# Testing

## 1 General Guidelines

Kernighan and Pike [5] present the following general guidelines for testing software.

- Know exactly what you are testing and what results you expect.
- Test as you go. Think about testing as you write a program. Because this is when you know best what the code should do. Test as you add parts.
- Test boundaries of variables and loop index, special cases such as empty array.
- Check pre- and post-conditions, for example, valid input.
- Use assertions for situations where a failure is unrecoverable. It is particularly useful in validating interfaces to assure consistency between caller and callee.
- Program defensively. Handle "can't happen" cases, for example, negative grades.
- Check error returns, for example, from system calls.
- Use inverse, if it exists, to verify the results. For example, encryption and decryption.
- Use two independent programs for the same problem.
- Use tools such as compiler profile to test coverage to assure every statement is tested.
- Test regressively. When fixing problems, there is tendency to check only the fix. When a problem is fixed, new problems may occur.

In addition to the above general guidelines, we suggest the following for testing numerical software.

- Test special values such as $\pm\infty$, $\pm 0$, NaN, overflow threshold, underflow threshold, if they are valid inputs.

- Use the results obtained by using higher precision as the accurate results to measure the errors in the results obtained by using lower precision.
- Random data usually does not work well in testing numerical software. Random matrices are usually well-conditioned, random numbers are seldomly near the special values.
- Use data from real applications for testing. For example, Harwell-Boeing has a collection of sparse matrices [2] and Tim Davis maintains a set of test matrices from various applications [1].
- Construct testing data so that the exact answer is known and the program is "attacked" rigorously. For example, we can construct matrices with known eigenvalues or singular values by multiplying diagonal matrix and random unitary matrices [6]. There are special matrices such as Hilbert, Cauchy, and Pascal matrices.

## 2 An Example

In the following we give an example of testing `sqrt` function.

In order to understand the testing, we first present an implementation of the function `sqrt`. The method used is the Newton's iteration for finding the zeros of a function $f(x)$. In general, Newton's iteration is

$$x_+ = x - f(x)/f'(x)$$

where $x$ is the current value and $x_+$ is the new value. In this case, $\sqrt{a}$ is a zero of the quadratic polynomial $f(x) = x^2 - a$. Thus the iteration is

$$x_+ = x - \frac{x^2 - a}{2x} = \frac{x + a/x}{2}.$$

A geometric interpretation of the above iteration is as follows. The problem of finding $\sqrt{a}$ is equivalent to finding the length of the side of a square whose area is $a$. We start with a rectangle with one side $x$ and area $a$. Then the other side is $a/x$. To make the rectangle more "square", we make a new rectangle with one side $x_+ = (x + a/x)/2$, the avarage of $x$ and $a/x$. Figure 1 depicts the iteration. We expect the constructed rectangles converge to a square with area $a$.

As any iterative method, we must address two issues:

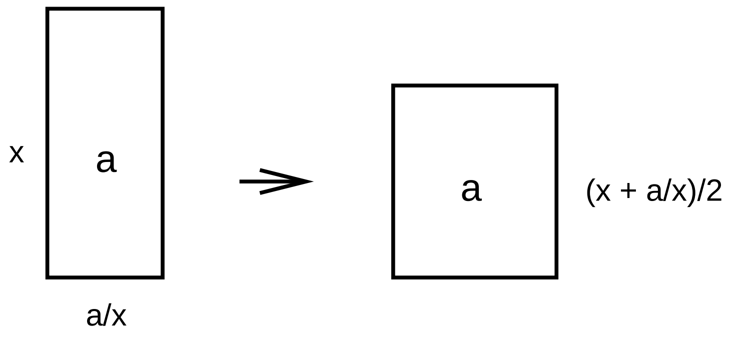

Figure 1: Geometric interpretation of the Newton's iteration for finding $\sqrt{a}$.

- The initial value;
- The stopping criterion.

Since the range of $a$ can be very wide, we first scale $a$ into the interval $[0.25, 1.0)$ by dividing or multipying by 4. Assuming $a$ has already been scaled, we use the following linear interpolation

$$x_0 = \frac{1 + 2a}{3}$$

as the initial guess. It is easy to see that the initial error

$$e_0 \equiv \sqrt{a} - x_0 = \sqrt{a} - (1 + 2a)/3$$

is maximized at $a = 9/16$ and the maximal $e_0$ is $1/24 < 2^{-4}$. For this quadratic polynomial, Newton's iteration converges quadratically, that is $e_{k+1} \leq c e_k^2$ where $c$ is a constant. Thus after three iterations, the error is smaller than $2^{-32}$, enough for single precision. And four iterations are enough for double precision.

Now we discuss testing. First we test the special values $\pm 0$, $\pm\infty$, and NaN. To test scaling, we use 1.0, $2^2$, $2^{-2}$, .... Then we test the significant part by choosing an $n$ and finding $k$ such that there are approximately $n$ consecutive integers between $2^k$ and $2^{k+1}$ ($k \approx \log_2 n$). We test whether $sqrt(x * x) = x$ exactly for these integers.

The hard part of testing numerical software is the construction of testing data. In the following, we describe a test for correctly rounded sqrt suggested by Kahan [4]. Specifically, we will show how to construct testing data to test if $sqrt(x)$ is correctly rounded, i.e., if $sqrt(x) = fl(\sqrt{x})$.

The idea is to construct an integer pair $(x, Y)$ where $Y$ is an $N$-bit integer and $x = 2^{N-j}X$ for $2^{N-1} \leq X < 2^N$ and $j = 0, 1$, so that

$$\sqrt{x} \approx Y + \frac{1}{2}.$$

Thus a small change in computation of $\sqrt{x}$ will round $sqrt(x)$ to either $Y+1$ or $Y$. In other words, the least significant digit of $sqrt(x)$ may be incorrect.

Suppose $(2Y+1)^2 = 4x + k$ for a small (relative to $(2Y+1)^2$) integer $k$. Since $x = 2^{N-j}X$, we have $(2Y+1)^2 \equiv k \bmod 2^{N+2-j}$. This implies that $2Y+1$ is a square root mod $2^{N+2-j}$ of $k$. It can be proved [4] that $k \equiv 1 \bmod 8$ (a necessary condition). Thus the possible values of $k$ are:

$$\cdots, -15, -7, 1, 9, 17, \cdots$$

Now we want to find $\sqrt{k}$. The following we describe an algorithm for generating a sequence $\{I_n\}$:

$$I_n^2 \equiv k \bmod 2^n, \qquad \text{for } n = 3, 4, 5, \cdots$$

Thus $I_{N+2-j}$ may be $2Y+1$ we are looking for. Once we get $Y$, we can find $x$.

Since $k \equiv 1 \bmod 8$ and $I_3^2 \equiv k \bmod 8$, we have $I_3 = 1$, the smallest positive square root in $(0, 2^3)$. The following algorithm generates the sequence $\{I_n\}$

```
I(3) = 1;
for n = 3, 4, 5, ...
    R(n) = (I(n)*I(n) - k)/2^n;
    if R(n) is even
         I(n+1) = I(n);
    else
         I(n+1) = 2^(n-1) - I(n):
    endif
endfor
```

To avoid the computation of $I_n^2$, square of a large integer, we observe that when $R_n$ is even

$$R_{n+1} = (I_{n+1}^2 - k)/2^{n+1} = (I_n^2 - k)/2^{n+1} = R_n/2$$

and when $R_n$ is odd

$$R_{n+1} = ((2^{n-1} - I_n)^2 - k)/2^{n+1} = 2^{n-3} + (R_n - I_n)/2.$$

The following algorithm generates sequences $\{I_n\}$ and $\{R_n\}$ without squaring large integers.

```
I(3) = 1;
R(3) = (1 - k)/2^3;
for n = 3, 4, 5, ...
    if R(n) is even
        I(n+1) = I(n);
        R(n+1) = R(n)/2;
    else
        I(n+1) = 2^(n-1) - I(n);
        R(n+1) = 2^(n-3) + (R(n) - I(n))/2;
    endif
endfor
```

For example, if $k = 1$, $I_n = 1$ and $R_n = 0$ for all $n$. If $k = 9$, then $I_3 = 1$, $R_3 = -1$ and $I_n = 3$, $R_n = 0$, for all $n > 3$. The following table lists some values (decimal) of $I_n$ and $R_n$ for $k = 17$

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|
| $I_n$ | 1 | 1 | 7 | 9 | 23 | 23 | 23 | 233 |
| $R_n$ | $-2$ | $-1$ | 1 | 1 | 4 | 2 | 1 | 53 |

Note that the sequence $I_3$, $I_4$, $I_6$, $I_{10}$ gives a $\sqrt{17}$ and $I_3$, $I_5$, $I_7$, $I_8$, $I_9$ its two's complement. From $I_{10} = 233$, we get $Y = 116$ (8 bits) and $x = 13568$ (14 bits). Note that

$$\sqrt{13568} \approx 116.48176 \approx 116 + \frac{1}{2}.$$

A small change in the computation could produce 117. This example shows that constructing testing data for numerical software can be tricky and problem specific.

## 3 Testing Stability

Stability of a numerical algorithm is an important issue. For example, we want to know how large the growth factor can be in a Gaussian elimination algorithm, how much a condition number estimator under-estimates the condition number. In general, we can apply error analysis, which requires mathematical techniques, or random testing, which requires delicate choice of testing data. In this section, describe an automated method for testing stability proposed by Higham [3, Page 477]. It uses optimization techniques to construct input data to reveal possible instability. To illustrate the idea, consider the problem of solving cubic equation:

$$x^3 + ax^2 + bx + c = 0.$$

The change of variable $x = y - a/3$ eliminates the quadratic term:

$$y^3 + py + q = 0, \qquad p = -\frac{a^2}{3} + b, \quad q = \frac{2}{27}a^3 - \frac{ab}{3} + c.$$

Then Vieta's substitution $y = w - p/(3w)$ yields

$$w^3 - \frac{p^3}{27w^3} + q = 0$$

and hence a quadratic equation in $w^3$: $(w^3)^2 + qw^3 - p^3/27 = 0$. Hence

$$w^3 = -\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}.$$

For either choice of sign, the three cube roots for $w$ yield the roots of the original cubic, on transforming back from $w$ to $y$ to $x$.

Are these formulae for solving a cubic numerically stable? We can use an optimization method to construct the coefficients $a$, $b$, and $c$ so that the relative error of the computed roots is maximized. We may use eigenvalue approach to get accurate roots. We can find the coefficients $a$, $b$, and $c$ such that the computed roots are inacurrate, while the roots are well separated (the eigenprolem is not ill-conditioned). Experiments show that the formulae, as programmed, are numerically unstable.

## References

[1] http://www.cis.ufl.edu/∼davis

[2] I.S. Duff, R. Grimes, and J. Lewis. Sparse Matrix Test Problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[3] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithm.* Society for Industrial and Applied Mathematics, 1996.

[4] W. Kahan. A Test for Correctly Rounded SQRT. http://www.cs.berkeley.edu/∼wkahan

[5] Brian W. Kernighan and Rob Pike. *The Practice of Programming.* Addison-Wesley Longman, Inc, 1999.

[6] G.W. Stewart. The effiicient generation of random orthogonal matrices with an application to condition estimators. *SIAM J. Numer. Anal.*, 17(3):403–409, 1980.