Homework 6 (Solution)

Math 3607

Summer 2021

Tae Eun Kim

Table of Contents


```
clear, close all, format short g
```

Solutions to *by-hand* problems are found in a separate pdf file named `hw06soln-by-hand.pdf`.

Problem 1 (Polynomial evaluation of matrices)

See `hw06soln-by-hand.pdf` for solution. Below, we test the code against MATLAB's built-in functions `polyval` (scalar and vector inputs) and `polyvalm` (square matrix inputs). One thing to remember (from Modules 2 and 3) is that MATLAB arranges polynomial coefficients in descending order, which is the opposite of our convention. So coefficients need to be flipped before getting fed to either one of these functions. To learn more about them, type

```
help polyval
```

```
  polyval Evaluate polynomial.
     Y = polyval(P,X) returns the value of a polynomial P evaluated at X. P
     is a vector of length N+1 whose elements are the coefficients of the
     polynomial in descending powers:

         Y = P(1)*X^N + P(2)*X^(N-1) + ... + P(N)*X + P(N+1)

     The polynomial P is evaluated at all points in X. See POLYVALM for
     evaluation of a polynomial P in a matrix sense.

     [Y,DELTA] = polyval(P,X,S) uses the optional output structure S created
     by POLYFIT to generate prediction error estimates DELTA. DELTA is an
     estimate of the standard deviation of the error in predicting a future
     observation at X by P(X).

     If the coefficients in P are least squares estimates computed by
     POLYFIT, and the errors in the data input to POLYFIT are independent,
     normal, with constant variance, then Y +/- DELTA will contain at least
     50% of future observations at X.
```

```
    Y = polyval(P,X,[],MU) or [Y,DELTA] = polyval(P,X,S,MU) uses XHAT =
    (X-MU(1))/MU(2) in place of X. The centering and scaling parameters MU
    are optional output computed by POLYFIT.

    Example:
       Evaluate the polynomial p(x) = 3x^2+2x+1 at x = 5,7, and 9:

       p = [3 2 1];
       x = [5 7 9];
       y = polyval(p,x)

    Class support for inputs P,X,S,MU:
       float: double, single

    See also polyfit, polyvalm.

    Documentation for polyval
    Other functions named polyval
```

or

```
help polyvalm
```

```
 polyvalm Evaluate polynomial with matrix argument.
    Y = polyvalm(P,X), when P is a vector of length N+1 whose elements
    are the coefficients of a polynomial, is the value of the
    polynomial evaluated with matrix argument X.  X must be a
    square matrix.

        Y = P(1)*X^N + P(2)*X^(N-1) + ... + P(N)*X + P(N+1)*I

    Class support for inputs p, X:
       float: double, single

    See also polyval, polyfit.

    Documentation for polyvalm
```

We will use $p(x) = (1+2x)^5 = 1 + 10x + 40x^2 + 80x^3 + 80x^4 + 32x^5$ as our testing polynomial with varying inputs. Let's encode the polynomial using its coefficients:

```
c = [1 10 40 80 80 32];
```

To simplify our testing procedure, let's define a short in-line functions:

```
diff_polyval = @(x) polyval(flip(c),x) - mypolyval(c,x);
diff_polyvalm = @(x) polyvalm(flip(c),x) - mypolyval(c,x);
```

Testing 1: Scalar input (against `polyval`)

```
x = 3/2;
diff_polyval(x)
```

```
ans =
     0
```

Testing 2: Vector input (against `polyval`)

```
x = rand(1, 10);
norm(diff_polyval(x))     % row vector
```

```
ans =
     0
```

```
norm(diff_polyval(x'))    % column vector
```

```
ans =
     0
```

Testing 3: Matrix input (against `polyvalm`)

For this testing, we will be careful to use a matrix with real eigenvalues.

```
x = tril(randn(8));        % lower triangular matrix; diagonal entries are eigenvalues
norm(diff_polyvalm(x))
```

```
ans =
   1.5872e-12
```

Note: As the dimenions of `x` increase, we notice that the error gets larger.

Testing 4: Incompatible inputs

If `x` is neither a scalar, a vector, nor a square matrix, the program should halt with an error message because of the line

```
error('Input x must be a scalar, a vector, or a square matrix');
```

Let's see if it indeed works. Let's input a rectangular matrix.

```
x = rand(20, 3);
mypolyval(c,x)
```

```
Error using hw06soln>mypolyval (line 60)
Input x must be a scalar, a vector, or a square matrix
```

Good! It prints out the error message we wrote.

Problem 2 (Singular values by hand)

See `hw06soln-by-hand.pdf`.

Problem 3 (SVD and the 2-norm)

See `hw06soln-by-hand.pdf`.

Problem 4 (Vandermonde matrix, SVD, and rank)

We have been playing with Vandermonde matrices several times in this semester.

```
x = linspace(0,1,1000)';
A = x.^(0:999);
```

This is certainly an overkill for this problem, but a 1000-by-1000 matrix is not big of a deal on modern machines.

(a) Let's print out singular values of A_1, A_2, and A_3. Since only singular values are needed, we use placeholders ~ to prevent `svd` from outputing `U` and `V`. In addition, we use the economy SVD to trim extra fat off from `S` matrix.

First, A_1 is a single column vector and it has only one singular value.

```
[~,S,~] = svd(A(:,1), 0);
diag(S)
```

```
ans =
       31.623
```

The matrix A_2 consists of the first two columns of A and so it has two singular values.

```
[~,S,~] = svd(A(:,1:2), 0);
diag(S)
```

```
ans = 2×1
       35.604
       8.1161
```

The matrix A_3 has three singular values.

```
[~,S,~] = svd(A(:,1:3), 0);
diag(S)
```

```
ans = 3×1
       37.531
        11.07
       1.6426
```

Let's use a loop to put the code into a nice script. I will use `fprintf` to format the outputs nicely for those who are familiar with it.

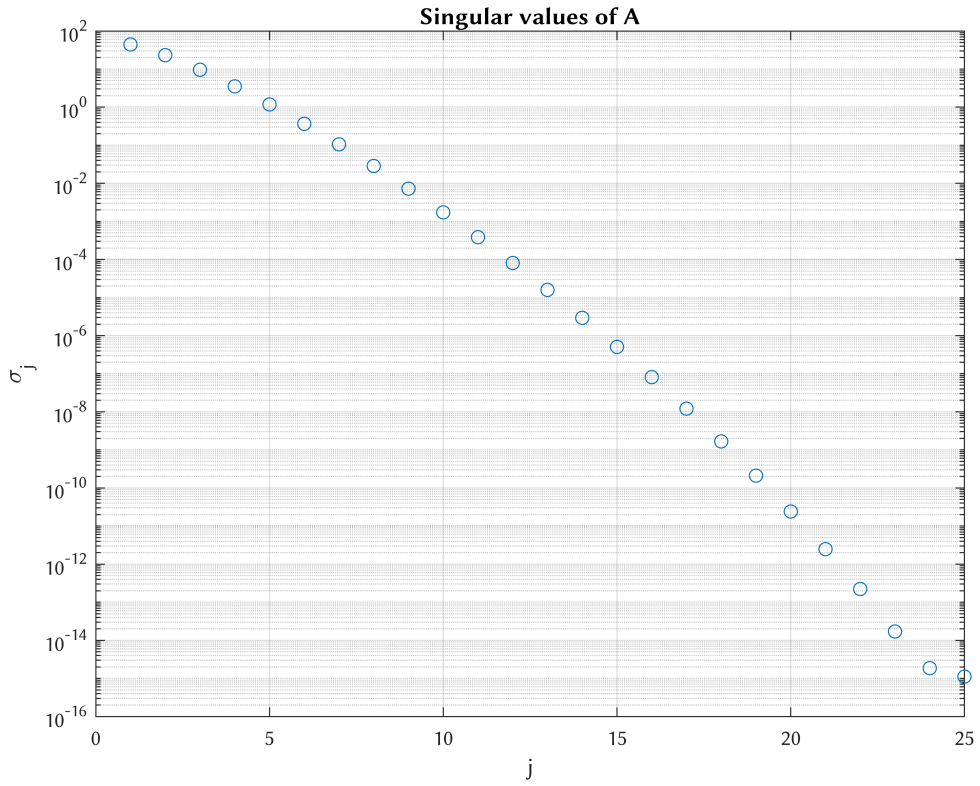
```
fmt = {'%5d', '%10.4f'};
for n = 1:3
    if n == 1
        fprintf('%5s %25s\n', 'n', 'singular values');
    end
    [~,S,~] = svd(A(:,1:n), 0);
    fprintf([fmt{[1 2*ones(1,n)]}, '\n'], n, diag(S)')
```

```
end
```

```
n           singular values
1   31.6228
2   35.6038    8.1161
3   37.5306   11.0704    1.6426
```

(b) For A_{25}, we need to plot the singular values on a semi-log plot. Since we are plotting only a couple dozen discrete values, it makes sense to plot them in dots or circles, rather than connecting them using a line.

```
n = 25;
[~,S,~] = svd(A(:,1:n),0);
clf
semilogy(1:n, diag(S), 'o')
xlabel('j'), ylabel('\sigma_j')
title('Singular values of A')
grid on
```



As we know from the theory of SVD, the singular values are arranged in descending order!

(c) In addition, we note from the previous plot that all singular values are non-negative. So, theoretically, the rank of A_{25} is 25, the number of nonzero singular values. However, MATLAB gives me a different answer:

```
A25 = A(:,1:25);
rank(A25)
```

```
ans =
    20
```

What is going on? As the problem instructs, let's dig into the help document for `rank`:

```
help rank
```

```
 rank   Matrix rank.
    rank(A) provides an estimate of the number of linearly
    independent rows or columns of a matrix A.

    rank(A,TOL) is the number of singular values of A
    that are larger than TOL. By default, TOL = max(size(A)) * eps(norm(A)).

    Class support for input A:
       float: double, single

    Documentation for rank
    Other functions named rank
```

The second paragraph says that the tolerance (TOL) is set to be the larger of the dimensions of the given matrix (in our case, 25) multiplied by a variant of the machine epsilon. In our case, this default tolerance is about 7×10^{-12}.

```
tol = max(size(A25)) * eps(norm(A25))
```

```
tol =
   7.1054e-12
```

This means that any singular values below this level is considered as zero taking into account the artifact of floating-point arithmetic. Let's confirm that this is indeed the case using logicals.

```
length( find(diag(S)>tol) )
```

```
ans =
    20
```

Bingo!

Unless you absolutely know what you are doing, I would not recommend modifying the tolerance level. But if you wish, we can do something like:

```
rank(A25, 1e-16)
```

```
ans =
    25
```

Functions Used

mypolyval

```
function y = mypolyval(c, x)
```

```
%MYPOLYVAL evaluates a polynomial at points  x  given its coeffs.
% Input:
%   c   coefficient vector (c_1, c_2, ..., c_n)^T
%   x   points of evaluation
%       - if x is a scalar or a vector, use Horner's method
%       - if x is a square matrix, use the result from (a)
%       - otherwise, produce an error message.
    [k,m] = size(x);
    if k==1 || m==1
        y = horner(c, x);
    elseif k==m
        [V,D] = eig(x);
        y = V*diag(horner(c, diag(D)))/V;  % implementing part (a)
    else
        error('Input x must be a scalar, a vector, or a square matrix');
    end
end
```

horner

```
function y = horner(c, x)
%HORNER Horner's method to evaluate polynomial
% Input:
%   c   coefficient vector (c_1, c_2, ..., c_n)^T
%   x   points of evaluation (either a scalar or a vector)
    n = length(c);
    y = c(n);
    for j = n-1:-1:1
        y = y.*x + c(j);  % Note the use of .*
    end
end
```