Polynomial Taylor Shifting
By Henrik Vestermark (hve@hvks.com)

Abstract

We present two classic methods for Polynomial Taylor shifting. Shaw-Traub and the Horner method. We present it using the Matrix form most often found in the literature. However, both methods' matrix forms can be boiled down to a set of simple row operations.

Contents

Abstract 1
Polynomial Taylor Shifting 2
Introduction 2
Shaw-Traub Taylor shifting 3
Algorithm for Polynomial Taylor shift with real coefficients 3
Optimized Algorithm for Polynomial Taylor shift with real coefficients 4
Taylor shifting using the Horner method 5
Algorithm for Horner Polynomial Taylor shift with real coefficients 5
Optimized Algorithm for Horner Polynomial Taylor shift with real coefficients 6
Reference 7

Polynomial Taylor Shifting

Introduction

Sometimes it can be practical not to solve a given Polynomial directly but instead solve a Polynomial where all the roots are shifted a certain distance from the original polynomial. A classic example is the Rutishauser QD method for finding Polynomial roots. One of the drawbacks of the Rutishauser QD method is that it requires all coefficients to be $a_i \neq 0 \; for \; i = 0,\ldots,n$. For the Polynomial below:

$$P(z) = a_n z^n + a_{n-1} z^{n-1} +, \ldots, a_1 z + a_0 \qquad 1$$

E.g., The Polynomial x^5-1 cannot be solved directly with that method. However, if we Taylor shift the roots to the left with 2 we get a new Polynomial
$x^5+10x^4+40x^3+80x^2+80x+31$
Now all the coefficients $a_i \neq 0 \; for \; i = 0,\ldots,n$. Moreover, we can now find the roots of the new Polynomial to be:

	X1=	-0.9999999999999998
	X2=	(-2.8090169943749466+i0.5877852522924708)
	X3=	(-2.8090169943749466-i0.5877852522924708)
	X4=	(-1.6909830056250537-i0.951056516295154)
	X5=	(-1.6909830056250534+i0.9510565162951539)

Adding the shifting value back (+2), you get:

	X1=	+0.9999999999999998
	X2=	(-0.8090169943749466+i0.5877852522924708)
	X3=	(-0.8090169943749466-i0.5877852522924708)
	X4=	(0.3090169943749745+i0.9510565162951536)
	X5=	(0.3090169943749745-i0.9510565162951535)

Which is the root of the Polynomial x^5-1.

J Gathen [1] is a good reference for fast Taylor shifts algorithms for a Polynomial in the form:

$$P(z) = a_n z^n + a_{n-1} z^{n-1} +, \ldots, a_1 z + a_0 \qquad 2$$

Shaw-Traub Taylor shifting

Here is the Shaw-Traub 1974 algorithm where z_0 is the shift value

Given $P(z) = a_n z^n + a_{n-1} z^{n-1}, \dots, a_1 x + a_0$ 3
We try to find Polynomial $Q(z) = q_n z^n + q_{n-1} z^{n-1}, \dots, q_1 x + q_0$
That represents the z_0-shifted Polynomial.
Arrange P(z) in a matrix form, where z_0 is the shift value:

$$M = \begin{bmatrix} a_{n-1}z_0^{n-1} & a_n z_0^n & & \dots & & \\ a_{n-2}z_0^{n-2} & & a_n z_0^n & \cdots & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 z_0^1 & & & \dots & a_n z_0^n & \\ a_0 z_0^0 & & & \dots & & a_n z_0^n \end{bmatrix}$$

Compute: $t_{i,,j+1}$=M[i,j+1]=M[i-1,j]+M[i-1,j+1] for j=0,1,…,n-1;
i=j+1,…n

$$M = \begin{bmatrix} a_{n-1}z_0^{n-1} & a_n z_0^n & & \dots & & \\ a_{n-2}z_0^{n-2} & t_{1,1} & a_n z_0^n & \dots & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 z_0^1 & t_{n-1,1} & t_{n-1,2} & \cdots & a_n z_0^n & \\ a_0 z_0^0 & t_{n,1} & t_{n,2} & \dots & t_{n,n-1} & a_n z_0^n \end{bmatrix}$$

Then $q_i = \frac{M[n,i+1]}{z_0^i} = \frac{t_{n,i+1}}{z_0^i}$ for i $= 0,1, \dots,$ n $- 1;$ *and* $q_n = a_n$

Algorithm for Polynomial Taylor shift with real coefficients

```
/*
Given the n - degree polynomial : p(x) = anx^n + an - 1x^n - 1 + ... + a1x + a0
We must obtain new polynomial coefficients qi, by Taylor shift q(x) = p(x + x0).
We'll use the matrix t of dimensions m x m, m=n+1 to store data.
Compute ti, 0 = an - i - 1x0^(n - i - 1) for i = 0..n - 1
Store ti, i + 1 = anx0^n for i = 0..n - 1
Compute ti, j + 1 = ti - 1, j + ti - 1, j + 1 for j = 0..n - 1, i = j + 1..n
Compute the coefficients : qi = tn, i + 1 / x0^i for i = 0..n - 1
The highest degree coefficient is the same: qn = an
*/
void taylorShift(const int n, double a[], double shift)
	{
	int i, j, m = n + 1;
	double **t;
	if (shift == 0) return; // No shift, no change
	t = new double *[m];
	for (i = 0; i < m; ++i)
		t[i] = new double[m];
	for (i = 0; i < n; ++i)
		{
		t[i][0] = a[ i + 1 ] * pow(shift, n - i - 1);
```

```
            t[i][i + 1] = a[0] * pow(shift, n);
            }
      for (j = 0; j < n; ++j)
            for (i = j+1; i <= n; ++i)
                  t[i][j + 1]=t[i-1][j]+t[i-1][j+1];
      for (i = 0; i < n; ++i)
            a[n-i] = t[n][i + 1 ] / pow(shift, i);
      for (i = 0; i < m; ++i)
            delete t[i];
      delete [] t;
      }
```

We can easily extend the above algorithm to Polynomial with Complex coefficients and Complex shifting values.

Quite simple and easy to do it however, we can optimize the algorithm by noticing that we generate each t's from a row based on summarizing two elements from the previous row. See below where $t_{1,1}$ is formed by adding $t_{1,1} = a_n z_0^n + a_n z_0^n$ from the previous row.

$$M = \begin{bmatrix} a_{n-1}z_0^{n-1} & a_n z_0^n & & \dots & & \\ a_{n-2}z_0^{n-2} & t_{1,1} & a_n z_0^n & \dots & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 z_0^1 & t_{n-1,1} & t_{n-1,2} & \cdots & a_n z_0^n & \\ a_0 z_0^0 & t_{n,1} & t_{n,2} & \dots & t_{n,n-1} & a_n z_0^n \end{bmatrix} \quad (4)$$

Or $t_{n,2}$ is formed by adding $t_{n,2} = t_{n-1,1} + t_{n-1,2}$ from the previous row.
Instead of arranging it as a matrix, we can simply just create a single row R that we initialize as follows.

$$R = [a_{n-1}z_0^{n-1} \quad a_n z_0^n \quad a_n z_0^n \quad \dots \quad a_n z_0^n \quad a_n z_0^n] \quad (5)$$

Now we use the same computational algorithm as before but start backward for each row to be able to merge it into a single-row operation.

Compute: t_i=R[i]=R[i-1]+R[i] for j=1,…,n; i=j,…1 and for each new j initialize R[0]=$a_{j+1}z_0^{n-j-1}$

Optimized Algorithm for Polynomial Taylor shift with real coefficients

```
// Optimized algorithm for Taylor shifting using only a single row instead of a
// matrix
void taylorShift(const int n, double a[], double shift)
      {
      int i, j, m = n + 1;

      if (shift == 0) return; // No shift, no change
```

```
        double *b = new double[m];
        b[0] = a[1] * pow(shift, n - 1);
        for (i = 1; i <= n; ++i)
                b[i] = a[0] * pow(shift, n);
        for (j = 1; j <= n; ++j)
                {
                for ( i = j; i >=1 ; --i)
                        b[i] = b[i] + b[i - 1];
                if (j == n) b[0] = 0; else
                b[0] = a[j + 1] * pow(shift, n - j - 1);
                }

        for (i = 0; i < n; ++i)
                a[n - i] = b[i + 1] / pow(shift, i);
        delete[] b;
        }
```

Taylor shifting using the Horner method

There is also another classic method that just uses a simple *Horner* schema to compute the Taylor shift. See Henrici [2] for the Polynomial:

$$P(z) = a_n z^n + a_{n-1} z^{n-1} +, \dots, a_1 z + a_0 \qquad 6$$

Again, when describing the method you usually set it up in a matrix form like the Shaw-Traub algorithm however, initialize it differently and z_0 is the shift value.

$$M = \begin{bmatrix} a_n & a_{n-1} & a_{n-2} & \dots & a_1 & a_0 \\ a_n & t_{1,1} & t_{1,2} & \cdots & t_{1,n-1} & t_{1,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_n & t_{n-2,1} & t_{n-2,2} & \dots & & \\ a_n & t_{n-1,1} & & \dots & & \\ a_n & & & \dots & & \end{bmatrix} \qquad 7$$

Compute: $t_{i,,j+1}$=M[j,i]=z_0M[j,i-1]+M[j-1,i] for j=1,…,n; i=1,…n-j+1
At the end of this computation, the new shifting polynomial coefficient is diagonal from the lower left corner to the upper right corner (blue color).

Algorithm for Horner Polynomial Taylor shift with real coefficients

```
// Taylor shift using Horner matrix
void taylorShiftHornermatrix(const int n, double a[], double shift)
        {
        int i, j;
        int m = n + 1;
        double **t;
        if (shift == 0) return; // No shift, no change
```

```
	t = new double *[m];
	for (i = 0; i < m; ++i)
		t[i] = new double[m];
	for (i = 0; i <= n; ++i)
		{
		t[0][i] = a[i];
		if(i!=0)
			t[i][0] = a[0];
		}
	for (i = 1; i <= n; ++i)
		a[i] = t[n - i + 1][i];
	for (j = 1; j <= n; ++j)
		for (i = 1; i <= n-j+1; ++i)
			t[j][i] = shift*t[j][i-1]+t[j-1][i];
	for (i = 1; i <= n; ++i)
		a[i] = t[n-i+1][i];
	for (i = 0; i < m; ++i)
		delete t[i];
	delete[] t;
	}
```

Somehow similar to the Shaw-Traub Algorithm, however you avoid the call to the pow() function at the expense of an extra multiplication. As we saw with the Traub-Shaw algorithm, the Polynomial can easily be extended to Complex coefficients and or Complex shift values.

As for the Shaw-Traub method, you can also optimize the algorithm and reduce the matrix to a set of single-row operations. As can be seen, the next *t* is computed using the previous row element and the current row element and therefore we can again boil down the algorithm to a very efficient version as outlined below.

Optimized Algorithm for Horner Polynomial Taylor shift with real coefficients

```
// Optimized Polynomial Taylor shift using the Horner method
void taylorShiftHorner(const int n, double a[], double shift)
	{
	int i, j;
	if (shift == 0) return; // No shift, no change
	for (j = 1; j <= n; ++j)
		for (i = 1; i <= n - j + 1; ++i)
			a[i] += shift*a[i - 1];
	}
```

This is the fastest of the two algorithms presented. The above algorithm has the advantage that you do not need to temporarily allocate memory to hold the matrix or a single row but can work directly on the coefficients on the original Polynomial.

Reference

1. J Gathen, Jürgen Fast Algorithm for Taylor sifts and certain Difference Equations
2. P. Henrici, Elements of Numerical Analysis, 1964 Willey