Matrix Algebra

MatLab offers a wide range of matrix algebra functions. However, while MatLab supports n dimensional arrays, matrix algebra is limited to 1D vectors and 2D matrices.

Solving Systems of Linear Equations

- A common problem in Matrix Algebra is to solve linear systems of equations. Suppose we have 4 equations in 4 unknowns, x_1, x_2, x_3 and x_4:

$$\begin{aligned} 4x_1 + 6x_2 + 2x_3 - 14x_4 &= 23 \\ -16x_1 + 20x_2 - 3x_3 + 5x_4 &= 0 \\ 25x_1 + 30x_2 + 40x_3 + 50x_4 &= 1 \\ 9x_1 - 13x_2 + 13x_3 - 28x_4 &= 9 \end{aligned}$$

- We can write these equation in matrix/vector form as:

$$\begin{bmatrix} 4 & 6 & 2 & -14 \\ -16 & 20 & -3 & 5 \\ 25 & 30 & 40 & 50 \\ 9 & -13 & 13 & -28 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 23 \\ 0 \\ 1 \\ 9 \end{pmatrix}.$$

or $A\mathbf{x} = B$, where 4×4 matrix A is:

$$A = \begin{bmatrix} 4 & 6 & 2 & -14 \\ -16 & 20 & -3 & 5 \\ 25 & 30 & 40 & 50 \\ 9 & -13 & 13 & -28 \end{bmatrix}$$

and 4×1 vectors $\mathbf{x}$ and B are:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 23 \\ 0 \\ 1 \\ 9 \end{pmatrix}.$$

- The solution to this set of linear system of equations are the values of x_1, x_2, x_3 and x_4 that simultaneously satisfy all four equations.
- In MatLab we setup this system of equations as:

```
A=[4 6 2 -14; -16 20 -3 5; 25 30 40 50; 9 -13 13 -28]
     4     6     2   -14
   -16    20    -3     5
    25    30    40    50
     9   -13    13   -28
B=[23; 0; 1; 9]
    23
     0
     1
     9
cond(A)
   11.3666
```

- Is this system of equations solvable? One check we can do is compute the condition number of matrix A (the smallest condition number is 1 and a low number indicates numerical stability). We see above that `cond(A)`

is 11.3666, which is quite low so the system of equations is solvable.

- There are 2 approaches to solving this system of equations, the second more numerically stable than the other.
- The first (less desirable solution method) involves using the inverse matrix of A, A^{-1}, where $AA^{-1} = A^{-1}A = I$ and I is the 4×4 identity matrix:

```
1     0     0     0
0     1     0     0
0     0     1     0
0     0     0     1
```

- Then the solution to $A\mathbf{x} = B$ is simply $\mathbf{x} = A^{-1}B$.

```
format long
x=inv(A)*B
   1.457544673236384
   1.246437482672581
  -0.810734178491318
  -0.808047483428687
```

- The second (more desirable solution method) is to use the left division operator \ (this solution method is more accurate, requires fewer floating point operations and is the simpler). So solve for $\mathbf{x}$ as `A\B`. The use of \ for this operation is an example of MatLab overloading an existing function. Which operation to be used is determined by the operands. In MatLab we compute:

```
format long
x=A\B
   1.457544673236385
   1.246437482672581
  -0.810734178491318
  -0.808047483428687
```

We see that both solution methods basically give the same solution (the last digit of x_1 only varies in the last digit by 1 with x_2, x_3 and x_4 being identical using `format long` to print the numbers). But this happens

because the system of equations is so well-conditioned (as shown by the very small condition number).

- Note that \ operation most likely uses an LU factorization approach to solve this system of equations but it does analyzes the structure of the matrix to determine which internal algorithms exactly to use. For example, if the matrix is upper or lower triangular, then MatLab does not re-factor the matrix, but just computes forward or backward substitution to find the solution.

- We can take the transpose of the equation $A\mathbf{x} = B$ to get $\mathbf{x}^T A^T = B^T$ where $\mathbf{x}^T$ and B^T are now row vectors. If we set $u = \mathbf{x}^T$, $B = A^T$ and $v = B^T$ then the system of equations $uB = v$ is valid and can be solved for in MatLab as `u=v/B`, where we have now used the right division

operator. Normally, this is not the way systems of equations are written.

Least Squares

- When the number of equations and unknowns differ, a unique solution usually does not exist. However, if we have an **over determined** system of equations (more equations than unknowns) the division operator / automatically find the solution that minimizes the norm of the squared residual.

- For the system of equations $A\mathbf{x} = B$ we compute the residual vector as $R = B - A\mathbf{x}$ and the L_2 (Euclidean) norm of the squared residual as $r = ||R||_2$.

- Consider an example were we add 2 more equations to the above $A\mathbf{x} = B$ system of equations:

$$A = \begin{bmatrix} 4 & 6 & 2 & -14 \\ -16 & 20 & -3 & 5 \\ 25 & 30 & 40 & 50 \\ 9 & -13 & 13 & -28 \\ 3 & -2 & 4 & 120 \\ 19 & -14 & 44 & -1 \end{bmatrix} \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \text{ and } B = \begin{pmatrix} 23 \\ 0 \\ 1 \\ 9 \\ -100 \\ 6 \end{pmatrix}.$$

Note that A is now a 6 row, 4 column matrix, $\mathbf{x}$ is still a 4 component column vector and B is now a 6 component column vector. That is, $A_{6\times 4}\mathbf{x}_{4\times 1} = B_{6\times 1}$. The column dimension of A is the same as the row dimension of $\mathbf{x}$ (both 4) and the result of the multiplication has size (6×1), the column dimension of A by the row dimension of $\mathbf{x}$.

- In MatLab, we have:

```
% add 2 more rows to A
A(5:6,:)=[3 -2 4 120; 19 -14 44 -1]
     4     6     2   -14
   -16    20    -3     5
    25    30    40    50
     9   -13    13   -28
     3    -2     4   120
    19   -14    44    -1
% add 2 more rows to B
B(5:6,:)=[-100; 6]
    23
     0
     1
     9
  -100
     6
% least squares solution
x=A\B
   0.689938282807723
   0.872578663078320
   0.010500035087486
  -0.827909384236666
% compute the residual vector
R=A*x-B
  -3.393043440810892
   2.241513710197055
```

```
   1.450349154208801
   9.183885140015303
   1.017531554216561
  -3.817362981663721
% compute the norm of the squared residual
norm(R)
  10.889996190813942
```

We see that the values for $\mathbf{x}$ are not "perfect" in the sense that they don't perfectly solve all the equations. The residual vector indicates which are the best and worst satisfied equations (4^{th} and 6^{th}).

- The least squares solution can also be computed using right division operator as described above.

- If there are more unknowns than equation we have the **under-determined** case and an infinite number of solutions exist. MatLab has methods to compute 2 of these: the solution with the minimum number of zero el-

ements and the solution with the smallest norm. See Hanselman and Littlefield for more details on this.

Sparse Matrices

- Sometimes matrices only have a few non-zero numbers as a percentage of the size of the matrix. If the dimensions of a matrix are large (say $\geq$ 100) and has a high percentage of zeros, it is wasteful of space to store all the zeros.
- One common form of sparse matrices are diagonal matrices (only diagonal elements are non-zero) and tri-diagonal matrices (where the upper and lower diagonals are also non-zero).
- One way to implement these **sparse** matrices is to store non-zero values only, along with their indices.
- Similarly, computational costs can be significantly reduced if we can

avoid doing arithmetic with zeros. Techniques to do sparse matrix operations are complex and MatLab hides these details from you. From the user point of view, computation on sparse matrices look the same as for non-sparse matrices. Operations on sparse matrices produce sparse matrices while operations on full matrices produce full matrices. Operations on sparse and full matrices generally result in full matrices.

- We present a simple MatLab example (the code is in **L17sparse_matrices_example1.m**).

- Consider the following MatLab code:

```
% setup a sparse matrix of all zeros
A=sparse(10,10)
% set element 1,1 to 100 - so only 1 non-zero element
A(1,1)=100
whos
B=rand(10,10)
C=A*B
whos
C=sparse(C)
whos
```

- This MatLab code produces the following output:

```
% setup a sparse matrix of all zeros
A=sparse(10,10)
A =
   All zero sparse: 10-by-10
A(1,1)=100
(1,1)       100
% The MatLab command, "whos" lists all the variables in workspace
% plus their types. Sparse A needs 10 columns plus 1 double value
% for the row index plus 1 double value for the non-zero value
% plus 1 double for the maximum number of non-zero values
% ==> 10*8+8+8+8=104 bytes
```

```
whos
  Name        Size              Bytes  Class     Attributes
  A          10x10                104  double    sparse
% Generate a 10*10 array of random numbers
B=rand(10,10)
0.7943 0.0838 0.9619 0.9106 0.6221 0.0497 0.3897 0.3532 0.2963 0.7757
0.3112 0.2290 0.0046 0.1818 0.3510 0.9027 0.2417 0.8212 0.7447 0.4868
0.5285 0.9133 0.7749 0.2638 0.5132 0.9448 0.4039 0.0154 0.1890 0.4359
0.1656 0.1524 0.8173 0.1455 0.4018 0.4909 0.0965 0.0430 0.6868 0.4468
0.6020 0.8258 0.8687 0.1361 0.0760 0.4893 0.1320 0.1690 0.1835 0.3063
0.2630 0.5383 0.0844 0.8693 0.2399 0.3377 0.9421 0.6491 0.3685 0.5085
0.6541 0.9961 0.3998 0.5797 0.1233 0.9001 0.9561 0.7317 0.6256 0.5108
0.6892 0.0782 0.2599 0.5499 0.1839 0.3692 0.5752 0.6477 0.7802 0.8176
0.7482 0.4427 0.8001 0.1450 0.2400 0.1112 0.0598 0.4509 0.0811 0.7948
0.4505 0.1067 0.4314 0.8530 0.4173 0.7803 0.2348 0.5470 0.9294 0.6443

% do matrix multiplication C=A*B for sparse matrix A and full matrix B
% Since A(1,1)=100 only the 1st row (multiplied by 100) is copied
% into C
C=
79.4285 8.3821 96.1898 91.0648 62.2055 4.9654 38.9739 35.3159 29.6321 77.5713
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
```

```
 0      0       0       0       0      0       0       0       0       0
 0      0       0       0       0      0       0       0       0       0
% get size and type information for variables A, B and C
whos
  Name        Size              Bytes  Class     Attributes
  A          10x10                104  double    sparse
  B          10x10                800  double
  C          10x10                800  double
% make C sparse
C=sparse(C)
   (1,1)       79.4285
   (1,2)        8.3821
   (1,3)       96.1898
   (1,4)       91.0648
   (1,5)       62.2055
   (1,6)        4.9654
   (1,7)       38.9739
   (1,8)       35.3159
   (1,9)       29.6321
   (1,10)      77.5713
% get size and type information again
whos
  Name        Size              Bytes  Class     Attributes
  A          10x10                104  double    sparse
  B          10x10                800  double
  C          10x10                248  double    sparse
```

- `C` has 10 columns (10 doubles for 80 bytes), 10 row indices (10 doubles for 80 bytes), 10 non-zero values (10 doubles for 80 bytes) and 1 double (8 bytes) for the number of non-zero values than can be used before the sparse array needs to be re-allocated.

- Another slightly more complex example of sparse matrices is given below and is in **L17sparse_matrices_example2.m**:

```
A=rand(5,5);
B=zeros(5,5,'double');
B(2,3)=1.0;
B(5,5)=1.0;

disp('full A');
A
disp('whos full A');
whos A
disp('full B');
B
disp('whos full B');
whos B
```

```
disp('full A*B');
A*B
```

- `whos A` gives the size and storage used by a variable `A`. Full `A` used 800 bytes (8 * 100 array elements) while sparse `A` uses 120 bytes (10 column indices for 80 bytes, 2 row indices for 16 bytes and 2 non-zero values for 16 bytes plus 1 double (8 bytes) for the maximum number of non-zero values that have been pre-allocated).

- This MatLab code has output:

```
full A
A = 0.8147    0.0975    0.1576    0.1419    0.6557
    0.9058    0.2785    0.9706    0.4218    0.0357
    0.1270    0.5469    0.9572    0.9157    0.8491
    0.9134    0.9575    0.4854    0.7922    0.9340
    0.6324    0.9649    0.8003    0.9595    0.6787
whos full A
  Name      Size              Bytes  Class     Attributes
  A         5x5                 200  double
```

```
full B
B =  0     0     0     0     0
     0     0     1     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     1
whos full B
  Name       Size              Bytes  Class     Attributes
  B          5x5                 200  double
full A*B
          0         0    0.0975         0    0.6557
          0         0    0.2785         0    0.0357
          0         0    0.5469         0    0.8491
          0         0    0.9575         0    0.9340
          0         0    0.9649         0    0.6787
```

- For matrix multiplication, the i^{th} row of A is multiplied by the j^{th} column of B (basically the dot product of these 2 row and column vectors) to give element A*B(i,j). Since B(3,2)=1, when multiplying all the rows of A by the 3^{rd} column of B, all the 2^{nd} elements of the rows become the corresponding elements of the 3^{rd} column of A*B. Similarly, since

B(5,5)=1, when multiplying all the rows of A by the 5^{th} column of B, all the 5^{th} elements of the rows become the corresponding elements of the 5^{th} column of A*B.

- Continuing with the code:

```
B=sparse(B);
% a double for each column, a double for each non-zero row index,
% a double for each non-zero value and a double for maximum number
% of non-zero values before re-allocation is needed.
% sparse array. 5*8+2*2*8+8=80 bytes
disp('sparse B');
B
disp('whos sparse B');
whos B
disp('C=A*sparse(B)');
C=A*B
disp('whos C=A*sparse(B)');
whos C
B=full(B);
disp('full B');
B
disp('whos full B');
whos B
```

- This code segment has the output:

```
sparse B
B = (2,3)        1
    (5,5)        1
whos sparse B
  Name      Size             Bytes  Class     Attributes
  B         5x5                 80  double    sparse
C=A*sparse(B)
         0         0    0.0975         0    0.6557
         0         0    0.2785         0    0.0357
         0         0    0.5469         0    0.8491
         0         0    0.9575         0    0.9340
         0         0    0.9649         0    0.6787
whos C=A*sparse(B)
  Name      Size             Bytes  Class     Attributes
  C         5x5                200  double
full B
B =  0     0     0     0     0
     0     0     1     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     1
whos full B
  Name      Size             Bytes  Class     Attributes
  B         5x5                200  double
```

- We can see that `C=A*full(B)` and `C=A*sparse(B)` gives the same result.

- Finally, MatLab code:

```
i=[2 5];
j=[3 5];
v=[1.0 1.0];
S=sparse(i,j,v,5,5,25);
% S is a 5*5 sparse array with a maximum of 25 values
% i are the row indices, j are the column indices and
% v gives the values
disp('S=sparse(i,j,v,5,5,25)');
S
% 5 doubles for column indices, 25 doubles for row indices,
% 25 doubles for matrix values, 1 double for value 25
% ==> 5*8+25*8+25*8+8=440 bytes
disp('whos S')
whos S
```

has output:

```
S=sparse(i,j,v,5,5,25)
```

```
S = (2,3)        1
    (5,5)        1
whos S
  Name      Size              Bytes  Class     Attributes
  S         5x5                 448  double    sparse
```

- Yes!!! This particular sparse array requires more than twice the space (448 bytes) than the full non-space array does (200 bytes). Storing a full matrix in sparse format is space inefficient. Indeed, if we stored this matrix without using 2 as the value of `nz` it would require 80 bytes [5 column indices (5*8 bytes) plus 2 row indices and 2 non-zero values (2*2*8 bytes) and the number of non-zero values (8 bytes)].

- `S = sparse(i,j,v,n,m,nz)` is a general form of the MatLab statement to construct a sparse matrix from the row and column indices stored in vectors `i` and `j` with corresponding non-zero values stored in `v`. The

matrix dimensions are `n` by `m`.

- If you do not specify `m` and `n`, then `sparse` uses the default values `m = max(i)` and `n = max(j)`.
- `nz` is the storage allocation for non-zero values, i.e. it is the maximum number of non-zero spaces to be allocated. If the number of non-zero values are less or than equal to `nz` then `A` does not have to be re-allocated when additional non-zero values are added to the array (up until the size exceeds `nz`). If, on the other hand, `nz` is less than the number of non-zero values, `A` would have to be re-allocated (this dynamic array reallocation is an expensive operation).
- `nz` must be greater than or equal to

`max([numel(i),numel(j),numel(v)])`.

  - The function `nnz` returns the number of non-zero values in the matrix while the function `nzmax` returns the amount of storage allocated for non-zero values in general. If `nnz(A)` and `nzmax(A)` return different results then more storage might be allocated than is actually required. For this reason, only set `nz` in anticipation of later addition of non-zero values to the array.

- `whos A` gives the size and storage used by a variable. For example, `whos A` gives:

```
  Name        Size              Bytes  Class     Attributes
  A          10x10                800  double
```

  for a full 10×10 matrix while `whos A` gives:

```
  Name          Size                Bytes  Class     Attributes
  A            10x10                  104  double    sparse
```

for a sparse 10×10 matrix with 2 non zero values. In this case we store 10 column indices as doubles (80 bytes), 2 row indices as doubles (16 bytes), 2 non-zero values as doubles (16 bytes) and 1 double (8 bytes) for the maximum non-zero values to be stored: 80+16+16+8=120 bytes. For this example, both `nnz(A)` and `nzmax(A)` have 2 as their values.

Least Squares Curve Fitting (Regression)

- You may be faced with the task of fitting a curve through data points in your academic or industrial careers.
- Sometimes, the curve interpolates the data points (passes through them) and sometimes the curve only **approximates** the data points (passes by them but not through them).
- We demonstrate least square curve fitting below (using `polyfit` and `polyval` and using Least Squares directly on the **Vandermonde** matrix), where we fit the data to a polynomial of order n.
- `polyfit` fits a polynomial of the specified order to a dataset using the appropriate Vandermonde matrix. `polyval` evaluates for a vector of

values. Note this vector can be much longer than the input data used to fit the polynomial coefficients.

- A general polynomial of order n can be written as:

$$y = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ... + a_2 x^2 + a_1 x + a_0.$$

- For a order 3 polynomial we have to solve the Vandermonde matrix:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{bmatrix} x_0^3 & x_0^2 & x_0^1 & x_0^0 \\ x_1^3 & x_1^2 & x_1^1 & x_1^0 \\ x_2^3 & x_2^2 & x_2^1 & x_2^0 \end{bmatrix} \begin{pmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix},$$

while for an order 10 polynomial the Vandermonde matrix becomes:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{pmatrix} = \begin{bmatrix} x_0^{10} & x_0^9 & x_0^8 & x_0^7 & x_0^6 & x_0^5 & x_0^4 & x_0^3 & x_0^2 & x_0^1 & x_0^0 \\ x_1^{10} & x_1^9 & x_1^8 & x_1^7 & x_1^6 & x_1^5 & x_1^4 & x_1^3 & x_1^2 & x_1^1 & x_1^0 \\ x_2^{10} & x_2^9 & x_2^8 & x_2^7 & x_2^6 & x_2^5 & x_2^4 & x_2^3 & x_2^2 & x_2^1 & x_2^0 \\ x_3^{10} & x_3^9 & x_3^8 & x_3^7 & x_3^6 & x_3^5 & x_3^4 & x_3^3 & x_3^2 & x_3^1 & x_3^0 \\ x_4^{10} & x_4^9 & x_4^8 & x_4^7 & x_4^6 & x_4^5 & x_4^4 & x_4^3 & x_4^2 & x_4^1 & x_4^0 \\ x_5^{10} & x_5^9 & x_5^8 & x_5^7 & x_5^6 & x_5^5 & x_5^4 & x_5^3 & x_5^2 & x_5^1 & x_5^0 \\ x_6^{10} & x_6^9 & x_6^8 & x_6^7 & x_6^6 & x_6^5 & x_6^4 & x_6^3 & x_6^2 & x_6^1 & x_6^0 \\ x_7^{10} & x_7^9 & x_7^8 & x_7^7 & x_7^6 & x_7^5 & x_7^4 & x_7^3 & x_7^2 & x_7^1 & x_7^0 \\ x_8^{10} & x_8^9 & x_8^8 & x_8^7 & x_8^6 & x_8^5 & x_8^4 & x_8^3 & x_8^2 & x_8^1 & x_8^0 \\ x_9^{10} & x_9^9 & x_9^8 & x_9^7 & x_9^6 & x_9^5 & x_9^4 & x_9^3 & x_9^2 & x_9^1 & x_9^0 \\ x_{10}^{10} & x_{10}^9 & x_{10}^8 & x_{10}^7 & x_{10}^6 & x_{10}^5 & x_{10}^4 & x_{10}^3 & x_{10}^2 & x_{10}^1 & x_{10}^0 \end{bmatrix} \begin{pmatrix} a_{10} \\ a_9 \\ a_8 \\ a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}.$$

- Naturally, all x_i^1 are just x_i and x_i^0 values are 1's for the last 2 columns of both of these matrices.
- MatLab has a function **vander** that evaluates this matrix given x.
- When `x=0:0.333333:1` as for the above order 3 fitting, the Vandermonde matrix (via `A=vander(x)`) is:

```
A=
         0         0         0    1.0000
    0.0370    0.1111    0.3333    1.0000
    0.2963    0.4444    0.6667    1.0000
    1.0000    1.0000    1.0000    1.0000
```

with a condition number `cond(A)` being `98.8679` which is quite good (the matrix is stable, well-conditioned and invertible).

- When `x=0:0.1,1` as for the above order 10 fitting, the Vandermonde matrix (via `A=vander(x)`) is:

```
A =
0       0       0       0       0       0       0       0       0       0       1.0000
0       0       0       0       0       0       0.0001  0.0010  0.0100  0.1000  1.0000
0       0       0       0       0.0001  0.0003  0.0016  0.0080  0.0400  0.2000  1.0000
0       0       0.0001  0.0002  0.0007  0.0024  0.0081  0.0270  0.0900  0.3000  1.0000
0.0001  0.0003  0.0007  0.0016  0.0041  0.0102  0.0256  0.0640  0.1600  0.4000  1.0000
0.0010  0.0020  0.0039  0.0078  0.0156  0.0312  0.0625  0.1250  0.2500  0.5000  1.0000
0.0060  0.0101  0.0168  0.0280  0.0467  0.0778  0.1296  0.2160  0.3600  0.6000  1.0000
0.0282  0.0404  0.0576  0.0824  0.1176  0.1681  0.2401  0.3430  0.4900  0.7000  1.0000
0.1074  0.1342  0.1678  0.2097  0.2621  0.3277  0.4096  0.5120  0.6400  0.8000  1.0000
```

```
0.3487 0.3874 0.4305 0.4783 0.5314 0.5905 0.6561 0.7290 0.8100 0.9000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
```

with a condition number `cond(A)=1.1558e+08`. This is slightly unstable already but probably can be inverted ok.

- Consider the Vandermonde matrix for `x=0:1:100`. `cond(vander(x))` returns `3.6872e+210` which is unsolvable as the matrix if very unstable and effectively is singular!!! The Vandermonde matrixes are characterized by having both small numbers and very large numbers (relatively speaking) in the matrix: this is what makes the matrix unstable.

- Given n and x and y we setup and solve a least squares system of equations to find the coefficients (x and y must have at least n elements).

- We first perform a linear regression fit for order $n = 1$ (a line fit), a quadratic fit for order $n = 2$, a cubic fit for order $n = 3$, a 6th order fit for $n = 6$, a 8th order fit for $n = 8$ an finally a 10^{th} fit for $n = 10$ using the MatLab function `polyfit`

- We illustrate this function using the original and modified versions of the 11 data points in Hanselman and Littlefield (page 358). The modified version has point 9 being an **outlier**, an obvious large error. This is opposed to the rest of the data points, which are **inliers** and have small error. The MatLab code **L17fit_polynomials.m** is given below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% L17fit_polynomials of various orders to the data                      %
% An order 1 fit is a line                                              %
% An order 2 fit is a quadratic                                         %
% An order 3 fit is a cubic                                             %
% Order 10 is the highest order polynomial that can be                  %
```

```
% fit to the data because we have 11 x,y data points only %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x=[0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0];
% Only inliers in y1
y1=[-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.59 9.48 9.3 11.2];
% y2 has 1 outlier at y2(9): -1.59 instead of 9.59
y2=[-0.447 1.978 3.28 6.16 7.08 7.34 7.66 -1.59 9.48 9.3 11.2];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fit using polyfit and polyval                              %
% polyfit fits the order n polynomial (n=1,3,6,8,10)         %
% to the y1 and y2 data via a least squares calculation      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

p1y1=polyfit(x,y1,1);
p1y2=polyfit(x,y2,1);
p2y1=polyfit(x,y1,2);
p2y2=polyfit(x,y2,2);
p3y1=polyfit(x,y1,3);
p3y2=polyfit(x,y2,3);
p6y1=polyfit(x,y1,6);
p6y2=polyfit(x,y2,6);
p8y1=polyfit(x,y1,8);
p8y2=polyfit(x,y2,8);
p10y1=polyfit(x,y1,10);
p10y2=polyfit(x,y2,10);
```

```
% use vectorized fprintf to print the coefficients
% of the fit polynomials
fprintf('Order 1 polyfit coefficients for y1\n');
fprintf('%f ',p1y1);
fprintf('\n\n');
fprintf('Order 1 polyfit coefficients for y2\n');
fprintf('%f ',p1y2);
fprintf('\n\n');

fprintf('Order 2 polyfit coefficients for y1\n');
fprintf('%f ',p2y1);
fprintf('\n\n');
fprintf('Order 2 polyfit coefficients for y2\n');
fprintf('%f ',p2y2);
fprintf('\n\n');

fprintf('Order 3 polyfit coefficients for y1\n');
fprintf('%f ',p3y1);
fprintf('\n\n');
fprintf('Order 3 polyfit coefficients for y2\n');
fprintf('%f ',p3y2);
fprintf('\n\n');

fprintf('Order 6 polyfit coefficients for y1\n');
fprintf('%f ',p6y1);
fprintf('\n\n');
fprintf('Order 6 polyfit coefficients for y2\n');
fprintf('%f ',p6y2);
```

```
fprintf('\n\n');

fprintf('Order 8 polyfit coefficients for y1\n');
fprintf('%f ',p8y1);
fprintf('\n\n');
fprintf('Order 8 polyfit coefficients for y2\n');
fprintf('%f ',p8y2);
fprintf('\n\n');

fprintf('Order 10 polyfit coefficients for y1\n');
fprintf('%f ',p10y1);
fprintf('\n\n');
fprintf('Order 10 polyfit coefficients for y2\n');
fprintf('%f ',p10y2);
fprintf('\n\n');

% generate xp as 100 evenly spaced values from 0 to 1
xp=linspace(0,1,100);
% compute 100 y1 and y2 values from sets of polynomial
% coefficients p1y1 to p10y1 and p1y2 to p10y2
% Use polyval to compute these 100 y1 or y2 values
% from the 100 xp values
y1order1=polyval(p1y1,xp);
y2order1=polyval(p1y2,xp);
y1order2=polyval(p2y1,xp);
y2order2=polyval(p2y2,xp);
y1order3=polyval(p3y1,xp);
y2order3=polyval(p3y2,xp);
```

```
y1order6=polyval(p6y1,xp);
y2order6=polyval(p6y2,xp);
y1order8=polyval(p8y1,xp);
y2order8=polyval(p8y2,xp);
y1order10=polyval(p10y1,xp);
y2order10=polyval(p10y2,xp);

figure
% Plot inlier data polynomial fits
plot(x,y1,'-ok',xp,y1order1,'-r',xp,y1order2,'-g',...
                xp,y1order3,'-b',xp,y1order6,'-y',...
                xp,y1order8,'-m',xp,y1order10,'-c',...
                'linewidth',2.0); % default linewidth is 0.5
xlabel('xp'); ylabel('y1');
title(['1st, 2nd, 3rd, 6th, 8th and 10th '...
       'Polynomial Fitting for Inliers Data']);
legend('Original Data','1st order fit','2nd order fit',...
       '3rd order fit','6th order fit','8th order fit',...
       '10th order fit','location','northwest');
print 'inlier_polynomial_fits_1_2_3_6_8_10.jpg' -djpeg

figure
% Plot outlier data polynomial fits
plot(x,y2,'-ok',xp,y2order1,'-r',xp,y2order2,'-g',...
                xp,y2order3,'-b',xp,y2order6,'-y',...
                xp,y2order8,'-m',xp,y2order10,'-c',...
                'linewidth',2.0); % default linewidth is 0.5
% Redraw the line to emphasize where it is
```

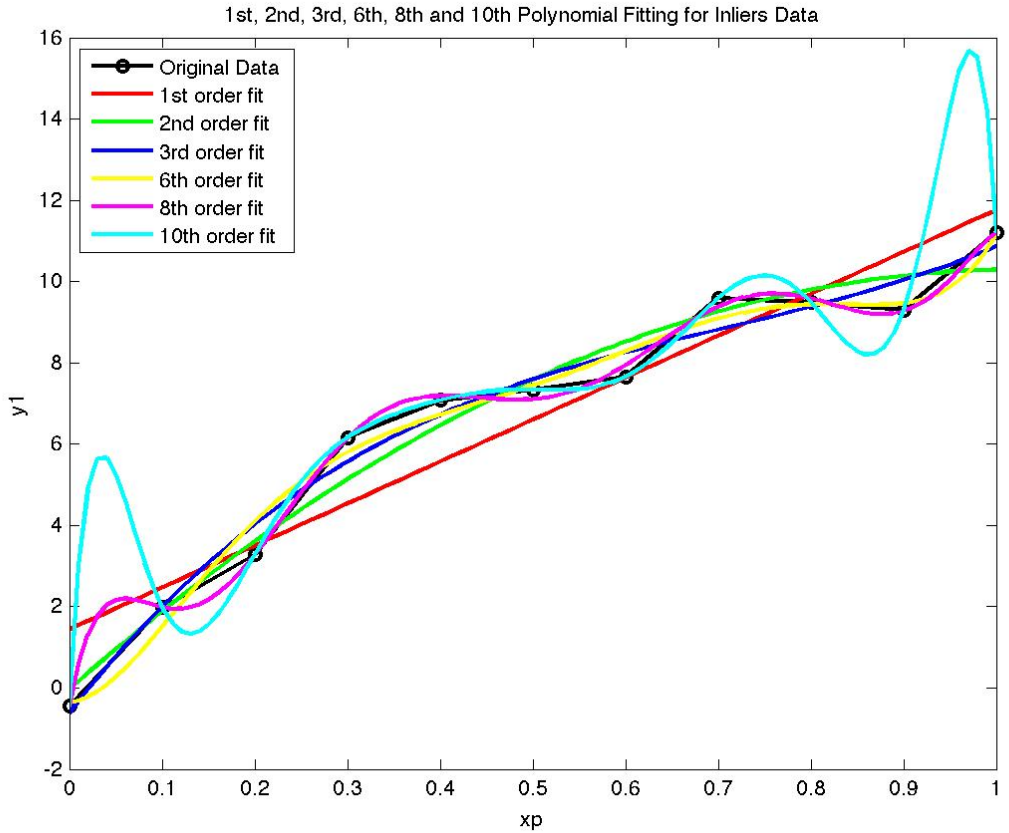
```
hold on
plot(xp,y2order1,'-r','linewidth',2.0);
xlabel('xp'); ylabel('y2');
title(['1st, 2nd, 3rd, 6th, 8th and 10th ' ...
       'Polynomial Fitting for Outliers Data']);
legend('Original Data','1st order fit','2nd order fit',...
       '3rd order fit','6th order fit','8th order fit',...
       '10th order fit','location','southwest');
print 'outlier_polynomial_fits_1_2_3_6_8_10.jpg' -djpeg
```

- This program computes the following polynomial coefficients:

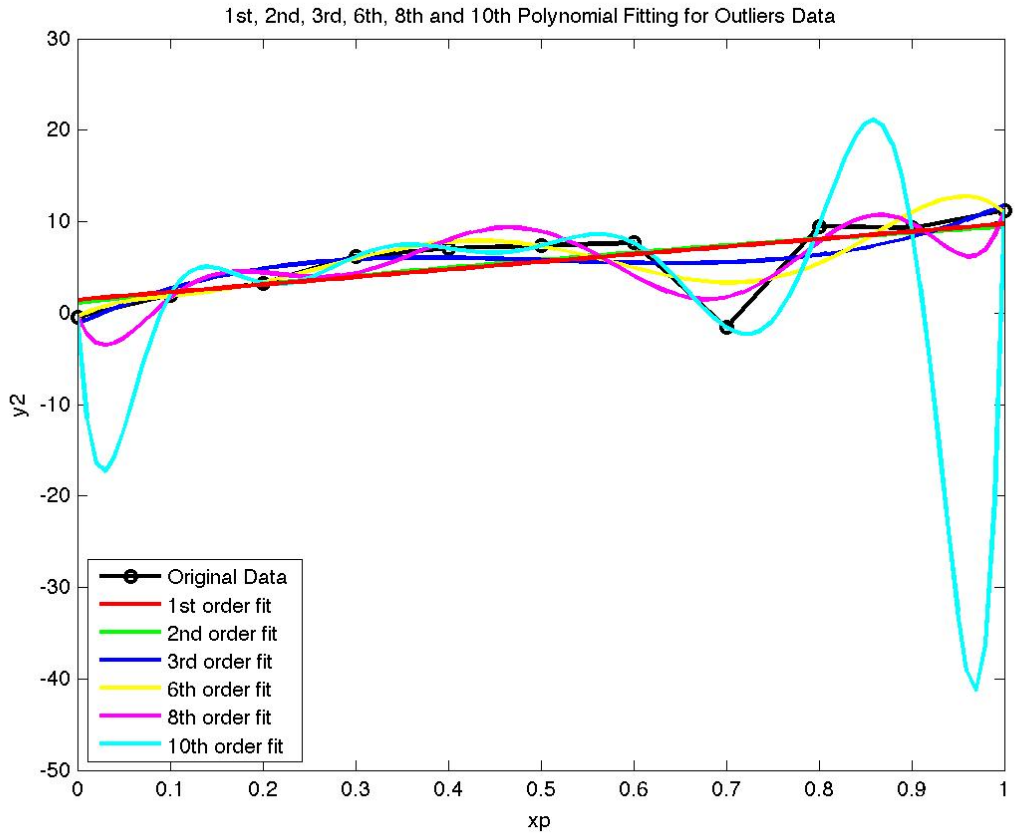
```
Order 1 polyfit coefficients for y1
10.323909 1.439955

Order 1 polyfit coefficients for y2
8.291182 1.439955

Order 2 polyfit coefficients for y1
-9.831818 20.155727 -0.034818

Order 2 polyfit coefficients for y2
-2.013636 10.304818 1.137909

Order 3 polyfit coefficients for y1
15.941725 -33.744406 29.274394 -0.608720

Order 3 polyfit coefficients for y2
```

```
65.891220 -100.850466 47.994596 -1.234175

Order 6 polyfit coefficients for y1
706.789216 -2106.761878 2371.762538 -1231.960189
269.994734 1.679947 -0.351209

Order 6 polyfit coefficients for y2
-2581.446078 7041.277338 -6894.315894 2927.580810
-535.370739 53.622567 -0.420193

Order 8 polyfit coefficients for y1
-16699.691938 70961.291095 -122810.832688 111004.082172
-55748.066128 15234.227853 -2052.393734 123.034442 -0.448106

Order 8 polyfit coefficients for y2
82537.985589 -324684.563738 516760.546740 -426511.819100
194478.226871 -48204.033069 5877.413896 -242.216490 -0.400180

Order 10 polyfit coefficients for y1
-474280.753994 2344109.623141 -4974920.635180 5935037.946731
-4373073.906467 2055752.526141 -612845.324186 110560.544698
-10769.509705 441.135821 -0.447000

Order 10 polyfit coefficients for y2
3222809.193264 -15401922.123702 31404444.445867 -35701589.039352
24811384.428269 -10864911.363390 2975387.506689 -486744.264860
42628.299822 -1475.435607 -0.447000
```

- The **L17fit_polynomials.m** program produces 2 plots of 7 curves for the $y1$ and $y2$ values. These are shown below:



Polynomial fits for order 1, 2, 3 ,6, 8 and 10 polynomials for the $y1$ inlier data.



Polynomial fits for order 1, 2, 3 ,6, 8 and 10 polynomials for the $y2$ outlier data.

- Some comments are in order:

  1. The outlier in the $y2$ data has a significant effect on how the curves appear versus those same curves for the $y1$ data, which consists of inlier data only. For example, the slope and y intercept of the line (order 1 polynomial) has changed significantly. Identification and removal of outlier data or a **robust** fitting of the polynomials to the data might be in order.

  2. The order 10 polynomials "perfectly" fit the data in that all the $(x, y1)$ or $(x, y2)$ data are interpolated. However, the Vandermonde matrix used in the least squares fitting has a larger and larger condition number as the order increases. This indicates numerical instability and the answer should not be trusted for large orders. For

order 10, estimation of data between interpolated data points can drastically wrong!!! Notice how the magnitude of the coefficients for higher order polynomials varies greatly.

3. The best fits are typically the low order polynomials, for example, the cubic polynomial fits the data, it numerically stable (low condition number) and is able to give good estimates for points between the data points. A line may not always be a good fit because the relationship among the data may not be linear!

4. MatLab does a "beautiful" job of plotting all these polynomials on a single graph!!!