EENG 212-M01
EENG 221-M01

Tutorial 8

MATLAB

PART A

Interpolation and Curve Fitting

We consider now two approaches to describe data. In interpolation, data is correct and what we want is to know what happens between the data points. Another method called curve fitting or regression looks for some smooth curve that " best fit " the data; in some way it is the average curve.

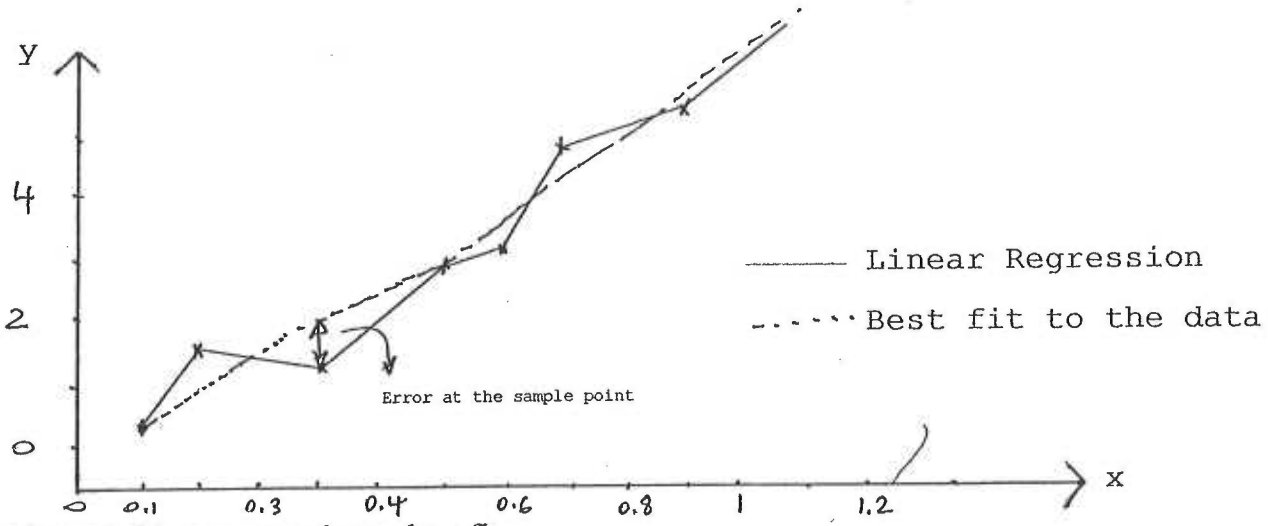


Figure 1: Linear regression vs best fit

1-Curve Fitting:

There are many "best fit" to data , but here by "best fit", it means minimizing the sum of the squared error at the data points and the curve used is restricted to polynomials: mathematically, it is the least squares curve fitting to a polynomial. In other words, the smooth curve makes the "sum of the squared error" as small as it can be.

In Matlab, the function **polyfit** solves the least squares curve fitting problem.

Consider the following data where y=f(x)
x=[0 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000 0.7000 0.8000 0.9000 1.0000]
y=[-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48 9.30 11];

To use the function polyfit, we must give to the above data the order of the polynomial we wish to best fit to the data. If we choose n=1 as the order, the best straight line approximation will be found (it is called linear regression). If we choose n=2, a quadratic polynomial will be found.

Example

```
x=0:0.1:1;
y=[-0.447  1.978  3.28  6.16  7.08  7.34  7.66 9.56  9.48 9.30  11 ];
n=2;
p=polyfit(x,y,n)
```

the result is

```
p =

 -10.1605   20.3880   -0.0568
```

In other words, y= -10.1605* x^2 + 20.1293* x -0.0317

To compare the curve fit solution with the data points, let's plot both in the following example

Example:

```
x=0:0.1:1;
y=[-0.447  1.978  3.28  6.16  7.08  7.34  7.66 9.56  9.48 9.30  11 ];
n=2;
p=polyfit(x,y,n);
xi=linspace(0,1,100);
z=polyval(p,xi); % evaluate p at the data points xi
plot (x,y,'*', xi,z,'0')
xlabel('x'), ylabel('y=f(x)')
title('second order curve fitting')
```

See figure 2 for results

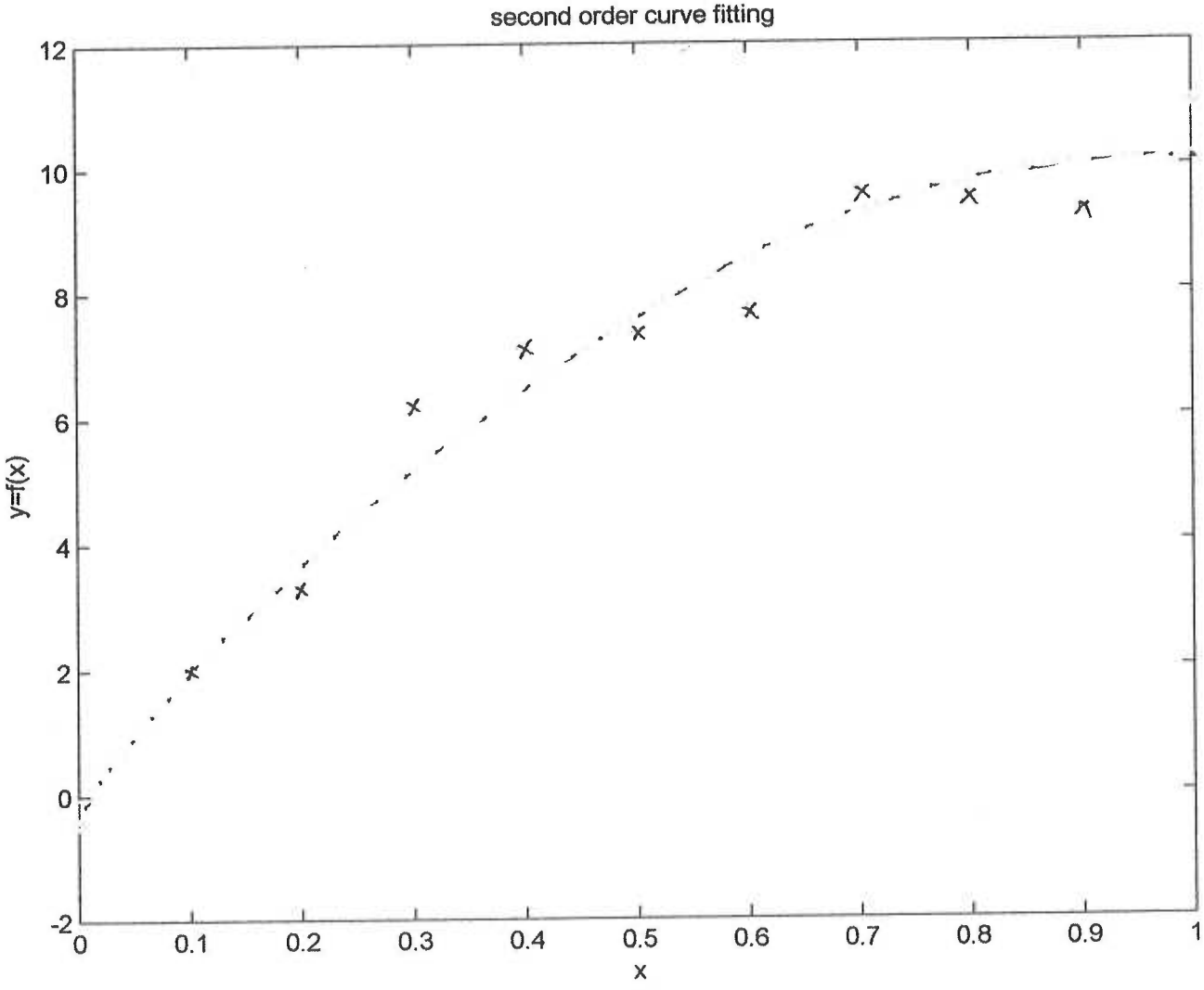

second order curve fitting
y=f(x)
x
12
10
8
6
4
2
0
-2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

figure 2

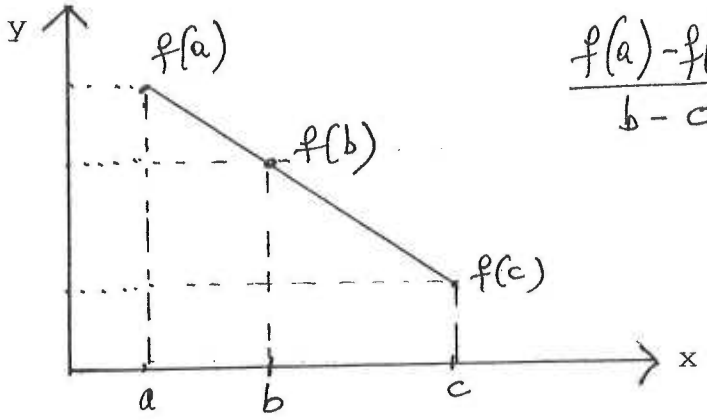
The choice of polynomial order is somewhat arbitrary since it takes (n+1) data points to uniquely specify an nth order polynomial. In our example, we had 11 data points, so we xould have chosen up to a 10^{th} order polynomial. Higher order polynomials do not give smooth curves.

2-Interpolation:

A method used to estimate values of a function between those given by a set of data points resulting, say, from some experimental measurements

2.1 One dimensional interpolation

Consider a set of x,y coordinates (y=f(x)). We need to estimate f(b) that is not one of the original set of data points, but "b" is between two of the x values from the set of data.
The most common technique for interpolation is called Linear Interpolation.



$$\frac{f(a)-f(b)}{b-a} = \frac{f(a)-f(c)}{c-a}$$

In the above figure we have two data points f(a) and f(c). Assuming between f(a) and f(c), function can be estimated with a straight line, then we can obtain any points between f(a) and f(c) by using equation derived from similar triangles.

f(b) = f(a) + (b-a)/(c-a) (f(c)-f(a))

Example:

```
%temperature recorded every hour
clg
hours=1:12;
temps=[ 5  8  9  15  25  29  31  30  22  25  27  24 ]; % in Celsius
plot(hours,temps,hours,temps,'*')
title('NY temperature in degree Celsius')
xlabel('Hour'), ylabel('Degree Celsius')
pause
clg
% find temperature at 9.3 (9H30)
t=interp1(hours,temps,9.3)
```

See figure 3 for results

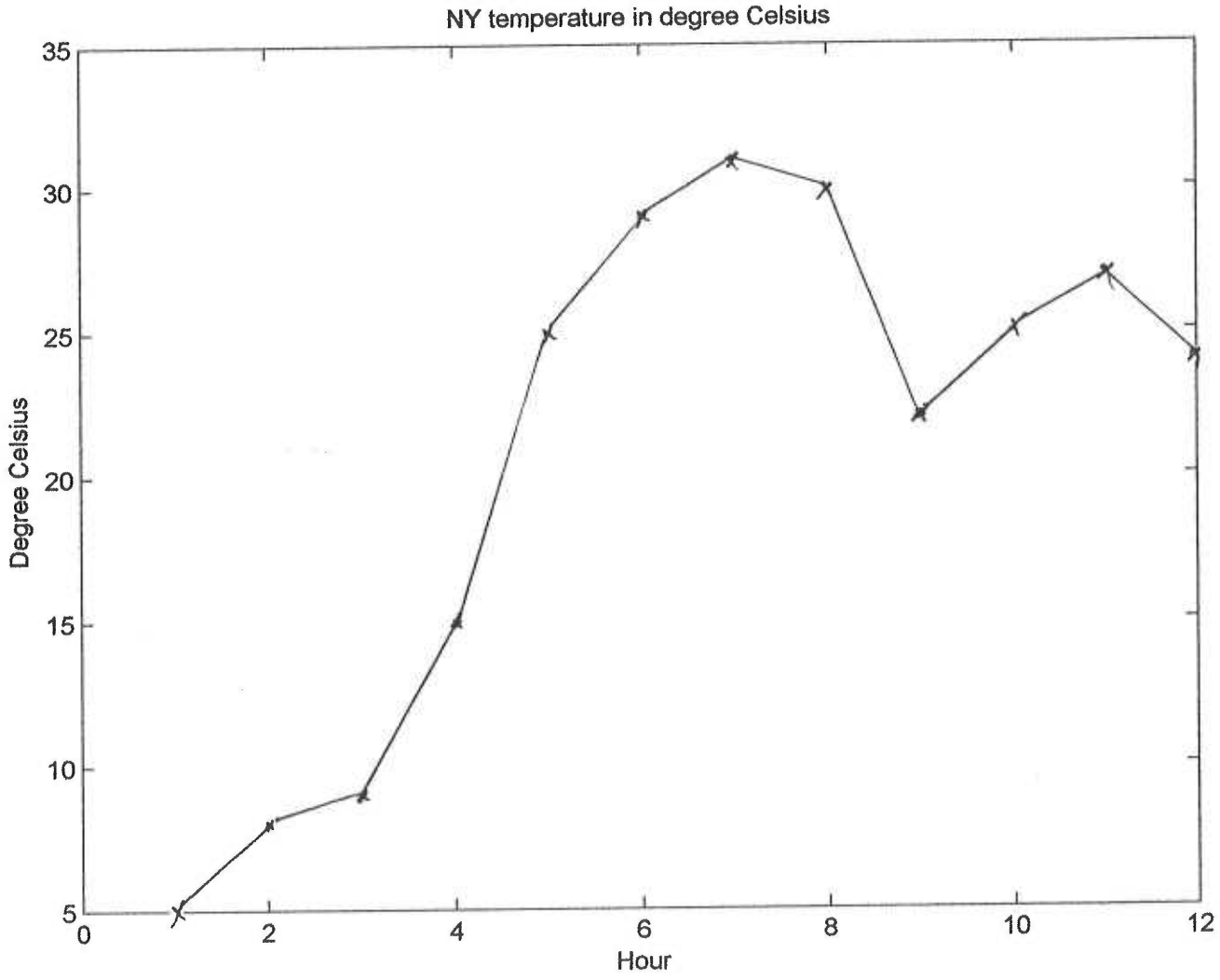

NY temperature in degree Celsius
Degree Celsius
Hour
35
30
25
20
15
10
5
0
2
4
6
8
10
12

figure 3

Matlab draws lines between points or linearly interpolate the data points. Of course more points means curve is smoother.
In the above example the resulting temperature is 22.9
But we can also interpolate more than one point as in the following Matlab command

t=interp1(hours, temps, [3.2, 6.5, 7.1, 11.7])

the answer is 10.20, 30.00 , 30.90, 24.90

If we assume a smoother line connects the data points, we also assume that a smother curve fits the data points; most common is that a third order polynomial i.e. a cubic polynomial is used. This type of interpolation is called **cubic splines** or just **splines.**

Example:
% as a continuation to our previous example
t=interp1(hours,temps,9.3,'spline');

the answer is: 21.8577

and for the case of multiple interpolations

t=interp1(hours,temps, [3.2 6.5 7.1 11.7], 'spline')

the answer is:
t= 9.6734
30.0427
31.1755
25.3820

Note: One of the most common use of spline interpolation is to smooth data (evaluate data as a fine interval)

Example:
h=1:0.1:12; % estimate temperature every 1/10 hour
t=interp1(hours,temps,h,'spline');
plot(hours,temps,'*',hours,temps,'+',h,t)

See figure 4

2.2 Two-dimensional interpolation
Same inderlying idea but now we interpolate functions of two variables z=f(x,y). We will make use of the function **interp2( ).** The following example should illustrate the use of interpolation in a two –dimensional space.

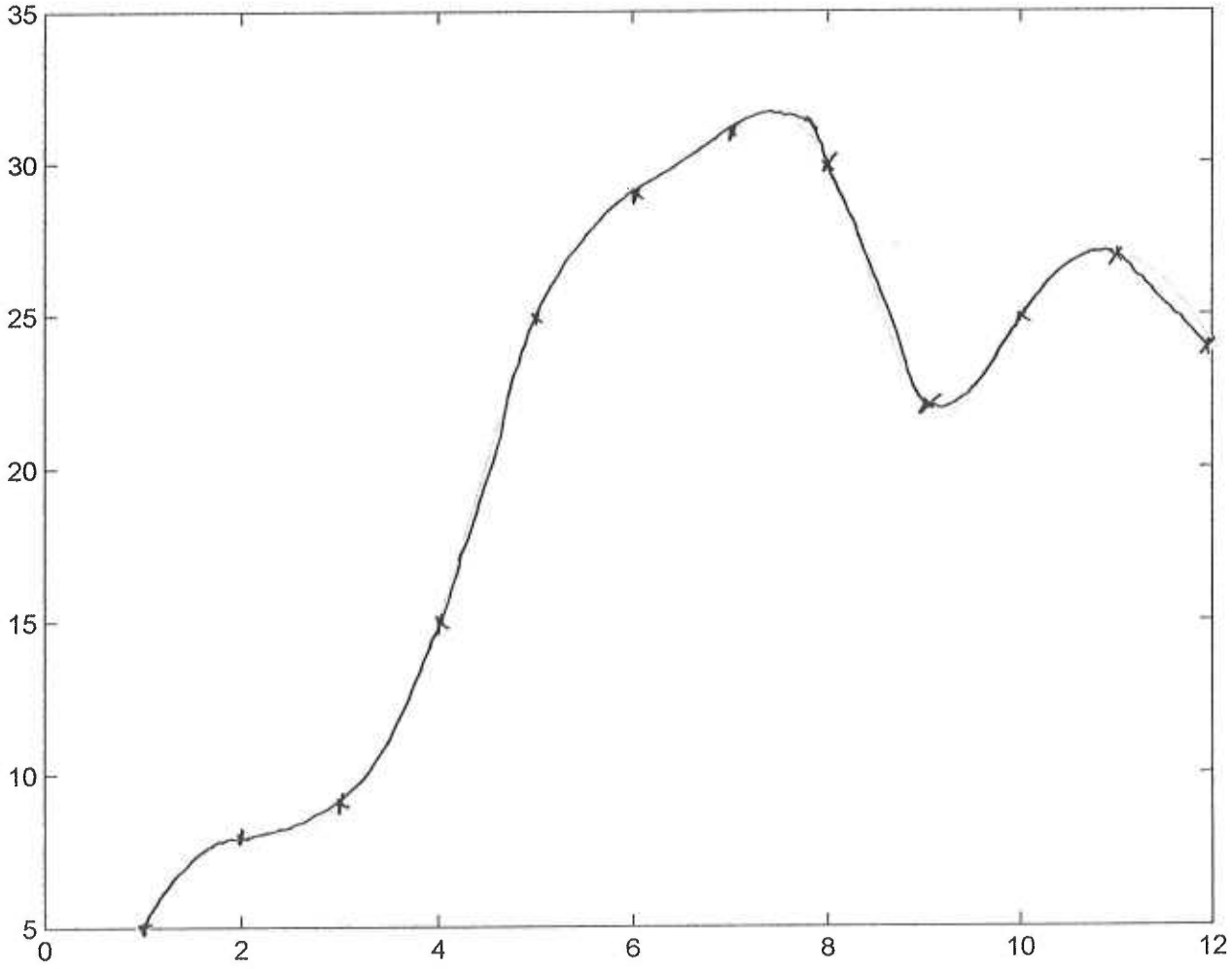
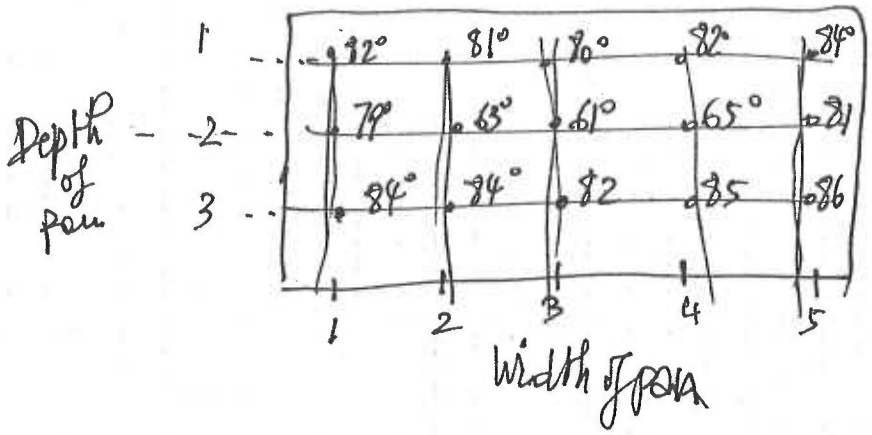
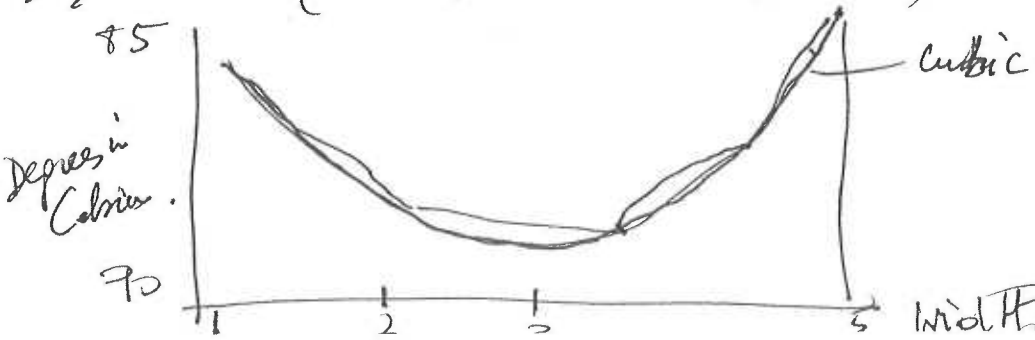


figure 4 : better fit !

ex: pan used to cook brownies. 3 x 5 grid.




```
% test to analyse data
% slice the pan at 1/2 depth, look across width
>> width = 1:5
>> depth = 1:3
>> temps = [82 81 80 82 84; 79 63 61 65 81;
            84 84 82 85 86];
>> wi = 1:0.2:5;   choose resolution per width
>> d = 2;  % center of pan.
>> z1 = interp2(width, depth, temps, wi, d, 'linear');
>> % linear interpolation
>> zc = interp2(width, depth, temps, wi, d, 'cubic');
>> plot(wi, z1, '-', wi, zc)   % plot linear and cubic.
>> xlabel('width of Pan')
>> ylabel('degree celsius')
>> title('Temperature at depth 2')
```



PART B

Numerical Integration and Differentiation

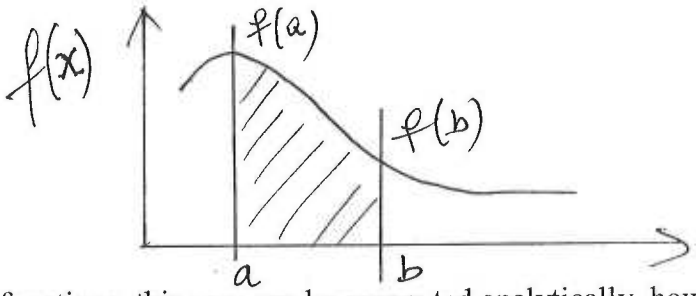
EENG 212
EENG 221

Lecture 7b
Numerical Integration and Differentiation

1-Numerical Integration and Differentiation
By now you must know that both topics are fundamental tools in calculus. Integration computes the area under a function and differentiation describes the slope or gradient of a function.

1.1 Integration
1.1.1 One-dimensional Integration

For the function f(x) show in figure 1, we can determine the area A as follows:



$$A = \int_a^b f(x)\,dx$$

Figure 1

For most functions, this area can be computed analytically, however in some situation, a numerical method is required to estimate the integral. Two methods can be used in this case to estimate the area A.

-The Trapezoidal Rule

The dashed area is represented by trapezoids and interval (a , b) is divided into n sections, then the area A is approximated as :

$$A = [(b-a)/2^n] / (f(x_0) + 2f(x_1) + 2f(x_2) + \dots f(x_n))$$

Where $x_0=a$ and $x_n = b$ and x_i's (i=0,1,2,..n) are the endpoints of the trapezoids.

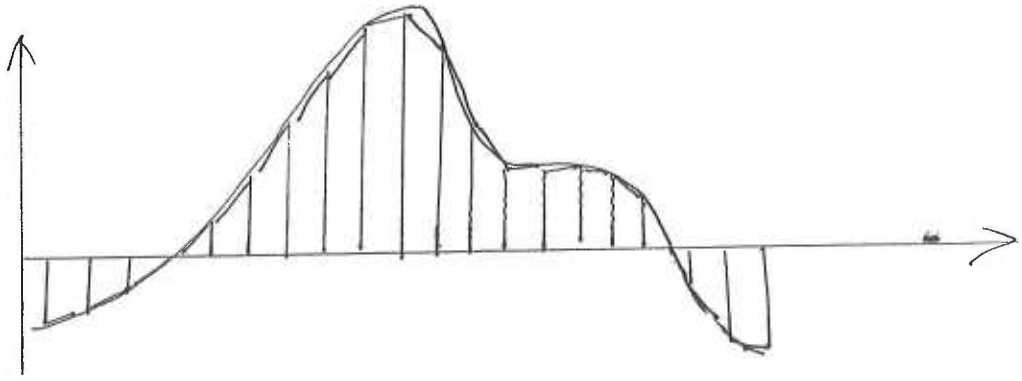


Figure2: Example of trapezoidal approximations

-The Simpson method

In this case the area A is found as

$$A= [h/3] (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots+2f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n}))$$

Here the area A is made of areas under quadratic sections of a curve,

$x_0 = a$, $x_{2n}= b$, and $h = (b-a)/2n$

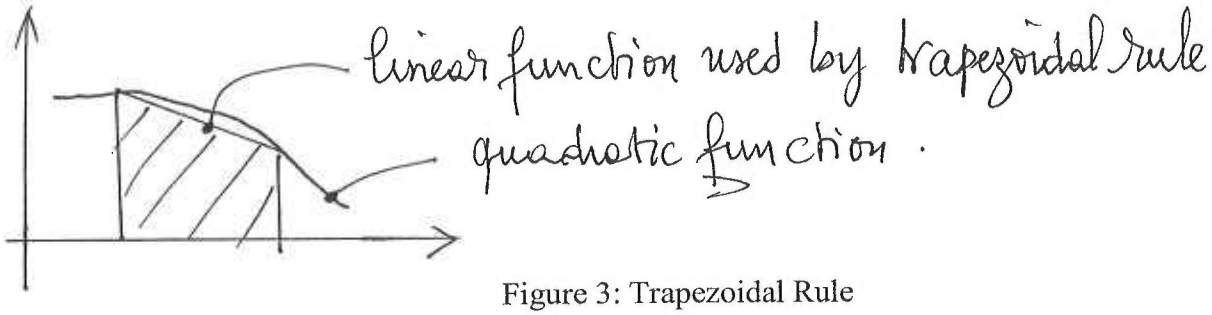


Figure 3: Trapezoidal Rule

When using the above methods, you should know that:

-the existence of singularities makes the answer almost impossible without a numerical integrator technique

- using quadratic sections improves the estimation
- in case higher order functions are used, the integration technique is referred to as Newton-Cotes integration technique.

Matlab provides functions for computing integrals:

The function **trapz**(x,y) is using for the Trapezoidal method.

The function **quad** si used when using Simpson's rule while the function **quad8** uses adaptive Newton Cotes eight panel rule. The function **dblquad** is sometimes referred to as **quad2.**

The format is dblquad('fname', xmin, x max, ymin, ymax)

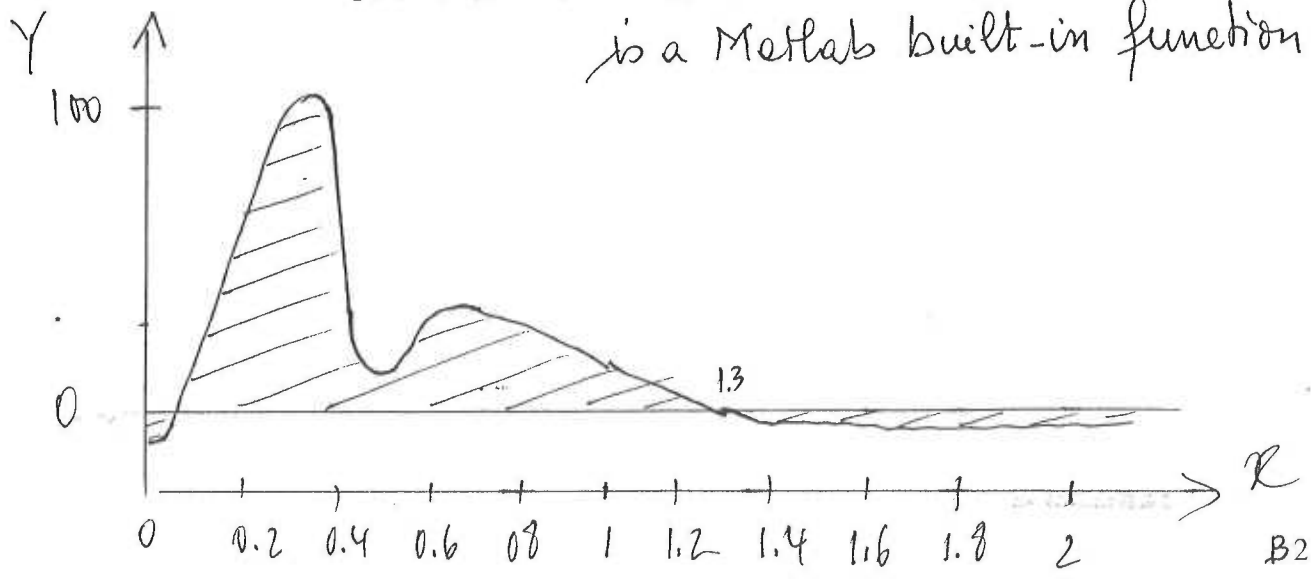
The basic use of the above functions requires 3 arguments:

quad('function name', a, b)

quad8('function name', a, b) is better with function having singularities.

Example: Evaluate the area under the nice looking function known as y=hump(x) using different approaches.

$$Y=\text{hump}(x)= (1/[(x-0.3)^2 +0.01]) + (1/[(x-0.9)^2 +0.04]) - 6$$



Example:
%evaluating the area under y=hump(x)
x=linspace(0,2);
y=hump(x) ;
plot(x,y);
grid on
area =trapz(x,y) % estimate the area under the function y using the trapezoidal rule ans=25.917
%or
format long
area =trapz(x,y) % ans=26.34473119524597
%now let's use quadrature concept. Use whatever intervals to get the most accurate result.
quad('humps',-1, 2) %ans=26.34497558341242
quad8('humps', -1, 2) %ans= 26.34496024631924

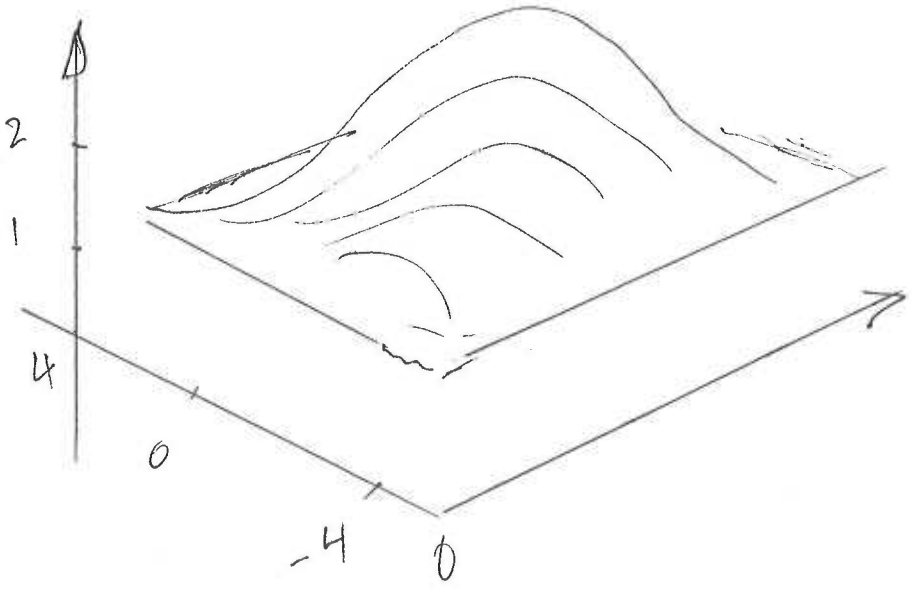
We can see that quad achieves 6 digit accuracy, while quad8 achieves 8 significant digit accuracy

1.1.2 Two-dimensional Integration

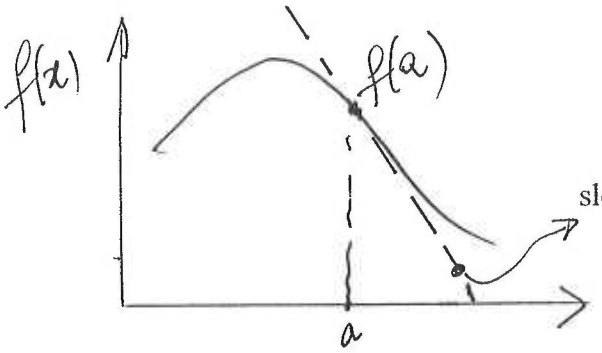
Matlab supports 2-dimesional integration with the function **dblquad**

$$\int_{y_{min}}^{y_{max}} \int_{x_{min}}^{x_{max}} f(x,y)\, dx\, dy$$

Example:
% first plot y=sin(x).*cos(y) +1 known for Matlab as **myfun** function
x=linspace(0,pi,20);
y=linspace(-pi,pi,20);
[xx,yy] =myfun(xx,yy);
mesh(xx,yy,zz) % how about a mesh plot
xlabel('x'), ylabel('y')
%volume under this function
dblquad('myfun', 0, pi, -pi, pi)
%absolute and relative tolerance can be added as a vector
% dblquad('myfun', 0, pi, -pi, pi, [rel_tol abs_tol])



2-Differentiation



slope of function or tangent at point 'a'

Figure 4: f'(a) is the derivative of f(x) at x=a

As you know from math, the derivative of f(x) at x is the rate of change of f(x) with respect to x

$$f'(x) = df(x)/dx$$

Evaluation of f'(x) using numerical differentiation

$f'(x_k) = (f(x_x) - f(x_k -1)) /(x_k - x_{k-1})$ know as backward number approximation

or $f'(x) = (f(x_{k+1}) - f(x_k)) / (x_{k+1} - x_k)$ known as forward number approximation

or $f'(x) = (f(x_{k+1}) - f(x_{k-1})) / (x_{k+1} - x_{k-1})$ central number approximation

The distance between the points makes a difference in terms of quality of approximation.
Now for second derivative :

$$f''(x) = (f'(x_k) - f'(x_{k-1})) / (x_k - x_{k-1})$$

Matlab differentiation function is **diff**
If x is a vector, **diff(x)** returns a vector containing differences between adjacent values.
For example is x= [2, 3, 6, -1, 3] then

```
diff(x) = [ 1, 3, -7, 4]    % one less value
```

Now if x is a matrix, the **diff(x)** returns a matrix containing differences
Between adjacent values in the columns; one less row should be obtained.
Now if y=f(x) then

```
dy= diff(y)./ diff(x)    % notice the use of the dot operator
```

Example:

```
x=-5 : 0.1: 5;
f=x.^4 -2.*x.^2 + 3.* x.^1 -3;
df=diff(f)./ diff(x);
%we can use the polyder function to obtain the derivative of a polynomial
Pd=polyder(f)
%now use central difference
end= length (f)
Df= (f(3:end) –f( 1: end -2)) / ( x(3)-x(1))
```

Example: Illustration of the maximum power transfer theorem with application to electrical circuits

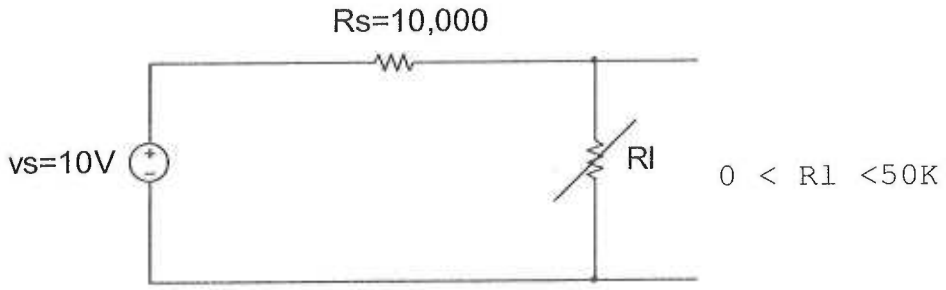


Figure 5: Maximum power transfer to Rl. Plot power dissipated by load. Verify that maximum power is dissipated at Rl=10K.

```
%pl is the power dissipated by the load
vs=10; rs=10e3 ; rl=0:1e3:50e3;
k=length (rl)   % k components in vector rl
%power dissipation calculation
for i=1:k
    pl(i) = (( vs/(rs +rl(i) )) ^2 )* rl(i)  ;    % p= r i^2 where i=v/r
end
% derivative of power calculated using backward difference
dp= diff(pl)./ diff(r);
rld=rl(2: length(dp)-1) ; % length of rld is one less than rl
%determination of critical points of derivative of power
prod= dp(1 : length (dp)-1 ).* dp( 2 : length(dp));
crit_pt=rld(find(prod < 0));
max_power= max (pl)    % max power is calculated
%print out results
fprint('max power occurs at % 8.2f  Ohms \n', crit_pt)
fprint('max power dissipation is  % 8.2f  watta \n', max_power)
%plot power versus load
plot (rl, pl, "*')
title('Power distributed to load')
xlabel('Rl in Ohms')
ylabel ('Power dissipation in Watts')
```