PHY 604: Computational Methods in Physics and Astrophysics II

Cyrus Dreyer
cyrus.dreyer@stonybrook.edu

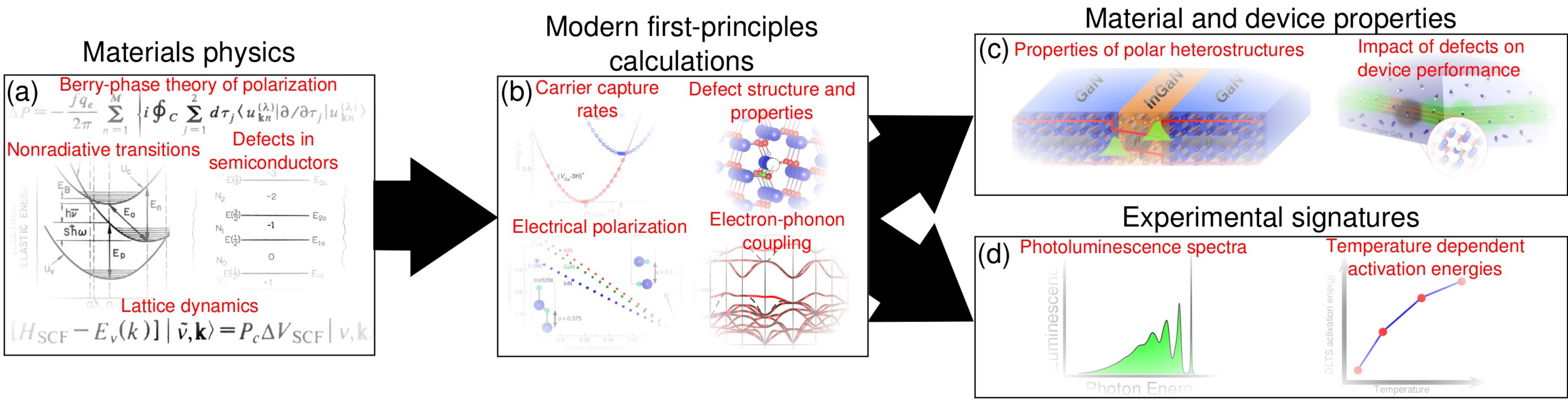
Fall 2021

My research interests: Computational condensed matter physics

Stony Brook University

FLATIRON INSTITUTE
Center for Computational Quantum Physics

https://www.simonsfoundation.org/flatiron/



Materials physics
(a) Berry-phase theory of polarization
Nonradiative transitions
Defects in semiconductors
Lattice dynamics
Modern first-principles calculations
(b) Carrier capture rates
Defect structure and properties
Electrical polarization
Electron-phonon coupling
Material and device properties
(c) Properties of polar heterostructures
Impact of defects on device performance
GaN
InGaN
GaN
Experimental signatures
(d) Photoluminescence spectra
Temperature dependent activation energies
Photon Energy
Temperature

https://you.stonybrook.edu/cdreyer/

Goals of the course:

- Learn how to solve problems in physics computationally
- Understand the limitations of numerical methods
- Have the ability to interpret numerical results presented in the literature
- Have exposure to computational tools
- Understand basic idea behind algorithms for performing common computational tasks

Technical points about the class: Programming Languages

- The assignments will involve writing computer programs
- You may use the programming language of your choice.* I would prefer:
  - Fortran
  - C++
  - Matlab
  - python
- * In general, and especially if your language is not on the list, you should provide some help for how to compile (if necessary) and run your code
- Examples will be given in fortran, C++, and python
  - I will introduce these codes early in the semester

Technical points about the class: Topics covered

- Basics of computation and programming constructions
- Good programming practices
- Numerical differentiation and integration
- Interpolation and root finding
- Ordinary differential equations
- Linear algebra
- Fast Fourier transforms
- Fitting
- Partial differential equations
- Monte Carlo techniques
- Genetic algorithms
- Parallel computing
- Machine learning

Technical points about the class: Assignments

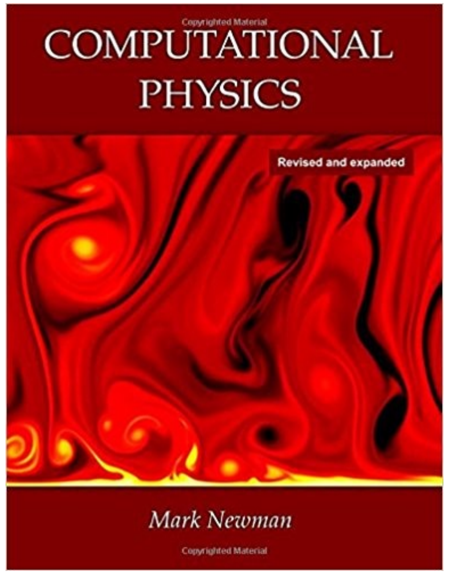
- Coding homework will be assigned roughly every two weeks
  - Homeworks will be 80% of the final grade

- You can use your choice of language (those listed on the previous slide are preferred)
  - Office hours: Mondays, 3:00pm to 4:00pm; Thursdays, 11:05am to 1:00pm
  - Please feel free to come to me for help!

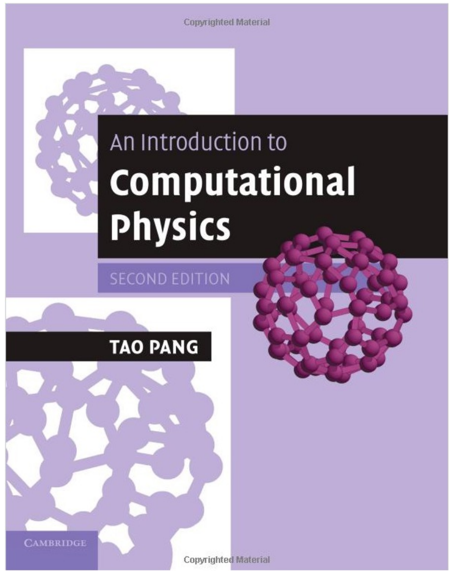
- There will be a final project at the end of the semester
  - Solve a physics problem computationally
  - Write up a short report, and present to the class
  - Final project is 20% of the final grade

Technical points about the class: Textbooks

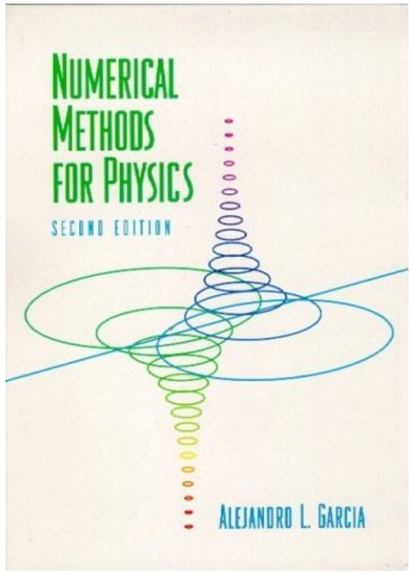
- **No textbook is required for this course**
  - Some recommended texts for further reading:



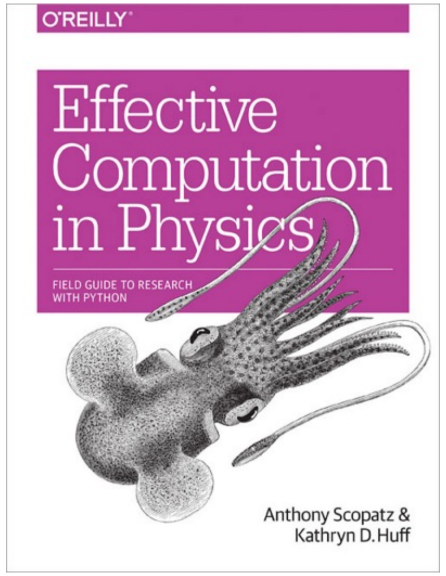
***Computational Physics*, by Mark Newman**
- Generally good coverage on most of the topics we'll discuss
- Lots of physics examples
- Inexpensive
- Main recommended book



***An Introduction to Computational Physics*, by Tao Pang**
- Also good coverage of the topics (up to PDEs)
- Lots of physics examples
- Inexpensive



***Numerical Methods for Physics* by Alejandro Garcia**
- Broad coverage
- More PDE stuff than Pang


O'REILLY
Effective Computation in Physics
FIELD GUIDE TO RESEARCH WITH PYTHON
Anthony Scopatz & Kathryn D. Huff

Effective Computation in Physics by Scopatz & Huff
- Introduces linux/unix shell
- Covers programming practices
- Introduces parallel programming

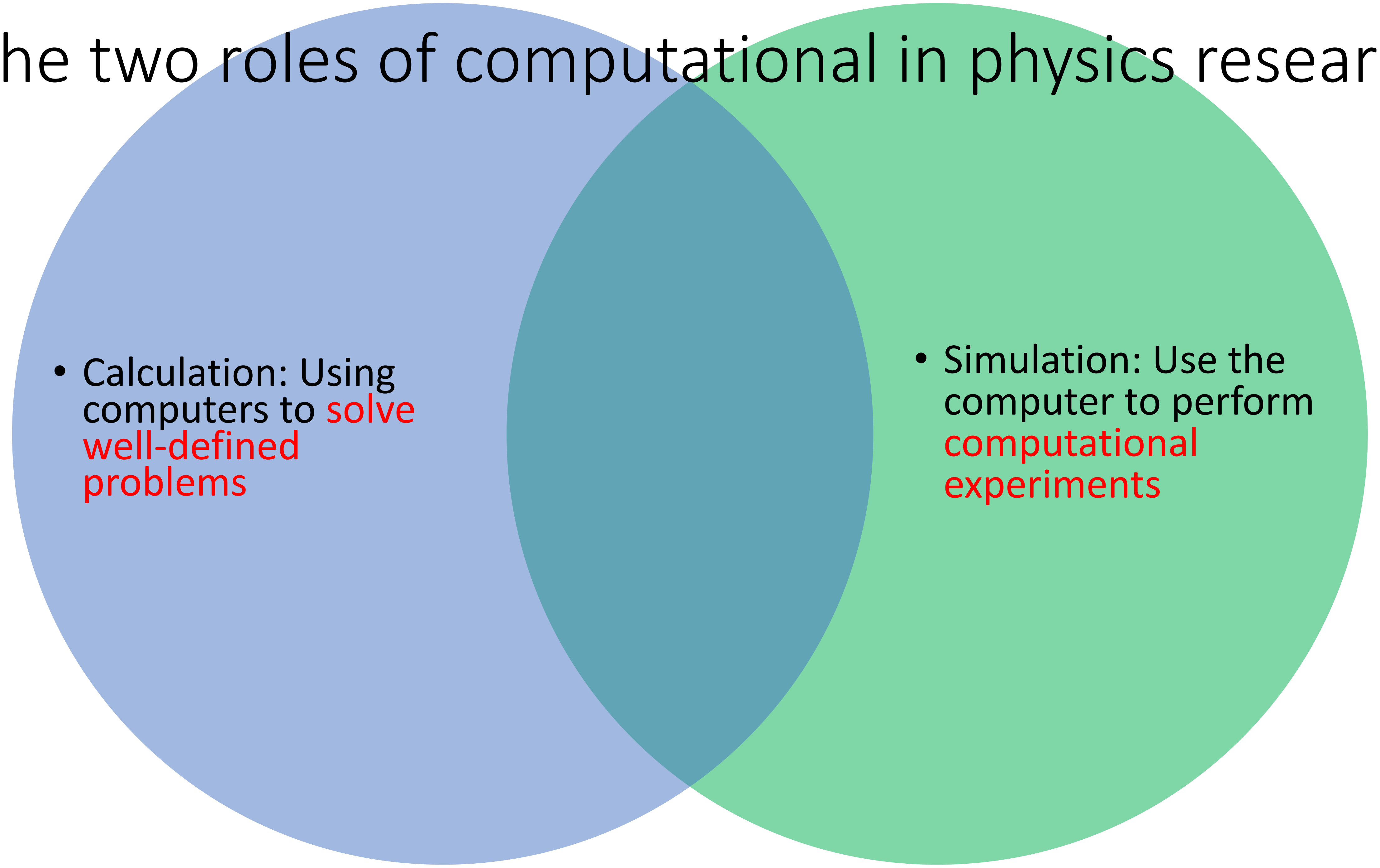
Why computation?

“Computational science now constitutes what many call the third pillar of the scientific enterprise, a peer alongside theory and physical experimentation.”

—President's information technology advisory committee (2005)

- Computation allows us to go beyond analytically solvable problems
- Computers allow us to perform repetitive tasks efficiently
- Computers allow us to generate and analyze large amounts of data

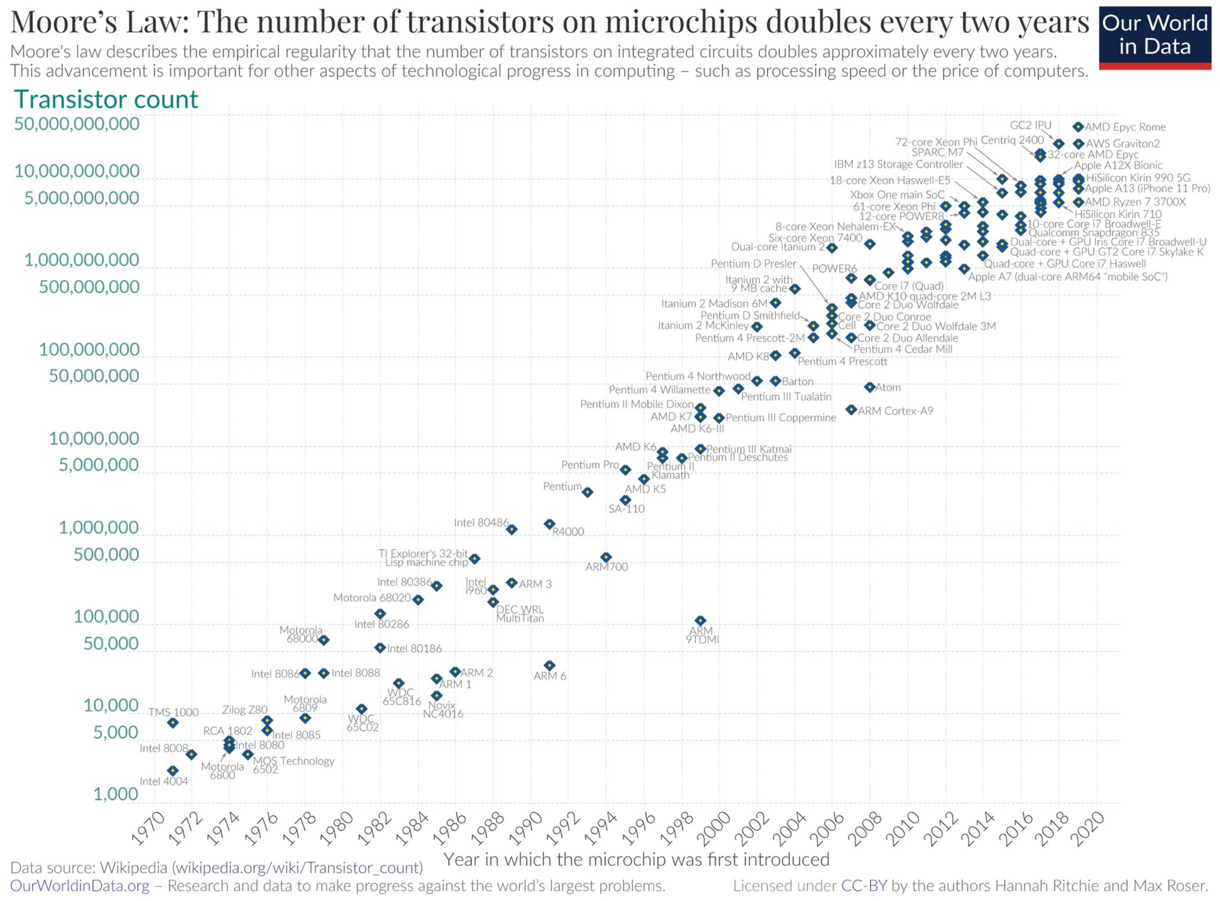
The two roles of computational in physics research

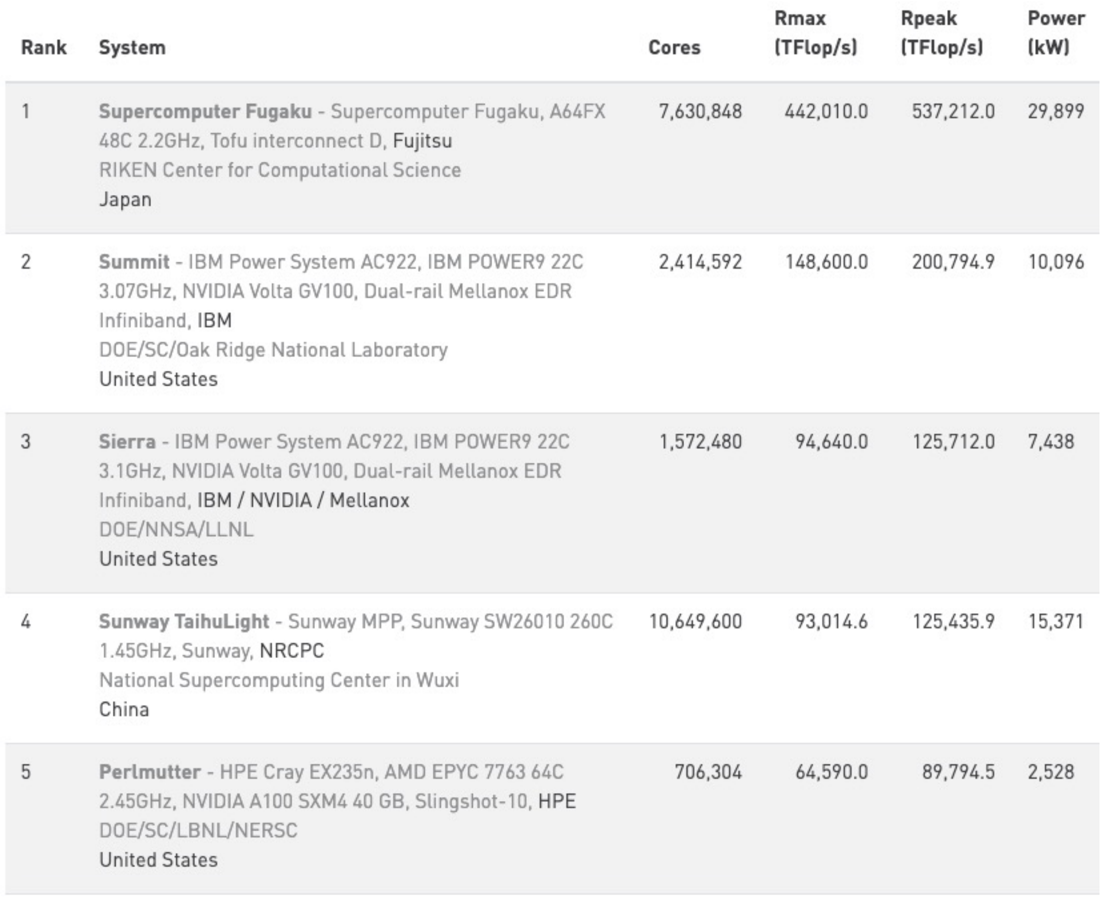


- Calculation: Using computers to solve well-defined problems

- Simulation: Use the computer to perform computational experiments

Computational science is driven by **more powerful computers**





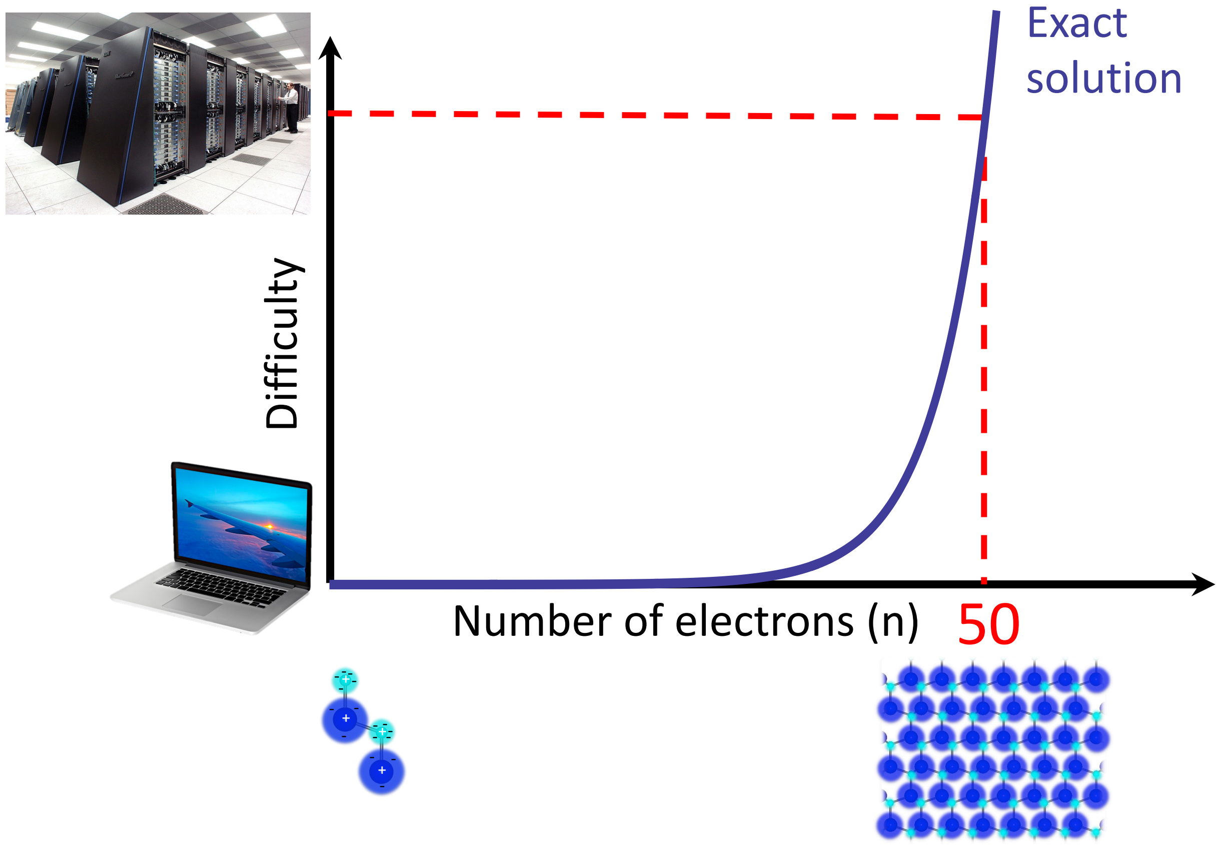
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	706,304	64,590.0	89,794.5	2,528

Computational science is driven by **better methods/algorithms**

$$\hat{H} = -\frac{\hbar}{2m_e}\sum_i \nabla_i^2 - \sum_{i,I}\frac{Z_I e^2}{|\mathbf{r}_i - \mathbf{R}_I|} + \frac{1}{2}\sum_{i\neq j}\frac{e^2}{|\mathbf{r}_i - \mathbf{r}_j|} - \sum_I \frac{\hbar^2}{2M_I}\nabla_I^2 + \frac{1}{2}\sum_{I\neq J}\frac{Z_I Z_J}{|\mathbf{R}_I - \mathbf{R}_J|}$$

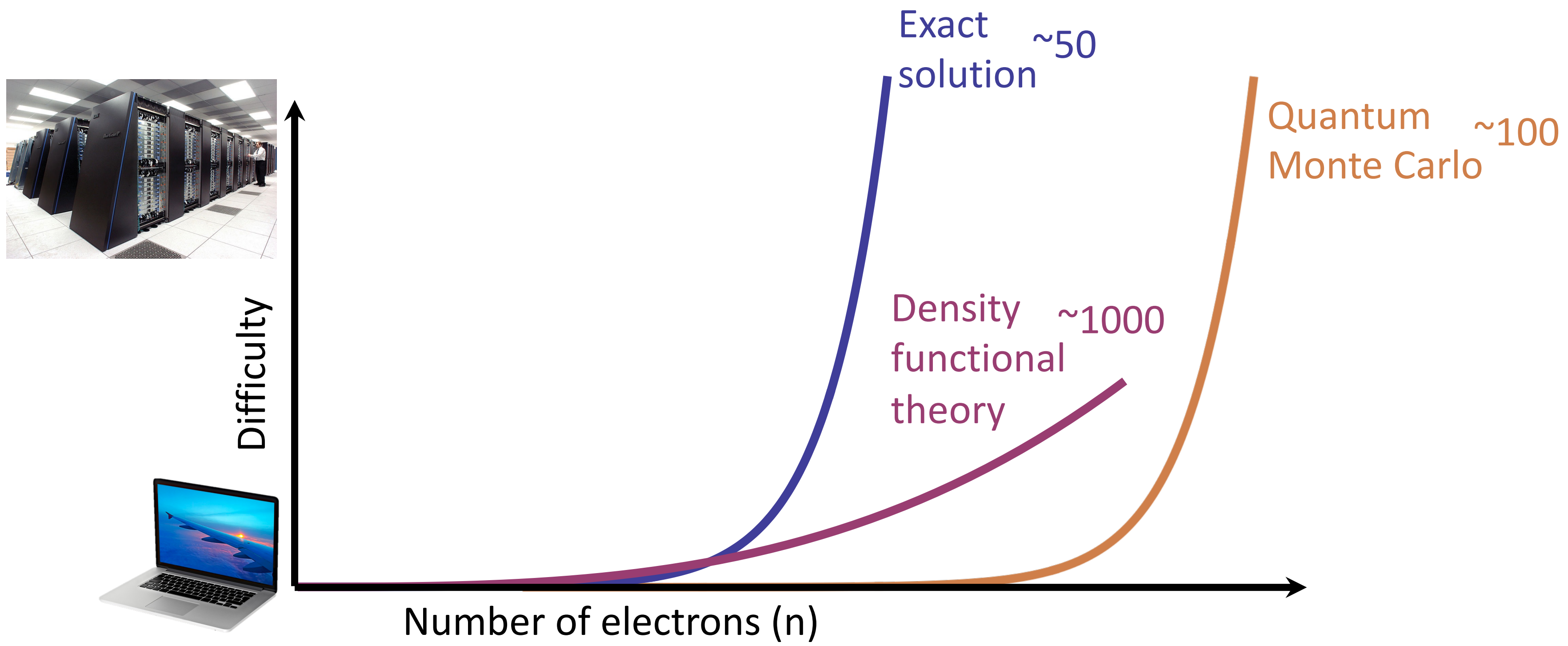
Computational science is driven by **better methods/algorithms**

$$\hat{H} = -\frac{\hbar}{2m_e}\sum_i \nabla_i^2 - \sum_{i,I}\frac{Z_I e^2}{|\mathbf{r}_i - \mathbf{R}_I|} + \frac{1}{2}\sum_{i\neq j}\frac{e^2}{|\mathbf{r}_i - \mathbf{r}_j|} - \sum_I \frac{\hbar^2}{2M_I}\nabla_I^2 + \frac{1}{2}\sum_{I\neq J}\frac{Z_I Z_J}{|\mathbf{R}_I - \mathbf{R}_J|}$$



Computational science is driven by **better methods/algorithms**

$$\hat{H} = -\frac{\hbar}{2m_e}\sum_i \nabla_i^2 - \sum_{i,I}\frac{Z_I e^2}{|\mathbf{r}_i - \mathbf{R}_I|} + \frac{1}{2}\sum_{i\neq j}\frac{e^2}{|\mathbf{r}_i - \mathbf{r}_j|} - \sum_I \frac{\hbar^2}{2M_I}\nabla_I^2 + \frac{1}{2}\sum_{I\neq J}\frac{Z_I Z_J}{|\mathbf{R}_I - \mathbf{R}_J|}$$



Goals for (the rest of) this lecture

- Representing numbers on the computer
    - Types
    - Finite precision of floating points
    - Comparing real numbers

Information in computer programs categorized by "**Type**"

C++ Type	**Fortran Equivalent**	**Description**	**Example**
`short` (also called **`short int`**)	**`integer(4)`**	Positive or negative number with no decimal places.	`56478, 3, -278`
`int`	**`integer`**		
`long` (also called **`long int`**)	**`integer(8)`**		
`float`	**`real`**	Positive or negative number with decimal places.	`3.0, 1.67e10, -3.2234e-20`
`double`	**`real(8)`**		
`long double`	**`real(16)`**		
`char`	**`character(1)`**	Single or multiple letters, numbers, symbols with no special interpretation	`a, abj3a, gh_&w`
`string` (string type implemented as a container in C++ standard library)	**`character(len=*)`** (as of Fortran 2008 standard)		
`bool`	**`logical`**	True or False	`.True., False`
`complex` (complex type implemented as a Template class in C++ standard library)	**`complex`**	Complex numbers	`3.0+5.6i`

All information in a computer stored as **bits**

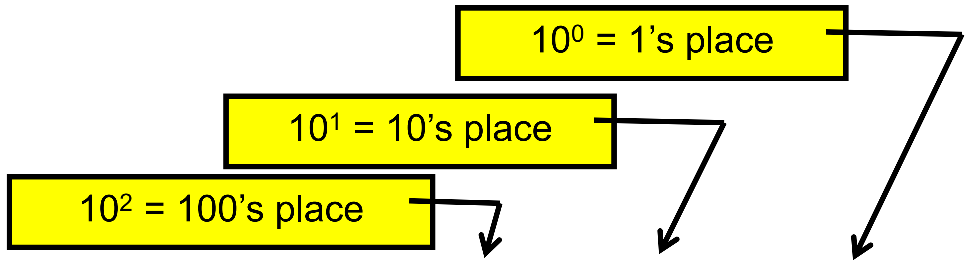
- Basic unit of information in a computer is a bit: **0 or 1**
    - 8 bits = 1 byte

- All types must be converted into some number of bytes

- Finite storage limits, e.g., the size or precision of a number

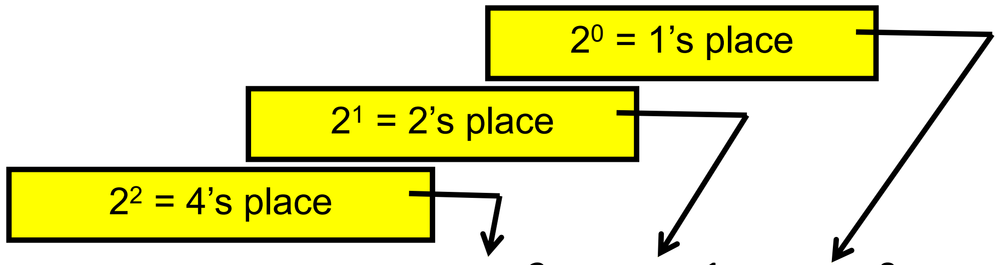
Binary data representation

- "Human" representation: Base ten (decimal)
  - Each digit multiplies a power of 10



$$137_{10} = 1\text{x}10^2 + 3\text{x}10^1 + 7\text{x}10^0$$

- "Computer" representation: Base two (binary)
  - Each digit multiplies a power of 2:

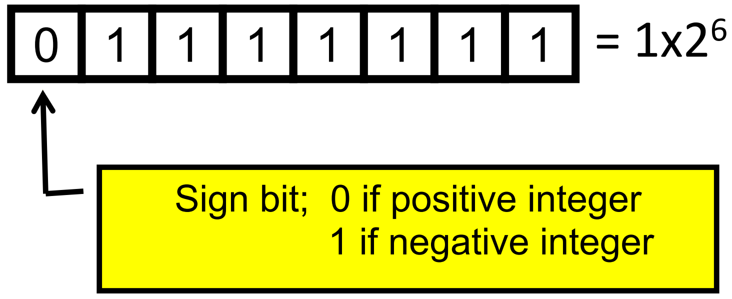


$$101_2 = 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$$

$= 5_{10}$ (in base 10 representation)

The amount of memory allocated to an integer determines largest number that can be stored

- E.g., 1 byte:



$= 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 1x2^2 + 1x2^1 + 1x2^0 = 127_{10}$

- 2-byte integer allows for 2^{16}-1 distinct values
  - This can store: -32,768 to 32,767 (signed)
  - Or it can store: 0 to 65,535 (unsigned)

- Standard in many languages is 4-bytes. This allows for 2^{32}-1 distinct values
  - This can store: -2,147,483,648 to 2,147,483,647 (signed)
    - C/C++: int (usually) or int32_t
    - Fortran: integer or integer(4)
  - Or it can store: 0 to 4,294,967,295 (unsigned)
    - C/C++: uint or uint32_t
    - Fortran (as of 95): unsigned

- For very big integers, 8-byte allows for 2^{64}-1
  - Fotran: integer(8)
  - C++: long

Overflow: Trying to put more information in a type than will fit

- What happens when you try to store an integer that too large for the memory allocated?
  - **Depends on the language!**

Overflow: Trying to put more information in a type than will fit

- What happens when you try to store an integer that too large for the memory allocated?
  - **Depends on the language!**

- Fortran: Just gives you the wrong result

Overflow: Trying to put more information in a type than will fit

- What happens when you try to store an integer that too large for the memory allocated?
  - **Depends on the language!**

- Fortran: Just gives you the wrong result

- Python: Allows the size of the integer to scale with the size of the number

Another aspect of integers to keep in mind: Integer division

- Multiplication of integers results in an integer; addition/subtraction of integers result in an integer; division of integers does not always result in an integer!!

- What happens if we divide two integers like: `1/2`?

Another aspect of integers to keep in mind: **Integer division**

- Multiplication of integers results in an integer; addition/subtraction of integers result in an integer; division of integers does not always result in an integer!

- What happens if we divide two integers like: 1 / 2?
  - In some codes, 1/2 gives 0, in others it converts to real and give 0.5
  - Common source of bugs!!

Real/Floating point numbers are more complicated

- Infinite real numbers on the number line need to be represented by a finite number of bits
- Finite memory results in limited **size and precision** of floating point numbers
  - Not all real numbers (even simple ones) can be stored in a finite number of digits in a base-2 representation
  - Example: $1/10=0.1_{10} = 0.0001100110011..._2$ does not have a finite representation in base 2 just as $1/3=0.333333..._{10}$ has no finite representation in base 10
- This means that even simple floating point numbers are often approximated with some small error
  - This means that floating point arithmetic is not exact! (on all computers and programming languages)
- Errors can compound if not treated carefully!

Real (a.k.a. floating point) data

- IEEE 754 mantissa-exponent form:

$$\underbrace{\text{mmmmmmmmmmmmmmmmmmmmmmmm}}_{\boxed{\text{24 bit mantissa}}}\underbrace{\text{eeeeeeee}}_{\boxed{\text{8 bit exponent}}}$$

- Value = mantissa x 2 $^{\text{exponent}}$
- Single precision:
  - Sign: 1 bit; exponent: 8 bits; significand: 24 bits (23 stored) = 32 bits
  - Range: 2^7-1 in exponent (because of sign) = 2^{127} multiplier ~ 10^{38}
  - Decimal precision: ~6 significant digits
- Double precision:
  - Sign: 1 bit; exponent: 11 bits; significand: 53 bits (52 stored) = 64 bits
  - Range: 2^{10}-1 in exponent = 2^{1023} multiplier ~ 10^{308}
  - Decimal precision: ~15 significant digits

Finite precision of floating points

- This means that most real numbers do not have an exact representation on a computer.
  - Spacing between numbers varies with the size of numbers
  - Relative spacing is constant

$$\text{relative roundoff error} = \frac{|\text{true number} - \text{computer number}|}{|\text{true number}|} \leq \epsilon$$

Overflows/underflows with reals

- Overflows and underflows can still occur when you go outside the representable range.
  - The floating-point standard will signal these (and compilers can catch them)
- Some special numbers:
  - `NaN` = 0/0 or $\sqrt{-1}$
  - `Inf` is for overflows, like 1/0
  - Both of these allow the program to continue, and both can be trapped (and dealt with)
- `-0` is a valid number, and `-0 = 0` in comparison
- Floating point is governed by an IEEE standard
  - Ensures all machines do the same thing
  - Aggressive compiler optimizations can break the standard

A result of finite precision: Need to be careful when comparing floats/reals

- Floating point numbers involve rounding and imprecision, which propagate in different ways under different operations

- Mathematically analogous expressions may yield slightly (or significantly as we will see!) different results

- In principle, this can be accounted for since floating point operations follow specific rules
    - see reading "What Every Computer Scientist Should Know About Floating-Point Arithmetic," by David Goldberg

- In practice, it best to do an "epsilon check"

Epsilon check for comparing floats

- Take two real numbers `a` and `b`
- We take `a==b` if `abs(a-b) < epsilon`

- Have to be very careful with this!!! We should think about:
  - The choice of `epsilon` based on the precision we require/expect for `a` and `b`
  - The choice of `epsilon` based on the magnitude of `a` and `b`
  - What will happen in special cases (`0, NaN, inf`)
  - …

Round-off error example

- Imagine that we can only keep track of 4 significant digits
- Compute $\sqrt{x+1} - \sqrt{x}$
- Take *x* = 1984. Keeping only 4 digits each step of the way:

$$\sqrt{x+1} - \sqrt{x} = 44.55 - 44.54$$

- We've lost a lot of precision
- Instead, consider:

$$\sqrt{x+1} - \sqrt{x} = (\sqrt{x+1} - \sqrt{x}) \left(\frac{\sqrt{x+1} + \sqrt{x}}{\sqrt{x+1} + \sqrt{x}} \right) = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

- Then

$$\sqrt{1985} - \sqrt{1984} = \frac{1}{\sqrt{1985} + \sqrt{1984}} = \frac{1}{44.55 + 44.54} = 0.01122$$

Roundoff error: Another example

- Consider computing exp(-24) via a truncated Taylor series:

$$e^x \simeq S(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!}$$

- Error in the approximation (**i.e., truncation error**) is less than:

$$\frac{|x|^{n+1}}{(n+1)!} \max\{1, e^x\}$$

- But if we compute S(-24) by adding terms until they are less than machine precision (8 byte):
  - S(-24)=3.7814382919759864E-007
  - Exp(-24)=3.7751345442790977E-011
  - Error is larger than the result (much larger than truncation error)!!
  - Looking at terms, we see we are relying on cancellations of terms

How can we make is more accurate? Choose a different algorithm

- Realize that:

$$e^{-24} = (e^{-1})^{24} \Rightarrow S(-24) = S(-1)^24$$

- $S(-1)^{24}$ = 3.7751345442791294E-011
- exp(-24) = 3.7751345442790977E-011

Truncation errors are different from roundoff

- Translating continuous mathematical expressions into discrete forms introduces truncation error
- For example: $e^x \simeq S(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}$
- Error: $\frac{|x|^{n+1}}{(n+1)!} \max\{1, e^x\}$
- Or $f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$ vs. $D_h(x) = \frac{f(x+h) - f(x)}{h}$

Floating point arithmetic not associative

- Adding lots of numbers together can compound round-off error

- One solution: sort and add starting with the smallest numbers

- Kahan summation (see reading list)
  - Algorithm for adding sequence of numbers while minimizing roundoff accumulation
  - Keeps a separate variable that accumulates small errors
  - Requires that the compiler obey parenthesis

Floating point arithmetic not associative:

```
! Purpose:  Test the precision of reals
! Author:    Cyrus Dreyer
! Date:         2/4/2019
program test_prec_reals
  implicit none         ! Turn off implicit typing
  ! Variable dictionary
  real :: factor1     ! Variable for factor 1
  real :: factor2     ! Variable for factor 2
  real :: prec_test_lhs ! Variable for result
  real :: prec_test_rhs ! Variable for result

  factor1 = 1.0           ! Assign a value to factor1
  factor2 = 1.0d-9        ! Assign a value to factor2

  prec_test_lhs = (factor1-factor1) + factor2 ! LHS of inequality on slide
  prec_test_rhs = factor1 + (-factor1 + factor2) ! RHS of inequality on slide

  ! Output
  write(*,'(a20,e20.12e2,a20,e20.12e2)') "Prec_test_lhs:",prec_test_lhs, &
       "Prec_test_rhs:", prec_test_rhs

  stop 0                ! Stop execution of the program
end program test_prec_reals
```

Integers versus real/floating-point: Mind the decimal place

- Keep in mind that in many languages, 2 and `2.0` (or `2.`) are interpreted differently
  - 2 is taken to be an integer
  - `2.` or `2.0` is taken to be real
- This can be important in a variety of contexts:
  - Integers are stored exactly, floats are approximate
  - Integer and floating-point division are different
  - Exponentiating negative numbers can be problematic `(–3.5)**(2)` is safer than `(–3.5)**(2.0)`
  - Integers must be used for, e.g., array indices

After class tasks

- Readings:
  - What every computer scientist should know about floating-point arithmetic
  - Wikipedia page on the Floating Point
  - Wikipedia page on the Kahan Summation Algorithm