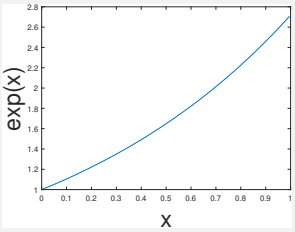
Answer to Question 1

There are four main functions created with a data set of randomly selected numbers. The Matlab code of each function is given beside its resulting figure.

The dataset with 100 random numbers in it, written in Matlab:

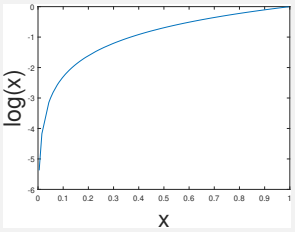
```
x = rand(1,100);
x = sort(x);
```

1 Exponential Function



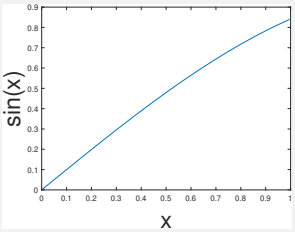

```
y= exp(x);
plot(x,y);
xlabel('x', 'FontSize', 30);
ylabel('exp(x)', 'FontSize', 30);
print -depsc exp
```

2 Logarithmic Function



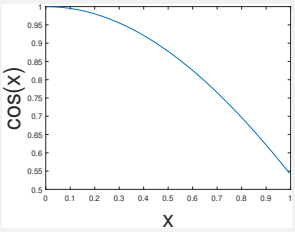

```
y= log(x);
plot(x,y);
xlabel('x', 'FontSize', 30);
ylabel('log(x)', 'FontSize', 30);
print -depsc log
```

3 Sine Function




```
y = sin(x);
plot(x,y);
xlabel('x', 'FontSize', 30);
ylabel('sin(x)', 'FontSize', 30);
print -depsc sin
```

4 Cosine Function




```
y= cos(x);
plot(x,y);
xlabel('x', 'FontSize', 30);
ylabel('cos(x)', 'FontSize', 30);
print -depsc cos
```

The command used to create .eps files to be displayed in LateX is:

```
print -depsc
```

Answer to Question 2

The mathematical expression proposed in the question, $P \equiv I - D^{-1/2}AD^{-1/2}$ indicates a Laplacian matrix since $A \equiv D^{-1/2}AD^{-1/2}$ is a normalized adjacency matrix for a given graph G where D is degree matrix, I is an identity matrix and A is an adjacency matrix. Therefore, the subtraction of the normalized adjacency matrix from a corresponding identity matrix is a normalized Laplacian matrix $L \equiv I - A$. Finally, the calculations in the following part will be done in Matlab with respect to this definition.

First, it is required to load raw data, formatted as (node1, node2, weight), from the input.txt into a Matlab table.

```
clear, clc;
T = readtable("input.txt");
```

The required component to create matrices is now obtained. Thus, the matrices to make a graph are:

```
x = table2array(T(:,1));
y = table2array(T(:,2));
z = table2array(T(:,3));
```

Then, it is compulsory to create a undirected graph to input the Laplacian matrix.

```
G = graph(x,y,z);
```

However, it is important to denote that Laplacian matrix will not accept a graph including looping vertices, a multigraph. Therefore, it is required to check whether the graph is a multigraph. If so, the looping vertices should be removed accordingly.

```
if ismultigraph(G)
  G = simplify(G);
end
```

Part A

Luckily, Matlab has a built-in function to allow creating Laplacian matrices. Therefore, the graph generated in the previous part will be inputted to the Laplacian function

```
L = laplacian(G);
```

Now, the first two eigenvalues and eigenvectors of the matrix L will be calculated. Before moving on, it is crucial to emphasize that the input file is too large to make use of *eig*() function. Therefore, *eigs*() function is utilized to calculate eigenvalues and eigenvectors where required. Also note that, *tictoc* function is utilized to report the running time.

```
tic
[V,D] = eigs(L,2);
toc
```

The running time is reported as:

```
Elapsed time is 0.117207 seconds.
```

The first two largest eigenvalues are calculated by a diagonal matrix since the eigenvalues are on the main diagonal of the matrix D:

```
d = diag(D);
e1 = d(1,:);
e2 = d(2,:);
```

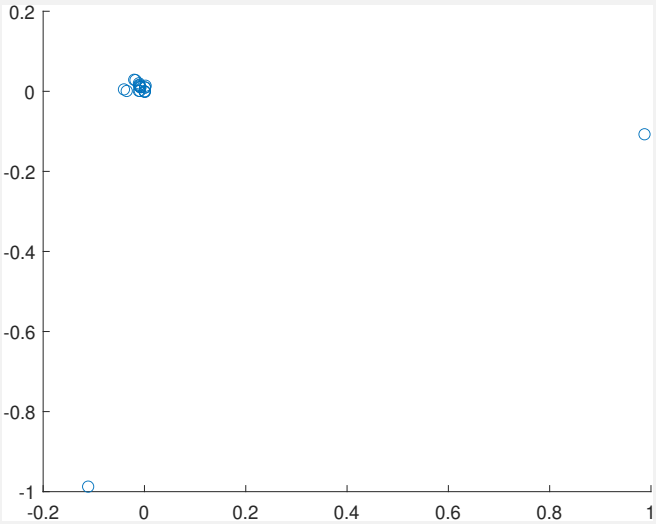
The first two largest eigenvalues are:

```
e1 = 116.2437
e2 = 110.0638
```

Part B

The plot of the first two largest eigenvectors is calculated with *scatter*() function and saved as .eps file:

```
s = scatter(V(:,1),V(:,2));
print -depsc s
```



Part C

The statement $P + \sigma I$ is calculated as the following:

```
sigma = 0.6;
sz = size(L);
```

```
szX = sz(:, 1);
I = eye(szX);
M1 = L + (sigma * I);
v1 = eig(M1);
```

The resultant vector has 10708 the eigenvalues of the statement. Therefore, the largest six values are give as the following:

```
77.5647
80.9354
90.7290
93.6526
110.6638
116.8437
```

Part D

P^{20} requires a massive amount of computation because the matrix P's itself has exactly 10708 rows and 10708 columns. It is likely for the user to run out of memory. However, there is a handy mathematical theorem, which is Eigen Decomposition theorem.

This theorem shows that $P^n = CD^nC^{-1}$ equality can be used to calculate the power of a matrix where D is a diagonal matrix of eigenvalues and C is the matrix composed of eigenvectors. Here is Matlab implementation:

```
[V1, D1] = eigs(L,10708);
D1 = D1^20;
result = V1 .* D1 .* V1^-1;
tic
[V2,D2] = eigs(result);
toc
```

The matrix composed of eigenvectors is $D2$ and the report of time:

```
Elapsed time is 1.098644 seconds.
```

Answer to Question 3

The new dataset for the calculation of this question:

```
s = [11 32 45 3 6 7 43 34 765 21 58 36 710 482 56];
t = [78 31 67 28 83 26 17 85 25 90 68 59 29 19 69];
```

The graph to create the new Laplacian matrix:

```
G = graph(s, t);
```

Subsequently composing the matrix $\hat{P} \equiv I - 0.85 * D^{-1/2}AD^{-1/2}$ in Matlab:

```
A = adjacency(G);
```

```
D = degree(G);
I = eye(765);
L = I - 0.85 .* D.^-1/2 .* A .* D.^-1/2;
```

Part A

The Matlab code to solve $\hat{P}x \equiv b$

```
b = zeros(765, 1);
for i=1:10
b(i) = 1/10;
end
tic
X = L^-1 * b;
toc
```

The report of time:

```
Elapsed time is 0.021504 seconds.
```

Part B

Since the problem is simply a system of linear equations, it can also be solved more efficiently by using the lower-upper decomposition as it gives a resultant matrix in the form of Gaussian Elimination enabling the matrix to be solved by row reduction and matrix-vector product. Moreover, the matrix $\hat{P}$ is a Laplacian matrix meaning that its eigenvalues are nonnegative. This property allows us to find the determinant of the matrix and invert the matrix.

The simple algorithm to solve this problem efficiently would be:

```
(1) Find the inverse matrix
(2) Find the determinant
(3) Apply Gauss Elimination, row reduction
(4) Substitute the output matrix into the lower-upper decomposition,
```

Answer to Question 4

Part A

The computation of 1000th number of Fibonacci sequence by using a recursive function, is too long to find it because its time complexity is $O(2^n)$. However, the 1000th will be found as the following, which uses a recursive function:

```
tic
dummy = fib(1000);
toc

function x = fib(n)
    if (n==1) || (n==2)
        x = 1;
    else
        x = fib(n-1) + fib(n-2);
    end
end
```

Part B

For Fibonacci sequence $F_n = F_{n-1} + F_{n-2}$, it is required to get a linear system for the first consecutive equations:

$F_n = F_{n-1} + F_{n-2}$

$F_n = F_{n-1} + 0F_{n-2}$

Thus, a matrix notation can be obtained to implement the matrix-vector product.

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

The traversal through the given N will be resulted as the multiplication of starting matrix with the consecutive matrix.

The initial matrix:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Therefore, the equation can be simplified into

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Finally, its implementation can be programmed in Matlab as the following:

```
i = 1000;
tic
result = matrixVectorProduct(i);
toc

function x = matrixVectorProduct(a)
u0 = [0 1; 1 1];
x = u0^a * [0; 1];
x = x(1,:);
end
```

The resultant number is $4.3467 * 10^{208}$. The report of time:

```
Elapsed time is 0.000670 seconds.
```

Part C

Calculation of the problem part (b) is also possible by converting the linear system into a dynamic linear model. The general equation for dynamic linear models can be shown as $Au_0 = u_1$. That is, an initial vector can be represented as a initial number of Fibonacci sequence.

Following this, we need to input an index k to find u_k, kth number in Fibonacci sequence. Therefore, we need to construct a equivalence by using eigenvalues and eigenvectors. S is the matrix of eigenvectors containing eigenvectors in columns, provided that the initial matrix A containing F_1 and F_2 is

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$\Lambda \in \mathbb{R}^{2x2}$ is a diagonal matrix containing eigenvalues on the main diagonal. By the definition of eigenvectors and eigenvalues, we can substitute A into $AS = S\Lambda$. Then, we add the initial vector u_0 giving $u_1 = S\Lambda S^{-1}u_1$. The more generalized for calculating kth number in Fibonacci sequence becomes $u_k = S\Lambda^k S^{-1}u_0$. Here is the implementation in Matlab:

F_1 and F_2

```
clear, clc;
A = [1 1; 1 0];
```

V = eigenvectors of A, D = eigenvalues in the diagonal matrix of A

```
[V,D] = eig(A);
d = diag(D);
```

Eigenvalues:

```
lambda_1 = d(2,:);
lambda_2 = d(1,:);
```

Eigenvectors:

```
v_1 = [d(2,:);1];
v_2 = [d(1,:);1];
```

S is composed of eigenvectors

```
S = [v_1 v_2];
```

Eigenvalues on the main diagonal

```
big_lambda = [v_1(1,:) 0; 0 v_2(1,:)];
```

Initial state vector

```
u0 = [1;1];
```

By that, we will be able to find consecutive Fibonacci numbers

```
c = S^-1*u0;
result = S * big_lambda^998 * c;
```

The time report:

```
Elapsed time is 0.001491 seconds.
```

Result

Consequently, using recursive function the worst way of calculating Fibonacci numbers due to its time complexity. Moreover, the calculation done by using matrix-vector is far better than eigenvectors. The comparison of running times by the parts (a), (b), (c):

$$(b) < (c) < (a)$$