

# Polynomial Derivative Predictor

Dennis E. Bahr, P.E.

October 2009

In the first paper of this series we found that we could write a polynomial equation that could be used to determine or predict the next dependent data value in sequence from a set of previous values of a polynomial provided that the independent variable values were equally spaced. Can we do something similar for derivatives and knowing the present and previous dependent values find an equation to predict the next derivative in sequence? The answer is a resounding yes. Unfortunately, we cannot simply take the equations that we derived in the last paper and differentiate them. We must derive these derivative equations from scratch. The usual method of developing derivative formulas is to use a Taylor series or a Z transform power series and multiply out all of the terms and then collect like terms. For higher order derivatives this can be a very tedious and complex process.

The new method, that I am proposing, provides a process to develop polynomial derivative predictor equations. Once we get some of the lower order equations developed we will find that a pattern is established and we will be able to derive the higher order derivative equations by using some very simple rules.

Let's start with a first order derivative and solve for the coefficients. First we find the derivative of  $y_i$  or  $D(y_i)$ .

$$D(y_i) = D(A + Bx_i) = B$$

Next, we solve for the coefficients (J and K) of the equation that describes a first order derivative.

$$D(y_i) = Jy_i + Ky_{i-1}$$

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

$$B = JA + JBx_i + KA + KBx_{i-1}$$

Replace all  $x_{i-1}$  terms with  $x_i - h$

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

$$B = JA + JBx_i + KA + KBx_i - KBh$$

Separating out the A terms we have:

$$0 = JA + KA$$

Dividing each term by A we obtain

$$J + K = 0$$

Now, separating out the B terms:

$$B = JBx_i + KBx_i - KBh$$

Dividing each element by B and collections terms

$$1 = Jx_i + Kx_i - Kh$$

$$1 = x_i (J + K) - Kh$$

Since  $J + K = 0$  we have

$$K = -1/h$$

Given this and knowing that  $J + K = 0$  we obtain

$$J = 1/h$$

Our equation for a derivative becomes:

$$D_1(y_i) = (y_i - y_{i-1}) / h$$

This, of course, is the derivative for a first order function

The Z transform can be written as

$$D_1 = (1 - z^{-1}) / h$$

[2.1]

The solution for a second order function can be found in a similar fashion. We begin by writing the general form of a second order derivative.

$$D_2(y_i) = Jy_i + Ky_{i-1} + Ly_{i-2}$$

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

$$K(A + Bx_{i-1} + Cx_{i-1}^2) +$$

$$L(A + Bx_{i-2} + Cx_{i-2}^2)$$

Multiplying the terms out

$$B + 2Cx_{i+1} = JA + JBx_i + JCx_i^2 +$$

$$KA + KBx_{i-1} + KCx_{i-1}^2) +$$

$$LA + LBx_{i-2} + LCx_{i-2}^2)$$

Collecting and solving for the A terms gives us the following

$$0 = JA + KA + LA$$

Divide each term by A and rearranging terms we have

$$J + K + L = 0$$

Next we Collect the B terms and set them equal to each other, reference all x's to  $x_i$  with the variable h, and divide by B.

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

$$1 = Jx_i + K(x_i - h) + L(x_i - 2 * h)$$

Multiplying and collecting terms we have

$$1 = x_i(J + K + L) - Kh - 2Lh$$

Replace  $J + K + L$  with zero from above

$$1 = - Kh - 2Lh$$

Finally we have

$$K + 2L = - 1 / h$$

Lastly, we collect the C terms

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

Dividing by C and multiplying out the terms

$$2x_i = Jx_i^2 + K(x_i^2 - 2x_i + h^2) + L(x_i^2 - 4x_i + 4h^2)$$

Multiplying out and collecting terms

$$2x_i = x_i^2(J + K + L) - 2Kx_ih + Kh^2 - 4Lx_ih + 4Lh^2$$

Since  $J + K + L = 0$ , we have

$$2x_i = -2Kx_ih + Kh^2 - 4Lx_ih + 4Lh^2$$

Collection and rearranging terms

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

Using the equation above  $Kh + 2Lh = -1$  we have,

$$2x_i = 2x_i + h^2(K + 4L)$$

$$0 = h^2(K + 4L) \text{ or } 0 = (K + 4L)$$

We now have three equations with three unknowns

$$J + K + L = 0$$

$$K + 2L = -1/h$$

$$K + 4L = 0$$

Since the equations for the terms are linear in nature one can easily solve them using Gauss Elimination or some other more sophisticated method.

Solving for the unknowns we finally have

$$J = 3 / 2h \quad k = -2 / h \quad L = 1 / 2h$$

The equation for the derivative can now be written as:

$$D_2(y_i) = 3y_i / 2h - 2y_{i-1} / h + y_{i-2} / 2h$$

This can also be written using the Z transform operator as

$$\boxed{D_2 = 1/2h (3 - 4z^{-1} + z^{-2})} \quad [2.2]$$

If we continue with a third order derivative we find that the simultaneous equations that we generate are

$$\begin{aligned} J + K + L + M &= 0 \\ K + 2L + 3M &= -1/h \\ K + 4L + 9M &= 0 \\ K + 8L + 27M &= 0 \end{aligned}$$

Solving these equations we find that

$$J = 11/6h \quad k = -3/h \quad L = 3/2h \quad M = -1/3h$$

An equation can be written using these coefficients and the Z transform operator as:

$$\boxed{D_3 = 1/6h (11 - 18z^{-1} + 9z^{-2} - 2z^{-3})} \quad [2.3]$$

We now have a pattern that allows us to write the equations for a fourth order derivative by expanding the matrix to:

$$\begin{aligned} J + K + L + M + N &= 0 \\ K + 2L + 3M + 4N &= -1/h \\ K + 4L + 9M + 16N &= 0 \\ K + 8L + 27M + 64N &= 0 \\ K + 16L + 81M + 256N &= 0 \end{aligned}$$

The solution to these equations is

$$J = 25/12h \quad K = -4/h \quad L = 3/h \quad M = -4/3h \quad N = 1/4h$$

Again, writing an equation using the Z transform operator we have

$$\boxed{D_4 = 1/12h (25 - 48z^{-1} + 36z^{-2} - 16z^{-3} + 3z^{-4})} \quad [2.4]$$

Now comes the interesting part. If we take the solution vector for the equations for the fourth order derivative above and divide each coefficient by 12 we get the following.

$$D_4 = (25/12 - 4/1z^{-1} + 6/2z^{-2} - 4/3z^{-3} + 1/4z^{-4}) / h$$

The last four terms of the solution matrix have coefficients with numerators that are from a row of Pascal's Triangle. Each of the last four terms is divided by an integer that increases in value by one. The first term is the sum of the coefficients of the other four terms with a sign change. When all of the z terms have the same value the sum of these terms becomes  $-25/12$  and the derivative is zero, as it should be.

Let us use the rule and derive the equation for the 5<sup>th</sup> order derivative predictor. We write down the terms from the fourth row of Pascal's triangle and divide each one by an increasing constant beginning with the number one. To calculate the first term we add the remaining five terms and change the sign. Therefore, we have

$$137/60 \quad -5/1 \quad 10/2 \quad -10/3 \quad 5/4 \quad -1/5$$

The difference equation using these coefficients then becomes

$$D_5 = (137/60 - 5/1z^{-1} + 10/2z^{-2} - 10/3z^{-3} + 5/4z^{-4} - 1/5z^{-5}) / h$$

Multiplying each of the coefficients by 60 and then placing 1/60 outside the parentheses one gets the equation that is usually found in the literature. Just looking at this equation, one does not get the sense of the beautiful underlying structure.

$$D_5 = 1/60h(137 - 300z^{-1} + 300z^{-2} - 200/3z^{-3} + 75/4z^{-4} - 12z^{-5}) / h$$

We can easily write the Z Transform equation for the solution where we used the rules.

$$D_5 = (137/60 - 5z^{-1}/1 + 10z^{-2}/2 - 10z^{-3}/3 + 5z^{-4}/4 - 1z^{-5}/5) / h \quad [2.5]$$

So, we can throw away long and complex derivations and write these equations using a few simple rules. The rules to find the predictor equation for derivatives is to pick the order, find the order row in Pascal's Triangle, write down the terms, divide them in sequence by an increasing integer, and then alternate the signs. The constant term is the sum of the other terms with a sign change.

The last paper in this series will deal with developing a pattern and equations for numerical integration and in this case I will develop equations to do both prediction and correction.

The Z transform equations that I derived in this paper can all be found on pages 303 and 304 in Jack Crenshaw's book<sup>1</sup>.

---

<sup>1</sup> Jack W. Crenshaw, MATH Toolkit for REAL-TIME Programming, CMP Books, 2000



```

    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("%5ld/%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].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++) {
        if(row != 1)
            a[N][row].setfraction(0, 1);
        else
            a[N][1].setfraction(-1, 1);
    }

    WriteOriginal(N, a);

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

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

```