Topic 9
Curve Fitting and Optimization

Material from MATLAB for Engineers, Moore, Chapters 13
Additional material by Peter Kovesi and Wei Liu
In this lecture we will

◊ interpolate between data points, using either linear or cubic spline models
◊ model a set of data points as a polynomial
◊ use the basic fitting tool
◊ consider the optimization problem for simple functions
◊ implement golden section search in Matlab
◊ review programming in Matlab.
Interpolation

◊ When you take data, how do you predict what other data points might be?

◊ Two techniques are:

  • Linear Interpolation
    – Assume data follows a straight line between adjacent measurements

  • Cubic Spline Interpolation
    – Fit a piecewise 3rd degree polynomial to the data points to give a “smooth” curve to describe the data.
What is the corresponding value of $y$ for this $x$?
Linear Interpolation

- Assume the function between two points is a straight line.

How do you find a point in between?

X=2, Y=?
Linear Interpolation – Connect the points with a straight line to find $y$
MATLAB Code

◊ interp1 is the MATLAB function for linear interpolation
◊ First define an array of x and y
◊ Now define a new x array, that includes the x values for which you want to find y values
◊ new_y=interp1(x,y,x_new)

◊ Exercise: How would you implement the interp1 function yourself? What control structures would you use?
The University of Western Australia

MATLAB 7.12.0 (R2011a)

Command Window

```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> interp1(x,y,3.5)
ans =
    4
```

Measured Data

- **x-axis**: 0, 1, 2, 3, 4, 5, 6
- **y-axis**: 15, 10, 9, 6, 2, 0
```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> interp1(x,y,3.5)
ans =
    4
>> new_x = 0:0.2:5;
>> new_y = interp1(x, y, new_x)
new_y =
   Columns 1 through 4
   15.0000  14.0000  13.0000  12.0000
   Columns 5 through 8
   11.0000  10.0000   9.8000   9.6000
   Columns 9 through 12
   Columns 13 through 16
   7.8000   7.2000   6.6000   6.0000
   Columns 17 through 20
   5.2000   4.4000   3.6000   2.8000
   Columns 21 through 24
   2.0000   1.6000   1.2000   0.8000
   Columns 25 through 26
   f2
```
Both measured data points and interpolated data were plotted on the same graph. The original points were modified in the interactive plotting function to make them solid circles.
Cubic Spline Interpolation

- A cubic spline creates a smooth curve, using a third degree polynomial.
- The curve does not correspond to a single cubic. Instead it is a set of cubic polynomials that meet at the measured data points.
- In fact, the polynomials are continuous up to their 2\textsuperscript{nd} derivative.
- Finding these polynomials can be reduced to the problem of solving a set of simultaneous linear equations, (but this is beyond the scope of this unit).
- Note, that all the original points remain on the curve.
We can get an improved estimate by using the spline interpolation technique.
```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> interp1(x,y,3.5,'spline')
ans =
    3.9417
>> new_x = 0:0.2:5;
>> new_y_spline = interp1(x,y,new_x,'spline');
>> plot(x,y, new_x,new_y_spline, '-o')
```
Cubic Spline Interpolation. The data points on the smooth curve were calculated. The data points on the straight line segments were measured. Note that every measured point also falls on the curved line.
Other Interpolation Techniques

◊ MATLAB includes other interpolation techniques including
  • Nearest Neighbor
  • Cubic
  • Two dimensional
  • Three dimensional

◊ Use the help function to find out more if you are interested
Curve Fitting

- There is scatter in all collected data.
- We can estimate the equation that represents the data by “eyeballing” a graph.
- There will be points that do not fall on the line we estimate.
- Curve fitting is used when we want to match an analytical (or symbolic) model to a set of measurements which may contain some error.
- Interpolation is used when we assume that all data points are accurate and we would like to infer new intermediate data points.
This line is just an “eyeballed guess”
Least Squares

- Finds the “best fit” straight line
- Minimizes the amount each point is away from the line
- It’s possible none of the points will fall on the line
- Linear Regression is used to define the line that minimizes the square of the errors (the distance each point is from the value predicted by the line).
- The same method is used by the slope function in Excel.
Polynomial Regression

- Linear Regression finds a straight line, which is a first order polynomial.
- If the data doesn’t represent a straight line, a polynomial of higher order may be a better fit.
- In MATLAB you do both linear and polynomial regression the same way – the only difference is the order.
polyfit and polyval

- **polyfit** finds the coefficients of a polynomial representing the data
  - Coef = polyfit(x,y,2); will set coef = [a,b,c] where a\(x^2\)+b\(x\)+c is the quadratic that minimizes the square of the error.

- **polyval** uses those coefficients to find new values of y, that correspond to the known values of x
  - newy = polyval(coef,X) will evaluate the polynomial corresponding to coef for each point in X and return the corresponding y-values. The degree of the polynomial is determined by the length of coef.
The coefficients of the first order polynomial describing the best fit line is:

\[ y = -2.9143x + 14.2857 \]

To evaluate how close the fit you've achieved by taking the difference between the measured and calculated points, you can use MATLAB to calculate the residuals. Here's how you can do it:

```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> coef = polyfit(x, y, 1);
>> new_y = polyval(coef, x);
```

The coefficients of the polynomial are `coef = [-2.9143 14.2857]`. To calculate the residuals, you can subtract the calculated values from the measured values:

\[ \text{Residuals} = y - \hat{y} \]

This will give you an idea of how well the polynomial fits the data.
Least Squares Fit

\[ \sum (y - y_{\text{calc}})^2 \]

The polyfit function minimizes this number
The University of Western Australia

Second Order Fit

\[ y = 0.0536 \times x^2 - 3.1821 \times x + 14.4643 \]

MATLAB Code:

```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> coef = polyfit(x, y, 2);
>> coef = [0.0536, -3.1821, 14.4643]
>> new_y = polyval(coef, x);
>> plot(x, y, 'o', x, new_y, '-x')
```
A fifth order polynomial gives a perfect fit to 6 points.
Improve your graph by adding more points

```matlab
>> x = 0:5;
>> y = [15, 10, 9, 6, 2, 0];
>> coef = polyfit(x,y,5);
>> new_x = 0:0.1:5;
>> new_y = polyval(coef, new_x);
>> plot(x,y,'o', new_x, new_y)
```
An example from the labs

◊ We can read in the Excel file from Lab 2 using the xlsread function:
  ◊ data = xlsread('Rings.xls');
◊ We can extract X and Y arrays and find the line of best fit:
  ◊ X = X(2:78);
  ◊ Y = Y(2:78);
  ◊ coef = polyfit(X,Y,1);
  ◊ NewY = polyval(coef,X)
$y = 0.074x + 1.8$
Using the Interactive Curve Fitting Tools

◊ MATLAB 7 includes new interactive plotting tools.

◊ They allow you to annotate your plots, without using the command window.

◊ They include
  • basic curve fitting,
  • more complicated curve fitting
  • statistical tools
Use the curve fitting tools…

◊ Create a graph
◊ Making sure that the figure window is the active window select
  • Tools-> Basic Fitting
  • The basic fitting window will open on top of the plot
```matlab
>> x = 0:5;
>> y = [0,20,60,68,77,110];
>> plot(x,y,'o')
>> axis([-1,7,-20,120])
```
Plot generated using the Basic Fitting Window

\[ y = 21x + 3.8 \]
\[ y = 1.1x^3 - 9.3x^2 + 41x - 3.1 \]
Residuals are the difference between the actual and calculated data points.
Basic Fitting Window

- Select date: data 1
- Center and scale x data
- Plot fits:
  - Check to display fits on figure
  - Show equations
  - Significant digits: 2
  - Plot residuals
    - Bar plot
    - Subplot
  - Show norm of residuals
- Numerical results:
  - Fit: cubic
  - Equation:
    \[ y = p1x^3 + p2x^2 + p3x + p4 \]
  - Coefficients:
    - \( p1 = 1.1019 \)
    - \( p2 = -9.1175 \)
    - \( p3 = 41.192 \)
    - \( p4 = -3.6556 \)
  - Norm of residuals: 15.385

Save to workspace...
You can also access the data statistics window from the figure menu bar. Select **Tools->Data Statistics** from the figure window.

This window allows you to calculate statistical functions interactively, such as mean and standard deviation, based on the data in the figure, and allows you to save the results to the workspace.
Golden Section Optimisation

◊ This is a common numerical problem.

◊ The optimisation problem: given some function $f$ which depends on one or more variables, find the values of these variables where $f$ is a minimum or maximum.
Terms of optimisation

◊ Extrema of the function in the interval \([x_1 .. x_2]\) are:

  • A, E, and G are local maxima. B and D are local minima.
  • C is the global maximum.
  • F is the global minimum.

◊ The tasks of finding the minimum or maximum of a function are related:

  \[
  \text{minimum of } f = \text{maximum of } (-f)
  \]
How to find global extrema

◊ There is very little theory about finding *global* extrema in general.

◊ Most literature really only applies to finding *local* extrema.

◊ There are two standard heuristics (rules of thumb) for finding global extrema:
  • Find local extrema starting from widely varying values of the variables and pick the most extreme of these.
  • Perturb a local extrema by taking some finite step away from it and then seeing if your algorithm finds a better point or "always" returns to the same point.
What is a good optimisation algorithm?

◊ Often the cost of evaluating the function $f$ (and perhaps its partial derivatives) can dominate the overall computation. In this case, a good algorithm is one that evaluates $f$ as few times as possible.

◊ There are two main classes of algorithms:
  • Those that only need evaluations of the function value.
  • Those that also need evaluations of the derivatives of the function.

◊ The second class of algorithms are generally more powerful, but the extra cost evaluating derivatives may not make it worthwhile.
Golden section search in one dimension

◊ We will start with looking at optimisation of functions of one variable. The golden section search is an algorithm that only uses evaluations of the function value.

◊ The first step is to find three points that *bracket* a minimum.

◊ We need three points such that \( a < b < c \) and that \( f(a) > f(b) < f(c) \).
We call \((a, b, c)\) a **bracketing triplet**.

Having bracketed a minimum we pick a new point, \(x\), either between \(a\) and \(b\) or \(b\) and \(c\).

Let’s assume that \(x\) lies between \(b\) and \(c\).

If \(f(b) < f(x)\), we form the new bracketing triplet \((a, b, x)\):
Diagram illustration (case 2)

◊ If \( f(b) > f(x) \), we form the new bracketing triplet \( (b, x, c) \):
Bracketing algorithm logic

◊ At any point in the algorithm, the middle point of the bracketing triplet represents the best minimum found so far.

◊ We keep bracketing until the distance between the two outer points becomes small.

◊ Two questions arise:
  1. How small is small?
  2. Where should the new point be placed in generating a new bracketing triplet?
When to stop bracketing

◊ You might think the limiting case to stop bracketing is:

\[ b (1 - \text{EPS}) < b < b (1 + \text{EPS}) \]

where EPS is the machine's floating point precision (about \(10^{-16}\) in double precision). But, not so!

◊ If we look at the way the function varies close to \(b\) (close to a minimum) using Taylor's theorem, we have:

\[
\frac{1}{1} f(x) = f(b) + f'(b) (x-b) + \frac{1}{2} f''(b) (x-b)^2 + \ldots
\]

Note that the term \(f'(b) (x-b)\) will be approximately zero since the slope of the function will be close to zero at an extrema.

◊ Hence, the difference between \(f(x)\) and \(f(b)\) is due to the higher order derivative terms.
Second derivative – the dominant

Of these higher derivative terms, the second derivative is likely to be the dominant term.
Derivation steps to find when to stop

◊ We observe \( f(x) \) will be indistinguishable from \( f(b) \) if:

\[
\frac{1}{2} \frac{f''(b)(x - b)^2}{f(b)} < \text{EPS}
\]

◊ Rearranging, we get:

\[
|x - b| < \sqrt{\frac{2 \text{EPS}|f(b)|}{f''(b)}}
\]

◊ Rewriting, we get:

\[
|x - b| < |b|\sqrt{\text{EPS}} \sqrt{\frac{2|f(b)|}{b^2|f''(b)|}}
\]

◊ For most functions, the magnitude of the second square-root term is of order 1. Hence, we have:

\[
|x - b| < |b|\sqrt{\text{EPS}}
\]

◊ Therefore, it is hopeless to form a bracketing interval of width less than \( \sqrt{\text{EPS}} \) times the bracketing interval's central value.
Generating a new bracketing triplet

◊ What is the best strategy for choosing a new bracketing point \( x \) given \( a, b, \) and \( c \)? Suppose \( b \) is a fraction \( W \) between \( a \) and \( c \):

\[
\begin{array}{c}
\underbrace{a \quad b \quad c} \\
W(c-a)
\end{array}
\]

◊ where

\[
\frac{b-a}{c-a} = W
\]

◊ Rewrite, we have

\[
\frac{c-b}{c-a} = 1 - W
\]
Generating a new bracketing triplet

◊ Suppose the next point $x$ is an additional fraction $Z$ beyond $b$:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>x</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(c–a)</td>
<td>Z(c–a)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

◊ where

$$\frac{x - b}{c - a} = Z$$
Generating a new bracketing triplet

◊ The next bracketing triplet will either be of length $W + Z$ relative to the current one, or of length $1 - W$.
◊ For maximum efficiency, we would want the segment size to be the same no matter which triplet is formed.
◊ Hence, we want:

$$W + Z = 1 - W$$
$$Z = 1 - 2W$$

◊ If the relative proportions of the points in the new interval are to be retained, we need:

$$|b - a| = |c - x|$$

◊ This implies $x$ must be placed in the larger segment.
Generating a new bracketing triplet

◊ This scale similarity means that $x$ should be the same fraction of the way from $b$ to $c$ as $b$ was from $a$ to $c$.

\[
\frac{Z}{1-W} = W
\]

◊ Hence, we want:

\[
\frac{1 - 2W}{1 - W} = W
\]

◊ Which implies

\[
W^2 - 3W + 1 = 0
\]

◊ Solving for $W$, we get:

\[
W = \frac{3 - \sqrt{5}}{2} = 0.38197
\]
Golden Ratio

◊ Thus, the optimal bracketing interval \((a, b, c)\) has \(b\) placed a fractional distance of 0.38197 from one end and 0.61803 from the other end.

◊ These two numbers are in the *golden ratio*:

\[
\frac{1-W}{W} = \frac{0.61803}{0.38197} = 1.6180
\]

◊ Hence, given a bracketing triplet of points, the next point that should be tried is a fraction 0.38197 into the larger segment from the centre point.

◊ Each step of the algorithm shrinks the size of the bracketing interval by 0.61803.

◊ If the initial three points have segments that do not form a golden ratio, this procedure will rapidly converge to a state...
1D Golden section minimisation (code): 1

% GOLDEN: 1D Golden section minimisation search.
%
% Usage:       [fmin, xmin] = golden(func, ax, bx, cx, tol)
%
% Arguments:  func - A string specifying the name of a function or
%              M-file that is evaluated on a single variable.
%             ax   - Left bracketting point.
%             bx   - Middle bracketting point.
%             cx   - Right bracketting point.
%             Note: we require ax < bx < cx and f(ax) > f(bx) < f(cx)
%             tol  - The terminating bracketting width.
%             The tolerance should be > sqrt(eps).
%
% Returns:    fmin - The minimum value of the function found.
%             xmin - The value that minimises the function.
function [fmin, xmin] = golden(func, ax, bx, cx, tol)

    if (ax > bx) | (bx > cx)
        error('ax, bx and cx must be ascending order');
    end

    if tol < eps
        warning('Tolerance should be sqrt(eps) or greater.');
        tol = sqrt(eps);
    end

    fa = feval(func, ax); % The feval function evaluates the
    fb = feval(func, bx); % function specified by the string
    fc = feval(func, cx); % func at the specified points.

    if (fb > fa) | (fb > fc)
        error('ax and cx do not bracket a minimum.');
    end
\[ W = \frac{3 - \sqrt{5}}{2}; \quad \text{% The golden fraction.} \]

\[ x_0 = ax; \quad x_3 = cx; \quad \text{% The function keeps track of} \]
\[ \text{% four points: } x_0 \ x_1 \ x_2 \ x_3. \]

\[
\begin{align*}
\text{if } &\text{abs}(cx - bx) > \text{abs}(bx - ax) \quad \text{% Make } x_0 \text{ to } x_1 \text{ the smaller segment.} \\
&x_1 = bx; \\
&x_2 = bx + W*(cx-bx); \quad \text{% Construct } x_2 \text{ between } b \text{ and } c. \\
\text{else} \\
&x_2 = bx; \\
&x_1 = bx - W*(bx-ax); \quad \text{% Construct } x_1 \text{ between } a \text{ and } b.
\end{align*}
\]

\[
\begin{align*}
&f_1 = \text{feval}(func, x_1); \quad \text{% Evaluate function at } x_1 \text{ and } x_2. \\
&f_2 = \text{feval}(func, x_2); \quad \text{% We never need the values at the} \\
&\quad \text{% end points.}
\end{align*}
\]
while abs(x3-x0) > tol*(abs(x1)+abs(x2))
    if f2 < f1                    % Use x1 x2 x3 to form new bracket.
        x0 = x1; f0 = f1;
        x1 = x2; f1 = f2;
        x2 = x1 + W*(x3-x1); f2 = feval(func, x2);
    else                        % Use x0 x1 x2 to form new bracket.
        x3 = x2; f3 = f2;
        x2 = x1; f2 = f1;
        x1 = x2 - W*(x2-x0); f1 = feval(func, x1);
    end
end
if f1 < f2                        % Finished - get the better
    fmin = f1; xmin = x1;
else
    fmin = f2; xmin = x2;
end
end
References:


◊ Code prepared by Peter Kovesi and modified by Luigi Barone.
Matlab Review

◊ This completes the section of the unit covering Matlab
◊ There is a great deal of Matlab functionality available, but you should be aware of the basics:
  • How data is represented using arrays and matrices
  • How variable are created and modified
  • How functions are defined and used
  • How conditions (if), loops (for, while), matrix operations, and IO can be combined to define a process.
  • How to plot and visualize data.