COS 323: Computing for the Physical and Social Sciences

COS 323

- People:
  - Szymon Rusinkiewicz
  - Sandra Batista
  - Victoria Yao
- Course webpage:
  - http://www.cs.princeton.edu/cos323

What's This Course About?

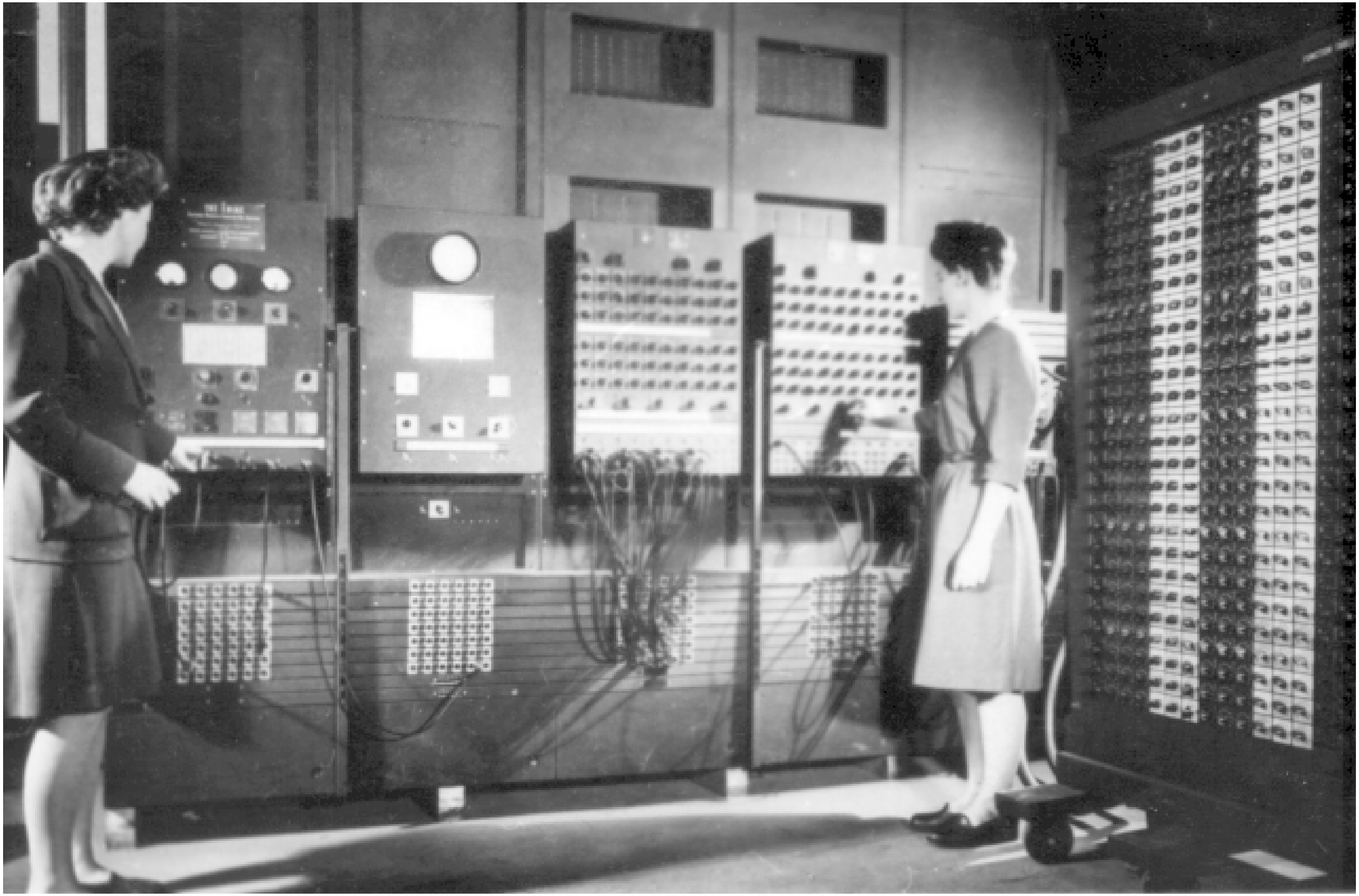
- Numerical Algorithms
- Analysis of Data
- Simulation

  - Learn through applications

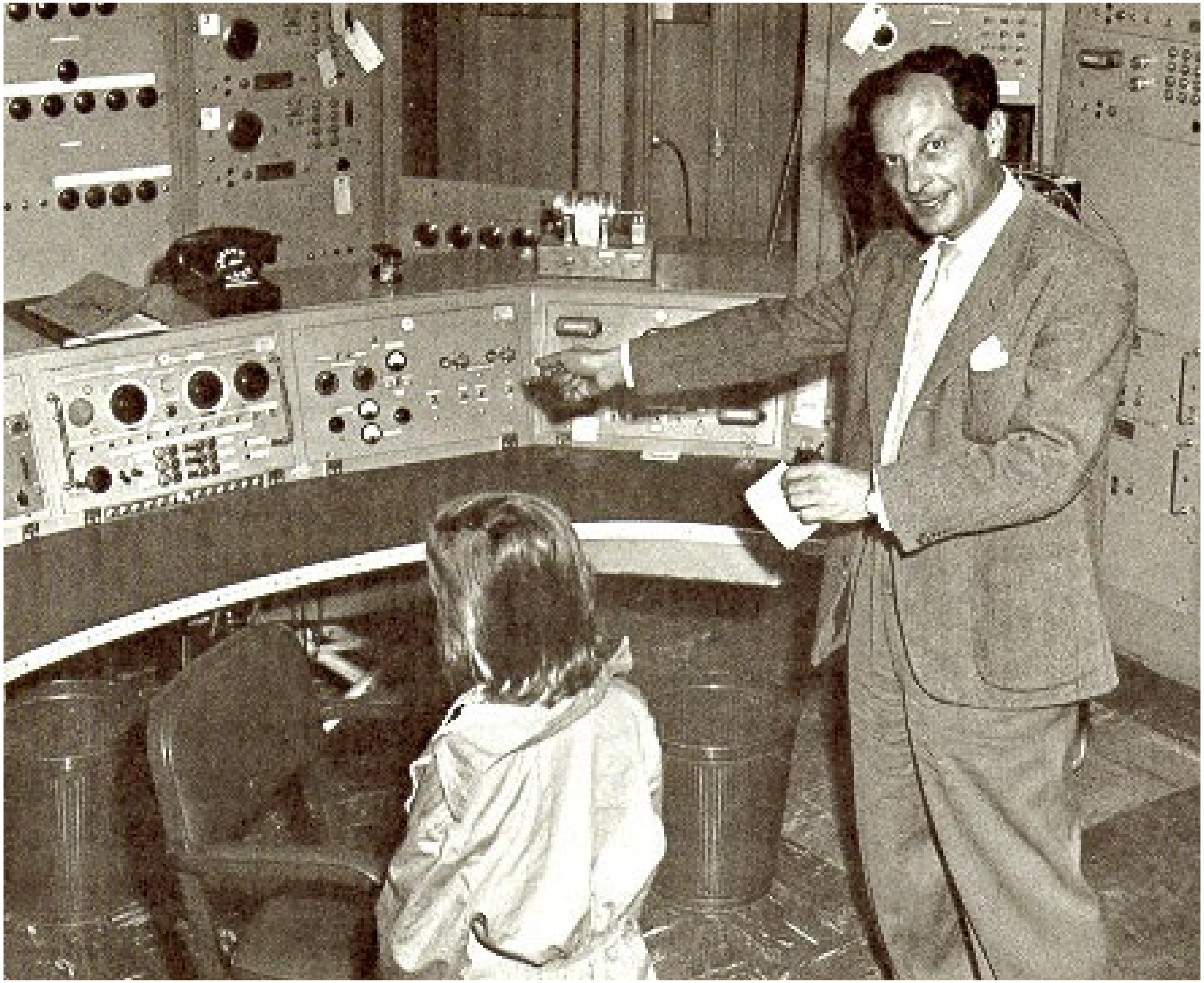
Scientific Computing

Computers, from their invention until the 70s/80s, were used mostly to solve problems

- Before "personal" computers (!)
- Users were scientists: producers of numerical "codes" rather than consumers of "applications"



Betty Jean Jennings and Fran Bilas with ENIAC I –
first general-purpose electronic computer



Stanisław Ulam with MANIAC I – about 10^4 ops/sec

from *SIAM News*, Volume 33, Number 4

The Best of the 20th Century: Editors Name Top 10 Algorithms

By Barry A. Cipra

Algos is the Greek word for pain. *Algor* is Latin, to be cold. Neither is the root for *algorithm*, which stems instead from al-Khwarizmi, the name of the ninth-century Arab scholar whose book *al-jabr wa'l muqabalah* devolved into today's high school algebra textbooks. Al-Khwarizmi stressed the importance of methodical procedures for solving problems. Were he around today, he'd no doubt be impressed by the advances in his eponymous approach.

Some of the very best algorithms of the computer age are highlighted in the January/February 2000 issue of *Computing in Science & Engineering*, a joint publication of the American Institute of Physics and the IEEE Computer Society. Guest editors Jack Don-garra of the University of Tennessee and Oak Ridge National Laboratory and Fran-cis Sullivan of the Center for Comput-ing Sciences at the Institute for Defense Analyses put togeth-er a list they call the "Top Ten Algorithms of the Century."

"We tried to assemble the 10 al-gorithms with the greatest influence on the development and practice of science and engineering in the 20th century," Dongarra and Sullivan write. As with any top-10 list, their selections—and non-selections—are bound to be controversial, they acknowledge. When it comes to picking the algorithmic best, there seems to be no best algorithm.

Without further ado, here's the CiSE top-10 list, in chronological order. (Dates and names associated with the algorithms should be read as first-order approximations. Most algorithms take shape over time, with many contributors.)

1946: John von Neumann, Stan Ulam, and Nick Metropolis, all at the Los Alamos Scientific Laboratory, cook up the Metropolis algorithm, also known as the **Monte Carlo method**.

The Metropolis algorithm aims to obtain approximate solutions to numerical problems with unmanageably many degrees of freedom and to combinatorial problems of factorial size, by mimicking a random process. Given the digital computer's reputation for deterministic calculation, it's fitting that one of its earliest applications was the generation of random numbers.

In terms of widespread use, George Dantzig's simplex method is among the most successful algorithms of all time.

1947: George Dantzig, at the RAND Corporation, creates the **simplex method for linear programming**.

In terms of widespread application, Dantzig's algorithm is one of the most successful of all time: Linear programming dominates the world of industry, where economic survival depends on the ability to optimize within budgetary and other constraints. (Of course, the "real" problems of industry are often nonlinear; the use of linear programming is sometimes dictated by the computational budget.) The simplex method is an elegant way of arriving at optimal answers. Although theoretically susceptible to exponential delays, the algorithm in practice is highly efficient—which in itself says something interesting about the nature of computation.

1950: Magnus Hestenes, Eduard Stiefel, and Cornelius Lanczos, all from the Institute for Numerical Analysis at the National Bureau of Standards, initiate the development of **Krylov subspace iteration methods**.

These algorithms address the seemingly simple task of solving equations of the form $Ax = b$. The catch, of course, is that A is a huge $n \times n$ matrix, so that the algebraic answer $x = b/A$ is not so easy to compute. (Indeed, matrix "division" is not a particularly useful concept.) Iterative methods—such as solving equations of the form $Kx_{i+1} = Kx_i + b - Ax_i$ with a simpler matrix K that's ideally "close" to A—lead to the study of Krylov subspaces. Named for the Russian mathematician Nikolai Krylov, Krylov subspaces are spanned by powers of a matrix applied to an initial "remainder" vector $r_0 = b - Ax_0$. Lanczos found a nifty way to generate an orthogonal basis for such a subspace when the matrix is symmetric. Hestenes and Stiefel proposed an even niftier method, known as the conjugate gradient method, for systems that are both symmetric and positive definite. Over the last 50 years, numerous researchers have improved and extended these algorithms. The current suite includes techniques for non-symmetric systems, with acronyms like GMRES and Bi-CGSTAB. (GMRES and Bi-CGSTAB premiered in *SIAM Journal on Scientific and Statistical Computing*, in 1986 and 1992, respectively.)

1951: Alston Householder of Oak Ridge National Laboratory formalizes the **decompositional approach to matrix computations**.

The ability to factor matrices into triangular, diagonal, orthogonal, and other special forms has turned out to be extremely useful. The decompositional approach has enabled software developers to produce flexible and efficient matrix packages. It also facilitates the analysis of rounding errors, one of the big bugbears of numerical linear algebra. (In 1961, James Wilkinson of the National Physical Laboratory in London published a seminal paper in the *Journal of the ACM*, titled "Error Analysis of Direct Methods of Matrix Inversion," based on the LU decomposition of a matrix as a product of lower and upper triangular factors.)

Alston Householder

1957: John Backus leads a team at IBM in developing the **Fortran optimizing compiler**.

The creation of Fortran may rank as the single most important event in the history of computer programming: Finally, scientists

8 out of the top 10 algorithms of the 20th century are numerical in nature

(we'll cover 6 of them)

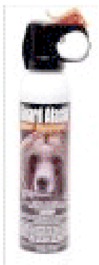

BETA
ANGRY BIRDS
LITE
PLAY
MOTOROLA

Hurricane Irene Models
hamweather.com
NHC
GFS
BAMM
UKMET
CMC
GFDL
NOGAPS
New York City
Richmond
Atlanta
Dallas
Bermuda
Austin
New Iberia
Panama City
Port OConnor
Brownsville
La Pesca
Cabo San Antonio
Cabo Rojo
Campeche
Veracruz
Grand Cayman
Belize City
Antigua/Barbuda
Salina Cruz
Sipacate
P. Cabezas
Barbados
Bluefields
Mon Aug 22 2011 07:14 PM EDT
Barranquilla
Caracas
Tobago

Dwight, Welcome to Your Amazon.com (If you're not Dwight K Schrute, click here.)

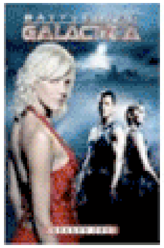
Today's Recommendations For You

Here's a daily sample of items recommended for you. Click here to see all recommendations. Page 1 of 44



Guard Alaska™ Bear Defense Spray
(8) $35.00
Fix this recommendation

Pickled Beets, Sliced by Barry Farm
(1) $4.49
Fix this recommendation

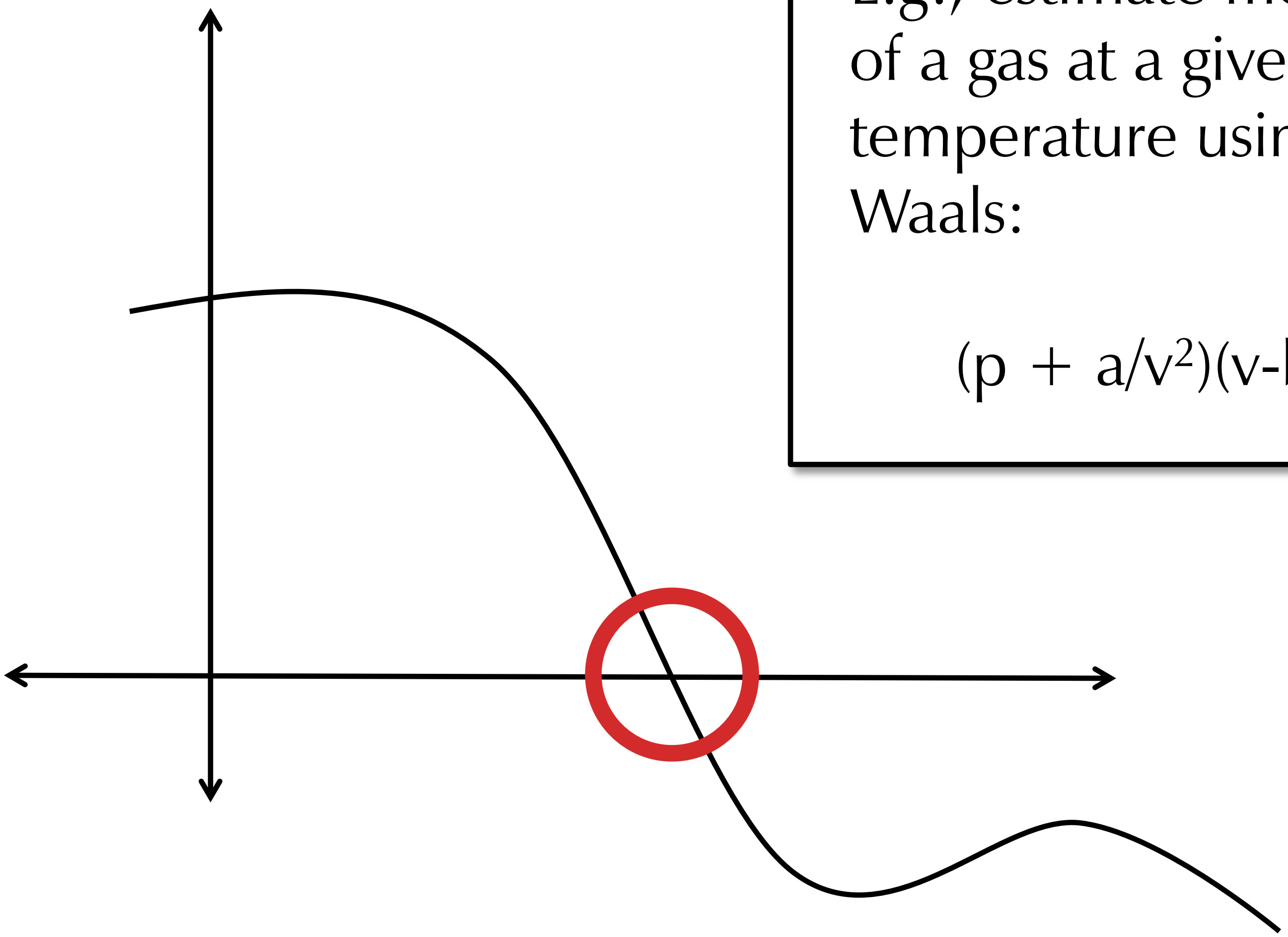


Battlestar Galactica - Season One
(553) $34.99
Fix this recommendation

Reebok 65cm Stability Ball by Reebok
(8) $18.78
Fix this recommendation

Some challenging but important & common problems…

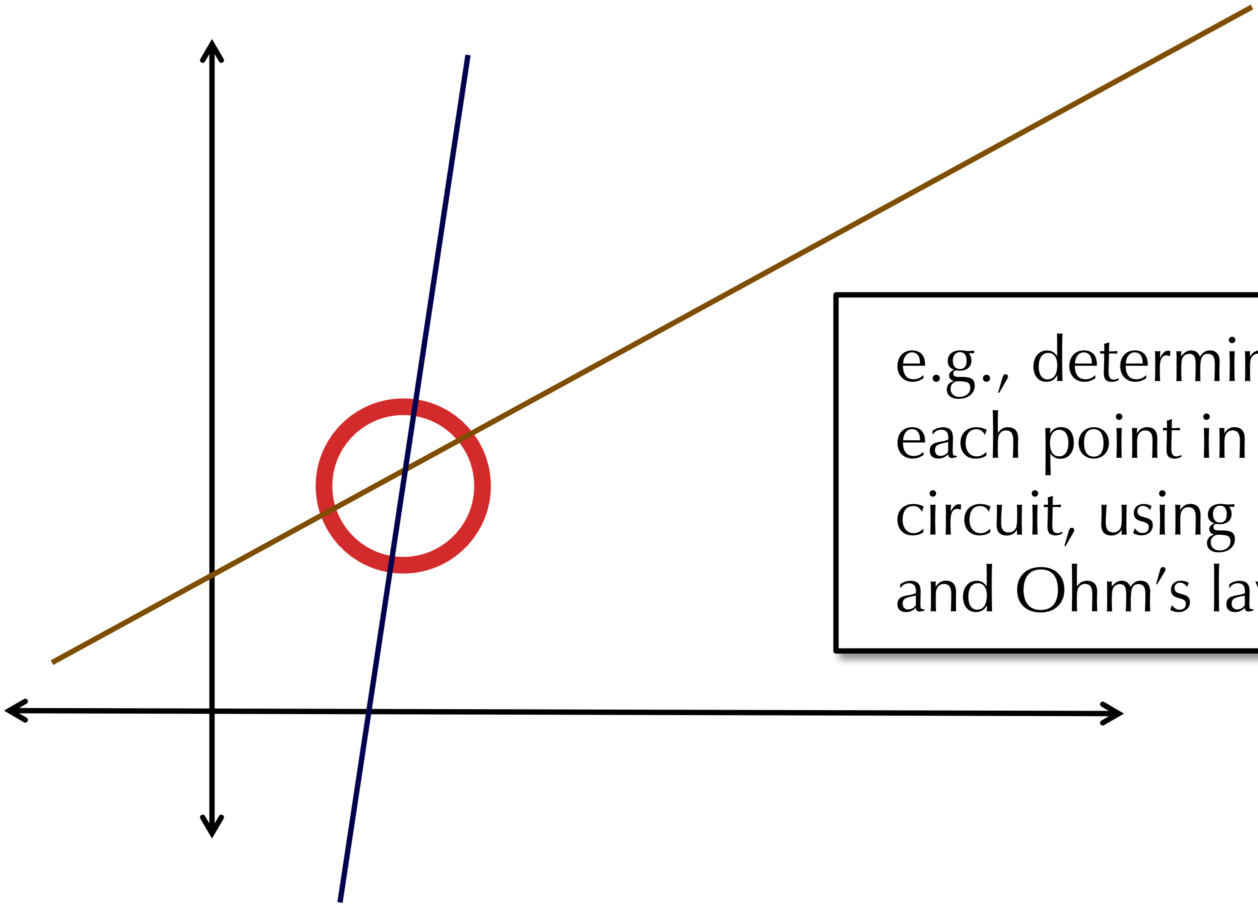
Root finding



E.g., estimate molal volume of a gas at a given pressure & temperature using van der Waals:

$$(p + a/v^2)(v-b) = RT$$

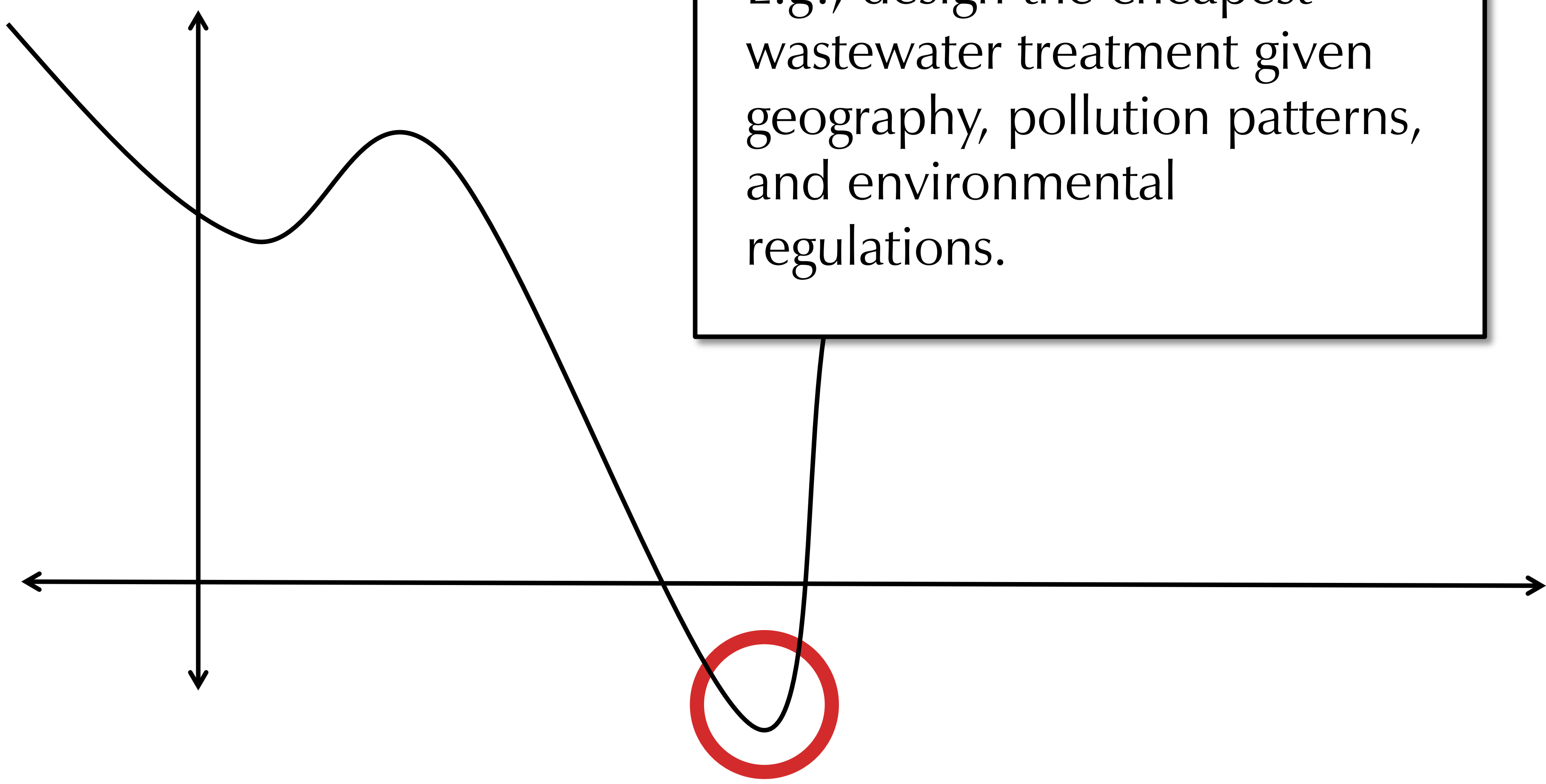
Solving systems of linear equations



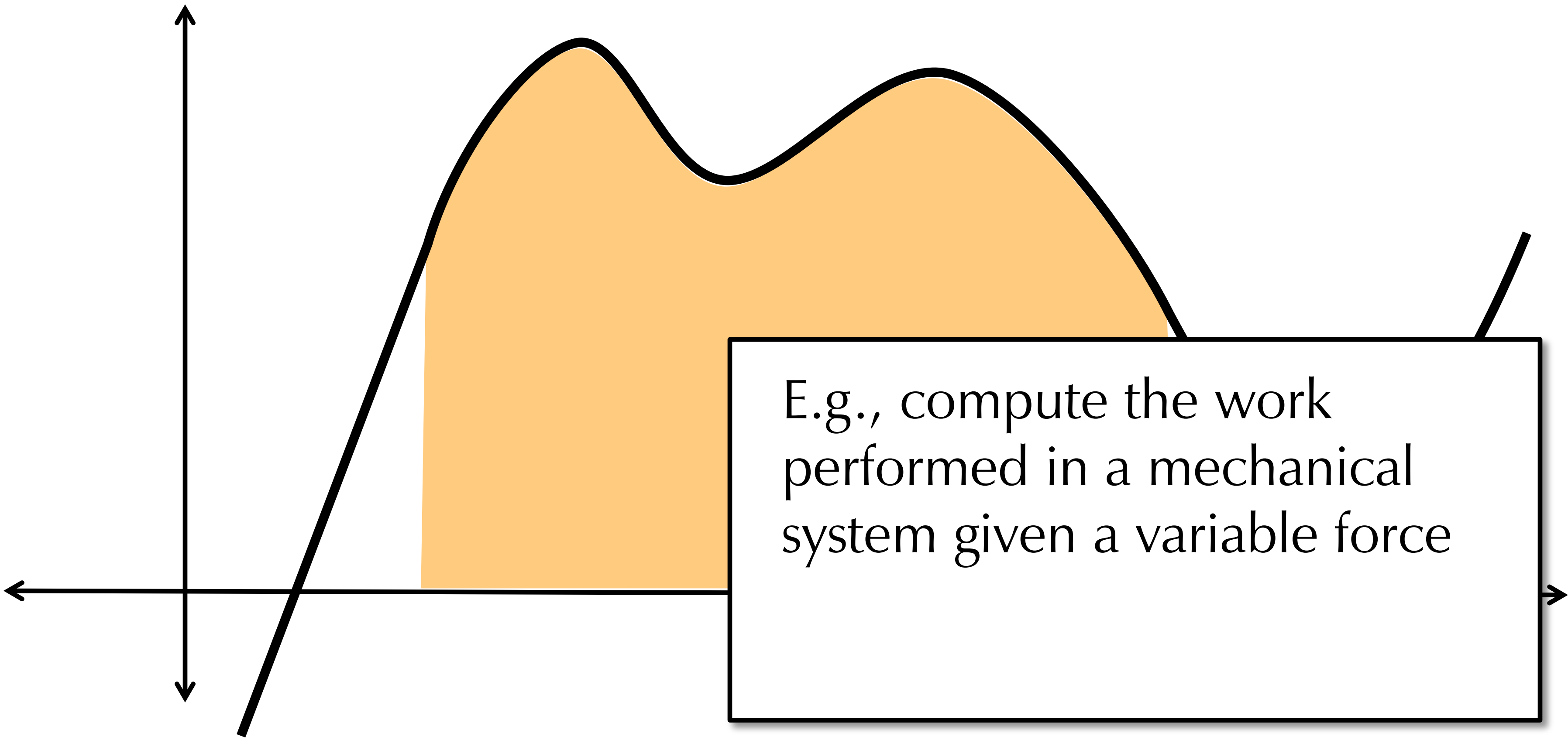
e.g., determine the current at each point in an electrical circuit, using Kirchoff's rule and Ohm's law.

Optimization

E.g., design the cheapest wastewater treatment given geography, pollution patterns, and environmental regulations.



Integration



E.g., compute the work performed in a mechanical system given a variable force

How do we solve these problems?

Numerical Analysis

- Algorithms for solving numerical problems
  - Calculus, algebra, data analysis, etc.
  - Used even if answer is not simple/elegant: “math in the real world”
- Analyze/design algorithms based on:
  - Running time, memory usage (both asymptotic and constant factors)
  - **Applicability, stability, and accuracy**

Why Is This Hard/Interesting?

- "Numbers" in computers ≠ numbers in math
  - Limited precision and range
- Algorithms sometimes don't give right answer
  - Iterative, randomized, approximate
  - Unstable
- Tradeoffs in accuracy, stability, and running time

Numbers in Computers

and their consequences

Numbers in Computers

- "Integers"
  - Implemented in hardware: fast
  - Mostly sane, except for limited range
- Floating point
  - Implemented in most hardware
  - Much larger range
    (e.g. -2^{31}... 2^{31} for integers, vs. -2^{127}... 2^{127} for FP)
  - Lower precision (e.g. 7 digits vs. 9)
  - "Relative" precision: actual accuracy depends on size

Floating Point Numbers

- Like scientific notation: e.g., *c* is

  $$2.99792458 \times 10^{8} \text{ m/s}$$

- This has the form

  $$(\text{multiplier}) \times (\text{base})^{(\text{power})}$$

- In the computer,
  - Multiplier is called mantissa
  - Base is almost always 2
  - Power is called exponent

Modern Floating Point Formats

- Almost all computers use IEEE 754 standard
- "Single precision":
  - 24-bit mantissa, base = 2, 8-bit exponent, 1 bit sign
  - All fits into 32 bits (!) – mantissa has implicit leading 1
- "Double precision":
  - 53-bit mantissa, base = 2, 11-bit exponent, 1 bit sign
  - All fits into 64 bits
- Sometimes also have "extended formats"

Other Number Representations

- Fixed point
  - *Absolute* accuracy doesn't vary with magnitude
  - Represent fractions to a fixed precision
  - Not supported directly in hardware, but can hack it
- "Infinite precision"
  - Integers or rationals allocated dynamically
  - Can grow up to available memory
  - No direct support in hardware, but libraries available

Consequences of Floating Point

- "Machine epsilon": smallest positive number you can add to 1.0 and get something other than 1.0
- For single precision: $\varepsilon \approx 10^{-7}$
  - No such number as 1.000000001
  - Rule of thumb: "almost 7 digits of precision"
- For double: $\varepsilon \approx 2 \times 10^{-16}$
  - Rule of thumb: "not quite 16 digits of precision"
- These are all *relative* numbers

So What?

- Simple example: add $^1/_{10}$ to itself 10 times

Yikes!

- Result: $^1/_{10} + {}^1/_{10} + \ldots \neq 1$
- Reason: 0.1 can't be represented exactly in binary floating point
  - Like $^1/_3$ in decimal

- Rule of thumb: comparing floating point numbers for equality is always wrong

More Subtle Problem

- Using quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

to solve $x^2 - 9999x + 1 = 0$

  - Only 4 digits: single precision should be OK, right?

- Correct answers: 0.0001… and 9998.999…
- Actual answers in single precision: 0 and 9999
  - First answer is 100% off!
  - Total cancellation in numerator because $b^2 >> 4ac$

Accuracy

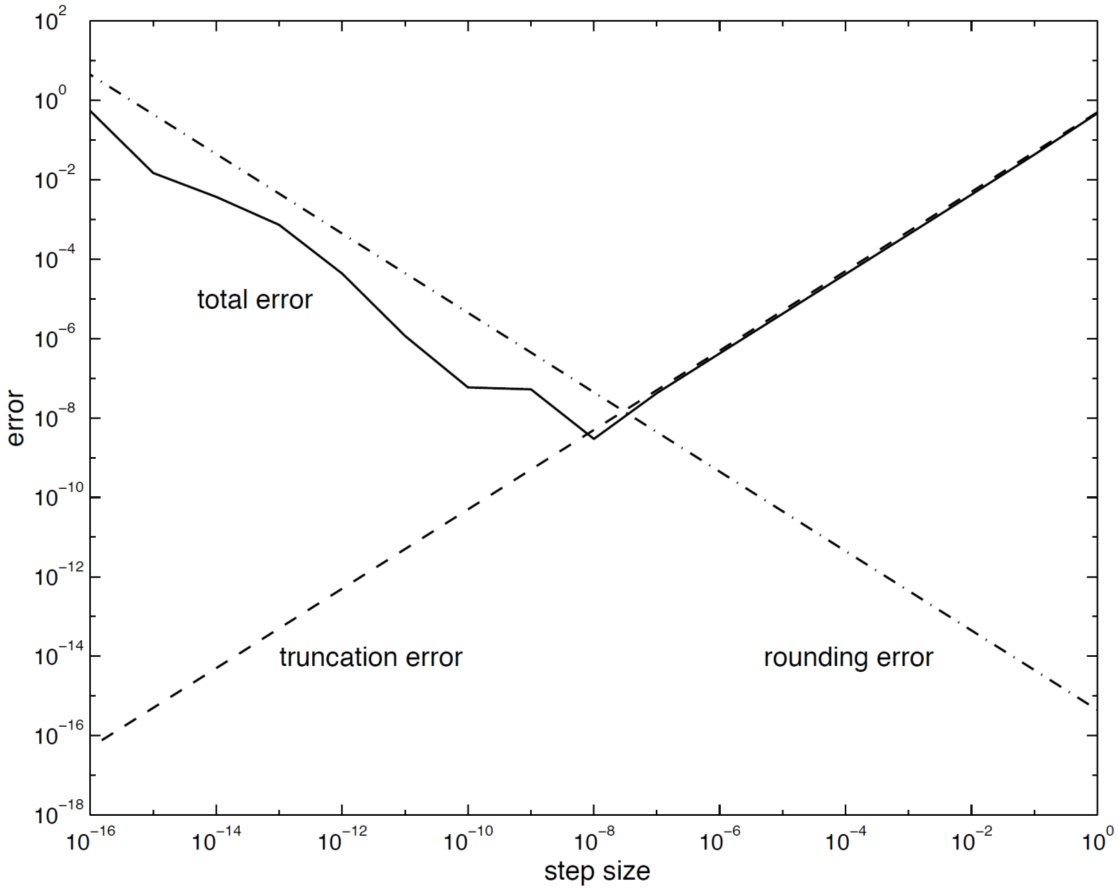
error is inevitable

Catalog of Errors

- Inherent error in data or model
  - "Garbage in, garbage out"
- Approximation errors in algorithm
  - Discretization error – e.g., too-big steps for derivative
  - Truncation error – e.g., too few terms of Taylor series
  - Convergence error – stopping iteration too early
  - Statistical error – too few random samples
- Roundoff error due to floating-point "numbers"

Error Tradeoff Example – Computing Derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$



[Heath]

Other Considerations of Problem Formulation & Algorithm

Sensitivity & conditioning, stability & accuracy

Well-Posedness and Sensitivity

- Problem is well-posed if solution
  - exists
  - is unique
  - depends continuously on problem data

  Otherwise, problem is ill-posed

- Solution may still be sensitive to input data
  - Ill-conditioned: relative change in solution much larger than that in input data

[Heath]

Sensitivity & Conditioning

- Some problems propagate error in bad ways
  - e.g., $y = \tan(x)$ sensitive to small changes in x near $\pi/2$
- Small error in input $\rightarrow$ huge error in solution: ill-conditioned
- Well-conditioned problems may have ill-conditioned inverses, and vice versa
  - e.g., $y = \text{atan}(x)$

Stability & Accuracy

- A stable algorithm introduces "only a little" computational error
  - Solution is an exact to solution to a "nearby" problem
  - Computational error is indistinguishable from a small data error
- An accurate algorithm produces a solution that is close to the true solution

stable algorithm + well-conditioned problem
$\rightarrow$ accurate solution

Running time

Running Time

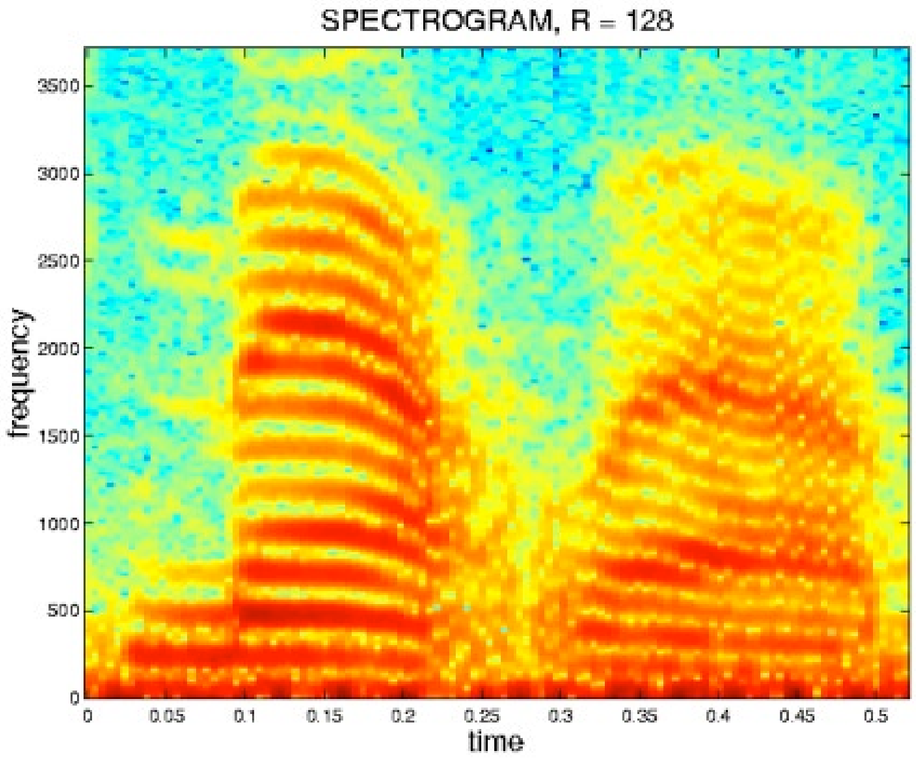
- Depending on algorithm, we'll look at:
  - Asymptotic analysis for noniterative algorithms (e.g., most methods for inverting an $n \times n$ matrix require time proportional to n^3)
  - Convergence order for iterative approximate algorithms (e.g., an answer to precision δ might require iterations proportional to $1/\delta$ or $1/\delta^2$)

Course Overview

Basic Techniques

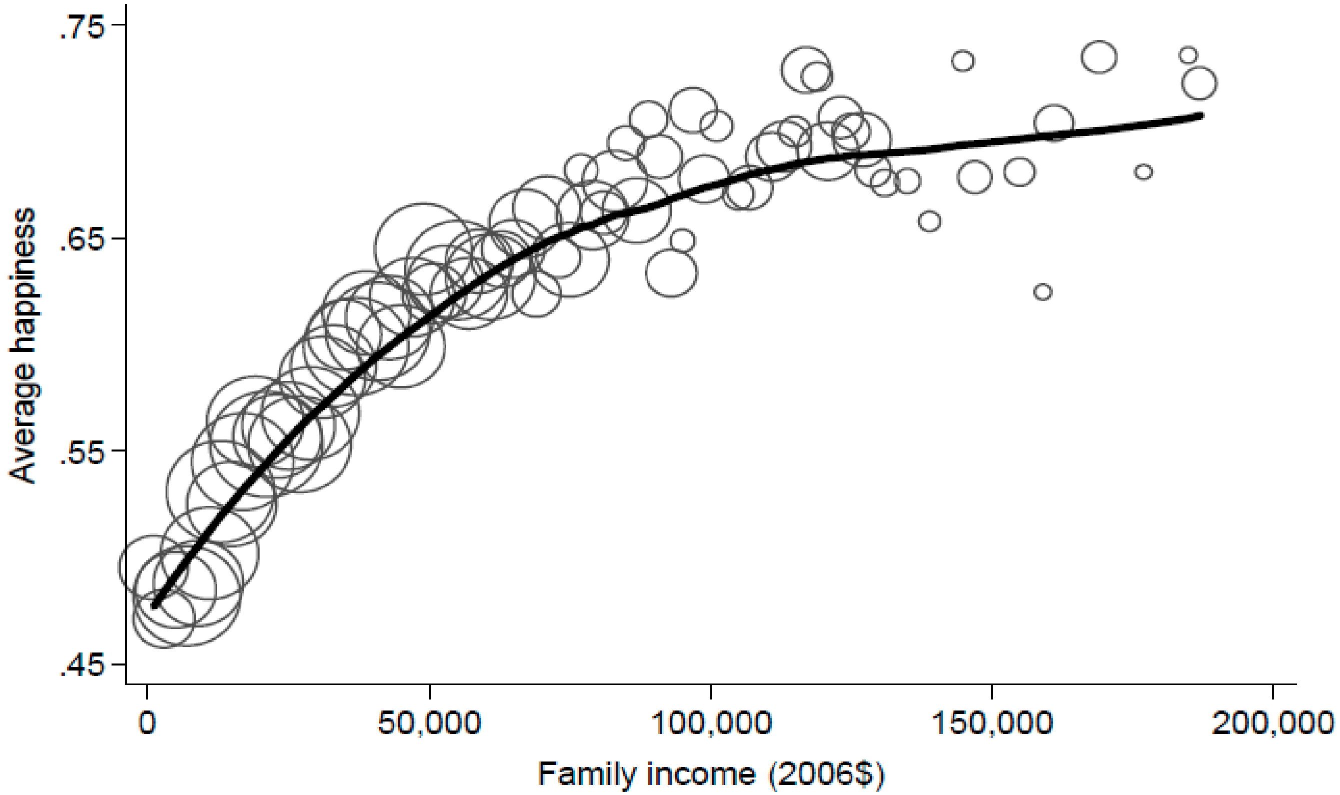
- root finding
- optimization
- linear systems
- integration
- ODEs, PDEs
- Plus…

Signal Analysis & Signal Processing



[Matusik & McMillan]

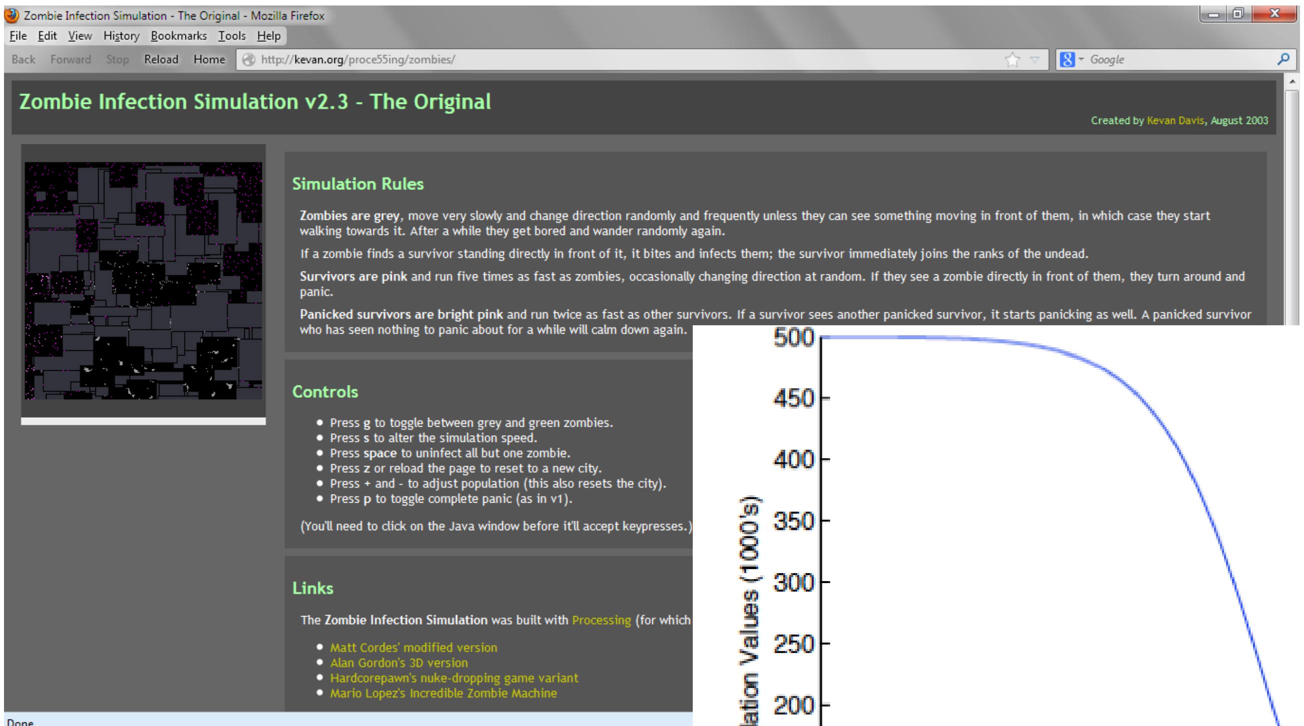
Data Analysis and Model Fitting

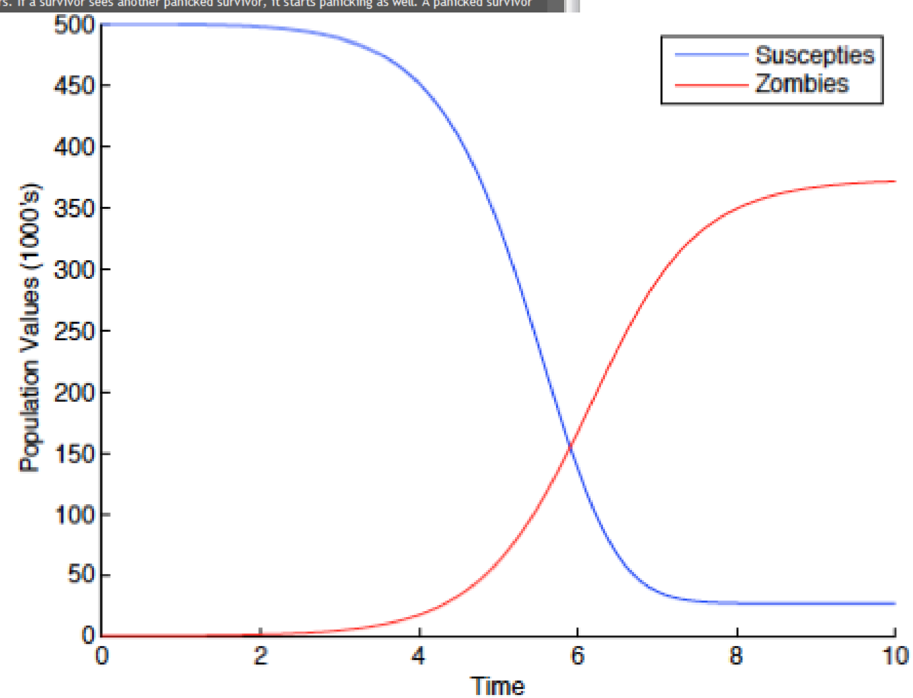


Note: 1972 to 2006. Sample size: 41,795. Each circle represents an income range of $2,000 (e.g., $10,001 to $12,000), in 2006$. Its diameter is proportional to the number of people in that range.

Source: My calculations from General Social Survey data.

Simulation


Zombie Infection Simulation - The Original - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Stop Reload Home
http://kevan.org/proce55ing/zombies/
Google
Zombie Infection Simulation v2.3 - The Original
Created by Kevan Davis, August 2003
Simulation Rules
Zombies are grey, move very slowly and change direction randomly and frequently unless they can see something moving in front of them, in which case they start walking towards it. After a while they get bored and wander randomly again.
If a zombie finds a survivor standing directly in front of it, it bites and infects them; the survivor immediately joins the ranks of the undead.
Survivors are pink and run five times as fast as zombies, occasionally changing direction at random. If they see a zombie directly in front of them, they turn around and panic.
Panicked survivors are bright pink and run twice as fast as other survivors. If a survivor sees another panicked survivor, it starts panicking as well. A panicked survivor who has seen nothing to panic about for a while will calm down again.
Controls
Press g to toggle between grey and green zombies.
Press s to alter the simulation speed.
Press space to uninfect all but one zombie.
Press z or reload the page to reset to a new city.
Press + and - to adjust population (this also resets the city).
Press p to toggle complete panic (as in v1).
(You'll need to click on the Java window before it'll accept keypresses.)
Links
The Zombie Infection Simulation was built with Processing (for which
Matt Cordes' modified version
Alan Gordon's 3D version
Hardcorepawn's nuke-dropping game variant
Mario Lopez's Incredible Zombie Machine
Done

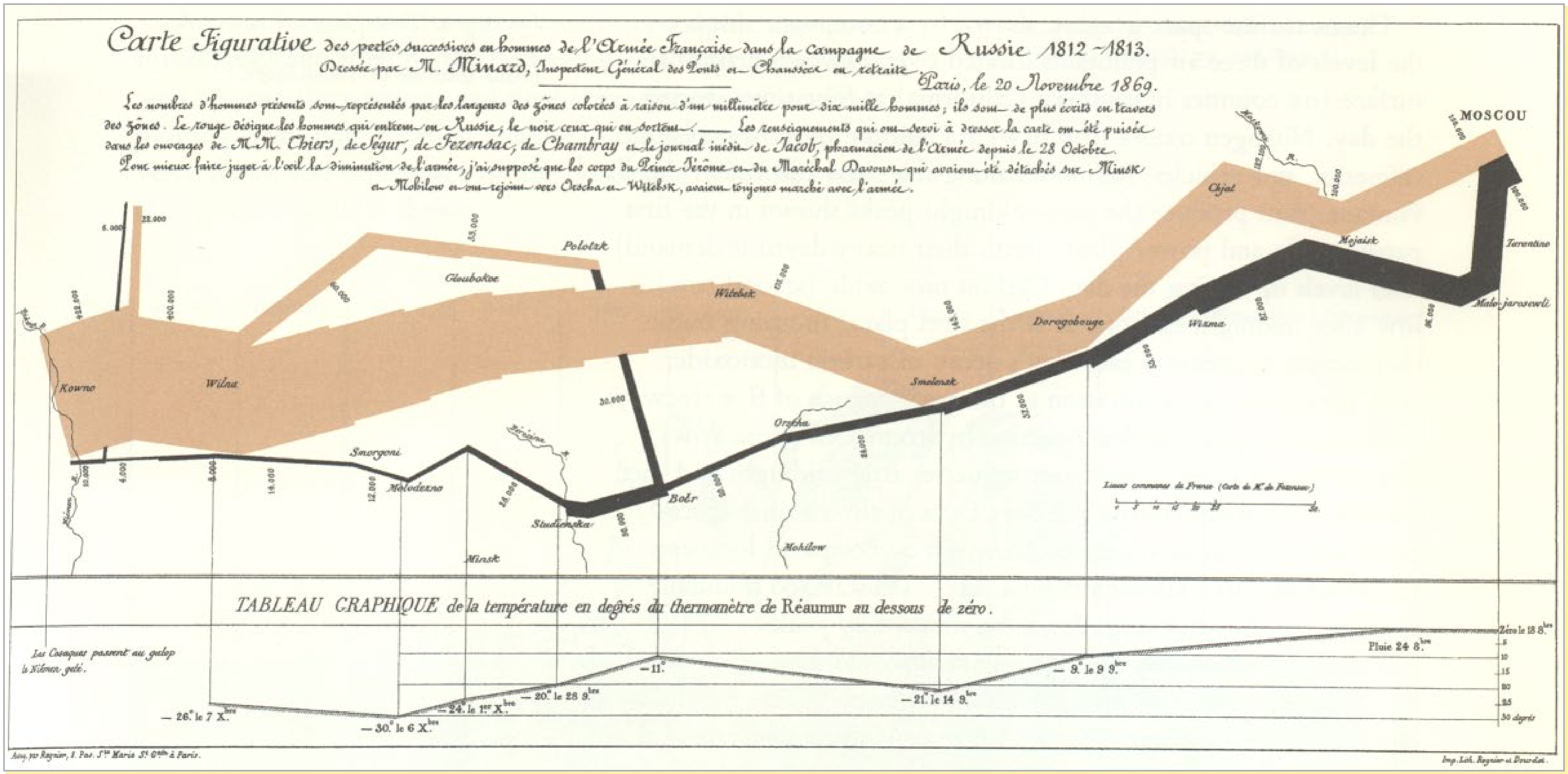

Suscepties
Zombies
Population Values (1000's)
Time
0
50
100
150
200
250
300
350
400
450
500
0
2
4
6
8
10

Simulation

" In summary, a zombie outbreak is likely to lead to the collapse of civilization, unless it is dealt with quickly. […] As seen in the movies, it is imperative that zombies are dealt with quickly, or else we are all in a great deal of trouble. "

– Munz et al., *Infectious Disease Modelling Research Progress*, 2009

Visualization



Course Information

Mechanics

- 5 programming assignments: 50%
  - Typically more thought than coding
  - MATLAB
  - Analysis, writeup counts a lot!
- 2 in-class exams: 25%
  - Short-answer, focusing on topics not covered in programming assignments
- Final project (in groups): 25%

To Do

- Course webpage:

  http://www.cs.princeton.edu/cos323

- MATLAB:
  - Install it now – instructions on webpage
  - Engineering school tutorial, or we'll do our own
- Assignment 0:
  - Available on course web page, due Tuesday Sep 24
- Sign up for Piazza