Welcome to 18.335J / 6.337J!

Course info/materials: github.com/mitmath/18.335J

→ Psets posted on Canvas
→ Discussion forum on Piazza

18.335J is an advanced intro to NLA + select topics

- Direct methods
- Iterative methods
- Randomized NLA
- Optimization, nonlinear problems, NLA in inf. dim. spaces, ...

Numerical methods (& analysis)

"Numerical analysis is the study of algorithms for the problems of continuous mathematics." - L.N. Trefethen

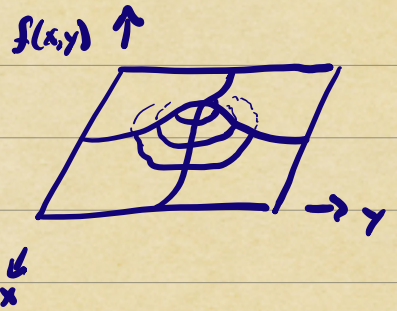
At the heart of those algorithms:

Numerical Linear Algebra (NLA)

Example 1: Numerical methods for PDEs

a) Poisson's equation $\Delta u = f$ $\quad u|_{\partial\Omega} = g$

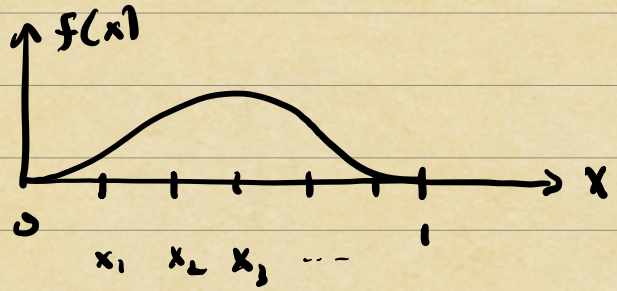
"source"



How do we solve on a computer?

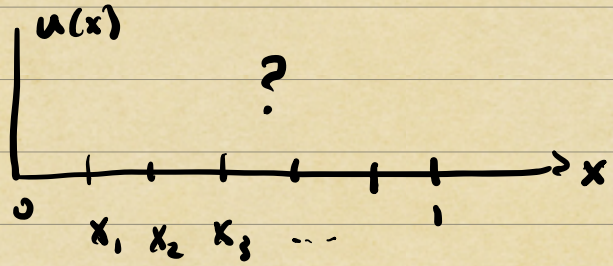
Illustrate in 1D

$$\frac{d^2}{dx^2} u(x) = f(x)$$



- Discretize w/ finite differences

$f_k = f(x_k)$, $u_k = u(x_k)$, $h = x_{k+1} - x_k$



$$u''(x_k) \approx \frac{u_{k+1} - 2u_k + u_{k-1}}{h^2}$$

$$\frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & 1 & -2 \end{bmatrix}\begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_N \end{bmatrix}$$

- Solve by LU factorization (Gaussian Elim.)

b) Schrodinger Equation

Quantized energies = eigenvalues

PDE Eigenvalue Problem

$$-\Delta u + Vu = Eu$$

- Discretize by finite differences

$$\frac{1}{h^2}\begin{bmatrix} 2+V_1h^2 & -1 & & \\ -1 & 2+V_2h^2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2+V_Nh^2 \end{bmatrix}\begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix} = E\begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix}$$

- Solve by QR Algorithm

Standard workflow: "discretize-then-solve"

$\Delta u = f$ s.t. $u|_{\partial\Omega} = g$

"discretize" ⇓ → see 18.336J Fast methods for PDE; IE

$Ax = b$

"solve" ⇓

18.335J, e.g. NLA

? → see 18.330 Intro to numerical analysis

$x \approx u$

Remarks:

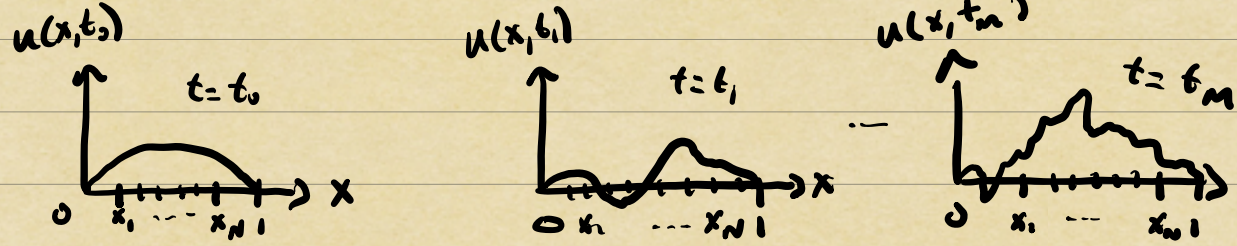
- In 2 and 3D, A can be HUGE!
- Standard factorization techniques become too expensive
- but A arising from PDE disc. (and many other settings) often has <u>structure</u> (e.g. tridiagonal, block tridiagonal, etc.) that allows us to compute Ax or $A^{-1}b$, or $A = X \Lambda X^{-1}$ very efficiently
- A central theme in modern NLA is finding and exploiting <u>structure</u> to design fast algorithms

Example 2: Numerical models from data

What if we don't know the PDE for a complex system, but we have lots of experimental data? Can we "learn" a model?

1D Illustration

Data (= "Snapshots")



Want to learn linear model that maps

$$A: u(x, t_k) \to u(x, t_{k+1})$$

Data matrix $(X)_{i,j} = u(x_i, t_j)$ ← Experimental data

$$X = \begin{bmatrix} | & & | \\ u(x,t_1) & \cdots & u(x,t_{m-1}) \\ | & & | \end{bmatrix}$$

$$X' = \begin{bmatrix} | & & | \\ u(x,t_2) & \cdots & u(x,t_m) \\ | & & | \end{bmatrix}$$

$\Rightarrow$ Find A s.t. $X' = AX$

(May have infinitely many, one, or no solutions)

Idea: minimize ↑ "size" of $X' - AX$ over $N \times N$ matrices A. (Usually $N \gg M$.)

Measure "size" of $X' - AX$ with a matrix norm, e.g., the Frobenius norm (generalizes Euclidean dist.)

$$\|M\|_F = \sqrt{\sum_{i,j} |M_{ij}|^2}$$

$\Rightarrow$ Least-squares problem. Solve by QR factorization or SVD decomp.

NLA is the workhorse under most numerical methods, but it works best in tandem with other fields of math.

Example 3: Nonlinear problems

For interacting quantum systems (e.g. Quantum Chem.)

$$-\Delta u + V(u)\,u = E u$$

↑ potential depends on atomic orbitals!

Iterative solution: linearization + NLA

Choose $u = u_0$ (initial guess)

Solve $-\Delta u_1 + V(u_0)\,u_1 = E u_1$ as above,

Solve $-\Delta u_2 + V(u_1)\,u_2 = E u_2$

(repeat) ⋮

Convergence?

Remarks

=> Using Calculus tools to "linearize" and then applying NLA is a powerful paradigm for nonlinear problems.

=> It is a key ingredient in many optimization algorithms.

New Topic / # Floating Pt. Numbers

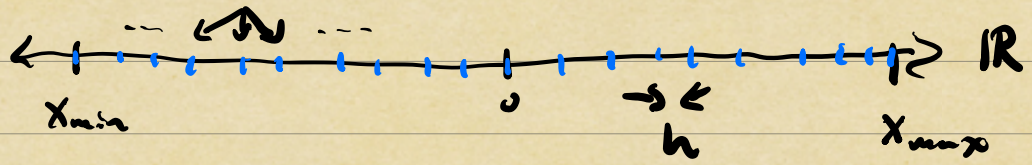
To do linear algebra, we need to do arithmetic ($+ - \times \div$) on $\mathbb{R}$ - the "real #'s" - or $\mathbb{C}$ (complex #'s)

We'll focus on $\mathbb{R}$, but not much changes for $\mathbb{C}$

Q: Computers have finite memory, so how do we work with real #s?

[Intuitively, The challenge is that the real #'s are unbounded and form a continuum.]

Idea 1: "Discretize" the continuum into equally spaced points



Mathematically, $F_h = \{ x_{min} \leq (\text{integer}) \times h \leq x_{max} \}$

On computer, store $\pm$ _ _ _ _ _ _ _ _ _ _ $\times h$

$\pm$: Sign

_ _ _ _ _ _ _ _ _ _: each bit stores a digit of the integer

Example: $h = 10^{-2}$, 7 digit integers

$$\pm\ 1\ 2\ 0\ 0\ 4\ .\ 7\ 2$$

↑ h controls position of decimal pt.

"Fixed point" arithmetic

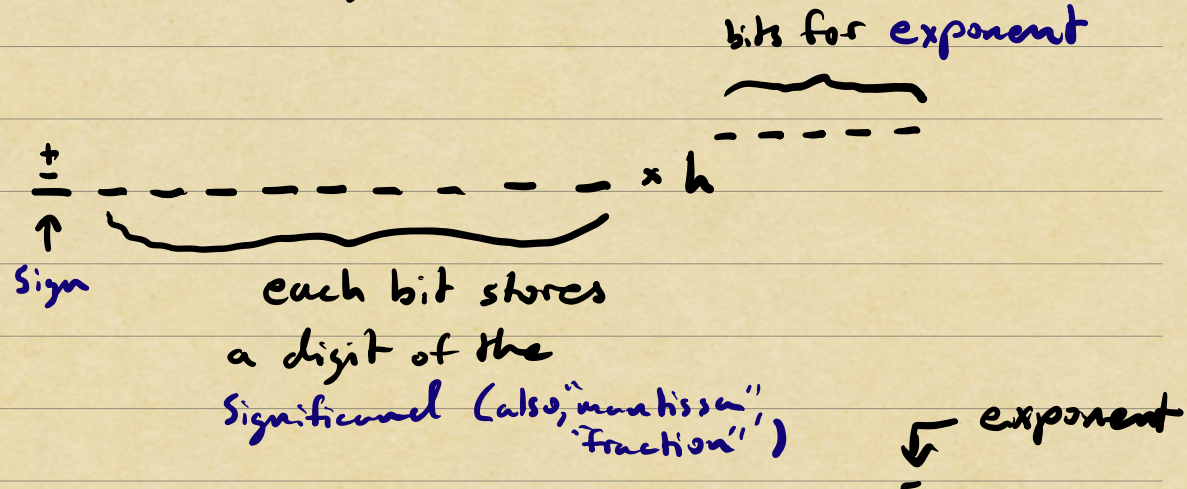
⇒ Round result of $\{+, -, \times, \div\}$ to nearest $x \in F_h$

$.02 \times .01 = .0002$

$.02 \boxtimes .01 = F_h(.0002) = 0$

Lost all significant digits!

Idea 2: "Floating Point"



mimics scientific notation: 1.263×10^{-5}

Significant digits

Mult. w/sci. not.

$$.02 \times .01 = (2 \times 10^{-2}) \times (1 \times 10^{-2})$$
$$= 2 \times 10^{-4} \leftarrow \text{update exp.}$$
$$\uparrow \text{update integer.}$$

Decimal location changes or "floats" depending on size of #. This frees up all bits of the mantissa to preserve sig. digits during multiplication/division.

floating point #'s (FF) have the form

$$x = \pm \underbrace{d_0.d_1 \cdots d_{p-1}}_{\text{Significand}} \times \beta^{\overbrace{e}^{\text{exponent}}} \qquad 0 \le d_x < \beta$$

p-precision (# digits)

In practice, finite storage for exponent means

$$e_{min} \le e \le e_{max} \qquad \begin{matrix}\text{overflow} \\ \text{underflow}\end{matrix}$$

(There is also 0, ±Inf, NaN.)

On most computers IEEE single & double prec.

$\beta = 2$ (binary)

	Single (32 bits)	double (64)
sign	1	1
exp	8	11
mantissa	23	52

Floating point #'s are <u>not</u> equally spaced!

The spacing changes relative to size of #

<u>Key property</u>: For all $x \in \mathbb{R}$, there is an $x' \in \mathbb{F}$ such that $|x' - x| \leq \varepsilon_{mach} |x|$

where $\varepsilon_{mach} = \frac{1}{2} \beta^{1-p}$

In Double prec. $\varepsilon_{mach} = 2^{-52} \approx 2.22 \times 10^{-16}$

Single prec. $\varepsilon_{mach} = 2^{-23} \approx 1.19 \times 10^{-7}$

<u>Floating Pt. Arithmetic</u>

<u>Real Arithmetic</u>	<u>F.P. Arithmetic</u>
$+, -, \times, \div : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$	$\oplus, \ominus, \otimes, \oslash : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$

$fl : \mathbb{R} \to \mathbb{F}$, $fl(x)$ rounds x to closest $x' \in \mathbb{F}$.

Exact rounding: $x \odot y = fl(x \cdot y)$

for any $x, y \in \mathbb{F}$ and $\cdot = +, -, \times, \div$

This means that for any $x, y \in \mathbb{F}$, we have

$$x \odot y = (x \cdot y)(1+\varepsilon)$$

where $|\varepsilon| \leq \varepsilon_{mach}$.

"Fundamental Axiom of Floating Point Arithmetic"

* This ignores over/underflow, when numbers are too big/small for e_{max}, e_{min} to capture.

Important points: (See iJulia notebook)

- Decimal input/output means rounding on inputs & outputs!
- F-P-A is not associative
  $(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$
- Over/Underflow, Inf, NaN
- Catastrophic cancellation
- Accumulation of Rounding Errors

New topic

Accuracy: Stability of FP algorithms

The algorithms of NLA are usually implemented in floating point arithmetic. They are built on $\{\oplus, \ominus, \otimes, \oslash\}$.

A key concern: how do rounding errors accumulate?

Suppose we want to compute $f: \mathbb{R}^n \to \mathbb{R}^m$

Ideally, we would like to have an algorithm $\tilde{f}: \mathbb{R}^n \to \mathbb{R}^m$ such that for any $x \in \mathbb{R}^n$

$$\|\tilde{f}(x) - f(x)\| = \|f(x)\| \mathcal{O}(\varepsilon_{mach})$$

↑ Big-$\mathcal{O}$ notation "on the order of"

I.e., relative error in the output is "small."

Such an algorithm is called forward stable.

Big "$\mathcal{O}$" notation $f(t) = \mathcal{O}(g(t))$ as $t \to \infty$ if there is a constant $C > 0$ and $t_0 > 0$ such

that $|f(t)| \leq C g(t)$ for all $t \leq t_0$.

E.g. 1 $p(t) = t + t^2 \leq 2t$ for all $t \leq 1$

$$\Rightarrow p(t) = O(t)$$

E.g. 2 $f(t) = \sin t = t - \frac{t^3}{6} + \frac{t^5}{120} + \dots$

$$= t + O(t^3)$$

Sensitivity and conditioning

Unfortunately, forward stability is often impossible to achieve. This is because functions can be highly sensitive to small perturbations in their inputs - we call such a function ill-conditioned.

E.g. $x \in \mathbb{R} \xrightarrow{\text{round}} \tilde{x} \in \mathbb{F}$ $\quad |x - \tilde{x}| \leq |x|(1 + \varepsilon_{mach})$

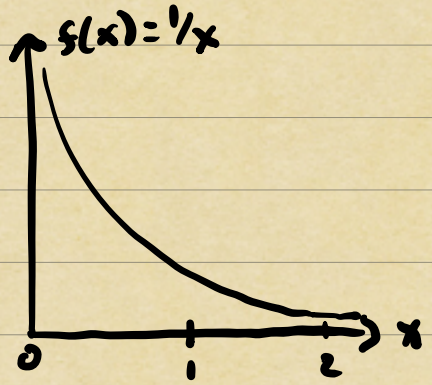
(Scalar functions) ↓ ↓

$f(x)$ $\quad f(\tilde{x}) = f(x + \varepsilon)$ $\quad |\varepsilon| \leq \varepsilon_{mach}$

if f continuously differentiable $\quad |f(x) - f(x+\varepsilon)| \approx |f'(x)| \varepsilon_{mach}$

Relative error: $|f(x) - f(x+\varepsilon)| \approx |f(x)| \underbrace{\frac{|f'(x)|}{|f(x)|}} \varepsilon_{mach}$

measures sensitivity



$$f'(x) = \frac{-1}{x^2} \Rightarrow \left|\frac{f'(x)}{f(x)}\right| = \frac{1}{x}$$

Backward Stability

In light of perturbations due to inputs, we usually can't get forward stable algorithms for problem classes that include ill-conditioned functions. How should we judge the quality of our algorithms then and understand their accumulation of rounding errors?

⇒ Ask that the algorithm behaves as if it computed exactly the right solution for a perturbed input.

We want $\tilde{f}: \mathbb{R}^n \to \mathbb{R}^m$ such that for any $x \in \mathbb{R}^n$

$$\tilde{f}(x) = f(\tilde{x}) \text{ for some } \tilde{x}$$

s.t. $\|x - \tilde{x}\| \leq \|x\| \, O(\varepsilon_{mach})$

We say the algorithm $\tilde{f}$ is **backward stable**.

Key idea: Backward stable (algorithm) + well-conditioned (problem)

$\star$ = "small" forward error

E.g. (back to scalar example)

$$|f(x) - \tilde{f}(x)| \overset{\text{backward stability}}{=} |f(x) - f(\tilde{x})|$$

$$\approx |f(x)| \underbrace{\frac{|f'(x)|}{|f(x)|}} \mathcal{O}(\varepsilon_{\text{mach}})$$

well conditioned = not too large

$\star$ have to be careful about what we mean by "small," constants in big "O" notation, etc.

Thm: Compute $f: \mathbb{R}^n \to \mathbb{R}^m$ with condition number κ at input $x \in \mathbb{R}^n$ on computer satisfying fundamental floating point axioms. Then,

$$\|\tilde{f}(x) - f(x)\| = \|f(x)\| \mathcal{O}(\kappa(x) \varepsilon_{\text{mach}})$$

Condition # κ measures sensitivity: next lecture

Before diving into them, let's see an example.

Example: Sum n numbers $\underbrace{x_1, \dots, x_n}_{\underline{x}} \in \mathbb{F}$

Problem $f(\underline{x}) = \sum_{i=1}^{n} x_i$

Algorithm

Pseudocode:

```
s = 0
for i = 1, ..., n
    s = s + x_i
end
```

floating point

$\tilde{s}_1 = x_1$

$\tilde{s}_i = \tilde{s}_{i-1} \oplus x_i$

$\tilde{f}(\underline{x}) = \tilde{s}_n$

1) Stability

Claim: Algorithm is backward stable.

Pf

We'll show $\tilde{f}(\underline{x}) = \sum_{i=1}^{n} x_i \prod_{k=1}^{n} (1+\varepsilon_k)$

with $|\varepsilon_k| \leq \varepsilon_{mach}$, using induction on n.

$n = 1$ $\quad \tilde{f}(\underline{x}) = x_1$ $\quad$ (WLOG, assume $x_1 \in \mathbb{F}$)

$n > 1$ Suppose $\tilde{S}_{n-1} = \hat{\sum_{i=1}^{n-1}} x_i \hat{\prod_{k=i}^{n-1}} (1+\varepsilon_k)$

$$\tilde{S}_n = \tilde{S}_{n-1} \oplus x_n = (\tilde{S}_{n-1} + x_n)(1+\varepsilon_n)$$

$$= \left[\hat{\sum_{i=1}^{n-1}} x_i \hat{\prod_{k=i}^{n-1}} (1+\varepsilon_i)\right](1+\varepsilon_n) \qquad |\varepsilon_n| < \varepsilon_{mach}$$

$$+ x_n(1+\varepsilon_n)$$

$$= \hat{\sum_{i=1}^{n}} x_i \hat{\prod_{k=i}^{n}} (1+\varepsilon_k) \quad \checkmark$$

So, $\tilde{f}(x) = \hat{\sum_{i=1}} x_i(1+\delta_i)$ where

$$(1+\delta_i) = \hat{\prod_{k=i}^{n}} (1+\varepsilon_k) = 1 + \hat{\sum_{k=i}^{n}} \varepsilon_k + \mathcal{O}(\varepsilon_{mach}^2)$$

$$= 1 + n\varepsilon_{mach} + \mathcal{O}(\varepsilon_{mach}^2)$$

This means $\tilde{f}(x) = f(\tilde{x})$ where $\tilde{x}_i = x_i(1+\delta_i)$

so $|x_i - \tilde{x}_i| = |x_i||\delta_i| \leq |x_i|\,\mathcal{O}(\varepsilon_{mach})$

$$\|x - \tilde{x}\| = \sqrt{\hat{\sum_{i=1}} |x_i - \tilde{x}_i|^2} \leq \underbrace{\sqrt{\hat{\sum_{i=1}} |x_i|^2}}_{\|x\|} \mathcal{O}(\varepsilon_{mach}) \quad \square$$

2) Accuracy

$$f(x) - \tilde{f}(x) = f(x) - f(\tilde{x}) = \hat{\sum_{i=1}} x_i \delta_i$$

$$\Rightarrow |f(x) - \tilde{f}(x)| \leq \sum_{i=1}^{n} |x_i| |\epsilon_i| \leq n \epsilon_{mach} \sum_{i=1}^{n} |x_i|$$

$$\Rightarrow |f(x) - \tilde{f}(x)| \leq n \epsilon_{mach} |f(x)| \left[\frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} \right] + \mathcal{O}(\epsilon_{mach}^2) + \mathcal{O}(\epsilon_{mach}^2)$$

$$\left[\frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} \right] = \text{condition } \#$$

$= 1$ when $x_i \geq 0$

$\gg 1$ when $|\sum_{i=1}^{n} x_i| \ll \sum_{i=1}^{n} |x_i|$ (cancellation)

It measures the sensitivity of f to perturbations in the inputs.

backward stable algorithm
+ well conditioned problem
building blocks for accurate NLA