8. INTERPOLATION

8.1 PRELIMINARIES

Interpolation in one dimension is defined in the following way ([p. 309] Heath 2002). When given *m* distinct data pairs -

$$(t_i, y_i), \quad i = 1{:}m,$$

for t arranged in ascending order ($t_{i-1} < t_i < t_{i+1}$), but not necessarily uniformly separated, determine a *reasonable* function, p(t), such that

$$p(t_i) \equiv y_i, \; i = 1{:}m.$$

Note the function, p(t), goes through the data points *exactly*. This function p(t) is called the *interpolant*. Occasionally additional constraints are employed, such as slopes, monotonicity, smoothness, or periodicity.

The interpolation function, p(t), is used ([p. 296] Ascher & Chen 2011) for:

- To estimate the underlying function, f(t), at t values other than the data abscissae, t_i, for i = 1:m.
- To determine approximations of derivatives and integrals of underlying function, f(t).

Prior to the introduction of personal computers, interpolation was utilized to determine functions from published data tables. Before the 1980's considerably more effort was expended on detailed[1] issues of interpolation.

Note that a fundamental distinction exists between interpolation and curve fitting. With interpolation data points, known very accurately, are fit *exactly* by smooth functions. These smooth functions often are polynomials of moderate order. Such interpolants fit the points exactly to within round-off error. When data possesses significant uncertainties data approximation, not interpolation, is used. In the case of noisy data the resultant fits goes close to but not through all the data points.

Usually a *linear*[2] *form* for interpolating functions is employed -

$$f(t) = \sum_{k=1}^{m} c_k \varphi_k(t) = c_1\varphi_1(t) + \cdots + c_m\varphi_m(t),$$

where c_k, with k = 1:m, are the *unknown* parameters, derived from the data, and $\varphi_k(t)$ are *predetermined* basis functions ([p.297] Ascher & Greif 2011). Moreover, the number of basis functions equals the number of data points. Note further that the basis functions assumed to be *linearly independent*[3]. Thus in general such interpolation schemes can be expressed as a system of *m* linear equations relating m unknown c_k -

[1] In Hildebrand's classic text, *Introduction to Numerical Analysis*, 1st Ed. (1956), pages 51 to 239 addressed interpolation.
[2] The term *linear* is employed to mean that f(t) is modeled as a *linear* combination of basis functions and not that f(t) or the basis functions themselves are linear in t.
[3] The assumption of linear independence requires that, if f(t) = 0 for all t, then c_k are all zero ([p.297] Ascher & Greif 2011).

$$\begin{bmatrix} \varphi_1(t_1) & \varphi_2(t_1) & \varphi_3(t_1) & \cdots & \varphi_m(t_1) \\ \varphi_1(t_2) & \varphi_2(t_2) & \varphi_3(t_2) & \cdots & \varphi_m(t_2) \\ \vdots & \vdots & \vdots & & \vdots \\ \varphi_1(t_m) & \varphi_2(t_2) & \varphi_3(t_m) & \cdots & \varphi_m(t_m) \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}.$$

Representations of interpolation schemes as systems of coupled linear equations are *usually* not employed to compute the unknown *c* parameters, but are derived by much more efficient algorithms.

Note that the choice of interpolant functions is not unique ([p. 311] Heath 2002). Issues in choosing such functions are:

- How should the chosen function act between the data points? *[a critical point]*
- Should global properties of the data, such as monotonicity, smoothness, or periodicity be considered?
- Are the values defining the interpolant physically meaningful in context of the problem?
- Does the resultant interpolant (& data) appear in Heath's words, *visually pleasing*?

Finally, the classes of functions used frequently as interpolants are following ([p. 312] Heath 2002):

- polynomials
- piecewise polynomials
- trigonometric functions
- exponential functions
- rational functions.

8.2 POLYNOMIAL INTERPOLATION

> *"Polynomial interpolants are rarely the end product of a numerical process. Their importance is more as building blocks for other, more complex algorithms in differentiation, integration, solutions of differential equations, approximation theory at large, and other areas. Hence polynomial interpolation arises frequently; indeed, it is one of the most ubiquitous tasks, both with the design of numerical algorithms and in their analysis"* ([p. 295] Ascher & Greif 2010).

The simplest and most used interpolant is the polynomial ([p. 313] Heath 2002). *Unique* polynomials of degree of m –1 can be fit to m *distinct* data pairs. However, there are many approaches in calculating and representing such polynomials, each with different computational advantages/disadvantages, although all methods must produce the same *m -1 polynomial fit*.

The primary reasons for employing polynomial schemes are their ease of use ([p. 298] Ascher & Chen 2011) in:

- Their construction and evaluation.
- Summing or multiplying where the resultant is also a polynomial.
- Differentiation or integration where the resultant is also a polynomial.

8.2.1 VANDERMONDE APPROACH

This method expresses the interpolant as a linear combination of 1, t, t^2, t^3, etc. Polynomials expressed in this form are termed *monomials*. Thus[4]

$$p_m(t) = \sum_{j=1}^{j=m+1} c_j \varphi_j(t), \quad \text{where } \varphi_j(t) \equiv t^{m+1-j}.$$

This monomial form of polynomials, although familiar to us, is not the *"best choice"* for the representation of polynomials; however, it a reasonable place to start an investigation ([p. 76] van Loan 2000).

Let us attempt to fit the data, (-2, 10), (-1, 4) (+1, 6), (+2, 3), or t = [-2 -1 1 2] and y = [10 4 6 3], by a cubic. Using the monomial representation, this cubic[5] is expressed as

$$P_3(t) = c_1 t^3 + c_2 t^2 + c_3 t + c_4.$$

Applying this representation to present data –

$$\begin{aligned}
P_3(-2) &= c_1(-8) + c_2(4) + c_2(-2) + c_4 = 10 \\
P_3(-1) &= c_1(-1) + c_2(1) + c_2(-1) + c_4 = 4 \\
P_3(+1) &= c_1(+1) + c_2(1) + c_3(+1) + c_4 = 6 \\
P_3(+2) &= c_1(+8) + c_2(4) + c_3(+2) + c_4 = 3,
\end{aligned}$$

expressing these equations in a matrix-vector format one obtains the following,

$$\begin{bmatrix} -8 & +4 & -2 & +1 \\ -1 & +1 & -1 & +1 \\ +1 & +1 & +1 & +1 \\ +8 & +4 & +2 & +1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 6 \\ 3 \end{bmatrix}.$$

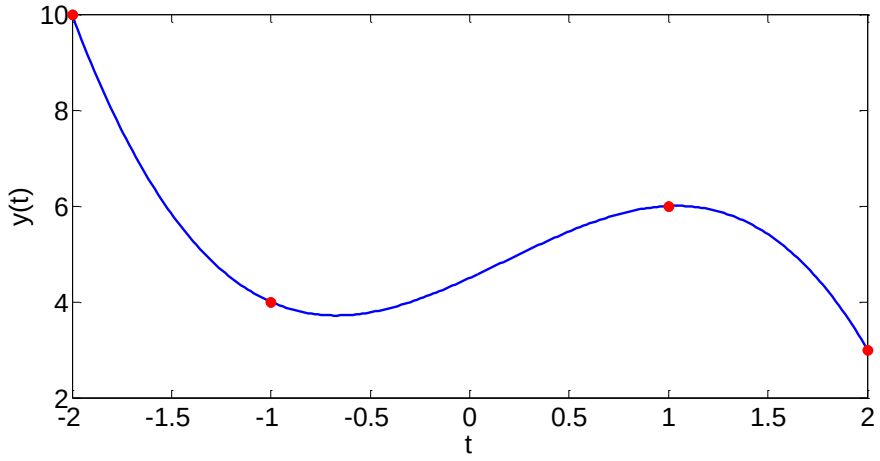
Solving this linear system, **`A*c = y`**, using Matlab's intrinsic function, **`linsolve`**,

```
>> A = [-8 4 -2 1; -1 1 -1  1; 1  1  1  1; 8  4   2  1]
A = -8     4    -2     1
    -1     1    -1     1
     1     1     1     1
     8     4     2     1
>> [c, Rc] = linsolve(A, y) % OR a = A\y;
c =      -0.91667
          0.5
          1.9167
          4.5
>> 1.0/Rc                  % Rc is the condition number of matrix A
ans = 11.955
```

Observe that the modest value, ~12, for the condition number of the coefficient matrix, A, means that the system of linear equations employed to derive the interpolating polynomial is nonsingular. Next we use this vector *c* to create a curve which passes through all the data points,

[4] The somewhat unusual ordering of monomials is employed here because in the MATLAB environment an nth-order polynomial, $p_n(t)$, is represented as $a_1 t^n + a_2 t^{n-1} + \ldots + a_n t + a_{n+1}$.
[5] Remember that a mth order polynomial must possess m+1 coefficients, c_k, k = 1:m+1.

```
>> tt = linspace(-2.0, +2.0, 250);
>> yy = c(1)*tt.^3 + c(2)*tt.^2 + c(3)*tt + c(4);
>> plot(tt, yy, 'b', t, y, 'ro')
```



Now we extend this cubic example to the general case

$$p_{m-1}(t) = c_1 t_i^{m-1} + c_2 t_i^{m-2} + \cdots + c_{m-1} t_i + c_m \equiv y_i \quad \text{for } i = 1{:}m.$$

Expressing the general case in matrix notation, (renaming[6] the coefficient matrix, A, V), we find

$$\begin{bmatrix} t_1^{m-1} & t_1^{m-2} & t_1^{m-3} & \cdots & 1 \\ t_2^{m-1} & t_2^{m-2} & t_2^{m-3} & \cdots & 1 \\ t_3^{m-1} & t_3^{m-2} & t_3^{m-3} & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_m^{m-1} & t_m^{m-2} & t_m^{m-3} & \cdots & 1 \end{bmatrix} * \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix},$$

$$V * c = y.$$

This approach is coded below -

```
  function [p, cond_number] = interp_vandermonde(t, y, tt)
% function [p, cond_number] = interp_vandermonde(t, y, tt)
% INPUT:
%      points to be interpolated:  y as function of t
%       t  = m by 1 vector
%       y  = m by 1 vector
%       tt = n by 1 vector points to be interpolated
% OUTPUT:
%       p           = n by 1 vector   p(tt)
%       cond_number = condition number calculated by linsolve
%
% Algorithm based on InterpV.m of van Loan C F (2000) [p. 79]
```

[6] A coefficient matrix with these properties is called a *Vandermonde* matrix (e.g. [p. 314] Heath 2002).

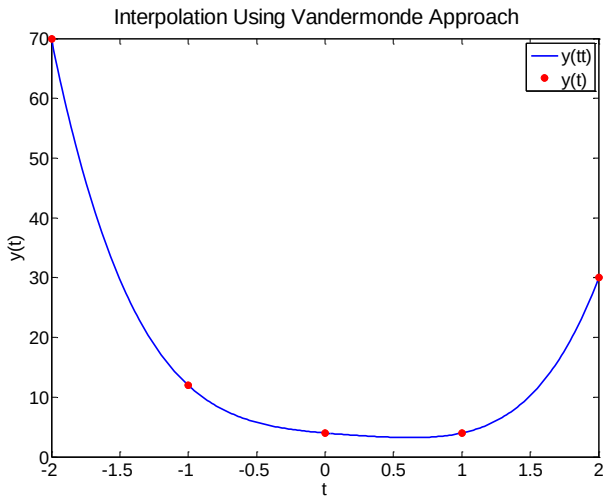
```
% Introduction to Matrix Computing: A Matrix-Vector Approach Using MATLAB
%  2nd Ed [Prentice-Hall: Upper Saddle River NJ]

    m       = length(t);
    t       = t(:);  % Forces t to be a column vector
    y       = y(:);  % forces y to be a column vector
    V       = zeros(m, m);
% Observe that coefficient matrix V in constructed from right to left -
%   => from the last column to the first column
    V(:, m) = ones(size(t));
      for j = 1:m - 1
         V(:, m - j) = t.^j;
      end

  % cis coefficient of polynomial & R is reciprocal of condition number
  [c, R] = linsolve(V, y);   cond_number = 1.0/R;

 % Employ Horner's method to evaluate the polynomial fit
  p = c(1)*ones(size(tt));
     for j = 2:m
      p = c(j) + tt.*p ;
     end
end % interp_vandermonde
```

In the above code, following van Loan [p. 81], the interpolating polynomial, p(tt), is evaluated employing Horner's rule, which uses *nested* multiplications. For example, a cubic would be evaluated in the following way -

$$P = c_1*\text{ones(size(t))};$$
$$P = P*t + c_2;$$
$$P = P*t + c_3;$$
$$P = P*t + c_4;$$
$$P = ((c_1t + c_2)t + c_3)t + c_4 \Rightarrow$$
$$P = c_1t^3 + c_2t^2 + c_3t + c_4.$$

The use of `interp_vandermonde.m` is illustrated below -

```
>> t = [-2 -1 0 +1 +2]';  y = [70 12 4 4 30]';
>> tt = linspace(-2.0, 2.0, 200);
>> [yy, cond_num] = interp_vandermonde(t, y, tt);
>> cond_num
cond_num =   56.667
>> plot(tt, yy, 'b', t, y, 'ro');
```

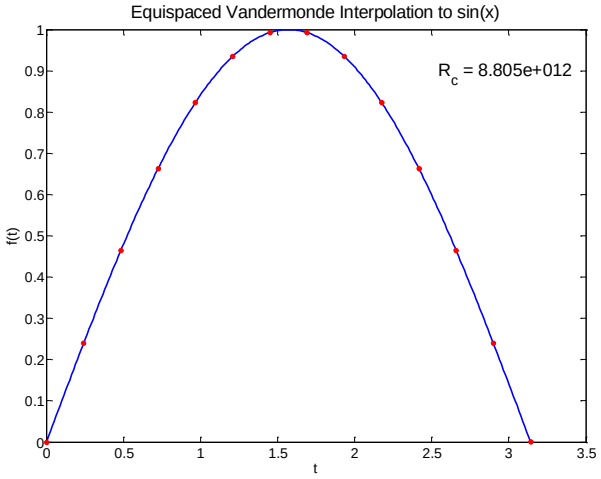


The use of the Vandermonde interpolation scheme has several disadvantages ([p. 301] Ascher & Greif):

- The computed values c_k (k = 1:m) not related in any simple way to the y_k values; this limits the application of the Vandermonde scheme in the construction of differentiation and integration algorithms.
- The Vandermonde matrix, V, can be ill-conditioned for modest-sized values of $m \geq 5$.
- Solving the m-by-m linear system of equations, $V \cdot c = y$, via Gaussian elimination requires $\geq \frac{2}{3}m^3$ floating-point operations, which can be time consuming when m is large.

The Vandermonde matrix can be ill conditioned for *high*-order polynomials. The origin of this ill-conditioning is straightforward. Note that the columns of the matrix V progress (from left to right) as x^{m-1}, x^{m-2}, x^{m-3}, ... x. As the number of data points to be interpolated, m, increases the columns of V become less distinct. Thus for large m, the columns of V become nearly linear dependent. As noted previously the linear independence of a matrix can be gauged by the condition number. Ferziger (1998) finds that the set of linear equations governing the Vandermonde algorithm "*become very ill conditioned*" for modest-sized data, i.e. *m* greater than 5! Below a 14-pt interpolation to sin(x) for $0 \leq x \leq \pi$ is shown, where the condition number approaches 10^{13} -

```
>> m = 14; t = linspace(0.0, pi, m)'; y = sin(t)';
>> [yy, Rc] = interp_vandermonde(t, sin(t), tt);
>> plot(tt, yy, 'b', t, sin(t), 'ro');
>> Rc
Rc = 8.805e+012
```



8.2.2 LAGRANGE INTERPOLATION

In the words of Ascher and Greif [p. 302], *"it would be nice if a polynomial basis is found such that $c_j = y_j$."* As they noted, such a representation would be straightforward to either differentiate or integrate. Lagrange interpolation possess such a format -

$$p_m(t) = \sum_{j=1}^{j=m+1} y_j L_j(t).$$

Consequently, the Lagrangian basis functions, $L_j(t)$, satisfy

$$L_j(t_i) = \begin{cases} 1, \text{ if } i = j, \\ 0, \text{ if } i \neq j. \end{cases}$$

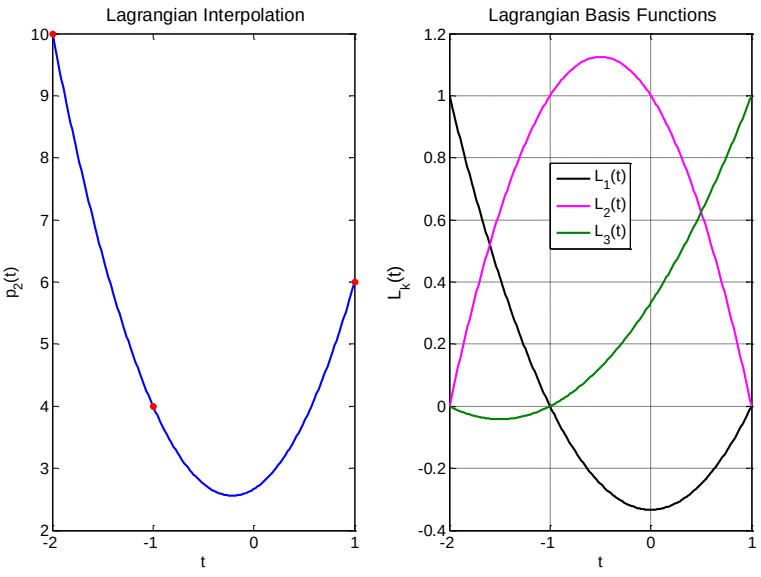
For t = [-2 -1 +1]' and y = [10 4 6]' one observes by inspection that

$L_1(t_1)= 1, L_1(t_2)= 0, L_1(t_3)= 0, \Rightarrow$
$L_1(t) = a(t + 1)(t - 1), L_1(-2) \equiv 1 \Rightarrow a = +1/3$
$L_1(t) = (t^2 - 1)/3$

$L_2(t_1)= 0, L_2(t_2)= 1, L_2(t_3)= 0, \Rightarrow$
$L_2(t) = a(t + 2)(t - 1), L_2(-1) \equiv 1 \Rightarrow a = - 1/2$
$L_2(t) = (2 - t^2 - t)/2$

$L_3(t_1)= 0, L_3(t_2)= 0, L_3(t_3)= 1, \Rightarrow$
$L_3(t) = a(t + 2)(t +1), L_3(+1) \equiv 1 \Rightarrow a = +1/6$
$L_3(t) = (t^2 + 3t + 2)/6$

```
>> t = [-2  -1  +1]'; y = [10 4 6]';
>> L1 = @(t) (t.^2 - 1)/3;
>> L2 = @(t) (2 - t.^2 - t)/2;
>> L3 = @(t) (t.^2 + 3*t+ 2)/6;
>> tt = linspace(-2, +1, 200);
>> yy = @(t) y(1)*L1(t) + y(2)*L2(t) + y(3)*L3(t);
>> subplot(1, 2, 1); plot(tt, yy(tt), 'b', t, y, 'ro');
>> subplot(1, 2, 2); plot(tt, L1(tt), 'k', tt, L2(tt), 'm', tt, L3(tt), 'g');
```



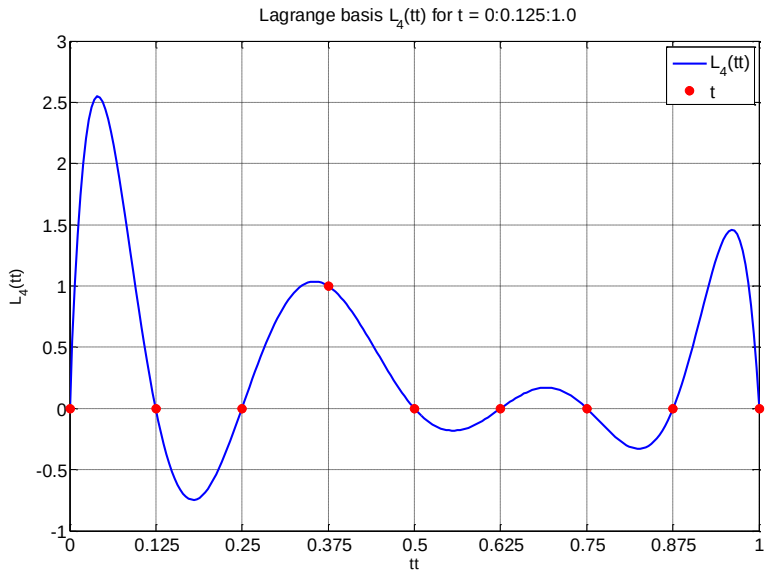
For an arbitrary set of *m* distinct data points a Lagrange basis function is

$$L_j(t) \equiv \frac{(t - t_1)\cdots(t - t_{j-1})(t - t_{j+1})\cdots(t - t_m)}{(t_j - t_1)\cdots(t_j - t_{j-1})(t_j - t_{j+1})\cdots(t_j - t_m)} = \frac{\prod_{k=1,\, k\neq j}^{m}(t - t_k)}{\prod_{k=1,\, k\neq j}^{m}(t_j - t_k)},$$

and thus *each* $L_j(t)$ itself is a polynomial of order m - 1, expressed in terms of its roots,

$$(t - t_1)\cdots(t - t_{j-1})(t - t_{j+1})\cdots(t - t_m),$$

but normalized to unity at t_j, i.e. $L_j(t_j) \equiv 1$. Observe that each basis, $L_j(t)$, has m -1 zero values and m - 2 extrema (alternating minima and maxima) (e.g. [p. 304] Ascher & Greif). Below is illustrated the j = 4 Lagrange basis, $L_4(t)$, for m = 9 equispaced data points between = 0.0 and 1.0.



One can represent Lagrangian interpolation as a system of coupled linear equations -

$$A \bullet c = y,$$

$$\begin{bmatrix} \varphi_1(t_1) & \varphi_2(t_1) & \varphi_3(t_1) & \cdots & \varphi_m(t_1) \\ \varphi_1(t_2) & \varphi_2(t_2) & \varphi_3(t_2) & \cdots & \varphi_m(t_2) \\ \vdots & \vdots & \vdots & & \vdots \\ \varphi_1(t_m) & \varphi_2(t_2) & \varphi_3(t_m) & \cdots & \varphi_m(t_m) \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix},$$

where

$$\varphi_j(t_k) \equiv L_j(t_k).$$

However,

$$L_j(t_k) = \begin{cases} 1, \text{ if } j = k, \\ 0, \text{ if } j \neq k, \end{cases}$$

and

$$c_j \equiv y_j, \text{for } j = 1{:}m.$$

which in turn requires that A·c = A·y = y. Consequently, the m-by-m matrix, *A*, is

$$A = \begin{bmatrix} 1.0 & 0 & 0 & \cdots & 0 \\ 0 & 1.0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1.0 \end{bmatrix},$$

the identity matrix.

Ascher and Greif present an efficient algorithm for implementing the Lagrange interpolation scheme. First, one computes the barycentric weights, w,

$$w_j = 1.0/\prod_{j \neq k}(t_j - t_k), \quad \text{for } j = 1{:}m.$$

This task can be performed via two nested for loops:

```
for j = 1:m
    w(j) = 1.0;
      for k = 1:m
         if k ~= j; w(j) = w(j)*(t(j) - t(k)); end
      end
    w(j) = 1.0/w(j);
  end
```

Alternatively this task can be performed using the built-in matrix functions, **`repmat`** and **`prod`**:

```
D  = repmat(t, 1, m) - repmat(t', m, 1) + diag(ones(m, 1));
w  = 1./prod(D)';    % Computes weights
```

Second, one computes the interpolation function, $p_{m-1}(tt)$, for $tt \neq t_j$, using the w parameter -

$$p_{m-1}(tt) = \frac{\sum_{j=1}^{m} \frac{w_j y_j}{(tt - t_j)}}{\sum_{j=1}^{m} \frac{w_j}{(tt - t_j)}}.$$

Also this task can be performed via two nested for loops:

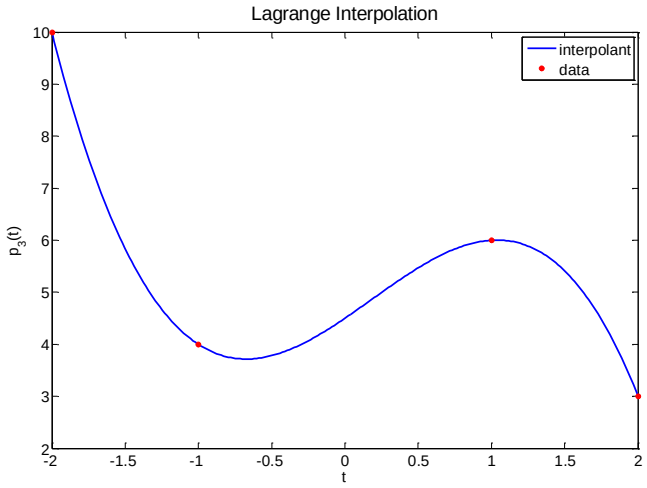
```
 for n = 1:length(tt)
   p(n) = 0.0;
   num  = 0.0;
   den  = 0.0
for j = 1:m
   if tt(n) == t(j); p(n) = y(j); continue; end  % skip remainder for tt == t
       num = num + w(j)*y(j)/(tt(n) - t(j));
       den = den + w(j)/(tt(n) - t(j));
end
   if den ~= 0.0; p(n) = num/den; end
 end
```

Alternatively this task can be performed using the built-in matrix functions, **repmat** and **prod**:

```
D  = repmat(tt, 1, m) - repmat(t', length(tt), 1);  % Computes tt - t
R  = 1./(D + eps*(D == 0.0)); % Reciprocals of tt - t & sets 0 elements to eps
yy = R*(w.*y)./(R*w);
```

```
  function p = interp_lagrange(t, y, tt)
% function p = interp_lagrange(t, y, tt) % Matrix based
% INPUT:
%      points to be interpolated:  y as function of t
%       t  = m by 1 vector
%       y  = m by 1 vector
%       tt = n by 1 vector points to be interpolated
% OUTPUT:
%       p           = n by 1 vector   p(tt)
% Algorithm (p. 305) from Ascher, U M &  Greif C 2011
% A First Course in NUMERICAL METHODS (SIAM: Philadelphia PA)

% Construction: compute barycentric weights
   t = t(:); y = y(:); tt = tt(:); % Ensures all inputs are column vectors
   m  = length(t);   nn = length(tt);
   D  = repmat(t, 1, m) - repmat(t', m, 1) + diag(ones(m, 1));
   w  = 1./prod(D)';    % Computes weights

 % Evaluate
   D  = repmat(tt, 1, m) - repmat(t', nn, 1);  % Computes tt - t
   R  = 1./(D + eps*(D == 0.0)); % Reciprocals of tt - t & sets zero elements to
eps
   p  = R*(w.*y)./(R*w);
 end
```

```
>> t = [-2 -1 1 2]'; y = [10 4 6 3]';
>> tt = linspace(-2, +2, 250)';
>> yy = interp_lagrange(t, y, tt);
>> plot(tt, yy, 'b', t, y, 'ro');
```



8.2.3 NEWTON APPROACH

Two interpolation schemes have been discussed. As noted by Ascher & Greif [p. 306], in the Vandermonde approach the derivation of the *c* coefficients is time consuming and subject to errors but evaluating the in-

terpolant, $P_{m-1}(t)$, via Horner's method is simple. Conversely, in the Lagrange scheme they observe that computing the basis functions,

$$\varphi_j(t) = \prod_{j \neq k}^{m} \frac{t - t_k}{t_j - t_k},$$

is straightforward, but evaluating the interpolant is involved. They suggest that the Newton scheme with its basis functions,

$$\varphi_j(t) = \prod_{k=1}^{j-1} (t - t_k),$$

can serve as a useful compromise between these two approaches.

If we fit a cubic interpolant to the data, t = [-2 -1 1 2] and y = [10 4 6 3], the Newton scheme employs the bases,

$$\varphi_1(t) = 1,$$
$$\varphi_2(t) = (t - t_1),$$
$$\varphi_3(t) = (t - t_1)(t - t_2),$$
$$\varphi_4(t) = (t - t_1)(t - t_2)(t - t_3),$$

Using these bases, we search for the coefficients, c_1, c_2, c_3, and c_4, such that

$$p_3(t) = c_1\varphi_1(t) + c_2\varphi_2(t) + c_3\varphi_3(t) + c_4\varphi_4(t),$$

such that

$$p_3(t_k) = y_k \text{ for } k = 1{:}4.$$

Thus

$$p_3(t_1) = y_1 = c_1$$
$$p_3(t_2) = y_2 = c_1 + c_2(t_2 - t_1)$$
$$p_3(t_3) = y_3 = c_1 + c_2(t_3 - t_1) + c_3(t_3 - t_1)(t_3 - t_2)$$
$$p_3(t_4) = y_4 = c_1 + c_2(t_4 - t_1) + c_3(t_4 - t_1)(t_4 - t_2) + c_4(t_4 - t_1)(t_4 - t_2)(t_4 - t_3).$$

Three separate schemes exist for solving for the coefficients, *c,* via a system of coupled linear equations, or, divided differences, or recursively. Now we solve by hand these four equations for *c* -

$$c_1 = y_1,$$
$$c_2 = \frac{y_2 - c_1}{t_2 - t_1},$$
$$c_3 = \frac{y_3 - (c_1 - c_2(t_3 - t_1))}{(t_3 - t_1)(t_3 - t_2)},$$
$$c_4 = \frac{(c_1 + c_2(t_4 - t_1) + c_3(t_4 - t_1)(t_4 - t_2))}{(t_4 - t_1)(t_4 - t_2)(t_4 - t_3)}.$$

As the prelude to the general solution, following van Loan ([p. 84]), let us solve for *c* here using the language of vectors and matrices which will permit to solve for the vector *c* via divided differences. First, we express the above four coupled equations in a matrix-vector format -

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & (t_2-t_1) & 0 & 0 \\ 1 & (t_3-t_1) & (t_3-t_1)(t_3-t_2) & 0 \\ 1 & (t_4-t_1) & (t_4-t_1)(t_4-t_2) & (t_4-t_1)(t_4-t_2)(t_4-t_3) \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}.$$

From this matrix solution one notes that $c_1 = y_1$. We eliminate c_1 from equations 2, 3, & 4 by subtracting equation 1 from equation 2, 3, & 4. Consequently, we find that

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (t_2-t_1) & 0 & 0 \\ 0 & (t_3-t_1) & (t_3-t_1)(t_3-t_2) & 0 \\ 0 & (t_4-t_1) & (t_4-t_1)(t_4-t_2) & (t_4-t_1)(t_4-t_2)(t_4-t_3) \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2-y_1 \\ y_3-y_1 \\ y_4-y_1 \end{bmatrix}.$$

Now we divide equation 2 by $(t_2 - t_1)$, divide equation 3 by $(t_3 - t_1)$, and equation 4 by $(t_4 - t_1)$. We find that

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & (t_3-t_2) & 0 \\ 0 & 1 & (t_4-t_2) & (t_4-t_2)(t_4-t_3) \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ (y_2-y_1)/(t_2-t_1) \\ (y_3-y_1)/(t_3-t_1) \\ (y_4-y_1)/(t_4-t_1) \end{bmatrix} \equiv \begin{bmatrix} y_1 \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix},$$

$$y_{21} \equiv (y_2-y_1)/(t_2-t_1),\ y_{31} \equiv (y_3-y_1)/(t_3-t_1),\ y_{41} \equiv (y_4-y_1)/(t_4-t_1).$$

Observe that $c_2 = y_{21}$. Subtract equation 2 from equation 3 and divide the resulting equation by $(t_3 - t_2)$. Subtract equation 2 from equation 4 and divide the resulting equation by $(t_4 - t_2)$ -

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & (t_4-t_3) \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{421} \end{bmatrix},$$

$$y_{321} \equiv (y_{31}-y_{21})/(t_3-t_2),\ y_{421} \equiv (y_{41}-y_{21})/(t_4-t_2).$$

Now we find that $c_3 = y_{321}$. Finally, subtract equation 3 from equation 4 and divide the resulting equation by $(t_4 - t_3)$ -

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{4321} \end{bmatrix},$$

$$y_{4321} \equiv (y_{431}-y_{321})/(t_4-t_3).$$

Obviously, $c_4 = y_{4321}$. The pattern for an algorithm to create a general *m*-sized interpolant parallels closely the above, where the y values are overwritten using two nested for loops -

```
    for j = 1:m - 1
      cj = yj;
       for i = j + 1:m
         yi = (yi - yj)/(ti - tj);
       end
    end
c(m) = y(m);
```

This scheme can be simplified even further using an array section to represent the inner for loop ([p. 87] van Loan) -

```
    for j = 1:m - 1
       y(j+1:m) = (y(j+1:m) - y(j))/(t(j+1:m) - t(j));
    end
  c = y;
```

Based on this code a newton interpolation scheme is expressed as

```
      function p = interp_newton(t, y, tt)
    % function p = interp_newton(t, y, tt)
    % INPUT:
    %      points to be interpolated:  y as function of t
    %       t  = m by 1 vector
    %       y  = m by 1 vector
    %       tt = n by 1 vector points to be interpolated
    % OUTPUT:
    %       p           = n by 1 vector   p(tt)
    %
    % Algorithm based on InterpN.m & HornerN of van Loan C F (2000) [p. 85]
    % Introduction to Matrix Computing: A Matrix-Vector Approach Using MATLAB
    %   2nd Ed [Prentice-Hall: Upper Saddle River NJ]

       m       = length(t);
       t       = t(:);   % Forces t to be a column vector
       y       = y(:);   % Forces y to be a column vector
       tt      = tt(:);  % Forces tt to be a column vector

       c       = newton_coefficients(t, y);

    % Employ Horner's method to evaluate the polynomial fit
    % Code fragment based HornerN (p. 87)
      p = c(m)*ones(size(tt));
         for j = m - 1:- 1:1
          p =  c(j) + (tt - t(j)).*p ;
         end
    end % interp_newton
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function c = newton_coefficients(t, y)
    % Algorithm based on InterpN.m of van Loan C F (2000) [p. 87]

      m    = length(t);
        for j = 1:m - 1
          y(j+1:m) = (y(j+1:m) - y(j))./(t(j+1:m) - t(j));
        end
      c = y;
    end
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

In the above code, following van Loan [p. 87], the interpolating polynomial, p(tt), is evaluated employing Horner's rule, which uses *nested* multiplications, similar, but not the same as, the scheme employed in the vandermonde algorithm. For the newton scheme a cubic would be expressed relative to its roots,

$$p_3(t) = c_4(t - t_3)(t - t_2)(t - t_1) + c_3(t - t_2)(t - t_1) + c_2(t - t_1) + c_1,$$

rearranging we find that

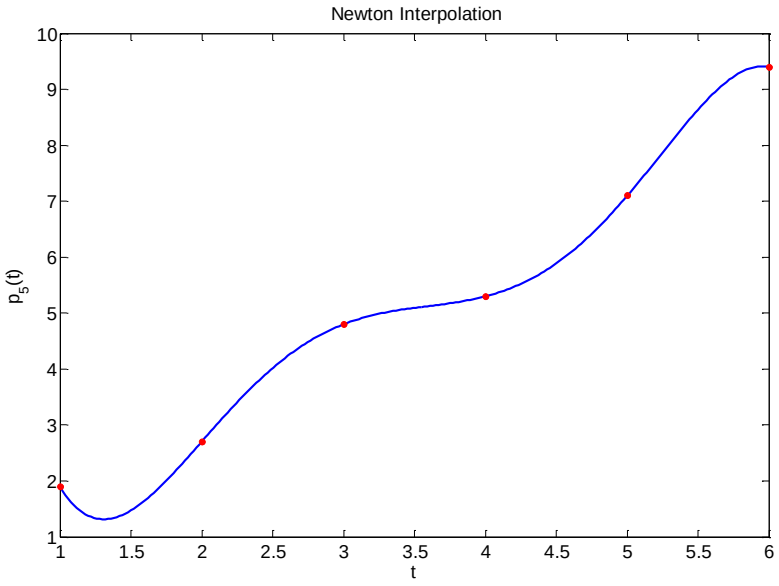
$$p_3(t) = \Big(\big(c_4\,(t-t_3)+c_3\big)(t-t_2)+c_2\Big)(t-t_1)+c_1.$$

This relationship can be represented by the following code fragment -

```
p = c(m)*ones(size(tt));
for k = m - 1:-1:1
    p = p.*(tt - t(k)) + c(k - 1);
end
```

Below is illustrated an example of newton interpolation -

```
>> t  = [1 2 3 4 5 6]'; y = [1.9 2.7 4.8 5.3 7.1 9.4]';
>> tt  = linspace(t(1), t(end), 250)';
>> yy  = interp_newton(t, y, tt);
>> plot(tt, yy, 'b', t, y, 'ro');
>> yyL = interp_lagrange(t, y, tt);
>> max(abs(yy - yyL))

ans = 5.3291e-015
>> [yyV, Rc] = interp_vandermonde(t, y, tt);
>> Rc
Rc =  1.2811e+006
>> max(abs(yy - yyV))
ans =  3.3218e-013
```



8.2.4 PROBLEMS WITH HIGH-ORDER POLYNOMIAL INTERPOLANTS

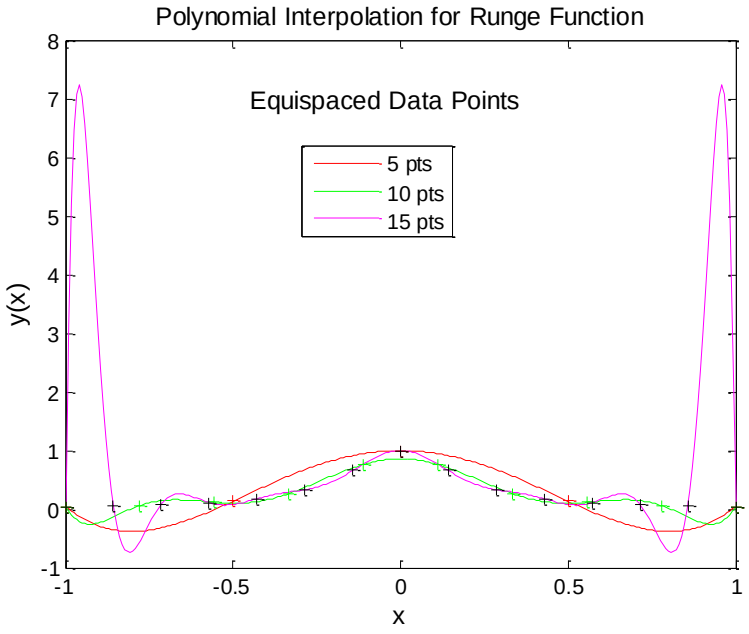
If f is a smooth function and p_{m-1} is a polynomial interpolant of order m -1 that interpolates f at t = t_1 to t_m, then for *any* t between t_1 and m, the error is the interpolant at t is, where an arbitrary $\theta \in (t_1 - t_m)$,

$$f(t) - p_m(t) = \frac{f^{(m)}(\theta)}{m!}(t - t_1)(t - t_2)\cdots(t - t_m),$$

Unfortunately θ is unknown for most cases, and the expression is only of interest if one possesses a bound on the derivatives. If a bound is known, $|f^{(m)}(t)| \le M$, and the step size (between x values), h = max($t_{i+1} - t_i$), where I = 1:m -1, thus

$$\max_{t \in [t_1, t_m]} \left| f(t) - p_{m-1}(t) \right| \le \frac{Mh^m}{4m}.$$

This error decreases as n increases (and h decreases) but *only* if $|f^{(m)}(t)|$ does not grow dramatically with m. This behavior can cause problems for *many* high-order interpolants. A polynomial of order m - 1 possesses a derivative with m - 2 zeros, and, consequently, m - 2 extrema (inflection points). Thus a high-order polynomial must have many *wiggles*, independent of the data points. Although the high-order interpolant must go through all the data points, it may oscillate violently between data points creating a useless interpolant. Such *wiggles* become discernible when continuous functions are interpolated by increasing number of equi-spaced points. A classic example of such wiggles generated by the seemingly benign Runge function is shown below (output from **`equispaced_runge.m`**) --



The rapid growth of extrema with number of equi-spaced data points is illustrated in the table below where the maximum absolute difference between the fitted curve and Runge function is determined as a function of nd:

```
>> nd = 4;
>> f = @(x) 1./(1 + (5.*x).^2);  % Vectorized Runge Function
>> xd = linspace(-1, +1, nd)'; yd = f(xd);
>> xr = linspace(-1, +1, 250)';
>> yr = interp_lagrange(xd, yd, xr);
>> plot(xd, yd, 'r+', xr, yr)
>> di = max(abs(yr-f(xr)))   %Maximum absolute difference between fit & function
```

Tabulation of d_i for n_d = 4, 8, 16, 32, 64, 128 is shown below -

n_d	Maximum difference [equi-spaced]
4	0.71
8	0.25
16	2.1
32	703.
64	1.42×10^{8}
128	5.85×10^{19}

8.2.5 CHEBYSHEV POLYNOMIALS & OPTIMUM POINTS FOR INTERPOLATION

If we are free to choose interpolating points can we find points which minimize the discretization error? If we employ the l_∞ norm, the optimal choice for points comes from Chebyshev polynomials. ([p. 374] Bradie 2006). For a non-negative integer, n, a Chebyshev polynomial, defined for $-1 \le x \le +1$, is represented as

$$T_n(x) = \cos(n\cos^{-1}(x)),$$

([p. 374], Bradie, 2006). Transforming the independent variable, $x = \cos\theta$, $T_n(x)$ simplifies to $\cos(n\theta)$ ([p. 179], Moin 2001; [p. 374], Bradie, 2006). Noting the following trigonometric identities,

$$T_{n+1}(\cos\theta) = \cos[(n+1)\theta] = \cos(n\theta)\cos(\theta) - \sin(n\theta)\sin\theta,$$
$$T_{n-1}(\cos\theta) = \cos[(n-1)\theta] = \cos(n\theta)\cos(\theta) + \sin(n\theta)\sin\theta,$$

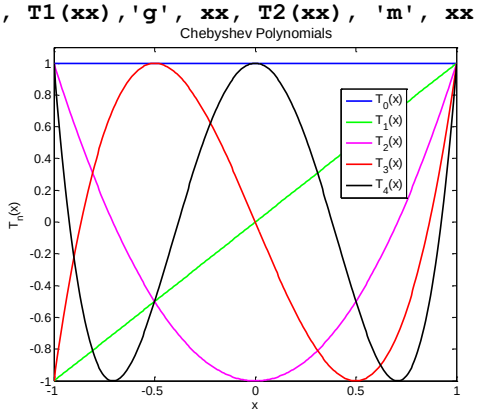
a recursive equation for $T_n(x)$ is identified;

$$T_{n+1}(\cos\theta) + T_{n-1}(\cos\theta) = 2\cos n\theta\cos\theta = 2\cos\theta T_n(\cos\theta), \text{ or } T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

From the definition of $T_n(x)$, it can be seen that $T_0(x) = 1$, and $T_1(x) = x$. Thus recurrence relation shows that

$$T_2(x) = 2xT_1(x) - T_0(x) = 2x^2 - 1,$$
$$T_3(x) = 2xT_2(x) - T_1(x) = 2x(2x^2 - 1) - x = 4x^3 - 3x,$$
$$T_4(x) = 2xT_3(x) - T_2(x) = 2x(4x^3 - 3x) - (2x^2 - 1) = 8x^4 - 8x^2 + 1,$$

```
>> T0 = @(x) ones(size(x));    T1 = @(x) x; T2 = @(x) 2.0*x.^2 - 1;
>> T3 = @(x) 4.0*x.^3 - 3.0*x; T4 = @(x) 8*x.^4 - 8*x.^2 + 1;
>> plot(xx, T0(xx), 'b', xx, T1(xx),'g', xx, T2(xx), 'm', xx, T3(xx), 'r', xx, T4(xx), 'k')
```



A Chebyshev polynomial, $T_n(x)$ has n zeros in [-1, +1] at

$$\xi_n^j = \cos\left(\frac{2j - 1}{2n}\pi\right), \quad j = 1, 2, \ldots, n,$$

the Chebyshev polynomial, $T_n(x)$ has $n+1$ extrema in [-1, +1] at

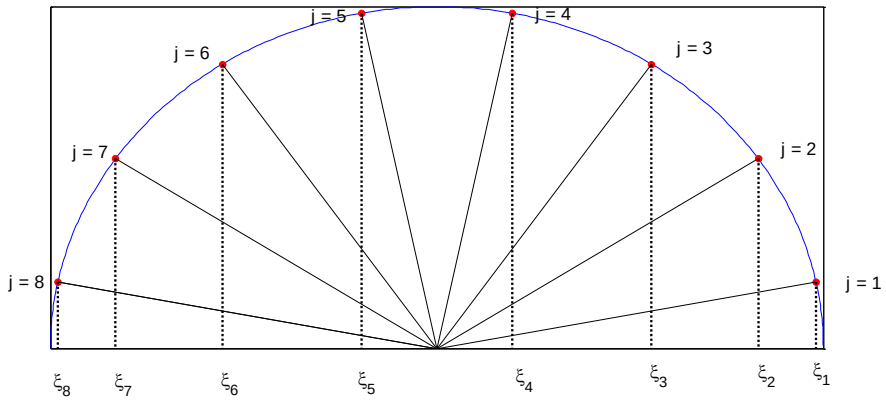
$$\eta_n^k = \cos\left(\frac{k}{n}\pi\right), \quad k = 0, 1, \ldots, n,$$

where $T_n(x)$ has a local minimum or maximum,

$$T_n(\eta_k^n) = (-1)^k, \quad k = 0, 1, \ldots, n$$

([p. 365] Ueberhuber).

Zeros of Chebyshev Polynomial, $T_8(x)$



If we interpolate a function at $-1 \le x \le +1$ at n+1 points, $x_0, x_1, \ldots, x_n$, and we want to choose these points to minimize the l_∞ norm of the polynomial

$$\omega(x) = (x - x_0)(x - x_1)(x - x_2)\cdots(x - x_n).$$

To achieve this goal we must set $\omega(x) = T_{n+1}(x)$ and must choose the interpolating points to be the roots of $T_{n+1}(x)$ ([p. 379], Bradie, 2006). Thus the interpolating points become

$$x_j = \cos\left(\frac{2j + 1}{2(n + 1)}\pi\right), \quad j = 0{:}n.$$

Thus the estimated error, over [-1, +1], becomes

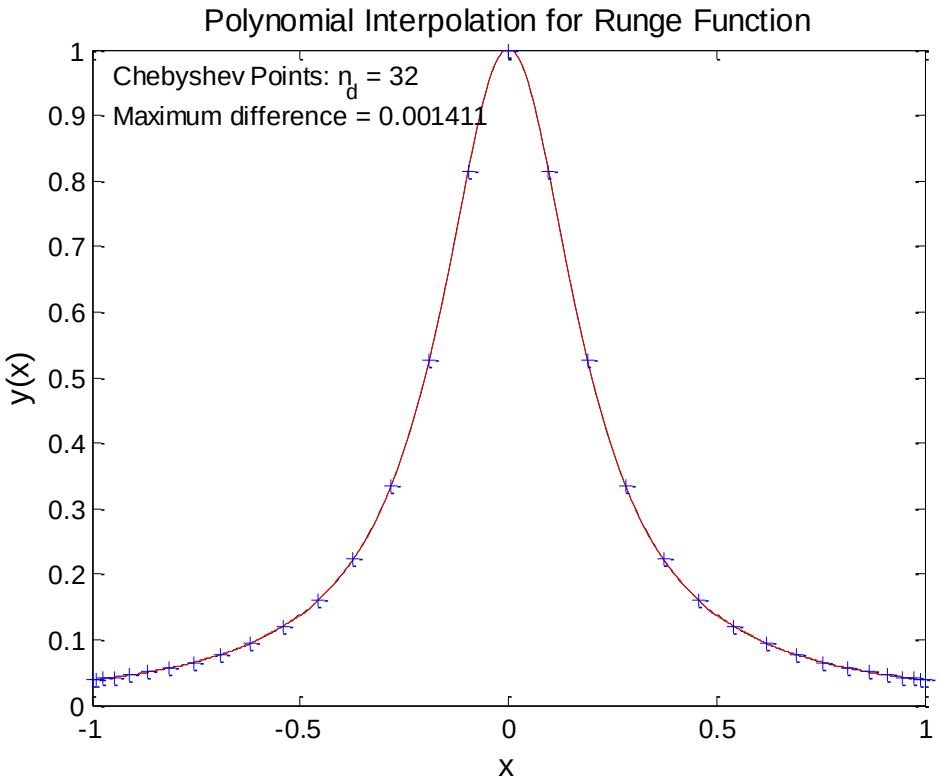
$$\|f - p_m\|_\infty \le \frac{\|f^{n+1}\|_\infty}{2^n (n+1)!}.$$

The rapid approach of the fits to the Runge function with Chebyshev data points is illustrated in the table below where the maximum absolute difference between the fitted curve and Runge function is determined as a function of n_d:

```
 % MATLAB code
>> nd = 15;
>> j = (0:nd)';  xd = cos( pi*(2*j + 1)/(2*nd + 2)); yd = f(xd);
>> xr = linspace(-1, 1, 500)';
>> yr = interp_lagrange(xd, yd, xr);
>> plot(xd, yd, 'r+', xr, yr)
>> di = max(abs(yr-f(xr)))
```

n_d	Maximum Difference Chebyshev Points
4	0.75
8	0.391
16	0.083
32	1.4×10^{-3}
64	2.5×10^{-6}
128	7.3×10^{-12}

Example of Chebyshev Fit:



8.3 PIECEWISE POLYNOMIALS (SPLINES)

The problems associated with wiggles in high-order interpolants have lead to the use of *piecewise* polynomial fits, in which a different polynomial is used in each interval of data, $[t_k, t_{k+1}]$. These data points are termed *knots* or *control points*. The most common class of piecewise polynomials are splines that are kth-order polynomials with k - 1 continuous derivatives at each knot. Cubic splines are the most commonly used splines. The basic properties of splines are investigated in an example.

Employ two cubic splines to interpolate *three* data points, (t_k, y_k), k = 1:3. Set

$$f_1(t) = \alpha_1+\alpha_2 t+\alpha_3 t^2+\alpha_4 t^3,\ t_1 \le t \le t_2.$$
$$f_2(t) = \beta_1+\beta_2 t+\beta_3 t^2+\beta_4 t^3,\ t_2 \le t \le t_3.$$

Note there are *eight* parameters to be determined. Evaluating $f_1(t)$ at t_1, t_2, and $f_2(t)$ at t_2, t_3, introduces four equations --

$$y_1 = \alpha_1+\alpha_2 t_1+\alpha_3 t_1^2+\alpha_4 t_1^3,$$
$$y_2 = \alpha_1+\alpha_2 t_2+\alpha_3 t_2^2+\alpha_4 t_2^3,$$
$$y_2 = \beta_1+\beta_2 t_2+\beta_3 t_2^2+\beta_4 t_2^3,$$
$$y_3 = \beta_1+\beta_2 t_3+\beta_3 t_3^2+\beta_4 t_3^3.$$

The continuity of derivatives supply two more equations:

$$\alpha_2 t_2+2\alpha_3 t_2+3\alpha_4 t_2^2 = \beta_2+2\beta_3 t_2+3\beta_4 t_2^2,$$
$$2\alpha_3+6\alpha_4 t_2 = 2\beta_3+6\beta_4 t_2.$$

The final two equations of constraint can be determined in *different* ways:

- Specify 1st derivatives at end points. This is termed *clamped boundary*.
- Specify 2nd derivative at end points to be zero, this is called a *natural* or *variational* spline.
- Force two consecutive cubics to be same at t_2 & t_{m-1}. This is termed *not-a-knot* spline.
- Force equality of the 1st derivative & 2nd derivatives at t_1 and t_m. The spline is *periodic*.

Please inspect the function, **`example_cubic_spline.m`**, which employs solves the above equations for a two-panel *natural* cubic spline fit for three data points. As a simple example you might want to use **`t = [-2 -1 0]`** and **`y = [4 -1 2]`**.

8.5 FOURIER TRANSFORMS

8.5.1 PRELIMINARIES

Now we propose to approximate periodic functions. A function, y(t), is termed *periodic* if for any t, y(t) = y(t + T), where T is the period. Suppose we have a sample of N data points, (t_k, y_k) k = 0:N -1. Following the terminology employed in signal analysis, we term such data a time series. Assume that data points are equi-spaced, are sampled every Δ time unit, and their number N is even. In most practical situations the data are real-valued (not complex) quantities; for purposes of this discussion assume that the time series are real valued. The application of the discrete Fourier transform is predicated on the assumption that the data are periodic with a period, T, equal to the range of the data. $T \equiv N\Delta$, is sometimes called the record length or fundamental period. The basic concept is that we can expand this time series as a sum of trigonometric functions.

8.5.2 FOURIER TRANSFORM

It can be shown that each y_j component can be expanded as a series of sines and cosines,

$$y_j = a_o + \sum_{k=1}^{N/2-1}\left[a_k\cos(2\pi kt_j/T)+b_k\sin(2\pi kt_j/T)\right]+a_{N/2}\cos(\pi Nt_j/T),$$

$$t_j = j\Delta = \frac{jT}{N},$$

$$y_j = a_o + \sum_{k=1}^{N/2-1}\left[a_k\cos(kj2\pi/N)+b_k\sin(kj2\pi/N)\right]+a_{N/2}\cos(\pi j).$$

$$a_o = \frac{1}{N}\sum_{j=0}^{N-1} y_j, \qquad a_{N/2} = \frac{1}{N}\sum_{j=0}^{N-1} y_j\cos(j\pi) = \frac{1}{N}\sum_{j=0}^{N-1} y_j(-1)^j,$$

$$a_k = \frac{1}{N}\sum_{j=0}^{N-1} y_j\cos(jk\frac{2\pi}{N}), \quad b_k = \frac{1}{N}\sum_{j=0}^{N-1} y_j\sin(jk\frac{2\pi}{N}),$$

$$1 \le k \le N/2.$$

The *a*'s and *b*'s are called *discrete Fourier cosine/sine transforms* of the *y*'s; the *y*'s are called the *inverse Fourier transforms* of a's and *b*'s. Note that a_o is the average of the y's and often is termed the *DC component*. Each specific sine or cosine represents k complete cycles in period T. Note that b_o and $b_{N/2}$ are absent in the above series, since these coefficients would be multiplied respectively by sin(0) and sin(πj), both of which are zero. We have N unknown coefficients: (N/2 - 1) of a_k, (N/2 - 1) of b_k, a_o, and $a_{N/2}$, which can be determined exactly from our N data pairs.

Each sine component inside the above square brackets has a period T/k for k = 1 to N/2 – 1; each cosine component has a period T/k for k = 1 to N/2. Consequently, the corresponding frequencies (in Hz or cycles/s) range from 1/T to (N/2 – 1)/T for sines and from 1/T to N/2T for cosines. Thus the step in frequency is 1/T. Note that the maximum frequency, f_{max}, is N/(2T). f_{max} is termed $_{Nyquist}$ frequency. f_{max} corresponds to two data points per cycle. A component with a frequency equal f_{max} cannot be resolved properly, since it does not possess a sine term. Moreover, the Fourier transform process *misidentifies* high-frequency ($f > f_{max}$) signals, it places such frequencies below ($f < f_{max}$)! Such behavior is called *aliasing*. For example, if Δ = 0.005 s, f_{max} = 100 Hz, and a component with a frequency of 125 Hz would appear at 75 Hz; a signal at 225 Hz would be resolved at 25 Hz. Avoid aliasing if possible, since it will mask completely the recognition of any signal with $f < f_{max}$.

We can express the above series as a sum of complex exponentials, the data representation employed by most Fourier transforms. Note that

$$i = \sqrt{-1},$$

$$\cos(kj2\pi/N) = \left[e^{ikj2\pi/N} + e^{-ikj2\pi/N}\right]/2,$$

$$\sin(kj2\pi/N) = \left[e^{ikj2\pi/N} - e^{-ikj2\pi/N}\right]/2i,$$

$$\exp(i2\pi(N-k)j/N) = \exp(-i2\pi kj/N).$$

where k = 1:N/2 –1. Rearranging the terms,

$$y_j = \sum_{k=0}^{N-1} c_k e^{ikj2\pi/N},\ j = 0,1,2,\cdots,N-1,$$

where

$$c_o = a_o, \qquad c_{N/2} = a_{N/2},$$

$$c_k = \frac{a_k - ib_k}{2}, \quad c_{N-k} = \frac{a_k + ib_k}{2}, \text{ for } k = 1{:}N/2-1.$$

Note that we have created a complex-valued vector, c_k, of length N. Each complex number possesses two components – a real and an imaginary. We created a 2N-element complex vector from only N pairs of (real) data. However, the information represented in either y_j or c_k must be the same. Thus only N elements of c_k are independent, and, consequently, c_k must contain N duplicate elements. If we have a real-valued time series y_j, j = 0… N – 1, resulting Fourier transform. c_k where k = 1 to N –1, satisfies

$$c_{N-n} = c_n^*.$$

Thus, only N/2 of the complex Fourier coefficients are independent!

How do we determine the frequencies corresponding to the *full* range of the complex coefficients, c_k? The standard (e.g., Press, Teukolsky, Vetterling, & Flannery, 1992; Ueberhuber, C. W. 1997) convention is to introduce negative frequencies! Remember that the transforms are periodic in k with period N. Varying k from 0 to N – 1 corresponds to a full period. Although the frequency maximum is f_{crit}, frequencies can be negative! Thus the standard convention is the following --

$$f_k \equiv \frac{k}{N\Delta}, \quad k = -\frac{N}{2}, \cdots, \frac{N}{2}.$$

Zero frequency corresponds to k = 0. The positive frequencies, $0 < f < f_{crit}$, correspond to $1 \le k \le (N/2 - 1)$; the negative frequencies, $-f_{crit} < f < 0$, correspond to $N/2 + 1 \le k \le N - 1$. N/2 corresponds to both $f = f_{crit}$ and $f = -f_{crit}$. We will come back to this critical issue in our first example.

8.5.3 INVERSE TRANSFORMS

Start with

$$y_j = \sum_{k=0}^{N-1} c_k e^{ikj2\pi/N},\ j = 0,1,2,\cdots,N-1,$$

and multiply both sides of the equation by

$$e^{-ijm\frac{2\pi}{N}},$$

where m is a positive integer between 0 and N – 1. Now sum over j form 0 to N – 1 both sides of the equation, and rearrange the sums on the right-hand side,

$$\sum_{j=0}^{N-1} y_j e^{-ijm\frac{2\pi}{N}} = \sum_{j=0}^{N-1}\sum_{k=0}^{N-1} c_k e^{ij(k-m)\frac{2\pi}{N}} = \sum_{k=0}^{N-1}\left(\sum_{j=0}^{N-1} c_k e^{ij(k-m)\frac{2\pi}{N}}\right).$$

It is clear that the sum in parentheses on the right side sums to N if m = k. What happens if $k \neq m$? Now we know from calculus that

$$\sum_{j=0}^{N-1} r^j = 1 + r + r^2 + \cdots + r^{N-1} = \frac{1-r^N}{1-r}.$$

Setting $r = e^{i(k-m)\frac{2\pi}{N}}$,

$$\sum_{j=0}^{N-1} c_k e^{ij(k-m)\frac{2\pi}{N}} = \frac{1 - \left[e^{i(k-m)\frac{2\pi}{N}}\right]^N}{1 - e^{ij(k-m)\frac{2\pi}{N}}} = \frac{1 - e^{i(k-m)2\pi}}{1 - e^{ij(k-m)\frac{2\pi}{N}}}.$$

However, k – m is an integer, since

$$e^{i(k-m)2\pi} = \cos([k-m]2\pi) + i\sin([k-m]2\pi) = 1 + i0,$$

$$\sum_{j=0}^{N-1} c_k e^{ij(k-m)\frac{2\pi}{N}} = N\delta_{km}, \text{ where}$$

$$\delta_{km} = N, \quad m = k,$$
$$= 0, \quad m \neq k.$$

Thus the inverse transform becomes

$$c_k = \frac{1}{N}\sum_{j=0}^{N-1} y_j e^{-ijk\frac{2\pi}{N}}, \text{ for } k = 0 \cdots N-1.$$

8.5.4 MATLAB INTRINSICS – `fft & ifft`

Remember that MATLAB numbers vectors from 1 to N, not 0 to N – 1, as virtually everyone else does. Thus

$$y_j = \sum_{k=1}^{N} c_k e^{i2\pi(k-1)\frac{(j-1)}{N}}, \; j = 1, \cdots, N,$$

$$c_k = \frac{1}{N}\sum_{j=1}^{N} y_j e^{-i2\pi(k-1)\frac{(j-1)}{N}}, \; k = 1, \cdots, N.$$

```
>> help fft        % MATLAB's On-line help
```

```
FFT Discrete Fourier transform.
FFT(X) is the discrete Fourier transform (DFT) of vector X.  For matrices, the FFT op-
eration is applied to each column.

FFT(X,N) is the N-point FFT, padded with zeros if X has less than N points and truncat-
ed if it has more .For length N input vector x, the DFT is a length N vector X, with
elements
```

$$X_k = \sum_{n=1}^{N} x_n e^{-i2\pi(k-1)(n-1)/N}, \; 1 \le k \le N.$$

```
 The inverse DFT (computed by IFFT) is given by
```

$$x_n = \frac{1}{N}\sum_{k=1}^{N} X_k e^{i2\pi(k-1)(n-1)/N}, \; 1 \le n \le N.$$

Example 1 (Lindfield and Penny 2000, p. 296):

```
y     = [2.8 -.77 -2.2 -3.1 -4.9 -3.2 4.83 -2.5 3.2 -3.6 -1.1 1.2 -3.2 3.3 -3.4 4.9];
 t    = 0:15;
 T    = max(t);
 D    = t(2) - t(1);   % DELTA
 N    = length(t);
 df   = 1/T;
 fmax = N/(2*T);
 sum(y)
 c    = fft(y);
 disp([t' c']);
```

```
>> [t' c']          Employs index numbering from k = 0 to N - 1.
time (NOT frequency)        Fourier coefficient
ans     0              -7.7400                c(0) = sum of y !
   1.0000               3.2959 - 8.3851i     c(1)
   2.0000              13.9798 -10.9313i     c(2)
   3.0000               8.0796 + 6.6525i     c(3)
   4.0000              -0.2300 - 4.7700i     c(4)
   5.0000               4.3150 - 6.8308i     c(5)
   6.0000              14.2202 - 1.4713i     c(5)
   7.0000             -17.2905 -15.0684i     c(N/2 - 1)
   8.0000              -0.2000                c(N/2)   ⇐ associated with Nyquist frequency
   9.0000             -17.2905 +15.0684i     c(9)  = c(7)*
  10.0000              14.2202 + 1.4713i     c(10) = c(6)*
  11.0000               4.3150 + 6.8308i     c(11) = c(5)*
  12.0000              -0.2300 + 4.7700i     c(12) = c(4)*
  13.0000               8.0796 - 6.6525i     c(13) = c(3)*
  14.0000              13.9798 +10.9313i     c(14) = c(2)*
  15.0000               3.2959 + 8.3851i     c(N-1)= c(1)*
```

```
% Remember -N/2 & +N/2 refer to same fmax
 n = (-N/2):(N/2 - 1);
 f = n/(N*D);   =>
```

Since our $-f_{max} < f < f_{max}$ we must *renumber* the Fourier transform!

A common task, so MATLAB developed a intrinsic function to do this!

```
>> help fftshift

 FFTSHIFT Shift zero-frequency component to center of spectrum.
 For vectors, FFTSHIFT(X) swaps the left and right halves of X.FFTSHIFT is
 Useful for visualizing the Fourier transform with the zero-frequency compo
 nent in the middle of the spectrum.

 [f' fftshift(c)']
  -0.5000           -0.2000                % -f_max
  -0.4375          -17.2905 +15.0684i
  -0.3750           14.2202 + 1.4713i
  -0.3125            4.3150 + 6.8308i
  -0.2500           -0.2300 + 4.7700i
  -0.1875            8.0796 - 6.6525i
  -0.1250           13.9798 +10.9313i
  -0.0625            3.2959 + 8.3851i
        0           -7.7400                % f = 0
   0.0625            3.2959 - 8.3851i
   0.1250           13.9798 -10.9313i
   0.1875            8.0796 + 6.6525i
   0.2500           -0.2300 - 4.7700i
   0.3125            4.3150 - 6.8308i
   0.3750           14.2202 - 1.4713i
   0.4375          -17.2905 -15.0684i
```

I suggest that you employ the ***full*** range (***negative*** & positive) frequencies only in rare circumstances, since the full range does not contain any information not found with the positive frequencies.

```
>> format short e
>> b = ifft(c);      % Application of the inverse transform
>> [t' b' (y' - b')]
ans =       0  2.8000e+000            0
  1.0000e+000 -7.7000e-001            0
  2.0000e+000 -2.2000e+000 -4.4409e-016
  3.0000e+000 -3.1000e+000            0
  4.0000e+000 -4.9000e+000            0
  5.0000e+000 -3.2000e+000            0
  6.0000e+000  4.8300e+000            0
  7.0000e+000 -2.5000e+000            0
  8.0000e+000  3.2000e+000            0
  9.0000e+000 -3.6000e+000            0
  1.0000e+001 -1.1000e+000  4.4409e-016
  1.1000e+001  1.2000e+000  2.2204e-016
  1.2000e+001 -3.2000e+000  8.8818e-016
  1.3000e+001  3.3000e+000            0
  1.4000e+001 -3.4000e+000  4.4409e-016
  1.5000e+001  4.9000e+000            0
```

8.9 `interp1` A MATLAB built-in function

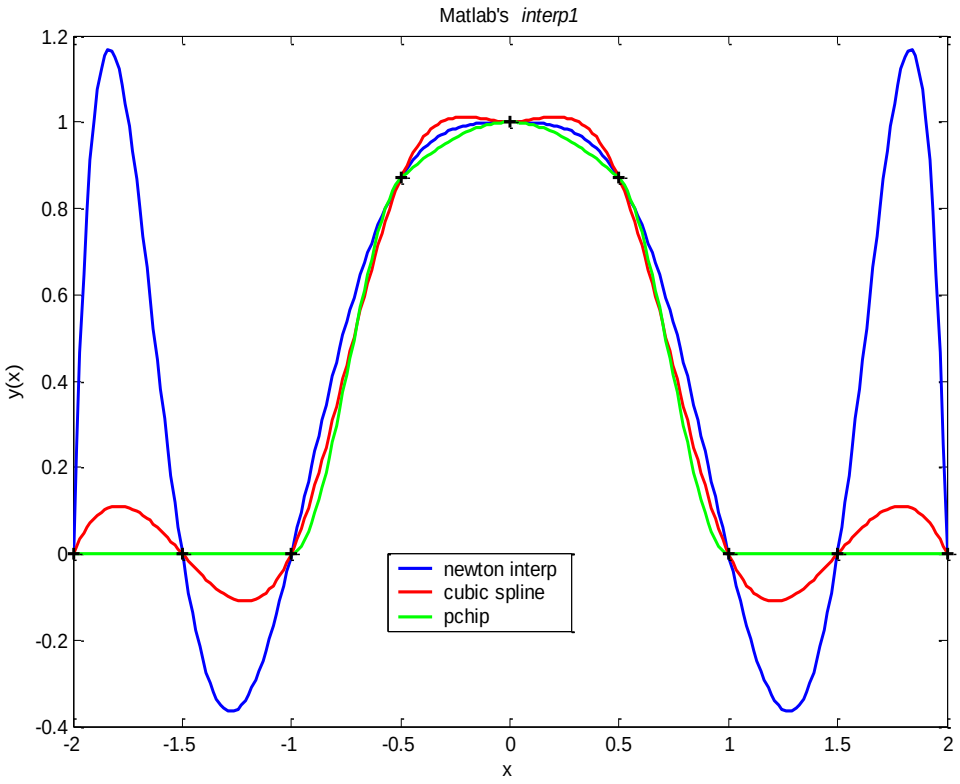
```
>> help interp1
```

INTERP1 1-D interpolation (table lookup). YI = INTERP1(X,Y,XI) interpolates to find YI, the values of the underlying function Y at the points in the vector XI. The vector X specifies the points at which the data Y is

given. If Y is a matrix, then the interpolation is performed for each column of Y and YI will be length(XI)-by-size(Y,2). YI = INTERP1(X,Y,XI,'method') specifies alternate methods. The default is linear interpolation. Available methods are:

'nearest'	- nearest neighbor interpolation
'linear'	- linear interpolation
'spline'	- piecewise cubic spline interpolation (SPLINE) [useful!]
'pchip'	- piecewise cubic Hermite interpolation (PCHIP) [useful!]

The use of **interp1** is illustrated in the function, **example_interp1.m**, whose output is shown below. NOTE: *pchip* preserves the monotonicity and shape of the data!



8.9 REFERENCES

Ascher, U. M., & Greif, C. 2011, *A First Course in NUMERICAL METHODS* (SIAM: Philadelphia, PA).

Bradie, B. 2006, *A Friendly Introduction to Numerical Analysis* (Prentice Hall: Upper Saddle River, NJ).

Ferziger, J. H. 1998, *Numerical Methods for Engineering Applications, 2nd Ed.* (John Wiley: NY, NY).

Garcia, A., 2000, *Numerical Methods for Physics. 2nd Ed.*, (Prentice Hall, Upper Saddle River, NJ).

Heath, M. T. 2002, *Scientific Computing: An Introductory Survey*, 2nd Ed. (McGraw-Hill: NY).

Kahaner, D., Moler, C., & Nash, S. 1989, *Numerical Methods and Software* (Prentice Hall, Upper Saddle River: NJ).

Lindfield, G., and Penny, J. 2000, *Numerical Methods Using MATLAB* (Prentice Hall: Upper Saddle River, NJ).

Moin, P. 2001, *Fundamentals of Engineering Numerical Analysis* (Cambridge University: Cambridge, UK).

Moler, C. 2004, *Numerical Computing with MATLAB* (SIAM: Philadelphia).

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, *Numerical Recipes in Fortran: The Art of Scientific Computing, 2nd Ed*, (Cambridge University Press: Cambridge, UK).

Quarteroni, A., & Saleri, F. 2003, *Scientific Computing with MATLAB*, (Springer-Verlag: Berlin).

Ueberhuber, C. W. 1997, *Numerical Computation 2: Methods, Software, and Analysis*, (Springer-Verlag: Berlin).

Van Loan, C. F. 2000, *Introduction to Scientific Computing: A Matrix-Vector Approach Using MATLAB*, (Prentice-Hall: Upper Saddle River, NJ).

8.10 QUESTIONS

1. True or false: There are arbitrarily many different mathematical functions that interpolate a given set of data points.

2. True or false: If an interpolating function accurately reproduces the given data values, then this fact implies that the coefficients in the linear combination of basis functions are well determined.

3. True or false: If the polynomial interpolating a given set of data points is unique, then so is the representation of that polynomial.

4. True or False: When interpolating a continuous function by a polynomial at equally spaced points on a given interval, the polynomial interpolant always converges to the function as the number of interpolation points increases.

5. What is the basic distinction between interpolation and approximation of a function?

6. State at least two different applications for interpolation.
7. Give two examples of numerical methods (for problems other than interpolation itself) that are based on polynomial interpolation.
8. Is it ever possible for two distinct polynomials to interpolate the same n data points? Ifso, under what conditions, and if not, why?
9. State at least two important criteria for choosing a particular set of basis functions for use in interpolation.
10. Determining the parameters of an interpolant can be interpreted as solving a linear system $Ax = y$, where the matrix A depends on the basis functions used and the vector y contains the function values to be fit. Describe in words the pattern of nonzero entries in the matrix A for polynomial interpolation using each of the following bases:

    (*a)* Monomial basis

    (*b)* Lagrange basis

    (c) Newton basis

11. *(a)* Is interpolation an appropriate procedure for fitting a function to noisy data? *(b)* If so, why, and if not, what

is a good alternative?

12. *(a)* For a given set of data points (t_i, y_i), i = 1:*n,* rank the following three methods for polynomial interpolation according to the cost of determining the interpolant (Le., determining the coefficients of the basis functions), from 1 for the cheapest to 3 for the most expensive: Monomial basis, Lagrange basis, Newton basis. (*b)* Which of the three methods has the best conditioned basis matrix *A,* where $A_{ij} = \varphi_j(t_i)$ (c) For which of the three methods is evaluating the resulting interpolant at a given point the most expensive?

13. *(a)* What is a Vandermonde matrix? *(b)* In what context does such a matrix arise? (c) Why is such a matrix often ill-conditioned when its order is relatively large?

14. Given a set of *n* data points, (t_i, y_i), i = 1:n, determining the coefficients *Xi* of the interpolating polynomial requires the solution of an *n* x *n* system of linear equations *Ax = y. (a)* If we use the monomial basis 1, t, t^2,..., give an expression for the entries a_{ij} of the matrix *A* that is efficient to evaluate.(*b)* Does the condition of *A* tend to become better, or worse, or stay about the same as *n* grows? (c) How does this change affect the accuracy with which the interpolating polynomial approximates the given data points.

15. List one advantage and one disadvantage of Lagrange interpolation compared with using the monomial basis for polynomial interpolation.

16. What is the computational cost (number of additions and multiplications) of evaluating a polynomial of degree *n* using Horner's method?

17. Why is interpolation by a polynomial of high degree often unsatisfactory?

18. *(a)* In interpolating a continuous function by a polynomial, what key features determine the error in approximating the function by the resulting interpolant? *(b)* Under what circumstances can the error be large even though the number of interpolation points is large?

19. How should the interpolation points be placed in an interval in order to guarantee convergence of the polynomial interpolant to sufficiently smooth functions on the interval as the number of points increases?

20. What does it mean for two polynomials p and q to be orthogonal to each other on an interval [a,b]?

21. *(a)* What is meant by a *Taylor* polynomial? *(b)* In what sense does it interpolate a given function?

22. In fitting a large number of data points, what is the main advantage of piecewise polynomial interpolation over interpolation by a single polynomial?

23. *(a)* How does Hermite interpolation differ from ordinary interpolation? *(b)* How does a cubic spline interpolant differ from a Hermite cubic interpolant?

24. In choosing between Hermite cubic and cubic spline interpolation, which should one choose (*a)* If maximum smoothness of the interpolant is desired? (*b)* If the data are monotonic and this property is to be preserved?

25. *(a)* How many times is a Hermite cubic interpolant continuously differentiable? *(b)* How many times is a cubic spline interpolant continuously differentiable?

28. *(a)* How many parameters are required to define a piecewise cubic polynomial with *n* knots? *(b)* Obviously, a similar number of equations is required to determine those parameters. Assuming the interpolating function is to be a natural cubic spline, explain how the requirements on the function account for the necessary number

of equations in the linear system to be solved for the parameters.

29. Which of the following interpolants to *n* data points are unique? (*a)* Polynomial of degree at most *n* -1. *(b)* Hermite cubic. (c) Cubic spline.

30. For which of the following types of interpolation is it possible, in general, to preserve monotonicity in a set of *n* data points (i.e., the interpolant is increasing or decreasing if the data points are increasing or decreasing)? (*a)* Polynomial of degree at most *n* -1. *(b)* Hermite cubic. (c) Cubic spline.

Exercises

1. Given the three data points (-1,1), (0,0), (1,1), determine the interpolating polynomial of degree two: (*a)* Using the monomial basis.(*b)* Using the Lagrange basis. (c) Using the Newton basis. Show that the three representations give the same polynomial.

2. Express the following polynomial in the of correct form for evaluation by Horner's method: $p(t) = 5t^3 - 3t^2 + 7t - 2.$

3. *(a)* Determine the polynomial interpolant t = [1 2 3 4 5], y = [11 29 65 125].using the monomial basis. *(b)* Determine the Lagrange polynomial interpolant to the same data and show that the resulting polynomial is equivalent to that obtained in part *a.* (c) Compute the Newton polynomial interpolant to the same data using each of the three methods given in the text (triangular matrix, incremental interpolation, and divided differences) and show that each produces the same result as the previous two methods.

4. *(a)* Write a routine that uses Horner's rule to evaluate a polynomial *pet)* given its degree *n,* an array *x* containing its coefficients, and the value t of the independent variable at which it is to be evaluated. *(b)* Add options to your routine to evaluate the derivative, p'(t).

5. Compute both polynomial and cubic spline interpolants to Runge's function, $f(t) = 1/(1 + 25t^2),$ using both *n* = 11 and 21 equally spaced points on the interval [-1,1]. Compare your results graphically by plotting both interpolants and the original function for each value of *n.*

6. An experiment has produced the following: t = [0.0 0.5 1.0 6.0 7.0 9.0] and y = [0.0 1.6 2.0 2.0 1.5 0.0]. We wish to interpolate the data with a smooth curve in the hope of obtaining reasonable values of y for values of t between the points at which measurements were taken. (a) Using any method you like, determine the polynomial of degree five that interpolates the given data, and make a smooth plot of it over the range $0 \le t \le 9$. (b) Similarly, determine a cubic spline that interpolates the given data, and make a smooth plot of it over the same range. (c) Which interpolant seems to give more reasonable values between the given data points? Can you explain why each curve behaves the way it does? (d) Might piecewise linear interpolation be a better choice for these particular data? Why?

7. The gamma function is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt, \quad x > 0.$$

For an integer argument n, the gamma function has the value

$$\Gamma(n) = (n - 1)!,$$

so interpolating the data points, t = [1 2 3 4 5 6], y = [1 1 2 6 24] should yield an approximation to the gamma function over the given range.
(a) Compute the polynomial of degree four that interpolates these five data points. Plot the resulting polynomial as well as the corresponding values given by the built-in `gamma` function over the domain [1,5].
(b) Use a cubic spline routine to interpolate the same data and again plot the resulting curve along with the built-in `gamma` function.

(c) Which of the two interpolants is more accurate over most of the domain?
(d) Which of the two interpolants is more accurate between 1 and 2?