Polynomial Integral Predictor / Corrector

Dennis E. Bahr, P.E.
October 2009

In the first two papers of this series we derived the equations to predict the future dependent variable values of polynomial functions and derivatives respectively. We can also develop equations that we can use to generate the integrals of polynomials by solving a set of equations as we did for the function and derivative. However, generating integrals is a bit more complicated and takes a lot more math.

In the case of an integral we want to calculate the area bounded by the dependent values y_i, y_{i+1}, and the independent values x_i and x_{i+1}. We are looking for the value of the integral at x_{i+1} for a predictor and x_i for a corrector knowing the values of the polynomial at x_{i-1} and x_i. As before, we assume that the spacing of the dependent values (x_i) is a constant and equal to the value h. I will develop equations for both prediction and correction, as these are what are required for solving differential equations using numerical analysis.

Let's begin with a first order predictor integral where the limits of integration are between x_{i-1} and x_i or x_{i-1} and $x_{i-1} + h$ and where we want to calculate the value of the integral at x_{i+1}. The general equation for such a first order indefinite integral can be written as:

$$I_{1P}(y_{i+1}) = Ax_{i+1}^2/2 + Bx_{i+1} + C - Ax_i^2/2 - Bx_i - C$$

We want to find an equation that uses the current data point and the previous point and finds the integral between x_i and x_{i+1}.

$$I_{1P}(y_{i+1}) = J(Ax_i + B) + K(Ax_{i-1} + B)$$

Putting these two together we have:

$$Ax_{i+1}^2 / 2 + Bx_{i+1} + C - Ax_i^2/2 - Bx_i - C =$$

$$JAx_i + JB + KAx_{i-1} + KB$$

$$A(x_i + h)^2/2 + B(x_i + h) - Ax_i^2/2 - Bx_i =$$

$$JAx_i + JB + KA(x_i - h) + KB$$

$$A(x_i^2 + 2x_ih + h^2)/2 + Bx_i + Bh - Ax_i/2 - Bx_i =$$

$$JA\ x_i + JB + KA\ x_i - KAh + KB$$

$$Ax_i^2/2 + Ax_ih + Ah^2/2 + Bx_i + Bh - Ax_i/2 - Bx_i =$$

$$JAx_i + JB + KAx_i - KAh + KB$$

Canceling like terms with opposite signs we have

$$Ax_ih + Ah^2/2 + Bh = JA\ x_i + JB + KA\ x_i - KAh + KB$$

Collecting the terms containing B

$$Bh = JB + KB \text{ or}$$

$$J + K = h$$

Collecting the terms containing A

$$Ax_ih + Ah^2/2 = JA\ x_i + KA\ x_i - KAh$$

$$x_ih + h^2/2 = J\ x_i + Kx_i - Kh$$

$$x_ih + h^2/2 = x_i(J + K) - Kh$$

Since J + K = h we have

$$x_ih + h^2/2 = x_ih - Kh$$

$$h^2/2 = -Kh$$

$$K = -h/2$$

Therefore:

$$J = 3h/2 \qquad \text{and} \qquad J + K = h$$

The equation for the first order integral predictor is therefore

$$I_{P1} = h(3y_i - y_{i-1})/2 + I_{P10}$$

This is equation 10.58 on page 329 in Jack Crenshaw's book[1].

This can be written using the Z transform operator as

$$I_{1P} = (3\backslash 2 - 1/2z^{-1})h + I_{1P0} \qquad [3.1]$$

The next step is to develop the first order corrector equation. Here we begin with the area bounded by the dependent values y_{i-1}, y_i and the independent values x_{i-1} and x_i and calculate the value of the integral at y_i.

$$I_{1C}(y_i) = J(Ax_i + B) + K(Ax_{i-1} + B)$$

$$Ax_i^2/2 + Bx_i + C - Ax_{i-1}/2 - Bx_{i-1} - C =$$

$$JAx_i + JB + KAx_{i-1} + KB$$

$$Ax_i^2/2 + Bx_i - A(x_i - h)^2/2 - B(x_i - h) =$$

$$JAx_i + JB + KAx_i - KAh + KB$$

$$Ax_i^2/2 - A(x_i^2 - 2x_ih + h^2)/2 + Bh =$$

$$JAx_i + JB + KA(x_i - h) + KB$$

$$Ax_ih - Ah^2/2 + Bh = JAx_i + JB + KAx_i - KAh + KB$$

Separating the A terms from the B terms we have

$$Bh = JB + KB$$

[1] Jack W. Crenshaw, **MATH Toolkit for REAL-TIME Programming**, CMP Books, 2000

Dividing out the B's

$J + K = h$

Now, let's take the A terms

$Ax_ih - Ah^2/2 = JAx_i + KAx_i - KAh$

Dividing out the A's

$x_ih - h^2/2 = x_i(J + K) - Kh$

But, we know from above that $J + K = h$

$x_ih - h^2/2 = x_ih - Kh$

Dividing by h yields

$x_i - h/2 = x_i - K$ or

Therefore:

$J = h/2$ and $K = h/2$

The equation for the first order integral corrector is therefore

$I_{1C} = h\backslash 2(y_i + y_{i-1}) + I_{1C0}$

This can be written using the Z transform operator as

$$\boxed{I_{1C} = (1\backslash 2 + 1/2z^{-1})h + I_{1C0}} \qquad [3.2]$$

This is equation 10.59 on page 329 in Jack Crenshaw's book[1].

Let's continue and develop the equations for a second order integral predictor.

$$I_{2P}(y_{i+1}) = J(Ax_i^2 + Bx_i + C) + K(Ax_{i-1}^2 + Bx_{i-1} + C)$$
$$+ L(Ax_{i-2}^2 + Bx_{i-2} + C)$$

[1] Jack W. Crenshaw, **MATH Toolkit for REAL-TIME Programming**, CMP Books, 2000

The term on the left hand side can be replaced with the general integral

$$I_{2P}(y_{i+1}) = Ax_{i+1}^3/3 + Bx_{i+1}^2/2 + Cx_{i+1} - Bx_i^3/3 - Bx_i^2/2 - Cx_i$$

Multiplying out terms on the left and right hand sides we have

$$Ax_{i+1}^3/3 + Bx_{i+1}^2/2 + Cx_{i+1} - Ax_i^3/3 - Bx_i^2/2 - Cx_i$$

$$= JAx_i^2 + JBx_i + JC + KAx_{i-1}^2 + KBx_{i-1} + KC$$

$$+ LAx_{i-2}^2 + LBx_{i-2} + LC$$

Replacing the subscript on the x terms with h we have

$$A(x + h)^3/3 + B(x + h)^2/2 + C(x + h) - Ax^3/3 - Bx^2/2 - Cx$$

$$= JAx^2 + JBx + JC + KA(x - h)^2 + KB(x - h) + KC$$

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

Expanding the powers

$$A(x^3 + 3x^2h + 3xh^2 + h^3)/3 + B(x^2 + 2xh + h^2)/2 + Cx + Ch$$
$$- Ax^3/3 - Bx^2/2 - Cx$$

$$= JAx^2 + JBx + JC + KA(x^2 - 2xh + h^2) + KBx - KBh + KC$$
$$+ LA(x^2 - 4xh + 4h^2) + LBx - 2LBh + LC$$

Multiplying out the terms and simplifying we have:

$$Ax^3/3 + Ax^2h + Axh^2 + Ah^3/3 + Bx^2/2 + Bxh + Bh^2/2 + Ch$$
$$- Ax^3/3 - Bx^2/2$$

$$= JAx^2 + JBx + JC + KA\,x^2 - 2KAxh + KAh^2 + KBx - KBh + KC$$
$$+ LAx^2 - 4LAxh + 4LAh^2 + LBx - 2LBh + LC$$

Simplifying terms we have:

$Ax^2h + Axh^2 + Ah^3/3 + Bxh + Bh^2/2 + Ch$

$= JAx^2 + JBx + JC + KA\ x^2 - 2KAxh + KAh^2 + KBx - KBh + KC$
$+ LAx^2 - 4LAxh + 4LAh^2 + LBx - 2LBh + LC$

Now we separate out the A, B, and C terms since they are orthogonal to each other and like terms on the left side must equal like terms on the right side. Let's start with the C terms since this is the simplest of the three equations.

$JC + KC + LC = Ch$ or

$J + K + L = h$

Next we do the B terms.

$Bxh + Bh^2/2 = JBx + KBx - KBh + LBx - 2LBh$

Dividing out the B's

$xh + h^2/2 = Jx + Kx - Kh + Lx - 2Lh$

We know from above the $J + K + L = h$ so we can write

$xh + h^2/2 = xh - Kh - 2Lh$

Simplify and divide out the h's

$h/2 = - K - 2L$

Finally we have

$K + 2L = - h/2$

Finally we do the A terms.

$Ax^2h + Ah^3/3 = JAx^2 + KA\ x^2 - 2KAxh + KAh^2 + LAx^2 - 4LAxh + 4LAh^2$

Divide out the A's

$x^2h + xh^2 + h^3/3 = Jx^2 + Kx^2 – 2Kxh + Kh^2 + Lx^2 – 4Lxh + 4Lh^2$

Since J + K + L = h we have

$x^2h + xh^2 + h^3/3 = x^2h – 2Kxh + Kh^2 – 4Lxh + 4Lh^2$

Divide all of the terms by h

$x^2 + xh + h^2/3 = x^2 – 2Kx + Kh – 4Lx + 4Lh$

We also know from above that K + 2L = – h/2 therefore,

$xh + h^2/3 = Kh + 4Lh + xh$

Again, divide each term by h and simplify

$K + 4L = h/3$

We now have three equations and with three unknowns

$J + K + 1L = h/1$
$K + 2L = – h/2$
$K + 4L = h/3$

If we solve for J, K, and L we find that

J = 23/12 K = –16/12 L = 5/12

Thus, we can write our second order predictor equation as

$I_{2P} = h\backslash 12(23y_i – 16y_{i-1} + 5y_{i-2}) + I_{2P0}$

Using the Z transform notation we have

$$I_{2P} = (23/12 – 4/3z^{-1} + 5/12z^{-2})h + I_{2P0} \qquad [3.3]$$

Going through the same derivation, but changing the equation to find the integral at x_i with an area bounded by $y_{i-1,}$ y_i and the independent variable between x_{i-1} and x_i we will get the equation for the corrector. The matrix for the second order corrector equation will look like this:

$$\begin{aligned} J + K + 1L &= h/1 \\ K + 2L &= h/2 \\ K + 4L &= h/3 \end{aligned}$$

Solving for J, K, and L we have

$$J = 12/5 \qquad K = 8/12 \qquad L = -1/12$$

Using these coefficients we can write the second order corrector equation

$$I_{2C} = h\backslash 12(5y_i + 8y_{i-1} - y_{i-2}) + I_{2C0}$$

The Z transform can be written as follows

$$\boxed{I_{2C} = (5/12 - 2/3z^{-1} + 1/12z^{-2})h + I_{2C0}} \qquad [3.4]$$

To develop equations of higher order we only need to expand the matrix to the order of the equation we wish to derive and solve for the coefficients. For a predictor, all of the terms on the right hand side with even numbered denominators are negative and for a corrector all of the terms on the right side are positive. Let's examine the matrix for a fourth order predictor.

$$\begin{aligned} J + K + 1L + 1M + 1N &= h/1 \\ K + 2L + 3M + 4N &= -h/2 \\ K + 4L + 9M + 16N &= h/3 \\ K + 8L + 27M + 64N &= -h/4 \\ K + 16L + 81M + 256N &= h/5 \end{aligned}$$

And now, examine the matrix for a fourth order corrector.

$$
\begin{array}{rrrrrrr}
J + & K + & 1L + & 1M + & 1N & = & h/1 \\
 & K + & 2L + & 3M + & 4N & = & h/2 \\
 & K + & 4L + & 9M + & 16N & = & h/3 \\
 & K + & 8L + & 27M + & 64N & = & h/4 \\
 & K + & 16L + & 81M + & 256N & = & h/5
\end{array}
$$

The terms on the left hand side are identical and form a nice geometric progression. The terms on the right hand side are also identical except for the differences in sign. Solving these matrices will give us the equations for a fourth order predictor and corrector.

The attached C^{++} program can solve for either the predictor or corrector equations by generating the matrix and solving for the unknowns.

Unfortunately, I could find no direct ties to Pascal's Triangle buried in these equations like I found in the function and derivative predictor equations. Many of the coefficients for the predictor and corrector integral equations are prime numbers indicating that any tie to Pascal's Triangle is deeply buried or simply not present.

Thus ends this series of predictor and corrector papers.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                          Integral.c                                   *
 *                                                                       *
 *               Calculates Coefficients for Integral                    *
 *                  Predictor / Corrector Equations                      *
 *                                                                       *
 *                      Dennis E. Bahr, P.E.                             *
 *                     Bahr Management, Inc.                             *
 *                   6632 North Chickahauk Trail                         *
 *                      Middleton, WI 53562                              *
 *                                                                       *
 *                        October 5, 2009                                *
 *                   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 5
#define N ORDER
#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];

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

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

   for(row=0; row<N; row++)     {
//      a[N][row].setfraction(pow(-1, row), row+1);       //Predictor
      a[N][row].setfraction(1, row+1);                  //corrector
   }

   WriteOriginal(N, a);

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

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