Lecture 6

Numerical Integration and Differentiation

Randall Romero Aguilar, PhD

This draft: October 10, 2018

Universidad de Costa Rica
SP6534 - Economía Computacional

Table of contents

Introduction

A Statistics Problem

What is the probability that a standard normal random variable $\tilde{Z}$ will realize a value less than or equal to z?

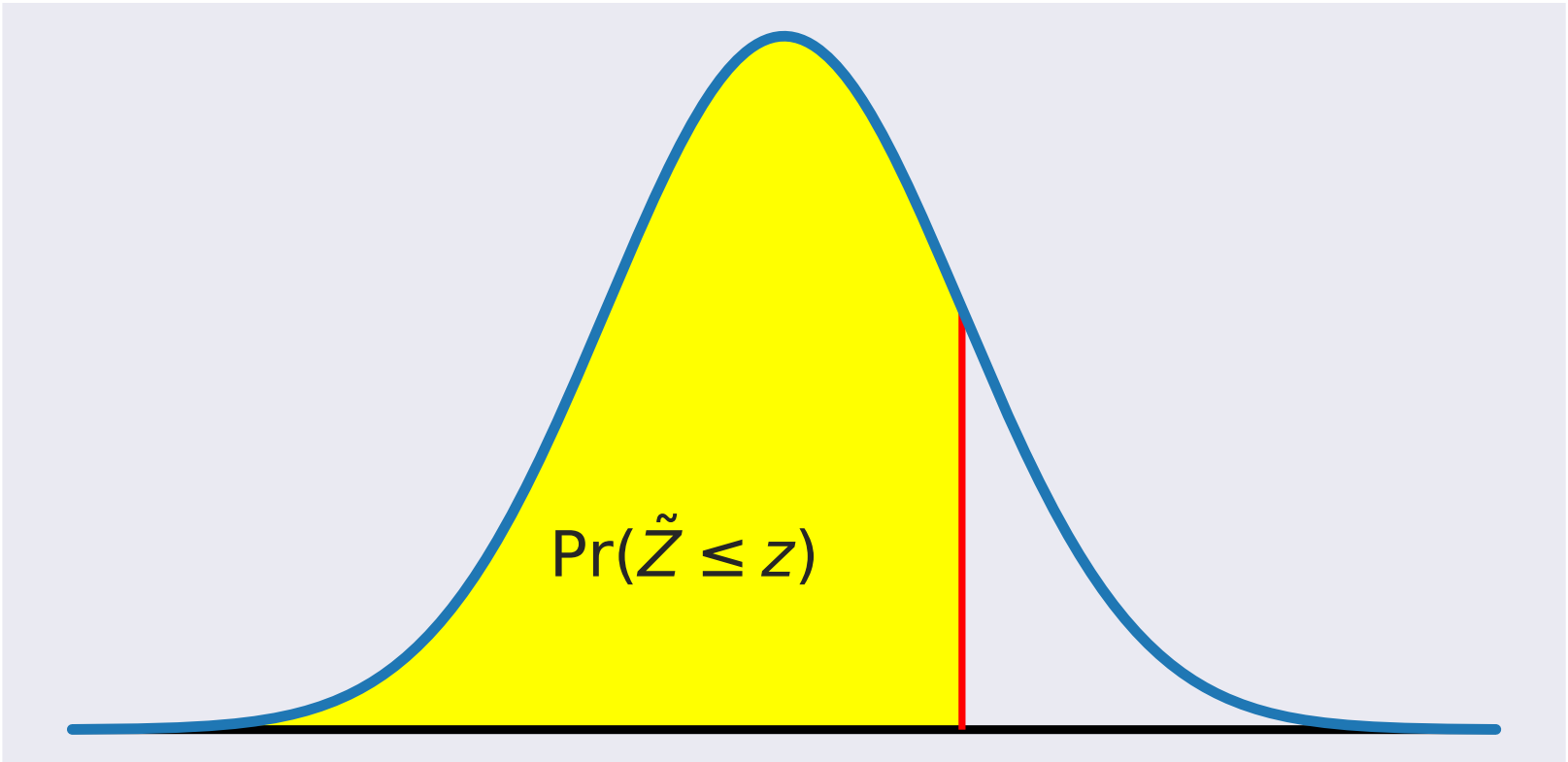


Figure 1: Standard Normal Density

- The probability is the area under the standard normal probability density function to the left of z:

$$\Pr(\tilde{Z} \leq z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} \exp\left(-\frac{t^2}{2}\right) \, dt$$

- Evaluate this integral for $z = 1$.
- Are you having difficulties?
- No surprise, since the integral lacks closed-form.
- Check the table in the back of your statistics book.
- It says the probability is 0.841.
- How did the author get this value?

A Risk Problem

- An agent's utility of income y exhibits constant absolute risk aversion $\alpha > 0$:

$$u(y) = -\exp(-\alpha y).$$

- The agent faces uncertain income $\tilde{y}$ that is lognormally distributed with log mean μ and log variance σ^2.
- Would this agent accept a certain income y^* in place of his uncertain income $\tilde{y}$?
- According to expected utility theory, yes, provided

$$u(y^*) > Eu(\tilde{y}).$$

- To answer the question definitively, we must evaluate the agent's expected utility with the uncertain income:

$$Eu(\tilde{y}) = -\frac{1}{\sigma\sqrt{2\pi}} \int_0^\infty \frac{1}{y} \exp\left(-\frac{(\ln(y)-\mu)^2}{2\sigma^2} - \alpha y\right) \, dy.$$

- Evaluate this expression when $\mu = 0$, $\sigma^2 = 0.1$, and $\alpha = 2$.
- Are you having difficulties?
- No surprise, since the integral lacks closed-form.

Numerical Quadrature

- In economics, we encounter two types of integration problems:
  - Evaluate the area under a curve, as in the “statistics” example.
  - Evaluate the expectation of a function of a random variable, as in the “risk” example.
- In many applications, the definite integral lacks an equivalent closed-form expression or is otherwise analytically intractable.
- However, such integrals typically may be easily and accurately evaluated numerically using quadrature methods.

- We discuss three classes of quadrature methods:
    - Newton-Cotes rules
    - Gaussian quadrature
    - Monte Carlo simulation
- All these methods have one thing in common: the definite integral is approximated using a weighted sum of function values at prescribed nodes, a simple task on a computer.
- Methods differ only in how the weights and nodes are chosen.

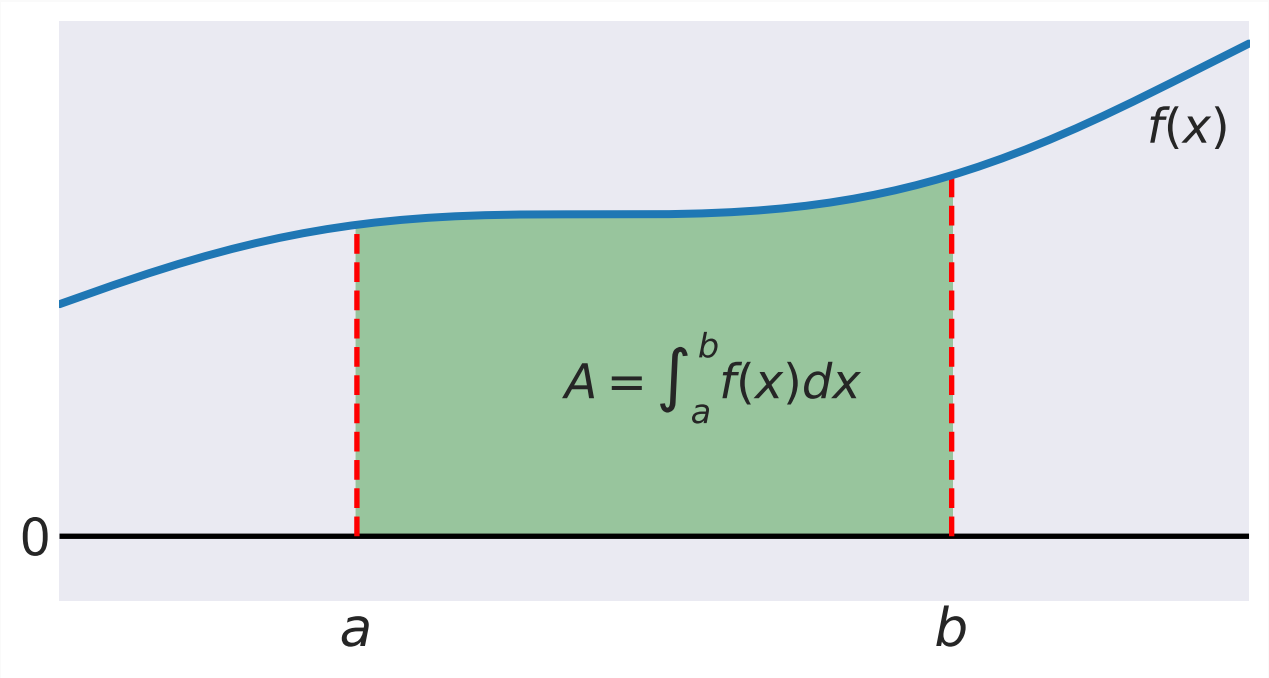
Area Under a Curve

Consider finding the area under a continuous real-valued function f over a bounded interval $[a, b]$:

$$A = \int_a^b f(x)\, dx$$

3 quadrature methods are commonly used to compute areas:

- Trapezoid rule
- Simpson's rule
- Gauss-Legendre quadrature



Trapezoid Rule

- The trapezoid rule approximates the area under a function f with the area under a piecewise linear approximation to f, $\tilde{f}$.
- Partition the interval $[a, b]$ into n subintervals of equal length $h = (b - a)/n$ defined by the nodes $x_i = a + ih$, $i = 0, 1, \ldots, n$.
- Compute the function values $y_i = f(x_i)$ at the nodes.
- Construct $\tilde{f}$ by connecting successive points (x_i, y_i) on the graph of f with straight lines.
- The area under $\tilde{f}$ is a series of trapezoids, giving the trapezoid rule its name.

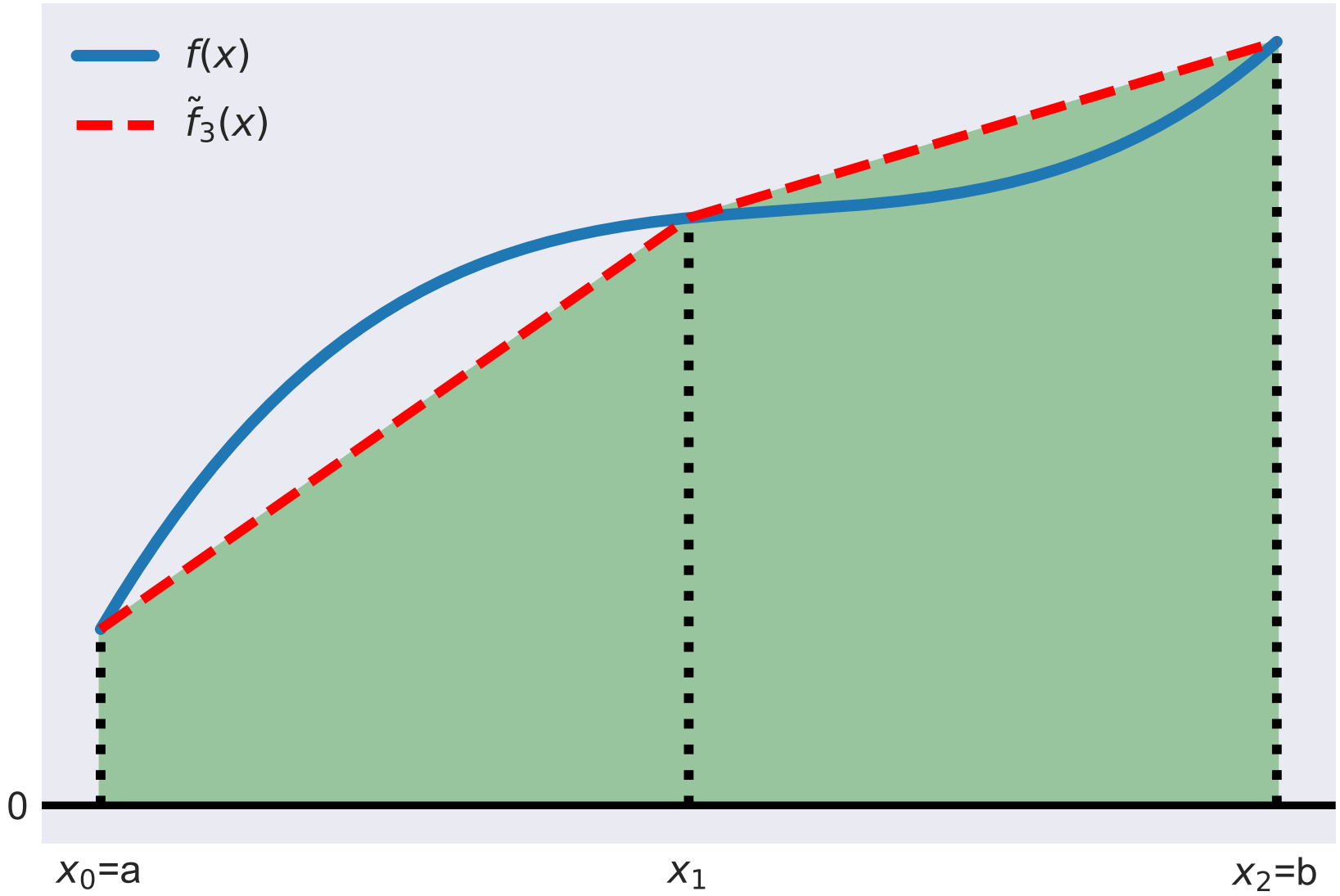


Figure 3: Trapezoid Rule, $n = 2$

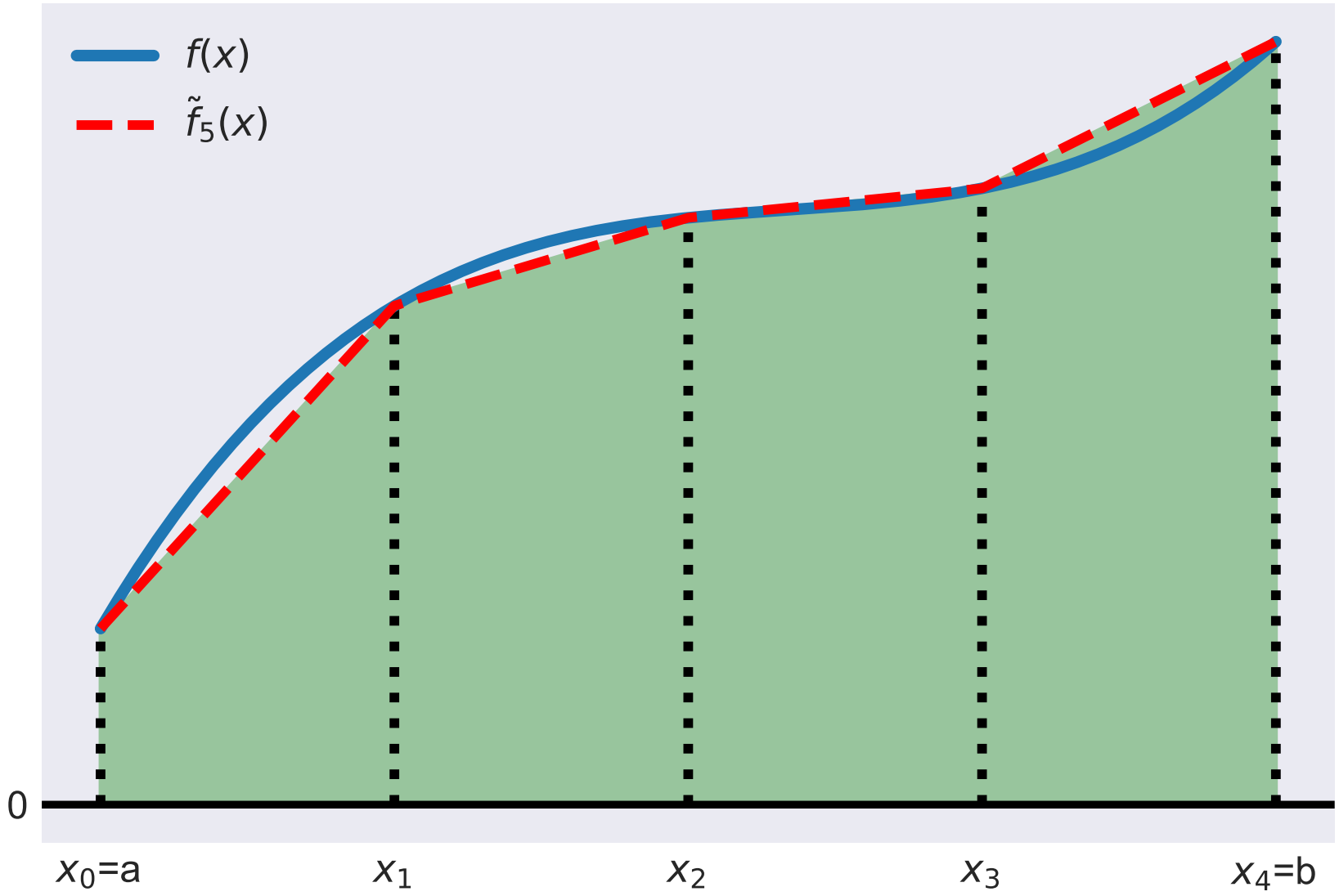


Figure 4: Trapezoid Rule, $n = 4$

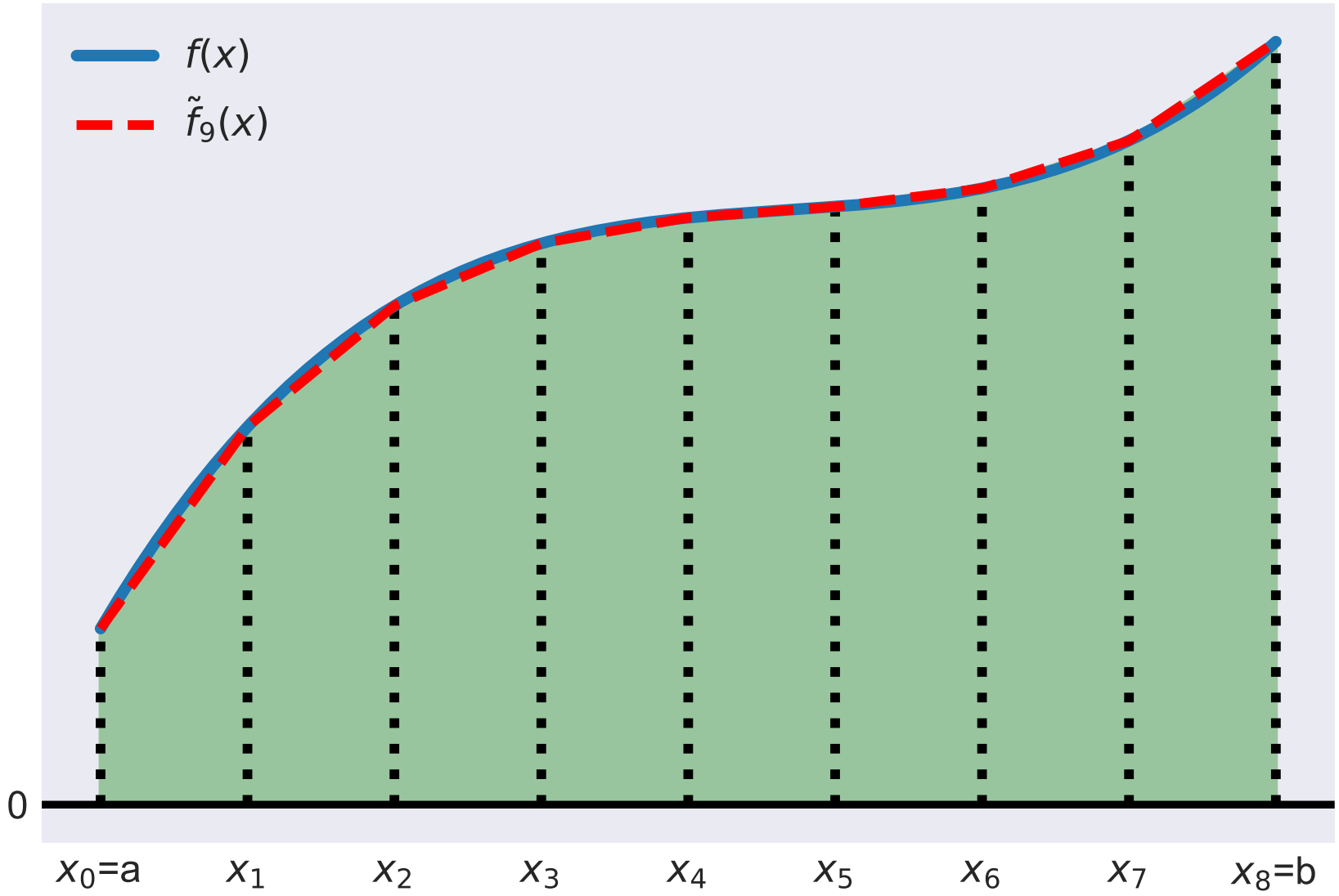


Figure 5: Trapezoid Rule, $n = 8$

- The area of the i^{th} trapezoid is:

$$\int_{x_{i-1}}^{x_i} \tilde{f}(x)\, dx = \frac{h}{2}[f(x_{i-1}) + f(x_i)].$$

- Summing the areas of all n trapezoids yields the trapezoid rule:

$$\int_a^b f(x)\, dx \approx \sum_{i=0}^{n} w_i\, f(x_i)$$

where

$$w_i = \begin{cases} h/2 & i = 0, i = n \\ h & \text{otherwise} \end{cases}$$

- The trapezoid rule is simple and robust.
- If f is smooth, the trapezoid rule affords an approximation error proportional to h^2.
- Doubling the number of nodes will reduce the approximation error by a factor of four.

Simpson's Rule

- Simpson's rule approximates the area under a function f with the area under a piecewise quadratic approximation to f, $\tilde{f}$.
- Partition the interval $[a, b]$ into n subintervals of equal length $h = (b - a)/n$ defined by the nodes $x_i = a + ih$, $i = 0, 1, \ldots, n$, n even.
- Compute the function values $y_i = f(x_i)$ at the nodes.
- Form a piecewise quadratic approximation $\tilde{f}$ for f by interpolating successive triplets of graph points (x_{i-2}, y_{i-2}), (x_{i-1}, y_{i-1}), and (x_i, y_i), $i = 2, 4, \ldots, n$, with quadratic functions.

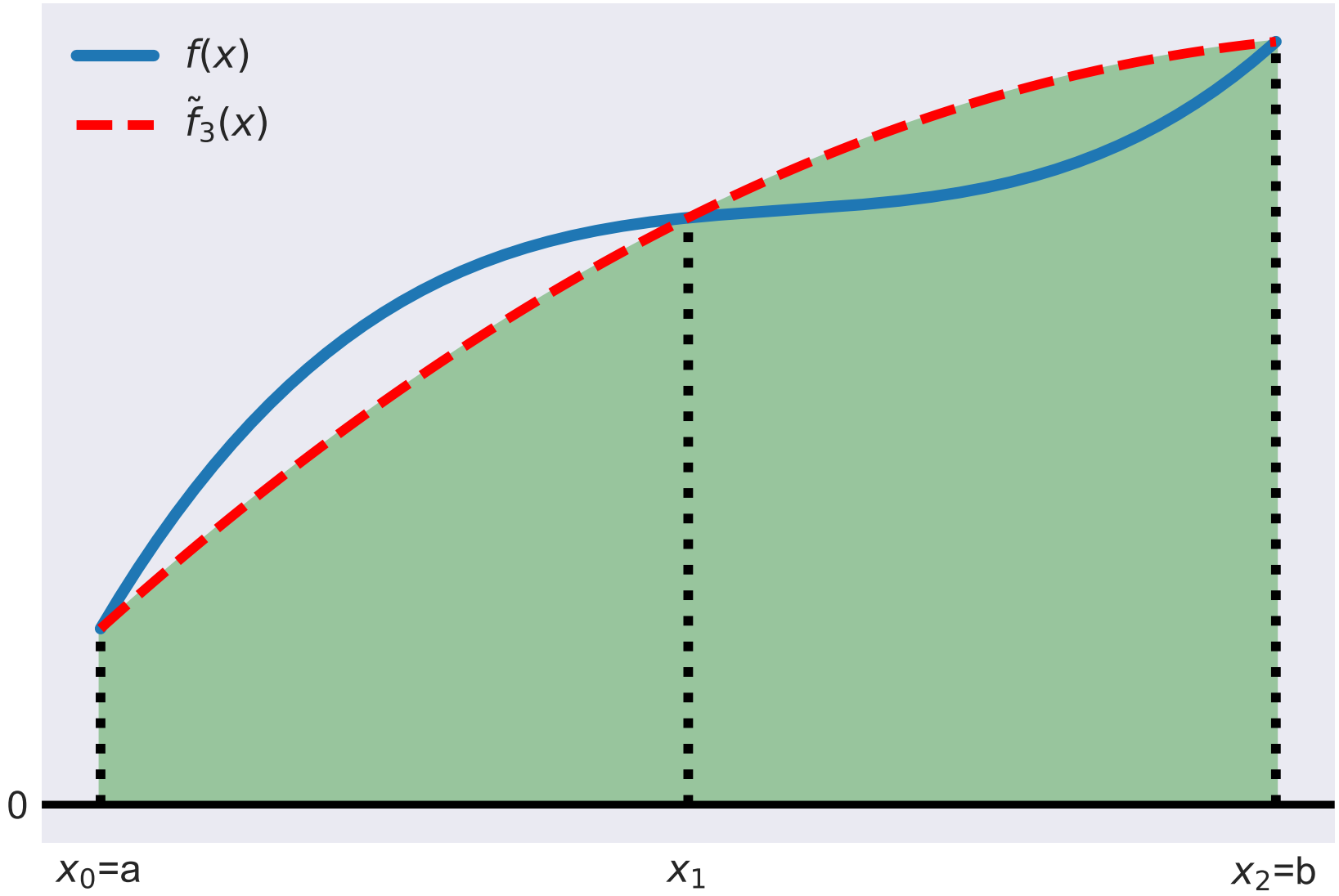


Figure 6: Simpson's Rule, $n = 2$

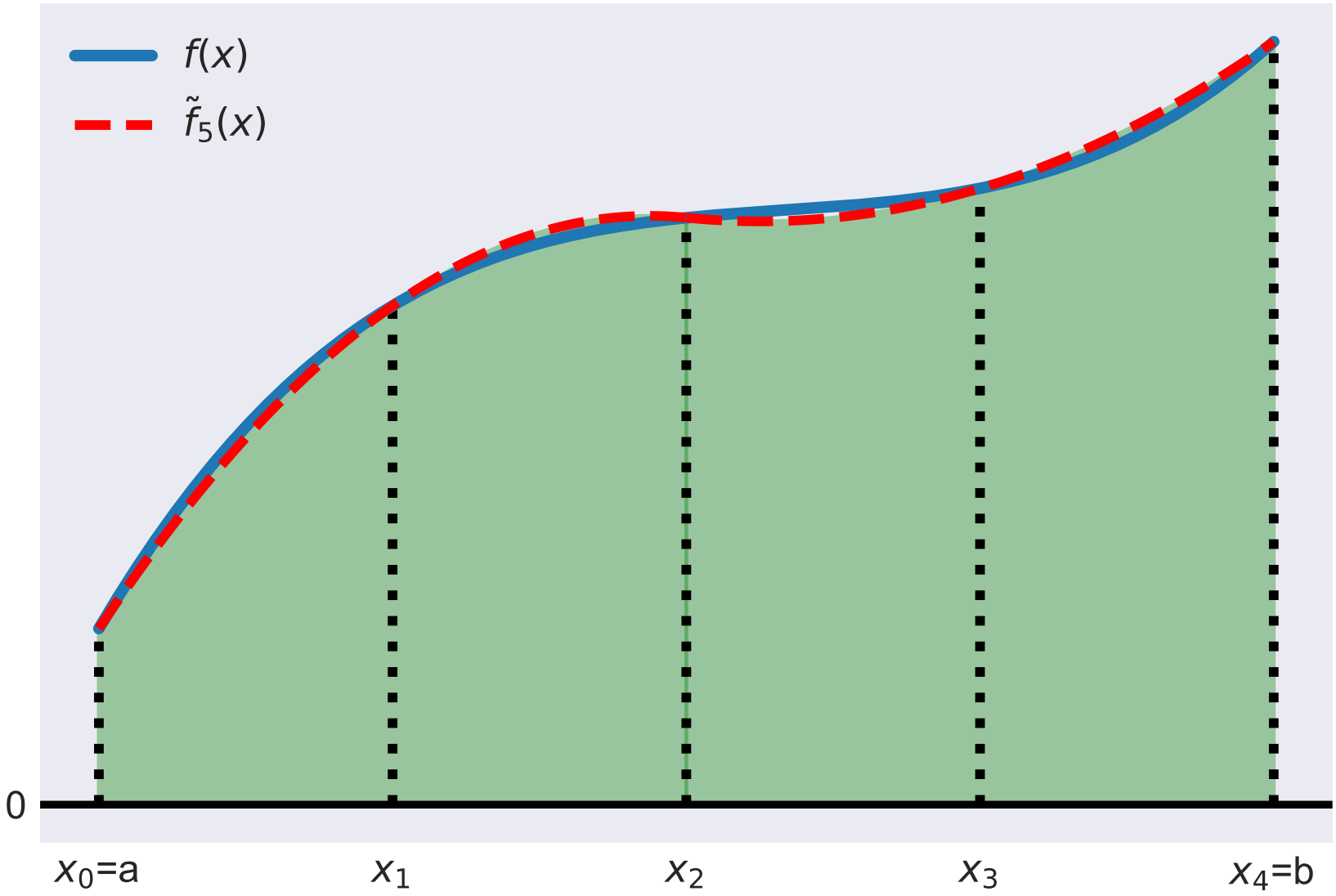


Figure 7: Simpson's Rule, $n = 4$

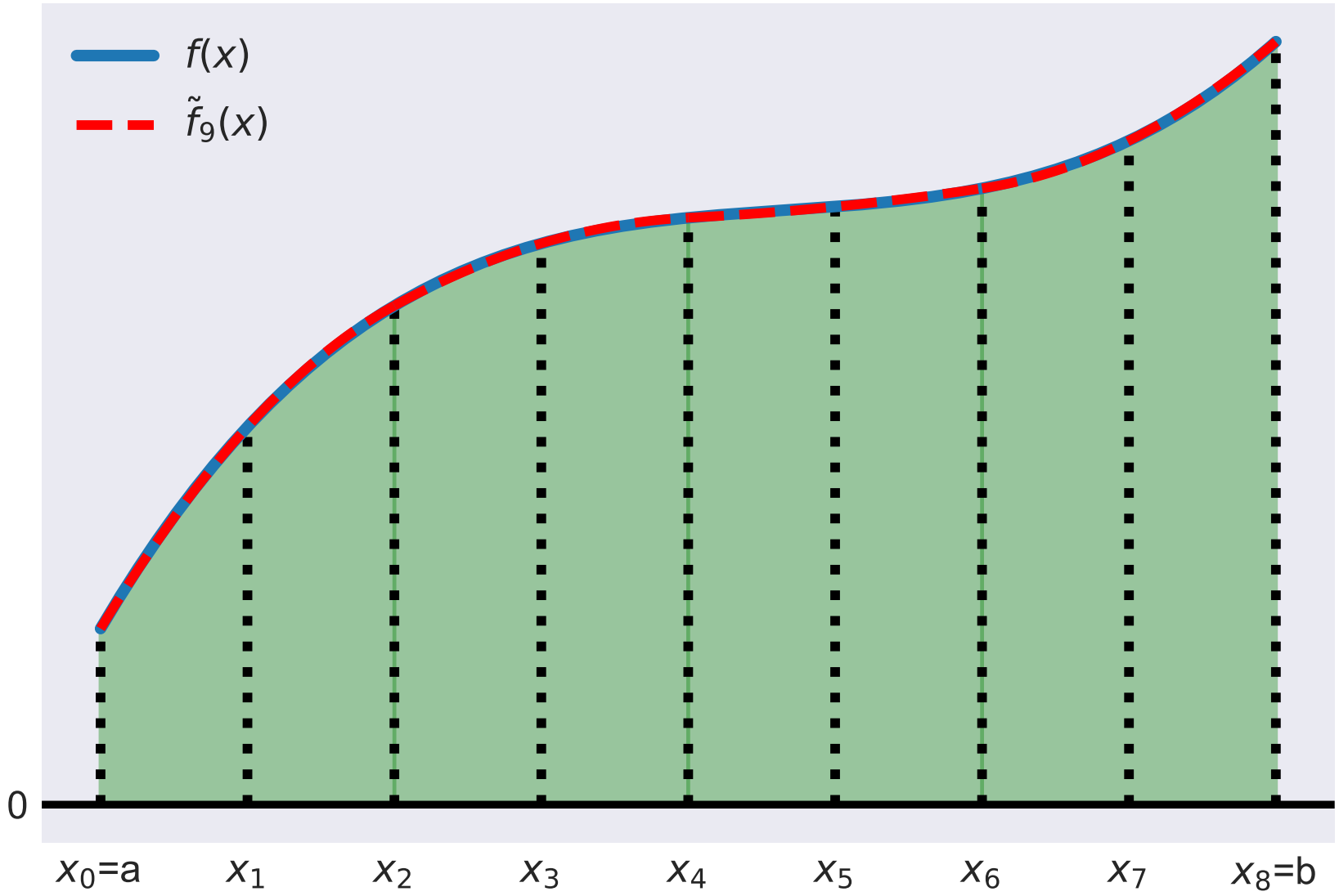


Figure 8: Simpson's Rule, $n = 8$

- The area under $\tilde{f}$ over subintervals $i-1$ and i, for $i = 2, 4, \ldots, n$, is:

$$\int_{x_{i-2}}^{x_i} \tilde{f}(x)\, dx = \frac{h}{3}\left(f(x_{i-2}) + 4f(x_{i-1}) + f(x_i)\right).$$

- Summing across successive pairs of subintervals yields Simpson's rule:

$$\int_a^b f(x)\, dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

where

$$w_i = \begin{cases} h/3 & i = 0, i = n \\ 4h/3 & 0 < i < n, i \text{ even} \\ 2h/3 & 0 < i < n, i \text{ odd} \end{cases}$$

- If f is smooth, Simpson's rule affords an approximation error proportional to h^4, the square of the trapezoid rule error.
- Doubling the number of nodes will reduce the approximation error by a factor of sixteen.
- Simpson's rule is preferred to the trapezoid rule because it is almost as simple, but far more accurate.

Newton-Cotes Rules

- The trapezoid rule and Simpson's rule are examples of Newton-Cotes rules, which replace the integrand with a piecewise polynomial of low degree.
- Newton-Cotes rules based on piecewise polynomials of third and higher degree can be defined, but are not practical.

Gauss-Legendre Quadrature

- Gauss-Legendre quadrature employs different logic to compute the area under a curve f over a bounded interval $[a, b]$.
- Specifically, the n quadrature nodes x_i and n quadrature weights w_i are chosen to exactly integrate polynomials of degree $2n - 1$ or less.
- This requirement imposes the $2n$ conditions

$$\int_a^b x^k \, dx = \sum_{i=1}^{n} w_i x_i^k, \qquad k = 0, \ldots, 2n-1,$$

  which may be solved for the n nodes and n weights using nonlinear equation methods.

- Unlike Newton-Cotes rules, Gauss-Legendre nodes are not uniformly spaced and do not include the integration limits.
- For example, the order-4 Gauss-Legendre nodes over the interval $[0, 1]$ are 0.069, 0.330, 0.670, and 0.931 and the corresponding weights are 0.174, 0.326, 0.326, and 0.174.

Gauss-Legendre quadrature outperforms Newton-Cotes rules if the integrand f is smooth, but otherwise may perform no better or worse.
For example, consider integrating e^{-x} and $\sqrt{|x|}$ on $[-1, 1]$.

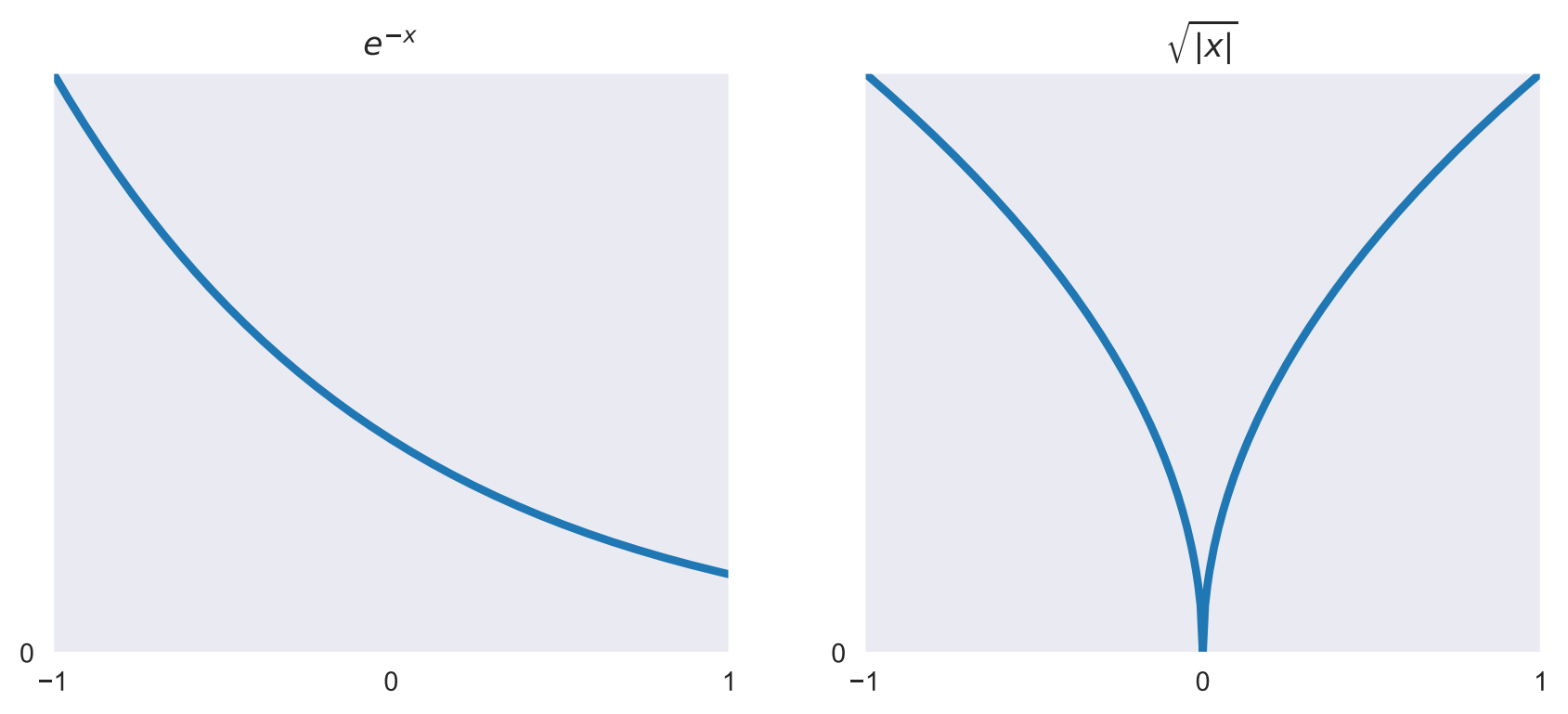


Figure 9: A smooth and a non-smooth function

Integral	Nodes n	Trapezoid rule	Simpson's rule	Gauss-Legendre
$\int_{-1}^{1} e^{-x} dx$	5	-1.7	-3.5	-9.5
	11	-2.5	-5.1	-14.3
	21	-3.1	-6.3	-14.7
	31	-3.4	-7.0	-inf
$\int_{-1}^{1} \sqrt{\lvert x \rvert} dx$	5	-1.0	-1.4	-0.9
	11	-1.6	-1.3	-1.4
	21	-2.0	-2.4	-1.8
	31	-2.3	-2.1	-2.0

Table 1: Log10 Definite Integral Relative Approximation Errors

CompEcon Utilities

- CompEcon utilities `qnwtrap`, `qnwsimp` and `qnwlege` generate the trapezoid rule, Simpson's rule, and Gauss-Legendre nodes and weights as follows:

```
x, w = qnwtrap(n,a,b)
x, w = qnwsimp(n,a,b)
x, w = qnwlege(n,a,b)
```

- Input: `n` the number of nodes, `a` the left integration limit, and `b` the right integration limit.
- Output: `x` and `w`, the `n`-vectors of quadrature nodes and weights, respectively.

Example 1:

Computing a probability

To compute the probability that a standard normal random variable is less than 1, execute the script

```
from numpy import exp, sqrt, pi
from compecon import qnwsimp

f = lambda x: exp(-x**2/2) / sqrt(2*pi)
x, w = qnwsimp(11, 0, 1)
prob = 0.5 + w.dot(f(x))
```

The computed answer, 0.8413, is correct to four significant digits.

This is how the table in your statistics book was computed.

Example 2:

Consumer surplus

- Suppose demand for a commodity is given by

$$q(p) = 0.15q^{-1.25}$$

  and the price drops from $p_1 = 0.7$ to $p_2 = 0.3$.
- Then the gain in consumer surplus

$$\int_{p_2}^{p_1} q(p)\, dp,$$

  may be computed by executing

```
from compecon import qnwlege


q = lambda x: 0.15*p**(-1.25)
p, w = qnwlege(11, 0.3, 0.7)
change = w.dot(q(p))
```

- The computed answer, 0.1548, is correct to four significant digits.

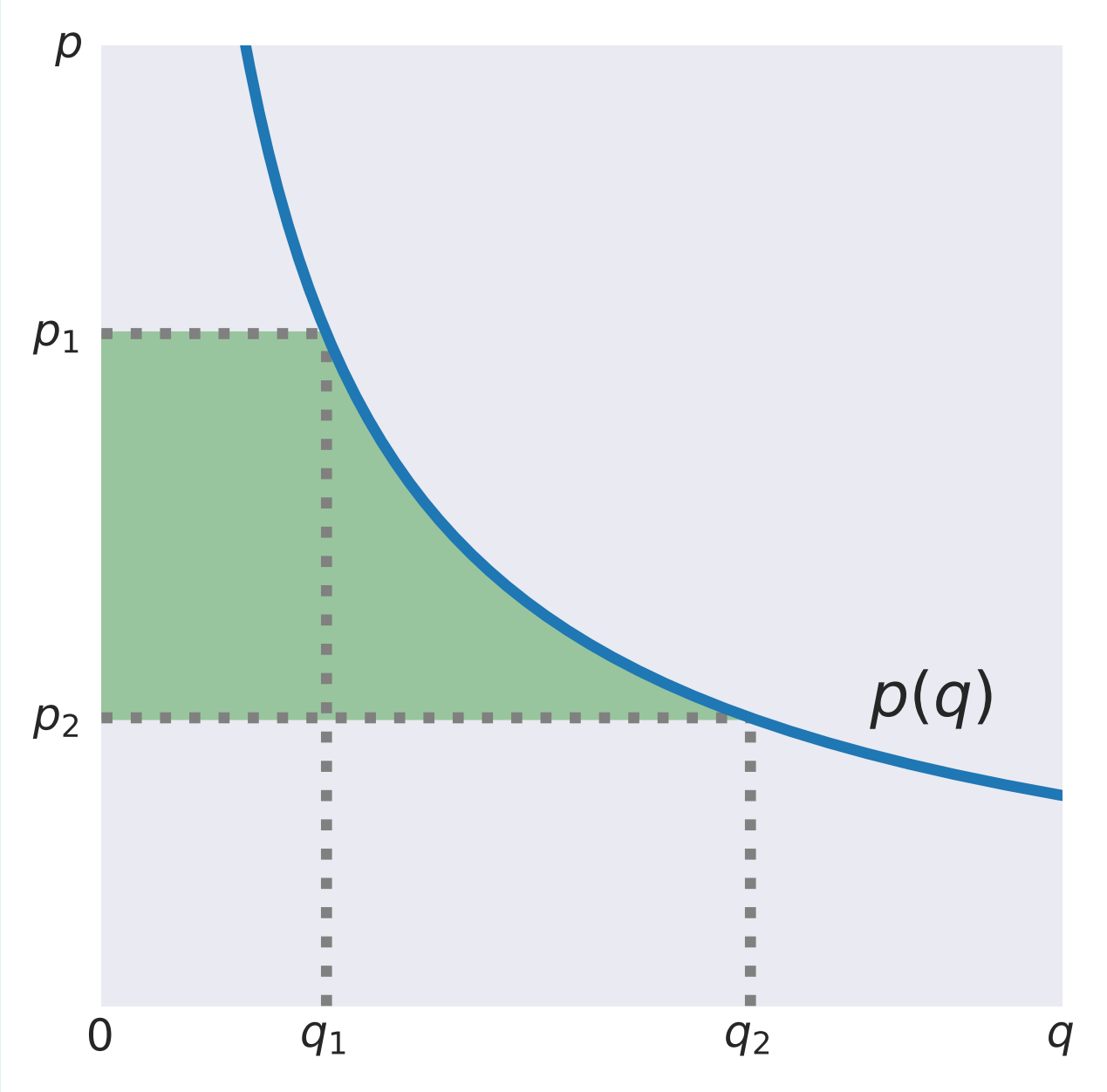


Figure 10: Change in Consumer Surplus

Computing Expectations

- In economics, we must often compute the expectation of a function f of a random variable $\tilde{X}$ with known probability density function w:

$$Ef(\tilde{X}) = \int f(x)w(x)\,dx.$$

- For example, in the "Risk Problem":
  - $\tilde{X}$ is the agent's random income,
  - f is the agent's utility of income function, and
  - w is the probability density function of income.

- Two numerical techniques are widely used to compute expectations in economics.
- Gaussian quadrature is especially powerful when the dimension of the random variable is low and the integrand f is smooth.
- Monte Carlo simulation, discussed separately in the next section, is easy to apply, and is especially useful when the random variable is high dimensional.

Gaussian Quadrature

- Gaussian quadrature replaces the continuous random variable $\tilde{X}$ with a discrete random variable that is easier to work with.
- Specifically, the n mass points x_i and n probabilities w_i of the discrete random variable are chosen so as to replicate the mean, variance, skewness, kurtosis, and, more generally, the same first $2n-1$ moments of $\tilde{X}$.
- This imposes $2n$ "moment matching" conditions

$$\sum_{i=1}^{n} w_i x_i^k = E\tilde{X}^k, \qquad k = 0, \ldots, 2n-1,$$

which may be solved for the n mass points and n probabilities using nonlinear equation methods.

- Given the discrete approximant for $\tilde{X}$, one may easily compute an approximation for the expectation of an arbitrary function f of $\tilde{X}$ as follows:

$$Ef(\tilde{X}) \approx \sum_{i=1}^{n} w_i f(x_i).$$

- By construction, the expectation approximation will be exact if f is a polynomial of degree $2n-1$ or less.
- This suggests that the expectation approximation should be accurate if f can be reasonably approximated by a polynomial of degree $2n-1$ or less; that is, if f is smooth.

Example 3:

Gaussian quadrature

- For $n = 3$, the Gaussian quadrature mass points and probabilities for a standard normal variable $\tilde{Z}$ are

$$\begin{array}{rcrrcr} x_1 & = & -\sqrt{3} & w_1 & = & 1/6 \\ x_2 & = & 0 & w_2 & = & 2/3 \\ x_3 & = & \sqrt{3} & w_3 & = & 1/6 \end{array}$$

- The exact value of $E \exp(\tilde{Z})$, expressed to four significant digits, is 1.6487.
- The Gaussian quadrature approximation,

$$E \exp(\tilde{Z}) \approx \frac{1}{6} \exp(-\sqrt{3}) + \frac{2}{3} \exp(0) + \frac{1}{6} \exp(\sqrt{3}) = 1.6382,$$

  is accurate to less than 1%, a remarkable fact given that we used a three-point approximation.

CompEcon Utilities

- The CompEcon Toolbox contains utilities for generating discrete approximants for common probability distributions.
- All utilities generate mass points `x` and probabilities `w` as output, and require the number of mass points `n` as input, but differ with respect to other inputs:

- Normal Distribution

```
x, w = qnwnorm(n,mu,var)
```

  Here, `mu` is the mean and `var` is the variance.

- Lognormal Distribution

```
x, w = qnwlogn(n,mu,var)
```

  Here, `mu` is the log mean and `var` is the log variance.

- Beta Distribution

```
x, w = qnwbeta(n,a,b)
```

  Here, `a` and `b` are the shape parameters.

- Gamma Distribution

```
x,w = qnwgamma(n,a,b)
```

  Here, `a` is the shape parameter and `b` is the scale parameter.

Example 4:

A risk problem

- Let us revisit the “risk problem”, in which an agent possesses utility of income $u(y) = -\exp(-\alpha y)$, $\alpha > 0$, and faces uncertain income $\tilde{y}$ that is lognormally distributed with parameters μ and σ^2.
- Would the agent accept a certain income $y^* = 1$ in place of the uncertain income $\tilde{y}$ if $\mu = 0$, $\sigma^2 = 0.1$, and $\alpha = 2$?

- To compute the agent's expected utility with random income, execute the script

```
from compecon import qnwlogn
from numpy import exp

n = 100
mu, var, alpha = 0, 0.1, 2
y, w = qnwlogn(n,mu,var)
Eu = -w.dot(exp(-alpha*y))
```

- This generates the approximation $Eu(\tilde{y}) = -0.148$, which is less than $u(y^*) = -0.135$.
- Yes, the agent would accept the certain income.

- The utility `qnwnorm` also generates discrete approximations for multivariate normal variates.
- To generate mass points and probabilities for d jointly distributed normal random variables, execute the script

```
x, w = qnwnorm(n,mu,var);
```

  where `n` is a $1 \times d$ vector indicating the number of mass points for each variable, `mu` is the $1 \times d$ mean vector, and `var` is the $d \times d$ covariance matrix.
- On output, `x` is an $d \times N$ matrix of mass points and `w` is an N vector of probabilities, where $N = n(1) \cdot n(2) \cdot \ldots \cdot n(d)$.

Example 5:

A farmer's problem

- A farmer's per-acre revenue is the product of the unit price $\tilde{p}$ and per-acre yield $\tilde{y}$, the logs of which are jointly normally distributed with mean vector and covariance matrix

$$\mu = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 0.2 & -0.1 \\ -0.1 & 0.4 \end{bmatrix}$$

- To compute the farmer's expected revenue using a grid of 150 mass points formed as the Cartesian product of 10 price nodes and 15 yield nodes:

```
from compecon import qnwnorm
from numpy import exp
mu, sigma = [1, 2], [[0.2,-0.1],[-0.1, 0.4]]
(p,y), w = qnwnorm([10,15], mu, sigma)
expectedrevenue = w.dot(exp(p+y))
```

Monte Carlo Simulation

- Monte Carlo simulation offers an alternative way to compute expectations that is especially useful when the random variable is high-dimensional.
- Monte Carlo simulation is motivated by the Strong Law of Large Numbers.
- The Law states that if $x_1, x_2, \ldots$ are independent realizations of a random variable $\tilde{X}$ and f is a continuous function, then

$$Ef(\tilde{X}) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

  with probability one.

- The Monte Carlo simulation scheme is simple.
- To compute an approximation for $Ef(\tilde{X})$, draw a random sample $x_1, x_2, \ldots, x_n$ from the distribution of $\tilde{X}$ and set

$$Ef(\tilde{X}) \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i).$$

- But how do you draw a random sample?

Random Number Generators

- A random number generator is an algorithm that generates what appears to be a sequence of independent realizations of a random variable with a specified distribution.
- A fundamental problem with so-called random number generators is that they employ purely deterministic, not random iteration rules.
- If you repeatedly initiate a generator at the same point, it will generate the same sequence of “random” numbers each time.
- The best that can be said of a random number generators is that a good one will generate realizations that pass certain statistical tests for randomness.
- For this reason, random number generators are perhaps best said to be “pseudo-random” number generators.

- `numpy.random` provides two intrinsic random number generators.
- `rand(m,n)` generates an $m \times n$ matrix of numbers that are independently uniformly distributed on the interval $[0, 1]$.
- `randn(m,n)` generates an $m \times n$ matrix of numbers that are independently standard normally distributed.

Example 6:

Generating random numbers from a standard normal distribution

- To generate 100,000 independent realizations of a standard normally distributed random variable, execute

```
from numpy.random import randn
x = randn(100_000)
```

- Based on your knowledge of normal distribution theory, what would you expect the following commands to produce:

```
x.mean()
x.std()
(x < 1.6449).mean()
```

- You would expect 0, 1, and 0.95.
- The computed values are close, but not exact.

Example 7:

Generating random numbers from a uniform distribution

- To generate 100,000 independent realizations of a uniform (0,1) random variable, execute

```
from numpy.random import rand
x = rand(100_000)
```

- Based on your knowledge of distribution theory, what would you expect the following commands to produce:

```
x.mean()
x.std()
(x < 0.2).mean()
```

- You would expect 0.5, $\sqrt{1/12} = 0.2887$, and 0.2.
- The computed values are close, but not exact.

- The `scipy.stats` library provides random number generators for over 90 distributions, including the Beta, Binomial, Exponential, Extreme Value, Gamma, Logistic, Lognormal, Normal, and Poisson distributions.
- To generate an $n \times m$ matrix of independent realizations of a given random variable, the calling protocol is

  ```
  x = dist.rvs(*parameters,size=[n,m])
  ```

  where `dist` is the name of the distribution and `parameters` are the parameters of the distribution.

Example 8:

Approximating an expected value

- To approximately compute $\mathbb{E}\,\tilde{X}^{-1}$ where $\tilde{X}$ is distributed Beta with shape parameters 1.5 and 3.0, execute the script

```
x = beta.rvs(1.5, 3.0, size=100_000)
(1 / x).mean()
```

- To approximately compute $\mathbb{E}\min(\tilde{X},\tilde{Y})$ where $\tilde{X}$ and $\tilde{Y}$ are independent, $\tilde{X}$ is Gamma distributed with shape parameter 1.5 and scale parameter 3.0, and $\tilde{Y}$ is exponentially distributed with mean 1, execute the script

```
x = gamma.rvs(1.5,3.0,size=100_000)
y = exponential.rvs(1,size=100_000)
np.minimum(x,y).mean()
```

Example 9:

Approximating a probability

- To compute an approximate value for $\Pr(\tilde{Y} < \tilde{X}^2)$ where $\tilde{X}$ and $\tilde{Y}$ are independent, $\tilde{X}$ is extreme value distributed with location parameter 0.5 and scale parameter 1.0, and $\tilde{Y}$ is geometrically distributed with probability parameter 0.3, execute the script

```
x = genextreme.rvs(0,0.5,1.0,size=100_000)
y = geom.rvs(0.3,size=100_000)
(y<x**2).mean()
```

- A more accurate way would be to execute the script

```
geom.cdf(x**2,0.3).mean()
```

Example 10:

A random commodity price

- A commodity price is governed by weekly price movements

$$\log(p_{t+1}) = \log(p_t) + \tilde{\epsilon}_t$$

  where the $\tilde{\epsilon}_t$ are i.i.d. normal with mean $\mu = 0.005$ and standard deviation $\sigma = 0.02$.
- To simulate three time series of 40 weekly price changes, starting from a price of 2, execute the script

```
m, n = 3, 40
mu, sigma = 0.005, 0.02
e = norm.rvs(mu,sigma,size=[n,m])
logp = np.zeros([n+1,m])
logp[0] = np.log(2)
for t in range(40):
    logp[t+1] = logp[t] + e[t]

plt.plot(np.exp(logp))
```

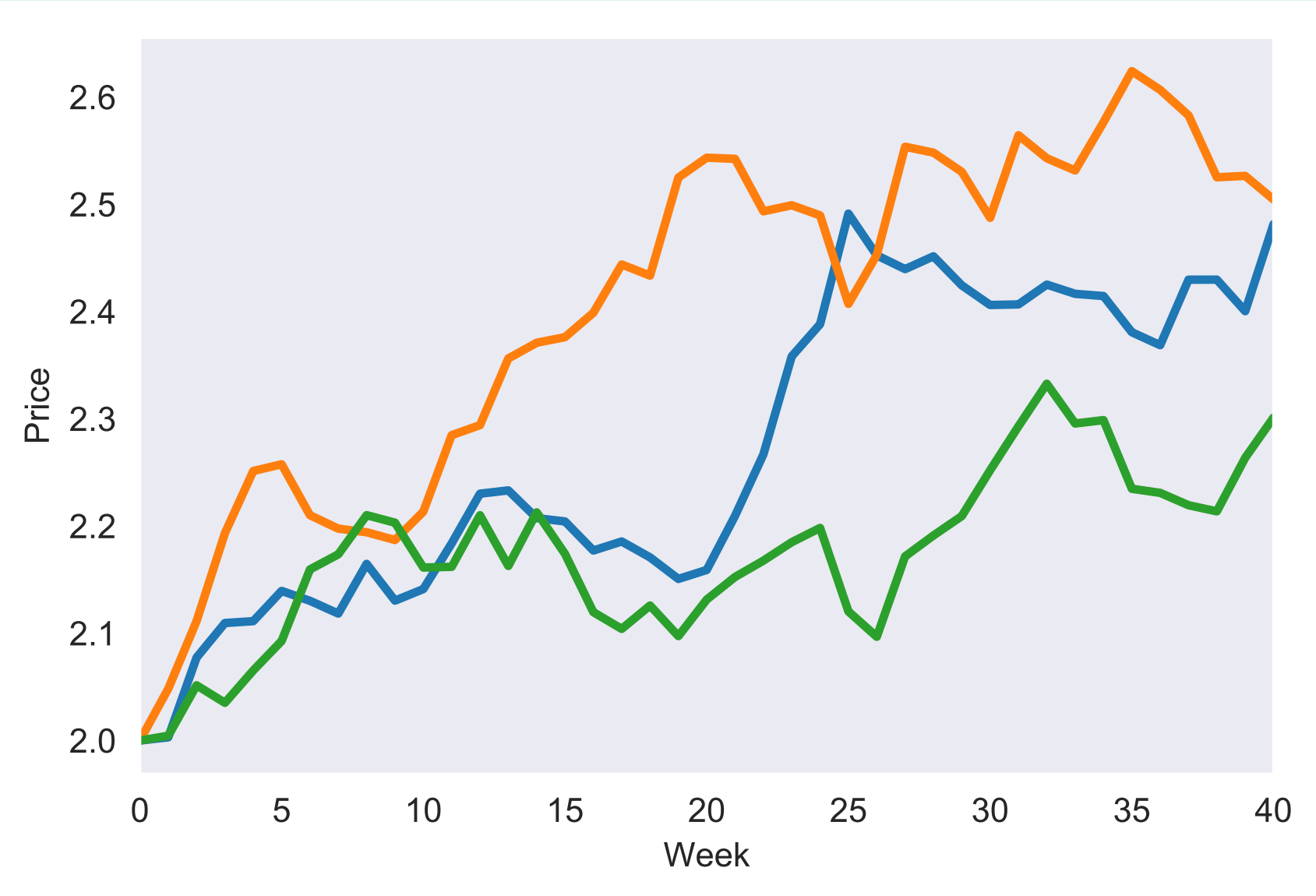


Figure 11: Time Series Simulation

- The `scipy.stats` library also provides a random number generator for multivariate normally distributed random vectors.
- To generate n independent realizations of a d-dimensional normally distributed random vector, the calling protocol is

```
r = multivariate_normal.rvs(mu, var, size=n)
```

where `mu` is a d mean vector and `var` is a $d \times d$ positive definite variance matrix.

Example 11:

A farmer facing random prices and yields

- A farmer's per-acre revenue is the product of the unit price $\tilde{p}$ and per-acre yield $\tilde{y}$, the logs of which are jointly normally distributed with mean vector and covariance matrix

$$\mu = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 0.2 & -0.1 \\ -0.1 & 0.4 \end{bmatrix}$$

- To compute the farmer's expected revenue using 100,000 independent realizations of the joint distribution of price and yield, execute:

```
mu, sigma = [1, 2], [[0.2, -0.1],[-0.1, 0.4]]
p, y = multivariate_normal.rvs(mu, sigma, size=100_000).T
expectedrevenue = np.exp(p + y).mean()
```

Pros and Cons

- Monte Carlo simulation possesses certain advantages.
- Monte Carlo simulation is easy to implement.
- Most software application packages (e.g., Excel) provide random number generators, but do not provide functions that compute Gaussian quadrature mass points and probabilities.
- Multidimensional Gaussian quadrature suffers from the "curse of dimensionality" – if you use n nodes in each of d directions, you end up with n^d mass points.
- Monte Carlo simulation does not suffer from the curse and is especially useful when simulating time-series of autocorrelated random variables, whose dimension equals the length of the series.

- Monte Carlo simulation, however, possesses some disadvantages.
- Approximations generated by Monte Carlo simulation will vary from one integration to the next, and are subject to a sampling error that cannot be bounded with certainty.
- The approximation can be made more accurate, in a dubious statistical sense, by increasing the size of the random sample, but this can be expensive.
- Monte Carlo simulation should be avoided when other methods are practicable, and used only when taking expectations over high-dimensional random variables.

Quasi-Monte Carlo Integration

- Quasi-Monte Carlo methods employ deterministic sequences of nodes x_i with the property that

$$\lim_{n\to\infty} \frac{b-a}{n} \sum_{i=1}^{\infty} f(x_i) = \int_a^b f(x)\, dx,$$

  for smooth functions f without regard to whether they pass tests of randomness.
- Deterministic sequences of nodes chosen to fill space in a regular manner typically provide more accurate integration approximations than pseudo-random sequences.

- There are numerous algorithms for generating equidistributed sequences, including the Neiderreiter and Weyl sequences.
- The algorithm are explained in detail in the Judd and Miranda & Fackler textbooks, but are not important for us here.
- Let us examine examples of equidistributed sequences on the unit square.

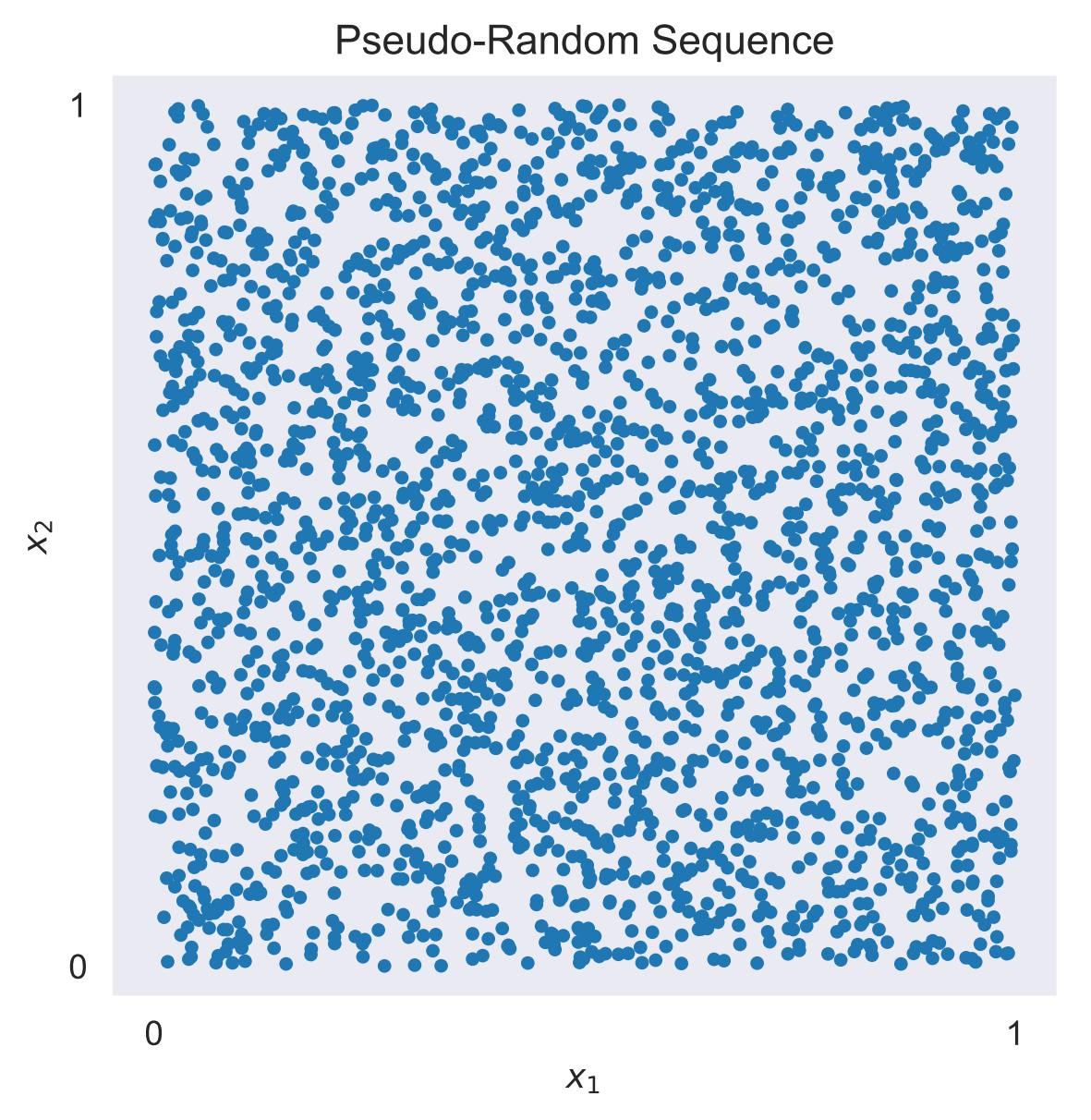


Figure 12: Pseudo-Random Sequence on Unit Square

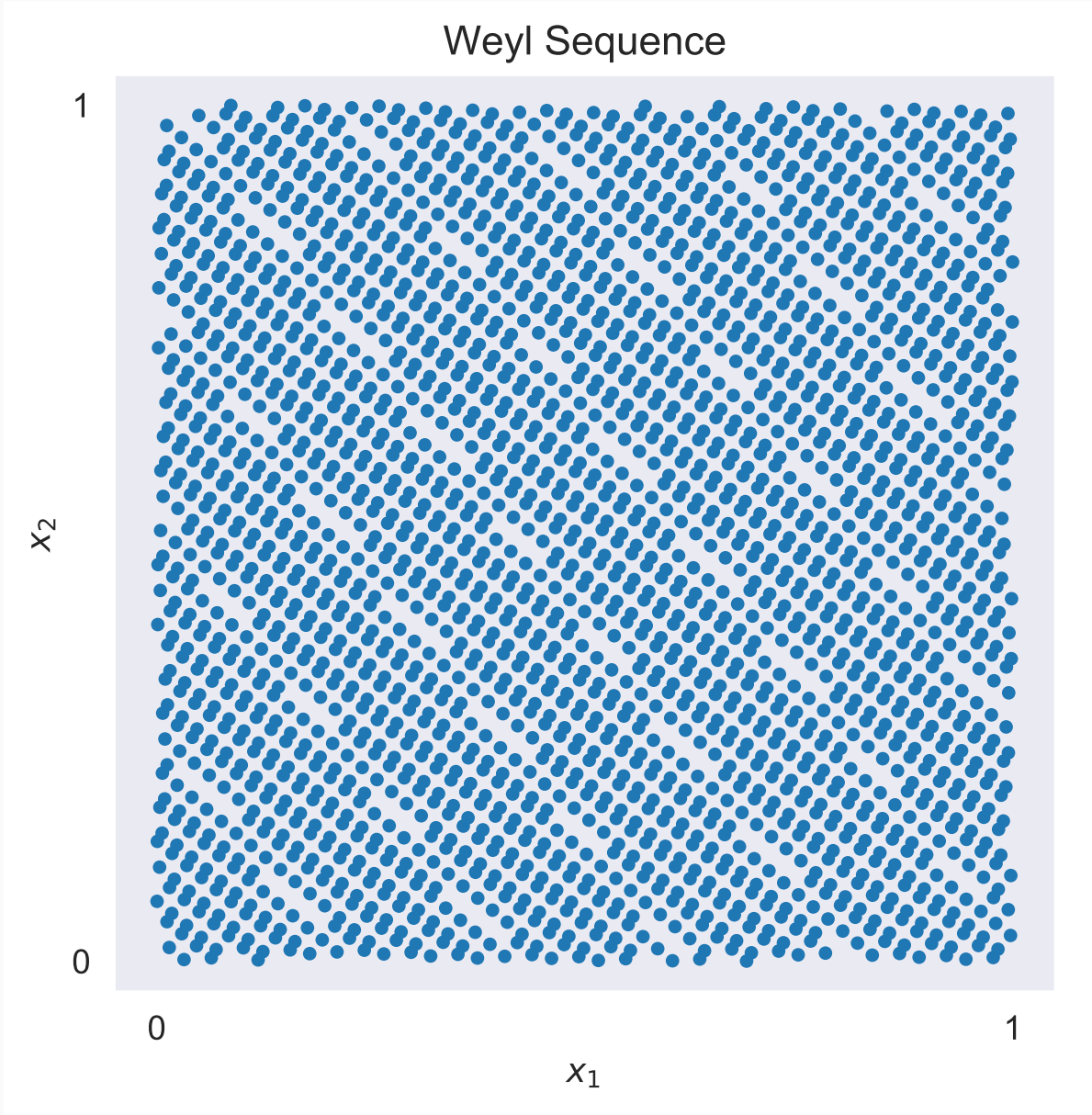


Figure 13: Weyl Sequence on Unit Square

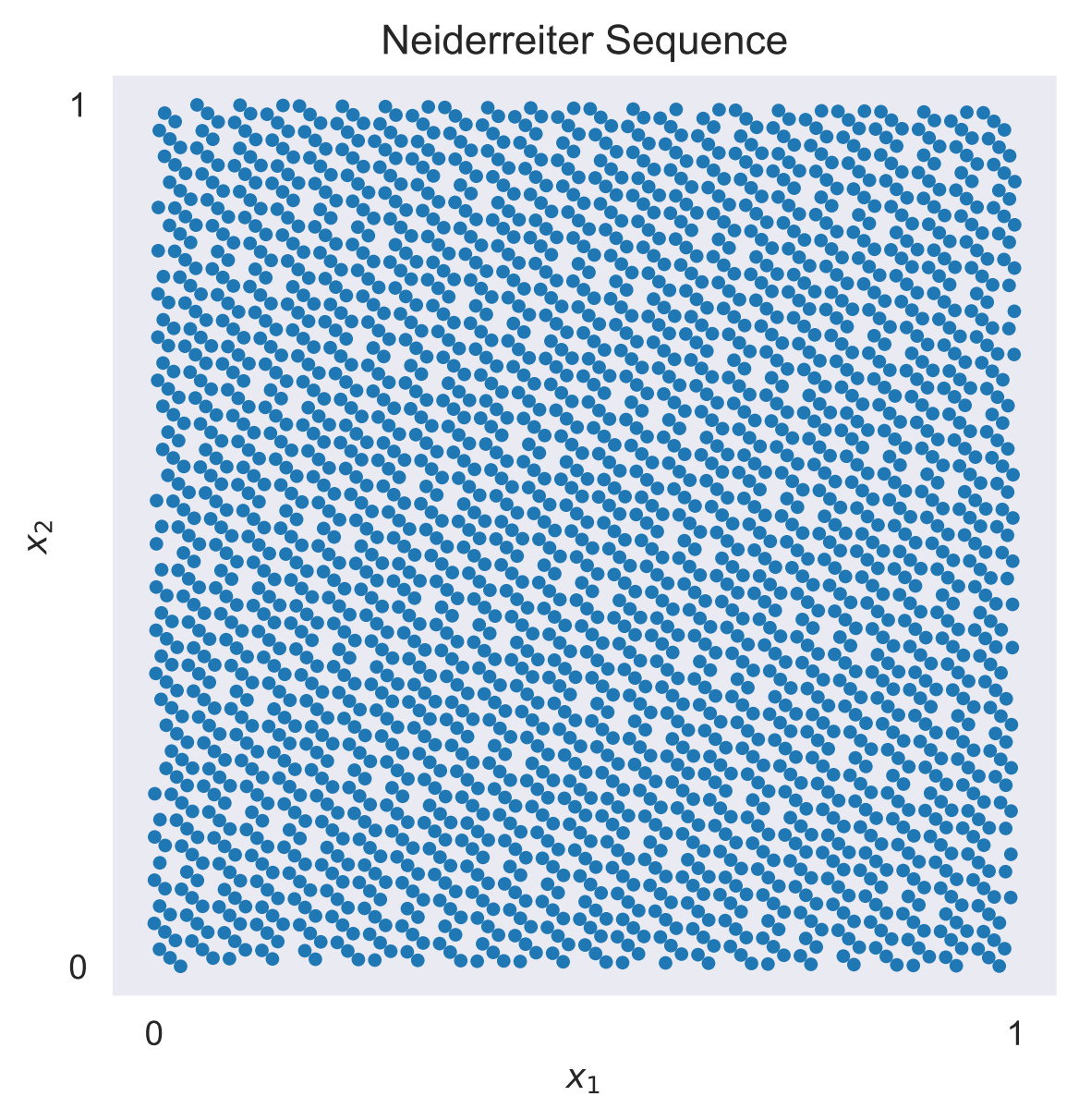


Figure 14: Neiderreiter Sequence on Unit Square

- CompEcon utility `qnwequi` generates equidistributed nodes and weights as follows

```
x, w = qnwequi(n,a,b,type)
```

- The inputs `n` = the number integration nodes, `a` = left integration limit, and `b` = right integration limit.
- The additional input `type` refers to the type of equidistributed sequence:

  `'N'` = Neiderrieter (the default),
  `'W'` = Weyl, and
  `'R'` = uniform pseudo-random.

- The integration limits are d vectors if the integration is taking place over a d-dimensional hypercube.

Example 12:

Quasi-Monte Carlo Integration

To seven significant digits,

$$
\begin{aligned}
A &= \int_{-1}^{1}\int_{-1}^{1} e^{-x_1}\cos^2(x_2)\,\mathrm{d}x_1\,\mathrm{d}x_2 \\
&= \int_{-1}^{1} e^{-x_1}\,\mathrm{d}x_1 \times \int_{-1}^{1}\cos^2(x_2)\,\mathrm{d}x_2 \\
&= \left(e-\tfrac{1}{e}\right)\times\left(1+\tfrac{1}{2}\sin(2)\right) && \approx 3.4190098
\end{aligned}
$$

To approximate the integral using a 10,000 node Neiderrieter scheme, execute the script

```
from numpy import exp, cos
from compecon import qnwequi
n, a, b = 10_000, [-1, -1], [1, 1]
(x1, x2), w = qnwequi(n,a,b,'N')
A = w.dot(exp(-x1) * cos(x2)**2)
```

```
A = 3.421441412
```

$$A = \int_{-1}^{1} \int_{-1}^{1} e^{-x_1} \cos^2(x_2) \, \mathrm{d}x_1 \, \mathrm{d}x_2 \approx 3.4190098$$

Nodes	Random	Neiderreiter	Weyl
10^3	-2.8	-2.9	-3.5
10^4	-2.5	-3.1	-3.9
10^5	-2.7	-4.0	-4.4
10^6	-3.2	-5.4	-5.7

Table 2: Log10 Approximation Errors for A, Alternative Equidistributed Sequences.

Numerical Differentiation

First-Order Derivatives

- The most natural way to approximate a derivative is to replace it with a finite difference:

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

- In theory, the approximation error disappears as h goes to 0, so if we pick h small enough, the error should be small.

- The approximation error can be bounded using Taylor's theorem, which states that

$$f(x+h) = f(x) + f'(x)h + O(h^2),$$

where $O(h^2)$ is proportional to the square of h.

- Rearranging,

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

since $O(h^2)/h = O(h)$, so the approximation error is proportional to $|h|$.

- However, there exist a more accurate finite difference approximation to the derivative of f at x.
- Consider the two second-order Taylor expansions

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + O(h^3)$$

$$f(x-h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} + O(h^3).$$

- Subtract the second expression from the first, rearrange, and divide by $2h$ to get

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

- We call
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
the centered finite difference approximation to the derivative of f at x.
- Its error $O(h^2)$ is one order more accurate than that of the one-sided finite difference approximation above.

- Since, in theory, finite difference approximation errors vanish as h approaches 0, one is tempted to make h as small as possible.
- Unfortunately, if h is made too small, rounding error can make the results meaningless.
- Consider the approximation error in the one- and two-sided finite difference derivatives of $\exp(x)$ at $x = 1$ as a function of the step size h.

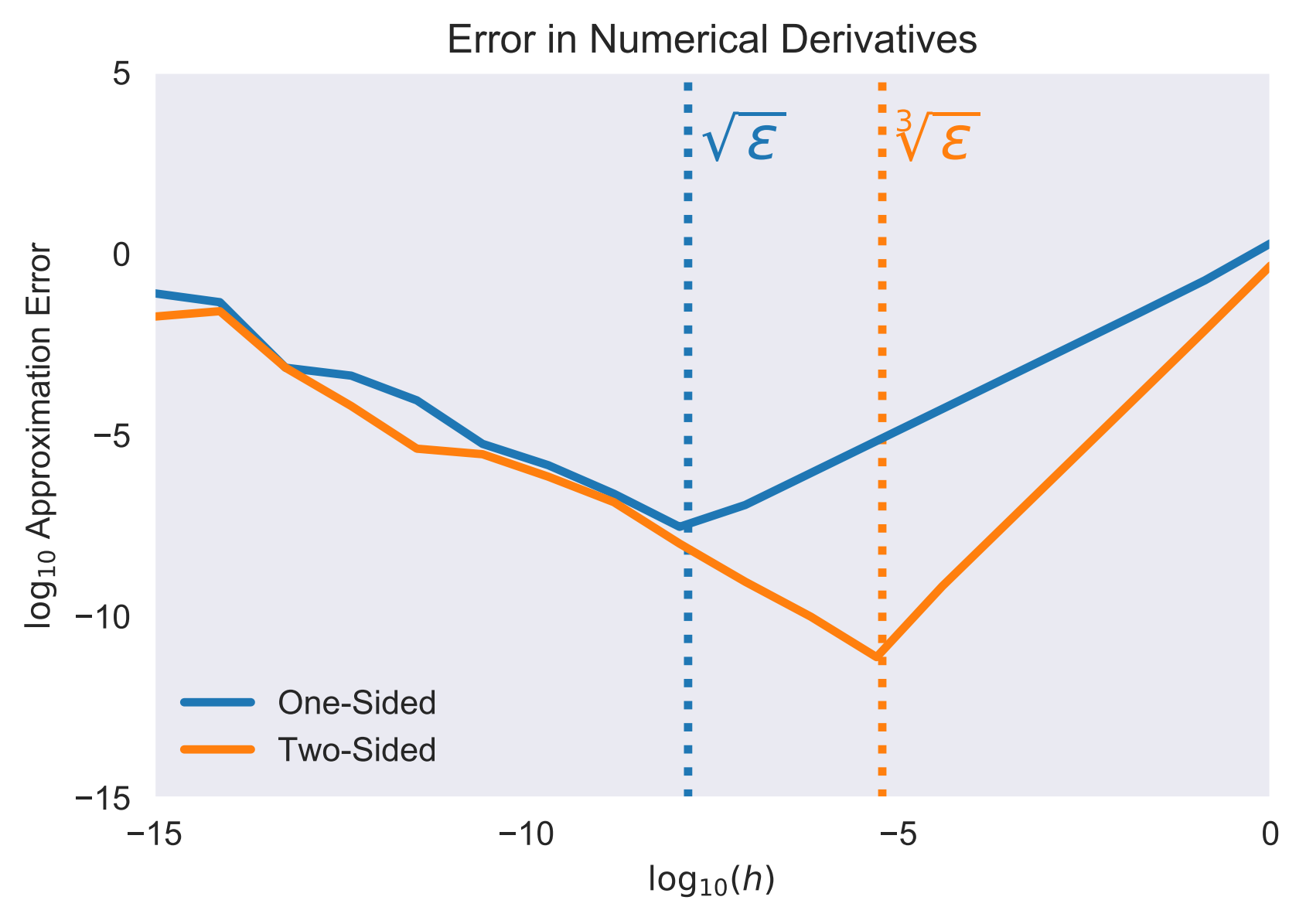


Figure 15: Approximation Error for One-Sided and Centered Finite Difference Derivatives of $\exp(x)$ at $x = 1$

- The centered finite difference approximation improves as h shrinks until it reaches cube root of machine precision $\sqrt[3]{\epsilon}$.
- Further reductions in h worsen approximation error because of rounding error.
- This suggests that we set $h \approx \sqrt[3]{\epsilon}$ relative to x for centered finite difference approximations.
- Similar empirical analysis suggests that we set $h \approx \sqrt{\epsilon}$ relative to x for one-sided finite difference approximations.

Higher-Order Derivatives

- Finite difference approximations for higher order derivatives can be found using a similar approach.
- For, example an order $O(h^2)$ centered finite difference approximation to the second derivative is

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

- To show the error is $O(h^2)$, add the two third-order Taylor expansions

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + O(h^4)$$

$$f(x-h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + O(h^4),$$

to get

$$f(x+h) + f(x-h) = 2f(x) + f''(x)h^2 + O(h^4),$$

rearrange, and divide by h^2.

CompEcon Utilities

- The CompEcon Toolbox utilities `jacobian` and `hessian` compute Jacobians and Hessians numerically.
- These were introduced earlier in the course.
- For convenience, we repeat examples of how to use them here.

- CompEcon utility `jacobian` computes the $m \times n$ finite difference Jacobian of an arbitrary function $f : \Re^n \mapsto \Re^m$.
- The calling protocol is

```
J = jacobian(f, #function of form fval=f(x)
             x) #evaluation point
```

- Output: `J` = Jacobian of `f` at `x`

Example 13:

Computing a Jacobian

- The exact Jacobian of

$$f(x_1, x_2) = \begin{bmatrix} \exp(x_1) - x_2 \\ x_1 + x_2^2 \\ (1 - x_1)\log(x_2) \end{bmatrix}$$

at $(0, 1)$ is

$$f'(x_1, x_2) = \begin{bmatrix} 1 & -1 \\ 1 & 2 \\ 0 & 1 \end{bmatrix}$$

To compute the Jacobian numerically, execute the script

```
def f(x):
    x1, x2 = x
    y = [np.exp(x1)-x2,
         x1 + x2**2,
         (1-x1)*np.log(x2)]
    return np.array(y)

np.set_printoptions(precision=15)
print(jacobian(f,np.array([0,1])))
```

This should return

```
[[ 1.000000000014386 -1.                ]
 [ 0.999999999996052  1.999999999990833]
 [ 0.                 1.000000000012223]]
```

- CompEcon utility `hessian` computes the $n \times n$ finite difference Hessian of an arbitrary function $f : \Re^n \mapsto \Re$.
- The calling protocol is

```
H = hessian(f, #function of form fval=f(x)
            x) #evaluation point
```

- Output: `H` = Hessian of `f` at `x`