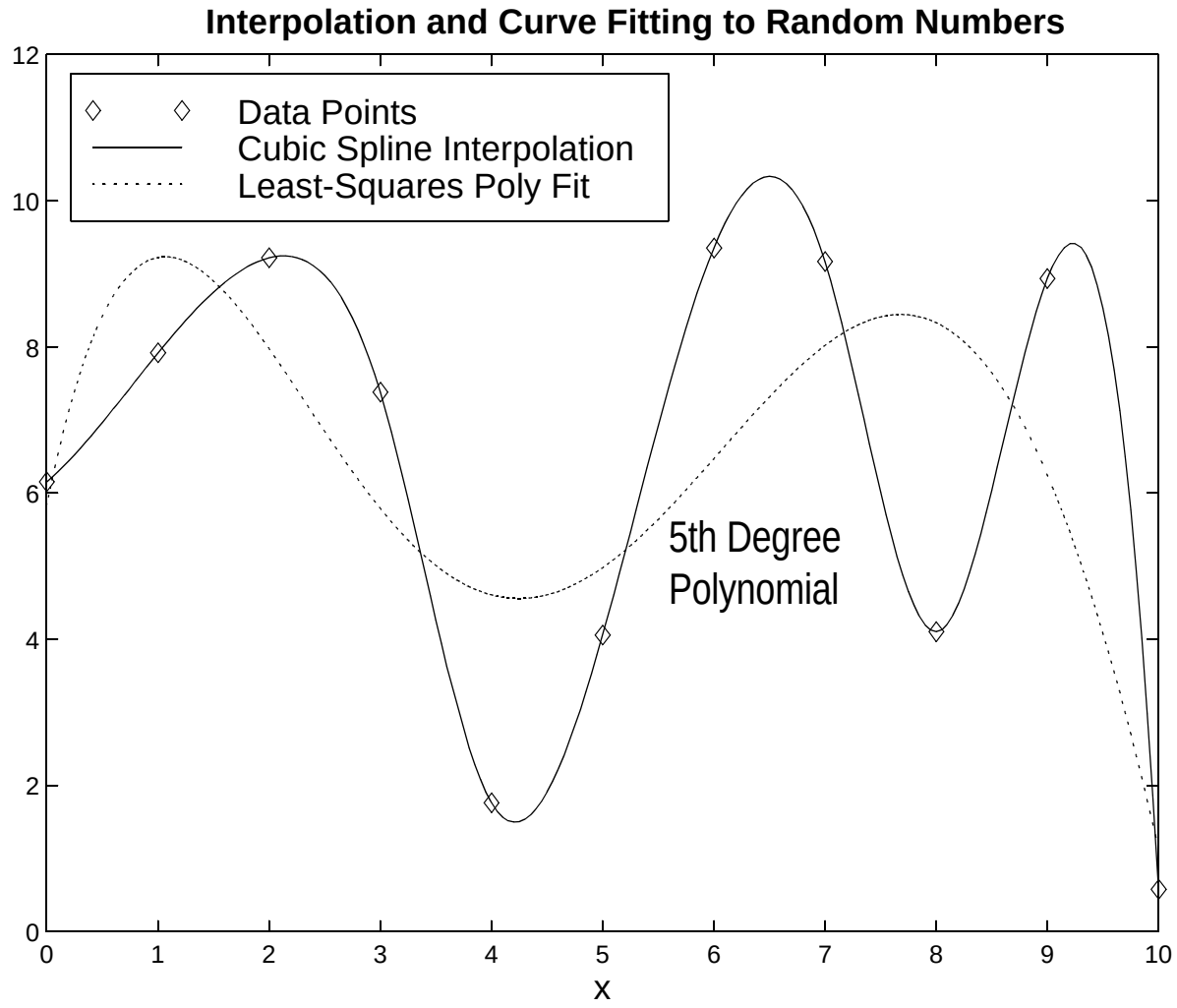
6 Interpolation and Curve Fitting

Overview

Given a set of data that results from an experiment (simulation based or otherwise), or perhaps taken from a real-life physical scenario, we assume there is some function $f(x)$ that passes through the data points and perfectly represents the quantity of interest at all non-data points. With *interpolation* we seek a function that allows us to approximate $f(x)$ such that functional values between the original data set values may be determined (estimated). The interpolating function typically passes through the original data set. With curve fitting we simply want a function that is a *good fit* (typically a *best fit* in some sense) to the original data points. With curve fitting the approximating function does not have to pass through the original data set.

- An example of interpolation using *spline* functions and *least-squares* curve fitting using a fifth degree polynomial is shown in the following figure
- The data set is a set of 10 random numbers generated using `10*rand(1,10)`
  - Note that the spline interpolation passes through the data points while the curve fit does not

– A higher degree polynomial would presumably give a better fit



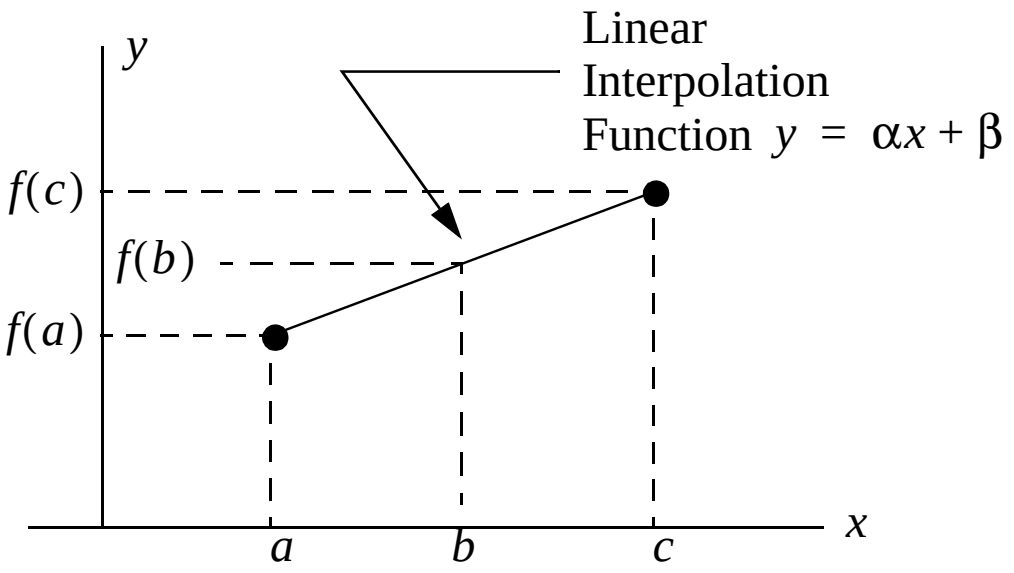
Interpolation

- The simplest type of interpolation is *linear interpolation*, which simply connects each data point with a straight line
- The polynomial that links the data points together is of first degree, e.g., a straight line

- A more exotic interpolation scheme is to connect the data points using third degree or cubic polynomials

Linear Interpolation

- Given data points $f(c)$ and $f(a)$, where $c > a$
- We wish to estimate $f(b)$ where $b \in [a, c]$ using linear interpolation



- The linear interpolation function for functional values between a and c can be found using similar triangles or by solving of system of two equations for two unknowns
- The slope intercept form for a line is

$$y = f(x) = \alpha x + \beta, x \in [a, c] \tag{6.1}$$

- As *boundary conditions* we have that this line must pass through the point pairs $(a, f(a))$ and $(c, f(c))$

- Thus we can write

$$f(a) = \alpha a + \beta \tag{6.2}$$

$$f(c) = \alpha c + \beta \tag{6.3}$$

- Now we solve for α and β by using (6.3) to solve for β and substituting this expression into (6.2)

$$\beta = f(c) - \alpha c$$

so

$$f(a) = \alpha a + f(c) - \alpha c$$

or

$$\alpha = \frac{f(c) - f(a)}{c - a} \tag{6.4}$$

- With α known we now solve for β

$$\beta = f(c) - \frac{f(c) - f(a)}{c - a} c \tag{6.5}$$

- In summary, the linear interpolation formula is to obtain $f(b)$ for $a \le b \le c$

$$f(b) = \frac{f(c) - f(a)}{c - a} b + \left[f(c) - \frac{f(c) - f(a)}{c - a} c \right] \tag{6.6}$$

- Note: With some algebraic manipulation we can get the equation in the text which is

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

- Given a data set, we can perform linear interpolation between each pair of data points to any desired resolution using the MATLAB function `interp1`
- Understanding how linear interpolation works is still very important if you are writing a custom algorithm or want to check the results of a MATLAB calculation
- The function

```
y_new = interp1(x,y,x_new,'linear');
```

  returns a vector `y_new` of the same size as `x_new`
  - The vectors `x` and `y` contain the raw data
  - If the fourth function argument, `'linear'`, is omitted `interp1` defaults to linear interpolation
  - It is assumed that the `x_new` vector has the same minimum and maximum values as `x`, but in `x_new` the points can be spaced to any desired resolution
  - The vector `y_new` contains the linearly interpolated values corresponding to `x_new`, and of course will match the original `y` vector values at the corresponding `x` values

Example: Suppose we have the following velocity versus time data (a car accelerating from a rest position)

Table 6.1: Car velocity data

Time, s	Velocity, m.p.h.
0	0
1	10

Table 6.1: Car velocity data (Continued)

Time, s	Velocity, m.p.h.
2	25
3	36
4	52
5	59

- Use MATLAB's interp1 function to estimate vehicle velocities on the interval [0,5] seconds with resolution of 0.01 seconds

```
» x = 0:5;
» y = [0 10 25 36 52 59];
» x_new = 0:.01:5;
» y_new = interp1(x,y,x_new,'linear');
» plot(x,y,'d')
» hold
Current plot held
» plot(x_new,y_new)
» title('Velocity versus Time','fontsize',18)
» ylabel('Miles per Hour','fontsize',14)
» xlabel('Time, seconds','fontsize',14)
» legend('Data Points','Linear Interpolation')
```

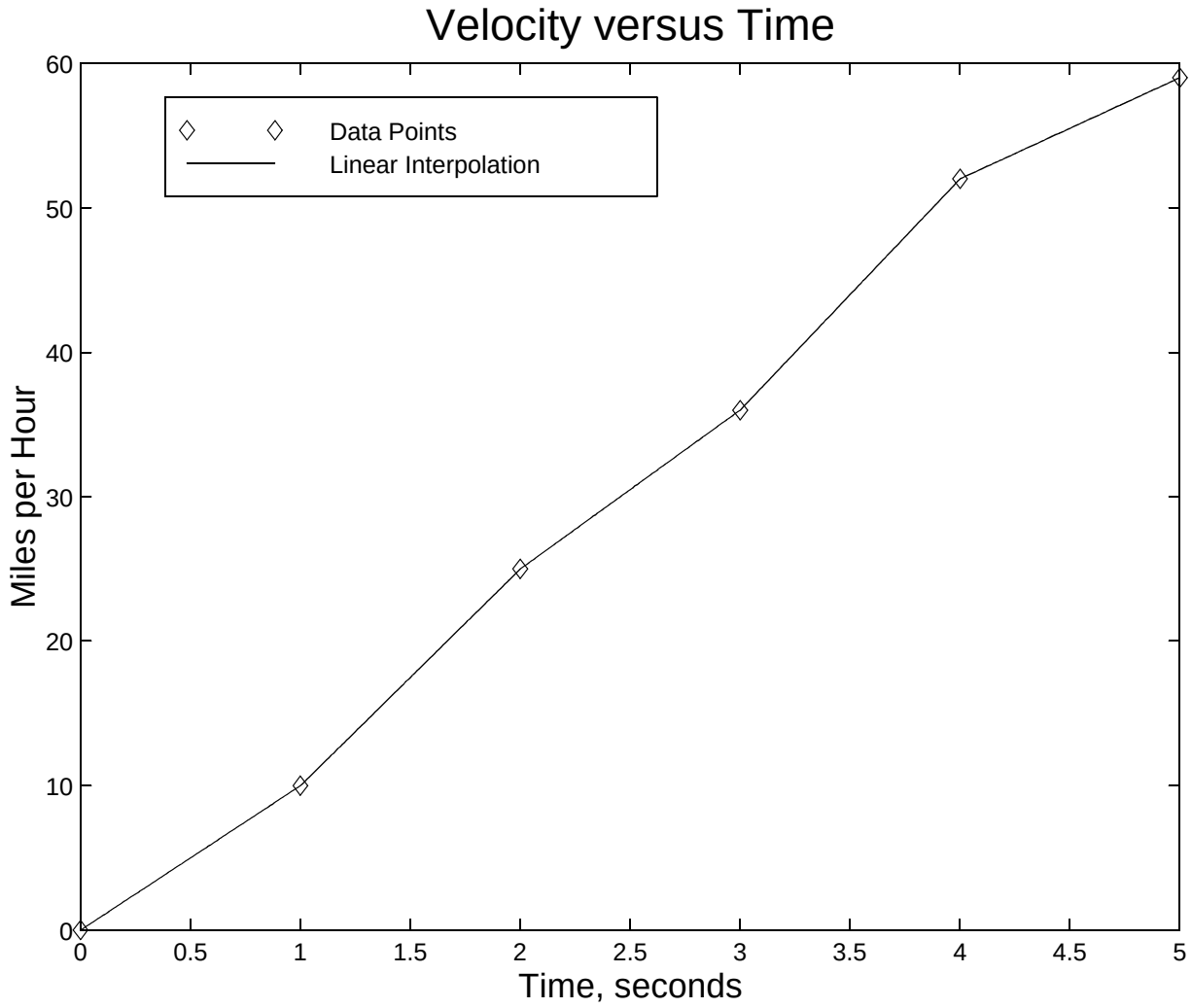
- To check the calculation suppose we wish to find $f(1.5)$, that is the vehicle velocity at 1.5 seconds
- Using (6.6) we enter the data point values for one and two seconds

$$f(1.5) = \frac{25-10}{2-1}1.5 + \left[25 - \frac{25-10}{2-1}2\right] = 17.5$$

- Does MATLAB agree?

```
» y_new = interp1(x,y,1.5,'linear')

y_new =   17.5000 %%% YES!
```



- If the `y` vector is composed of columns of data, all corresponding to a single `x` vector of data, the `interp1` function will output a row vector for each value in the `x_new` vector

Cubic-Spline Interpolation

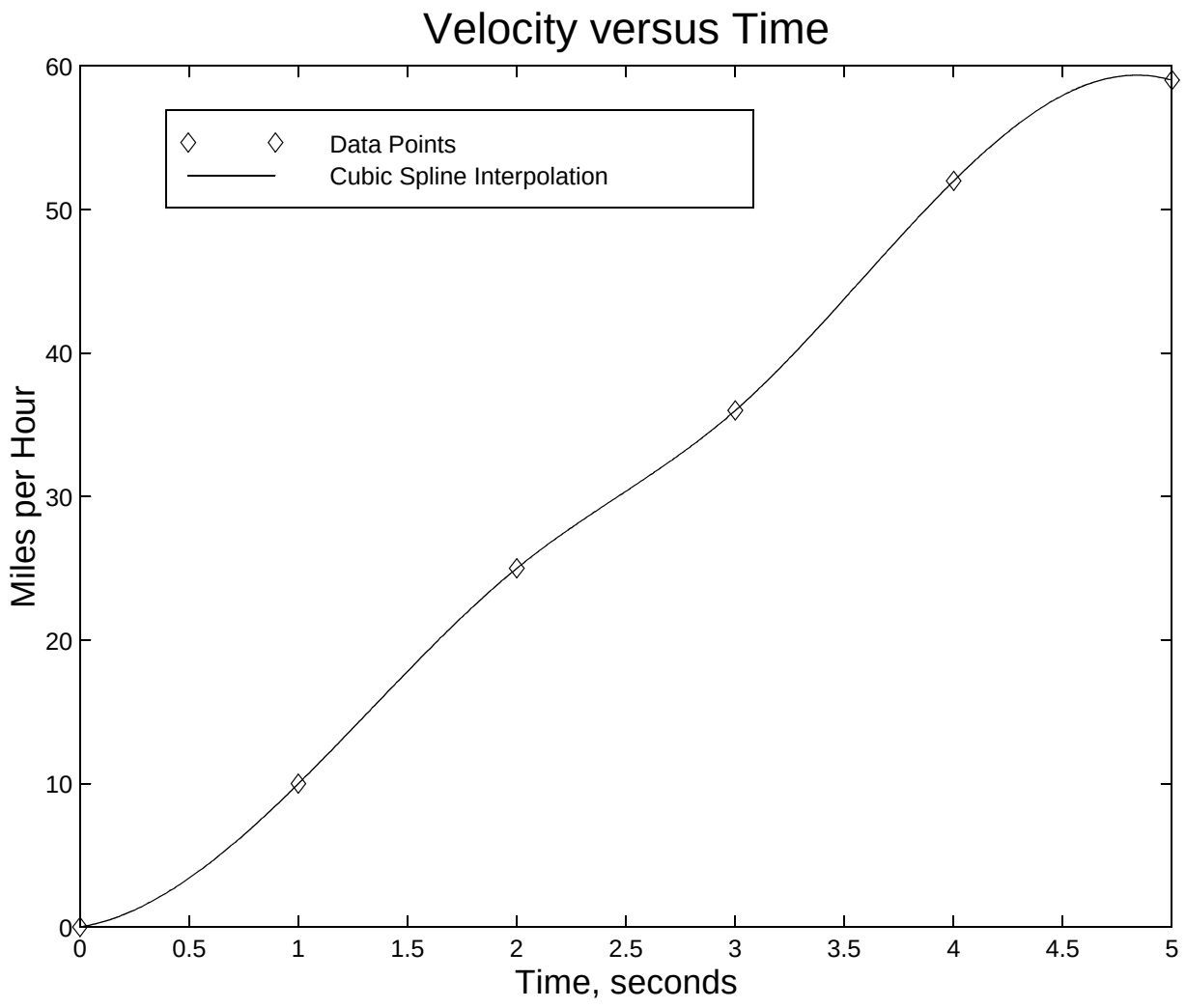
- As we can see from the previous example, linear interpolation produces a rather jagged result if the data points are not closely spaced and don't lie in a straight line
- An improved interpolation procedure is to replace the straight line connecting the data points with a third degree polynomial
- The third degree polynomial is of the form

$$y = f(x) = \alpha_3 x^3 + \alpha_2 x^2 + \alpha_1 x + \beta$$

  - As with linear interpolation a new set of coefficients must be used for each interval between the available data points
  - This polynomial is known as a cubic spline function
  - It is very special since the coefficients are chosen to give a smooth transition as we pass from one point and the next
  - This smooth behavior is accomplished by computing the polynomial coefficients for each interval using more than just the adjacent data points (recall linear interpolation uses just the interval end points to determine α and β)
- The MATLAB function `interp1` implements cubic spline interpolation by simply changing the fourth argument from `'linear'` to `'spline'`

Example: Repeat the vehicle velocity example, except now use cubic spline interpolation

```
» x = 0:5;
» y = [0 10 25 36 52 59];
» x_new = 0:.01:5;
» y_new = interp1(x,y,x_new,'spline');
» plot(x,y,'d')
» legend off
» hold
Current plot held
» plot(x_new,y_new)
» title('Velocity versus Time','fontsize',18)
» ylabel('Miles per Hour','fontsize',14)
» xlabel('Time, seconds','fontsize',14)
» legend('Data Points','Cubic Spline Interpolation')
```



- The `interp1` function provides several other interpolation modes as well

```
» help interp1 % A portion of the on-line help)
YI = INTERP1(X,Y,XI,'method') specifies alternate meth-
ods.
    The default is linear interpolation.  Available
methods are:

      'nearest' - nearest neighbor interpolation
      'linear'  - linear interpolation
      'spline'  - cubic spline interpolation
      'cubic'   - cubic interpolation

    All the interpolation methods require that X be
monotonic. Variable spacing is handled by mapping the
given values in X,Y, and XI to an equally spaced domain
before interpolating. For faster interpolation when X is
equally spaced and monotonic, use the methods '*linear',
'*cubic', '*nearest', or '*spline'.  Faster linear
interpolation when X is non-uniformly spaced see
INTERP1Q.
```

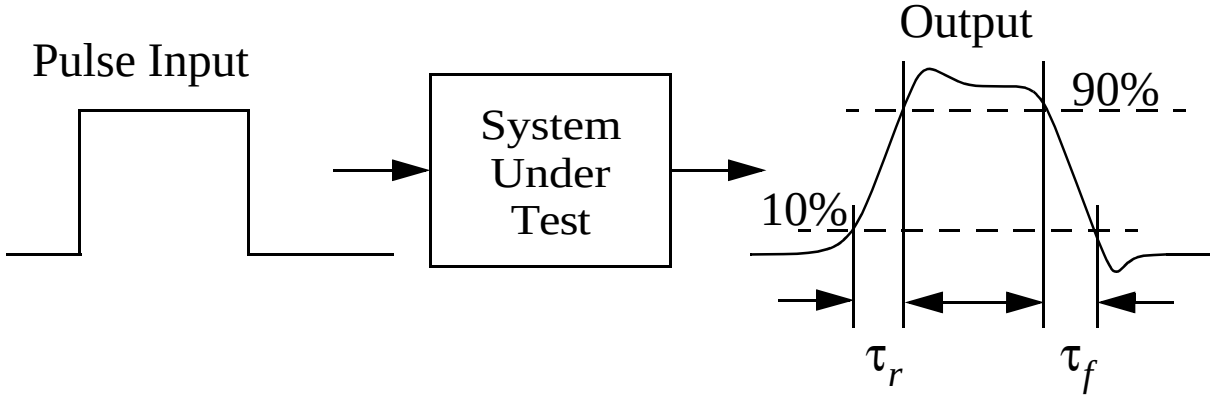
- In the Practice! on p. 170 you explore another aspect of interpolation
  - Thus far we have interpolated y-axis values by allowing for an arbitrary resolution to be specified on the x-axis
  - It is perfectly ok to interchange the roles of x and y to allow x-axis values to be interpolated by allowing for an arbitrary resolution to be specified on the y-axis (part 3)
  - In part 2 of Practice p. 170, also **hand calculate** the linear interpolation of the temperature at 0.3 seconds

Problem Solving Applied: Rise-Time and Fall-Time Determination

In the design of electronic devices/circuits/systems that process pulse type signals, we are often interested in a performance characterization known as rise-time/fall-time.

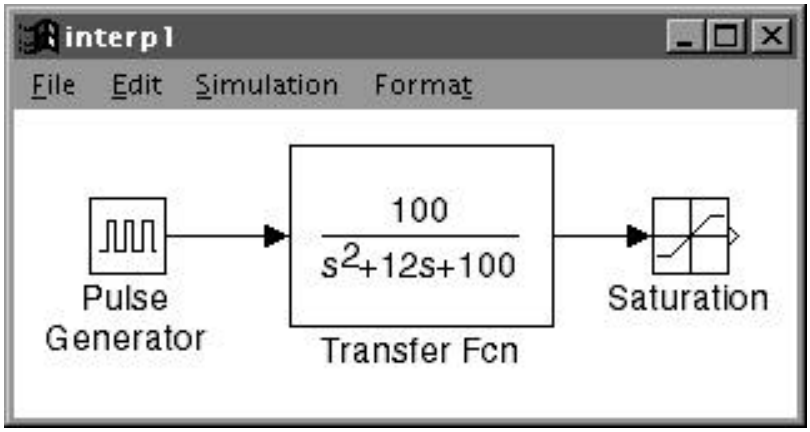
Rise-Time and Fall-Time Definitions

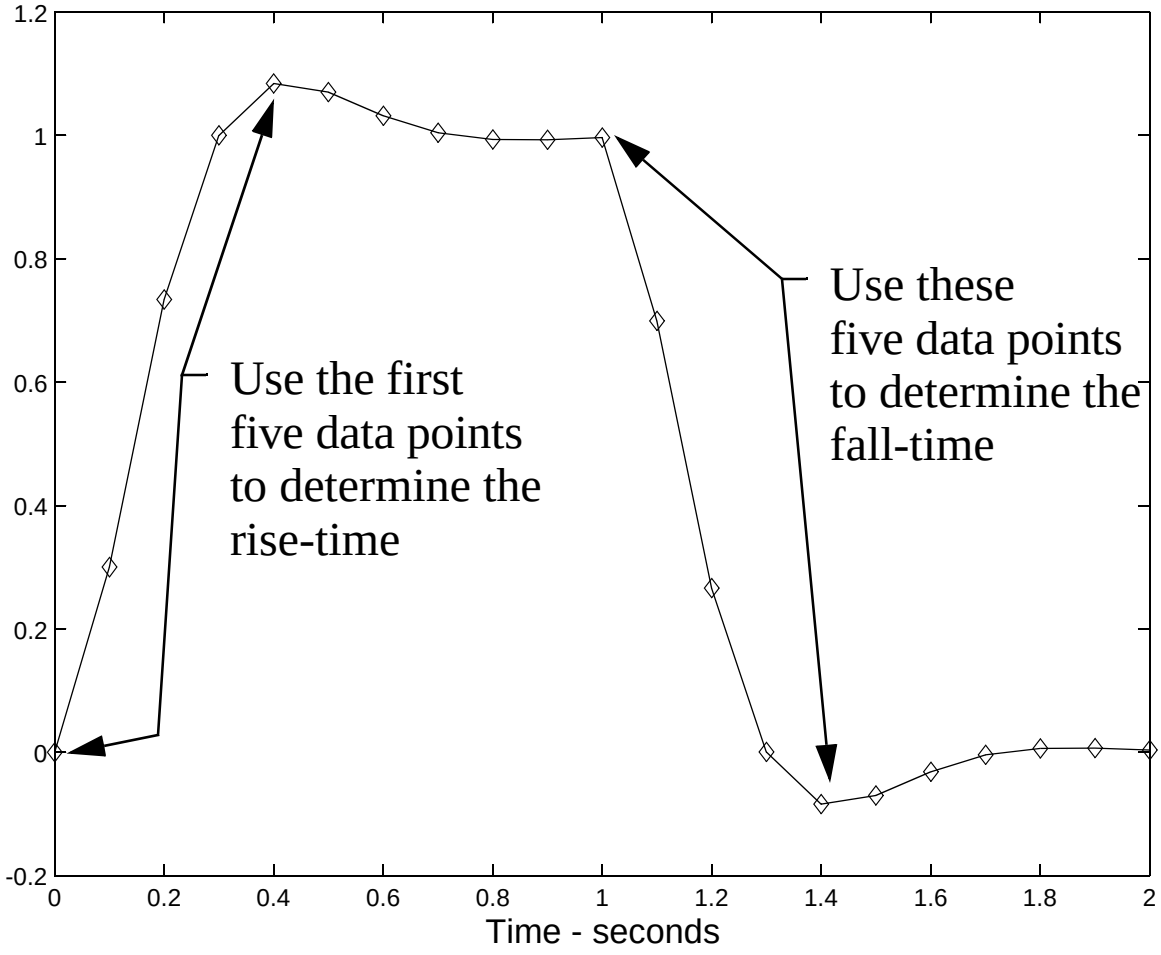
- For an amplifier the rise-time is typically defined as the time required for the output of the amplifier to change from a low output level (say an at rest value of zero volts) to some high output level, in response to a step input signal
- The concept of rise-time also applies to digital logic gates and flip-flops when one defines the switching time for logical zero-one or one-zero transitions
- Traditionally the rise-time, τ_r, is the time it takes the output to transition from 10% of the final value to 90% of the final value
- The fall-time, τ_f, is the 10% to 90% transition time going in the opposite direction; in general they are different



Using Interpolation to find τ_r and τ_f

- Interpolation can help us solve for the rise-time and fall-time if the resolution of the experimental data is limited
- A system model is created in MATLAB *Simulink*





- The data points are stored in vectors `tout` and `simout`

```
» [tout, simout]

ans =
         0         0 %%
    0.1000    0.3000 %
    0.2000    0.7340 % Use these points for rise-time
    0.3000    1.0002 %
    0.4000    1.0844 %%
    0.5000    1.0701
    0.6000    1.0316
    0.7000    1.0041
    0.8000    0.9934
    0.9000    0.9932
    1.0000    0.9964 %%
    1.1000    0.6992 %
    1.2000    0.2664 % Use these points for fall-time
    1.3000    0.0004 %
    1.4000   -0.0840 %%
    1.5000   -0.0700
    1.6000   -0.0317
    1.7000   -0.0042
    1.8000    0.0065
    1.9000    0.0068
    2.0000    0.0036
```

- To obtain the rise-time and fall-time more precisely we will interpolate on the time axis (*x* and *y* axis roles reversed)
  - We have to be very careful to make sure that the y-axis input data is strictly increasing or strictly decreasing on the interval of interest

```
» t2 = interp1(simout(1:5),tout(1:5),0.9,'spline')

t2 =    0.2364

» t1 = interp1(simout(1:5),tout(1:5),0.1,'spline')

t1 =    0.0351

» tr = t2 - t1

tr =    0.2013 %% Rise-time in seconds

» t2 = interp1(simout(11:15),tout(11:15),0.1,'spline')

t2 =    1.2368

» t1 = interp1(simout(11:15),tout(11:15),0.9,'spline')

t1 =    1.0343

» tr = t2 - t1

tr =    0.2025 %% Fall-time in seconds
```

- We could have automated the above sequence of commands into an m-file

Problem Solving Applied: Robot Arm Manipulators

A control system application involving interpolation is in setting up a *safe* trajectory for a robot arm to move through in three dimensions. To simplify the problem, here we only consider motion in a plane (two dimensions). The robot arm must pass through a specified set of points (like data points in an interpolation problem).

- For the problem of interest the robot arm starts at a home position; has various intermediate positions; has a location where it must grasp an object; has a location where it must release the object; returns to the home position, perhaps via intermediate positions
- A data file which contains the required *x*-*y* coordinate data points, is saved with a third column of numbers which corresponds to the type of point it is, e.g.,

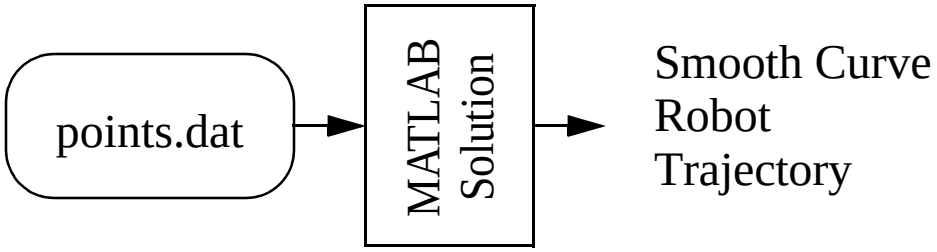
Table 6.2: Third column coding of robot arm 2D trajectory.

Code	Type of Point
0	Home position
1	Intermediate positions
2	Location of object to grasp
3	Location to release object

Problem Statement

Design a smooth curve using cubic-spline interpolation to guide the robot manipulator through a complete cycle of leaving the home location, grasping an object, moving the object to a release location, and returning to the home location.

Input/Output Description



Hand Calculation

Consider the data set shown below.

Table 6.3: Sample manipulator arm path nodes

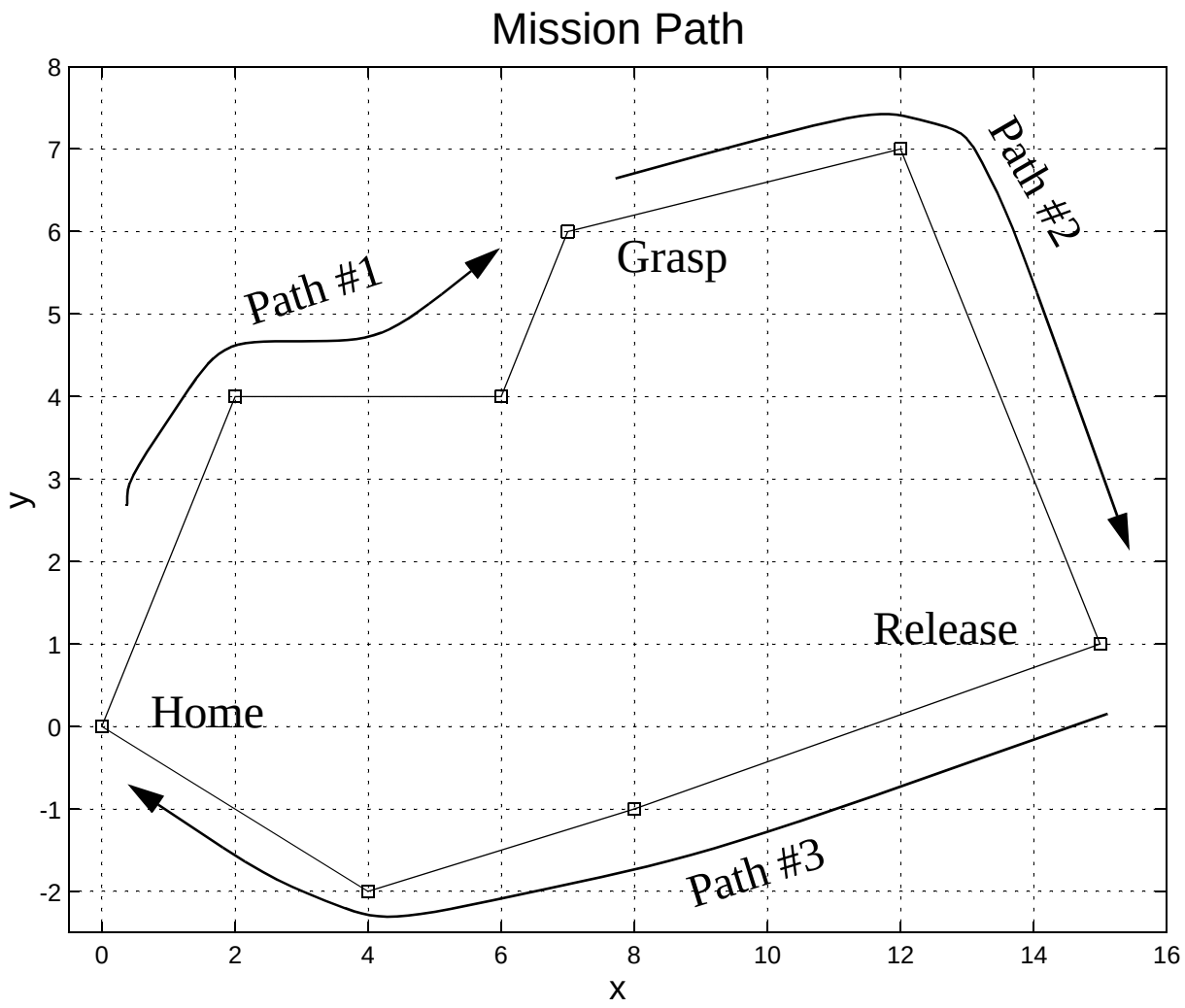
x	y	Code	Interpretation
0	0	0	Home position
2	4	1	Intermediate position
6	4	1	Intermediate position
7	6	2	Grasp object
12	7	1	Intermediate position
15	1	3	Release object

Table 6.3: Sample manipulator arm path nodes (Continued)

x	y	Code	Interpretation
8	-1	1	Intermediate position
4	-2	1	Intermediate position
0	0	0	Home position

- Load the points into MATLAB to parse them and then plot the points (connect the dots)

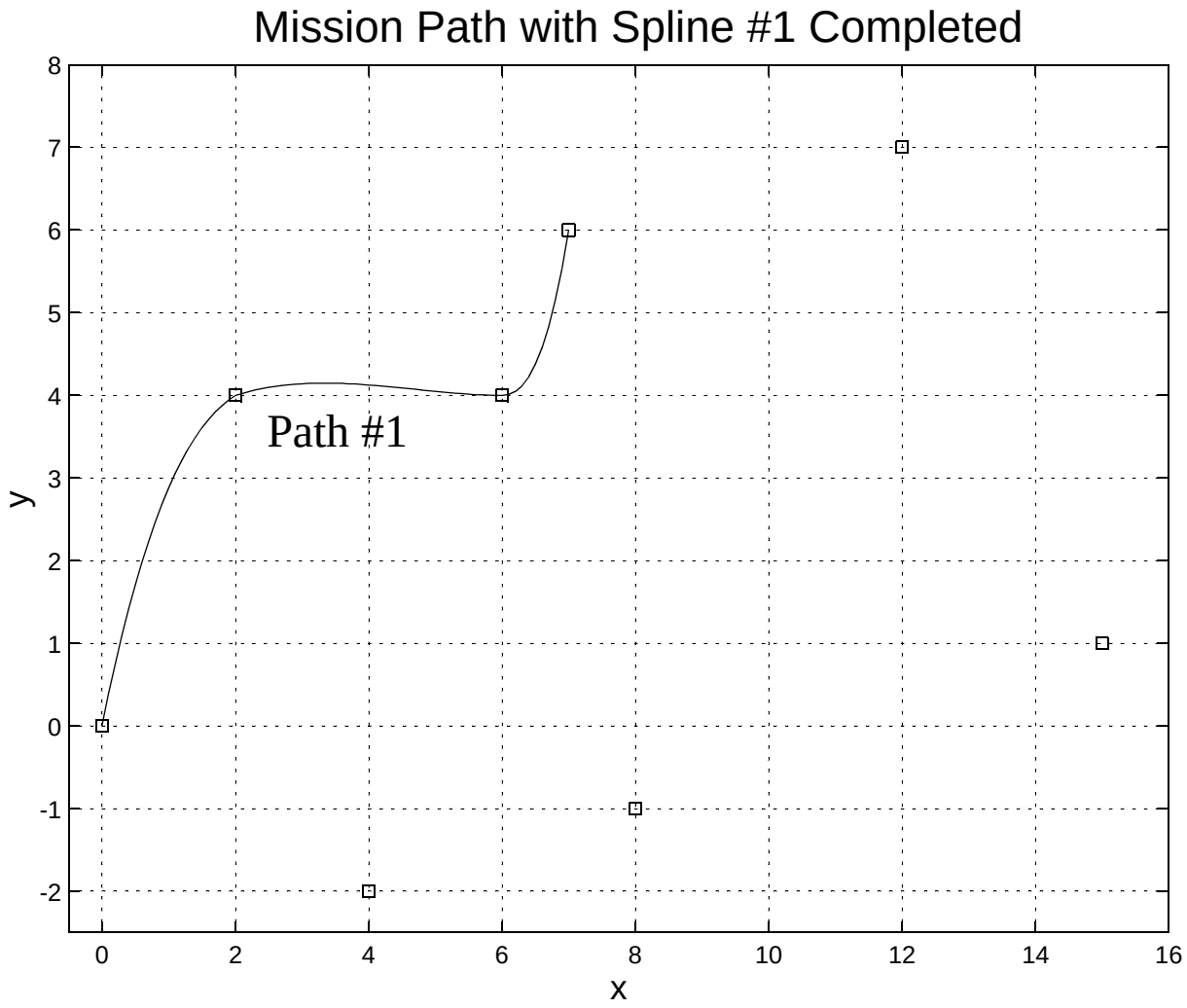
```
» load m0_points.dat;
» x = m0_points(:,1);
» y = m0_points(:,2);
» code = m0_points(:,3);
» plot(x,y,'s')
» axis([-.5 16 -4 10])
» grid
» hold
Current plot held
» plot(x,y)
» axis([-.5 16 -2.5 10])
» axis([-.5 16 -2.5 8])
» title('Mission Path','fontsize',18)
» ylabel('y','fontsize',14)
» xlabel('x','fontsize',14)
```



- A cubic-spline path needs to be created for each of three path types
  - From home to grasp
  - From grasp to release
  - From release to home
- Before writing the MATLAB code we need to calculate at least one path by hand, that is manually parsing the data files and loading the appropriate points into MATLAB's `interp1` function

- Path #1 runs from row one through row four in the data file
- The interpolation step size on the *x*-axis will be chosen as one tenth the minimum absolute change in *x*-coordinate over the entire data set, i.e., `min(abs(x(i+1)-x(i)))` as `i` runs from 1 to 9 – 1; for the hand calculation we can just observe this value by scanning the sample data
  - The minimum *x* step is 1

```
» t1 = 0:1/10:7; % Interpolated steps along the x-axis
» s1 = interp1(x(1:4),y(1:4),t1,'spline');
» hold
Current plot released
» plot(x,y,'s')
» axis([-.5 16 -2.5 8])
» hold
Current plot held
» plot(t1,s1)
» grid
» title('Mission Path with Spline #1 Completed',...
    'fontsize',18)
» ylabel('y','fontsize',14)
» xlabel('x','fontsize',14)
```



MATLAB Solution

Now we write a MATLAB script file that will automatically parse the data file into the three distinct paths and compute the corresponding cubic-splines.

- For the complete program see the text p. 173

Least-Squares Curve Fitting

- Given a set of data points a *best fit* to the data set may be a straight line
- A least-squares curve fit can be used to find a straight line such that the squared distance from each data point to the line is minimized
  - It could be that the line will not pass through any data point
- The curve that we fit to the data is not limited to straight lines
- In general the curve we fit to the available data may be a polynomial of any order, an exponential function, trigonometric function, etc.

Linear Regression

- With linear regression a linear equation, $y1 = mx + b$, is chosen that fits the data points such that the sum of the squared error between the data points and the line is minimized
  - The squared distance is computed with respect to the *y*-axis
- Given a set of data points

$$(x_k, y_k), k = 1, \ldots, N$$

the *mean squared error* (mse) is defined as

$$\text{mse} = \frac{1}{N}\sum_{k=1}^{N}[y_k - y1_k]^2 = \frac{1}{N}\sum_{k=1}^{N}[y_k - (mx_k + b)]^2$$

- The minimum mse is obtained for particular values of *m* and *b*
- Using calculus we compute the derivative of the mse with respect to both *m* and *b*
- Since the mse is nonnegative setting the derivative equal to zero occurs at the minimum mse
- The resulting *m* and *b* values give us the best straight line (linear) fit to the data
- The equations are rather complex, i.e.,

$$m_{\text{opt}} = \frac{N\sum_{k=1}^{N} x_k y_k - \left(\sum_{k=1}^{N} x_k\right)\left(\sum_{k=1}^{N} y_k\right)}{N\sum_{k=1}^{N} x_k^2 - \left(\sum_{k=1}^{N} x_k\right)^2}$$

and

$$b_{\text{opt}} = \frac{1}{N}\sum_{k=1}^{N} y_k - m_{\text{opt}}\frac{1}{N}\sum_{k=1}^{N} x_k$$

- MATLAB has a powerful function, `polyfit`, which handles the linear regression case as well as polynomial regression

Polynomial Regression

- A logical extension to the linear regression curve fit is to use a higher-order polynomial such as

$$y = f(x) = a_0 x^N + a_1 x^{N-1} + \ldots + a_{N-1} x + a_N$$

- A third degree or cubic polynomial fit, different from cubic splines since here there is only one, is of the form

$$y = g(x) = a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

  – Note: First degree polynomial regression is the same as linear regression

- The MATLAB function which performs polynomial regression is

```
a = polyfit(x,y,n);
```

  where

  – `a` is vector of `n+1` coefficients corresponding to decreasing powers of `x`

  – `x` and `y` are the supplied data points

  – `n` is the polynomial degree

- For simple linear regression we would use

```
a = polyfit(x,y,1);
```

- To plot the resulting curve fit we can use the `polyval` which was introduced in Chapter 3

Example: Consider the vehicle velocity data of Table 6.1.

- Try polynomial regression of orders 1 (linear), 2, 5, and 10

```
» x = 0:5;
» y = [0 10 25 36 52 59];
» x_ls = 0:0.01:5;
» y1 = polyval(polyfit(x,y,1),x_ls);
» subplot(221)
» plot(x,y,'s')
» hold
Current plot held
» plot(x_ls,y1)
» axis([0 5 0 60]); grid;
» title('First-Degree (Linear) Fit','fontsize',16)
» ylabel('mph','fontsize',14)
» subplot(222)
» y2 = polyval(polyfit(x,y,2),x_ls);
» plot(x,y,'s')
» hold
Current plot held
» plot(x_ls,y2)
» axis([0 5 0 60]); grid;
» title('Second-Degree Fit','fontsize',16)
» ylabel('mph','fontsize',14)
» subplot(223)
» y5 = polyval(polyfit(x,y,5),x_ls);
» plot(x,y,'s')
» hold
Current plot held
» plot(x_ls,y5)
» axis([0 5 0 60]); grid;
» title('Fifth-Degree Fit','fontsize',16)
» ylabel('mph','fontsize',14)
» xlabel('Time-Seconds','fontsize',14)
```

```
» subplot(224)
» y10 = polyval(polyfit(x,y,10),x_ls);
» plot(x,y,'s')
» hold
Current plot held
» plot(x_ls,y10)
» axis([0 5 0 60]); grid;
» title('Tenth-Degree Fit','fontsize',16)
» ylabel('mph','fontsize',14)
» xlabel('Time-Seconds','fontsize',14)
```