

# **What's a Compiler**

# Programs are Source Code

- Matlab code
- Python code
- C++ code
- Assembly Code
- x86-64 ELF (binaries produced by gcc)
- these are all source code

# Different Source Languages are for Different Audiences

hello.asm

```
SECTION .data
msg: db "Hi World",10
len: equ $-msg
```

```
SECTION .text
global main
```

main:

```
mov.   edx,len
mov.   ecx,msg
mov.   ebx,1
mov.   eax,4
int.   0x80
mov.   ebx,0
mov.   eax,1
int.   0x80
```

00000000	7F 45 4C 46	01 01 01 00	00 00 00 00	00 00 00 00	.ELF.....
00000010	02 00 03 00	01 00 00 00	80 80 04 08	34 00 00 00	.....4...
00000020	F8 00 00 00	00 00 00 00	34 00 20 00	02 00 28 00	.....4. ....
00000030	05 00 04 00	01 00 00 00	00 00 00 00	00 80 04 08	.....
00000040	00 80 04 08	A2 00 00 00	A2 00 00 00	05 00 00 00	.....
00000050	00 10 00 00	01 00 00 00	A4 00 00 00	A4 90 04 08	.....
00000060	A4 90 04 08	09 00 00 00	09 00 00 00	06 00 00 00	.....
00000070	00 10 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
00000080	BA 09 00 00	00 B9 A4 90	04 08 BB 01	00 00 00 B8	.....
00000090	04 00 00 00	CD 80 BB 00	00 00 00 B8	01 00 00 00	.....
000000A0	CD 80 00 00	48 69 20 57	6F 72 6C 64	0A	....Hi World.

# Compilers are Source Transformers

- Translator from one source format into another
- Typically from a high-level (easy to read/write) language into a low-level (easy for machine to interpret)
- Efficient program in language A, after undergoing transformation, may no longer be efficient in language B!

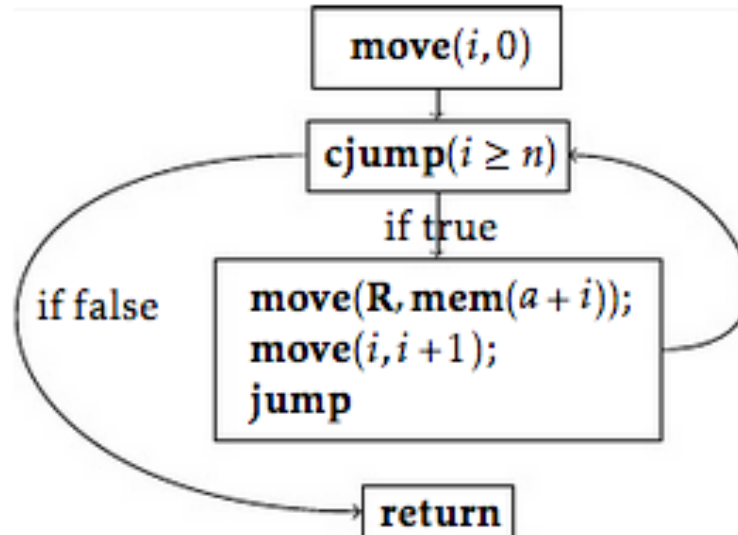
# Optimizing Compilers

# Inefficiency detection

- Compilers perform **static analysis** on either source or some intermediate program
- Most common technique revolves around a cause-effect system known as **dataflow analysis**
- In dataflow analysis, you form a correspondence with an abstract analysis domain, and instructions are abstract functions.
- When the domain satisfies certain order theoretic properties, the least fixed point of the composition of the abstract functions is computable, and represents an overapproximation of the correct analysis
- In general it's impossible to derive true properties

# Dataflow Framework

Figure 4: Another CFG whose nodes are Basic Blocks



Each box represents a function; arrows denotes the flow in analysis information throughout the program, this then reduces to a linear system of “flow-valued” equations. Note: while loops are recursive, so we need the ability to take fixed points of this system.

# To Learn More

If you want to learn more about compilers/programming

- CS4120/4121 - Compilers
- CS4110/6110 - (Advanced) Programming languages
- CS61xx - courses related to programming languages
- CS7190/92 - PL Seminar and Program Refinement Seminar
- A gentle introduction to program analysis <http://www.scribd.com/doc/112917241/Dataflow-Odyssey>



# Common Optimizations

- Data-cache optimization
- Procedure inlining
- Loop-unrolling
- Constant/copy propagation
- Partial redundancy elimination
- Induction-variable optimizations
- Dead-code elimination
- Loop-motion invariant code hoisting
- Common subexpression elimination
- Constant folding/algebraic simplification
- etc...

# Modern Compilers are Smart

- Compiler design has come a long way
- Most experts agree that even low-effort optimizing compilers do better jobs than hand optimization
- Unfortunately, compilers cannot make clever algebraic reductions: **no order of magnitude decrease in flops**
- Using -OInfinity is not cure for poorly designed algorithms (well, maybe one day)

# Spot the Difference

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            for (int k = 0; k < n; k++)  
                c[i][j] = c[i][j] + a[i][k]*b[k][j];  
}
```

```
void MultiplyMatrices_2(int **a, int **b, int **c, int n){  
    for (int i = 0; i < n; i++)  
        for (int k = 0; k < n; k++)  
            for (int j = 0; j < n; j++)  
                c[i][j] = c[i][j] + a[i][k]*b[k][j];  
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

[c](#) [algorithm](#) [matrix](#) [matrix-multiplication](#) [gprof](#)

[share](#) | [edit](#) | [flag](#)

edited Sep 13 '11 at 2:47



[templatetypedef](#)

119k ● 23 ● 273 ● 485

asked Sep 13 '11 at 0:29



[kevlar1818](#)

1,341 ● 4 ● 25

[add comment](#)

# Constant Propagation/Dead Code

```
a = 7
b = 3
c = a + b
if c == 4:
    print "Fail"
else:
    print "Yay"
```

=>

```
c = 7 + 3
if c == 4:
    print "Fail"
else:
    print "Yay"
```

- at the point  $c = a + b$ , **a** and **b** must be 7 and 3
- we can propagate these constants
- after propagation, **a** and **b** no longer “live”
- we remove “dead” (not “live”) variables

# Constant Folding

```
c = 7 + 3
if c == 4:
    print "Fail"
else:
    print "Yay"
```




```
c = 10
if c == 4:
    print "Fail"
else:
    print "Yay"
```

The expression  $7+3$  is always 10, so we can replace it with 10.

# Constant Folding/Dead Branch

```
c = 10
if c == 4:
    print "Fail"
else:
    print "Yay"
```



```
print "Yay"
```

- Another constant propagation of `c` into `10` gives a conditional (`10 == 4`)
- Constant folding again replaces it with `False`
- Therefore, the true branch is a dead branch, and can be eliminated entirely

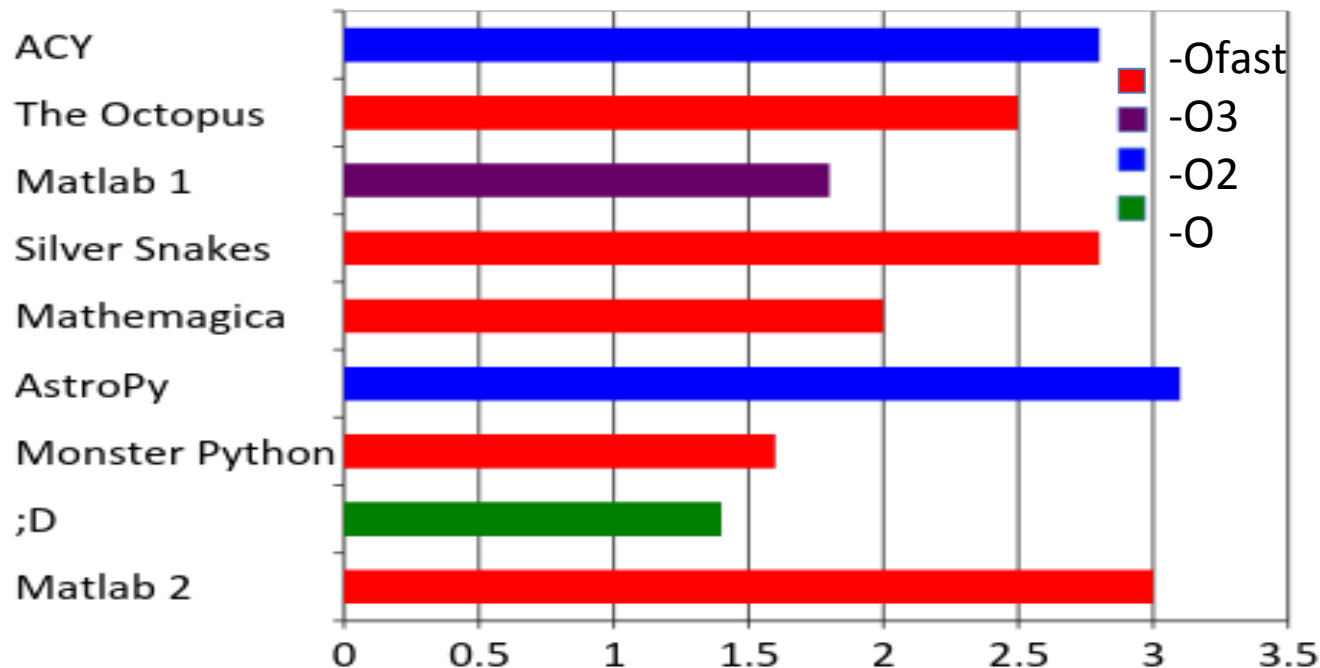
# **GCC Optimization Flags**

- `-g` - add debug information
- `nothing` - default
- `-funroll-loops` - loop unrolling
- `-O` - level 1
- `-O2` - level 2
- `-O3` - level 3
- `-Ofast` - level 3 + unsafe optimization

We can form an ordering on these flags, `-O` contains everything `nothing` does, and `-Ofast` contains everything

# Effects of optimization: Speed

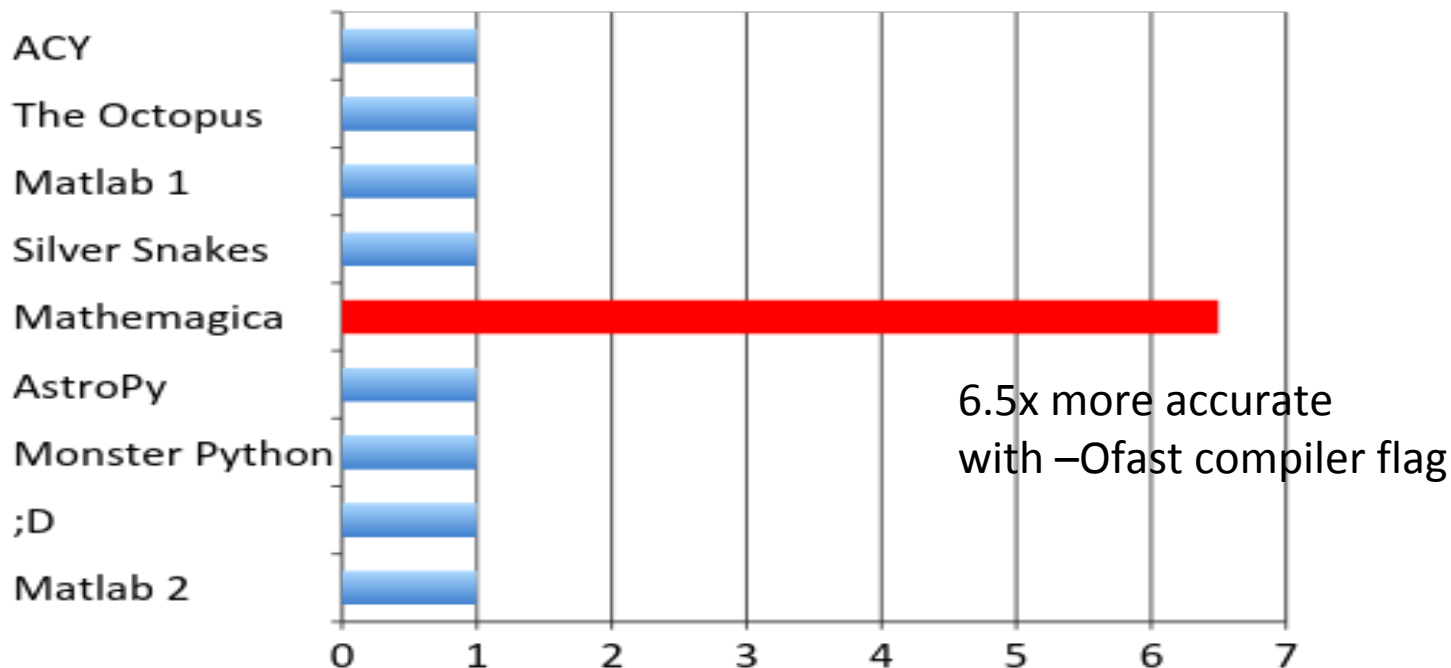
(default time/fastest optimized time)



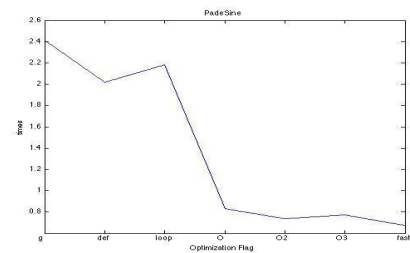
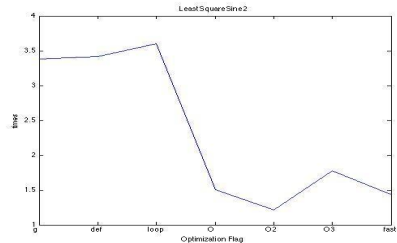
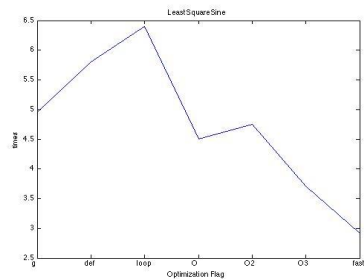
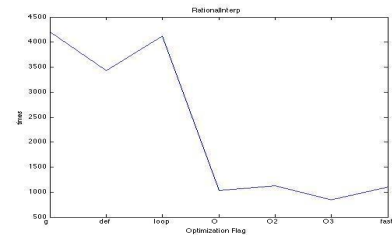
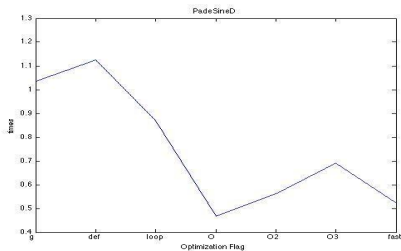
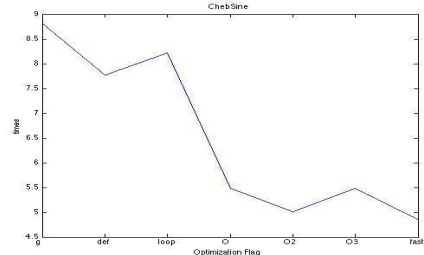
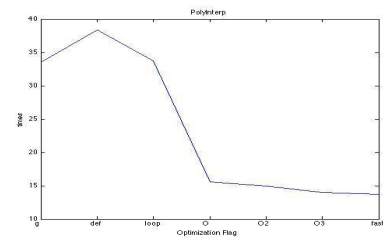
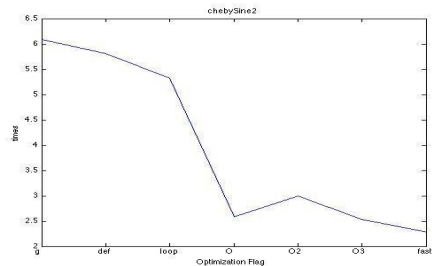
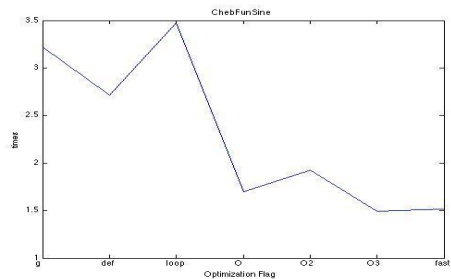


# Effects of optimization: Error

(default error/smallest optimized error)



# Speed vs flag plots

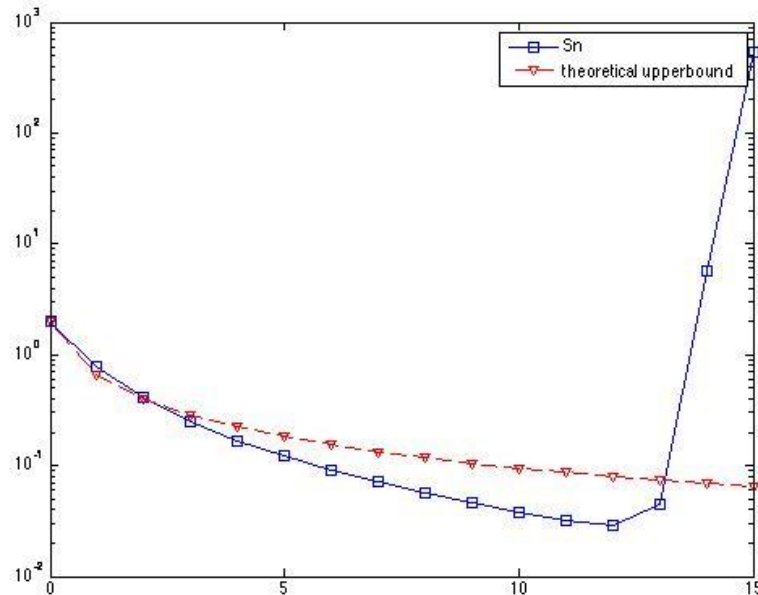


# Accuracy & Optimization

- Our group (Mathemagica) is the only group for whom - Ofast ended up giving different results than the safe flags
- We hypothesize that this increase of accuracy is created because of the violation that floating point arithmetic is not associative!
- We documented our journey converging towards this hypothesis.

# Difficulties in evaluating Inner Products

$$S_k = \int_0^\pi \frac{x^n}{\pi} \sin(x) dx \implies S_k = 1 - \frac{2}{\pi^2} \binom{2n+1}{2} S_{k-1}, S_0 = 1$$



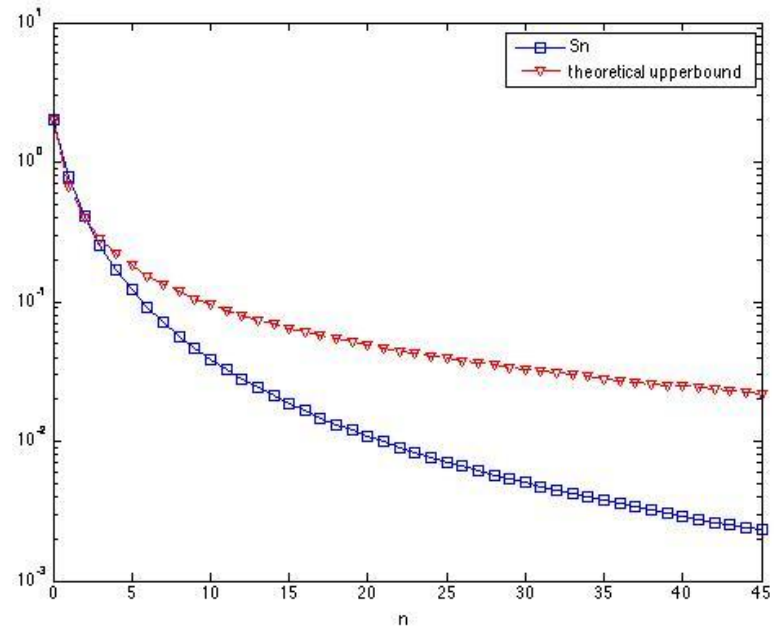
???

Clever, but not stable...

# A stable series

$$\Rightarrow S_k = \left| \sum_{\Delta=1}^{\infty} (-1)^{\Delta} \frac{\pi^{2\Delta}}{\left[ \frac{2(k+\Delta)+1}{2\Delta} \right]} \right|$$

For the full derivation, see <https://www.dropbox.com/s/6tvombinbixp6jx/lsg2.pdf>



# Finding the “bug”, attempt 2

- Turns out the round off of the evaluation of the inner product isn't the culprit, since we can make it arbitrarily close to machine precision
- What about some other part of our code?

evaluate

$$\langle L_n, L_n \rangle = \int_{-\pi}^{\pi} L_n \left( \frac{x}{\pi} \right)^2 dx = \frac{2\pi}{2n+1}$$

where

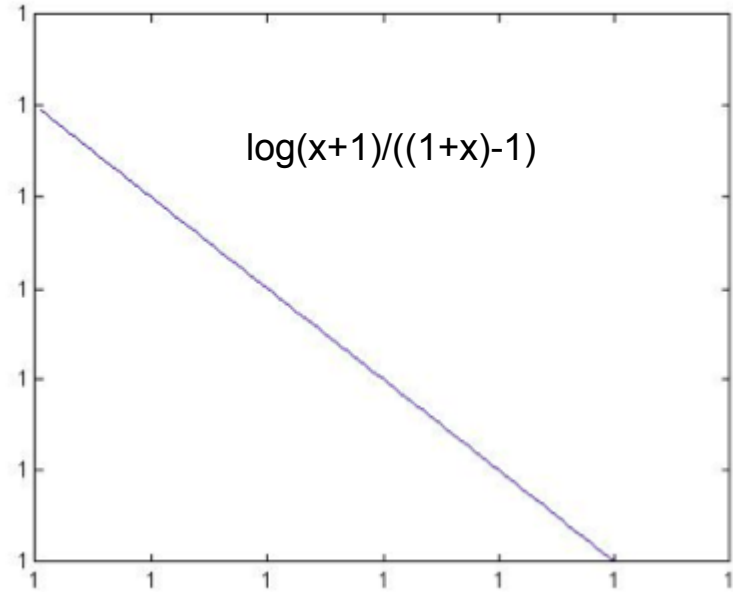
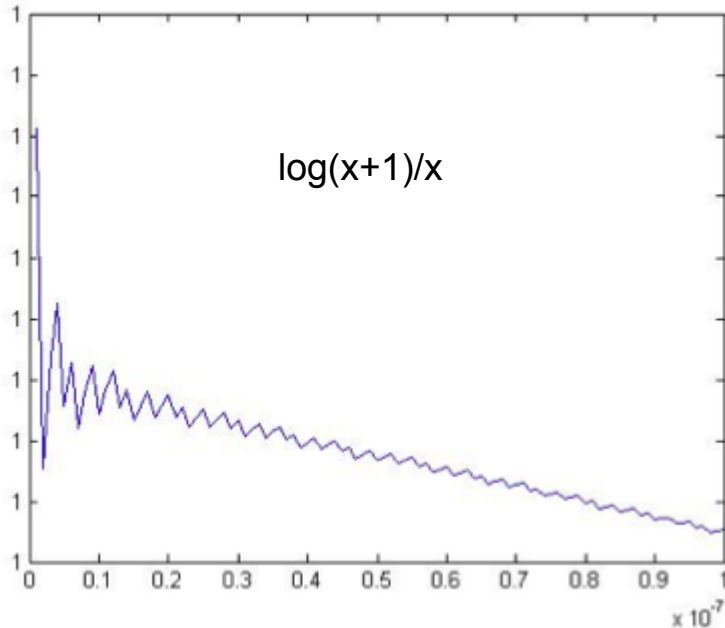
$$L_n(x) = \frac{(2n-1)xL_{n-1}(x) - (n-1)L_{n-2}}{n}$$

$$\frac{L_n(x)}{\langle L_n, L_n \rangle}$$

- However, this method isn't stable either. Sure, the value of  $\langle L[n], L[n] \rangle$  is perfect, but  $L[n](x)/\langle L[n], L[n] \rangle$  diverges!
- Problem: relative round off error is additive when we multiply. When we compute the polynomial using Horner's rule, most of the error comes from the amplification of the round off error in the first few round; subsequent error is negligible. Therefore,  $L[n](x)$  has large error, but  $\langle L[n], L[n] \rangle$  does not.
- Solution, compute  $\langle L[n], L[n] \rangle$  lossily, also using Horner's rule!
- Method is stable now :) but the optimization-induced bug is still there!
- (Note: we originally computed the coefficient of our polynomial explicitly, and evaluated it, we found the resulting method to be divergent: this is the fix.)

# Unintuitive Cancellation

We can demo this phenomenon with the computation of  $\log(1+x)/x$  around 0



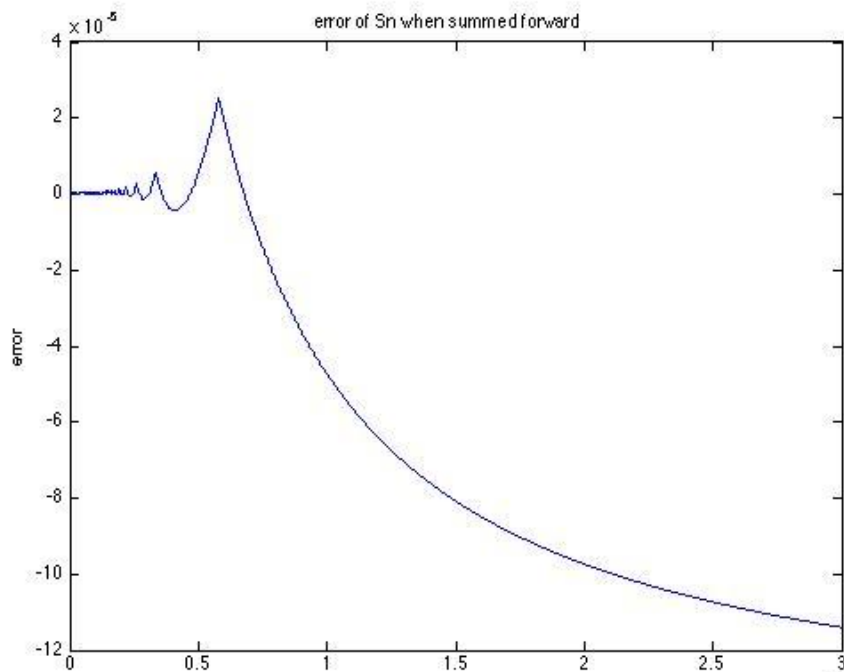


# Finding the “bug”: third time’s the charm

- This phenomenon is a combination of loop unrolling and algebraic simplification (using `-ffast-math`)
- In our original scheme for evaluation of the polynomial, we proceeded linearly in  $k$ :  $P(x) = \sum(k, \text{coef}[k] * L[k](x))$
- The coefficients are all decreasing and alternating at an exponential rate
- Since double precision only stores 16 digits, if you add  $x$  to a number much smaller than it, we are going to lose digits. At the  $k$ th iteration, we add a number around 1 to a number around  $10^{-k}$ , so we lose  $k$  places. After 16 iterations, all precision is lost.
- It turns out that computing the series backwards is stable. At the  $k$ th iteration, we add some number around  $10^{-\{k-1\}}$  with an number around  $10^{-k}$ , so we only lose one place overall.

# The unintuitive summation

Consider  $S_n = \sum_k^n \frac{1}{k(k+1)} = 1 - \frac{1}{n+1}$



The forward error is around  $1e-4$ , the backwards error is around  $1e-8$ . That's four magnitudes of difference!

# Solution

After loop unrolling, -ffast-math reordered the sequence of computations within the evaluation of the polynomial so that we're no longer just adding large numbers with progressively smaller numbers every time, hence improving accuracy.