Title: A Simple Approach to Modeling Uncertainty in C++

Author : Evan Planning

Introduction

Many classes have been introduced in the pages of this magazine and elsewhere which use overloading and other features of C++ to produce specialized mathematical classes which can be used like built-in types: more compact representation, rational numbers, complex numbers, vectors, matrices, and more. This article presents the first of a family of classes which generalize floating-point. numbers to include an estimate of the uncertainty of each number.

Motivation

As an example of a floating point application consider a spacecraft trajectory simulator. Such a program might start with an initial position and velocity for the spacecraft, and project it forward in time. For each timestep it would move the spacecraft in accordance with its current velocity and adjust the velocity to account for the gravitational attraction of significant bodies. If the program is implemented perfectly, then for a given set of initial conditions it will be able to predict perfectly where the spacecraft will be at any future moment.

But in the real world there may be some uncertainty in our knowledge of the spacecraft's initial position and velocity, as well as the positions and masses of planets. So what is really needed is a simulator that can track the effect of initial uncertainties and provide an assessment of the uncertainty in the final position. When a spacecraft is headed for a close encounter with an asteroid, "mission planners will need to know more than whether the most likely path of the spacecraft impacts the asteroid. They will need to know the probability of a collision.

If our example program were coded in C, the options for adding uncertainty appraisal to this application would be: (1) ignore it, (2) try a number of plausible sets of inputs and look at the output, or (3) rewrite the application to keep an assessment of uncertainty alongside the expected value of each uncertain value . With C++ we add option 4: use an UncertainDouble variable in place of each possibly uncertain double, and let the UncertainDouble class take care of all the bookkeeping.

Implementation

The code presented here follows the Gaussian model of uncertainty (see sidebar) and is built around two private double data members: value (also called mean) and uncertainty (also called sigma). This class is known as UDoubleMS for uncertain double mean-sigma. Listing 1 is the definition of this class, and for the most part looks like the definition of any mathematical. class. mean() and deviation() member functions give read-only access to the data members. Other differences are discussed below.

Listing 2 gives the implements.ti.ons of the constructors and destr uctors for UDoubleMS. The constructors initialize the value and uncertainty data members. The destructor does nothing.

C++'s built-in double data type can be seen as the degenerate case of UDoubleMS with uncertainty equal to zero. In fact when uncertainty is zero, members of this class do behave exactly like the built-in double type. For this reason, one constructor accepts a single argument of type double. This constructor uses the argument as the value and defaults the uncertainty to 0.0. 'I'his is the default conversion from double to UDoubleMS.

The uncertainty associated with a number tells us how many digits are significant, and so allows us to pı int that. number more intelligently than is usual with doubles.
The function uncertain_print() (Listing 3)
takes a mean and a deviation (and an optional ostream) and prints out mean +/- deviation, carefully printing only as many digits of the mean as correspond to the first two digits of the deviation.
("+/-" is read "plus or minus".)
So 1.2345 +/- 0.2387 prints as "1.23 +/- 0.24" and 0.012345 +/- 5.321 prints as "0.0 +/- 5.3". This function is used by UDoubleMS's operator<< () (not shown). l-t is implemented outside of the UDoubleMS class so that it can be used by other UDouble classes.

Operator>>() is much simpler and is similarly implemented in terms of uncertain_read() .

Propagation of Single Uncertainties

If x is 0.5 +/- 0.1, what. is f(x)? We could make a good guess by looking at. f(0.5) for the mean and then at f(0.4) and f(0.6) to estimate the deviation. Mathematically, in the Gaussian approximation we need to know the value of f() at 0.5 (f(0.5)) and the slope of f() in the neighborhood of 0.5 (f' (0.5)) (the slope or derivative is the ratio of small changes in f(x) to small change in x) .
The mean of f(x) is f() applied to the mean of x and the deviation of f(x) is the deviation of x scaled by the slope of f() at the mean of x: f(x) = f(<x> +/- dx) = f(<x>) +/- f' (<x>)dx. This formula may look daunting but, it is really quite simple to use when the slope of f() is known. And the slope is known for all functions we need to make UDoubleMS act like double: unary +, unary -, acos() , asin(), atan(), atan2(), ceil(), cos(), cosh(), exp(), fabs(), floor(), fmod(), frexp(), ldexp(), log(), log10(), modf(), POW(), sine, sinh(), sqrt(), tan(), and tanh().

Unary + and - have slopes of 1.0 and -1.0 respectively and so are easily implemented (Listing 4) . The rest of these functions are written using knowledge of slopes. For example, the slope of sino is cos(), the slope of expo is exp(), the slope of ceil() is O (except at integers, where it is infinite.) (Listing 5).

When the slope of a function is not known in advance, it can be approximated by taking the difference between f(mean + sigma) and f(mean - sigma) . Listing 6 presents the one argument version of PropagateUncertaintiesBySlope (), which does exactly this.

Multiple Sources of Uncertainty

Many operations combine two potentially uncertain inputs into one uncertain output. The simplest of these is binary "+": if a is 1.0 +/- 0.1. andb is 1.0 +/- 0.1 what is c = a + b? The value of c follows familiar rules: 1.0 + 1..0 =- 2.0. But the uncertainty of c will depend on whether or not a and b arc correlated, that is whether or not their uncertainties share a common source. As extreme cases where a and b are positively and negatively correlated we can consider the possibilities that b is a and that b is (2 - a) .

'I'he first case might arise when we have two blocks known to be identical and we measure one. We are then asking how long two of the blocks laid end-to-end are, and of course this is exactly twice the length of a single block. In this case
c = a + b = a + a = 2.0 * a = 2.0 * (1.0 +/- 0,1) = 2.0 +/-- 0.2.
Here the uncertainties of a and b have the same source and the same sign, so they simply add. You might think of them as parallel vectors. [Fig la] Using an ideal uncertain double (UDouble) class this case might be coded:

```
    UDouble a(1.0, 0.1), b;
    b = a ;
    tout << a << " +  " << b << " =  " << (a + b) << endl;
```

which prints:

```
    1.00 +/- 0.10 +  1.00 +/- 0.10 =  2.00 +/- 0.20
```

The case where b is (2 - a) is somewhat harder to imagine, but perhaps we have a box known to be 2 meters long and two blocks that together fill it perfectly. So even though our measurement of block a is imperfect we know that b is (2 - a). In this case
c = a + b = a + (2 - a) = 2 + (a - a) = 2.0 +/- 0.0
Since the uncertainties of a and b have a common source but opposite sense they subtract. You might think of them as parallel vectors pointing in opposite directions (antiparallel vectors) [Fig lb]. This case might be coded:

```
    UDouble a(1.0, 0.1), b;
    b = 2.0 - a;
    tout << a << " +  " << b << " ==  " << (a + b) << endl;
```

which prints:

```
    1.00 +/- 0.10 +  1.00 +/- 0.10 =  2.0000000 +/- 0.0
```

UDoubleMS is a class template in order to allow it to expand to two almost identical. classes. 'I'he int parameter is_correlated is conceptually a boolean, but I didn't use the new boolean type because it is not yet widely available. When is_correlated is true uncertainties add simply; when it is false uncertainties add by hypotenuse as we will. see below. The (correlated version of UDoubleMS allows
the uncertainty private data member to Lake on negative values so that anti-correlated uncertainties can cancel when added. The first case above would have the internal representations a = (1.0, 0.1), b = (1.0, 0.1) but in the second case this would be a = (1.0, 0.1), b = (1.0, -0.1). So adding the corresponding components gives the answers we derived above. Listing 7 is operator+=() and binary operator+() .

The uncorrelated case arises when the uncertainties in a and b come from independent sources. In this case the uncertainty vectors would be (on average) at right angles and their sum would
be their hypotenuse, or the square root. of the sum of their squares:
sqrt(0.1^2 + 0.1^2) = 0.1 * sqrt(2) -= 0.14 [Fig 1.c] so
c = a + b = 2,00 +/- 0.14.
This case is an application of the uncorrelated version of the UDoubleMS<is_correlated> class, UDoubleMS<0>. This case can be coded:

```
    UDouble a(1.0, 0.1), b(1.0, 0.1);
    tout_ << a << " + " << b << " = " << (a + b) << endl;
```

which prints:

```
    1.00 +/- 0.10 +  1.00 +/- 0.10 =  2.00 +/- 0.14
```

The most complicated case of adding uncertainties is when the two operands are partially correlated. If a and b are independent and c is their sum, as in the previous case, then a and c are neither perfectly correlated nor perfectly independent. a is correlated with the a portion of c but uncorrelated with the b portion of c. [Fig 1.d] The code presented here cannot trace such partial correlations, but more advanced methods can do so. Such classes may be presented in future articles here and preliminary implementations are included on the code disk(?) . One such case might be coded:

```
    UDouble a(1.0, 0.1), b(1.0, 0.1), c;
    // make "c" be half correlated with "a" and half with "b"
    // renormalized to be 1.00 +/- 0.10
    c = (a + b) / sqrt (2.0) + 1.0 - sqrt(2.0);
    tout << a << " +- " << c << " = " << (a + c) << endl;
```

which prints:

```
    1.00 +/- 0.10  +  1.00 +/-  0.10  =  2.00 +/- 0.18
```

All binary functions use slope to figure out the uncertainty from each source and then add the two uncertainties either simply (if correlated) or as hypotenuse (not correlated). The operators which work this way are +=, -=, *=, /=, and binary +, -, *, and /, and fmod(), atan2(), and pow() . Some examples are given in Listings 7 & 8.

Listing 9 shows how uncertainties are propagated by slope through an unknown function of two uncertain variables.

The only operations that are defined for type double that are not also defined for UDoubleMS are casting to other numerical types and relational operators. This is because UDoubles are conceptually multi-valued. What should

```
   UDoubleMS<0> = ud(100.0, 3.0);
   int i = ud;
```

yield? ud is 100.0 +/- 3.0 and so is likely to be near 97 or 98 or 99 or 100 or 101 or 102 or 103, but could well be anywhere from 90 to 110, and in theory might be -10,230. The most sensible single value is the mean, and if this is wanted it is available through int i = ud.mean() ; . Similarly there is no single answer to the question of whether 100.0 +/- 3.0 is greater than 101.0 +/- 6.0. It is possible to assign a probability to this value but if that probability is expressed as a simple floating-point. number between 0.0 and 1.0, then expressions like if (uda > udb) will almost always evaluate as true. If a comparison of means is wanted then the appropriate idiom is if (uda.mean() > udb.mean()) .

Implementation Issues

I have found it useful during development to include all function definitions in situ in the class template definition. While many dislike this approach because it may use more compile time and

because it clutters the class definition,
it saves a great deal of time in development. when only one file must be changed for any change of interface and it decreases the total code size.

Another thing that has proved useful during this development effort is that this package contains multiple very different implementations of the same functionality. This collection of classes now contains a UDoubleTest class that has members of the other UDouble types and distributes all operations to the members . In this way it is easy to compare the output of various methods and see at a glance where they differ.

This code uses one newish C++ feature: a template with a parameter that is not. a type. Using a new feature limited my portability enough that I decided against trying to add any other new features like exceptions and the bool type.

Demo Program

The code disk also contains a program that puts UDoubleMS through its paces and explains the results. This demo program is implemented mostly in terms of another class, UDoubleTest, which is composed of one private UDoubleMS<0> member and one UDoubleMS<1>, and distributes most operations to those classes. Listing 10 is part of class UDoubleTest, Listing 11 is part of the demo program, and Listing 12 is part of the output from the demo program.

Practical Use of UDoubleMS

An application that uses UDoubleMS must include header uncertain.h and must change all double variables that can be uncertain to UDoubleMS. Input can be handled by operator>>() if it is formatted as "mean +/- sigma"; otherwise custom input routines will be needed. Output should be formatted correctly by the overloaded operator<<() without modification.

The class UDoubleMS<1> should be used in cases with only one source of uncertainty. UDoubleMS<0> can be used where there are multiple uncertainties and each independent uncertainty gets mixed with other uncertainties exactly once. Many applications will fit neither of these sets of restrictions, and so will need the more advanced classes in this collection.

Speed Issues

Depending on the operations used in a program, changing variables from double to UDoubleMS will probably slow the program down by about a factor of three. For many applications this slowdown will not be a problem because computers hav'e become so much faster in recent years and because for many applications 1/0 or graphics take more CPU time than floating-point. calculations.

In cases where this slowdown is unacceptable, a typedef could be used for the type of all variables which may need to be UDouble. Then a compile-time definition could choose to make a double version or a UDouble version, and the UDouble version might be used only occasionally to check assumptions of uncertainty.

Speed could still be improved somewhat by adding versions of all binary operations that accept one double operand and one UDoubleMS operand. These versions could be faster than the full two-UDoubleMS versions, but would contribute to code bloat.

Other UDouble Classes

UDoubleMS offers the simplest possible model of uncertainty. The other UDouble classes on the code disk offer more accurate, but more computationally expensive, solutions to the problem of modeling uncertainties. The UDoubleMSC class uses some knowledge of the second derivative of functions (curve) to improve accuracy and (optionally) to warn when curves begin to break down the applicability of the Gaussian model. It also warns when discontinuities threaten the applicability of this model.. The correlation tracking class, UDoubleCT, uses the same underlying Gaussian model as the UDoubleMS class, but keeps track of uncertainties from multiple sources correctly. The ensemble class, UDoubleEnsemble, does not depend on the Gaussian model but instead models each uncertain variable with an ensemble of possible values.

Conclusion

I like to think of these classes as adding intelligence to an application, A carefully designed application using UDoubleMS is not only making the basic calculations that a more primitive application would make but also “thinking” about the accuracy of its results, With the UDoubleMSC class with Gaussian breakdown checking, the application could even check the accuracy of the first-order check on accuracy.

Acknowledgements

This work was partly done at the Jet Propulsion Laboratory, California Institute of Technology, under a contract from the National Aeronautics and Space Administration.

Sidebar: The Gaussian Approximation

Errors are most frequently modeled as belonging to a Gaussian, or “bell curve” distribution. in this approximation each quantity is fully characterized by only two parameters: the central value, or mean, and the deviation, or uncertainty. The mean tells what the most likely value is, while the deviation tells how far the actual value is likely to be spread around the mean. For a pure Gaussian distribution there is about a 32% chance that the value is more than one deviation away from the mean, a 4.5% chance that the value is more than 2 deviations away from the mean, and a 0.3% chance that the value is more than three deviations away from the mean.
When the deviation is zero then the distribution is 100% certain to have the value of the mean.

The probability density for a Gaussian distribution is proportional to [need formula printing here] e^-[(x-mean)^2/(2*sigma^2)]

One reason for using the Gaussian model is how well it matches many real. distributions. In fact, the Central Limit. Theorem guarantees that for any
distribution with a mean and a deviation, the sum of n

variables with this distribution will **become** more and more like a Gaussian distribution as n gets larger.

The other reason for the popularity of the Gaussian model is its computational simplicity. The sum of two variables with Gaussian distributions has a Gaussian distribution. The distribution is smooth and differentiable. It even Fourier transforms into another Gaussian distribution.

Drawbacks

But this model. is not always good enough. There are many examples of real-world distributions that are not. Gaussian. Time read from a perfectly accurate system clock has uniformly distributed error between two consecutive ticks. (eg. if the resolution is seconds, then a reading of 12:00:00 is equally likely to be 12:00:00.01, 12:00:00.50, and 12:00:00.99 but absolutely will not be 11:59:59:99 or 12:00:01 .00.)

One particularly limiting problem with the Gaussian model is that its "tails" (the edges of the distribution) are infinite. There is a small but finite chance that the value is 100 deviations away from the mean. But infinite tails cannot be made to model cases where the distribution must have a limit. I may say that a block is 1.0 +/- 0.1 inches wide. The Gaussian interpretation of this statement allows a chance that the actual block has a negative width, but we know this cannot be true.

Expanding the Gaussian model

The mean is sometimes referred to as the first moment of a distribution, and is calculated from a set of data simply by averaging the values of the data. The deviation is the second moment and can be calculated using the mean and the average of the squares of the values. Distributions which are almost Gaussian can be described more fully using a few more moments (the third moment is called skew and generally reflects the asymmetry of the distribution) . But higher moments are increasingly difficult to compute accurately, and must be avoided or used with care. Without an infinite number of moments, however, it is impossible to describe some important practical cases, such as tails with limits.

Loss of "Gaussianness"

I stated earlier that sum of Gaussian distributions is also Gaussian. Unfortunately, most operations can produce distributions that are not Gaussian even when the the operands are . In the Gaussian approximation, application of a function to an initial Gaussian variable is approximat.ed by transformation by a tangent to the true function. Figure A shows failures of this model wh'en the function is discontinuous or curved.

Author's Background:

Evan Manning has degrees in Applied Physics from Caltech and Stanford. He has been working as a self-taught C programmer in defense and space applications for the past 8 years. Currently he works for Telos Information Systems as a consultant at NASA's Jet Propulsion Laboratory. He can be reached at manning@alumni .caltech .edu.