Numerical Analysis – Lab 1

1 Programming with MATLAB

Instruction. This is a quick start. There is also a mini-course available on the learning mall.

Variables. In MATLAB (MATrix LABoratory), all quantities are matrices. A scalar is simply a 1×1 matrix and a vector is an $n \times 1$ (or $1 \times n$) matrix. To assign a value to a variable, just type the name of the variable, =, and then the value. So

```
>> a=6.2
```

assigns the value 6.2 to the variable a, while

```
A=[1  4  5; 6  7  3]
```

defines A to be the 2×3 matrix $\begin{bmatrix} 1 & 4 & 5 \\ 6 & 7 & 3 \end{bmatrix}$ and

```
v=[5 ; 7 ; 1 ; 2]
```

defines the vector $\begin{bmatrix} 5 \\ 7 \\ 1 \\ 2 \end{bmatrix}$. To see the value of a variable, just type its name. To see what variables are currently stored in memory type `who`. Typing `A(2,1)` will display the entry (6) in row 2 , column 1 , of A, while typing `v(3)` will show the third entry (1) of `v` . `A(:, 2)` displays the second column of A, and `A(2,:)` displays the second row. MATLAB has a command to define a (row) vector with evenly spaced entries. The vector $\begin{bmatrix} 1 & 1.2 & 1.4 & 1.6 & 1.8 & 2 \end{bmatrix}$ is entered as

```
>> v=1: .2: 2
```

where `.2` is an abbreviation of 0.2 with the zero integer part omitted. In general `a: h: b` is the vector with components $a, a+h, a+2h, \ldots, a+nh$, where n is the largest integer such that $a+nh \leq b$ (h doesn't need to divide $b-a$). If h is omitted, 1 is used, so the vector $\begin{bmatrix} 5 & 6 & 7 & 8 & 9 \end{bmatrix}$ is entered as `5: 9`.

Basic Algebraic Operations. MATLAB can be used like a calculator to perform basic operations of addition (+), subtraction (-), multiplication (*), division (/) and exponentiation ($\wedge$). For instance, to compute $1.6 \cdot 7 + \frac{2.3}{1.4} - 7.3^2$, enter

```
>> 1.6*7 + 2.3/1.4 - 7.3^2
```

When the quantities involved are vectors or matrices, addition and subtraction are componentwise, while multiplication is matrix multiplication (recall that the matrix product AB is defined as the matrix of dot products of rows of A with columns of B, so the number of columns of A must equal the number of columns of B). In the example below, notice that AB is not defined so MATLAB gives an error message. Also notice that B^3 is the matrix product BBB, while $A^3 = AAA$ is not defined since A is not a square matrix.

```
>> A = [1  2  3; 3  2  1]
>> B = [3  5; -1  1]
>> A * B
>> B * A
>> B^3
>> A^3
```

In addition to these basic operations, MATLAB also can perform component-wise multiplication (.∗), division (./) and exponentiation (.∧). These operations are useful when we wish to to the same algebraic operations with a list of numbers (i.e. a vector). For instance, if we wanted to square all the numbers in the vector $\begin{bmatrix} 1 & 1.2 & 1.4 & 1.6 & 1.8 & 2 \end{bmatrix}$ we would enter the following.

```
>> v=1: .2: 2
>> v.^2
```

Functions. MATLAB has numerous built-in functions. A function is simply a program that takes some input and returns an output. All the standard elementary functions such as `sin`, `cos`, `tan`, `exp` (e^x), `log` (which means the natural log: $\ln(x)$), and so forth, are included in MATLAB's family of functions. Other useful functions are `sqrt`, `sign` (sign(x) is 1 if $x > 0$, 0 if $x = 0$ and -1 if $x < 0$), `det` (determinant of a square matrix), `inv` (inverse of a square matrix) and `factor` (gives the prime factorization of an integer). To obtain information about any function type `help` followed by the function name. A description of the function will appear, including the type of input expected and what the output will be. It is also possible to define your own functions in MATLAB using the inline command. For example

```
>> f=inline('x*sin(x) - exp(3*x)')
```

defines the function $f(x) = x\sin(x) - e^{3x}$. (**If you copy the code to Matlab, you need to retype the quote ' otherwise MATLAB would complain that it is a bad character.**)

Simple Plotting. To plot a set of points $(x_1, y_1), \ldots (x_n, y_n)$ in MATLAB, first enter the x and y values into two separate vectors x and y. Then type `plot(x, y)`. For example to plot $(0, 3), (1, 2)$ and $(2, 5)$:

```
>> x=[0  1  2]
>> y=[3  2  5]
>> plot(x, y)
```

MATLAB's default is to connect the dots with straight line segments. To plot only the points, type

```
>> plot(x, y, '.')
```

To plot the graph of a function, we first need to select a set of points on the graph to plot. It is usually wise to choose a large number of points so that the graph looks smoother. For example, to plot the graph of $f(x) = x^3 - x$ on the interval $[-1, 2]$, we might choose points with x-coordinates spaced 0.05 units apart.

```
>> x=-1: .05: 2;
>> y=x.^3 - x;
>> plot(x, y)
```

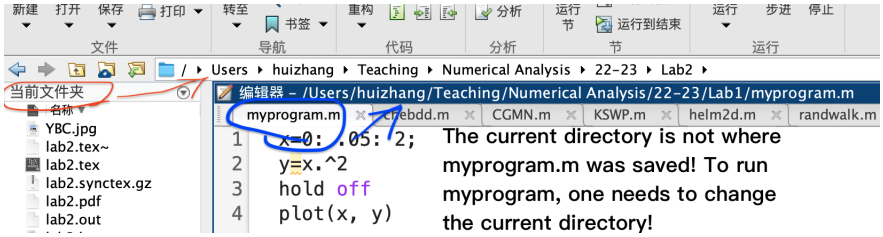
(The semicolon at the end of a command tells MATLAB not to display the output of the command.) To plot multiple graphs together type `hold on` after plotting the first graph.

```
>> hold on
>> z=3*x-x.^2;
>> plot(x, z, 'r')
```

The option `'r'` tells MATLAB to plot the graph in red to distinguish from the first graph.

Programming. MATLAB programs are usually saved as m-files using a filename of the form `filename.m` and can then be executed by typing `filename` (without the `.m`). When a program is executed, MATLAB simply executes each command in the program line-by-line. For example, if we save the lines

```
x=0: .05: 2;
y=x.^2
hold off
plot(x, y)
```



in a file named `myprogram.m`, and **ensure in MATLAB the current directory is where `myprogram.m` was saved**, then typing `myprogram` in MATLAB will execute the four lines and produce a plot of the function x^2 over the interval $[0, 2]$.

One of the most important concept in programming is that of a loop. A loop is simply a way for a program to execute the same commands multiple times. For example, the following program computes the sum of the squares of the first 10 integers.

```
s=0
for n=1: 10
    n
    s=s+n^2
end
```

The program begins by setting $s = 0$. The second line tells MATLAB to let the variable n run from 1 through 10. For each of these values of n, MATLAB then executes the two lines between the `for` and the `end`. So it first displays n, then adds n^2 to s. It then increments n by 1 and repeats this process, finally stopping after the 10^{th} time through. The `for` and `end` define the scope of the loop, so everything between these lines is executed with each iteration of the loop. As another example, consider a sequence defined recursively by $x_0 = 1$ and $x_{n+1} = 1 - \frac{1}{2}x^2$. Here is a program that will list the terms of the sequence up to x_{30}.

```
format long
format compact
x=1
for n=1:30
    x=1-(1/2)*x^2
end
```

The `format long` and `format compact` commands tell MATLAB to display values with less spacing and with 15 significant figures. Another type of loop is the `while` loop, which executes the commands within the loop while a given condition holds. For instance, the harmonic series $\sum_{n=1}^{\infty} \frac{1}{n}$ diverges, so its partial sums approach infinity. What is the first partial sum that is greater than 5 ?

```
s=0
n=0
while s<=5
   n=n+1
   s=s+1/n
end
```

The 83^{rd} partial sum is the first to exceed 5. Loops may also be nested, meaning that there is a loop within a loop. The following program is an example of this. It computes the first n rows of Pascal's triangle.

```
n=input('Number of rows=');
clear A
for i=1 : n
    A(i, 1)=1;
    A(i, i)=1;
    for j=2: i-1
        A(i, j)=A(i-1, j-1)+A(i-1, j);
    end
end
A
```

Save this as `pascaltri.m` and type `pascaltri`. The input statement asks you to enter the number of rows desired. In the main for loop, the counter i ranges from 1 to n and represents the row that is being computed. The lines `A(i, 1)=1;` and `A(i, i)=1;` set the first and last entries of that row equal to 1. To fill in the remaining entries of row i, a second for loop has j range from 2 to $i-1$. Within this inner loop, the j^{th} entry of row i is computed using the recursion relation for the triangle (each entry is the sum of the two adjacent entries in the previous row). The output is an $n \times n$ matrix, where MATLAB has filled in the extra entries that weren't part of the triangle with zeros.

Many programs are functions, where an input is provided and the result of executing the program is an output. The first line of the m-file that defines a function must be of the form

`function` *output variables = functionname (input variables)*

and the name of the m-file must be `functionname.m`. For example:

```
function y=g(x)
if x<0
    y=x^2;
else
    y=sin(2*x);
end
```

This program, which would be saved as `g.m`, consists of a single conditional statement that tells MATLAB to compute x^2 if $x < 0$, or compute $\sin(2x)$ otherwise. To call the function `g`, ensure the current directory is where `g.m` was saved, then type e.g. `g(0.1)` in MATLAB. The following function `pfactor` returns the prime factorization of the input n as a vector factors.

```
function factors=pfactor(n)
factors=[];
for k=2:n
    while rem(n,k)==0
         factors=[factors k];
         n=n/k;
    end
end
```

The program begins by setting `factors` equal to the empty vector `[]`, and then runs through all the possible factors k from 2 to n, dividing out by k and adding it to the list of factors if the remainder upon dividing n by k is zero. (Note: There are ways to speed up factoring algorithms. MATLAB's `factor` function does the same thing as this program, but faster.) As another example, suppose we want to approximate the derivative of a function f on some interval $[a, b]$. By the definition of the derivative

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

so when h is small, $f'(x)$ can be approximated by the expression $\frac{f(x+h)-f(x)}{h}$. The program

```
function df=derivative(f,x,h)
df=[]
for k=1:length(x)
     df(k)=(f(x(k)+h)-f(x(k)))/h;
end
```

takes as input an inline function `f`, a vector `x` of x-values, and `h`, and produces as output a vector `df` consisting of difference quotient approximations to $f'(x)$ at the x-values. Entering the following will define $f(x) = x^4$ and use the `derivative` function to compute the difference quotient approximations of f using $h = 0.01$ at points in $[-1, 1]$ spaced 0.1 apart.

```
>> f=inline('x^4')
>> x=-1: .1: 1;
>> g=derivative(f, x, .01)
```

The difference quotient approximations can then be compared to the actual derivative $f'(x) = 4x^3$

```
>> plot(x,4*x.^3)
>> hold on
>> plot(x,g,'r')
```

Notice that the graphs are pretty close, but do not match perfectly. Plotting the difference between the approximation and the actual derivative gives a better illustration of the error.

```
>> hold off
>> plot(x,g-4*x.^3)
```

The error is nearly zero at $x = 0$, and around 0.06 near $x = \pm 1$.

Exercises (optional). (The reference programms can be found on the learning mall.)

1. Write a program to compute the first 25 terms in the sequence $x_n = n^2 \sin\left(n^2\right)$.
2. Write a program to compute the first 25 terms in the sequence defined recursively by $x_0 = 1$, and $x_{n+1} = 3\cos(x_n)$.
3. Write a program to compute the n^{th} partial sum of the series $\sum_{k=1}^{\infty} \frac{1}{k^2}$. Your program should take n as input and produce the n^{th} partial sum as its output. Use your program to compute the $10^{th}, 100^{th}$ and 1000^{th} partial sums.
4. (a) Write a program to compute the product of the first n odd numbers.

   (b) Denoting the n^{th} such product p_n, write a program to compute the sum of p_1 through p_n.
5. Recall that the right-hand sum of a function f on an interval $[a, b]$ using n subintervals is

$$R_n = \sum_{k=1}^{n} f(x_k)\,\Delta x$$

   where $\Delta x = (b - a)/n$ and $x_k = a + k\Delta x$. Write a function `riemannsum` that takes as input a function f, endpoints a and b, and number of subintervals n, and produces the right-hand sum as its output.
6. Apply the program in Exercise 5 to the function $f(x) = e^{-x}$ on $[0, 2]$ using the following values of $n : 10, 100, 1000$. Compare these with the value of the integral $\int_0^2 e^{-x} dx$.

2 Round-off errors

Rounding. Note that the rounding already takes place when we input the data to the machine. To see this effect, let us try to compute $10^{16} \bmod(2\pi)$ ($r := x \bmod y$ for $x > y > 0$ is the smallest non-negative number such that $(x - r)/y$ is an integer) in MATLAB:

```
>> mod(1e16,2*pi)
```

where `1e16` is an abbreviation of 10^{16} on the machine. The answer returned by MATLAB on my computer is 2.6372. Let us check whether it is consistent:

```
>> sin(mod(1e16,2*pi))
>> sin(1e16)
```

which returned 0.4832 and 0.7797 on my computer: inconsistent! We can compute it with much higher precision using the function `vpa`:

```
>> digits
>> mod(vpa(1e16),vpa(2*pi))
```

where `vpa` tells MATLAB to store the input with high precision (vpa is a short-hand for variable precision arithmetic), and `digits` shows the default significant decimal digits to be stored on the machine for the high precision. This time I got 2.2474... Let us check the consistency again:

```
>> sin(mod(vpa(1e16),vpa(2*pi)))
>> sin(vpa(1e16))
```

which returned 0.7796880.. twice (the same). A questions arises: which of the above results can be trusted? Let eps be the machine precision used. By default, MATLAB uses the double precision for which eps= $2^{-53} \approx 10^{-16}$, and for the default `vpa` precision we have eps= 10^{-32}. The input 10^{16} stored on the machine has the round-off error about 10^{16}eps, which would be 1 in double precision but 10^{-16} in `vpa` precision. (It is still a mystery to me how MATLAB gave a reasonably good result for `sin(1e16)`.)

Swamping. Next let us see one type of round-off effect in the addition arithmetic, known as swamping error. First, try to compute $1 + 10^{-16}$:

```
>> 1 + 1e-16
```

What do you see? The answer from MATLAB is 1. The small number 10^{-16} was swamped by the large number 1. This itself seems harmless, but:

```
>> (1 + 1e-16) - 1
```

gives 0 and the relative error 100%. To see one more implication of swamping, let us now compute the partial sum of the series $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$ using

```
function s = psum(n)
s = 0;
for k = 1:n
    s = s + 1/k^2;
end
```

which is saved in the file `psum.m` and we call it with $n = 10^4, \ldots, 10^{10}$:

```
>> format long
>> for k= 4:10
       psum(10^k)
   end
```

which gave the answers (black digits are correct, while red are wrong, compared to the exact series):

1.644834071848065
1.644924066898242
1.644930668487700
1.644933966847260
1.644934057834575
1.644934057834575
1.644934057834575

while the exact value of the series is $\frac{\pi^2}{6} = 1.644934066848226....$

The partial sum stagnates to converge from $n \geq 10^8$. What happens after $n \geq 10^8$? Let us check:

```
>> s = psum(1e8)
>> k = 1e8 + 1
>> 1/k^2
>> s + 1/k^2
```

which shows that the current partial sum is s=1.644934057834575 while the next term $1/k^2 = 1/(10^8 + 1)^2 \approx 10^{-16}$. So in the addition `s + 1/k^2`, the $1/k^2$ is swamped by the s. **Is there a quick fix for this problem?** Think about it ... (the answer can be found on the next page)

Cancellation. When we subtract two nearly equal numbers, the equal leading significant figures of the numbers will cancel which causes loss of meaningful significant figures. In the section 1, we wrote the function `derivative` to use $(f(x+h) - f(x))/h$ for approximating $f'(x)$. Note that when $h \to 0$, we have $f(x+h)$ becomes closer and closer to $f(x)$ so there are more and more cancellation of significant figures. Let us try

```
>> f = inline('sin(x)')
>> for k = 2:2:12
     (f(1+10^(-k))-f(1))*10^k
   end
```

which gave

0.536085981011869
0.540260231418621
0.540301885121330
0.540302302898255
0.540302247387103
0.540345546085064

while the exact derivatie is given as $\cos 1$=0.540302305868140...

We see that the approximation first tends to be more accurate for $h = 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}$ but then gets larger error from $h = 10^{-10}$. It is because that $f(x+h) - f(x)$ cancels many significant figures:

```
>> sin(1+1e-10)
ans =
   0.841470984861927
>> sin(1)
ans =
   0.841470984807897
```

so that `sin(1+1e-10)-sin(1)` contains essentially only 5 significant digits. The division by h does not change this fact. (You may have noticed that for $h = 10^{-10}$ the approximation of derivative has 6 correct digits, not only 5. This is either by luck or by the extra significant figures stored in the register (a high speed small memory inside CPU).)

3 Convergence rates

Let us return to the derivative approximation problem used for illustration of the cancellation in the section 2. Indeed, there are not only the round-off error due to cancellation but also the numerical approximation error due to $f'(x) \approx \frac{f(x+h)-f(x)}{h}$. We know that $\frac{f(x+h)-f(x)}{h} \to f'(x)$ as $h \to 0$, but how fast? Using the Taylor expansion, we can find

$$\frac{f(x+h)-f(x)}{h} = f'(x) + \frac{1}{2}f''(\xi)h$$

for some $\xi = \xi(x,h)$ between x and $x+h$. Hence,

$$|f'(x) - \frac{f(x+h)-f(x)}{h}| = \frac{1}{2}|f''(\xi)|h.$$

If $f(x) = \sin x$, then $f''(x) = -\sin x$ satisfies $|f''(\xi)| \le 1$. So $|f'(x) - \frac{f(x+h)-f(x)}{h}| \le \frac{1}{2}h$ and we can write $\frac{f(x+h)-f(x)}{h} = f'(x) + O(h)$. The convergence of $\frac{f(x+h)-f(x)}{h}$ to $f'(x)$ is the order of convergence $O(h)$. That explains why we see in the section 2 that the correct digits extends by two places when h is divided by 100, for $h = 10^{-2}, 10^{-4}, 10^{-6}$. It would go all the way like that if there was not any round-off error (i.e. if we had a machine with unlimited precision). But as $h \to 0$, the round-off error becomes larger and larger, albeit the approximation error decreases. If we think of the two errors as functions of h and plot their graphs, then there is a cross point at which the two types of errors equal, and after which the round-off error dominates.

Now, let us return to the partial sum problem used for illustration of the swamping in the section 2. We indicated that **a quick fix for the swamping problem** in evaluating $\sum_{k=1}^{\infty} \frac{1}{k^2}$. The idea is to add the smaller numbers first:

```
function s = psum2(n)
s = 0;
for k = n:-1:1
    s = s + 1/k^2;
end
```

where we simply changed the loop to start from n, step by -1 and end at 1, and the function is named as `psum2`. We call it with

```
>> format long
>> for k= 4:10
       psum2(10^k)
   end
```

which gave the answers

1.644834071848060
1.644924066898226
1.644933066848726
1.644933966848231
1.644934056848226
1.644934065848226
1.644934066748226

while the exact series $\sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6} = 1.644934066848226...$ We see that the partial sum continually tends to the series after the swamping problem is resolved. But, how fast? We need to estimate the remainder

$$|\sum_{k=1}^{\infty} \frac{1}{k^2} - \sum_{k=1}^{n} \frac{1}{k^2}| = \sum_{k=n+1}^{\infty} \frac{1}{k^2} \le \frac{1}{n(n+1)} + \frac{1}{(n+1)(n+2)} + \ldots \le \frac{1}{n}.$$

We have also the remainder $\geq \frac{1}{n+1}$ which is of the same order as $\frac{1}{n}$. So the convergence of $\sum_{k=1}^{n} \frac{1}{k^2}$ to $\sum_{k=1}^{\infty} \frac{1}{k^2}$ is the order of convergence $1/n$, and we write

$$\sum_{k=1}^{n} \frac{1}{k^2} = \sum_{k=1}^{\infty} \frac{1}{k^2} + O(\frac{1}{n}),$$

which explains why we got one more correct digit when n is multiplied by 10.