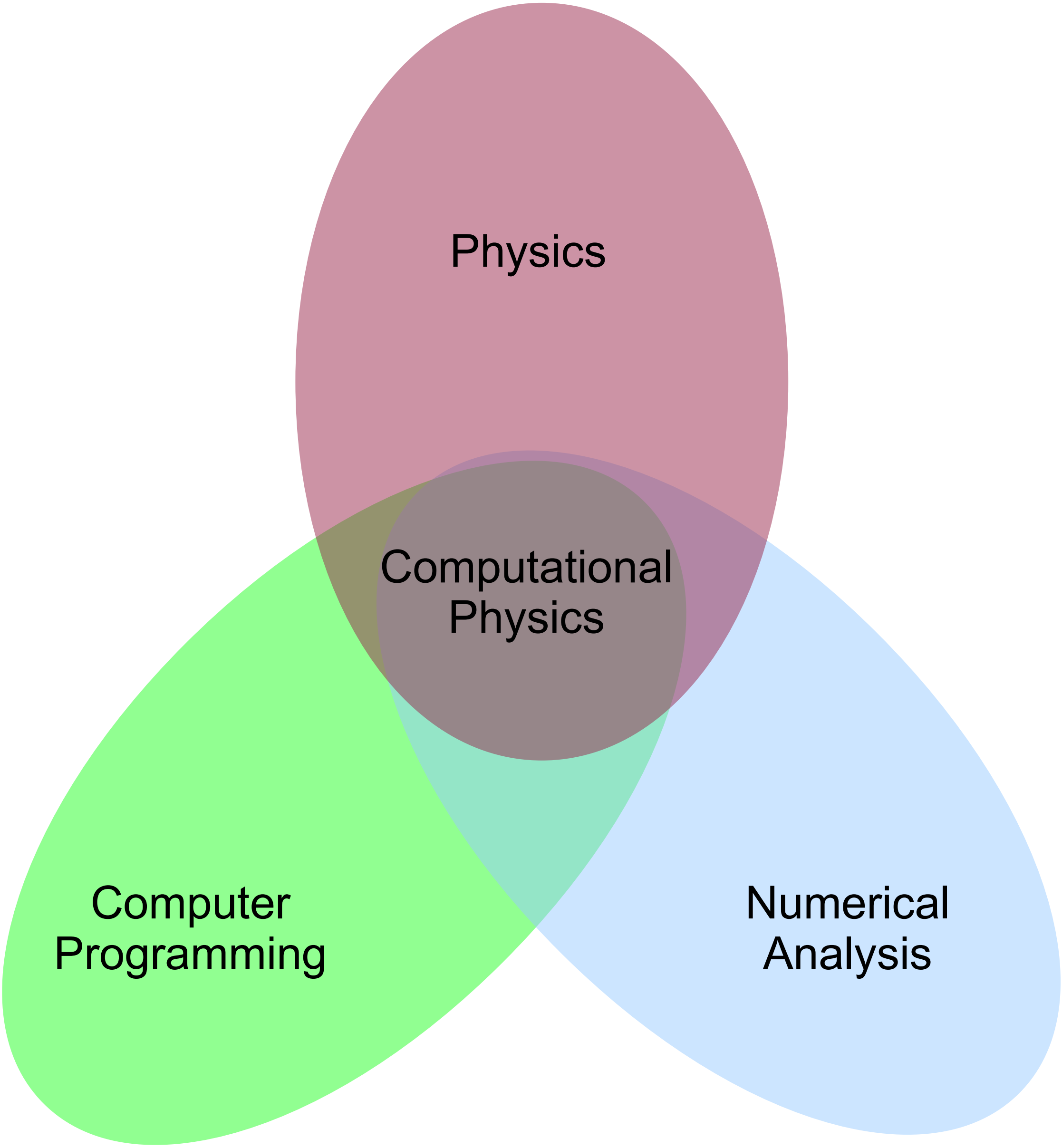
Physics 3340
Computational Physics
Fall 2017
Dr. John Fattaruso
http://www.physics.smu.edu/fattarus/

Dr. John Fattaruso
Background

- Ph.D. Electrical Engineering, U. C. Berkeley; minors Electromagnetic theory, Statistics
- ~22 years at Texas Instruments; Analog circuit and solid state device design; Distinguished Member of the Technical Staff
- ~40 years of numerical programming in machine languages, Fortran, C, C++, Java

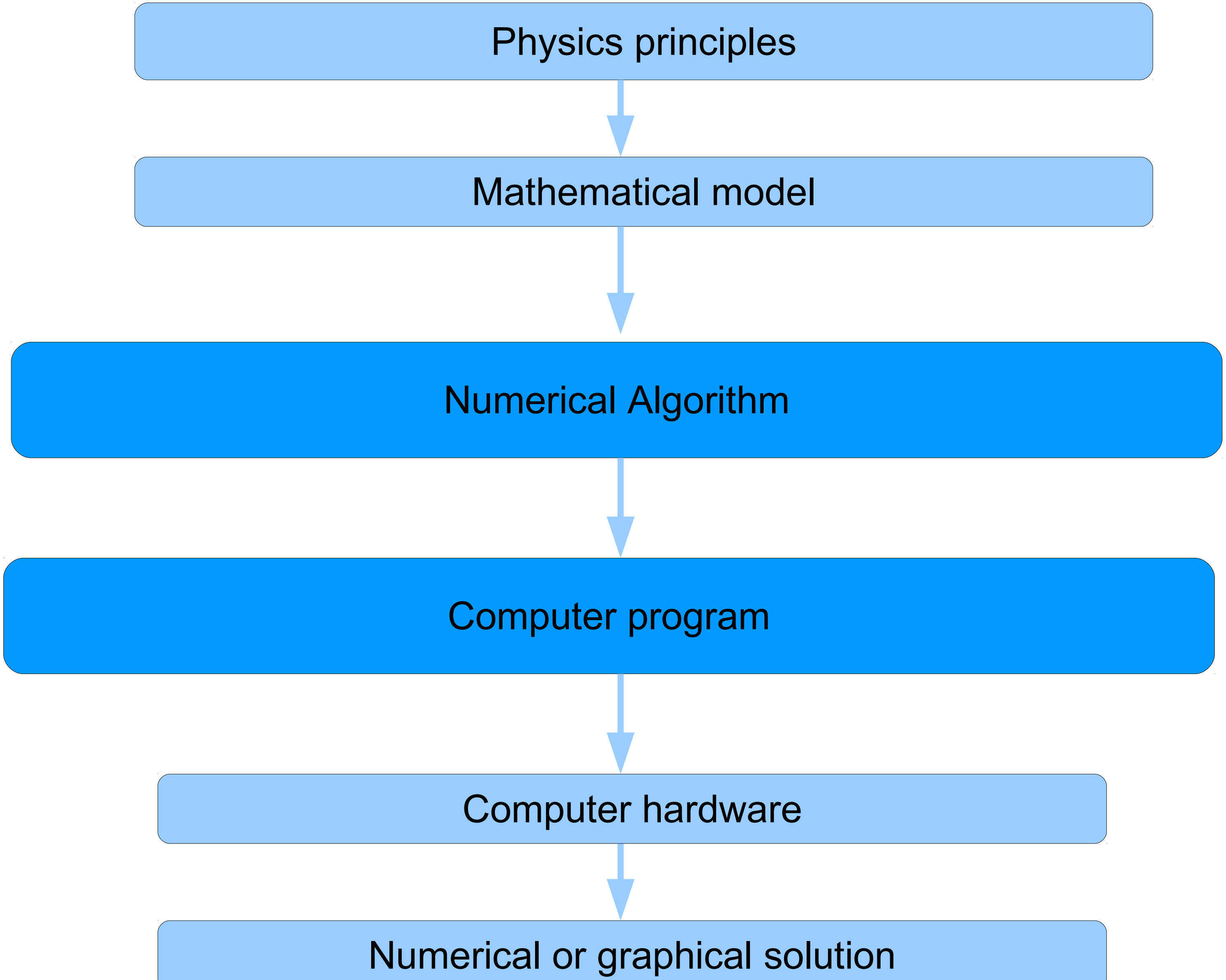
What Type of Course is Physics 3340?



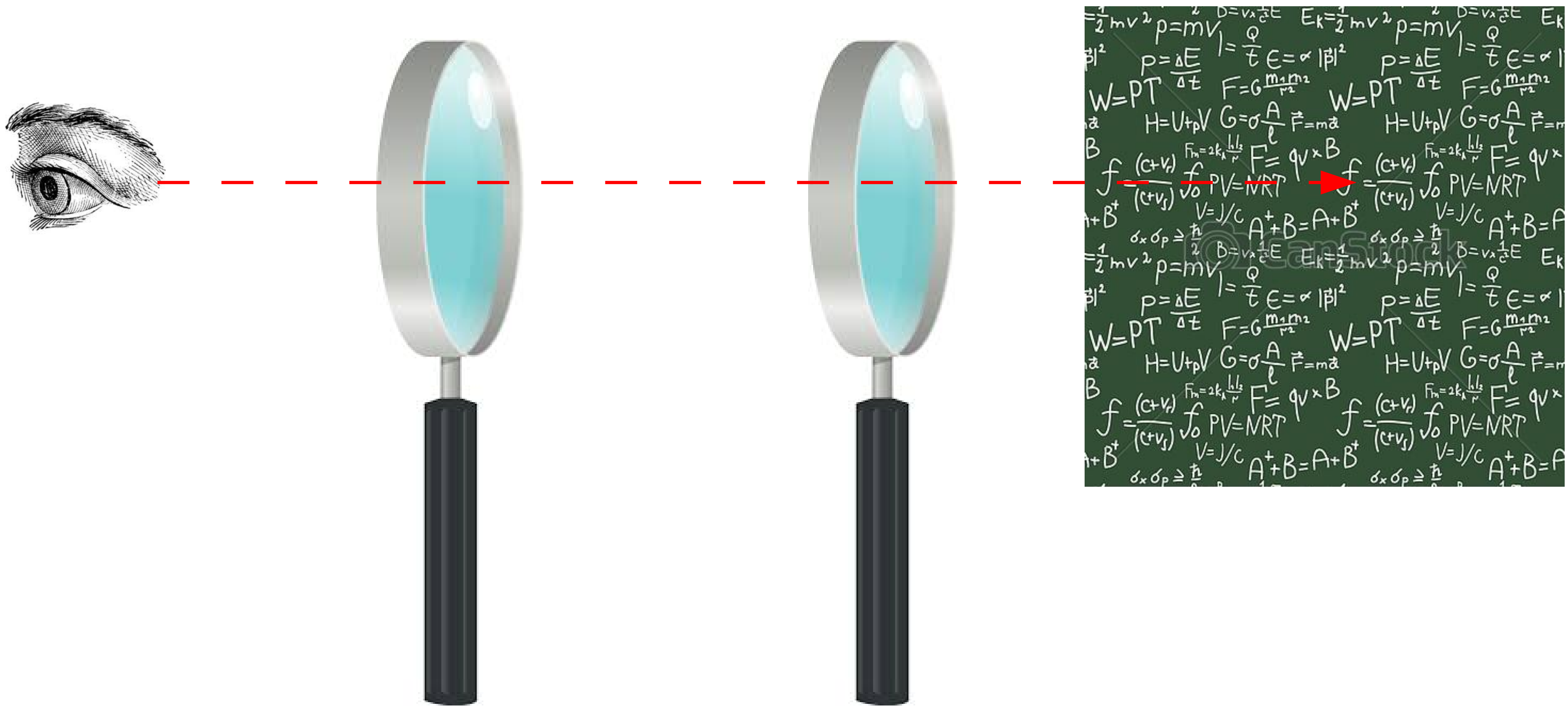
Contributors to Early Numerical Analysis

- Newton
- Gauss
- Lagrange
- Euler
- Legendre

Where Does Numerical Analysis Fit In?



Solving Physics Problems Numerically



Computer Hardware

Numerical Algorithm

Each of these tools has its own properties and limitations that must be understood

Why Study Numerical Analysis Down at C Level?

- Prepackaged tools like *octave* or prepackaged libraries like *gsl* don't always do just what you want
- All numerical solutions involve tradeoffs between accuracy, convergence, computation speed; these tradeoffs should be understood with the fundamental algorithms before using prepackaged tools
- “Never in the history of mankind has it been possible to produce so many wrong answers so quickly” - Carl Erik Fröberg
- Computational speed is like closet space; you'll always need more than you have

Why Program Numerical Code in C?

- C is a ubiquitous language, compiled for any processor and universally understood by programmers
- Languages such as Java, Tcl/Tk or Perl, and higher level analysis tools such at Matlab or Octave can link to low level functions coded in C
- Numerical programs tend to be predominantly computation intensive around iterative loops, rather than having to deal with large scale data structures or inherited object hierarchies
- Data file formats tend to be predominantly columnar numerical data, appropriate for standard I/O routines in C

Class Outline

- Introduction to Linux and numerical programming in C
- Visualization of numerical data with gnuplot
- Roots of nonlinear equations
- Solutions of systems of linear equations
- Solutions of systems of nonlinear equations
- Monte Carlo simulation with pseudorandom numbers
- Interpolation of sparse data points
- Numerical integration
- Solutions of ordinary differential equations
- Boundary value problems

Class Grading

- Weekly assignments 30%
- Two midterms 20% each
- Final project 30%

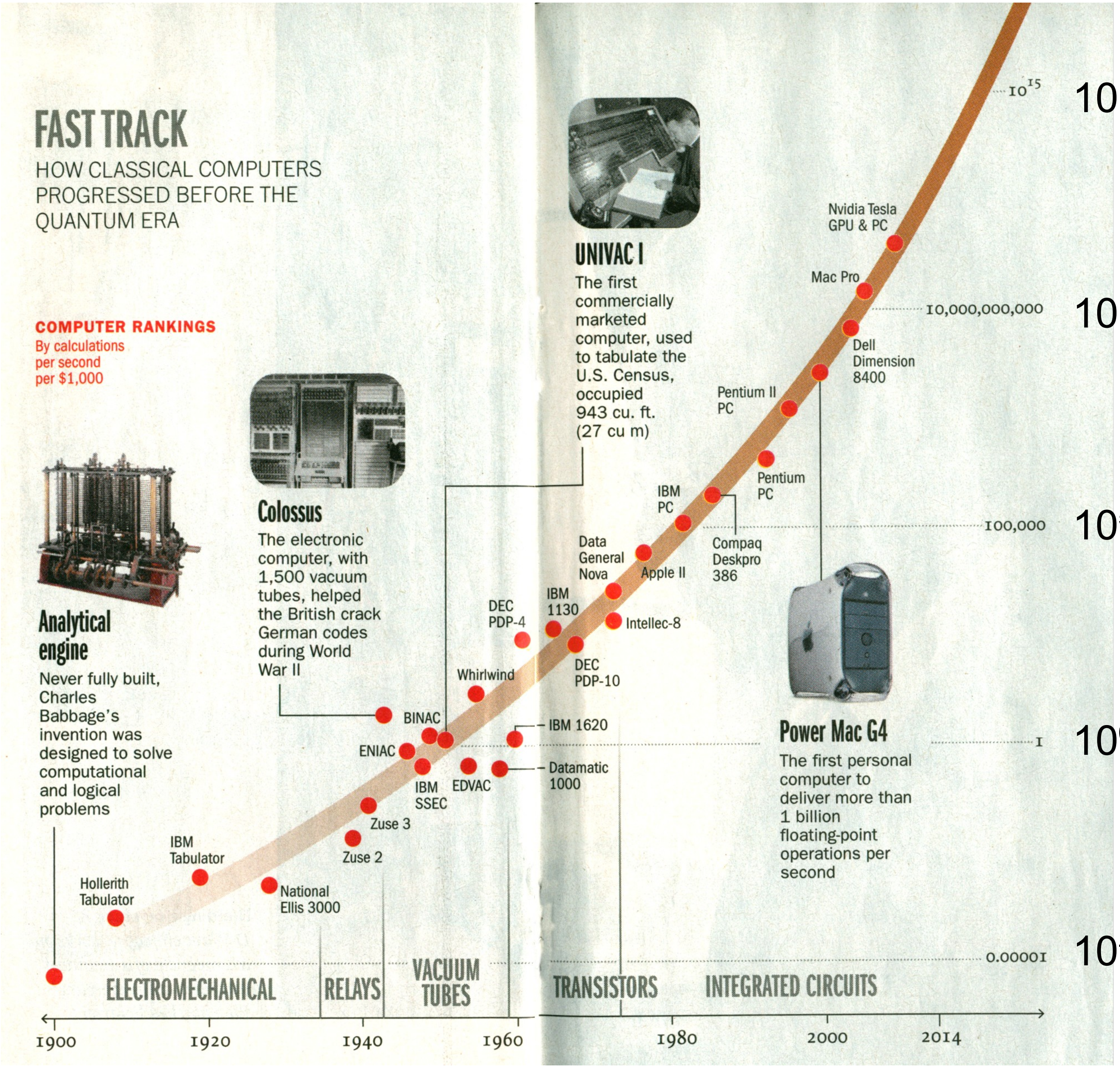
- Homework assignments posted on Canvas on Mondays, due the following Monday on Canvas by date and time deadline
- All source code must be strictly authored by each student

Class Standards

- You may:
  - Collaborate with fellow students deciding on general approaches to assignment problems
  - Help debug each other's programs

- You may not:
  - Copy lines of code directly from another student's programs
  - Copy another student's assignment file

History of Computational Economy



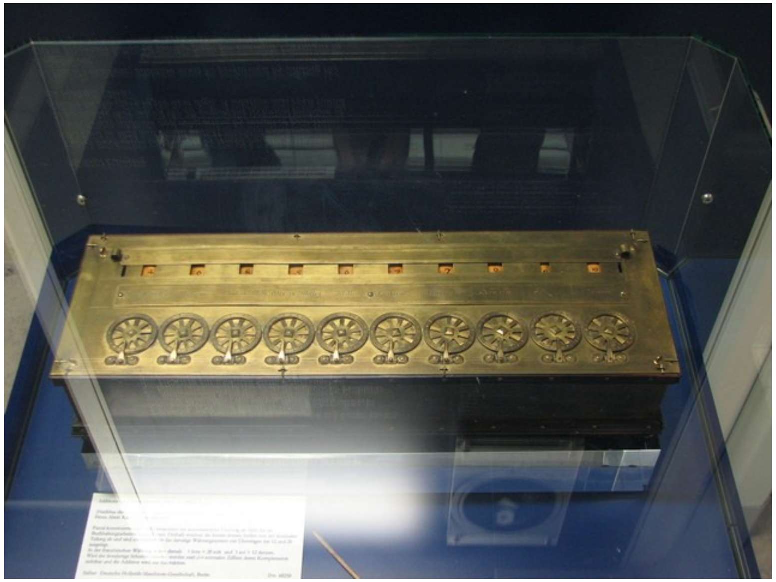
10^{15}
10^{10}
10^{5}
10^{0}
10^{-5}

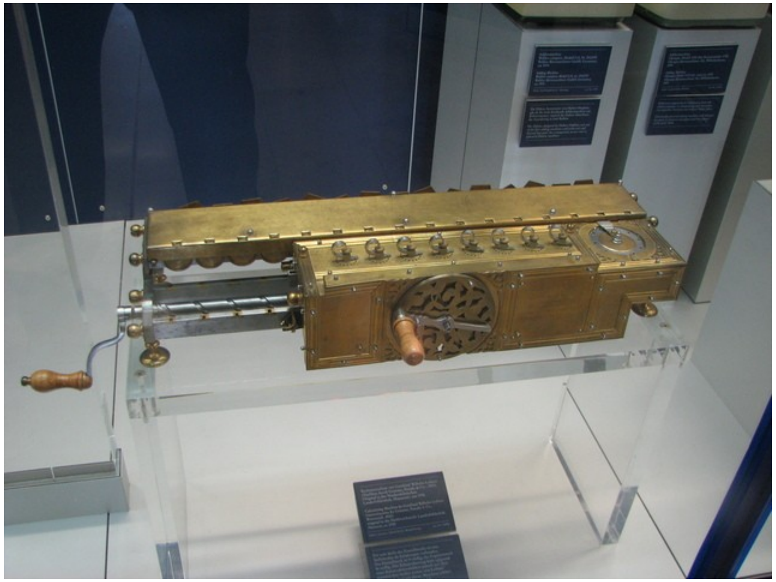
Calculations per second per $1,000

(Time magazine, 17 Feb 2014)

1900 1920 1940 1960 1980 2000 2014

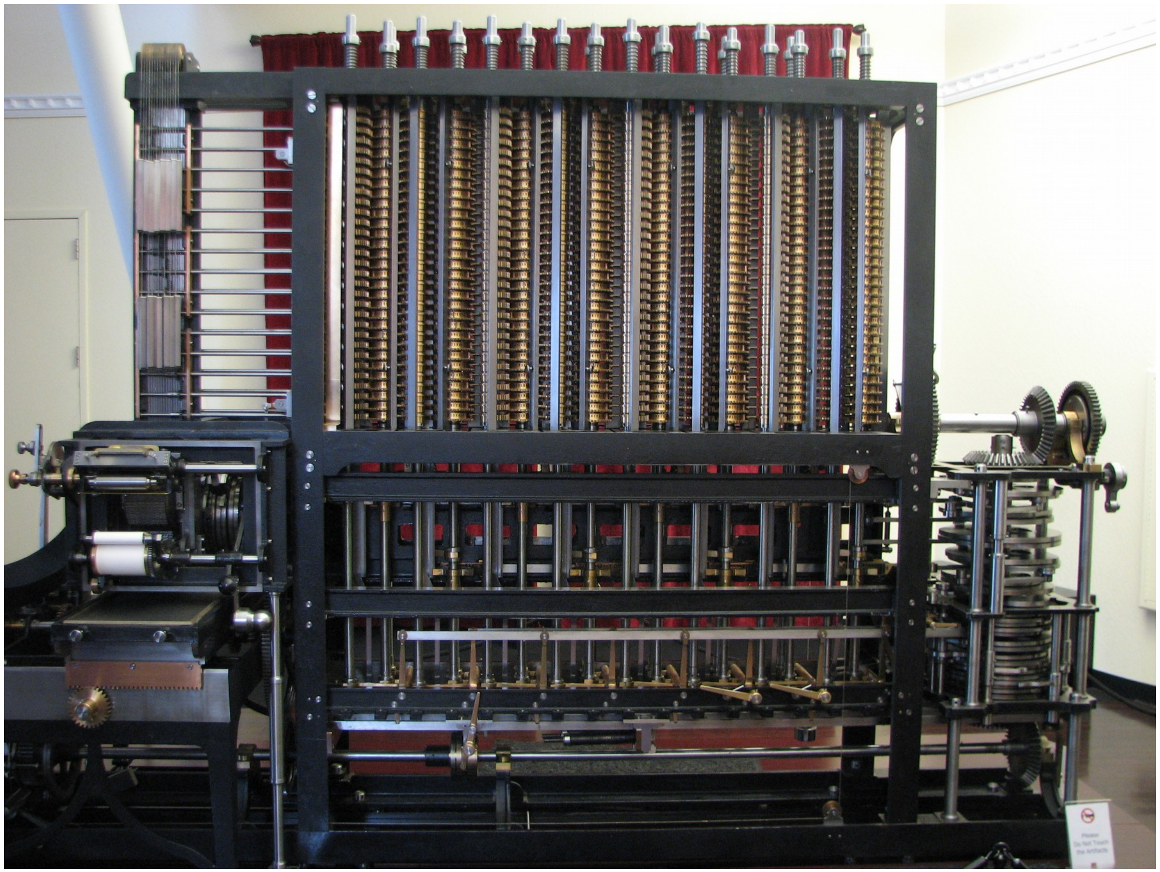
Historical Computing Devices





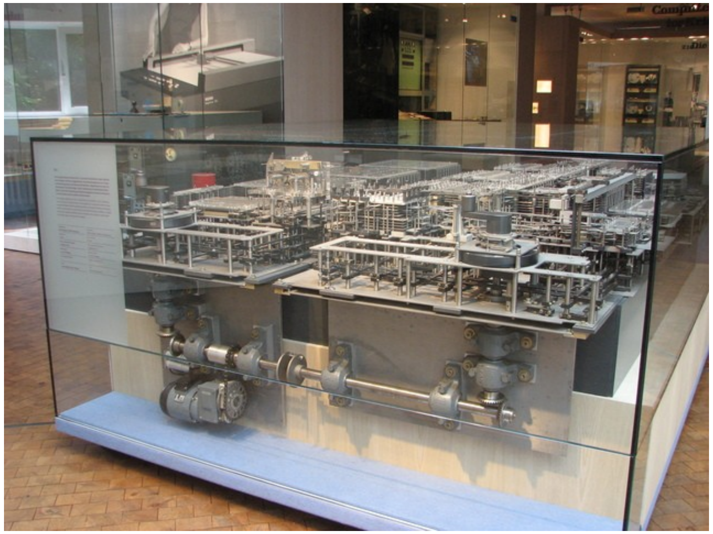
(Displayed at the Deutches Museum of Science and Technology, Munich, Germany)

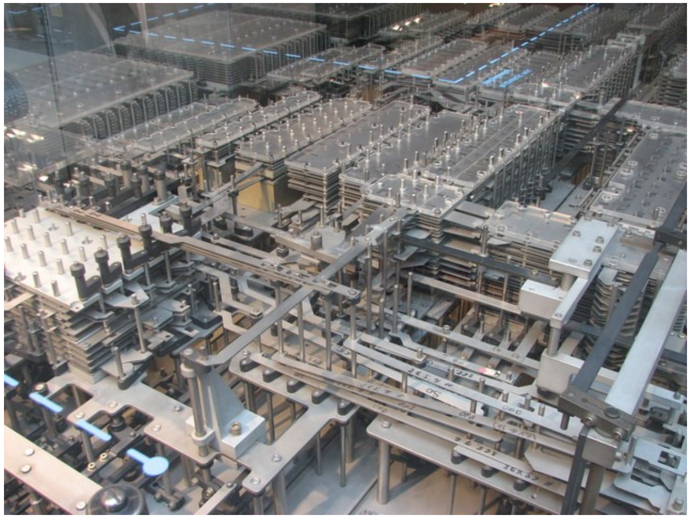
Babbage “Difference Engine” (circa 1850)



(Reconstruction, displayed at the
Computer History Museum, San Jose, CA)

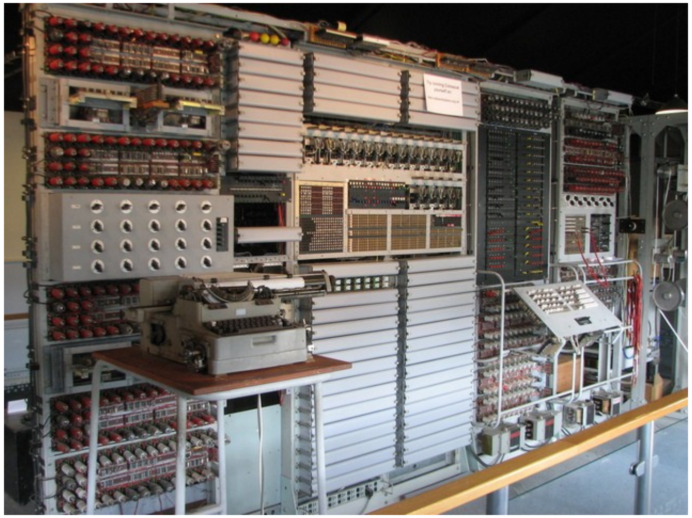
The Zuse ZI (circa 1935)

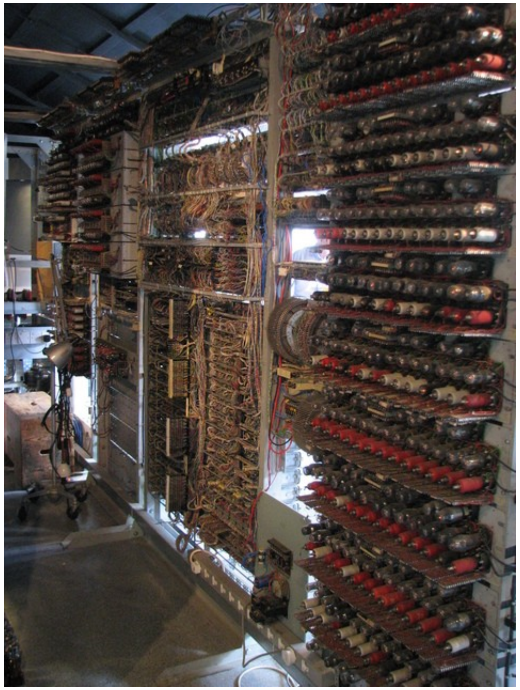


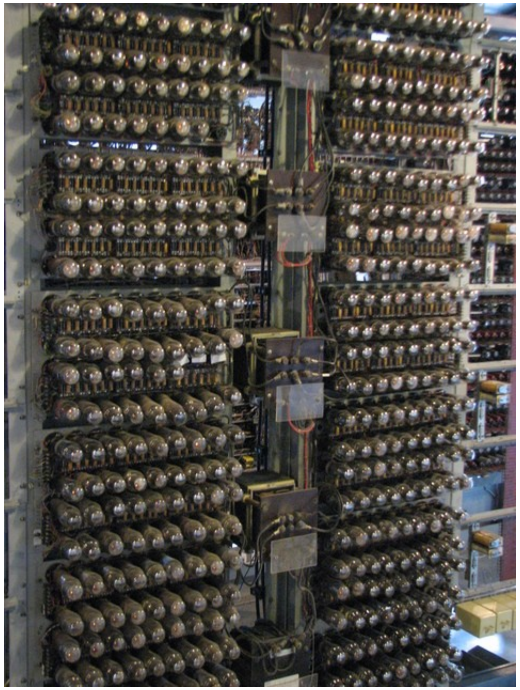


(Reconstruction, displayed at the German Museum of Technology, Berlin, Germany)

The Colossus (circa 1940)

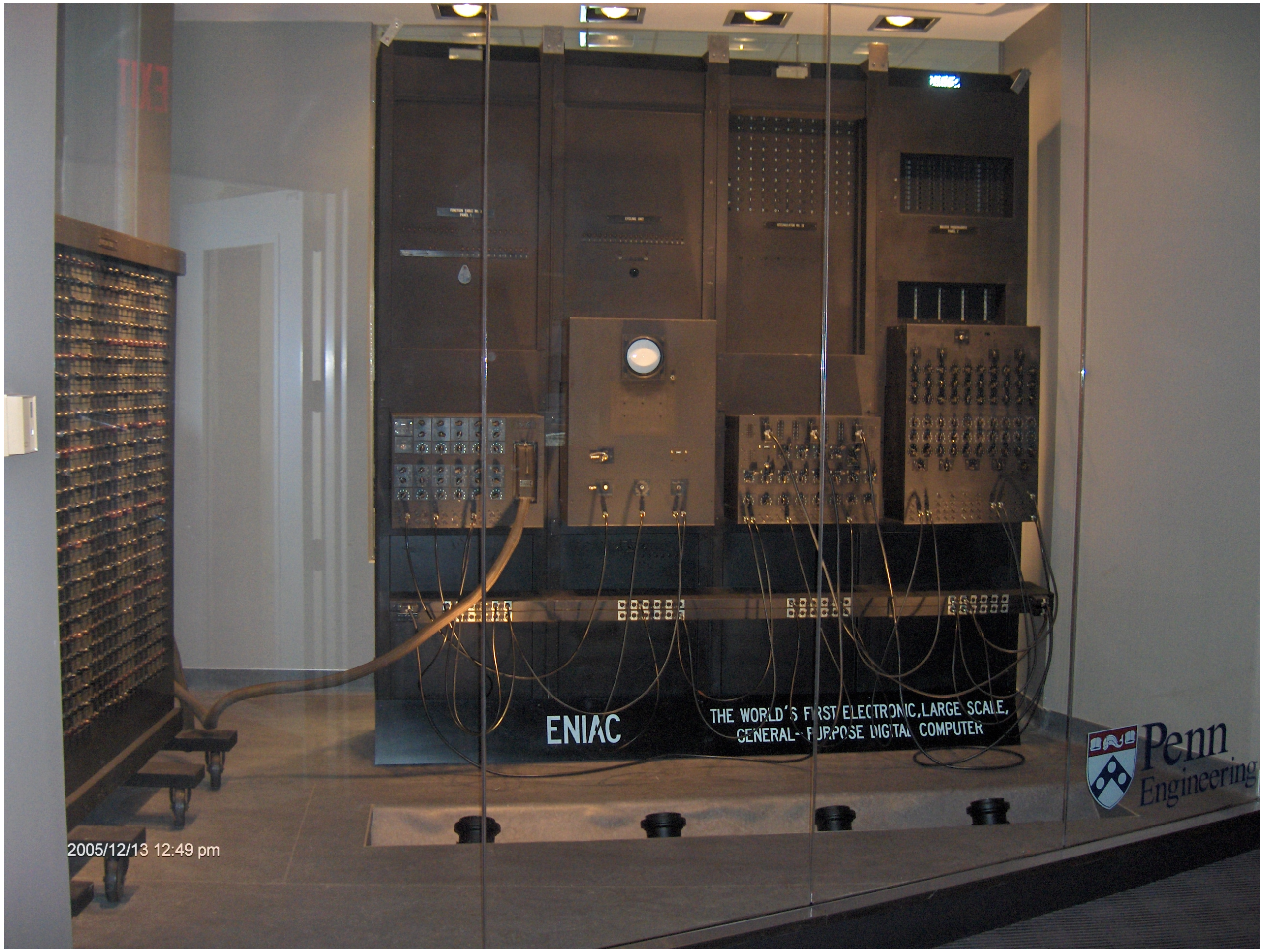


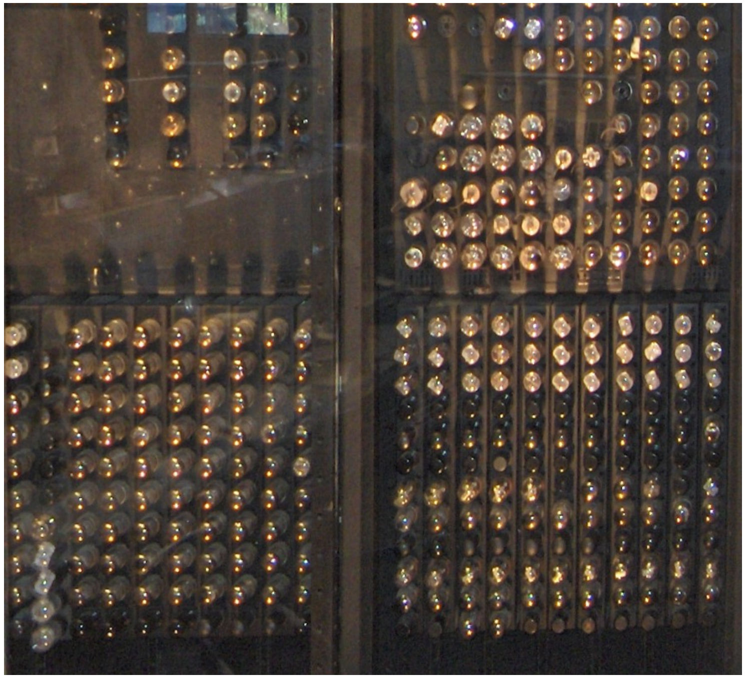




(Reconstruction, displayed at the National Museum of Computing, Bletchley Park, England)

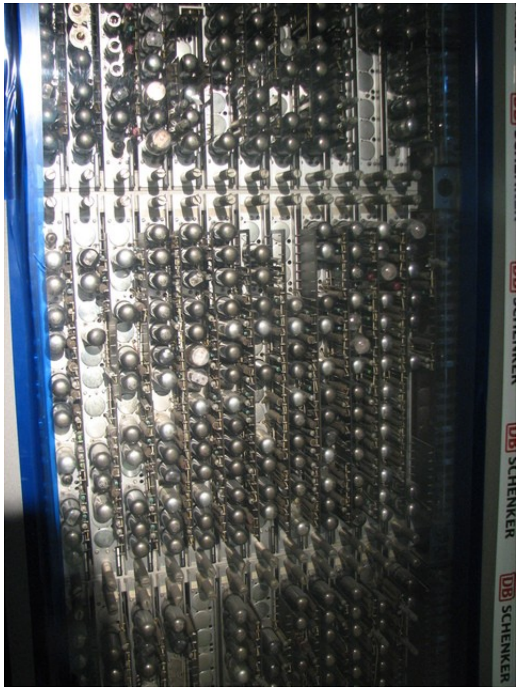
The ENIAC (circa 1943)

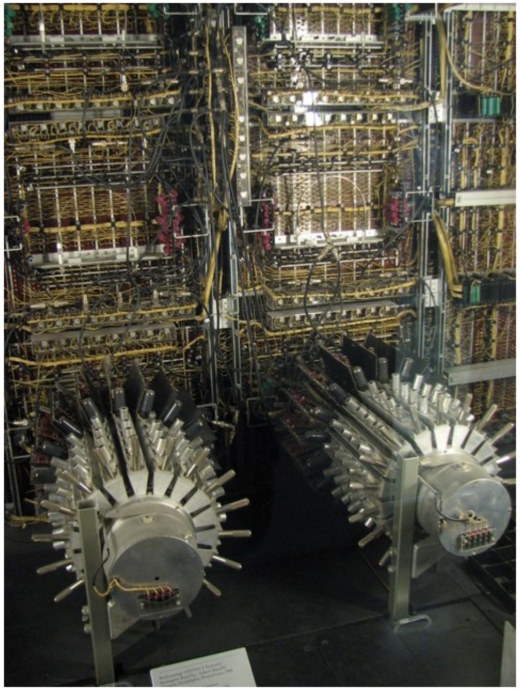




(Displayed at the University of Pennsylvania and
at the Smithsonian Museum, Washington D.C.)

The UNIVAC I (circa 1950)





(Displayed at the Deutches Museum of Science and Technology, Munich, Germany)

The IBM 1620 (circa 1960)

The Control Data Corporation 6600

Traditional Computer Terminal - the ASR33

CRT Computer Terminal - the ADM3A

Some Remnants of Mainframe/Terminal Technology in the C Language and the Windows and Unix/Linux OSs

Separate “Carriage Return” and “Newline” characters for EOL in Windows date back to driving separate motors in teletype terminals; Unix uses single “Newline”

Pseudo terminals can be started and run in separate windows, resembling CRT terminal screens

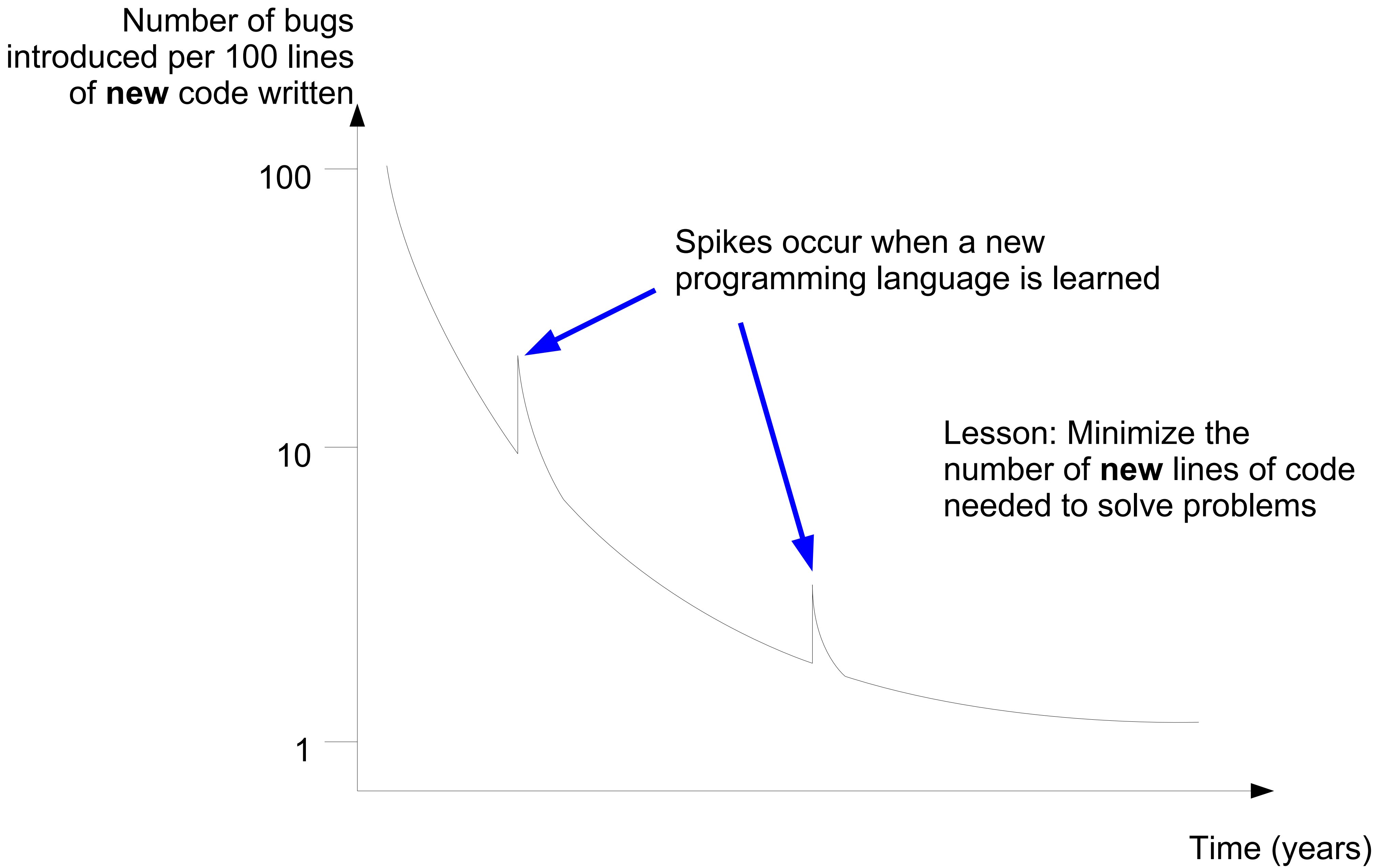
'Print working directory' command `pwd` displays on a terminal, as if it were a teletype

Tape archive command `tar` is used to bundle and compress any number of files

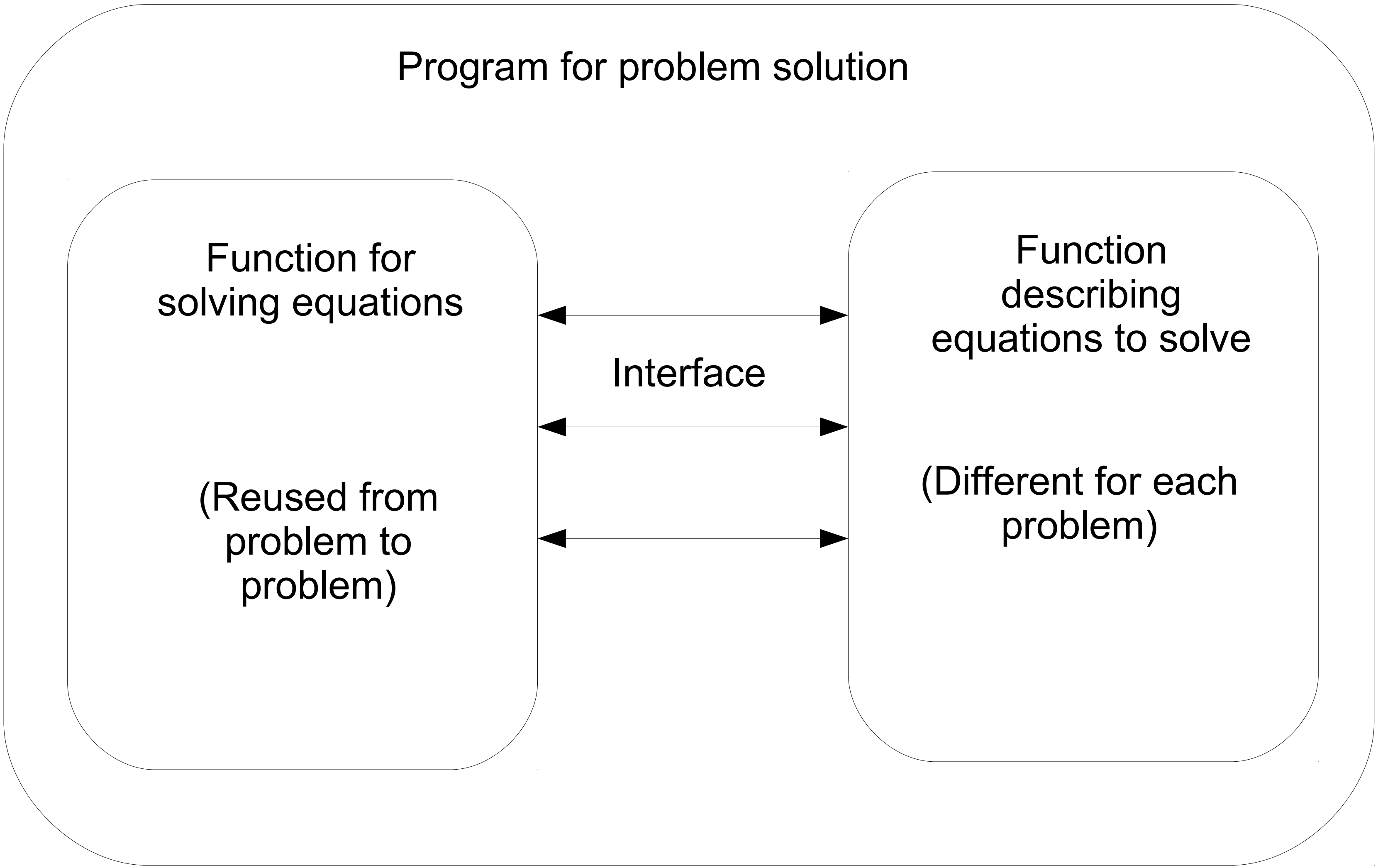
Formatted print library function named `printf` displays on a terminal, as if it were a teletype

'Break' key on keyboards traces back to a 'Break' key on the teletype that would open serial terminal line and signal for attention

Programming Learning Curve



Reusing Program Code



Use of Fonts in Class Slides

- Descriptive text is in Arial font
- Text intended for program code or example computer output is in `Courier` font
- A symbolic label intended to have an actual name substituted for it is in *Times Roman Italic* font
- Examples:
  - This is a slide
  - `x = 42.0 * sin(theta);`
  - Read file *file_name*

Employing Computational Physics

- Beyond basic problems, many problems in the real world do not admit analytic solution
- Nonlinearities in physical system, described by transcendental or other strongly nonlinear equations that do not admit analytic solution
- Complexity that gives systems of simultaneous equations of too high an order for analytic solution
- Visualization of mathematical dependencies, even for linear or simple systems
- Verification of complex analytic solutions

Class Progress

Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
  (Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
  (Midterm 2)
Solutions of ordinary differential equations

“Pseudocode” in Cheney and Kincaid

Declare variables and their types

integer $i, n;$ **real** $x, y;$ **real array** $(a_i)_{0:n}$

Array notation means n+1 real values indexed as a_0 through a_n

⋮

for $i = 0$ **to** n **do**

⋮

end for

Iterative 'for' loop: body of loop is executed iteratively n+1 times with the variable i assigned the values 0, 1, 2 , ... , n-2, n-1, n in sequence

⋮

$x \leftarrow 1.0$

⋮

while $x > 0.0$ **do**

⋮

end while

Iterative 'while' loop: body of loop is executed iteratively while the condition is true

$\leftarrow$ is the 'assignment' operator:

the value of the expression on the right is assigned to the variable on the left.

"Pseudocode" in Cheney and Kincaid

if $i = 0$ **then**

$\vdots$

else if $i = 1$ **then**

$\vdots$

else

$\vdots$

end if

$\vdots$

output $i\,, x$

Code to execute if the value of *i* is 0

Code to execute if the value of *i* is 1

Code to execute otherwise

Output the values of *i* and *x*

Example: Numerical Evaluation of Polynomials

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n = \sum_{i=0}^{n} a_i x^i$$

Naive method of computation would be to evaluate the *i*th term separately as

$$a_i \cdot x \cdot x \cdot x \cdot \ \cdots \ \cdot x$$

And then sum all n+1 terms. This requires *n* additions and

$$0 + 1 + 2 + 3 + \cdots + (n-1) + n \ = \ \frac{n(n+1)}{2}$$

multiplications

Polynomial Evaluation in Pseudocode

Declare variables and their types

n, x and a_i are assumed to have assigned values

$$\textbf{integer } i, n; \ \textbf{real } p, x; \ \textbf{real array } (a_i)_{0:n}$$

Array notation means n+1 real values indexed as a_0 through a_n

Code statements to execute in sequence

$$p \leftarrow 0.0$$
$$\textbf{for } i = 0 \textbf{ to } n \textbf{ do}$$
$$\quad p \leftarrow p + a_i x^i$$
$$\textbf{end for}$$

Iterative 'for' loop: body of loop is executed iteratively n times with the variable i assigned the values 0, 1, 2 , ... , n-2, n-1, n in sequence

$\leftarrow$ is the 'assignment' operator:

the value of the expression on the right is assigned to the variable on the left.

The value that is accumulated in the variable p after n+1 iterations will be the value of the polynomial $p(x)$

Image for Algorithmic Variables

Expanding the **for** Loop

$$\text{initialize: } p = 0$$

$$i = 0 : p = a_0$$

$$i = 1 : p = a_0 + a_1 x$$

$$i = 2 : p = a_0 + a_1 x + a_2 x^2$$

$$i = 3 : p = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$\vdots$$

$$\text{Result: } p = \sum_{i=0}^{n} a_i x^i$$

More Detailed Pseudocode

Now include detailed pseudocode for evaluating x^i

$$\textbf{integer } i, j, n; \ \textbf{real } p, q, x; \ \textbf{real array } (a_i)_{0:n}$$

$$p \leftarrow 0.0$$

$$\textbf{for } i = 0 \textbf{ to } n \textbf{ do}$$

$$\quad q \leftarrow a_i$$

$$\quad \textbf{for } j = 0 \textbf{ to } i-1 \textbf{ do}$$

$$\qquad q \leftarrow q * x$$

$$\quad \textbf{end for}$$

$$\quad p \leftarrow p + q$$

$$\textbf{end for}$$

Outer 'for' loop

Inner 'for' loop

Nested Evaluation of Polynomials

From section 1.1 of Cheney and Kincaid

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

is much more efficiently computed as

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n))\cdots))$$

Requiring just *n* multiplications and *n* additions

Pseudocode for Nested Algorithm

integer $i, n;$ **real** $p, x;$ **real array** $(a_i)_{0:n}$

$p \leftarrow a_n$

for $i = n - 1$ **to** 0 **do**

  $p \leftarrow a_i + xp$

end for

Expanding the for Loop

$$\text{initialize: } p = a_n$$
$$i = n-1: p = a_{n-1} + a_n x$$
$$i = n-2: p = a_{n-2} + a_{n-1} x + a_n x^2$$
$$i = n-3: p = a_{n-3} + a_{n-2} x + a_{n-1} x^2 + a_n x^3$$
$$i = n-4: p = a_{n-4} + a_{n-3} x + a_{n-2} x^2 + a_{n-1} x^3 + a_n x^4$$
$$\vdots$$

$$\text{Result: } p = \sum_{i=0}^{n} a_i x^i$$

Synthetic Division of Polynomials

$$\text{Let } p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

$$\text{Find } q(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-2} x^{n-2} + b_{n-1} x^{n-1}$$

$$\text{such that } p(x) = (x - r) q(x) + p(r) \text{ for some } r$$

Equate coefficients of x^i in both expressions of *p(x)*
The algorithm for finding b_i recursively from b_{i+1} and a_{i+1} appears
It is the same algorithm as nested evaluation of the polynomial *p(x)* except that intermediate terms are stored in an array and are the b_i coefficients

Pseudocode for Synthetic Division Algorithm

integer $i, n;$ **real** $p, r;$ **real array** $(a_i)_{0:n}, (b_i)_{-1:n-1}$

$b_{n-1} \leftarrow a_n$

for $i = n-1$ **to** 0 **do**

$\quad b_{i-1} \leftarrow a_i + r b_i$

end for

After n iterations

b_i contain coefficients of $q(x)$

b_{-1} contains $p(r)$

See
http://en.wikipedia.org/wiki/Horner%27s_method
http://en.wikipedia.org/wiki/Polynomial_remainder_theorem

Taylor's Theorem with Remainder for f(x)

for $a \leq x \leq b$ and $a \leq c \leq b$

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(c)}{k!}(x-c)^k \;+\; E_{n+1}$$

The error term $E_{n+1} = \dfrac{f^{(n+1)}(\xi)}{(n+1)!}(x-c)^{n+1}$

where ξ is between c and x

Taylor's Theorem with Remainder for f(x+h)

for $a \leq x \leq b$ and $a \leq x + h \leq b$

$$f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k \quad + \quad E_{n+1}$$

The error term $E_{n+1} = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1}$

where ξ is between x and $x + h$

Example Taylor Series for $\sqrt{(1+h)}$

Two terms: $\sqrt{(1+h)} \approx 1 + \frac{h}{2} - \frac{1}{8}\xi^{-\frac{3}{2}}h^2$

Three terms: $\sqrt{(1+h)} \approx 1 + \frac{h}{2} - \frac{h^2}{8} + \frac{1}{16}\xi^{-\frac{5}{2}}h^3$

Four terms: $\sqrt{(1+h)} \approx 1 + \frac{h}{2} - \frac{h^2}{8} + \frac{h^3}{16} - (\,)\xi^{-\frac{7}{2}}h^4$

where ξ is between 1 and $1+h$

Pseudocode for Series Approximations

2 terms:

real h, f, e, d;
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
  $f \leftarrow 1 + x/2$
  $e \leftarrow \left| f - \sqrt{1+h} \right|$
  output h, e
  $h \leftarrow h \cdot 10^{1/d}$
end while

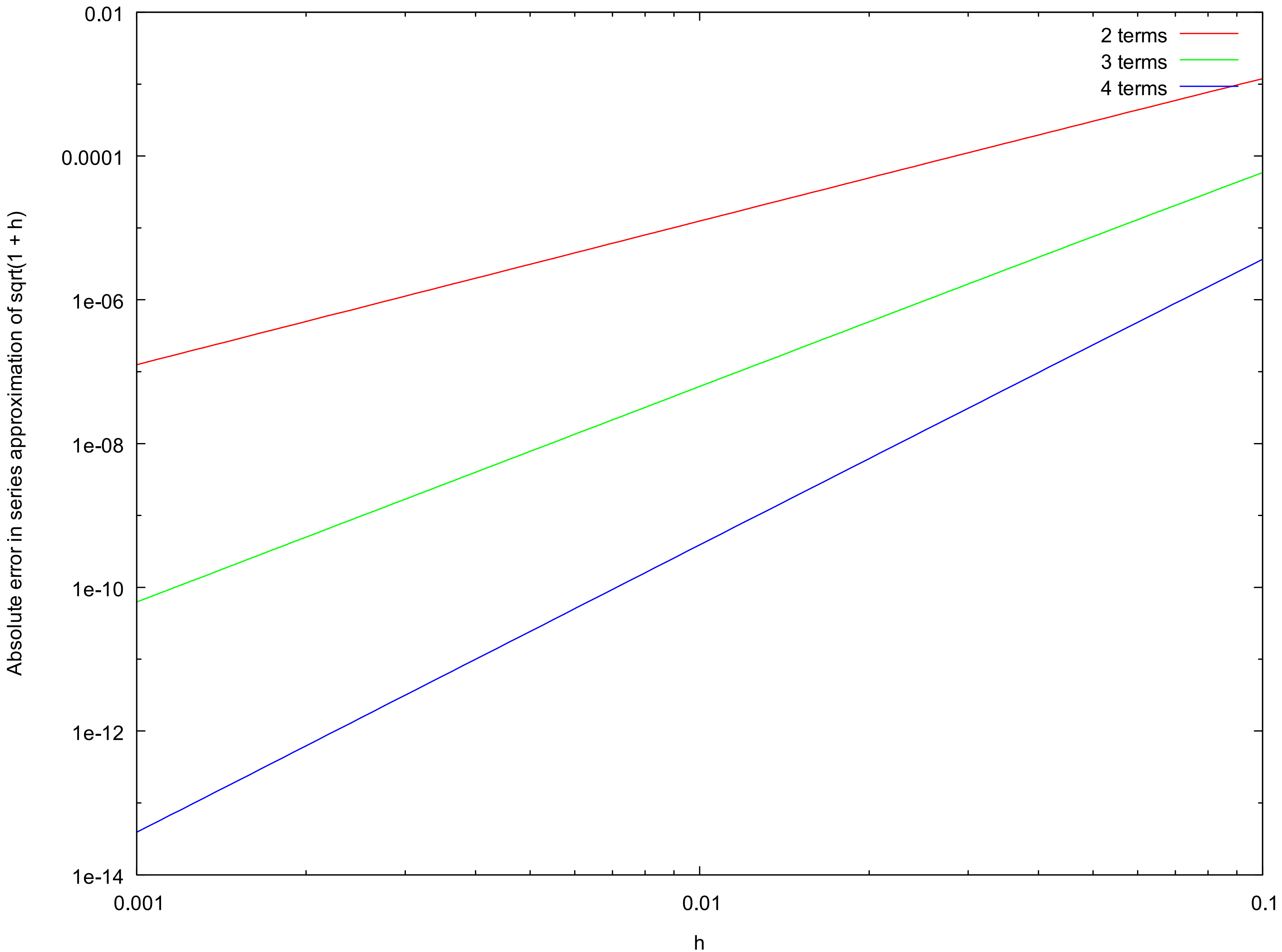
3 terms:

real h, f, e, d;
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
  $f \leftarrow 1 + x/2 - h^2/8$
  $e \leftarrow \left| f - \sqrt{1+h} \right|$
  output h, e
  $h \leftarrow h \cdot 10^{1/d}$
end while

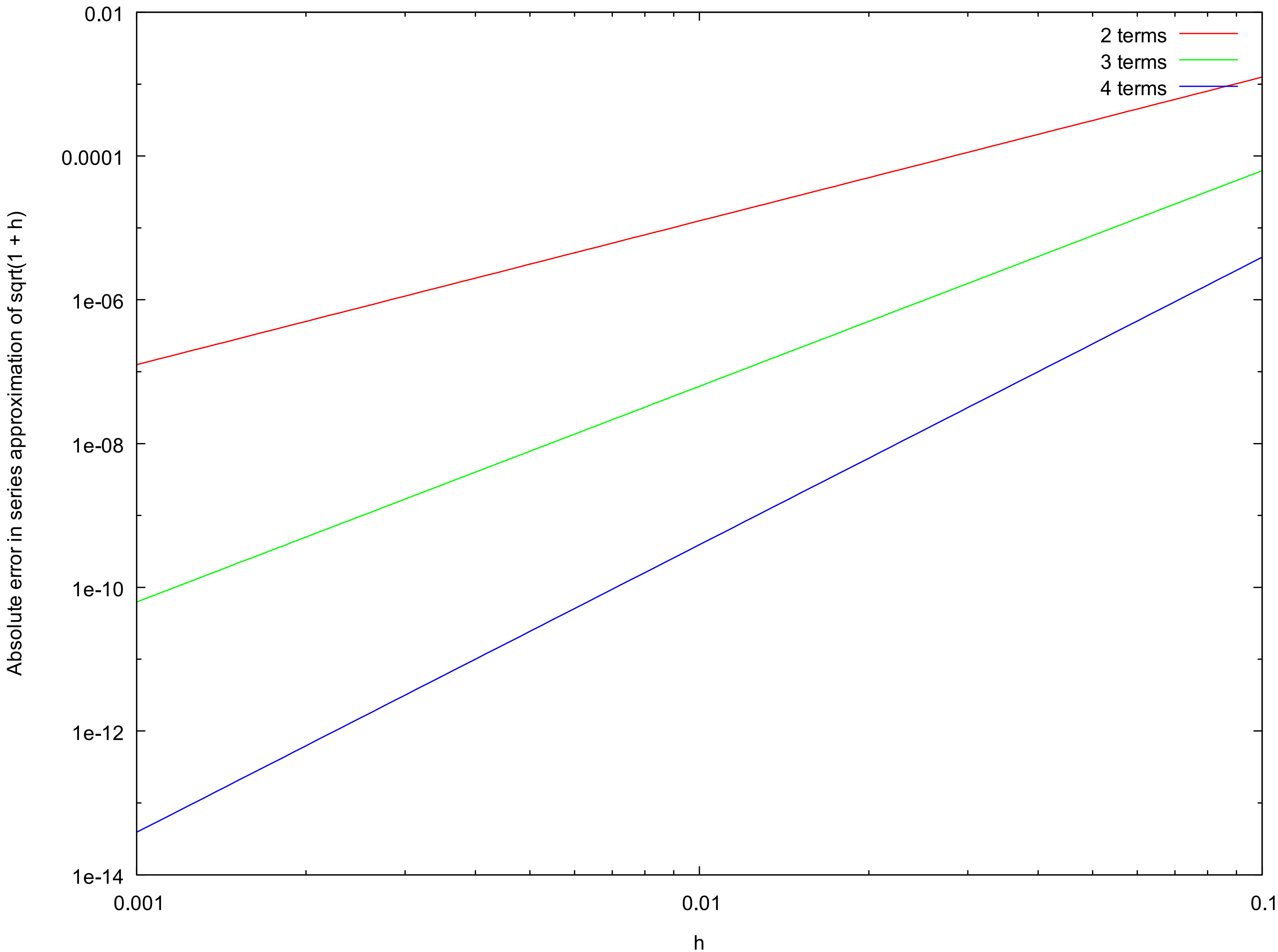
4 terms:

real h, f, e, d;
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
  $f \leftarrow 1 + x/2 - h^2/8 + h^3/16$
  $e \leftarrow \left| f - \sqrt{1+h} \right|$
  output h, e
  $h \leftarrow h \cdot 10^{1/d}$
end while

Actual Error in Series Approximation



Estimated Error in Series Approximation



Pseudocode for Series Error Estimates

2 terms:

real $h, e, d;$
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
    $e \leftarrow h^2/8$
    output h, e
    $h \leftarrow h \cdot 10^{1/d}$
end while

3 terms:

real $h, e, d;$
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
    $e \leftarrow h^3/16$
    output h, e
    $h \leftarrow h \cdot 10^{1/d}$
end while

4 terms:

real $h, e, d;$
$h \leftarrow 0.001$
while $h \leqslant 0.1$ **do**
    $e \leftarrow (\,)h^4$
    output h, e
    $h \leftarrow h \cdot 10^{1/d}$
end while

Stirling Approximation

From Cheney and Kincaid, problem 1.2.47

$$n! \geqslant \sqrt{2\pi n}\cdot n^{n}\cdot e^{-n}$$

Very handy when calculating upper bounds of Taylor series error terms!

Accuracy of the Stirling Approximation