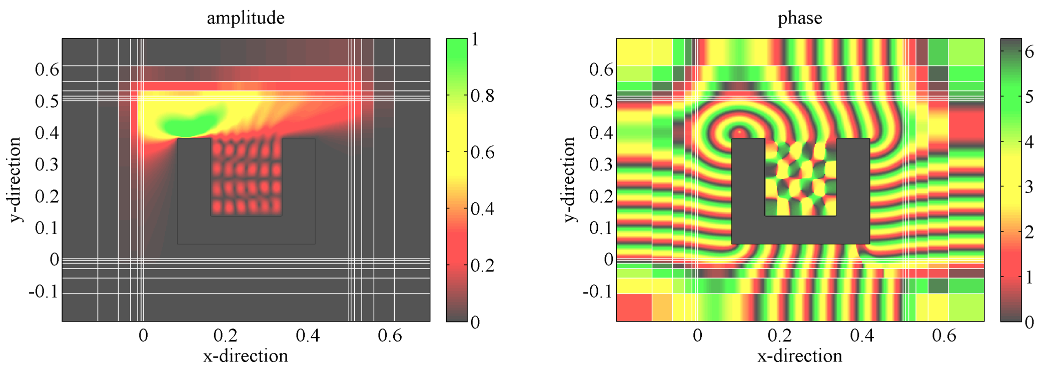
Introduction to MATLAB

Stefan Güttel

October 15, 2020

Contents



1 Introduction

In this course of 4 hours we will give an introduction to MATLAB. This course material is based on the MATLAB documentation which can be accessed by typing `doc` in the MATLAB command window. Another useful command is `help`. In combination with a function name it returns a help text for that function. Example: `help sum`

Apart from the excellent MATLAB documentation, there are many MATLAB tutorials and books available. Here are a few:

- The MathWorks website `https://uk.mathworks.com/` contains a well-managed support section with documentation, examples, and an active user community. Most importantly, *MatlabCentral* is a way to share useful MATLAB functions between users:

  `https://uk.mathworks.com/matlabcentral/`

- Cleve Moler, one of the MATLAB authors, has written two interesting books on numerical computing with MATLAB. These books are freely available at

  `https://uk.mathworks.com/moler.html`

- The *MATLAB Guide* by D. J. Higham and N. J. Higham (SIAM 2017) is another useful book to learn about MATLAB.

Familiarize yourself with the command window, workspace, command history, and navigation in the MATLAB documentation. MATLAB code can be stored in so-called *m-files* and it is convenient to use these files also during the development phase of a project. Everything that follows the character `%` is treated as a comment. Use the cursor keys to navigate up and down the history in the command window. Use the Tab key for auto-completion.

You can leave MATLAB by typing `exit` or closing the window. Long-running MATLAB codes can (often) be interrupted by pressing Ctrl+C. Example: `while 1, end`

Useful commands: `doc, help, clc, clear, clear all`

2 Matrices and Arrays

Learn how to create a matrix, the fundamental data structure in MATLAB, as an explicit list of elements, via index assignment, and using MATLAB commands. Learn how to suppress outputs to the command line. Understand the colon : operator. Understand how to concatenate matrices and how to delete rows/columns.

Commands: `[], [1,2,3], [1;2;3], A(1,2), +, -, *, /, .*, ./, sum, prod, transpose, diag, :, linspace, rand, randn, end, zeros, ones, eye, gallery, sparse, full, size, length`

Exercise 1: Create the matrix

```
A =
     0     0     0     0     0
     0     0     0     0     0
     5     4     3     2     1
```

with a single line of MATLAB code. Then assign random numbers to its 2nd row. Then assign the 1st row such that all columns sum to 0. Then reorder the columns reversely and transpose the matrix. (4 commands)

Exercise 2: Create the matrix

```
B =
     1     2     3     4     5
    -1    -2    -3    -4    -5
     1     2     3     4     5
    -1    -2    -3    -4    -5
     1     2     3     4     5
```

with a single line of MATLAB code.

Exercise 3: When accessing elements of a matrix with just a single index, the elements will be addressed columnwise starting with the first column. For example, `B(6)` with the above 5×5 matrix will return 2, the element in the $(1, 2)$ position. Create a sparse zero matrix C of size $n \times n$ via

```
n = 4;
C = sparse(n,n);
```

and populate its entries with 3 lines of MATLAB code such that

```
full(C)
ans =
     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
```

Do you recognize this matrix?

3 Expressions

In MATLAB numbers are just 1×1 matrices. There are some special numbers like `i`, `pi`, `eps`, `realmin`, `realmax`, `Inf`, `NaN` (the latter two are not case-sensitive). MATLAB allows to redefine these numbers as variables, which can cause confusion. So it is better to avoid such variable names. Nevertheless, it is good practice to write `1i` for the imaginary unit, as this will give the expected result even if `i` has been redefined. Variables do not need to be declared; they are instantiated with their first definition using the assignment operator `=`.

Example 4:

```
rho = 1.6180;

a = abs(3+4i)
a =
   5

z = sqrt(besselk(4/3,rho-1i))
z =
   0.3730+ 0.3214i

huge = exp(log(realmax))
huge =
   1.7977e+308

toobig = pi*huge
toobig =
   Inf
```

Note how the final `;` in the first line suppresses the output. In order to refer to the last output use the `ans` variable.

4 Basic Linear Algebra commands

MATLAB is a very powerful tool for Linear Algebra calculations. Matrices (and vectors) can be added and subtracted (`+` and `-`) and multiplied (`*`). Remember that matrix–matrix multiplication is different from element-wise multiplication (`.*`)!

One of the most important commands is the backslash \ (or equivalently, `mldivide`) for the solution of linear systems or least squares problems.

Other commands: `cond, rank, null, norm`

Exercise 5: Create a magic square via `A = magic(5)` and a vector `b = ones(5,1)`. Solve the linear system $Ax = b$, obtaining the numerical solution $\widetilde{x}$. What do you notice about the solution vector $\widetilde{x}$ and can you explain it? What is the exact solution x? Check numerically if the following inequalities are satisfied:

$$\frac{1}{\mathrm{cond}_2(A)}\frac{\|b - A\widetilde{x}\|_2}{\|b\|_2} \leq \frac{\|x - \widetilde{x}\|_2}{\|x\|_2} \leq \mathrm{cond}_2(A)\frac{\|b - A\widetilde{x}\|_2}{\|b\|_2}.$$

More commands: `qr, chol, schur, eig, eigs, svd, svds, fft`

Exercise 6: Familiarize yourself with the `eig` command and compute the eigenvectors and eigenvalues of the matrix A from Exercise 5. The *Perron theorem* implies that all eigenvalues λ_j of a matrix with positive entries and equal row sums σ satisfy $|\lambda_j| \le \sigma$, and that there is exactly one eigenvalue $\lambda_k = \sigma$. Verify this numerically.

5 Graphics

It is very easy to generate nice plots and animations with MATLAB. There are many functions for 2D/3D plots and images and we will only list a few of them.

Commands: `figure, close, hold, spy, plot, semilogx, semilogx, loglog, bar, hist, meshgrid, contour, contourf, surf, pcolor, image, imagesc`

Example 7: Plot the function $f(x) = \sin(x)$ and its zeros on $[0, 2\pi]$.

```
figure(1)
x = linspace(0,2*pi,200); % 200 typically enough for plotting
fx = sin(x);
plot(x,fx,'b-') % this plots a (b)lue line (-)
hold on
plot([0,pi,2*pi],[0,0,0],'ro') % this plots (r)ed circles (o)
```

There are also commands for changing the axes, and adding legends or titles. Some of the most useful ones are listed below.

Commands: `axis, xlim, ylim, xlabel, ylabel, legend, title, grid`

Example 8: Following the above example, we can make our plot more descriptive.

```
figure(1)
xlabel('x'); ylabel('f(x)');
legend('sin(x)','zeros')
title('a simple graph')
axis([0,2*pi,-1.2,1.2])
grid on
```

For generating a surface plot of a function $f(x, y)$ we need to create a parametrization of its (x, y) domain. This can be done conveniently using the `meshgrid` command.

Example 9: Plot $|\sin(x + iy)|$ over the domain $[-1, 7] \times [-1, 1]$.

```
figure(2)
x = linspace(-1,7,200); y = linspace(-1,1,200);
[X,Y] = meshgrid(x,y);
Z = X + 1i*Y;
F = sin(Z);
```

```
surf(X,Y,abs(F),'LineStyle','none')
axis tight
colorbar
```

6 Programming Scripts

MATLAB offers a very intuitive and clear programming language. Programs are typically written as *m-files* in the editor. When working with MATLAB interactively (as we have done so far), it is recommended to type everything which requires more than one line in the editor. This allows one to modify and debug the code very efficiently. Saved m-files can be called by their name from the MATLAB command line or by pressing Ctrl+Enter.

Keywords: `for`, `while`, `if else`, `continue`, `break`, `switch case`

Logical: `==`, `&&`, `||`, `~`, `all`, `any`, `isempty`

Example 10: Fibonacci sequence (`fibonacci.m`)

```
%% clear workspace and close figures
close all; clear all; clc

%% initialize variable f to store Fibonacci numbers
f = zeros(20,1); f(1) = 0; f(2) = 1;

%% Fibonacci sequence
for j = 3:length(f)
    f(j) = f(j-2) + f(j-1);
    if f(j) <= 10
        str = sprintf('%2drd Fibonacci number is smaller/equal to 10',j);
    else
        str = sprintf('%2drd Fibonacci number is greater than 10',j);
    end
    if j == 19, str = [ str ' --- puh, almost done...' ]; end
    disp(str)
end

%% semilogy plot of sequence and golden section growth
semilogy(f); hold on
q = (1 + sqrt(5))/2;
semilogy(q.^(1:length(f)),'r--')
xlabel('n')
legend('Fibonacci sequence f_n','1.6180^n','Location','NorthWest')
```

Exercise 11: Use `meshgrid` to create a grid of complex numbers $z = x + iy$ with $x \in [-2, 1]$ and $y \in [-1, 1]$. For each number z in this grid run 200 iterations of the Mandelbrot recursion $r_{n+1} = r_n^2 + z$ starting with $r_0 = 0$. Use the `pcolor` function to visualize $|r_{200}|$ for each number z. Challenge for MATLAB gurus: no more than 10 lines!

7 Functions

Essentially, MATLAB functions are m-files that accept input arguments and produce outputs. In its simplest form, a function is defined in an m-file with the same name.

Example 12: If this is the content of an m-file with name `quadroots.m`:

```
function [z1,z2] = quadroots(p,q)
%QUADROOTS  Compute the solutions of z^2 + p*z + q = 0.
%   Given the scalar inputs p, q this functions returns
%   the roots z1 and z2 of a quadratic equation.
  z1 = -p/2 - sqrt(p^2/4 - q);
  z2 = -p/2 + sqrt(p^2/4 - q);
  return % not necessarily required
end
```

then we can call it by its name, e.g., like

```
z1 = quadroots(0,1); % z2 will be ignored
[z1,z2] = quadroots(0,1);
```

Note that `help quadroots` will output the comments following the `function` keyword.

Many MATLAB functions are implemented as m-files and can be viewed by using the `edit` command. For example, `edit magic` will open the implementation of the function we used above for generating magic squares.

Another powerful construct are so-called *inline functions*, which are single-line functions returning a single output. They are defined using the `@` operator. For example, by grouping the two roots z_1, z_2 in a two-dimensional vector, we can rewrite `quadroots` as

```
quadroots1 = @(p,q) [ -p/2-sqrt(p^2/4-q) , -p/2+sqrt(p^2/4-q) ];
```

Or even shorter:

```
quadroots2 = @(p,q) -p/2 + [-1,1]*sqrt(p^2/4 - q);
```

8 Numerical Integration and Differential Equations

MATLAB provides many routines for standard tasks in computing, ranging from elementary math operations, over linear algebra, statistics and random numbers, interpolation, optimization, Fourier analysis and filtering, sparse matrix computation, to computational geometry. More information can be found, as always, in the MATLAB documentation (type `doc`). In this section we will discuss with the help of two examples some functions for numerical integration and the solution of differential equations.

There are a number of routines for calculation definite integrals $\int_a^b f(x)\,\mathrm{d}x$ and their multi-dimensional extensions. These MATLAB function require as their first input a *function handle* to $f(x)$ like, e.g., an inline function as discussed above. Note that this function should be *vectorized*, i.e., it should be evaluatable for vector arguments x.

Commands: `integral, integral2, integral3` (see also `quad`)

Example 13: Integrate $f(x) = \exp(-x^2)\log(x)^2$ from 0 to $+\infty$:

```
f = @(x) exp(-x.^2).*log(x).^2; % vectorized!
q = integral(f,0,Inf,'AbsTol',1e-6,'RelTol',1e-6)
```

MATLAB also provides a collection of integrators for ordinary differential equations.

Commands: `ode45, ode15s, ode23, odeset, odeexamples`

Exercise 14: Solve the 1D heat equation on an interval, discretized in space by second-order finite differences, and in time by MATLAB's `ode15s`. More precisely, solve

$$u_t(t,x) = u_{xx}(t,x), \quad x \in [0,L], \quad t \in [0,T],$$

with given initial and boundary data

$$u(t=0,x) = b(x), \quad u(t,x=0) = 0, \quad u'(t,x=L) = 0.$$

Let us define equispaced spatial grid points $x_j = jh$, $h = L/n$, $j = 1,\ldots,n$. A finite difference approximation of the second derivative of a function $u(x)$ at $x = x_j$ is

$$u''(x_j) \approx \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}, \quad j = 1,\ldots,n-1,$$

with u_j denoting approximations to $u(x_j)$ and the convention that $u_0 = 0$. For $x = x_n$ we can use the approximation

$$u''(x_n) \approx \frac{-2u_j + 2u_{j-1}}{h^2}.$$

1. Define variables $L = 1$, $T = 1$, $n = 100$, and an inline function $b(x) = x(L-x)^2$.

2. Create a column vector `xx` of the grid points $[x_1, x_2, \ldots, x_n]^T$ using the colon operator `:`. Define a vector `u0` of the initial data, $[b(x_1), b(x_2), \ldots, b(x_n)]^T$. (If the inline function for $b(x)$ is vectorized this can be done via `u0 = b(xx)`.)

3. Create the $n \times n$ matrix A which maps column vectors u to the finite difference approximations of the second derivative (i.e., Au will be an approximation for $u''(x)$ at the grid points $x_1, x_2, \dots, x_n$). You can use the command `gallery('tridiag',n)`, which will generate a sparse tridiagonal matrix

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n}.$$

4. Define the inline function `f = @(t,u) A*u` and a vector `tt` of 100 equidistant time points in $[0, T]$. Then call

```
[tt,uu] = ode15s(f,tt,u0);
figure
for j = 1:length(tt)
   plot(xx,uu(j,:));
   title(['t = ' num2str(tt(j))])
   axis([0,L,0,norm(u0,inf)])
   shg; pause(0.2)
end
```

   This will create an animation of the solution of the heat equation.

5. You may also want to try other MATLAB integrators like, e.g., `ode45`, and compare which one works best for this problem.

9 Useful tools and tricks

Timings and profiler: The `pause` command will pause the program execution until a key is pressed. With `pause(1)` one can pause a program for 1 second. The commands `tic` and `toc` can be used to measure time that passed in between. Example: `tic; pause(1); toc`

The MATLAB profiler is an extremely valuable tool for speeding up complex programs, as it allows one to identify the computationally most intensive parts of a code. It is activated via `profile on`. After program execution one types `profile report` to obtain the report. The profiler is cleared or deactivated via `profile clear` or `profile off`, respectively.

Vectorization and memory allocation: Operations on arrays should be vectorized whenever possible. Somewhat oversimplified this boils down to "avoiding `for` loops", because these can extremely slow down your code. Also, it is important to allocate memory for large arrays in advance, as a dynamic resize may cause noticeable overhead.

Example 15: Compute the square root of $5 \cdot 10^6$ random numbers.

```
clear all
v = rand(5e6,1);

tic; % no memory allocation and no vectorization
for j = 1:length(v)
    s1(j) = sqrt(v(j));
end
t1 = toc;

tic; % with memory allocation but without vectorization
s2 = zeros(size(v));
for j = 1:length(v)
    s2(j) = sqrt(v(j));
end
t2 = toc;

tic; % with memory allocation and vectorization
s3 = sqrt(v);
t3 = toc;

% compare the timings
bar([ t1 , t2 , t3 ])
```

Exercise 16: Run the above code through the profiler and find the line where most of computation time is spent on.

Publishing MATLAB code: When publishing MATLAB code in LaTeX inside the `verbatim` environment, it is recommended to include the package `upquote` to make sure all quotes `'` can be copied-and-pasted correctly into the MATLAB command window (this is what one gets without this package: ’).

Another option for publishing MATLAB in LaTeX is the `mcode` package, which highlights keywords, comments, and strings in different colors (as is done in this document).

Graphics workflow: When writing a paper or thesis, often a lot of time is spent on generating graphics. It is therefore worth spending some time in advance for optimizing the workflow of including figures in your work. I recommend to generate all graphics with a dedicated script and to include these directly into the LaTeX file without generating copies. (Warning: multiple instances of the same file may cause severe headaches!)

10 Assignment

Task: Implement and test the Runge-Kutta method and numerically solve an ordinary differential equation. Make sure to comment your work by adding MATLAB comments.

1. Copy and paste the following MATLAB code into an m-file called `rk4.m`:

```
function [tt,yy] = rk4(f,tt,y0)
%RK4  Solve y' = f(t,y) with initial value y0 using Runge-Kutta.
%  The argument f is a function handle with input parameters (t,y).
%  The vector tt contains the time points at which to solve the ODE.
%  The row vector y0 is the initial value at tt(1).
%  Each column of the output matrix yy corresponds to the Runge-Kutta
%  approximation of y at each time point in tt.

%  ... ADD YOUR CODE HERE ...

end
```

2. Modify `rk4.m` for implementing the classical Runge-Kutta method defined via

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_1) \\ k_3 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + h\,\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}, \end{aligned}$$

where at $h = t_{n+1} - t_n$ at each time step. (This can be done in less than 10 lines of code, but doesn't need to!)

3. Create a new m-file called `test1.m` to test your Runge-Kutta implementation. To this end consider the simple initial value problem

$$y'(t) = -y, \quad y(0) = 1, \tag{1}$$

to be solved at equispaced time points $t = 0, 0.1, 0.2, \dots, 2$. Visually compare your numerical solution $\widetilde{y}(t)$ (the output `yy`) with the exact solution $y(t) = e^{-t}$ for $t \in [0, 2]$ by plotting both in the same figure. (Hint: Both curves should look almost identical.)

4. Create a new m-file called `test2.m` to solve the same test problem (1) over the time interval $[0, 2]$ and measure the error of the computed solution $\widetilde{y}(t)$ at $t = 2$ compared to the exact solution $y(t) = e^{-t}$, i.e., evaluate err $= |\widetilde{y}(2) - y(2)|$. Do this for $n = 5, 10, 15, \dots, 50$ equal time steps in $[0, 2]$ and plot the error versus n. It's best to use a `loglog` plot for this. (Hint: You should see a straight line and the error should decrease approximately like C/n^4 for some constant C.)

5. Attach the three files `rk4.m`, `test1.m`, `test2.m` to an email and

   send to `stefan.guettel@manchester.ac.uk` by **November 13th, 2020**.

   You should receive a confirmation of submission via email until November 16th. If you do not receive this confirmation, you need to get in contact immediately.

   Note that the submission deadline is absolutely strict.
   Aim to submit your work a week before that deadline.
   Late submissions will not be accepted!