MA2507 Computing Mathematics Laboratory: Week 1

1. Generating vectors and matrices

```
>> format compact     % reduce space for display
>> a=[1 2 3]          % generate a vector
a =
     1     2     3
>> a=[1,2,3]
a =
     1     2     3
>> b=4:6
b =
     4     5     6
>> b=4:1:6
b =
     4     5     6
>> b=4:0.5:6
b =
    4.0000    4.5000    5.0000    5.5000    6.0000
>> b=linspace(4,6,5)
b =
    4.0000    4.5000    5.0000    5.5000    6.0000
>> c=[1;1;-2]                % column vector, ";" for next row
c =
     1
     1
    -2
>> A=[1 2 3                   % generate a matrix
      3 4 5
      2 2 1]
A =
     1     2     3
     3     4     5
     2     2     1
>> A=[1 2 3; 3 4 5; 2 2 1]    % ";" for next row
A =
     1     2     3
     3     4     5
     2     2     1
>> A=[1, 2, 3; 3, 4, 5; 2, 2, 1]
A =
     1     2     3
     3     4     5
     2     2     1
>> A(2:3,2:3)                % sub-matrix, not saved, stored in ans
ans =
```

```
     4     5
     2     1
>> A(2,3)                   % (2,3) entry of A
ans =
     5
>> diag(A)                  % get the diagonal elements
ans =
     1
     4
     1
>> diag(c)                  % create a matrix with given diagonal
ans =
     1     0     0
     0     1     0
     0     0    -2
>> diag(c,1)                % create a matrix
ans =
     0     1     0     0
     0     0     1     0
     0     0     0    -2
     0     0     0     0
>> diag(c,-1)               % create a matrix
ans =
     0     0     0     0
     1     0     0     0
     0     1     0     0
     0     0    -2     0
>> eye(3)                   % identity matrix
ans =
     1     0     0
     0     1     0
     0     0     1
>> ones(3)                  % matrix with all entries 1
ans =
     1     1     1
     1     1     1
     1     1     1
>> ones(3,3)
ans =
     1     1     1
     1     1     1
     1     1     1
>> ones(3,1)
ans =
     1
     1
```

```
     1
>> ones(1,3)
ans =
     1     1     1
>> rand                    % random number, uniform in [0,1]
ans =
    0.8147
>> rand                    % random number, again
ans =
    0.9058
>> rand(3)                 % generate 3x3 random matrix
ans =
    0.1270    0.0975    0.9575
    0.9134    0.2785    0.9649
    0.6324    0.5469    0.1576
>> rand(3,1)               % generate 3x1 random vector
ans =
    0.9706
    0.9572
    0.4854
>> rand(3,3)               % generate 3x3 random matrix
ans =
    0.8003    0.9157    0.6557
    0.1419    0.7922    0.0357
    0.4218    0.9595    0.8491
```

2. Basic operations: `+`, `-`, `*`, `\`, `/`, `^`, `'`, linear systems and least squares

```
>> a=[1 2 3];          % hide results with ";"
>> c=[7 8 9];
>> f = c'              % transpose for real vector
f =
     7
     8
     9
>> a*f                 % row multiplies column
ans =
    50
>> f*a                 % column multiplies a row
ans =
     7    14    21
     8    16    24
     9    18    27
>> a/10                % divided by 10
ans =
    0.1000    0.2000    0.3000
>> 10\a                % multiplied by (1/10)
```

```
ans =
    0.1000    0.2000    0.3000
>> u=2+3i             % complex number
u =
   2.0000 + 3.0000i
>> v=4+i              % complex number
v =
   4.0000 + 1.0000i
>> i=100              % over-write internal i
i =
   100
>> v=4+i
v =
   104
>> v=4+1i            % complex number without internal i
v =
   4.0000 + 1.0000i
>> u'                 % complex conjugate
ans =
   2.0000 - 3.0000i
>> conj(u)            % complex conjugate
ans =
   2.0000 - 3.0000i
>> A = [1 2 3; 3 4 5; 2 2 1];
>> det(A)                    % determinant
ans =
     2
>> rank(A)
ans =
     3
>> inv(A)                     % inverse matrix
ans =
   -3.0000    2.0000   -1.0000
    3.5000   -2.5000    2.0000
   -1.0000    1.0000   -1.0000
>> A'                         % tranpose for real matrix
ans =
     1     3     2
     2     4     2
     3     5     1
>> b=[1 1 -2]';
>> x=A\b                       % solve Ax=b, IMPORTANT!
x =
    1.0000
   -3.0000
    2.0000
```

```
>> c=[1 1 -2];
>> y=c/A                        % solve yA=c
y =
    2.5000   -2.5000    3.0000
>> B=A(:,1:2)                   % submatrix of A, 1st and 2nd columns
B =
     1     2
     3     4
     2     2
>> z = B\b                      % least squares problem min||Bz-b||
z =
   -3.0000
    2.3333
>> B*z                          % check Bz ?= b
ans =
    1.6667
    0.3333
   -1.3333
>> A(2,3)=5+2i                 % reset one entry
A =
   1.0000 + 0.0000i   2.0000 + 0.0000i   3.0000 + 0.0000i
   3.0000 + 0.0000i   4.0000 + 0.0000i   5.0000 + 2.0000i
   2.0000 + 0.0000i   2.0000 + 0.0000i   1.0000 + 0.0000i
>> A'                          % transpose and complex conjugate
ans =
   1.0000 + 0.0000i   3.0000 + 0.0000i   2.0000 + 0.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i   2.0000 + 0.0000i
   3.0000 + 0.0000i   5.0000 - 2.0000i   1.0000 + 0.0000i
>> transpose(A)                % just transpose
ans =
   1.0000 + 0.0000i   3.0000 + 0.0000i   2.0000 + 0.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i   2.0000 + 0.0000i
   3.0000 + 0.0000i   5.0000 + 2.0000i   1.0000 + 0.0000i
>> conj(A)                     % complex conjugate
ans =
   1.0000 + 0.0000i   2.0000 + 0.0000i   3.0000 + 0.0000i
   3.0000 + 0.0000i   4.0000 + 0.0000i   5.0000 - 2.0000i
   2.0000 + 0.0000i   2.0000 + 0.0000i   1.0000 + 0.0000i
```

3. Built-in functions `sqrt, exp, log, log10, sin, cos, tan, cot`

```
>> A=[1 2; -1 0]
A =
     1     2
    -1     0
>> exp(A)
ans =
```

```
    2.7183    7.3891
    0.3679    1.0000
>> B=(pi/2)*A
B =
    1.5708    3.1416
   -1.5708         0
>> cos(B)
ans =
    0.0000   -1.0000
    0.0000    1.0000
>> sin(B)
ans =
    1.0000    0.0000
   -1.0000         0
>> a = [-2, -2+0.001i, -2-0.001i]
a =
  -2.0000 + 0.0000i  -2.0000 + 0.0010i  -2.0000 - 0.0010i
>> sqrt(a)   % discontinuity along negative real axis
ans =
   0.0000 + 1.4142i   0.0004 + 1.4142i   0.0004 - 1.4142i
```

4. The dot: element-wise operations

```
>> a=[1 2 3]; b=[4 5 6];
>> a.*b    % element-wise multiplication
ans =
     4    10    18
>> a.\b    % element-wise left division
ans =
    4.0000    2.5000    2.0000
>> a./b   % element-wise right division
ans =
    0.2500    0.4000    0.5000
>> 1./a   % vector with elements 1/a(i)
ans =
    1.0000    0.5000    0.3333
>> a.^3    % element-wise power
ans =
     1     8    27
>> b.^a   % element-wise power
ans =
     4    25   216
>> A=[1 2;3 4]  % generate matrix
A =
     1     2
     3     4
>> 2.^A        % matrix with entries 2^A(i,j)
```

```
ans =
     2     4
     8    16
>> B=[2 -1; 1 3]
B =
     2    -1
     1     3
>> A.^B        % matrix with entries A(i,j)^B(i,j)
ans =
    1.0000    0.5000
    3.0000   64.0000
>> A.*B        % matrix with entries A(i,j)*B(i,j)
ans =
     2    -2
     3    12
>> A.\B        % matrix with entries A(i,j)\B(i,j)
ans =
    2.0000   -0.5000
    0.3333    0.7500
>> A./B        % matrix with entries A(i,j)/B(i,j)
ans =
    0.5000   -2.0000
    3.0000    1.3333
>> A(1,2)=2+3i  % reset one element of A
A =
   1.0000 + 0.0000i   2.0000 + 3.0000i
   3.0000 + 0.0000i   4.0000 + 0.0000i
>> A.'          % transpose only
ans =
   1.0000 + 0.0000i   3.0000 + 0.0000i
   2.0000 + 3.0000i   4.0000 + 0.0000i
```

5. Script file. You can put your MATLAB commands into a file, with a ".m" extension, the so-called script file, then run the script file, by typing the file name. Let us consider the follow example. Solve the linear system $Ax = b$, where A is a 100×100 matrix defined through vectors c and d, by

$$A = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix} + \begin{bmatrix} c_1d_1 & c_1d_2 & c_1d_3 & \dots \\ c_2d_1 & c_2d_2 & c_2d_3 & \dots \\ c_3d_1 & c_3d_2 & c_3d_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix},$$

where $b_j = c_j = 1/j$ and $d_j = \sin(\sqrt{2}j)$, for $1 \le j \le 100$. Show x_1, x_2, ..., x_5. Here is the script file saved as `w1example1.m`:

```
%% Week 1, Example 1
%  solve a 100x100 linear system, show x(1), x(2), ..., x(5) only
n = 100;
```

```
jj = 1:n;
b = (1./jj)';            % column vector
d = sin(sqrt(2)*jj);     % row vector
a1 = ones(1,n);
a2 = ones(1,n-1);
A = diag(a2,1) + diag(a2,-1) - 2*diag(a1) + b*d;
x = A\b;
x(1:5)
```

Now, we run it and get

```
>> w1example1
ans =
   42.7809
   75.3692
  102.8613
  126.9559
  148.5024
```

You can also use the `publish` command to produce a web page or a file, including both the program and the output.

6. Exercise: Given n points (x_j, y_j), $1 \le j \le n$, we can try to fit them by a quadratic polynomial $P(x) = c_1 + c_2 x + c_3 x^2$ using least squares. That means, we solve the minimization problem $\min \sum_{j=1}^{n} [c_1 + c_2 x_j + c_3 x_j^2 - y_j]^2$. This leads to $\min_{\mathbf{c}} ||A\mathbf{c} - \mathbf{y}||$, where

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Now for $n = 10$, $x_j = 1 + (j-1)/(n-1)$ and $y_j = e^{x_j}$, find c_1, c_2 and c_3.

Solution: Here is the script file, called `w1ex.m`:

```
%% Week 1 Exercise 1
% Solve a least squares problem for data fitting by quadratic polynomial.
x = linspace(1,2,10);
y = exp(x);
A = [ones(10,1),x',(x.^2)'];
c = A\y'
```

If we run the script, we get

```
>> w1ex
c =
    2.7021
   -2.2472
    2.2885
```

MA2507 Computing Mathematics Laboratory: Week 2

1. Eigenvalue problem. For a square matrix A, the eigenvalues satisfy $\det(\lambda I - A) = 0$. For each eigenvalue λ_j, you can find an eigenvector $\mathbf{v}_j$ such that $A\mathbf{v}_j = \lambda_j \mathbf{v}_j$. There are many eigenvalue problems for linear operators (often given as differential operators). For a string (e.g. of a violin), there is an eigenvalue problem given by an ordinary differential equation, for which the eigenvalues are related to the vibration frequencies (the vibrations give rise to the sound). In quantum mechanics, the energy of an electron in an atom cannot take arbitrary values. The allowed energey levels are the eigenvalues of a differential operator (given in the so-called Schrödinger equation). Eigenvalue problems of linear operators can be reduced to matrix eigenvalue problems.

```
>> A=[1 2 3; 4 5 6; 3 1 2]
A =
     1     2     3
     4     5     6
     3     1     2
>> eig(A)    % find the eigenvalues
ans =
    8.5772
   -1.3528
    0.7756
>> [V,D]=eig(A) % find eigenvalues and eigenvectors
V =
   -0.3514   -0.6970    0.0526
   -0.8884   -0.2090   -0.8327
   -0.2954    0.6860    0.5512
D =
    8.5772         0         0
         0   -1.3528         0
         0         0    0.7756
>> v1=V(:,1)   % first column of V is the first eigenvector
v1 =
   -0.3514
   -0.8884
   -0.2954
>> ev(1)=D(1,1) % (1,1) entry of D is the first eigenvalue
ev =
    8.5772
>> A*v1         % Matrix multiplies an eigenvector
ans =
   -3.0143
   -7.6200
   -2.5334
>> ev(1)*v1     % eigenvalue multiplies the eigenvector
ans =
   -3.0143
   -7.6200
```

```
   -2.5334
>> A*V          % AV=VD, not DV
ans =
   -3.0143    0.9429    0.0408
   -7.6200    0.2828   -0.6459
   -2.5334   -0.9280    0.4275
>> V*D
ans =
   -3.0143    0.9429    0.0408
   -7.6200    0.2828   -0.6459
   -2.5334   -0.9280    0.4275
>> V*D*inv(V)  % eigenvalue decomposition A=V*D*(V inverse)
ans =
    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    3.0000    1.0000    2.0000
>> V*D/V      % inv of V replaced by right division
ans =
    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    3.0000    1.0000    2.0000
>> s=diag(D)  % diagonals of D are the eigenvalues
s =
    8.5772
   -1.3528
    0.7756
>> ss=sort(s) % order real eigenvalues from small to large
ss =
   -1.3528
    0.7756
    8.5772
>> [ss,II]=sort(s)  %II gives the index for the ordering
ss =
   -1.3528
    0.7756
    8.5772
II =
     2
     3
     1
>> VV=V(:,II)   % re-order the eigenvectors
VV =
   -0.6970    0.0526   -0.3514
   -0.2090   -0.8327   -0.8884
    0.6860    0.5512   -0.2954
>> A=[4 -1 1; 1 2 0; -1 1 1]     % A is not diagonalizable
```

```
A =
     4    -1     1
     1     2     0
    -1     1     1
>> [V,D]=eig(A)
V =
  -0.7071 + 0.0000i   0.0000 - 0.0000i   0.0000 + 0.0000i
  -0.7071 + 0.0000i  -0.7071 - 0.0000i  -0.7071 + 0.0000i
   0.0000 + 0.0000i  -0.7071 + 0.0000i  -0.7071 + 0.0000i
D =
   3.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   2.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   2.0000 - 0.0000i
>> real(V)      % show the real part only
ans =
   -0.7071    0.0000    0.0000
   -0.7071   -0.7071   -0.7071
    0.0000   -0.7071   -0.7071
>> real(D)      % should be all real
ans =
    3.0000         0         0
         0    2.0000         0
         0         0    2.0000
>> det(V)      % should be exactly zero
ans =
   5.5511e-17 - 1.8795e-08i
```

2. Simple graphics. MATLAB has excellent capability for graphics. Here, we use command `plot` to draw a Bessel function. The result is shown in the next page.

```
>> x=0:0.05:40;
>> y=besselj(0,x);
>> plot(x,y)
>> xlabel('x')     % add a label for the x-axis.
>> title('Bessel function J_0(x)')   % add a title to the figure.
```

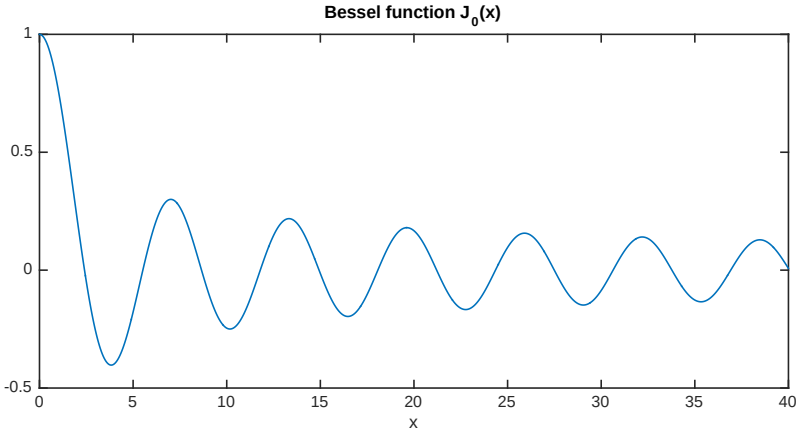
There are two kinds of Bessel functions $J_n(x)$ and $Y_n(x)$, where n is an index called the order. In the graphics window, you can save the figure in a proper format. The MATLAB figure format is useful if you want to load the figure and modify the figure later. For easy communications, you may use JPEG image format. I saved the figure in EPS file format to produce this document using LaTeX.

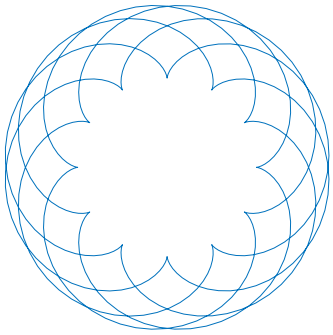
For two constants a and b, the curve given by

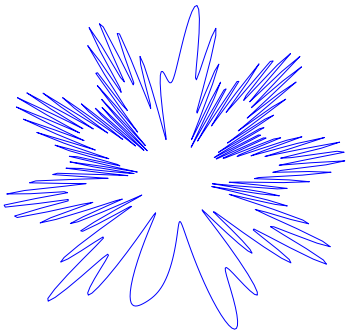
$$x = (a+b)\cos(t) - b\cos[(a/b+1)t], \quad y = (a+b)\sin(t) - b\sin[(a/b+1)t]$$

is quite interesting. The script

```
a=12;
```







```
b=5;
t=linspace(0,10*pi,500);
x=(a+b)*cos(t)-b*cos((a/b+1)*t);
y=(a+b)*sin(t)-b*sin((a/b+1)*t);
plot(x,y)
axis equal
axis off
```

draws the left-lower curve above for $a = 12$, $b = 5$ and $0 \leq t \leq 10\pi$, with x and y using the same scaling, and the axes removed. For a complex vector, the `plot` command use the real part as the horizontal axis and the imaginary part as the vertical axis. The following script

```
x=linspace(-1,1,500);
f=(3+sin(10*pi*x)+sin(61*exp(.8*sin(pi*x)+.7))).*exp(1i*pi*x);
plot(f,'b')
axis off
axis equal
```

produces the right-lower curve above.

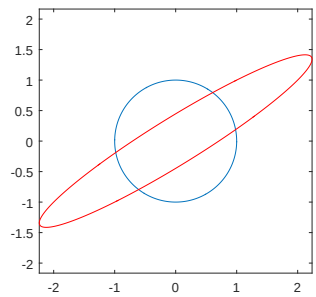
3. Matrix norm and SVD. In MA2503, we studied a little bit of singular value decomposition (SVD). That is, $A = USV^*$, where U and V are unitary matrices (inverse is conjugate transpose) and S is the diagonal matrix for the singular values. The largest singular value of a matrix is also the matrix norm which is defined as $||A|| = \max||Ax||/||x||$. Based on SVD, we claimed that in general a 2×2 matrix transforms the unit circle to an ellipse. Here is a script file that tests these things.

```
% Matrix multiplication maps a circle to an ellipse.
% This script file is saved as cirell.m
t=linspace(0,2*pi,300);   % 300 points for angle from 0 to 2pi
x = [cos(t); sin(t)];     % the unit circle
A=[1,2;1,1]               % a 2x2 matrix
y = A*x;
plot(x(1,:),x(2,:),y(1,:),y(2,:),'r') % draw two curves
axis equal                % use same scale for two axes
norm(A)                   % matrix norm
svd(A)                    % singular values only
[U,S,V]=svd(A)            % SVD of matrix A
```

If we run th escript in MATLAB, we get

```
>> A=[1,2;1,1]                % a 2x2 matrix
A =
     1     2
     1     1
>> norm(A)
ans =
    2.6180
>> svd(A)
ans =
    2.6180
    0.3820
>> [U,S,V]=svd(A)
U =
   -0.8507   -0.5257
   -0.5257    0.8507
S =
    2.6180         0
         0    0.3820
V =
   -0.5257    0.8507
   -0.8507   -0.5257
```

and the following figure



4. Exercise: For $j = 1, 2, ..., 99$, we define $a_j = 1 + 1/j$, $b_j = \sqrt{j}$ and the 100×100 matrix

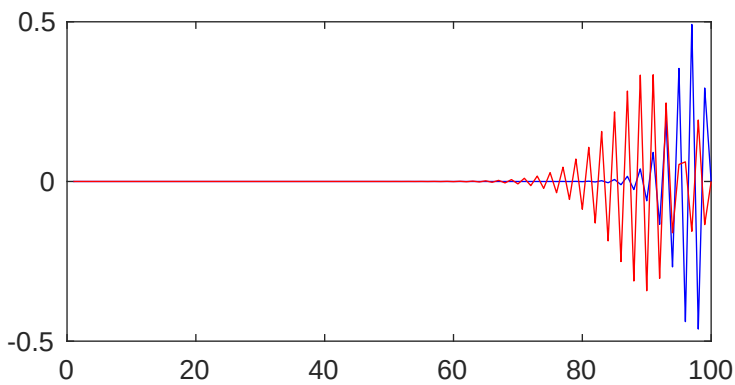
$$A = \begin{bmatrix} 0 & a_{99} & & & & b_{99} \\ a_{99} & 0 & a_{98} & & & \vdots \\ & a_{98} & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & a_2 & b_2 \\ & & & a_2 & 0 & b_1 \\ b_{99} & & \dots & b_2 & b_1 & 0 \end{bmatrix}.$$

Find the three smallest eigenvalues of A, and plot the eigenvectors corresponding to the 2nd and 3rd smallest eigenvalues.

Solution: We need to reverse the index, and define vectors $\mathbf{a} = [a_{99}, a_{98}, ..., a_1]$ and $\mathbf{b} = [b_{99}, b_{98}, ..., b_1]$. For real symmetric matrices, MATLAB automatically order the eigenvalues from the smallest to the largest. Therefore, we do not need to sort the eigenvalues. The MATLAB program is

```
kk = 99:-1:1;
a = 1+1./kk;
A = diag(a,1)+diag(a,-1);
b = sqrt(kk)
A(100,1:99) = b;
A(1:99,100) = b';
[V,D] = eig(A);
s = diag(D);
s(1:3)
jj = 1:100;
plot(jj,V(:,2),'b',jj,V(:,3),'r')
```

We get the following figure and the small eigenvalues:




```
ans =
  -69.3545
   -2.3687
   -2.1354
```

MA2507 Computing Mathematics Laboratory: Week 3

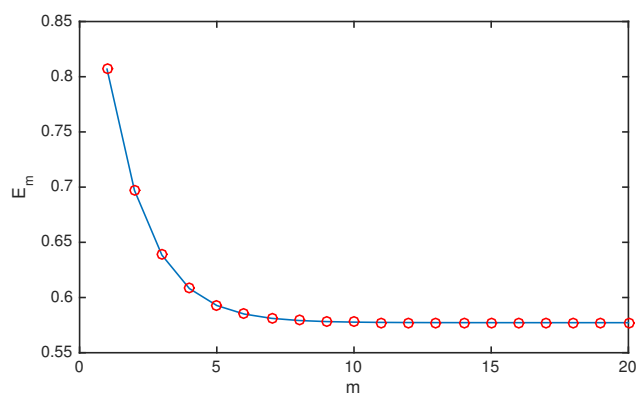
1. The "for" loop. MATLAB is a lot more than a progarmming language, but it certainly has the necessary functionalities of a programming language. Here, we start with the "for" loop which is a basic element of all programming languages. As an example, we calculate

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

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

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

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

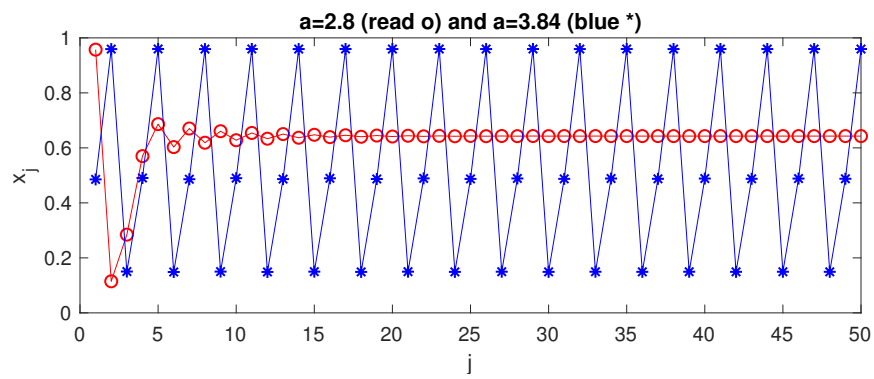
$$x_j = f(x_{j-1}) = ax_{j-1}(1 - x_{j-1}), \quad j = 2, 3, ...$$

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,



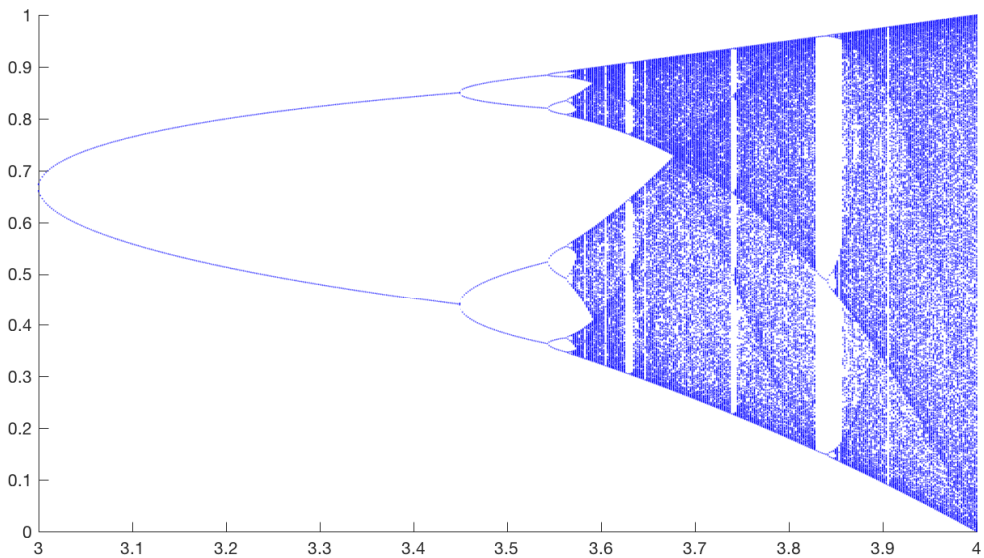
and for $a = 3.84$, the sequences approaches three constants. We call these three constants the limiting values of the sequence for $a = 3.84$. The sequence depends on x_1, but the limiting values are almost always independent of x_1, that is, for a random x_1, we get the same limiting values with probability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

5. Exercise: For $0 \le a \le 5$, calculate the eigenvalues of

$$A = \begin{bmatrix} 0 & 1 & a \\ 1 & 0 & 2 \\ -a & 2 & 1 \end{bmatrix},$$

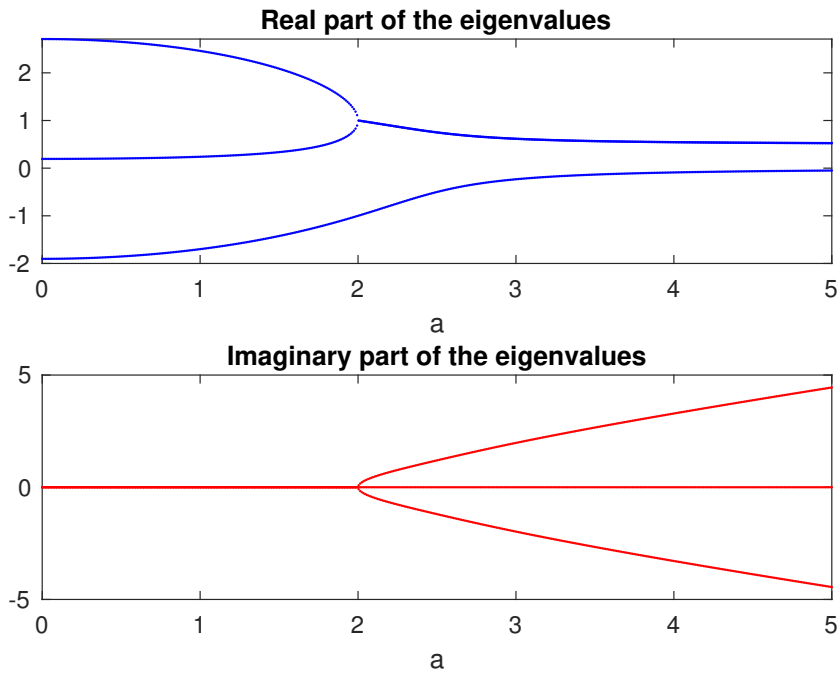
show the real and imaginary parts of the eigenvalues as functions of a.

Solution: We discretize a by $n = 500$ points, store the eigenvalues of A (for all a) in a $3 \times n$ matrix, then plot the eigenvalues as blue or red dots. We need the `subplot` command. The following MATLAB program

```
n=500;
a = linspace(0,5,n);
ss = zeros(3,n);
for j=1:n
   A=[0 1 a(j); 1 0 2; -a(j) 2 1];
   ss(:,j) = eig(A);
end
subplot(2,1,1)
plot(a, real(ss),'b.','MarkerSize',1)
xlabel('a')
title('Real part of the eigenvalues')
subplot(2,1,2)
```

```
plot(a, imag(ss),'r.','MarkerSize',1)
xlabel('a')
title('Imaginary part of the eigenvalues')
```

gives rise to the following figure.



MA2507 Computing Mathematics Laboratory: Week 4

1. The "while" loop. When we use a "for" loop to evaluate some formula repeatedly, we need to know the number of iterations. Often, the number of iteration is unknown, then the "while" loop becomes useful. Let us consider Newton's method for solving a nonlinear equation $f(x) = 0$. Starting from an initial guess x_1, Newton's method gives us a sequence x_2, x_3, ..., by

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}.$$

If the equation $f(x) = 0$ has a solution x_* and if x_1 is sufficiently close to x_*, then under some suitable conditions, $\lim_{j\to\infty} x_j = x_*$. In reality, we only calculate a finite number of iterations, and use the last iteration as the approximate solution. Typically, we use a small number `tiny` to control the "while" loop, and to finish the loop when some kind of "relative error" is smaller than `tiny`. For Newton's method, typically, we can use the ratio $(x_{j+1} - x_j)/x_{j+1}$ as the "relative error". Now, for $f(x) = x^2 - a$, we use Newton's method to find the square root of a. The following script file calculates $\sqrt{3}$.

```
% solve f(x) = x^2 -a by Newton's method.
a= 3;
x = 1.5;          % initial guess
tiny = 1.0e-10;   % small number
rat = 1;          % any number larger than tiny
while rat > tiny
    dx = (x^2-a)/(2*x);   % f(x_j)/f'(x_j)
    x = x - dx
    rat = abs(dx/x);
end
```

The program gives the following

```
x = 1.750000000000000
x = 1.732142857142857
x = 1.732050810014728
x = 1.732050807568877
x = 1.732050807568877
```

It agrees with the "exact" value of $\sqrt{3}$ very well.

2. Catastrophic cancellation. Let us write a program for

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

If we write the program based on the "for" loop, we need to specify an integer for the maximum number of terms. Meanwhile, we can define a variable $y = x^n/n!$, and update y in each iteration. This leads to

```
nmax=50;   % truncate the Taylor series to power nmax.
s=1;       % initialize s, the final s approximates exp(x)
y=1;
```

```
for n=1:nmax
   y=y*x/n;
   s=s+y;
end
```

This is not ideal, since when x is large (e.g. $x = 50$), the approximation is not at all accurate. For small x, it is possible that we have already kept too many terms. To improve this, we can use a "while" loop that compares $|y/s|$ with a given small number. But in the "while" loop, we no longer have an integer index. Therefore, we need to initialize the index and manually increase the index in the loop.

```
s=1;
y=1;
n=0;     % integer index
rat=1;
while rat > 1.0e-14
    n=n+1;   % increase the index by 1
    y=y*x/n;
    s=s+y;
    rat=abs(y/s);
end
s
```

Notice that we have used a small number 10^{-14} to control the "while" loop. Using `format long` and for $x = 2$, the above program gives $e^x \approx 7.389056098930645$, but the MATLAB internal program gives the slightly more accurate value 7.389056098930650.

However, the program has trouble if x is negative and $|x|$ is large. For $x = -21$, the above program gives $s = -3.164859560770680 \times 10^{-9}$. For $x = -50$, the program gives $s = 1.107293338289196 \times 10^4$. This is very bad, since $e^{-21} \approx 7.582560427911907 \times 10^{-10}$ and $e^{-50} \approx 1.928749847963918 \times 10^{-22}$. To understand that, we need to think about the magnitude of y for those negative x with large absolute values. For n near $|x|$, $y = x^n/n!$ has very large absolute value, but we can only store y for about 16 digits. The truncation error (caused by only keeping 16 digits of y) could be much larger than the exact value of e^x.

3. The "if ... else ... end" statement. If we divide the unit square $[0,1] \times [0,1]$ to $m \times m$ small squares (of equal size) and drop m^2 random points on the unit square, how many small squares (call them boxes) will be hit by one or more of these points on average? This is a problem in probability theory. We will do a simulation here. We will use an $m \times m$ matrix B to keep track of the boxes. The matrix B is initialized as a zero matrix. If the (i, j) box is hit by a point, then we turn the (i, j) entry of B to 1. For a point (x, y), we need to figure out which box it belongs to.

```
m = 100;
B = zeros(m,m);
numbox = 0;
for k = 1:m^2
    x = rand;            % get a random point (x,y)
    y = rand;
    ix = ceil(m*x);      % (x,y) goes to box (ix,iy)
```

```
    iy = ceil(m*y);      % ceil(z) is the integer >or= z.
    if B(ix,iy) == 0
        B(ix,iy) = 1;
        numbox = numbox+1;
    end
end
numbox
```

The typical answer is around 6300.

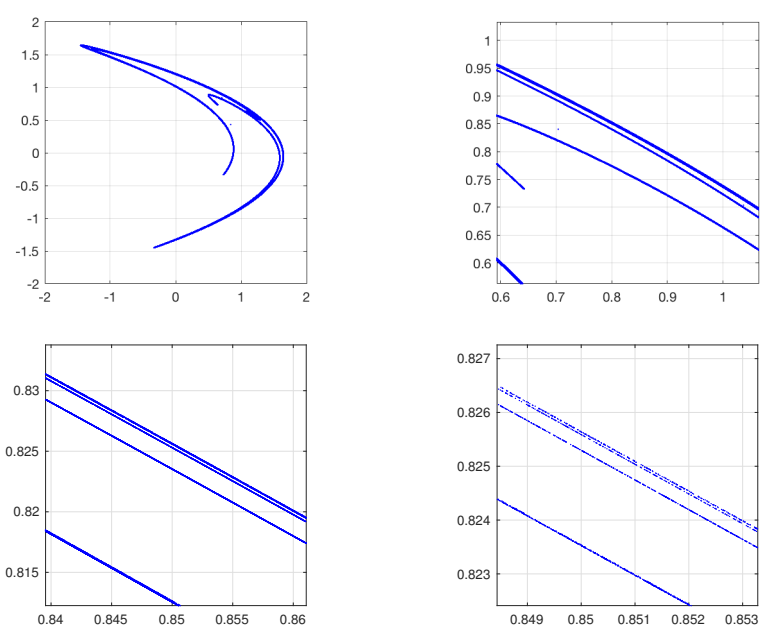
4. Hénon map: Staring from a point (x_1, y_1), we can generate an sequence in the plane by

$$x_{j+1} = a - x_j^2 + by_j, \quad y_{j+1} = x_j, \quad j = 1, 2, ...$$

where a and b are constants. For some a and b, the sequence $\{(x_j, y_j)\}$ can have very complicated behavior. The following program shows 10^7 points for $a = 1.28$ and $b = 0.3$.

```
N = 10^7;
x(1) = rand;
y(1) = rand;
for j=1:N-1
    x(j+1) = 1.28 - (x(j))^2 + 0.3*y(j);
    y(j+1) = x(j);
end
plot(x,y,'b.','MarkerSize',1)
axis equal
axis([-2 2 -2 2])
```

The above gives the first plot below. The others are zoom-in plots.



5. Exercise: Divide the square $[-2,2]\times[-2,2]$ to 128×128 small squares (boxes of equal size), generate a sequence of points by the Hénon map (for $a = 1.28$ and $b = 0.3$) with a random initial point in $[0,1]\times[0,1]$. Assuming the entire sequence is inside the big square $[-2,2]\times[-2,2]$, count the number of boxes the sequence hits, until you cannot find a new box for 1000 iterations.

   Solution: We need a "while" loop and an integer `NoNewBox` for the number of steps without finding a new box. The program is given below.

```
m = 128;
B = zeros(m,m);
N = 1000;
numbox = 0;
NoNewBox = 0;
x = rand;       % only the first point is random
y = rand;
while NoNewBox < N
    ix = ceil((x+2)*m/4);
    iy = ceil((y+2)*m/4);
    if B(ix,iy) == 0
        B(ix,iy) = 1;
        numbox = numbox + 1;
        NoNewBox = 0;
    else
        NoNewBox = NoNewBox + 1;
    end
    xold = x;   % get new point by Henon map without saving in a squenece
    x = 1.28-x^2+0.3*y;  % a = 1.28,  b=0.3
    y = xold;
end
numbox
```

   We typically get a number larger than 500.

MA2507 Computing Mathematics Laboratory: Week 5

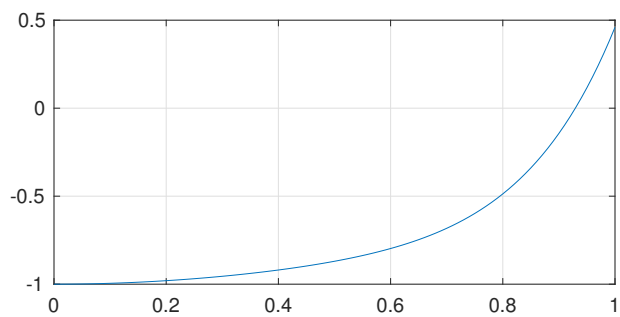
1. Functions. When we execute a MATLAB script file, all variables, vectors and matrices that appear in the script file go into the main memory of MATLAB. In order to keep a clean account of the memory space and to have a clean control for input and output, it is extremely useful to use functions. Usually, a function with the name `funcname` must be stored in the file called `funcname.m` and it should be in the current working folder. Here is a function for $f(x) = x^7 - \cos(x)$ and it is saved as `f.m`.

```
function y=f(x)
y = x.^7 - cos(x);
end
```

In the folder where `f.m` is stored, we can simply evaluate it in the MATLAB main command window, or use the function in any script file. For example, the following lines

```
>> x = 0:0.01:1;
>> plot(x, f(x)); grid
```

produce the figure below. We can see that the function has a zero between 0.9 and 1.



If the function is only useful for a particular script, we can put the function below the main script (in the more recent versions of MATLAB). Let us try to find the zero of function $f(x)$ by Newton's method. For that we need another function for its derivative. We have the following script file:

```
% main script file: Newton's method
format long
x = 0.9;
rat = 1;
while rat > 1.0e-12
    dx = f(x)/fp(x);
    x = x - dx;
    rat = abs(dx/x);
end
x

% function below the main script
function y=f(x)
y = x.^7 - cos(x);
```

```
end

% another function below the main script
function z=fp(x)
z = 7*x.^6 + sin(x);
end
```

The above gives $x = 0.929273104148150$. Since we still have `f.m` in the folder, we can also use MATLAB internal function `fzero` to find the solution:

```
>> fzero(@f,0.9)
ans =
   0.929273104148150
```

Notice that the first input of `fzero` is the name of a function. This is made possible by adding the symbol `@` in front of the name, otherwise (i.e, if you do not add `@`), MATLAB would think that you are trying to run the function without the input. We can write a program for Newton's method that also inputs functions.

```
% a very short main script
format long
mynewt(@f, @fp, 0.9)

% a very general Newton's method that inputs functions
function sol= mynewt(F,Fp,x)
rat = 1;
while rat > 1.0e-10
    dx = feval(F,x)/feval(Fp,x);
    x = x - dx;
    rat = abs(dx/x);
end
sol = x;
end

% a function f(x)
function y=f(x)
y = x.^7 - cos(x);
end

% the derivative of f(x)
function z=fp(x)
z = 7*x.^6 + sin(x);
end
```

Newton's method requires the derivative of $f(x)$. Secant method is useful, since it does not need the derivative, but it needs two initial guesses. Given x_1 and x_2, the secant method gives x_3, x_4, ..., by

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

To have an efficient code, we do not use a vector x. We implement the first step for computing x_3, after that, we swap the variables, so that x_4, x_5, ..., are all called x_3. Meanwhile, we save and swap $f(x_j)$, so that f is only evaluated once in each iteration. Here is the complete program.

```
% a very short main script
format long
x = mysecm(@f,0.8,0.9)

% a very general function for the secant method
function sol= mysecm(F,x1,x2)
rat = 1;
y1 = feval(F,x1);
y2 = feval(F,x2);
while rat > 1.0e-10
    dx = y2*(x2-x1)/(y2-y1);
    x3 = x2 - dx;
    y3 = feval(F,x3);
    rat = abs(dx/x3);
    x1=x2;
    x2=x3;
    y1=y2;
    y2=y3;
end
sol = x3;
end

% a function below
function y=f(x)
y = x^7 - cos(x);
end
```

2. Barnsley fern: We can use a random variable (uniform in [0,1]), and the "if ... elseif ... else ... end" statement to perform different tasks with different probabilities. Here is an example for generating the so-called Barnsley fern. Given four 2×2 matrices A_1, A_2, A_3, A_4 and four vectors b_1, b_2, b_3 and b_4 below

```
A1 = [0.85, 0.04; -0.04, 0.85]
A2 = [0.2, -0.26; 0.23, 0.22]
A3 = [-0.15, 0.28; 0.26, 0.24]
A4 = [0, 0; 0, 0.16]
b1 = [0; 1.6]
b2 = b1
b3 = [0; 0.44]
b4 = [0; 0]
```

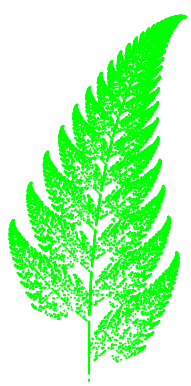
and the zero staring point $(x_1, y_1) = (0, 0)$, we can generate points (x_j, y_j) using

$$\begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} = A_m \begin{bmatrix} x_j \\ y_j \end{bmatrix} + b_m,$$

where the choice of integer m follows the following probability: $P(m=1)=0.85$, $P(m=2)=0.07$, $P(m=3)=0.07$ and $P(m=4)=0.01$. Notice that $0.85+0.07=0.92$ and $0.85+0.07+0.07=0.99$. We calculate 20000 points using the following program.

```
m=20000;
x=zeros(2,m);
for j=2:m
    a=rand;
    if a < 0.85
        A = [0.85, 0.04; -0.04, 0.85];
        b = [0; 1.6];
    elseif a < 0.92
        A = [0.2, -0.26; 0.23, 0.22];
        b = [0; 1.6];
    elseif a < 0.99
        A = [-0.15, 0.28; 0.26, 0.24];
        b = [0; 0.44];
    else
        A = [0, 0; 0, 0.16];
        b = [0; 0];
    end
    x(:,j)= A*x(:,j-1)+b;
end
plot(x(1,:),x(2,:),'g.')
axis equal
axis off
```

I have stored the points in a matrix with two rows and plot the points as green dots. The last two lines make sure the same scaling is used for x and y, and the box for the plot is removed. The above program gives the following figure.



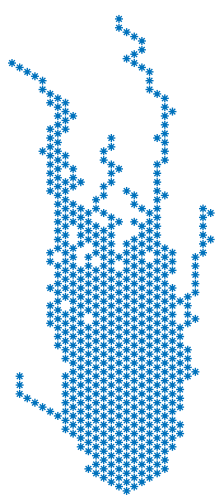
3. Creepy animal: Here is another example (last year's test question) using `rand` to control different probabilities. Starting from the origin $(x_1, y_1) = (0, 0)$, generate 10000 points by the following rule: (a) with probability 0.4, $(x_{j+1}, y_{j+1}) = (x_j, y_j) + (0, 1)$ (move up by distance 1); (b) with probability

0.25, $(x_{j+1}, y_{j+1}) = (x_j, y_j) + (\sqrt{3}/2, 1/2)$ (move up-right by distance 1); (c) with probability 0.25, $(x_{j+1}, y_{j+1}) = (x_j, y_j) + (-\sqrt{3}/2, 1/2)$ (move up-left by distance 1); (d) with probability 0.1, reset (x_{j+1}, y_{j+1}) to the origin. Plot the 10000 points as "*".

We can keep all x_j and y_j in a 10000×2 matrix, called it `x`. The index j refers to j-th row. Here is the MATLAB script file.

```
x(1,:) = [0 0];
for j=2:10000
    p = rand;
    if p < 0.40
        x(j,:) = x(j-1,:) + [0,1];
    elseif p <0.65
        x(j,:) = x(j-1,:) + [sqrt(3)/2, 0.5];
    elseif p < 0.9
        x(j,:) = x(j-1,:) + [-sqrt(3)/2, 0.5];
    else
        x(j,:) = [0 0];
    end
end
plot(x(:,1), x(:,2), '*')
axis equal
axis off
```

We run the above program and get a typical figure like the one below.



4. Exercise: Solve $f(x) = x^7 - \cos(x) = 0$ using the inverse quadratic interpolation method (IQIM). Given three initial guesses x_1, x_2, x_3, IQIM calculates x_4, x_5, ..., by

$$x_{j+1} = x_j - c_1 y_j + c_2 y_j^2, \quad j = 3, 4, ...$$

where $y_j = f(x_j)$, c_1 and c_2 satisfy

$$c_1 + (y_{j-1} - y_j)c_2 = \frac{x_{j-1} - x_j}{y_{j-1} - y_j}$$

$$c_1 + (y_{j-2} - y_j)c_2 = \frac{x_{j-2} - x_j}{y_{j-2} - y_j}$$

The method is derived as follows. We find a quadratic polynomial of y, say $x = P(y)$, such that $x_j = P(y_j)$, $x_{j-1} = P(y_{j-1})$ and $x_{j-2} = P(y_{j-2})$, then let $x_{j+1} = P(0)$. We let the polynomial be given as

$$P(y) = x_j + c_1(y - y_j) + c_2(y - y_j)^2,$$

then $x_j = P(y_j)$ is already satisfied. The conditions $x_{j-1} = P(y_{j-1})$ and $x_{j-2} = P(y_{j-2})$ give the two equations for c_1 and c_2. Like the secant method, IQIM does not require the derivative of f. But it requires three initial guesses. It can be proved that the method is faster than secant method. We have the following script file which includes the general IQIM and the function $f(x) = x^7 - \cos(x)$.

```
% a very short main script
format long
x = iqim(@f,0.7,0.8,0.9)

% function for a very general IQIM program
function sol = iqim(f,x1,x2,x3)
y1 = feval(f,x1);
y2 = feval(f,x2);
y3 = feval(f,x3);
rat = 1;
while rat > 1.0e-10
    A = [1, y2-y3; 1, y1-y3];
    b = [(x2-x3)/(y2-y3); (x1-x3)/(y1-y3)];
    c = A\b;
    dx = -c(1)*y3+c(2)*y3^2;
    x4 = x3 + dx;
    y4 = feval(f,x4);     % only 1 evaluation of f in each iteration
    rat = abs(dx/x4);
    x1=x2; x2=x3; x3=x4;  % three lines saved in one line
    y1=y2; y2=y3; y3=y4;
end
sol = x4;
end

% a particular function
function y=f(x)
y = x^7-cos(x);
end
```

Of course, we get the same answer $x = 0.929273104148150$. You can write a less general IQIM without using `feval`.

MA2507 Computing Mathematics Laboratory: Week 6 Test

1. Let A be a 300×300 matrix and $\mathbf{b}$ be a column vector of length 300, given by

$$A = \begin{bmatrix} T & I & \mathbf{0} \\ I & T & I \\ \mathbf{0} & I & T \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix},$$

where I is the 100×100 identity matrix, T is a 100×100 matrix,

$$T = \begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & 4 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 100 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} 1 \\ 1/2 \\ 1/3 \\ \vdots \\ 1/100 \end{bmatrix}, \quad \mathbf{b}_3 = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \\ \vdots \\ 100^2 \end{bmatrix}.$$

Solve the linear system $A\mathbf{x} = \mathbf{b}$, and show the first five elements of $\mathbf{x}$.
Solution: There are different ways to generate matrix A. We use the block version.

```
format long
n = 100;
w = ones(n,1);
u = ones(n-1,1);
T = diag(4*w)+diag(u,1)+diag(u,-1);
I = eye(n);
Z = zeros(n);
A = [T, I, Z; I, T, I; Z, I, T];
s = 1:n;
b = [s, 1./s, s.^2]';
x = A\b;
x(1:5)
```

The above script gives

```
>> w6q1
ans =
   0.090733511194292
   0.380293794481529
   0.526205354431658
   0.761554850494849
   0.965233997728883
```

2. Let A_n be an $n \times n$ matrix whose (i, j) entry is $a_{ij} = 1/(i + \sqrt{j} - 1)$. The singular values of matrix A can be found by `svd(A)` in MATLAB. Denote the smallest singular value of A_n by s_n. Plot s_n as a discrete function of n for $n = 1$, 2, 3, ..., 10. Use MATLAB command `semilogy` instead of `plot`. In addition, find an approximate formula:

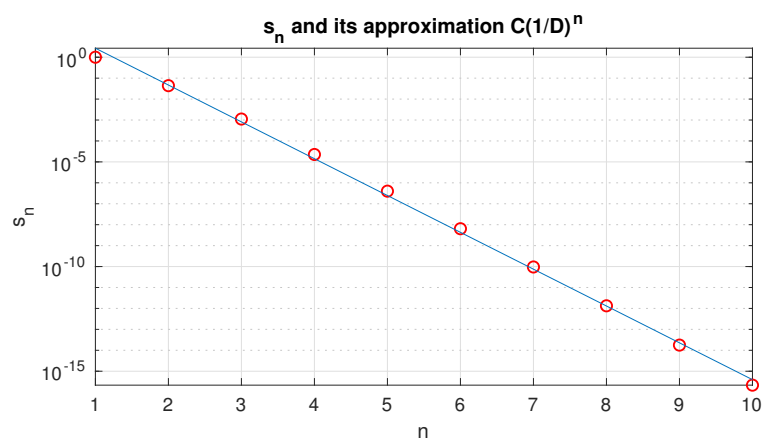
$$s_n \approx C \left(\frac{1}{D}\right)^n$$

based on least squares approximation to $\ln(s_n) \approx \ln(C) - n\ln(D)$ for $n = 1$, 2, ..., 10. Show results for C and D. In MATLAB, $\ln(x)$ is `log(x)`

Solution: We only need to generate the 10×10 matrix A_{10}. For $n < 10$, A_n is the top-left $n \times n$ block of A_{10}. Even if you do not know what the singular values are, you can still do the first part. Least squares was considered in the assignment for week 1. That is, we try to minimize $\sum_{n=1}^{10} [\ln(C) - n\ln(D) - \ln(s_n)]^2$.

```
% Part 1
n=10;
for i=1:n
    for j=1:n
        A(i,j) = 1/(i+sqrt(j)-1);
    end
end
for k=1:n
    ss=svd(A(1:k,1:k));
    s(k) = ss(k);
end
% Part 2
B = [ones(n,1), -(1:n)'];
b = log(s);
x = B\b';
C = exp(x(1))
D = exp(x(2))
% you only need to plot s_n, here I plot both s_n and the approximation
y = C*(1/D).^(1:n);
semilogy(1:n,s,'ro',1:n, y)
grid
xlabel('n')
ylabel('s_n')
title('s_n and its approximation C(1/D)^n')
```

We get the following figure, and `C=1.546119646299158e+02`, `D=57.432956826982704`.



3. Find a zero of Bessel function $J_0(x)$ near 2.4 by the rational interpolation method (RIM) given below. In MATLAB, $J_0(x)$ is `besselj(0,x)`. To solve $f(x) = 0$, RIM requires 3 initial guesses x_1, x_2, and

x_3, and it calculates x_4, x_5, ..., by

$$x_{j+1} = R(0), \quad j \geq 3$$

where $R(y)$ (depending on j) is a rational function given by

$$R(y) = x_j + \frac{c_1(y - y_j)}{1 + c_2(y - y_j)},$$

c_1 and c_2 are determined from the conditions $x_{j-1} = R(y_{j-1})$ and $x_{j-2} = R(y_{j-2})$. In the above, $y_k = f(x_k)$ for any integer $k \geq 1$.

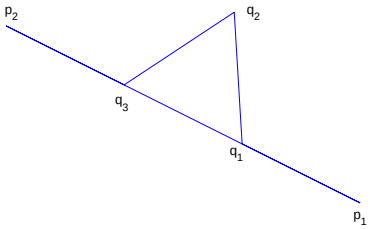
Solution: This problem is very similar to the assignment of week 5. However, you have to write down the two equations for c_1 and c_2. The program is best written for the first step that calculates x_4, then swap the variables. Below is a not-very-general code.

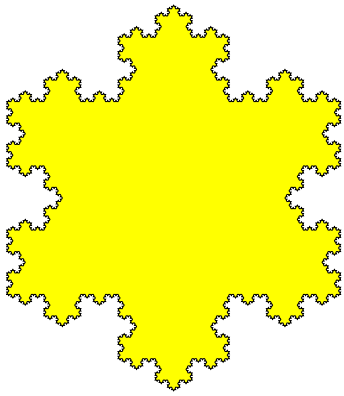
```
% main script including initial guesses and the method
x1 = 2.2;
x2 = 2.3;
x3 = 2.4;
y1 = f(x1);
y2 = f(x2);
y3 = f(x3);
rat = 1;
while rat > 1.0e-10
    A = [1, x3-x1; 1, x3-x2];                % New line
    b = [(x1-x3)/(y1-y3); (x2-x3)/(y2-y3)];  % New line
    c = A\b;
    dx = -c(1)*y3/(1 -c(2)*y3);              % New line
    x4 = x3 + dx;
    rat = abs(dx/x4);
    y4 = f(x4);
    x1=x2; x2=x3; x3=x4;
    y1=y2; y2=y3; y3=y4;
end
x4

% a function given below the main script
function y=f(x)
y = besselj(0,x);
end
```

We get the solution 2.404825557695773. You can make it more general using `feval`.

MA2507 Computing Mathematics Laboratory: Week 7

1. Koch snowflake. The Koch snowflake (also known as the Koch curve) is a mathematical curve constructed by Swedish mathematician Helge von Koch. As a first step, we need to construct a function called `addq`. Starting from two points $\mathbf{p}_1$ and $\mathbf{p}_2$ in the plane, it calculates $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$, as shown in the left figure below. It outputs four points, $\mathbf{p}_1$, $\mathbf{q}_1$, $\mathbf{q}_2$, $\mathbf{q}_3$ in a 2×4 matrix, where





each point is a column vector of length 2. The Koch curve is generated by starting from a triangle, then applying the function `addq` to each edge recursively and indefinitely. In practice, we only run `addq` for small number of levels to draw a picture. The program is given below.

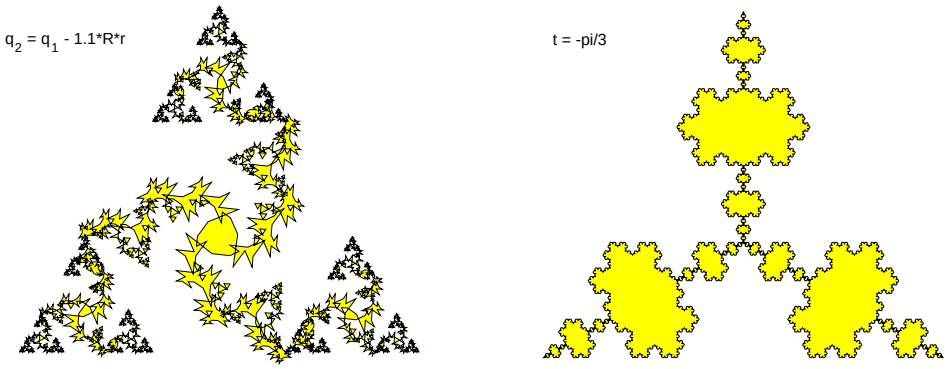
```
% main script for Koch's snowflake, requires function "addq"
% set the number of levels.
mmax = 5;
% points of a triangle as columns of p, repeating the first point
p = [0, 0; 1, 0; 1/2, sqrt(3)/2; 0 0]';
for m=1:mmax
    pnew = [];      % start an empty matrix pnew
    n = length(p);  % should give the number of columns
    for j=1:n-1
        q = addq(p(:,j),p(:,j+1));
        pnew = [pnew, q];    % make existing matrix pnew bigger
    end
    p=[pnew, p(:,1)];     % repeating the first point
end
fill(p(1,:),p(2,:),'y')
axis equal off

% a function below the main script
function q = addq(p1,p2)
r = (p2-p1)/3;
q1 = p1 + r;
t = pi/3;             % rotation angle
R = [cos(t), sin(t); -sin(t), cos(t)];  % rotation matrix, clockwise
```

```
q2 = q1 + R*r;
q3 = p1 + 2*r;
q = [p1,q1,q2,q3];
end
```

Starting from a triangle and using five levels (`mmax`= 5), we obtan the right figure above. The true Koch curve is the one for `mmax`= ∞. The length of Koch curve is infinite. Just to have some fun, I replace the line for $\mathbf{q}_2$ in `addq` by

```
q2 = q1 -1.1*R*r;
```

and get the left figure below. If I change the rotation angle to `t=-pi/3`, I get the righ figure below.



2. Polynomial roots. MATLAB can find the roots of a polynomial by the command `roots`. For

$$p(x) = a_1x^n + a_2x^{n-1} + ... + a_nx + a_{n+1}$$

the coefficient vector is $\mathbf{a} = [a_1, a_2, ..., a_{n+1}]$, then you can get the roots by `roots(a)`. For example

```
>> a = [1, -6, 11, -6]
a =
     1    -6    11    -6
>> roots(a)
ans =
   3.000000000000002
   1.999999999999998
   1.000000000000000
```

The exact roots of the polynomial with that coefficient vector `a` should be integers 1, 2 and 3. We see some small errors already. MATLAB also has a program called `poly` that calculates the polynomial coefficients for given roots. For example,

```
>> rs = 1:3
rs =
```

```
     1     2     3
>> poly(rs)
ans =
     1    -6    11    -6
```

Let us try to write our own code for `poly`. For that, we need to think about the step of multiplying $(x - r_j)$. That is,

$$\begin{aligned}&(a_1x^{j-1} + a_2x^{j-2} + ... + a_j)(x - r_j)\\&= a_1x^j + (a_2 - a_1r_j)x^{j-1} + (a_3 - a_2r_j)x^{j-2} + ... + (a_j - a_{j-1}r_j)x + (a_{j+1} - a_jr_j)\end{aligned}$$

where we have set $a_{j+1} = 0$. We want to replace $a_{j+1} - a_jr_j$ by a new a_{j+1}, then replace $a_j - a_{j-1}r_j$ by a new a_j, and finally, replace $a_2 - a_1r_j$ by a new a_2. Here is my program:

```
% give r1, r2, r3, ..., rn
% find a1, a2, ..., a_(n+1), such that
% (x-r1)(x-r2)... (x-rn) = a1 x^n + a2 x^(n-1) + ... + a_(n+1)
%
function a = mypcoef(r)
n = length(r);
a = zeros(1,n+1);
a(1) = 1;
for j=1:n
    for k=j+1:-1:2        % backwards, for decreasing k
        a(k)=a(k)-a(k-1)*r(j);
    end
end
end
```

Now, we have

```
>> mypcoef(rs)
ans =
     1    -6    11    -6
```

Earlier, we see small errors when `roots` is used to solve $x^3 - 6x^2 + 11x - 6 = (x-1)(x-2)(x-3) = 0$. Now, we consider $p(x) = (x - 1)(x - 2)...(x - 20)$.

```
>> rs = 1:20
rs =
  Columns 1 through 16
     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15    1
  Columns 17 through 20
    17    18    19    20
>> a = poly(rs);
>> roots(a)
ans =
  19.999874055724192
```

```
  19.001295393676987
  17.993671562737585
  17.018541647321989
  15.959717574548915
  15.059326234074415
  13.930186454760916
  13.062663652011070
  11.958873995343460
  11.022464271003383
   9.991190949230132
   9.002712743189727
   7.999394310958664
   7.000096952230211
   5.999989523351082
   5.000000705531480
   3.999999973862455
   3.000000000444877
   1.999999999998383
   0.999999999999949
```

Now, we see that the errors for some roots are very large. If you do the rounding, there are only 2 correct digits for roots 13, 14 and 15. Why MATLAB produces such inaccurate results, yet we still spend money to buy it? That is because there is no way to do it better. The roots of a polynomial are often very sensitive to the coefficients. Since MATLAB uses double-precision floating point numbers, only about 16 digits are kept in all the numbers. The small errors in the computation could eventually lead to very inaccurate solutions. It is easy to see that

$$(x-1)(x-2)...(x-20) = x^{20} - 210x^{19} + ...$$

Wilkinson proposed to perturb the coefficient -210 by a small amount, and check the roots of the perturbed polynomial. We add -10^{-9} to -210, and get the following

```
>> rs = 1:20;
>> a = poly(rs);
>> a(2) = a(2)-1.0e-9;
>> a(2)
ans =
    -2.100000000010000e+02
>> roots(a)
ans =
 20.039813559447666 + 0.000000000000000i
 18.674529328758680 + 0.362401392514759i
 18.674529328758680 - 0.362401392514759i
 16.580239938319032 + 0.904752660971922i
 16.580239938319032 - 0.904752660971922i
 14.379230160362074 + 0.819866950038492i
 14.379230160362074 - 0.819866950038492i
 12.371751653022132 + 0.260330199806903i
```

```
 12.371751653022132 - 0.260330199806903i
 10.938384167179578 + 0.000000000000000i
 10.010823659914928 + 0.000000000000000i
  8.999745911541700 + 0.000000000000000i
  7.999643456507326 + 0.000000000000000i
  7.000099195412345 + 0.000000000000000i
  5.999987161655663 + 0.000000000000000i
  5.000000725730995 + 0.000000000000000i
  4.000000004576920 + 0.000000000000000i
  2.999999998066851 + 0.000000000000000i
  2.000000000042657 + 0.000000000000000i
  0.999999999999891 + 0.000000000000000i
```

While the command `roots` does not give accurate solsutions, it is definitely true that the perturbed polynomial does not have 20 real roots. Instead, it has 4 complex conjugate pairs and 12 real roots.

3. Ill-conditioned linear systems. If A is a square invertible matrix (also called non-singular matrix), then the linear system $A\mathbf{x} = \mathbf{b}$ has a unique solution. The famous Hilbert matrix with (i, j) entry $1/(i + j - 1)$ is invertible. It even has an internal MATLAB command `hilb`. For example,

```
>> A = hilb(3)
A =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

Now, suppose the solution $\mathbf{x} = [1, 1, ..., 1]^T$ is known, we can get the right hand side by matrix-vector multiplication, then try to solve $\mathbf{x}$. In principle, we should get the exact correct solution, but look at the example below for the 12×12 Hilbert matrix.

```
>> A = hilb(12);
>> x = ones(12,1);
>> b = A*x;
>> xx = A\b
xx =
   0.999999953328950
   1.000005957492877
   0.999811517811304
   1.002581059925782
   0.980997425546507
   1.083799188899740
   0.765776881515429
   1.425134190494669
   0.500393563981414
   1.366669356762649
   0.847266909645277
   1.027564028520094
```

So, we get wrong answers without doing anything wrong! You also see a warning message saying that the matrix is near singular, RCOND $= 2.6845 \times 10^{-17}$, and the results may be inaccurate. The so-called condition number, $\kappa(A)$, is the ratio between the largest singular value to the smallest singular value of A. If $\kappa(A)$ is very large, then the linear system $\mathbf{Ax} = \mathbf{b}$ is ill-conditioned, and accurate solutions are difficult or even impossible to get. For this Hilbert matrix, the condition number is

```
>> cond(A)
ans =
     1.621163904747500e+16
```

Here `rcond` is the reciprocal condition estimator. It is some approximation to $1/\kappa(A)$. It is not exactly $1/\kappa(A)$, because $\kappa(A)$ is more expensive to calculate.

Eigenvalue problems can also be "ill-conditioned", although the definition of condition number for eigenvalues is different. For ill-conditioned eigenvalue problems, accurate eigenvalues are difficult or impossible to get.

4. Exercise: *Write a MATLAB script with two functions. The first function (call it* `polyg`*) inputs a positive integer n and outputs n corners of a regular polygon with n sides, assuming the corners are on the unit circle, and there must be a horizontal bottom side. The second function (call it* `rdpoly`*) inputs a large integer m and n corners of a polygon (z_1, z_2, ..., z_n), outputs m points p_j for $j = 1$, 2, ..., m, by choosing p_1 as a random point in $[-0.5, 0.5] \times [-0.5, 0.5]$, and generating other points by*

$$p_{j+1} = p_j + 0.6(z_k - p_j)$$

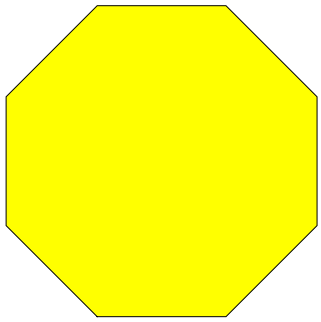
where $k \in \{1, 2, ..., n\}$ is randomly chosen for each j. The main script draws two figures. The first one is just a regular polygon with $n = 8$ sides (filled with some color). The second one shows $m = 10000$ points produced by `rdpoly` *for $n = 5$ as dots.*

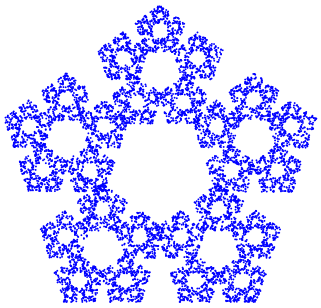
Solution: For this problem, it is convenient to use complex numbers. A point (x, y) in the plane, now becomes a complex number $z = x + iy$. I have the following MATLAB program.

```
% Main script
% Part 1
z = polyg(8);
subplot(1,2,1)
fill(real(z),imag(z),'y')
axis equal off
% Part 2
n = 5;
m = 10000;
z = polyg(n);
p = rdpoly(z,m);
subplot(1,2,2)
plot(real(p),imag(p),'b.','MarkerSize',1)
axis equal off

% a function to generate the corners of a polygon
function z = polyg(n)
```

```
    t = 2*pi/n;
    t0 = -pi/2-t/2;
    tt = t0 + t*(0:n-1);
    z = exp(1i*tt);
end

% another function below the main script
function p = rdpoly(z,m)
    n = length(z);
    p(1) = rand-0.5 + 1i*(rand-0.5);
    for j=1:m-1
        k = randi(n);
        p(j+1) = p(j)+0.6*(z(k)-p(j));
    end
end
```

It produces the figures below.



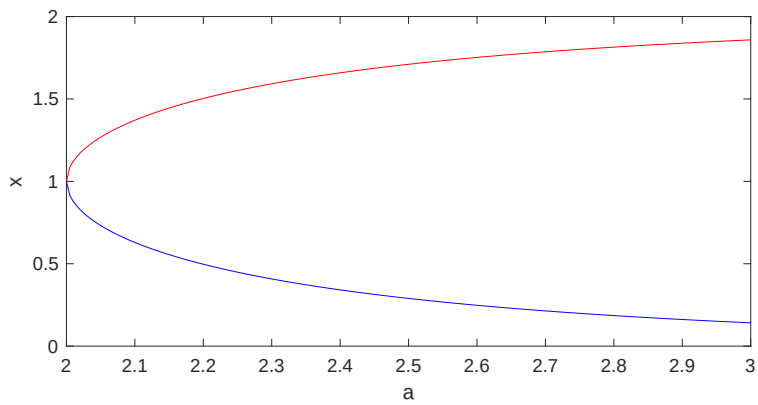


MA2507 Computing Mathematics Laboratory: Week 8

1. Anonymous function. This is a quick method to define short and simple functions. Consider the function $f(x,a) = xe^{a(1-x)} + x - 2$. You can verify that $f(1,a) = 0$. For $a > 2$, we have two more solutions $x = p_1(a)$ and $x = p_2(a)$ satisfying $p_1 + p_2 = 2$. In the following program, we solve p_1 and p_2 for $2 \le a \le 3$, and show p_1 and p_2 as functions of a. To solve p_1, we use MATLAB internal program `fzero`. It requires a single-variable function and an initial guess.

```
n=200;
a = linspace(3,2,n);     % start at a=3, go down to a=2
x0 = 2;                  % initial guess for a=3
for j=1:n
    f = @(x) x*exp(a(j)*(1-x))+x-2;
    p(j) = fzero(f, x0);
    x0 = p(j);           % use p(j) as initial guess for next a
end
% plot p and (2-p) as functions of a
plot(a, p, 'r', a, 2-p, 'b')
xlabel('a'), ylabel('x')
```

At $a = 2$, the two solutions merge to $x = 1$, thus it is easier to solve p_1 for $a = 3$. Therefore, we vary a from 3 to 2. Once p_1 is solved for a_j, we use the solution as the initial guess to solve the equation for a_{j+1}. The program gives the following figure.



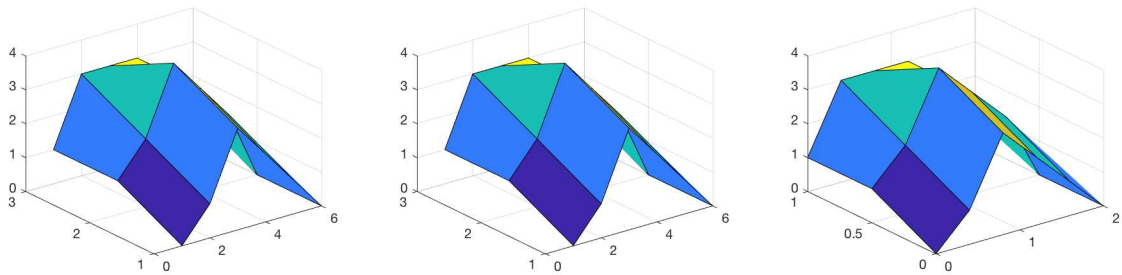
2. 2D plots. MATLAB has a few different commands for plotting 2D figures. First, we take a look at `surf`. Consider the following program

```
f = [0 1 3 2 1 0; 1 2 4 3 2 0; 1 3 3 3 2 1]
subplot(1,3,1), surf(f)
jj = 1:6;
ii = 1:3;
subplot(1,3,2), surf(jj,ii,f)
x = linspace(0,2,6);
y = linspace(0,1,3);
subplot(1,3,3), surf(x,y,f)
```

It gives the matrix `f`:

```
f =
     0     1     3     2     1     0
     1     2     4     3     2     0
     1     3     3     3     2     1
```

and the figure below. The first plot uses only the matrix `f` as the input, the two horizontal axes



are just integers from 1 to 6 (shown as from 0 to 6), and from 1 to 3. In our usual coordinate system, the x direction corresponds to integer vector $[1, 2, ..., 6]$, and the y direction corresponds to integer vector $[1, 2, 3]$. As a test, I define two integer vectors `jj` and `ii` and plot the figure again, using `jj` and `ii` also as inputs. This gives the second plot which is identical to the first one. Usually, we think the matrix `f` is obtained from a function $f(x, y)$ for some discrete values of x and y. Let us assume x is from $[0, 2]$ and y is from $[0, 1]$, then I sample x by 6 points and sample y by 3 points, and now use x and y as inputs. This gives the third plot. It is still the same figure. Since the first two plots actually show a horizontal axis from 0 to 6 (instead of from 1 to 6), we see a slight difference betwen the third and the first two plots. Importanly, the (i, j) entry of `f` is NOT $f(x_i, y_j)$, it is $f(x_j, y_i)$. Namely,

$$f_{ij} = f(x_j, y_i)$$

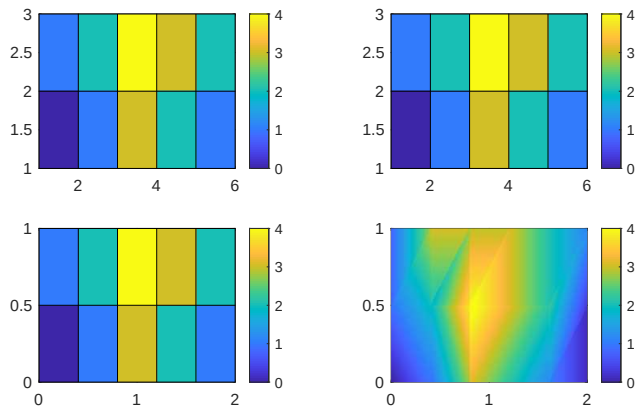
We have six x_j (for $1 \leq j \leq 6$) and three y_i (for $1 \leq i \leq 3$), the above gives a 3×6 matrix which is exactly right. If you set $\mathtt{f}_{ij} = f(x_i, y_j)$, you will get a 6×3 matrix. Finally, we see that `surf` gives continuous and piecewise plane segments. That means `surf` interpolates the points linearly.

A more useful command is `pcolor`. Let us look at the following program.

```
f = [0 1 3 2 1 0; 1 2 4 3 2 0; 1 3 3 3 2 1];
subplot(2,2,1)
pcolor(f), colorbar
jj = 1:6;
ii = 1:3;
subplot(2,2,2)
pcolor(jj,ii,f), colorbar
x = linspace(0,2,6);
y = linspace(0,1,3);
subplot(2,2,3)
pcolor(x,y,f), colorbar
subplot(2,2,4)
```

```
pcolor(x,y,f), colorbar
shading interp
```

The program produces the following figures. The first two are identical. That means if we only



input the matrix `f`, MATLAB uses interger vectors as the x and y coordinates. In the 3rd and 4th plots, we force MATLAB to use our own x and y vectors. The first three plots show piecewise constant figures. There is no interpolation. More importantly, notice that only two rows and five columns are shown (instead of 3 rows and 6 columns). What we see in the first three plots are 2×5 elements of `f` arranged as

$$\begin{matrix} f_{21} & f_{22} & f_{23} & f_{24} & f_{25} \\ f_{11} & f_{12} & f_{13} & f_{14} & f_{15} \end{matrix}$$

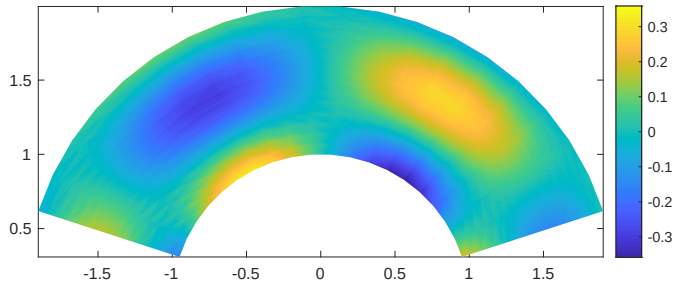
The first row is at the bottom. That is good, because we usually want to have y increase in the vertical direction. The last row and last column are missing. You can get the last row, last column back by `shading interp`. It also gives you linear interpolation, so that the figure is now continuous. This is what we see in the 4th plot.

Another way of using `pcolor` is to input three matrices X, Y and Z. This is useful, if the area we are plotting is not a rectangle in the xy plane. The following example plots a function for $r \in [1, 2]$ and $\theta \in [0.1\pi, 0.9\pi]$, where r and θ are polar cooridnates. The function is $J_3(5r)\cos(3\theta)$, where $J_m(z)$ is a Bessel function.

```
m = 15;
r = linspace(1,2,m);
n = 20;
t = pi*linspace(0.1,0.9,n);
for j=1:n
    for i=1:m
        Z(j,i) = myfn(r(i),t(j));
        X(j,i) = r(i)*cos(t(j));
        Y(j,i) = r(i)*sin(t(j));
    end
end
pcolor(X,Y,Z), colorbar
shading interp
```

```
axis equal tight

function f = myfn(r,t)
f = besselj(3,5*r)*cos(3*t);
end
```

The program gives the following figure.



3. Finding minima. To find a minimum of a function, we can use `fminbnd`. For example, the following lines

```
>> f =@(x) x^4+x^2 - 3*sin(x);
>> format long
>> fminbnd(f,0,2)
ans =
   0.648846918337408
```

find a minimum of $f(x) = x^4 + x^2 - 3\sin(x)$ in the interval $[0, 2]$. It gives you the position of x where the minimum is achieved. If you want to know the minimum value of f, just put that x into the function f. The first input of `fminbnd` is the name of a function. Since the first line above defines the function f, you do not need to add the symbol `@`. But for the following script file

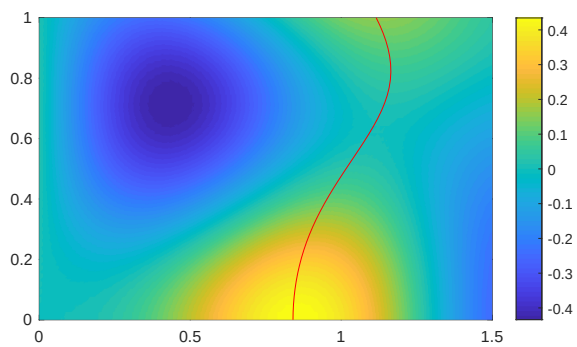
```
% main script
format long
fminbnd(@f,0,2)
%
% function defined after the main script
function y=f(x)
y = x^4+x^2 - 3*sin(x);
end
```

the function f is defined below the main script. You need to add `@` in front of f. Otherwise, MATLAB will give you an error message.

Now for the function $f(x, y) = J_3(5r)\cos(3\theta)$, we first plot it in the rectangle $[0, 1.5] \times [0, 1]$, then find the maximum of f, as a function of x only, for each fixed $y \in [0, 1]$.

```
% plot f(x,y) on [0,1.5]x[0,1]
n = 60;
x = linspace(0,1.5,n);
m = 40;
y = linspace(0,1,m);
for i=1:m
    for j=1:n
        F(i,j) = myfn(x(j),y(i));
    end
end
pcolor(x,y,F), shading interp, colorbar, axis equal tight
%
% find max of f(x,y) for each fixed y
for i=1:m
    g =@(x) -myfn(x,y(i));     % multiple -1, max becomes min
    xx(i) = fminbnd(g,0,1.5);
end
hold on
plot(xx,y,'r')
hold off

% function J_3(5r)cos(3theta)
function f = myfn(x,y)
z = x+1i*y;
r = abs(z);
t = angle(z);
f = besselj(3,5*r)*cos(3*t);
end
```

Here, we use a loop for different y, define a function of x for each y, then use `fminbnd`. The positions of the maxima (for each y) are plotted as a curve on top of the figure for f. The program gives the following figure.



4. SVD and image compression. An $m \times n$ matrix F may have many small singular values. For a real matrix, the so-called singular value decomposition (SVD) is the relation $F = USV^{\mathsf{T}}$, where

U and V are two orthogonal matrices (an orthogonal matrix is one whose inverse is simply the transpose), S is a diagonal matrix with the singular values on the diagonal, that is

$$S = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \ddots \end{bmatrix},$$

where $\sigma_1 \geq \sigma_2 \geq ... \geq 0$. If there are many small singular values, we can choose an integer k, and replace σ_j by 0 for $j > k$. This means that F is approximated by

$$F_k = \sum_{i=1}^{k} \sigma_i u_i v_i^{\mathsf{T}},$$

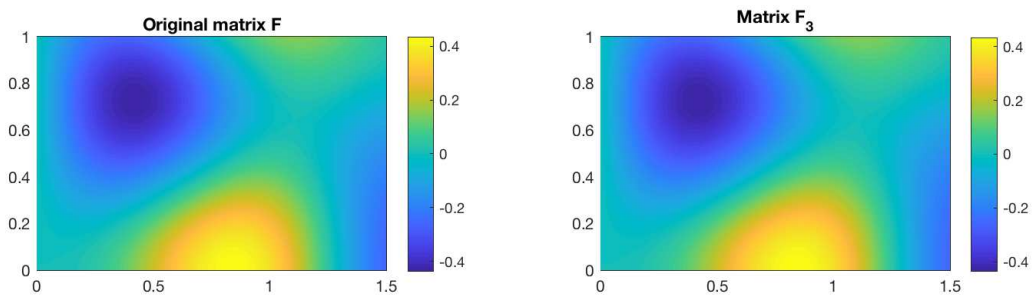
where u_1, u_2, ..., are the columns of U, v_1, v_2, ..., are the columns of V. The following program constructs a matrix F for the function $J_3(5r)\cos(3\theta)$ on $[0, 1.5] \times [0, 1]$, calculates its SVD, approximates F by F_3 (keeping only three non-zero singular values), shows the first 5 singular values, and compares the figures for F and F_3.

```
% main script
m = 100;
y = linspace(0,1,m);
n = 150;
x = linspace(0,1.5,n);
for i=1:m
    for j=1:n
        F(i,j) = myfn(x(j),y(i));
    end
end
subplot(1,2,1)
pcolor(x,y,F), title('Original matrix F')
shading interp, colorbar, axis equal tight
%
[U,S,V] = svd(F);
S(1:5,1:5)
F3 = U(:,1:3)*S(1:3,1:3)*(V(:,1:3))';
subplot(1,2,2)
pcolor(x,y,F3), title('Matrix F_3')
shading interp, colorbar, axis equal tight

% a function below the main script
function f = myfn(x,y)
z = x+1i*y;
r = abs(z);
t = angle(z);
f = besselj(3,5*r)*cos(3*t);
end
```

The first five singular values are shown by the line `S(1:5,1:5)`.

```
ans =
   19.8646         0         0         0         0
         0   13.4750         0         0         0
         0         0    3.1757         0         0
         0         0         0    0.0037         0
         0         0         0         0    0.0001
```

The program produces the following two figures. It is hard to see any difference between the



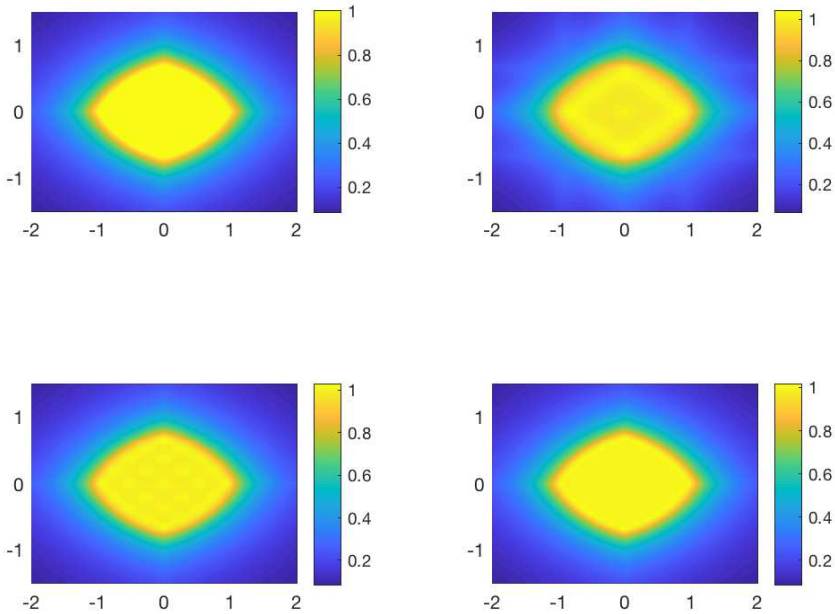
two. For F_k, it is only necessary to store k columns of U, k columns of V, and k singular values, Therefore, if F represents an image, we have a way to compress it. That is useful for storage and transmission of images.

5. Exercise: *Let $g(x,y) = 1/(x^2 + 2y^2 + |xy| + 0.0001)$, and $f(x,y) = g(x,y)$ if $g(x,y) \leq 1$ and $f(x,y) = 1$ if $g(x,y) > 1$. Construct a 150×200 matrix F for function f on $[-2,2] \times [-1.5,1.5]$. For $k = 3$, 6 and 9, approximate F by F_k using SVD and keeping k non-zero singular values. Compare the images of F, F_3, F_6 and F_9.*
Solution: The program is given below.

```
% main script
n = 200;
x = linspace(-2,2,n);
m = 150;
y = linspace(-1.5,1.5,m);
for i=1:m
    for j=1:n
        F(i,j) = f(x(j),y(i));
    end
end
subplot(2,2,1)
pcolor(x,y,F), shading interp, colorbar, axis equal tight
%
[U,S,V]  = svd(F);
for p=1:3
    k = 3*p;
    Fk = U(:,1:k)*S(1:k,1:k)*(V(:,1:k))';
```

```
    subplot(2,2,p+1)
    pcolor(x,y,Fk), shading interp, colorbar, axis equal tight
end

% function f(x,y)
function z =f(x,y)
    z = 1/(x^2 + 2*y^2 + abs(x*y)+0.0001);
    if z > 1
        z = 1;
    end
end
```

We get the following figure.



MA2507 Computing Mathematics Laboratory: Week 9

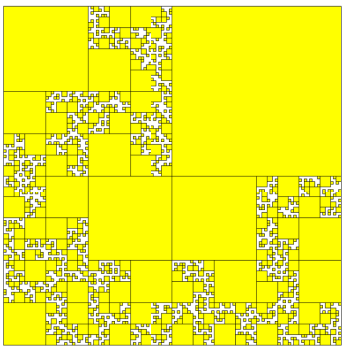
1. Recursive functions. A recursive function is a function that calls itself. Sometimes, you can write elegant programs using recursive functions. Our first example is *random boxes*. We have a recursive function `drawbox(x0,x1,y0,y1,m)`, where $x0$, $x1$, $y0$, $y1$ are the coordinates of a square, m is a control integer. If m is positive, the program divides the square into four smaller ones, colors one small square by yellow at random, and continues `drawbox` on the other three smaller squares with control integer $m-1$. The main program draws a unit square and then carries out `drbox` for some small and positive m.

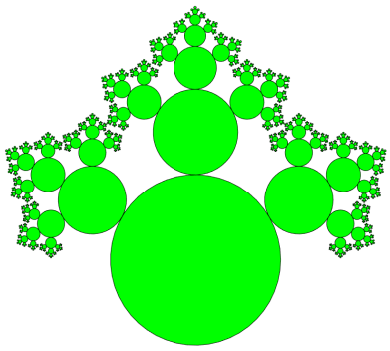
```
% main script
clf
plot([0 1 1 0 0],[0 0 1 1 0],'k')
hold on
axis equal off
drawbox(0,1,0,1,7)
hold off

% recursive function, after the main script
function drawbox(x0,x1,y0,y1,m)
% given a square [x0,x1] x [y0, y1]
if m > 0
    x = [x0, (x0+x1)/2, x1]; % x-coordinates of smaller squares
    y = [y0, (y0+y1)/2, y1]; % y-coordinates of smaller squares
    k = randi(4);
    for i=1:2
        for j=1:2
            kk = 2*i+j-2;   % convert (i,j) to kk in {1, 2, 3, 4}
            if kk == k
                fill([x(i),x(i+1),x(i+1),x(i)],[y(j),y(j),y(j+1),y(j+1)],'y')
            else
                drawbox(x(i),x(i+1),y(j),y(j+1),m-1)
            end
        end
    end
end
end
```

We get the yellow figure on the next page.

As another example, we write a program to draw many disks of different size. The recursive function is `w9exr(c,r,a,m)` where c is the coordinates of the the current disk, r is the radius, m is an integer to control the level, a is an angle with the y axis for one of the three disks at level $m-1$. These three disks are tangent to the current disk, and they have radius $0.5r$, $0.4r$ and $0.4r$, respectively. The straight lines connecting the center of the current disk to the centers of the three smaller disks form angles a, $a+\pi/3$ and $a-\pi/3$ with the y axis, respectively. The main script stats the program with a unit disk and angle $a=0$ for some $m>0$.

```
% main script
```





```
clf
hold on
w9exr([0,0],1,0,6)
axis equal off
hold off

% a function after the main script
function w9exr(c,r,a,m)
% c -- vector for the location of the center
% r -- radius of the current disk
% a -- angle for central disk of the next level
t = linspace(0,2*pi,50);
x=c(1)+r*cos(t);
y=c(2)+r*sin(t);
fill(x,y,'g')
if m > 0
    c1 = c + 1.5*r*[sin(a),cos(a)];
    w9exr(c1,0.5*r,a,m-1);
    a2 = a +pi/3;
    c2 = c + 1.4*r*[sin(a2),cos(a2)];
    w9exr(c2,0.4*r,a2,m-1);
    a3 = a - pi/3;
    c3 = c + 1.4*r*[sin(a3),cos(a3)];
    w9exr(c3,0.4*r,a3,m-1);
end
end
```

The above program gives the green figure above.

2. Fast Fourier Transform: In the following, we write a recursive function to implement the so-called fast Fourier transform (FFT). FFT is an efficient method for computing the discrete Fourier transform (DFT). DFT is the following matrix-vector multiplication:

$$\boldsymbol{y} = \mathbf{F}_N \boldsymbol{x}, \tag{1}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors of length N, $\mathbf{F}_N$ is an $N \times N$ matrix whose $(j+1, k+1)$ entry

equals to ω^{jk} where $\omega = e^{i2\pi/N}$. More explicitly,

$$\mathbf{F}_N = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix}.$$

The matrix $\mathbf{F}_N$ is symmetric, but not Hermitian. It is not unitary, but it is close enough. We can prove that

$$\mathbf{F}_N \overline{\mathbf{F}}_N = N\mathbf{I} \tag{2}$$

Therefore, $(1/\sqrt{N})\mathbf{F}_N$ is unitary. Usually, when DFT is studied, it is easier to start index from 0 for the vectors $\boldsymbol{y}$ and $\boldsymbol{x}$, that is

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}.$$

Now Eq. (1) implies

$$y_j = \sum_{k=0}^{N-1} \omega^{jk} x_k = \sum_{k=0}^{N-1} x_k e^{i2\pi jk/N}, \quad j = 0, 1, ..., N-1. \tag{3}$$

From Eqs. (2) and (1), we get

$$\boldsymbol{x} = \frac{1}{N} \overline{F}_N \mathbf{y} \tag{4}$$

which is equivalent to

$$x_k = \frac{1}{N} \sum_{j=0}^{N-1} y_j e^{-i2\pi jk/N}, \quad k = 0, 1, ..., N-1. \tag{5}$$

Now, we describe FFT assuming N is a power of 2. We split $\boldsymbol{y}$ into two vectors of length $N/2$ by

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix},$$

and put the even and odd elements of $\boldsymbol{x}$ into two vectors

$$\boldsymbol{x}_1 = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ \vdots \end{bmatrix}, \quad \boldsymbol{x}_2 = \begin{bmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \end{bmatrix}.$$

The basic principle of FFT is the following. If $\boldsymbol{a} = \mathbf{F}_{N/2}\boldsymbol{x}_1$ and $\boldsymbol{b} = \mathbf{F}_{N/2}\boldsymbol{x}_2$, then

$$\boldsymbol{y}_1 = \boldsymbol{a} + \mathbf{W}\boldsymbol{b}, \quad \boldsymbol{y}_2 = \boldsymbol{a} - \mathbf{W}\boldsymbol{b},$$

where $\mathbf{W}$ is an $(N/2) \times (N/2)$ diagonal matrix with diagonal entries 1, ω, ω^2, ... You can prove this result starting from Eq. (3). It is all elementary mathematics. Notice that $e^{i2\pi} = 1$.

Now, we have the following very short recursive program for FFT.

```
% Fast Fourier Transform (FFT), assuming N is power of 2.
% This is equivalent to N*ifft(x) in MATLAB.
function y=yfft(x)
N = length(x);
omg = exp(1i*2*pi/N);
w = omg.^((0:N/2-1)');
if N == 1
    y=x;
else
    a = yfft(x(1:2:N-1));
    b = yfft(x(2:2:N));
    c = w.*b;
    y = [a+c; a-c];
end
```

This program is not optimal. Notice that the powers of ω are repeatedly calculated. This can be avoided. FFT is a very important algorithm, since it evaluates DFT very quickly. DFT is related to Fourier transform and Fourier series which are important in signal processing and many other applications.

3. Exercise: *For a function $f(x)$, its Fourier transform can be defined as*

$$g(y) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi xy} dx. \tag{6}$$

The most important property is that $f(x) = \int_{-\infty}^{\infty} g(y) e^{i2\pi xy} dy$. Use FFT, calculate (approximately) the Fourier transform of $f(x) = [\cos(6\pi x) + \sin(7\pi x)]e^{-\pi x^2}$. Suggestions: truncate x to $[-L, L)$ for $L = 20$ with $N = 2m = 4096$ points, truncate y to $[-m/(2L), (m-1)/(2L)]$ with also $2m$ points, plot $f(x)$ on $[-2, 2]$ and plot real and imaginary parts of g on $[-7, 7]$.

Solution: We need to approximate the integral relation and write it to something like Eq. (5). Define $x_j = jL/m$ for $j = -m, -m+1, ..., 0, 1, ..., m-1$, define $y_k = k/(2L)$ for $k = -m, -m+1, ..., m-1$. Notice that $x_j y_k = jk/(2m) = jk/N$. We can approximate Eq. (6) as

$$g(y_k) = \frac{L}{m} \sum_{j=-m}^{m-1} f(x_j) e^{-i2\pi x_j y_k} = \frac{L}{m} \sum_{j=-m}^{m-1} f(x_j) e^{-i2\pi jk/N}, \quad k = -m, -m+1, ..., m-1.$$

The above is somewhat similar to Eq. (5), but the ranges for j and k are different. For DFT, we should have indices from 0 to $N-1$. Now, suppose j is negative, we can add N to it and define $j+N$ as the new j. Notice that since $e^{-i2\pi k} = 1$ for any integer k, we have $e^{-i2\pi jk/N} = e^{-i2\pi(j+N)k/N}$. Similarly, for a negative k, we can replace it by $k + N$. This implies that if we can define

$$\tilde{\mathbf{f}} = \begin{bmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \vdots \\ \tilde{f}_{N-1} \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{m-1}) \\ f(x_{-m}) \\ \vdots \\ f(x_{-2}) \\ f(x_{-1}) \end{bmatrix}, \quad \tilde{\mathbf{g}} = \begin{bmatrix} \tilde{g}_0 \\ \tilde{g}_1 \\ \tilde{g}_2 \\ \vdots \\ \tilde{g}_{N-1} \end{bmatrix} = \begin{bmatrix} g(y_0) \\ g(y_1) \\ \vdots \\ g(y_{m-1}) \\ g(y_{-m}) \\ \vdots \\ g(y_{-2}) \\ g(y_{-1}) \end{bmatrix}$$

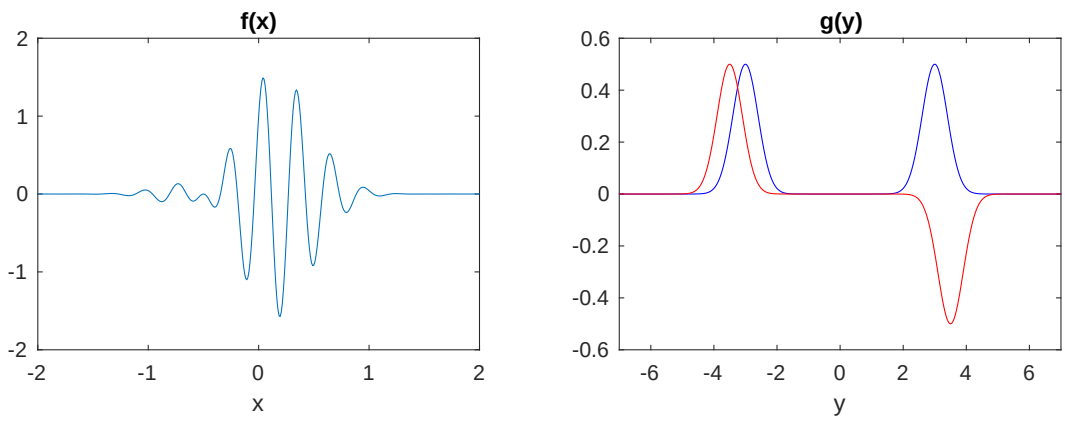
then

$$\tilde{g}_k = \frac{L}{m} \sum_{j=0}^{N-1} \tilde{f}_j e^{-i2\pi jk/N}, \quad k = 0, 1, ..., N-1.$$

In MATLAB, the program `fft` calculates above without the coefficient L/m. So we just have to multiply L/m. Our program is as follows.

```
% main script for Fourier transform
L = 20;
m = 2048;
N = 2*m;
x = (L/m)* (-m: m-1);
f = myfn(x);
subplot(1,2,1)
plot(x,f), xlabel('x'), title('f(x)')
xlim([-2 2])      % show plot from x=-2 to x=2
ff = [f(m+1:N), f(1:m)];
gg = fft(ff)*(L/m);
g = [gg(m+1:N), gg(1:m)];
y = (-m : m-1)/(2*L);
subplot(1,2,2)
plot(y, real(g),'b',y, imag(g),'r'), xlabel('y'), title('g(y)')
xlim([-7, 7])

function z = myfn(x);
z = (cos(6*pi*x)+sin(7*pi*x)).* exp(-pi*x.^2);
end
```

The program gives the following figure.



MA2507 Computing Mathematics Laboratory: Week 10

1. Lorenz equation: In a standard ODE course, you learn analytic methods for solving ordinary differential equations, including first order ODEs, second and higher order ODEs, and system of ODEs. Notice the major difference between linear and nonlinear ODEs. Most nonlinear ODEs cannot be solved analytically. Most linear ODEs have "simple" hehavior. If the ODE is for a function $y(t)$, then as $t \to \infty$, $y(t)$ may converge to a constant, may diverge, or may approach a periodic solution. Can ODEs have more exotic behavior? Yes, it was first observed by Edward Lorenz in 1963 in the following system of ODEs:

$$\begin{aligned} x' &= -\sigma(x-y) \\ y' &= -xz + Rx - y \\ z' &= xy - bz, \end{aligned}$$

where σ, R and b are constants. For $R = 28$, $\sigma = 10$ and $b = 8/3$, Lorenz found chaos (the mathematical term chaos was only introduced in the 70's). The solution does not converge to a constant or a periodic solution. Two solutions that are very close at $t = 0$ separate rapidly (actually exponentially) as t is increased. This is called "sensitive dependence on initial conditions", also called "butterfly effect". Lorenz is a meteorologist, he realized that if the differential equations for weather also have "sensitive dependence on initial conditions", then long-term weather forecast will never be possible. Indeed, this is now widely believed. A mathematical proof of chaos in Lorenz equation was only completed around year 2000.

We can solve the Lorenz equation using MATLAB internal program `ode45`. To do that, we must introduce a column vector $\mathbf{y} = [x, y, z]^T$, since the ODE solver only works for first order systems written as

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \tag{1}$$

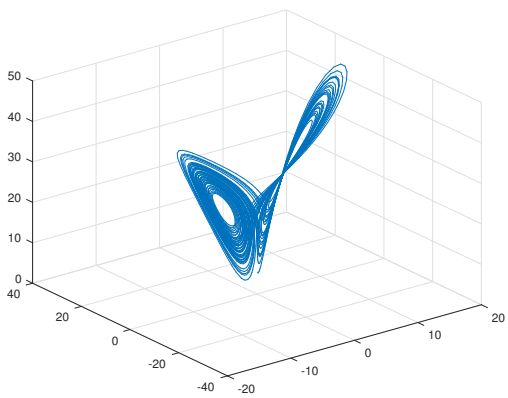
and we need a function for the right hand side of Eq. (1) as a column vector. Below is a MATLAB program.

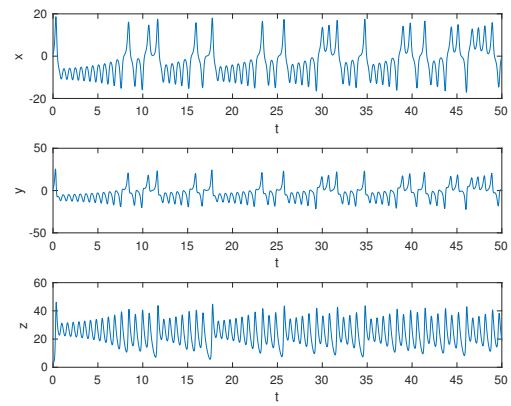
```
% main script
tspan = [0,50];                 % from t=0 to t=50
yzero = [1, 2.3, 4.1];          % initial conditions
[tt,yy] = ode45(@lorenz,tspan,yzero);
plot3(yy(:,1),yy(:,2),yy(:,3))  % 3D curves
grid

% a function for the right hand side of the ODE system, as a column vector
function k = lorenz(t,y)
s = 10; b = 8/3; r = 28;
k = [s*(y(2)-y(1)); y(1)*(r-y(3))-y(2); y(1)*y(2)-b*y(3)];
end
```

Even though t does not appear in the Lorenz equation, we must define the function as a function of t and $\mathbf{y}$. We use `tspan` to specify the starting and ending time for the solution. `yzero` is a vector of the initial conditions, that is, $x(0)$, $y(0)$ and $z(0)$. The program above gives the figure (3D curve) on next page. The solutions are given in a column vector `tt` and 3-column matrix `yy`. The vector `tt` discretizes t from $t = 0$ to $t = 50$, the columns of `yy` give the corresponding x, y and z. If we replace the last two lines by

```
subplot(3,1,1), plot(tt, yy(:,1)), xlabel('t'), ylabel('x')
subplot(3,1,2), plot(tt, yy(:,2)), xlabel('t'), ylabel('y')
subplot(3,1,3), plot(tt, yy(:,3)), xlabel('t'), ylabel('z')
```

then, we get a figure showing x, y and z as functions of t. You can also use the commands `comet`





and `comet3` to see animated plots.

2. KdV equation: The Korteweg-de Vries (KdV) equation (1895)

$$u_t + uu_x + \delta^2 u_{xxx} = 0,$$

where δ is a constant, models waves in shallow water. The KdV equation has soliton solutions which were first observed by J. Scott Russell on the Edinburgh-Glasgow canal in 1834. Many mathematical properties of the KdV equation were found after the initial numerical study of Zabusky and Kruskal in 1965. They solved the KdV equation for $\delta = 0.022$ and $0 \le x \le 2$ assuming periodic boundary condition $u(x+2,t) = u(x,t)$ for all x and all t, and an initial condition $u(x,0) = \cos(\pi x)$. We will repeat this calculation with our own method. Discretizing x by $x_j = 2j/n$ for $n = 200$ for $1 \le j \le n$, the KdV equation is approximated by the following system of ODEs:

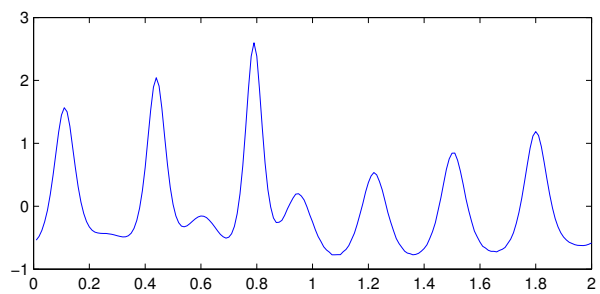
$$\frac{du_j}{dt} = -u_j \frac{u_{j+1} - u_{j-1}}{2d} - \frac{\delta^2}{2d^3}(u_{j+2} - 2u_{j+1} + 2u_{j-1} - u_{j-2}),$$

for $1 \le j \le n$, where $d = 2/n$ and $u_j(t) = u(x_j, t)$. Due to the periodic boundary condition, we need to set $u_0 = u_n$, $u_{-1} = u_{n-1}$, $u_{n+1} = u_1$ and $u_{n+2} = u_2$ in the equations for $j = 1$, 2, $n-1$ and n. The MATLAB program is given below.

```
% main script
n=200;
x =(2/n)*(1:n);
u= cos(pi*x);          % initial condition
tspan = [0, 1.5];      % time interval
[tt,uu] = ode45(@kdv, tspan, u);
m = length(tt);
for j = 1:10:m
    plot(x,uu(j,:))    % show solution every 10 time steps
```

```
    axis([0 2 -1 3])
    pause(0.01);      % show the plot only for 0.01 seconds
end
plot(x,uu(m,:))

% right hand side of the ODE system approximating thge KdV equation
% u, f  -- column vectors
function f = kdv(t,u)
  n = length(u);
  u = [u; u(1:4)];  %  extend the vector u by 4 elements periodically
  f = zeros(n+2,1); %  define a longer f
  delta = 0.022;
  d = 2/n;
  c = delta^2/(2*d^3);
  for j=3:n+2
      f(j)=-u(j)*(u(j+1)-u(j-1))/(2*d)-c*(u(j+2)-2*u(j+1)+2*u(j-1)-u(j-2));
  end
  f(1:2)=f(n+1:n+2);  % get f(1) and f(2)
  f = f(1:n);         % reset f as a vector of length n
end
```

The j loop above gives the time evolution of the solution. The final line plots the solution at $t = 1.5$.



3. Water droplet: The cross-sectional shape of a water droplet on a flat surface is given by the following ODE

$$u'' + (1-u)\left[1+(u')^2\right]^{3/2} = 0, \quad -1 < x < 1,$$

with boundary conditions

$$u(-1) = u(1) = 0,$$

where $u = u(x)$ is a function of x, the base of the droplet is assumed to be $[-1, 1]$. This is an ODE boundary value problem (BVP). To solve it in MATLAB, we must write the second order ODE as a first order system, for $\mathbf{y} = [u, u']^T$, and must also write down the two boundary conditions in a column. The first order ODE system is

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y})$$

and we need a function for $\mathbf{f}$. The boundary conditions are

$$\mathbf{r} = \mathbf{r}(\mathbf{y}_a, \mathbf{y}_b) = \mathbf{0}$$

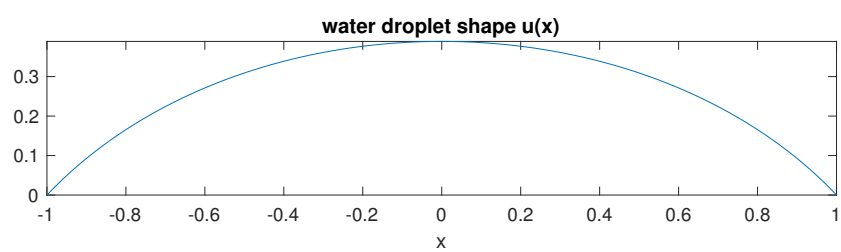
where

$$\mathbf{y}_a = \begin{bmatrix} u(-1) \\ u'(-1) \end{bmatrix}, \quad \mathbf{y}_b = \begin{bmatrix} u(1) \\ u'(1) \end{bmatrix},$$

and we need a function for $\mathbf{r}$. This is a nonlinear ODE, we also need an initial guess. The initial guess is for $\mathbf{y}$. Here, we will use $0.5\cos(\pi x/2)$ as the initial guess for $u(x)$, then the initial guess for $u'(x)$ is $-0.25\pi\sin(\pi x/2)$. We use MATLAB program `bvp4c` to solve the ODE BPV. It requires `bvpinit` to set up the initial data. Here is the propgram.

```
% main script
x = linspace(-1,1,100);    % discretize x
s0 = bvpinit(x,@myigs);    % s0 is a structure for initial data
s = bvp4c(@myde,@mybc,s0); % s is a structure for the final solution
plot(s.x, s.y(1,:))
axis equal tight
xlabel('x'), title('water droplet shape u(x)')

% function for initial guess
function y0 = myigs(x)
  y0 = [0.5*cos(pi*x/2); -0.25*pi*sin(pi*x/2)];
end

% function for the ODE written as a 1st order system
function f = myde(x,y)
  f = [y(2); (y(1)-1)*(1 + (y(2))^2)^(3/2)];
end

% function for boundary conditions, writte as a column vector
function r = mybc(ya,yb)
  r = [ya(1); yb(1)];
end
```

The program gives the following figure.



4. Exercise: *Solve the following ODE BVP:*

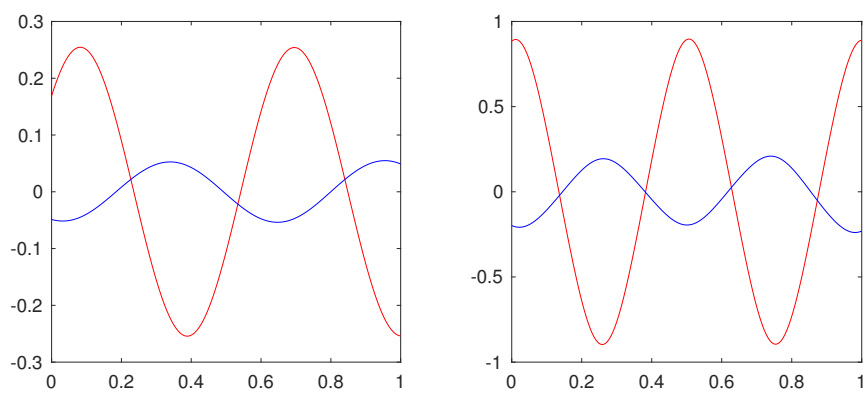
$$\begin{aligned} &u'' + 100(1 + |u|^2)u = 0, \quad 0 < x < 1, \\ &u'(0) + iu(0) = 2, \\ &u'(1) - iu(1) = 0. \end{aligned}$$

based on the initial guess $u(x) = 1$ *and* $u'(x) = 0$.

Solution: This problem appears in nonlinear optics. It is a nonlinear problem and it has multiple solutions. In the following program

```
x = linspace(0,1,51);
s0 = bvpinit(x,@myigs);
s = bvp4c(@myde,@mybc,s0);
plot(s.x, real(s.y(1,:)),'r', s.x, imag(s.y(1,:)),'b')

function y = myigs(x)
  y = [1; 0];
end

function f = myde(x,y)
  f = [y(2); -100*(1 + abs(y(1))^2)*y(1)];
end

function r = mybc(ya,yb)
 r = [ya(2)+1i*ya(1)-2; yb(2)-1i*yb(1)];
end
```

the interval $[0, 1]$ is discretized by 51 points, and we get the left figure below. If 51 is changed to



52, we get a different solution given in the right. This is a little bit strange, but for a general ODE BPV, existence and uniqueness are not guaranteed. In the above, $\mathbf{y}_a$ and $\mathbf{y}_b$ are

$$\mathbf{y}_a = \begin{bmatrix} u(0) \\ u'(0) \end{bmatrix}, \quad \mathbf{y}_b = \begin{bmatrix} u(1) \\ u'(1) \end{bmatrix}.$$

and the boundary conditions are

$$\mathbf{r} = \begin{bmatrix} u'(0) + iu(0) - 2 \\ u'(1) - iu(1) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

MA2507 Computing Mathematics Laboratory: Week 11

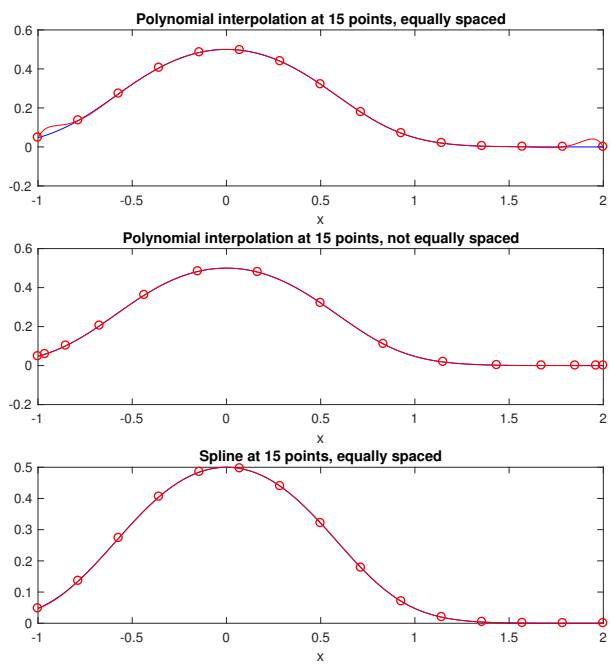
1. Polynomial interpolation: Given $n+1$ points (x_i, y_i) for $0 \le i \le n$, there is a unique polynomial $P(x)$ with a degree $\le n$, such that $P(x_i) = y_i$ for $0 \le i \le n$. This is the so-called polynomial interpolation. In MATLAB, this can be calculated by `polyfit`. The output of `polyfit` is the polynomial coefficients. We can use `polyval` to evaluate the polynomial. Let us try on

$$f(x) = \frac{1}{1+e^{3x^2}}$$

for $-1 \le x \le 2$.

```
f =@(x) 1./(1+exp(3*x.^2));
xx = linspace(-1,2,200);   % 200 points for drawing figures
n = 14
x = linspace(-1,2,n+1);
c = polyfit(x,f(x),n);
yy = polyval(c,xx);
subplot(3,1,1)
plot(xx,f(xx),'b',x,f(x),'ro', xx,yy,'r')
xlabel('x'), title('Polynomial interpolation at 15 points, equally spaced')
```

The above produces the top plot in the following figure. In the above, we take $n+1 = 15$ equally



spaced points from function $f(x)$, then use MATLAB commands `polyfit` and `polyval` to find the polynomial interpolant. The command `polyfit` requires the $n+1$ points given as the first and second inputs, but there is a thrid input for the polynomial degree. You can use a number less than n, it will try to "least squares". We see that the polynomial does not fit the original function well near the two end points. This is the so-called Runge phenomenon.

There is a way to improve this. The idea is to choose more points near the two end points. This can be done by first choosing $n+1$ points in $[-1,1]$ by

$$t_j = \cos\left(\frac{j\pi}{n}\right), \quad j = 0, 1, ..., n,$$

then transform t_j to x_j by $x_j = 0.5 - 1.5t_j$. These points t_j for $0 \le j \le n$ are the so-called Chebyshev points. Following the above MATLAB program, we add the following lines

```
t = cos((0:n)*pi/n);
x = 0.5-1.5*t;
c = polyfit(x,f(x),n;
yy = polyval(c,xx);
subplot(3,1,2)
plot(xx,f(xx),'b',x,f(x),'ro', xx,yy,'r')
xlabel('x'), title('Polynomial interpolation at 15 points, not equally spaced')
```

This leads to the second plot in the figure above.

2. Spline: Given $n+1$ points, we can use the command `spline` to produce a nice smooth curve. It produce a polynomial of degree 3 (or less) on each interval (x_i, x_{i+1}), and also the final function is smooth with continuous second order derivative. Now, let us try `spline` on 15 equally spaced points. We add the following lines to the above MATLAB program.

```
x = linspace(-1,2,n+1);
yy = spline(x,f(x),xx)
subplot(3,1,3)
plot(xx,f(xx),'b',x,f(x),'ro', xx,yy,'r')
xlabel('x'), title('Spline at 15 points, equally spaced')
```

This produces the third plot in the above figure. In the following, we consider n points (x_j, y_j) for $j = 1, 2, .., n$, assume $x_1 < x_2 < ... < x_n$, and denote the spline function as $S(x)$. We have $S(x_j) = y_j$ for $j = 1, 2, ..., n$, and S, S' and S'' are continuous. In addition, there are three choices

(a) $S''(x_1) = S''(x_n) = 0$.

(b) S' is given at x_1 and x_n.

(c) S''' is continuous at x_2 and x_{n-1}.

MATLAB `spline` gives (c) or (b) with extra input for S'. Here, we give some details about (a).

First, we define

$$h_j = x_{j+1} - x_j, \quad d_j = 6\frac{y_{j+1} - y_j}{x_{j+1} - x_j}, \quad j = 1, 2, ..., n-1.$$

Now, let $z_j = S''(x_j)$ for $j = 1, 2, ..., n$. For choice (a), we set $z_1 = z_n = 0$ and solve z_2, z_3, ..., z_{n-1} from

$$\begin{bmatrix} 2(h_1+h_2) & h_2 & & \\ h_2 & 2(h_2+h_3) & \ddots & \\ & \ddots & \ddots & h_{n-2} \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix} \begin{bmatrix} z_2 \\ z_3 \\ \vdots \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} d_2 - d_1 \\ d_3 - d_2 \\ \vdots \\ d_{n-1} - d_{n-2} \end{bmatrix}.$$

On (x_j, x_{j+1}), $S(x)$ is a cubic polynomial given as

$$S(x) = c_{1j}(x_{j+1} - x)^3 + c_{2j}(x - x_j)^3 + c_{3j}(x_{j+1} - x) + c_{4j}(x - x_j),$$

where

$$c_{1j} = \frac{z_j}{6h_j}, \quad c_{2j} = \frac{z_{j+1}}{6h_j}, \quad c_{3j} = \frac{y_j}{h_j} - \frac{z_j h_j}{6}, \quad c_{3j} = \frac{y_{j+1}}{h_j} - \frac{z_{j+1} h_j}{6}.$$

Here is the MATLAB program.

```
function yy = nspline(x,y,xx)
% Spline with 2nd order derivatoive = 0 at endpoints.
% Part 1: calculate the spline function S(x)
n = length(x);
for j=1:n-1
    h(j) = x(j+1)-x(j);
    d(j) = 6*(y(j+1)-y(j))/h(j);
end
A = zeros(n-2,n-2);
b = zeros(n-2,1);
A(1,1)=2*(h(1)+h(2));
b(1) = d(2)-d(1);
for j=2:n-2
    A(j,j)=2*(h(j)+h(j+1));
    A(j-1,j) = h(j);
    A(j,j-1) = h(j);
    b(j) = d(j+1)-d(j);
end
z = A\b;
z = [0; z; 0];
for j = 1:n-1
    c1(j)=z(j)/(6*h(j));
    c2(j)=z(j+1)/(6*h(j));
    c3(j)= y(j)/h(j)-z(j)*h(j)/6;
    c4(j)= y(j+1)/h(j)-z(j+1)*h(j)/6;
end
% Part 2: evaluate function S on vector xx
k = 1;
for j= 1: length(xx)
    if xx(j) > x(k+1)
        k = k+1;
    end
    t = x(k+1)-xx(j);
    s = xx(j)-x(k);
    yy(j) = c1(k)*t^3+c2(k)*s^3+c3(k)*t+c4(k)*s;
end
end
```

3. Graphical input. MATLAB allows you to input points by clicking the mouse using `ginput`, but

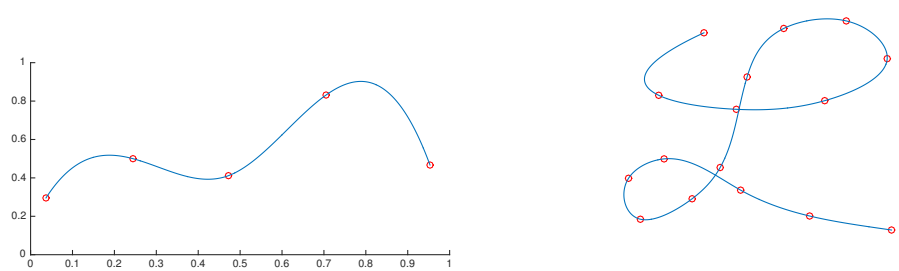
it does not show the points in the graphical window. I wrote the following program that draws a little circle for each point you input.

```
function [x,y]=myginp
% axis([0 1 0 1]); hold on
% hit Return to stop
j=0;
click=1;
while click > 0
    [xin,yin] = ginput(1);
    click = length(xin);
    if click == 1
        plot(xin,yin,'ro');
        j=j+1;
        x(j) = xin;
        y(j) = yin;
    end
end
```

You can use `myginp` with `spline` to draw a smooth function connecting a few points that you input by clicking the mouse.

```
axis([0 1 0 1])
hold on
[x,y]=myginp
xx = linspace(min(x),max(x),400);
yy = spline(x,y,xx);
plot(xx,yy)
hold off
```

The above program is used to draw the left plot in the following figure. We can also use `myginp`



to input a few points (x_i, y_i) for $1 \le i \le n$, and use `spline` to draw a smooth curve (not y as a function of x) connecting these points. We need to define t_i for $1 \le i \le n$, such that $t_1 = 0$, and $t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$, and draw spline functions of t. We have the following program:

```
axis([0 1 0 1])
hold on
```

```
[x,y]=myginp;
n=length(x);
t(1)=0;
for j=2:n
    t(j)=t(j-1)+sqrt((x(j)-x(j-1))^2+(y(j)-y(j-1))^2);
end
tt=linspace(0,t(n),500);
xx=spline(t,x,tt);
yy=spline(t,y,tt);
plot(xx,yy)
axis off
hold off
```

It is used to produce the right plot in the figure above.

It is also useful to use periodic boundary conditions. Namely, we want $S(x_1) = S(x_n)$, $S'(x_1) = S'(x_n)$ and $S''(x_1) = S''(x_n)$. That means, for the input, we must insist $y_n = y_1$. Now, we solve z_1, z_2, ..., z_{n-1} from

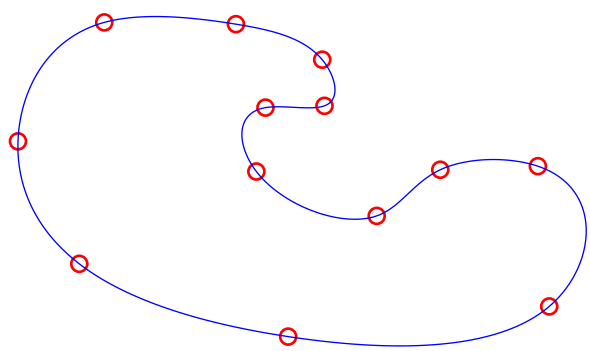
$$\begin{bmatrix} 2(h_{n-1}+h_1) & h_1 & 0 & \cdots & h_{n-1} \\ h_1 & 2(h_1+h_2) & h_2 & & 0 \\ 0 & h_2 & 2(h_2+h_3) & \ddots & \\ \vdots & & \ddots & \ddots & h_{n-2} \\ h_{n-1} & 0 & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 - d_{n-1} \\ d_2 - d_1 \\ d_3 - d_2 \\ \vdots \\ d_{n-1} - d_{n-2} \end{bmatrix}.$$

After that we set $z_n = z_1$. The formula for S is the same.

4. Exercise: *Write a program that inputs a few points from the screen and draw a* **close** *smooth curve based on* `myginp` *and a cubic spline program with periodic boundary condition.*
Solution: If we input a few points by `myginp`, we add one extra point which is exactly the same as the first point. This can be done by the following main script.

```
% draw closed smooth curves connecting a few points
clf
axis([0 1 0 1])
hold on
[x, y] = myginp;
n = length(x)+1;     % new line
x(n)=x(1);           % new line
y(n)=y(1);           % new line
t(1)=0;
for j=2:n
    t(j)=t(j-1)+sqrt((x(j)-x(j-1))^2+(y(j)-y(j-1))^2);
end
tt = linspace(0, t(n), 500);
xx = pspline(t,x,tt);    % changed to psline
yy = pspline(t,y,tt);    % changed to psline
plot(xx,yy,'b')
axis off
hold off
```

The program gives the following figure. The program `psline` is the program for spline with periodic boundary condition.



```
function yy = pspline(x,y,xx)
% spline with periodic boundary condition
% Part 1: calculate the spline function
n = length(x);
for j=1:n-1
    h(j) = x(j+1)-x(j);
    d(j) = 6*(y(j+1)-y(j))/h(j);
end
A = zeros(n-1,n-1);
b = zeros(n-1,1);
A(1,1)=2*(h(n-1)+h(1));
b(1) = d(1)-d(n-1);
for j=2:n-1
    A(j,j)=2*(h(j-1)+h(j));
    A(j,j-1) = h(j-1);
    A(j-1,j) = h(j-1);
    b(j) = d(j)-d(j-1);
end
A(1,n-1) = h(n-1);
A(n-1,1) = h(n-1);
z = A\b;
z = [z; z(1)];
for j = 1:n-1
```

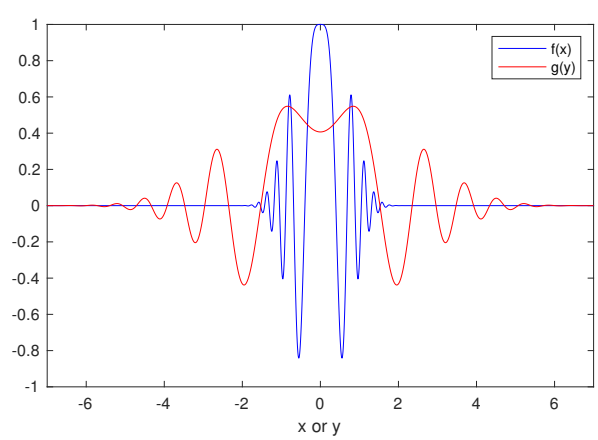
```
    c1(j)=z(j)/(6*h(j));
    c2(j)=z(j+1)/(6*h(j));
    c3(j)= y(j)/h(j)-z(j)*h(j)/6;
    c4(j)= y(j+1)/h(j)-z(j+1)*h(j)/6;
end
% Part 2: same as before
k = 1;
for j= 1: length(xx)
    if xx(j) > x(k+1)
        k = k+1;
    end
    t = x(k+1)-xx(j);
    s = xx(j)-x(k);
    yy(j) = c1(k)*t^3+c2(k)*s^3+c3(k)*t+c4(k)*s;
end
end
```

MA2507 Computing Mathematics Laboratory: Week 12 Test

1. Let $f(x) = \cos(10x^2)e^{-|x|^3}$ and $g(y)$ be the Fourier transform of $f(x)$. Find g approximately based on MATLAB internal function `fft`. You are required to discretize x and y by $N = 16384$ points as $x_j = j/\sqrt{N}$ for $-N/2 \le j < N/2$, $y_k = k/\sqrt{N}$ for $-N/2 \le k < N/2$, and plot both f and g (real part of g) in the same figure on interval $[-7, 7]$.
   Solution: This question follows the assignment of week 9, except that by choosing the same set of points for discretizing x and y, the program becomes a little bit simpler.

```
N = 16384;
m = N/2;
d = 1/sqrt(N);
x = d*(-m:m-1);     % y = x, same set of points
f = myfn(x);
ff = [f(m+1:N),f(1:m)];
gg = d*fft(ff);
g = [gg(m+1:N),gg(1:m)];
plot(x,f,'b',x,real(g),'r')
xlim([-7,7])
xlabel('x or y')
legend('f(x)','g(y)')

function z = myfn(x)
  z = exp(- abs(x).^3).* cos(10*x.^2);
end
```

   We obtain the following figure.

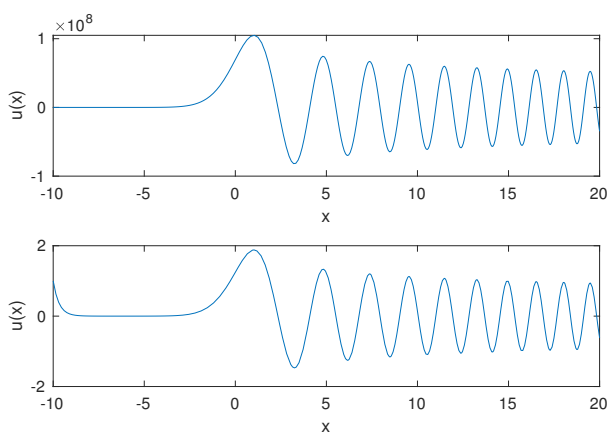


2. For differential equation $u'' + xu = 0$, where u is a function of x,
   (a) solve an IVP by ode45 from $x = -10$ to $x = 20$ with initial conditions $u(-10) = 1$, $u'(-10) = -3$, and plot $u(x)$ in a subplot;
   (b) solve a BVP by bvp4c on interval $(-10, 20)$ with boundary conditions $u(-10) = 1$ and $u'(-10) = u'(20)$ (use your own initial guess), plot $u(x)$ in another subplot.

Solution: As usual, we need to define a column vector $\mathbf{y} = [u, u']^T$, then write the equation as the first order system $d\mathbf{y}/dx = \mathbf{f}(x, \mathbf{y})$. For part (b), we have $\mathbf{y}_a$ for $\mathbf{y}$ at $x = -10$, and $\mathbf{y}_b$ for $\mathbf{y}$ at $x = 20$.

```
% Part 1: IVP from x=-10 to x=20, u(-10)=1, u'(-10)=-3
[tt,yy] = ode45(@myde,[-10,20],[1,-3]);
subplot(2,1,1)
plot(tt,yy(:,1))
xlabel('x'), ylabel('u(x)')
% Part 2: BVP
x = linspace(-10,20,300);
s0 = bvpinit(x,@myigs);
s = bvp4c(@myde,@mybc,s0);
subplot(2,1,2)
plot(s.x, s.y(1,:))
xlabel('x'), ylabel('u(x)')

function f = myde(x,y)
  f = [y(2); -x*y(1)];    % f(x,y) of the differential equation
end

function y = myigs(x)
  y = [1; 0];             % initial guess for y
end

function r = mybc(ya,yb)
  r = [ya(1)-1; ya(2)-yb(2)];  %boundary conditions
end
```

The above program gives the following figure. Although these two solutions look similar, they are

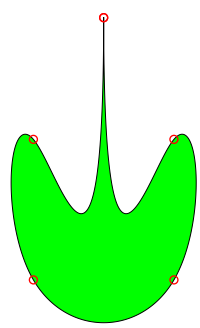


very different. The solution of the initial value problem is very large. The solution of the BVP is $O(1)$. Both solutions satisfy $u(-10) = 1$, but their values of $u'(-10)$ are different.

3. The figure below is obtained using MATLAB internal function `spline` based on the six points shown as the red circles and given in the table (the 1st and last points are identical). The boundary of the

j	1	2	3	4	5	6
x_j	0	0.5	0.5	-0.5	-0.5	0
y_j	$\sqrt{3}/2$	0	-1	-1	0	$\sqrt{3}/2$

domain is obtained by two spline functions, x as a function of t and y as a function of t, where the t-values corresponding to the six points are $[t_1, t_2, ..., t_6] = [0, 1, 2, 3, 4, 5]$. The domain is shown using the MATLAB `fill` command. The MATLAB `spline` command is used to specify the derivative of the spline functions at the first and last points. For example, to force $x'(0) = a$ and $x'(5) = b$, we input the vector $[a, x_1, ..., x_6, b]$ instead of $[x_1, x_2, .., x_6]$. Write a MATLAB program to draw a figure as close to the one below as possible with your choices for $x'(0)$, $x'(5)$, $y'(0)$ and $y'(5)$.



Solution: From the figure, we see that when t is increased starting from 0, $x(t)$ increases very slowly, and $y(t)$ decreases very rapidly. This suggests $x'(0) = 0$ and $y'(0)$ is negative (with large magnitude). When $t \to 5$, we see that $x(t)$ increases very slowly to 0, but $y(t)$ increases very rapidly, thus $x'(5) = 0$, and $y'(5)$ is positive and large. Due to the symmetry, we need to keep $y'(0) = -y'(5)$. The figure is obtained when $y'(0) = -7$ and $y'(5) = 7$.

```
x = [0, 0.5, 0.5, -0.5, -0.5, 0];
y = [sqrt(3)/2, 0, -1, -1, 0, sqrt(3)/2];
t = 0:5;
tt = 0:0.01:5;
xx = spline(t,[0,x,0],tt);
yy = spline(t,[-7,y,7],tt);
fill(xx,yy,'g')
hold on
plot(x,y,'ro')
axis equal off
hold off
```