Numerical Tools for Physical Scientists

Ramses Van Zon, Jonathan Dursi
SciNet, Jan/Feb 2012

The Course

- We've covered some basics of programming for scientists in the last class:
  - C/C++
  - git for version control
  - unit tests
- Here we're going to focus on specifics of numerical computing for physical scientists.

Course Outline

- Today: Numerics, Random Numbers
- Jan 20: Integration, ODE solvers
- Jan 27: Numerical Linear Algebra - BLAS, LAPACK
- Feb 3: Fast Fourier Transforms - FFTW

Today

- What numerical computing is, and how to think about it
- Modelling vs reality; Validation & Verification
- Real arithmetic on computers - floating point math
- Random Number Generators

Computational Science

- “Third Leg” of Science?
- Different than theoretical science or experimental science; requires skills of each
- “Experimental theory” - exploring complex regions of theory
- Requires note-taking, methodical approach of experimentalists; mathematical chops of theorists; and other knowledge too.

Computational Science

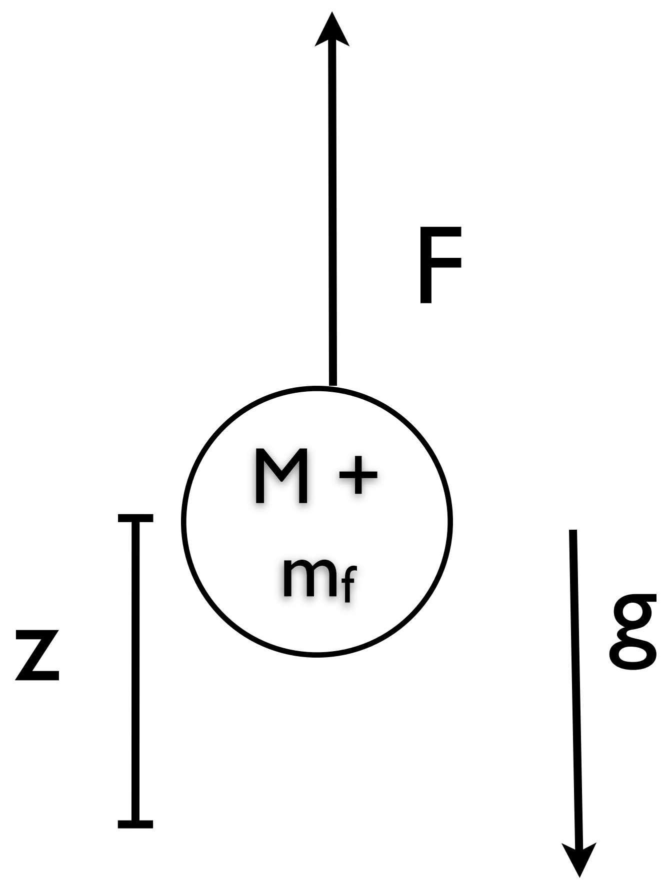
- Often done incredibly badly
- If experimentalists work was of quality of much computational work, we still would be arguing over the charge on an electron
- Experimentalists, theoreticians have had centuries to determine best practices for their disciplines
- Computationalists starting to develop ours - eg this course.

Computational Science

- Computational science, like experimental or theoretical science, is a *modelling* endevor
- Creating simplified picture of reality that includes (only) bits you want to study.

Phenomenon

Conceptual/ Mathematical Model



$$\ddot{z} = m(t)(F - g)$$

$$\dot{m_f} \propto F$$

Numerical Computation

```
double dfdt(double time, double mass, double fuelmass,
           double vel, double force, double z, double dt,
           double *df) {

        double g = Mearth*newtonG/(z*z);

        df[0] = (mass+fuelmass)*(alpha*burnrate-g);
        df[1] = vel;
        df[2] = burnrate;

}
~
~
                                             1,1           All
```

SciNet compute • calcul CANADA

Phenomenon

model

Conceptual/
Mathematical Model

model

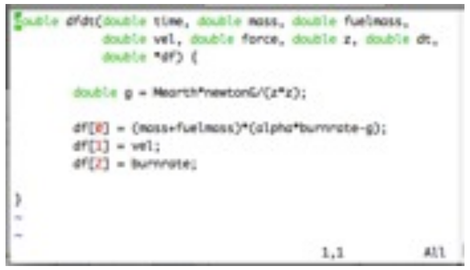
Numerical
Computation

Can go wrong at each step

Are we solving the right equations?

$$\ddot{z} = m(t)(F - g)$$

$$\dot{m}_f \propto F$$



```
double dfdt(double time, double mass, double fuelmass,
        double vel, double force, double z, double dt,
        double *df) {

    double g = Mearth*newtonG/(z*z);

    df[0] = (mass+fuelmass)*(alpha*burnrate-g);
    df[1] = vel;
    df[2] = burnrate;

}
~
~
                                        1,1           All
```

Can go wrong at each step

$$\ddot{z} = m(t)(F - g)$$

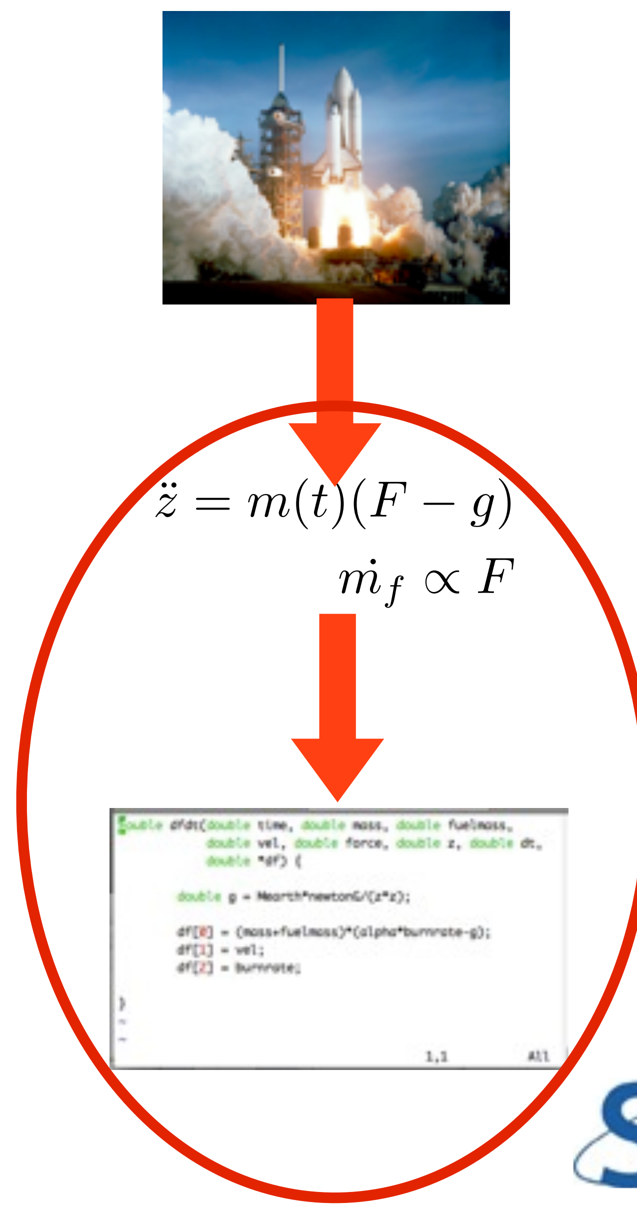
$$\dot{m}_f \propto F$$

```
double dfdt(double time, double mass, double fuelmass,
        double vel, double force, double z, double dt,
        double *df) {

    double g = Mearth*newtonG/(z*z);

    df[0] = (mass+fuelmass)*(alpha*burnrate-g);
    df[1] = vel;
    df[2] = burnrate;

}
```

Are we solving the equations right?

Verification: Testing math ⇒ numerics

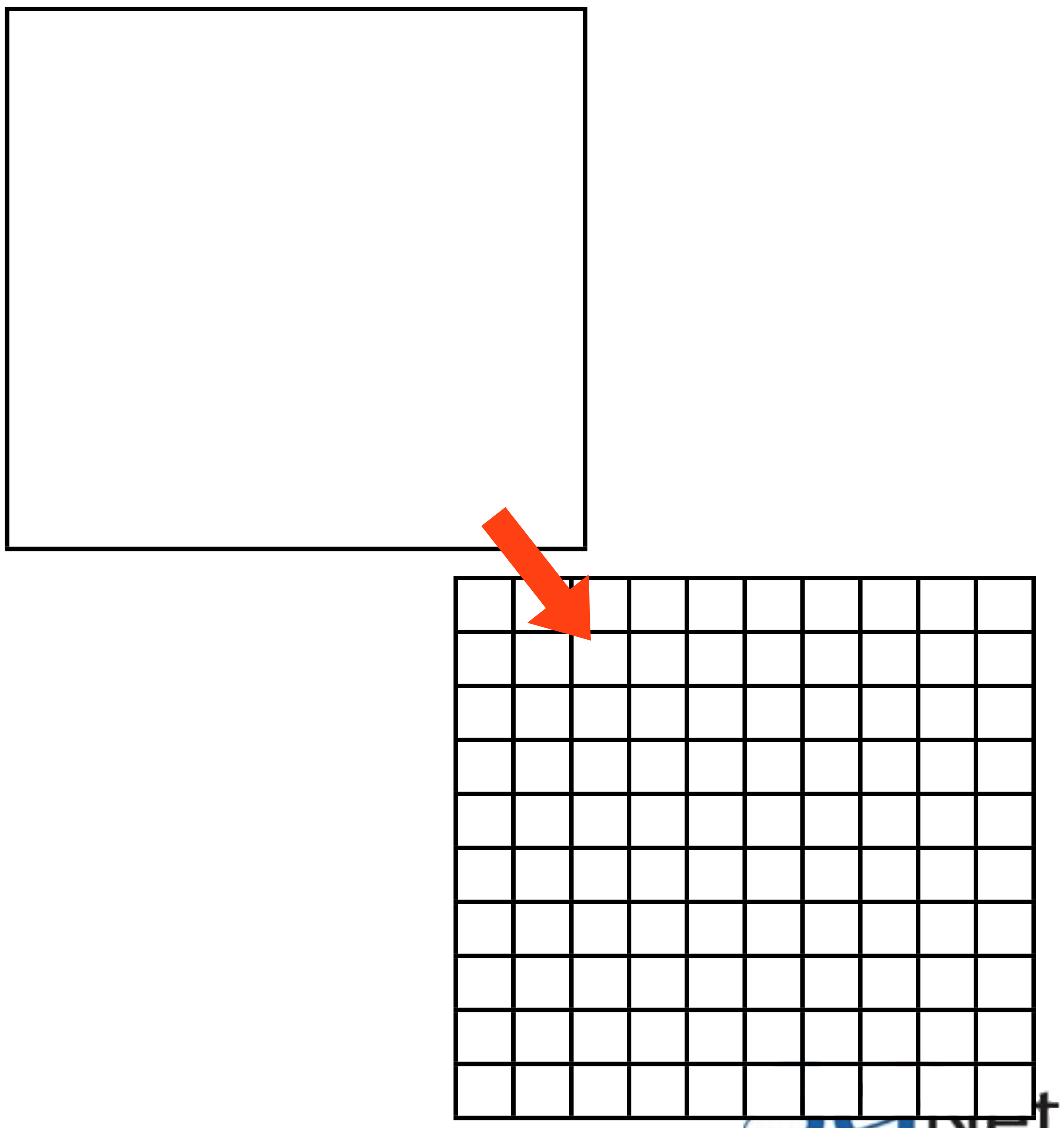
- Can go wrong in translation from mathematical model to computational model
- Discretization error, Truncation error, roundoff, ... or just plain bug.
- Process of testing this: Verification



SciNet compute • calcul CANADA

Discretization error

- Error caused by going from continuum to discrete domain
- Eg: grid in space; discrete timesteps; etc.
- Should decrease as you increase resolution.



Truncation Error

- Typically occurs when an expansion is truncated

$$e^x \approx 1 + x + \frac{x^2}{2}$$

Roundoff

- Floating point mathematics can go wrong (more on this later)

$$(a+b)+c \neq a+(b+c)$$

Just plain bugs

- Scientific software can get large, complex
- Bugs creep in
- Unit testing, version control can greatly help
- Still happens

Verification: Analytics, Bechmarks, Comparisons

- Trying to make sure we are correctly solving the intended equations in the regime of interest.
- Comparison to known analytic solutions:
  - Easy to do
  - Solutions tend to be of very simple situations - not hard tests of the computation, particularly integrated.
  - But very useful for unit tests.

Verification: Analytics, Bechmarks, Convergence

- Trying to make sure we are correctly solving the intended equations in the regime of interest.
- Benchmarking a complex solution from your code to that of another code (could be: same code last year, saved results)
    - CAN NOT show that either solution is *correct*
    - CAN show that at least one code/version has a problem, or that something has caused changes.

Verification: Analytics, Bechmarks, Convergence

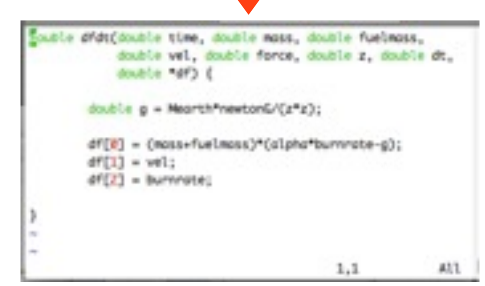
- Trying to make sure we are correctly solving the intended equations in the regime of interest.
- Convergence testing: compare solutions at higher and higher resolution (or terms in expansion, or...)
- Again, doesn't mean converges to correct result, but lack of convergence indicates a problem
- Relatedly - does slightly varying input parameters, result in robust result, or do huge changes occur in relevant metrics?

Validation: Testing reality ⇒ numerics

- Can go wrong in translation from phenomenon to mathematical model

$$\ddot{z} = m(t)(F - g)$$

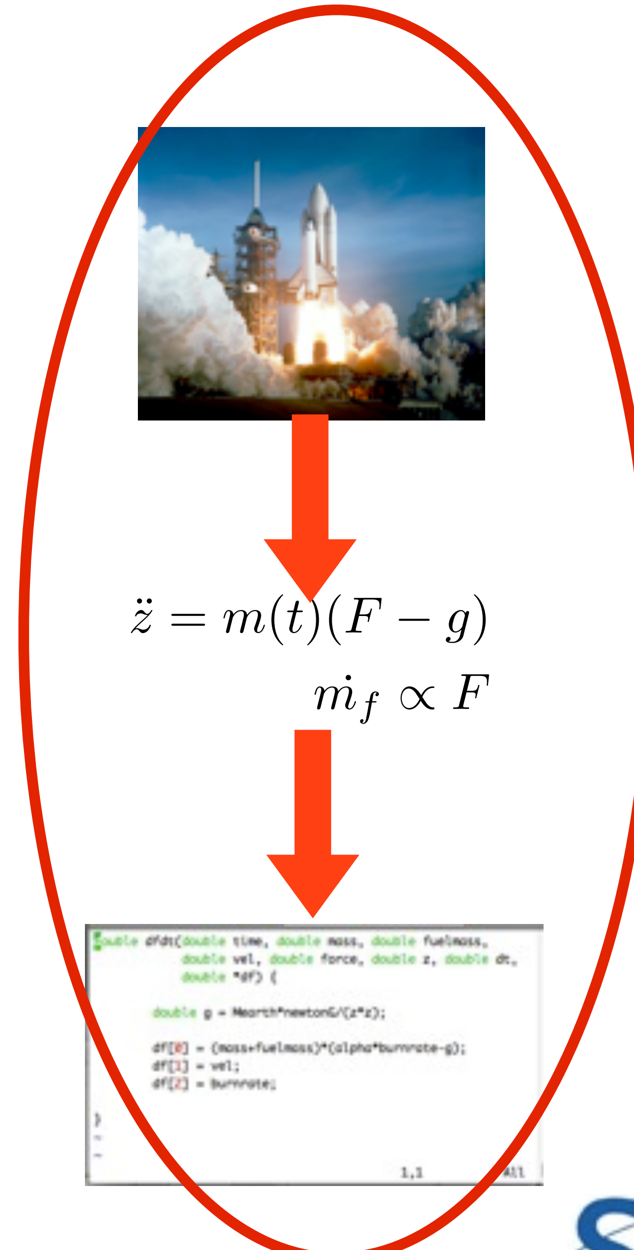
$$\dot{m}_f \propto F$$



```
double dfdt(double time, double mass, double fuelmass,
        double vel, double force, double z, double dt,
        double *df) {

    double g = Mearth*newtonG/(z*z);

    df[0] = (mass+fuelmass)*(alpha*burnrate-g);
    df[1] = vel;
    df[2] = burnrate;

}
~
~
                                        1,1          All
```

Validation: Testing reality ⇒ numerics

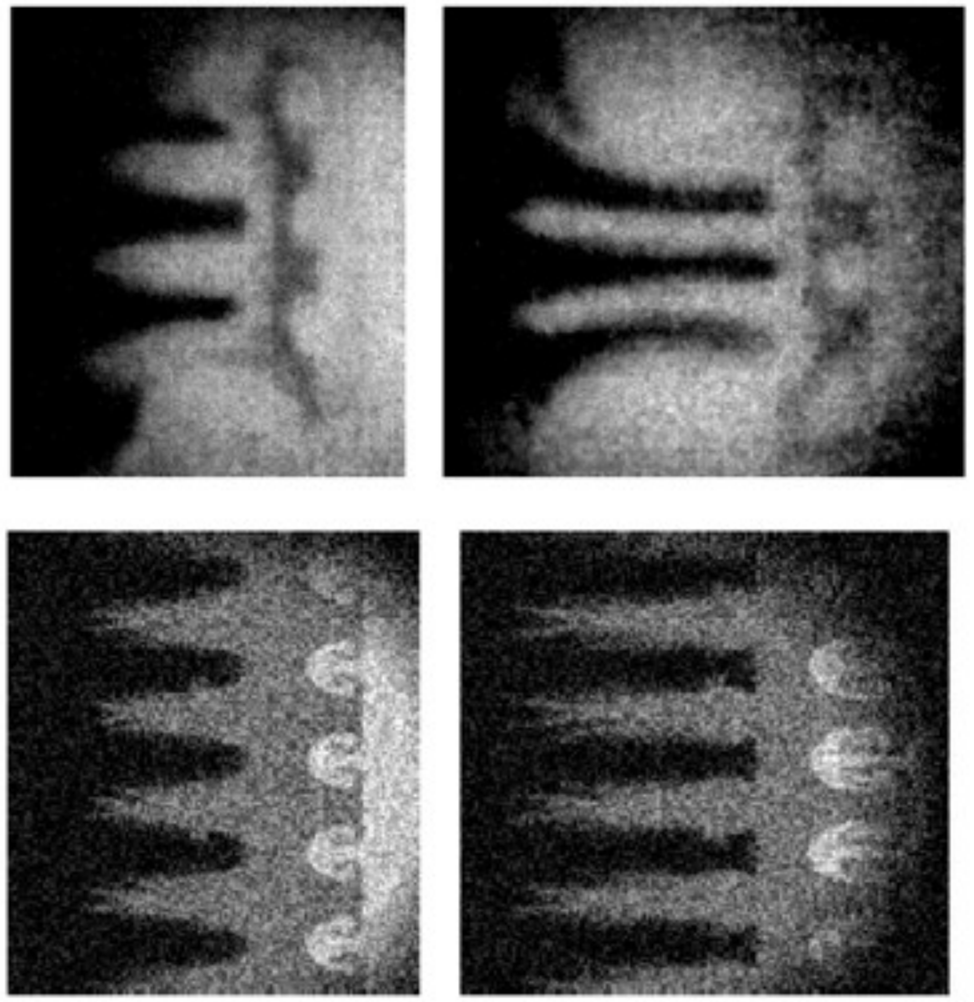
- Can go wrong in translation from phenomenon to mathematical model
- Typically only implementation of full mathematical model you have is the code
- Testing code against reality



SciNet compute • calcul CANADA

Validation: Code/ Experiment comparisons

- Only way to do validation is to compare directly to experimental results
- Must be in regime you are realistically interested in, but still experimentally accessible
- Requires collaboration with experimentalists.
- Proves that there's a regime in which your code accurately reproduces reality.



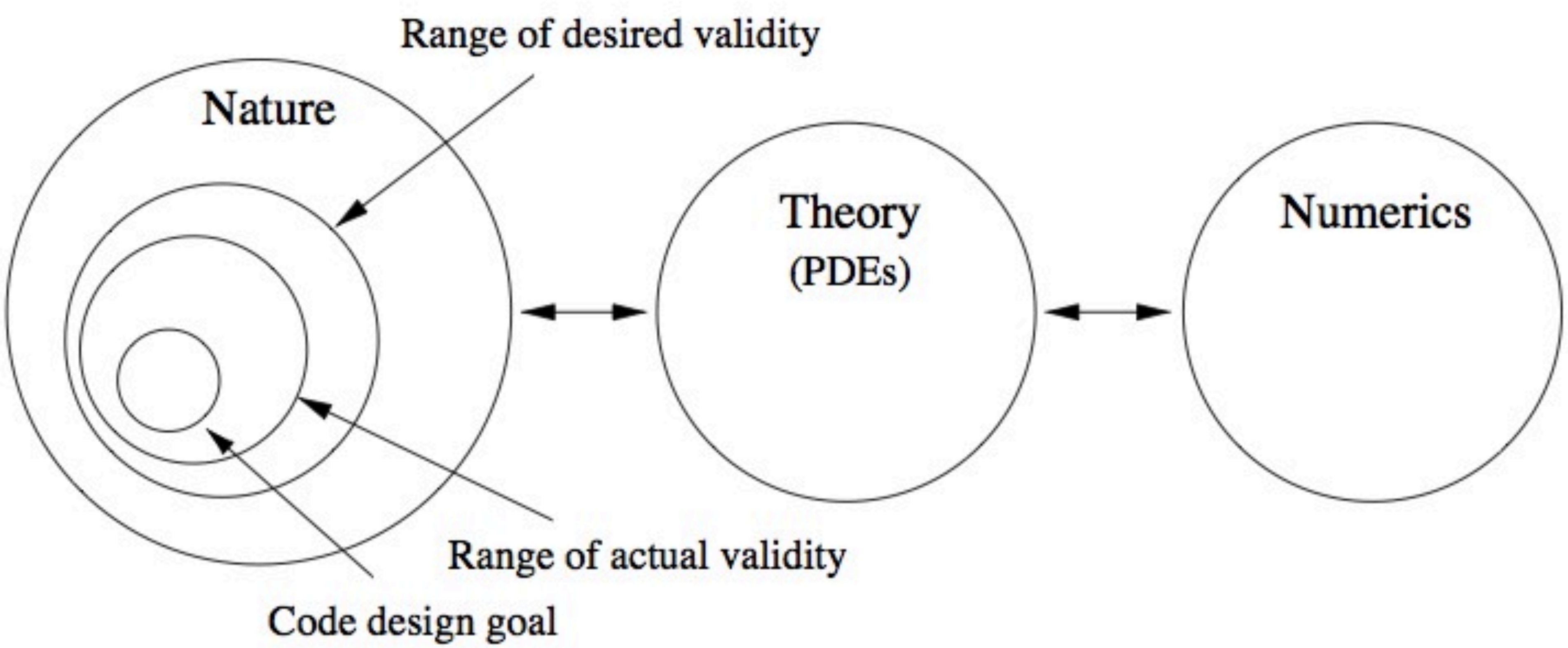
arXiv:astro-ph/0206251

Not a one-off

- Even after an extensive V&V effort, code changes
- Still giving right answer?
- Unit tests, **regular** integrated tests crucial to *maintaining* correctness
- Who cares that your code once gave correct answer *once*, some Thursday two years ago?

SciNet compute • calcul CANADA

Regimes of Interest



Floating Point Mathematics

Like real numbers, but different.

Integer Math and Computers

- Infinite number of integers
- Finite size of integer representation
- Finite range. One bit for sign; can go from -2^{31} to (almost) 2^{+31} (−2,147,483,648 to 2,147,483,647)
- Unsigned: $0..2^{32}-1$

int: 32 bits = 4 bytes

Integer Math and Computers

- long long int:

- One bit for sign; can go from -2^{-63} to (almost) 2^{+63} (−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

- Unsigned: $0..2^{64}-1$

long long int: 64 bits = 8 bytes

SciNet compute • calcul CANADA

Integer Math and Computers

- All integers within range are exactly representable.
- Absolute spacing (1) constant across range; relative spacing varies
- All operations (+,-,*) between representable integers represented unless overflow (with either sign)

Fixed point numbers

- Could treat real numbers like integers - 0...INT_MAX, with say the last two digits 'behind decimal point'.
- Financial stuff often uses this; only ever need/want two decimal points
- Horrifically bad for scientific computing - relative precision varies with magnitude; cannot represent small and large numbers at same time.

Floating Point Numbers

- Analog of numbers in scientific notation
- Inclusion of an exponent means point is “floating”
- Again, one bit dedicated to sign

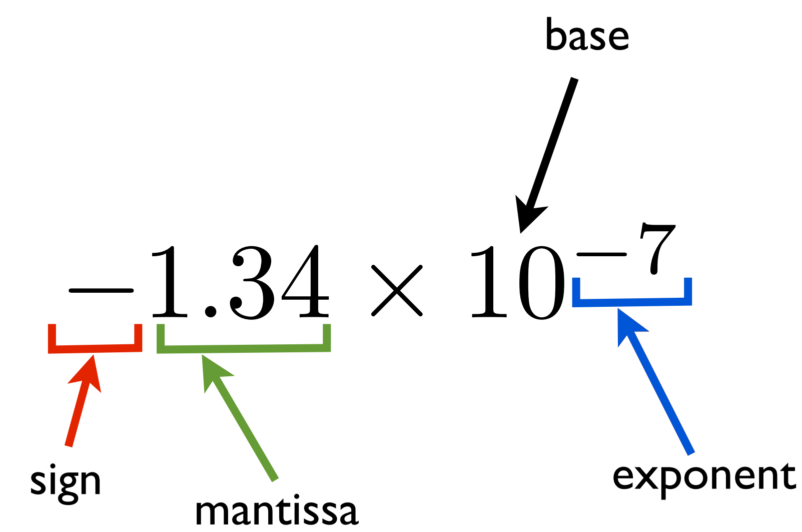
base

$$-1.34 \times 10^{-7}$$

sign mantissa exponent

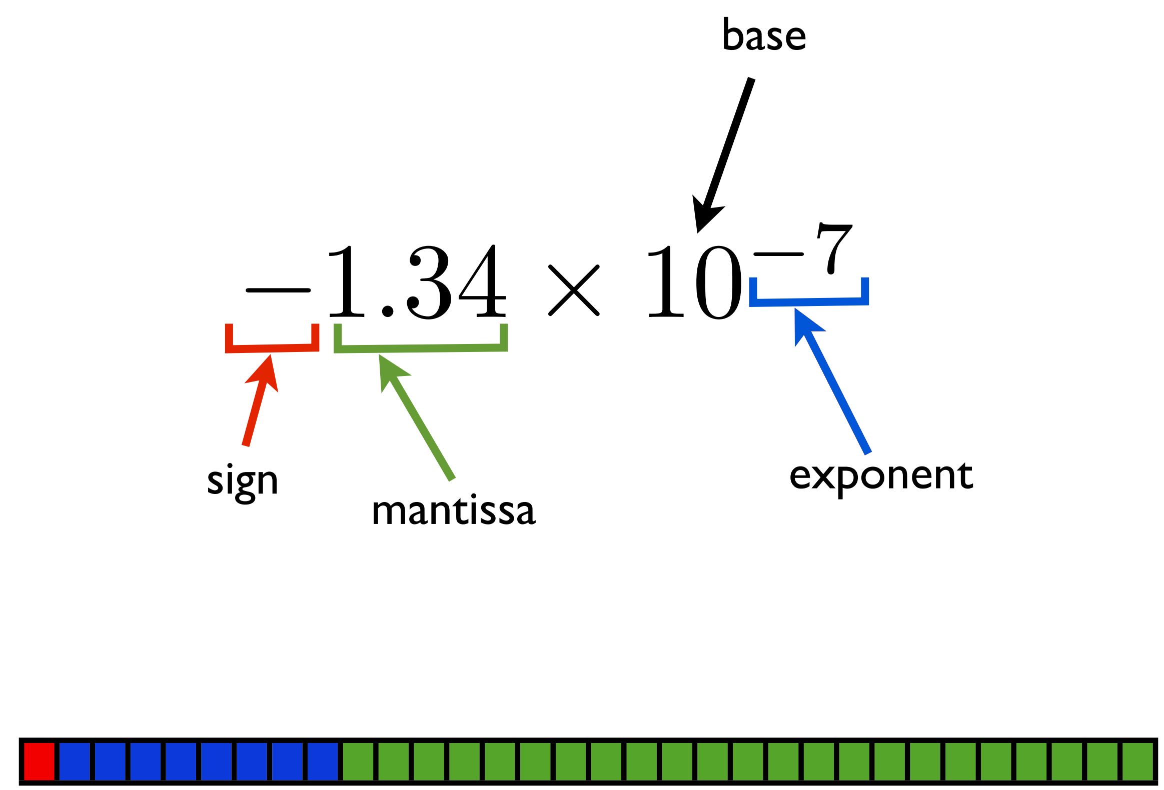
Floating Point Numbers

- **Standard**: IEEE 754
- Single precision real number (float):
- 1 bit sign
- 8 bit exponent (-126..127)
- 23 bit mantissa
- double precision: 1/11/52



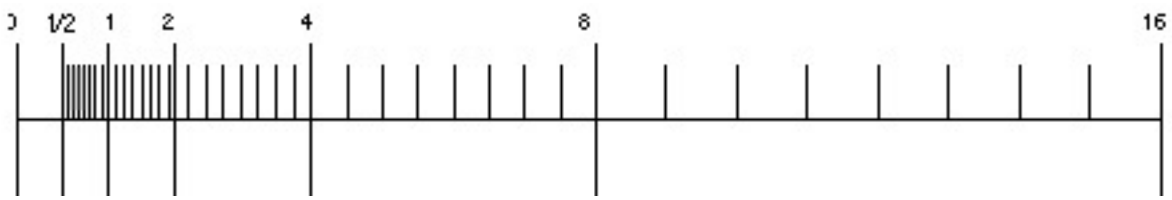
Floating Point Numbers

- To ensure uniqueness of represention (don't waste patterns), first bit of mantissa always 1.
- Since always 1, don't need to store it
- Really 24 (53) bits of mantissa
- Normalized numbers

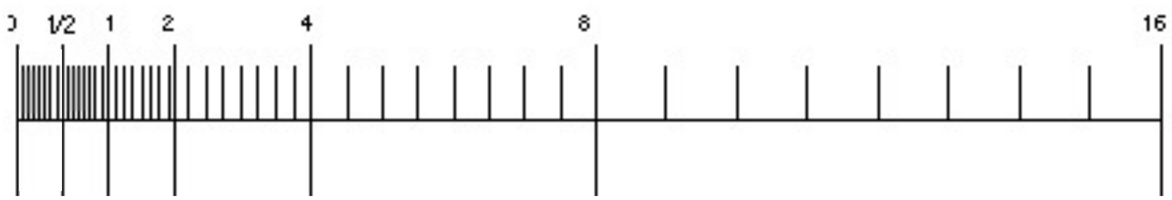


Denormal numbers

- This actually leads to a big jump between smallest possible number and zero



- Relative accurracy doesn't degrade gracefully



- So *if* exponent = minimum, assume first bit of mantissa = 0

Special “Numbers”

- There’s room in the format for the storing of a few special numbers:
- Signed infinities (+Inf, -Inf): result of overflow, or divide by zero.
- Signed zeros - signed underflow, or divide by +/-Inf
- Not a Number - NaN.   Sqrt of a negative, 0/0, Inf/Inf, etc.
- All of the events which lead to these are (usually) errors and can be made to cause exceptions.

Underflow: mostly harmless?

- Try the following:
- Repeatedly take sqrts, then square a number
- Plot this from 0..2
- What should you get? What do you get?
- Like cancellation, loss of precision in early stage of calcuation can cause problem

```
In [9]: def sqrts(x):
   ...:     y = x
   ...:     for i in xrange(128):
   ...:         y = sqrt(y)
   ...:     for i in xrange(128):
   ...:         y = y*y
   ...:     return y
   ...:

In [10]: x = linspace(0.,2.,1000)

In [11]: y = sqrts(x)

In [12]: plot(x,y,'o-')
```

SciNet compute • calcul CANADA

Floating Point Exceptions

- Let's look at the following Fortran code
- Second division should fail
- If compile and run as is, will just print NaN for C
- Can have it stop at error:

```
program nantest
    real :: a, b, c

    a = 1.
    b = 2.

    c = a/b
    print *, c,a,b

    a = 0.
    b = 0.

    c = a/b
    print *, c,a,b

    a = 2.
    b = 1.

    c = a/b
    print *,c,a,b
end program nantest
```

Floating Point Exceptions

- Compiling with gfortran, can give -ffpe-trap options
- will trap (throw exception, stop) for invalid, divide by zero, overflow
- Could also do underflow
- Debugger stops at line that causes exception

```
$ gfortran -o nantest nantest.f90
   -ffpe-trap=invalid,zero,overflow -g

$ gdb nantest
[...]
(gdb) run
Starting program: /scratch/ljdursi/Testing/fortran/na
  0.50000000       1.0000000       2.0000000

Program received signal SIGFPE, Arithmetic exception.
0x0000000000400384 in nantest () at nantest.f90:13
13            c = a/b
Current language:  auto; currently fortran
```

Floating Point Exceptions

- C: include fenv.h, and use feenableexcept (enable exceptions)
- constants defined
- gdb again works

```
#include <stdio.h>
#include <fenv.h>

int main(int argc, char **argv) {
    float a, b, c;
    feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW

    a = 1.;
    b = 2.;

    c = a/b;
    printf("%f %f %f\n", a, b, c);

    a = 0.;
    b = 0.;

    c = a/b;
    printf("%f %f %f\n", a, b, c);

    a = 2.;
    b = 1.;

    c = a/b;
    printf("%f %f %f\n", a, b, c);

    return 0;
}
```

Floating Point Exceptions

- C: include fenv.h, and use feenableexcept (enable exceptions)
- constants defined
- gdb again works

```
$ gcc -o nantest nantest.c -lm -g
$ gdb ./nantest
[...]
(gdb) run
Starting program: /scratch/s/scinet/ljdursi/Testing/
exception/nantest
1.000000 2.000000 0.500000

Program received signal SIGFPE, Arithmetic exception.
0x00000000004005d0 in main (argc=1, argv=0x7fffffffe4
nantest.c:17
17      c = a/b;
```

SciNet compute • calcul CANADA

Floating Point Math

- Fire up Python, and try the following:

```
In [1]: print 1
1

In [2]: print 1.e-17
1e-17

In [3]: print (1. + 1.e-17) - 1.
??

In [4]: print (1. - 1.) + 1.e-17
??
```

Floating Point Math

- Fire up Python, and try the following:

```
In [1]: print 1
1

In [2]: print 1.e-17
1e-17

In [3]: print (1. + 1.e-17) - 1.
0.0
```

SciNet compute • calcul CANADA

Errors in Floating Point Math

- Assigning a real to a flaotting point variable involves truncation

- Error of 1/2 ulp (unit in last place) due to rounding due to assignment to finite precision

- (single precision: 1 part in 2^{-24} ~ 6e-8; double, 1e-16)

$$
\begin{aligned}
x = 1/5 &= 0.2_{10} \\
&= 0.001100110011\ldots_2 \\
&\ominus 0.0011_2
\end{aligned}
$$

Rounding

- Rounding should not introduce any systematic biases
- IEEE 754 defines 4 rounding modes:
  - to nearest (even in ties): default
  - to 0 (truncate)
  - to +Inf (round up)
  - to -Inf (round down)

Don’t test for equality!

- Because of this error in assignments, and other small perturbations we’ll see, testing for floating point equality is prone to failure.

- Generally don’t test for x== 0, or x==y

- abs(x) < tolerance, or abs(x-y) < tolerance

Rounding

- Can set rounding mode
  - C: #include <fenv.h>, fesetround()
    - FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOZERO
  - Fortran: use,intrinsic IEEE_ARITHMETIC
    - call IEEE_SET_ROUNDING_MODE(),
    - IEEE_DOWN, IEEE_UP, IEEE_TO_ZERO, IEEE_NEAREST

Machine Epsilon

- Let's work in base 10, with mantisa precision=3 and exponent precision=2.
- (ignore denormal/ normalized for now; weird with non-binary)
- 1 + 0.001

Machine Epsilon

- Let's work in base 10, with mantisa precision=3 and exponent precision=2.
- (ignore denormal/normalized for now; weird with non-binary)
- 1 + 0.001
- There are numbers x such that 1 + x = 1 even though x isn't 0!

$$1 + 10^{-3}$$

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\ 1.00 \times 10^{-3} \\ \hline \end{array}$$

$$\begin{array}{r} 1.00 \quad \times 10^{0} \\ +\ 0.001 \times 10^{-3} \\ \hline =\ 1.00 \times 10^{0} \end{array}$$

Machine Epsilon

- Defined to be the smallest number s.t. 1+x != 1
- (or sometimes, the largest number s.t. 1+x = 1)
- single IEEE precision: ~1.19209e-07; double, ~2.22045e-16
- By repeated halving, try to see if you can calculate machine epsilon this way. What precision is default floating point number in python?

Machine Epsilon

```
In [4]: x = 1.

In [5]: while 1. + x > 1.:
   ...:     print x, 1.+x
   ...:     x = x / 2.
[...]
2.22044604925e-16 1.0
```

First lesson of floating point numbers

- Be wary of adding numbers that are potentially of very different magnitude
- Relative size ~ machine epsilon, regardless of absolute magnitude (eg, $10 + 10e_{mach} \sim 10$).
- What should we do when adding large series of numbers, even if of roughly same magnitude?

Subtraction: cancellation

- The same effect in opposite with subtraction
- Be wary subtracting very **similar** numbers.

$$
\begin{array}{rl}
 & 1.23 \times 10^{0} \\
- & 1.22 \times 10^{0} \\
\hline
= & 1.00 \times 10^{-2}
\end{array}
$$

Subtraction: cancellation

- The same effect in opposite with subtraction
- Be wary subtracting very **similar** numbers.
- "catastrophic cancellation" - lose precision
- Dangerous in intermediate results

3 sig fig

3 sig fig

$$\begin{array}{r} 1.23 \times 10^{0} \\ -\ 1.22 \times 10^{0} \\ \hline = 1.00 \times 10^{-2} \end{array}$$

1 sig fig

SciNet compute • calcul CANADA

Things you **do** know

- Subtraction: if x, y floating-point representable numbers and x within a factor of 2 of y, then FP subtraction exact

$$x/2 < y < 2 \Rightarrow x \ominus y = x - y$$

- Rounding error when adding FP x and y is an FP number, and can be computed:

$$r = x + y - (x \oplus y) \Rightarrow r = b \ominus ((a \oplus b) \ominus a)$$

- subtraction is addition of a negative
- similar results exists for multiplication

Things you **do** know

- Math libraries typically provide functions (sin, cos, sqrt, pow, etc) results accurate to ~1-3 ulp, for given FP input
- For exact details, check manual

Be cautious, but don't despair

- FP errors are normally not a concern ~ 1 ulp
- Shouldn't normally be biased one way or another - error of N calculations ~sqrt(N) ulp
- (Note: iterate trillion computations in single precision - likely have O(1) errors)
- BUT need to be careful, especially of repeatedly iterated calculations or of awkward things early in a long calculation - if (eg) lose much precision early in a multi-stage computation

SciNet compute • calcul CANADA

How do you know if there's a problem?

- Can test:
    - Change precision (single to or from double; fortran allows quad). Does answer significantly change?
    - Perturb calculation at ulp level by changing rounding behaviour. Does answer significantly change?
    - Perturb calculation by slightly changing inputs. Does answer significantly change?
- If you pass 3 tests, some evidence you're doing ok.

Floating Point and Compilers

- You generally want to turn on heavy levels of optimization when compiling (-O2, -O3); this can speed up your code significantly
- At -O3 levels (by convention), the compiler is allowed to re-order mathematical operations in such a way that, mathematically, give same answer
- But numerically may not!
- Overview of optimization flags for intel compilers: http://wiki.scinethpc.ca/wiki/images/7/77/Snug_techtalk_compiler.pdf

Floating Point and Compilers

- If your code is already demonstrated to be numerically stable, these perturbations shouldn't be a big issue.
- BUT:
  - Compiler may, by dumb luck, stumble on a reordering which is numerically unstable. Test with different optimization flags.
  - If your code includes something you've carefully written for numerical stability that you don't want reordered, put it in a separate file and compile it with -O2 or less

Floating Point and Compilers

- Other optimization flags include things like -ffast-math (or -funsafe-optimizations) which do more agressive changes
- -ffast-math likely does things like replace divisions with multiplication-by-reciprical, which is less accurate; may use less accurate but faster math functions. Worth trying, but be careful with this.
- -funsafe-math-optimizations: ditto.

SciNet compute • calcul CANADA

Architecture and Floating Point

- In theory, all IEEE-754 compliant hardware should give same results.
- Mostly true; some small non-compliances here and there. Not normally a big worry (change in compilers more likely to cause numerical changes)
- Biggest difference: x86 does FP math on variables in registers in "extended precision" - 80 bits vs. 64.
- Higher precision, but depends on whether variables are in registers, etc.
- Can cause difference between x86 and other achitectures.
- Be aware of this.

Easiest way to avoid problem

- Don't write numerical code when you don't have to!
- If there exist numerical libraries for things you want to do (ODE, integration, FFTs, linear algebra, solvers), *use them*.
- Amongst other benefits, the numerical issues have been worked out in most mature, highly-used code bases.

Things to avoid

- Subtractions of like-sized variables early in calcuation
- Sumations of large amounts of numbers, or numbers of widely varying magnitudes
- Testing for exact floating-point equality

Things to do

- Try to keep values normalized in some sense so that all the values you're likely to deal with are of order unity (avoids machine epsilon problems)
- Try to use existing libraries when necessary
- Routinely test your code

Random Number Generation

Introducing uncertainty on purpose

Based on: "Random Numbers in Scientific Computing: An Introduction", Katzgrabber, arXiv:1005.4117

SciNet compute • calcul CANADA

Need Random Numbers

- For randomly sampling a domain
- Monte Carlo / MCMC simulations
- Stochastic algorithms

Required Properties

- What is a random sequence of numbers?
- Follow some desired distribribution
- Unpredictable
- Fast (we may need billions of them)
- Long period (we may need billions of them)
- Uncorrelated

SciNet compute • calcul CANADA

Real Random Numbers

- Can be generated by a physical process, and stored as a list or used in real-time by computer
- Physical process - lava lamp (lavarnd.org), quantum stuff
- Network process - /dev/urandom
- Generally slow, expensive, hard/impossible to reproduce for debugging
- Often hard to characterize underlying distribution

Pseudo Random Number Generators

- PRNG
- Software-based; deterministic sequences of numbers based on some starting seed
- "Seem" random, but reproducible (with same seed), often very fast.
- Will assume uniform distribution on [0,1); given this, can create other distributions

Common Tests: Correlations

- Simple pairwise correlations:
- Want to avoid correlations between pairs of numbers

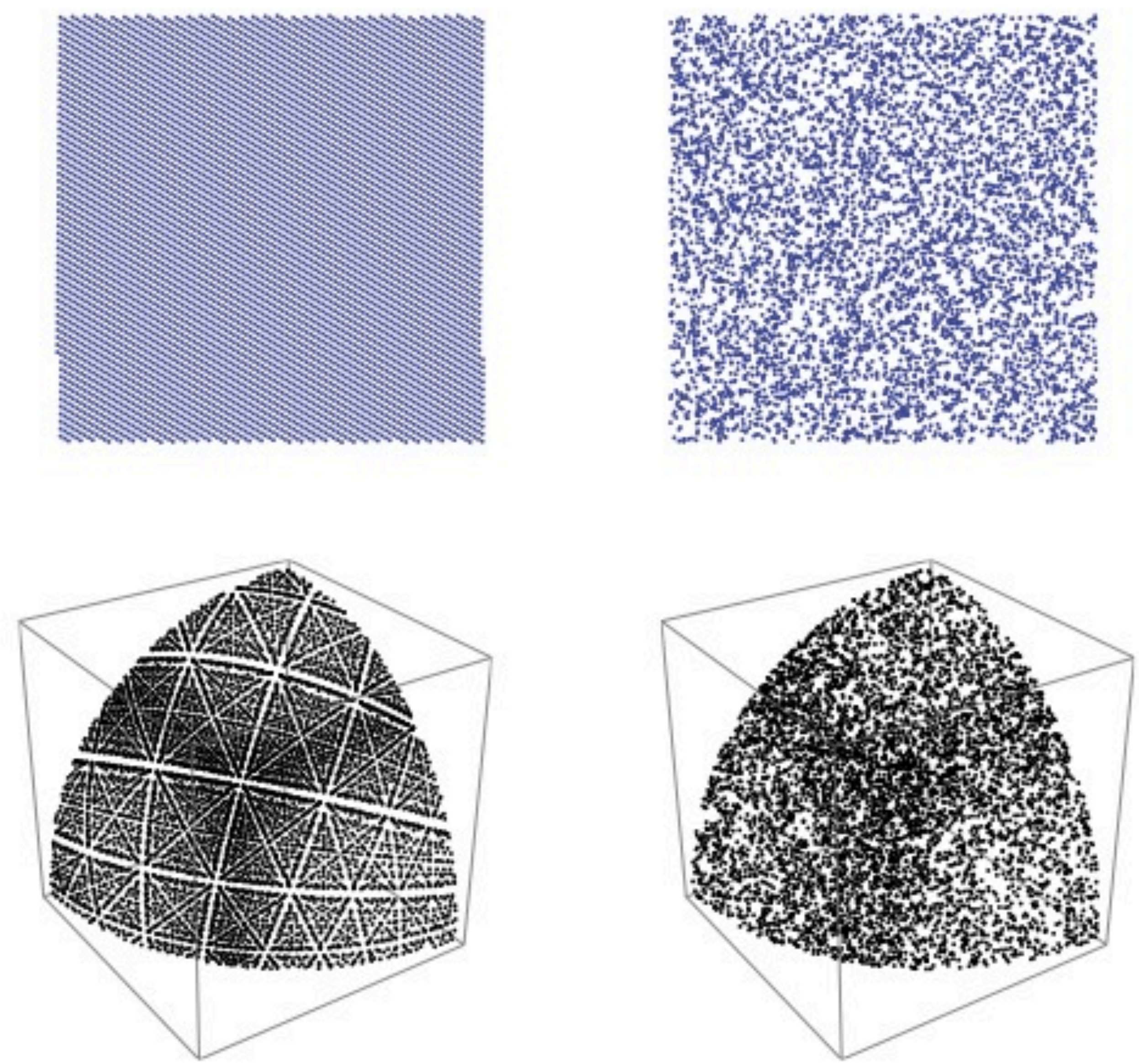
$$\varepsilon(N,n) = \frac{1}{N}\sum_{i=1}^{N} x_i x_{i+n} - E(x)^2$$

$$E(x) = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$E(N,n) = O(N^{-1/2}) \quad \forall n$$

Correlations

- What correlations look like in 2d domain
- Left: bad LCG; right: Mersenne Twister



From Katzgraber

Common Tests: Moments

$$\mu(N,k) = \left| \frac{1}{N} \sum_{i=1}^{N} x_i^k - \frac{1}{k+1} \right|$$

- Ensure moments of random numbers also have desired properties

$$\mu(N,k) = O(N^{-1/2}) \quad \forall k$$

Other Tests

- Overlapping permutations: Analyze orders of five consequitive random numbers. The 5! possible permutations should occur with equal probability

- Parking lot test: pairs of random numbers placed in 2-d domain, exclude others within certain distance. After N attempts, points should follow well known distribution

- Spacings: spacings between random points should follow poisson integral if uniformly distributed

- Binary rank test - test ranks of 32x32 binary matrix

Test suites

- NIST test suite:
  http://csrc.nist.gov/groups/ST/toolkit/rng/index.html
  Very well documented, explain tests.

- Pierre L'Ecuyer, U de Montréal:
  http://www.iro.umontreal.ca/~simardr/testu01/tu01.html
  Test suite in C, includes several PRNGs

- Best test: one that is related to the properties you need for your problem.

Linear Congruential Generators

- x_0 is a seed
- m - large integer; determines period of sequence
- For U(0,1), divide x_i by m.
- For good results: c relatively prime to m, a-1 a multiple of p for every prime divisor p of m, a-1 is multiple of 4 if m is multiple of 4.

$$x_{i+1} = (ax_i + c) \mod m$$

SciNet compute • calcul CANADA

Linear Congruential Generators

- Common, but not very good
- Period limited by size of integers; not enough for some applications.
- Hard to do well in parallel
- **Easy** to mess up, with long history of bad LCGs in standard implementations, literature.

$$x_{i+1} = (a x_i + c) \mod m$$

Linear Feedback Shift Register Generators

$$x_i = (a_1 x_{i-1} + \cdots + a_n x_{i-n}) \mod p$$

- Generalization of LCG
- Good period iff characteristic polynomial defined by a_i is primitive modulo p
- Requires big seed (n x_is); typically use small seed + good small PRNG to seed
- Still not great - better period (p^n).
- Mersene Twister is a (good) generalization of this.

Lagged Fibonacci

- Some binary operator between previous items in sequence

$$x_i = (x_{i-j} \odot x_{i-k}) \mod m$$

- Requires some memory
- Requires large seed block again
- m typically large power of 2

Lagged Fibonacci

$$x_i = (x_{i-j} \odot x_{i-k}) \mod m$$

- r1279: k=1279. Period is 10^{394} ; passess tests, and can be fast
- Standard in (eg) GSL

Lagged Fibonacci

- r250: k = 250, using xor.
- Also fast, passed all common tests at time
- In 1992, Ferreberg et al did MC simulation of Ising model
- Estimate of energy/per spin was 42σ off!
- PRNGs are hard; **don't** implement yourself.

$$x_i = (x_{i-j} \odot x_{i-k}) \mod m$$

Some good PRNGs

- r1279
- Mersenne twister (mt19937)
- WELL generators

Not-good PRNGs

- r250
- Anything from Numerical Recipies - short periods, slow, ran0 & ran1 spectacularly fail statistical tests.
- Standard Unix generators (rand(), drand48()) - not a disaster, but short period, correlations.

Non-Uniform Distributions

- Transformation law of probabilities
- Starting with a known distribution (eg, uniform, p(u) = 1 in 0..1), can transform to another distribution (q(y)) if can invert function

$$|q(y)dy| = |p(u)du|$$

$$\Rightarrow q(y) = p(u)\left|\frac{du}{dy}\right|$$

Exponential Dist.

- Example: exponential distribtion
- Easy to invert, differentiate
- Can get exponential distribution by taking ln of uniform random numbers.

$$|q(y)dy| = |p(u)du|$$
$$\Rightarrow q(y) = p(u)\left|\frac{du}{dy}\right|$$

$$q(y) = a\exp(-ay)$$
$$\left|\frac{du}{dy}\right| = a\exp(-ay)$$
$$u(y) = \exp(-ay)$$

$$y = -\frac{1}{a}\ln(u)$$

Box-Muller: Gaussian Random Numbers

- Same process can be applied to more complex dists, with some tricks.
- For gaussian, can't do it in 1d, but can in 2
- Generate 2 gaussian RNs (unit σ, zero mean) from 2 uniform

$$x = \sqrt{-2\ln(u_2)}\cos(2\pi u_1)$$

$$y = \sqrt{-2\ln(u_2)}\sin(2\pi u_1)$$

Shifting distribution

- If just need to shift distribution, easy
- U(a,b): (b-a)*(u + a) where u from U(0,1)
- Can similarly shift gaussian distribution from unit, zero-mean gaussian to others

Acceptance/Rejection

- If can't invert your desired distribution f(x), can still generate RN
- Numerically invert (tabulate)
- Or:
  - Generate distribution you can on same domain, g(x)
  - Reject numbers with probability 1-f(x)/g(x) (eg, generate random number u[0,1], x from g;
    accept if u< f(x)/g(x)
  - Faster if g tightly bounds f (less rejected guesses

GSL - Gnu Scientific Library

- Gsl has several good implementations of good PRNGs
- Seperates the generator from the distribution you want

```c
#include <stdio.h>
#include <gsl/gsl_rng.h>

int main(int argc, char **argv) {
    gsl_rng *rng;
    int i;
    double u;

    rng = gsl_rng_alloc(gsl_rng_mt19937);
    gsl_rng_set(rng, 1);

    for (i=0; i<100; i++) {
        u = gsl_rng_uniform(rng);
        printf("%d %f\n", i, u);
    }

    gsl_rng_free(rng);

    return 0;
}
```

Create, seed PRNG

Generate Random #s

Clean up

```
$ gcc -o gsl gsl.c -I${SCINET_GSL_INC}
  -L${SCINET_GSL_LIB} -lgslcblas -lgsl
```

Python

- Numpy.random - series of random number generators, distributions.
- Based on mersenne twister
- Good, but would be nice to have choice...

SciNet compute • calcul CANADA

Notes on Seeding

- For random seeds, taking system time is common
- If doing in parallel, need to make sure different processes/ threads have different seeds!
- Factor rank, thread num, pid, etc in there somehow

Homework: 1

- Consider the sequence of numbers: 1 followed by 10^8 values of 10^{-8}
- Should Sum to 2
- Write code which sums up those values in order. What answer does it get?
- Add to program routine which sums up values in reverse order. Does it get correct answer?
- How would you get correct answer?
- Submit code, Makefile, text file with answers.

Homework: 2

- Implement an LCG with a = 106, c = 1283, m = 6075 that generates random numbers from 0..1
- Using that and MT: generate pairs (dx, dy) with dx, dy each in -.1.. +.1. Generate histograms of dx and dy (say 200 bins). Look ok? What would you expect variation to be?
- For 10,000 pts: take random walks from 0,0 until exceed radius of 2, then stop. Plot histogram of final angles for the two PRNGs. What do you see?
- Submit makefile, code, plots, VC log