Software Design and Development

Some Common Numerical Issues

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Overview

This is NOT a Numerical Analysis course!
A minimal thorough one is a MPhil on its own

Describes some very common classes of problem
And gives some approaches for resolving them

1: Low–level issues (cancellation etc.)
2: Accuracy issues for linear systems etc.
3: Other widespread, important issues

It's the problems you don't expect that catch you

Beyond the Course

There is more detail and further reading on:

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Arithmetic/
https://www-internal.lsc.phy.cam.ac.uk/nmm1/Development/

https://www.cl.cam.ac.uk/teaching/1819/...
...../Numerical_Analysis_2019.pdf

http://www.damtp.cam.ac.uk/user/hf323/...
...../L19-IB-NA/index.html

Numerical Analysis References

There are lots of good numerical analysis books
Regrettably, I don't know of ones to recommend

The NAG library documentation is also very good
https://www.nag.com/content/...
...software–documentation

Do NOT trust Numerical Recipes an inch
And I strongly advise NOT using its code
A lot of what it says is completely wrong
Yes, it has improved – the first edition was even worse

Authoritative References

The best modern reference:
The Accuracy and Stability of Numerical Algorithms
by Nicholas J. Higham

The classic reference:
The Algebraic Eigenvalue Problem by
J.H. Wilkinson

Also very highly regarded:
Matrix Computations (now 4th ed.) by
Gene H. Golub and Charles F. Van Loan

Reminder: Libraries

Recommended to use a package or library:

- NAG library is most general reliable library
- Good open−source libraries (e.g. LAPACK)
- Many others are seriously unreliable or worse
- Do NOT trust Numerical Recipes or the Web

Your field may well have a preferred one
As usual, check with experts in your field

Low-Level Accuracy Issues

You may have been taught several of these

If so, consider this as a reminder

Garbage In, Garbage Out

Results almost never better than input (GIGO)
And they can be much less accurate

- Do NOT assume machine precision in result

Some forms of error can be reduced statistically
But not all, and there are issues (see later)

N digits means at most $10^{-(N-1)}$ accuracy
Last few digits on data loggers may not be correct
Also, many physical constants are imprecise

Don't Ask the Impossible

Given a function F, and error in input δ
Absolute error in $F(X)$ is at least $F'(X) \times \delta$
Consider $atan$ near $\pi/2$, for example

Any function with singularities has this problem
But can encounter it in functions with none

- Always do a quick check for such issues

If a problem, need to rethink the approach

Cancellation

- Low–level cause of most loss of accuracy

Caused by subtracting two nearly–equal values

Obviously, includes adding two with different signs

- But also dividing (and multiplying by inverse)

Assume numbers have P digits of precision

X and Y have Q leading digits in common

$\Rightarrow$ X–Y and X/Y–1.0 have precision P–Q

- Restructuring expressions can help a lot

Expression Reordering

Where it matters, consider changes like the following:

(X+D)**2–X**2 ⇒ (2*X+D)*D

x^5–y^5 ⇒ (x^4+x^3*y+x^2*y^2+x*y^3+y^4)*(x–y)

sin(x+d)–sin(x) ⇒ sin(x)*(cos(d)–1.0)+cos(x)*sin(d)

I haven't used this, but you might like to try:

http://herbie.uwplse.org/

Approximating Functions

A continued fraction is usually the best method
A lot of nice properties and rarely much cancellation

- Don't rush in – also look up Lentz algorithm

But Taylor series are easier to derive

Don't have to use derivatives – can just fit polynomials

Also Padé approximants (ratios of polynomials)
Can also get by expanding continued fractions
Often converge lot faster than polynomials

But cancellation is often a serious issue for both

Polynomials, Taylor Series etc. (1)

Multinomials generally, Padé approximants etc.

Cancellation often associated with slow convergence

$$log(1+X) \;=\; X - X^2/2 + X^3/3 - ... - (-x)^k/k + ...$$

But not always, unfortunately

$$exp(X) \;=\; 1 + X + X^2/2 + X^3/6 - ... + x^k/k! + ...$$

Now consider $X = \pm 100$ and blench

Both have foul cancellation problems

Mentioned later: "The Perfidious Polynomial"

Polynomials, Taylor Series etc. (2)

Asymptotic expansions (e.g. Stirling's) are odd
There is a best number of terms for any argument

For all such approximations, if an issue:

1) Reduce range if feasible
2) Reorganise to reduce cancellation
3) Reorganise to accelerate convergence
4) Solve a related function and convert

E.g. logarithm can be done by inverting exponential

5) Consider other approaches (e.g. integrals)

Yes, really – normal scores are best done that way!

Large Reductions (1)

Main ones are summations and inner products
For $\prod X_i$, generally use logarithms
But watch out for $1 + \epsilon \approx 1$ and losses

Kahan summation is usually more accurate
I use a similar method, which has some advantages
Most reliable is to emulate extra precision

Too tricky to teach now, for two reasons:
1) Needs advanced floating–point hackery
2) Compiler optimisation often breaks them

Large Reductions (2)

But, for example code, see:

https://www-internal.lsc.phy.cam.ac.uk/nmm1/...
.../Development/Programs/fancy_accumulate.cpp
.../Development/Programs/fancy_inner.cpp
.../Development/Programs/double_emulation.cpp

Warning: those files contain extreme geekery
Don't assume even Kahan summation is trivial
Look at the comments in the files for details

Implicit Cancellation

Cancellation may be implicit in the algorithm
There is no specific expression where it occurs

- You won't fix that by the above methods

Common with ones that use numerical derivatives
Ideally, you have an algebraic form for them

- You can then provide them, avoiding cancellation

But sometimes it is deep within the logic

- Only real solution is a more stable algorithm

Summary

- The above is NOT an exhaustive list!

It's just some of the most common problem areas

Keep a watch out for such low–level inaccuracies

You can usually reduce them considerably if needed

Think laterally – how else could you do it?

Book on This Topic

Real Computing Made Real:
Preventing Errors in Scientific and
Engineering Calculations

by Foreman S. Acton

Good, clear book on avoiding precision loss etc.
Explains only how to prevent some forms of error!

Higher Level Issues

This is where classical numerical analysis comes in
It is applied mathematics, not low–level computing

Applies to all arithmetics modelling real numbers
and to all algebras derived from them
Complex numbers, quaternions, matrices, ...

Will mention a lot of techniques but not describe them
Look them up – start with Wikipedia!
If actually need to use them, find a proper reference

Reminder: Transformations

Transforming variables can help immensely
Will mention only a couple of reasons

Often taught to improve convergence
And faster convergence often reduces cancellation
Can also sometimes avoid asymptotic expansions

Also for singularities and discontinuities
3–D rotations using roll, pitch and yaw is evil
Convert to direction cosines and all is well

Root Finding, Minimisation (1)

Applies to polynomials, eigenvalues,
non–linear systems and parameter estimation

Newton–Raphson is simple and moderately good

Fancier methods faster if function is well–behaved
Near–quadratic in region of well–separated roots etc.
Close roots makes them much slower, and can fail

Very similar remarks apply to minimisation
There isn't a semi–canonical method, though

Root Finding, Minimisation (2)

But sometimes functions are not well−behaved

Binary chop needs only monotonicity
Fibonacci minimisation is similar for a minimum
And so is steepest descents, perhaps fiddled a bit

Still will have problems with very close roots
And truly evil n−D problems can run like drains
E.g. exponential helix with solution at centre

But nothing else will do any better in such cases

Linear Systems

This includes most uses of matrices
This area is very well–understood
Algorithms have known, reliable error bounds

Numerical analysis books go into huge detail
We shall not be doing so!

We shall start with some general rules
Often applicable much more widely

Matrix Theory

I am assuming that you know basic matrix theory

If you don’t understand anything, PLEASE ASK
Not at the end, but AT THE TIME

Warning: C/C++ use Algol order for matrices
This is the other way round to Fortran
If converting code, often reverse loop order
Alternatively, reverse subscript order

Errors in Results

- Errors are usually relative to largest result
  but sometimes largest (input) element

Very small results have a huge relative error

Matrix errors are usually $N \times eps \times cond.\ no.$
N is size of matrix, eps is accuracy of data
The condition number is how ‘evil’ the matrix is

Appropriate condition number varies with analysis
‘Nice’ problems are near 1, ‘nasty’ ones much higher
An infinite one means that the problem is ill–posed
i.e. there are no or multiple answers

It's Not The Arithmetic

Errors bounds are inherent in the mathematics
eps is maximum of input error and $\approx 10^{-15}$
Also for measurements and physical constants

Usually, no point in using extra arithmetic precision

- But critical to use an appropriate algorithm

Sometimes, using more accurate accumulation helps

Start by using good library or reference book
Look at NAG library, LAPACK etc.
Aside: FFTs are $N \times eps$ at worst, if done right

Matrix Guidelines

Real symmetric and Hermitian matrices are simple
They have faster and more robust algorithms

Pivoting and scaling are less likely to be needed
For positive (semi–)definite they never are
Not mentioned further – look them up if necessary

Sparsity adds a great many difficulties
Both in performance and in robustness of algorithms
But a lot of work has been done on it
E.g. the book Direct methods for sparse matrices
by Duff, Erisman and Reid

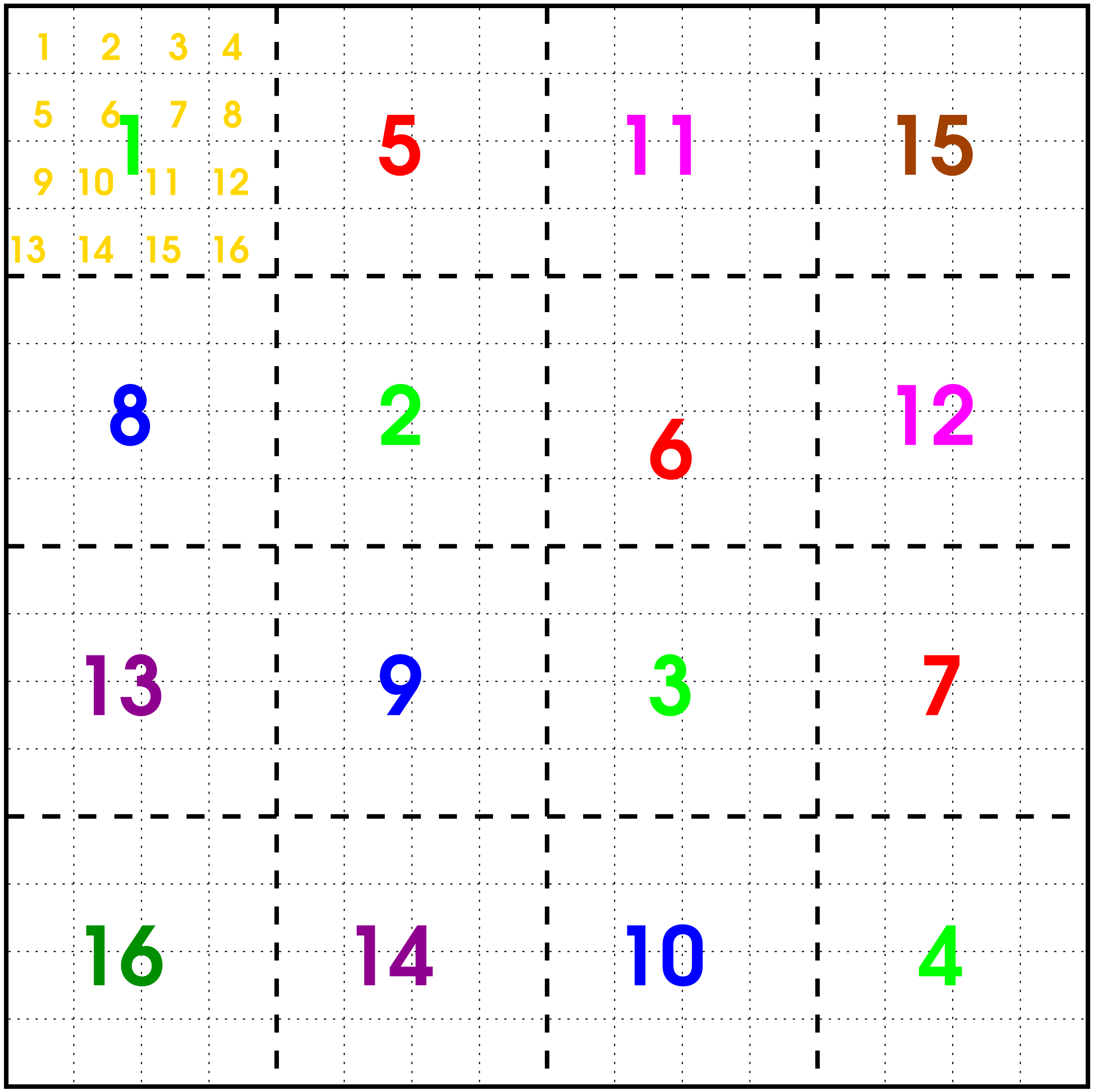
Performance

Obviously, depends critically on the algorithm
Fastest often not most accurate or robust
Esp. for sparse matrices and borderline problems

Large matrix codes can be very cache unfriendly
Best solution is to use blocked algorithms
Not always possible, but often tens of times faster
Messy coding, so reason to use a good library

Even for simple algorithms like transposition
And the optimal code will depend on your system

ONE Blocked Transposition



Solving Equations and Inversion

Mainly $L.L^T$ (Cholesky) and $L.U$ decompositions

Generally not a problem, except for near–singularity
Causes inaccuracy and overflow, if not detected
Avoid inversion, but needed for multivariate statistics

Both sometimes involve singular matrices
Well–posed equations have the zeroes cancelling
In this case, there are solutions (see later)

Easiest is to use SVD and limit inverse values
But check that your near–zeroes really do cancel!

Eigensystems

Consider a $N \times N$ matrix
Always N eigenvalues, but may be equal
Just another form of root finding – covered above
Most common algorithms are QL/QR ones

Equal ones mean eigenvectors are ill–defined
They can be any vector within their subspace
Good algorithms will return orthonormal basis

Real symmetric and Hermitian always have all N
Very nasty unsymmetric ones may not

$$\begin{matrix} 1 & 1 \\ 0 & 1 \end{matrix}$$

Other Decompositions

Lots of them, and use is very domain–specific

Particularly useful in problematic cases
E.g. $L.D.L^T$ instead of $L.L^T$ (Cholesky) where
L is lower triangular and D diagonal

Most general is Singular Value Decomposition (SVD)
Doesn't have any of above problems

Can sometimes take short–cuts, and may need to
Especially for positive semi–definite matrices
E.g. adding $\delta \times N \times maxval.$ to diagonal

Characteristic Polynomial

Polynomial that has the eigenvalues as roots

- Generally best avoided

It has a lot of unobvious numerical issues
Polynomials look simple, numerically, but aren't

- They should always be viewed suspiciously

Look for "The Perfidious Polynomial"
by J.H. Wilkinson
https://en.wikipedia.org/wiki/...
.../Wilkinson's_polynomial

Determinants

A useful example, because can be done many ways

Cholesky or $L.U$ and product of diagonals

Product of eigenvalues (using QL)

From characteristic polynomial in two ways

And more ...

- Use one of the first two

With a 8×8 Hilbert matrix, rotated:

Using Cholesky:	Error: 1.5E–07
Eigenvalue product:	Error: 1.9E–07
Polynomial constant:	Error: 2.4E+02 $\Leftarrow$
Polynomial root product:	Error: 6.3E+02 $\Leftarrow$

Roots of Polynomial

The eigenvalues vary from 1.11E–10 to 1.70
The absolute errors vary from 2.7E–08 to 4.4E–16
Yes, the smallest ones have the largest errors
The relative errors are ridiculous!

Largely due to rounding error and cancellation
In this case, it might be 'fixable' using extra precision

- But not in general ...

General rule: use the right algorithm for the task!
Often not the fastest, when such problems arise

Non-Linear Systems

PDEs, ODEs are just most common examples

This is where things get much trickier
As dependent on actual function as algorithm
As well as domain of the function evaluation

A classic is fluid flow – see Reynolds number
Low–speed is easy, high–speed isn't and transonic?

Similar problems may need different approaches

- Always watch out for unexpected results

Find A Specialist Expert

Approaches for related problems may work
They are always worth at least looking at

If not, need someone who knows about your problem
Or a reliable reference that addresses it
Or sit down and analyse it yourself

- Never just bull ahead and ignore issues
- Always watch out for weird results
- And remember experts aren't omniscient

Instability and Chaotic Systems

Errors can often build–up (super–)exponentially
In this case, single $\Rightarrow$ double will not help

- No option – must improve algorithm

Trivial (not very realistic) example, given above:
K'th differences of x^K, x=0.5,0.51,...,1.99,2.0
In D.P., 1 sig. fig. at K=7, nonsense thereafter

Worst examples are now called chaotic ones

- There may be NO useful algorithm

In which case, you must take another approach

Solving Unstable Systems

Often you can calculate some properties
But only representative details – e.g.:

Long–term orbital mechanics – the orbit is easy
But exactly where will the planets be?

Turbulent fluid flow – not easy, but it can be done
But, to predict individual vortices exactly?

Weather forecasting – it's now pretty reliable
But not "Will it rain at noon at Great St Mary's?"
Nor even local patterns over the long term

Parameter Estimation

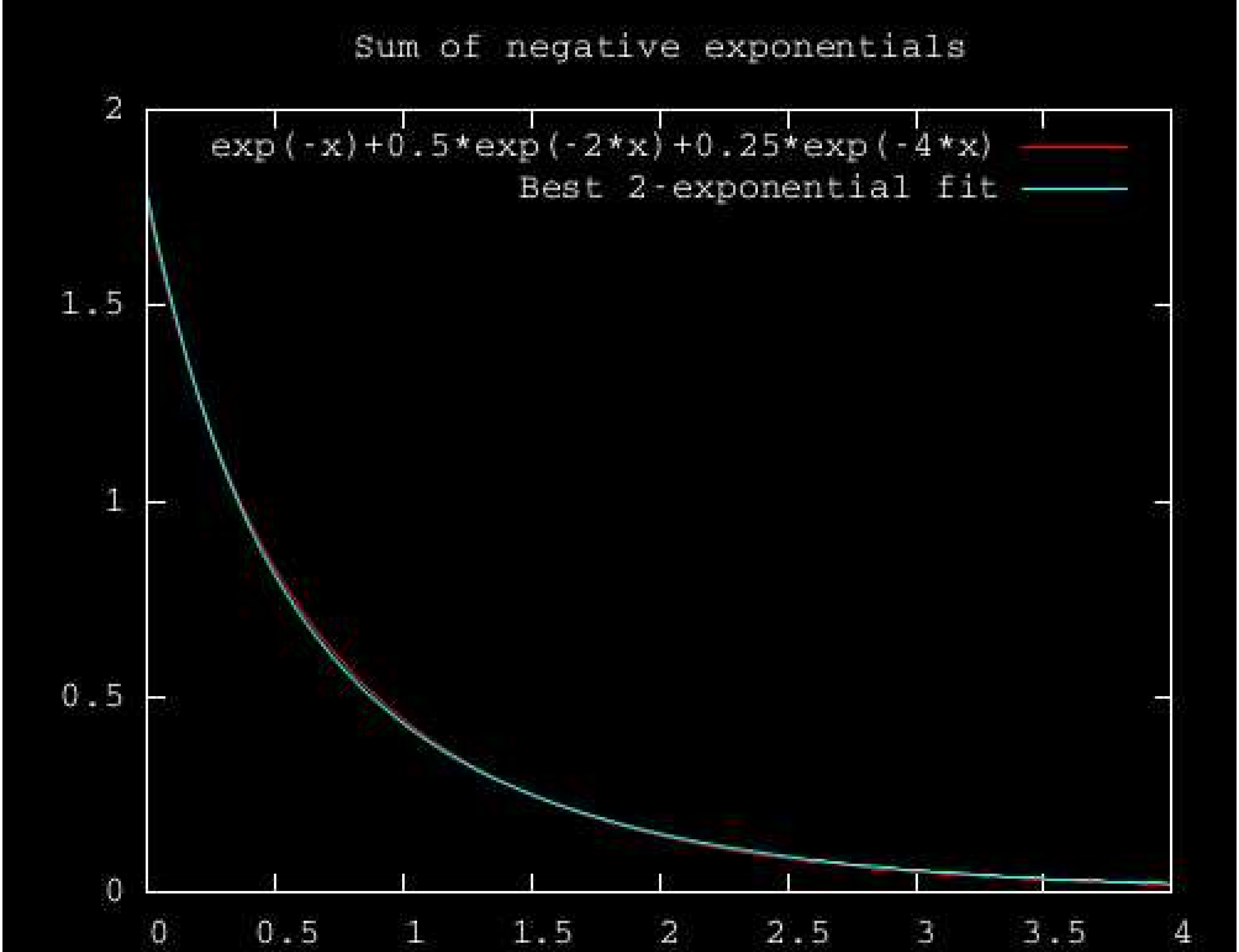
Very easy to fit data as closely as you like
But estimating parameters is hard

Problems often factorial in number of parameters
Because many different functions will also fit

- Even a good fit does not mean useful estimates

Parameters can be too related to be estimated well

Extreme example is sum of negative exponentials
Virtually impossible above $N = 2$ – seriously!


Sum of negative exponentials
exp(-x)+0.5*exp(-2*x)+0.25*exp(-4*x)
Best 2-exponential fit
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4

Errors in Estimates

Only solution is to estimate errors in estimates

This is a multivariate problem – watch out
Each parameter may be precise, if others are fixed
But the estimates are hopeless (as in that example)

You need the inverse of the 2nd deriv. matrix
Near–singularity means too many parameters
as in the horrible example above

Errors are then diagonal elements, scaled
Scaling depends mainly on confidence you need

Factorials and Friends (1)

Extremely common in statistics and combinatorics
Including gamma and beta functions and more
Widespread elsewhere, and often counter–intuitive
N! in 32–bit integers overflows at N = 14
N! in 64–bit reals overflows at N = 171

Logarithms? Avoids overflow, but slow for large N
Solution is to use Stirling's formula in logarithm form:

More on this in notes

$$log(N!) = (N log(N) - N + 0.5 log(2\pi N)) + 1/12N - \ldots$$

Error in expansion that far is $-1/360N^3$

Factorials and Friends (2)

Can be accuracy problems, esp. in combinatorics

Consider $\boldsymbol{Binomial(10^9, 0.4, 4 \times 10^8)}$

Relative error of $\mathbf{4 \times 10{-}4}$ using usual formula

$$\boldsymbol{Bin(N,P,K) \;=\; N!{\times}P^K(1{-}P)^{(N-K)}/(K!(N{-}K)!)}$$

Use Stirling's formula algebraically to fix problem

$$\boldsymbol{Bin(N,P,K) \;=\; (NP/K)^K{\times}(N(1{-}P)/(N{-}K))^{(N-K)}{\times}}$$

$$\boldsymbol{\sqrt{N/(K(N-K)2\pi}{\times}(1{+}(1/N{-}1/K{-}1/(N{-}K))/12)}$$

Algebra packages (not just Mathematica) can help

Better for checking your work than doing it

Monte-Carlo Simulations

Any program that uses randomised data fits here

Far too many people assume this is simple
Statisticians know better, but it's specialised

- Most Web references and books are unreliable

Worse, a great many are actually erroneous

Error is always $O(N^{-0.5})$ – no way round it
But can reduce the constant considerably
Look up "Monte Carlo Methods"

More on this in notes

Problem Distributions (1)

However .... Does the distribution have a mean?
Not all distributions do, either in theory or practice
Need a second moment for decent convergence

Ratio of two independent Gaussian variances
Yes, an extreme case, but a realistic one

Sample of 100: mean was 11.3 ± 6.39 (using 2 SE)
Sample of 1e4: mean was $2,460 \pm 1,919$
Sample of 1e6: mean was $1.34e10 \pm 6.67e10$

Equivalent to the integral being infinite

Problem Distributions (2)

Can sometimes use a transformation
Also look up truncated sampling in this context

Can always use median and its estimated range
2 SE equivalent is (sorted) element $N/2 \pm N^{0.5}$

Median of 100 was 0.87, range 0.59 to 1.37
Median of 1e4 was 0.998, range 0.866 to 1.172
Median of 1e6 was 0.997, range 0.993 to 1.001

Neither estimates quite the same thing, of course

Random Number Generators

You need at least the following:

- Return U(0,1) (not int) in double precision

Also need at least 50 independent bits in number

- A long period (at least 10^{18})

Ideally, at least square of total numbers used

- Good pseudo–independence between all numbers
- And with few rare or subtle failure modes

More on how to test generators in notes

Testing Generators

Test suites: Diehard is not much good – use TestU01
http://simul.iro.umontreal.ca/testu01/tu01.html

Adjacency properties start failing surprisingly early
Always by 10^{15} numbers, sometimes by 10^{7}
Partly due to precision, but not entirely

Some popular generators fail by 2×10^{4}
This is a common cause of erroneous results
I have much better tests for adjacency problems

Common Generators (1)

Ghastly or worse: some Numerical Recipes,
gfortran rand, ISO C rand, C++ minstd_rand(0),
g++ default_random_engine,
any 32–bit or float generators

Usable but flawed: anything mod. 2^{64} or less,
($x = x \times 13^{13} + 1 \; |2^{64}|$ is probably best)
C++ ranlux48_base, C++ knuth_b

With a few weaknesses: Mersenne Twisters,
my own dprand

Common Generators (2)

Pass all tests: gfortran RANDOM_NUMBER,
C++ ranlux24_base, C++ ranlux24,
ranlux48, the better crytographic ones,
my tweaked dprand

C++ ranlux24 is slow and C++ ranlux48 worse

- But all generators have weaknesses

Including true random ones (e.g. /dev/random)

Using Multiple Generators

Different initialisations give an indication of variability

- But only due to the actual number sequence

Always re–run critical results using another generator

- Which must be based on different principles

And, if you are really cautious, try a third

Spurious results due to interactions are common
Even in generators that have passed all known tests

Parallel Sequences

This is a seriously difficult area
Most Web pages and books are wrong, here
Ask me offline if you want to know more about it

- Firstly, disjointness is not independence

It may be the best you can do, but is no more

Can generate truly pseudo–independent sequences
But it is not easy to do in a scalable fashion
And even that is only in a limited sense

Using a Common Instance

Applies only to threaded programs, not MPI
All threads call the same instance of same generator

- Don't use them unsynchronised

At best, it will be slow, and may fail horribly

Can buffer lots of them, as for I/O, synchronised
Then can extract them, unsynchronised, efficiently
Refill buffer, synchronised when needed

Using Separate Instances

Each thread or process has its own instance

Need a very long period, and use different sections
Must be a very high–quality generator

Ideally, need all sequences to be disjoint
May be possible to arrange only probabilistically
Details of how depend on details of generator

- Avoid using similar seeds (e.g. 1...N) to start
Unless the generator randomises them before use

Non-Uniform Distributions (1)

Normal (Gaussian) is the most common case
But there are too many others to describe
All convert one or more U(0,1) generators

Like special functions, but more techniques available

- And with correspondingly more failure modes

Probably no good test suite, even for common ones
It needs statistical skills and is distribution–dependent
I can describe how, but not here

Non-Uniform Distributions (2)

Be cautious with them, as they vary a lot in quality
Sensitive simulations go wrong very easily

Exactly like special functions, watch out for:

- Overall poor accuracy
- Inaccuracies in the tails
- Breaking invariants in subtle ways
- Rare, input–dependent failure

But also poor independence between numbers

Using more than one, just like basic generators, helps