Matlab has been adapted to many tasks over the years, but its fundamental task has mainly been numerical computation. This lecture will focus on using Matlab’s built in numerical functionality to solve a variety of common numerical problems that arise in a variety of fields and subjects.

**Function Handles**

Many of the numerical solvers in Matlab require the evaluation of functions and Matlab provides a variety of methods for passing functions to other functions for evaluation. This capability is also useful in a variety of other programming applications, such as GUI design, but here we will be focusing on the essentials how these constructs operate.

**Function Handles**

To create a function handle we use the following syntax

```matlab
>> fhandle = @myfuncname;
```

*fhandle* can then be passed to other functions for evaluation or called along with *myfuncname*’s argument list to cause the function to be evaluated.

**Example**

```matlab
>> H=@humps;
>> H(2),humps(2)
```

We’ll see examples of passing functions handles to other functions in later sections.

**Anonymous Functions**

The @ syntax can also be used to define what are known as anonymous functions. Anonymous functions are a way of creating simple functions on the fly without using m-files. They have the following syntax

```matlab
>> fhandle=@(arglist) expr
```
Example 1

```
>> sqr=@(x)x.^2;
>> sqr(5)
```

Example 2

```
>> a = 1.3;   b = .2;   c = 30;
>> parabola = @(x) a*x.^2 + b*x + c;
>> fplot(parabola, [-25 25])
```

Note: Changing a, b and c and then calling parabola again will not cause the parameters a, b and c to change within the function.

Example 3

```
>> add=@(x,y) x+y;
>> add(3,5)
```

Inline Functions

There is one last method of creating functions to be evaluated.

```
>> f=inline(expr)
```

where expr is a character string. Matlab will parse the string and pull out variable names automatically. Inline functions can be a bit slower to work with than anonymous functions, but can be useful when manipulating strings that describe desired functions. Note these inline functions are not function handles.

Example

```
>> f = inline('sin(alpha*x)')
```

```
f =
    Inline function:
    f(alpha,x) = sin(alpha*x)
```

Linear Algebra

Matlab has been designed to handle numerical linear algebra problems in an efficient way. There are several built in functions to handle some of the most common problems of interest.

Matrix Decomposition

There are several tasks in numerical computing and engineering that involve decomposing a matrix into some format that makes solving the underlying problem easier or more efficient. One common problem takes the form \( Ax = b \), where \( A \) is a matrix of parameters, \( b \) is some desired output and \( x \) is a vector of unknown inputs.
Example
Solve 9x+3y=5 and 7x-9y=10 for x and y.

\[
\begin{align*}
\text{>> } & A=\begin{bmatrix}9 & 3 \\ 7 & -9\end{bmatrix}; b=\begin{bmatrix}5 \\ 10\end{bmatrix}; \\
\text{>> } & A\backslash b \\
\text{>> } & A*\text{ans}
\end{align*}
\]

While this was a simple example, in engineering linear systems of this form occur quite often, e.g., data fitting, linearizing a nonlinear problem around some operating point, linear dynamic systems, control systems.

In the previous example, A was square and as such there is exactly one solution, but in practice it is more common that problems will be over determined and one seeks a solution than minimizes some measure of error. In this case, Matlab will find a least squares solution, i.e., it minimizes \( \text{norm}(Ax-b)^2 \).

Example
Solve 9x+3y=5, 7x-9y=10, 16x+10y=6 for x and y.

\[
\begin{align*}
\text{>> } & A(3,:)=[16,10]; b(3)=6; \\
\text{>> } & A\backslash b, A*\text{ans}
\end{align*}
\]

In the case of square matrixes, Matlab will employ a method called LU decomposition that decomposes the matrix A into two matrices L (lower triangular) and U(upper triangular) such that A= L*U. It then becomes very efficient to solve the problem as Ly=b, Ux=y using forward and backwards substitution.

Example

\[
\begin{align*}
\text{>> } & [L,U]=lu(A(1:2,:)) \\
\text{>> } & y=L\backslash b \\
\text{>> } & x=U\backslash y \\
\text{>> } & A(1:2,:)*x
\end{align*}
\]

If the matrix is not square, such as in the least squares problem, Matlab will make use of a decomposition known as the QR decomposition. This decomposition will generate an orthogonal matrix Q (inv(Q)=Q') and an upper triangular matrix R such that QR = A. This decomposition can be used to find efficient solutions to the least squares problem.

Example

\[
\begin{align*}
\text{>> } & [Q,R]=qr(A) \\
\text{>> } & y=Q'*b \\
\text{>> } & x=R\backslash y \\
\text{>> } & A*x
\end{align*}
\]
A third decomposition that has much importance both theoretically and in numerical applications is the Singular Value Decomposition (SVD). The SVD has a wide range of applications ranging from signal processing to the alignment of 3D models and much more. One common example is solving a system of the form \( Ax = 0 \). (For instance, this type of problem occurs quite often in computer vision tasks)

**Example**

```matlab
>> [U,D,V] = svd(A)
>> U*D*V'
```

Find \( x \) such that \( Ax = 0 \);

```matlab
>> A = [2,4,1,3; -1,-2,1,0; 0,1,2,2; 3,6,2,5];
>> [U,D,V] = svd(A)
>> A*V(:,4)
>> null(A)
```

**Useful Matrix Functions**

```matlab
>> norm(A)
>> cond(A)
>> rank(A)
>> det(A)
>> inv(A)
>> pinv(A)
>> eig(A)
>> eigs(A)
```

**Exercises**

1. Type `hilb(15)\ones(15,1)` and examine the error message.
2. Define a matrix: \( H = \text{hilb}(15) \). The Hilbert matrix \( H \) is an example of an extremely poorly conditioned matrix. Compute the LU, QR and SVD decompositions of \( H \) as well as all of the various useful matrix functions.
3. Repeat question 2 for \( V = \text{vander}(1:5) \)
Polynomials, Data Fitting, and Interpolation

Given measurements from an experiment, one often wants to find a simple model that explains the observed data. Using this model it is possible to predict (or interpolate) new values that are likely to follow the same underlying pattern as the observed data. Polynomial models are some of the simplest models that can be used for interpolating data and Matlab has several tools for working with polynomials. Polynomial expressions are also of importance to fields such as filter design or coding theory.

Matlab also offers additional tools for interpolating data without explicitly computing an interpolating function. These tools allow the user to specify some class of interpolating function, but the details are never returned to the user.

Polynomials

We will start by defining a polynomial and examining various functions related to polynomials. Matlab represents a polynomial
\[ p(x)=p_1x^n+p_2x^{n-1}+\ldots+p_nx+p_{n+1} \]
as a row vector of the coefficients.

Example

```
>> p = [5,3,6];
>> polyval(p,[5,7,10])
>> z = roots(p)
>> p1 = poly(z)
>> pd = polyder(p)
>> A = [0 1; 1 1];
>> pm = poly(A)
>> polyvalm(pm,A)
```

Additionally we can fit polynomials to datasets to generate polynomial coefficients using polyfit.

Example

```
>> x = linspace(-2,2,20);
>> y = 1./(x+(1-x).^2);
>> p = polyfit(x,y,3);
>> xx = linspace(-2,2,100);
>> plot(x,y,".",xx,polyval(p,xx),'-','MarkerSize',30,'LineWidth',2)
```

More On Interpolation

Matlab also supports several additional piecewise data interpolation functions. These include spline, pchip, interp1, griddata and interp2. Spline and pchip both fit piecewise continuous cubic polynomials to the
data. However they enforce different continuity conditions and behavior between points.

**Example**

```matlab
>>x=[-12:4:12];
>>y=atan(x);
>>t=[-12:.1:12];
>>p=pchip(x,y,t);
>>s=spline(x,y,t);
>>plot(x,y,'o',t,p,'-',t,s,'-.','LineWidth',1.25)
>>xlim([-12,12])
>>legend('data','pchip','spline','Location','NW')
```

It is also possible to extract the coefficients of the spline or pchip commands using `pp=spline(x,y)` or `pp=pchip(x,y)` then use the command `ppval` to evaluate the coefficients just like `polyval`.

Interp1 can perform linear, nearest neighbor, cubic spline and piecewise hermitian (same as pchip) interpolation. It requires that the x values be monotonically increasing.

To interpolate data in two dimensions, Matlab provides the functions `griddata` and `interp2`. These functions have similar arguments but like `interp1`, `interp2` requires monotonically increasing x and y paramenters..

**Example**

```matlab
>>x=rand(100,1)*4-2;y=rand(100,1)*4-2;
>>z=x.*exp(-x.^2-y.^2);
>>hi=-2:.1:2;
>>[XI,YI]=meshgrid(hi);
>>ZI=griddata(x,y,z,XI,YI);
>>mesh(XI,YI,ZI), hold on
>>plot3(x,y,z,'o') ,hold off
```

**Note:** Matlab can also find the roots of general functions of one variable (`fzero`) and find the minimum of a function of one variable (`fminbnd`) or of n variables (`fminsearch`).

**Exercises**

1. Define the following vectors: `x=[0 pi/4 3*pi/8 3*pi/4 pi];` `y=sin(x);` and `xi=linspace(0,pi,40);`
   Use the interp1 function to compute `yi` using four different methods:
   ‘linear’, ‘nearest’, ‘spline’ and ‘pchip’
   Plot `yi` vs `xi` for all 4 different results into a single plot window.
(You can use different colors for different plots. Look up the help on plot to figure out how to do this).

2. Polynomial division and multiplication can be accomplished using the conv and deconv functions in Matlab. Read the help files for these two functions.
- Define the following polynomials: \( p = [3 \ 2 \ 1] \) and \( q = [4 \ 5 \ 2] \).
- Multiply \( p \) and \( q \) by using \( \text{conv}(p,q) \).
- Define two polynomials \( g = [1 -6 12 -8] \) by \( h = [1 \ -2] \).
- Divide \( g \) by \( h \) using \( \text{deconv}(g,h) \).

3. Now, define a new polynomial of your choice. (It can be of any order). Find the roots of the polynomial and plot the polynomial over some range of values (an example range could be \( x = -1:0.1:1 \)).

4. 2D Interpolation: Let us see how we can repeat the griddata example using interp2.
- Define \( x,y \) as follows: \( [x,y] = \text{meshgrid}(-2:.3:2) \);
- Define \( z, h, XI \) and \( YI \) exactly as in the example.
- Compute \( z_{\text{nearest}} \) using interp2 and 'nearest' interpolation.

Numerical Integration (Quadrature)

Integral equations play a role in numerous fields of engineering and mathematics. Matlab provides several functions for evaluating integrals numerically when they are well defined over a finite interval. For infinite integrals or functions that have singularities over the desired interval, there are a variety of approaches available to generate problems that Matlab can handle. A standard numerical analysis textbook will cover many of these approaches.

The Matlab functions for numerical integration are:

- \( \text{quad}(\text{fun},a,b,\text{tol}) \)
- \( \text{quadl}(\text{fun},a,b,\text{tol}) \)
- \( \text{trapz}(x,y) \)
- \( \text{dblquad}(\text{fun},x_{\text{min}},x_{\text{max}},y_{\text{min}},y_{\text{max}},\text{tol}) \)
- \( \text{triplequad}(\text{fun},x_{\text{min}},x_{\text{max}},y_{\text{min}},y_{\text{max}},z_{\text{min}},z_{\text{max}},\text{tol}) \)

Two standard methods for this process are quad and quadl. These functions differ in the underlying methods used with quadl being more accurate but slower. See Matlab’s help files for more details regarding the underlying methods.

To integrate \( x \log(x) \) over \([2,4]\) we call
\[
\text{quad(@(x)x.*log(x),2,4)}
\]
\[
\text{ans = 6.041}
\]
• Note that we have introduced an anonymous function syntax in this example as quad requires that we pass it a function handle which accepts vector arguments and returns vector arguments.

To see the efficiency difference between quad and quadl we can check the following commands

\[
\texttt{tic, quad(@(x)x.*log(x),2,4), toc} \\
\texttt{tic, quadl(@(x)x.*log(x),2,4), toc}
\]

**Example (Fresnel Integrals)**

\[
n=2000; \\
x=\text{zeros}(1,n); y=x; \\
t=\text{linspace}(0,8*\pi,n+1); \\
\text{for } i=1:n \\
\quad x(i)=\text{quadl}(@(x)\cos(x.^2),t(i),t(i+1),1e-3); \\
\quad y(i)=\text{quadl}(@(x)\sin(x.^2),t(i),t(i+1),1e-3); \\
\text{end} \\
x=\text{cumsum}(x); y=\text{cumsum}(y); \\
\text{plot([-x(end:-1:1) 0 x],[-y(end:-1:1) 0 y])} \\
\text{axis equal}
\]

The trapz function takes a vector x and a vector f(x) as input and will evaluate the integral using the trapezoid rule with the endpoints specified by the two vector inputs

\[
\text{>>x=\text{linspace}(0,2*\pi,10);} \\
\text{>>f=\text{sin}(x).^2./\text{sqrt}(1+\text{cos}(x).^2)} \\
\text{>> \text{trapz}(x,f)} \\
\text{ans} = 2.8478
\]

To evaluate double and triple integrals, the functions dblquad and triplequad can be used. These functions have the same calling conventions as quad/quadl, but have the additional requirement that the function passed to the command accept a vector and scalar (or 2 scalars) input and return a vector.

\[
\text{function out = fxy(x,y)} \\
\quad \text{out} = y^2*\exp(x)+x*\cos(y);
\]

\[
\text{>>dblquad(@fxy,0,1,4,6)} \\
\text{ans=} 87.2983
\]

**Exercises**

1) Integrate the function \(x \sin(x)^2\) over the range \([0,4*\pi]\) using quad, quadl and trapz and anonymous functions.

2) Integrate the function \(x*y*z+x*\exp(z)+y*\cos(z)+4*x*\sin(y)\) over \(x=[0,1], y=[9,15], z=[3,7]\) using a function written in an m-file and a function handle.
Ordinary Differential Equations

ODEs are very common in several engineering applications. Many systems can be described by ODEs and Matlab has several built in solvers to perform numerical evaluation of ODEs given initial values. They all have the following form

\[ [T,Y]=\text{solver}(\text{defun},\text{trange},\text{IV}) \]

where defun is a function of \( y \) and \( t \), trange is the time range over which the solver will operate and IV is the initial value of the system.

This command structure will solve systems of the form \( y'(t)=f(t,y(t)) \) with \( y(0)=IV \)

Note the function passed to the solver should accept a vector and return a vector.

Additionally the help files will mention that some solvers are better for stiff or non-stiff problems. Stiff problems are those for which various numerical solvers become unstable for all but extremely small step sizes, i.e. the solver must work very slowly else the solution will fail to converge.

- \text{ode45}(\text{defun},\text{trange},\text{iv}) \) is generally the first solver that should be tried and will work in many cases. It is designed for non-stiff problems.

Examples(Simple Pendulum Equation)

Solve the simple pendulum equation \( a''(t)+\sin(a(t))=0 \) and plot the phase plot.

First convert to a system of first order equations \( y_1(t)=a(t), \)
\( y_2(t)=a'(t) \)

\[
\text{function } \text{yprime=pend}(t,y) \\
\text{yprime}=[y(2),-\text{sin}(y(1))];
\]

\[
\gg \text{tspan = [0 10];} \\
\gg \text{yazero=[1;1];ybzero=[-5;2];yczero=[5;-2];} \\
\gg \text{[ta,ya]=ode45(@pend,tspan,yazero);} \\
\gg \text{[tb,yb]=ode45(@pend,tspan,ybzero);} \\
\gg \text{[tc,yc]=ode45(@pend,tspan,yczero);} \\
\gg \text{[y1,y2]=meshgrid(-5:.5:5,-3:.5:3);} \\
\gg \text{Dy1Dt=y2;Dy2Dt=-sin(y1);} \\
\gg \text{quiver(y1,y2,Dy1Dt,Dy2Dt)} \\
\gg \text{hold on}
\]
Partial Differential Equations and Boundary Value Problems
Matlab can also solve PDEs and BVPs and has several built in solvers. These go strongly beyond the scope of this class and are mentioned only for completeness. They operate similarly to the ODE solvers and help bvp4c and help pdepe can provide additional information on these subjects.

Exercises
1. Let \( y'(t) = -y - 5 \exp(-t) \sin(5t) \). Using function handles and ode45 solve this problem for tspan=[0 3] and yzero=1. Plot the results.
   Steps: Define a function called calculate_yprime which takes in \( y \) and \( t \) returns a variable \( yprime \) according to the equation above.
   Next, apply the ode45 function which takes in three arguments: a function handle to calculate_yprime, tspan and yzero as and solves the system to return ta and ya.
   Plot ya vs ta.
2. Solve the same problem but use tspan=1:4. Notice the difference in the outputs.
3. Solve the problem one more time using tspan=[0, -.5, -1]. This shows that the solver can solve both backwards and forwards in time.