JAMES HAUSER AND CARLA PURDY

Better Prototyping Through Genetics

Lookup tables and Taylor series are two common methods for interpolating between experimentally gathered data or for generating a known function such as a sine wave. This article proposes a third approach.

Embedded software developers often use a fast, floating-point processor to prototype systems that will eventually run on a fixed-point processor. To avoid an inaccurate specification, it's important to realistically simulate the fixed-point processor while generating algorithms based on the prototype. Here we describe a technique that uses a genetic algorithm (GA) to provide realistic approximations of known functions or interpolation of sampled data in a fixed-point environment.

Simple multiplication and addition aren't problematic, but current systems involve filters, trigonometry, and other complex functions. If the specification doesn't take into account the limited word size of the fixed-point processor, the developer has little choice but to rely on software floating-point libraries to meet the system requirements, which increases cost, size, and frustration. The GA-generated integer-based function approximation described here can be used during prototyping as well as directly implemented in the target fixed-point processor.

Common problems

There's no silver bullet that will resolve all specification vs. implementation issues, but you can use technology to reduce the gap.

Consider the problem of interpolating function values from a set of sampled data points for a highly nonlinear function. An interpolating polynomial of high degree can easily be generated using many commercially available tools for the purpose of prototyping. The coefficients may range from very small to very large, making normalization difficult if not impos-

The genetic algorithm takes minutes of processing time compared to more than a year for enumeration in typical cases.

sible. Alternatively, a 2-dimensional lookup procedure will not obtain the same degree of accuracy unless the table is large.

Suppose instead that we are given an equation for a highly nonlinear function. Using this equation during prototyping/modeling in conjunction with a floating-point processor would lead to accuracy expectations unachievable in a fixed point processor, unless floating point emulation was used.

Common solutions

There are two common methods for interpolating between experimentally gathered data or for generating a known function such as a sine wave. The first is the table look-up method, which is used for applications that don't require extreme accuracy.[2] To increase accuracy, this method requires increased memory.

The second method is the Taylor series. Let's look at the sine Taylor series expansion published in the TMS320C54X DSP Applications Guide.[3] It uses five terms:

$$-x\left(\frac{1-x^2}{6}\left(\frac{1-x^2}{20}\left(\frac{1-x^2}{42}\left(\frac{1-x^2}{72}\right)\right)\right)\right)$$

and computes a 16-bit result. From 0 to $\pi/2$, the error is about 1 bit when implemented in floating point; this corresponds to 0.00003 radians. A fixed-point implementation yields an error exceeding 0.2 at $\pi/2$. In other words, the accuracy of the sine function computed by this Taylor series depends heavily on the implementation.

These two methods are not the only options. A standard fixed-point function has been designed that computes an output using a set of 3rd degree polynomials.[4] The function is generic and the table of 3rd degree polynomials is specific to the problem being solved. That is, the 3rd degree polynomials might be for computing temperature from the counts read from an A/D converter or they might be for computing the sine of an angle (or for approximating another known function). Once the standard function is coded, only one copy is needed in the program (or ASIC) to solve a function evaluation or interpolation problem.

The polynomial coefficients are specific to the problem. The number of polynomials depends on the accuracy required. In itself this is not novel—just efficient.[4] What is novel is how the coefficients for the set of 3rd degree polynomials are determined.

100% compatible

The question to ask during algorithm development is not how accurate will the fixed-point implementation be, but how accurate do we need it to be. The true problem being addressed is the translation from a solution found for the set of real numbers to the set of integers. The way to solve this problem is to find the best solution limited to the set of integers.

Since most fixed-point processors and digital signal processors (DSPs) are 16-bit processors with 16-bit by 16-bit multiplication units, the integer space will be limited to +32767 and –32768. This does not limit the processor type required for implementing our standard function. For example, we have implemented the standard function in an 8-bit PIC processor that doesn't even have a multiplication unit.

The obvious way to convert from real coefficients to integer coefficients is to normalize the coefficients and to use Q-arithmetic. The numbers used in Q-arithmetic are integers with an implied decimal point. It is the responsibility of the programmer to manage decimal point alignment. In many cases, good results can be attained by normalization and scaling. The implementation will be unique for each function and won't always get the same results attained during simulation. This method is more of an art than a science. In the past, this was the only option available to the implementer.

Rather than using the coefficients obtained using the least squares method, normalizing and shifting them to fit the target architecture, and hoping that the accuracy is good enough, we propose finding the best set of 16-bit integer coefficients with the desired accuracy directly.

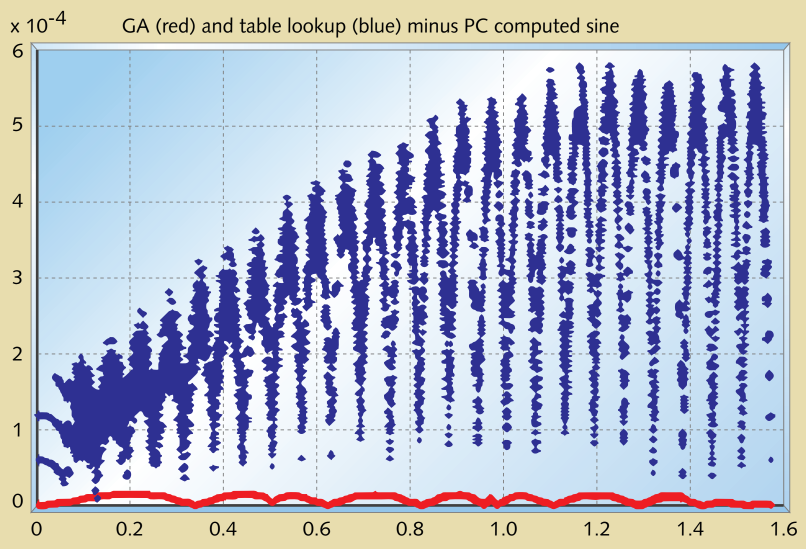
The solution space for four 16-bit coefficients contains 264 possible combinations. Typical numerical methods cannot be used to find the optimum set of integer coefficients, and enumerating the space takes about a week of processing time per candidate subspan of the data to be fit. The problem of searching an integer-valued space for a set of coefficients is integer programming—a known nondeterministic polynomial time problem.

Since there aren't any numerical methods for finding the optimal set of 3rd degree polynomials and their integer restricted coefficients, and since enumeration is time prohibitive, some heuristic is needed. One such heuristic is a *genetic algorithm* (GA). We have developed a GA that can find the set of piecewise 3rd degree polynomials with 16-bit integer valued coefficients that meet a prescribed accuracy constraint. The GA takes minutes of processing time compared to more than a year

TABLE 1 Coefficients and breakpoints for 3rd degree polynomial

Breakpoint No.	Breakpoint Value	Coefficient A	Coefficient B	Coefficient C	Coefficient D
1	0	–21272	–70	32767	0
2	8200	–16345	–16110	28762	7862
3	15900	–7974	–27662	18539	13519
4	22600	–3291	–32051	6222	16085

FIGURE 1 GA and table lookup error relative to the Intel Pentium III FPU



for enumeration in typical cases.

Genetic algorithms

For various problems, populations of possible solutions evolve according to the biological principles of natural selection and "survival of the fittest." In particular, GAs combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old population; an occasional new part is tried for good measure.

Each individual in the population is a possible solution.[5] In our case, each individual represents the four 16-bit coefficients for a single 3rd degree polynomial. The goal of the GA is to maximize the number of points spanned by each piecewise polynomial such that the error at each training point is less than some constraint and the coefficients are in the range +32767 to –32768.

The actual GA implementation is beyond the scope of this article.[1,6] Hence, we limit the discussion to results.

As previously stated, a 5-term Taylor series expansion, implemented in fixed-point arithmetic, diverges when the angle approaches π/2 radians. A 50-word lookup table for the range 0 to π with linear interpolation is superior to the Taylor series. Because of the symmetry of the sine wave, only 0 to π/2 radians will be analyzed. These results can be extended directly to the cosine function.

Since the lookup table for 0 to π/2 radians requires 26 words, the table of polynomial coefficients will be restricted to the same storage limitation. Four 3rd degree polynomials require 20 words of storage. Our goal is to do better than table lookup with less storage.

We obtain the input to the GA was by sampling the sine function over the range from 0 to π/2. The sampled input and output were both multiplied by 214. The training error constraint is 0.75. Additional accuracy can be obtained by increasing the sampling frequency. An alternative is to decrease the training error. But, this will increase the number of polynomials and storage required.

The 3rd degree polynomial evaluated by the standard function has the following form:

```
a*(input - breakpoint)^3 +
  b*(input - breakpoint)^2 +
  c*(input - breakpoint) + d
```

Notice that the lower `breakpoint` value is subtracted from the `input`. This is done to limit the size, and this result is treated as sign + 15 binary places. Thus, the coefficients can be constrained to having integer values. The precision loss resulting from cubing and squaring can be controlled by shifting, using a most-significant-bit detection scheme. Table 1 lists the coefficients and breakpoints for the four piecewise 3rd degree polynomials needed to evaluate sine (θ).

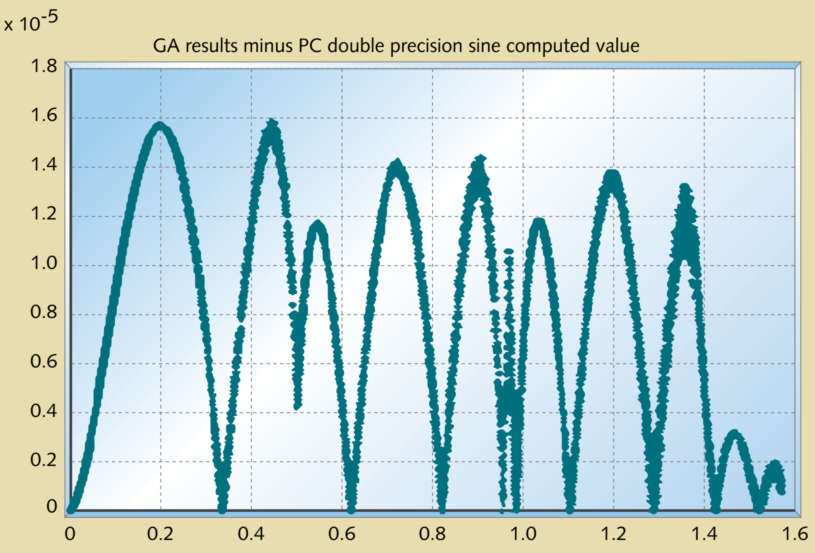
To compare the GA to the lookup table methods, we can test them both over the entire input range from 0 to 214 counts. This corresponds to a range of 0 to π/2. The maximum error between the floating-point computed sine value and the GA value is less than 1.7×10^{-5}. In general, the GA is significantly superior to the lookup table approach, as shown in Figure 1.

Figure 2 plots the GA error against the PC-computed value. For those concerned with throughput, it takes 0.000001126 seconds per sine PC-computation and it takes 0.0000018256 seconds per sine GA standard function computation on a 450MHz Pentium II.

Proper prototyping

Great products do not occur by accident; they are engineered. It is true that the specification should be implementation independent, but it is also true that the specification should not propose performance and accuracy capabilities that are not realizable by the target processor. Some of these issues can be avoided by using the same standard function evaluation module during rapid prototyping that will be used in the product software. Although it is desirable not to constrain the design by over-specifying the product, it is still necessary to show at the end of the specification phase that the proposed delivered product will indeed meet the customer's requirements and be delivered within cost. **esp**

FIGURE 2 GA error relative to the Intel Pentium III FPU



Jim Hauser is an assistant professor of computer science at Northern Kentucky University. He has a PhD from the University of Cincinnati and has worked in industry for 18 years as an embedded/DSP programmer for RCA (Cape Canaveral), Delco Electronics, and Harris Corporation. You can reach him at hauserj@nku.edu.

Carla Purdy is an associate professor of electrical and computer engineering and computer science at the University of Cincinnati. She has a PhD in mathematics from the University of Illinois and a PhD in computer science from Texas A&M University. Her email address is carla.purdy@uc.edu.

References

1. Hauser, James and Carla Purdy, "Sensor Data Processing Using Genetic Algorithms," Proc. 43rd Midwest Symposium on Circuits and Systems, August 2000.
2. Texas Instruments, "Sine, Cosine on the TMS320C2xx," Application Report Literature Number: BPRA047, 1997.
3. Texas Instruments., "TMS320C54xDSP Applications Guide," Literature Number SPRU173, 1996.
4. Conte, Samuel and Carl de Boor, *Elementary Numerical Analysis*, McGraw-Hill, 1980.
5. Goldberg, David, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
6. Hauser, James, "Approximation of Nonlinear Functions for Fixed-Point and ASIC Implementations Using a Genetic Algorithm," Ph.D. diss., University of Cincinnati, 2001.

Notes

The method proposed in this article does not depend on knowing the function to be approximated, but it can be used if the function is known. Source code and other related information may be found at www.embedded.com/code.htm.