Computing for Economists

Numerical Differentiation & Integration

Toren Fronsdal and Ruby Zhang

Harvard University

Table of Contents

Introduction

Motivation

Numerical differentiation and integration are key steps in a number of estimation exercises in economics. For example:

- Optimization and non-linear equation solving (gradients, Hessians, Jacobians)
- Computation of expectations
- Computation of consumer surplus

Resources

These slides are primarily based on **Judd (1998)**, Chapters 7-9, supplemented with lecture notes by **Jesús Fernández-Villaverde** and **Pablo Guerrón** and by **Shunsuke Tsuda**.

Numerical Methods

Differentiation:

1. Finite Differences
2. Symbolic Differentiation
3. Automatic Differentiation

Integration:

1. Quadrature Methods
2. Monte Carlo
3. Quasi-Monte Carlo

Numerical Differentiation

Forward Difference

The derivative of a function $f(\cdot)$ is

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

Thus, a natural approximation is "forward differencing" scheme:

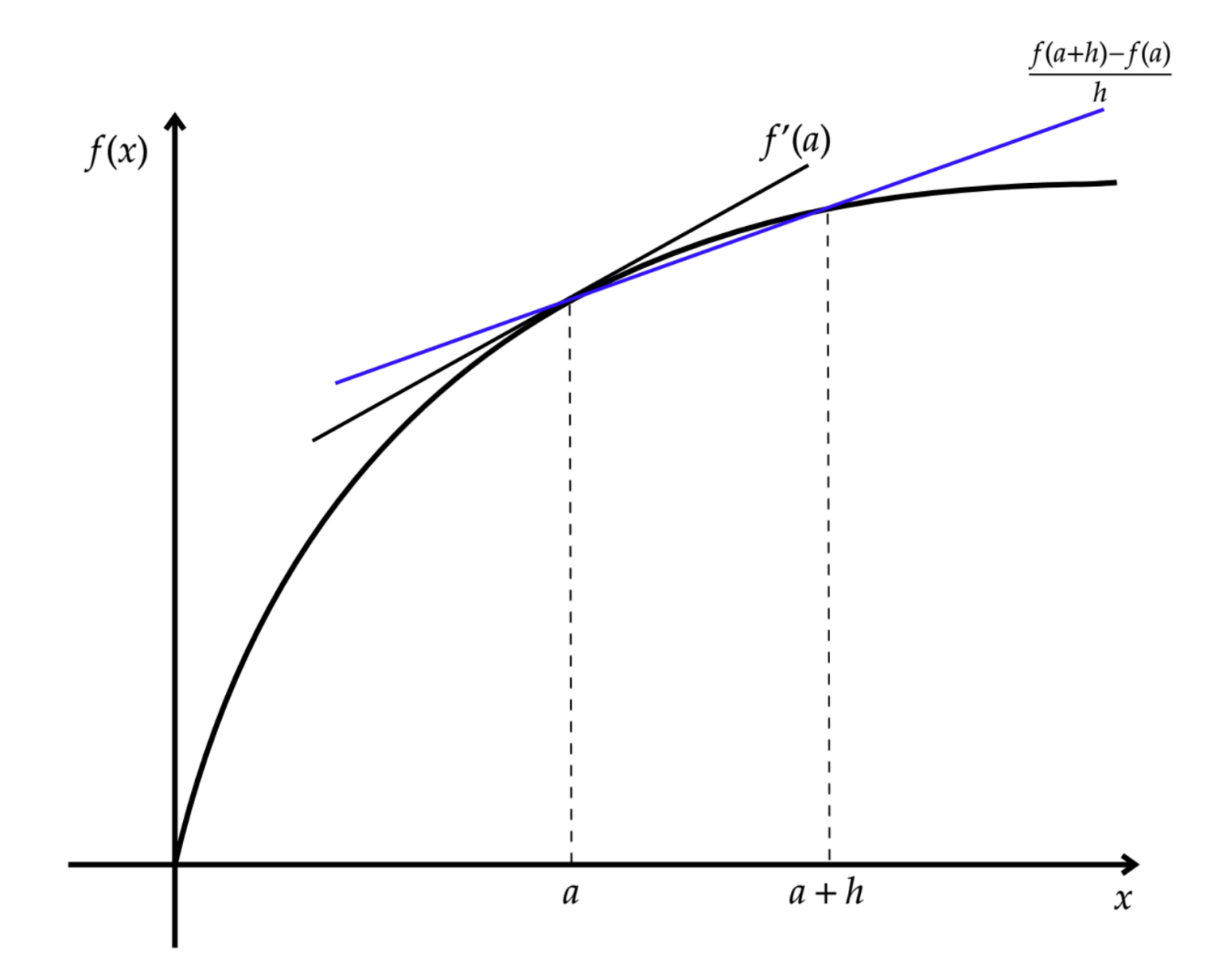
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

for a small $h > 0$.

Extension to multivariate functions is straightforward:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i + h_i, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Forward Difference



Round-Off and Truncation Errors

Round-off errors occur due to the finite precision of a computer.

Truncation errors arise from the use of an approximation in place of an exact mathematical procedure.

Assuming f is twice differentiable, we can use the Taylor expansion

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi), \quad \xi \in (x, x+h).$$

Rearranging gives

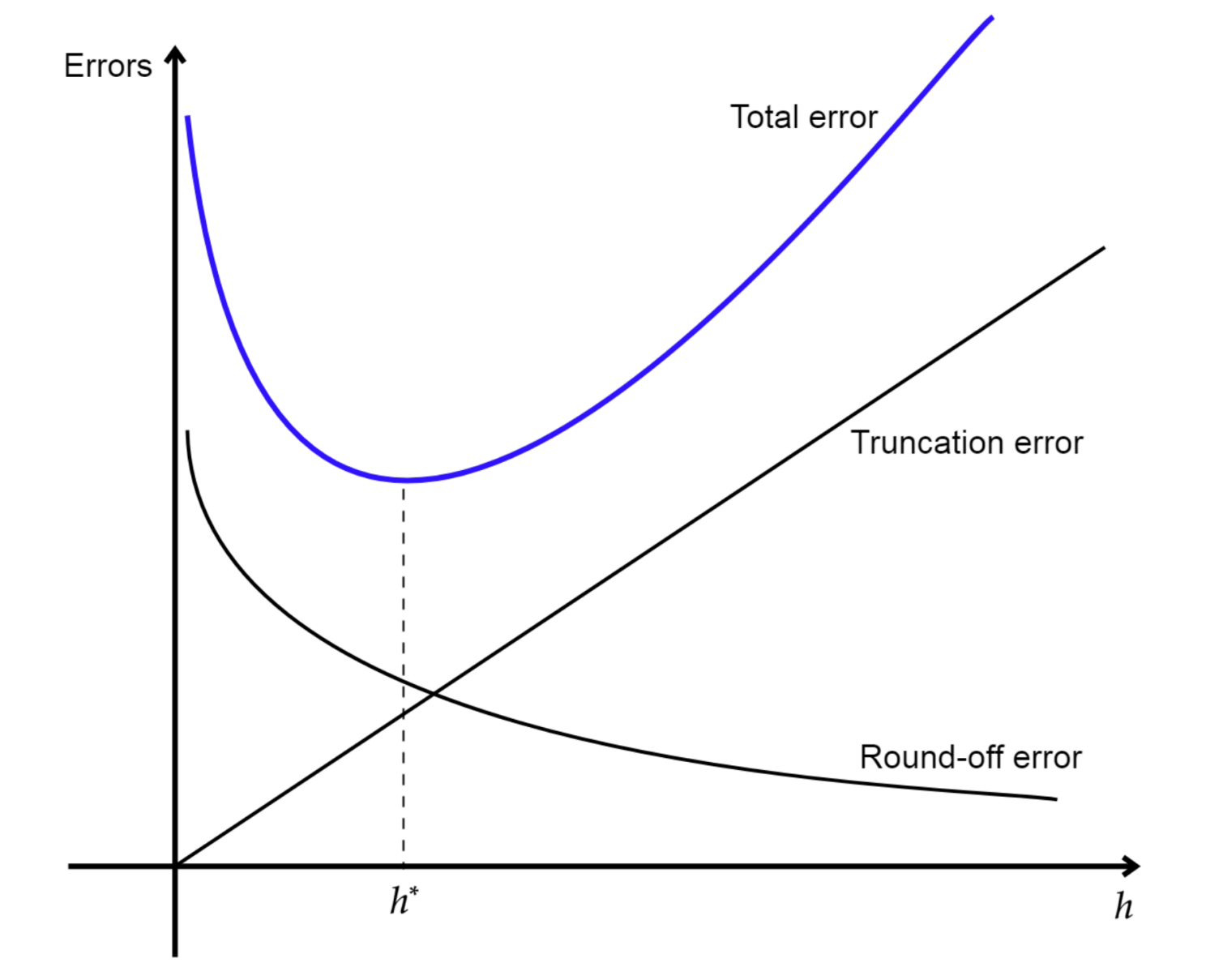
$$\frac{f(x+h) - f(x)}{h} - f'(x) = h\frac{f''(\xi)}{2}.$$

So the truncation error is bounded by

$$\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \leq \frac{hM}{2}, \quad M \equiv \max_{\xi \in [x, x+h]} |f''(\xi)|$$

$\Rightarrow$ error on the first order ($\mathcal{O}(h)$)

Round-Off and Truncation Errors



How to choose h?

Minimize the sum of round-off and truncation errors.

If ε is the machine precision and $\hat{f}(x)$ is the computed value of $f(x)$, then the round-off error is bounded by

$$\left| \frac{f(x+h) - f(x)}{h} - \frac{\hat{f}(x+h) - \hat{f}(x)}{h} \right| \leq \frac{2\epsilon L_f}{h}, \quad \text{where } L_f \equiv \max_{\xi \in [x, x+h]} |f(\xi)|.$$

Optimal h: $h^* = \arg\min_h \frac{2\epsilon L_f}{h} + \frac{hM}{2} = 2\sqrt{\frac{\epsilon L_f}{M}}$.

Rule of thumb: $h = \max(|x|, 1)\sqrt{\varepsilon}$.

Backward Difference

"Backward differencing" scheme:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Extension to multivariate functions is:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i, \ldots, x_n) - f(x_1, \ldots, x_i - h_i, \ldots, x_n)}{h}$$

Also, an approximation error of order h.

Thus, in practice, one uses forward or backward differences depending on whether we care more about left or right derivatives.

Central Difference

"Centered differencing" scheme:

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

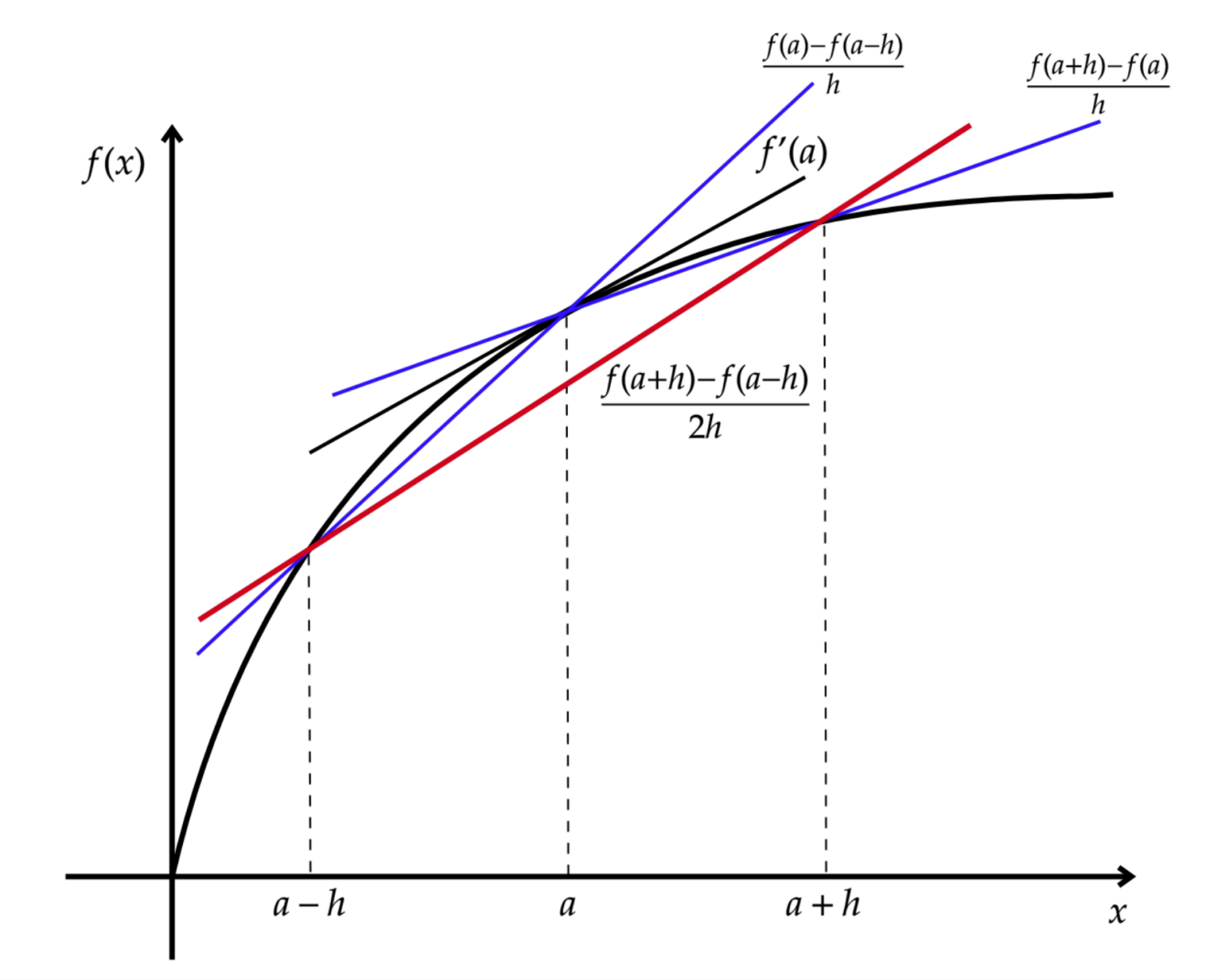
Rule for selecting h:

$$h = \max(|x|, 1)\sqrt[3]{\epsilon}$$

This choice minimizes the sum of round-off and truncation error.

`numpy.gradient()` uses second-order central differences in the interior points and either first or second-order forward or backward differences at the boundaries.

Central Difference



Central Difference

Truncation error can be found by subtracting Taylor expansions of $f(x+h)$ and $f(x-h)$:

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{\left(f'''(\xi_1) + f'''(\xi_2)\right)}{6}h^3 \Rightarrow$$
$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}\left(h^2\right)$$

Centered differencing is more precise: truncation error of order h^2.

Trade-off between accuracy and efficiency for functions of higher dimensions.

Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$, then computing the Jacobian matrix of f requires $m(n+1)$ function evaluations using the forward differencing scheme and $2mn$ evaluations using the centered differencing scheme, roughly twice as many when n is large.

Richardson Extrapolation

Recall that centered differencing generates errors of order $\mathcal{O}\left(h^2\right)$.

We can increase the efficiency by using **Richardson extrapolation**.

Basic idea: Combine two approximations with $\mathcal{O}(h^2)$ to form an approximation with $\mathcal{O}(h^4)$.

If $f(x-2h)$ and $f(x+2h)$ are known,

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + \mathcal{O}\left(h^4\right),$$

a fourth-order approximation.

Symbolic Differentiation

Symbolic differentiation manipulates mathematical expressions symbolically to compute derivatives.

- Provides exact derivatives as mathematical expressions
- Useful for simple functions but can become complex for intricate expressions
- Can result in complex and redundant expressions

Example: WolframAlpha

SymPy is a Python library for symbolic mathematics.

Automatic Differentiation

Automatic differentiation divides the function to derivate into small parts and then applies the chain rule to solve for the derivative. As a result, no truncation error.

Efficient, accurate, and versatile. Ideal for complex models.

- **JAX** is a Python package to automatically differentiate Python and NumPy code.
- *`torch.autograd`* is PyTorch's automatic differentiation engine that powers neural network training.
  - Backpropagation is the special case of autodiff applied to neural nets

Not covered in this course; see **Bayden et al. (2018)** or **Griewank and Walther (2003)** for an overview.

Numerical Integration

Motivating Example

Numerical evaluation of a definite integral is a frequent problem encountered in economic modeling.

For example, the BLP model (and mixed logit) depends on an equation equating predicted and observed market shares

$$s_{jt}\left(\delta_{jt};\theta_2\right) = \int \frac{\exp\left(\delta_{jt}+\mu_{jt}(\nu)\right)}{1+\sum_{k\in J}\exp\left(\delta_{kt}+\mu_{kt}(\nu)\right)} dF(\nu)$$

- Accuracy $\Rightarrow$ correct point estimates
- Efficiently $\Rightarrow$ complete calculations (Nested Fixed Point Algorithm)

Numerical Integration

Most rules approximate a (multidimensional) integral

$$I_f \equiv \int_\Omega f(x)w(x)dx, \quad \Omega \subset \mathbb{R}^d, w(x) \geq 0 \quad \forall x \in \Omega$$

as

$$\hat{I}_f \equiv \sum_{j=1}^{R} w_j f\left(y_j\right), y_j \in \Omega$$

- The crucial issue is how to choose the weights and nodes, $\left\{w_j, y_j\right\}$
- Ideally, a rule should have $\lim_{R\to\infty} \hat{I}_f = I_f$, i.e. converge to the truth

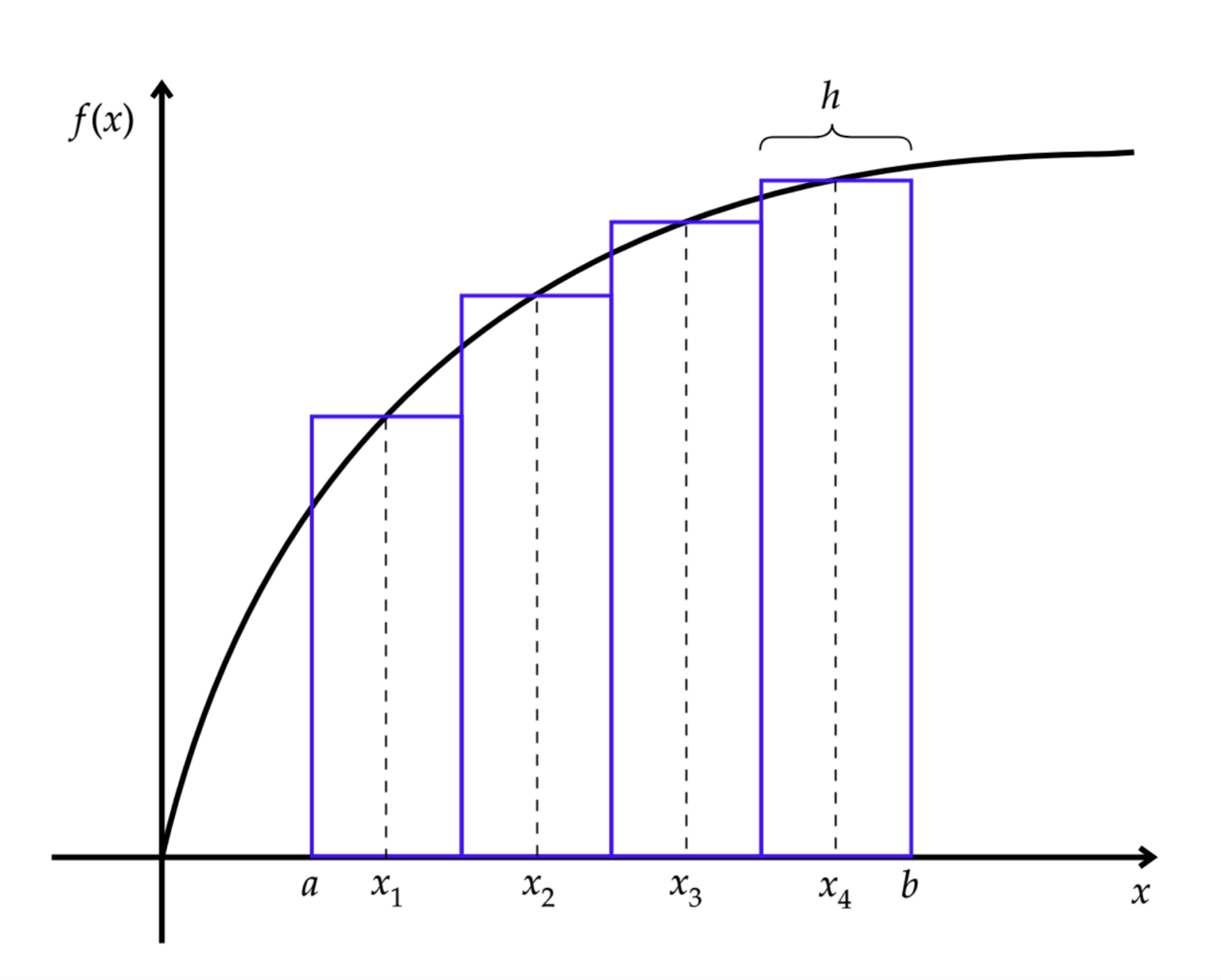
Numerical Integration

Newton–Cotes formulas: methods that are based on equally spaced nodes.

- Midpoint rule
- Trapezoidal rule
- Simpson's rule

Gaussian quadrature formulas: methods that are based on nodes that are *not* equally spaced.

Midpoint Rule



Midpoint Rule

Approximates the area using rectangles at midpoints.

Assume that $f(x)$ is continuous on $[a, b]$. Let n be a positive integer and $h = \frac{b-a}{n}$. If $[a, b]$ is divided into n subintervals, each of length h, and x_i is the midpoint of the i^{th} subinterval, set

$$M_n = h \sum_{i=1}^{n} f(x_i)$$

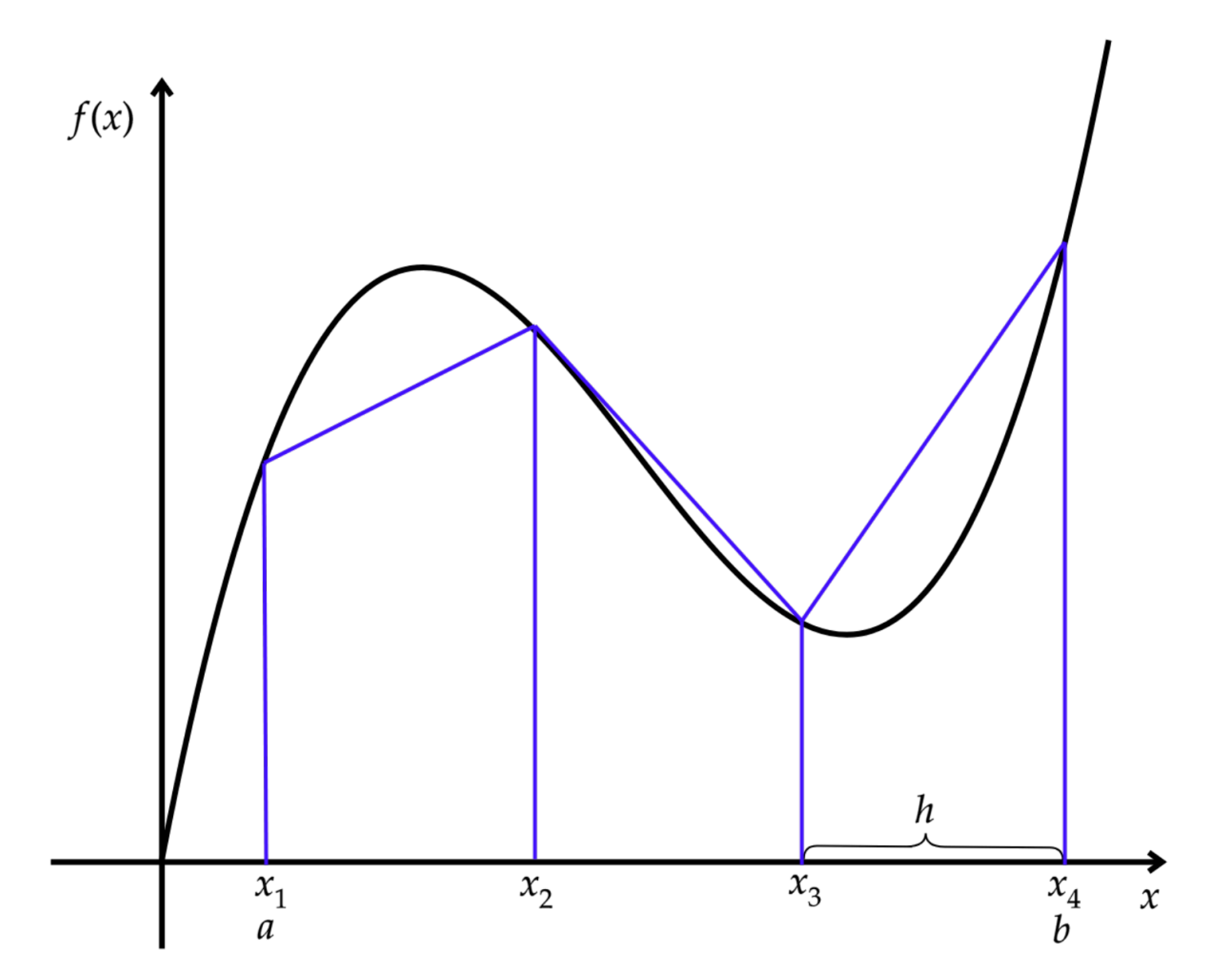
Then $\lim_{n\to\infty} M_n = \int_a^b f(x)dx$.

$$\int_a^b f(x)dx = M_n + \frac{h^2(b-a)}{24} f''(\xi), \qquad \text{for some } \xi \in [a, b]$$

so error on the order of $\mathcal{O}(h^2)$.

The choice of h is crucial: adaptive quadrature iterates on n until M_n stops changing.

Trapezoidal Rule



Trapezoidal Rule

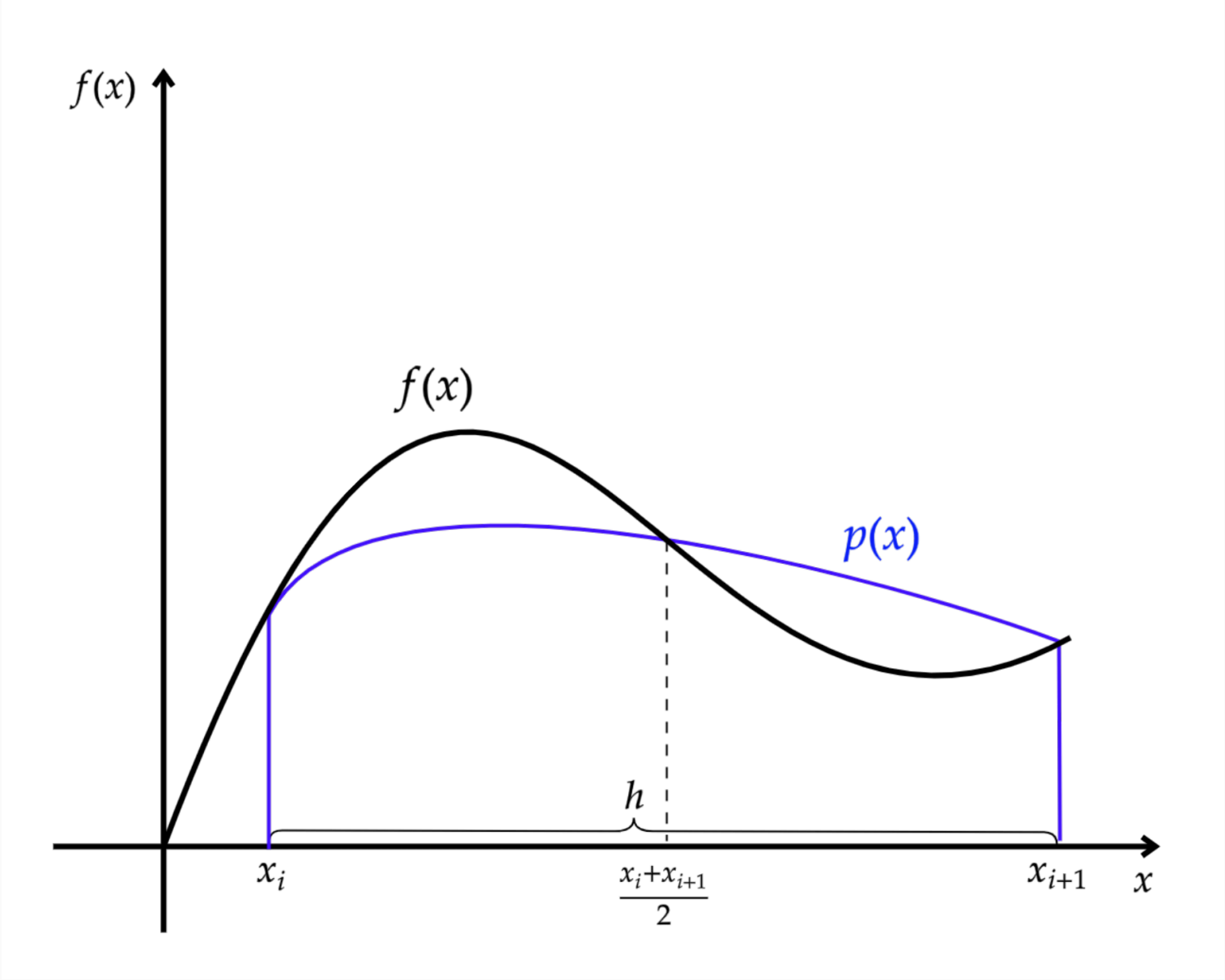
Approximates the area using trapezoids.

$$
\begin{aligned}
T_n &= h \sum_{i=1}^{n} \frac{f(x_i) + f(x_{i+1})}{2} \\
&= h \left(\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(x_{i+1}) \right)
\end{aligned}
$$

Then, $\lim_{n \to \infty} T_n = \int_a^b f(x)dx$.

The error is $\frac{h^2(b-a)}{12} f''(\xi)$, so like the midpoint rule, the error is also proportional to h^2.

Simpson's Rule



Simpson's Rule

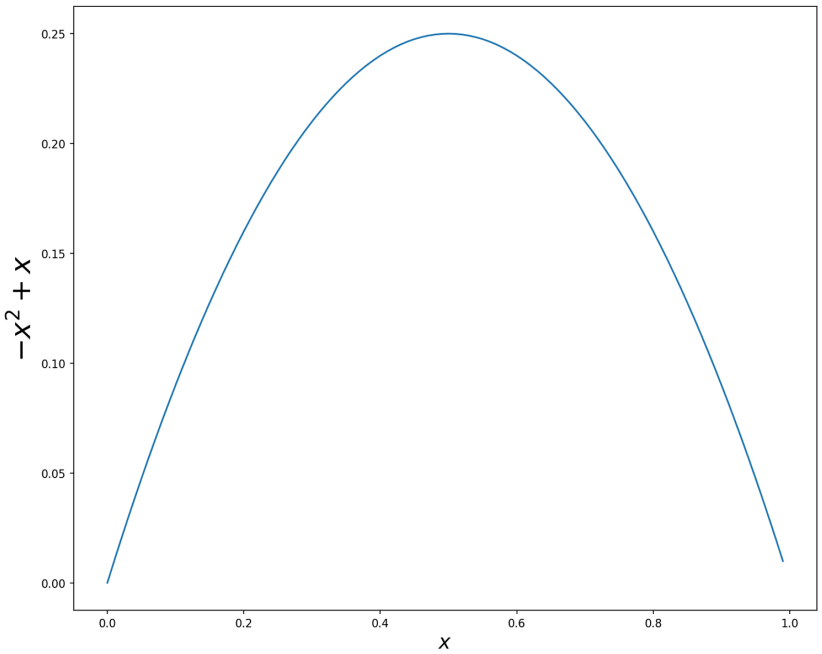
Approximates the area using piecewise quadratic functions.

$$S_n = \frac{h}{3}\left[f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 4f_{n-1} + f_n\right] - \frac{h^4(b-a)}{180} f^{(4)}(\xi)$$

Essentially takes three consecutive x_i nodes, uses an interpolating quadratic function to approximate f locally, and integrates the interpolating quadratic to approximate the integral over that interval.

Error is $\mathcal{O}(h^4)$.

Numerical Integration Rules for Simple Integrals



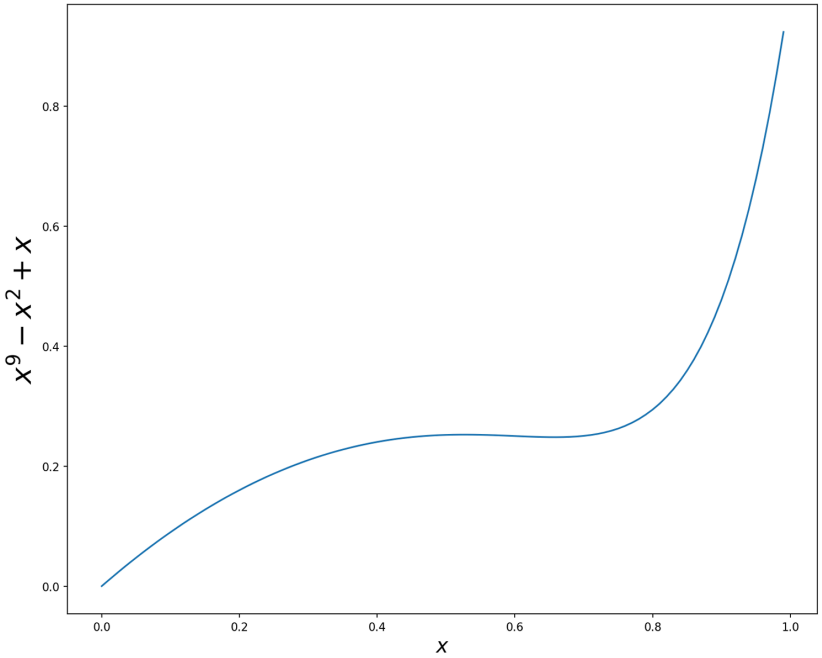


Table 1: Minimum nodes to achieve approximation error $< 10^{-6}$

Object	$\int_0^1 \left(-x^2 + x\right) dx$			$\int_0^1 \left(x^9 - x^2 + x\right) dx$		
Method	Midpoint	Trapezoid	Simpson's	Midpoint	Trapezoid	Simpson's
n	289	410	2	541	764	42
Computation time (sec.)	0.000271	6.13 e-05	0.000122	0.000684	0.000138	0.00012

Numerical Integration Rules for Simple Integrals

Rule	Number of points	$\int_0^1 x^{1/4} dx$	$\int_1^{10} x^{-2} dx$	$\int_0^1 e^x dx$	$\int_1^{-1} (x + 0.05)^+ dx$
Trapezoid	4	0.7212	1.7637	1.7342	0.6056
	7	0.7664	1.1922	1.7223	0.5583
	10	0.7797	1.0448	1.7200	0.5562
	13	0.7858	0.9857	1.7193	0.5542
Simpson	3	0.6496	1.3008	1.4662	0.4037
	7	0.7816	1.0017	1.7183	0.5426
	11	0.7524	0.9338	1.6232	0.4844
	15	0.7922	0.9169	1.7183	0.5528
Gauss-Legendre	4	0.8023	0.8563	1.7183	0.5713
	7	0.8006	0.8985	1.7183	0.5457
	10	0.8003	0.9000	1.7183	0.5538
	13	0.8001	0.9000	1.7183	0.5513
Truth		0.80000	0.90000	1.7183	0.55125

Gaussian Quadrature

Newton-Cotes methods are based on nodes $\{x_i\}_1^n \in [a, b]$ chosen arbitrarily. Gaussian quadrature provides a way to more efficiently set nodes and weights.

Gaussian quadrature is also more general than Newton-Cotes:

$$\int_a^b f(x)w(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i)$$

Gaussian quadrature sets the nodes and the weights in such a way that the approximation is exact when f is a low-order polynomial.

- Yields exact result for polynomials of degree $2n - 1$ or less

$$\int_a^b f(x)w(x)dx = \sum_{i=1}^{n} \omega_i f(x_i), \quad \forall f \in \mathcal{F}_{2n-1}$$

Gaussian Quadrature Formulas

Formula	Domain	Weight
Gauss-Legendre	$[-1, 1]$	$w(x) = 1$
Gauss-Chebyshev	$[-1, 1]$	$w(x) = \left(1 - x^2\right)^{-\frac{1}{2}}$
Gauss-Hermite	$[-\infty, \infty]$	$w(x) = e^{-x^2}$
Gauss-Laguerre	$[0, \infty]$	$w(x) = e^{-x}$

Can perform a change of variables to transform rules to other intervals.

Gauss-Hermite is useful when integrating normally distributed random variables.

Simple Integral Example

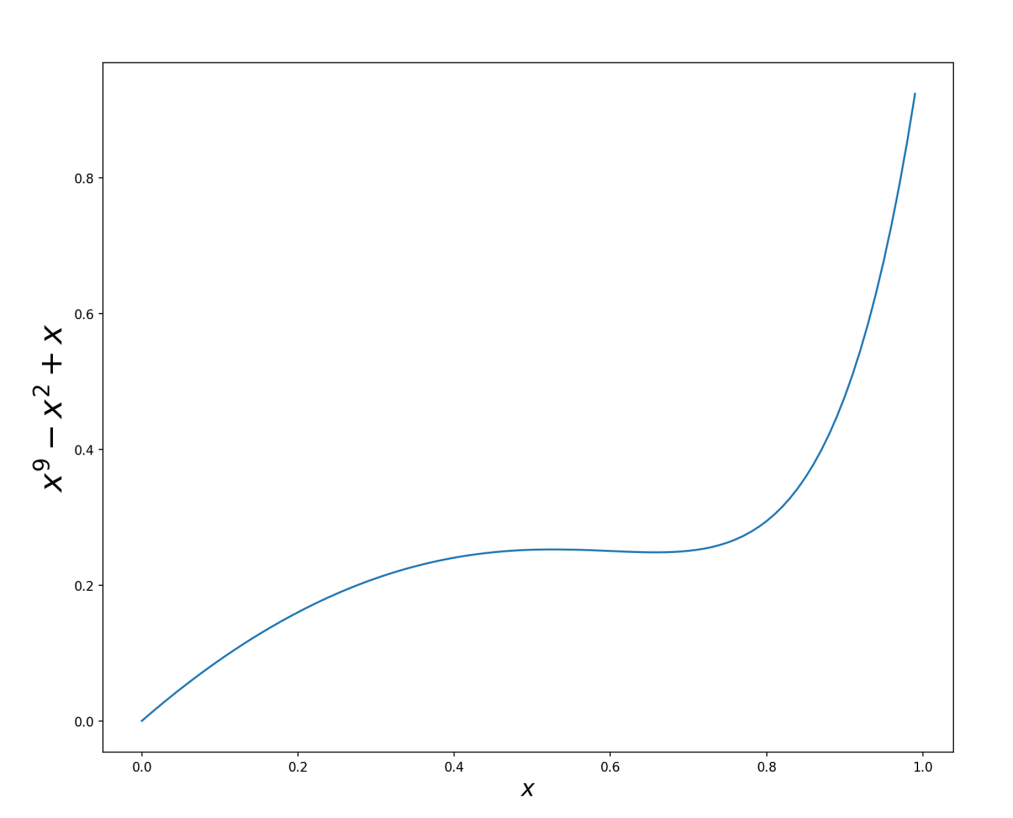


Table 2: Minimum nodes to achieve approximation error $< 10^{-6}$

Object	$\int_0^1 \left(x^9 - x^2 + x\right) dx$			
Method	Midpoint	Trapezoid	Simpson's	**Gaussian Quadrature**
n	541	764	42	**5**
Computation time (sec.)	0.000684	0.000138	0.00012	**0.000327**

Numerical Integration in Python

`scipy.integrate` is a sub-package that provides several numerical integration techniques.

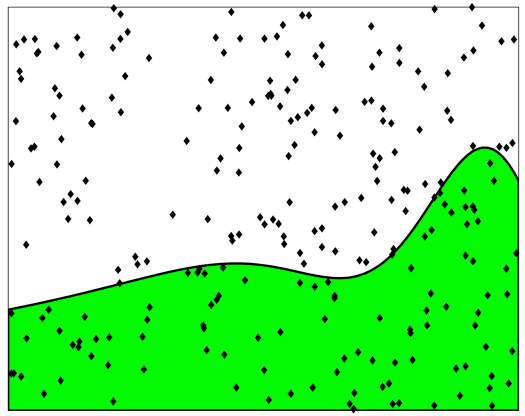
It contains the following functions:

- `trapezoid()`: composite trapezoidal rule
- `simpson()`: composite Simpson's rule
- `quad()`: adaptive quadrature using Fortran library QUADPACK

Monte Carlo Integration

Monte Carlo methods rest on the central limit theorem and the law of large numbers.

Since they are probabilistic in nature, they put structure on the error of approximation.



General approach:

1. Draw points at random
2. Approximate the integral as the total area times the fraction of points that fall under the curve $f(x)$

Crude Monte Carlo Integration

A crude Monte Carlo estimate of $\mathbb{E}[f(X)] = \int_0^1 f(x)dx$ is calculated by generating N draws from $U[0,1]$, $\{x_i\}_{i=1}^N$, and takes the form

$$\hat{I}_f = \frac{1}{N}\sum_{i=1}^{N} f(x_i)\,,$$

where $\hat{I}_f$ is also a random variable with variance

$$\sigma^2_{\hat{I}_f} = \frac{1}{N}\int_0^1 \left(f(x) - I_f\right)^2 dx = \frac{\sigma_f^2}{N}.$$

Since σ_f^2 is unknown, we can estimate of the variance of $f(X)$ as

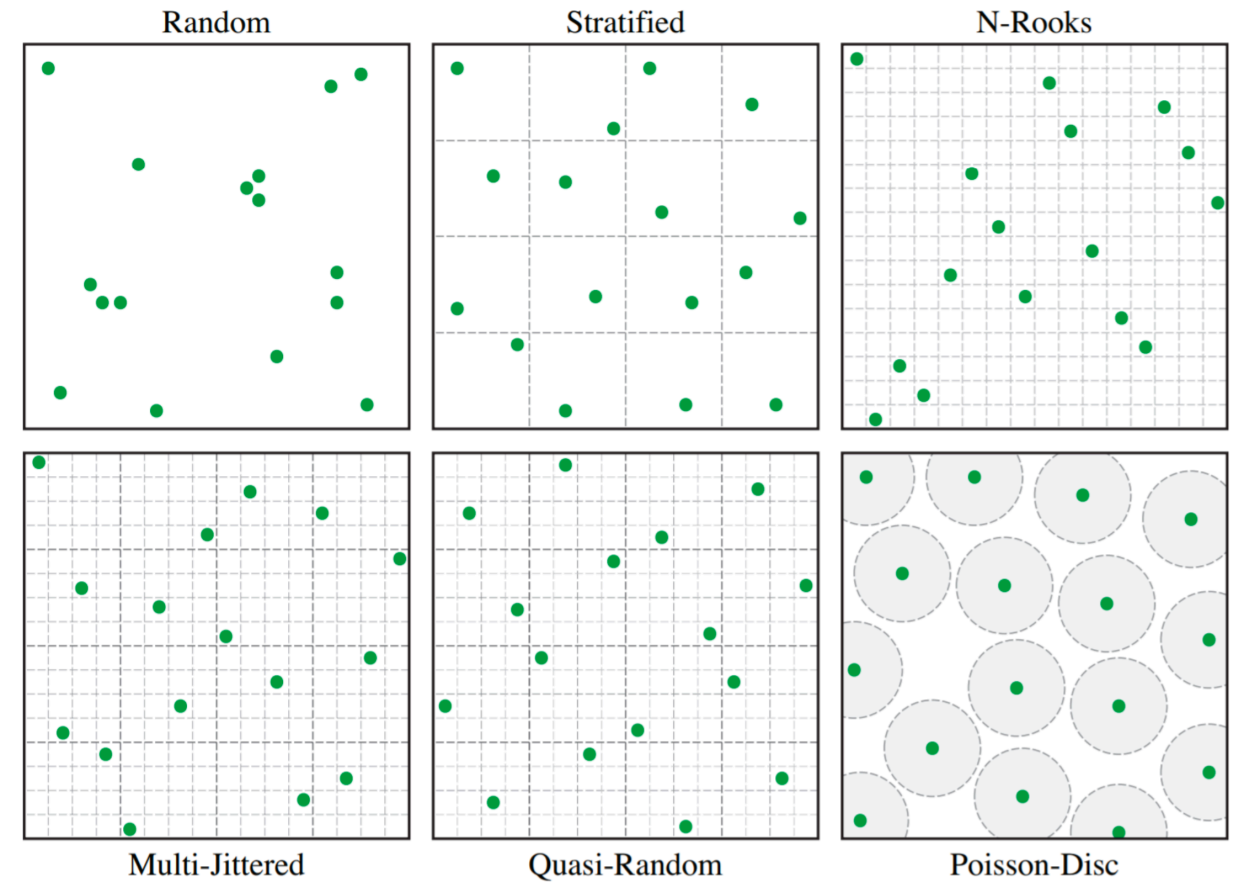
$$\hat{\sigma}_f^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(f(x_i) - \hat{I}_f\right)^2.$$

Although this estimator is unbiased, it is not commonly used, because of its large variance.

Techniques to Reduce Variance

There are a variety of simple techniques that can reduce the variance, but retain its unbiasedness.

- Stratified sampling
- Importance sampling
- Antithetic variates
- Control variates
- Quasi-Monte Carlo


Random
Stratified
N-Rooks
Multi-Jittered
Quasi-Random
Poisson-Disc

Stratified Sampling

Approach: split the sample space into distinct regions and randomly sample within regions.

For example, suppose we split $[0, 1]$ into the regions $[0, \alpha]$ and $[\alpha, 1]$. Then, sampling N points from both regions, we form the estimate

$$\hat{I}_f = \frac{\alpha}{N} \sum_i f(x_{1i}) + \left(\frac{1-\alpha}{N} \right) \sum_i f(x_{2i}) \,,$$

where $x_{1i} \in [0, \alpha]$ and $x_{2i} \in [\alpha, 1]$, $i = 1, \ldots, N$. Its variance is

$$\frac{\alpha}{N} \int_0^\alpha f^2 + \frac{(1-\alpha)}{N} \int_\alpha^1 f^2 - \frac{\alpha}{N} \left(\int_0^\alpha f \right)^2 - \frac{(1-\alpha)}{N} \left(\int_\alpha^1 f \right)^2 .$$

A good α equates the variance over $[0, \alpha]$ with that over $[\alpha, 1]$.

- Ensures draws don't "clump" in one region

Importance Sampling

Approach: sample more intensively where f is large, which is where $f(x)$ makes the greatest contribution to the integral.

The value of f in some parts of $[0, 1]$ is more "important" than other parts.

If x_i is drawn with density $p(x)$, then

$$\hat{I}_f^p = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

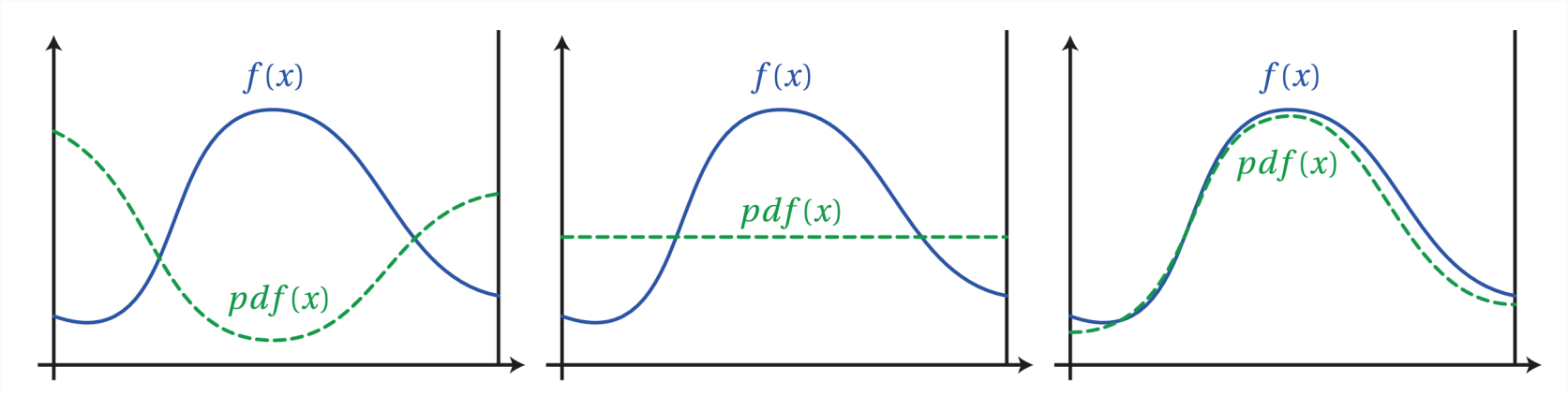
is an unbiased estimate of I_f, and its variance is

$$\sigma^2_{\hat{I}_f^p} = \frac{1}{N} \left(\int_0^1 \frac{f(x)^2}{p(x)} dx - \left(\int_0^1 f(x) dx \right)^2 \right)$$

Use a $p(x)$ that is somewhat similar to the shape of $f(x)$, to minimize the variance of $f(x)/p(x)$.

Importance Sampling

Approach: sample more intensively where f is large, which is where $f(x)$ makes the greatest contribution to the integral.



Quasi–Monte Carlo Integration

Relies on ideas from number theory and Fourier analysis.

Can result in a faster rate of convergence than Monte Carlo.

Main difference from Monte Carlo: use low-discrepancy **deterministic** sequences instead of pseudo-random sequences.

Low-discrepancy sequence: a sequence with the property that for all values of N, its subsequence $x_1, \ldots, x_N$ has a low discrepancy with respect to interval $[a, b]$:

$$D_N = \sup_{a \leq c \leq d \leq b} \left| \frac{\#\left|\{x_1, \ldots, x_N\} \cap [c, d]\right|}{N} - \frac{d-c}{b-a} \right|$$

Main choices: Halton sequence, Sobol sequence, and Faure sequence.

Multidimensional Integration

Curse of dimensionality: time complexity grows exponentially with the number of dimensions.

- **Quadrature methods** suffer from the curse of dimensionality: with n nodes in each dimension for a d-dimensional problem, n^d functional evaluations are required
- **Monte Carlo integration** convergence is $O(n^{-1/2})$ and independent of dimensionality
- **Quasi-Monte Carlo** can get can get convergence approaching $O(n^{-1})$

$\implies$ Monte Carlo integration is generally used in high dimensions when quadrature methods are intractable.

Example: estimating $\int_0^1 \int_0^1 e^x e^y dx dy$ with 5,000 nodes takes 229.14 seconds with the midpoint rule and 0.02 seconds with a Monte Carlo approach.

Practical Advice for Integration

1. Solve integral analytically, if possible
2. Use quadrature approach if feasible (e.g., low-dimension integration)
    - *`scipy.integrate.quad`* efficiently implements adaptive quadrature methods
3. If that fails, use a Monte Carlo approach
    - Simple to implement, but can also use efficient, parallelized implementations such as the Python package *`vegas`*