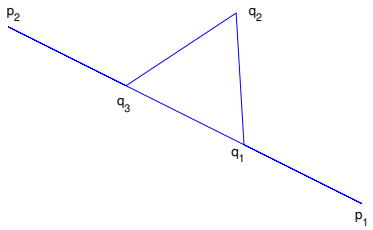
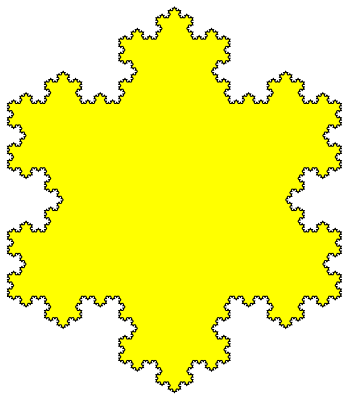
MA2507 Computing Mathematics Laboratory: Week 6

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.

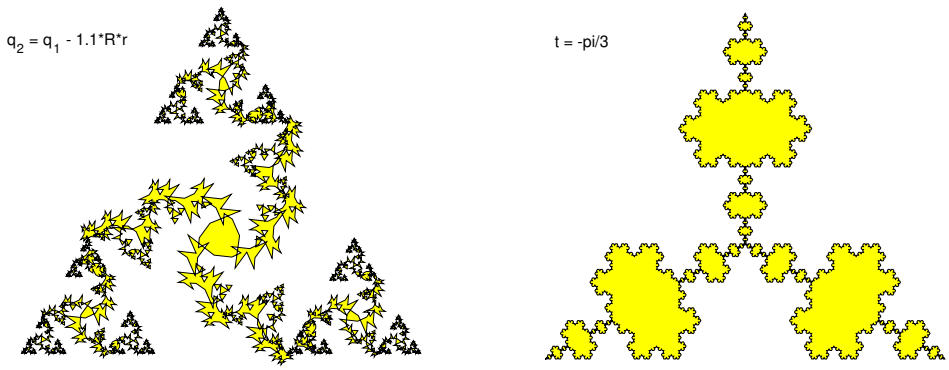
```
% main script for Koch's snowflake, requires function "addq"
% set the number of levels.
mmax = 5;
% points of a triangle as columns of p, repeating the first point
p = [0, 0; 1, 0; 1/2, sqrt(3)/2; 0 0]';
for m=1:mmax
    pnew = [];      % start an empty matrix pnew
    n = length(p);  % should give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

$$p(x) = a_1 x^n + a_2 x^{n-1} + ... + a_n x + a_{n+1}$$

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    16
  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.