Matrices and Matrix Operations

- The MatLab command `A=[1 2 5; 3 9 0]` makes matrix

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 9 & 0 \end{bmatrix}.$$

- `A` is a 2D array, with the first index specifying the row and the second index specifying the column. Thus A(2,3) is the element in the 2^{nd} row, 3^{rd} column, namely 0.

- We can perform operation directly on such matrices. `exp(A)` computes the exponential of each element of `A`:

```
exp(A)

ans =

   1.0e+03 *

    0.0027    0.0074    0.1484

    0.0201    8.1031    0.0010
```

- MatLab command `B=[2*x log(x)+sin(y); 5i 3+2i]` makes matrix

$$B = \begin{bmatrix} 2x & \ln x + \sin y \\ 5i & 3 + 2i \end{bmatrix}.$$

The current x and y values are used to evaluate these expressions.

For `x=1` and `y=1` we obtain:

```
B =

   2.0000 + 0.0000i   0.8415 + 0.0000i

   0.0000 + 5.0000i   3.0000 + 2.0000i
```

Note that the first row elements are output as complex but with 0i as the imaginary parts of these numbers.

- Vectors are 1D matrices. `u=[1 3 9]` is a row vector while `v=[1;3;9]` is a column vector.

```
u=[1 2 3]

u =
```

```
      1      2      3

v =

      1

      2

      3
```

- A scalar is a vector with 1 row and 1 column. In this case we can leave the brackets off. `a=[6]` or just `a` has the value 6.
- A matrix with 0 rows and 0 columns is the null/empty matrix and is indicated by empty brackets, i.e. `X=[]`.
- If it is not possible to type an entire row on one line than the continuation periods ... can be used to indicate the input continues on the next line.

Thus the following 3 commands are equivalent:

```
A=[1 3 9;5 10 15; 0 0 -5];
A=[1 3 9
   5 10 15
   0 0 -5]
A=[1 3 9; 5 10 ...
   15; 0 0 -5];
```

- Continuation periods (...) can be used anywhere white spaces can be used. For example:

```
a = 4 + 5 + ...
     6 + 7
```

```
a =

    22
```

- Remember, elements of a matrix can be accessed by specifying their row and column indices. Then `A(i,j)` refers to the element at the i^{th} row and j^{th} column of matrix `A`.

- A range of rows and columns can be specified. For example, `A(m:n,k:l)` specified rows m to n and columns k to l. When the rows (or columns) to be specified range over all rows (or columns) we can use a `:` to specify this. Thus `A(:,5:20)` refers to columns 5 to 20 of all rows of `A`.

```
A=[ 1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
A =
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
A(2:3,2:3)
ans =
     6     7
    10    11
```

- Matrix dimensions are automatically determined by MatLab. We can use `size(A)` to find the number of rows and columns of matrix `A`. If `A` is 2D then `[m,n]=size(A)` assigns the number of rows to `m` and the number of columns to `n`. For the above matrix:

```
size(A)

ans =

     4     4
```

- Note that `size(A,1)` and `size(A,2)` return the 1^{st} and 2^{nd} dimensions of `A`.

- When a matrix is first specified, MatLab creates a matrix just big enough to accommodate it. If matrices `B` and `C` do not already exist then `B(2,3)=5;` produces:

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

while `C(3,1:3)=[1 2 3]` produces

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 3 \end{bmatrix}.$$

In both cases, undefined elements of `B` and `C` that must exist are assigned value 0.

Matrix Manipulation

- Consider the following examples:

```
% Matrices are entered row-wise. Rows are
% separated by semicolons and columns
% are separated by spaces or commas
```

```
A=[1 2 3; 4 5 6; 7 8 8]

A =

   1   2   3

   4   5   6

   7   8   8

A(2,3) % Element A(2,3) of matrix A is accessed.

ans =

    6

% Changing any entry is easy through indexing

A(3,3) = 9

A

   1   2   3
```

```
    4   5   6

    7   8   9

% Any submatrix of A is obtained by using

% range specifies for row and column indices

B=A(2:3,1:2)

B =

    4   5

    7   8

% The column by itself as a row or a column index

% specifies all rows or columns of the matrix

B=A(2:3,:)

B =
```

```
    4   5   6

    7   8   9

% A row or a column of a matrix is deleted by

% setting it to []

B(:,2)=[]

B =

    4   6

    7   9
```

- If `A` is a 10×10 matrix, `B` is a 5×10 matrix and y is a 20 element row vector then

```
A([1 3 6 9],:) = [B(1:3,:); y(1:10)]
```

replaces the 1^{st}, $3^{rd}$and 6^{th} rows of `A` by the first 3 rows of `B` and the 9^{th} row of `A` by the first 10 elements of `y`. This statement requires that matrix sizes be compatible (so B must have 10 columns and A also ends up with 10 columns).

- If

$$Q = \begin{bmatrix} 2 & 3 & 6 & 0 & 6 \\ 0 & 0 & 20 & -4 & 3 \\ 1 & 2 & 3 & 9 & 8 \\ 2 & -5 & 5 & -5 & 6 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

and $v = [1\ 4\ 5]$ then

$$Q(v;:) = \begin{bmatrix} 2 & 3 & 6 & 0 & 5 \\ 2 & -5 & 5 & -5 & 6 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

and

$$Q(:, v) = \begin{bmatrix} 2 & 0 & 6 \\ 0 & -4 & 3 \\ 1 & 9 & 8 \\ 2 & -5 & 6 \\ 5 & 20 & 25 \end{bmatrix}.$$

- In MatLab 5 and later versions, to get the 1^{st}, 4^{th} and 5^{th} rows of `Q` you can do:

$$v = logical([1\ 0\ 0\ 1\ 1]); \quad Q(v, :).$$

Reshaping Matrices

- Matrices can be reshaped as a vector: `b=A(:)` strings out the elements of `A` column-wise as a column vector `b`.

- If `A` is a $m \times n$ matrix it can be resized into a $p \times q$ matrix, as long as $m \times n = p \times q$, with the command `reshape(A,p,q)`. Thus, for a 6×6 matrix `A`, `reshape(A,9,4)` transforms `A` into a 9×4 matrix and `reshape(A,3,12)` transforms `A` into a 3×12 matrix.

Transpose of a Matrix

- The transpose of `A` is denoted as `A'`. For a real matrix `A`, `B=A'` yields $B = A^T$ and for a complex matrix `A`, `B=A'` is the complex conjugate transpose $B = \bar{A}^T$.
- If $A = \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix}$ then `B=A'` gives $B = \begin{bmatrix} 2 & 6 \\ 3 & 7 \end{bmatrix}$.
- If $C = \begin{bmatrix} 2 & 3+i \\ 6i & 7i \end{bmatrix}$ then C^T=`C'` yields $C^T = \begin{bmatrix} 2 & -6i \\ 3-i & -7i \end{bmatrix}$.

- If `u=[0 1 2 ... 9]` then `v=u(3:6)'` gives `v` as $\begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$.

Row Major versus Column Major Storage

- Most programming languages like C or C++ store their arrays in row major order.
- Matlab (and Fortran and Java) store their arrays in column major order.
- To the programmer, this makes no difference unless a large array's indices are accessed in the opposite order to the order used to store the array (lots of paging may result, slowing down execution, but you will

still get the correct result). [Note that paging is done by the operating system and hidden from the programmer.]

- Sometimes, data created in row major order by another program has to be read by MatLab (so row major order has to be converted to column major order).

- The difference between row-major order and column-major order is simply that the order of the dimensions is reversed. That is, in row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster

- In 2D, one simply needs to take the transpose of the 2D array to convert

row to column or column to row major order.

- For higher dimensions things are a bit more complicated. The ordering determines which dimensions of the array are more consecutive in memory.

- Any multi-dimensional array is really a 1D array with an memory-offset calculation based on the indices of each of its elements. You can do such a calculation in a language like C but not in MatLab.

- In row-major order, the last dimension is contiguous, so that the memory-offset of this element in a $N_1 \times N_2 \times ...N_d$ array, $k = 1, ..., d$ for element $(n_1, n_2, ...n_d)$, where $n_k \in [0, N_k - 1]$ is a zero based index, is given

by:

$$n_d+N_d(n_{d-1}+N_{d-1}(n_{d-2}+N_{d-2}(\ ...\ N_2n_1)...)) = \sum_{k=1}^{d}\left(\prod_{l=k+1}^{d} N_l\right) n_k$$

- In column-major order, the first dimension is contiguous, so that the memory-offset of this element in a $N_1 \times N_2 \times ...N_d$ array, $k = 1, ..., d$ for element $(n_1, n_2, ...n_d)$, where $n_k \in [0, N_k - 1]$ is a zero based index, is given by:

$$n_1 + N_1(n_2 + N_2(n_3 + N_3(\ ...\ N_{d-1}n_d)...)) = \sum_{k=1}^{d}\left(\prod_{l=1}^{k-1} N_l\right) n_k$$

- We can use MatLab's `reshape` and `permute` to do this conversion.

- `B=permute(A,order)` rearranges the dimensions of A so that they are in the order specified by the vector `order`. `B` has the same values as

`A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of order must be unique. For example:

```
A = [1 2; 3 4]
A =
     1     2
     3     4
permute(A,[2 1])
ans =
     1     3
     2     4
```

For 2D this is the same as the transpose of the original 2D matrix.

- Consider a 3D array with size `size_z * size_x * size_y)` generated by a C program on a SUN workstation: the data is stored in row major order and Big Endian (PCs and Macs are in Little Endian as the bytes of integers and floats are reversed). For example, bytes `1 2 3 4` in Big Endian are converted to bytes `4 3 2 1` in Little Endian.

- Consider some MatLab code to do this conversion on a 3D array:

```
pathname='/Volumes/barron/DATA3D'; % Some path
stemname='sin3D.'; % Some stem
% Open /Volumes/barron/DATA3D/sin3D.9 in Big Endian
% mode for reading. Little Endian would require
```

```
% the 'b' be changed to 'l'
% 'l'/'b' for little/big endian (32 bit)
% 'a'/'s' for little/big endian (64 bit)
% 'r' stands for raw or binary data
fd=fopen([pathname '/' stemname num2str(9)],...
         'r','b');

% Read size_z*size_x*size_y of unsigned short data
% (2 bytes): in MatLab, use datatype 'uint16'

volume=fread(fd,size_z*size_x*size_y,'uint16');
```

```
% Reshape the array to have the opposite dimensions
% and then permute that array so that the 3rd
% dimension becomes the 1st dimension, the 2nd
% dimension remains the same and the 3rd dimension
% becomes the first ==> the data is now in column
major order with the correct dimensions

volume=permute(reshape(volume,...
                [size_x size_y size_z]),[3 2 1]);
```

Initalization and Appending Rows/Columns

- An $m \times n$ matrix, `A`, can be initialized to a zero matrix by the command `A=zeros(m,n,'double')`. The initialization reserves a contiguous block of the computer's memory at `A` (in column major order) with a enough room for $m \times n$ double elements, This is necessary if loops are to be compiled by JIT (MatLab's Just In Time compiler, see below) and hence executed more efficiently.

- If the rows or columns of a matrix are computed in a loop and appended to the matrix in each iteration of the loop (definitely not efficient!!!), then you can initialize the matrix to the null matrix `A=[]` and then append a row or column of any size to `A`.

- The command `A=[A u]` appends column vector `u` to `A` while `A=[A;v]` appends the row vector `v` to the rows of the original `A`.
- If $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $u = [5\ 6\ 7]$ and $v = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$, then
  - `A=[A; u]` using the original `A` produces $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 5 & 6 & 7 \end{bmatrix}$, making `A` a 4×3 matrix.
  - `A=[A v]` using the original `A` produces $A = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \end{bmatrix}$, making `A` a 3×4 matrix.
  - `A=[A u']` (note `u'` is the transpose of `u`) using the original `A`

produces $A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 7 \end{bmatrix}$, a 3×4 matrix

- `A=[A u]` produces an error because the original `A` has 3 rows and 3 columns and `u` has 1 row and 3 columns.
- `B=[]; B=[B;1 2 3]` produces array `B=[1 2 3]` and
- `B=[];`

```
for k=1:3
B=[B; k k+1 k+2]
end
```

produces:

```
B =
     1     2     3
```

```
B =

     1     2     3

     2     3     4

B =

     1     2     3

     2     3     4

     3     4     5
```

- Any row(s) or column(s) of a matrix can be deleted by setting the row or column to the null vector. Examples:

```
u=[1 2 3 4 5 6 7 8 9]

u =

   1 2 3 4 5 6 7 8 9
```

```
length(u)

ans =

     9

% delete all elements of u except 1 to 4

u(5:length(u))=[]

u =

     1  2  3  4

length(u)

ans =

     4

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

A=[1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15]
```

```
A =

     1     2     3     4     5

     6     7     8     9    10

    11    12    13    14    15

% delete the 2nd row of A

A(2,:)=[]

A =

     1     2     3     4     5

    11    12    13    14    15

% delete the 3rd through 5th columns of A

A(:,3:5)=[]

A =
```

```
     1     2
    11    12
```

Utility Matrices

- Some MatLab utility matrices:

```
% returns a m by n matrix with ones on the main diagonal
eye(m,n)
% returns a m by n matrix of zeros
zeros(m,n)
% returns a m by n matrix of ones
ones(m,n)
% returns a m by n matrix of random numbers
```

```
rand(m,n)
% returns a m by n matrix of normally distributed numbers
randn(m,n)
% generate a square n by n matrix where the sum of rows,
% columns and diagonals is the same
magic(n)
% generates a diagonal matrix with vector v
% on the diagonal
diag(v)
% extracts the first diagonal of matrix A as a vector
diag(A)
% extracts the 1st upper off-diagonal vector of A
```

```
diag(A,1)
```

- Some functions that can be used in matrix manipulation:

```
% rotate a matrix by 90 degrees
rot90
% flip a matrix from left to right
fliplr
% flip a matrix from up to down
flipud
% extract the lower triangular part of a matrix
tril
% extract the upper triangular part of a matrix
triu
```

```
% change the shape of a matrix: the number of
% elements in the changed matrix must be the same
reshape
```

- Examples of matrix manipulation using utility matrices and functions:

```
% create a 3*3 identity matrix. Commands zeros, ones
% and rand work in a similar way
eye(3) % assumes a square matrix
ans =
    1 0 0
    0 1 0
    0 0 1
% eye for a rectangular matrix
```

```
eye(2,3)

ans =

     1     0     0

     0     1     0

% Create matrix B using submatices: ones,

% zeros and the identity matrix of specified sizes

B=[ones(3) zeros(3,2); zeros(2,3) 4*eye(2)]

B =

   1 1 1 0 0

   1 1 1 0 0

   1 1 1 0 0

   0 0 0 4 0
```

```
   0  0  0  0  4


% Pull out the diagonal of B in a row vector.
% Without the transpose operation it would be a
% column vector
diag(B)'
ans =
    1   1   1   4   4


% Transpose the first upper diagonal vector of B
% If you use a  negative values for the 2nd argument
% you get the first lower off-diagonal vector
```

```
diag(B,-1)'

ans =

    1   1   0   0

% positive numbers for the 2nd argument give

% first upper off-diagonal vector

diag(B,1)'

ans =

    1   1   0   0

% Create D by putting vector d on the main diagonal,

% vector d1 on the first upper diagonal and vector

% d2 on the second lower diagonal with all other

% elements being zeros. Note that d, d1 and d2 must
```

```
% be the right sizes, otherwise you get an error.

d=[2 4 6 8];

d1=[-3 -3 -3];

d2=[-1 -1];

D=diag(d)+diag(d1,1)+diag(d2,-2)
```

```
D =

   2    -3    0    0

   0     4   -3    0

  -1     0    6   -3

   0    -1    0    8

% Values of diag(d), diag(d1,1) and diag(d2,2)

diag(d)

ans =

      2      0      0      0

      0      4      0      0

      0      0      6      0

      0      0      0      8
```

```
diag(d1,1)

ans =

     0    -3     0     0

     0     0    -3     0

     0     0     0    -3

     0     0     0     0

diag(d2,2)

ans =

     0     0    -1     0

     0     0     0    -1

     0     0     0     0

     0     0     0     0
```

Clearly, `diag(d)+diag(d1,1)+diag(d2,-2)` add the corresponding elements of these 3 matrices.

- The general MatLab command to creat a matrix of numbers over a given range with a specified increment is:

```
v = InitialValue: Step/Increment: FinalValue
```

- For example:

```
% produce a=[0 10 20 30 ... 100] (step 10),
a=0:10:100
a =
   0 10 20 30 40 50 60 70 80 90 100
% produce b=[0 pi/50 2*pi/50 ... pi/4], a linearly
```

```
% spaced vector from 0 to pi/4 spaced by step pi/50
b=0:pi/50:pi/4
b =
  0         0.0628    0.1257    0.1885    0.2513
  0.3142    0.3770    0.4398    0.5027    0.5655
  0.6283    0.6912    0.7540
% produce a=[-2 -1 0 1 2 3 4 ... 10] (step 1)
a=-2:10
a =
 -2 -1 0 1 2 3 4 5 6 7 8 9 10
```

- Square brackets are only required if vector concatenation is required. `u=[1:10 33:-2:23]` requires square brackets to concatenate 2 vec-

tors `[1 2 3 ... 10]` and `[33 31 29 ... 23]`

```
u=[1:10 33:-2:23]
u =
  1 2 3 4 5 6 7 8 9 10 33 31 29 27 25 23
```

- `linspace(a,b,n)` generates a linearly spaced vector of length `n` from `a` to `b`. `u=linspace(0,20,5)` generates `u=[0 5 10 15 20]`.
- In general, `u=linspace(a,b,n)` is the same as `u=a:(b-a)/(n-1):b` but note that we **don't** need to know the `step` or `increment (b-a)/(n-1)` when using `linspace`.
- So `u=linspace(0,20,5)` is the same as `a=0:5:20`, i.e. `(b-a)/(n-1)` is `(20-0)/(5-1)=5`.

- `logspace(a,b,n)` generates a logarithmically spaced vector of length `n` from `10^a` to `10^b`. Thus `v=logspace(0,3,4)` generates the vector `v=[1 10 100 1000]`. Hence `logspace(a,b,n)` is the same as `10.^(linspace(a,b,n))`.

```
logspace(0,3,4)
ans =
           1          10         100        1000
linspace(0,3,4)
ans =
     0     1     2     3
10.^(linspace(0,3,4))
ans =
           1          10         100        1000
```

Matrix Multiplication

- Matrix multiplication is `C=A*B` where `A` is a $m \times n$ matrix and `B` is a $n \times k$ matrix. `C` is a $m \times k$ matrix. So the number of columns of `A` must equal the number of rows of `B`. `C=A*B` is equivalent to the multiplication performed in the nested loop below :

```
% Matrix Multiplication
% Multiply A_m,k by B_k,n (#columns of A=#rows of B)
% to obtain matrix C_m,n
A=[1 2 3 4 5; 6 7 8 9 10]; % A_2,5
B=[1 2 3; 4 5 6; 7 8 9; 10 11 12; 13 14 15]; % B_5,3
[m,p]=size(A)
```

```
m =

     2

p =

     5

[p,n]=size(B)

p =

     5

n =

     3

for i=1:m

for j=1:n

   C(i,j)=0;
```

```
    for k=1:p
      C(i,j)=C(i,j)+A(i,k)*B(k,j);
    end
end
end
A
A =
     1     2     3     4     5
     6     7     8     9    10
B


B =
```

```
     1     2     3
     4     5     6
     7     8     9
    10    11    12
    13    14    15


C % Matrix Multiplication by Nested Loops
C =
   135   150   165
   310   350   390
C=A*B % MatLab Matrix Multiplication
C =
```

```
   135   150   165
   310   350   390
% Note what happens when the number of columns
% of the first matrix does not equal the number
% of rows of the second matrix, as in B*A
B*A
Error using  *
Inner matrix dimensions must agree.
```

- `A+B` or `A-B` add or subtract matrices `A` and `B` by adding or subtracting the corresponding elements of `A` or `B`, if `A` and `B` are the same size.

```
A=[1 2; 3 4];
```

```
B=[4 3; 2 1];
A+B
ans =
     5     5
     5     5
A-B
ans =
    -3    -1
     1     3
```

- `A/B` is valid for same sized matrices and equals AB^{-1}

```
A/B
```

```
ans =

    1.5000   -2.5000

    2.5000   -3.5000
```

- `A^2` is `A*A` makes sense if `A` is square.

```
A^2

A =

     1     2

     3     4
```

- If α is a scalar then $A+\alpha$ or $A-\alpha$ add or subtracts α from each element of A.

```
A =[ 1 2; 3 4];
```

```
A-3

ans =

    -2    -1

     0     1
```

- Similarly, $A * \alpha$ (or $\alpha * A$) multiplies each element of A by α.

```
A*3

ans =

     3     6

     9    12
```

- Vectors are single row/column matrices. If u and v are n sized vectors ($n \times 1$ matrices) then $u * v$ produces an error but $u * v'$ or $u' * v$ produce

the outer and inner products of the two vectors.

```
u=[1 2 3];
v=[4 5 6];
u*v
Error using  *
Inner matrix dimensions must agree.
u*v'
ans =
    32
u'*v
ans =
     4     5     6
```

```
     8    10    12
    12    15    18
```

Note that `u'*v` is a **singular** matrix as any row or column is any other row or column scaled by a constant factor. The rows and columns are not **linearly independent** of each other. For example, 2 times the 1^{st} row equals the 3^{rd} row and 1.5 time the 1^{st} column equals the 3^{rd} column.

- In addition to normal or right division (/) there is left division (\) in MatLab. The matrix equation `As=B` has solution `s=A\B`. Thus `A\B` is almost the same as `inv(A)*B` but faster and more numerically stable. The next section explains solving linear systems of equations, which is what `s=A\B` does!

- In the case of scalars, `5\3` is 0.6, which is 3/5 or $3 * 5^{-1}$.

Solving Linear Systems of Equations

- Consider the solution for the unknowns, `x`, `y` and `z` for the following 3 equations:

$$\begin{aligned} 3x + 4y - 7z &= 16 \\ 9x + 2y + 102z &= 1002 \\ -6x * 4y + z &= 14 \end{aligned}$$

This is 3 linear equations in 3 unknowns (and usually has a unique solution). These 3 equations can be written in matrix form as:

$$\begin{pmatrix} 3 & 4 & -7 \\ 9 & 2 & 102 \\ -6 & 4 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{pmatrix} 16 \\ 1002 \\ 14 \end{pmatrix}.$$

We can write this as $A\vec{s} = B$ where $\vec{s} = (x, y, z)$.

```
A = [3 4 -7; 9 2 102; -6 4 1];
B = [16 ; 1002; 14];
s=A\B
s =
   8.0862e+00
   1.3418e+01
   8.8470e+00
```

$\vec{s}$ satisfies all 3 equations simultaneously:

```
% 1st equation
s(1)*3+s(2)*4+s(3)*-7-16
ans =
  -2.8422e-14
% 2nd equation
s(1)*9+s(2)*2+s(3)*102-1002
ans =
     0
% 3rd equation
s(1)*-6+s(2)*4+s(3)*1-14
ans =
```

```
  -1.7764e-14
% The residual vector is 0 within roundoff error
r=A*s-B
r =
 1.0e-13 *
   -0.2842
         0
   -0.1776
% The norm of the residual is computed as
% the length of the vector r
% Called Euclidean norm or L2 norm
% sqrt(r(1)*r(1)+r(2)*r(2)+r(3)*r(3))
```

```
sqrt(r(1)*r(1)+r(2)*r(2)+r(3)*r(3))
ans =
   3.3516e-14
% Built-in Matlab function norm computes this
norm(r)
ans =
   3.3516e-14
```

Least Squares

- Lets add two 2 rows to `A` (we also need to add 2 more rows to `B`). These numbers were just arbitrarily chosen! Then when we solve $As = B$ we are finding the `s` vector that "best" fits the data. This is the **least squares**

solution to an over constrained linear system of equations.

```
A = [3 4 -7; 9 2 102; -6 4 1; 2 2 2; 4 -19 6];
B = [16 ; 1002; 14; 20; 0 ];
s=A\B
s =
    6.5426
    4.8799
    9.1282
% Residual
r=A*s-B
r =
  -40.7504
```

```
    -2.2763
   -24.6078
    21.1014
   -11.7780
norm(r)
ans =
    53.4351
```

We can see the residual vector and its norm are not that close to 0 now: the solution vector `s` is not a perfect fit to the system of equations now.

Vectorization versus Serialization

- How does one compute a dot product of two vectors, which requires element by element multiplication and summing? Element by element multiplication, division and exponentiation between 2 matrices of the same size is done by preceding the corresponding operator by a period:

```
.*    %  element by element multiplication
./    %  element by element left division
.\    %  element by element right division
.^    %  element by element exponentiation
.'    %  non-conjugated transpose
```

This is called **vectorization** and is a powerful MatLab tool.

- Consider `A.*B`. This is equivalent to

```
% The size of A and B must be the same in
% each dimension
for i=1:size(A,1)
for j=1:size(A,2)
    C(i,j)=A(i,j)*B(i,j)
end % for j
end % for i
```

This nested loop is the **serialization** of matrix element by element multiplication. **Just In Time** (JIT) compilation of this serialized code can produce code that executes almost as fast as the vectorized version of the

code.

JIT - Just In Time Compiler

- MatLab is an **interpreted** language in that the program source code directly executed, statement by statement.
- **Compiled** languages require that the code first be explicitly translated into a lower-level machine language executable and then this is executed.
- Usually compiled code runs much faster than interpreted code but interpreted code is considerably more flexible. For example, MatLab lets array sizes change in loops!
- In the early days, interpretation was line by line but these days many

interpreted languages, such as Java or Python, use an intermediate representation, which combines compiling and interpreting. In this case, a compiler may output some form of bytecode or threaded code, which is then executed by a bytecode interpreter.

- JIT (or Just In Time) compilation evaluates loop code the first time the code is executed to see if it is suitable for compilation: if it is bytecode is produced, which usually runs much faster.

- MatLab code containing loops may benefit from JIT acceleration if the code has the following properties:

    1. The loop is a For Loop.

    2. The loop contains only logical, character, double precision vari-

ables.

3. The loop uses arrays that are 3D or less.
4. All variables in the loop are defined prior to loop execution.
5. Memory for all variables maintain constant size and type during loop execution.
6. Loop indices are scalar quantities.
7. Only built-in MatLab functions are called.
8. Conditional statements (**if-then-else** and **switch-case**) involve only scalar comparisons.
9. Each line within the loop contain no more than 1 assignment statement.

- An example: compute $\sin(x)$ at 10^7 points.

```
N=1e7;
% generate sin(x) at 1e7 points - vectorized solution
disp('Vectorized solution time:');
tic
x=linspace(0,2*pi,N);
y=sin(x);
elapsed_time1=toc

% generate sin(x) at 1e7 points - JIT solution
disp('JIT solution time:');
tic
```

```
i=0; % define variable i before the loop
% allocate space for x and initialize as 0's
x=zeros(1,N);
% allocate space for y and initialize as 0's
y=zeros(1,N);
for i=1:N % scalar loop index
   y(i)=sin(2*pi*(i-1)/N);  % built-in fct sin
end
elapsed_time2=toc

disp('Vectorization speed up:');
elapsed_time2/elapsed_time1
```

produces the output:

Machine	Vectorized Time	JIT Time	Speedup Speedup
SUN 64bit (brown) MatLab 5 (1999)	5.1008	68.1895	13.3683
SUN 64bit (mccarthy) MatLab 2009b	1.0018	5.8502	5.8376
MAC 2.16GHz 32bit, Intel Core Duo , MatLab 2010a, 2003	1.1854	1.2942	1.0919
MAC 3.06GHz 64 bit, Intel dual core, MatLab 2013a, 2009	0.3305	1.4421	4.3334
MAC 2.70HHz 64 bit, Intel quad core, MatLab 2013a, 2012	0.1495	0.4133	2.7651

- Vectorized code is usually superior to nested loops (even when JIT is used). JIT did not exist for MatLab 5!!!

- Both the JIT and vectorized solutions take about the same time on the older machine Mac notebook but on the newer machines (with later versions of MatLab) vectorization is definitely faster. Before MatLab 6.5, the **for** Loop would have taken an order of magnitude more time to execute (see the MatLab 5 result).

- The 2009 and 2012 machines are faster because of the multi-core architectures.

- Here, JIT is not helpful because vectorizing the solution is so simple. In more substantial problems, it may be difficult or impossible to derive a vectorized solution. In this case, JIT acceleration would be very useful!!!

- MatLab has a `feature` command: `feature('accel','on')` or `feature('accel','off')` will turn JIT compilation on or off explicitly. Note that the loops have to be compilable.

More on Vectorization

- For vectors `u` and `v`, `u.*v` produces $[u_1v_1 \;\; u_2v_2 \;\; u_3v_3...]$,

```
u=[1 2 3];
v=[4 5 6];
u.*v
ans =
     4    10    18
```

`u./v` produces $[u1/v1 \;\; u2/v2 \;\; u3/v3...]$ while `u.^v` produces $[{u_1}^{v_1} \;\; {u_2}^{v_2} \;\; {u_3}^{v_3}...]$.

```
u./v
ans =
    0.2500    0.4000    0.5000
u.^v
```

```
ans =

     1    32   729
```

- For same sized matrices A and B, $C = A.*B$ produces a matrix with elements $C_{i,j} = A_{i,j} * B_{i,j}$. `A^2` is matrix multiplication $A * A$ while `A.*2` computes $A_{i,j} * 2$.

```
A=[1 2; 3 4];
A*A
ans =
     7    10
    15    22
A.*A
ans =
```

```
     1     4
     9    16
```

- For scalars, `1./v` computes $[1/v_1, 1/v_2, 1/v_3 ...]$ and `pi.^v` computes $[\pi^{v_1}\ \pi^{v_2}\ \pi^{v_3} ...]$.

```
v=[4 5 6];
1./v
ans =
    0.2500    0.2000    0.1667
```

- Some more examples:

```
A=[1 2 3; 4 5 6; 7 8 9];
```

```
x=A(1,:)'

x =

   1

   2

   3

% Here x is a column vector

% and x' is a row vector

x'*x   % inner or dot product

ans =

    14

x*x'   % outer product

ans =
```

```
     1   2   3
     2   4   6
     3   6   9
A*x     % matrix multiplied by a vector
ans =
    14
    32
    50
A^2     % A is square and A^2 is A*A
ans =
    30   36   42
    66   81   96
```

```
    102 126 150

A.^2   % elements of A are raised to the power of 2

ans =

     1   4   9

    16  25  36

    49  64  81
```