MA2501: Numerical Methods
Spring 2016

Solutions to exercise set 0

Norwegian University of Science and Technology
Department of Mathematics

This set of exercises was meant to give a short introduction into the usage of MATLAB.

1 Linear algebra and plotting:

Find and plot the polynomial of degree 3 that interpolates the points given in the following table:

i	1	2	3	4
x_i	-2	0	1	3
y_i	-16	-3	-1	24

In other words: Find a polynomial

$$p(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

that satisfies $p(x_i) = y_i$ for $i = 1,\ 2,\ 3,\ 4$.

a) Verify that the coefficients satisfy the linear system

$$\begin{pmatrix} 1 & -2 & 4 & -8 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 3 & 9 & 27 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} -16 \\ -3 \\ -1 \\ 24 \end{pmatrix}.$$

b) Use MATLAB to solve the linear system.

c) Use MATLAB for plotting the interpolation polynomial.

Possible solution:

The solution of the linear system is $(a_0, a_1, a_2, a_3) = (-3, 3/2, -1/2, 1)$ and thus

$$p(x) = x^3 - \frac{1}{2}x^2 + \frac{3}{2}x - 3.$$

It can be obtained in MATLAB with:

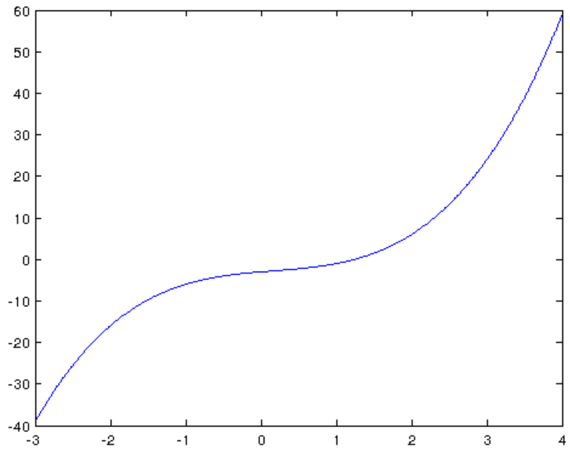
`A = [1,-2,4,-8;1,0,0,0;1,1,1,1;1,3,9,27];`	define the matrix
`b = [-16;-3;-1;24];`	define the vector
`a = A\b`	solve the equation, store it as the variable `a`, and show it

Note that it is important to keep track of the correct dimensions: The variable `b` above is a 4×1 vector. Also note that the semicolon (`;`) at the end of a line surpresses the visual output of the result of a calculation.

The function p can (in the possibly interesting interval $[-3, 4]$) be plotted with:

`x = [-3:0.01:4];`	discretise the interval $[-3, 4]$
`p = -3 + 1.5*t - 0.5*t.^2 + t.^3;`	evaluate the function at the discretisation points
`plot(t,p)`	a simple plot

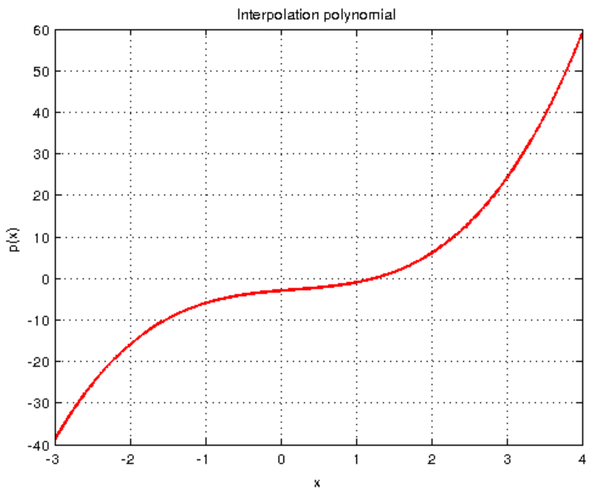
This yields the following:



Now it is possible to play around with the result a bit. For instance:

`plot(t,p,'Color','red','LineWidth','2');`	change color and line width
`xlabel('x');`	add a label to the x-axis
`ylabel('p(x)');`	add a label to the y-axis
`title('Interpolation polynomial');`	add a title
`grid on;`	add a grid

yields



2 **Some simple programming:**

Euler's number e can, for instance, be computed using either of the formulas

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

or

$$e = \sum_{k=0}^{\infty}\frac{1}{k!}.$$

a) Write two MATLAB-programs that compute the numbers

$$a_n = \left(1+\frac{1}{n}\right)^n$$

and

$$b_m = \sum_{k=0}^{m}\frac{1}{k!}$$

for different values of n and m and compare the results with the true value of e.

b) One of the two methods does not seem to converge to e. Which one? Why?

Possible solution:

a) A program for the first method can for instance be:

```
function a = myeuler1(n)
a = (1+1/n)^n;
```

A possibility for the (slightly more complicated) second method is:

```
function b = myeuler2(m)
c = 1;
b = 1;
for k = 1:m
  c = c/k;
  b = b + c;
end
```

A different possibility that takes advantage of the capabilities of MATLAB of working with vectors and the inbuilt function `factorial` is:

```
function b = myeuler3(m)
b = sum(1./factorial(0:m));
```

b) Testing the second program, we see[1] that the result does not change for $m \geq 17$ and in fact coincides with the result of the computation `exp(1)`.

In contrast, the first program requires a fairly large number n to yield a reasonable result. For $n = 100$, the error is about 10^{-2}, for $n = 10^4$, it is about 10^{-4}, finally, for

[1] Usually MATLAB only shows 5 significant digits. Using the command `format long`, one can increase this to 15 digits for double precision.

$n = 10^8$ it is of the order of 10^{-8}. Increasing n further, however, tends to decrease the accuracy: If we choose $n = 10^{12}$, then the error increases to about 10^{-4}.

This behaviour can be explained by understanding that the total error of the program can be decomposed into two parts: first, the approximation error, which comes from the fact that the formula is only exact for "$n = \infty$", and, second, computational (i.e., rounding) errors, which come mainly from the fact that the division $1/n$ is, in general, inexact. Now note that the division $1/n$ can be performed exactly, if n is some power of 2. Indeed, choosing $n = 2^{40}$ (which is about the same as 10^{12}) yields an error of about 10^{-12}. Choosing $n = 2^{52}$, we basically obtain an exact result. If, however, we choose $n = 2^{53}$, then $1 + 1/n$ is indistinguishable from 1 in double precision. Thus the result of the algorithm for the input $n = 2^{53}$ is simply 1.

3 Consider the following two segments of pseudocode:

Program A:

Data: a vector $a = [a_0, a_1, \dots, a_n]$ of real numbers, a real number x;
Output: a real number y;
Initialization: $y \leftarrow a_0$;
for $k = 1$ **to** n **do**
  $y \leftarrow y + a_k x^k$;
end

Program B:

Data: a vector $a = [a_0, a_1, \dots, a_n]$ of real numbers, a real number x;
Output: a real number y;
Initialization: $y \leftarrow a_n$;
for $k = n - 1$ **to** 0 **by** -1 **do**
  $y \leftarrow a_k + xy$;
end

a) What do these programs actually do?

b) In theory, both programs should yield the same result. Can they be expected to do so also numerically?

c) Which of the programs is usually preferable?

Possible solution:

The loop in the first program (obviously) gives the result

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

In the second program we obtain

$$y = a_0 + x\big(a_1 + x\big(a_2 + x\big(a_3 + \cdots + x\big(a_{n-1} + x a_n\big)\cdots\big)\big)\big),$$

which can, again, be simplified to

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Thus both programs provide the same result, if all operations are performed without rounding errors. Because rounding errors usually cannot be avoided, the *numerical* results can be expected to be different, though.

In order to answer the question, which of the programs is preferable, we first have to clarify what "preferable" actually means—and there are several possible interpretations:

1. Can one of the two programs be expected to yield more accurate results?
2. Is one of the two programs most probably faster?
3. Are there noticeable differences in memory usage?

Next we look at these points separately:

1. From the viewpoint of accuracy of the result, at first glance none of the two methods is *obviously* better. Rounding errors in the form of cancellation may occur in both programs in their main steps (either the calculation of $y + a_k y^k$ or $a_k + xy$).
2. The situation is different, however, if one counts the number of operations:

   Program B requires in each iteration one multiplication and one addition, totalling in n multiplications and n additions.

   For Program A, the total number of additions is again n, but the number of multiplications is larger. At first glance, the computation of x^k seems to require $k - 1$ multiplications. This would amount to $0 + 1 + 2 + \ldots + n - 1 = n(n - 1)/2$ multiplications. Exploiting the fact that, for instance, $x^4 = (x^2)^2$, this number can be decreased quite a bit. Even more, it would be possible (and very sensible) to keep the value x^k in the memory and to compute x^{k+1} in the next step by multiplying the stored value with x. The main code line would then be replaced by something like

   $$\begin{aligned} z &\leftarrow xz, \\ y &\leftarrow y + a_k z. \end{aligned}$$

   Still, this requires two multiplications in each step, leading to a total of $2n$ (or $2n-1$ if one discounts the unnecessary first one).
3. Apparently, memory usage is no real issue in both programs.

Roughly spoken, these considerations imply that the second program is almost twice as fast as the (optimal implementation of the) first program without sacrificing any accuracy. Thus it should in general be preferred.

Program B is usually known as *Horner's Algorithm* (see also Cheney and Kincaid, pp. 8 sqq.).