Computer System Support for Scientific and Engineering Computation

Lecture 21 - July 12, 1988 (notes revised June 14, 1990)

1 The Orbit of Pluto

Recently, Prof. Gerald Sussman of Massachusetts Institute of Technology received a lot of press (including an article in *Science*) about his calculations on the orbits of the outer planets, which suggest that the orbit of Pluto is unstable. What he does is to use Newton's equations of motion

$$\ddot{\mathbf{x}}_i = C \sum_j m_j \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_i - \mathbf{x}_j\|^3} \tag{1}$$

and solve them numerically using a formula of the form

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \text{Correction} + O(\Delta t^{14}) \tag{2}$$

If $\{\mathbf{x}_i(t)\}$ is a solution to Newton's equations, then $\{\mathbf{x}_i(-t)\}$ is also a solution. Thus given a set of initial conditions for time t_0, we can run the equations forward to time $t+t_0$ using (2), and then take the values of $\{\mathbf{x}_i(t+t_0), -\dot{\mathbf{x}}_i(t+t_0)\}$ as initial conditions for time $t+t_0$ and use (2) again to get back to time t_0. The difference between what we get and our original initial conditions will give us an estimate of the accuracy of the answer computed by (2), we hope. Sussman also used things like conservation of energy to check the accuracy of his solution. What he discovered is that for certain values of Δt the error was dramatically smaller than for other values of Δt. A variation of 10% in Δt could make a difference of a factor of 1000 in the error.

How can this be explained? The error consists of two parts, one due to the factor $O(\Delta t^{14})$, and the other due to roundoff error. For a fixed t, as Δt gets smaller, the first source of error will decrease, and the second source of error will increase (since more steps will be required). His "lucky" values of Δt have an error that is much less than the worst case error predicted by error analysis, so probably some symmetry in the calculation is causing the roundoff errors to cancel. He can only get this result when he uses VAX style rounding, that is, rounding 2.5 to 3 rather than to 2 as in the IEEE standard. Perhaps this is one of those anomalies that will never be explained.

2 Exception Handling

In the next few sections, we will examine the variation of exception handling that exists on commercial hardware.

2.1 Inmos T-800

The Inmos T-800 chip combines the Invalid Operation, Overflow, and Divide by Zero exceptions into a single bit. A situation where this can cause a problem is in evaluating continued fractions as outlined in *Presubstitution, and Continued Fractions.* In that method, divide by zero is harmless, but invalid operations can occur and are not harmless. If you are programming the T-800 to evaluate continued fractions in this way, you will have to turn on trapping in order to distinguish an invalid operation from a divide by zero, and every time you get a trap you will have to backtrack to see whether you got a harmless divide by zero, or a serious invalid operation exception.

2.2 CDC 6600, 6400, 7600, Cyber 17x

These CDC machines have numbers that *partially underflow.* Such a number `X` has the property that `X .NE. 0.0` but `1.0*X .EQ. 0.0` and `0.00123/X` causes a divide by zero. The reason is that when doing a multiply or divide, the CDC machines check for the special case of the arguments being zero. But when it does that check, it only looks at the first 12 bits, which contain the sign bit and the exponent field. Thus numbers with the smallest possible exponent will be treated as if they were 0 for multiplication and division (but not for addition, subtraction or comparison).

CDC was the first American computer to use NaNs (they call them *indefinites*) and ∞. However, the user does not get a trap when an ∞ is generated, only when it is referenced. If trapping on ∞ is turned off, then operating with ∞ can generate NaNs. There are several anomalies with these "indefinite" values. They misbehave in comparisons. And $1.0/(-\infty) = +0.0$, so $1.0/(1.0/(-\infty)) = +\infty$.

Underflows never signal, and always flush to 0.

2.3 Cray

On the Cray, `1.0*X` can overflow, even when `X` is prefectly representable. To illustrate how this can happen, consider a decimal machine with 4 significant figures and an exponent range of -99 to $+99$, where the largest representable number is $.9999 \times 10^{99}$. If you multiply 1.0 times 1×10^{98}, you will be multiplying $.1 \times 10^{1}$ times $.1 \times 10^{99}$ and will first get $.01 \times 10^{99+1}$ and then upon normalizing 0.1×10^{99}. Thus the exponenent was temporarily bigger than 99, even though the normalized result had an exponent of 99. On the Cray, if the exponent ever gets larger than the maximum allowed, it signals overflow even if the final result is representable as in the calculation above.

On the Cray, `X/Y` is computed by converting it to a multiplication by a reciprocal `X*(1/Y)`. Thus even when the exact value of `0.00123/X` is in range on the Cray, it can overflow because `1/X` overflows for the smallest values of `X`. However, since the exponent range on the Cray is quite large (15 bits of exponent compared with 11 in the IEEE standard), overflow and underflow occur much more rarely than on other machines. As on CDC machines, underflow never signals, and always flushes to 0.

2.4 IEEE 754 Implementations

The IEEE standard says that when underflow occurs and traps are enabled, the trap handler should get the exact result times an adjustment factor 2^{α} to bring the exponent back into range. The result is rounded after multiplication by the adjustment factor. For example,

consider a decimal machine similar to the previous example : if $\alpha = 144$, then 22.99×10^{-3} multiplied by $.5 \times 10^{-99}$ will underflow, so the trap handler will get $.11495 \times 10^{-101+144}$ which when rounded gives $.1150 \times 10^{43}$. If the user wants to continue computation with the correct denormalized result, then the trap handler might naively multiply $.1150 \times 10^{43}$ by 10^{-144} to get $.1150 \times 10^{-101} = .001150 \times 10^{-99}$. Rounding to 4 significant digits gives $.0012 \times 10^{-99}$, which is the wrong answer, because of double rounding. In order for the trap handler to work correctly, it needs more information than just the result scaled by 2^{α}. Knowing whether the operation was inexact plus whether there was a round up (a total of 2 bits of state) would be enough. The Intel 8087 keeps these two bits, although they are undocumented. The newest Weitek chips chop rather than round, so they only need to keep an inexact bit. However, that means you cannot do correctly rounded over/underflow counting.

In general, code that works on at least two different pre-existing machines will also run correctly on a machine with IEEE arithmetic. There is one serious caveat to this. Codes that would halt with an exception on a non-IEEE machine, will return a NaN or ∞ and keep going on an IEEE machine. It is possible that such codes might get into an infinite loop, especially if they involve iteration, because comparing a number to a NaN should return false (and give an invalid operation exception). This problem can be avoided by enabling traps. A less serious problem is that a code which checks for underflow by testing for zero will not detect an underflow because of gradual underflow.

2.5 Pyramid 98xx

The Pyramid 98xx series resembles IEEE 754, with the following notable differences.

- There is no gradual underflow. Numbers with the smallest exponent represent ± 0.
- All NaN's are signalling.
- The boolean expression $x > y$ will sometimes return true when one of the arguments is a NaN (the standard says any comparision involving a NaN should be false), although it will raise the invalid flag.* There is no condition code for unordered.
- Division does not always deliver the infinitely precise quotient rounded to working precision. Errors of 1 ulp have been observed. Furthermore, divide does not correctly set the inexact flag.
- The trap handler for over/underflows does not receive the result with the exponent adjusted by 2^{α}.
- The default action for Invalid Operation, Overflow and Divide by Zero is to abort the program (IEEE says the default should be to continue).

If an exception is generated at compile time (for example, a compile time constant of 1/0) the compiler halts.

*The problem is that the compiler will sometimes convert $x > y$ into not $x \leq y$, which makes it more difficult to get comparisions involving NaNs to work properly.

376

2.6 HP

HP machines can save the results of up to 7 floating point exceptions. This works as follows. The FPA has 16 64-bit floating point registers. The first 4 of these registers are used for exception handling, and are sub-divided into 8 32-bit registers, the first of which contains status bits, and the other seven of which are exception registers. An exception registers holds the offending instruction. Since part of the opcode contains bits specifying that the operation is a floating point operation, and this information is redundant, those bits are used to indicate what the exception was. When the trap handler wants to continue from an exception, it will emulate the instructions still in the pipeline. Because of hardware interlocks, in the sequence `C = A*B; A = E + F` the `ADD` instruction will not complete (and clobber `A`) until the hardware can be sure that the `MUL` won't trap.[†] Thus the trap handler will still have access to all the operands necessary to emulate the instructions.

Other points: there are 2 extra bits for the underflow handler so that it can correctly convert to a denormal. The compiler converts $-x$ to $0 - x$, which gives the wrong answer when x is 0, because $-(0)$ is -0, but $0 - (+0)$ is $+0$. The power function raises invalid operation on 0^0 and ∞^0[‡]

On HP machines, transcendental functions do not change the inexact flag. The reasoning is that since transcendental functions are almost always inexact, there is essentially no information conveyed by setting the inexact flag. In older releases, sine, cosine and tangent could raise the non-IEEE exception TLOSS/ERANGE, which occurs if argument reduction destroys all the precision of the argument. This doesn't happen in the current software release. They raise the invalid operation exception when given an argument of NaN or $\pm\infty$.

2.6.1 Trigonometric Functions

Raising the correct exception for trigonometric functions can be tricky. Suppose $\arccos(x)$ is implemented as $\arccos(x) = 2\arctan(\sqrt{\frac{1-x}{1+x}})$. Then evaluating `arccos(3.0)` will give an invalid operation exception, which is reasonable, although it will be for taking a negative square root, which may be misleading. On the other hand `arccos(-1.0)` will raise the divide by zero exception, which is definitely not what the user wants, although it will correctly deliver π as the result. There are two solutions to this problem. The first would be to explicitly check for the argument -1.0. The other would be to save the divide by zero flag before beginning the operation, and restore it afterword. Although this method might seem preferable because it doesn't involve a special test, it may well be slower because changing the exception flags often involves flushing the pipeline.

2.7 Sun-3 and Sun-4

The standard Unix™ signal handling mechanism is not sufficient for an IEEE trap handler. About all you can reliably do is to set a global variable and continue or else abort. It isn't possible to distinguish one floating point exception from another: they are all SIGFPE. Sun has an improved signal handling mechanism that allows you to get the address of the instruction that caused the exception, which may be different than the value of the PC at the time the handler was called. It also delivers the type of the exception. However, on

[†] In all the implementations so far, the `ADD` won't complete until the `MUL` completes.

[‡] Kahan suggests that x^0 should always return 1.

underflow or overflow you do not get a wrapped-exponent result. And on Sun-3, the trap handler may not be able to find the 68881's operands.

To implement IEEE arithmetic with Weitek 1164/5 chips, there is a system trap handler independent of any user-defined trap handler. The system trap handler is in a library on Sun-3's and in the kernel on Sun-4's. It smooths over the differences between the IEEE standard and the 1164/5 arithmetic. For example, the Weitek chip does not support gradual underflow. The system-level trap handler traps on any underflow, obtains the operands, and *recomputes* the results and exceptions, signaling a user-level trap if enabled.

Sun supports the IEEE default of not trapping on exceptions, despite a few customers' complaints.

Some Puzzles in Exception-Handling

Inmos T-800 :

Invalid Op'n, Overflow, Divide-by-Zero all share the same flag. To distinguish them, the program must back-track and examine the operands of whatever operation(s) raised the flag. Therefore, harmless Divide-by-Zero in a continued fraction cannot be passed over.

CDC 6600, 6400, 7600, Cyber 17x :

"Partial Underflow" occurs in smallest binade; these numbers test non-zero and behave non-zero in add/subtract, but behave like zero in multiply/divide. Therefore, in Fortran,

```
IF ( X .NE. 0.0 )    Q = 0.00123/X
```

must be re-written

```
IF ( 1.0*X .NE. 0.0 )    Q = 0.00123/X .
```

"Indefinites" misbehave in comparisons.

$1.0/(+\infty) = +0.0$, so $1.0/(1.0/(-\infty)) = +\infty$.

Underflow never signals (it flushes to 0.0).

Kahan 12 July 88

CRAYs

Underflow never signals (it flushes to 0.0).

1.0*X can Overflow if X is in the largest finite binade.

0.00123/X can Overflow if X is in the smallest nonzero binade.

Some IEEE 754 Implementations :

It is almost impossible to PAUSE on underflow and then resume, underflowing gradually as if the program had not stopped, because no provision is made to keep an extra bit or two that permit a double-rounding to be avoided.

A ONE-LINER :

ARCCOS(X) := 2.0 * ARCTAN (SQRT ((1.0-X)/(1.0+X)))

ARCCOS (3.0) signals "SQRT (NEGATIVE)" .

ARCCOS (-1.0) raises "DIVIDE-BY-ZERO" FLAG, but delivers π correct enough.

Better :

```
ARCCOS (X) :
        Save    DIVIDE-BY-ZERO flag ;
        A := 2.0 * ARCTAN ( SQRT ( (1.0-X)/(1.0+X) ) ) ;
        restore DIVIDE-BY-ZERO flag ;
        return  A .
```

Pyramid Floating-Point Exceptions
Wendy Thrash & Ken Drottar
July 11, 1988

Pyramid 98xx (AAU) arithmetic resembles IEEE 754, but

1) There is no gradual underflow. Any number with zero exponent represents plus or minus zero.

2) All NaNs are signaling.

3) Trichotomy is enforced in comparisons: although a NaN never compares equal to anything (including itself), it will be either < or > any given number (including itself). Any comparison of a NaN (including comparison for equality) raises the invalid flag.

   All this happens because comparisons are done via condition codes, with no code or combination representing unordered.

4) There are peculiarities in divide:

   a) Division does not always produce the correctly rounded result; errors of 1 ulp have been observed.

   b) Rounding mode can affect the accuracy of division.

   c) The inexact flag is generally meaningless following a divide.

5) Re-biasing the results of a trapped overflow or underflow is not supported.

IEEE Exception	Default Action
Invalid	Abort program
Overflow	Abort program
Division by Zero	Abort program
Underflow	Ignore (flush to zero)
Inexact	Ignore

All exceptions are indicated by flags in the program status word as they occur. Exception status accumulates to the end of the process. A trap is taken if an exception occurs while its interrupt enable bit is set in the PSW. Enabling the interrupt for an exception whose status bit is already set will not cause a trap until the next time that exception occurs.

Programs can be run with floating-point traps disabled, though this requires a bit of effort on the user's part. When programs are executed in this manner, exceptions are handled as follows:

IEEE Exception	Action/Result
Invalid	Raise invalid flag / NaN
Overflow	Raise overflow flag / +- Infinity
Division by Zero	Raise divide-by-zero flag / +- Infinity
Underflow	Raise underflow flag / +- Zero
Inexact	Raise inexact flag / +- Rounded result

These comments do not apply to Pyramid's older 90x/98x systems with FPA arithmetic or floating-point software.

```
float finvaliderr(opnd,namestr,msg_code,error_code)
float *opnd;
char *namestr;
int msg_code, error_code;
{
        int math_err(), getout();
        struct exception excpt;
        int int_opnd, on_action;
        unsigned int status;
        union {
            unsigned int i[2];
            double d;
        } temp;

        /* get integer copy of opnd */
        int_opnd = *((int *)opnd);

        /* Is opnd a Nan? */
        if (((int_opnd >> 23) & 0xff) == 0xff &&     /* exponent = Emax */
            (int_opnd & 0x007fffff) != 0) {          /* mantissa != 0 */
                /* Since *opnd is a Nan, do the convert manually. */
                temp.i[0] = (int_opnd >> 3) | 0x7ff00000;
                temp.i[1] = int_opnd << 29;
                excpt.arg1 = temp.d;

                /* if *opnd is signaling Nan, make returned value quiet */
                if (temp.i[0] & 0x00080000)
                        temp.i[0] = (temp.i[0] & 0xfff7ffff) | 0x00040000;
                excpt.retval = temp.d;
        }
        else {
                excpt.arg1 = (double) *opnd;

                /* returned value will be a double  precision quiet Nan */
                temp.i[0] = 0x7ff40000;
                temp.i[1] = 0x0;
                excpt.retval = temp.d;
        }

        /* Is invalid trap enabled? */
        if ((status = get_fpu_status()) & 0x10) {
            /* set up struct excpt */
            excpt.type = DOMAIN;                    /* error type */
            excpt.name = namestr;                   /* routine name */
            if (!matherr(&excpt)) {
                errno = EDOM;
                if (error_code) {
                    /* LIBF or LIBH */
                    (void) FTN_GETLIBTRAP(&on_action,&excpt.retval,&error_code);
                    if (on_action == on_abort) {
                        (void) math_err(msg_code);
                        (void) getout();
                    }
                    else if (on_action != on_proc && on_action != on_ignore)
                        (void) math_err(msg_code);
                }
                /* LIBM */
                else (void) math_err(msg_code);
            }
        }
        else xchg_fpu_status(status | 0x80000000);      /* set invalid flag */

        return excpt.retval;                        /* return value */
}
```

Zums 12 July 88

INEXACT EXCEPTION for TRANSCENDENTAL FUNCTIONS

Most transcendental functions return transcendental results for most of their algebraic operands. Therefore, transcendental functions are almost always inexact.

For functions of interest to us the exceptions are :

acos(0), asin(0), atan(0), atan2(0,y), cos(0), cosh(0), cosh(-∞), cosh(+∞), erf(0), erf(+∞), erfc(0), erfc(+∞), exp(0), exp(-∞), exp(+∞), log(0), log(+∞), log10(0), log10(+∞), sin(0), sinh(0), sinh(-∞), sinh(+∞), tan(0), tanh(0), tanh(-∞), tanh(+∞), loggamma(1), loggamma(2), loggamma(+∞) (and if you don't mind the odd order poles : loggamma(0) & loggamma(negative integers)), Jn(0), Jn(+∞), Yn(0), Yn(+∞) and power(for many operands)

COSINE, SINE, TANGENT

INVALID: NaNs, $\pm\infty$. In these cases, trap on DOMAIN/EDOM if enabled, otherwise return quiet(x). If the range reduction used has an upper bound beyond which there is a total loss of precision, any $|x| >$ some max. In this case, trap on TLOSS/ERANGE if enabled otherwise return quiet(x).

OVERFLOW: None.

UNDERFLOW: None known to exist.

SPECIALS: None.

EXPONENTIAL

INVALID: NaNs only. Trap on DOMAIN/EDOM if enabled, otherwise return quiet(x).

OVERFLOW: If $x > 88.02968667$ (single) or $x > 709.0895658$ (double). Trap on OVERFLOW/ERANGE if enabled, otherwise return $+\infty$.

UNDERFLOW: If $x < -87.33654476$ (single) or $x < -708.3964186$ (double). Trap on UNDERFLOW/ERANGE if enabled, otherwise return denorm result.

SPECIALS: $\exp(-\infty) = +0$, $\exp(+\infty) = +\infty$.

POWER

INVALID: NaNs, power(0,0), power(x,y) for negative x and non-integer y, power(∞,0). Trap on DOMAIN/EDOM if enabled, otherwise return quiet(x) or quiet(y).

OVERFLOW: Trap on OVERFLOW/ERANGE if enabled, otherwise return ∞.

UNDERFLOW: Trap on UNDERFLOW/ERANGE if enabled, otherwise return denorm result.

SPECIALS: power(0,y) = 0 for $y > 0$, power(0,y) = ∞ for $y < 0$, power(1,y) = 1 for non-NaN y, power(x,0) = 1 for $x > 0$, power(+∞,y) = +∞ for $y > 0$, power(+∞,y) = 0 for $y < 0$, power(x,+∞) = +∞ for $x > 1$, power(x,-∞) = 0 for $x > 1$, power(x,+∞) = 0 for $0 < x < 1$, power(x,-∞) = +∞ for $0 < x < 1$.

IEEE Exception Handling

SunOS 4.0 and Fortran 1.1

Sun-3 -f68881

Sun-3 -ffpa

Sun-4

David Hough
11 July 1988

Accrued-Exception Bits

Simple operations according to IEEE 754
nonstop not always popular

Binary-decimal conversion
decimal_to_floating(3)
floating_to_decimal(3)

Transcendental functions
68881 atomic - read manual
some FPA
software in Sun-4

ieee_flags(3m)

```
char *out; int k, ieee_flags( );
ieee_flags ("clear","exception","all",&out);
   /* clear all accrued exceptions */

...
... (code that generates three exceptions:
   overflow, invalid, inexact)
...

k = ieee_flags("get","exception","overflow",&out);

/* then out = ''overflow'', and on a Sun-3, k=25. */
```

How do you get IEEE out of Weitek 1164/5 ?

Recomputation

- **user mode in FPA via SIGFPE**
- **inexact: first exception and trapping**
- **Sun-4 in kernel**

Performance

- **underflow & nonstandard_arithmetic()**
- **NaNs**

IEEE Trapping

What can you expect from a standard Unix signal handler?

Bad vibes: SVID, POSIX, X3J11, X/Open, ...
set a global variable
abort

setjmp/longjmp?

no pre-substitution
no continuation with new result

SunOS SIGFPE

Does...
- **signal, code, context, instruction address**
- **confusion with other types of exceptions**
- **FPA recomputation works**

Doesn't...
- **no exceptional operand**
- **no 68020 effective address**
- **no exponent wrap**

Little used, little debugged
- **setting f-regs doesn't work on Sun-4**

```
void sample_handler( sig, code, scp, addr)
int sig ;       /* sig == SIGFPE always */
int code ;
struct sigcontext *scp ;
char *addr ;
{
      /*
        Sample user-written sigfpe code handler.
        Prints a message and continues.
        struct sigcontext is defined in <signal.h>.
       */
      printf("ieee exception code %x occurred at pc %X \n",
           code,scp->sc_pc);
}

extern void sample_handler();

main()
{
#include <floatingpoint.h>

      sigfpe_handler_type hdl;

/*
* set new overflow handler to sample_handler() and set
* new invalid handler to SIGFPE_ABORT
*/
      hdl = (sigfpe_handler_type) sample_handler;
      if(ieee_handler("set","overflow",hdl) != 0)
           printf("ieee_handler can't set overflow \n");
      if(ieee_handler("set","invalid",SIGFPE_ABORT) != 0)
           printf("ieee_handler can't set invalid \n");
      ...
```