Polynomial interpolation with Lagrange basis

We (again) construct a polynomial $p_n(x)$ of degree at most n, that interpolates the data (x_i, f_i), $i = 0, \cdots, n$. Instead of using the monomials 1, x, x^2, $\cdots$, x^n, as basis functions, we use the Lagrange basis functions, $l_0^{(n)}(x)$, $l_1^{(n)}(x)$, $l_2^{(n)}(x)$, $\cdots$, $l_n^{(n)}(x)$, defined by

$$l_j^{(n)}(x) \equiv \frac{(x-x_0)(x-x_1)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n)}{(x_j-x_0)(x_j-x_1)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)}$$
$$= \frac{\prod_{i=0, i\neq j}^{n}(x-x_i)}{\prod_{i=0, i\neq j}^{n}(x_j-x_i)},$$

and having the property

$$l_j^{(n)}(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}.$$

Notice that each $l_j^{(n)}(x)$ is a polynomial of degree exactly n. In most cases, since the degree n is obvious from the context, we drop the superscript (n) from the notation and use $l_j(x)$ for $l_j^{(n)}(x)$. Also note that, if $x_j = x_i$, for some $i \neq j$, then $l_j(x)$ and $l_i(x)$ are not defined.

Examples of Lagrange basis functions

Consider the case $n = 0$. Recall also that, by convention, the product of no terms is set to 1. Thus, if $n = 0$, $l_0(x) = \frac{\prod_{i=0, i\neq 0}^{0}(x-x_i)}{\prod_{i=0, i\neq 0}^{0}(x_0-x_i)} = 1$.

Consider the case $n = 1$. Then

$$l_0(x) = \frac{\prod_{i=0, i\neq 0}^{1}(x-x_i)}{\prod_{i=0, i\neq 0}^{1}(x_0-x_i)} = \frac{x-x_1}{x_0-x_1}, \quad l_1(x) = \frac{\prod_{i=0, i\neq 1}^{1}(x-x_i)}{\prod_{i=0, i\neq 1}^{1}(x_1-x_i)} = \frac{x-x_0}{x_1-x_0}.$$

Polynomial interpolation with Lagrange basis

Let $p_n(x) = \sum_{j=0}^{n} a_j l_j(x)$. With the representation of $p_n(x)$ in terms of the Lagrange basis functions, the coefficients a_j are trivial to compute. This is because the interpolating equations $p_n(x_i) = f_i$, $i = 0, \cdots, n$, reduce to $\sum_{j=0}^{n} a_j l_j(x_i) = f_i$, and due to the property of the Lagrange basis functions given above we have $a_i = f_i$, $i = 0, \cdots, n$. Thus, we can set a_i to f_i, without doing any computation.
We can view the interpolating equations as a linear system $\mathbf{I}$a = f, where $\mathbf{I}$ is the identity matrix, a the vector of coefficients a_j, $j = 0, \cdots, n$, and f the vector of data values f_i, $i = 0, \cdots, n$. Of course, "solving" a system with the identity matrix is trivial.

The Lagrange basis functions give us an explicit form for the polynomial interpolant p_n of the data (x_i, f_i), $i = 0, \cdots, n$,

$$p_n(x) = \sum_{j=0}^{n} f_j l_j(x).$$

Polynomial interpolation with Lagrange basis -- examples

Example: Consider the case $n = 0$, i.e. the case of constant interpolation. In this case $p_0 = f_0 l_0 = f_0$. This result agrees with the result obtained through the monomial basis representation of p_0.

Example: Consider the case $n = 1$, i.e. the case of linear interpolation. In this case $p_1(x) = f_0 l_0(x) + f_1 l_1(x) = f_0 \frac{x-x_1}{x_0-x_1} + f_1 \frac{x-x_0}{x_1-x_0}$. This result agrees with the result obtained through the monomial basis representation of p_1, $p_1(x) = \frac{x_0 f_1 - x_1 f_0}{x_0 - x_1} + \frac{f_0 - f_1}{x_0 - x_1} x$. That is, the polynomial does not change. Only the form in which it is written is different.

Notes:

- The coefficients of one representation are (in general) different from the respective ones of the other representation, since the basis functions are different. However, the two representations are equivalent, and denote the same polynomial.
- If $x_0 = x_1$, then the basis functions l_0 and l_1 are not defined.
- If $f_0 = f_1$, then p_1 is of degree 0.

Polynomial interpolation with Lagrange basis -- numerical example

Consider the data $(0, 5)$, $(-1, 7)$, $(2, 13)$. Here, $n = 2$. The polynomial of degree at most 2 interpolating the data is

$$\begin{aligned} p_2(x) &= f_0 l_0(x) + f_1 l_1(x) + f_2 l_2(x) = 5l_0(x) + 7l_1(x) + 13l_2(x) \\ &= 5\,\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + 7\,\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + 13\,\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \\ &= 5\,\frac{(x+1)(x-2)}{(0+1)(0-2)} + 7\,\frac{(x-0)(x-2)}{(-1-0)(-1-2)} + 13\,\frac{(x-0)(x+1)}{(2-0)(2+1)} \\ &= -5\,\frac{(x+1)(x-2)}{2} + 7\,\frac{x(x-2)}{3} + 13\,\frac{x(x+1)}{6} \end{aligned}$$

It is easy to bring this polynomial in monomial form and verify that it is the same polynomial as the one computed through the monomial basis function representation, i.e. $p_2(x) = 5 + 2x^2$.

Existence and uniqueness of the interpolating polynomial

Theorem: Given data (x_i, f_i), $i = 0, \cdots, n$, with the x_i being distinct, there exists a unique polynomial of degree at most n, interpolating the data.

Proof:

The existence of the polynomial is shown by using the Lagrange form of the interpolating polynomial. Since x_i are distinct, the basis functions $l_j(x)$ can be constructed, and therefore $p_n(x) = \sum_{j=0}^{n} f_j l_j(x)$ is an interpolating polynomial.

To prove uniqueness, suppose there are two polynomials, $p_n(x)$ and $q_n(x)$, of degree at most n, which both interpolate the data. That is, $p_n(x_i) = f_i$, and $q_n(x_i) = f_i$, $i = 0, \cdots, n$.

Consider $r(x) = p_n(x) - q_n(x)$. This is a polynomial of degree at most n. However, we have $r(x_i) = p_n(x_i) - q_n(x_i) = f_i - f_i = 0$, for $i = 0, \cdots, n$, i.e., $r(x)$ has at least $n+1$ roots. Since $r(x)$ is of degree at most n and has at least $n+1$ roots, it must be the zero polynomial, i.e., $r(x) = 0$, that is $p_n(x) = q_n(x)$. Thus, the two polynomials of degree at most n interpolating the data must be the same. QED.

Polynomial interpolation with Newton's basis

We (once again) construct a polynomial $p_n(x)$ of degree at most n, that interpolates the data (x_i, f_i), $i = 0, \cdots, n$. We use yet another set of basis functions, the Newton basis functions. These are defined by

$$b_j(x) \equiv (x - x_0)(x - x_1)\cdots(x - x_{j-1}) = \prod_{i=0}^{j-1}(x - x_i).$$

Notice that $b_j(x)$ is a polynomial of degree exactly j. Also note that the b_j's are built gradually, i.e. each b_j is the product of b_{j-1} times one more term, $(x - x_{j-1})$. Thus, $b_j(x_i) = 0$, for $i = 0, \cdots, j-1$, and for $j = i+1, \cdots, n$, i.e. for $i < j$.

Examples of Newton's basis functions:

$b_0(x) = 1$, $b_1(x) = x - x_0$, $b_2(x) = (x - x_0)(x - x_1)$, ... ,
$b_n(x) = (x - x_0)(x - x_1)\cdots(x - x_{n-1})$.

Let $p_n(x) = \sum_{j=0}^{n} a_j b_j(x)$. With the representation of $p_n(x)$ in terms of Newton's basis functions, the coefficients a_j are not too difficult to compute. There is a recursive procedure that uses the so-called *Newton's Divided Differences* to compute the coefficients.

Polynomial interpolation with Newton's basis

To understand how the procedure works, consider the interpolating equations and take into account the form of Newton's basis functions. That is,

$p_n(x_0) = f_0 \Rightarrow a_0 = f_0$

$p_n(x_1) = f_1 \Rightarrow a_0 + a_1(x_1 - x_0) = f_1 \Rightarrow a_1 = \dfrac{f_1 - f_0}{x_1 - x_0}$

$p_n(x_2) = f_2 \Rightarrow a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f_2 \Rightarrow$

$$a_2 = \frac{\dfrac{f_2 - f_1}{x_2 - x_1} - \dfrac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}.$$

We can write more interpolation equations and derive the formulae for more coefficients, but at this point we start seeing a pattern in the formulae, and we need to generalize the pattern. To generalize, we first give a definition.

For a function $f(x)$ or a set of data $f_i = f(x_i)$, $i = 0, \cdots, n$, we define the *Newton's Divided Differences* (NDDs) by

$f[x_i] \equiv f_i$,

$f[x_i, x_{i+1}] \equiv \dfrac{f_{i+1} - f_i}{x_{i+1} - x_i}$ and

$f[x_i, x_{i+1}, \cdots, x_{i+k}] \equiv \dfrac{f[x_{i+1}, x_{i+2}, \cdots, x_{i+k}] - f[x_i, x_{i+1}, \cdots, x_{i+k-1}]}{x_{i+k} - x_i}$. Notice that this

is a recursive definition. It can be shown that $a_i = f[x_0, x_1, \cdots, x_i]$. That is, in order to compute the coefficients a_j, we need to compute the divided differences of the data. When computing the NDDs by hand, we use the so-called NDD table. We write the x_i's and respective f_i's in the two leftmost columns of the table. We then fill the entries in the table, so that they form a triangular shape (arrow), i.e. column $k+1$ has one less entry than column k. An entry in the kth column of the table is computed by differencing the corresponding pair of nearby entries in the $(k-1)$st column and dividing by the appropriate difference $x_{i+k} - x_i$. The coefficients a_j are the entries along the upper diagonal of the table.

Newton's Divided Difference (NDD) Table

x_0	f_0				
		$f[x_0, x_1]$			
x_1	f_1		$f[x_0, x_1, x_2]$		
		$f[x_1, x_2]$		$f[x_0, x_1, x_2, x_3]$	
x_2	f_2		$f[x_1, x_2, x_3]$		$f[x_0, x_1, x_2, x_3, x_4]$
		$f[x_2, x_3]$		$f[x_1, x_2, x_3, x_4]$	
x_3	f_3		$f[x_2, x_3, x_4]$		...
		$f[x_3, x_4]$		...	
x_4	f_4		...		
.	.	...			
		$f[x_{n-1}, x_n]$			
x_n	f_n				

Note that the first two columns of the table are given data, while the rest of the table is computed.

Polynomial interpolation with Newton's basis -- examples

Example: Consider the case $n = 0$, i.e. the case of constant interpolation. In this case, the NDD table is just

x_0	f_0

and $a_0 = f_0$. Thus $p_0 = f_0 b_0 = f_0$. This result agrees with the result obtained through the monomial and Lagrange basis representations of p_0.

Example: Consider the case $n = 1$, i.e. the case of linear interpolation. In this case, the NDD table is

x_0	f_0	
		$f[x_0, x_1]$
x_1	f_1	

and $a_0 = f_0$, $a_1 = f[x_0, x_1]$. Thus

$$p_1(x) = f_0 b_0(x) + f[x_0, x_1] b_1(x) = f_0 + f[x_0, x_1](x - x_0) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0).$$

This result agrees with the result obtained through the monomial and Lagrange basis representations of p_1, $p_1(x) = \frac{x_0 f_1 - x_1 f_0}{x_0 - x_1} + \frac{f_0 - f_1}{x_0 - x_1} x = f_0 \frac{x - x_1}{x_0 - x_1} + f_1 \frac{x - x_0}{x_1 - x_0}$.

That is, the polynomial does not change. Only the form in which it is written is different. This is expected since we have shown that there is a unique polynomial of degree at most 1, interpolating the data (x_0, f_0), (x_1, f_1).

Polynomial interpolation with Newton's basis -- numerical example

Consider the data $(0, 5)$, $(-1, 7)$, $(2, 13)$. Here, $n = 2$. Construct the NDD table for the data.

0	5		
		-2	
-1	7		2
		2	
2	13		

Using the table, we have $p_2(x) = 5 - 2(x - 0) + 2(x - 0)(x + 1)$. It is easy to bring this polynomial in monomial form and verify that it is the same polynomial as the one computed through the monomial and the Lagrange basis function representations, i.e. $p_2(x) = 5 + 2x^2$.

What if we want to add another piece of data? The NDD table makes this easy.
We can add the extra (x_i, f_i) pair at the bottom of the table and compute the arising new entries. We then only need to add one more term to $p_2(x)$ to obtain $p_3(x)$.

Polynomial interpolation with Newton's basis -- adding more data example

Example: Consider including the data $(1, 5)$ in the above table. Now $n = 3$. The updated NDD table becomes

0	5			
		-2		
-1	7		2	
		2		1
2	13		3	
		8		
1	5			

Notice that this table (with $n = 3$) is expanded from the previous one (with $n = 2$), by the bottom diagonal. Then we have $p_3(x) = p_2(x) + 1(x - 0)(x + 1)(x - 2) = 5 + 2x^2 + x(x + 1)(x - 2)$.

Polynomial interpolation with Newton's basis -- note

The computation of the entries of the NDD table, and, thus, of the coefficients a_j of the Newton representation of the interpolating polynomial is equivalent to the solution of a triangular linear system $C\mathrm{a} = \mathrm{f}$, where

$$C = \begin{bmatrix} b_0(x_0) & b_1(x_0) & b_2(x_0) & \cdots & b_n(x_0) \\ b_0(x_1) & b_1(x_1) & b_2(x_1) & \cdots & b_n(x_1) \\ b_0(x_2) & b_1(x_2) & b_2(x_2) & \cdots & b_n(x_2) \\ . & . & . & \cdots & . \\ b_0(x_n) & b_1(x_n) & b_2(x_n) & \cdots & b_n(x_n) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & x_1 - x_0 & 0 & \cdots & 0 \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & \cdots & 0 \\ . & . & . & \cdots & . \\ 1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \cdots & (x - x_0)(x - x_1)\cdots(x - x_{n-1}) \end{bmatrix},$$

$$\mathrm{a} = \begin{bmatrix} a_0 \\ a_1 \\ . \\ a_n \end{bmatrix}, \mathrm{f} = \begin{bmatrix} f_0 \\ f_1 \\ . \\ f_n \end{bmatrix}.$$

Comparison of the three bases

- Monomials: $1, x, x^2, \cdots, x^n$
- The polynomial $p_n(x)$ is efficiently evaluated (Horner's rule),
- and easy to differentiate and integrate.
- Computing the coefficients a_i requires solving a linear system $B\mathrm{a} = \mathrm{f}$ of size $n+1$, where B is dense, and ill-conditioned for large n.
- Adding a piece of data requires recomputation of all a_i's.
- Lagrange: $l_0(x), l_1(x), \cdots, l_n(x)$, with $l_j(x) = \dfrac{\prod_{i=0, i\neq j}^{n} (x - x_i)}{\prod_{i=0, i\neq j}^{n} (x_j - x_i)}$
- The polynomial $p_n(x)$ is a bit messy to evaluate,
- and even messier to differentiate and integrate.
- Computing the coefficients a_i is trivial: $a_i = f_i$. (This is equivalent to solving a linear system with the identity matrix.)
- Adding a piece of data requires updating all the $l_j(x)$'s.

Comparison of the three bases

- Newton: $1, x - x_0, (x - x_0)(x - x_1), \cdots, (x - x_0)(x - x_1)\cdots(x - x_{n-1})$
- The polynomial $p_n(x)$ is efficiently evaluated (algorithm similar to Horner's rule),
- but a bit messy to differentiate and integrate.
- Computing the coefficients a_i is relatively simple, with the help of Newton's Divided Differences and the use of a recursive algorithm. The work is equivalent to solving a triangular linear system (forward substitution) of size $n+1$.
- Adding a piece of data is easy. It requires the computation of one additional coefficient. All other coefficients remain the same.

For computational purposes, the Newton basis functions are preferred, as they balance the various criteria well.
For hand calculations (small n), we may choose one of the three bases depending on which one is more convenient to us for our purposes.

Error in polynomial interpolation

The following theorem give us insight about the quality of polynomial interpolation, and information about the magnitude of the error and special cases when the error is zero.

We first introduce some notation.
The *spread* of a set of points $\{s_1, s_2, \cdots, s_N\}$, denoted by $\text{spr}\{s_1, s_2, \cdots, s_N\}$, is the smallest interval containing $s_1, s_2, \cdots, s_N$. The *open spread* of a set of points $\{s_1, s_2, \cdots, s_N\}$, denoted by $\text{ospr}\{s_1, s_2, \cdots, s_N\}$, is the largest open interval which is a subset of $\text{spr}\{s_1, s_2, \cdots, s_N\}$.

Note that, with the above notation, the points do not need to be ordered. If, though, the points are ordered, i.e. $s_1 \le s_2 \le \cdots \le s_N$, then $\text{spr}\{s_1, s_2, \cdots, s_N\} = [s_1, s_N]$, and $\text{ospr}\{s_1, s_2, \cdots, s_N\} = (s_1, s_N)$.

We now give the statement of the Theorem.

Error in polynomial interpolation -- remarks

3. The point ξ is unknown, therefore $f^{(n+1)}(\xi)$ and the error are unknown. We also know that ξ changes with x. However, in some cases, we can calculate a (upper) bound on $|f^{(n+1)}(x)|$, when x is in a certain interval. In that case, if we also have a bound on $|W(x)|$, we can calculate a bound for the error.
4. The value of $W(x)$ can be calculated for a particular x. For x in a certain interval, a bound on $|W(x)|$ can be calculated.
   For example, if $n \le 2$, $W(x)$ is cubic, therefore $W'(x)$ is quadratic. Thus it is easy to find the roots r_1 and r_2 of $W'(x)$. Then, $\max_{a \le x \le b} |W(x)| = \max\{|W(x)|, x = a, x = b, x = r_1, x = r_2\}$.
   If $n > 2$, the roots of $W'(x)$ can be computed by nonlinear iterative techniques. If r_i, $i = 1, \cdots, n$ are the roots of $W'(x)$, $\max_{a \le x \le b} |W(x)| = \max\{|W(x)|, x = a, x = b, x = r_i, i = 1, \cdots, n\}$.
   If the points x_i are equidistant, simpler techniques apply to calculate a bound on $|W(x)|$. It can be shown that, if $x_i = a + ih$, $i = 0, \cdots, n$, with $h = (b-a)/n$, then $\max_{x_0 \le x x_n} |W(x)| \le \dfrac{(x_n - x_0)^{n+1}}{2^{n+1}}$

Error in polynomial interpolation

Theorem (error in polynomial interpolation): If $f(x)$ has $n+1$ continuous derivatives and $p_n(x)$ is the polynomial of degree at most n interpolating f at distinct points $x_0, x_1, \cdots, x_n$, then, for any x,

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{j=0}^{n} (x - x_j),$$

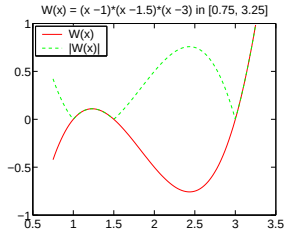
where ξ is an unknown point in $\text{ospr}\{x_0, x_1, \cdots, x_n, x\}$, that depends on x.

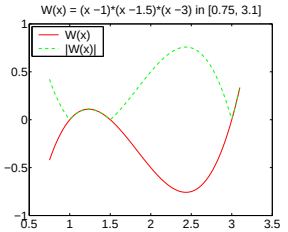
We do not give a proof of this theorem, but make remarks about what it implies in practical cases.

Remarks:

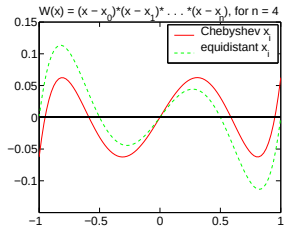
0. If f happens to be a polynomial of degree at most n, since $f^{(n+1)} = 0$, the error is zero, that is, the interpolant is the function itself. This is expected from the uniqueness of polynomial interpolation theorem.
1. The error of the interpolant is zero on the data points x_i, $i = 0, \cdots, n$. This is expected from the definition of the interpolant.
2. The error formula consists of two main parts, one is $f^{(n+1)}(\xi)$, the other is $W(x) = \prod_{j=0}^{n} (x - x_j) = (x - x_0)(x - x_1) \cdots (x - x_n)$.

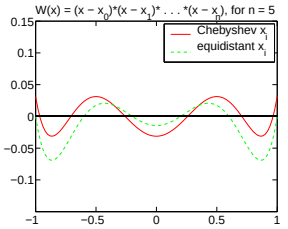
Error in polynomial interpolation -- remarks





If the points x_i are equidistant, $W(x)$ tends to be larger towards the endpoints and smaller towards the midpoint $(a+b)/2$.

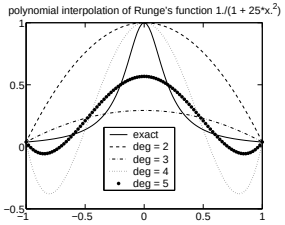


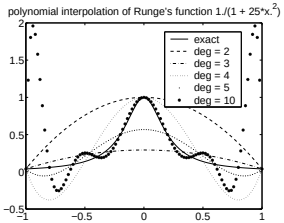


Certain choices for non-equidistant x_i's (denser towards the endpoints and sparser towards the midpoint) even-up $W(x)$ along $[a, b]$, so that $\max |W(x)|$ is minimized. Such choices are the so-called Chebychev points.

Pitfalls of polynomial interpolation

In order to get a fairly good approximation to a function $f(x)$ by an interpolating polynomial of nth degree, it is often necessary to use a fairly large n. Unfortunately, polynomials of high degree often exhibit an oscillatory behaviour towards the endpoints.





n	max error
2	6.4615e-01
3	7.0701e-01
4	4.3813e-01
5	4.3269e-01
10	1.9156e+00
20	5.8278e+01

In the above plots, we sample the function $f(x) = \dfrac{1}{1+25x^2}$ at an appropriate number of equidistant points in $(-1, 1)$ in order to construct interpolating polynomials of the degrees indicated, then plot the interpolants and the function itself on lots (101) of evaluation points. To the right, we table the maximum (in absolute value) error of the interpolant.

Pitfalls of polynomial interpolation

This problem is partly overcome by using the Chebyshev points, as data points. The Chebyshev points are denser towards the endpoints and sparser in the middle of the interval. However, if the function is given in tabular form, we do not have a choice which data points to use.

In addition, high degree polynomials are not desirable for various reasons relating to numerical instability and computational inefficiency.

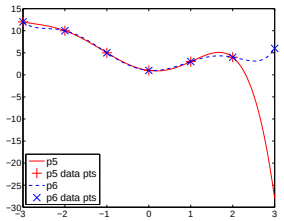
The construction of high degree polynomials often leads to ill-conditioned computations. When the polynomial degree is large, matlab (`polyfit`) gives warnings about ill-conditioned polynomials.

High degree polynomials are also often sensitive to small perturbations of the coefficients, therefore small errors in the computation of the coefficients may get amplified when doing the evaluation of the polynomial.

The addition of a single piece of data may substantially change the behaviour of the resulting polynomial.

Sensitivity of a polynomial to added data -- Example

```
xi = [-3, -2, -1, 0, 1, 2, 3];
yi = [12, 10,  5, 1, 3, 4, 6];
p5 = polyfit(xi(1:6), yi(1:6), 5)
p6 = polyfit(xi,      yi,      6)
xv = linspace(-3, 3, 100);
yv5= polyval(p5, xv);
yv6= polyval(p6, xv);
plot(xv, yv5, 'r-', xi(1:6), yi(1:6), 'r+', ...
     xv, yv6, 'b--', xi, yi, 'bx')
legend('p5','p5 data pts', 'p6','p6 data pts', 3);
```




```
p5 =
-0.1083 -0.5000  0.3750  3.5000 -1.2667  1.0000
p6 =
 0.0472  0.0333 -0.7361 -0.3333  3.6889 -0.7000 1.0000
```


Pitfalls of polynomial interpolation

The NDD procedure takes $O(n^2)$ flops, and the evaluation of the polynomial in NDD form takes $O(n)$ flops for one point of evaluation. Often, the number of evaluation points is much larger than the number of data points, therefore, the evaluation procedure is of higher cost than $O(n^2)$.

For these reasons, polynomial interpolation with large numbers of data is not used in practice.

Piecewise polynomial interpolation is a good alternative, that resolves the above problems, as it

- avoids oscillatory behaviour for large n;
- guarantees that the error decreases as n increases;
- often leads to well-conditioned matrices;
- requires $O(n)$ flops for construction and $O(1)$ flops for evaluation per data point.

We will study piecewise polynomial interpolation methods next in the course.

