A FAST GRADIENT FILL ALGORITHM

CHRIS LOMONT

Abstract. Repeated evaluation of some mathematical function is the bottleneck in many digital applications, from image and sound processing, to video games and CAD systems. Techniques to speed up such functions are needed to make the applications usable, but many programmers are unaware of some useful methods to improve code speed. This article explains a few such techniques by working through a simple yet instructive example: optimizing a circular gradient fill used in paint programs.

1. Introduction

We all wish our code were error free, easily maintained, sold a million copies, and of course we want it to execute *fast*. Well, there is not enough space in this article to show you how to do all of that (even if I could!), but I will show you techniques to speed up that slow section of code. In particular, if you have to evaluate a costly math function often, like filters used in image or sound processing, this article should allow a performance boost. An even greater boost may be obtained for a very small error. For many applications, like video games, this is certainly allowable. Please do not use the latter trick in life critical code!

2. The Problem

I will demonstrate how to apply polynomial approximation, forward differencing, and fixed point math to obtain a 20 fold increase in the speed of a simple application - a circular gradient fill (see Image 1 for example). The problem we wish to code is as follows:

Fill a $2w$ by $2h$ pixel area with a radial gradient; the center color is denoted c_1 and the outer color c_2. Assume the center of the rectangle has coordinates $(0, 0)$, the upper right corner is (w, h), and to be precise we use pixels coordinates that are integer plus $\frac{1}{2}$, i.e., in the first quadrant the pixel closest to the origin has coordinates $(\frac{1}{2}, \frac{1}{2})$. For position (x, y) we then wish to draw a color defined as follows:

Let $M = \sqrt{(w - 0.5)^2 + (h - 0.5)^2}$ be the largest distance from the origin a pixel needs plotted. Let $r = \sqrt{x^2 + y^2}$ be the distance from the

origin. Then linearly interpolating the color along the radial direction, the blending ratio between the two end colors is $\alpha = \frac{r}{M}$. So the *exact* color at (x, y) should be $c_e = \alpha c_2 + (1 - \alpha)c_1$, which we can write as $c_e = \alpha(c_2 - c_1) + c_1$. Letting $\Delta c = c_2 - c_1$, and assuming the color is rounded down the the nearest integer by the hardware, we label the color written c_w as

$$c_w = \lfloor r\frac{\Delta c}{M} + c_1 \rfloor$$

where $\lfloor\ \rfloor$ denotes rounding down to the lowest integer (we could also add $\frac{1}{2}$ inside the brackets to round to nearest integer). This has the advantage that $\frac{\Delta c}{M}$ is constant throughout the the fill, and only has to be calculated once. Thus we get the first piece of code - see Listing 1 (GradientFill_1 in the final source listing). We assume pixel values are integers in the range 0=black to 255=white. Colored gradients can be done by interpolating each of the RGB components. For consistency, we will use white at the center of the gradient and black at the outermost pixels.

Using a Dual PIII 1000 MHZ with 1.5 GB of RAM, this takes 5158 ms to do 250 fills on a 400 by 400 gradient. This will be our baseline time and image (Image 1) to which we compare further algorithms.

3. Initial Optimizations

3.1. **Exploiting Symmetry.** We use the symmetry to compute one quadrant, and reuse each computation to draw four pixels. This reduces the time to 1597 ms with identical output, for a net 3.31 fold speedup. This is not quite a fourfold speedup since we still have to draw the pixels, and gives function GradientFill_2 in the listing. Next we write directly to the surface instead of calling "SetPixel" each pixel, giving 1186 ms with identical output to the initial function, and a 4.46 fold speedup. The result is replacing the drawing loop with the loop in Listing 2 (GradientFill_3). From now on we only need to compute quadrant I, that is, where the x and y coordinates are positive (remember the "first" pixel is coordinate $(\frac{1}{2}, \frac{1}{2})$). Now on to the mathematical speedups!

3.2. **Polynomial Approximation.** Two real time killers in the pixel drawing loop are the square roots to compute distance, and the floating to integer color conversion for each pixel plotted. So we will address these in turn. We want to remove the square root and replace it with something faster. The first idea will be approximation with a polynomial, but there are many types from which to choose. In general

higher degree polynomials will give better approximations, but this is not *always* true. The first one we will try is a Taylor series, which most students see in a calculus class for the first time, and is the only polynomial they think of to approximate other functions. Also, we will try a interpolating polynomials: one chosen to interpolate evenly spaced points, another chosen to interpolate the *Chebyshev* points [3], and finally ones we will call "Super Polynomials" designed specifically for our function. We call them TPolys, IPolys, CPolys, and SPolys respectively.

To help in choosing a polynomial, we construct a harness using the exact square root function. We will first scale the dimensions of the gradient so it fits in a square with coordinates $[-1, 1]$x$[-1, 1]$, which makes it easier to develop the polynomials. After debugging the harness to make sure it produces identical output to the correct gradient fill, we can replace the square root call with a function pointer that evaluates a polynomial of choice. That way we can see what polynomials look acceptable, before we choose a specific polynomial to further optimize. This function is shown in Listing 3 (GradientFill_4 in the final listing).

Then, for a *fixed* value of y in $[0, 1]$ we will approximate a scanline (the left to right span of pixels with constant y value) with a polynomial approximating the true scanline function $f(x) = \sqrt{x^2 + y^2}$. One way to construct our polynomials (besides by hand!) is to use a package like Maple [5] or Mathematica [6]. We will create degree two, three, and four polynomials, each in four flavors - the TPoly, IPoly, CPoly, and SPoly as listed above. As we will see, the oddly spaced interpolation points of the CPolys give a better fit than the equally spaced IPolys, which raises the question: can we pick even better point spacing to interpolate this function? The answer is yes, although it is hard to get the truly best ones due to the difficulty of the calculation (setting up exactly what I want kills all the symbolic math packages - the resulting problem is very hard!). However, we can find interpolation points that minimize the total squared error over [0,1], and so we add such polynomials, and call them "Super Polynomials" (named after Super-Man, of course, unless there are copyright problems).

The Taylor series is expanded about $x = \frac{1}{2}$ to reduce error over the range $[0, 1]$. The Mathematica code, Section I, shows how to construct the quadratic polynomials in four flavors; the rest are similar. We use the Mathematica command **CForm** to get text we can cut and paste into C code. For example, we create a function for a degree 2 Taylor series approximating $f(x)$, shown in Listing 4 (function TPoly2 in the final listing).

So now we have 12 polynomials to test. We expect the higher degree versions of each type to make better images, but what about comparing between the types? There are theoretical reasons the CPolys should be superior to the TPolys and IPolys, but as programmers say, the proof is in the pixels. So we compare each of the 12 images to the correct image to obtain Image 2 (the function ShowPolys in the final listing makes this image).

Left to right are degrees 2, 3, and 4. The top row are the TPolys, with their errors relative to the correct image to the right of each gradient. The error is computed as the absolute difference in the pixel values (0-255), then is multiplied by 60 to show small errors formed, and is clamped at a max of 255. The second row is the IPolys, then the CPolys, and the last row is the SPolys. Notice the error pattern in the top row of TPolys, and how this error pattern decreases from left to right. Since we expanded the Taylor polynomial around $\frac{1}{2}$, they have little error in the middle of each quadrant, which corresponds the the blackness in the columns 1/4 and 3/4 of the way across each image, but the error increases near the edges of the quadrants, where there are white error pixels. Between the same degree IPolys and CPolys, clearly the CPolys are superior. Looking at the images, the quadratic CPoly has some vertical bulge, but the cubic and quartic ones make very good images. Finally, the SPolys do pretty well, and seem a little better than the equal degree CPolys. The following table shows the error data:

Type	deg 2	deg 3	deg 4
TPoly	253, 1.556	12, 0.511	9, 0.241
IPoly	7, 1.806	4, 0.371	3, 0.109
CPoly	6, 1.037	3, 0.240	2, 0.077
SPoly	7, 0.890	4, 0.211	2, 0.066

Each entry shows the maximal pixel error, and the average pixel error. For example, the degree 3 "Super Polynomial" approximation creates a maximal error of 4 values out of 255, and averages 0.211 error per pixel on the test gradient.

For the next step of our optimization we will replace the polynomial with a forward differencing scheme [2]. Larger degrees are a bit slower per pixel, so we will choose to optimize a cubic, since they have much better images then the quadratic polynomials, and are less expensive to compute than the quartics. The best two cubics are the CPoly and the SPoly, but since the coefficients of the CPoly are easier to evaluate, and it does seem to have a slightly smaller max error than the SPoly,

we choose the cubic CPoly. Chebyshev, step on down! You are our next contestant!

For fun (I don't get out much) we time a version using the cubic CPoly (cleaned up from the Mathematica version to remove redundant computation) and find it requires 3991 ms initially (GradientFill_4) when evaluating the whole polynomial each pixel. If we precompute the polynomial coefficients once per scanline (GradientFill_5), it requires 746 ms , for a 7.1 fold increase over baseline performance. Most of the error in the final image we will produce comes from this step. If your application cannot tolerate much error, you need better polynomials or other approximants.

3.3. **Forward Differencing.** Now we turn to the next optimization topic: forward differencing. The idea is to replace costly polynomial evaluations with repeated addition. The idea almost seems like magic (and is yet another reason to read up on math topics!), and goes as follows: Suppose we need to evaluate a polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1}+...+a_1 x^1+a_0$ at the *equally spaced* points $x_0, x_0+d, x_0+2d, ...$ Then there is a method, explained below with an example, to do it using n additions per evaluation, after the magic setup of n values. This is much better than the usual n additions and $n-1$ multiplications per evaluation.

For example, suppose we need to evaluate a quadratic polynomial $p(x) = ax^2 + bx + c$ at the points $x_0, x_0 + d, x_0 + 2d, ..., x_0 + md$. We begin by defining the *first forward difference polynomial* $\Delta p(x)$ as

$$\begin{aligned} \Delta p(x) &= p(x+d) - p(x) \\ &= a(x+d)^2 + b(x+d) + c - (ax^2 + bx + c) \\ &= 2adx + bd + ad^2 \end{aligned}$$

This can be rewritten as $p(x + d) = p(x) + \Delta p(x)$, which is true for any value of x! So if we know $p(x_0)$ and $\Delta p(x_0)$, we can compute $p(x_0+d)$ by just adding the two known values at x_0. $\Delta p(x)$ is a function that tells how much to add to $p(x)$ to get $p(x+d)$, thus the clever name "forward difference". Notice $\Delta p(x)$ is linear, so is quicker to evaluate than the quadratic $p(x)$. Thus we replaced evaluating the 2nd degree polynomial $p(x)$ with evaluating the first degree one $\Delta p(x)$. We repeat this trick on $\Delta p(x)$ getting the *second forward difference polynomial* $\Delta^2 p(x)$

$$\begin{aligned}\Delta^2 p(x) &= \Delta p(x+d) - \Delta p(x)\\ &= (2ad(x+d) + bd + ad^2) - (2adx + bd + ad^2)\\ &= 2ad^2\end{aligned}$$

This is just a constant, so can be precomputed outside of the loop. Thus to evaluate the sequence of points we want, we precompute $p(x_0)$, $\Delta p(x_0)$, and $\Delta^2 p(x_0)$, from which we can compute the rest of the points by clever additions. This enables us to replace the (pseudo C code) loop

```
for (i = x0; i < x0 + 1000d; i += d)
    print a*i*i+b*i+c;
```

with the much faster

```
p0 = ax0^2 + bx0 + c;
Δp = 2adx0 + bd + ad^2;
Δ^2p = 2ad^2;
for (i = x0; i < x0+1000d; i += d)
    {// Notice the loop only needs 2 additions per pass!
    print p0;
    p0 += Δp;
    Δp += Δ^2p;
    }
```

To turn higher degree polynomials into forward differencing loops, just repeat the derivation of the second, third, fourth, etc., forward differences.

We are now interested in turning the cubic CPoly into such a method, so to save time we derive the computations in Mathematica (or by hand...). See the Mathematica code section II. We obtain for a starting cubic polynomial $p(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0$ and stepsize δ the following forward differences:

$$\begin{aligned}p_0 &= a_3x_0^3 + a_2x_0^2 + a_1x_0^1 + a_0\\ \Delta p_0 &= 3a_3\delta x_0^2 + (2a_2 + 3\delta a_3)\delta x_0 + (a_1 + \delta a_2 + \delta^2 a_3)\delta\\ \Delta^2 p_0 &= 6a_3\delta^2 x_0 + 2\delta^2(a_2 + 3a_3\delta)\\ \Delta^3 p_0 &= 6a_3\delta^3\end{aligned}$$

We convert Listing 4 to use the cubic CPoly, and use forward differencing, to obtain Listing 5 (GradientFill_6). This routine uses 424 ms , a 12.5 fold speed improvement over the baseline gradient, with

very little visual quality loss (see Image 3). Although converting polynomials to forward differencing can introduce some roundoff error, it turns out in this case it does not, since we round to integers on output. This is checked in the final code by the call CompareMethods(GradientFill_5,GradientFill_6). Thus replacing polynomial approximation with forward differencing almost doubled the throughput! A useful technique indeed!

3.4. **Fixed Point.** The other bottleneck mentioned was the floating point to integer conversion done each pixel. We will convert the floating point math in the loop to fixed point [1], which is a way to approximate floating point calculations with integer code. Since we want 8 bits of color in the output, and choose to fit in a 32 bit architecture, we will take 8 bits of significant data and 24 bits of fraction. Listing 6 (GradientFill_7) is our final version of the routine. It takes 233 ms to run the test above, for an incredible 22.7 fold speed improvement. For a final comparison, this function on test images gets a max error of 3 and per pixel error of 0.2401. That means our final optimized version is within 3 color values out of 255 over the entire gradient, and averages being off 1 value about 1 out of 5 pixels. Visually, it is a very good gradient. (See Image 4). Note switching to fixed point only caused a few errors over the last function (only 16 pixels were wrong, each by 1 value, over a 200 by 200 gradient).

4. Conclusion

In summary, we have seen several techniques for speeding up nonlinear functions evaluated on gridpoints. The first one was to approximate expensive functions over a bounded range with appropriate polynomials. Probably the best place to start is using polynomials with roots at Chebyshev points spread over the interval. The next technique was to speed up polynomial evaluation. If the evaluation is over equally spaced grid points, use forward differencing, which replaces degree n polynomial evaluation with n additions, at some loss of precision. Finally, in the gradient example, we could replace the floating point math with faster integer math, since the output was not affected much by such a change.

The source code listing gives a complete program to test and play with these ideas. It writes text data to the console, and in Windows will write directly on the desktop, avoiding the extra code needed to make a windows friendly program. The main function has numerous tests you can run by uncommenting the lines you want. At the top of the file are variables defining the gradient size.

Homework:

1. Can you beat the above algorithms using again (gasp!) square roots (but very optimized ones! search the web for fast square root routines or see [4]) and fast float to integer conversion tricks (again, the code is on the web!)?

2. Think of a use involving more variables, and see if these techniques still are useful.

References

[1] http://www.cstone.net/ kyoung/fix1FAQ.html
http://www.cstone.net/ kyoung/fix2FAQ.html.
[2] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, *Computer Graphics: Principles and Practice in C (2nd Edition)*, Addison-Wesley, 1995.
[3] Walter Gautschi, *Numerical Analysis: An Introduction*, Birkhauser, 1997.
[4] Andrew Glassner (editor), *Graphics Gems*, Academic Press, 1990.
[5] Maple Software - www.maplesoft.com
[6] Mathematica - www.mathematica.com

Department of Mathematics, 150 North University Street, Purdue University, West Lafayette, Indiana 47907-2067
Email address: clomont@math.purdue.edu
First written: Nov 2002

Image 1 - The original circular gradient

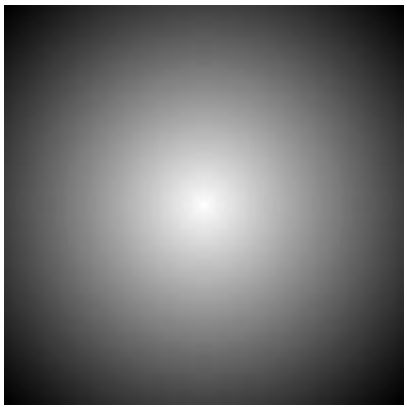


Image 2 - Errors

Each gradient of degree 2,3,4 left to right has to its right the error to the original image. The rows are, top to bottom, the TPoly, IPoly, CPoly, and SPoly.

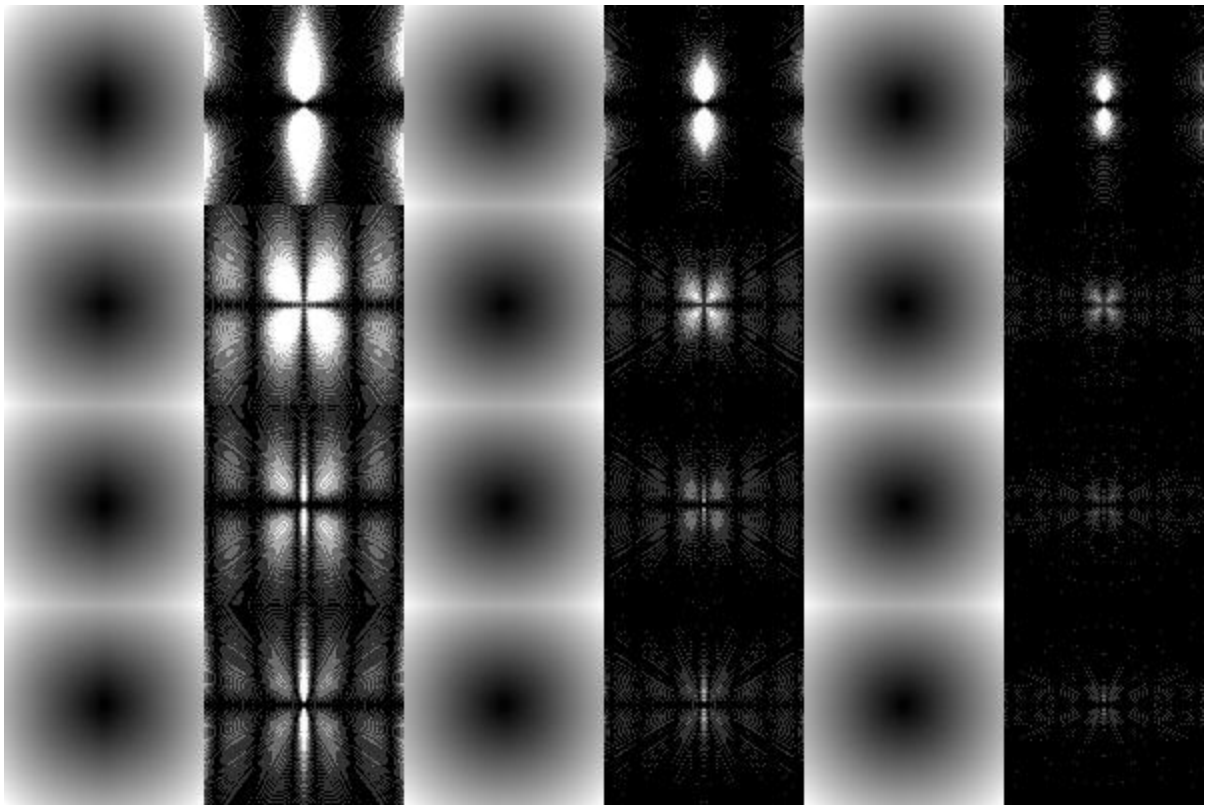


Image 3

The error from switching to a forward difference, polynomial approximation.
The images are the original gradient, the new one, and the relative and total error.

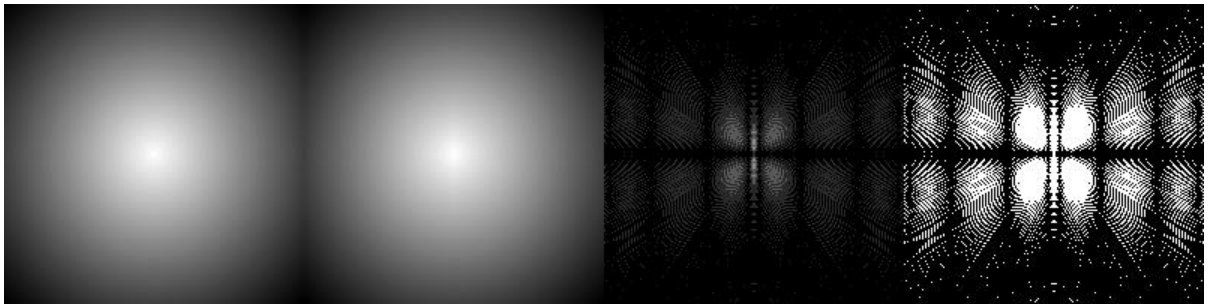
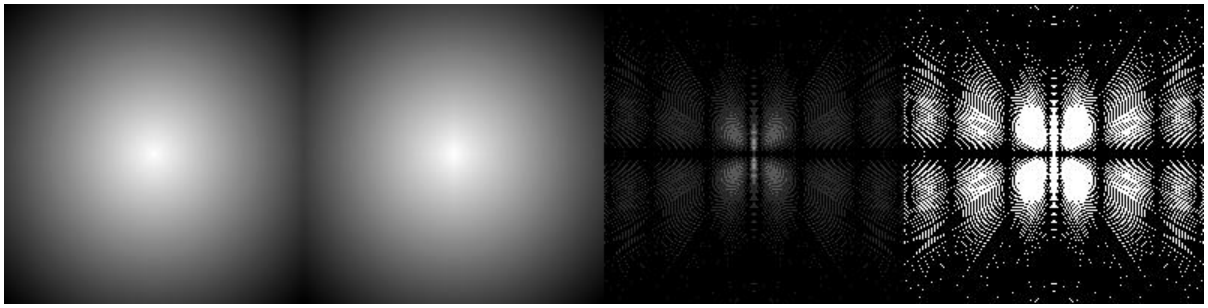


Image 4

The original gradient, final gradient, and the relative and total error versus the original image



END OF IMAGES

Section I. Polynomial Approximations:
for the scanline with fixed j in [0 to 1], with x values going from 0 to 1, we derive several polynomials approximating the distance from 0,0.

We start with degree 2 Taylor approximations

```
In[1]:= deg = 2; (* change this to generate other degree polynomials below *)

In[2]:= f[x_] := Sqrt[x^2 + j^2]

In[3]:= FullSimplify[Series[f[x], {x, 1/2, deg}]]
```

$$\text{Out[3]}= \sqrt{\frac{1}{4}+j^2}+\frac{x-\frac{1}{2}}{\sqrt{1+4j^2}}+\frac{4j^2\left(x-\frac{1}{2}\right)^2}{(1+4j^2)^{3/2}}+O\left[x-\frac{1}{2}\right]^3$$

The C Code we use for this polynomial

```
In[4]:= Sqrt[1/4 + j^2] + (x - 1/2)/Sqrt[1 + 4 j^2] + (4 j^2 (x - 1/2)^2)/(1 + 4 j^2)^(3/2) // CForm

Out[4]//CForm=
Sqrt(0.25 + Power(j,2)) + (-0.5 + x)/Sqrt(1 + 4*Power(j,2)) +
   (4*Power(j,2)*Power(-0.5 + x,2))/Power(1 + 4*Power(j,2),1.5)
```

Now make some functions to create interpolating points of two types: evenly spaced over [0,1] and as roots of Chebyshev polynomials scaled to [0,1]

```
In[5]:= evenlySpaced[n_] := Table[{x, f[x]}, {x, 0, 1, 1/n}]

In[6]:= chebyRoot[n_, k_] := (Cos[(2 k - 1) / (2 n) Pi] + 1) / 2

In[7]:= chebySpaced[n_] := Union[Table[{chebyRoot[n, k], f[chebyRoot[n, k]]}, {k, 1, n}]]
```

To see the Chebyshev interpolation points - notice they are closer to the endpoints of [0,1] than evenly spaced points

```
In[8]:= Take[Flatten[N[chebySpaced[4]]], {1, 8, 2}]

Out[8]= {0.96194, 0.691342, 0.308658, 0.0380602}
```

Now we create the evenly spaced polynomial in a format we can paste into C code. Note we do it numerically, since the computation is much faster for higher degree versions

```
In[9]:= Collect[Simplify[N[InterpolatingPolynomial[evenlySpaced[deg], x]]], x] // CForm
```

```
Out[9]//CForm=
      Sqrt(Power(j,2)) + (-3.*Sqrt(Power(j,2)) + 4.*Sqrt(0.25 + Power(j,2)) -
            1.*Sqrt(1. + Power(j,2)))*x + (2.*Sqrt(Power(j,2)) - 4.*Sqrt(0.25 + Power(j,2)) +
            2.*Sqrt(1. + Power(j,2)))*Power(x,2)
```

Now we create the Chebyshev spaced polynomial in a format we can paste into C code

```
In[10]:= Collect[Simplify[N[InterpolatingPolynomial[chebySpaced[deg + 1], x]]], x] // CForm
```

```
Out[10]//CForm=
      1.2440169358562927*Sqrt(0.00448729810778068 + Power(j,2)) -
         0.3333333333333335*Sqrt(0.25 + Power(j,2)) +
         0.08931639747704095*Sqrt(0.8705127018922193 + Power(j,2)) +
         (-2.4880338717125854*Sqrt(0.00448729810778068 + Power(j,2)) +
            2.666666666666667*Sqrt(0.25 + Power(j,2)) -
            0.1786327949540819*Sqrt(0.8705127018922193 + Power(j,2)) -
            0.5*(2.666666666666667*Sqrt(0.00448729810778068 + Power(j,2)) -
               5.333333333333335*Sqrt(0.25 + Power(j,2)) +
               2.666666666666667*Sqrt(0.8705127018922193 + Power(j,2))))*x +
         (2.666666666666667*Sqrt(0.00448729810778068 + Power(j,2)) -
            5.333333333333335*Sqrt(0.25 + Power(j,2)) +
            2.666666666666667*Sqrt(0.8705127018922193 + Power(j,2)))*Power(x,2)
```

Now we look for better interpolation points by minimizing the total squared error over the interval [0,1]. We start with quadratic polynomials. Doing cubic and higher degree polynomials are similar

```
In[11]:= g = Integrate[(a x^2 + b x + c - Sqrt[x^2 + y^2])^2, {x, 0, 1}]
```

$$\text{Out[11]}= \frac{1}{60}\left(20 + 12\,a^2 + 30\,a\,b + 20\,b^2 + 40\,a\,c + 60\,b\,c + 60\,c^2 + 60\,y^2 + 40\,b\,\left(y^2\right)^{3/2} - 30\,a\,\sqrt{1+y^2} - 40\,b\,\sqrt{1+y^2} - 60\,c\,\sqrt{1+y^2} - 15\,a\,y^2\,\sqrt{1+y^2} - 40\,b\,y^2\,\sqrt{1+y^2} + 60\,c\,y^2\,\text{Log}\left[\sqrt{y^2}\right] - 15\,a\,y^4\,\text{Log}\left[\sqrt{y^2}\right] - 60\,c\,y^2\,\text{Log}\left[1+\sqrt{1+y^2}\right] + 15\,a\,y^4\,\text{Log}\left[1+\sqrt{1+y^2}\right]\right)$$

We assume that a critical point is a minimum since the maximum is clearly unbounded (or can test more thoroughly)

```
In[12]:= r = Simplify[Solve[{D[g, a] == 0, D[g, b] == 0, D[g, c] == 0}, {a, b, c}], y ≥ 0][[1]];
```

```
In[13]:= Simplify[a x^2 + b x + c /. r]
```

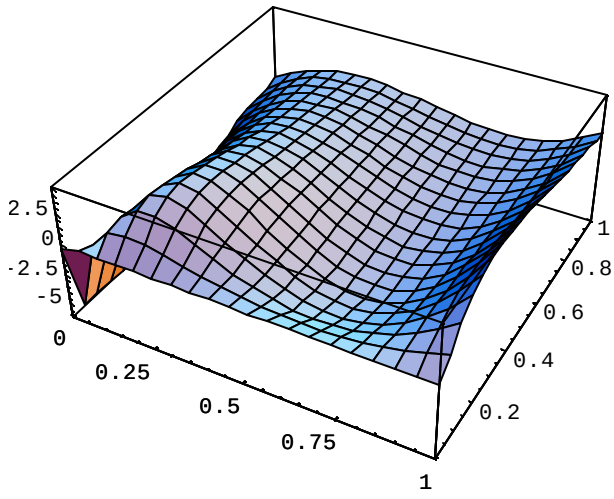
$$\text{Out[13]}= \frac{1}{4}\left(48 y^3 - 256 x y^3 + 240 x^2 y^3 + 4 x \sqrt{1+y^2} - 33 y^2 \sqrt{1+y^2} + 166 x y^2 \sqrt{1+y^2} - 150 x^2 y^2 \sqrt{1+y^2} + 3 y^2 \left(-6 + 5 y^2 + 10 x^2 \left(-2 + 3 y^2\right) - 6 x \left(-4 + 5 y^2\right)\right) \text{Log}[y] - 3 y^2 \left(-6 + 5 y^2 + 10 x^2 \left(-2 + 3 y^2\right) - 6 x \left(-4 + 5 y^2\right)\right) \text{Log}\left[1 + \sqrt{1+y^2}\right]\right)$$

The best polynomial is now

```
In[14]:= p[x_, y_] := 1/4 (48 y^3 - 256 x y^3 + 240 x^2 y^3 + 4 x Sqrt[1 + y^2] - 33 y^2 Sqrt[1 + y^2] + 166 x y^2 Sqrt[1 + y^2] -
           150 x^2 y^2 Sqrt[1 + y^2] + 3 y^2 (-6 + 5 y^2 + 10 x^2 (-2 + 3 y^2) - 6 x (-4 + 5 y^2)) Log[y] -
           3 y^2 (-6 + 5 y^2 + 10 x^2 (-2 + 3 y^2) - 6 x (-4 + 5 y^2)) Log[1 + Sqrt[1 + y^2]])
```

The error in [-256,256] over the unit square [0,1]x[0,1]

```
In[15]:= Plot3D[(p[x, y] - Sqrt[x^2 + y^2]) * 256, {x, 0, 1}, {y, 0.001, 1}, PlotPoints → {20, 20}]
```




```
Out[15]= - SurfaceGraphics -
```

Finally, the polynomial used in the C code

```
In[16]:= p[x, y] // CForm
```

```
Out[16]//CForm=
      (48*Power(y,3) - 256*x*Power(y,3) + 240*Power(x,2)*Power(y,3) + 4*x*Sqrt(1 + Power(y,2))
      -
           33*Power(y,2)*Sqrt(1 + Power(y,2)) + 166*x*Power(y,2)*Sqrt(1 + Power(y,2)) -
           150*Power(x,2)*Power(y,2)*Sqrt(1 + Power(y,2)) +
           3*Power(y,2)*(-6 + 5*Power(y,2) + 10*Power(x,2)*(-2 + 3*Power(y,2)) -
              6*x*(-4 + 5*Power(y,2)))*Log(y) -
           3*Power(y,2)*(-6 + 5*Power(y,2) + 10*Power(x,2)*(-2 + 3*Power(y,2)) -
              6*x*(-4 + 5*Power(y,2)))*Log(1 + Sqrt(1 + Power(y,2))))/4.
```

Section II. Now we derive the precomputed forward difference coefficients for the cubic: p(x)=a3 x^3 + a2 x^2 + a1 x + a0

```
In[17]:= p = a3 x^3 + a2 x^2 + a1 x + a0;
```

The difference operator takes a function and a variable and simplifies....

```
In[18]:= Delta[f_] := Collect[Simplify[(f /. x → x + δ) - (f)], x]
```

Take the first difference, leaving a second degree in x polynomial. Then define dp(x), and repeat

```
In[19]:= dp = Delta[p]
Out[19]= 3 a3 x^2 δ + x δ (2 a2 + 3 a3 δ) + δ (a1 + a2 δ + a3 δ^2)

In[20]:= ddp = Delta[dp]
Out[20]= 6 a3 x δ^2 + 2 δ^2 (a2 + 3 a3 δ)

In[21]:= dddp = Delta[ddp]
Out[21]= 6 a3 δ^3
```

Finally, a constant (not depending on x, only on coefficients and stepsize).

Sourcecode Listings

Listing 1 - Non-optimized Gradient

```
void GradientFill_1(const color_t & c1, const color_t & c2)
          { // basic circular gradient fill (assumes width and height are even!)
            // see paper for description of variables, etc
          double r,x,y,M,dc,K,cx,cy;

          // the center of the surface
          cx = (double)width/2.0;
          cy = (double)height/2.0;

          // compute max distance M from center
          M = sqrt(cx*cx+cy*cy);

          // the color delta
          dc = c2-c1;

          // and constant used in the code....
          K = dc/M;

          color_t ce; // the exact color computed for each square

          for (int j = 0; j < height; j++)
                    for (int i = 0; i < width; i++)
                              {
                              // coodinates relative to center, shifted to pixel centers
                              x = i - cx + 0.5;
                              y = j - cy + 0.5;
                              r = sqrt(x*x+y*y); // the distance
                              // the "exact" color to place at this pixel
                              ce = (color_t)(r*K+c1);
                              SetPixel(i,j,ce);
                              }
          } // GradientFill_1
```

Listing 2 - Faster Gradient

```
          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          for (int j = 0; j < height/2; j++)
                    {
                    p1 = surface + j*width;
                    p2 = p1 + width - 1;
                    p3 = surface + (height - 1 - j)*width;
                    p4 = p3 + width - 1;
                    for (int i = 0; i < width/2; i++)
                              {
                              // coodinates relative to center, shifted to pixel centers
                              x = i - cx + 0.5;
                              y = j - cy + 0.5;
                              r = sqrt(x*x+y*y);
                              ce = (color_t)(r*K+c1); // the "exact" color

                              // now draw exact colors - 4 pixels
                              *p1++ = ce;
                              *p2-- = ce;
                              *p3++ = ce;
                              *p4-- = ce;
                              }

                    }
```

Listing 3 - Testbed for Polynomials

```
void GradientFill_4(const color_t & c1, const color_t & c2)
          { // version using polynomial approximation
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square

          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          double maxDimension = max(height,width);

          // and constant used in the code....
          double t1,t2; // temp values
          t1 = width/maxDimension;
          t2 = height/maxDimension;

          K = dc/(sqrt(t1*t1+t2*t2));

          for (int j = 0; j < height/2; j++)
                    {
                    double d,alpha, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                    beta  = ((double)(height/2-1-j)+0.5)/(maxDimension/2.0);
                    p1 = surface + (height/2-j)*width+width/2+1;
                    p2 = p1 - 1;
                    p3 = surface + (height/2+j)*width+width/2+1;
                    p4 = p3 - 1;

                    p1 = surface + j*width;
                    p2 = p1 + width - 1;
                    p3 = surface + (height - 1 - j)*width;
                    p4 = p3 + width - 1;

                    for (int i = 0; i < width/2; i++)
                              {
                              // get color and update forward differencing stuff
                              alpha = ((double)(width/2-1-i)+0.5)/(maxDimension/2.0);
                              d = interpolatingPoly(beta,alpha); // call the polynomial
                              ce = (color_t)(d*K + c1);

                              // now draw exact colors - 4 pixels
                              *p1++ = ce;
                              *p2-- = ce;
                              *p3++ = ce;
                              *p4-- = ce;
                              }
                    }
          } // GradientFill_4
```

Listing 4 - Quadratic Taylor Polynomial

```
double TPoly2(double j, double x)
          { // Taylor series quadratic polynomial
          double j2,r2;
          j2 = j*j;
          r2 = sqrt(1+4*j2);
          x -= 0.5; // center it
          return r2/2 + x/r2 + 4*j2*x*x/(r2*r2*r2);
          } // TPoly2
```

Listing 5 - Cubic Chebyshev with Forward Differencing

```
void GradientFill_6(const color_t & c1, const color_t & c2)
          { // version using Chebyshev polynomial approximation,
            // and forward differencing
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square
```

```
color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

double maxDimension = max(height,width);

// and constant used in the code....
double t1,t2; // temp values
t1 = width/maxDimension;
t2 = height/maxDimension;

K = dc/(sqrt(t1*t1+t2*t2));

double delta = 1.0/(maxDimension/2.0); // stepsize

// initial pixel relative x coord
double alpha = (1.0)/maxDimension;

for (int j = 0; j < height/2; j++)
          {
          double d, beta; // pixel coords in rectangle [-1,1]x[-1,1]
          beta  = ((double)(height/2-1-j)+0.5)/(maxDimension/2.0);

          p1 = surface + j*width+width/2;
          p2 = p1 - 1;
          p3 = surface + (height - 1 - j)*width+width/2;
          p4 = p3 - 1;

          double a0,a1,a2,a3; // polynomial coefficients
          double j2,r1,r2,r3,r4; // temp values

          j2 = beta*beta;

          r1 = sqrt(0.0014485813926750633 + j2);
          r2 = sqrt(0.095269361691366900 + j2);
          r3 = sqrt(0.4779533685342265000 + j2);
          r4 = sqrt(0.9253281139039617000 + j2);

          a0 = 1.2568348730314625*r1 - 0.3741514406663722*r2 +
                    0.16704465947982383*r3 - 0.04972809184491411*r4;
          a1 = -7.196457548543286*r1 + 10.760659484982682*r2 -
                    5.10380523549030050*r3 + 1.53960329905090450*r4;
          a2 = 12.012829501508346*r1 - 25.001535905017075*r2 +
                    19.3446816555246950*r3 - 6.35597525201596500*r4;
          a3 = -6.122934917841437*r1 + 14.782072520180590*r2 -
                    14.7820725201805900*r3 + 6.12293491784143700*r4;

          // forward differencing stuff
          double d1,d2,d3;

          // initial color value and differences
          d = ((a3*alpha+a2)*alpha+a1)*alpha+a0+c1/K;
          d1 = 3*a3*alpha*alpha*delta + alpha*delta*(2*a2+3*a3*delta) + delta*(a1+a2*delta+a3*delta*
          d2 = 6*a3*alpha*delta*delta + 2*delta*delta*(a2 + 3*a3*delta);
          d3 = 6*a3*delta*delta*delta;

          d  *= K;   // we can prescale these here
          d1 *= K;
          d2 *= K;
          d3 *= K;

          for (int i = 0; i < width/2; i++)
                    {
                    // get color and update forward differencing stuff
                    ce = (color_t)(d);
                    d+=d1; d1+=d2; d2+=d3;

                    // now draw 4 pixels
                    *p1++ = ce;
                    *p2-- = ce;
                    *p3++ = ce;
                    *p4-- = ce;
                    }
          }
} // GradientFill_6
```

Listing 6 - Final Version: Cubic Chebyshev, Forward Differencing, and Fixed Point Math

```
void GradientFill_7(const color_t & c1, const color_t & c2)
          { // stuff above, with fixed point math
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square

          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          double maxDimension = max(height,width);

          // and constants used in the code....
          double t1,t2; // temp values
          t1 = width/maxDimension;
          t2 = height/maxDimension;

#define _BITS 24 // bits of fractional point stuff
#define _SCALE (1<<_BITS) // size to scale it

          K = dc/sqrt(t1*t1+t2*t2)*_SCALE;

          double delta = 2.0/maxDimension; // stepsize
          double delta2,delta3;   // powers of delta
          delta2 = delta*delta;
          delta3 = delta2*delta;

          // initial color value and differences
          double alpha = 1.0/maxDimension;

          for (int j = 0; j < height/2; j++)
                    {
                    double d, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                    beta  = ((double)(height-1-(j<<1)))/maxDimension;

                    p1 = surface + j*width+width/2;
                    p2 = p1 - 1;
                    p3 = surface + (height - 1 - j)*width+width/2;
                    p4 = p3 - 1;

                    double a0,a1,a2,a3; // polynomial coefficients
                    double j2,r1,r2,r3,r4; // temp values

                    j2 = beta*beta;

                    // numbers from the analysis to create the polynomial
                    r1 = sqrt(0.0014485813926750633 + j2);
                    r2 = sqrt(0.0952699361691366900 + j2);
                    r3 = sqrt(0.4779533685342265000 + j2);
                    r4 = sqrt(0.9253281139039617000 + j2);

                    a0 = 1.2568348730314625*r1 - 0.3741514406663722*r2 +
                              0.16704465947982383*r3 - 0.04972809184491411*r4;
                    a1 = -7.196457548543286*r1 + 10.760659484982682*r2 -
                              5.10380523549030050*r3 + 1.53960329905090450*r4;
                    a2 = 12.012829501508346*r1 - 25.001535905017075*r2 +
                              19.3446816555246950*r3 - 6.35597525201596500*r4;
                    a3 = -6.122934917841437*r1 + 14.782072520180590*r2 -
                              14.7820725201805900*r3 + 6.12293491784143700*r4;

                    // forward differencing variables
                    double d1,d2,d3;

                    // initial color value and differences
                    d = ((a3*alpha+a2)*alpha+a1)*alpha+a0+c1/K*_SCALE;
                    d1 = delta*(3*a3*alpha*alpha + alpha*(2*a2+3*a3*delta) + a2*delta + a3*delta2 + a1);
                    d2 = 2*delta2*(3*a3*(alpha + delta) + a2);
                    d3 = 6*a3*delta3;

                    // now fixed point stuff
                    int color,dc1,dc2,dc3;

                    color = (int)(d*K+0.5); // round to nearest value
                    dc1   = (int)(d1*K+0.5);
                    dc2   = (int)(d2*K+0.5);
                    dc3   = (int)(d3*K+0.5);

                    for (int i = 0; i < width/2; i++)
```

```
                                {
                                // get color and update forward differencing stuff
                                ce = (color>>_BITS);
                                color += dc1; dc1 += dc2; dc2 += dc3;

                                // now draw 4 pixels
                                *p1++ = ce;
                                *p2-- = ce;
                                *p3++ = ce;
                                *p4-- = ce;
                                }
                        }
#undef _BITS  // remove these defines
#undef _SCALE
          } // GradientFill_7
```

Total Sourcecode Listing

```
// code to derive and demonstrate a fast gradient fill algorithm
// Chris Lomont VC++ 7.0 Nov 2002

#include <windows.h> // used for timing and graphics
#include <cmath>
#include <iostream>

using namespace std;
typedef unsigned char color_t;

// constants defining the surface to fill - change to check different sizes
static const int width  = 200;
static const int height = 200;

// a global surface to fill, and a backup for comparisons
static color_t surface[width*height],backupSurface[width*height];

// gradient function pointer
typedef void (*GradientFillPtr)(const color_t & c1, const color_t & c2);

// function pointer to polynomial to test for drawing
typedef double (*PolyPtr)(double j, double x);
PolyPtr interpolatingPoly;

void SetPixel(int x, int y, const color_t & color)
          { // draw a pixel in the global surface
          surface[x+y*width] = color;
          } // SetPixel

void ComputeDifferences(bool relative, int mult,
                                                                double * totalError = NULL, double * maxError =
          { // show difference between surface and backup
            // overwrites global surface with values showing differences
            // if relative false, set surface[pos] to 255 wherever
            // there is a difference, else 0
            // if relative true, set value to abs of difference times mult
            // also, return error - total and max values, if non NULL entries

          // reset these
          if (NULL != totalError)
                    *totalError = 0;
          if (NULL != maxError)
                    *maxError = 0;

          for (int pos = 0; pos < sizeof(surface); pos++)
                    {
                    int err;
                    err = abs(surface[pos] - backupSurface[pos]);
                    if (0 != err)
                              {
                              if (false == relative)
                                        surface[pos] = 255;
                              else
                                        surface[pos] = min(255,err*mult);

                              if (NULL != totalError)
                                        *totalError += err;
                              if (NULL != maxError)
```

```
                                        {
                                        if (err > *maxError)
                                                *maxError = err;
                                        }
                                }
                        else
                                surface[pos] = 0;
                        }
                } // ComputeDifferences

        void ShowSurface(int xpos, int ypos)
                { // for debugging, show the global surface at given position
                  // draws straight to screen - not a good Win32 app :)
                HDC hdc = GetDC(NULL); // WIN32 stuff in here
                color_t c;
                for (int j = 0; j < height; j++)
                        for (int i = 0; i < width; i++)
                                {
                                c = surface[i+j*width];
                                SetPixel(hdc,xpos+i,ypos+j,RGB(c,c,c));
                                }
                ReleaseDC(NULL,hdc);
                } // ShowSurface

        void GradientFill_1(const color_t & c1, const color_t & c2)
                { // basic circular gradient fill (assumes width and height are even!)
                  // see paper for description of variables, etc
                double r,x,y,M,dc,K,cx,cy;

                // the center of the surface
                cx = (double)width/2.0;
                cy = (double)height/2.0;

                // compute max distance M from center
                M = sqrt(cx*cx+cy*cy);

                // the color delta
                dc = c2-c1;

                // and constant used in the code....
                K = dc/M;

                color_t ce; // the exact color computed for each square

                for (int j = 0; j < height; j++)
                        for (int i = 0; i < width; i++)
                                {
                                // coodinates relative to center, shifted to pixel centers
                                x = i - cx + 0.5;
                                y = j - cy + 0.5;
                                r = sqrt(x*x+y*y); // the distance
                                // the "exact" color to place at this pixel
                                ce = (color_t)(r*K+c1);
                                SetPixel(i,j,ce);
                                }
                } // GradientFill_1

        void GradientFill_2(const color_t & c1, const color_t & c2)
                { // faster version using symmetry (assumes width and height are even!)
                double r,x,y,M,dc,K,cx,cy;

                // the center
                cx = (double)width/2.0;
                cy = (double)height/2.0;

                // compute max distance M from center
                M = sqrt(cx*cx+cy*cy);

                // the color delta
                dc = c2-c1;

                // and constant used in the code....
                K = dc/M;

                color_t ce; // the exact color computed for each square

                for (int j = 0; j < height/2; j++)
                        for (int i = 0; i < width/2; i++)
                                {
```

```
                                // coodinates relative to center, shifted to pixel centers
                                x = i - cx + 0.5;
                                y = j - cy + 0.5;
                                r = sqrt(x*x+y*y);
                                ce = (color_t)(r*K+c1); // the "exact" color

                                // now draw exact colors - 4 pixels
                                SetPixel(i,j,ce);
                                SetPixel(width - 1 - i,j,ce);
                                SetPixel(i,height - 1 - j,ce);
                                SetPixel(width - 1 - i,height - 1 - j,ce);
                                }
            } // GradientFill_2

void GradientFill_3(const color_t & c1, const color_t & c2)
            { // faster version using symmetry and direct writing to memory
              // (assumes width and height are even!)
            double r,x,y,M,dc,K,cx,cy;

            // the center
            cx = (double)width/2.0;
            cy = (double)height/2.0;

            // compute max distance M from center
            M = sqrt(cx*cx+cy*cy);

            // the color delta
            dc = c2-c1;

            // and constant used in the code....
            K = dc/M;

            color_t ce; // the exact color computed for each square

            color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

            for (int j = 0; j < height/2; j++)
                        {
                        p1 = surface + j*width;
                        p2 = p1 + width - 1;
                        p3 = surface + (height - 1 - j)*width;
                        p4 = p3 + width - 1;
                        for (int i = 0; i < width/2; i++)
                                    {
                                    // coodinates relative to center, shifted to pixel centers
                                    x = i - cx + 0.5;
                                    y = j - cy + 0.5;
                                    r = sqrt(x*x+y*y);
                                    ce = (color_t)(r*K+c1); // the "exact" color

                                    // now draw exact colors - 4 pixels
                                    *p1++ = ce;
                                    *p2-- = ce;
                                    *p3++ = ce;
                                    *p4-- = ce;
                                    }
                        }
            } // GradientFill_3

/* now follow the interpolating polynomials */
double IPoly2(double j, double x)
            { // interpolating quadratic polynomial
            double j2 = j*j;
            return j + (-3.*j + 4.*sqrt(0.25 + j2) - 1.*sqrt(1. + j2))*x +
                        (2.*j - 4.*sqrt(0.25 + j2) + 2.*sqrt(1. + j2))*x*x;
            } // IPoly2

double IPoly3(double j, double x)
            { // interpolating cubic polynomial
            double j2 = j*j;
            return j + (-5.5*j + 9.*sqrt(0.1111111111111111 + j2) -
                        4.5*sqrt(0.4444444444444444 + j2) + 1.*sqrt(1. + j2))*x +
                        (9.*j - 22.5*sqrt(0.1111111111111111 + j2) +
                        18.*sqrt(0.4444444444444444 + j2) - 4.5*sqrt(1. + j2))*x*x +
                        (-4.5*j + 13.5*sqrt(0.1111111111111111 + j2) -
                        13.5*sqrt(0.4444444444444444 + j2) + 4.5*sqrt(1. + j2))*x*x*x;
            } // IPoly3

double IPoly4(double j, double x)
```

```
          { // interpolating quartic polynomial
          double j2 = j*j,r1,r2,r3,r4;
          r1 = sqrt(0.0625 + j2);
          r2 = sqrt(0.25 + j2);
          r3 = sqrt(0.5625 + j2);
          r4 = sqrt(1 + j2);

          return j + (-8.333333333333332*j + 16*r1 - 12*r2 + 5.333333333333333*r3 -
                    r4)*x + (23.333333333333332*j - 69.33333333333333*r1 + 76*r2 -
                    37.33333333333333*r3 + 7.333333333333333*r4)*x*x +
                    (-26.666666666666664*j + 96*r1 - 128*r2 + 74.66666666666667*r3 -
                    15.999999999999998*r4)*x*x*x + (10.666666666666666*j -
                    42.666666666666664*r1 + 64*r2 - 42.666666666666664*r3 +
                    10.666666666666666*r4)*x*x*x*x;
          } // IPoly4

double TPoly2(double j, double x)
          { // Taylor series quadratic polynomial
          double j2,r2;
          j2 = j*j;
          r2 = sqrt(1+4*j2);
          x -= 0.5; // center it
          return r2/2 + x/r2 + 4*j2*x*x/(r2*r2*r2);
          } // TPoly2

double TPoly3(double j, double x)
          { // Taylor series cubic polynomial
          double j2,r2;
          j2 = j*j;
          r2 = sqrt(1+4*j2);
          x -= 0.5; // center it
          return r2/2 + x/r2 + 4*j2*x*x/(r2*r2*r2) - 8*j2*x*x*x/(r2*r2*r2*r2*r2*r2);
          } // TPoly3

double TPoly4(double j, double x)
          { // Taylor series quartic polynomial
          double j2,r2;
          j2 = j*j;
          r2 = sqrt(1+4*j2);
          x -= 0.5; // center it
          return r2/2 + x/r2 + 4*j2*x*x/(r2*r2*r2) - 8*j2*x*x*x/(r2*r2*r2*r2*r2*r2) -
                    16*j2*(j2 -1)*x*x*x*x/(r2*r2*r2*r2*r2*r2*r2*r2);
          } // TPoly4

double CPoly2(double j, double x)
          { // Chebyshev point interpolating quadratic polynomial
          double j2 = j*j,r1,r2,r3;
          r1 = sqrt(0.00448729810778068 + j2);
          r2 = sqrt(0.25 + j2);
          r3 = sqrt(0.8705127018922193 + j2);
          return 1.2440169358562927*r1 - 0.3333333333333335*r2 +
                    0.08931639747704095*r3 + (-2.4880338717125854*r1 +
                    2.666666666666667*r2 - 0.1786327949540819*r3 -
                    0.5*(2.666666666666667*r1 - 5.333333333333335*r2 +
                    2.666666666666667*r3))*x + (2.666666666666667*r1 -
                    5.333333333333335*r2 + 2.666666666666667*r3)*x*x;
          } // CPoly2

double CPoly3(double j, double x)
          { // Chebyshev point interpolating cubic polynomial
          double j2 = j*j,r1,r2,r3,r4;
          r1 = sqrt(0.0014485813926750633 + j2);
          r2 = sqrt(0.09526993616913669 + j2);
          r3 = sqrt(0.4779533685342265 + j2);
          r4 = sqrt(0.9253281139039617 + j2);
          return
                    1.2568348730314622*r1 - 0.37415144066637296*r2 +
                    0.16704465947982494*r3 - 0.049728091844914335*r4 +
                    (-1.3065629648763768*r1 + 0.3889551651687712*r2 -
                    0.17365397017533368*r3 + 1.0912617698829394*r4 -
                    0.9619397662556434*(6.122934917841437*r1 -
                    10.782072520180591*r2 + 5.125218270688209*r3 -
                    0.4660806683490568*r4))*x + (6.122934917841437*r1 -
                    10.782072520180591*r2 + 5.125218270688209*r3 - 0.4660806683490568*r4 -
                    0.9619397662556434*(-6.122934917841437*r1 + 14.78207252018059*r2 -
                    14.78207252018059*r3 + 6.122934917841437*r4))*x*x +
                    (-6.122934917841437*r1 + 14.78207252018059*r2 - 14.78207252018059*r3 +
                    6.122934917841437*r4)*x*x*x;
          } // CPoly3
```

```
double CPoly4(double j, double x)
          { // Chebyshev point interpolating quartic polynomial
          double j2 = j*j,r1,r2,r3,r4,r5;
          r1 = sqrt(0.0005988661492916429 + j2);
          r2 = sqrt(0.04248024955689501 + j2);
          r3 = sqrt(0.25 + j2);
          r4 = sqrt(0.6302655018493681 + j2);
          r5 = sqrt(0.9516553824444453 + j2);
          return
                    1.2627503029350091*r1 - 0.3925221011010298*r2 +
                    0.20000000000000007*r3 - 0.10190508989888575*r4 +
                    0.031676888064907274*r5 + (-11.537170939541092*r1 +
                    17.721650625490284*r2 - 9.6*r3 + 4.966893194508029*r4 -
                    1.551372880457229*r5)*x + (33.65141886601196*r1 -
                    71.0262271348634*r2 + 60.79999999999999*r3 -
                    34.5056569091295*r4 + 11.08046517798098*r5)*x*x +
                    (-39.166991453338284*r1 + 95.01686363257255*r2 -
                    102.39999999999998*r3 + 70.66981681541661*r4 -
                    24.11968899465095*r5)*x*x*x + (15.821670111997312*r1 -
                    41.4216701119973*r2 + 51.19999999999999*r3 -
                    41.42167011199729*r4 + 15.821670111997303*r5)*x*x*x*x;
          } // CPoly4

double SPoly2(double y, double x)
          { // the super quadratic function
            // do not call with y = 0!
          double y2,y3,x2,r2;
          y2 = y*y; y3 = y2*y;
          x2 = x*x;
          r2 = sqrt(1 + y2);

          return
                    (48*y3 - 256*x*y3 + 240*x2*y3 + 4*x*r2 - 33*y2*r2 + 166*x*y2*r2 -
                    150*x2*y2*r2 + 3*y2*(-6 + 5*y2 + 10*x2*(-2 + 3*y2) -
                    6*x*(-4 + 5*y2))*log(y) - 3*y2*(-6 + 5*y2 + 10*x2*(-2 + 3*y2) -
                    6*x*(-4 + 5*y2))*log(1 + r2))/4;
          } // SPoly2

double SPoly3(double y, double x)
          { // the super cubic function
            // do not call with y = 0!
          double y2,y3,y4,y5,x2,x3,r2;
          y2 = y*y; y3 = y2*y; y4 = y3*y; y5 = y4*y;
          x2 = x*x; x3 = x*x2;
          r2 = sqrt(1 + y2);

          return
                    40.*y3 - 400.*x*y3 + 900.*x2*y3 - 560.*x3*y3 - 18.666666666666664*y5 +
                    224.*x*y5 - 560.*x2*y5 + 373.3333333333333*x3*y5 + x*r2 -
                    19.333333333333332*y2*r2 + 174.5*x*y2*r2 - 370.*x2*y2*r2 +
                    221.66666666666666*x3*y2*r2 + 18.666666666666664*y4*r2 - 224.*x*y4*r2 +
                    560.*x2*y4*r2 - 373.3333333333333*x3*y4*r2 + y2*(-8. + 30.*y2 +
                    x3*(70 - 525*y2) + x*(60. - 337.5*y2) + x2*(-120 + 810*y2))*log(y)+
                    y2*(8 - 30*y2 + x2*(120. - 810.*y2) +x*(-60. + 337.5*y2) + x3*(-70. +
                    525*y2))*log(1. + r2);
          } // SPoly3

double SPoly4(double y, double x)
          { // the super quartic function
            // do not call with y = 0!
          double y2,y3,y4,y5,x2,x3,x4,r2;
          y2 = y*y; y3 = y2*y; y4 = y3*y; y5 = y4*y;
          x2 = x*x; x3 = x*x2; x4 = x*x3;
          r2 = sqrt(1 + y2);

          return
                    (12*x*(-3200*y3 + 7168*y5 + 2*r2 + 1009*y2*r2 - 5593*y4*r2 +
                    75*y2*(4 - 63*y2 + 21*y4)*log(y) - 75*y2*(4 - 63*y2 +
                    21*y4)*log(1 + r2))+ 630*x4*y2*(160*y - 448*y3 - 44*r2 +
                    343*y2*r2 - 3*(4 - 90*y2 + 35*y4)*log(y) + 3*(4 - 90*y2 + 35*y4)*
                    log(1 + r2)) + 5*y2*(480*y - 896*y3 - 172*r2 + 707*y2*r2 -
                    3*(20 - 210*y2 + 63*y4)*log(y) + 3*(20 - 210*y2 +
                    63*y4)*log(1 + r2))- 140*x3*y2*(1536*y - 4096*y3 - 434*r2 +
                    3151*y2*r2 - 15*(8 - 168*y2 + 63*y4)*log(y) + 15*(8 - 168*y2 +
                    63*y4)*log(1 + r2))+ 210*x2*y2*(720*y - 1792*y3 - 212*r2 +
                    1387*y2*r2 - 3*(20 - 378*y2 + 135*y4)*log(y) + 3*(20 - 378*y2 +
                    135*y4)*log(1 + r2)))/24.;
          } // SPoly4
```

```
void GradientFill_4(const color_t & c1, const color_t & c2)
          { // version using polynomial approximation
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square

          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          double maxDimension = max(height,width);

          // and constant used in the code....
          double t1,t2; // temp values
          t1 = width/maxDimension;
          t2 = height/maxDimension;

          K = dc/(sqrt(t1*t1+t2*t2));

          for (int j = 0; j < height/2; j++)
                    {
                    double d,alpha, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                    beta  = ((double)(height/2-1-j)+0.5)/(maxDimension/2.0);
                    p1 = surface + (height/2-j)*width+width/2+1;
                    p2 = p1 - 1;
                    p3 = surface + (height/2+j)*width+width/2+1;
                    p4 = p3 - 1;

                    p1 = surface + j*width;
                    p2 = p1 + width - 1;
                    p3 = surface + (height - 1 - j)*width;
                    p4 = p3 + width - 1;

                    for (int i = 0; i < width/2; i++)
                              {
                              // get color and update forward differencing stuff
                              alpha = ((double)(width/2-1-i)+0.5)/(maxDimension/2.0);
                              d = interpolatingPoly(beta,alpha); // call the polynomial
                              ce = (color_t)(d*K + c1);

                              // now draw exact colors - 4 pixels
                              *p1++ = ce;
                              *p2-- = ce;
                              *p3++ = ce;
                              *p4-- = ce;
                              }
                    }
          } // GradientFill_4

void GradientFill_5(const color_t & c1, const color_t & c2)
          { // version using Chebyshev polynomial and constants precomputed
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square

          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          double maxDimension = max(height,width);

          // and constant used in the code....
          double t1,t2; // temp values
          t1 = width/maxDimension;
          t2 = height/maxDimension;

          K = dc/(sqrt(t1*t1+t2*t2));

          for (int j = 0; j < height/2; j++)
                    {
                    double d,alpha, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                    beta  = ((double)(height/2-1-j)+0.5)/(maxDimension/2.0);

                    p1 = surface + j*width;
                    p2 = p1 + width - 1;
                    p3 = surface + (height - 1 - j)*width;
```

```
                    p4 = p3 + width - 1;

                    double a0,a1,a2,a3; // polynomial coefficients
                    double j2,r1,r2,r3,r4; // temp values

                    j2 = beta*beta;

                    r1 = sqrt(0.0014485813926750633 + j2);
                    r2 = sqrt(0.09526993616913669 + j2);
                    r3 = sqrt(0.4779533685342265 + j2);
                    r4 = sqrt(0.9253281139039617 + j2);

                    a0 = 1.2568348730314625*r1 - 0.3741514406663722*r2 +
                              0.16704465947982383*r3 - 0.04972809184491411*r4;
                    a1 = -7.196457548543286*r1 + 10.760659484982682*r2 -
                              5.10380523549030050*r3 + 1.53960329905090450*r4;
                    a2 = 12.012829501508346*r1 - 25.001535905017075*r2 +
                              19.3446816555246950*r3 - 6.35597525201596500*r4;
                    a3 = -6.122934917841437*r1 + 14.782072520180590*r2 -
                              14.7820725201805900*r3 + 6.12293491784143700*r4;

                    for (int i = 0; i < width/2; i++)
                              {
                              // get color
                              alpha = ((double)(width/2-1-i)+0.5)/(maxDimension/2.0);
                              // evaluate approximating polynomial
                              d = ((a3*alpha+a2)*alpha+a1)*alpha+a0;
                              ce = (color_t)(d*K + c1);

                              // now draw exact colors - 4 pixels
                              *p1++ = ce;
                              *p2-- = ce;
                              *p3++ = ce;
                              *p4-- = ce;
                              }
                    }
          } // GradientFill_5

void GradientFill_6(const color_t & c1, const color_t & c2)
          { // version using Chebyshev polynomial approximation,
            // and forward differencing
          double K,dc;

          // the color delta
          dc = c2-c1;

          color_t ce; // the exact color computed for each square

          color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

          double maxDimension = max(height,width);

          // and constant used in the code....
          double t1,t2; // temp values
          t1 = width/maxDimension;
          t2 = height/maxDimension;

          K = dc/(sqrt(t1*t1+t2*t2));

          double delta = 1.0/(maxDimension/2.0); // stepsize

          // initial pixel relative x coord
          double alpha = (1.0)/maxDimension;

          for (int j = 0; j < height/2; j++)
                    {
                    double d, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                    beta  = ((double)(height/2-1-j)+0.5)/(maxDimension/2.0);

                    p1 = surface + j*width+width/2;
                    p2 = p1 - 1;
                    p3 = surface + (height - 1 - j)*width+width/2;
                    p4 = p3 - 1;

                    double a0,a1,a2,a3; // polynomial coefficients
                    double j2,r1,r2,r3,r4; // temp values

                    j2 = beta*beta;
```

```
                        r1 = sqrt(0.0014485813926750633 + j2);
                        r2 = sqrt(0.0952699361691366900 + j2);
                        r3 = sqrt(0.4779533685342265000 + j2);
                        r4 = sqrt(0.9253281139039617000 + j2);

                        a0 = 1.2568348730314625*r1 - 0.3741514406663722*r2 +
                                0.16704465947982383*r3 - 0.04972809184491411*r4;
                        a1 = -7.196457548543286*r1 + 10.760659484982682*r2 -
                                5.10380523549030050*r3 + 1.53960329905090450*r4;
                        a2 = 12.012829501508346*r1 - 25.001535905017075*r2 +
                                19.3446816555246950*r3 - 6.3559752520159650*r4;
                        a3 = -6.122934917841437*r1 + 14.782072520180590*r2 -
                                14.7820725201805900*r3 + 6.12293491784143700*r4;

                        // forward differencing stuff
                        double d1,d2,d3;

                        // initial color value and differences
                        d = ((a3*alpha+a2)*alpha+a1)*alpha+a0+c1/K;
                        d1 = 3*a3*alpha*alpha*delta + alpha*delta*(2*a2+3*a3*delta) + delta*(a1+a2*delta+a3*delta*
                        d2 = 6*a3*alpha*delta*delta + 2*delta*delta*(a2 + 3*a3*delta);
                        d3 = 6*a3*delta*delta*delta;

                        d  *= K;   // we can prescale these here
                        d1 *= K;
                        d2 *= K;
                        d3 *= K;

                        for (int i = 0; i < width/2; i++)
                                {
                                // get color and update forward differencing stuff
                                ce = (color_t)(d);
                                d+=d1; d1+=d2; d2+=d3;

                                // now draw 4 pixels
                                *p1++ = ce;
                                *p2-- = ce;
                                *p3++ = ce;
                                *p4-- = ce;
                                }
                        }
                } // GradientFill_6

        void GradientFill_7(const color_t & c1, const color_t & c2)
                { // stuff above, with fixed point math
                double K,dc;

                // the color delta
                dc = c2-c1;

                color_t ce; // the exact color computed for each square

                color_t * p1, *p2, *p3, *p4; // 4 quadrant pointers

                double maxDimension = max(height,width);

                // and constants used in the code....
                double t1,t2; // temp values
                t1 = width/maxDimension;
                t2 = height/maxDimension;

        #define _BITS 24 // bits of fractional point stuff
        #define _SCALE (1<<_BITS) // size to scale it

                K = dc/sqrt(t1*t1+t2*t2)*_SCALE;

                double delta = 2.0/maxDimension; // stepsize
                double delta2,delta3;   // powers of delta
                delta2 = delta*delta;
                delta3 = delta2*delta;

                // initial color value and differences
                double alpha = 1.0/maxDimension;

                for (int j = 0; j < height/2; j++)
                        {
                        double d, beta; // pixel coords in rectangle [-1,1]x[-1,1]
                        beta  = ((double)(height-1-(j<<1)))/maxDimension;
```

```
                        p1 = surface + j*width+width/2;
                        p2 = p1 - 1;
                        p3 = surface + (height - 1 - j)*width+width/2;
                        p4 = p3 - 1;

                        double a0,a1,a2,a3; // polynomial coefficients
                        double j2,r1,r2,r3,r4; // temp values

                        j2 = beta*beta;

                        // numbers from the analysis to create the polynomial
                        r1 = sqrt(0.0014485813926750633 + j2);
                        r2 = sqrt(0.0952699361691366900 + j2);
                        r3 = sqrt(0.4779533685342265000 + j2);
                        r4 = sqrt(0.9253281139039617000 + j2);

                        a0 = 1.2568348730314625*r1 - 0.3741514406663722*r2 +
                                0.16704465947982383*r3 - 0.04972809184491411*r4;
                        a1 = -7.196457548543286*r1 + 10.760659484982682*r2 -
                                5.10380523549030050*r3 + 1.53960329905090450*r4;
                        a2 = 12.012829501508346*r1 - 25.001535905017075*r2 +
                                19.3446816555246950*r3 - 6.35597525201596500*r4;
                        a3 = -6.122934917841437*r1 + 14.782072520180590*r2 -
                                14.7820725201805900*r3 + 6.12293491784143700*r4;

                        // forward differencing variables
                        double d1,d2,d3;

                        // initial color value and differences
                        d = ((a3*alpha+a2)*alpha+a1)*alpha+a0+c1/K*_SCALE;
                        d1 = delta*(3*a3*alpha*alpha + alpha*(2*a2+3*a3*delta) + a2*delta + a3*delta2 + a1);
                        d2 = 2*delta2*(3*a3*(alpha + delta) + a2);
                        d3 = 6*a3*delta3;

                        // now fixed point stuff
                        int color,dc1,dc2,dc3;

                        color = (int)(d*K+0.5); // round to nearest value
                        dc1   = (int)(d1*K+0.5);
                        dc2   = (int)(d2*K+0.5);
                        dc3   = (int)(d3*K+0.5);

                        for (int i = 0; i < width/2; i++)
                                {
                                // get color and update forward differencing stuff
                                ce = (color>>_BITS);
                                color += dc1; dc1 += dc2; dc2 += dc3;

                                // now draw 4 pixels
                                *p1++ = ce;
                                *p2-- = ce;
                                *p3++ = ce;
                                *p4-- = ce;
                                }
                        }
#undef _BITS  // remove these defines
#undef _SCALE
          } // GradientFill_7

unsigned long TimeFunction(void (func)(const color_t & c1, const color_t & c2),const color_t & c1, const color_t
          { // call gradient function the number of times, and return ms elapsed
          unsigned long startTime=0, endTime=0;
          memset(surface,0,sizeof(surface)); // clear it out
          startTime = timeGetTime(); // WIN32
          for (int pos = 0; pos < count; pos++)
                    func(c1,c2);
          endTime   = timeGetTime(); // WIN32
          return endTime - startTime;
          } // TimeFunction

void ShowPolyErrors(void)
          { // show the polynomial approximants and their error patterns
            // and output error data to cout
          int mult = 60; // error brightness
          color_t c2 = 255, c1 = 0; // gradient color
          double totalError, maxError;

          // function pointers to polynomials to test for drawing
          static const PolyPtr interpolatingPolys[] = {
```

```
                TPoly2, TPoly3, TPoly4,
                IPoly2, IPoly3, IPoly4,
                CPoly2, CPoly3, CPoly4,
                SPoly2, SPoly3, SPoly4,
                };
        static const char * polyNames[] = {
                "TPoly2", "TPoly3", "TPoly4",
                "IPoly2", "IPoly3", "IPoly4",
                "CPoly2", "CPoly3", "CPoly4",
                "SPoly2", "SPoly3", "SPoly4",
                };

        // do default method - make backup - this is our target image
        GradientFill_1(c1,c2);
        memcpy(backupSurface,surface,sizeof(surface));

        cout << "Error information\n";
        cout << "Polynomial name : total error , max error, error/pixel\n";
        double area = width*height;

        for (int pos = 0; pos < 12; pos++)
                { // show all polys
                int xpos,ypos; // drawing positions

                xpos = 2*(pos%3)*width + 10;
                ypos = (pos/3)*height  + 10;

                interpolatingPoly = interpolatingPolys[pos]; // test this one
                GradientFill_4(c1,c2);
                ShowSurface(xpos,ypos);
                // show difference between surface and backup, and get error data
                ComputeDifferences(true,mult, &totalError, &maxError);
                cout << polyNames[pos] << ": " << totalError << ",\t";
                cout << maxError << ",\t" << (totalError/area) << endl;
                ShowSurface(xpos+width,ypos);
                }
        } // ShowPolyErrors


void CompareMethods(GradientFillPtr func1, GradientFillPtr func2)
        { // compare two methods, show errors between and main image
        unsigned long timeElapsed;
        int count = 50, mult = 50;

        color_t c1 = 255, c2 = 0;

        // do default method
        timeElapsed = TimeFunction(GradientFill_1,c1,c2,count);
        cout << "Time for Gradient_1: " << timeElapsed << " ms\n";
        ShowSurface(10,10+2*height); // make backup - this is our target image
        memcpy(backupSurface,surface,sizeof(surface));

        timeElapsed = TimeFunction(func1,c1,c2,count);
        cout << "Time for function 1: " << timeElapsed << " ms\n";
        ShowSurface(10,10);
        ComputeDifferences(true,mult); // difference between surface and backup
        ShowSurface(10+width,10);

        timeElapsed = TimeFunction(func2,c1,c2,count);
        cout << "Time for function 2: " << timeElapsed << " ms\n";
        ShowSurface(10,10+height);
        ComputeDifferences(true,mult); // difference between surface and backup
        ShowSurface(10+width,10+height);

        // lastly, compare top two methods
        double totalError, maxError;
        func1(c1,c2);
        memcpy(backupSurface,surface,sizeof(surface)); // backup of func1 image
        func2(c1,c2);
        // show difference between function 2 and function 1
        ComputeDifferences(false,2*mult, &totalError, &maxError);
        ShowSurface(10+2*width,10);

        double area = width*height;
        cout << "Total, max, and per pixel error: " << totalError << ",\t";
        cout << maxError << ",\t" << (totalError/area) << endl;

        } // CompareMethods
```

```
void TimeMethods(void)
          { // time the various methods
          unsigned long timeElapsed, baseTimeElapsed;
          int count = 250, mult = 60;

          cout << "Timing: size = " << width << 'x' << height;
          cout << "  count = " << count << endl;

          static const GradientFillPtr gradientFunctions[] = {
                    GradientFill_1,
                    GradientFill_2,
                    GradientFill_3,
                    GradientFill_4,
                    GradientFill_5,
                    GradientFill_6,
                    GradientFill_7
                    };

          color_t c1 = 255, c2 = 0;

          // the base time
          baseTimeElapsed = TimeFunction(GradientFill_1,c1,c2,count);

          for (int pos = 0; pos < 7; pos++)
                    {
                    int xpos, ypos;
                    xpos = (pos % 3)*width;
                    ypos = (pos / 3)*height;
                    timeElapsed = TimeFunction(gradientFunctions[pos],c1,c2,count);
                    cout << "Time for GradientFill_" << pos+1 << ": ";
                    cout << timeElapsed << " ms, ";
                    cout << baseTimeElapsed/(double)(timeElapsed) << " fold increase\n";
                    }

          } // TimeMethods

int main(void)
          {
          // make this the default poly for functions needing it
          interpolatingPoly = CPoly3;

          // call this to show the basic gradient, and the polynomial approximations
          // also output error numbers to cout
          ShowPolyErrors();

          // call this to time the various methods
//        TimeMethods();

          // use this to show time and errors between any two methods versus baseline
//        CompareMethods(GradientFill_1,GradientFill_2); // identical output
//        CompareMethods(GradientFill_2,GradientFill_3); // identical output
//        CompareMethods(GradientFill_3,GradientFill_4); // switch to poly - errors
//        CompareMethods(GradientFill_4,GradientFill_5); // identical - same polys
//        CompareMethods(GradientFill_5,GradientFill_6); // identical - same polys + forward diff
//        CompareMethods(GradientFill_6,GradientFill_7); // switch to integers - some error
//        CompareMethods(GradientFill_7,GradientFill_1); // final comparison

          return 0;
          } // main

// end - FastGradient.cpp
```

END OF LISTINGS