Computing the Hermite Interpolant Polynomial

John Burkardt
Department of Scientific Computing
Florida State University

.....

https://people.sc.fsu.edu/~jburkardt/presentations/hermite_interpolant_2011_fsu.pdf

February 1, 2024

Abstract

The standard interpolating polynomial is defined by the requirement that it match a prescribed set of function values at given abscissas. The Hermite interpolating polynomial is a generalization, for which at each abscissa the function value as well as one or more derivative values are prescribed. In particular, the "simple" Hermite interpolating polynomial prescribes just the function value and first derivative. This document discusses a technique for computing the simple Hermite interpolating polynomial by modifying a common approach used for the standard interpolating polynomial.

1 The Hermite Interpolation Problem

The standard interpolating polynomial problem starts with n pairs of values (x_i, y_i), with the x values assumed to be distinct, and the y values representing arbitrary data, or possibly the results of evaluation of some function $f(x)$. For this problem data, it is required to construct a polynomial $p(x)$ of degree $n-1$ (or, perhaps more naturally, *"order"* n), which interpolates the data, that is:

$$p(x_i) = y_i \quad \text{for } i = 1, ..., n.$$

There is a generalization of this problem, known as the *Hermite interpolating polynomial problem.* Again, we have n distinct abscissa values x_i, but now each abscissa has m associated data values, $(x_i, y_{i,0}, y_{i,1}, ..., y_{i,m-1})$, and it is required to construct a polynomial with the properties that:

$$\begin{aligned} p(x_i) &= y_{i,0} \\ p'(x_i) &= y_{i,1} \\ &... \\ p^{(m-1)}(x_i) &= y_{i,m-1} \quad \text{for } i = 1, ..., n. \end{aligned}$$

In general, the Hermite interpolating polynomial satisfying these conditions will be of degree $m * n - 1$ or order $m * n$.

In the case of the "simple" Hermite interpolating polynomial, the value of m is 2, so that at each abscissa x_i we have prescribed both function and derivative data, which we may prefer to write as (y_i, y_i'). The interpolating conditions are:

$$\begin{aligned} p(x_i) &= y_i \\ p'(x_i) &= y_i' \quad \text{for } i = 1, ..., n. \end{aligned}$$

and the simple Hermite interpolating polynomial will be of degree $2 * n - 1$ or order $2 * n$.

Our concern is how to take the problem data (x_i, y_i, y_i'), convert it into a form that represents the simple Hermite interpolating polynomial, and then use that representation to evaluate the interpolant and its derivative at any point we desire. The natural approach is to consider methods for the standard interpolating problem, and investigate whether these can be modified in some way to apply to the extended problem.

2 The Polynomial Representation for the Standard Problem

Given a set of x and y data pairs, consider any consecutive subsequence of elements of y, which we may denote by $y_k, y_{k+1}, ...y_{k+j}$, and denote the *forward divided difference* of this data using the following recursive definition:

$$\begin{aligned}
[y_k] &= y_k \\
[y_k, y_{k+1}] &= \frac{[y_{k+1}] - [y_k]}{x_{k+1} - x_k} \\
&... \\
[y_k, y_{k+1}, ...y_{k+j}] &= \frac{[y_{k+1}, y_{k+2}...y_{k+j}] - [y_k, y_{k+1}, ...y_{k+j-1}]}{x_{k+j} - x_k}
\end{aligned}$$

For the standard interpolation problem, the Newton representation of the interpolating polynomial has the form:

$$p(x) = \sum_{j=1}^{n} c_j \, n_j(x)$$

Here, given the abscissas x_i, the function $n_j(x)$ is the j-th *Newton basis polynomial*:

$$n_j(x) = \prod_{i=1}^{j-1} (x - x_i)$$

with $n_1(x) \equiv 1$ and each $n_j(x)$ being a monic polynomial of degree $j-1$ or order j.

The polynomial coefficients c_j may be computed using forward divided differences. That is:

$$\begin{aligned}
c_1 = [y_1] &= y_1 \\
c_2 = [y_1, y_2] &= \frac{y_2 - y_1}{x_2 - x_1} \\
&... \\
c_j = [y_1, y_2, ...y_j] &= \frac{[y_2, y_3...y_j] - [y_1, y_2, ...y_{j-1}]}{x_j - x_1}
\end{aligned}$$

We will refer to these values as the divided difference coefficients.

Given the polynomials $\{n_j(x)\}$ and coefficients $\{c_j\}$, it can be shown that the polynomial $p(x)$ that they define is of degree $n-1$ and that $p(x_i) = y_i$ for each $i = 1, ..., n$. In other words, $p(x)$, so constructed from the given data, is the solution to the standard interpolating polynomial problem. Notice that the polynomial $p(x)$ can be determined in a straightforward way from the set of n abscissas x_j and the n divided difference coefficients c_j. The pair of vectors (x, c) forms a divided difference table, and constitutes our representation of the polynomial $p(x)$.

3 Divided Differences for the Standard Interpolant

The evaluation of the Newton polynomials $n_j(x)$ can be put off until it is necessary to evaluate $p(x)$. That means that, in order to determine the representation of the polynomial, we need only to work out the

actual values of the divided difference coefficients c_j. An efficient computation is suggested by the following procedure:

```
  for ( i = 0; i < n; i++ )
  {
    c[i] = y[i];
  }
  for ( i = 1; i < n; i++ )
  {
    for ( j = n - 1; i <= j; j-- )
    {
      c[j] = ( c[j] - c[j-1] ) / ( x[j] - x[j-i] );
    }
  }
```

Notice that the first iteration of the double loop replaces $c[j]$ by an finite difference estimate of the slope at that point.

4 Divided Differences for the Simple Hermite Interpolant

Because the simple Hermite interpolant is a polynomial, it can be represented by a difference table of the same form as the interpolating polynomial for the standard problem. However, it is not at first obvious how to compute the difference table for the simple Hermite interpolant. Now we have n abscissa values x_i and $2 * n$ values that are either function or derivative values, and we need to do something new in order to fit this data into the finite difference machinery.

It turns out that all we have to do is expand the x data vector in an obvious way, interleave the y and y' data, and carry out the first iteration of the loop by hand, computing first differences in alternate entries, and derivatives in the others. After that, the ordinary divided difference computation can proceed correctly, and the resulting information correctly represents the Hermite interpolant.

Specifically, if $x[]$, $y[]$ and $yp[]$ are arrays of dimension n, the following procedure will produce the arrays $xd[]$ and $cd[]$, both of dimension $2 * n$, where c contains the divided difference coefficients, and xd is the "doubled" abscissa array which must be associated with cd in order that the pair (xd, cd) forms a divided difference table that can later be evaluated in the usual way.

```
//
//  Make a copy of the X array that is 2*N long.
//  This augmented array must also be used later, when evaluating the Newton polynomials.
//
  for ( i = 0; i < n; i++ )
  {
    xd[0+i*2] = x[i];
    xd[1+i*2] = x[i];
  }
//
//  Use step 1 to copy Y and interleave Y' values.
//
  cd[0] = y[0];
  for ( i = 1; i < n; i++ )
  {
    cd[0+2*i] = ( y[i] - y[i-1] ) / ( x[i] - x[i-1] );
  }
  for ( i = 0; i < n; i++ )
```

```
  {
    cd[1+2*i] = yp[i];
  }
//
//  Steps 2 through 2*N-1 go in the usual way.
//
  for ( i = 2; i < 2 * n; i++ )
  {
    for ( j = 2 * n - 1; i <= j; j-- )
    {
      cd[j] = ( cd[j] - cd[j-1] ) / ( xd[j] - xd[j-i] );
    }
  }
```

5 Evaluating the Standard Interpolant

For the standard interpolant, our forward divided difference table is contained in the n pairs of values (x, c). The standard interpolant $p(x)$ is simply the sum of the Newton polynomials weighted by the finite difference coefficients, and so the evaluation $yv = p(xv)$ can be written as:

```
yv = c[n-1];
for ( i = n - 2; 0 <= i; i-- )
{
  yv = c[i] + ( xv - x[i] ) * yv;
}
```

6 Evaluating the Simple Hermite Interpolant

Recall that, for the simple Hermite interpolant, we had to create an augmented abscissa array $xd[]$ in which each original abscissa appeared twice. This array, and the corresponding divided difference coefficients cd, were now each of length $nd = 2 * n$. Keeping these differences in mind, the procedure for evaluating the simple Hermite interpolant at a point xv is analogous to that for the standard interpolant:

```
yv = cd[nd-1];
for ( i = nd - 2; 0 <= i; i-- )
{
  yv = cd[i] + ( xv - xd[i] ) * yv;
}
```

7 The Derivative of a Simple Hermite Interpolant

It is often desirable to evaluate the derivative of an interpolation polynomial. When using a simple Hermite interpolant, one obvious reason to do this is simply to verify that the derivative interpolation property has been enforce, that is, that $p'(x_i) = y'_i$.

One procedure for doing this is to calculate the divided difference table for the polynomial that is the derivative of $p(x)$. This can be done by shifting the divided difference table so that it is essentially in standard polynomial form, and then differentiating term by term, essentially replacing the coefficient c_i by $i * c_{i+1}$. The result is a set of divided difference coefficients cdp for $p'(x)$, containing $2 * n - 1$ entries. Because they were derived by shifting all abscissas to zero, the corresponding abscissa array xdp is a vector of $2 * n - 1$ zeros. The pair (xdp, cdp) forms a divided difference table, and can be evaluated in the same way as $p(x)$.

The details of this calculation are accessible by examining the function **dif_deriv()** within the **sandia_rules** library.

8 Software

The software library **sandia_rules** has been extended to include functions necessary to compute and evaluate the simple Hermite interpolant. There are two functions of primary interest to the user.

The function:

```
void hermite_interpolant ( int n, double x[], double y[], double yp[],
double xd[], double yd[], double xdp[], double ydp[] )
```

accepts n sets of data (x_i, y_i, y'_i) and returns a finite difference table (xd, yd) for $p(x)$, and a table (xdp, ydp) for $p'(x)$.

The function

```
void hermite_interpolant_value ( int nd, double xd[], double yd[], double xdp[],
  double ydp[], int nv, double xv[], double yv[], double yvp[] )
```

accepts the finite difference tables (xd, yd) and (xdp, ydp), of lengths $nd = 2 * n$ and $2 * n - 1$ respectively, and a vector xv of length nv, containing evaluation points. It returns in yv and yvp the values of $p(x)$ and $p'(x)$ at the evaluation points.

9 Example Program

There is a test program associated with **sandia_rules** called **sandia_rules_prb**. A new function has been added to this program specifically to demonstrate the behavior of the simple Hermite interpolant.

The function sets up data, creates a simple Hermite interpolant and tabulates it. The auxilliary function **r8vec_linspace_new()** is used to create a vector of abscissas equally spaced between the given limits.

```
//****************************************************************************80

void test38 ( )

//****************************************************************************80
//
//  Purpose:
//
//    TEST38 tabulates the Hermite interpolant and its derivative.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    01 November 2011
//
//  Author:
//
//    John Burkardt
//
```

```
{
  int i;
  int n;
  int nd;
  int ndp;
  int ns;
  double *x;
  double *xd;
  double *xdp;
  double *xs;
  double *y;
  double *yd;
  double *ydp;
  double *yp;
  double *ys;
  double *ysp;

  std::cout << "\n";
  std::cout << "TEST38\n";
  std::cout << "  HERMITE_INTERPOLANT sets up the Hermite interpolant.\n";
  std::cout << "  HERMITE_INTERPOLANT_VALUE evaluates it.\n";
  std::cout << "  Consider data for y=sin(x) at x=0,1,2,3,4.\n";

  n = 5;
  y = new double[n];
  yp = new double[n];

  nd = 2 * n;
  xd = new double[nd];
  yd = new double[nd];

  ndp = 2 * n - 1;
  xdp = new double[ndp];
  ydp = new double[ndp];

  x = webbur::r8vec_linspace_new ( n, 0.0, 4.0 );
  for ( i = 0; i < n; i++ )
  {
    y[i] = std::sin ( x[i] );
    yp[i] = std::cos ( x[i] );
  }

  webbur::hermite_interpolant ( n, x, y, yp, xd, yd, xdp, ydp );
/*
  Now sample the interpolant at NS points, which include data values.
*/
  ns = 4 * ( n - 1 ) + 1;
  ys = new double[ns];
  ysp = new double[ns];

  xs = webbur::r8vec_linspace_new ( ns, 0.0, 4.0 );
```

```
  webbur::hermite_interpolant_value ( nd, xd, yd, xdp, ydp, ns, xs, ys, ysp );

  std::cout << "\n";
  std::cout << "  In the following table, there should be perfect\n";
  std::cout << "  agreement between F and H, and F' and H'\n";
  std::cout << "  at the data points X = 0, 1, 2, 3, and 4.\n";
  std::cout << "\n";
  std::cout << "  In between, H and H' approximate F and F'.\n";
  std::cout << "\n";
  std::cout << "     I       X(I)          F(X(I))         H(X(I)) ";
  std::cout << "        F'(X(I))        H'(X(I))\n";
  std::cout << "\n";
  for ( i = 0; i < ns; i++ )
  {
    std::cout << "  " << std::setw(4) << i
              << "  " << std::setw(14) << xs[i]
              << "  " << std::setw(14) << sin ( xs[i] )
              << "  " << std::setw(14) << ys[i]
              << "  " << std::setw(14) << cos ( xs[i] )
              << "  " << std::setw(14) << ysp[i] << "\n";
  }

  delete [] x;
  delete [] xd;
  delete [] xdp;
  delete [] xs;
  delete [] y;
  delete [] yd;
  delete [] ydp;
  delete [] yp;
  delete [] ys;
  delete [] ysp;

  return;
}
```

10 Results of Example Program

The program constructs a simple Hermite interpolant, using the abscissas $x = 0, 1, 2, 3, 4$ with data values $y = sin(x)$ and $y' = cos(x)$.

The interpolant is then evaluated at 17 evenly spaced points in the interval [0,4], which includes the original data points. The exact function and derivative are compared to those of the simple Hermite interpolant. At the original data values, the interpolant agrees perfectly with the original data, as required by the interpolation condition.

```
  TEST38
  HERMITE_INTERPOLANT sets up the Hermite interpolant.
  HERMITE_INTERPOLANT_VALUE evaluates it.
  Consider data for y=sin(x) at x=0,1,2,3,4.
```

```
In the following table, there should be perfect
agreement between F and H, and F' and H'
at the data points X = 0, 1, 2, 3, and 4.

In between, H and H' approximate F and F'.
```

I	X(I)	F(X(I))	H(X(I))	F'(X(I))	H'(X(I))
0	0	0	0	1	1
1	0.25	0.247404	0.247407	0.968912	0.968921
2	0.5	0.479426	0.479428	0.877583	0.877575
3	0.75	0.681639	0.681639	0.731689	0.731683
4	1	0.841471	0.841471	0.540302	0.540302
5	1.25	0.948985	0.948985	0.315322	0.315324
6	1.5	0.997495	0.997495	0.0707372	0.0707368
7	1.75	0.983986	0.983986	-0.178246	-0.178247
8	2	0.909297	0.909297	-0.416147	-0.416147
9	2.25	0.778073	0.778073	-0.628174	-0.628172
10	2.5	0.598472	0.598473	-0.801144	-0.801143
11	2.75	0.381661	0.381661	-0.924302	-0.924304
12	3	0.14112	0.14112	-0.989992	-0.989992
13	3.25	-0.108195	-0.108194	-0.99413	-0.994124
14	3.5	-0.350783	-0.350781	-0.936457	-0.936451
15	3.75	-0.571561	-0.571559	-0.820559	-0.820567
16	4	-0.756802	-0.756802	-0.653644	-0.653644

References

[1] Philip Davis, **Interpolation and Approximation**, Dover, 1975.

[2] Carl de Boor, **A Practical Guide to Splines**, Springer, 2001,