Relevant Sections of the Textbook

- §§16.1, 16.4
- Part of §16.5
- §19.8
- See also §32.6
Solving Matrix Problems

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 $x_4$: 
\[4x_1 + 6x_2 + 2x_3 - 14x_4 = 23\]
\[-16x_1 + 20x_2 - 3x_3 + 5x_4 = 0\]
\[25x_1 + 30x_2 + 40x_3 + 50x_4 = 1\]
\[9x_1 - 13x_2 + 13x_3 - 28x_4 = 9\]

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

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

or \(Ax = b\), where \(4 \times 4\) 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 $4 \times 1$ vectors $x$ and $b$ are:

$$
\begin{pmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    x_4 \\
\end{pmatrix}
\text{ and }
\begin{pmatrix}
    23 \\
    0 \\
    1 \\
    9 \\
\end{pmatrix}.
$$

- 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:

```matlab
A = [4 6 2 -14; -16 20 -3 5; 25 30 40 50; 9 -13 13 -28]
B = [23; 0; 1; 9]
```
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 $\text{cond}(A)$ 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 $4 \times 4$ identity matrix:
Then the solution to $Ax = B$ is simply $x = A^{-1}b$.

```
format long
x=inv(A)*B
1.457544673236385
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 simpler to code). So solve for $x$ as $A\backslash b$. 
• 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:

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

We see that both solution methods give the same solution. But this happens because the system of equations is so well-conditioned (as shown by the very small condition number).
• Note that the \ operation most likely uses an LU factorization \ ($A = LU$ with $L$ lower triangular and $U$ upper triangular) approach to solve this system of equations but it does analyze 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 $Ax = b$ to get

$$x^T A^T = b^T,$$

where $x^T$ and $b^T$ are now row vectors.
• If we set \( u = x^T \), \( B = A^T \) 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 finds the solution that minimizes the norm of the residual.
• For the system of equations $Ax = b$ we compute the residual vector as $r = b - Ax$ and the (Euclidean) $2$-norm of the residual as $R = \|r\|_2$.

• Consider an example where we add 2 more equations to the above $Ax = B$ system of equations:

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

Note that $A$ is now a 6 row, 4 column matrix, $x$ is still a 4 component column vector and $b$ is now a 6 component column vector.
• That is, $A_{6\times4}x_{4\times1} = b_{6\times1}$. The column dimension of $A$ is the same as the row dimension of $x$ (both 4) and the result of the multiplication has size $(6 \times 1)$, the row dimension of $A$ by the column dimension of $x$.

• In MATLAB, we have:

```matlab
% 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
  0.01050035087486
-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 $x$ are not “perfect” in the sense that they
don’t solve all the equations exactly. The residual vector indicates
which are the best and worst satisfied equations ($4^{th}$ and $6^{th}$).
• If we round \( x \) to the nearest \( n \)th decimal place, using \( \text{round}(x,n) \), we see that the norm of the residual always increases. For example:

\[
\begin{align*}
xx &= \text{round}(x,2) \\
xx &= \\
&= 0.690000000000000 \\
&= 0.870000000000000 \\
&= 0.010000000000000 \\
&= -0.830000000000000 \\
\end{align*}
\]

\[
\begin{align*}
rr &= A \times xx - b \\
rr &= \\
&= -3.380000000000003 \\
&= 2.180000000000001 \\
&= 1.250000000000002 \\
&= 9.270000000000000 \\
&= 0.770000000000010 \\
&= -3.800000000000001 \\
norm(rr) &= \\
\text{ans} &= \\
&= 10.895187010786003
\end{align*}
\]

• You can try this yourself by modifying the code in L17matrix_computations.m.
• The least squares solution can also be computed for transposed equations of the form $uB = v$ using right division operator as described above.

• If there are more unknowns than equation we have the underdetermined case and an infinite number of solutions exist. MATLAB has methods to compute certain particular solutions, the solution with the minimum number of zero elements and the solution with the smallest norm, as well as strategies for getting (more) general solutions. See Hanselman and Littlefield and the MATLAB document page Systems of Linear Equations (in Documentation > MATLAB > Mathematics > Linear Algebra) 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.
• 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, computations 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:

```matlab
% setup a sparse matrix of all zeros
A=sparse(10,10)

% the number of nonzero elements of A
disp('nnz(A)')
nnz(A)

% the number of possible nonzero elements before reallocation is needed
disp('nzmax(A)')
nzmax(A)

% set element 1,1 to 100 - so only 1 non-zero element
A(1,1)=100

disp('nnz(A)')
nnz(A)

% The MATLAB command, "whos" lists all the variables in workspace
% plus their types and the amount of memory used
disp('whos')
whos
```
• This MATLAB code produces the following output:

```matlab
A = 
    All zero sparse: 10x10
nnz(A) 
ans = 
    0
nzmax(A) 
ans = 
    1
A = 
    (1,1) 100
nnz(A) 
ans = 
    1
nzmax(A) 
ans = 
    1
whos
    Name      Size      Bytes  Class Attributes
A        10x10      104    double    sparse
```

• A has only one possible entry when it’s created, so the underlying array needs to be reallocated when more than one element is added.
• Continuing with the code:

```
B=rand(10,10)

disp('C=A*B')
C=A*B

disp('whos')
whos

disp('C=sparse(C)')
C=sparse(C)

disp('whos')
whos

% add an element to A, exceeding the number of possible nonzero elements
A(1,2) = 2

disp('nzmax(A)')
nzmax(A)

disp('whos')
whos
```
• This code has the output:

\[
B = \\
\begin{bmatrix}
0.2729 & 0.1253 & 0.1265 & 0.8929 & 0.1662 & 0.4022 & 0.5306 & 0.4243 & 0.2405 & 0.8236 \\
0.0372 & 0.1302 & 0.1343 & 0.7032 & 0.6225 & 0.6207 & 0.8324 & 0.4294 & 0.7639 & 0.1750 \\
0.6733 & 0.0924 & 0.0986 & 0.5557 & 0.9879 & 0.1544 & 0.5975 & 0.1249 & 0.7593 & 0.1636 \\
0.4296 & 0.0078 & 0.1420 & 0.1844 & 0.1704 & 0.3813 & 0.3353 & 0.0244 & 0.7406 & 0.6660 \\
0.4517 & 0.4231 & 0.1683 & 0.2120 & 0.2578 & 0.1611 & 0.2992 & 0.2902 & 0.7437 & 0.8944 \\
0.6099 & 0.6556 & 0.1962 & 0.0773 & 0.3968 & 0.7581 & 0.4526 & 0.3175 & 0.1059 & 0.5166 \\
0.0594 & 0.7229 & 0.3175 & 0.9138 & 0.0740 & 0.8711 & 0.4226 & 0.6537 & 0.6816 & 0.7027 \\
0.3158 & 0.5312 & 0.3164 & 0.7067 & 0.6841 & 0.3508 & 0.3596 & 0.9569 & 0.4633 & 0.1536 \\
0.7727 & 0.1088 & 0.2176 & 0.5578 & 0.4024 & 0.6855 & 0.5583 & 0.9357 & 0.2122 & 0.9535 \\
0.6964 & 0.6318 & 0.2510 & 0.3134 & 0.9828 & 0.2941 & 0.7425 & 0.4579 & 0.0985 & 0.5409
\end{bmatrix}
\]

\[
C = A*B \\
C = \\
\begin{bmatrix}
27.2939 & 12.5332 & 12.6500 & 89.2922 & 16.6204 & 40.2184 & 53.0629 & 42.4335 & 24.0478 & 82.3574 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10x10</td>
<td>104</td>
<td>double</td>
<td>sparse</td>
</tr>
<tr>
<td>B</td>
<td>10x10</td>
<td>800</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10x10</td>
<td>800</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

C = sparse(C)
C =

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>27.2939</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,2)</td>
<td>12.5332</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,3)</td>
<td>12.6500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,4)</td>
<td>89.2922</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,5)</td>
<td>16.6204</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,6)</td>
<td>40.2184</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,7)</td>
<td>53.0629</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,8)</td>
<td>42.4335</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,9)</td>
<td>24.0478</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,10)</td>
<td>82.3574</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nzmax(C)
an =
10

whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10x10</td>
<td>104</td>
<td>double</td>
<td>sparse</td>
</tr>
<tr>
<td>B</td>
<td>10x10</td>
<td>800</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10x10</td>
<td>248</td>
<td>double</td>
<td>sparse</td>
</tr>
</tbody>
</table>
\[
A = \\
\begin{pmatrix}
1,1 & 100 \\
1,2 & 2
\end{pmatrix}
\]

\text{nzmax}(A)
\text{ans} = 
11

\text{whos}
\begin{array}{cccc}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} & \text{Attributes} \\
A & 10x10 & 264 & \text{double} & \text{sparse} \\
B & 10x10 & 800 & \text{double} & \\
C & 10x10 & 248 & \text{double} & \text{sparse} \\
\end{array}

- Converting \( C \) from dense to sparse converts it from needing 800 bytes (8 * 100 array elements), as is needed for \( B \), to needing only 248 bytes.
• Before adding a second element \( A \) needed only 104 bytes, but after exceeding \( \text{nzmax}(A)=1 \), reallocation of arrays was needed, and \( A \) now needs 264 bytes, more than \( C \) needs, and \( C \) has more elements!

• This shows that MATLAB is making decisions about how much space to allocate for spare matrix data, based on some sort of general case.

• Indeed, we see that by adding a second element to \( A \), the number of possible nonzero elements before reallocation has been raised from 1 to 11. This is because reallocation is expensive in terms of time, so MATLAB is giving a bunch of space to add elements before reallocation is necessary again.
• On the other hand, \( \text{nzmax}(C) = 10 \), exactly the number of nonzero elements in \( C \), which is why \( C \) uses less space.

• This shows that it is better to avoid having to change the maximum number of nonzero elements where possible, and that if \( \text{nzmax} \) is fixed for a particular matrix, then MATLAB can use space more efficiently.

• The general way MATLAB stores sparse matrix data uses three arrays in the following way:

  1. The nonzero elements are stored in a one-dimensional array of double-precision reals, in column major order (if the matrix is complex, the imaginary parts are stored in another such array).

  2. A second array of integers stores the row indices.

  3. A third array of \( n + 1 \) integers stores the index into the first two arrays of the leading entry in each of the \( n \) columns, and a terminating index whose value is \( \text{nnz} \).
• Thus a real $m \times n$ sparse matrix with $nzmax$ nonzeros uses $nzmax$ reals and $nzmax + n + 1$ integers.

• So for $A$ with $nzmax=1$, it uses 8 bytes (1 element) for the data values, 8 bytes for the row index, $8 \times 10$ bytes for the column-change index, and 8 bytes for $nnz$, totalling $8 + 8 + (10 + 1) \times 8 = 104$ bytes.

• Once $nzmax$ has been raised to 11, $A$ needs $11 \times 8$ bytes for the data values, $11 \times 8$ bytes for the row indices, $8 \times 10$ bytes for the column-change indices, and 8 bytes for $nnz$, totalling $88 + 88 + 80 + 8 = 264$ bytes.

• For $C$ on the other hand, $nzmax$ is only 10, so 16 fewer bytes are needed.
Another slightly more complex example of sparse matrices is given below and is in \texttt{L17sparse\_matrices\_example2.m}:

```matlab
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
```
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 $[A \ast B](i, j)$. Since $B(2, 3) = 1$, but otherwise $B(i, 3) = 0$, 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 $A \ast B$. Similarly, since $B(5, 5) = 1$, but otherwise $B(i, 5) = 0$, 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 $A \ast B$. 
• Continuing with the code:

```matlab
B = sparse(B);
% a double for each data possible entry, a double for each row index of a data entry,
% a double for the index of the start of each column in the data array,
% and a double for the number of non-zero values in the sparse array.
% 2*8 + 2*8 + 5*8 + 8 = 80 bytes
disp('sparse B');
B
disp('nnz(B)')
nnz(B)
disp('nzmax(B)')
nzmax(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:

```matlab
sparse B
B = (2,3) 1
    (5,5) 1

nnz(B)
ans =
    2

nzmax(B)
ans =
    2

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
```
matrix 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
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>5x5</td>
<td>200</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

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

- Finally, MATLAB code:

```matlab
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

% 25 doubles for each possible data entry, 25 doubles for row indices,
% 5 doubles for the index of the start of each column in the data,
% and 1 double for the number of nonzero entries (2 in this case)
% ==> 25*8+25*8+5*8+8=448 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 25 as the value of \texttt{nzmax} using \texttt{S=sparse(i,j,v,5,5)}, \texttt{nzmax} would be set to 2, requiring only 80 bytes [2 data entries (2*8 bytes), 2 row indices (2*8 bytes), 5 column indices (5*8 bytes) and the number of non-zero values (8 bytes)].

• \texttt{S = sparse(i,j,v,n,m,nzmax)} is a general form of the MATLAB statement to construct a sparse matrix from the row and column indices stored in vectors \texttt{i} and \texttt{j} with corresponding non-zero values stored in \texttt{v}. The matrix dimensions are \texttt{n} by \texttt{m}.
  
  – If you do not specify \texttt{m} and \texttt{n}, then \texttt{sparse} uses the default values \texttt{m = max(i)} and \texttt{n = max(j)}. 

- \texttt{nzmax} 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 (\texttt{nnz}) are less or than equal to \texttt{nzmax} then \texttt{A} does not have to be re-allocated when additional non-zero values are added to the array (up until the size exceeds \texttt{nzmax}).

- If, on the other hand, \texttt{nzmax} is less than the number of non-zero values, \texttt{A} would have to be re-allocated (this dynamic array reallocation is an expensive operation).

- \texttt{nzmax} must be greater than or equal to

\[
\text{max([numel(i),numel(j),numel(v)])}.
\]
The function \texttt{nnz} returns the number of non-zero values in the matrix while the function \texttt{nzmax} returns the amount of storage allocated for non-zero values in general. If \texttt{nnz}(A) and \texttt{nzmax}(A) return different results then more storage might be allocated than is actually required. For this reason, only set \texttt{nzmax} in anticipation of later addition of non-zero values to the array.

The best way to allocate sparse matrices is to use this routine, since setting individual values is expensive in general and can force expensive reallocation of an entire sparse matrix.
• The sparse matrix data structure may seem awkward (and, as mentioned, it is not efficient for manipulating matrices one element at a time) but it is efficient for the purpose of matrix algorithms specifically designed for sparse matrices.

• In terms of matrix algebra operations sparse matrices can make an enormous difference, as is shown by examining the code in L17sparse_matrices_computations.m.
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:

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

• For a order $n$ polynomial we must solve the Vandermonde matrix problem given pairs of points $(x_i, y_i), 1 \leq i \leq n$. For $n = 3$,

$$
\begin{pmatrix}
  y_1 \\
  y_2 \\
  y_3
\end{pmatrix}
= 
\begin{bmatrix}
  x_1^3 & x_1^2 & x_1^1 & x_1^0 \\
  x_2^3 & x_2^2 & x_2^1 & x_2^0 \\
  x_3^3 & x_3^2 & x_3^1 & x_3^0
\end{bmatrix}
\begin{pmatrix}
  a_3 \\
  a_2 \\
  a_1 \\
  a_0
\end{pmatrix},
$$
while for an order 10 polynomial the Vandermonde matrix, and corresponding problem, becomes:

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

- 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 \texttt{vander} that evaluates this matrix given a vector \(x\).
• When \( x=0:0.333333:1 \) as for the above order 3 fitting, the Vandermonde matrix (via \( A=vander(x) \)) is:

\[
A= \\
\begin{bmatrix}
0 & 0 & 0 & 1.0000 \\
0.0370 & 0.1111 & 0.3333 & 1.0000 \\
0.2963 & 0.4444 & 0.6667 & 1.0000 \\
1.0000 & 1.0000 & 1.0000 & 1.0000 \\
\end{bmatrix}
\]

with a condition number \( \text{cond}(A) \) 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:

\[
A = \\
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0.0010 & 0.0100 & 0.1000 & 1.0000 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0.0003 & 0.0016 & 0.0080 & 0.0400 & 0.2000 & 1.0000 \\
0 & 0 & 0 & 0 & 0.0001 & 0.0003 & 0.0016 & 0.0041 & 0.0102 & 0.0256 & 0.0640 & 0.1600 & 0.4000 & 1.0000 \\
0.0001 & 0.0003 & 0.0007 & 0.0016 & 0.0041 & 0.0102 & 0.0256 & 0.0640 & 0.1600 & 0.4000 & 0.6000 & 0.9000 & 1.0000 \\
0.0010 & 0.0020 & 0.0039 & 0.0078 & 0.0156 & 0.0312 & 0.0625 & 0.1250 & 0.2500 & 0.5000 & 1.0000 & 1.0000 \\
0.0060 & 0.0101 & 0.0168 & 0.0280 & 0.0487 & 0.0778 & 0.1296 & 0.2160 & 0.3600 & 0.6000 & 0.9000 & 1.0000 \\
0.0282 & 0.0404 & 0.0576 & 0.0824 & 0.1176 & 0.1681 & 0.2401 & 0.3430 & 0.4900 & 0.7000 & 1.0000 & 1.0000 \\
0.1074 & 0.1342 & 0.1678 & 0.2097 & 0.2621 & 0.3277 & 0.4096 & 0.5120 & 0.6400 & 0.8000 & 1.0000 & 1.0000 \\
0.3487 & 0.3874 & 0.4305 & 0.4783 & 0.5314 & 0.5905 & 0.6561 & 0.7290 & 0.8100 & 0.9000 & 1.0000 & 1.0000 \\
1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000
\]

with a condition number \( \text{cond}(A) = 1.1558e+08 \). This is slightly unstable already but probably can be inverted reliably if we have a sufficiently large error tolerance.
• Consider the Vandermonde matrix for \( x=0:1:100 \). \( \text{cond}(\text{vander}(x)) \) returns \( 3.6872e+210 \) which is unsolvable as the matrix if very unstable and effectively is singular!

• The Vandermonde matrices 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 \( n = 6 \),
- an 8th order fit for \( n = 8 \), and finally
- a \( 10^{th} \) order 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:
Fit 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

\[
xdata=[0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0];
\]
% Only inliers in y1
\[
y1data=[-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.59 9.48 9.3 11.2];
\]
% y2 has 1 outlier at y2(9): -1.59 instead of 9.59
\[
y2data=[-0.447 1.978 3.28 6.16 7.08 7.34 7.66 -1.59 9.48 9.3 11.2];
\]

Fit using poltfit and ployval
polyfit fits the order n polynomial (n=1,2,3,6,8,10) to the y1 and y2 data via a least squares calculation
polyval computes the values p of the polynomial specified by the coefficients output pcoefs from polyfit at the values of an input vector x
pcoefs = polyfit(xdata,y1data,n);
\[
x = linspace(a,b,m);
\]
p = polyval(pcoefs,x);
p1y1=polyfit(xdata,y1data,1);
p1y2=polyfit(xdata,y2data,1);
p2y1=polyfit(xdata,y1data,2);
p2y2=polyfit(xdata,y2data,2);
p3y1=polyfit(xdata,y1data,3);
p3y2=polyfit(xdata,y2data,3);
p6y1=polyfit(xdata,y1data,6);
p6y2=polyfit(xdata,y2data,6);
p8y1=polyfit(xdata,y1data,8);
p8y2=polyfit(xdata,y2data,8);
p10y1=polyfit(xdata,y1data,10);
p10y2=polyfit(xdata,y2data,10);

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

');
fprintf('Order 1 polyfit coefficients for y2
');
fprintf('%f ',p1y2);
fprintf('

');
fprintf('Order 2 polyfit coefficients for y1
');
fprintf('%f ',p2y1);
fprintf('

');
fprintf('Order 2 polyfit coefficients for y2
');
fprintf('%f ',p2y2);
fprintf('

');
fprintf('Order 3 polyfit coefficients for y1
');
fprintf('%f ',p3y1);
fprintf('

');
fprintf('Order 3 polyfit coefficients for y2
');
fprintf('%f ',p3y2);
fprintf('

');
fprintf('Order 6 polyfit coefficients for y1
');
fprintf('%f ',p6y1);
fprintf('

');
fprintf('Order 6 polyfit coefficients for y2
');
fprintf('%f ',p6y2);
fprintf('

');
fprintf('Order 8 polyfit coefficients for y1
');
fprintf('%f ',p8y1);
fprintf('

');
fprintf('Order 8 polyfit coefficients for y2
');
fprintf('%f ',p8y2);
fprintf('

');
fprintf('Order 10 polyfit coefficients for y1
');
fprintf('%f ',p10y1);
fprintf('

');
fprintf('Order 10 polyfit coefficients for y2
');
fprintf('%f ',p10y2);
fprintf('

');
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 x as 100 evenly spaced values from 0 to 1
x=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,x);
y2order1=polyval(p1y2,x);
y1order2=polyval(p2y1,x);
y2order2=polyval(p2y2,x);
y1order3=polyval(p3y1,x);
y2order3=polyval(p3y2,x);
y1order6=polyval(p6y1,x);
y2order6=polyval(p6y2,x);
y1order8=polyval(p8y1,x);
y2order8=polyval(p8y2,x);
y1order10=polyval(p10y1,x);
y2order10=polyval(p10y2,x);
figure

% Plot inlier data polynomial fits
plot(xdata,y1data,'-ok',x,y1order1,'-r',x,y1order2,'-g',...
     x,y1order3,'-b',x,y1order6,'-y',...
     x,y1order8,'-m',x,y1order10,'-c',...
     'linewidth',1.5); % default linewidth is 0.5
xlabel('x'); ylabel('y1');
ylim([-10 20]);
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.png' -dpng

figure

% Plot outlier data polynomial fits
plot(xdata,y2data,'-ok',x,y2order1,'-r',x,y2order2,'-g',...
     x,y2order3,'-b',x,y2order6,'-y',...
     x,y2order8,'-m',x,y2order10,'-c',...
     'linewidth',1.5); % default linewidth is 0.5

% Redraw the line to emphasize where it is
hold on
plot(x,y2order1,'-r','linewidth',1.5);
xlabel('x'); ylabel('y2');
ylim([-10 20]);
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','northwest');
print 'outlier_polynomial_fits_1_2_3_6_8_10.png' -dpng
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 $y_1$
-16699.691938 70961.291095 -122810.832688 111004.082172
-55748.066128 15234.227853 -2052.393734 123.034442 -0.448106

Order 8 polyfit coefficients for $y_2$
82537.985589 -324684.563738 516760.546741 -426511.819101
194478.226872 -48204.03069 5877.413896 -242.216490 -0.400180

Order 10 polyfit coefficients for $y_1$
-474280.753982 2344109.623080 -4974920.635044 5935037.946559
-4373073.906332 2055752.526072 -612845.324163 110560.544693
-10769.509704 441.135821 -0.447000

Order 10 polyfit coefficients for $y_2$
3222809.193191 -15401922.123328 31404444.445032 -35701589.038299
24811384.427439 -10864911.362966 2975387.506548 -486744.264830
42628.299818 -1475.435607 -0.447000

- The `L17fit polynomials.m` program produces 2 plots of 7 curves for the $y_1$ and $y_2$ 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 $y^2$ outlier data.
• Some comments are in order:

1. The outlier in the $y_2$ data has a significant effect on how the curves appear versus those same curves for the $y_1$ data, which consists of inlier data only. For example, the slope of the line (order 1 polynomial) has changed significantly (though the $y$ intercept is identical). 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 $(x, y_1)$ or $(x, y_2)$ data are interpolated. However, there are two issues with this:
(i) The Vandermonde matrix used in the least squares fitting has an increasingly large condition number as the order increases. This indicates numerical instability, meaning the result is unreliable for large orders. For order 10, estimation of data between the data points can be drastically wrong! Notice also the large variation in the magnitude of the coefficients for higher order polynomials.

(ii) Using too high order polynomial overfits the data, i.e., assumes that all variations in the data are meaningful, so takes them too seriously. This is reflected in how large the variations of the higher order curves are, which stray very far from the data between data points.
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!