Introduction to Numerical Integration, Optimization, Differentiation and Ordinary Differential Equations

Ralph C. Smith
Department of Mathematics
North Carolina State University

Overview: Elements of Numerical Analysis

- Numerical integration
- Optimization
- Numerical differentiation
- Ordinary Differential equations (ODE)

Motivation

What does an integral represent?

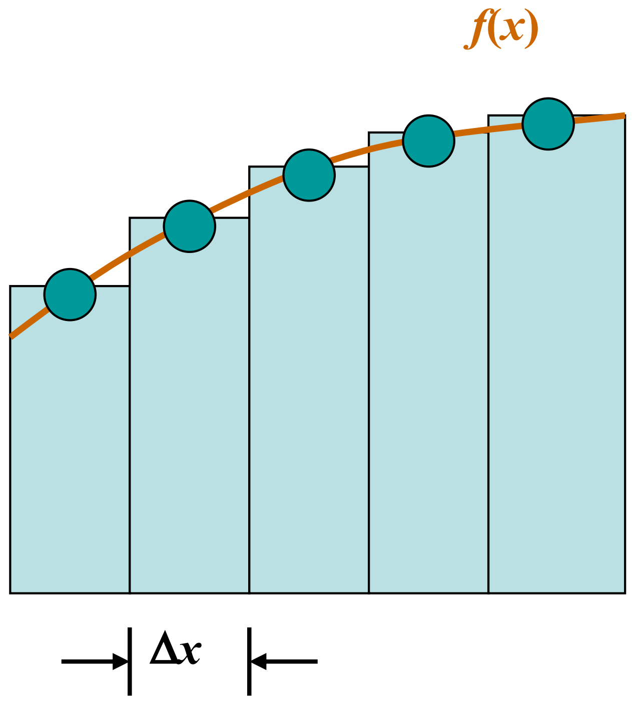
$$\int_a^b f(x)dx = \text{area} \qquad \int_c^d \int_a^b f(x)dxdy = \text{volume}$$

Basic definition of an integral:

$$\int_a^b f(x)dx = \lim_{n\to\infty}\sum_{k=1}^{n} f(x_k)\Delta x$$

where $$\Delta x = \frac{b-a}{n}$$

sum of height × width



Numerical Quadrature

Motivation: Computation of expected values requires approximation of integrals

$$\mathbb{E}[u(t,x)] = \int_{\mathbb{R}^p} u(t,x,q)\rho(q)dq$$

Example: HIV model

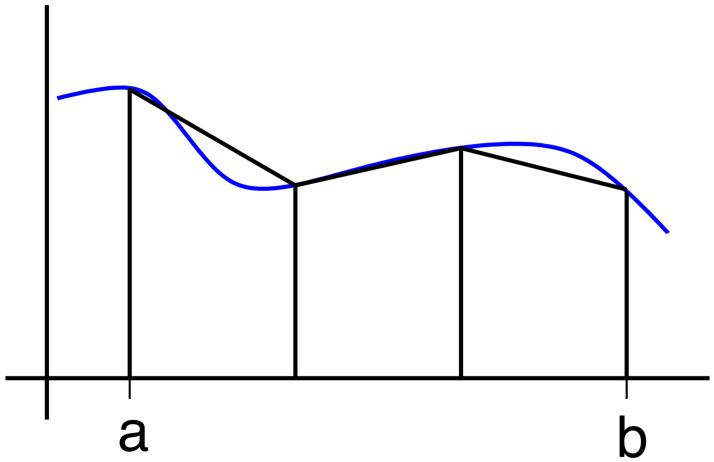
$$\mathbb{E}[V(t)] = \int_{\mathbb{R}^6} V(t,q)\rho(q)dq$$

Numerical Quadrature:

$$\int_{\mathbb{R}^p} f(q)\rho(q)dq \approx \sum_{r=1}^{R} f(q^r)w^r$$

Questions:

- How do we choose the quadrature points and weights?
  - E.g., Newton-Cotes; e.g., trapezoid rule



$$\int_a^b f(q)dq \approx \frac{h}{2}\left[f(a) + f(b) + 2\sum_{r=1}^{R-2} f(q^r)\right]$$

$$q^r = a + hr\ ,\ \ h = \frac{b-a}{R-1}$$

Error: $\mathcal{O}(h^2)$

Numerical Quadrature

Motivation: Computation of expected values requires approximation of integrals

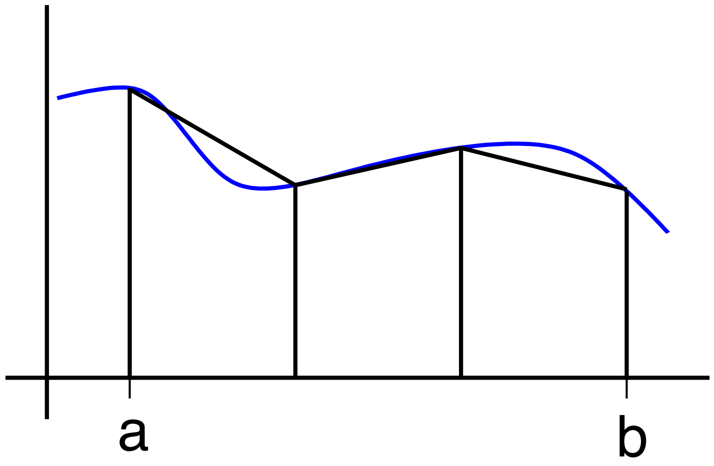
$$\mathbb{E}[u(t,x)] = \int_{\mathbb{R}^p} u(t,x,q)\rho(q)dq$$

Numerical Quadrature:

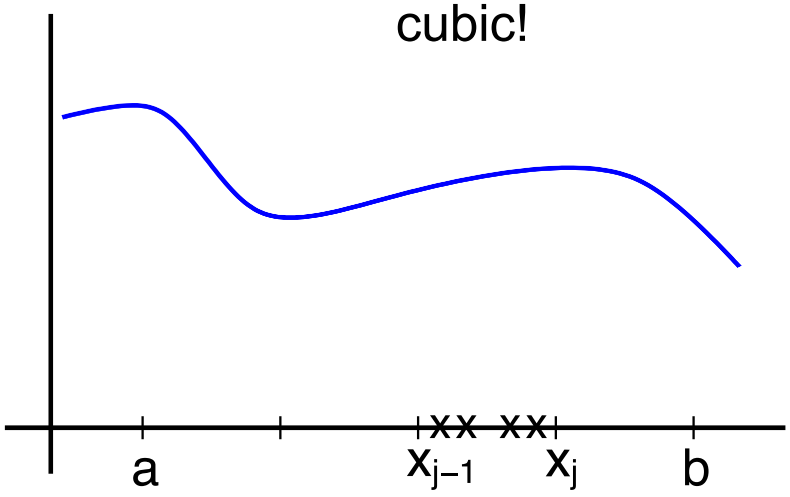
$$\int_{\mathbb{R}^p} f(q)\rho(q)dq \approx \sum_{r=1}^{R} f(q^r)w^r$$

Questions:

- How do we choose the quadrature points and weights?
    - E.g., Newton-Cotes, Gaussian algorithms



Error: Exact for polynomials up to cubic!



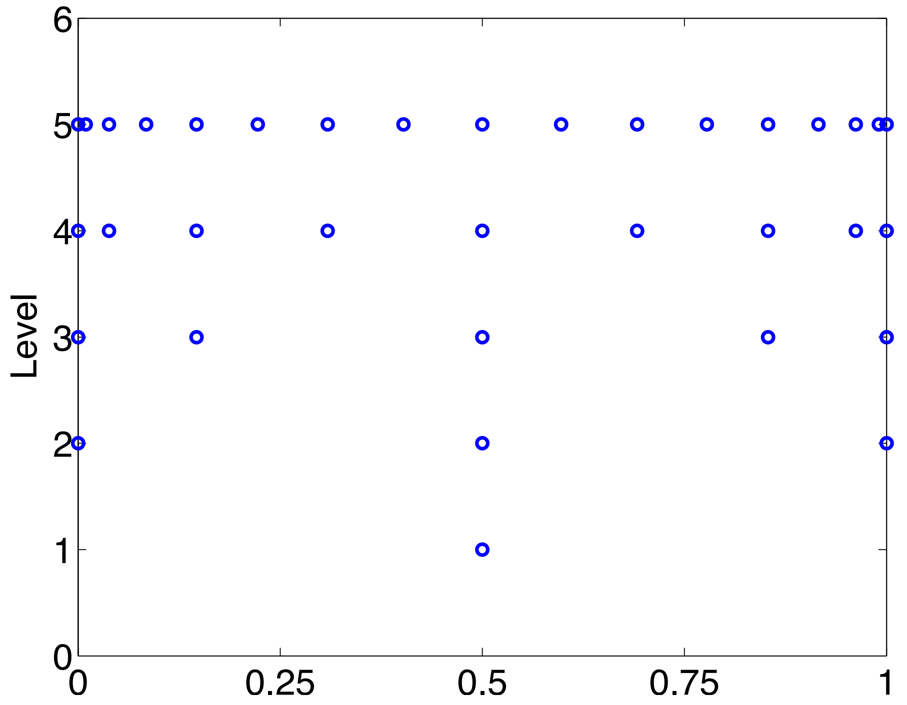
Numerical Quadrature

Numerical Quadrature:

$$\int_{\mathbb{R}^p} f(q)\rho(q)dq \approx \sum_{r=1}^{R} f(q^r)w^r$$

Questions:

- Can we construct nested algorithms to improve efficiency?
    - E.g., employ Clenshaw-Curtis points

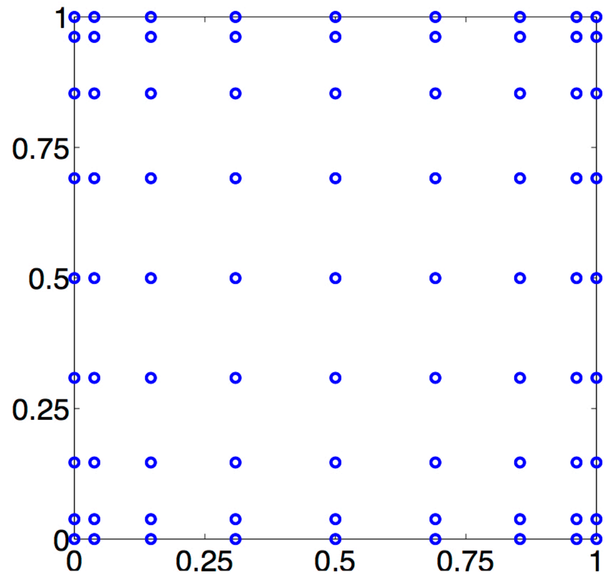


Numerical Quadrature

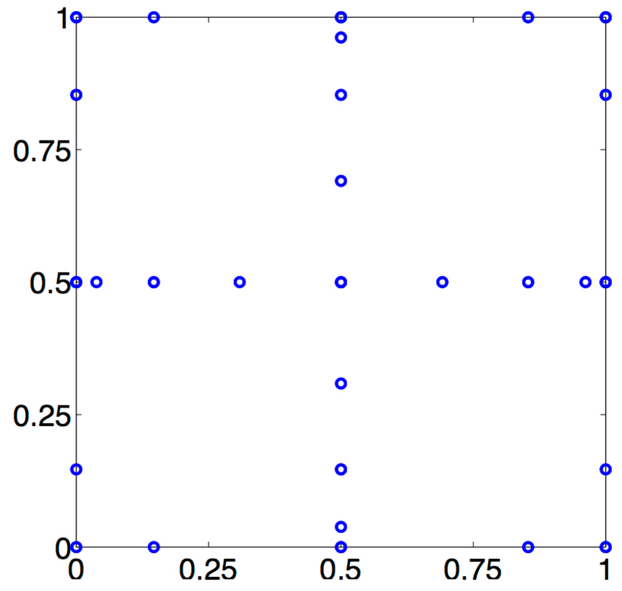
Questions:

- How do we reduce required number of points while maintaining accuracy?

Tensored Grids: Exponential growth



Sparse Grids: Same accuracy



p	R_ℓ	Sparse Grid $\mathcal{R}$	Tensored Grid $R = (R_\ell)^p$
2	9	29	81
5	9	241	59,049
10	9	1581	$> 3 \times 10^9$
50	9	171,901	$> 5 \times 10^{47}$
100	9	1,353,801	$> 2 \times 10^{95}$

Numerical Quadrature

Problem:

- Accuracy of methods diminishes as parameter dimension p increases
- Suppose $f \in C^{\alpha}([0,1]^p)$
- Tensor products: Take R_ℓ points in each dimension so $R = (R_\ell)^p$ total points
- Quadrature errors:

Newton-Cotes: $E \sim \mathcal{O}(R_\ell^{-\alpha}) = \mathcal{O}(R^{-\underline{\alpha/p}})$

Gaussian: $E \sim \mathcal{O}(e^{-\beta R_\ell}) = \mathcal{O}\left(e^{-\beta \underline{\sqrt[p]{R}}}\right)$

Sparse Grid: $E \sim \mathcal{O}\left(\mathcal{R}^{-\alpha} \log(\mathcal{R})^{\underline{(p-1)(\alpha+1)}}\right)$

Numerical Quadrature

Problem:

- Accuracy of methods diminishes as parameter dimension p increases
- Suppose $f \in C^\alpha([0,1]^p)$
- Tensor products: Take R_ℓ points in each dimension so $R = (R_\ell)^p$ total points
- Quadrature errors:

$$\text{Newton-Cotes: } E \sim \mathcal{O}(R_\ell^{-\alpha}) = \mathcal{O}(R^{-\underline{\alpha/p}})$$

$$\text{Gaussian: } E \sim \mathcal{O}(e^{-\beta R_\ell}) = \mathcal{O}\left(e^{-\beta \underline{\sqrt[p]{R}}}\right)$$

$$\text{Sparse Grid: } E \sim \mathcal{O}\left(\mathcal{R}^{-\alpha} \log(\mathcal{R})^{\underline{(p-1)(\alpha+1)}}\right)$$

- Alternative: Monte Carlo quadrature

$$\int_{\mathbb{R}^p} f(q)\rho(q)dq \approx \frac{1}{R}\sum_{r=1}^{R} f(q^r) \quad , \quad E \sim \left(\frac{1}{\sqrt{R}}\right)$$

- Advantage: Errors independent of dimension p
- Disadvantage: Convergence is very slow!

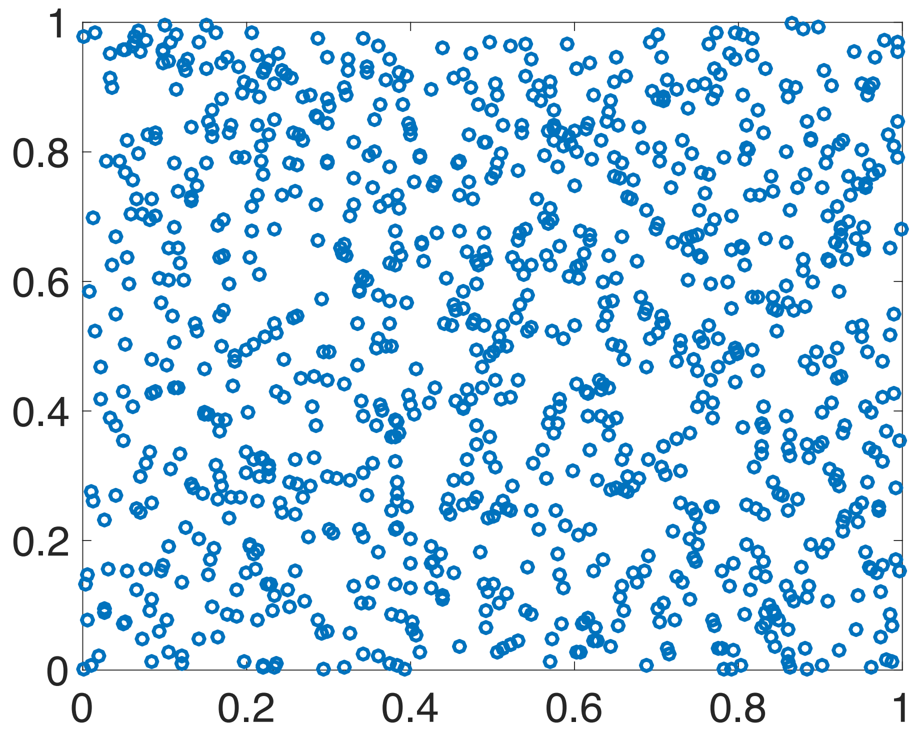
Conclusion: For high enough dimension p, monkeys throwing darts will beat Gaussian and sparse grid techniques!

Monte Carlo Sampling Techniques

Issues:

- Very low accuracy and slow convergence
- Random sampling may not “randomly” cover space …

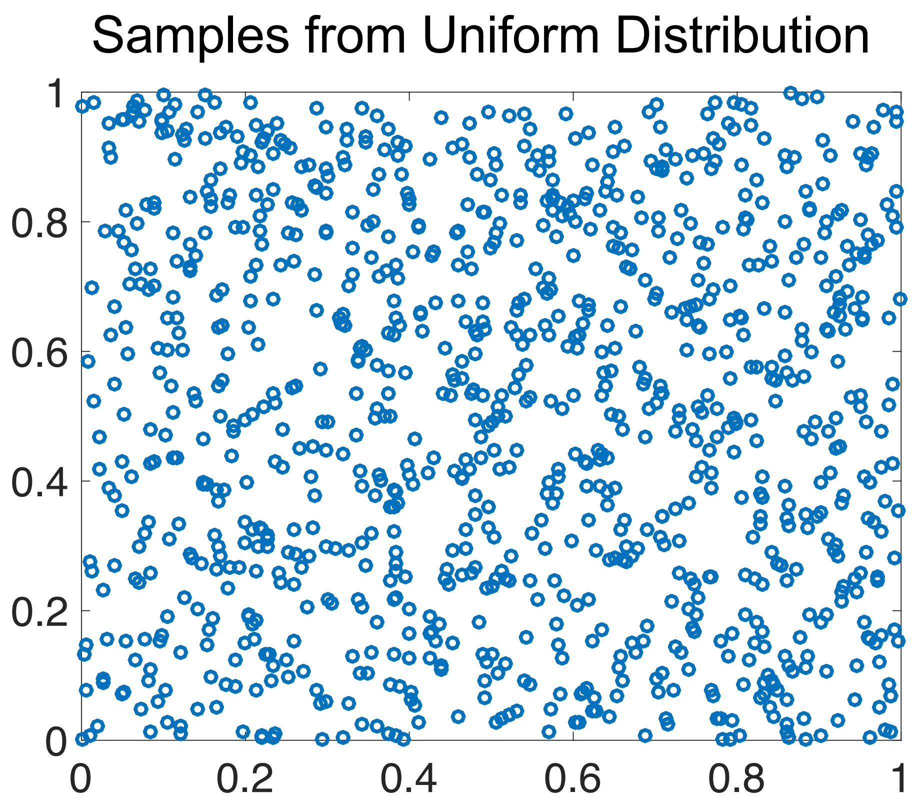
Samples from Uniform Distribution

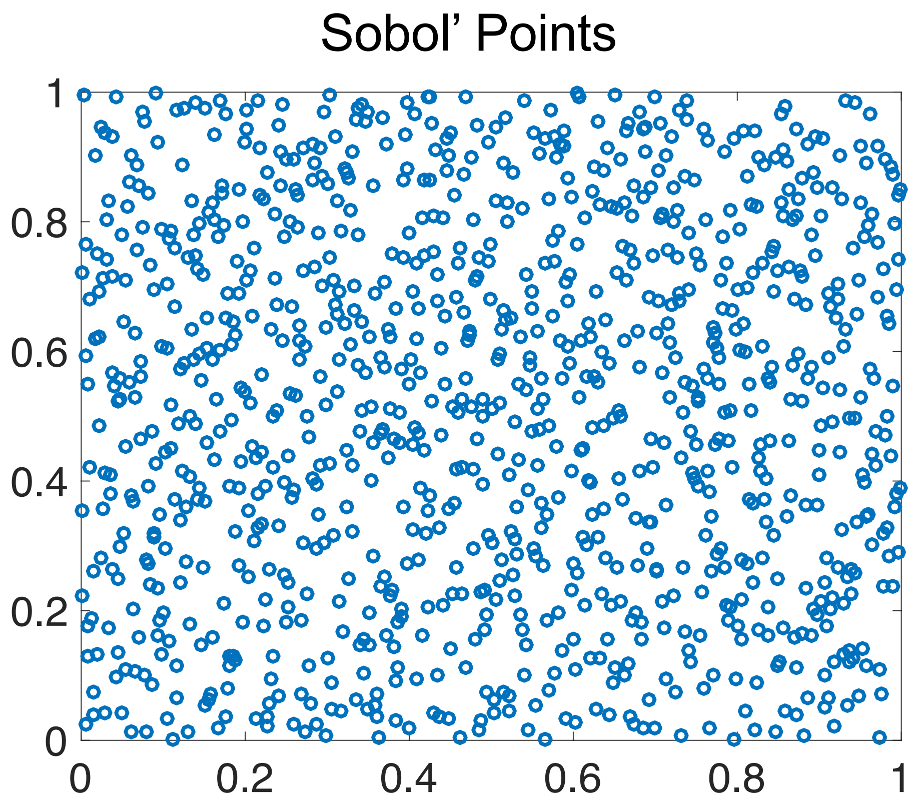


Monte Carlo Sampling Techniques

Issues:

- Very low accuracy and slow convergence
- Random sampling may not “randomly” cover space …





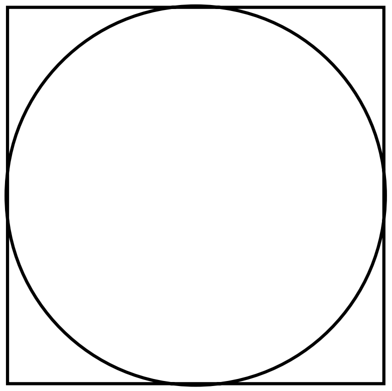
Sobol’ Sequence: Use a base of two to form successively finer uniform partitions of unit interval and reorder coordinates in each dimension.

Monte Carlo Sampling Techniques

Example: Use Monte Carlo sampling to approximate area of circle

$$\frac{A_c}{A_s} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

$$\Rightarrow A_c = \frac{\pi}{4} A_s$$



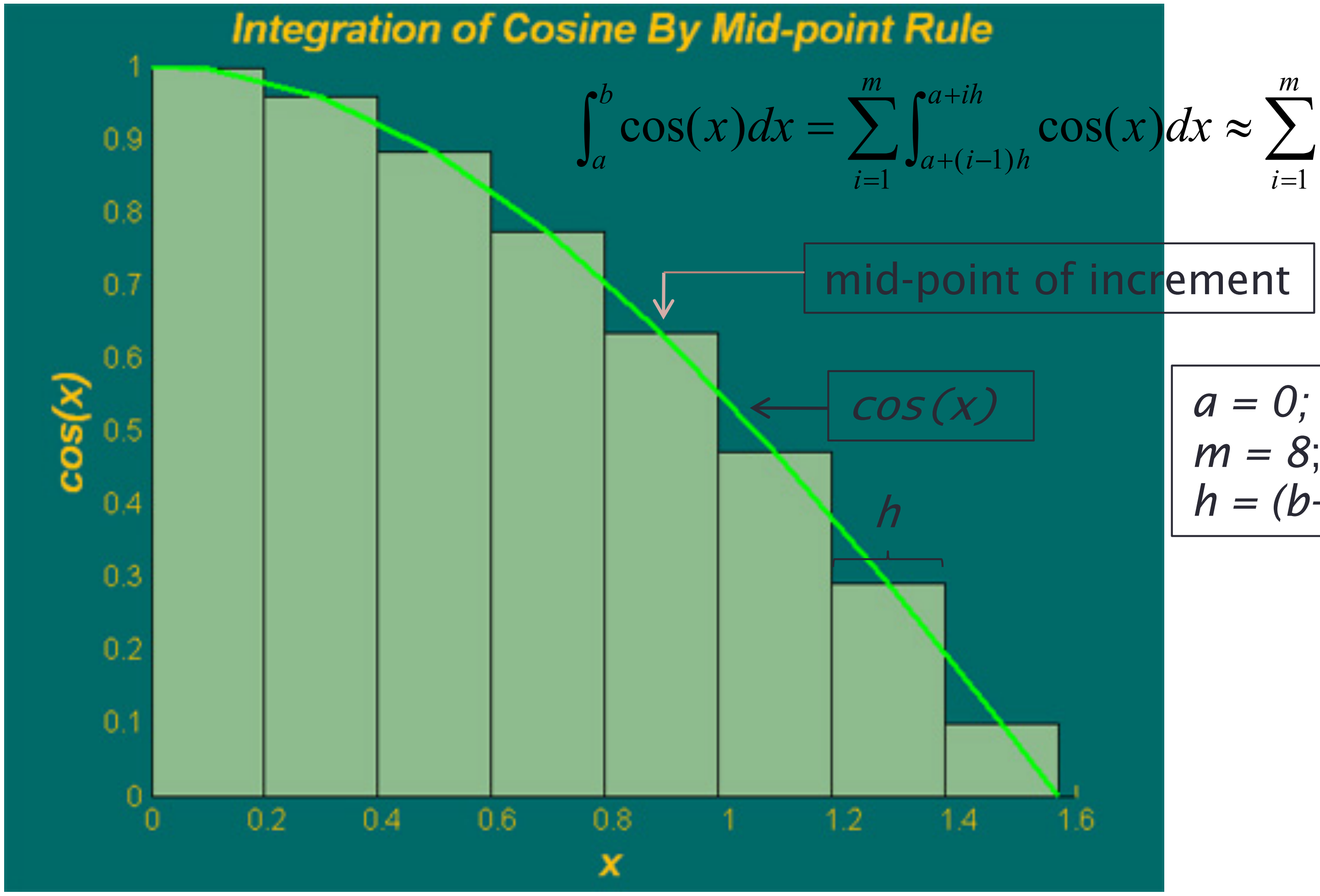
Strategy:

- Randomly sample N points in square $\Rightarrow$ approximately $N\frac{\pi}{4}$ in circle
- Count M points in circle

$$\Rightarrow \pi \approx \frac{4M}{N}$$

Numerical Integration: Example

- Integration of cosine from 0 to $\pi/2$.
- Use mid-point rule for simplicity.

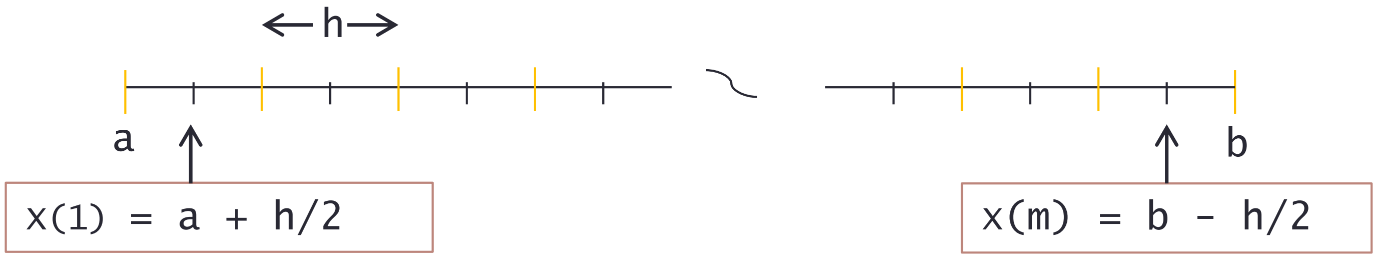


$$\int_a^b \cos(x)dx = \sum_{i=1}^{m} \int_{a+(i-1)h}^{a+ih} \cos(x)dx \approx \sum_{i=1}^{m} \cos(a+(i-\tfrac{1}{2})h)h$$

```
a = 0; b = pi/2;  % range
m = 8;  % # of increments
h = (b-a)/m;  % increment
```

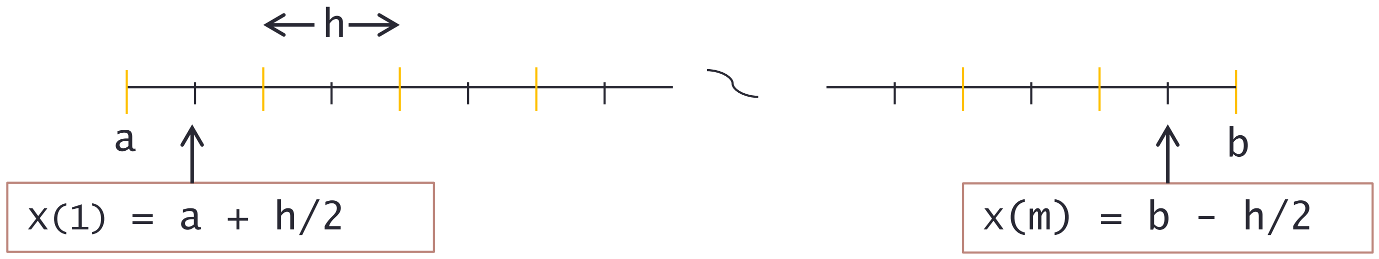
Numerical Integration

```
% integration with for-loop
tic
   m = 100;
   a = 0;                 % lower limit of integration
   b = pi/2;              % upper limit of integration
   h = (b-a)/m;           % increment length
   integral = 0;          % initialize integral
   for i=1:m
     x = a+(i-0.5)*h;     % mid-point of increment i
     integral = integral + cos(x)*h;
   end
toc
```



Numerical Integration

```
% integration with vector form
tic
   m = 100;
   a = 0;                   % lower limit of integration
   b = pi/2;                % upper limit of integration
   h = (b-a)/m;             % increment length
   x = a+h/2:h:b-h/2;       % mid-point of m increments
   integral = sum(cos(x))*h;
toc
```



Optimization

Example: Helmholtz energy $\psi(P,q) = \underline{\alpha_1} P^2 + \underline{\alpha_{11}} P^4 + \underline{\alpha_{111}} P^6$

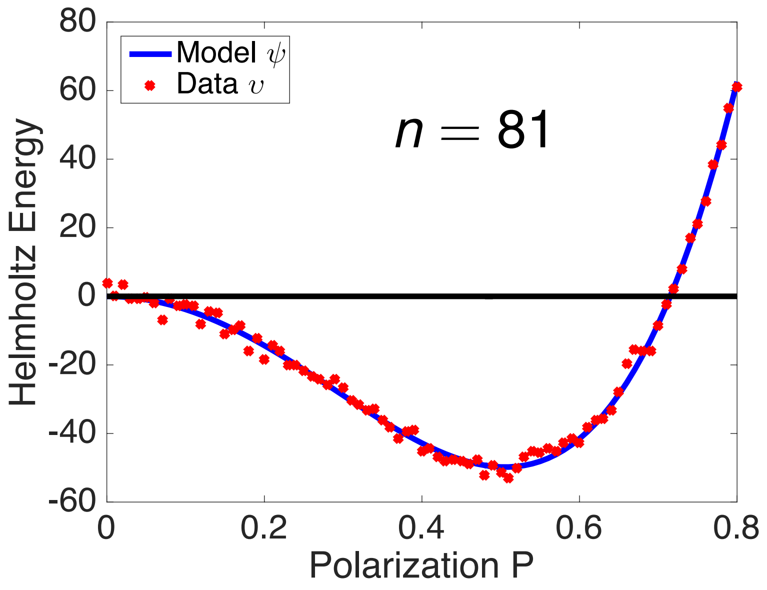
$$q = [\alpha_1, \alpha_{11}, \alpha_{111}]$$

Statistical Model: Describes observation process

$$\upsilon_i = \psi(P_i, q) + \varepsilon_i \quad , \; i = 1, \dots, n$$

Point Estimates: Ordinary least squares

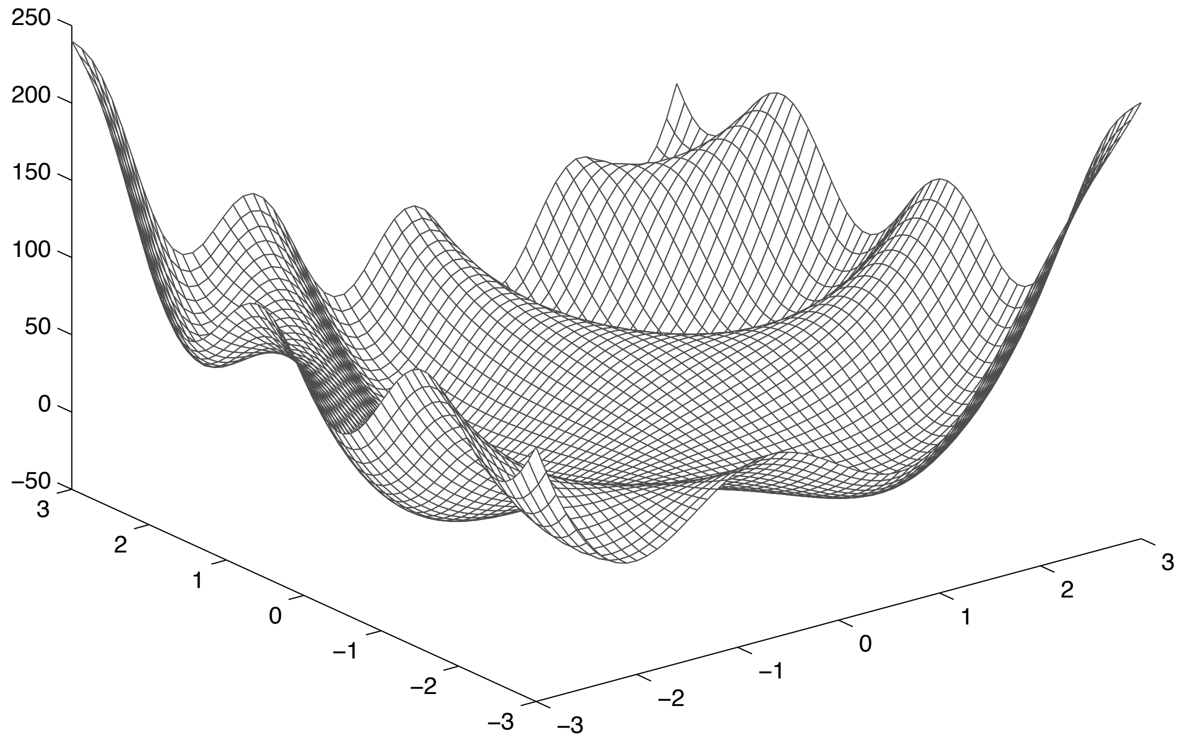
$$q^0 = \arg\min_q \frac{1}{2} \sum_{j=1}^{n} [\upsilon_i - \psi(P_i, q)]^2$$



Note: Optimization is critical for model calibration and design

Optimization

Issues: Nonconvex problems having numerous local minima



Tie between Optimization and Root Finding

Problem 1: minimize $f(x)$, $f : \mathbb{R}^n \to \mathbb{R}$

Problem 2: solve $F(x) = 0$ where $F : \mathbb{R}^n \to \mathbb{R}^n$

Note:

- If x^* solves (1), it also solves (2) with $F(x) = \nabla f(x)$
- If x^* solves (2), it solves (1) with $f(x) = \|F(x)\|^2 = F(x)^T F(x)$

Optimization

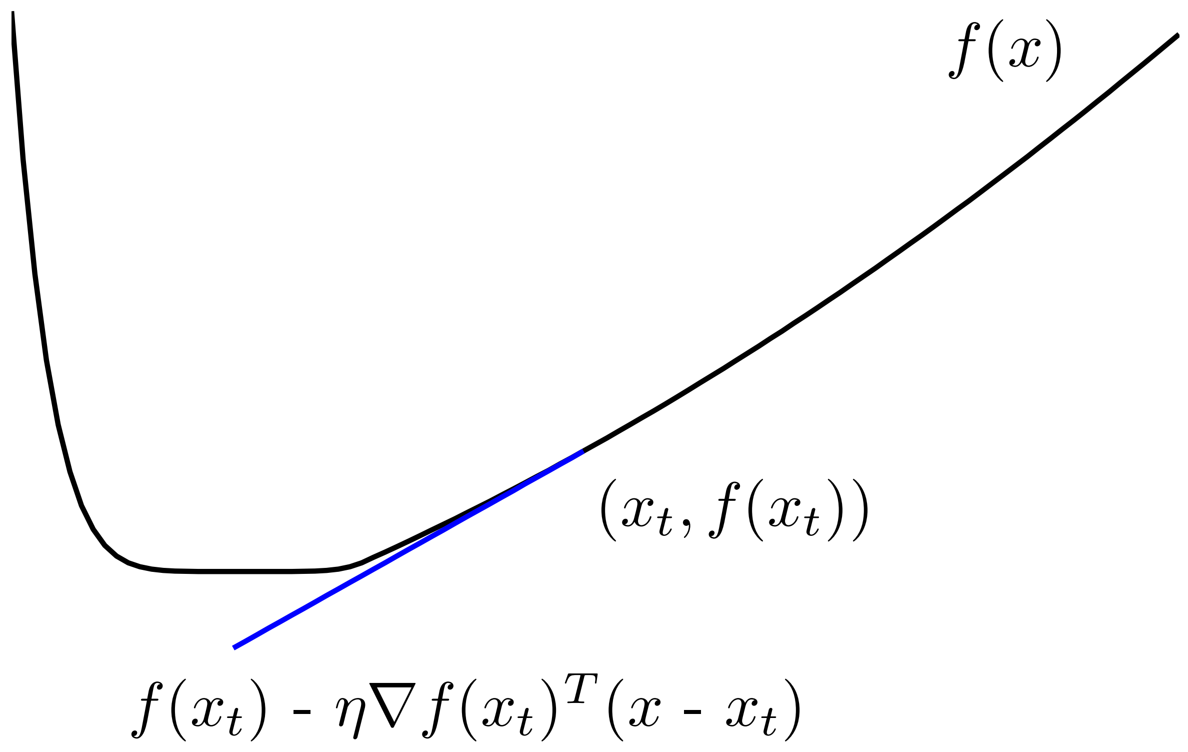
Method 1: Gradient descent

Goal: $\min_x f(x)$

Strategy: Employ iteration

$$x_{t+1} = x_t - \eta_t \nabla f(x_t)$$

where η_t is a stepsize



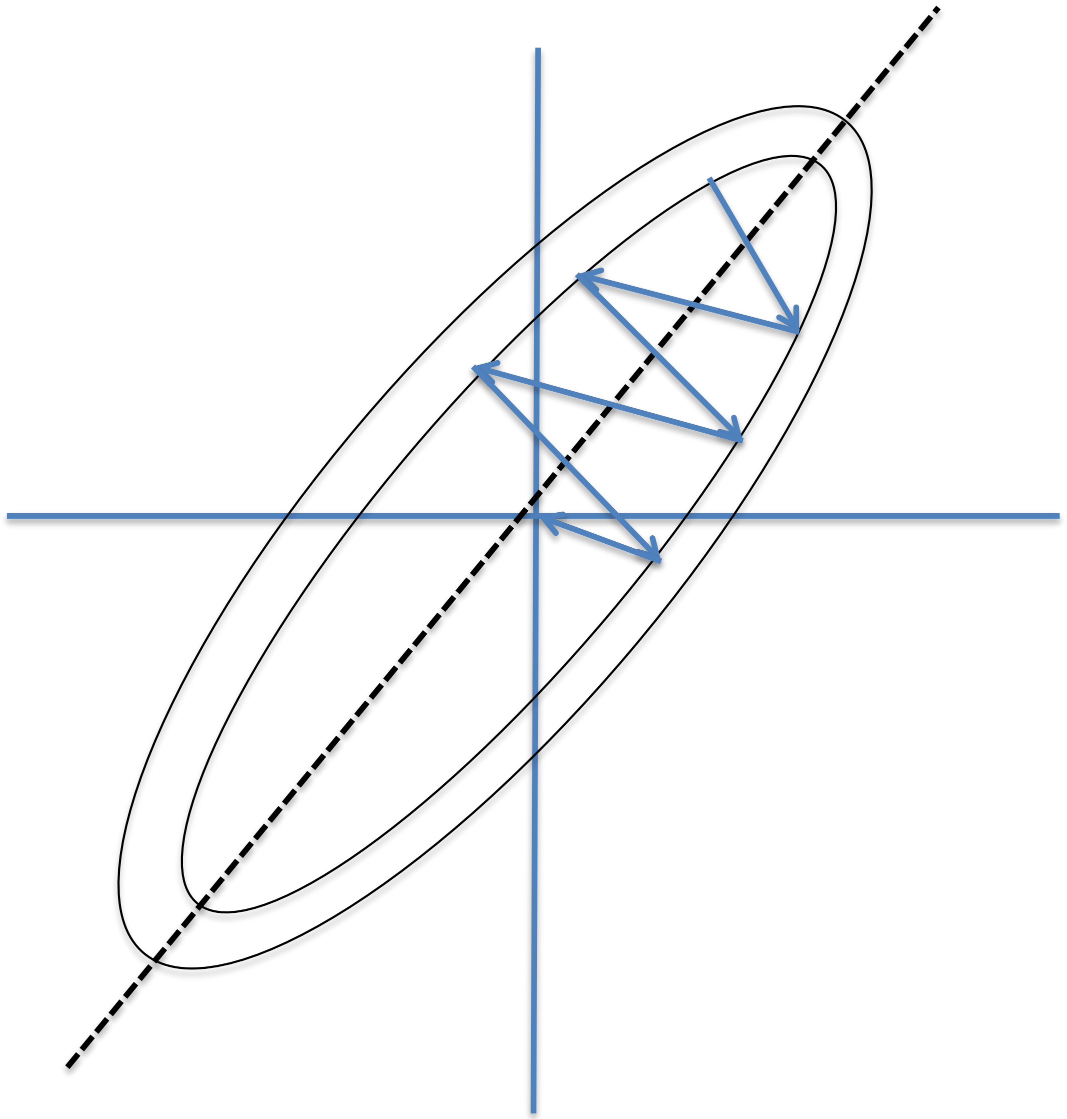
Strategy: Employ iteration

Note: Stochastic gradient descent employed in machine learning including artificial neural nets.

Optimization

Method 1: Gradient descent

Potential issue:



Newton's Method

Problem 1: minimize $f(x)$, $f : \mathbb{R}^n \to \mathbb{R}$

Problem 2: solve $F(x) = 0$ where $F : \mathbb{R}^n \to \mathbb{R}^n$

Note:

- If x^* solves (1), it also solves (2) with $F(x) = \nabla f(x)$
- If x^* solves (2), it solves (1) with $f(x) = \|F(x)\|^2 = F(x)^T F(x)$

Newton's Method (n=1): Let x_j approximate the root p with $F'(x_j) \neq 0$. Then

$$F(x) = F(x_j) + F'(x_j)(x - x_j) + \frac{(x - x_j)^2}{2} F''(\xi)$$

$$\Rightarrow 0 \approx F(x_j) + F'(x_j)(p - x_j)$$

$$\Rightarrow p \approx x_j - \frac{F(x_j)}{F'(x_j)}$$

Iteration: $x_{j+1} = x_j - \frac{F(x_j)}{F'(x_j)}$

Note: Quadratic convergence if function is sufficiently smooth and 'reasonable' initial value

Newton's Method

Newton's Method (n>1): Consider $F(x) = \nabla f(x) = 0$

Iteration: $x_{j+1} = x_j + s_j$ where s_j solves

$$F(x_j) + F'(x_j)s_j = 0$$

$$\Rightarrow x_{j+1} = x_j - H(x_j)^{-1}\nabla f(x_j)$$

Hessian:

$$F'(x) = H(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}$$

Note: Hessian computation is expensive so several techniques to approximate its action; e.g., Limited-memory Broyden –Fletcher-Goldfarb-Shanno (L-BFGS) employed in machine learning.

MATLAB Optimization Routines

Note: There is significant documentation for the Optimization Toolbox

Minimization:

- fmincon: Constrained nonlinear minimization
- fminsearch: Unconstrained nonlinear minimization (Nelder-Mead)
- fminunc: Unconstrained nonlinear minimization (gradient-based trust region)
- quadprog: Quadratic programming

Equation Solving:

- fsolve: Nonlinear equation solving
- fzero: scalar nonlinear equation solving

Least Squares:

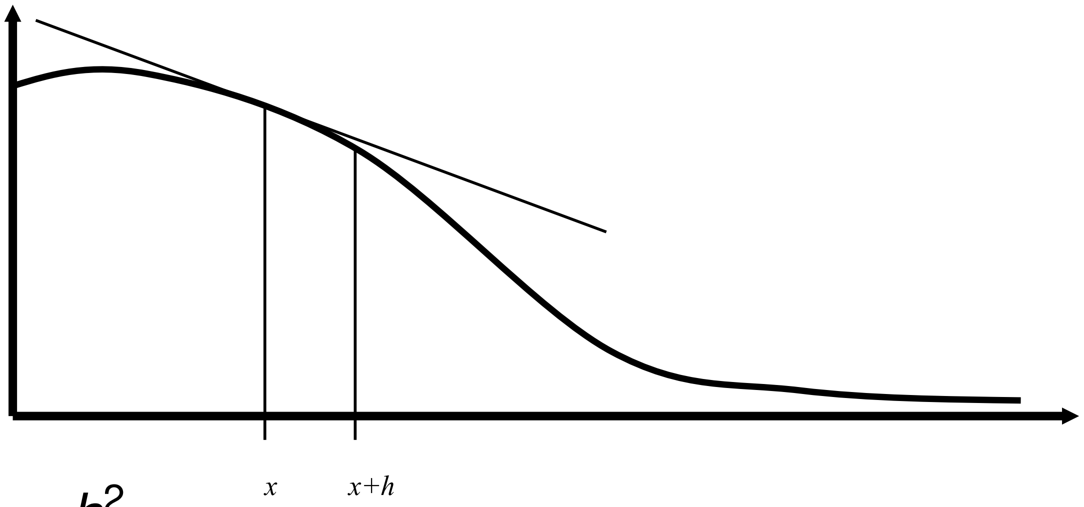
- lsqlin: Constrained linear least squares
- lsqnonlin: Nonlinear least squares
- lsqnonneg: Nonnegative linear least squares

Kelley's Routines: Available at the webpage http://www4.ncsu.edu/~ctk/

Numerical Differentiation

Derivative:

$$f'(x) = \lim_{h\to 0} \frac{(x+h)-f(x)}{h}$$

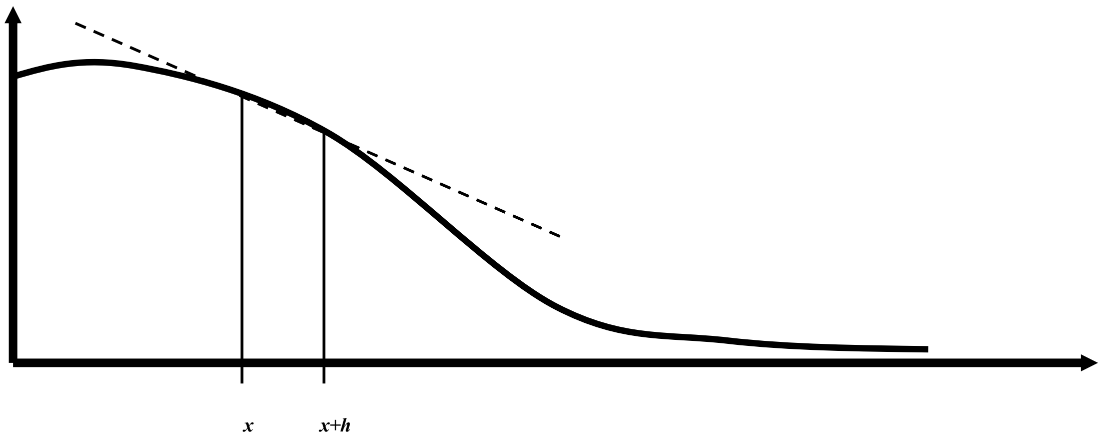


Note:

$$f(x+h) = f(x) + f'(x)h + f''(\xi)\frac{h^2}{2!}$$

$$\Rightarrow f'(x) = \tfrac{f(x+h)-f(x)}{h} - f''(\xi)\tfrac{h}{2!}$$

Forward Difference: $f'(x) = \dfrac{f(x+h)-f(x)}{h} + \mathcal{O}(h)$



Numerical Differentiation

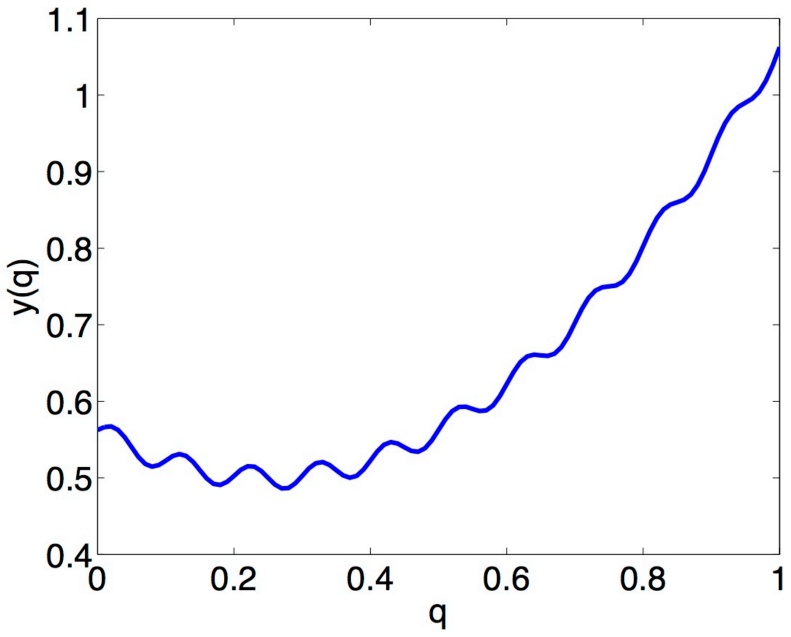
More Accuracy: Central differences

$$f'(x) = \frac{f(x+h) - f(x-h)}{h} + \mathcal{O}(h^2)$$

Numerical Differentiation

Issue: Suppose we have the following "noisy" function or data

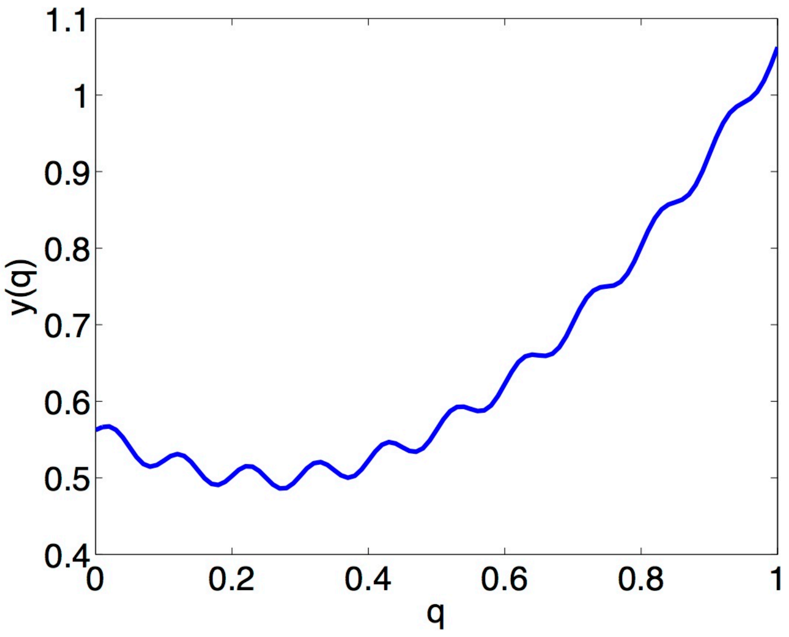
- What is the issue with doing finite-differences to approximate derivative?



Numerical Differentiation

Issue: Suppose we have the following “noisy” function or data

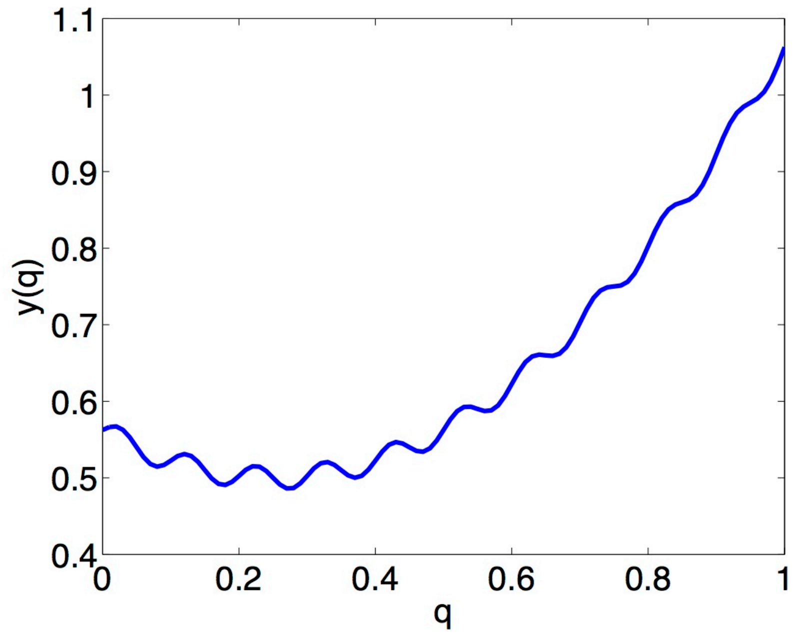
- What is the issue with doing finite-differences to approximate derivative?
- Derivatives can grow unboundedly due to noise.



Numerical Differentiation

Issue: Suppose we have the following "noisy" function or data

- What is the issue with doing finite-differences to approximate derivative?
- Derivatives can grow unboundedly due to noise.



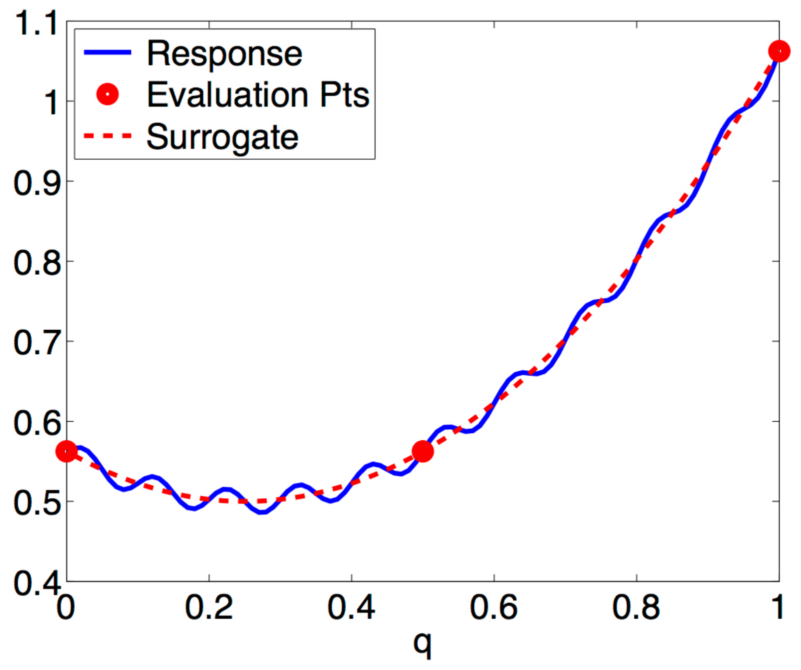
Example: Quadratic polynomial

$$y_s(q) = (q - 0.25)^2 + 0.5$$

Note: Solve linear system

Solution:

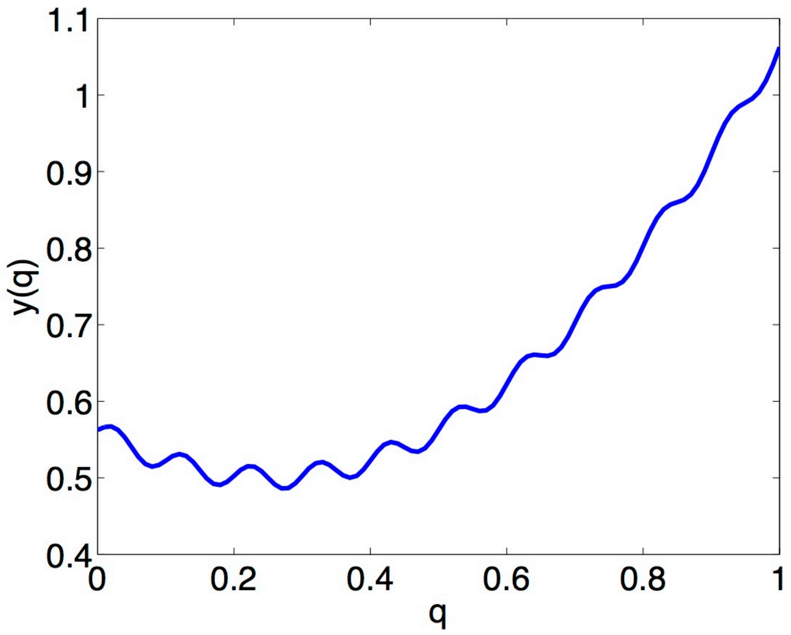
- Fit "smooth" function that is easy to differentiate.
- Interpolation



Numerical Differentiation

Issue: Suppose we have the following “noisy” function or data

- What is the issue with doing finite-differences to approximate derivative?
- Derivatives can grow unboundedly due to noise.

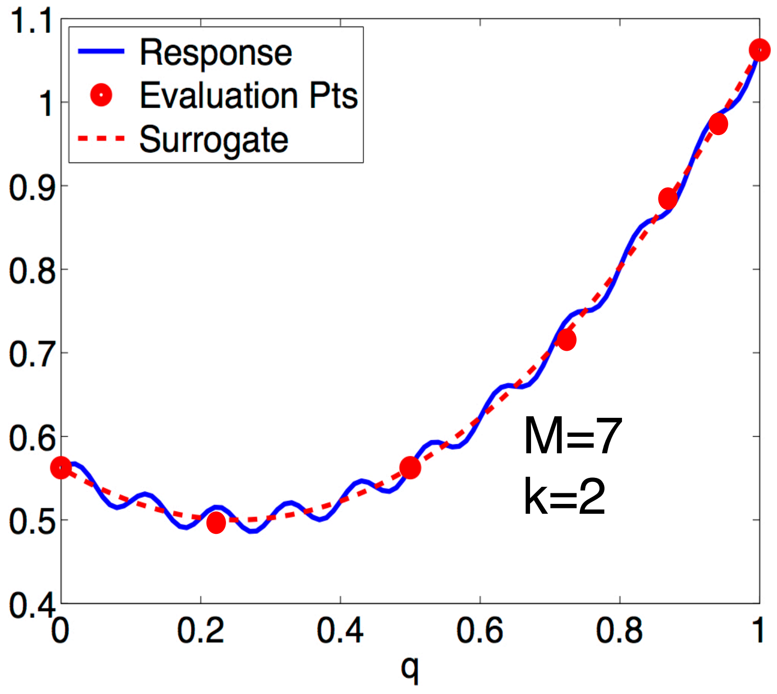


Example: Quadratic polynomial

$$y_s(q) = (q - 0.25)^2 + 0.5$$

Solution:

- Fit "smooth” function that is easy to differentiate.
- Regression



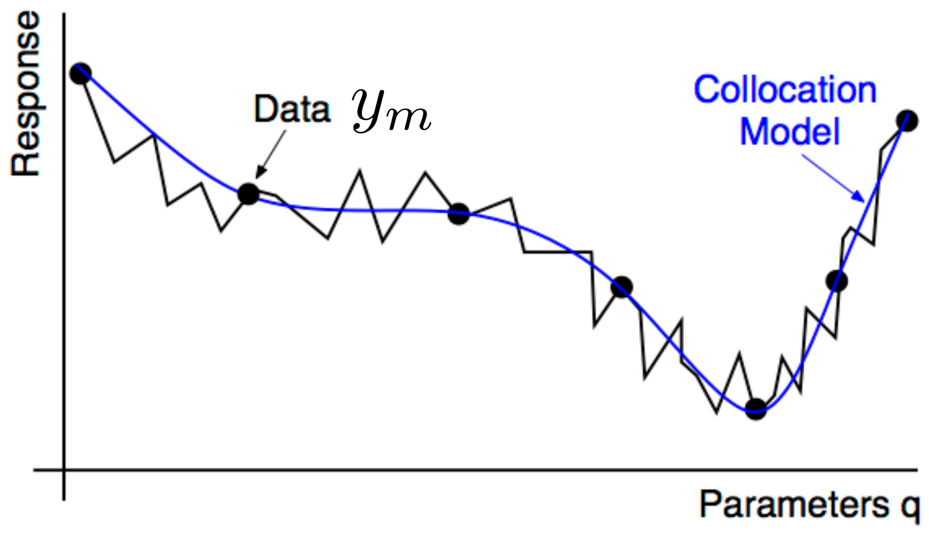
Lagrange Polynomials

Strategy: Consider high fidelity model

$$y = f(q)$$

with M model evaluations

$$y_m = f(q^m)\ ,\ m = 1, \ldots, M$$



Lagrange Polynomials:

$$Y^M(q) = \sum_{m=1}^{M} y_m L_m(q)$$

where $L_m(q)$ is a Lagrange polynomial, which in 1-D, is represented by

$$L_m(q) = \prod_{\substack{j=0 \\ j \neq m}}^{M} \frac{q - q^j}{q^m - q^j} = \frac{(q - q^1)\cdots(q - q^{m-1})(q - q^{m+1})\cdots(q - q^M)}{(q^m - q^1)\cdots(q^m - q^{m-1})(q^m - q^{m+1})\cdots(q^m - q^M)}$$

Note:

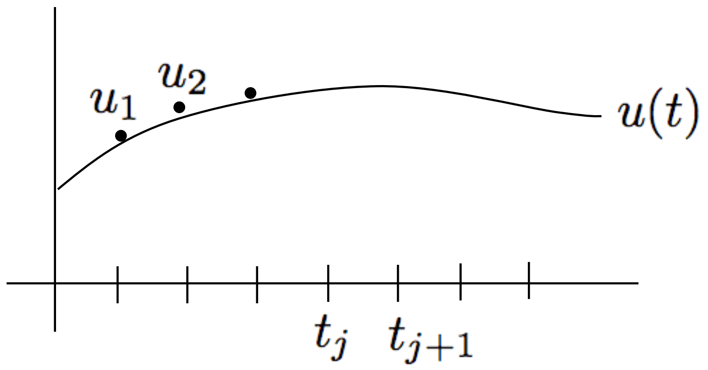
$$L_m(q^j) = \delta_{jm} = \begin{cases} 0 & , \quad j \neq m \\ 1 & , \quad j = m \end{cases}$$

Result: $Y^M(q^m) = y_m$

Numerical Methods for IVP: Euler's Method

Initial Value Problem:

$$\frac{du}{dt} = f(t,u)\ ,\ t \geq 0$$
$$u(0) = u_0$$



Notation: $t_j = jk$ for $j = 0, 1, \cdots$

$u_j \approx u(t_j)$

Taylor Series:

$$u(t_{j+1}) = u(t_j) + k\frac{du}{dt}(t_j) + \frac{k^2}{2}\frac{d^2u}{dt^2}(t_j) + \cdots + \frac{k^n}{n!}\frac{d^nu}{dt^n}(t_j) + \frac{k^{n+1}}{(n+1)!}\frac{d^{n+1}u}{dt^{n+1}}(\xi)$$

Euler's Method:

$$u(t_{j+1}) = u(t_j) + k\dot{u}(t_j) + \mathcal{O}(k^2)$$
$$\Rightarrow u_{j+1} = u_j + kf(t_j, u_j)$$

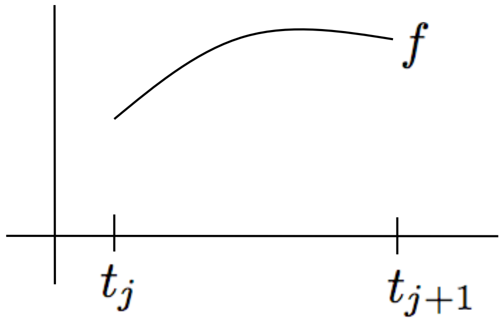
Accuracy: Local truncation error $\mathcal{O}(k^2)$

Global truncation error $\mathcal{O}(k)$

Euler and Implicit Euler Methods

Note:

$$u(t_{j+1}) = u(t_j) + \int_{t_j}^{t_{j+1}} f(t, u(t))dt$$



Euler's Method: Left Endpoint

$$u_{j+1} = u_j + kf(t_j, u_j)$$

Implicit Euler: Right Endpoint

$$u_{j+1} = u_j + kf(t_{j+1}, u_{j+1})$$

Stability: Apply method to

$$\dot{u}(t) = \lambda u \ , \ \ \lambda < 0$$

$$u(0) = u_0$$

Forward Euler

$$\begin{aligned} u_{j+1} &= u_j + \lambda k u_j \\ &= (1 + \lambda k)^{j+1} u_0 \end{aligned}$$

$$\Rightarrow |1 + \lambda k| < 1$$

$$\Rightarrow k < \frac{-2}{\lambda}$$

Implicit Euler

$$\begin{aligned} u_{j+1} &= u_j + \lambda u_{j+1} \\ &= \left(\frac{1}{1 - \lambda k}\right)^{j+1} u_0 \end{aligned}$$

$$\Rightarrow 1 < |1 - \lambda k|$$

$$\Rightarrow k > 0$$

Runge-Kutta-Feylberg Methods

4th Order Runge-Kutta:

$$k_1 = kf(t_j, u_j)$$

$$k_2 = kf\left(t_j + \frac{k}{2}, u_j + \frac{k_1}{2}\right)$$

$$k_3 = kf\left(t_j + \frac{k}{2}, u_j + \frac{k_2}{2}\right)$$

$$k_4 = kf(t_{j+1}, u_j + k_3)$$

$$u_{j+1} = u_j + \frac{1}{6}(k_1 + k_2 + k_3 + k_4)$$

Accuracy: Local Truncation error is 4th-order if u(t) has five continuous derivatives.

Runge-Kutta-Feylberg: Use R-K method with 5th order truncation error to estimate local error in 4th order R-K method to choose appropriate stepsize.

MATLAB ODE Routines

Algorithms: From the MATLAB ODE documentation

- **ode45** is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver - in computing y(tn), it needs only the solution at the immediately preceding time point, y(tn-1). In general, ode45 is the best function to apply as a "first try" for most problems.

- **ode23** is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of moderate stiffness. Like ode45, ode23 is a one-step solver.

- **ode113** is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances and when the ODE file function is particularly expensive to evaluate. ode113 is a multistep solver - it normally needs the solutions at several preceding time points to compute the current solution.

- The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

- **ode15s** is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like ode113, ode15s is a multistep solver. Try ode15s when ode45 fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem.

- **ode23s** is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective.

- **ode23t** is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. ode23t can solve DAEs.

- **ode23tb** is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like ode23s, this solver may be more efficient than ode15s at crude tolerances.

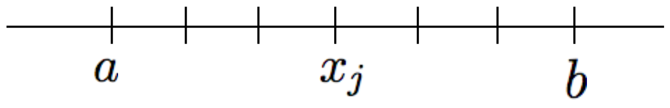
MATLAB ODE Routines: From the Documentation

Solver	Problem Type	Order of Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. This should be the first solver you try.
ode23	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
ode113	Nonstiff	Low to High	For problems with stringent error tolerances or for solving computationally intensive problems.
ode15s	Stiff	Low to Medium	If ode45 is slow because the problem is stiff
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
ode23t	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

Numerical Methods for BVP: Finite Differences

Problem:

$$y'' = p(x)y' + q(x)y + f(x)\ ,\ a \le x \le b$$
$$y(a) = \alpha\ ,\ y(b) = \beta$$



Grid: $x_j = a + jh\ ,\ h = (b-a)/(N+1)$ Note: N interior grid points

Centered Difference Formulas: (From Taylor expansions)

$$y''(x_j) = \frac{1}{h^2}\left[y(x_{j+1}) - 2y(x_j) + y(x_{j-1})\right] - \frac{h^2}{24}y^{(4)}(\xi_j)$$
$$y'(x_j) = \frac{1}{2h}\left[y(x_{j+1}) - y(x_{j-1})\right] - \frac{h^2}{6}y'''(\eta_j)$$

System:

$$\frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2} = p(x_j)\left[\frac{y(x_{j+1}) - y(x_{j-1})}{2h}\right] + q(x_j)y(x_j)$$
$$+ f(x_j) - \frac{h^2}{12}\left[2p(x_j)y'''(\eta_j) - y^{(4)}(\xi_j)\right]$$

Finite Difference Method for BVP

Finite Difference System: Define $y_0 = \alpha$, $y_{N+1} = \beta$ and consider

$$\left(\frac{2y_j - y_{j+1} - y_{j-1}}{h^2}\right) + p(x_j)\left(\frac{y_{j+1} - y_{j-1}}{2h}\right) + q(x_j)y_j = -f(x_j)$$

$$\Rightarrow -\left(1 + \frac{h}{2}p(x_j)\right) y_{j-1} + (2 + h^2 q(x_j))y_j - \left(1 - \frac{h}{2}p(x_j)\right) y_{j+1} = -h^2 f(x_j)$$

for $j = 1, 2, \cdots, N$

Matrix System:

$$\begin{bmatrix} 2 + h^2 q(x_1) & -1 + \frac{h}{2}p(x_1) & 0 & \\ -1 - \frac{h}{2}p(x_2) & 2 + h^2 q(x_2) & -1 + \frac{h}{2}p(x_2) & \\ & \ddots & \ddots & \ddots \\ & & & -1 + \frac{h}{2}p(x_{N-1}) \\ & 0 & -1 - \frac{h}{2}p(x_N) & 2 + h^2 q(x_N) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} -h^2 f(x_1) + \left(1 + \frac{h}{2}p(x_1)\right)\alpha \\ -h^2 f(x_2) \\ \vdots \\ -h^2 f(x_{N-1}) \\ -h^2 f(x_N) + \left(1 - \frac{h}{2}p(x_N)\right)\beta \end{bmatrix}$$