Advanced
MATLAB

Christos Xenophontos
Department of Mathematics & Statistics
University of Cyprus
NOTES

The notes are separated into ‘topics’, as described by their title in bold capitals. These notes are not meant to be a complete treatise on the subject, but rather a collection of useful information for making your MATLAB experience more efficient. They are arranged in order, with the next topic relying possibly on the previous one(s).

PREALLOCATION:

In order to avoid variables growing inside a loop, we pre-allocate:

For example:
```matlab
a = zeros(1, 100);
for n=1:100
    res = % Very complex calculation %
    a(n) = res;
end
> Variable a is only assigned new values. No new memory is allocated
```

Usually, in a program this amounts to initializing the variables.

VECTORIZATION:

The most time-consuming operation in MATLAB is performing loops. MATLAB offers vectorization capabilities that allow us to avoid using loops. For example, consider

```matlab
a=rand(1,100);
b=zeros(1,100);
for n=1:100
    if n==1
        b(n)=a(n);
    else
        b(n)=a(n-1)+a(n);
    end
end
```

which adds consecutive terms in a random array of 100 entries. A much more efficient way of doing the same thing is

```matlab
a=rand(1,100);
b=[0 a(1:end-1)]+a;
```

The command `find` is quite useful when avoiding loops. It basically returns indices of non-zero values and it can simplify code. For example,
As another example, let $x = \sin(\text{linspace}(0,10\pi,100))$. How many of the entries are positive?

Using a loop and if/else

```matlab
count=0;
for n=1:length(x)
    if x(n)>0
        count=count+1;
    end
end
```

Being more clever

```matlab
count=length(find(x>0));
```

Is there a better way?!

<table>
<thead>
<tr>
<th>length(x)</th>
<th>Loop time</th>
<th>Find time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.1</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.5</td>
<td>0.04</td>
</tr>
</tbody>
</table>
This code computes the sine of 1,001 values ranging from 0 to 1:

\[
\begin{align*}
    i &= 0; \\
    \text{for } t &= 0:.01:10 \\
    \quad i &= i + 1; \\
    \quad y(i) &= \sin(t); \\
    \text{end}
\end{align*}
\]

This is a vectorized version of the same code:

\[
\begin{align*}
    t &= 0:.01:10; \\
    y &= \sin(t);
\end{align*}
\]

This code computes the cumulative sum of a vector at every fifth element:

\[
\begin{align*}
    x &= 1:10000; \\
    \text{ylength} &= \text{(length}(x) - \text{mod}(\text{length}(x),5))/5; \\
    y(1:ylength) &= 0; \\
    \text{for } n &= 5:5:\text{length}(x) \\
    \quad y(n/5) &= \text{sum}(x(1:n)); \\
    \text{end}
\end{align*}
\]

Using vectorization, you can write a much more concise MATLAB process. This code shows one way to accomplish the task:

\[
\begin{align*}
    x &= 1:10000; \\
    xsum = \text{cumsum}(x); \\
    y &= xsum(5:5:\text{length}(x));
\end{align*}
\]

*Remember:* avoid loops, whenever possible.

**SPARSE MATRICES**

When a matrix contains mostly 0’s, the use of the sparse matrix capabilities of MATLAB is recommended. For example,

\[
\begin{align*}
    \text{M_full} &= \text{magic}(1100); & \text{Create 1100-by-1100 'magic' matrix.} \\
    \text{M_full}(\text{M_full} > 50) &= 0; & \text{Set elements >50 to zero.} \\
    \text{M_sparse} &= \text{sparse}(\text{M_full}); & \text{Create sparse matrix of same.}
\end{align*}
\]

\[
\begin{array}{cccc}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} & \text{Attributes} \\
\hline
\text{M_full} & 1100x1100 & 9680000 & \text{double} & \text{sparse} \\
\text{M_sparse} & 1100x1100 & 9608 & \text{double} & \text{sparse}
\end{array}
\]

So a sparse matrix uses less resources and it should be used whenever possible, especially for very large matrices.

The basic syntax is:
\[ S = \text{sparse}(m, n) \rightarrow \text{creates an } m \times n \text{ zero sparse matrix} \]

\[ S = \text{sparse}(A) \rightarrow \text{converts full matrix into sparse form} \]

(See help sparse for more uses.)

As an example, we consider the matrix

\[
D = \begin{bmatrix}
4 & -1 & \cdots & 0 \\
-1 & 4 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & -1 \\
-1 & 0 & \cdots & 4
\end{bmatrix} \in \mathbb{R}^{n \times n}.
\]

which appears in the discretization of differential equations. The size \( n \) of \( D \) depends on the discretization parameters, but it is usually large, e.g. \( O(10^5) \). Let's see how to take advantage of MATLAB's sparse capabilities and define the matrix \( D \) in an efficient way, for say \( n = 10000 \). The number of non-zero elements in \( D \) is given by \( n + 2(n - 1) = 3n - 2 \). We then 'allocate memory' for an \( n \times n \) sparse matrix with (at most) \( 3n - 2 \) non-zero entries as follows:

\[
\begin{align*}
&> n=10000; \\
&> D = \text{spalloc}(n,n,3*n-2);
\end{align*}
\]

Type help spalloc for more information.

Next, we insert the non-zero elements using, well, a loop:

\[
\begin{align*}
&> D(1,1) = 1; \\
&> D(n,n) = 1; \\
&> \text{for } i=2:n-1 \\
&\quad \quad D(i,i) = 2; \\
&\quad \quad D(i,i-1) = -1; \\
&\quad \quad D(i,i+1) = -1; \\
&\text{end}
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} & \text{Attributes} \\
\hline
D & 10000x10000 & 559976 & \text{double} & \text{sparse} \\
\hline
\end{array}
\]

If the matrix was not defined as sparse, then the above result would have been

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} & \text{Attributes} \\
\hline
D & 10000x10000 & 800000000 & \text{double} & \\
\hline
\end{array}
\]

As another example, we consider the matrix
\[
A = \begin{bmatrix}
D & -I \\
-1 & D \\
& & \ddots \\
D & -1 \\
-1 & 4 \\
& & & \ddots \\
& & & & \ddots \\
& & & & & -1 \\
-1 & 4 \\
& & & & & & -1 \\
\end{bmatrix} \in \mathbb{R}^{2n \times 2n},
\]

where
\[
I = \begin{bmatrix}
1 & \cdots & 1 \\
& & \\
& & \\
& & \\
& & \\
& & \\
\end{bmatrix} \in \mathbb{R}^{n \times n},
D = \begin{bmatrix}
4 & -1 \\
-1 & 4 \\
& & \ddots \\
& & & \ddots \\
& & & & -1 \\
-1 & 4 \\
& & & & & -1 \\
\end{bmatrix} \in \mathbb{R}^{n \times n}.
\]

This matrix appears in the discretization of partial differential equations. First, we note that
\[
A = \begin{bmatrix}
B & 0 \\
0 & B \\
\end{bmatrix}, \quad \text{where } B = \begin{bmatrix}
D & -I \\
-1 & D \\
\end{bmatrix}.
\]

Suppose the size of \(I\) and \(D\) is \(n \times n\) and of \(A\) is \(2mn \times 2mn\), for some \(m\) and \(n\). Using the matrix \(D\) we have already defined, we have

```
>> n = 10; m = 4; A = [];
>> I = eye(n);
>> B = [D,-I;-I,D];
>> for i=1:m, A = blkdiag(A,B); end
>> whos A
Name              Size              Bytes  Class       Attributes
A     80000x80000          5759496 double     sparse
>> spy(A)
```

![Spy Plot](image)
When a sparse matrix is used in the solution of a linear system \( Ax = b \), using the ‘backslash’ command, \( x = A \backslash b \), MATLAB detects and takes advantage of it, by using the most appropriate linear solver.

**LOGICAL ARRAYS**

A *logical array* of 1 (true) and 0 (false) values is returned as a result of applying logical operators to arrays, e.g.,

```matlab
>> a = [4 0 -2 7 0]
a =
   4     0    -2     7     0
>> a > 0
ans =
   1×5 logical array
     1   0   0   1   0
>> a == 7
ans =
   1×5 logical array
     0   0   0   1   0
>> a ~= 0
ans =
   1×5 logical array
     1   0   1   1   0
>> (a >= 0) & (a <= 4)
ans =
   1×5 logical array
     1   0   0   0   1
>> (a < 0) | (a > 4)
ans =
   1×5 logical array
     0   0   1   1   0
>> ~((a < 0) | (a > 4))
```
A logical array may be used just like an index array to select and change the elements of an array.

e.g.,
```matlab
>> a(a>0)
ans =
4    7
>> a(a == 7) = 8
a =
4    0   -2    8    0
>> a(a ~= 0) = a(a ~= 0) + 1
a =
5    0   -1    9    0
```

MULTIDIMENSIONAL ARRAYS

We are mostly used to working with two-dimensional arrays (or matrices), but MATLAB allows us to define arrays whose entries are arrays themselves, thus creating a multi-dimensional array.

Every value and variable is an array and has a size:

```matlab
>> a = 3;
>> x = [ 5 6 7 ];
>> A = [ 1 2 3; 4 5 6 ];
>> whos
Name        Size         Bytes    Class Attributes
a           1x1            8    double
x           1x3            24    double
A           2x3           48    double
```

A scalar is a 0-dimensional array, a vector is a 1-dimensional array and a matrix is a 2-dimensional array … conceptually … in practice they are all two-dimensional.

Array concatenation:
Universal function

\[ \text{cat}(\text{dimension}, \text{argument}_1, \ldots, \text{argument}_N) \]

- **horizontal:**
  ```matlab
  >> [a, a]
  ans =
      3   3
  >> horzcat(a, a, a)
  ans =
      3   3   3
  ```

- **vertical:**
  ```matlab
  >> [x; x]
  ans =
      5   6   7
      5   6   7
  >> vertcat(x, x, x)
  ans =
      5   6   7
      5   6   7
      5   6   7
  ```

If we use 3 for the dimension, then we construct a 3-dimensional array

```matlab
>> X = cat(3, A, 2*A);
```
It has three indices:

```matlab
>> X
X(:,:,1) =
    1    2    3
    4    5    6
X(:,:,2) =
    2    4    6
    8   10   12
```

- X is now a 3-dimensional array:

  ```matlab
  >> ndims(X)
  ans =
        3
  ```

  ```matlab
  >> size(X)
  ans =
        2    3    2
  ```

  ```matlab
  >> whos X
  Name    Size        Bytes  Class   Attributes
  X       2x3x2        96  double
  ```

- arrays can have an arbitrarily high number of dimensions

3D Array

**shape**: (4, 3, 2)
CELLS AND CELL ARRAYS

A cell is a universal type that can hold anything:

```matlab
>> a = { 5 }
a =
    [5]
>> b = { [ 1 2; 3 4] }
b =
    [2x2 double]
>> whos
    Name      Size Bytes Class Attributes
    a    1x1     68 cell
    b    1x1     92 cell
```

- a is a single cell (1x1 cell) holding a scalar (1x1 double)
- b is a single cell (1x1 cell) holding a matrix (2x2 double)

We can also have cell arrays:

```matlab
>> x = { 1, 2, 3, 4 }
x =
    [1]    [2]    [3]    [4]
>> A = { 1 2 3; 4 5 6 }
A =
    [1]    [2]    [3]
>> whos A x
    Name      Size Bytes Class Attributes
    A    2x3    408 cell
    x    1x4    272 cell
```

1 by 4 vector and 2 by 3 matrix, each cell containing a scalar.

- it can contain anything, really:
  ```matlab
  >> y = { 12, [4 5 6], [1 2; 3 4], true, 'hi' }
  y =
    [12]    [1x3 double]    [2x2 double]    [1]    'hi'
  >> whos y
    Name      Size Bytes Class Attributes
    y    1x5    369 cell
  ```

- even cell arrays:
  ```matlab
  >> z = { 4 { 5 6 } }
  z =
    [4]    {1x2 cell}
  ```
- `cell` function creates an empty cell array:
  ```matlab
  >> cell(2)
  ans =
      []   []
      []   []
  >> cell(2, 4)
  ans =
      []   []   []   []
      []   []   []   []
  >> cell([1, 5])
  ans =
      []   []   []   []   []
  ```
  - it behaves exactly like `zeros` function

- example:
  ```matlab
  >> y = { 12; [4 5 6]; [1 2; 3 4], true, 'hi' }
  ```

- we can access elements by indexing with `{ and }`:
  ```matlab
  >> y{1}
  ans =
      12
  >> y{2}
  ans =
      4   5   6
  >> y{4}
  ans =
      1
  >> y{5}
  ans =
  hi
  ```

- actually, we can also use indexing with `( and )`:
  ```matlab
  >> A = y(3)
  A =
      2x2 double
  >> whos A
  Name      Size          Bytes  Class      Attributes
  A          1x1            92  cell
  ```
  - but what we get is a cell
  - indexing with `{ and }` returns the content of the cell
- cell arrays are just arrays, so we can slice:
  \[
  \begin{align*}
  \text{\texttt{X}} &= \{ 1 \ [2 \ 3] \ [4 \ 5 \ 6; \ 7 \ 8 \ 9] \ 0 \} \\
  \text{\texttt{X(1:2)}} &= \begin{bmatrix} 1 & \text{1x2 double} \end{bmatrix} \begin{bmatrix} 2x3 \text{ double} \end{bmatrix} \ [0] \\
  \text{\texttt{X(3)}} &= \begin{bmatrix} \text{1x2 double} \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \ [0]
  \end{align*}
  \]

- and concatenate:
  \[
  \begin{align*}
  \text{\texttt{Y}} &= \begin{bmatrix} \text{X(1:2)} \text{X(4)} \end{bmatrix} \\
  \text{\texttt{Y(1)}} &= \begin{bmatrix} 1 & \text{1x2 double} \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix} \\
  \text{\texttt{Y(2)}} &= \begin{bmatrix} 1 & \text{1x2 double} \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix}
  \end{align*}
  \]

- and delete:
  \[
  \begin{align*}
  \text{\texttt{X(3)}} &= \begin{bmatrix} \end{bmatrix} \\
  \text{\texttt{X}} &= \begin{bmatrix} \end{bmatrix}
  \end{align*}
  \]

- \texttt{celldisp} describes a cell array:
  \[
  \begin{align*}
  \text{\texttt{C}} &= \{ 'Hello' \ 15; [ 1 \ 2; 3 \ 4] \ 3 \} ; \\
  \text{\texttt{C(2,3)}} &= \text{true}; \\
  \text{\texttt{celldisp(C)}} &= \\
  \text{\texttt{C(1,1)}} &= \text{Hello} \\
  \text{\texttt{C(2,1)}} &= \begin{bmatrix} 1 & 2 \\
 3 & 4 \end{bmatrix} \\
  \text{\texttt{C(1,2)}} &= 15 \\
  \text{\texttt{C(2,2)(1)}} &= 3 \\
  \text{\texttt{C(1,3)}} &= [] \\
  \text{\texttt{C(2,3)}} &= 1
  \end{align*}
  \]

- \texttt{cellplot} displays it graphically (\texttt{'legend'} is optional):
  \[
  \begin{align*}
  \text{\texttt{cellplot(C, \ 'legend')}}
  \end{align*}
  \]
There are also the commands: `cell2mat`, `mat2cell`, `num2cell`.

```matlab
>> help cell2mat

cell2mat

Convert the contents of a cell array into a single matrix.

M = cell2mat(C) converts a multidimensional cell array with contents of the same data type into a single matrix. The contents of C must be able to concatenate into a hyperrectangle. Moreover, for each pair of neighboring cells, the dimensions of the cell's contents must match, excluding the dimension in which the cells are neighbors. This constraint must hold true for neighboring cells along all of the cell array's dimensions.

The dimensionality of M, i.e. the number of dimensions of M, will match the highest dimensionality contained in the cell array.

cell2mat is not supported for cell arrays containing cell arrays or objects.

Example:

C = {
    [1] [2 3 4]; [5; 9] [6 7 8; 10 11 12];
}
M = cell2mat(C)

See also mat2cell, num2cell
```

As an example, let us consider the following: we define a cell C which contains the name, age and networth of, say 5 individuals.

```matlab
>> C = {'Lucy', 7, 45000; 
'Sally', 18, 150000; 
'Martha', 64, 75000; 
'Bob', 47, 200000; 
'Ed', 36, 0};

The following command gives the average net worth of all individuals:

```matlab
>> mean(cell2mat(C(:,3)))
ans =
    94000
```

The following one gives the number of individuals who are above 20 years old and have net worth exceeding €50000:
>> length(find(cell2mat(C(:,2))>20 & cell2mat(C(:,3))>50000))

ans =
2

(Note that the size of the class is irrelevant in the above commands.)

The most common use of a cell array is to hold lines of text. If they were all the same number of characters they would fit in a char array; if not, we put each line in a separate cell, eg:

>> a = dir('m*')

a =
14×1 struct array with fields:
    name    folder    date    bytes    isdir    datenum

>> names = {a.name}'

names =
14×1 cell array
    {'markov.mat'    }
    {'markus.m'      }
    {'matlab.mat'    }
    {'max_hpmixed.eps'}
    {'mazas.m'       }
    {'mesh_graded.eps'}
    {'mesh_nongraded.eps'}
    {'mich.m'        }
    {'minimax.m'     }
    {'mixedhp4th.mat' }
    {'movingBL.m'    }
    {'mps1.asv'      }
    {'mps1.m'        }
    {'myclock.m'     }

As a final example of the use of cells, we mention the capability of storing the various data obtained by, e.g. the command besselj:
>> help besselj

besselj Bessel function of the first kind.

J = besselj(NU,Z) is the Bessel function of the first kind, J_nu(Z).
The order NU need not be an integer, but must be real.
The argument Z can be complex. The result is real where Z is positive.

J = besselj(NU,Z,SCALE) returns a scaled J_nu(Z) specified by SCALE:
0 - (default) is the same as besselj(NU,Z)
1 - scales J_nu(Z) by exp(-abs(imag(Z)))

Class support for inputs NU and Z:
float: double, single

See also airy, besselh, besseli, besselk, bessely.

It basically evaluates the Bessel function \( J_\nu(z) \), where \( \nu \) is a number and \( z \) is the variable.

Suppose we want to get a handle on the first \( n \) Bessel functions, so that we could perhaps plot them etc. If \( n \) is small, we may do this manually, but if \( n \) is large then the use of a cell could prove efficient. Let us take \( n = 10 \), and define the (empty) cell \( C \):

```matlab
>> n = 10;
>> C = cell(n,1);
```

We also define a vector of 1001 points in, say \([0, 20]\):

```matlab
>> z=linspace(0,20,1001);
```

Then we place in each position in \( C \), one of the Bessel functions \( J_\nu(z) \), as follows:

```matlab
>> for i=1:n
    C{i} = besselj(i,z);
end
```

```
C =
10x1 cell array
{1x1001 double}
{1x1001 double}
{1x1001 double}
{1x1001 double}
{1x1001 double}
{1x1001 double}
{1x1001 double}
{1x1001 double}
```
Let’s plot some of them:

```matlab
>> plot(z,C{5},z,C{8},z,C{10})
>> xlabel('x')
>> ylabel('Bessel function \( J_\nu (z) \)')
>> legend('
\( \nu = 5 \)','
\( \nu = 8 \)','
\( \nu = 10 \)')
```

STRUCTURES

Structures is yet another way to group things together, similar to a database. We first mention the commands `cell2struct` and `struct2cell` (get help on them).
We may define a structure, using the command `struct`. As an example, consider the following, which defines a structure called `point`, with three fields `x`, `y` and `c`, with values 12, –8, ‘r’.

```plaintext
>> point = struct('x', 12, 'y', -8, 'c', 'r')
point =
    x: 12
    y: -8
    c: 'r'
```
an empty structure:

```
>> struct()
ans =
1x1 struct array with no fields.
```

an empty array of structures:

```
>> struct([], [])
ans =
0x0 struct array with no fields.
>> struct(['name', {}])
ans =
0x0 struct array with fields:
    name
```

an array of structures:

```
>> triangle = struct('x', { 5 5 15 }, ...
    'y', { 0 10 10 }, ...
    'c', 'k')
triangle =
1x3 struct array with fields:
    x
    y
    c
```

indexing returns a single structure:

```
>> triangle(2)
ans =
    x: 5
    y: 10
    c: 'k'
```

selecting a field returns a cell array of values:

```
>> triangle.y
ans =
    0
ans =
    10
ans =
    10
```
As an example, consider creating a structure array called `student`, with fields `Name`, `SSN`, `Email` and `Tests`. Such an array structure can be quite useful when managing a class. For a class with `n` students, we will have a structure with `n` members, each having their own value for each one of the fields mentioned above. For example, consider the following two members:

```plaintext
As an example, consider creating a structure array called `student`, with fields `Name`, `SSN`, `Email` and `Tests`. Such an array structure can be quite useful when managing a class. For a class with `n` students, we will have a structure with `n` members, each having their own value for each one of the fields mentioned above. For example, consider the following two members:
```
Now, to define such a structure, we type

```matlab
>> student(1).Name='John Smith';
>> student(2).Name='Mary Jones';
>> student(1).SSN='392-77-1786';
>> student(2).SSN='431-56-9832';
>> student(1).Email='smithj@myschool.edu';
>> student(2).Email='jonesm@myschool.edu';
>> student(1).tests=[67,75,84];
>> student(2).tests=[84,78,93];
>> student

student =
  1×2 struct array with fields:
    Name    SSN    Email    tests

Now imagine that we have such a structure but with \( n \) members, i.e. we have the data of \( n \) students. We could, e.g. find the mean score test of all the students with the command

```matlab
>> mean(mean(student(1:end).tests))
```

ans =

75.3333
CLASSES (OBJECT ORIENTED PROGRAMING):

A class within MATLAB is defined using a file called `classname.m` where `classname` is the name of the class you wish to define. Within this file the two most basic elements are: (1) a declaration of the data of the class (the properties) and (2) a constructor for your class. The properties are simply a list of the variables that you will use to reference the data stored in instances of your class (objects). The constructor is a method (function) with the same name as your class which contains the rules for storing the data associated with instances (objects) of the class. It is needed to create objects.

We would like to create a class for the rational numbers, named `ratnum`. Thus, the class definition file will be called `ratnum.m`. The simplest version will look like this:

```matlab
classdef ratnum
    % RATNUM -- class for rational numbers
    % Properties for the class
    properties (Access = protected)
        n % Numerator
        d % Denominator
    end

    methods
        function r = ratnum(numerator,denominator)
            % Usage: r = ratnum(numerator,denominator)
            % Purpose: Constructor for rational number objects
            % Input: numerator -- numerator for rational number
            % denominator -- denominator for rational number
            % Output: r -- rational number object
            r.n = numerator;
            r.d = denominator;
        end
    end
end
```

The attribute (Access = protected) is optional but we will usually employ it as it enforces a stricter object-oriented framework on us. The `methods` block is terminated with an `end` tag. It contains a single method for our class, the constructor `ratnum`. The constructor has the same name as the class and takes two arguments, the values of the numerator and the denominator. It then stores them in what looks like a structure, `r`, and returns this as its output. MATLAB will interpret this variable as being of class `ratnum`.

If one were for instance to type in the command window

```matlab
>> a = ratnum(1,3)
```

```
a =
ratnum with no properties.
```

MATLAB would create (construct) a rational number (1/3) and the result would be bound to the variable `a` which in the lingo is now a `ratnum` object (or instance of the `ratnum` class).
Look in the workspace window and you will see that it is listed as being of class `ratnum` or try typing `class(a)`. Since we do not have any methods, beyond the constructor, in our class there is little that we can do with a. We cannot even look at the numerator or denominator; attempting to do so generates an error.

```matlab
>> a.n
No public property 'n' for class 'ratnum'.
```

In a strict OOP framework one must have a method defined if one wants to do anything with an object. This is known as the concept of data encapsulation. The internal data of the object is (encapsulated) protected from other parts of the program. A user of a class does not need to know the internal storage details.

As a first method beyond the constructor, let us add the `disp` method to our class definition so that rational numbers can print in the command window. The only thing the method has to do, is print out the rational number. The methods within the class definition have access to the internal data of instances of the class so this is rather easy to write. A simple version is as follows:

```matlab
function disp(r)
    % Usage: disp(r)
    % Purpose: display a rational number object
    % Input: r -- rational number object
    % Output: display the rational number
    if (r.d ~= 1)
        fprintf('%d/%d
',r.n,r.d);
    else
        fprintf('%d
',r.n);
    end
end
```

This method is placed within the `methods` block of our class definition file. If we now go back and create our rational number object, we will see 1/3 printed to the screen. (Every time you change your class file you should type `clear classes` at the command prompt otherwise MATLAB will not be able to use your changes.)

```matlab
>> a = ratnum(1,3)
a =
1/3
```

By default, when an object is created in MATLAB and MATLAB needs to print it to the screen, MATLAB calls the `disp` method in the class definition file for the object. If you wish to directly print the object, you can also call its `disp` method; e.g. `disp(a)`.

Continuing, let us now add some real functionality to the class. Let us create a method `add` which will add two rational numbers together and return the output as a rational number object. Formally the addition rule is given as

\[
\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd}.
\]

The required method should look like this:

```matlab
function r = add(r1,r2)
```
% Usage: r = add(r1,r2)
% Purpose: add two rational numbers
% Inputs: r1 -- rational number object
% r2 -- rational number object
% Output: r -- Sum of r1 and r2 as a rational number object

r = ratnum(r1.n*r2.d + r2.n*r1.d,r1.d*r2.d);
end

Note that the output is a rational number which we created by calling the rational number constructor ratnum. Also note that the method has access to the internal data of the two rational number objects. As an example, if one typed

```matlab
>> a = ratnum(1,3);
>> b = ratnum(1,2);
>> c = add(a,b)
```

then `c` would be an instance of our class and the screen output would be 5/6.

In the class definition we have set up so far, there is no mechanism for looking at the numerator or the denominator of a rational number object nor is there a way to reset these values once an object has been instantiated (created). When designing a class, one has to decide if these are useful features that you wish to give users of the class. If they are then they are added to the methods block of the class definition. Methods which allow you to query property values are known as getters and the methods that permit you to set property values are known as setters. For example, if we wanted to permit users to directly get and set the numerator and denominator of a rational number object, then we would define four separate methods for these purposes: `setN`, `setD`, `getN`, and `getD`. The appropriate code, placed within the methods block for the numerator getter and setter would like:

```matlab
function n = getN(r)
% Usage: n = getN(r)
% Purpose: Get the numerator of a rational number object
% Input: r -- rational number object
% Output: n -- the value of the numerator
    n = r.n;
end

function r = setN(r, numerator)
% Usage: r = setN(r, numerator)
% Purpose: Set the numerator of a rational number object
% Input: r -- rational number object
% numerator -- new numerator value
% Output: r -- reset rational number object
    r.n = numerator;
end
```

Then the usage would then look like:

```matlab
>> r = ratnum(3,7)
r =
3/7

>> getN(r)
ans =
3```
Note that getters are quite common in class definitions. Setters on the other hand should only be set up after careful consideration (does the user really need direct access to the internal data of an object?).

**Assignment:** finish all necessary code, in order for the class to function precisely as described. In addition, think of and implement your own methods for anything you might want to do with rational numbers. (You should think of at least two.) Provide code and appropriate examples.

**Overloading**

*Overloading* is the idea that a given method name will do different things depending upon the class of its input arguments. We have already seen this in action with the `disp.m` method. Every time MATLAB tries to display any object it uses the display method defined within the corresponding class definition; i.e. it first looks up the classname of the object it is trying to display and then it executes the appropriate method upon the object. This is enormously useful for a number of reasons, the most obvious being that we don't have to invent different names for methods that essentially do the same thing on different objects (such as display them).

MATLAB even allows one to overload operators like + and * etc. This is done by creating methods with special reserved names. For instance, if we have two rational numbers `r1` and `r2`, we can add them using `add(r1,r2)`. But it would be nice to add them by typing `r1 + r2`. This is accomplished by using MATLAB’s reserved name, `plus`, for +. Thus all we need to do is rename our addition method `add` to `plus`. MATLAB will then interpret statements like `r1 + r2` as `plus(r1,r2)`. Other reserved names can be found by typing `help matlab/ops` (scroll to the top of the list and you will see names on the left that are the reserved names corresponding to the operator on the right).

**Assignment:** use overloading twice in the class `ratnum`.

Next, we would like to create a class for polynomials, named `polynomial.m`. This value class handles polynomials of the form

\[ p(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_m x^m. \]

in a convenient and simple way.

The class definition shown below, gives specific information about this class and the methods you should implement.

```matlab
classdef polynomial

%POLYNOMIAL
    properties (GetAccess=public,SetAccess=private)
```
coeffs=0;
order =0;
end
methods
  function self = polynomial(arg)
  function [tf] = iszero(poly)
  function [y] = evaluate(poly,x)
  function [apoly] = plus(poly1,poly2)
  function [mpoly] = minus(poly1,poly2)
  function [ipoly] = integrate(poly,const)
  function [dpoly] = differentiate(poly)
  function [iseq] = eq(poly1,poly2)
  function [] = plot_it(poly,x,pstr,ax)
  function [] = disp(poly)
end

Once written, the class should work as the following examples show.

```matlab
>> p1 = polynomial([1,2,3]);  %3x^2+2x+1
>> p2 = polynomial(p1);       %3x^2+2x+1
>> p3 = polynomial([1,2,3,0]); %3x^2+2x+1

>> p(1,7) = polynomial([1,2,3]); %3x^2+2x+1
>> length(p)
anst =
  7

>> p(3)
anst =
  0.0000

>> p(7)
anst =
  1.0000 + 2.0000 x + 3.0000 x^2

>> p1.order
anst =
  2

>> p2.coeffs
anst =
  1 2 3

p3.coeffs = [5,2,3];
??? Setting the 'coeffs' property of the ...
'polynomial' class is not allowed.

>> p3.iszero()
anst =
  0

>> p3.evaluate(0:0.25:1.0)
anst =
  1.0000 1.6875 2.7500 4.1875 6.0000

>> p4 = polynomial(0);

>> p4.iszero()
anst =
  1
```
Think of at least two more methods to implement and illustrate the implementation of all the methods with additional examples.