# 2 Lecture 2, Jan 7

## Announcements

- Location change: class meeting is changed to CHS 51-279, effective Jan 7.
- Enrollment cap is raised to 38 (open).
- Use RStudio for R programming.

## Last time

- Introduction. Gauss (least squares to find Ceres)–optimization, Marc Coram (decipher a note circulating in jail)–sampling. Two major modes of statistical computing.
- Course content, logistics.
- Computer representation of characters (ASCII, unicode) and integers (fixed point number system).

## Today

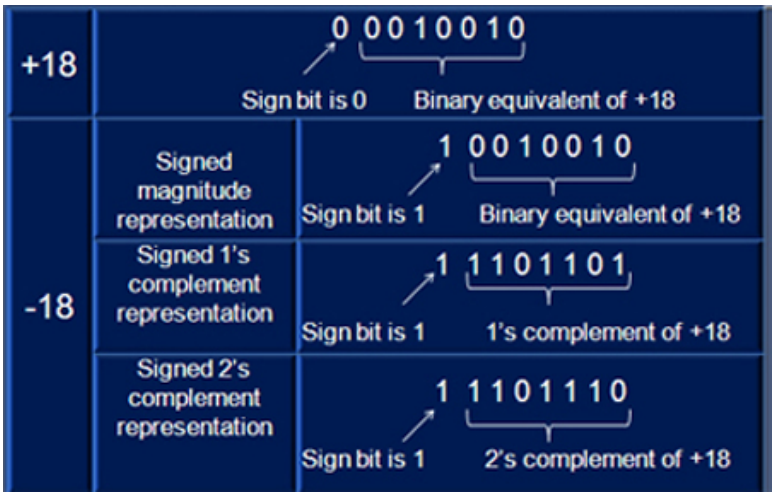
- Computer representation and arithmetic of integers: fixed-point numbers.
- Computer representation and arithmetic of real numbers: floating-point numbers.

## Fixed-point number system

Fixed-point number system $\mathbb{I}$ is a computer model for integers $\mathbb{Z}$. One storage unit may be $M = 8/16/32/64$ bit.

- The number of bits and method of representing negative numbers vary from system to system. The `integer` type in R has $M = 32$ bits. MATLAB has (u)int8, (u)int16, (u)int32, (u)int64. JULIA has even more choices such as `Int128`, `UInt128`, `BigInt`, ...
- First bit indicates sign: 0 for nonnegative numbers, 1 for negative numbers.

- "two's complement representation" for negative numbers. (i) Sign bit is set to 1, (ii) remaining bits are set to opposite values, (iii) 1 is added to the result.

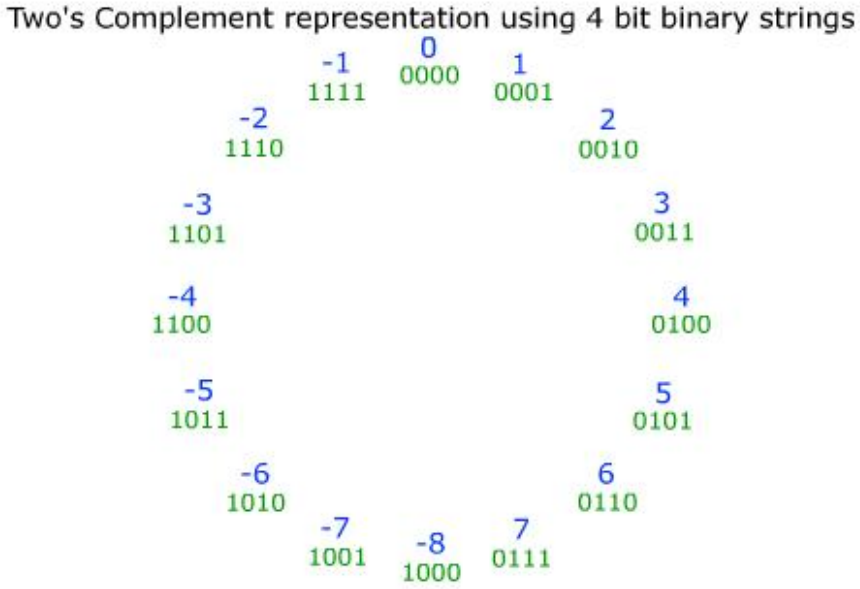



- Range of representable integers by $M$-bit storage is $[-2^{M-1}, 2^{M-1}-1]$ (don't need to represent 0 anymore so *could* have capacity for $2^{M-1}$ negative numbers).

- For $M = 8$, $[-128, 127]$.
  For $M = 16$, $[-65536, 65535]$.
  For $M = 32$, $[-2147483648, 2147483647]$.

- The smallest representable integer in R is $-2^{31} + 1 = -2147483647$.

- For unsigned integers such as in MATLAB, the range is $[0, 2^M - 1]$.




- An integer $-i$ in the interval $[-2^{M-1}, -1]$ would be represented by the same bit pattern by which the nonnegative integer $2^M - i$ is represented, treating the sign bit as a regular numeric bit. For example, (-18) = 11101110, $(2^8) - (18)$=(238)=11101110. Two's complement is subtracting the nonnegative integer $i$ from $1 \ldots 1 + 1 = 10 \ldots 0 = (2^M)$.

- Addition and subtraction are much simpler in two's complement representation. Why? It respects modular arithmetic nicely (look at the diagram). E.g., (3)+(4) = 0011 + 0100 = 0111 = (7).

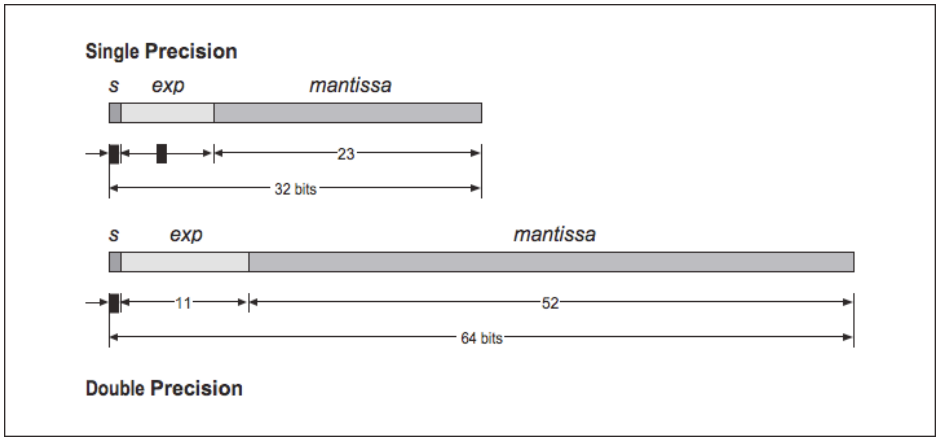
- Keep track of overflow and underflow. If the result of a summation is $R$, which

must be in the set $[-2^{M-1}, 2^{M-1}+1]$, there are only three possibilities for the true sum: $R$, $R+2^M$ (overflow), or $R-2^M$ (underflow).

  - When adding two nonnegative integers: 0XX...X + 0YY...Y, where X and Y are arbitrary binary digits, we only need to keep track of overflow in this case. If the resulting binary number has a leading bit of 1, we know overflow occurs since the sum cannot be negative. We should treat that sign bit as a regular numeric bit. In other words, we crossed the upper boundary 100 and should add $2^M$ to the result. If the resulting binary number has a leading bit of 0, no overflow occurs.
  - When adding two negative integers: 1XX...X + 1YY...Y, where X and Y are arbitrary binary digits, we only need to keep track of underflow in this case. If the resulting binary number has a leading bit of 0, we know underflow occurs since the sum cannot be positive. We crossed the lower boundary 000 and should subtract $2^M$ from the result. If the resulting binary number has a leading bit of 1, no underflow occurs.
  - When adding a negative integer and a nonnegative integer: 1XX...X + 0YY...Y, the result is always between the two summands. No overflow or underflow could happen.

- R reports `NA` for integer overflow and underflow. Other languages, e.g., Julia simply outputs the result according modular arithmetic.

## Floating-point number system




Floating-point number system $\mathbb{F}$ is a computer model for real numbers $\mathbb{R}$.

- A real number is represented by $\pm d_0.d_1d_2\cdots d_p \times b^e$ (scientific notation).
- Parameters for a floating-point number system: *base* (or *radix*), range of *fraction* (or *mantissa*, *significand*), range of *exponent*.
- Non-uniqueness of representation. *normalized/denormalized* significant digits. E.g., $+18 = +1.0010 \times 2^4$(normalied) $= +0.10010 \times 2^5$(denormalized).
- *Bias* (or *excess*): actual exponent is obtained by subtracting bias from the value of exponent evaluated regardless of sign digit.
- IEEE 754.
  - *Single precision* (32 bit): base 2, $p = 23$ (23 significant bits), $e_{\max} = 127$, $e_{\min} = -126$ (8 exponent bits), bias=127. $e_{\min} - 1$ and $e_{\max} + 1$ are reserved for special numbers. This implies a maximum magnitude of $\log_{10}(2^{127}) \approx 38$ and precision to $\log_{10}(2^{23}) \approx 7$ decimal point. $\pm 10^{\pm 38}$.
  - *Double precision* (64 bit): base 2, $p = 52$ (52 significant bits), $e_{\max} = 1023$, $e_{\min} = -1022$ (11 exponent bits), bias=1023. This implies a maximum magnitude of $\log_{10}(2^{1023}) \approx 308$ and precision to $\log_{10}(2^{52}) \approx 16$ decimal point. $\pm 10^{\pm 308}$.

- "$(+18) = (2^4 + 2^1) = +1.0010 \times 2^4$ in single precision

  $$(0)(10000011)(00100000000000000000).$$

  First is sign bit. Next 8 bits are exponent 131 in ordinary base 2 with a bias of 127. Remaining 23 bits represent the fraction beyond the leading bit, known to be 1. In summary it represents $(+18)$ as $+1.0010 \times 2^4$ in the binary format. $(-18)$ is represented by the same bits except changing the sign bit to 1.

- Special floating-point numbers:
  - Exponent $e_{\max} + 1$ plus a mantissa of 0 means $\pm\infty$.
  - Exponent $e_{\max} + 1$ plus a nonzero mantissa means NaN. NaN could be produced from `0 / 0`, `0 * Inf`, ... In general $NaN \neq NaN$ bitwise.
  - Exponent $e_{\min} - 1$ with a mantissa of all 0s represents the real number 0.
  - Exponent $e_{\min} - 1$ with a nonzero mantissa are for numbers less than $b^{e_{\min}}$. Numbers are de-normalized in the range $(0, b^{e_{\min}})$ – "graceful underflow".

- $\mathbb{F}$ is not a subset of $\mathbb{R}$, although $\mathbb{I} \subset \mathbb{Z}$.

- For the history of establishment of IEEE-754 standard, see an interview with William Kahan.
  `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`

To summarize

- Single precision: $\pm 10^{\pm 38}$ with precision up to 7 decimal digits.

- Double precision: $\pm 10^{\pm 308}$ with precision up to 16 decimal digits.

- Irrational numbers such as $\pi$ do not exist in $\mathbb{F}$.

- Exercise: what is the floating point representation of the number 0.1?

- The floating-point numbers do not occur uniformly over the real number line.

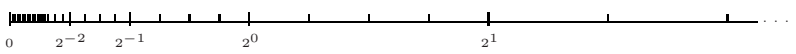



**Fig. 2.4.** The Floating-Point Number Line, Nonnegative Half

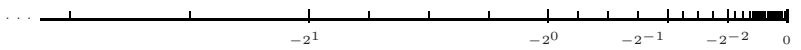



**Fig. 2.5.** The Floating-Point Number Line, Nonpositive Half

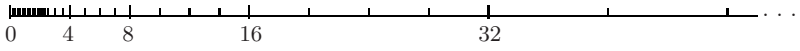



**Fig. 2.6.** The Floating-Point Number Line, Nonnegative Half; Another View

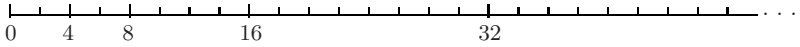



**Fig. 2.7.** The Fixed-Point Number Line, Nonnegative Half

- *Machine epsilons* are the spacings of numbers around 1. $\epsilon_{\min} = b^{-p}$ and $\epsilon_{\max} = b^{1-p}$.

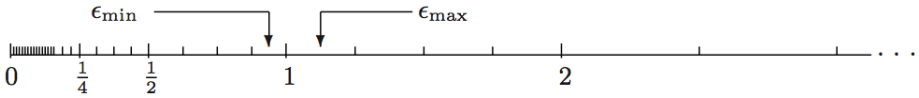



**Fig. 2.8.** Relative Spacings at 1: "Machine Epsilons"

- The variable `.Machine` in R contains numerical characteristics of the machine.
- How to test `inf` and `nan`? In R, `is.nan()`, `is.finite()`, `is.infinite()`. In MATLAB, `isinf()`, `isnan()`.
- R only uses double (64-bit) and 32-bit integer. It can be a downside when dealing with big data.

## Integers and floating-point numbers in Julia

Julia offers a rich collection of primitive data types. Read (the excellent) documentation `http://docs.julialang.org/en/release-0.4/manual/integers-and-floating-point-numbe`

Notable things include: `Int128`, `UInt128`, half precision numbers `Float16`, arbitrary precision numbers `BigInt` and `BigFloat`, rational numbers, ...

## Consequences of computer storage/arithmetic

- Be memory conscious when dealing with big data. E.g., human genome has about $3 \times 10^9$ bases, each of which belongs to $\{A, C, T, G\}$. How much storage if we store $10^6$ SNPs (single nucleotide polymorphisms) of 1000 individuals (1000 Genome Project) as characters (1GB), single (4GB), double (8GB), int32 (4GB), int16 (2GB), int8 (1GB), PLINK binary format 2bit/SNP (250MB)?

- Know the limit. *Overflow* and *Underflow*. For double precision, $\pm 10^{\pm 308}$. In most situations, underflow is preferred over overflow. Overflow often causes crashes. Underflow yields zeros. E.g., in logistic regression, $p_i = \frac{\exp(\boldsymbol{x}_i^T \boldsymbol{\beta})}{1+\exp(\boldsymbol{x}_i^T \boldsymbol{\beta})} = \frac{1}{1+\exp(-\boldsymbol{x}_i^T \boldsymbol{\beta})}$. The former expression can easily lead to $\infty/\infty = NaN$, while the latter expression leads to *graceful underflow*.

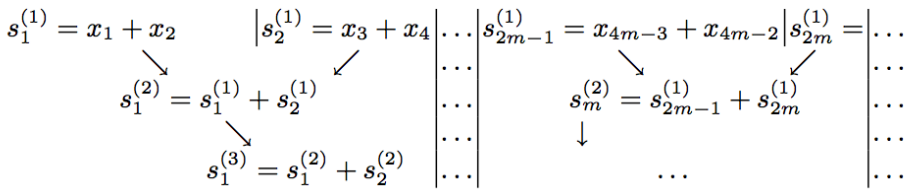
- Be aware of non-uniform distribution of floating-point numbers, in contrast to fixed-point numbers. There are the same number of floating-point numbers in $[b^i, b^{i+1}]$ and $[b^{i+1}, b^{i+2}]$ for $e_{\min} \le i \le e_{\max} - 2$. It is more dense when closer to zero.

- "Catastrophic cancellation 1". Addition or subtraction of two numbers of widely different magnitudes: $a + b$ or $a - b$ where $a \gg b$ or $a \ll b$. We loose the precision in the number of smaller magnitude. Consider $a = x.xxx... \times 2^0$ and $b = y.yyy... \times 2^{-53}$. What happens when computer calculates $a + b$? We get $a + b = a$!

- Another example: What happens when compute $\sum_{x=1}^{\infty} x$ in order? Will the partial sum reach `Inf`? "A divergent series converges."

- Always try to add numbers of similar magnitude. Rule 1: add small numbers together before adding larger ones. Rule 2: add numbers of like magnitude

together (paring). When all numbers are of same sign and similar magnitude, add in pairs so each stage the summands are of similar magnitude.



- "Catastrophic cancellation 2". Subtraction of two nearly equal numbers eliminates significant digits. $a - b$ where $a \approx b$. Consider $a = x.xxxxxxxxxx1ssss$, $b = x.xxxxxxxxxx0tttt$. The result is $1.vvvvu...u$ where $u$ are unassigned digits.

- E.g., evaluating $e^{-20}$ by Taylor series

$$e^{-x} = 1 - x + x^2/2! - x^3/3! + \cdots$$

  gets `6.138e-09`, while the true value is about `2.061e-09`. Many cancellations accumulate. Anyway, it's *not* the way to evaluate $e^x$!

- In general floating-point arithemetics may violate the associative and distributive laws.

- Sometimes catastrophic cancellation can be avoided. Roots of the quadratic function $ax^2 + bx + c$ are

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

  When one root is close to 0, cancellation can happen. We may evaluate one of the root (away from 0) by the formula and then compute the other by relationship $x_1 x_2 = c/a$.

## Algorithms

- *Algorithm* is loosely defined as a set of instructions for doing something. Input $\rightarrow$ Output.

- **?**: (1) finiteness, (2) definiteness, (3) input, (4) output, (5) effectiveness

- Basic unit for measuring efficiency is flop. A *flop* (floating point operation) consists of a floating point multiply (or divide) and the usually accompanying addition, fetch and store. Some books such as **?** and **?** consider addition as a separate flop.

- How to measure efficiency of an algorithm? Big O notation. If $n$ is the size of a problem, an algorithm has order $O(f(n))$, where the leading term in the number of flops is $c \cdot f(n)$.

- E.g., matrix-vector multiplication `A %*% b`, where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$, takes $O(mn)$ flops. Matrix-matrix multiplication `A %*% B`, where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times p}$, takes $O(mnp)$ flops.

- *Exponential* order $O(b^n)$ (NP-hard="horrible"), *polynomial* order $O(n^q)$ (doable), $O(n \log n)$ (fast), linear order $O(n)$ (fast), *log* order $O(\log n)$ (super fast).

- One goal of this course is to get familiar with the flop counts for some common numerical tasks in statistics.

  > The form of a mathematical expression and the way the expression should be evaluated in actual practice may be quite different.

- Compare flops of the following two expressions:

  ```
  G %*% Xt %*% y
  G %*% (Xt %*% y)
  ```

  where `G` $\in \mathbb{R}^{p \times p}$, `Xt` $\in \mathbb{R}^{p \times n}$, and `y` $\in \mathbb{R}^n$. "Matrix multiplication is expensive."

- Hardware advancement, e.g., CPU clock rate, only affects constant $c$. Unfortunately, data size $n$ is increasing too and often at a faster rate.

- Classification of data sets by **??**.

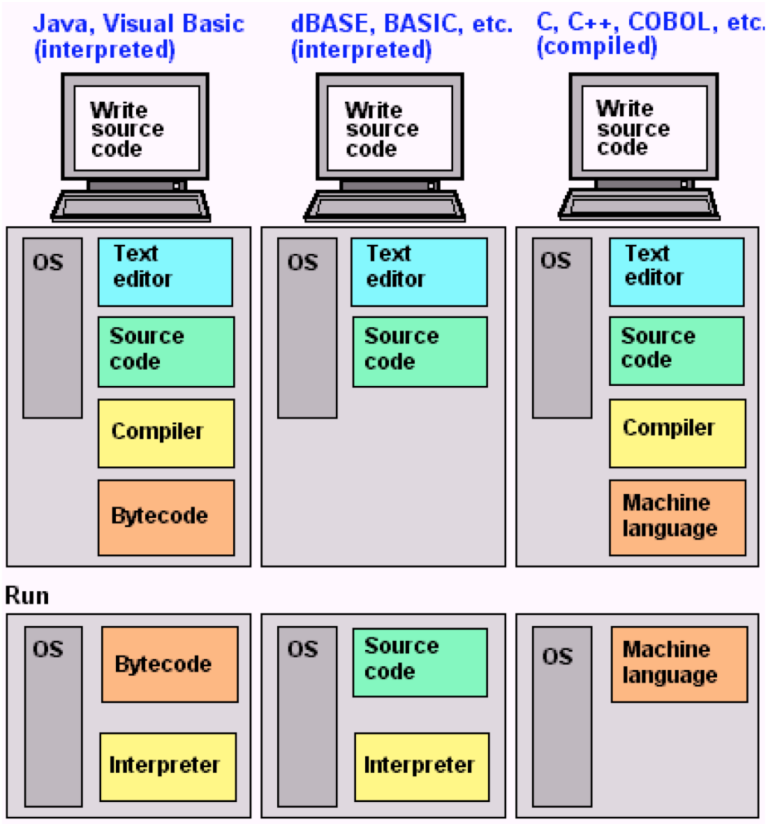
| Data Size | Bytes | Storage Mode |
|---|---|---|
| Tiny | $10^2$ | Piece of paper |
| Small | $10^4$ | A few pieces of paper |
| Medium | $10^6$ (megabyte) | A floppy disk |
| Large | $10^8$ | Hard disk |
| Huge | $10^9$ (gigabytes) | Hard disk(s) |
| Massive | $10^{12}$ (terabytes) | RAID storage |

- Difference of $O(n^2)$ and $O(n \log n)$ on massive data. Suppose we have a teraflop supercomputer capable of doing $10^{12}$ flops per second. For a problem of size $n = 10^{12}$, $O(n \log n)$ algorithm takes about $10^{12} \log(10^{12})/10^{12} \approx 27$ seconds. $O(n^2)$ algorithm takes about $10^{12}$ seconds, which is approximately 31710 years!
- QuickSort and FFT are celebrated algorithms that turn $O(n^2)$ operations into $O(n \log n)$. *Divide-and-conquer* is a powerful technique. Another example is the Strassen's method, which turns $O(n^3)$ matrix multiplication into $O(n^{\log_2 7})$.

## Computer Languages

> **工欲善其事，必先利其器**
>
> *To do a good job, an artisan needs the best tools.*
>
> *The Analects* by Confucius (about 500 BC)

- What features are we looking for in a language?
  - Efficiency (in both run time and memory) for handling big data.
  - IDE support (debugging, profiling).
  - Open source.
  - Legacy code.
  - Tools for generating dynamic report (reproducibility).
  - Adaptivity to hardware evolution (parallel and distributed computing).

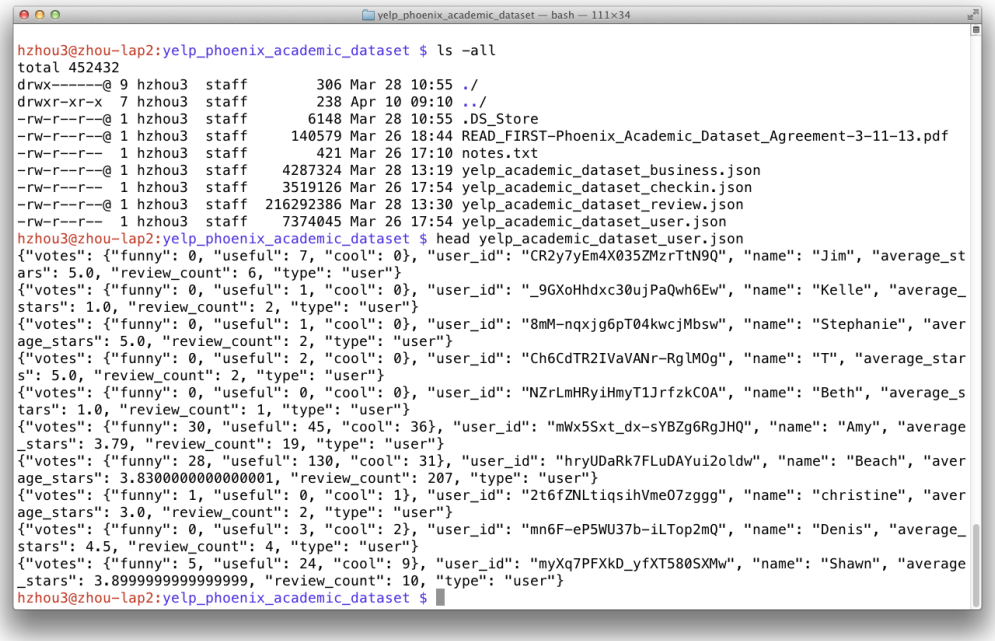



- Types of languages

  1. Compiled languages: C/C++, Fortran, ...
     - Directly compiled to machine code that is executed by CPU
     - Pros: fast, memory efficient
     - Cons: longer development time, hard to debug
  2. Interpreted language: R, Matlab, SAS IML, JavaScript, ...
     - Interpreted by interpreter
     - Pros: fast prototyping
     - Cons: excruciatingly slow for loops
  3. Mixed (dynamic) languages: Julia, Python, JAVA, Matlab (JIT), R (JIT), ...
     - Compiled to bytecode and then interpreted by virtual machine

- Pros: relatively short development time, cross-platform, good at data preprocessing and manipulation, rich libraries and modules
- Cons: not as fast as compiled language

4. Script languages: shell scripts, Perl, ...
   - Extremely useful for data preprocessing and manipulation
   - E.g., massage the Yelp (`http://www.yelp.com/dataset_challenge`) data before analysis




5. Database language: SQL, Hadoop. Data analysis never happens if we do not know how to retrieve data from databases.

- Messages
  - To be versatile in the big data era, be proficient in at least one language in each category.
  - To improve efficiency of interpreted languages such as R or MATLAB, avoid loops as much as possible. Aka, vectorize code
    "The only loop you are allowed to have is that for an iterative algorithm."

For some tasks where looping is necessary, consider coding in C or Fortran. It is "convenient" to incorporate compiled code into R or Matlab. But do this **only after profiling!**
Success stories: `glmnet` and `lars` packages in R are based on Fortran.

- Julia is trying to bridge the gap between interpreted and compiled languages. That is to achieve efficiency without vectorizing code.

- Language features of R, Matlab, and Julia

| Features | R | Matlab | julia |
|---|---|---|---|
| Open source | ☺ | ☹ | ☺ |
| IDE | R Studio ☺☺ | ☺☺☺ | ☹ |
| Dynamic document | ☺☺☺(RMarkdown) | ☺☺☺ | ☺☺☺(IJulia) |
| Multi-threading | `parallel` pkg | ☺ | ☺ |
| JIT | `compiler` pkg | ☺ | ☺ |
| Call C/Fortran | wrapper | wrapper | no glue code |
| Call shared library | wrapper | wrapper | no glue code |
| Typing | ☹ | ☺☺ | ☺☺☺ |
| Pass by reference | ☹ | ☹ | ☺☺☺ |
| Linear algebra | ☹ | MKL, Arpack | OpenBLAS, Eigpack |
| Distributed computing | ☹ | ☺ | ☺☺☺ |
| Sparse linear algebra | ☹ (`Matrix` package) | ☺☺☺ | ☺☺☺ |
| Documentation | ☹ | ☺☺☺ | ☺☺ |
| Profiler | ☹ | ☺☺☺ | ☺☺☺ |

- Benchmark code `R-benchmark-25.R` from `http://r.research.att.com/benchmarks/R-benchmark-25.R` covers many commonly used numerical operations used in statistics. We ported (literally) to Matlab and Julia and report the run times (averaged over 5 runs) here.

  Matlab benchmark code: `http://hua-zhou.github.io/teaching/biostatm280-2016winter/benchmark_matlab.m`
  Julia benchmark code: `http://hua-zhou.github.io/teaching/biostatm280-2016winter/benchmark_julia.jl`

Machine specs: Intel i7 @ 2.6GHz (4 physical cores, 8 threads), 16G RAM, Mac OS 10.11.2.

| Test | R 3.2.2 | MATLAB R2014a | JULIA 0.4.2 |
|---|---|---|---|
| Matrix creation, trans, deformation ($2500 \times 2500$) | 0.77 | 0.17 | **0.14** |
| Power of matrix ($2500 \times 2500$, $A.^{1000}$) | 0.26 | **0.11** | 0.21 |
| Quick sort ($n = 7 \times 10^6$) | 0.76 | **0.24** | 0.69 |
| Cross product ($2800 \times 2800$, $A^T A$) | 10.72 | 0.35 | **0.31** |
| LS solution ($n = p = 2000$) | 1.28 | **0.07** | 0.09 |
| FFT ($n = 2,400,000$) | 0.40 | **0.04** | 0.64 |
| Eigen-values ($600 \times 600$) | 0.82 | **0.31** | 0.57 |
| Determinant ($2500 \times 2500$) | 3.76 | 0.18 | **0.17** |
| Cholesky ($3000 \times 3000$) | 4.23 | **0.15** | 0.20 |
| Matrix inverse ($1600 \times 1600$) | 3.37 | **0.16** | 0.21 |
| Fibonacci (vector calc) | 0.34 | **0.17** | 0.68 |
| Hilbert (matrix calc) | 0.21 | 0.07 | **0.01** |
| GCD (recursion) | 0.31 | 0.14 | **0.12** |
| Toeplitz matrix (loops) | 0.36 | **0.0014** | 0.02 |
| Escoufiers (mixed) | 0.45 | 0.40 | **0.21** |

- A slightly more complicated (or realistic) example taken from Doug Bates's slides `http://www.stat.wisc.edu/~bates/JuliaForRProgrammers.pdf`. The task is to use Gibbs sampler to sample from bivariate density

$$f(x, y) = kx^2 \exp(-xy^2 - y^2 + 2y - 4x), x > 0,$$

using the conditional distributions

$$\begin{aligned} X|Y &\sim \Gamma\left(3, \frac{1}{y^2+4}\right) \\ Y|X &\sim \mathcal{N}\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right). \end{aligned}$$

Let's sample 10,000 points from this distribution with a thinning of 500.

  - How long does R take? 42 seconds.
    `http://hua-zhou.github.io/teaching/biostatm280-2016winter/gibbs_r.html`
  - How long does Julia take? 0.43 seconds.
    `http://hua-zhou.github.io/teaching/biostatm280-2016winter/gibbs_julia.html`

- With similar coding efforts, Julia offers $\sim$ 100 fold speed-up! JIT in R didn't kick in. Neither does Matlab, which took about 20 seconds.
- Julia offers the capability of strong typing of variables. This facilitates the optimization by compiler.
- With little efforts, we can do parallel and distributed computing using Julia.

Benchmark of the same example in other languages including Rcpp is available in the blogs by Darren Wilkinson (`http://bit.ly/IWhJ52`) and Dirk Eddelbuettel's (`http://dirk.eddelbuettel.com/blog/2011/07/14/`).

> *"As some of you may know, I have had a (rather late) mid-life crisis and run off with another language called Julia.* ***`http://julialang.org`*** *"*
>
> Doug Bates (on the `knitr` Google Group)