MATHEMATICAL INSTITUTE

UNIVERSITY OF OXFORD

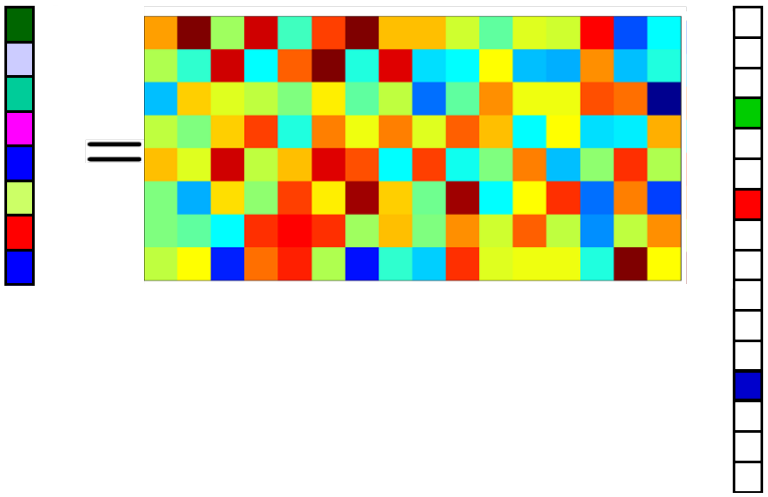
Computational Mathematics

Students' Guide

Hilary Term 2019

by

Prof Vidit Nanda and Dr Alberto Paganini



Acknowledgments: The course guide is based on previous versions by Dr Andrew Thompson and others before him.We are grateful to all previous authors.

Contents

Chapter 1

Introduction

The use of computers is widespread in all areas of life, and at universities they are used in both teaching and research. The influence and power of computing is fundamentally affecting many areas of both applied and pure mathematics. MATLAB is one of several systems used at Oxford for doing mathematics by computer; others include Mathematica, Maple, Sage and SciPy/NumPy. These tools are sufficiently versatile to support many different branches of mathematical activity, and they may be used to construct complicated programs.

1.1 Objectives

The objective of this practical course is to discover more about mathematics using MATLAB. Last term you were introduced to some basic techniques, by working through the Michaelmas Term Students' Guide which you are due to complete near the beginning of this term. After this, for the rest of Hilary Term, you will work alone on two projects.

While MATLAB complements the traditional part of the degree course, we hope the projects help you revise or understand topics which are related in some way to past or future lectures. It is hoped that at the end of this MATLAB course you will feel sufficiently confident to be able to use MATLAB (and/or other computer tools) throughout the rest of your undergraduate career.

1.2 Schedule

Deadlines

- 12 noon, Monday, week 6 (February 18) Submission of first project.
- 12 noon, Monday, week 9 (March 11) Submission of second project.

Computer and demonstrator access

This term, the practical sessions with demonstrators in Weeks 1 and 2 will be the same as those for weeks 7 and 8, respectively, of Michaelmas Term. From Week 3 onwards, there are no fixed hours for each college, and you may use the timetabled classrooms at the Mathematical Institute whenever suits you within the following times:

- Weeks 3–8: Mon 3pm–4pm; Thurs 3pm–4pm.

There will be additional sessions ahead of the project deadlines:

- Week 5: Weds 3pm–5pm; Fri 3pm–5pm.

- Week 8: Fri 3pm–5pm.

It's probably a good idea to check the course website (https://courses.maths.ox.ac.uk/node/37598) for any possible updates to these times.

Note that you will need to bring your laptop to the drop-in sessions. If you need to borrow a laptop, you will need to inform the Academic Administrator (acadadmin@maths.ox.ac.uk) of your chosen drop-in session times in advance.

A demonstrator will be present during the above times. Demonstrators will help resolve general problems that you encounter in trying to carry out the project instructions, but will not assist in the actual project exercises.

1.3 Completing the projects

To carry out a project successfully, you need to master two ingredients: the actual mathematics of the topic under investigation, and the construction of the MATLAB commands needed to solve the relevant problems. Picking up the mathematics is probably a familiar activity that you practice when you attend a lecture or read your notes. Building up a repertoire of MATLAB commands and algorithmic ideas requires a perhaps different skill that in some ways is more akin to learning a language. There is a tendency to do things in an inefficient way to begin with, but eventually one achieves fluency in most practical situations.

Before you get started on a project, it is a good idea to glance through the exercises to try to appreciate what is being asked. To answer most of the exercises you will have to find the relevant commands that enable MATLAB to do what you want. There are clues and guidance given for this within each project, although it will often be necessary (or at least helpful) to consult the MATLAB help system.

Each project is divided into several exercises, and earns a total of 20 marks. The projects must be completed to your satisfaction and submitted electronically before the respective deadlines in weeks 6 and 9, according to the instructions given below. The marks will count towards Prelims and will not be released until after the Preliminary Examinations.

Your answers will ideally display both your proficiency in MATLAB and appreciation of some of the underlying mathematics.

Each project has some marks set aside for "MATLAB code which is elegant and concise". The lecturer will (try to) give examples of such during the MATLAB lectures this term. As always, the presentation of your work also counts towards your grade.

1.3.1 Getting help

You may discuss with the demonstrators and others the techniques described in the Michaelmas Term Students' Guide, the commands listed in the Hilary Term Students' Guide, and those found in the MATLAB help pages. You may also ask the the Course Director to clarify any obscurity in the projects.

The projects must be your own unaided work. You will be asked to make a declaration to that effect when you submit them.

1.3.2 Debugging and correcting errors

'Debugging' means eliminating errors in the lines of code constituting a program. When you first devise a program for an exercise, do not be too disheartened if it does not work when you first try to run it. In that case, before attempting anything else, type `clear` at the command line and run it again. This has the effect of resetting all the variables, and may be successful at clearing the problem.

If the program still fails, locate the line where the problem originates. Remove semicolons from commands if necessary, so that intermediate calculations are printed out and you can spot the first line where things fail. You may also want to display additional output; the `disp()` command can be useful. If the program runs but gives the wrong answer, try running it for very simple cases, and find those for which it gives the wrong answer. Remove all code that is not used in that particular calculation, by inserting comments so that MATLAB ignores everything that follows on that line.

Website

A copy of this manual can be found at:

https://courses.maths.ox.ac.uk/node/37598

This site will also incorporate up-to-date information on the course, such as corrections of any errors, possible hints on the exercises, and instructions for the submission of projects.

Legal stuff

Both the University of Oxford and the Mathematical Institute have rules governing the use of computers, and these should be consulted at https://www.maths.ox.ac.uk/members/it/it-notices-policies/rules.

Chapter 2

Preparing your project

To start, say, Project A, find the template 'projAtemplate.m' on the course website https://courses.maths.ox.ac.uk/node/37598 . Save this file as `projectA.m`, in a folder/directory also called `projectA`. Do not use other names.

You will be submitting this entire folder so please make sure it contains only files relevant to your project. You will almost certainly end up creating several `.m` files within this folder as part of your project.

2.1 Matlab publish

The file `projectA.m` should produce your complete answer. We will use the MATLAB 'publish' system.

```
publish('projectA.m','pdf')
```

This will create a PDF report in `projectA/html/projectA.pdf`. The lecturer will give examples of 'publish' in your MATLAB lectures and post an example file on the course website. You should also read '`help publish`' and '`doc publish`'.

The examiners will read this published report in assessing your project. It is important that the report be well-presented.

- Divide `projectA.m` into headings for each exercise (perhaps more than one heading for each exercise).
- You can and should call other functions and scripts from within `projectA.m`
- Make sure you answer all the questions asked using text in comment blocks—if it asks why explain why!
- Some questions ask you to create a function in an external file. A good way to make this code appear in your published results is to include '`type other_function.m`' where appropriate in your `projectA.m`.
- Include appropriate MATLAB output: don't include pages and pages of output, but you must show that you have answered the question. This will require some thought and good judgment but it's worth the effort to avoid losing points if the examiners cannot determine your answer.

The examiners may also run your various codes and test your functions.

Make sure you run publish one last time before submitting your project. Then double-check the results.

2.2 Zip up your files

Make a `projectA.zip` or `projectA.tar.gz` file of your projectA folder or directory including all files and subfolders or subdirectories. No `.rar` files please. It is highly recommended you make sure you know how to do this well before the deadline.

Double-check that you have all files for your project and only those files for your project.

2.3 Submitting the projects

Full instructions on the submission system will be emailed to you nearer the time. The projects are to be submitted electronically at https://courses.maths.ox.ac.uk/node/37598/assignments (from anywhere with internet access). Submission deadlines are given in Section 1.2. These deadlines are *strict*. It is *vital* that you meet them because the submission system will not allow submissions after the above times. You should therefore give yourself plenty of time to submit your projects, preferably at least a day or two in advance of the deadline. Penalties for late submission are specified in the Examination Conventions https://www.maths.ox.ac.uk/members/students/undergraduate-courses/examinations-assessments/examination-conventions.

You will need your University Single Sign On username and password in order to submit each project, and also your examination candidate number (available from Student Self-Service). If you have forgotten your details you must contact OUCS well before the first deadline. The system will only allow one submission per project.

Chapter 3

Solving nonlinear equations (Project A)

In this project, we investigate numerical methods to solve nonlinear equations.

Let $f : \mathbb{R} \to \mathbb{R}$ denote the following nonlinear scalar function

$$f(x) \coloneqq 2x + \frac{1}{2} e^{-x^2} \cos(2\pi x^3) \,. \tag{3.1}$$

Exercise 3.1. *Plot the function f in the interval $[-3, 3]$.*

This plot shows that there is a real number $x^* \in [-3, 3]$ such that $f(x^*) = 0$. In one lecture, we have seen that MATLAB has a built-in function to solve nonlinear equations: `fsolve`.

Exercise 3.2. *Use the `fsolve` function to find x^*. Then, add the point $(x^*, f(x^*))$ to the previous plot.*

We will now study an algorithm to compute x^* that is inspired by the intermediate value theorem. This theorem states that if a scalar function f is continuous in an interval $[a, b]$, and c is a real number in the interval $[\min(f(a), f(b)), \max(f(a), f(b))]$, then there is a real number $x \in [a, b]$ such that $f(x) = c$.

The intermediate value theorem is useful, because it implies that it is sufficient to evaluate f at two points a and b to find out whether f has a zero in the interval $[a, b]$.

Exercise 3.3. *Write an anonymous function `haszero = @(g, a, b) ...;` that returns 1 if, using the intermediate value theorem, it is possible to conclude that the continuous function g has at least a zero in the interval $[a, b]$, and returns 0 otherwise. Test the correct behaviour of your implementation by verifying that the function f from equation (3.1) has at least a zero in the interval $[-3, 3]$, and that f may not have a zero in the interval $[1, 3]$.*

Once `haszero` has been written, you should test it with functions whose behaviour you understand well (such as polynomials or trigonometric functions) in order to verify that it is indeed returning the correct answer. Next, it is time to refine the interval which contains x^*, which will be handled by the following exercise.

Exercise 3.4. *Write a function `shrinkInterval` whose inputs are a function g (which we assume is continuous) and two real numbers a and b, where $a \leq b$. If the function g has a zero in the interval $[a, (a+b)/2]$, then the function `shrinkInterval` must return the values a and $(a + b)/2$. Otherwise, if the function g has a zero in the interval $[(a + b)/2, b]$, then `shrinkInterval` must return the values $(a + b)/2$ and b. If you cannot guarantee that the function g has a zero in the interval $[a, b]$, then `shrinkInterval` must throw an error.*

Finally, plot the solution that you obtain after several iterations of `shrinkInterval`.

Exercise 3.5. *Create two variables $a = -3$ and $b = 3$ and add a black line that connects the points $(a, -2)$ and $(b, -2)$ to the previous plot. Then, write a `for-loop` that shrinks the interval $[a, b]$ 10 times using the function `shrinkInterval` (with $g = f$ from (3.1)). At each iteration, add a new black line that connects the points $(a, -2 + 0.2 * k)$ and $(b, -2 + 0.2 * k)$ to the previous plot, where k denotes the iteration number. What do you observe?*

Chapter 4

Nonlinear boundary value problems (Project B)

In this project, we investigate numerical methods to solve nonlinear boundary value problems. In particular, we consider the **Bratu equation**

$$\frac{\partial}{\partial x}\left(q(u(x))\frac{\partial}{\partial x}u(x)\right) = 0 \quad \text{for } x \in (0,1), \tag{4.1}$$

where $q : \mathbb{R} \to \mathbb{R}^+$ is a strictly positive and sufficiently smooth scalar function. Additionally, we require that the solution u to (4.1) satisfies the following *Dirichlet boundary conditions*

$$u(0) = 0 \quad \text{and} \quad u(1) = 1\,. \tag{4.2}$$

The simplest approach to solve such a boundary value problem numerically is to employ finite differences. For an $N \in \mathbb{N}$, let $h \coloneqq N^{-1}$, and define the uniform spatial grid $\{x_j\}_{j=0}^N$, where $x_j \coloneqq jh$. Using finite differences, the left-hand side of (4.1) can approximated as follows

$$\frac{\partial}{\partial x}\left(q(u(x))\frac{\partial}{\partial x}u(x)\right) \approx \frac{1}{h^2}\Big(q(u(x))u(x+h) - u(x)\big(q(u(x)) + q(u(x-h))\big) + q(u(x-h))u(x-h)\Big)\,.$$

Using this formula, we can approximate the solution to (4.1) and (4.2) by solving the following system of equations

$$\begin{cases} q(u_i)u_{i+1} - u_i\big(q(u_i) + q(u_{i-1})\big) + q(u_{i-1})u_{i-1} = 0 \quad \text{for } i = 1, 2, \dots, N-1\,, \\ u_0 = 0\,, \\ u_N = 1\,, \end{cases} \tag{4.3}$$

where u_i denotes an approximation of $u(x_i)$. Unfortunately, it is not straightforward to solve (4.3) because it is nonlinear. In this case, the standard approach is to employ **Picard iteration**, which works as follows: First, one select an initial guess $\{u_i^{(0)}\}_{i=0}^N$. Then, one computes a better approximation $\{u_i^{(1)}\}_{i=0}^N$ by solving

$$\begin{cases} q(u_i^{(0)})u_{i+1}^{(1)} - u_i^{(1)}\big(q(u_i^{(0)}) + q(u_{i-1}^{(0)})\big) + q(u_{i-1}^{(0)})u_{i-1}^{(1)} = 0 \quad \text{for } i = 1, 2, \dots, N-1\,, \\ u_0^{(1)} = 0\,, \\ u_N^{(1)} = 1\,. \end{cases} \tag{4.4}$$

The advantage of (4.4) is that it is linear in the unknown variables $\{u_i^{(1)}\}_{i=0}^N$. By introducing the symbol $\mathbf{u}^{(0)}$ to denote the vector with entries $\{u_i^{(0)}\}_{i=0}^N$ and the symbol $\mathbf{u}^{(1)}$ to denote the vector with entries $\{u_i^{(1)}\}_{i=0}^N$, we can summarize (4.4) in the following form

$$\mathbf{A}(q(\mathbf{u}^{(0)}))\mathbf{u}^{(1)} = \mathbf{b}\,, \tag{4.5}$$

where the matrix $\mathbf{A}(q(\mathbf{u}^{(0)})) \in \mathbb{R}^{(N+1)\times(N+1)}$ depends $q(\mathbf{u}^{(0)})$ and $\mathbf{b}$ is a vector whose first N entries are zeros, and whose last entry is 1.

Exercise 4.1. *Write a* MATLAB*-function* `A = createMatrix(u0, q)` *that, given vector* $\mathbf{u}^{(0)}$ *and an anonymous function* q*, returns the matrix* $\mathbf{A}(q(\mathbf{u}^{(0)}))$ *from equation* (4.5)*. The output should be in sparse format. To check the correctness of your implementation, verify that*

```
createMatrix(rand(3,1), @(u) ones(size(u)))
```

returns the following matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

Hint: to assemble the matrix `A`*, use the* MATLAB*-command* `spdiags`*.*

Exercise 4.2. *Let* $q(u) \coloneqq 1$*, and let* $\mathbf{u}^{(0)}$ *be a vector with 845 random entries. Compute* $\mathbf{u}^{(1)}$ *and plot it with using appropriate* x*-coordinates. If your implementation is correct, the result is a straight line that connects the points (0,0) and (1,1).*

Having computed $\mathbf{u}^{(1)}$, we can iterate the process and create a sequence of improved solutions $\{\mathbf{u}^{(k)}\}_{k\in\mathbb{N}}$ by solving

$$\mathbf{A}(q(\mathbf{u}^{(k)}))\mathbf{u}^{(k+1)} = \mathbf{b} \quad \text{for } k = 0, 1, 2, \ldots \tag{4.6}$$

Exercise 4.3. *Write a function* `uNext = PicardIteration(uPrevious, q)` *that performs one step of* (4.6)

Exercise 4.4. *Let* $q(u) = 1 + u^2$ *and* $\mathbf{u}^{(0)}$ *be a zero vector with 39 entries. Compute and plot in the same figure* $\mathbf{u}^{(0)}$*,* $\mathbf{u}^{(1)}$*,* $\mathbf{u}^{(2)}$*, and* $\mathbf{u}^{(3)}$*. To verify that your implementation is correct, in the same figure plot the reference solution using the provided function* `y = refsol(x)`*.*

Finally, we would like to investigate how quickly the sequence $\{\mathbf{u}^{(k)}\}_{k\in\mathbb{N}}$ converges to the true solution. In particular, we would like to assess the dependence of the approximation error on k and N. To quantify the error, we can use the following function

$$err(\mathbf{u}^{(k)}) = \max_{i=0,1,\ldots,N} \left| u_i^{(k)} - u(ih) \right|$$

(Remember that $h = N^{-1}$).

Exercise 4.5. *Let* $q(u) = 1+u^2$*. For* $N = 2^6, 2^7, \ldots, 2^{14}$*, compute the approximation errors of* $\mathbf{u}^{(1)}, \mathbf{u}^{(2)}, \ldots, \mathbf{u}^{(15)}$ *and store them in a* 15×9 *matrix. Then, plot the* $\log_2$ *values of this matrix using the* MATLAB*-command* `surf`*, and add suitable axis labels. Adjust the axis orientation using the* MATLAB*-command* `view`*. What do you observe?*

Chapter 5

Solving an initial value problem (Project C)

In this project, we experiment with several methods to solve the scalar initial value problem

$$y' = \sin(t^2)y \quad \text{and} \quad y(0) = 1 \tag{5.1}$$

using MATLAB.

Exercise 5.1. *Use* MATLAB*'s symbolic toolbox to compute the analytic solution to* (5.1)*. Then, plot this solution for t the time interval* $[0, 8]$.

By simple manipulations, it is easy to see that the exact solution to (5.1) is $y(t) = e^{\int_0^t \sin(x^2)\,\mathrm{d}x}$. Therefore we can easily evaluate the function $y(t)$ by approximating the integral $\int_0^t \sin(x^2)\,\mathrm{d}x$.

To approximate integrals, we can use the trapezium rule, whose formula reads

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2N} \sum_{k=1}^{N} \left(f\left(a + (k-1)\frac{(b-a)}{N}\right) + f\left(a + k\frac{(b-a)}{N}\right)\right)$$

Exercise 5.2. *Use the trapezium rule to approximate $\int_0^t \sin(x^2)\,\mathrm{d}x$ for $t = 0.1, 0.2, \ldots, 8$. Then, plot $y(t)$ using this approximation.*

MATLAB includes several routines to solve initial value problems numerically.

Exercise 5.3. *Use* MATLAB*'s built-in function* `ode23s` *to solve* (5.1) *in the time interval* $[0, 8]$*. Use the command* `odeset` *to set the absolute and relative tolerances to* 10^{-10}.

An alternative way to solve (5.1) is to use **Heun's method**. This method approximates the solution to a generic initial value problem $y' = f(t, y)$ and $y(t_0) = y_0$ using the formula

$$y_{i+1} = y_i + \frac{h}{2}\left(f(t_0 + hi, y_i) + f(t_0 + hi + h, y_i + hf(t_0 + hi, y_i))\right).$$

where h is a chosen step size and y_i is the approximation of $y(ih)$.

Exercise 5.4. *Write a function* `[t, y] = Heun(f, t0, T, h, y0)` *that read an anonymous function $f(t, y)$, an initial time t_0, a final time T, a step size h, and an initial value y_0, and returns the approximation of the solution to $y' = f(t, y)$, $y(t_0) = y_0$ computed with Heun's method. The output* `t` *is a vector with entries $t_0, t_0 + h, \ldots, T$.*

Then, compute and plot the solution to (5.1) *obtained with Heun's method using $T = 8$ and $h = 8/40, 8/60, 8/80$. What do you observe?*