Today

Last time

- Condition of problems
- Stability of algorithms

Today

- Float-point numbers in IEEE format
- Rounding, propagation of errors, and cancellation
- Truncation errors
- Matlab recap

Announcements

- Homework 1 was posted last week; is due next week Mon, Sep 26 *before class*
- No office hour this week: Grader's office hour or send an email if you have questions.
- Next week, Professor Stadler will fill in

Recap: Condition of a problem

- Terms such as"little bit" and a "small amount" already point to that we need to measure something
- Therefore, we assume the map f is given as

$$f : U \subset \mathbb{R}^n \to \mathbb{R}^m$$

  and we are interested in the norm $\| \cdot \|$
- The input error is then

$$\|x - \hat{x}\| \leq \delta \text{ (absolute)} \qquad \|x - \hat{x}\| \leq \delta \|x\| \text{ (relative)}$$

- Correspondingly we measure the output error $f(x) - f(\hat{x})$ in $\| \cdot \|$ (we could also have looked at a componentwise error)

Recap: Condition of a problem (cont'd)

- Absolute condition number at x is

$$\kappa_{\text{abs}} = \lim_{\delta \to 0} \sup_{\|x - \hat{x}\| \le \delta} \frac{\|f(x) - f(\hat{x})\|}{\|x - \hat{x}\|}$$

- Relative condition number at x is

$$\kappa_{\text{rel}} = \lim_{\delta \to 0} \sup_{\|x - \hat{x}\| \le \delta} \frac{\|f(x) - f(\hat{x})\| / \|f(x)\|}{\|x - \hat{x}\| / \|x\|}$$

- If f is differentiable in x, then

$$\kappa_{\text{abs}} = \|f'(x)\| \qquad \kappa_{\text{rel}} = \frac{\|x\|}{\|f(x)\|} \|f'(x)\| ,$$

where $\|f'(x)\|$ is the norm of the Jacobian $f'(x)$ in the operator norm

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \sup_{\|x\| = 1} \|Ax\|$$

Recap: Condition of a problem (cont'd)

- If $\kappa_{\text{rel}} \sim 1$,

Recap: Condition of a problem (cont'd)

- If $\kappa_{\text{rel}} \sim 1$, then the problem is well conditioned: If the relative error in the data/input is small, then the relative error in the answer/output is similarly small
- If $\kappa_{\text{rel}} \gg 1$,

Recap: Condition of a problem (cont'd)

- If $\kappa_{\text{rel}} \sim 1$, then the problem is well conditioned: If the relative error in the data/input is small, then the relative error in the answer/output is similarly small
- If $\kappa_{\text{rel}} \gg 1$, then the problem is poorly conditioned: Small relative input error can lead to large relative output error
- If κ_{rel} (and κ_{abs}) do not exist, then the problem is ill conditioned.

- What is poorly conditioned depends on desired accuracy: if the input accuracy is low but we expect a high output accuracy, then problems are quickly poorly conditioned. If we are happy with a less accurate output, we might consider the problem still well conditioned.
- Sometimes, the possibly large error in the output does not matter and so we can solve poorly conditioned problems (think of early design stages, rapid prototyping, etc); but we should be very much aware of the condition of the problem.

Recap: Condition number of a matrix

Consider a matrix $A \in \mathbb{R}^{n \times n}$. Its condition number is

$$\kappa(A) = \|A\| \|A^{-1}\|$$

Widely used is the $\| \cdot \|_2$ norm and then

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

with the maximal and minimal singular value $\sigma_{\max}(A)$ and $\sigma_{\min}(A)$ of A

Consider a system of linear equations $Ax = b$. Then, the problems $A \mapsto A^{-1}b$ and $b \mapsto A^{-1}b$ have relative condition numbers

$$\kappa_{\text{rel}} \leq \kappa(A)$$

Recap: Backward stability

Backward stability: Pass the errors of the algorithm back and interpret as input errors.

An algorithm $\tilde{f}$ for a problem f is backward stable if for each $x \in X$ we have $\tilde{f}(x) = f(\tilde{x})$ for an $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|}$$

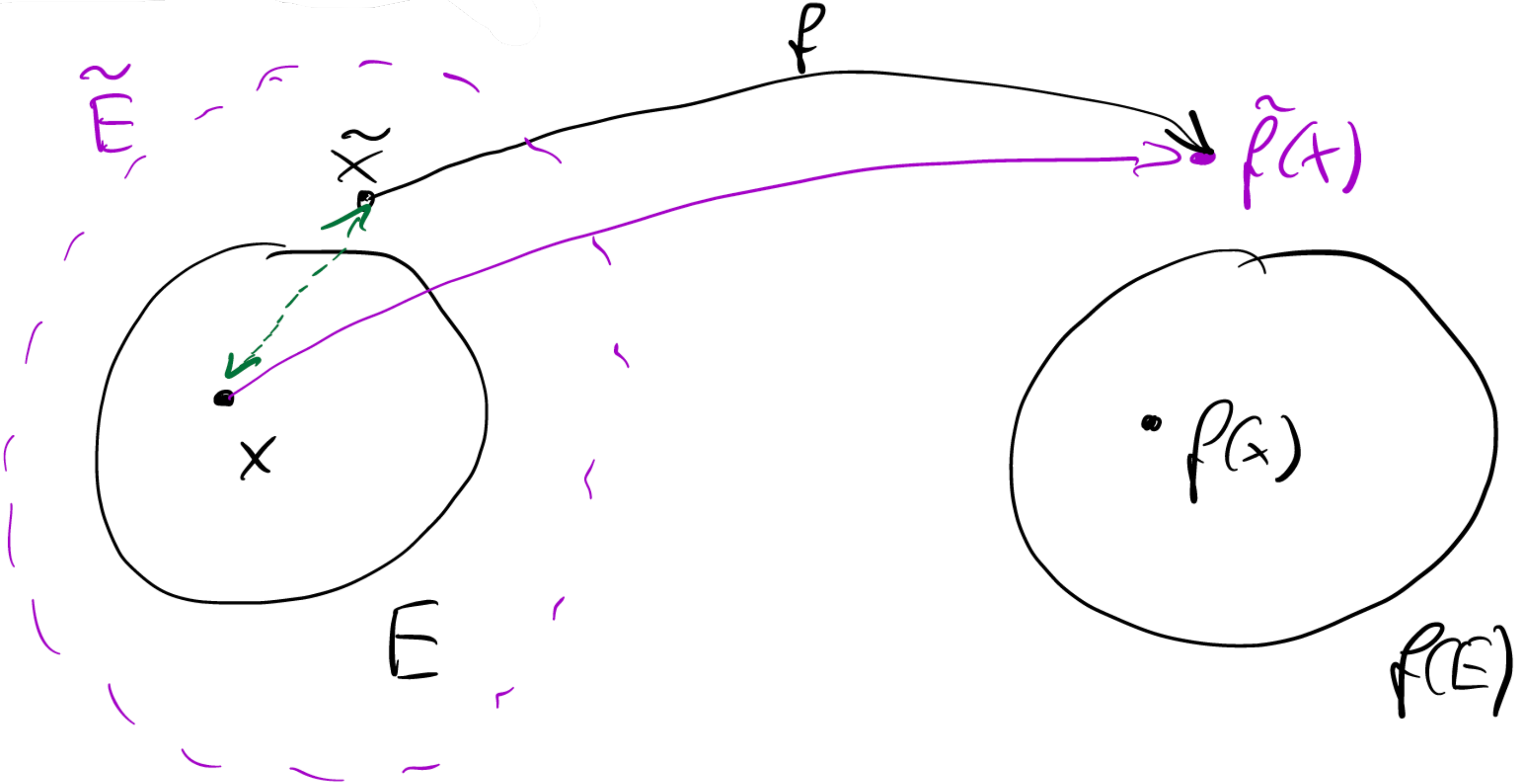
small

This is a tightening of the definition of stability of the previous slide:

> *A backward stable algorithm gives exactly the right answer to nearly the right question.*

In backward error analysis one calculates, for a given output, how much one would need to perturb the input in order for the answer to be exact.

Recap: Backward stability (cont'd)



Representing real numbers

Representing real numbers

- Computers represent everything using bit strings, i.e., integers in base 2. A finite number of integers can thus be exactly represented. But not real numbers! This leads to roundoff errors.
- Assume we have N digits to represent real numbers on a computer that can represent integers using a given number system, say decimal for human purposes.
- Fixed-point representation of numbers

$$x = (-1)^s \cdot [a_{N-2}a_{N-3}\cdots a_k.a_{k-1}\cdots a_0]$$

has a problem of representing either small or larger numbers because the decimal point . is fixed at position k

Floating-point numbers

- Instead, let's use floating-point representation

$$x = (-1)^s \cdot [0\,.\,a_1 a_2 \cdots a_t] \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}$$

similar to the common scientific number representation

$$0.1156 \cdot 10^1 = 1156 \cdot 10^{-3} \qquad t = 4$$

- A floating-point number in base β is represented using one sign bit $s = 0$ or 1, a t-digit integer mantissa

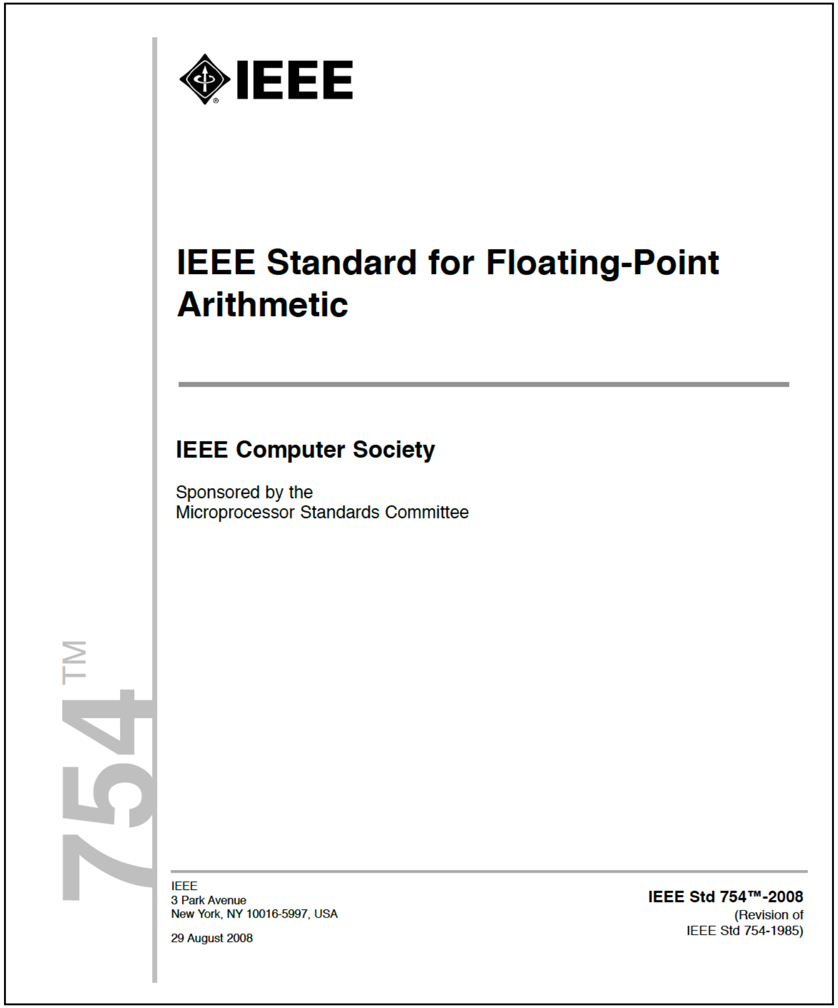
$$0 \leq m = [a_1 a_2 \cdots a_t] \leq \beta^t - 1$$

and an integer exponent $L \leq e \leq U$

- Computers today use binary numbers and so $\beta = 2$

IEEE 754 standard

- Formats for representing and encoding real numbers using bit strings (single and double precision).
- Rounding algorithms for performing accurate arithmetic operations (e.g., addition, subtraction, division, multiplication) and conversions (e.g., single to double precision).
- Exception handling for special situations (e.g., division by zero and overflow).


IEEE

IEEE Standard for Floating-Point Arithmetic

IEEE Computer Society

Sponsored by the
Microprocessor Standards Committee

754™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

29 August 2008

IEEE Std 754™-2008
(Revision of
IEEE Std 754-1985)

Single precision IEEE floating-point numbers have the standardized storage format:

$$\text{sign} + \text{power} + \text{fraction}$$

with

$$N_s + N_p + N_f = 1 + 8 + 23 = 32 \text{ bits}$$

and are interpreted as

$$x = (-1)^s \cdot 2^{p-127} \cdot (1.f)_2$$

- Sign $s = 1$ for negative numbers
- Power $1 \leq p \leq 254$ determines the exponent
- Fractional part of the mantissa f
- `single` in Matlab, `float` in C/C++, REAL in Fortran)

IEEE representation example

Take the number $x = 2752 = 0.2752 \cdot 10^4$. Converting 2752 to the binary number system

$$\begin{aligned} x &= 2^{11} + 2^9 + 2^7 + 2^6 = (101011000000)_2 = 2^{11} \cdot (1.01011)_2 \\ &= (-1)^0 2^{138-127} \cdot (1.01011)_2 = (-1)^0 2^{(10001010)_2 - 127} \cdot (1.01011)_2 \end{aligned}$$

On the computer:

$$\begin{aligned} x &= [s \quad | \quad p \quad | \quad f] \\ &= [0 \quad | \quad 100,0101,0 \quad | \quad 010,1100,0000,0000,0000,0000] \\ &= (452c0000)_{16} \end{aligned}$$

```
format hex;
>> a=single(2.752E3)
a =
   452c0000
```

Double precision IEEE numbers

Double precision IEEE numbers (default in Matlab, `double` in C/C++) follow the same principle but use 64 bits to give higher precision and range

$$N_s + N_p + N_f = 1 + 11 + 52 = 64 \text{ bits}$$

$$x = (-1)^s \cdot 2^{p-1023} \cdot (1.f)_2$$

Even higher (extended) precision formats are not really standardized or widely implemented/used.

There is also software-emulated variable precision arithmetic in, e.g., Maple

Extremal exponent values

The extremal exponent values have special meaning (here single precision)

value	power p	fraction f
± 0	0	0
$\pm\infty$	255	0
Not a number (NaN)	255	> 0

Important facts about floating-point numbers

- Not all real numbers x, or even integers, can be represented exactly as a floating-point number. Instead, they must be *rounded* to the nearest floating point number $\hat{x} = \mathrm{fl}(x)$

- Floating-point numbers have a *relative rounding error* that is smaller than the *machine precision* or *roundoff-unit* u

$$\frac{|\hat{x} - x|}{|x|} \leq u = 2^{-(N_f+1)} = \begin{cases} 2^{-24} \sim 6.0 \cdot 10^{-8}\,, & \text{for single precision} \\ 2^{-53} \sim 1.1 \cdot 10^{-16}\,, & \text{for double precision}\,. \end{cases}$$

- Often the machine precision/roundoff-unit is denoted as ϵ

- **The rule of thumb is that single precision gives 7-8 digits of precision and double 16 digits.**

- There is a smallest and largest possible number due to limit for the exponent.

Two axioms

Ignoring over- and underflow, we assume the following two "axioms" to hold for computers we work with:

1. For all $x \in \mathbb{R}$, there exists ϵ with $|\epsilon| \le u$ (roundoff unit) such that

   $$\mathrm{fl}(x) = x(1+\epsilon)\,,$$

   where $\mathrm{fl}(\cdot)$ rounds to the the closest floating point approximation.

2. Consider two floating point numbers x, y. The floating-point operation $\circledast$ (=add, sub, mult, div) of $*$ (=add, sub, mult, div) satisfies

   $$x \circledast y = \mathrm{fl}(x * y)$$

Axiom 1 and 2 imply that for two floating-point numbers x, y, there exists ϵ with $|\epsilon| \le u$ such that

$$x \circledast y = (x * y)(1+\epsilon)\,.$$

Floating-point exceptions

Computing with floating point values may lead to exceptions, which may halt the program:

Floating-point exceptions

Computing with floating point values may lead to exceptions, which may halt the program:

- **Divide-by-zero**: if the result is $\pm\infty$, e.g., $1/0$

Floating-point exceptions

Computing with floating point values may lead to exceptions, which may halt the program:

- **Divide-by-zero**: if the result is $\pm\infty$, e.g., $1/0$
- **Invalid**: If the result is a *NaN*, e.g., taking $\sqrt{-1}$ (note that Matlab supports complex numbers...)

```
>>> x = math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Floating-point exceptions

Computing with floating point values may lead to exceptions, which may halt the program:

- **Divide-by-zero**: if the result is $\pm\infty$, e.g., $1/0$
- **Invalid**: If the result is a *NaN*, e.g., taking $\sqrt{-1}$ (note that Matlab supports complex numbers...)

```
>>> x = math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

- **Overflow**: If the result is too large to be represented, e.g., adding two numbers, each on the order of *realmax*

Floating-point exceptions

Computing with floating point values may lead to exceptions, which may halt the program:

- **Divide-by-zero**: if the result is $\pm\infty$, e.g., $1/0$
- **Invalid**: If the result is a *NaN*, e.g., taking $\sqrt{-1}$ (note that Matlab supports complex numbers...)

```
>>> x = math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

- **Overflow**: If the result is too large to be represented, e.g., adding two numbers, each on the order of *realmax*
- **Underflow**: If the result is too small to be represented, e.g., dividing a number close to *realmin* by a large number.

Avoiding overflow

Numerical software needs to be careful about avoiding exceptions:

Mathematically equivalent expressions are not necessarily computationally equivalent!

- For example, computing $\sqrt{x^2 + y^2}$ may lead to overflow in computing $x^2 + y^2$ even though the result does not overflow
- Matlab's hypot function guards against this:

$$\sqrt{x^2 + y^2} = |x|\sqrt{1 + \left(\frac{y}{x}\right)^2} \text{ ensuring that } |x| > |y|$$

  works correctly
- These kind of careful constructions may have higher computational cost (more CPU operations) or make roundoff errors worse.

Floating-point in practice

- Most scientific software **uses double precision** to avoid range and accuracy issues with single precision (better be safe then sorry). Single precision may offer speed/memory/vectorization advantages however (e.g. GPU computing).
- **Do not compare floating point numbers** (especially for loop termination), or more generally, do not rely on logic from pure mathematics.
- Optimization, especially in compiled languages, can rearrange terms or perform operations using **unpredictable** alternate forms (e.g., wider internal registers).
  Using parenthesis helps , e.g. $(x + y) - z$ instead of $x + y - z$, but does not eliminate the problem.
- Library functions such as sin and ln will typically be computed almost to full machine accuracy, but do not rely on that for special/complex functions.

Propagation of errors

- Assume that we are calculating something with numbers that are not exact, e.g., a rounded floating-point number $\hat{x}$ versus the exact real number x.
- For IEEE representations, recall that

$$\frac{|\hat{x}-x|}{|x|} \leq u = 2^{-(N_f+1)} = \begin{cases} 2^{-24} \sim 6.0 \cdot 10^{-8}\,, & \text{for single precision} \\ 2^{-53} \sim 1.1 \cdot 10^{-16}\,, & \text{for double precision}\,. \end{cases}$$

- In general, the *absolute error* $\delta x = \hat{x} - x$ may have contributions from each of the different types of error (roundoff, truncation, propagated, statistical).
- Assume we have an estimate or bound for the relative error

$$\left|\frac{\delta x}{x}\right| \lesssim \epsilon_x \ll 1$$

based on some analysis, e.g., for roundoff error the IEEE standard determines $\epsilon_x = u$ (roundoff-unit)

Propagation of errors

How does the relative error change (propagate) during numerical calculations?

$$\varepsilon_{xy} = \left| \frac{(x+\delta x)(y+\delta y) - xy}{xy} \right|$$

$$= \left| \cancel{\frac{xy}{xy}} + \frac{\delta x\, \cancel{y}}{x\, \cancel{y}} + \frac{\cancel{x}\, \delta y}{\cancel{x}\, y} + \frac{\delta x\, \delta y}{xy} - \cancel{\frac{xy}{xy}} \right|$$

$$\left| \frac{\delta x}{x} \right| \ll 1 \qquad \left| \frac{\delta y}{y} \right| \ll 1$$

$$\frac{\delta_x \delta_y}{xy} \le \max\left\{\frac{\delta_x}{x}, \frac{\delta_y}{y}\right\}$$

$$\left| \frac{\delta_x}{x} + \frac{\delta_y}{y} + \frac{\delta_x \delta_y}{xy} \right| \le 2(\varepsilon_x + \varepsilon_y)$$

$$\lesssim \underline{\varepsilon_x} + \underline{\varepsilon_y}$$

Propagation of errors: Numerical experiment [From A. Donev]

Harmonic sum

$$H(N) = \sum_{i=1}^{N} \frac{1}{i}$$

Matlab implementation prone to error propagation

```
function nhsum = harmonic(N)
nhsum = 0;
for i = 1:N
    nhsum = nhsum + 1.0/i;
end
end
```

What are the numerical issues of this implementation?

Propagation of errors: Numerical experiment [From A. Donev]

Harmonic sum

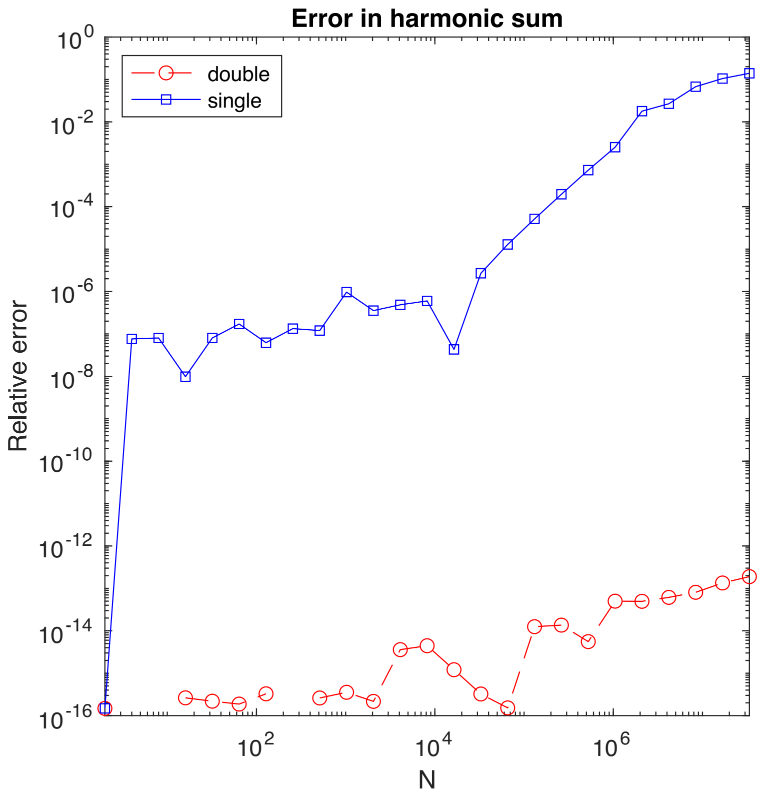
$$H(N) = \sum_{i=1}^{N} \frac{1}{i}$$

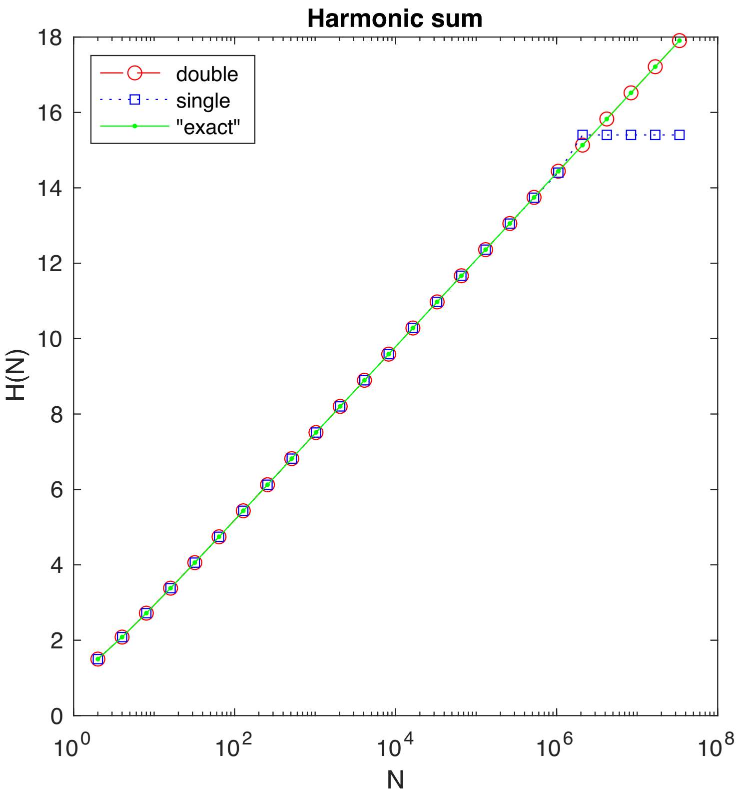
Matlab implementation prone to error propagation

```
function nhsum = harmonic(N)
nhsum = 0;
for i = 1:N
    nhsum = nhsum + 1.0/i;
end
end
```

What are the numerical issues of this implementation?

$\rightsquigarrow$ Adds very small number $1/i$ to potentially large number `nhsum`



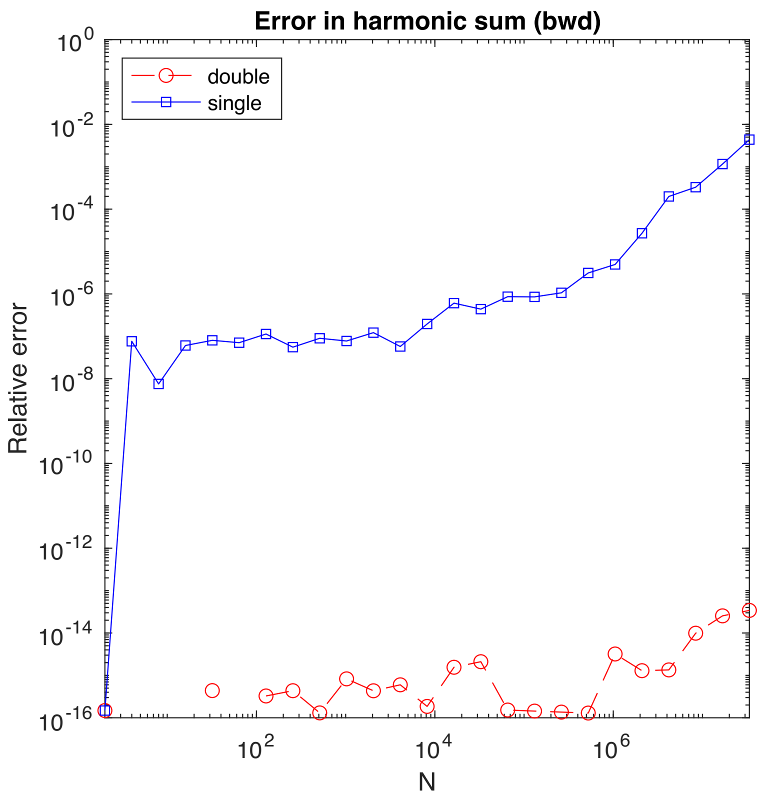


What can we do about it?

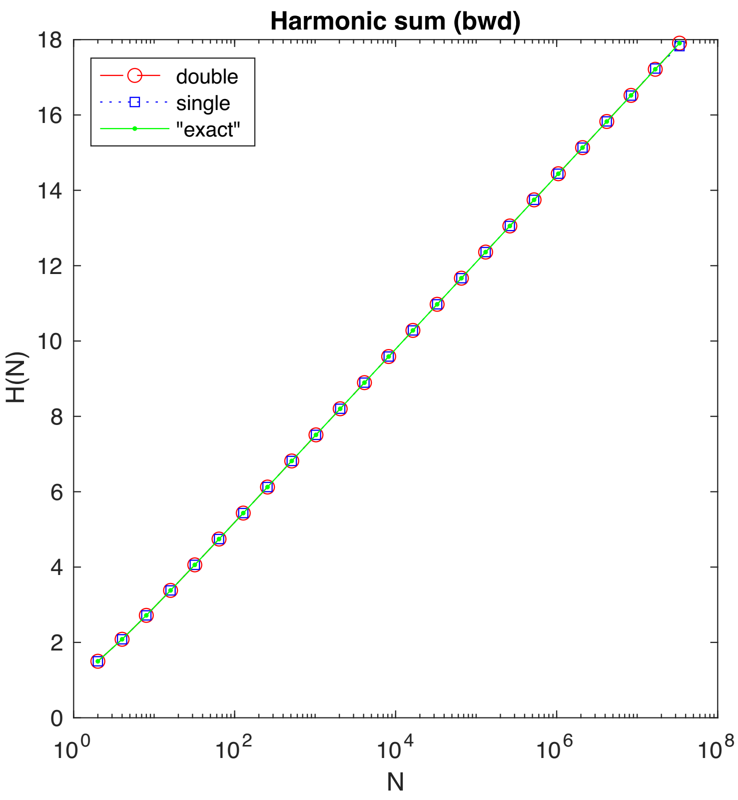
Implementation with backward summation

```
function nhsum = harmonicBwd(N)
nhsum = 0;
for i = N:-1:1
      nhsum = nhsum + 1.0/i;
end
end
```

Better, because adds small numbers to small numbers and larger numbers to large numbers.



Error in harmonic sum (bwd)
Relative error
N
double
single



Harmonic sum (bwd)
H(N)
N
double
single
"exact"

Numerical cancellation

If x and y are close to each other, then $x - y$ can have reduced accuracy due to catastrophic cancellation.

Consider computing the smaller root of the quadratic equation

$$x^2 - 2x + c = 0$$

for $|c| \ll 1$ and focus on propagation/accumulation of roundoff errors.

$$x^2 - 2x + c = 0 \qquad |c| \ll 1$$

Solution $x = 1 - \sqrt{1-c}$

Taylor approximation

$$f(z) = \sqrt{1+z} \quad , \quad f'(z)$$

$$f(z) = f(0) + \frac{f'(0)}{1!}(z-0) + \frac{f''(0)}{2!}(z-0)^2 + \dots$$

$$\Rightarrow \sqrt{1+z} \doteq 1 + \frac{1}{2} z$$

For $|c|$ small

$$x = 1 - \sqrt{1-c} \doteq 1 - (1 - \frac{c}{2}) = \frac{c}{2}$$

Case 1: $|c| < |u|$

$$fl(1-c) = 1$$

$\Rightarrow$ no point in moving fwd

Case 2: $|u| < |c| \ll 1$

calculate $1-c$

$$fl(1) \ominus fl(c) = fl(fl(1) - fl(c))$$

$$= (fl(1) - fl(c))(1+\varepsilon) \qquad |\varepsilon| \le u$$

$$= (1(1+\varepsilon_1) - c(1+\varepsilon_c))(1+\varepsilon), \qquad |\varepsilon_1|, |\varepsilon_c| \le u$$

$$\underbrace{fl(1) \ominus fl(c) - (1-c)}_{\delta(1-c)} = \underbrace{\underbrace{(1-c)}_{\text{order } 1}\varepsilon}_{\text{order } u} + \underbrace{\underbrace{(\varepsilon_1 - \varepsilon_c)}_{\text{order } u}(1+\varepsilon)}_{\text{order } u}$$

$$\left| \frac{\delta(1-c)}{1-c} \right| \quad \text{order } |u|$$

Sqrt:

$$\sqrt{x+\delta x} = \sqrt{x}\left(1 + \frac{\delta x}{x}\right)^{1/2}$$

$$\doteq \sqrt{x}\left(1 + \frac{\delta x}{2x}\right) \qquad u$$

Quick analysis

$\sqrt{1-c}$ error of order u

$$x = 1 - \sqrt{1-c}$$

$$fl(1) \ominus fl(y) - (1-y) = \delta(1-y)$$

order $\underline{|1-y| \cdot |u| + |u|}$

$$\left| \frac{\delta(1-y)}{1-y} \right| \approx \frac{|1-y| \cdot |u| + |u|}{|1-y|} = |u| + \frac{|u|}{|1-y|}$$

To avoid cancellation, we should not directly implement $1 - \sqrt{1-c}$

Rather, we can take the Taylor approximate $x \approx \frac{c}{2}$, which provides a good approximation for small c.

Even better, we could use the **mathematically equivalent but numerically preferred form**:

$$1 - \sqrt{1-c} = \frac{c}{1+\sqrt{1-c}}$$

which does not suffer from cancellation problems as c becomes smaller.
(Notice that $1 - \sqrt{\dots}$ is avoided and therefore the cancellation problem shown by our analysis is avoided. We showed that $\sqrt{1-c}$ is safe.)

To avoid cancellation, we should not directly implement $1 - \sqrt{1-c}$

Rather, we can take the Taylor approximate $x \approx \frac{c}{2}$, which provides a good approximation for small c.

Even better, we could use the **mathematically equivalent but numerically preferred form**:

$$1 - \sqrt{1-c} = \frac{c}{1+\sqrt{1-c}}$$

which does not suffer from cancellation problems as c becomes smaller.
(Notice that $1 - \sqrt{\ldots}$ is avoided and therefore the cancellation problem shown by our analysis is avoided. We showed that $\sqrt{1-c}$ is safe.)

```
>>> c = 1e-10 # solution roughly 5.000000000125 x 10^-11
>>> 1 - math.sqrt(1 - c)
5.000000413701855e-11
>>> c/(1 + math.sqrt(1 - c))
5.000000000125e-11
```

Big $\mathcal{O}$ notation

Useful to compare growth of functions.

We write $f \in \mathcal{O}(g)(x \to \infty)$ if there exists constant $C > 0$ such that for an x_0 the following holds

$$\forall x \geq x_0 : \quad |f(x)| \leq C|g(x)|$$

We also write $f \in \mathcal{O}(g)(x \to 0)$ if there exists a constant $C > 0$ such that for an $x_0 > 0$ the following holds

$$\forall |x| \leq x_0 : \quad |f(x)| \leq C|g(x)|$$

In many cases we do not write explicitly whether we mean $x \to \infty$ or $x \to 0$ because it is clear from the context.

Question: In practice, would you prefer an algorithm with costs growing as $c_1(x) \in \mathcal{O}(x)$ or $c_2(x) \in \mathcal{O}(x^2)$? Why?

Question: In practice, would you prefer an algorithm with costs growing as $c_1(x) \in \mathcal{O}(x)$ or $c_2(x) \in \mathcal{O}(x^2)$? Why?

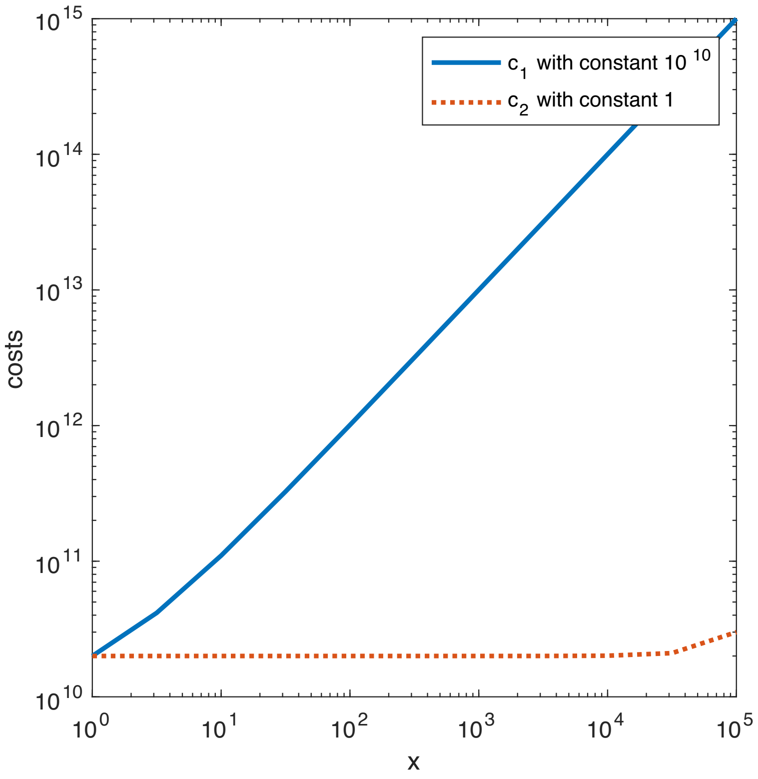
Answer: It depends on the hidden constants C and x_0. If c_1 and c_2 have roughly the same constants, then probably c_1.

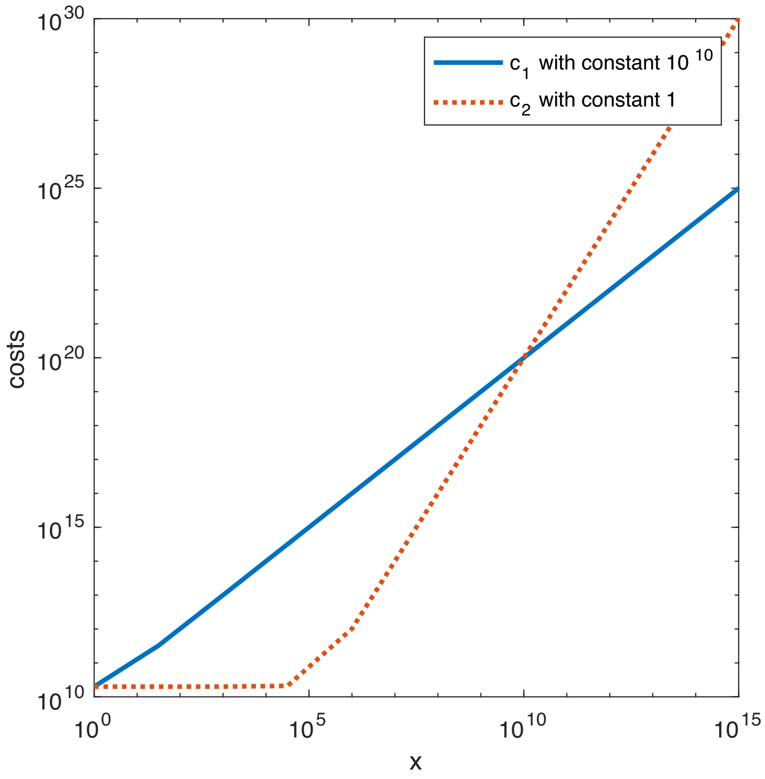
However, if the constant for c_1 is $x_0 = 10^{10}$ and the constant for c_2 is $x_0 = 1$, then in most practical situations we prefer c_2 because we most likely will never reach the asymptotics of $x > x_0$ for c_1 in practice!

Warning: The Big $\mathcal{O}$ notation tells us something about the asymptotics. The constants x_0 and C that are hidden in $\mathcal{O}(\cdot)$ do matter in practice!

Costs (i.e., $x \to \infty$) of $c_1(x) = 10^{10} + 10^{10}x$ and $c_2(x) = 2 \times 10^{10} + x^2$. Then, $c_1 \in \mathcal{O}(x)$ and $c_2 \in \mathcal{O}(x^2)$ for $x \to \infty$. I.e., asymptotically the costs of c_2 grow faster than c_1.

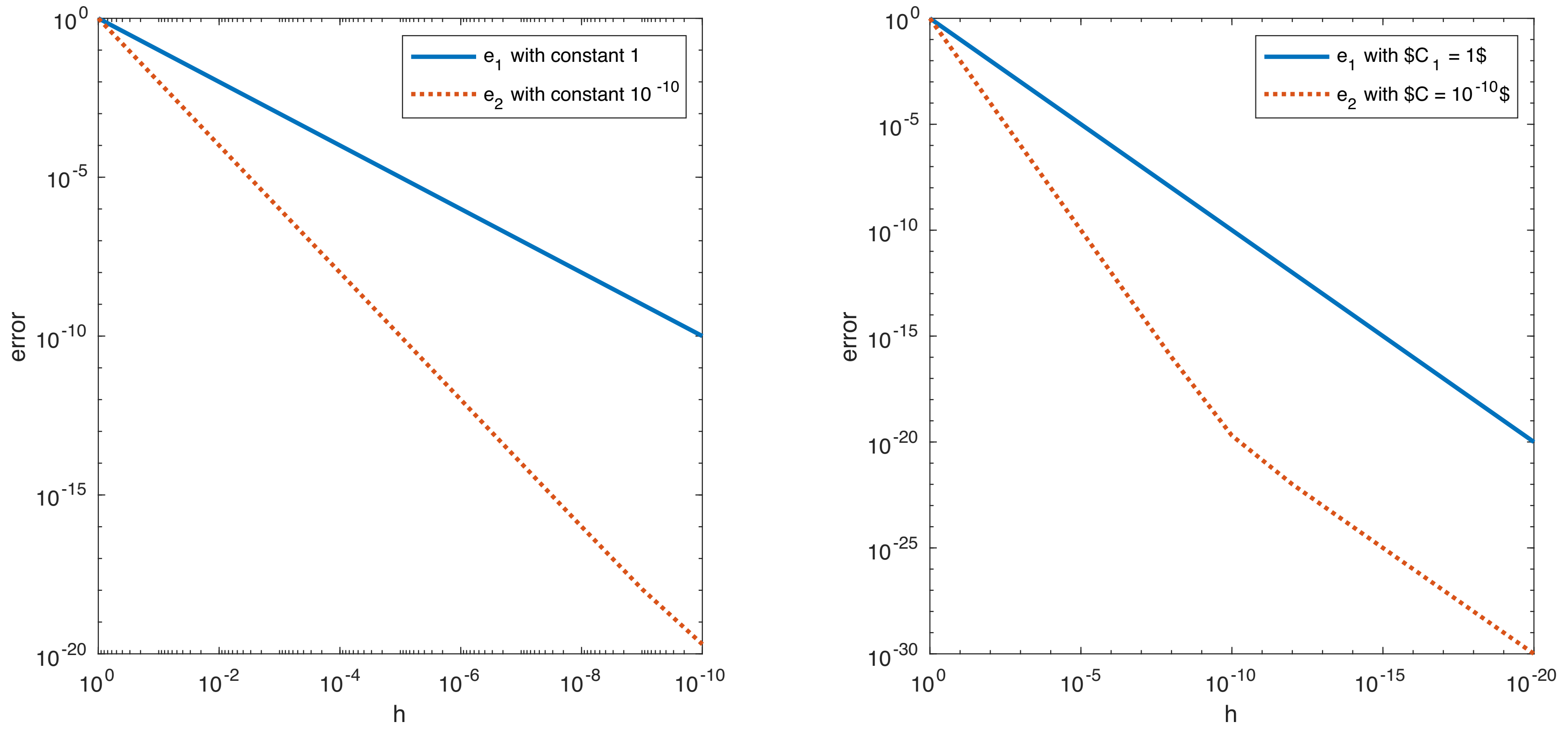
Costs (i.e., $x \to \infty$) of $c_1(x) = 10^{10} + 10^{10}x$ and $c_2(x) = 2 \times 10^{10} + x^2$. Then, $c_1 \in \mathcal{O}(x)$ and $c_2 \in \mathcal{O}(x^2)$ for $x \to \infty$. I.e., asymptotically the costs of c_2 grow faster than c_1.





Warning: The Big $\mathcal{O}$ notation tells us something about the asymptotics. The constants x_0 and C that are hidden in $\mathcal{O}(\cdot)$ do matter in practice!

Set now error: $e_1(h) = h$ and $e_2(h) = 10^{-10}h + h^2$. Then, $e_1 \in \mathcal{O}(h)$ and $e_2 \in \mathcal{O}(h)$ for $h \to 0$.



Warning: The Big $\mathcal{O}$ notation tells us something about the asymptotics. The constants h_0 and C that are hidden in $\mathcal{O}(\cdot)$ do matter in practice!

Revisiting stability

Recall that we said: An algorithm $\tilde{f}$ for a problem f is backward stable if for each $x \in X$ we have $\tilde{f}(x) = f(\tilde{x})$ for an $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|}$$

small.

We now can be more precise: An algorithm $\tilde{f}$ for a problem f is backward stable if for each $x \in X$ we have $\tilde{f}(x) = f(\tilde{x})$ for an $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} \in \mathcal{O}(u)\,,$$

where u is the roundoff unit

- Recall that, loosely speaking, the symbol $\mathcal{O}(u)$ means "on the order of the roundoff unit."
- By allowing $u \to 0$ (which is implied here by the $\mathcal{O}$), we consider an idealization of a computer (in practice, u is fixed). So what we mean is that the error should decrease in proportion to u or faster.

Suppose a backward stable algorithm is applied to solve a problem $f : X \to Y$ with relative condition number κ. Then, the relative errors satisfy

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \in \mathcal{O}(\kappa(x)u)\,.$$

Proof board

Bwd stable $\tilde{f}(x) = f(\tilde{x})$

$$\frac{\|\tilde{x} - x\|}{\|x\|} \in O(u)$$

condition

$$k = \lim_{\delta \to 0} \sup_{\|x - \hat{x}\| \le \delta} \frac{\|f(x) - f(\hat{x})\|}{\|f(x)\|} \Big/ \frac{\|x - \hat{x}\|}{\|x\|}$$

$$\tilde{f}(x) = f(\tilde{x})$$

$$\frac{\|f(x) - \tilde{f}(x)\|}{\|f(x)\|} \le k(x) \underbrace{\frac{\|x - \tilde{x}\|}{\|x\|}}_{\in O(u)}$$

$$\Rightarrow \begin{array}{l} \delta \to 0 \\ u \to 0 \end{array} \quad O(k(x)\, u)$$

Conclusions and summary

- No numerical method can compensate for a poorly conditioned problem. But not every numerical method will be a good one for a well conditioned problem.
- A numerical method needs to control the various computational errors (approximation, truncation, roundoff, propagated, statistical) while balancing computational cost.
- A numerical method must be consistent and stable in order to converge to the correct answer.
- The IEEE standard standardizes the single and double precision floating-point formats, their arithmetic, and exceptions. It is widely implemented.
- Numerical overflow, underflow and cancellation need to be carefully considered and avoided: Mathematically equivalent forms are not numerically equivalent.

Matlab peculiarities [Following slides: A. Donev]

- MATLAB is an **interpreted language**, meaning that commands are interpreted and executed as encountered. MATLAB caches some stuff though...
- Many of MATLAB's **intrinsic routines** are however compiled and optimized and often based on well-known libraries (BLAS, LAPACK, FFTW, etc.).
- Variables in scripts/worspace are global and persist throughout an interactive session (use *whos* for info and *clear* to clear workspace).
- Every variable in MATLAB is, unless specifically arranged otherwise, a matrix, **double precision float** if numerical.
- Vectors (column or row) are also matrices for which one of the dimensions is 1.
- **Complex arithmetic** and complex matrices are used where necessary.

Matrices [Slide: A. Donev]

```
>> format compact; format long
>> x=-1; % A scalar that is really a 1x1 matrix
>> whos('x')
  Name      Size            Bytes  Class     Attributes
  x         1x1                 8  double

>> y=sqrt(x) % Requires complex arithmetic
y =                  0 + 1.000000000000000i
>> whos('y')
  Name      Size            Bytes  Class     Attributes
  y         1x1                16  double    complex

>> size(x)
ans =      1     1
>> x(1)
ans =     -1
>> x(1,1)
ans =     -1
>> x(3)=1;
>> x
x =     -1     0     1
```

Vectorization/Optimization [Slide: A. Donev]

- MATLAB uses **dynamic memory management** (including garbage collection), and matrices are re-allocated as needed when new elements are added.
- It is however much better to **pre-allocate space** ahead of time using, for example, *zeros*.
- The **colon notation** is very important in accessing array sections, and x is different from $x(:)$.
- **Avoid for loops** unless necessary: Use array notation and intrinsic functions instead.
- To see how much CPU (computing) time a section of code took, use *tic* and *toc* (but beware of timing small sections of code).
- MATLAB has built-in **profiling tools** (*help profile*).

Pre-allocation [Slide: A. Donev]

```
format compact; format long
clear; % Clear all variables from memory

N=100000; % The number of iterations

% Try commenting this line out:
f=zeros(1,N); % Pre-allocate f

tic;
f(1)=1;
for i=2:N
    f(i)=f(i-1)+i;
end
elapsed=toc;

fprintf('The result is f(%d)=%g, computed in %g s\n', ...
        N, f(N), elapsed);
```

Vectorization [Slide: A. Donev]

```
function vect(vectorize)
   N=1000000; % The number of elements
   x=linspace(0,1,N); % Grid of N equi-spaced points

   tic;
   if(vectorize) % Vectorized
      x=sqrt(x);
   else % Non-vectorized
      for i=1:N
         x(i)=sqrt(x(i));
      end
   end
   elapsed=toc;

   fprintf('CPU time for N=%d is %g s\n', N, elapsed);
end
```

Matlab examples [Slide: A. Donev]

```
>> fibb % Without pre-allocating
The result is f(100000)=5.00005e+09, computed in 6.53603 s

>> fibb % Pre-allocating
The result is f(100000)=5.00005e+09, computed in 0.000998 s

>> vect(0) % Non-vectorized
CPU time for N=1000000 is 0.074986 s

>> vect(1) % Vectorized -- don't trust the actual number
CPU time for N=1000000 is 0.002058 s
```

Vectorization/Optimization [Slide: A. Donev]

- Recall that everything in MATLAB is a double-precision matrix, called **array**.
- Row vectors are just matrices with first dimension 1. Column vectors have row dimension 1. Scalars are 1×1 matrices.
- The syntax x' can be used to construct the **conjugate transpose** of a matrix.
- The **colon notation** can be used to select a subset of the elements of an array, called an **array section**.
- The default arithmetic operators, $+$, -, *****, / and ^ are **matrix addition/subtraction/multiplication**, linear solver and matrix power.
- If you prepend a **dot before an operator** you get an **element-wise operator** which works for arrays of the same shape.

Matrices [Slide: A. Donev]

```
>> x=[1 2 3; 4 5 6] % Construct a matrix
x =      1      2      3
         4      5      6

>> size(x) % Shape of the matrix x
ans =      2      3

>> y=x(:) % All elements of y
y =      1      4      2      5      3      6

>> size(y)
ans =      6      1

>> x(1,1:3)
ans =      1      2      3

>> x(1:2:6)
ans =      1      2      3
```

Matrices [Slide: A. Donev]

```
>> sum(x)
ans =
     5     7     9

>> sum(x(:))
ans =
    21

>> z=1i; % Imaginary unit

>> y=x+z
y =
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
   4.0000 + 1.0000i   5.0000 + 1.0000i   6.0000 + 1.0000i

>> y'
ans =
   1.0000 - 1.0000i   4.0000 - 1.0000i
   2.0000 - 1.0000i   5.0000 - 1.0000i
   3.0000 - 1.0000i   6.0000 - 1.0000i
```

Matrices [Slide: A. Donev]

```
>> x*y
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> x.*y
ans =
    1.0000 + 1.0000i    4.0000 + 2.0000i    9.0000 + 3.0000i
   16.0000 + 4.0000i   25.0000 + 5.0000i   36.0000 + 6.0000i

>> x*y'
ans =
   14.0000 - 6.0000i   32.0000 - 6.0000i
   32.0000 -15.0000i   77.0000 -15.0000i

>> x'*y
ans =
   17.0000 + 5.0000i   22.0000 + 5.0000i   27.0000 + 5.0000i
   22.0000 + 7.0000i   29.0000 + 7.0000i   36.0000 + 7.0000i
   27.0000 + 9.0000i   36.0000 + 9.0000i   45.0000 + 9.0000i
```

Coding guidelines [Slide: A. Donev]

- Learn to reference the **MATLAB help**: Including reading the examples and "fine print" near the end, not just the simple usage.
- **Indendation, comments, and variable naming** make a big difference! Code should be readable by others.
- Spending a few extra moments on the code will pay off when using it.
- Spend some time learning how to **plot in MATLAB**, and in particular, how to plot with different symbols, lines and colors using *plot, loglog, semilogx, semilogy.*
- Learn how to **annotate plots**: *xlim, ylim, axis, xlabel, title, legend.* The intrinsics *num2str* or *sprintf* can be used to create strings with embedded parameters.
- Finer controls over fonts, line widths, etc., are provided by the intrinsic function *set*...including using the LaTex interpreter to typeset mathematical notation in figures.