MATLAB - Lagrange and Spline Interpolations

Lagrange Interpolating Polynomial. The following MATLAB function receives the independent vector `x`, dependent vector `y` and the value xx in the range of vector x to estimate `y=f(xx)` using Lagrange interpolation.

```
function yint = Lagrange(x,y,xx)
% Lagrange: Lagrange interpolating polynomial
% yint = Lagrange(x,y,xx): Uses an (n ? 1) ?order
% Lagrange interpolating polynomial based on n data points
% to determine a value of the dependent variable (yint) at
% a given value of the independent variable, xx.
% input:
% x = independent variable
% y = dependent variable
% xx = value of independent variable at which the
% interpolation is calculated
% output:
% yint = interpolated value of dependent variable
n = length(x);
if length(y) ~= n,
    error('x and y must be same length');
end
s = 0;
for i = 1:n
    product = y(i);
    for j = 1:n
        if i ~= j
            product = product*(xx - x(j))/(x(i) - x(j));
        end
    end
    s = s + product;
end
yint = s;
end
```

Test this function for `T=[-40 0 20 50]` and `d = [1.52 1.29 1.2 1.09]` to estimate `d(15)`.

Spline Interpolations

Table lookup. A table lookup is useful for performing repeated interpolations from a table of independent and dependent variables. Suppose that you would like to set up an M-file that would use piecewise interpolation to estimate the function value at a particular number. The approach would be to pass in vectors containing all the data and have the M-file determine the bracket.

For ordered data, there are two simple ways to find the interval. The first is called a ***sequential search.*** This method involves comparing the desired value with each element of the vector in sequence until the interval is located. For example, for linear spline, consider the following code:

```
    function yi = TableLook(x, y, xx)
    n = length(x);
    if xx < x(1) | xx > x(n)
        error('Interpolation outside range')
    end
    % sequential search
    i = 1;
    while(1)
        if xx <= x(i + 1),
            break,
        end
    i = i + 1;
    end
    % linear interpolation
    yi = y(i) + (y(i + 1) - y(i))/(x(i + 1) - x(i))*(xx - x(i));
    end
```

For situations for which there are lots of data, the sequential sort is inefficient because it must search through all the preceding points to find values. In these cases, a simple alternative is the ***binary search.*** Here is an M-file that performs a binary search followed by linear interpolation:

```
function yi = TableLookBin(x, y, xx)
n = length(x);
if xx < x(1) | xx > x(n)
    error('Interpolation outside range')
end
% binary search
iL = 1;
iU = n;
while (1)
    if iU - iL  == 1,
        break,
    end
iM = fix((iL + iU) / 2); %fix(x)rounds x to the nearest integers towards zero.

if x(iM) < xx
    iL = iM;
else
    iU = iM;
end
end
% linear interpolation
yi = y(iL) + (y(iL + 1) - y(iL))/(x(iL + 1) - x(iL))*(xx -
x(iL));
end
```

Test these two codes for the following data to estimate `d(350)`

```
>> T = [-40 0 20 50 100 150 200 250 300 400 500];

>> d = [1.52 1.29 1.2 1.09 .946 .935 .746 .675 .616 .525 .457];
```

Cubic Spline. Cubic splines can be easily computed with the built-in MATLAB function, spline. It has the general syntax

```
>> yy = spline (x, y, xx)
```

where `x` and `y` = vectors containing the values that are to be interpolated, and `yy` = a vector containing the results of the spline interpolation as evaluated at the points in the vector `xx`. By default, spline uses the *not-a-knot condition*. In this condition, it forces continuity of the third derivative at the second and the next-to-last knots. However, if *y* contains two more values than *x* has entries, then the first and last value in *y* are used as the derivatives at the end points. Consequently, this option provides the means to implement the *clamped-end* condition.

Try the following:

```
x = linspace(-1,1,9);
y = 1./(1 + 25*x.^2);
xx = linspace(-1,1);
yy = spline(x,y,xx);

%Not-a-Knot condition
yr = 1./(1 + 25*xx.^2);
plot(x,y,'o',xx,yy,xx,yr,'--')

hold on
% Clamped-end condition
yc = [1 y  -4];
yyc = spline(x,yc,xx);
plot(x,y,'o',xx,yyc,xx,yr,'--')
```

Built-in `interp1` function

The built-in function `interp1` provides a handy means to implement a number of different types of piecewise one-dimensional interpolation. It has the general syntax

```
>> yi = interp1(x, y, xi, 'method')
```

where `x` and `y` = vectors containing values that are to be interpolated, `yi` = a vector containing the results of the interpolation as evaluated at the points in the vector `xi`, and '`method`' = the desired method. The various methods are

- '`nearest`'—nearest neighbor interpolation. This method sets the value of an interpolated point to the value of the nearest existing data point. Thus, the interpolation looks like a series of plateaus, which can be thought of as zero-order polynomials.

- '`linear`'—linear interpolation. This method uses straight lines to connect the points.

- '`spline`'—piecewise cubic spline interpolation. This is identical to the spline function.

- '`pchip`' and '`cubic`'—piecewise cubic Hermite interpolation.

  Note. If the '`method`' argument is omitted, the default is linear interpolation.

Try the following:

```
>> t = [0 20 40 56 68 80 84 96 104 110];

>> v = [0 20 20 38 80 80 100 100 125 125];

>> tt = linspace(0,110);

>> vl = interp1(t,v,tt);

>> v2=interp1(t,v,tt,’spline’)

>> plot(t,v,'o',tt,v1)

>> hold on

>> plot(t,v,'o',tt,v1)
```