Math 451: Project 2

Therin Irwin

February 23, 2012

Abstract

In a small, embedded system where small size and power efficiency are crucial, a programmer must limit the use of memory and processor power to absolutely essential tasks. The firmware we are developing for our new implantable pacemaker is no exception. Our developers have expressed interest in using the square root function, but the microchip we have chosen does not have built-in functionality for this calculation.This report describes our options for implementing $\sqrt{x}$ where $0 \leq x \leq 4$, given the restraints on processing power and memory footprint.

1 Overview

We have decided to limit our implementation of $\sqrt{x}$ to where $0 \leq x \leq 4$ to ensure the accuracy in this range while still using a maximum of 10 pre-stored points. The accuracy desired of this computation is two decimal places, therefore the error $e < 0.005$.

1.1 Techniques Discussed

During the course of this report, Newton's Method, Natural Cubic Splines, and Polynomial Interpolation will be thoroughly analyzed as feasible techniques. Several other possible techniques, Hermite Interpolation, Linear Splines, and Taylor Series will be discussed in short in Section 3. They are the least likely to fit the function within the required error bound and thus merited the least analysis.

1.2 Comparisons Applied

There is a need to compare the techniques discussed. Each will be critiqued based on how well it fits the domain of the true function (the error bound) and how the technique would perform on a very small system.

Error bound will be assessed by comparing the technique function to the correct function, $\sqrt{x}$:

$$\varphi(x) = f_T(x) - \sqrt{x}$$

Where $\varphi(x)$ is the error function at a point x and $f_T(x)$ is the function that approximates $\sqrt{x}$ using a technique T.

The discussion presented will attempt to derive a complexity for each of the techniques presented as well as an error bound on the domain. When representing complexity, big-O notation will be used.[1] This is the same notation used to describe the limiting behavior of a function; when the notation is used a remark will be made about which definition is applied.

2 Technique Analysis

This section details the analysis of each technique mentioned earlier. Each starts with a brief overview of how the technique works and how one would implement the technique.

If the reader is familiar with one of the techniques, the "overview" section of each technique can be skimmed or skipped.

2.1 Newton's Method

This technique is a feasible option because of its high degree of accuracy regardless of the data points chosen to be stored. When working with a continuous and differentiable function f, Newton's Method is effective because it converges quickly on a solution, if one is to be found. One immediately noticeable downside is that the lack of storage requires every call to the function to iteratively search for a solution.

2.1.1 Overview

This technique works by starting with an initial value q, and then calculating the intersection of the tangent line at $f(q)$ and the x-axis. This intersection will give us the value q for the next iteration of the algorithm. Iteration can be continued until a suitably accurate root value is found, or it has become obvious that there is no root available near the initial value. We can apply it to find the square root of p by finding the roots of $f(x) = x^2 - p$.

[1]More information about Big-O notation in Computer Science can be found by googling "Big-O notation in Computer Science."

2.1.2 Implementation

Since an algorithm for solving Newton's Method would need to be implemented on the microcontroller chosen for our project, it is worthwhile to show an example of pseudocode of how Newton's Method might work. This will take up significantly more memory than the functions required for implementing the other two techniques discussed.

```
// function f – we're finding the root of this.
// int x – the initial root guess.
Procedure NewtonMethod (func f, func f', int x)
Real fp
For some fixed number of iterations n
   fp = f evaluated at current guess x
   if stopping criteria are met for fp
      return best guess x
   end
   x = root of tangent line at current guess x
end
```

2.1.3 Analysis

Initial Guess Selection. In lieu of the 10 points of storage which are not necessary when using this technique, we can store the exact values for ten equally spaced points along $0 \leq x \leq 4$ for $\sqrt{x}$. This will allow us to first find a close point and then use Newton's Method to refine the guess to be accurate to 2 decimal places.

Stopping Criteria. Given the details discussed in section 2.1.2, we must select suitable stopping criteria that will limit the number of iterations. Given that Project 1 instituted the same error bound of 0.005 on its function, we can use the same stopping criteria:

$$\left|\frac{f(x_n)}{f'(x_n)}\right| \leq \varepsilon_A + cx_n \qquad (5)$$

Where $f(x_n) = x_n^2 - p$, ε_A is the absolute error, and c is the maximum error of the function. In our case, $c = 0.005$. Finally, since our maximum granularity is in steps of 0.01 and we are working with a domain of 0 to 4, the maximum number of steps that would ever be needed to reach the required accuracy is 9. This was reached by determining the maximum number of binary search steps needed to accurately identify $\frac{4}{0.01}$ items:

$$2^n = \frac{4}{0.01}$$

$$\log_2 \frac{4}{0.01} = 8.644 < 9$$

So, the maximum number of Newton's Method steps we will use is 9.

Accuracy of this Technique. The beauty of using an iterative solution such as Newton's Method is that the solution can be as accurate as the programmer desires it to be within the physical limitations of the machine. However, the accuracy comes at a price, as more accuracy requires more iterations of the algorithm.

In our case, the error must be within 0.005 when using the above stopping criteria. This is not always as accurate as some of the other solutions, but it is the only solution to provide a reasonably accurate answer across the entire domain.

Computational Complexity. Newton's Method will be the most CPU-intensive to use to approximate $\sqrt{x}$, although not by much. Closer analysis of the pseudocode presented suggests that the algorithm will approach O(log n), where$n = \frac{distance\ \ to\ closest\ \ knot}{0.005}$. This is nearly trivial, but still more complex than some of the other techniques presented.

2.2 Natural Cubic Splines

Cubic splines are one of the most accurate ways to interpolate a set of points using a piecewise-defined polynomial. The main advantage that such a technique has over an iterative technique is that the approximate answer can be obtained immediately through evaluating the polynomial at x.

2.2.1 Overview

Once a cubic spline is calculated for a set of points, it can be programmed into the microcontroller and a function value can be retrieved by simply passing in an x value. This makes evaluation of a spline very fast, but it also can be very inaccurate, especially near the end points (in our case, 0 and 4.)

Generating the cubic spline for $\sqrt{x}$ (again, this only has to be done one time) will be very tricky. The hardest part is picking a set of 10 points, or knots, with which to create a cubic spline that models the function accurately. For most non-polynomial functions (such as $\sqrt{x}$), the points cannot be chosen equally spaced, or an incredibly inaccurate spline could result. Knot-choosing will be discussed in detail in the theoretical analysis.

After a set of knots are chosen, a natural cubic spline is derived using a suitable algorithm. This will be discussed in the next section.

2.2.2 Implementation

The implementation required on the microcontroller requires simply evaluating the piecewise polynomial function, as stated previously. Savings in the code base of this technique over Newton's Method will allow sufficient memory for storage of 10 piecewise cubic polynomials. This negates the need to derive the cubic spline every time the function is called.

2.2.3 Analysis

Complexity of the Algorithm. As stated earlier, the evaluation of a cubic spline will be very fast, with each call to the approximate square root function taking only the time needed to evaluate a 3^{rd} degree polynomial (hence *cubic*). So we can say the *complexity* of evaluating this function on a microchip is O(1), or linear. In other words, it does not matter what value of x you pass to the spline; it will always take the same amount of time to compute.

Choosing the Knots. The difficulty in constructing a natural cubic spline lies in selecting the set of knots with which to build the spline.

Theorem 1. There is a cubic spline on a uniformly spaced set of points with spacing h such that

$$\max_{a \le x \le b} |f(x) - S(x)| \approx O(h^4)$$

Where S(x) is the cubic spline function.

So let's try to construct a uniformly spaced set of points, and see how many we would need in $0 \le x \le 4$ to satisfy our accuracy constraint. Assume that the error is equal to h^3, a conservative estimate. Then $h^3 = 0.005 \rightarrow h = \sqrt[3]{0.005} = 0.171$. Dividing 4 (length of the domain) by h gets us 23.48, which we round to 23. Therefore, 23 equally spaced points would be needed to reach our error bound of 0.005. Unfortunately, this would result in a piecewise function with 23 pieces necessary, which does not meet our memory footprint requirements.

Knowing that uniform knot placement would not work for our project, alternative ways to place the knots were explored. It was found through intuition and some experimentation that the best way to place the knots is to put more of them near the left edge of the graph (closer to $x = 0$). Figures 2.2.1 and 2.2.2 illustrate the spline curve generated with the uniformly spaced points and the error of that spline.

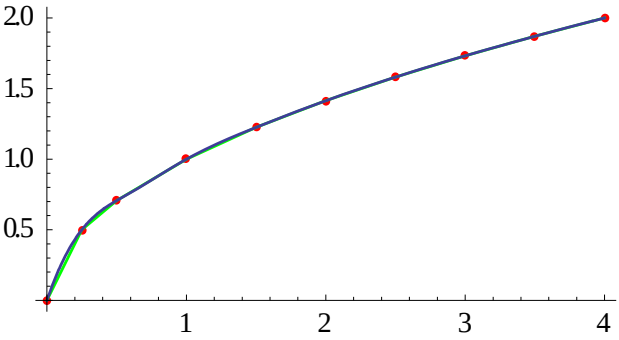

Fig. 2.2.1: Spline with uniformly spaced knots (one extra knot at 0.25).

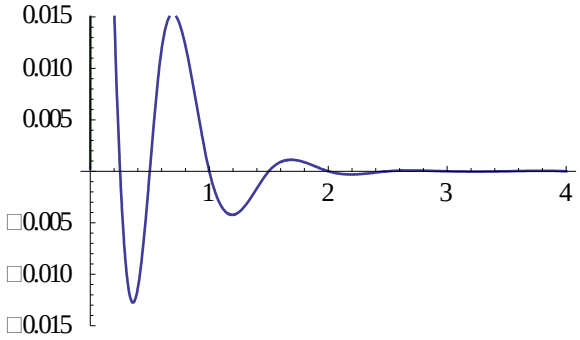

Fig 2.2.2: Plot of error of spline in Fig 2.2.1.

Note that the error is greater than the allowed error of 0.005 at most points when $x < 1$. When we compare that to Figs. 2.2.3 and 2.2.4, the difference is striking.

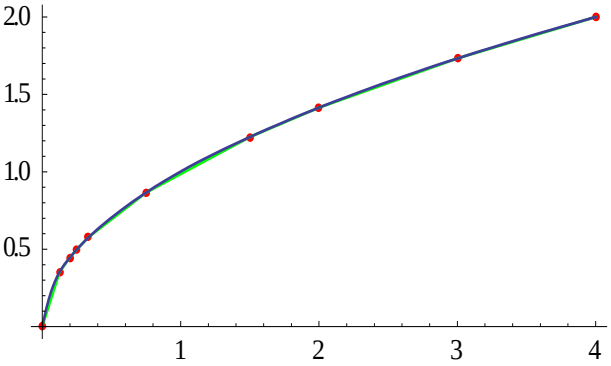

Fig 2.2.3: Spline with non-uniformly spaced points, gathered near $x = 0$.

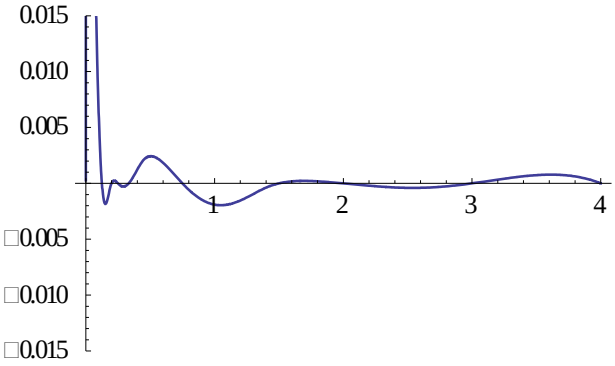

Fig 2.2.4: Plot of error of spline in Fig 2.2.3.

As you can see, the error involved when placing the points nearer to $x = 0$ is much less, making the Natural Cubic Spline a much better looking option for our application.

However, it seemed that no matter what the knot arrangement, the error near $x = 0$ was always much higher than anywhere else. We will have to decide if this is something that we can tolerate for our application before choosing this technique.

2.3 Polynomial Interpolation

Polynomial interpolation was chosen as a candidate technique because of the very low storage space required. A polynomial interpolation, depending on the degree, would require $1/10^{th}$ the storage space of a Cubic Spline with 10 sections.

2.3.1 Overview

A polynomial interpolation is a technique that simply takes any number of points n and uniquely generates a polynomial of degree $n-1$. If desired, a polynomial of smaller degree could also be generated, but then the polynomial is not guaranteed to be unique. Like a cubic spline, once we generate the function once, we will never have to do so again. We can simply store the polynomial generated.

To generate an $n-1$ degree polynomial given n points, we would use Newton's Form:

$$p(x) = a_0 x + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0) \dots (x - x_{n-1})$$

The $x_1 \dots x_n$ are taken from the set of points given and the $a_1 \dots a_n$ are calculated using the top-most elements of a Divided Difference Table. The polynomial $p(x)$ is then said to fit the set of points used to calculate it.

2.3.2 Implementation

As with the cubic spline technique, the implementation on the microcontroller would consist simply of evaluating the polynomial generated and stored in the microcontroller's memory. This polynomial would likely be stored as a table of coefficients.

2.3.3 Analysis

Complexity of the Algorithm. As with the Cubic Spline technique, the evaluation of $\sqrt{x}$ using a polynomial will be very fast, with each call to the approximate square root function taking only the time needed to evaluate ann^{th} degree polynomial. So we can say the *complexity* of evaluating this function on a microchip is O(1), or linear. In other words, it does not matter what value of x you pass to the polynomial interpolation; it will always take the same amount of time to compute.

Selection of Polynomial Degree. Experiments such as Runge's function, which get worse when a higher degree polynomial is used, suggest that a lower degree polynomial is more accurate for interpolating $\sqrt{x}$. We also know that higher-degree polynomials have a steeper curvature, something that is not desired when interpolating $\sqrt{x}$.

It is easy to show through a simple glance at the graph of $\sqrt{x}$ that a polynomial of degree 1 will not work. A degree 1 polynomial willbe a straight line, and no straight line through $\sqrt{x}$ will ever interpolate it within 0.005.

Starting with degree 3 (4 points) and going up to degree 9 (10 points), experiments were performed using Mathematica 8 to establish the feasibility of using this technique. All

polynomial degrees proved to be ineffective for our purposed because all had extremely inaccurate results except on the points provided.

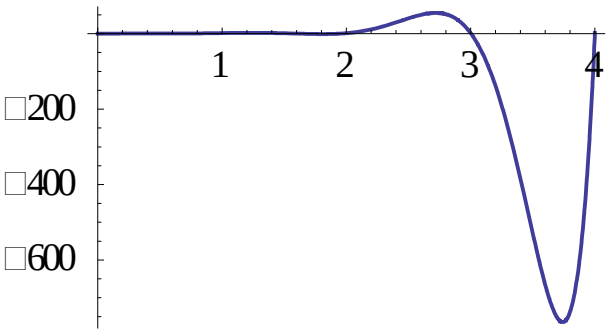

Fig. 2.3.1: Interpolating polynomial of degree 9.

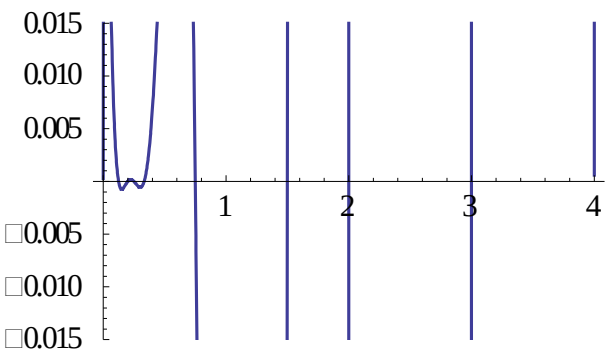

Fig. 2.3.2: Error of degree 9 interpolation.

Given these terrible accuracy results, it is extremely unlikely that Polynomial Interpolation could be a viable solution to our problem.

3 Brief Analysis of Other Methods

3.1 Hermite Interpolation

Hermite interpolation, similar to Polynomial interpolation but using derivative values as well as function values, was initially thrown out because of the lack of memory available for use. It simply did not seem logical to dedicate memory to storing derivative points for the function when only 10 points can be stored in memory.

3.2 Linear Splines

This technique is similar to the Cubic Spline technique, but each part of the piecewise function is a polynomial of degree 1. Cubic Splines are nearly always more effective at interpolating a function. Neither the 1st or 2nd derivatives are continuous when using a linear spline, while they are for a cubic spline. These derivatives provide curvature to the graph that more accurately interpolates a function.

3.3 Taylor Series

This technique uses an infinite sum to represent a function using only the function's derivatives at one point. Given that many, many iterations of the sum would be needed to get an accurate enough answer for the edge cases ($x = 0, 4$), this technique is definitely not the best choice for our application.

4 Comparison of Presented Options

4.1 Algorithm Complexity

Considering the energy efficiency requirements of the device's microchip,

In terms of computational complexity, Polynomial Interpolation and Cubic Splines are the clear winners over Newton's Method. Newton's Method requires a significant algorithm with iteration to be implemented on the microchip, while the other two options presented only require evaluation of a single polynomial each time a function value is required.

However, Newton's Method's computational complexity of O(log n) is not much worse than the other two methods, whose computational complexity is O(1) for both. This is especially so considering the very small domain required of $\sqrt{x}$ and the low degree of accuracy required.

4.2 Memory Requirement

Memory usage was also a large concern of the development team, as the device must be as small and as energy efficient as possible.

In terms of memory usage, Newton's Method would again probably use the most, simply because of the algorithm that will be implemented on the microchip. Since Polynomial Interpolation and Cubic Splines have a very simple polynomial calculation algorithm, the code for these two methods will take much less memory to store.

4.3 Accuracy Requirement

The accuracy requirement of this project is arguably the most important aspect. Given that the accuracy is already low at only 2 significant digits, it is unlikely that the device will be able to function correctly if any higher error is encountered.

Newton's Method is the only technique presented that can satisfy this requirement across the entire domain of $0 \leq x \leq 4$, while staying within the memory requirement of the device. If one or two more points can be tolerated, it is possible that the Cubic Spline technique could meet this requirement as well.

5 Conclusion

To sum up the analyses, we can say that Newton's Method is the only method that is consistently accurate to 2 decimal places along the entire interval of $0 \leq x \leq 4$. Therefore, Newton's Method is the logical choice with which to implement $\sqrt{x}$.

Additionally, if more accuracy is deemed to be required, it is a simple task to edit the stopping criteria of Newton's Method to obtain an even more accurate answer. However, this will have a negative impact on performance of the algorithm.

A properly implemented Newton's Method algorithm will ensure fast, accurate approximations to $\sqrt{x}$ for our implantable device.