Computer Assignment 3

1. Write a function with header `[B] = myMakeLinInd(A)`, where `A` and `B` are matrices. Let the `rank(A) = n`, then `B` should be a matrix containing the first `n` columns of `A` that are all linearly independent.

   Solution:

```
function B = myMakeLinInd(A)
B = [A(:,1)];
n = 2;
for k=2:size(A,2)
    if rank([B A(:,k)]) == n
        B = [B A(:,k)];
        n = n+1;
    end
end
```

2. Write a function `alpha = myPolyfit(n,p,x)` that finds the coefficients of a polynomial $p(x)$ of degree `n` that fits the data in `p` and `x`. Your function should solve this problem as a linear system of equations and show an error if there is either no solution or an infinite number of solutions.

   Solution:

```
function alpha = myPolyfit(x,p,n)
A = x.^(n:-1:0);
if rank(A) == n+1
    alpha = (A\p)';
else
    error('there is no unique solution')
end
```

3. Repeat the question above but using the least square method instead. Note that now there is always a unique solution, independently of the length `p` and `x`. You can check your results with the MATLAB built-in function `polyfit`.

   Solution:

```
function alpha = myPolyfit_bis(x,p,n)
A = x.^(n:-1:0);
alpha = (pinv(A)*p)';
end
```

4. Using the bisection method, write a function `r = myRoots(alpha)` that outputs the (real) roots of a polynomial whose coefficients are the elements of the (real-valued) array `alpha`. You can check your method with the MATLAB built-in function `roots`.

   Hint: Find the monotonicity intervals by finding the roots of the derivative of the polynomial.

Solution:

```
function roots = myRoots(alpha)
if size(alpha,2) ~= 1
    error('alpha must be a column vector')
end
n = length(alpha)-1;
if n == 1
    roots = -alpha(2)/alpha(1);
else
    d_alpha = pol_derivative(n)*alpha;
    turning_points = myRoots(d_alpha);
    f = @(x) coeff2pol(alpha,x);
    Intervals = intervals(turning_points, f);
    roots = [];
    if Intervals
        for i=1:size(Intervals,1)
            roots(i) = mybisection(f, Intervals(i,1),
               Intervals(i,2),1e-10);
        end
    end
end
end

function D = pol_derivative(n)
D = [diag(n:-1:1) zeros(n,1)];
end

function I = intervals(turning_points, f)
I = [];
k = 1;
if turning_points
    n = length(turning_points);
    if f(-100)*f(turning_points(1))<0
        I(1,:) = [-100, turning_points(1)];
        k = 2;
    end
    for i = 1:n-1
        if f(turning_points(i))*f(turning_points(i+1))<0
            I(k,:) = [turning_points(i) turning_points(i+1)
               ];
            k = k+1;
        end
    end
    if f(turning_points(n))*f(100)<0
```

```
            I(end+1,:) = [turning_points(n), 100];
        end
    elseif f(-100)*f(100)<0
        I = [-100, 100];
    end
    end


    function root = mybisection(f,a,b,tol)
        m = (a+b)/2;
        while abs(f(m))> tol
            if f(a)*f(m)<0
                b = m;
            else
                a = m;
            end
            m = (a+b)/2;
        end
        root = m;
    end
```

5. The eigenvalues λ of a (square) matrix A correspond to the roots of the function $p(\lambda) = \det(A - \lambda I)$, where I denotes the identity matrix. Explain why if A is of size n, then $p(\lambda)$ is a polynomial of degree n. Next, using question 3 and question 4, code a function that finds the real eigenvalues A and their corresponding eigenvectors.

   Solution:

```
function [V, e] = myRealEig(A)
n = size(A,1);
cpol =@(lda) det(A-lda*eye(n));
lda = randn(n+1,1);
for i=1:(n+1)
    p(i,1) = cpol(lda(i));
end
alpha = myPolyfit(lda,p,n);
e = myRoots(alpha');
for i=1:length(e)
    V(:,i) = (A-e(i)*eye(n))\randn(n,1);
    V(:,i) = V(:,i)/norm(V(:,i));
end
```

6. The singular value decomposition of a matrix A of size `n`×`m`, is a factorisation of A in the form $A = USV^t$, where both U and V are (full rank) (orthonormal) square matrices and S is a non-necessarily-square diagonal matrix whit non-negative elements. The non-zero elements of the diagonal of S, called singular values of A, correspond to the

square root of the non-zero eigenvalues of AA^t (or A^tA). The matrix V is formed by the eigenvectors of A^tA and the matrix U is formed by the eigenvectors of AA^t. Using `eig`, implement a function `[U,S,V] = mySVD(A)` which computes the SVD decomposition of a matrix `A`.

Solution:

```
function [U,S,V] = mySVD(A)
n = rank(A'*A);
[V , e] = eig(A'*A);
S_d = diag(real(e));
[~, idx] = sort(S_d, 'descend');
S_d = real(sqrt(S_d(idx))); V = V(:,idx);
S = zeros(size(A)); S(1:n,1:n) = diag(S_d(1:n));
U = [A*V(:,1:n)*diag((S_d(1:n)).^(-1)) null(A*A')];
end
```

7. Note that the rank of a matrix A is given by the number of non-zero singular values of A (why?). Write a function that take as input a matrix A, and outputs a new matrix A_k, which is k-rank version of A, computed by keeping the k-largest singular values of A. Use this function to show a low rank version of the image of question 10 of Assignment 1.

   Solution:

```
function A_ = reducedRank(A,k)
[U,S,V] = svd(A);
S_diag = diag(S);
m = length(S_diag)-k;
S_diag(m:end) = 0;
S_ = diag(S_diag);
A_ = U*S_*V';
end
```

8. Find regression curves for the average runtime data $T_1(n)$ and $T_2(n)$, corresponding to the runtime of the code of question 10 of Assignment 2, and its efficient version, respectively, where n is the size of the input matrix `M`. Plot your regression curves along with the runtime data. Can you quantify now how faster is the efficient implementation with respect to the inefficient one?

   Solution:

```
N = 100; K = 100;
T1 = zeros(N,K);
T2 = zeros(N,K);
for k=1:K
    for n = 1:N
        M = rand(n);
```

```
            tic,
            myFunction(M);
            T1(n,k) = toc;
            tic,
            myEfficientFunction(M);
            T2(n,k) = toc;
        end
    end
    T1 = 1/K*sum(T1,2);
    T2 = 1/K*sum(T2,2);

    % least square regressions
    A = (1:N)'.^(2:-1:0); % complexity is O(n^2) for both
       functions
    alpha1 = pinv(A)*T1;
    alpha2 = pinv(A)*T2;
    plot(1:N, T1, '*'), hold on, plot(1:N, A*alpha1), plot
       (1:N, T2, '.'), plot(1:N, A*alpha2,'--')
    legend('inefficient  function', 'inefficient function
       regression', 'efficient function', 'efficient
       function regression')
    xlabel('size of the input matrix'); ylabel('time [s] in
        logarithmic scale');
    set(gca, 'YScale', 'log')
    title('average runtime')
```

To quantify how much faster is the efficient method we can look at the first coefficient of the regression curves. The ratio between the two is

```
>> alpha1(1)/alpha2(1)

ans =

   12.5880
```

So the efficient algorithm is around 12 times faster.

9. Implement a MATLAB function that take as input two arrays `f` and `x`, representing the values of a real valued function $f(x)$; the array `x` should be evenly spaced. Your function should:

   (a) create a new array `f_s` which replace each element of `f` with the average of its `k` nearest neighbours (`k` should also be an input of your function) to the left and to the right. The function `f_s` is a way of regularising a noisy or irregular function.

   (b) returns the numerical derivative of f_s using a centred first order finite difference scheme that you should also implement.

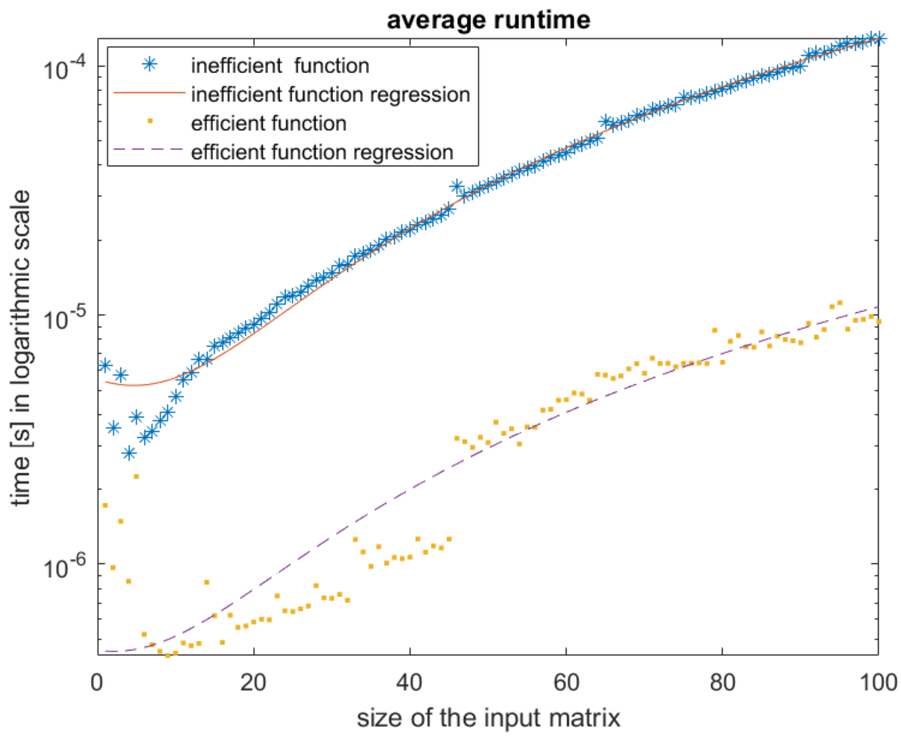

Figure 1: Q3

Test your code with `x = linspace(0,2*pi,1000)` and `f = sin(x) + 0.1*randn(size(x))`, for different values of k.

Solution:

```
function [f_s, df] = denoising(x, f, k)
    f_s = zeros(size(f));
    for i = 1:length(f)
    idx = max(1, i-k):min(length(f), i+k);
    f_s(i) = mean(f(idx));
    end
    df = zeros(size(f_s));
    for i = 2:length(f_s)-1
    df(i) = (f_s(i+1) - f_s(i-1)) / (x(i+1) - x(i-1));
    end
    df(1) = (f_s(2) - f_s(1)) / (x(2) - x(1));
    df(end) = (f_s(end) - f_s(end-1)) / (x(end) - x(end-1))
       ;
    end
```

10. Write a function `I = myTrapez(f, a, b, n)`, which computes the approximation of $\int_a^b f(x)\,dx$ by a trapezoidal rule: $\int_a^b f(x)\,dx \approx h\left[\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(x_k)\right]$, where $x_k = a + hk$, and $h = \frac{b-a}{n}$.Your function should not use any built-in Matlab functions. Test

your function by computing $\int_0^1 \sqrt{1-x^2}\,dx$, with $n = 10, 20$, and 40. Given that the exact value of the integral is $\pi/4$, how does the error of the approximateresult scale with n?

Solution:

```
function I = myTrapez(f, a, b, n)
h = (b - a) / n;
I = 0;
for k = 1:n-1
xk = a + k*h;
I = I + f(xk);
end
I = h* (f(a)/2 + I + f(b)/2);
end
```