MA2507 Computing Mathematics Laboratory: Week 3

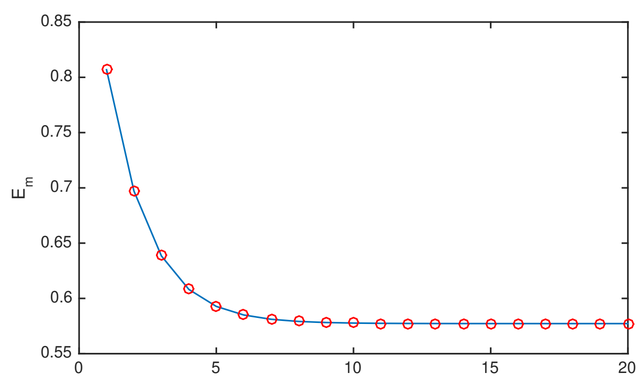
1. The "for" loop. MATLAB is a lot more than a progarmming language, but it certainly has the necessary functionalities of a programming language. Here, we start with the "for" loop which is a basic element of all programming languages. As an example, we calculate

$$E_m = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{2^m} - \log(2^m)$$

for $m = 1, 2, ..., 20$. As m tends to infinity, E_m converges to a constant, the so-called Euler's constant $\gamma = 0.5772156649\ldots$. Notice that E_m is not defined as $1 + 1/2 + ... + 1/m - \log_2 m$. Here is a program using a double "for" loop.

```
for m=1:20
    E(m)=0;
    n=2^m;
    for j=1:n
        E(m)=E(m)+1/j;
    end
    E(m)=E(m)-log(n);
end
mm=1:20;
plot(mm,E,mm,E,'ro')
xlabel('m')
ylabel('E_m')
```

Notice that m is not a vector. After the "for" loop is completed, we have $m = 20$. The above generates the following figure.



Actually, in MATLAB, loops tend to slow down the program. If we use MATLAB vector operations, we can remove the j-loop. Now, the main "for" loop in m becomes

```
for m=1:20
    n=2^m;
    jj=1:n;
    E(m) = sum(1./jj)-log(n);
end
```

Notice that we are repeating many calculations. To improve the efficiency (which is not important for this small problem), we can rewrite the program as follows:

```
E(1)=1.5;      % for m=1, first set E(1)=1+1/2.
for m=2:20     % for m from 2 to 20, calculate 1+1/2+...+1/n
    jj=2^(m-1)+1:2^m;
    E(m) = E(m-1)+sum(1./jj);
end
mm=1:20;
E = E - mm*log(2);  % do not forget to subtract log(n).
```

2. Ultra slow methods. Some methods that you have learnt before may be ultra slow and thus have no practical use for "large" problems. Here, we use a "for" loop to calculate the number of operations needed to compute the determinant of an $n \times n$ square matrix by the co-factor expansion method. Let $F(n)$ be the number of required operations for an $n \times n$ matrix, then $F(1) = 0$, and

$$F(n) = nF(n-1) + 2n - 1.$$

We can use the time needed to multiply two 5000×5000 matrices to estimate the number of operations that our computer can do in one second.

```
>> n=5000;
>> A=rand(n); B=rand(n);
>> tic; C=A*B; toc
Elapsed time is 3.139875 seconds.
```

My computer can do nearly 8×10^{10} operations in one second.

```
>> sp=2*n^3/3.139875    % 2n^3 operations needed for matrix multiplication
sp =
   7.9621e+10
```

Then, we find the time needed (in years) to calculate the determinant of a 22×22 matrix by co-factor expansion.

```
>> F(1)=0;
>> for j=2:22
     F(j)=j*F(j-1)+2*j-1;
   end
>> F(22)/(sp*3600*24*365) % time in years needed to calculate det
ans =
   1.2168e+03
```

Therefore, you need more than 1200 years to calculate the determinant of a 22×22 matrix by the method of co-factor expansion.

3. Numerical instability. The quadratic equation $\lambda^2 + (5/6)\lambda - 1 = 0$ has two roots: $\lambda_1 = 2/3$ and $\lambda_2 = -3/2$. Consider the linear recurrence

$$a_j + (5/6)a_{j-1} - a_{j-2} = 0, \quad j = 3, 4, \ldots$$

It can be shown that the general solution of the above is

$$a_j = C_1(2/3)^j + C_2(-3/2)^j,$$

where C_1 and C_2 are constants. These constants can be determined from the values of of a_1 and a_2. A particular solution is $a_j = (2/3)^j$. It can be obtained for $C_1 = 1$ and $C_2 = 0$. We try to calculate this solution by the linear recurrence relation and compare it with the exact value $(2/3)^j$. Here is a MATLAB program:

```
n=200;
a(1)=2/3;
a(2)=4/9;
for j=3:n
    a(j)=a(j-2)-(5/6)*a(j-1);
end
b =  (2/3).^(1:n);
[b', a']
```

For small values of j, the results by the linear recurrence are accurate, but for large j, the results are terribly wrong! But the MATLAB program is perfectly correct. In principle, with the first two lines, we should get exactly $C_1 = 1$ and $C_2 = 0$, but the reality is that $C_2 \neq 0$ (but it is very small, on the order of 10^{-16}). This is related to the so-called round-off errors. For example, if you set $x = 1.1$ in MATLAB, i.e.

```
>> x=1.1
x =
     1.100000000000000e+00
```

you do not really get $x = 1.1$, because MATLAB stores x as a double precision floating point number which can be written as $p/2^q$ for integers p and q. You can easily prove that 1.1 cannot be written as $p/2^q$ for any integers p and q.

4. Logistic map. Starting from any real number x_1, we can define a sequence $\{x_n\}$ using the recursion formula

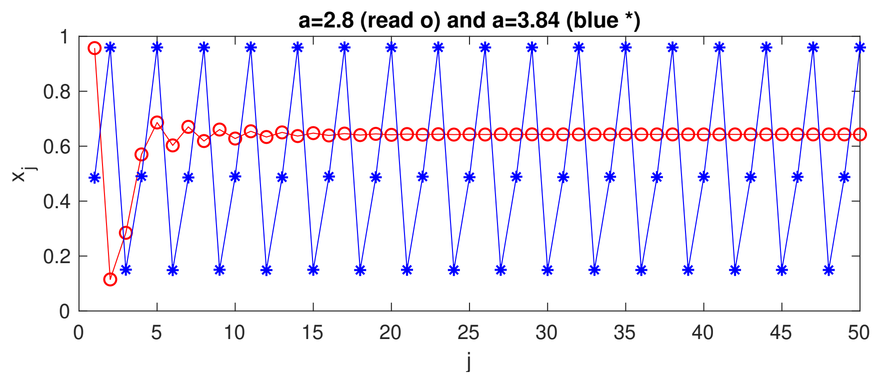
$$x_j = f(x_{j-1}) = ax_{j-1}(1 - x_{j-1}), \quad j = 2, 3, ...$$

where a is a given real number. This quadratic function $f(x) = ax(1-x)$ is often called the logistic map. The question is what happens to the sequence $\{x_n\}$ as $n \to \infty$. The answer depends on a. In the simplest case, $\lim x_n$ exists, so x_n tends to a constant. In other cases, $\lim x_n$ does not exist, but the sequence can be split into a finite number of convergent subsequences, thus the sequence approaches a finite set of numbers. In more complicated cases, the sequence approaches an infinite set of numbers. Here, we show the first 50 points for $a = 2.8$ and $a = 3.84$. Using the following MATLAB program

```
m = 50;
a = 2.8;
x(1) = rand;
for j = 2:m
    x(j) = a*x(j-1)*(1-x(j-1));
```

```
end
plot(1:m,x,'r',1:m,x,'ro')
hold on
a = 3.84;
x(1) = rand;
for j = 2:m
    x(j) = a*x(j-1)*(1-x(j-1));
end
plot(1:m,x,'b',1:m,x,'b*')
xlabel('j')
ylabel('x_j')
title('a=2.8 (red o) and a=3.84 (blue *)')
hold off
```

we obtain the following figure. It can be seen that for $a = 2.8$, the sequence converges to a constant,



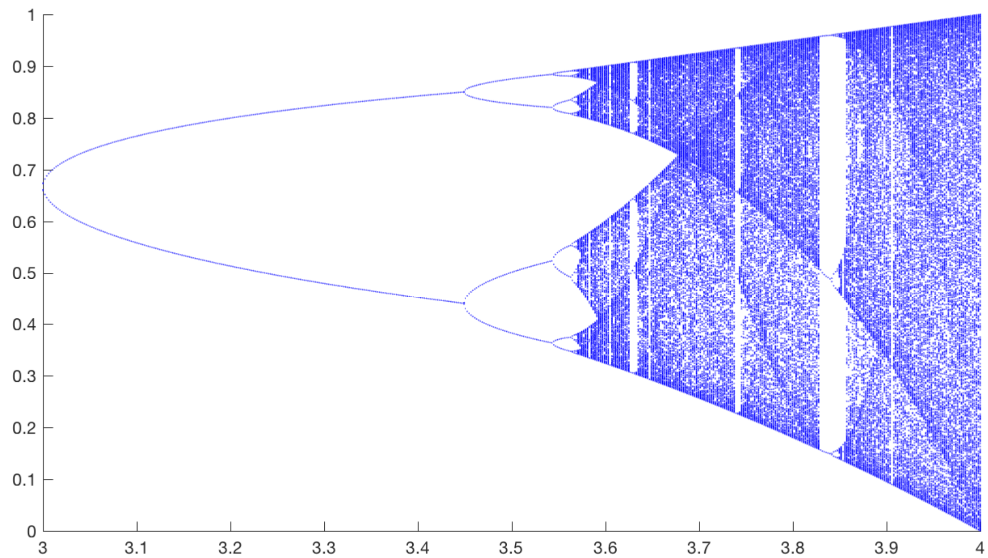
and for $a = 3.84$, the sequences approaches three constants. We call these three constants the limiting values of the sequence for $a = 3.84$. The sequence depends on x_1, but the limiting values are almost always independent of x_1, that is, for a random x_1, we get the same limiting values with probability 1.

Now, we want to calculate and show the limiting values for $n = 500$ different a between 3 and 4. That is, we discretize the interval $[3, 4]$ as $[a_1, a_2, ..., a_n]$, and calculate the limiting values for each a_i. To calculate the limiting values, we do $m + p$ iterations, and keep the last p iterations as the approximate limiting values. Here is a MATLAB program for $m = 1000$ and $p = 500$. We show the last p iterations as blue dots on a vertical line for the corresponding a_i.

```
n = 500;
m = 1000;
p = 500;
a = linspace(3,4,n);
hold on
for i=1:n
    x(1)=rand;
    for j= 2 : m+p
        x(j)=a(i)*x(j-1)*(1-x(j-1));
    end
    aa = a(i)*ones(1,p);
```

```
    plot(aa,x(m+1:m+p),'b.','MarkerSize',1)
end
hold off
```

The program gives the figure below. I have changed the value of `MarkerSize` to 1 to reduce the size of the dots. The programs above use the `plot` command n times. If you are willing to save



the points in a matrix (which requires more computer memory), you can use `plot` only once.