T=6 is maintained and at all the other three edges T=0. Initially, T=0 in the whole plate away from the edge.

a) Solve this problem by the explicit Euler time marching and Second-order central differencing for the spatial derivatives. Create a MATLAB code to perform the computation (you are not allowed to use any of the MATLAB toolboxes such as the PDE toolbox).

Investigate numerically to obtain the stability condition in terms of dt/dx^2. In addition, include in your discussion the following.

b) What is the value for *t* (the time) in the exact centre of the plate when T reaches exactly 1 there? Explain your method and what step(s) have you taken to increase the accuracy in your answer.

c) Compute to a large enough final time. Do you find that the solution for T at the centre of the plate converges to a particular value? Is this value in agreement with the steady solution you found in the 2D Laplace Equation exercise?

Suggestion: You can consider reusing parts of `diffusion2d_fdm_implicit.m` as well as some of the techniques employed there. This m-file should be made available to you together with an explanatory note.

Solutions

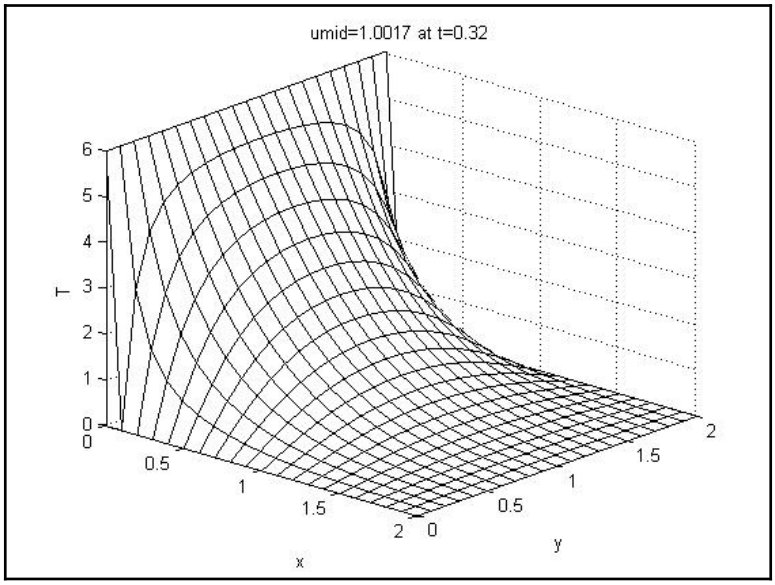
a) The PDE given in the question is discretised by Explicit Euler time marching and Second-Order Central Differencing for the spatial derivatives as

$$u_{i,j}^{(n+1)}=a(dt)\left[\frac{u_{i+1,j}^{(n)}-2u_{i,j}^{(n)}+u_{i-1,j}^{(n)}}{\Delta x^2}+\frac{u_{i,j+1}^{(n)}-2u_{i,j}^{(n)}+u_{i,j-1}^{(n)}}{\Delta y^2}\right]+u_{i,j}^{n}.$$

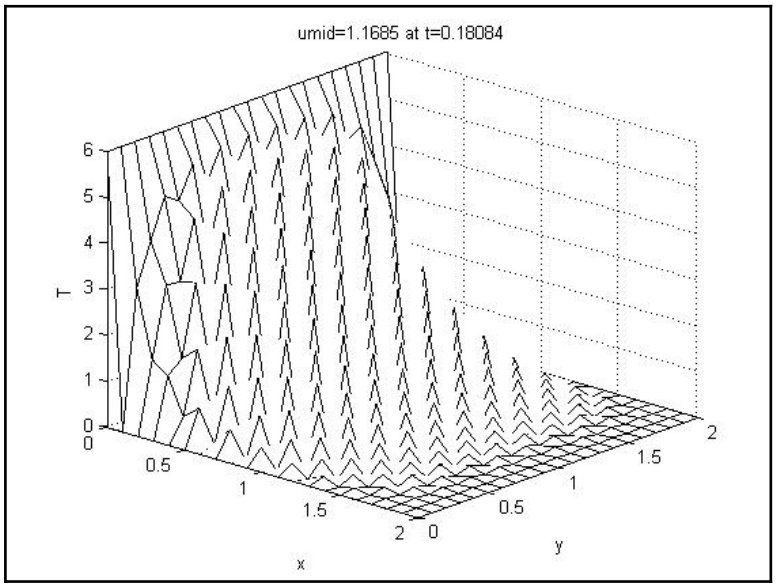
Like suggested in the question, I took `diffusion2d_fdm_implicit.m` and modified it. First, I set `xmax=2` and `ymax=2` in line 15 and `Thot=6` in line 24.

Then I deleted the part from line 37 to line 53 in `diffusion2d_fdm_implicit.m` as this is only necessary if we do a Backward Euler time-marching. The m-file I created to solve the current question is called `diffusion2d_fdm_explicit.m`. Obviously, there is usually more than one way to implement the coding. Here is just a suggested solution. Lines 39 to 42 complete the timestepping like the discretised equation above. See also the comments in the mfile.

I first ran the code with `C=0.1` (line 22). I have an if loop in line 44 to break the timestepping as soon as the solution at the middle (umid) of the plate exceeds 1. The following figure is obtained:



I then reran with `C=0.2` and `C=0.3`. I noticed that the method is unstable at `C=0.3`. Therefore, I ran a case with `C=0.25`. And it was successful. 'Funny' solutions start to happen when I used C greater than 0.25. For example at `C=0.27`:



So I conclude that for numerical stability $C=dt/dx^2<=0.25$ (Which, by the way, agrees exactly with the theory).

b) To accomplish this part. I first obtained the result using $C=0.1$ and $dx=0.1$. I get `istep*dt= 0.320000000000000` and `(istep-1)*dt= 0.319000000000000`. At this stage, I am probably only confident that the answer is $t=0.3$...
I then ran all these cases with dx=dy=0.05:

C	(istep-1)*dt	(istep)*dt
0.1	0.32	0.32025
0.05	0.320125	0.32025
0.02	0.32015	0.3202
0.01	0.3202	0.320225

At this stage, my confidence could be $t=0.3202$ when umid=1.0.

Just for the sake of experimenting I do the following two cases with $dx=dy=0.02$:

C	(istep-1)*dt	(istep)*dt
0.01	0.320272	0.320276
0.005	0.320274	0.320276

The above two cases took a while to run. Now I may be confident that t=0.32027... At this point I stopped.

c) This should be easy. I set `tlast=5` in line 31. And found that umid≈1.5 at the end. This agrees with the steady-state solution obtained in the 2D Laplace Equation exercise.

diffusion2d_fdm_explicit.m

```
% sample solution m-file.
% sample solution m-file.
% by martin king, 4 Sept 2008
%------------------------------
% solves 2D Diffusion Equation by Explicit Euler timestepping.
% du/dt=a(u_xx+u_yy) in domain [0,2]x[0,2]
% bc's: one side u=6, 3 sides u=0.
% ic: u=0.

clear all; close all;
format long;

a=1.; %the coefficient of diffusion

xmin=0.; xmax=2.0; ymin=0.; ymax=2.0;  % defining domain
dx=0.1; dy=0.1;  % grid spacing; this code will only work for dx=dy
x=[xmin:dx:xmax]; y=[ymin:dy:ymax];  % defining mesh
[X,Y]=meshgrid(x,y);  %X,Y are only used for plotting later
[nx,ny]=size(X);  % setting number of grid points

%where C=dt/dx^2, dt is timestep size
C=0.1; dt=C*dx.^2;

Thot=6.0; Tcold=0.;  % boundary values
wallhot=find(X==xmin);  % grid point indices for hot wall
wallcold=find(X==xmax|Y==ymin|Y==ymax);  % grid point indices for cold
wall

u=zeros(nx,ny);  %declaring u
u(wallcold)=Tcold;u(wallhot)=Thot;  %boundary conditions

tlast=5.;  %final time
ntsteps=ceil(tlast/dt);  %number of timesteps
dt=tlast/ntsteps;  %re-correct dt
C=dt./dx.^2;  %re-correct C

xmid=ceil(nx/2);ymid=ceil(ny/2);

for istep=1:ntsteps  %time stepping
     rhstemp=a*dt*4*del2(u,dx,dy);  %forming the rhs by taking DEL^2 of
known matrix u.
     rhs=rhstemp(2:end-1,2:end-1);  %the internal points in rhstemp
nicely forms L[u]
     rhs=rhs+u(2:end-1,2:end-1);  %add the most current u to complete
making rhs for the internal points
     utemp=rhs;  %internal points at next timestep are now just rhs
     u(2:end-1,2:end-1)=utemp;  %update internal points
%     if u(xmid,ymid)>=1.
%         break;
%     end
%{
% 'animation'
    if(mod(istep,2)==0.)
        figure(2)
        utemp=reshape(utemp,nx-2,ny-2);  %reshaping into a matrix
        u(2:end-1,2:end-1)=utemp;  %update internal points
        surf(X,Y,u); shading interp; view(35,25); drawnow;
        xlabel x, ylabel y, zlabel T;
    end
%}
end
```

```
%
%plotting the last solution
        utemp=reshape(utemp,nx-2,ny-2);  %reshaping into a matrix
        u(2:end-1,2:end-1)=utemp;  %update internal points


        figure(1)
        mesh(X,Y,u), axis([xmin xmax ymin ymax 0. 6]), view(42,26);
        xlabel x, ylabel y, zlabel T;
        colormap(1e-6*[1 1 1]);


        title(['umid=',num2str(u(xmid,ymid)),' at
t=',num2str(istep*dt)]);
        umid=u(xmid,ymid)
%
```

Exercise 5. ODE – Numerical order of accuracy

Exercise

The radioactivity of an unstable isotope may be modelled by the following equation:

$$\frac{du(t)}{dt} = \alpha u(t).$$

The function *u(t)* represents the concentration of the isotope. This concentration decays by a factor of two during a time interval *T* (the half-life). Find an analytical solution to the above ODE and show that

$$\alpha = \frac{-\ln 2}{T}.$$

The objective of this question is to investigate numerically the order of accuracy for the Explicit Euler's method, the Heun's method and the Midpoint method which are covered in the lectures.

Use the m-file `ode_accuracy_lab5_student.m` as a template. As an example, the Euler's method has been coded at line 30. You are required to code in the Heun's method (near line 33) and the Midpoint method (near line 37). If you find it necessary to, you may add to or modify other part of the code.

For all the methods, report and explain your findings. Do the orders of accuracy obtained agree with the theoretical expressions (usually derived using Taylor series) given during the lectures? Can you suggest a reason for the order of accuracy you observe for the Midpoint method?

Solutions

The solution to the ODE is

$$u(t) = u(t=0)e^{\alpha t}.$$

Since *T* is the half-life,

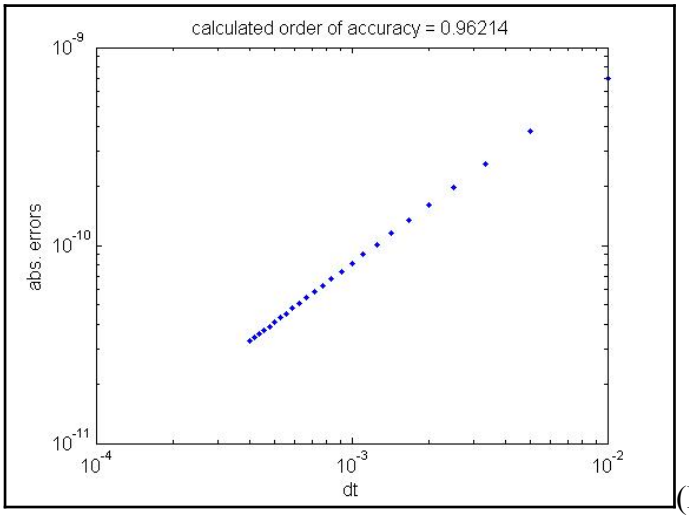
$$\frac{u(t=0)}{2} = u(t=0)e^{\alpha T},$$

from here, it is easily shown that

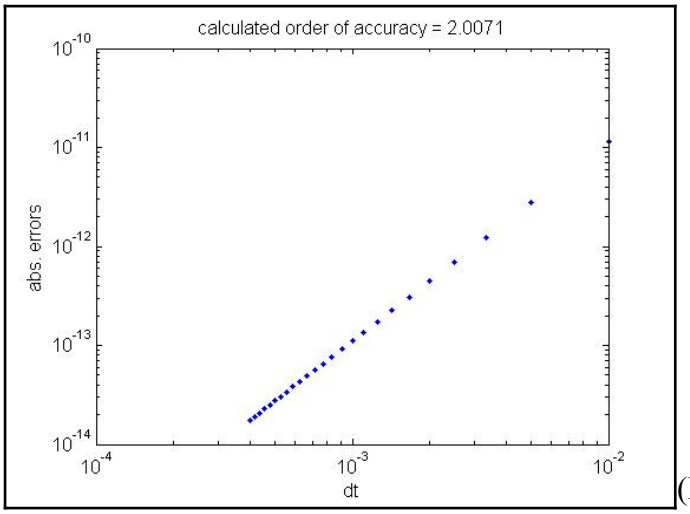
$$\alpha = \frac{-\ln 2}{T}.$$

The suggested coding for case 2 (Heun) and case 3 (Midpoint) can be found in the provided m-file (`ode_accuracy_lab5_solution.m`).

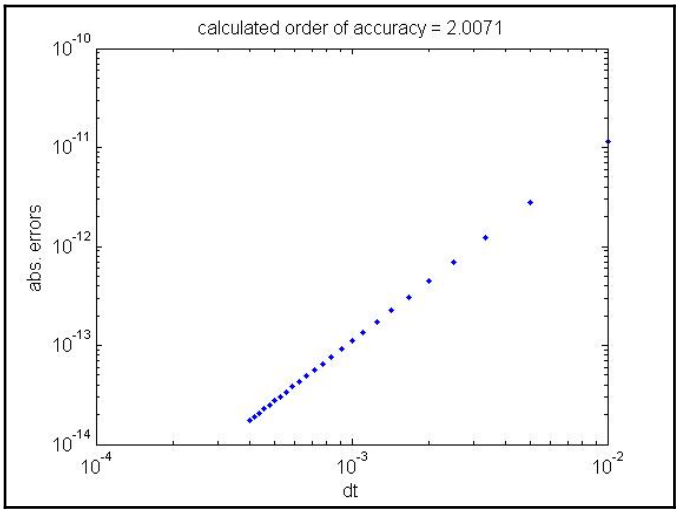
The following figures of absolute errors against *dt* (in log-log plots) are plotted for the Explicit, Heun and Midpoint methods respectively:



(Explicit)



(Heun)



(Midpoint)

These are exact errors at the end of the time integration. For the explicit Euler method, the order of error, $Ea=O(dt^2)$, is for a local truncation error, i.e. for advancing a single *dt* step. Since the total number of timesteps integrated is proportional to *1/dt* , the error at the end of the time integration (i.e. the total error or the global truncation error) is $\sim O(dt)$.

Both the Heun's and the Midpoint methods have local truncation error of $\sim O(dt^3)$ and therefore, global truncation error $\sim O(dt^2)$.

The results from our little numerical experiments confirm the theoretical analysis. Since Heun's and the Midpoint methods are higher order methods, the absolute errors are also smaller than those for the Euler's method for a particular *dt* used. This fact can be seen clearly in the figures above.

ode_accuracy_lab5_solution.m

```
% Sample solution m-file.
% Suggested solutions to guide tutors.
%--------------------------------
% Investigating the order of accuracy for some common
% numerical schemes for ODES.
% Numerically solving the ODE
% du/dt=alpha*u

clear all;  % clearing variables from workspace
close all;  % closing figure windows
format long; % display all the decimal places

alpha=-4.;  % suggested alpha
finaltime=5.;  % suggested final time
dtvect=[];  % declaring two empty matrices to store dt's and errors
errorvect=[];

% loop for different timestep sizes
for ndt=2:2:50
 u0=1.;  % initial condition
 u=u0;  % set first value of f to f0
 dt=0.02/ndt;  % timestep sizes
 itimelast=round(finaltime/dt);  % total number of timesteps required

 for itime=0:itimelast-1  % timestepping loop
  switch 3  % use switch to choose which case to run
   case 1  %Explicit Euler
%BEGIN: WRITE YOUR CODE HERE, time-marching in u
    u=u+dt*(alpha*u);
%END
   case 2  %Heun
%BEGIN: WRITE YOUR CODE HERE, time-marching in u
    k1=dt*(alpha*u);
    k2=dt*(alpha*(u+k1));
    u=u+0.5*(k1+k2);
%END
   case 3  %Midpoint method
%BEGIN: WRITE YOUR CODE HERE, time-marching in u
   umid=u+0.5*dt*(alpha*u);
   u=u+dt*(alpha*umid);
%END
  end
  t=dt*(itime+1);  %advance to next time instant
  exact=exp(alpha*t)*u0;  %the exact solution to the ODE
  error=abs(u-exact);
 end
  figure(1)
  loglog(dt,error,'.','MarkerSize',16)  %plotting errors against dt
```

```
  hold on;
% store dt and final error to dt_vect and error_vect respectively
  dtvect=[dtvect; dt]; errorvect=[errorvect; error];
end
hold off;

p=polyfit(log10(dtvect),log10(errorvect),1);  %finding gradient of the
points in figure 1
title(['calculated order of accuracy = ',num2str(p(1))]);  %print it
out as title
xlabel('dt');
ylabel('abs. Errors');
```

Exercise 6. ODE – Numerical order of accuracy and stability

Exercise

We have encountered the following ODE in the previous exercise:

$$\frac{du(t)}{dt} = -4u(t).$$

The function *u(t)* may represent the concentration of a radioactive isotope.

a) Use the m-file `ode_accuracy_lab6_student.m` as a template. As before, the Euler's method has been coded near line 30. This m-file is essentially identical to `ode_accuracy_lab5_student.m` which you have used in the previous exercise. However, here you are required to code in the Runge-Kutta 4th-Order method for case 2. Investigate the order of accuracy for this method.

b) Obtain a stability condition (analytical) for the Euler's method for this ODE, in terms of *dt*, the size of the timestep. Then, investigate this stability condition numerically on the above ODE. Does your finding agree with the theoretical derivation?

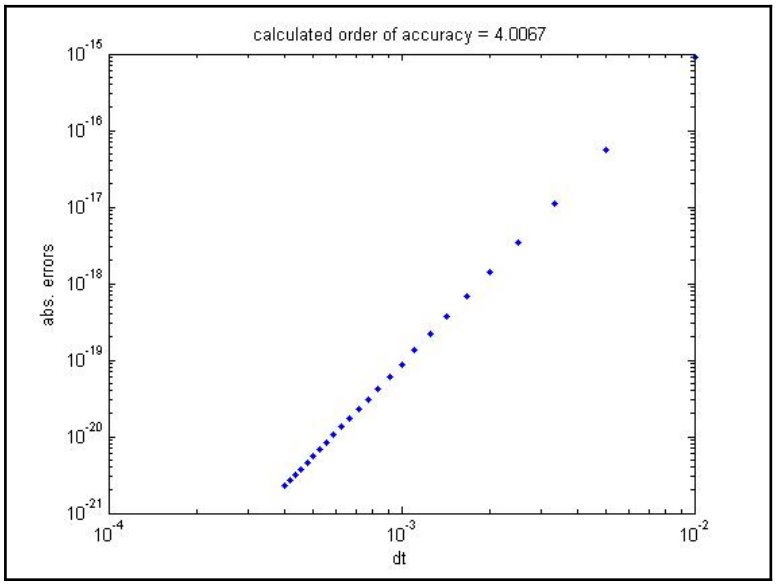
Hint: For part b, you can make some minimal changes to either one of the two m-files above.

Solutions

a) Please asee `case 2` in `ode_accuracy_lab6_solution.m`. The following snippet of MATLAB coding is used for RK4.

```
k1=dt*(alpha*u);
k2=dt*(alpha*(u+k1/2));
k3=dt*(alpha*(u+k2/2));
k4=dt*(alpha*(u+k3));
u=u+1/6*(k1+2*k2+2*k3+k4);
```

If the RK4 is coded correctly (without a change to other default values) the following figure will result:



Thus, the RK4 method applied to the present ODE has an order of accuracy which agrees with the theoretical accuracy for RK4.

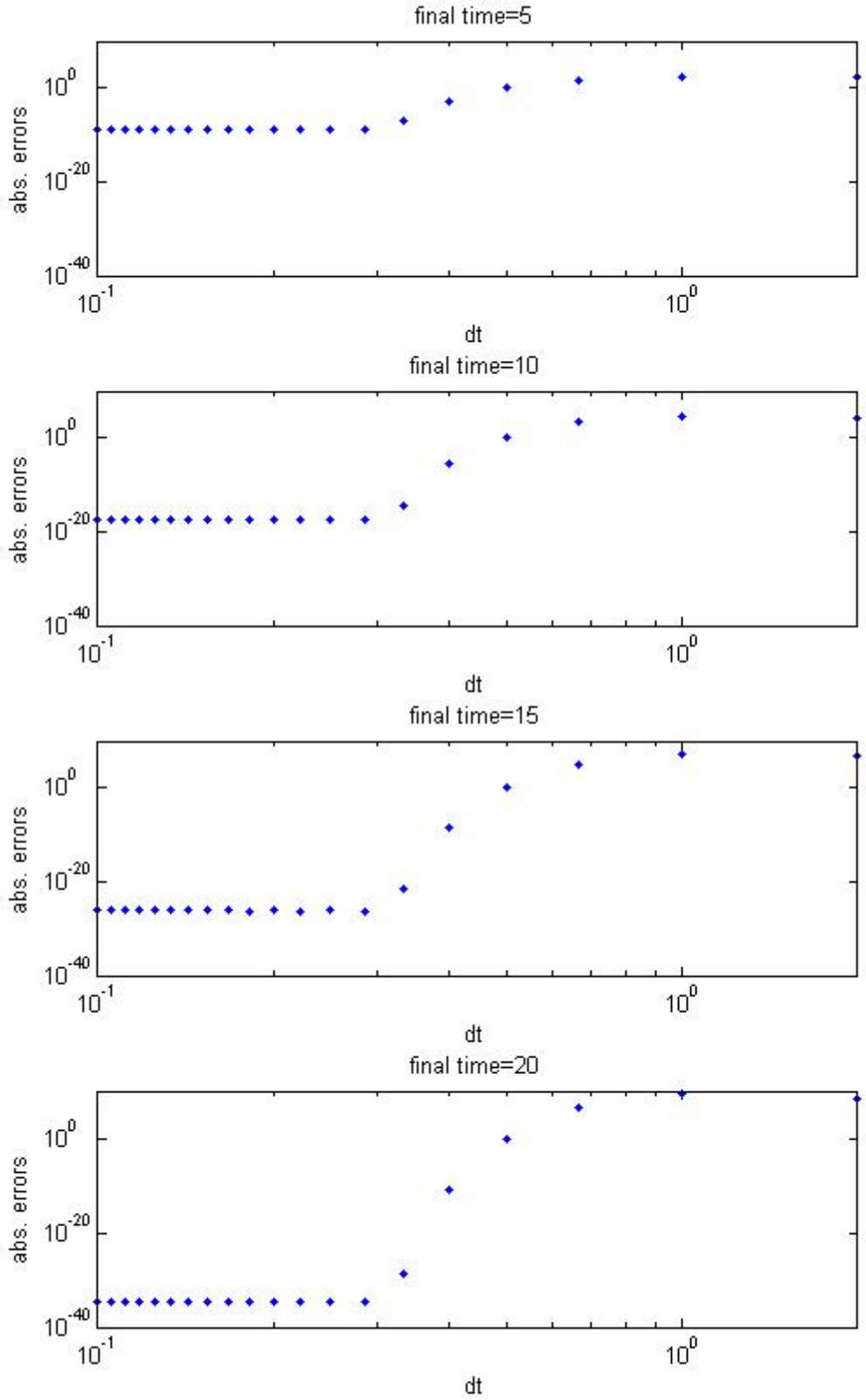
b) From an example in the lectures, the stability condition for an ODE of the present form is:

$$-2<(dt)k<0.$$

Here k=-4. Therefore:

$$0<dt<1/2.$$

You should notice that in part a, the largest *dt* used is 0.01, thus all the cases carried out in part a are numerically stable. To carry out a numerical investigation for the stability condition, I perform computations with a larger values of *dt* (see line 25 in `ode_stability_lab6_solution.m`), i.e from *dt*=0.1 to 2. I also investigate cases for 4 different final times (see the outermost for loop at line 16). If you just go ahead and run this code, you should obtain this figure:


final time=5
abs. errors
10^{0}
10^{-20}
10^{-40}
10^{-1}
10^{0}
dt
final time=10
abs. errors
10^{0}
10^{-20}
10^{-40}
10^{-1}
10^{0}
dt
final time=15
abs. errors
10^{0}
10^{-20}
10^{-40}
10^{-1}
10^{0}
dt
final time=20
abs. errors
10^{0}
10^{-20}
10^{-40}
10^{-1}
10^{0}
dt

From these figures, I am able to conclude that the method starts to deteriorate at *dt=0.3*. The errors appear to reach an unacceptable O(1) at *dt=0.5*. Also, from the figures, we can see that above *dt=0.5*, as longer durations are being integrated (going down the panel of figures), the errors are larger. Whereas the reverse is true for *dt<0.3*, i.e. errors become smaller as larger final time are being integrated. This is consistent with the concept of numerical stability in the current context. Thus I may conclude that numerically, the Euler's method applying to this ODE obeys the theoretical stability condition, although the errors may already become unacceptable at around *dt=0.4*, especially in cases in which the final times integrate to are short (see top figure).

N.B. There is no one fixed way to carry out the investigation for part b. What is shown here is only a guide or a suggestion. As long as you can perform an investigation numerically with a range of *dt*, and arriving at the same findings and conclusions, then regardless of methods or coding used, the solutions may be regarded as successful and complete.

ode_stability_lab6_solution.m

```
% Sample solution m-file.
% Suggested solutions.
%--------------------------------
% Investigating the order of accuracy for some common
% numerical schemes for ODES.
% Numerically solving the ODE
% du/dt=alpha*u

clear all;  % clearing variables from workspace
close all;  % closing figure windows
format long; % display all the decimal places

alpha=-4.;  % suggested alpha
finaltime=5.;  % suggested final time
dtvect=[];  % declaring two empty matrices to store dt's and errors
errorvect=[];

% loop for different timestep sizes
for ndt=2:2:50
 u0=1.;  % initial condition
 u=u0;  % set first value of f to f0
 dt=0.02/ndt;  % timestep sizes
 itimelast=round(finaltime/dt);  % total number of timesteps required

 for itime=0:itimelast-1  % timestepping loop
  switch 2  % use switch to choose which case to run
   case 1  %Explicit Euler
%BEGIN: WRITE YOUR CODE HERE, time-marching in f
    u=u+dt*(alpha*u);
%END
   case 2  %Runge-Kutta 4-th Order
%BEGIN: WRITE YOUR CODE HERE, time-marching in f
    k1=dt*(alpha*u);
    k2=dt*(alpha*(u+k1/2));
    k3=dt*(alpha*(u+k2/2));
    k4=dt*(alpha*(u+k3));
    u=u+1/6*(k1+2*k2+2*k3+k4);
%END
  end
  t=dt*(itime+1);  %advance to next time instant
  exact=exp(alpha*t)*u0;  %the exact solution to the ODE
  error=abs(u-exact);
 end
```

```
  figure(1)
  loglog(dt,error,'.','MarkerSize',6)  %plotting errors against dt
  hold on;
% store dt and final error to dt_vect and error_vect respectively
  dtvect=[dtvect; dt]; errorvect=[errorvect; error];
end
hold off;

p=polyfit(log10(dtvect),log10(errorvect),1);  %finding gradient of the
points in figure 1
title(['calculated order of accuracy = ',num2str(p(1))]);  %print it
out as title
xlabel('dt');
ylabel('abs. errors');
```

Exercise 7. ODE – Lorenz Equations

Exercise a

The most famous equations displaying chaos behaviour are arguably the Lorenz equations:

$$\frac{dX}{dt} = -\sigma X + \sigma Y$$
$$\frac{dY}{dt} = -XZ + rX - Y$$
$$\frac{dZ}{dt} = XY - bZ$$

E.N. Lorenz (1917-2008) derived them as a simplified model for convective cells in the atmosphere. It turned out that Lorenz's work on this simple set of equations would revolutionalise the whole field of dynamical systems.

For this question, you will investigate one important property of a chaotic system. Obtain the mfiles `lorenz_lab7q1_student.m` and `lorenzrhs.m`. Run `lorenz_lab7q1_student.m`, using *X0*=10 (the initial condition for *X*), and final time =100. This m-file is coded with the explicit Euler's time-marching method. It will take some time to finish running, be patient.

Make sure that you can obtain the famous butterfly-shaped Lorenz attractor. The solutions, which have a distinctive structure, are attracted (hence the name) to a particular part of the *X-Y-Z* space. But you should recognise that the solutions are non-periodic with complicated fluctuations (see figure 2). You might also see from figure 2 that the solutions also jump from one 'wing' of the butterfly to another.

a) The time history of v=[*X Y Z*] is stored in a matrix called `vsave` (near line 51). Save `vsave` to a MATLAB data file (MAT-File). Run the code again, but decrease *X0* by 0.1%. Also save `vsave` for this second case to another MAT-File. Therefore now you have the time history of two vectors. Clear your MATLAB workspace of previous variables and load these two vectors. Plot the separation (i.e. the distance between two three-dimensional vectors) for these two vectors against time (you should scale the separation by the magnitude of one of the two vectors). In other words, plot:

$$\left\| v_1 - v_2 \right\| / \left\| v_1 \right\|$$

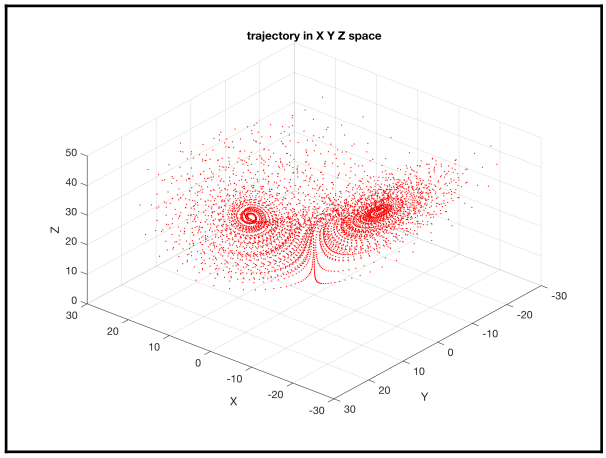
against *t*. Write another simple mfile to carry out this task if you need to. What can you say about the separations of the two solutions? For how long do the vectors stay within 10% of each other? **Hint:** you may find the MATLAB function `norm()` useful.

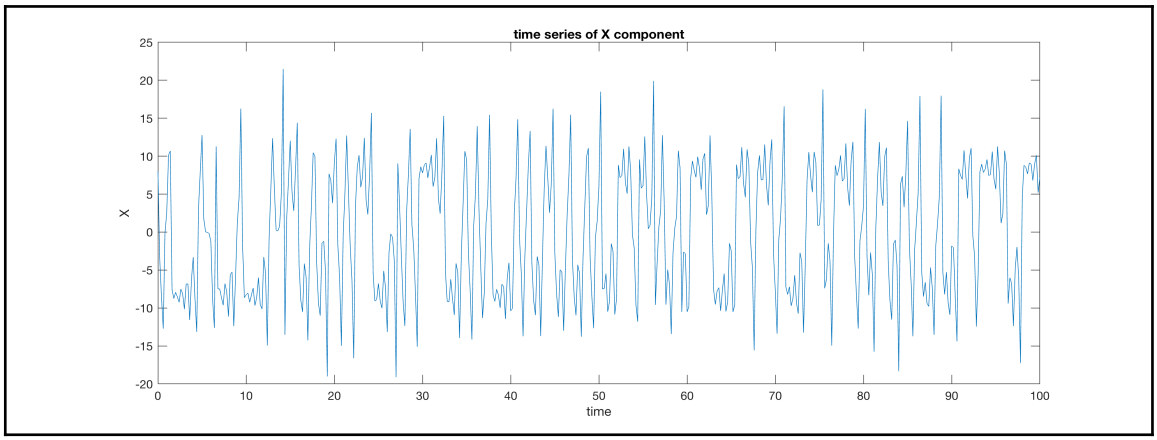
b) Repeat the investigation in a) by using a smaller timestep, say *dt=0.002* (note the default is set to *dt=0.02*). What do you observe?

c) With, again, *dt=0.02* but using an extremely small difference in initial conditions (e.g. 1e-13), repeat the investigation. What do you observe?

Solutions a

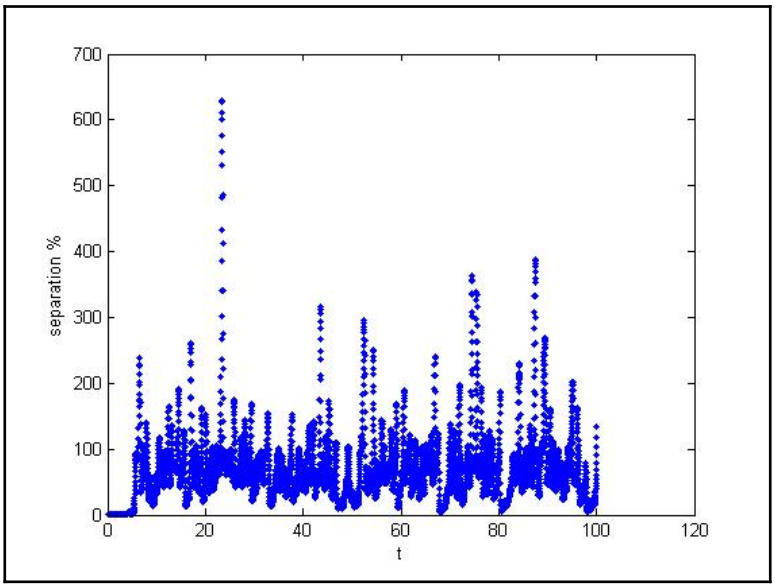
a) I first run `lorenz_lab7q1_student.m` as suggested. The following figures about the Lorenz attractor is obtained.





Next, I uncomment line 69 in `lorenz_lab7q1_student.m` so I can save the matrix `vsave`. I then run two cases with *X0=10.* and *X0=9.99* (which is *0.1%* from *10*). I then use `lorenz_loadplot_solution.m` to load and plot the normalized separations of the two solutions against integration time.

It is found that the solutions stay within *10%* of each other until *t=5.4*. From then on, the solutions are observed to have large but bounded differences. The differences are so large that they can be regarded as two different solutions.



b) I repeat the two cases with *dt=0.002*. And find that the solutions stay within *10%* for *t=5.5*. After that the differences in the solutions quickly grow to be as large as those in case **a)**. It seems that a more accurate time integration may not increase the lead time in prediction.

c) I run a case with *dt=0.02* and *X0=10-(1e-13)*. Comparing with the solution for the case *dt=0.02* and *X0=10*, the solutions stay within *10%* until *t=21.8*. Thereafter, the differences of two solutions also become very large.

It is concluded that Lorenz equations are very sensitive to the initial condition used. Using a smaller timestep does not delay the time to reach large separation. It seems that obtaining two different solutions are ultimately inevitable, even in cases which very similar initial conditions are used. Note importantly that you may not get identical numerical results to mine because tiny differences between our softwares and computers can cause the divergence.

Exercise b

This question is a continuation of Exercise a.

a) A valid assumption is that a high-order method may help delay or restrict the separation of the two solutions which are started with a small difference in initial conditions. Modify `lorenz_lab7q1_student.m` to employ the Runge-Kutta 4th-order method. Repeat the investigations you carried out in part a of Exercise a) using the Runge-Kutta 4th-order method (with a difference in *X0* of 0.1% and *dt*=0.02). What do you observe?

b) What if now *dt* is being decreased to *dt*=0.002 (i.e. repeating as part b of Exercise a)?

c) Repeat the computation as in part c of Exercise a.

d) On many occasions in this course, you have encountered and applied the concepts of accuracy and stability, especially during the computing sessions. A third pillar of numerical methods is *consistency*. For a method to be *consistent*, the solutions must converge when smaller and smaller *dt* is used while other parameters remain the same. Using *X0*=10, and a suggested final time=50, verify that neither the explicit Euler's method nor the Runge-Kutta 4th-Order is *consistent* for the Lorenz equations.

Hint: You may want to cancel plotting and cancel storing `vsave` to speed up the computations. You can check on the solutions to the equations at final times (just `v`) to see if they converge. You can try dt down to 2d-6. Be patient, cases with small timesteps may take quite a while to run.

N.B. Regarding part **c)**: In fact, it's been recently suggested that no consistent numerical method has been found for the integrations of ODEs that exhibit chaotic behaviours. The computed numbers are not solutions of the continuous differential equations (in the limit dt→0), but are merely numerical noise which (luckily?) also resides in the attractor. If interested see Yao, L.S. and D. Hughes, 2008, Comment on "Computational periodicity as observed in a simple system" by Edward N. Lorenz, Tellus A, 60A, pp. 803-805.

Solutions b

a) I code in the R-K4 method in `lorenz_lab7q2_solution.m`.

```
k1=dt.*lorenzrhs(v,r,sig,b);
k2=dt.*lorenzrhs(v+k1/2,r,sig,b);
k3=dt.*lorenzrhs(v+k2/2,r,sig,b);
k4=dt.*lorenzrhs(v+k3,r,sig,b);
v=v+(k1+2.*k2+2.*k3+k4)/6.;
```

And redo the investigation as I do for part a of Exercise a. I find that the two solutions stay within *10%* of each other until *t=5.06*. As always, the differences quickly grow to be large after that. It appears that a higher order method may not provide better predictability.

b) Now I redo part a) but with *dt=0.002*. It is found that the solutions stay close until *t=5.06*.

c) I find *t=33.0*. From the simple exercise performed so far, it may be commented that the only factor that can help in delaying the time when large separations start to occur is in having initial conditions which are closer to each other.

d) I run both the Euler's and R-K 4 methods with the suggested *X0=10* and final time=*50*. First using *dt=0.02* and subsequently a number of cases with further reductions of *dt* until *dt=0.000002*. There is no indication that the final solution for **v** at *t=50* is reaching some converging values. This may point to the serious problem in consistency for applying Euler's and R-K 4 methods on systems of equations that exhibit chaos.

`lorenz_lab7q2_solution.m`

```
% Lab7 Q2.
% Sample solution m-file.
```

```
%-------------------------------------------------
% time integration of "chaotic" Lorenz 3-component system.
% employs 4th-Order Runge-Kutta. this code needs lorenzrhs.m to run
% Martin King, Aug 2008.

clear all;
close all;

%setting the initial conditions
disp('initial Y and Z are zero. Specify initial 0 < X <= 10 when asked
')
X0 = input('give an initial X value  ');
Y0 = 0;
Z0 = 40;

time = input('final integration time units (at least 100) ');

% define constants of the system
r = 28;
sig = 10;
b = 8/3;

%timestep size
dt = 0.02;

v = [X0 Y0 Z0];  % vector of initial conditions
vprime = zeros(1,3);

% setting up figure 1 for plotting
figure(1)
axis([-30 30 -30 30 0 50]), view(-140,45)
xlabel('X'), ylabel('Y'), zlabel('Z')
title('trajectory in X Y Z space')
hold on

n = 0;  % counter
vsave=[];

for t = 0:dt:time
%R-K 4
% lorenzrhs is a function written in lorenzrhs.m and returns the
% vprime vector
    k1=dt.*lorenzrhs(v,r,sig,b);
    k2=dt.*lorenzrhs(v+k1/2,r,sig,b);
    k3=dt.*lorenzrhs(v+k2/2,r,sig,b);
    k4=dt.*lorenzrhs(v+k3,r,sig,b);
    v=v+(k1+2.*k2+2.*k3+k4)/6.;

    p = plot3(v(1),v(2),v(3),'r.','MarkerSize',4);

   drawnow

    vsave=[vsave; v];

    if mod(t,.2) == 0
        n = n+1;
        Xhist(n) = v(1);
        thist(n) = t;
    end
end
grid
hold off
```

```
scrsz = get(0,'ScreenSize');
%[left, bottom, width, height]
figure('Position',[2 2 scrsz(3)/2. scrsz(4)/3.]);

figure(2)
plot(thist(1:n),Xhist(1:n))
xlabel('time'); ylabel('X')
title('time series of X component')

%uncomment the following line to save vsave to file1.mat
%save file1 vsave;
%
```

Exercise 8. ODE – Eigenvalues and eigenvectors

Exercise

This question originates from the physical example of a slender column under compression given in the lectures. It concerns the eigenvalue problem for the ODE:

$$\frac{-d^2 y}{d x^2} = p^2 y .$$

If this equation is discretised by the Second-order Central Differencing, the matrix equation in the following form will result:

$$\mathbf{D}\boldsymbol{y} = \lambda \boldsymbol{y}.$$

where **D** is a tridiagonal matrix, ***y*** a column vector containing the unknown values of y in the internal points and λ a scalar. Therefore this is an eigenvalue problem for the matrix **D.**

a) Take the length of the column as L=4, and 19 internal nodes for the discretisation. Then use MATLAB to calculate and plot the first 4 eigenmodes for the column fixed at both ends under compression. Show also their corresponding eigenvalues (the 4 smallest ones).

b) Since $p^2 = P/EI$, we can rewrite the above ODE as

$$-E\frac{d^2 y}{d x^2} = \frac{P}{I} y .$$

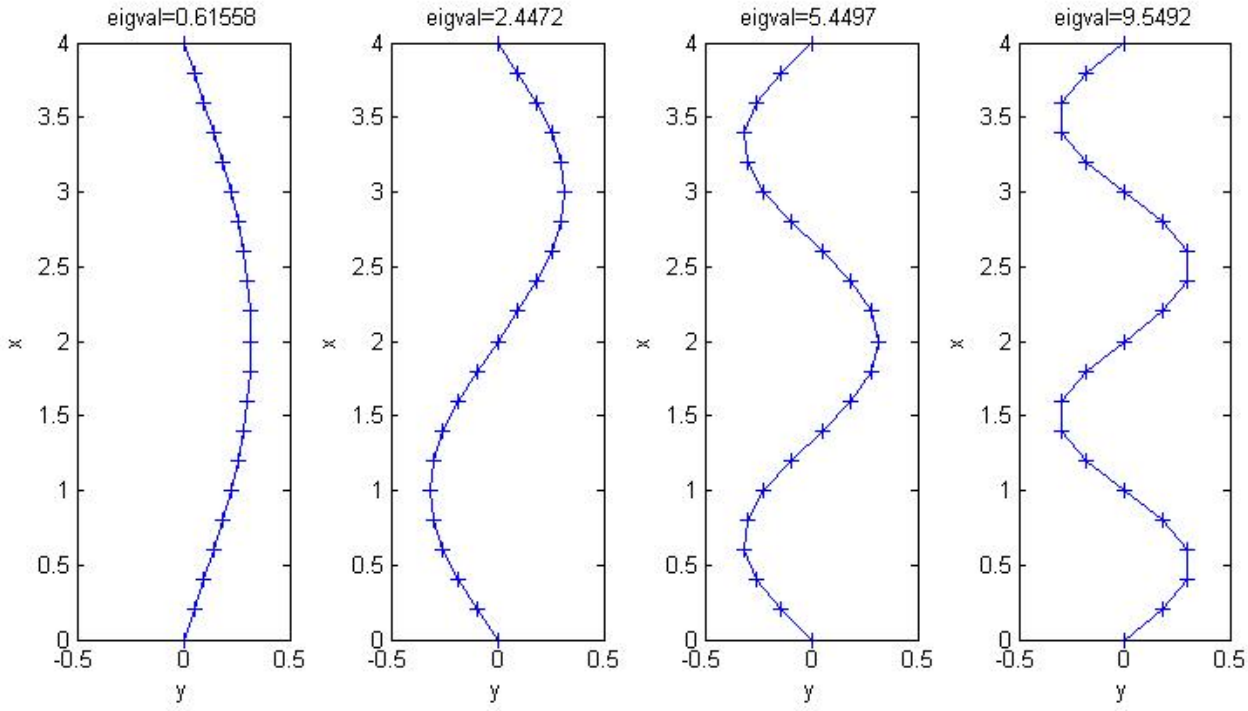
Now, a concrete column with modulus E which changes linearly (along the column) from 30 to 100 is installed. Repeat part a).

Hint: You can use the MATLAB built-in function `eig()` to find the eigenvectors and eigenvalues. You are reminded that the results returned by `eig()` are not necessarily sorted in order in the eigenvalues.

Solutions

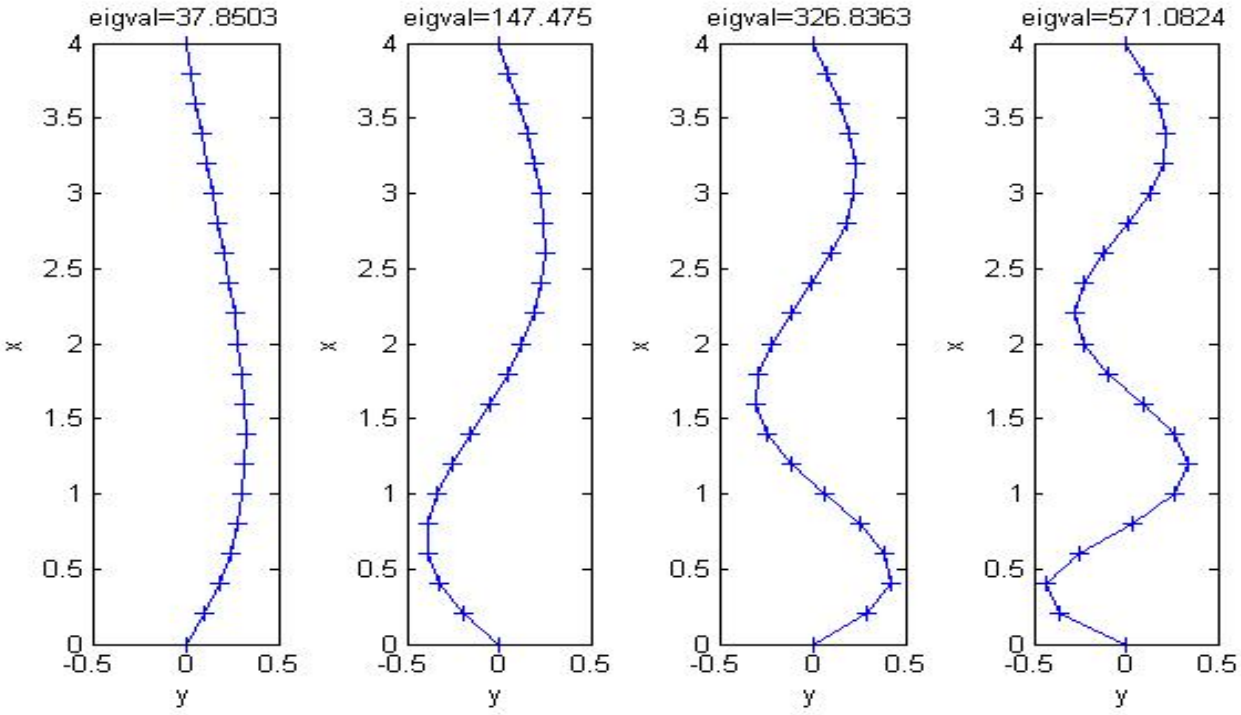
a) The **D** is a tridiagonal matrix of 19 rows x 19 columns containing -dx^(-2)*(1, -2, 1) in its three main diagonals respectively. I create **D** in line 11 in `eigenvector_solution.m`. Although I call it `A` there. I used `eig()`, to find the

eigenvectors and eigenvalues in line 22. In line 23, I sorted the eigenvalues. At the same time I obtained the order that it should be sorted and stored the indices in `isort`. In the for loop from line 25, I plotted the first 4 eigenvectors, with the titles as the eigenvalues (p^2) . The following is the figure I obtained:



b) This part is slightly tricky because we need to incorporate the linearly varying E in **D**. The part from lines 20 to 23 do that. I first create EE, which is a tridiagonal matrix containing the internal E's in the principal diagonal (`diag(E(2:end-1),0)`). And the diagonal entries below the principal diagonal end with the second last internal point of E's (`diag(E(2:end-2),-1)`), while the diagonal above starts with the second internal point of E's `diag(E(3:end-1),1)`. To complete, I multiply `EE` by the respective entries in `A` in line 23.

I obtained:



This is interesting because we now find that the ‘buckling’ (the wavy parts) shifted to the weaker part of the column near the bottom of the column.

Exercise 9. ODE – Boundary value problems

Exercise

(This exercise originated from Ravi Jagadeeshan, Monash University. But the sample solutions are mine.)

Consider the flow of a Newtonian fluid between two flat wide plates which are distance *2H* apart, and which is governed by the equation:

$$\frac{d}{dy}\left(\mu \frac{du}{dy}\right)=\frac{-\Delta p}{L}$$

subject to the boundary conditions:

$$\frac{du}{dy}(0)=0; u(\pm H)=0$$

(a) Use MATLAB to solve this two-point boundary value problem for $\Delta p=2.8\times10^5$ *Pa,* $\mu=0.492$ *Pa s,* $L=4.88$ *m* and $H=0.0025$ *m.*

(b) Plot the dependence of velocity on position.

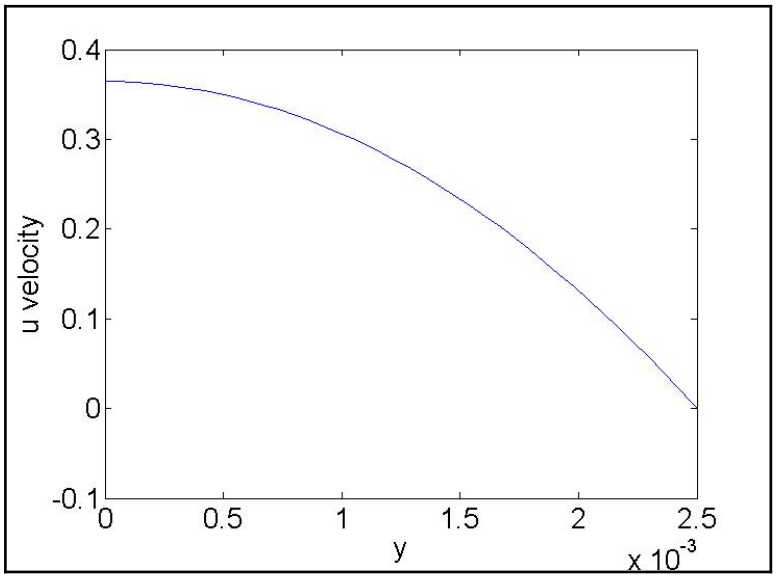
(c) Compare your numerical prdiction with the analytical solution by plotting the two solutions on the same plot.

(d) Compute the average velocity of the fluid between the plates and compare your result with the analytical prediction.

Solutions

I 'cheated' with using MATLAB's ODE45 function. But you gain from how to use it. If you have written a code from scratch to solve this, bravo! We can compare our answers below.

(a) and (b)
The codes (`main.m`, `twoode.m`, and `res.m`) I used are provided. Since the boundary condition is also given in the mid-plane (at *y=0*), I only solved half of the flow field. from *y=0* to *y=0.0025*. Here's the solution I obtained:

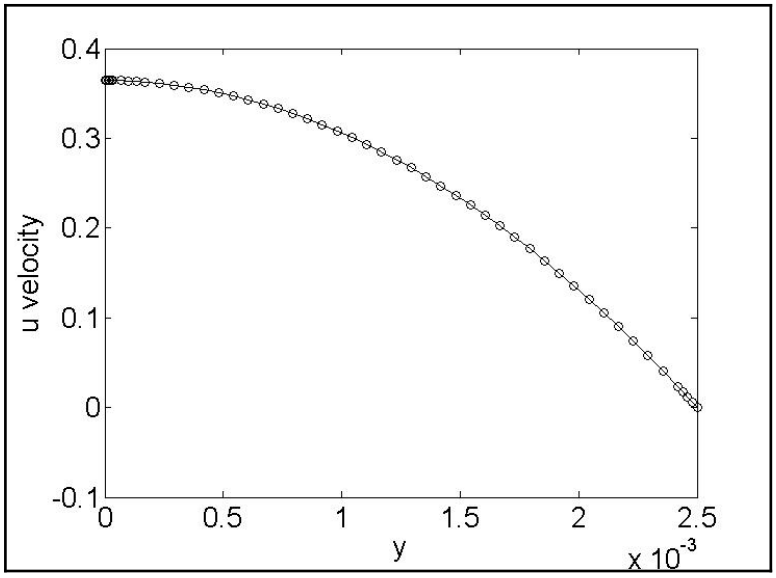


The solution for the complete flow field is symmetrical about $y=0$.

(c) You should be able to integrate the 2nd-order ODE twice (and using the appropriate boundary conditions given) to obtain the following analytical solution:

$$u=\frac{\Delta p}{2L\mu}\left(-y^2+0.0025^2\right).$$

Now, I plotted the analytical solution in a solid line and the numerical solution in open circles. As can be seen from the figure, the numerical and analytical solutions agree very well.



(d) The average velocity is the velocity that would give the same flow rate as the parabolic velocity.

$$U_{ave}(2\mathrm{H}) = \int_{-H}^{H} u\,dy.$$

From this, it is found that U_{ave}=0.2429... The average velocity from the numerical solution is found by

```
2*trapz(y,u(:,1))/(2*H)
```

which gives also identical solution to the above. This is not too surprising as the numerical solution for velocity is so close to the analytical solution.

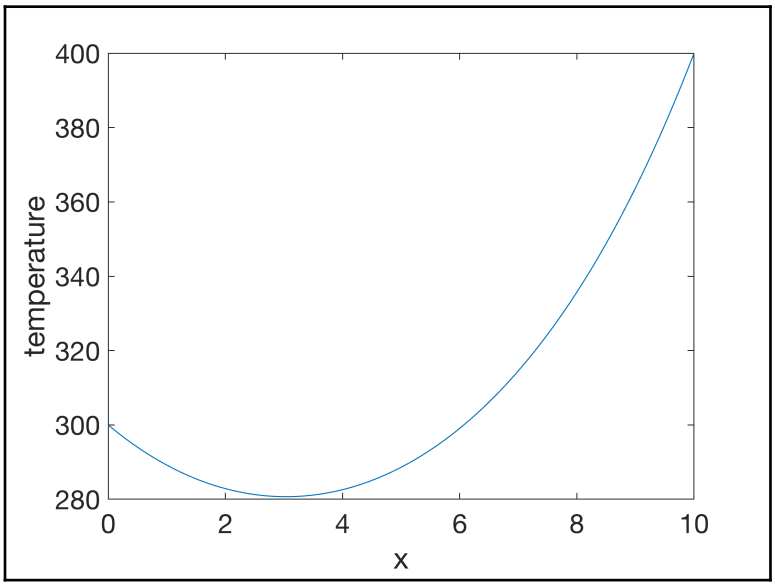
Bonus

Here I will show a slightly different MATLAB technique of solving same type of equation. This ODE can model heat conduction in a one-dimensional solid with a temperature-dependent heat source/sink applied along it.

$$\frac{d^2T}{dx^2} = -h(T_a - T)$$

x=[0, 10], T(0)=300, T(10)=400, Ta=200, and h=0.05.

You can read the m-files `main_heat.m`, `twoode_heat.m`, and `twobc_heat.m` provided. The following figure shows the numerical solution:



Exercise 10. Finite Element Method

To be completed.

Exercise 11. Interpolation

To be completed.

Exercise 12. Filters

To be completed.