Polynomial Function Predictor

Dennis E. Bahr, P.E.
September 2009

This is the first of a series of three papers that describe the derivation of equations that can be used to calculate the next dependent variable value in sequence for any sampled polynomial, its derivative, or its integral knowing the current and previous dependent variable values of the polynomial. For these papers, I will assume that the independent variable has equally spaced values. These equations are used predominately for computer modeling and embedded system control and are known as predictor or extrapolation equations. Such equations are often generated using the Taylor series, but I will show how one can derive the equations by finding the solution to a set of unique matrices. Once one sees the pattern in the matrices one can write the equations for almost any order by using a few simple rules.

In general, data collected in a real world situation cannot be described by a polynomial without having some error between the data and the polynomial describing the data. Thus, when these *predictor* equations are used for numerical integration they are often are combined with companion *corrector* equations.

All of the equations that I discussed above are related in one way or another to Pascal's Triangle. Pascal's triangle is built as a series of numbers ordered into rows and columns as shown below:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
```

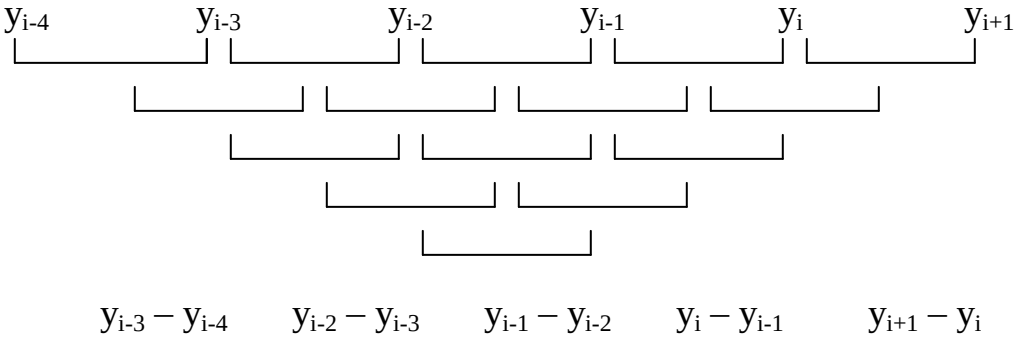
Pascal's Magnificent Triangle

One normally uses the triangle to expand expressions like $(x + y)^n$ where x and y are variables and/or numbers and n is a whole number. As an example of a cube we can write:

$$(x + y)^3 = x^3 + 3x^2y + 3yx^2 + y^3$$

You will notice that the coefficients can be found in the fourth row in the triangle. There are various ways to calculate the rows in the triangle, but most of them are just methods and formulas and don't describe the real magic in the triangle.

I will now derive predictor equations for polynomial functions by first using a very simple and non-rigorous approach and later I will do the derivation again using a much more sophisticated and rigorous method. The first method is the method of finite differences. We start with a set of independent variable points that equally spaced such that $x_{i+1} - x_i = h$ where h can be any value and is a constant for all independent variable values. If we take the difference between adjacent ordinate values, and then take the differences of these differences, etc. we end up with an equation for predicting future values of a polynomial. Let's start with a set of data points, y_n, that have equal spacing between samples and begin by taking the forward difference of all adjacent points.



If we now take the last two differences on the right and calculate the forward difference again we obtain the following:

$$(y_{i+1} - y_i) - (y_i - y_{i-1}) = y_{i+1} - 2y_i + y_{i-1}$$

This result is the second order difference of the last three data points and the coefficients can be found in the third row of the triangle.

Let us use all of the first order differences calculated above and take the forward differences of these pairs.

$$(y_{i-2} - y_{i-3}) - (y_{i-3} - y_{i-4}) \qquad (y_{i-1} - y_{i-2}) - (y_{i-2} - y_{i-3})$$

$$(y_i - y_{i-1}) - (y_{i-1} - y_{i-2}) \qquad (y_{i+1} - y_i) - (y_i - y_{i-1})$$

$$y_{i-2} - 2y_{i-3} + y_{i-4} \qquad y_{i-1} - 2y_{i-2} + y_{i-3} \qquad y_i - 2y_{i-1} + y_{i-2} \qquad y_{i+1} - 2y_i + y_{i-1}$$

We continue by taking the forward differences of these triplets and end up with three equations containing four terms each.

$$(y_{i-1} - 2y_{i-2} + y_{i-3}) - (y_{i-2} - 2y_{i-3} + y_{i-4})$$

$$(y_i - 2y_{i-1} + y_{i-2}) - (y_{i-1} - 2y_{i-2} + y_{i-3})$$

$$(y_{i+1} - 2y_i + y_{i-1}) - (y_i - 2y_{i-1} + y_{i-2})$$

$$y_{i-1} - 3y_{i-2} + 3y_{i-3} - y_{i-4} \qquad y_i - 3y_{i-1} + 3y_{i-2} - y_{i-3} \qquad y_{i+1} - 3y_i + 3y_{i-1} - y_{i-2}$$

The coefficients of these equations can all be found in the fourth row of the triangle. Let's continue by combining like terms. The terms are set equal to zero since the highest order derivative of any polynomial is equal to zero.

$$(y_i - 3y_{i-1} + 3y_{i-2} - y_{i-3}) - (y_{i-1} - 3y_{i-2} + 3y_{i-3} - y_{i-4}) +$$

$$(y_{i+1} - 3y_i + 3y_{i-1} - y_{i-2}) - (y_i - 3y_{i-1} + 3y_{i-2} - y_{i-3}) = 0$$

$$y_i - 3y_{i-1} + 3y_{i-2} - y_{i-3} - y_{i-1} + 3y_{i-2} - 3y_{i-3} + y_{i-4} +$$

$$y_{i+1} - 3y_i + 3y_{i-1} - y_{i-2} - y_i + 3y_{i-1} - 3y_{i-2} + y_{i-3}) = 0$$

Combining terms and solving for y_{i+1} gives us the final result.

$$\boxed{y_{i+1} = 5y_i - 10y_{i-1} + 10y_{i-2} - 5y_{i-3} + y_{i-4}}$$

The equation that we end up with has coefficients from the sixth row of the triangle except that the signs alternate. You will notice that the y_{i+1} term has been placed on the left side of the equation. By knowing the five previous data points we can determine the next data point in sequence. Since the equation was derived using differences and differences of differences, etc., the equation has taken into account all of the derivatives for fourth order on down. The first six orders are shown below and one can easily see that the coefficients track the elements of Pascal's Triangle except for alternating changes in sign. This method is related to Newton and Neville's interpolation methods since both deal with finite differences and equally spaced independent variable points.

Order	Predictor Equation	
0	$y_{i+1} = 1y_i$	[1.0]
1	$y_{i+1} = 2y_i - y_{i-1}$	[1.1]
2	$y_{i+1} = 3y_i - 3y_{i-1} + y_{i-2}$	[1.2]
3	$y_{i+1} = 4y_i - 6y_{i-1} + 4y_{i-2} - y_{i-3}$	[1.3]
4	$y_{i+1} = 5y_i - 10y_{i-1} + 10y_{i-2} - 5y_{i-3} + y_{i-4}$	[1.4]
5	$y_{i+1} = 6y_i - 15y_{i-1} + 20y_{i-2} - 15y_{i-3} + 6y_{i-4} - y_{i-5}$	[1.5]
6	$y_{i+1} = 7y_i - 21y_{i-1} + 35y_{i-2} - 35y_{i-3} + 21y_{i-4} - 7y_{i-5} + y_{i-6}$	[1.6]

It is interesting to note that if one uses the backward difference or central difference instead of the forward difference, the exact same equations will be obtained. It is important however, that one use the same difference method for all calculations when using this difference approach. One can write these equations by using Pascal's triangle and there are simple formulae available to allow one to calculate the coefficients for any row of Pascal's Triangle.

How can we now use these equations to do something useful? For example, lets start with a randomly selected polynomial equation that fits some set of data perfectly.

$$y = 4y^3 - 7y^2 + 3y - 2 \qquad [1.7]$$

Using arbitrary chosen abscissa points we get the following table:

x	y	position
+0	–2	y_{i-3}
+1	–2	y_{i-2}
+2	+8	y_{i-1}
+3	+52	y_i
+4	+154	y_{i+1}

All of these values were obtained using equation [1.7]. Using equation [1.3] and inserting the first four ordinate or y values from the table above into it, we can calculate the y_{i+1} term as follows:

$$y_{i+1} = 4 * (+52) - 6 * (+8) + 4 * (-2) - (-2)$$

$$y_{i+1} = 208 - 48 - 8 + 2 = 154 \qquad \text{value obtained from [1.3]}$$

Notice that we get the same value we got from equation [1.7] using the polynomial predictor. Using the same equations, but a different table obtained by sliding along the abscissa, we get the following:

x	y	position	all values obtained from [1.7]
–2	–68	y_{i-3}	
–1	–16	y_{i-2}	
0	–2	y_{i-1}	
+1	–2	y_i	
+2	+8	y_{i+1}	

Now, using equation [1.3] above we can calculate the y_{i+1} term as follows:

$$y_{i+1} = 4 * (-2) - 6 * (-2) + 4 * (-16) - (-68)$$

$$y_{i+1} = -8 + 12 - 64 + 68 = 8 \qquad \text{value obtained from [1.3]}$$

It appears that one can solve a polynomial equation by using the polynomial equation directly and finding one point at a time if one has a set of data points and knows the order of that data, one can always find the next data point in sequence without actually knowing the polynomial equation that describes the data points. This process can be continued over and over again by sliding along the abscissa and doing the calculation again and again. One does need to be careful because the results may deviate due to round off or truncation errors. If one does not know the order of the equation or system from which the points were generated, one can always use any of the higher order equations from the set above and get the correct answer. If the order of the data is not known, one can try using the equations in decreasing order until the y_{i+1} value from two of the equations disagree. The higher order equation can then be chosen as the correct equation to use and the order of the data is the order of that equation. Again, for real world data a *predictor-corrector* method would most likely be required.

Let us now use a more rigorous method to derive these equations. For a first order polynomial we are looking for an equation of the form:

$$y_{i+1} = Jy_i + Ky_{i-1}$$

We begin with the general equation for a first order system and expand it out.

$$A + Bx_{i+1} = J(A + Bx_i) + K(A + Bx_{i-1})$$

$$A + Bx_{i+1} = JA + JBx_i + KA + KB\ x_{i-1}$$

Let us assume that the sample points are equally spaced and equal to a variable that we will call "h".

$$A + B(x_i + h) = JA + JBx_i + KA + KB(x_i - h)$$

$$A + B(x_i + h) = JA + JBx_i + KA + KBx_i - KBh$$

This equation is true for all values of y if and only if all of the terms containing A on the left side of the equation are equal to the terms containing an A on the right side of the equation. The also applies to the B terms since the A terms and the B terms are orthogonal to each other. Let's begin with the A terms.

$$A = JA + KA$$

Dividing each term by A yields

$$J + K = 1$$

For the B terms we have

$$Bx_i + Bh = JBx_i + KBx_i - KBh$$

Now divide each term by B

$$x_i + h = Jx_i + Kx_i - Kh$$

$$x_i + h = x_i(J + K) - Kh$$

Since J + K = 1 we can write

$$x_i + h = x_i - Kh$$

$$h = - Kh$$

$$K = - 1$$

Therefore, since J + K = 1

$$J - 1 = 1 \text{ or } J = 2$$

The resulting equation then becomes:

$$\boxed{y_{i+1} = 2y_i - y_{i-1}}$$

For a second order system we begin by looking for an equation of the form:

$$y_{i+1} = Jy_i + Ky_{i-1} + Ly_{i-2}$$

The general equation then becomes:

$$A + Bx_{i+1} + Cx_{i+1}^2 = J(A + Bx_i + Cx_i^2) +$$

$$K(A + Bx_{i-1} + Cx_{i-1}^2) + L * (A + B * x_{i-2} + C * x_{i-2}^2)$$

$$A + B(x_i + h) + C(x_i + h)^2 =$$

$$JA + JBx_i + JCx_i^2 + KA + KB(x_i - h) + KC(x_i - h)^2$$

$$+ LA + LB(x_i - 2h) + LC(x_i - 2h)^2$$

Separating the A terms from the equation above we have:

$$A = JA + KA + LA$$

Dividing each term by A

$$J + K + L = 1 \qquad [2.1]$$

Separating out the B terms we have:

$$B(x_i + h) = JBx_i + KB(x_i - h) + LB(x_i - 2h)$$

Dividing each term by B

$$x_i + h = Jx_i + K(x_i - h) + L(x_i - 2h)$$

Multiplying out the terms

$$x_i + h = Jx_i + Kx_i - Kh + Lx_i - 2Lh$$

$$x_i + h = x_i(J + K + L) - Kh - 2Lh$$

Grouping terms

$$h = - Kh - 2Lh$$

$$1 = - K - 2L$$

$$K = - 2L - 1 \qquad [2.2]$$

Separating the C terms from the initial equation

$$C(x_i + h)^2 = JCx_i^2 + KC(x_i - h)^2 + LC(x_i - 2h)^2$$

Divide each term by C and multiplying out terms

$$x_i^2 + 2x_ih + h^2 = Jx_i^2 + Kx_i^2 - K2x_ih + Kh^2 +$$

$$Lx_i^2 - 4Lx_ih + 4Lh^2$$

Grouping terms

$$x_i^2 + 2x_ih + h^2 = x_i^2(J + K + L) – K2x_ih + Kh^2 – 4Lx_ih + 4Lh^2$$

Since J + K + L =1

$$2x_ih + h^2 = – K2x_ih + Kh^2 – 4Lx_ih + 4Lh^2$$

Divide each term by h

$$2x_i + h = – K2x_i + Kh – 4Lx_i + 4Lh$$

Combining terms

$$2x_i + h = 2x_i(– K – 2L) + Kh + 4Lh$$

Using equation [2.2] this can be reduced to

$$h = Kh + 4Lh$$

Dividing each term by h again and rearranging we get:

$$K = 1 – 4L \qquad [2.3]$$

Using equations [2.1], [2.2], and [2.3] we can solve for J, K, and L

$$J = 3 \qquad K = – 3 \qquad \text{and} \quad L = 1 \qquad [2.4]$$

Therefore our equation becomes:

$$y_{i+1} = 3y_i – 3y_{i-1} + y_{i-2} \qquad [2.5]$$

If we continue the derivations with higher and higher orders we find that a pattern emerges. Using this pattern we can write a set of three simultaneous equations for the case that we just solved. I have decided not to bore you with the derivations, but instead show the results.

$$J + 1K + 1L = 1$$

$$J + 2K + 3L = 0$$

$$J + 4K + 9L = 0$$

If these equations are solved for J, K, & L we find that we find that we get the same results as shown in [2.4] above. However, this form of the

simultaneous equations allows us to easily expand the set of equations to higher orders.

For a third order equation such as:

$$y_i = Jy_{i-1} + Ky_{i-2} + Ly_{i-3} + My_{i-4}$$

The linear equations to be solved look like:

$$J + 1K + 1L + 1M = 1$$

$$J + 2K + 3L + 4M = 0$$

$$J + 4K + 9L + 16M = 0$$

$$J + 8K + 27L + 64M = 0$$

The solution to these equations is:

$$J = +4 \qquad K = -6 \qquad L = +4 \qquad \text{and} \quad M = -1$$

These constants form the basis for the following equation:

$$y_{i+1} = 4y_i - 6y_{i-1} + 4y_{i-2} - y_{i-3}$$

Since we now have the pattern, we no longer need to go through the process of deriving the linear set of equations for higher order polynomials. In fact, since the coefficients of the solutions are elements of Pascal's Triangle, we also no longer need to solve any sets of linear equations. We can use the available formulary for Pascal's Triangle directly and add alternating signs to the elements in our row of choice and obtain all of the coefficients for our prediction equations.

The second paper discusses how to derive the equations for determining a future derivative of a polynomial function. It turns out that this is only slightly more complicated than what we did for functions. The third article of this series will show how to derive equations for determining the predictor and corrector integrals of a polynomial function. These last two papers will require that we actually solve the matrices that we generate, and to that end I have developed a C++ library that treats fractions as an object and where results are returned as fractions. Doing our math this way allows us to obtain more accurate results and the equations that we will derive are usually shown as fractions in publications. More on that later.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Function.c                                   *
 *                                                                       *
 *    Calculates Coefficients for Polynomial Predictor Equations         *
 *                                                                       *
 *                       Dennis E. Bahr, P.E.                            *
 *                     Bahr Management, Inc.                             *
 *                   6632 North Chickahauk Trail                         *
 *                       Middleton, WI 53562                             *
 *                                                                       *
 *                         October 5, 2009                               *
 *                           Version 1.0                                 *
 *                   Copyright (c) 2001-2009 USA                         *
 *                       All Rights Reserved                             *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <math.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>

#define ORDER 4
#define N ORDER+1
#define TRUE  1
#define FALSE 0

__int64 test;

class fraction {
   private:
      long int numerator, denominator;

   public:
      fraction(long int numerator = 1, long int denominator = 1);

      fraction operator + (fraction);
      fraction operator - (fraction);
      fraction operator * (fraction);
      fraction operator / (fraction);
         bool operator == (fraction);
          bool operator > (fraction);
          void operator = (fraction);

      fraction reduction(fraction);
          void setfraction(long int, long int);
          fraction fabs(fraction);
          void fpow(long int, long int);
          void oprint();
          void sprint();
};

fraction::fraction(long int n, long int d) {
   numerator = n;
   denominator = d;
}

fraction fraction::operator + (fraction f) {
   fraction temp, out;
   temp.numerator = numerator * f.denominator + f.numerator * denominator;
   temp.denominator = f.denominator * denominator;
   out = temp.reduction(temp);
   return out;
}

fraction fraction::operator - (fraction f) {
   fraction temp, out;
   temp.numerator = numerator * f.denominator - f.numerator * denominator;
   temp.denominator = f.denominator * denominator;
   out = temp.reduction(temp);
```

```
   return out;
}

fraction fraction::operator * (fraction f) {
   fraction temp, out;
   temp.numerator = numerator * f.numerator;
   temp.denominator = denominator * f.denominator;
   out = temp.reduction(temp);
   return out;
}

fraction fraction::operator / (fraction f) {
   fraction temp;
   temp.numerator = numerator * f.denominator;
   temp.denominator = denominator * f.numerator;
   return temp.reduction(temp);
}

bool fraction::operator == (fraction f) {
   return (numerator * f.denominator == f.numerator * denominator);
}

bool fraction::operator > (fraction f) {
   return (numerator * f.denominator > f.numerator * denominator);
}

void fraction::operator = (fraction f) {
        numerator = f.numerator;
        denominator = f.denominator;
}

void fraction::setfraction(long int n, long int d) {
   numerator = n;
   denominator = d;
}

fraction fraction::fabs( fraction out ) {
   out.numerator = labs(numerator);
   out.denominator = labs(denominator);
   return out;
}

void fraction::fpow(long int x, long int y) {
   numerator = pow(x, y);
   denominator = 1;
}

void fraction::oprint() {
   printf("%7ld", numerator);
}

void fraction::sprint() {
   printf("%6ld/%ld", numerator, denominator);
}

//--------------------------------------------------------------------------//
//      Euclid's Algorithm for fraction reduction
//--------------------------------------------------------------------------//

fraction fraction::reduction(fraction f) {
   fraction save, temp;
   save = f;
   while(f.denominator != 0) {
          temp.numerator = f.numerator;
          f.numerator = f.denominator;
          f.denominator = temp.numerator % f.denominator;
   }
   temp.numerator = save.numerator / f.numerator;
   temp.denominator     = save.denominator / f.numerator;
   // for negative fractions set numerator negative
   if(temp.denominator < 0) {
```

```
     temp.numerator = -temp.numerator;
         temp.denominator = - temp.denominator;
   }
   return temp;
}

//---------------------------------------------------------------------------//
//     Write the original matrix to the terminal
//---------------------------------------------------------------------------//

void WriteOriginal(long int n, fraction a[][N]) {
   long int j, k;

   printf("\n");
   for(j=0; j<n; j++) {
      for(k=0; k<n+1; k++)
        a[k][j].sprint();          // oprint
          printf("\n");
    }
   printf("\n");
}

//---------------------------------------------------------------------------//
//     Write the solution to the terminal
//---------------------------------------------------------------------------//

void WriteSolution(long int n, fraction a[][N], fraction x[]) {
   long int j, k;

   printf("\n");
   for(j=0; j<n; j++) {
      for(k=0; k<n+1; k++)
        a[k][j].sprint();
          printf(" |");
      x[j].sprint();
          printf("\n");
   }
   printf("\n");
}

//---------------------------------------------------------------------------//
//     Solve system of equations using Gaussian Elimination
//---------------------------------------------------------------------------//

double GaussSolve(long int n, fraction a[][N], fraction x[]) {
   /* n is the number of equations and unknowns */
   /* a[][n] is the input augmented matrix */
   /* x[n] is the output solution */

   long int i, j, k, maxrow;
   fraction tmp;
   fraction left;
   fraction right;

   for(i=0; i<n; i++) {
       /* Find the row with the largest first value */
      maxrow = i;
      for(j=i+1; j<n; j++) {
                a[i][j].fabs(left);
                a[i][maxrow].fabs(right);
         if(left > right)
            maxrow = j;
      }

      /* Swap the maxrow and ith row */
      for(k=i; k<n+1; k++) {
         tmp = a[k][i];
         a[k][i] = a[k][maxrow];
         a[k][maxrow] = tmp;
      }
```

```
      /* Singular matrix? */
      a[i][i].fabs(left);
        if(left == 0) return(FALSE);

      /* Eliminate the ith element of the jth row */
      for(j=i+1; j<n; j++) {
         for(k=n; k>=i; k--)
            a[k][j] = a[k][j] - a[k][i] * a[i][j] / a[i][i];
      }
   }

   /* Do the back substitution */
   for(j=n-1; j>=0; j--) {
      tmp = 0;
      for(k=j+1; k<n; k++)
         tmp = tmp + a[k][j] * x[k];
      x[j] = (a[n][j] - tmp) / a[j][j];
   }

   return(TRUE);
}

double main(void) {
   long int row, col;
   fraction a[N+1][N];    // a[col][row]
   fraction b[N];         // b[row]

   for(row=1; row<N; row++) {
     for(col=0; col<N; col++) {
         a[col][row].fpow(col+1, row);
     }
   }

   a[N][0].setfraction(1, 1);
   for(row=1; row<N; row++)     {
      a[N][row].setfraction(0, 1);
   }

   WriteOriginal(N, a);

   if(!GaussSolve(N, a, b))     {
      printf ("The Matrix is Singular");
          return -1;
   }

   WriteSolution(N, a, b);
   return 0;
}
```