18. Mid-term review

Goals for today

- Conceptual review of first half of course
- Technical review

What is numerical analysis about?

- Numerical analysis is about solving **problems** $y = \phi(x)$
- I.e. given **input** x, calculate **output** y

What is numerical analysis about?

- Numerical analysis is about solving **problems** $y = \phi(x)$
- I.e. given **input** x, calculate **output** y

- Example: Given a function ("x"), find a root ("y")

What is numerical analysis about?

- Numerical analysis is about solving **problems** $y = \phi(x)$
- I.e. given **input** x, calculate **output** y

- Example: Given a function ("x"), find a root ("y")

- Cannot solve exactly
- So construct **approximations** $y = \tilde{\phi}(x)$
- Call $\tilde{\phi}$ the **constructed** or **engineered problem**

What is numerical analysis about?

- Numerical analysis is about solving **problems** $y = \phi(x)$
- I.e. given **input** x, calculate **output** y

- Example: Given a function ("x"), find a root ("y")

- Cannot solve exactly
- So construct **approximations** $y = \tilde{\phi}(x)$
- Call $\tilde{\phi}$ the **constructed** or **engineered problem**

- E.g.: "Calculate $(x - 1)$ using floating-point arithmetic"

How do we do this?

- Can break down into three (large!) steps:
  1. Find a way to **construct** an approximation
  2. Show **how good** an approximation it is
  3. **Implement** it via an **algorithm**

Floating-point arithmetic

- Approximate real numbers as **floating-point numbers**

Floating-point arithmetic

- Approximate real numbers as **floating-point numbers**

- Approximate as closest **dyadic rational**:

$$x \simeq \pm 2^e \left(1 + \sum_{n=1}^{d} b_n 2^{-n} \right)$$

with binary digits $b_n \in \{0, 1\}$

Floating-point arithmetic

- Approximate real numbers as **floating-point numbers**

- Approximate as closest **dyadic rational**:

$$x \simeq \pm 2^e \left(1 + \sum_{n=1}^{d} b_n 2^{-n} \right)$$

  with binary digits $b_n \in \{0, 1\}$

- IEEE 754 standard: "Double precision" (`Float64`): $d = 52$ and 11 digits for e

Floating-point arithmetic

- Approximate real numbers as **floating-point numbers**

- Approximate as closest **dyadic rational**:

$$x \simeq \pm 2^e \left(1 + \sum_{n=1}^{d} b_n 2^{-n} \right)$$

  with binary digits $b_n \in \{0, 1\}$

- IEEE 754 standard: "Double precision" (`Float64`): $d = 52$ and 11 digits for e

Evaluating functions

- Evaluate functions like $\exp(x)$ and $\sin(x)$

Evaluating functions

- Evaluate functions like $\exp(x)$ and $\sin(x)$
- Use Taylor polynomial approximations for *small* x

Evaluating functions

- Evaluate functions like $\exp(x)$ and $\sin(x)$
- Use Taylor polynomial approximations for *small* x
- **Argument reduction** for other x (negative and large)
- Relate value of $f(x)$ to $f(r)$ with r small

Convergence

- Often construct problem containing a **parameter** N or h

Convergence

- Often construct problem containing a **parameter** N or h

- Design such that solution of constructed problem $\tilde{\phi}_N$ **converges** to true solution $\phi(x)$ as $N \to \infty$ or $h \to 0$:

$$\phi_N(x) \to \phi(x) \quad \text{as } N \to \infty$$

- *Even though original problem does not contain parameter!*

Convergence II

- Example of convergence: Fixed-point algorithm for root finding
- Want to find a root x^* of f, i.e. $f(x^*) = 0$
- To do so, construct an auxiliary ("helper") problem g such that a **fixed point** $g(x^*) = x^*$ is a root of f
- E.g. $g(x) = x - Cf(x)$, with $C \neq 0$ some constant

Convergence III

- If x^* is a stable fixed point of g then if x_0 is close enough to x^*, the sequence x_n defined by

$$x_{n+1} = g(x_n)$$

  satisfies $x_n \to x^*$
- By running the iteration long enough we obtain an approximation for the root x^*
- Running it longer (with high enough precision) we can obtain approximations that are *arbitrarily good* (i.e. as close as we want)
- E.g. Picard iteration for ODEs: $x^{(n+1)} = x_0 + \int f(x^{(n)})$

Rate of convergence

- How *fast* does $x_n \to x^*$?
- Make a plot of $\delta_n := |x_n - x^*|$ as function of n

Rate of convergence

- How *fast* does $x_n \to x^*$?
- Make a plot of $\delta_n := |x_n - x^*|$ as function of n
- Usually decays too fast to get information from plot
- So plot $\log(\delta_n)$ against n – straight line $\Rightarrow$ $\delta_n \sim C \exp(-\alpha n)$
- Plot $\log(\delta_n)$ against $\log(n)$ – straight line $\Rightarrow \delta_n \sim C x^{-\alpha}$

Order of convergence

- *Rate* of convergence tells us how fast δ_n tends to 0
- **Order** of convergence instead relates consecutive values of δ_n
- Defined as $\lim_{n\to\infty} \frac{\delta_{n+1}}{\delta_n^\alpha}$ if limit exists

Order of convergence

- *Rate* of convergence tells us how fast δ_n tends to 0
- **Order** of convergence instead relates consecutive values of δ_n
- Defined as $\lim_{n\to\infty} \frac{\delta_{n+1}}{\delta_n^\alpha}$ if limit exists

- I.e. $\delta_{n+1} \sim (\delta_n)^\alpha$
- E.g. for $\alpha = 1$ have $\delta_n \sim \exp(-\alpha n)$
- For $\alpha = 2$ have $\Rightarrow \delta_n \sim \exp(-\alpha n^2)$

Fixed-point iterations

- How do we design a fixed-point iteration?

Fixed-point iterations

- How do we design a fixed-point iteration?

1. Rearrange expression for f into fixed-point expression
2. Choose C appropriately in definition of g
3. General methods: secant, Newton, …

Newton method

- Solve nonlinear equation $f(x) = 0$ by repeatedly solving *linear* equations
- $x_{n+1} = x_n - J^{-1} f(x_n)$
- Where $J := \left(\frac{\partial f_i}{\partial x_j}\right)$ is Jacobian matrix

Newton method

- Solve nonlinear equation $f(x) = 0$ by repeatedly solving *linear* equations
- $x_{n+1} = x_n - J^{-1} f(x_n)$
- Where $J := \left(\frac{\partial f_i}{\partial x_j} \right)$ is Jacobian matrix

- Need to be able to calculate derivatives

Automatic differentiation

- Method to calculate derivatives automatically
- By encoding the basic rules

Automatic differentiation

- Method to calculate derivatives automatically
- By encoding the basic rules

- Define dual number type $a + \epsilon\, b$
- $f(a + \epsilon b) = f(a) + \epsilon\, f'(a)\, b$

Automatic differentiation

- Method to calculate derivatives automatically
- By encoding the basic rules

- Define dual number type $a + \epsilon\, b$
- $f(a + \epsilon b) = f(a) + \epsilon\, f'(a)\, b$

- Derivative is given by coefficient of ϵ

Errors

- Consider a problem $y = \phi(x)$
- Suppose perturb input by Δx to $\tilde{x}$
- Output changes by $\Delta y := \phi(\tilde{x}) - \phi(x) = \tilde{y} - y$

Errors

- Consider a problem $y = \phi(x)$
- Suppose perturb input by Δx to $\tilde{x}$
- Output changes by $\Delta y := \phi(\tilde{x}) - \phi(x) = \tilde{y} - y$

- **Absolute error**: $\Delta x := |\tilde{x} - x|$
- **Relative error**: $\delta x := \frac{\Delta x}{x}$

Conditioning

- Conditioning tells us how sensitive a *problem* is$
- Concept of conditioning is independent of which *algorithm* we use to solve problem

Conditioning

- Conditioning tells us how sensitive a *problem* is$
- Concept of conditioning is independent of which *algorithm* we use to solve problem
- (Relative) condition number $\kappa := \left\|\frac{\delta y}{\delta x}\right|$

Conditioning

- Conditioning tells us how sensitive a *problem* is$
- Concept of conditioning is independent of which *algorithm* we use to solve problem

- (Relative) condition number $\kappa := \left\|\frac{\delta y}{\delta x}\right\|$

- $-\log_{10}(\kappa)$ is number of accurate digits lost in calculation

Interpolation

- Given data $(t_i, y_i)_{i=0}^{N}$, find polynomial of degree N that passes through all data points

Interpolation

- Given data $(t_i, y_i)_{i=0}^{N}$, find polynomial of degree N that passes through all data points
- Exactly solvable (in exact arithmetic):

Interpolation

- Given data $(t_i, y_i)_{i=0}^N$, find polynomial of degree N that passes through all data points

- Exactly solvable (in exact arithmetic):

  - Solve linear system (Vandermonde matrix)
  - Or write down explicit Lagrange interpolant (see also later)

Interpolation II

- Given function f, data from **sampling** f
- Define $y_i := f(t_i)$

Interpolation II

- Given function f, data from **sampling** f
- Define $y_i := f(t_i)$

- For smooth f can get polynomial *arbitrarily close* to f
- By interpolating in Chebyshev points t_i and taking $N \to \infty$

Applications of interpolation

. . .

\ \

- Use to find finite-difference approximations of derivatives
- E.g.

$$f'(x) \simeq \frac{f(x-h) + f(x+h)}{2}$$

Applications of interpolation

. . .

\ \

- Use to find finite-difference approximations of derivatives
- E.g.

$$f'(x) \simeq \frac{f(x-h) + f(x+h)}{2}$$

- Use to calculate approximations of integrals (quadrature)
- E.g.

Ordinary differential equations (ODEs)

- Solve $\dot{x} = f(x)$
- I.e. $\dot{x}(t) = f(x(t)) \quad \forall t$

Ordinary differential equations (ODEs)

- Solve $\dot{x} = f(x)$
- I.e. $\dot{x}(t) = f(x(t)) \quad \forall t$

- Solution is a *function* $t \mapsto x(t)$
- We need to approximate the unknown function $x(t)$

Numerical methods for ODEs

- Simplest numerical method: **Euler method**:

$$x(t+h) = x(t) + hf(x)$$

Numerical methods for ODEs

- Simplest numerical method: **Euler method**:

$$x(t+h) = x(t) + hf(x)$$

- Taylor methods: expand in Taylor series in h
- $x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t) + \cdots$

Numerical methods for ODEs

- Simplest numerical method: **Euler method**:

$$x(t+h) = x(t) + hf(x)$$

- Taylor methods: expand in Taylor series in h
- $x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t) + \cdots$

- Runge–Kutta: Several **stages**, each an Euler step
- RK reproduce Taylor expansion
- Order-p method if local error $\mathcal{O}(h^{p+1})$

Adaptive ODE methods

- Run two different ODE methods to estimate local error
- Use local error estimate to choose size h of time step

Adaptive ODE methods

- Run two different ODE methods to estimate local error
- Use local error estimate to choose size h of time step

- Allows to control global error

Adaptive ODE methods

- Run two different ODE methods to estimate local error
- Use local error estimate to choose size h of time step

- Allows to control global error

- Embedded Runge–Kutta methods reuse evaluations of f to create two methods with different orders

Summary

- Design approximate problem (algorithm) that converges to true solution
- Find rate of convergence
- Implement