OSE scientific computing

Prof. Dr. Philipp Eisenhauer

Dec 14, 2021

CONTENTS

> The purpose of (scientific) computing is insight, not numbers.
>
> —Richard Hamming.

The sound analysis of computational economic models requires expertise in economics, statistics, numerical methods, and software engineering. We first provide an overview of basic numerical methods for optimization, numerical integration, approximation methods, and uncertainty quantification. We then deepen our understanding of each of these topics in the context of a dynamic model of human capital accumulation using respy. We conclude by showcasing basic software engineering practices such as the design of a collaborative and reproducible development workflow, automated testing, and high-performance computing.

Students learn how to use Python for advanced scientific computing. They acquire a toolkit of numerical methods frequently needed for the analysis of computational economic models, obtain an overview of basic software engineering tools such as GitHub and pytest, and are exposed to high-performance computing using multiprocessing and mpi4py.

Guest lectures organized by institutions from the public and private sector are an integral part of our curriculum. These events connect students directly with employment opportunities that match their interests and skill set and provide students with insights into scientific computing applications in a variety of settings.

We build the course on the Nuvolos.cloud as an integrated research and teaching platform. The platform provides a simple, browser-based environment that allows for complete control over students' computational environment and simplifies the dissemination of teaching material. It enables students to seamlessly scale up their course projects from a prototype to a high-performance computing environment.

CHAPTER

ONE

LABS

Throughout the course we will make heavy use of Python and its SciPy ecosystem. We provide a set of learning labs as Jupyter Notebooks.

1.1 Tooling

We showcase the basics of Python programming and point students to useful resources to study further. There are numerous excellent introductory lectures on Python programming in economics available online. Among them is Python programming for economics and finance and we will sample some of their material for our labs.

1.2 Linear equations

We explore different solution algorithms to solve linear equations. We look at direct methods building on L-U decomposition as well as iterative methods. We study the impact of ill-conditioned matrices on the performance of algorithms. In the process, we learn some basic ideas behind testing and benchmarking numerical routines.

1.2.1 Linear Equations

Outline

1. Setup
2. Special cases
3. L-U Factorization
4. Pivoting
5. Benchmarking
6. Ill conditioning
7. Iterative methods
8. Resources

Setup

The linear equation is the most elementary problem that arises in computational economic analysis. In a linear equation, an $n \times n$ matrix A and an n-vector b are given, and one must compute the n-vector x that satisfies $Ax = b$.

Linear equations arise naturally in many economic applications such as

- Linear multicommodity market equilibrium models
- Finite-state financial market models
- Markov chain models
- Ordinary least squares

They more commonly arise indirectly from numerical solution to nonlinear and functional equations:

- Nonlinear multicommodity market models
- Multiperson static game models
- Dynamic optimization models
- Rational expectations models

Applications often require the repeated solution of very large linear equation systems. In these situations, issues regarding speed, storage requirements, and preciseness of the solution of such equations can arise.

[1]:
```
import pandas as pd
import numpy as np

from linear_algorithms import backward_substitution
from linear_algorithms import forward_substitution
from linear_algorithms import gauss_seidel
from linear_algorithms import solve

from linear_plots import plot_operation_count

from temfpy.linear_equations import get_ill_cond_lin_eq
from linear_problems import get_inverse_demand_problem
```

Special cases

We can start with some special cases to develop a basic understanding for the core building blocks for more complicated settings. Let's start with the case of a lower triangular matrix A, where we can solve the linear equation by a simple backward or forward substitution. Let's consider the following setup.

$$A = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Consider an algorithmic implementation of forward-substitution as an example.

$$x_i = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) / a_{ii}$$

[2]:
```
??forward_substitution
```

[3]:
```
def test_problem():
    A = np.tril(np.random.normal(size=(3, 3)))
    x = np.random.normal(size=3)
    b = np.matmul(A, x)
    return A, b, x


for _ in range(10):
    A, b, x_true = test_problem()
    x_solve = forward_substitution(A, b)
    np.testing.assert_almost_equal(x_solve, x_true)
```

Questions

- How can we make the test code more generic and sample test problems of different dimensions?
- Is there a way to control the randomness in the test function?
- Is there software out there that allows to automate parts of the testing?

If we have an upper triangular matrix, we can use backward substitution to solve the linear system

[4]:
```
??backward_substitution
```

Exercise

- Implement the same testing setup as above the backward-substitution function.

We can now build on these two functions to tackle more complex tasks. This is a good example on how to develop scientific software step-by-step ensuring that each component is well tested before integrating into more involved settings.

L-U Factorization

Most linear equations encountered in practice, however, do not have a triangular A matrix. Doolittle and Crout have shown that any matrix A can be decomposed into the product of a (row-permuted) lower and upper triangular matrix L and U, respectively $A = L \times U$ using **Gaussian elimination**. We will not look into the Gaussian elimination algorithm, but there is an example application in our textbook where you can follow along step by step. The L-U algorithm is designed to decompose the A matrix into the product of lower and upper triangular matrices, allowing the linear equation to be solved using a combination of backward and forward substitution.

Here are the two core steps:

- Factorization phase

$$A = LU$$

- Solution phase:

$$Ax = (LU)x = L(Ux) = b,$$

where we solve $Ly = b$ using forward-substitution and $Ux = y$ using backward-substitution.

Adding to this the two building blocks we developed earlier `forward_substitution` and `backward_substitution`, we can now write a quite generic function to solve systems of linear equations.

[5]:
```
??solve
```

Let's see if this is actually working.

[6]:
```
A = np.array([[3, 1], [1, 2]])
x_true = np.array([9, 8])
b = A @ x_true
x_solve = solve(A, b)
np.testing.assert_almost_equal(x_true, x_solve)
```

Pivoting

Rounding error can cause serious error when solving linear equations. Let's consider the following example, where ϵ is a tiny number.

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

It is easy to verify that the right solution is

$$x_1 = \frac{1}{1-\epsilon}$$
$$x_2 = \frac{1-2\epsilon}{1-\epsilon}$$

and thus x_1 is slightly more than one and x_2 is slightly less than one. To solve the system using Gaussian elimination we need to add $-1/\epsilon$ times the first row to the second row. We end up with

$$\begin{bmatrix} \epsilon & 1 \\ 0 & 1-\frac{1}{\epsilon} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2-\frac{1}{\epsilon} \end{bmatrix},$$

which we can then solve recursively.

$$x_2 = \frac{2-1/\epsilon}{1-1/\epsilon}$$
$$x_1 = \frac{1-x_2}{\epsilon}$$

Let's translate this into code.

[7]:
```
eps = 1e-17
A = np.array([[eps, 1], [1, 1]])
b = np.array([1, 2])
```

We can now use our solution algorithm.

[8]:
```
solve(A, b)
```

[8]:
```
array([0., 1.])
```

But we have to realize that the results are grossly inaccurate. What happened? Is there any hope to apply `numpy`'s routine?

[9]:
```
np.linalg.solve(A, b)
```

[9]:
```
array([1., 1.])
```

This algorithm does automatically check whether such rounding errors can be avoided by simply changing the order of rows. This is called **pivoting** and changes the recursive solution to

$$x_2 = \frac{1 - 2\epsilon}{1 - \epsilon}$$
$$x_1 = 2 - x_2$$

which can be solved more accurately. Our implementation also solves the modified problem well.

[10]:
```
A = np.array([[1, 1], [eps, 1]])
b = np.array([2, 1])
solve(A, b)
```

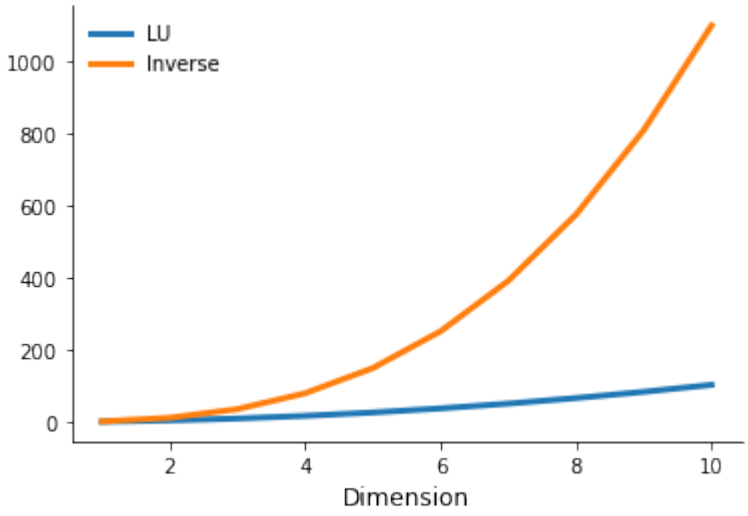
[10]:
```
array([1., 1.])
```

Building your own numerical routines is usually the only way to really understand the algorithms and learn about all the potential pitfalls. However, the default should be to rely on battle-tested production code. For linear algebra there are numerous well established libraries available. Building your own numerical routines is usually the only way to really understand the algorithms and learn about all the potential pitfalls. However, the default should be to rely on battle-tested production code. For linear algebra there are numerous well established libraries available.

Benchmarking

How does solving a system of linear equations by an $L - U$ decomposition compare to other alternatives of solving the system of linear equations.

[11]:
```
plot_operation_count()
```



The right setup for your numerical needs depends on your particular problem. For example, this trade-off looks very different if you have to solve numerous linear equations that only differ in b but not A. In this case you only need to compute the inverse once.

Exercise

- Set up a benchmarking exercise that compares the time to solution for the two approaches for $m = \{1, 100\}$ and $n = \{50, 100\}$, where n denotes the number of linear equations and m the number of repeated solutions.

Ill Conditioning

Some linear equations are inherently difficult to solve accurately on a computer. This difficulty occurs when the A matrix is structured in such a way that a small perturbation δb in the data vector b induces a large change δx in the solution vector x. In such cases the linear equation or, more generally, the A matrix is said to be **ill conditioned**.

One measure of ill conditioning in a linear equation Ax = b is the "elasticity" of the solution vector x with respect to the data vector b

$$\epsilon = \sup_{||\delta b||>0} \frac{||\delta x||/||x||}{||\delta b||/||b||}$$

The elasticity gives the maximum percentage change in the size of the solution vector x induced by a 1 percent change in the size of the data vector b. If the elasticity is large, then small errors in the computer representation of the data vector b can produce large errors in the computed solution vector x. Equivalently, the computed solution x will have far fewer significant digits than the data vector b.

In practice, the elasticity is estimated using the condition number of the matrix A, which for invertible A is defined by $\kappa \equiv ||A|| \cdot ||A^{-1}||$. The condition number is always greater than or equal to one. Numerical analysts often use the rough rule of thumb that for each power of 10 in the condition number, one significant digit is lost in the computed solution vector x. Thus, if A has a condition number of $1,000$, the computed solution vector x will have about three fewer significant digits than the data vector b.

Let's look at an example, where the solution vector is all ones but but the linear equation is notoriously ill-conditioned.

[12]:
```
??get_ill_cond_lin_eq
```

How does the solution error depend on the condition number in this setting.

[13]:
```
rslt = dict((("Condition", []), ("Error", []), ("Dimension", [])))
grid = [5, 10, 15, 25]

for n in grid:
    A, b, x_true = get_ill_cond_lin_eq(n)
    x_solve = np.linalg.solve(A, b)
    rslt["Condition"].append(np.linalg.cond(A))
    rslt["Error"].append(np.linalg.norm(x_solve - x_true, 1))
    rslt["Dimension"].append(n)
```

[14]:
```
pd.DataFrame.from_dict(rslt)
```

[14]:

	Condition	Error	Dimension
0	2.616969e+04	1.278000e+03	5
1	2.106258e+12	1.762345e+09	10
2	2.582411e+21	4.947153e+16	15
3	2.035776e+22	2.383979e+20	25

There is a more general lesson here. Always be skeptical about the quality of your numerical results. See these two papers for an exploratory analysis of econometric software packages and nonliner optimization. Yes, they are rather old and much progress has been made, but the general points remain valid.

- McCullough, B. D., and Vinod, H. D. (2003). Verifying the Solution from a Nonlinear Solver: A Case Study. *American Economic Review*, 93 (3): 873-892.
- McCullough, B., D., and Vinod H. D. (1999). The Numerical Reliability of Econometric Software. *Journal of Economic Literature*, 37 (2): 633-665.

Exercise

Let's consider the following example as well, which is taken from Johansson (2015, p.131).

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & \sqrt{p} \\ 1 & \frac{1}{\sqrt{p}} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

This system is singular for $p = 1$ and for p in the vicinity of one is ill-conditioned.

- Create two plots that show the condition number and the error between the analytic and numerical solution.

Iterative methods

Algorithms based on Gaussian elimination are called exact or, more properly, direct methods because they would generate exact solutions for the linear equation $Ax = b$ after a finite number of operations, if not for rounding error. Such methods are ideal for moderately sized linear equations but may be impractical for large ones. Other methods, called iterative methods, can often be used to solve large linear equations more efficiently if the A matrix is sparse, that is, if A is composed mostly of zero entries. Iterative methods are designed to generate a sequence of increasingly accurate approximations to the solution of a linear equation, but they generally do not yield an exact solution after a prescribed number of steps, even in theory.

The most widely used iterative methods for solving a linear equation $Ax = b$ are developed by choosing an easily invertible matrix Q and writing the linear equation in the equivalent form

$$Qx = b + (Q - A)x$$

or

$$x = Q^{-1}b + (I - Q^{-1}A)x$$

This form of the linear equation suggests the iteration rule

$$x^{k+1} \leftarrow Q^{-1}b + (I - Q^{-1}A)x^k$$

which, if convergent, must converge to a solution of the linear equation. Ideally, the so-called splitting matrix Q will satisfy two criteria. First, $Q^{-1}b$ and $Q^{-1}A$ should be relatively easy to compute. This criterion is met if Q is either diagonal or triangular. There are two popular approaches:

- The **Gauss-Seidel** method sets Q equal to the upper triangular matrix formed from the upper triangular elements of A.
- The **Gauss-Jacobi** method sets Q equal to the diagonal matrix formed from the diagonal entries of A.

[15]:
```
??gauss_seidel
```

Exercise

- Implement the Gauss-Jacobi method.

[16]:
```
from linear_solutions_tests import gauss_jacobi  # noqa: E402
```

Let's conclude with an economic application as outlined in Judd (1998). Suppose we have the following inverse demand function $p = 10 - q$ and the following supply curve $q = p/2 + 1$. Equilibrium is where supply equals demand and thus we need to solve the following linear system.

$$\begin{bmatrix} 10 \\ -2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix} \times \begin{bmatrix} p \\ q \end{bmatrix}$$

[17]:
```
A, b, x_true = get_inverse_demand_problem()
```

Now we can compre the two solution approaches and make sure that they in fact give the same result.

[18]:
```
x_seidel = gauss_seidel(A, b)
x_jacobi = gauss_jacobi(A, b)
np.testing.assert_almost_equal(x_seidel, x_jacobi)
```

Resources

- **The PARDISO Solver Project**: https://www.pardiso-project.org
- **LAPACK—Linear Algebra PACKage**: http://www.netlib.org/lapack
- Robert Johansson. *Numerical Python: scientific computing and data science applications with NumPy, SciPy and Matplotlib*. Apress, 2018.
- William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. *Numerical recipes: The art of scientific computing*. Cambridge University Press, 1986.

[ ]:
```

```

1.2.2 Functions

This module contains the algorithms for the linear equations lab.

The materials follow Miranda and Fackler (2004, [MF04]) (Chapter 2). The python code heavily draws on Romero-Aguilar (2020, [RA20]) and Foster (2019, [Fos19]).

`labs.linear_equations.linear_algorithms.`**`backward_substitution`**`(`a, b`)`

Perform backward substitution to solve a system of linear equations.

Solves a linear equation of type $Ax = b$ when for an *upper triangular* matrix A of dimension $n \times n$ and vector b of length n.

Parameters

- **a** (*numpy.ndarray*) – Lower triangular matrix of dimension $n \times n$.
- **b** (*numpy.ndarray*) – Vector of length n.

Returns **x** – Solution of the linear equations. Vector of length n.

Return type numpy.ndarray

labs.linear_equations.linear_algorithms.**forward_substitution**(a, b)

Perform forward substitution to solve a system of linear equations.

Solves a linear equation of type $Ax = b$ when for a *lower triangular* matrix A of dimension $n \times n$ and vector b of length n. The forward subsititution algorithm can be represented as:

$$x_i = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) / a_{ii}$$

Parameters

- **a** (*numpy.ndarray*) – Lower triangular matrix of dimension $n \times n$.
- **b** (*numpy.ndarray*) – Vector of length n.

Returns **x** – Solution of the linear equations. Vector of length n.

Return type numpy.ndarray

labs.linear_equations.linear_algorithms.**gauss_seidel**(*a*, *b*, *x0=None*, *lambda_=1.0*, *max_iterations=1000*, *tolerance=1.4901161193847656e-08*)

Solves linear equation of type $Ax = b$ using Gauss-Seidel iterations.

In the linear equation, A denotes a matrix of dimension $n \times n$ and b denotes a vector of length n The solution method performs especially well for larger linear equations if matrix :math`A`is sparse. The method achieves fairly precise approximations to the solution but generally does not produce *exact* solutions.

Following the notation in Miranda and Fackler (2004, [MF04]), the linear equations problem can be written as

$$Qx = b + (Q - A)x \Rightarrow x = Q^{-1}b + (I - Q^{-1}A)x$$

which suggest the iteration rule

$$x^{(k+1)} \leftarrow Q^{-1}b + (I - Q^{-1}A)x^{(k)}$$

which, if convergent, must converge to a solution of the linear equation. For the Gauss-Seidel method, Q is the upper triangular matrix formed from the upper triangular elements of A.

Parameters

- **a** (*numpy.ndarray*) – Matrix of dimension $n \times n$
- **b** (*numpy.ndarray*) – Vector of length n.
- **x0** (*numpy.ndarray, default None*) – Array of starting values. Set to be if None.
- **lambda** (*float*) – Over-relaxation parameter which may accelerate convergence of the algorithm for $1 < \lambda < 2$.
- **max_iterations** (*int*) – Maximum number of iterations.
- **tolerance** (*float*) – Convergence tolerance.

Returns **x** – Solution of the linear equations. Vector of length n.

Return type numpy.ndarray

Raises **StopIteration** – If maximum number of iterations specified by *max_iterations* is reached.

labs.linear_equations.linear_algorithms.**naive_lu**(a)

Apply a naive LU factorization.

LU factorization decomposes a matrix A into a lower triangular matrix L and upper triangular matrix U. The naive LU factorization does not apply permutations to the resulting matrices and thus only works reliably for diagonal matrices A:

Parameters **a** (*numpy.ndarray*) – Diagonal square matrix.

Returns

- **l** (*numpy.ndarray*)
- **u** (*numpy.ndarray*)

labs.linear_equations.linear_algorithms.**solve**(a, b)

Solve linear equations using L-U factorization.

Solves a linear equation of type $Ax = b$ when for a nonsingular square matrix A of dimension $n \times n$ and vector b of length n. Decomposes Algorithm decomposes matrix A into the product of lower and upper triangular matrices. The linear equations can then be solved using a combination of forward and backward substitution.

Two stages of the L-U algorithm:

1. Factorization using Gaussian elimination: $A = LU$ where L denotes a row-permuted lower triangular matrix. U denotes a row-permuted upper triangular matrix.

2. Solution using forward and backward substitution. The factored linear equation of step 1 can be expressed as

$$Ax = (LU)x = L(Ux) = b$$

The forward substitution algorithm solves $Ly = b$ for y. The backward substitution algorithm then solves $Ux = y$ for x.

Parameters

- **a** (*numpy.ndarray*) – Matrix of dimension $n \times n$
- **b** (*numpy.ndarray*) – Vector of length n.

Returns **x** – Solution of the linear equations. Vector of length n.

Return type numpy.ndarray

Example

```
>>> b = np.array([1, 2, 3])
>>> a = np.array([[4, 0, 0], [0, 2, 0], [0, 0, 2]])
>>> solve(a, b)
array([0.25, 1.  , 1.5 ])
```

1.3 Nonlinear equations

We explore different solution algorithms to solve nonlinear equations. We start with the bisection method. We then turn to function iteration before exploring Newton's method for nonlinear equations. Finally, we look at Quasi-Newton methods and benchmark their performance in solving a standard Cournot problem. We briefly discuss some criterion to choose the right algorithm for the problem at hand.

1.3.1 Nonlinear Equations

Outline

1. Setup
2. Bisection method
3. Function iteration
4. Newton's method
5. Quasi-Newton method
6. Choosing a solution method
7. Convergence
8. Resources

Setup

The nonlinear equation takes one of two forms.

- **Rootfinding problem** Given a function $f : R^n \mapsto R^n$,compute an n-vector x^*, called root of f, such that $f(x^*) = 0$.
- **Fixed-point problem** Given a function $g : R^n \mapsto R^n$, compute an n-vector x^* called fixed point of g such that $g(x^*) = x^*$

The two forms are equivalent.

- A root of f is a fixed-point of $g(x) = x - f(x)$.
- A fixed-point of g is a root ot $f(x) = x - g(x)$.

Nonlinear equations arise naturally in economics:

- Multicommodity market equilibrium models
- Multiperson static game models
- Unconstrained optimization models

Nonlinear equations also arise indirectly when numerically solving economic models involving functional equations:

- Dynamic optimization models
- Rational expectations models
- Arbitrage pricing models

Building on our earlier lab on the solution to linear systems of equations, often the solution to a nonlinear problem is computed iteratively by solving a sequence of linear problems.

- sensitivity to intial condition

- susceptible to rounding error

[2]:
```
# from temfpy.nonlinear_equations import exponential  # noqa: F401
from scipy import optimize
import pandas as pd
import numpy as np

from nonlinear_algorithms import bisect
from nonlinear_plots import plot_bisection_test_function

from nonlinear_algorithms import fixpoint
from nonlinear_plots import plot_fixpoint_example
from nonlinear_plots import plot_newton_pathological_example
from nonlinear_plots import plot_convergence
from nonlinear_plots import plot_newtons_method
from nonlinear_plots import plot_secant_method

from nonlinear_algorithms import newton_method

from nonlinear_problems import bisection_test_function
from nonlinear_problems import function_iteration_test_function
from nonlinear_problems import newton_pathological_example
```

Bisection method

We start with the implementation (and testing) of the bisection algorithm.

- The bisection method is perhaps the simplest and most robust method for computing the root of a continuous real-valued function defined on a bounded interval of the real line. The bisection method is based on the Intermediate Value Theorem, which asserts that if a continuous real-valued function defined on an interval assumes two distinct values, then it must assume all values in between. In particular, if f is continuous, and $f(a)$ and $f(b)$ have different signs, then f must have at least one root x in $[a, b]$.
- Each iteration begins with an interval known to contain or to bracket a root of f, because the function has different signs at the interval endpoints. The interval is bisected into two subintervals of equal length. One of the two subintervals must have endpoints of different signs and thus must contain a root of f. This subinterval is taken as the new interval with which to begin the subsequent iteration. In this manner, a sequence of intervals is generated, each half the width of the preceding one, and each known to contain a root of f. The process continues until the width of the bracketing interval containing a root shrinks below an acceptable convergence tolerance.

[3]:
```
??bisect
```

```
Signature: bisect(f, a, b, tolerance=1.5e-08)
Source:
def bisect(f, a, b, tolerance=1.5e-8):
    """Apply bisect method to root finding problem.

    Iterative procedure to find the root of a continuous real-values function :math:
↪`f(x)` defined
    on a bounded interval of the real line. Define interval :math:`[a, b]` that is known␣
↪to contain
    or bracket the root of :math:`f` (i.e. the signs of :math:`f(a)` and :math:`f(b)`␣
↪must differ).
    The given interval :math:`[a, b]` is then repeatedly bisected into subintervals of␣
↪equal length.
```

(continues on next page)

(continued from previous page)

```
    Each iteration, one of the two subintervals has endpoints of different signs (thus␣
↪containing
    the root of :math:`f`) and is again bisected until the size of the subinterval␣
↪containing the
    root reaches a specified convergence tolerance.

    Parameters
    ----------
    f : callable
        Continuous, real-valued, univariate function :math:`f(x)`.
    a : int or float
        Lower bound :math:`a` for :math:`x \\in [a,b]`.
    b : int or float
        Upper bound :math:`a` for :math:`x \\in [a,b]`. Select :math:`a` and :math:`b` so
        that :math:`f(b)` has different sign than :math:`f(a)`.
    tolerance : float
        Convergence tolerance.

    Returns
    -------
    x : float
        Solution to the root finding problem within specified tolerance.

    Examples
    --------
    >>> x = bisect(f=lambda x : x ** 3 - 2, a=1, b=2)[0]
    >>> round(x, 4)
    1.2599

    """
    # Get sign for f(a).
    s = np.sign(f(a))

    # Get staring values for x and interval length.
    x = (a + b) / 2
    d = (b - a) / 2

    # Continue operation as long as d is above the convergence tolerance threshold.
    # Update x by adding or subtracting value of d depending on sign of f.
    xvals = [x]

    while d > tolerance:
        d = d / 2
        if s == np.sign(f(x)):
            x += d
        else:
            x -= d

        xvals.append(x)

    return x, np.array(xvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_algorithms.py
```

(continues on next page)

(continued from previous page)

```
Type:      function
```

Let's visualize a test function to get a sense of what result can should expect.

[4]:
```
??bisection_test_function
```

```
Signature: bisection_test_function(x)
Source:
def bisection_test_function(x):
    """Get test function for bisection."""
    return x ** 3 - 2
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_problems.py
Type:      function
```

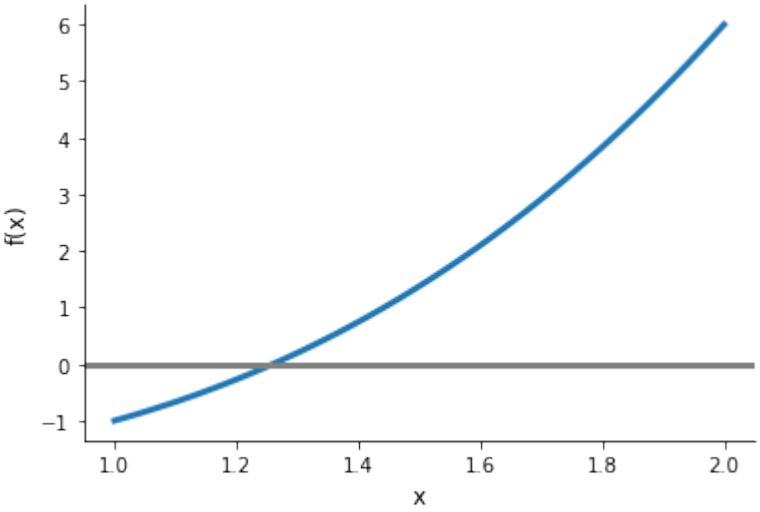
[5]:
```
??plot_bisection_test_function
```

```
Signature: plot_bisection_test_function(f)
Source:
def plot_bisection_test_function(f):
    """Plot bisect example."""
    fig, ax = plt.subplots()
    grid = np.linspace(1, 2)

    values = []
    for value in grid:
        values.append(f(value))
    ax.plot(grid, values)

    ax.axes.axhline(0, color="grey")
    ax.set_ylabel("f(x)")
    ax.set_xlabel("x")
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_plots.py
Type:      function
```

- The ability to vectorize functions using np.vectorize comes in very handy when evaluating functions over a grid. However, be aware that this is a simple replacement of a `for loop` and thus does not improve performance.

We are ready to go ...

[6]:
```
plot_bisection_test_function(bisection_test_function)
```



Now we are ready to check our implementation and investigate the sensitivity of results to alternative tuning parameter.

[26]:
```
lower, upper = 1, 2
x, xvals = bisect(bisection_test_function, lower, upper)
print(f"The root of our test function is {x:3.2f}.")
```

```
The root of our test function is 1.26.
```

Exercises

1. Write a short test that ensures that in fact found a root of the function.
2. Create a simple plot that shows each iterate of x and label the two axis appropriately.

Function iteration

- Function iteration is a relatively simple technique that may be used to compute a fixed point $x = g(x)$ of a function from R^n to R^n. The technique is also applicable to a rootfinding problem $f(x) = 0$ by recasting it as the equivalent fixed-point problem $g(x) = x - f(x)$. Function iteration begins with the analyst supplying a guess $x^{(0)}$ for the fixed point of g. Subsequent iterates are generated using the simple iteration rule

$$x^{(k+1)} \leftarrow g(x^{(k)})$$

Since g is continuous, if the iterates converge, they converge to a fixed point of g.

[8]:
```
??fixpoint
```

```
Signature: fixpoint(f, x0, tolerance=0.0001)
Source:
def fixpoint(f, x0, tolerance=10e-5):
    """Compute fixed point using function iteration.

    Parameters
    ----------
    f : callable
```

(continues on next page)

(continued from previous page)

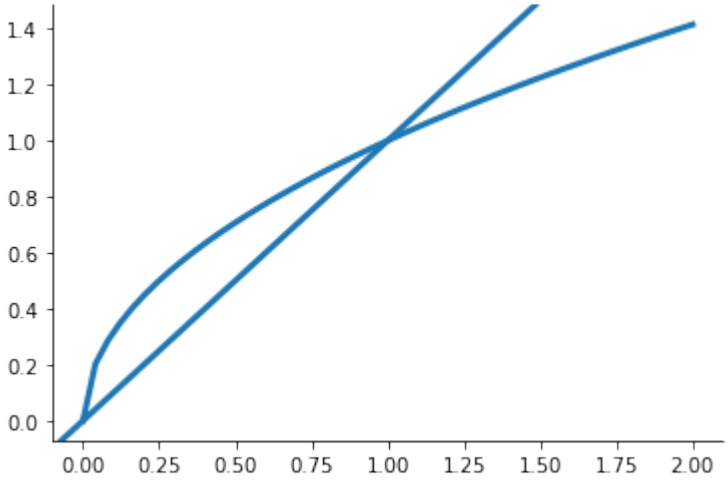
```
        Function :math:`f(x)`.
    x0 : float
        Initial guess for fixed point (starting value for function iteration).
    tolerance : float
        Convergence tolerance (tolerance < 1).

    Returns
    -------
    x : float
        Solution of function iteration.

    Examples
    --------
    >>> import numpy as np
    >>> x = fixpoint(f=lambda x : x**0.5, x0=0.4, tolerance=1e-10)[0]
    >>> np.allclose(x, 1)
    True

    """
    e = 1
    xvals = [x0]

    while e > tolerance:
        # Fixed point equation.
        x = f(x0)
        # Error at the current step.
        e = np.linalg.norm(x0 - x)
        x0 = x
        xvals.append(x0)
    return x, np.array(xvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_algorithms.py
Type:      function
```

Let's visualize a test function to get a sense of what result can should expect.

[9]:
```
??function_iteration_test_function
```

```
Signature: function_iteration_test_function(x)
Source:
def function_iteration_test_function(x):
    """Get test function for function iteration."""
    return np.sqrt(x)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_problems.py
Type:      function
```

[10]:
```
plot_fixpoint_example(function_iteration_test_function)
```



Let's run and test our implementation.

[28]:
```
x = fixpoint(function_iteration_test_function, 0.4, tolerance=1e-5)[0]

try:
    np.testing.assert_almost_equal(function_iteration_test_function(x), x)
except AssertionError as msg:
    print(msg)
```

```
Arrays are not almost equal to 7 decimals
 ACTUAL: 0.9999965046343594
 DESIRED: 0.9999930092809364
```

- This is an example of using a try-except block to explicitly handle the `AssertionError`. Details on exception handling in Python, see this Wiki for some more examples.

We set two parameters manually, which one is it? How about choosing different starting values?

[33]:
```
for x0 in np.linspace(0.1, 0.9):
    x = fixpoint(function_iteration_test_function, x0, tolerance=1e-5)[0]
    try:
        np.testing.assert_almost_equal(function_iteration_test_function(x), x)
        print("Success for x0 {x0}")
    except AssertionError:
        pass
```

Let's look at the `tolerance` setting instead.

Exercises

1. Find the fixpoint of $g(x) = \sqrt{x + 0.2}$.
2. How many iterations does the function iteration need? Does it depend on the starting value?

Newton's method

Newton's method is an algorithm for computing the root of a function $f : R^n \mapsto R^n$. Newton's method employs a strategy of successive linearization. The strategy calls for the nonlinear function f to be approximated by a sequence of linear functions whose roots are easily computed and, ideally, converge to the root of f. In particular, the $k + 1^{th}$ iterate

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

is the root of the Taylor linear approximation of f around the preceding iterate x_k.

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k).$$

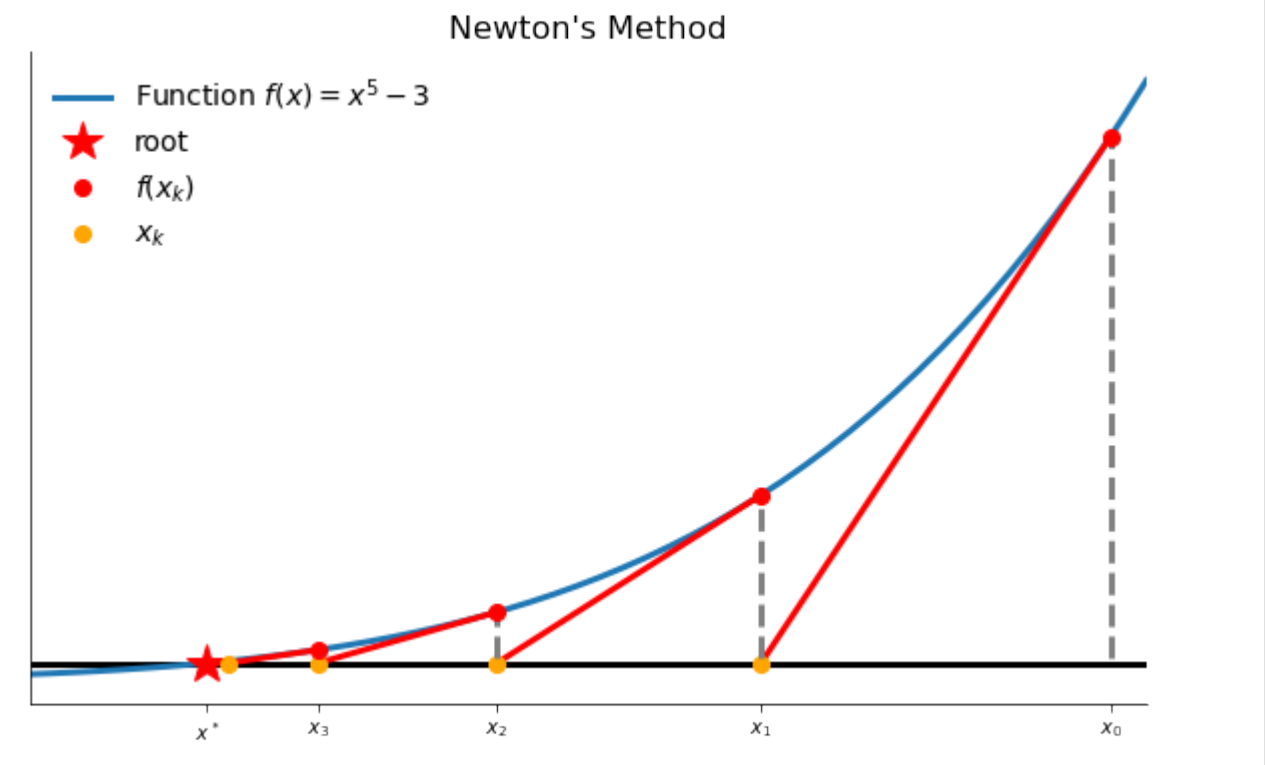
This yields the following iteration rule:

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

If $n = 1$, the iteration rule takes the simpler form

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

[13]:
```
plot_newtons_method()
```



Exercises

1. Plot the function $f(x) = x^4 - 2$ and provide a rough estimate of its fix point.
2. Let's compute the root of the function using Newton's method by employing the proper iteration rule:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(c_k)} = x_k - \frac{x_k^4 - 2}{4x_k^3}$$

We now look at a more general implementation.

[14]:
```
??newton_method
```

```
Signature: newton_method(f, x0, tolerance=1.5e-08)
Source:
def newton_method(f, x0, tolerance=1.5e-8):
    """Apply Newton's method to solving nonlinear equation.

    Solve equation using successive linearization, which replaces the nonlinear problem
    by a sequence of linear problems whose solutions converge to the solution of the␣
↪nonlinear
    problem.

    Parameters
    ----------
    f : callable
        (Univariate) function :math:`f(x)`.
    x0 : float
        Initial guess for the root of :math:`f`.
    tolerance : float
        Convergence tolerance.

    Returns
    -------
    xn : float
        Solution of function iteration.

    """
    x0 = np.atleast_1d(x0)

    # This is tailored to the univariate case.
    assert x0.shape[0] == 1

    xn = x0.copy()

    while True:
        fxn, gxn = f(xn)
        if np.linalg.norm(fxn) < tolerance:
            return xn
        else:
            xn = xn - fxn / gxn
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_algorithms.py
Type:      function
```

Question

- What are the important generalizations?

Let' explore one last test function to test the interface of our function and learn something about private functions in the process.

$$f(x) = x^3 - 2$$

[36]:
```
def _jacobian(x):
    return 3 * x ** 2


def _value(x):
    return x ** 3 - 2


def f(x):
    return _value(x), _jacobian(x)


x = newton_method(f, 0.4)
np.testing.assert_almost_equal(f(x)[0], 0)
```

A potential shortcoming of Newton's method is that the derivatives required for the Jacobian may not be available may be difficult to calculate analytically, or time-consuming to approximate numerically ... or that it might actually fail or result in cycles.

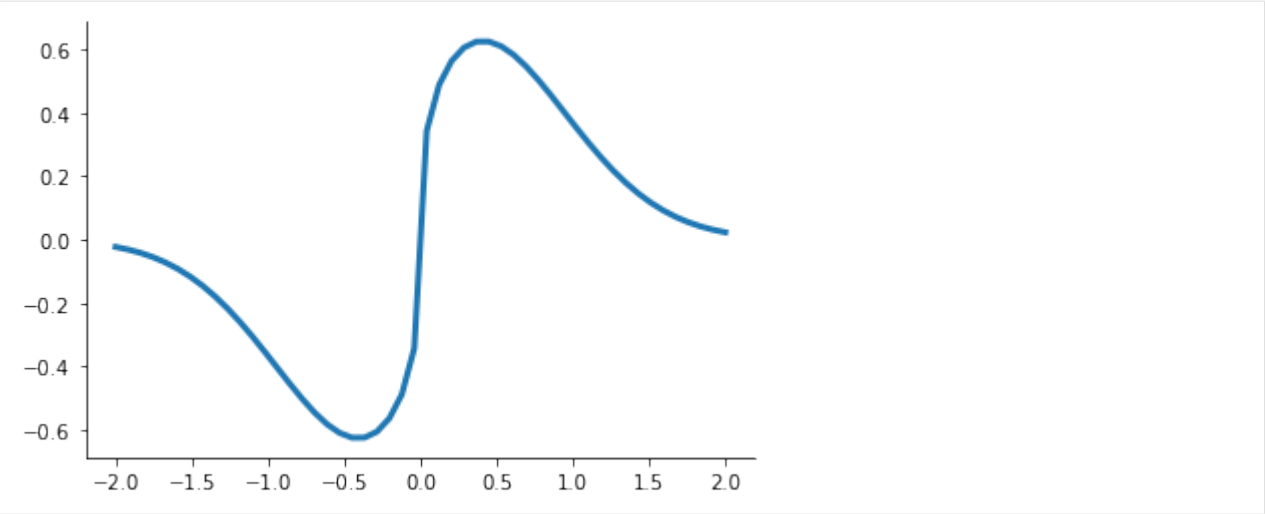
Consider the following pathological example which has a unique root at 0.

$$f(x) = x^{\frac{1}{3}} e^{-x^2}$$

[37]:
```
??newton_pathological_example
```

```
Signature: newton_pathological_example(x)
Source:
def newton_pathological_example(x):
    """Get Newton Pathological example."""
    fval = newton_pathological_example_fval(x)
    fjac = newton_pathological_example_fjac(x, newton_pathological_example_fval)
    return fval, fjac
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/nonlinear_equations/nonlinear_problems.py
Type:      function
```

[17]:
```
plot_newton_pathological_example(newton_pathological_example)
```



We can derive the Newton iterates as:

$$x_{k+1} = x_k \left(1 - \frac{3}{1 - 6x_k^2}\right)$$

Questions

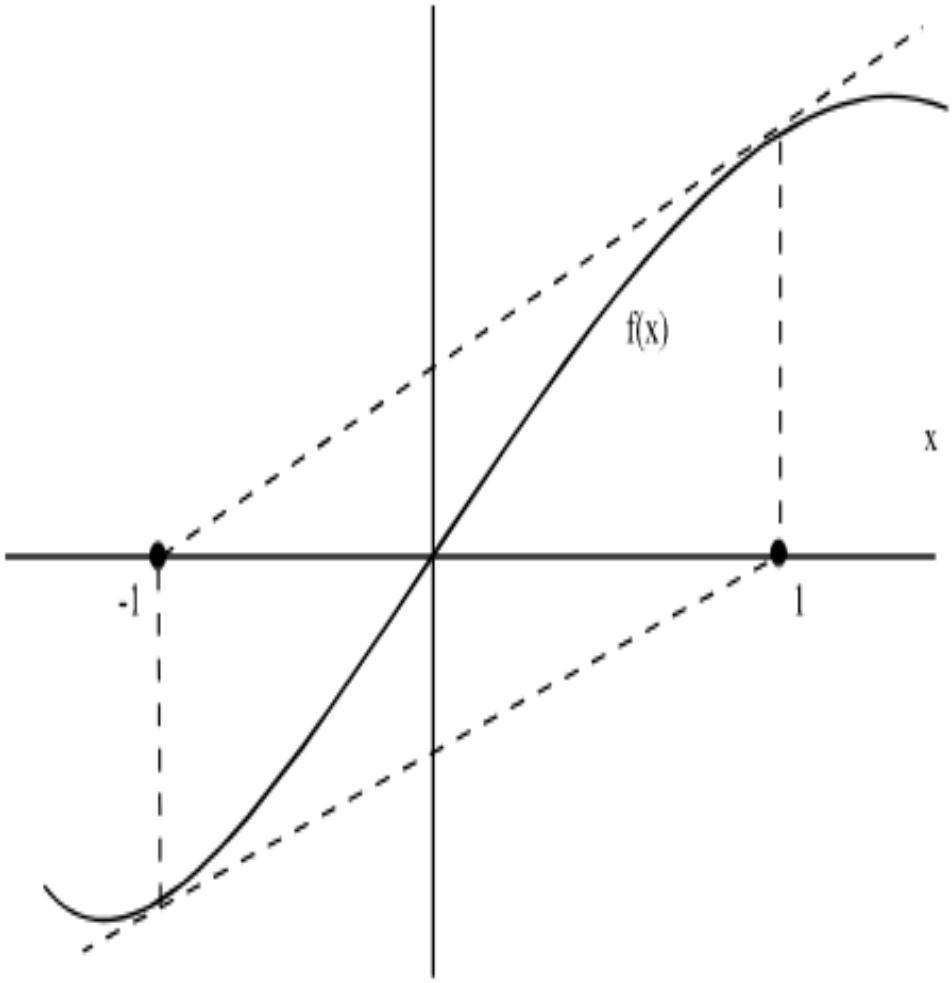
1. What happens when you apply Newton's method to solve this function for different starting values?
2. What does the Newton iterate look like for very small and very large values?

- For x_k small, the iteration reduces to $x_{k+1} = -2x_k$ and so it converges to 0 only if $x_0 = 0$.
- For x_k large, the iteration becomes

$$x_{k+1} = x_k \left(1 + \frac{3}{x_k^2}\right)$$

which diverges, but will eventually satisfy the stopping rule.

Another problems might be cycles.



Quasi-Newton method

Quasi-Newton methods replace the Jacobian in Newton's method with an estimate that is easier to compute. Specifically, quasi-Newton methods use an iteration rule

$$x_{k+1} = x_k - A_k^{-1} f(x_k)$$

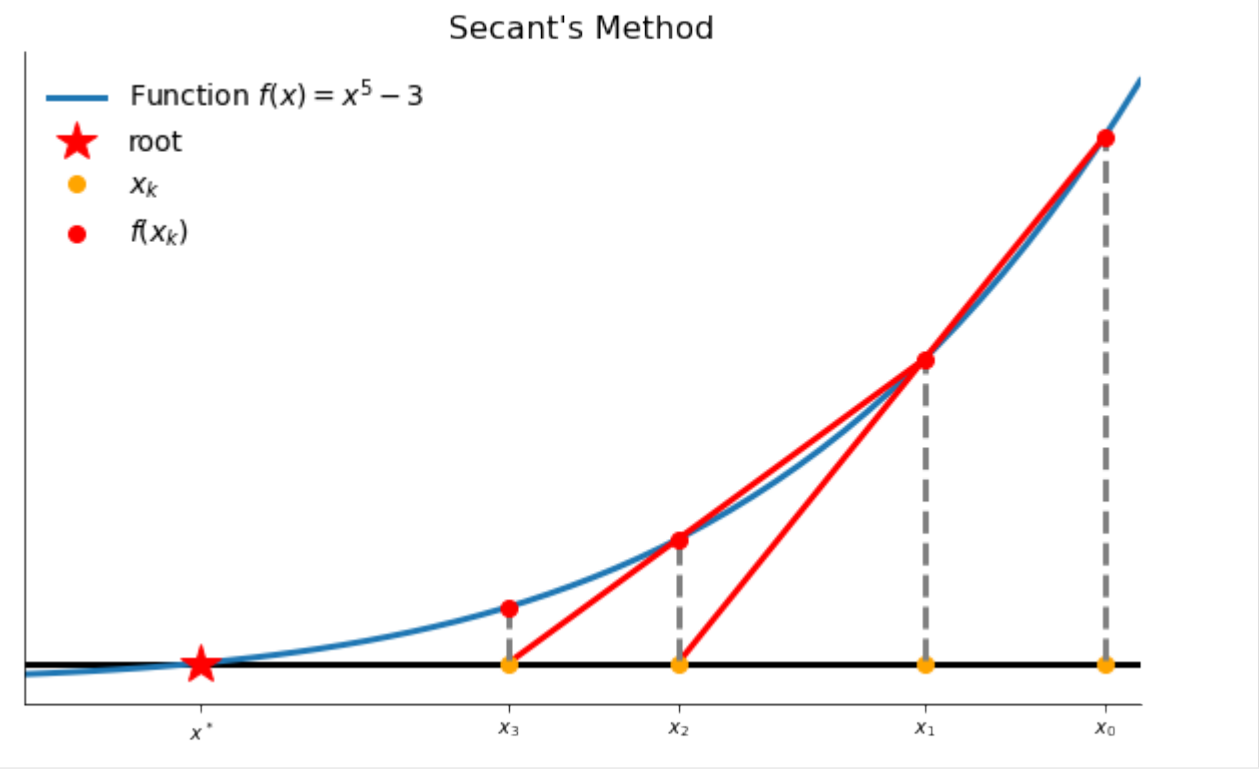
where A_k is an estimate of the Jacobian $f'(x_k)$.

The **secant method** replaces the derivative in Newton's method with the estimate

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

The secant method is so called because it approximates the function f using *secant* lines drawn through successive pairs of points on its graph.

[18]:
```
plot_secant_method()
```



Let's start with a crude implementation to make sure we understood the setup correctly. We will use this to show the `lambda` functions, see here for details.

[19]:
```
f = lambda z: z ** 4 - 2  # noqa: E731
x, xlag = 2.3, 2.4

for it in range(50):
    d = (f(x) - f(xlag)) / (x - xlag)
    x, xlag = x - f(x) / d, x
    if abs(x - xlag) < 1.0e-10:
        break

print(f"Root of our example function {x:5.3f}")
```

```
Root of our example function 1.189
```

Broyden's method is the most popular multivariate generalization of the univariate secant method. **Broyden's method** replaces the Jacobian in Newton's method with an estimate A_k that is updated by making the smallest possible change (measured by the Frobenius norm) that is consistent with the secant condition:

$$f(x_{k+1}) - f(x_k) = A_{k+1}(x_{k+1} - x_k).$$

This yields the iteration rule

$$A_{k+1} = A_k + (f(x_{k+1}) - f(x_k) - A_k d_k)\frac{d_k'}{d_k' d_k}$$

where $d_k = x_{k+1} - x_k$. Often A_0 is equal to the numerical approximation of f at x_0. The remarkable feature of Broyden's method is that it is able to generate a reasonable approximation to the Jacobian matrix with no additional evaluations of the function. This approach is readily available in scipy.

[20]:
```
rslt = optimize.root(f, 1.0, method="broyden1")
print(f"Root of our example function {rslt['x']:5.3f}")
```

```
Root of our example function 1.189
```

Consider a market with two firms producing the same good. Firm i's total cost of production is a function of the quantity q_i it produces.

$$C_i(q_i) = \frac{\beta_i}{2} q_i^2$$

The market clearing price is a function of the total quantity produced by both firms.

$$P(q_1 + q_2) = (q_1 - q_2)^{-\alpha}$$

Firm i chooses production q_i so as to maximize its profit taking the other firm's output as given.

$$\pi_i(q_1, q_2) = P(q_1 + q_2)q_i - C_i(q_i).$$

Thus in equilibrium,

$$\frac{\partial \pi_i}{\partial q_i} = (q_1 - q_2)^{-\alpha} - \alpha(q_1 + q_2)^{-(\alpha+1)} q_i - \beta_i q_i = 0 \qquad \text{for} \quad i = 1, 2.$$

Exercise

- Compute the market equilibrium quantities using Broyden's method for $\alpha = 0.6$ and $\beta = [0.6, 0.8]$.

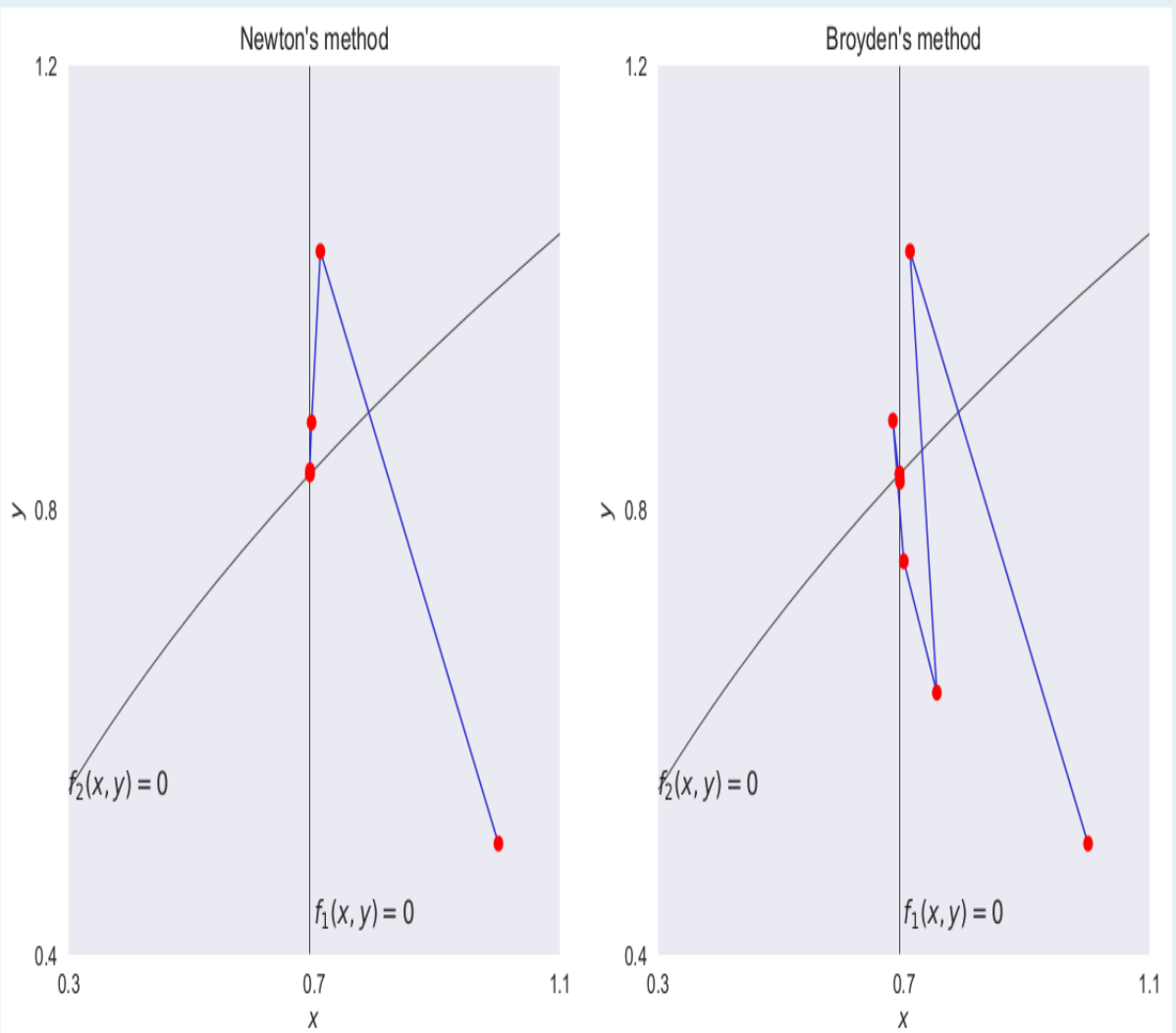


Figure 4: Convergence Paths for Newton and Broyden's Methods

Choosing a solution method

We now consider a more challenging task and compare the performance of `scipy`'s root finding algorithms. We will use one of temfpy's test functions for this.

[21]:
```
??exponential
```

```
Object `exponential` not found.
```

Now let's see which of the algorithms performs best for a ten dimensional problem.

[22]:
```
METHODS = ["broyden1", "broyden2", "anderson", "Krylov"]
DIMENSION = 10
PROBLEMS = 5
```

We are ready to run our benchmarking exercise.

[23]:
```
def exponential_function(x):

    x = np.atleast_1d(x)

    p = x.shape[0]
    rslt = np.tile(np.nan, p)

    for i in range(p):
        xi = np.clip(x[i], None, 300)
        if i == 0:
            rslt[i] = np.exp(xi) - 1
        else:
            rslt[i] = (i / 10) * (np.exp(xi) + x[i - 1] - 1)

    return rslt


np.testing.assert_almost_equal(exponential_function([0, 0]), [0.0] * 2)

columns = ["Algorithm", "Sample", "Success", "Iteration"]
df = pd.DataFrame(columns=columns)

for method in METHODS:

    # Ensure that all have same test problems.
    np.random.seed(123)

    for _ in range(PROBLEMS):

        x0 = np.random.uniform(size=DIMENSION)

        rslt = optimize.root(exponential_function, x0, method=method)

        info = [[method, _, rslt["success"], rslt["nit"]]]
        df = df.append(pd.DataFrame(info, columns=columns), ignore_index=True)
```

```
/home/peisenha/.local/lib/python3.8/site-packages/scipy/optimize/nonlin.py:1016:␣
↪RuntimeWarning: invalid value encountered in true_divide
  d = v / vdot(df, v)
/home/peisenha/.local/lib/python3.8/site-packages/scipy/optimize/nonlin.py:1016:␣
↪RuntimeWarning: divide by zero encountered in true_divide
  d = v / vdot(df, v)
/home/peisenha/.local/lib/python3.8/site-packages/scipy/optimize/nonlin.py:771:␣
↪RuntimeWarning: invalid value encountered in multiply
  self.collapsed += c[:,None] * d[None,:].conj()
```

Now we can explore the performance of the alternative implementations.

[24]:
```
df.head()
```

[24]:

	Algorithm	Sample	Success	Iteration
0	broyden1	0	False	1100
1	broyden1	1	False	1100
2	broyden1	2	False	1100

(continues on next page)

(continued from previous page)

```
3  broyden1      3   False      1100
4  broyden1      4   False      1100
```

Convergence

Two factors determine the speed with which a properly coded and initiated algorithm will converge to a solution:

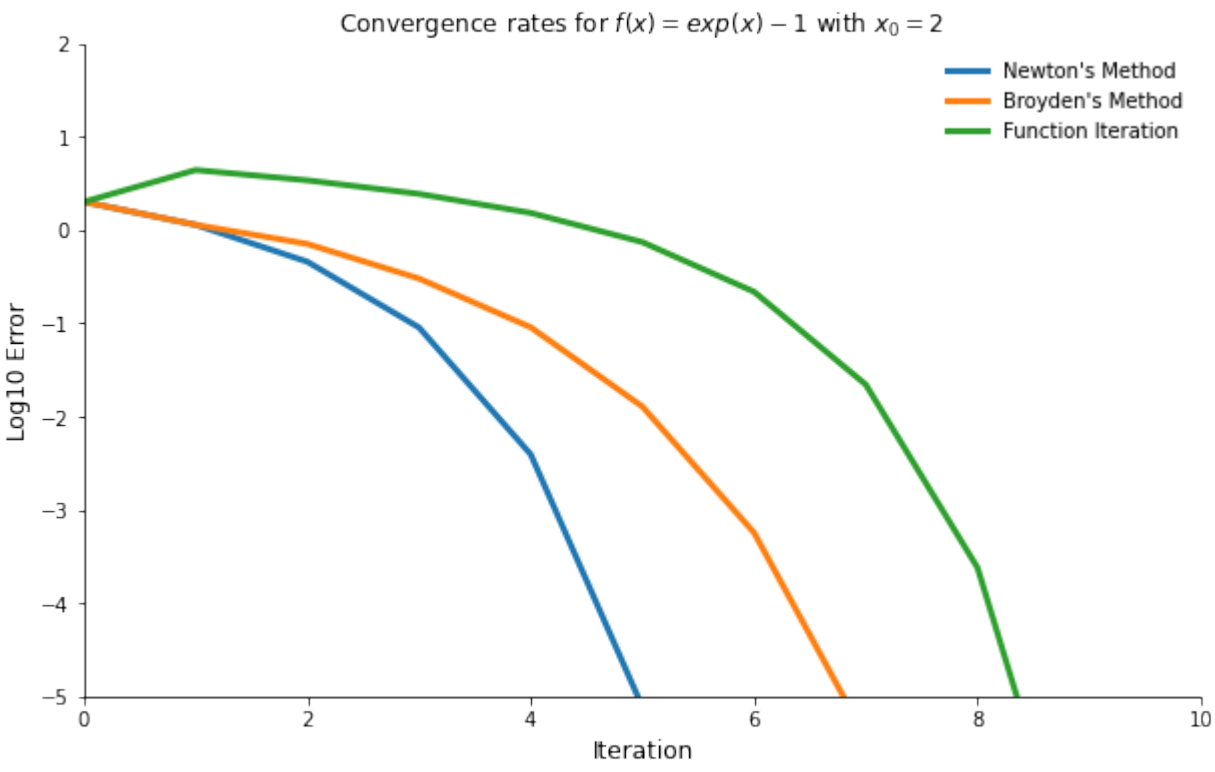
- Asymptotic rate of convergence
- Computational effort per iteration

The asymptotic rate of convergence measures improvement afforded per iteration near the solution. A sequence x_k converges to x^* at an asymptotic rate of order p if there is constant $C > 0$ such that for k sufficiently large,

$$||x_{k+1} - x^*|| \leq C||x_k - x^*||^p.$$

- Function iteration converges at a "linear" rate with $p = 1$ and $C < 1$.
- Broyden's method converges at a "superlinear" rate with $p \approx 1.62$.
- Newton's method converges at a "quadratic" rate with $p = 2$.

[25]:
```
plot_convergence()
```



However, algorithms differ in computations per iteration.

- Function iteration requires a function evaluation.
- Broyden's method additionally requires a linear solve.
- Newton's method additionally requires a Jacobian evaluation.

Thus, a faster rate of convergence typically can be achieved only by investing greater computational effort per iteration. The optimal tradeoff between rate of convergence and computational effort per iteration varies across applications.

Resources

- **HOMPACK**: https://www.netlib.org/hompack

1.3.2 Functions

This module contains the algorithms for the nonlinear equations lab.

The materials follow Miranda and Fackler (2004, [MF04]) (Chapter 3). The python code draws on Romero-Aguilar (2020, [RA20]).

labs.nonlinear_equations.nonlinear_algorithms.**bisect**(*f*, *a*, *b*, *tolerance=1.5e-08*)

Apply bisect method to root finding problem.

Iterative procedure to find the root of a continuous real-values function $f(x)$ defined on a bounded interval of the real line. Define interval $[a, b]$ that is known to contain or bracket the root of f (i.e. the signs of $f(a)$ and $f(b)$ must differ). The given interval $[a, b]$ is then repeatedly bisected into subintervals of equal length. Each iteration, one of the two subintervals has endpoints of different signs (thus containing the root of f) and is again bisected until the size of the subinterval containing the root reaches a specified convergence tolerance.

Parameters

- **f** (*callable*) – Continuous, real-valued, univariate function $f(x)$.
- **a** (*int or float*) – Lower bound a for $x \in [a, b]$.
- **b** (*int or float*) – Upper bound a for $x \in [a, b]$. Select a and b so that $f(b)$ has different sign than $f(a)$.
- **tolerance** (*float*) – Convergence tolerance.

Returns **x** – Solution to the root finding problem within specified tolerance.

Return type float

Examples

```
>>> x = bisect(f=lambda x : x ** 3 - 2, a=1, b=2)[0]
>>> round(x, 4)
1.2599
```

labs.nonlinear_equations.nonlinear_algorithms.**fischer**(*u*, *v*, *sign*)

Define Fischer's function.

$$\phi_i^{\pm}(u, v) = u_i + v_i \pm \sqrt{u_i^2 + v_i^2}$$

Parameters

- **u** (*float*) –
- **v** (*float*) –
- **sign** (*float or int*) – Gives sign of equation. Should be either 1 or -1.

Returns

Return type callable

labs.nonlinear_equations.nonlinear_algorithms.**fixpoint**(*f*, *x0*, *tolerance=0.0001*)
Compute fixed point using function iteration.

Parameters

- **f** (*callable*) – Function $f(x)$.
- **x0** (*float*) – Initial guess for fixed point (starting value for function iteration).
- **tolerance** (*float*) – Convergence tolerance (tolerance < 1).

Returns **x** – Solution of function iteration.

Return type float

Examples

```
>>> import numpy as np
>>> x = fixpoint(f=lambda x : x**0.5, x0=0.4, tolerance=1e-10)[0]
>>> np.allclose(x, 1)
True
```

labs.nonlinear_equations.nonlinear_algorithms.**funcit**(*f*, *x0=2*)
Apply function iteration using the fixpoint method.

labs.nonlinear_equations.nonlinear_algorithms.**newton_method**(*f*, *x0*, *tolerance=1.5e-08*)
Apply Newton's method to solving nonlinear equation.

Solve equation using successive linearization, which replaces the nonlinear problem by a sequence of linear problems whose solutions converge to the solution of the nonlinear problem.

Parameters

- **f** (*callable*) – (Univariate) function $f(x)$.
- **x0** (*float*) – Initial guess for the root of f.
- **tolerance** (*float*) – Convergence tolerance.

Returns **xn** – Solution of function iteration.

Return type float

1.4 Optimization

We discuss the key attributes of optimization algorithms that determine the choice of a suitable optimization algorithm. We explore the role of noise in the criterion function and ill-conditioning for different groups of optimizers: local vs. global, derivative-based vs. derivative-free. We conclude with some programming exercises for nonlinear least squares problems and implement a simple maximum likelihood estimation.

1.4.1 Optimization

[1]:
```
from functools import partial
from temfpy.optimization import carlberg

from scipy import optimize as opt
import matplotlib.pyplot as plt
from scipy.stats import norm
import seaborn as sns
import scipy as sp
import pandas as pd
import numpy as np

from optimization_problems import get_test_function_gradient
from optimization_problems import get_parameterization
from optimization_problems import get_test_function
from optimization_auxiliary import process_results
from optimization_auxiliary import get_bounds
from optimization_plots import plot_contour
from optimization_plots import plot_surf
from optimization_plots import plot_optima_example
from optimization_plots import plot_true_observed_example
```

Outline

1. Setup
2. Algorithms
3. Gradient-based methods
4. Derivative-free methods
5. Benchmarking exercise
6. Special cases

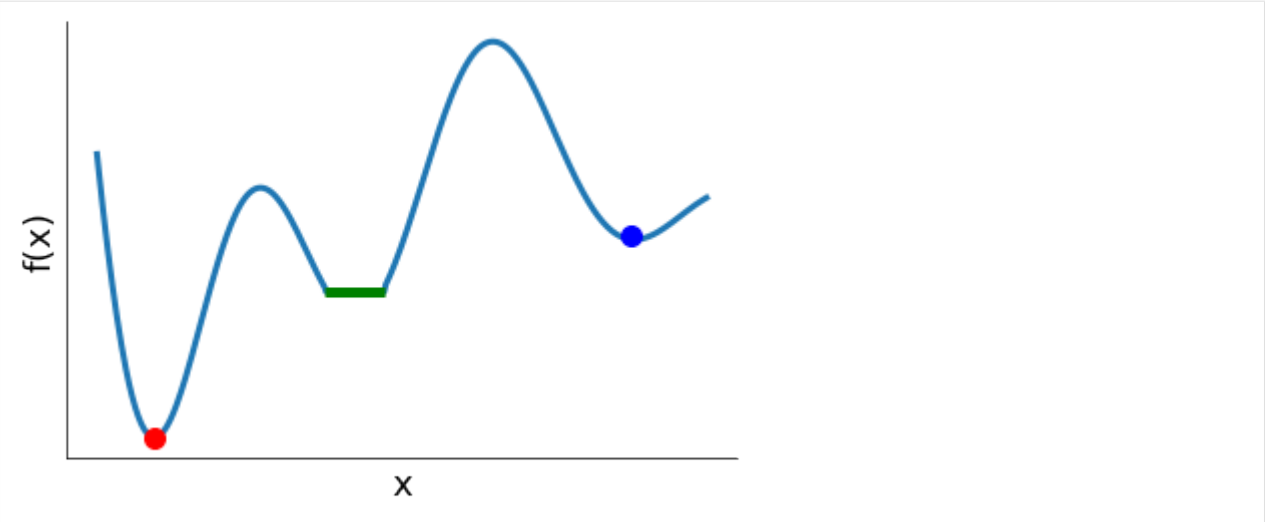
Setup

In the finite-dimensional unconstrained optimization problem, one is given a function $f : R^n \mapsto R$ and asked to find an x^* such that $f(x^*) \geq f(x)$ for all x. We call f the objective function and x^* , if it exists, the global minimum of f. We focus on minimum - to solve a minimization problem, simply minimize the negative of the objective.

We say that $x^* \in R^n$ is a ...

- strict global minimum of f if $f(x^*) > f(x)$ for all $x \neq x^*$.
- weak local minimum of f if $f(x^*) \geq f(x)$ for all x in some neighborhood of x^*.
- strict local minimum of f if $f(x^*) > f(x)$ for all $x \neq x^*$ in some neighborhood of x^*.

[2]:
```
plot_optima_example()
```



Let $f : R^n \mapsto R$ be twice continuously differentiable.

- **First Order Necessary Conditions:** If x^* is a local minimum of f, then $f'(x^*) = 0$.
- **Second Order Necessary Condition:** If x^* is a local minimum of f, then $f''(x^*)$ is negative semidefinite.

We say x is a critical point of f if it satisfies the first-order necessary condition.

- **Sufficient Condition:** If $f'(x^*) = 0$ and $f''(x^*)$ is negative definite, then x^* is a strict local minimum of f.
- **Local-Global Theorem:** If f is concave, and x^* is a local minimum of f, then x^* is a global minimum of f.

Key problem attributes

- Convexity: convex vs. non-convex
- Optimization-variable type: continuous vs. discrete
- Constraints: unconstraint vs. constraint
- Number of optimization variables: low-dimensional vs. high-dimensional

These attributes dictate:

- ability to find solution
- problem complexity and computing time
- appropriate methods
- relevant software

$\Rightarrow$ Always begin by categorizing your problem

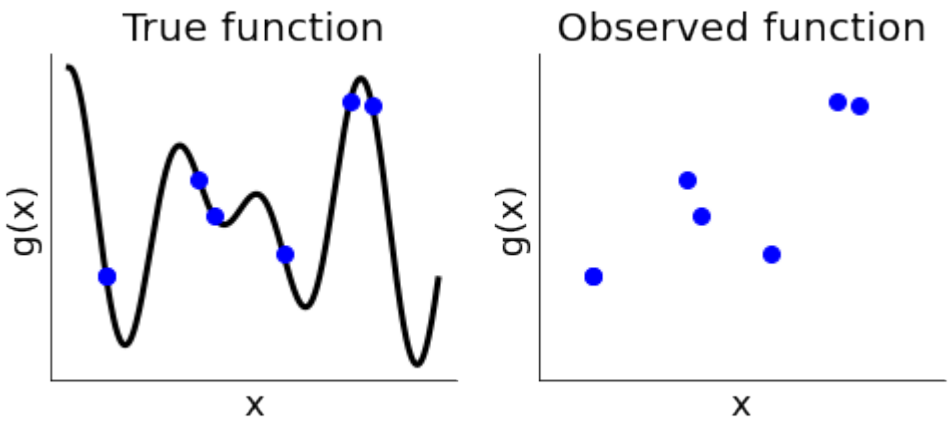
Optimization problems are ubiquitous in economics:

- Government maximizes social welfare
- Competitive equilibrium maximizes total surplus
- Ordinary least squares estimator minimizes sum of squares
- Maximum likelihood estimator maximizes likelihood function

Algorithms

We are mostly blind to the function we are trying to minimize and can only compute the function at a limited number of points. Each evaluation is computationally expensive.

[3]:
```
plot_true_observed_example()
```



Goals

- reasonable memory requirements
- low failure rate, convergence conditions are met
- convergence in a few iterations with low cost for each iteration

Catergorization

- gradient-based vs. derivative-free
- global vs. local

Question

- How to compute derivatives?

Gradient-based methods

Benefits

- efficient for many variables
- well-suited for smooth objective and constraint functions

Drawbacks

- requires computing the gradient, potentially challenging and time-consuming
- convergence is only local
- not-well suited for noisy functions, derivative information flawed

Second derivative are also very useful, but …

- Hessians are $n \times n$, so expensive to construct and store
- often only approximated using quasi-Newton methods

Questions

1. How to use gradient-based algorithms to find a global optimum?
2. Any ideas on how to reduce the memory requirements for a large Hessian?

Algorithm 1 Gradient-based framework

Choose initial guess x_0, set $k \leftarrow 0$

while (not converged) **do**

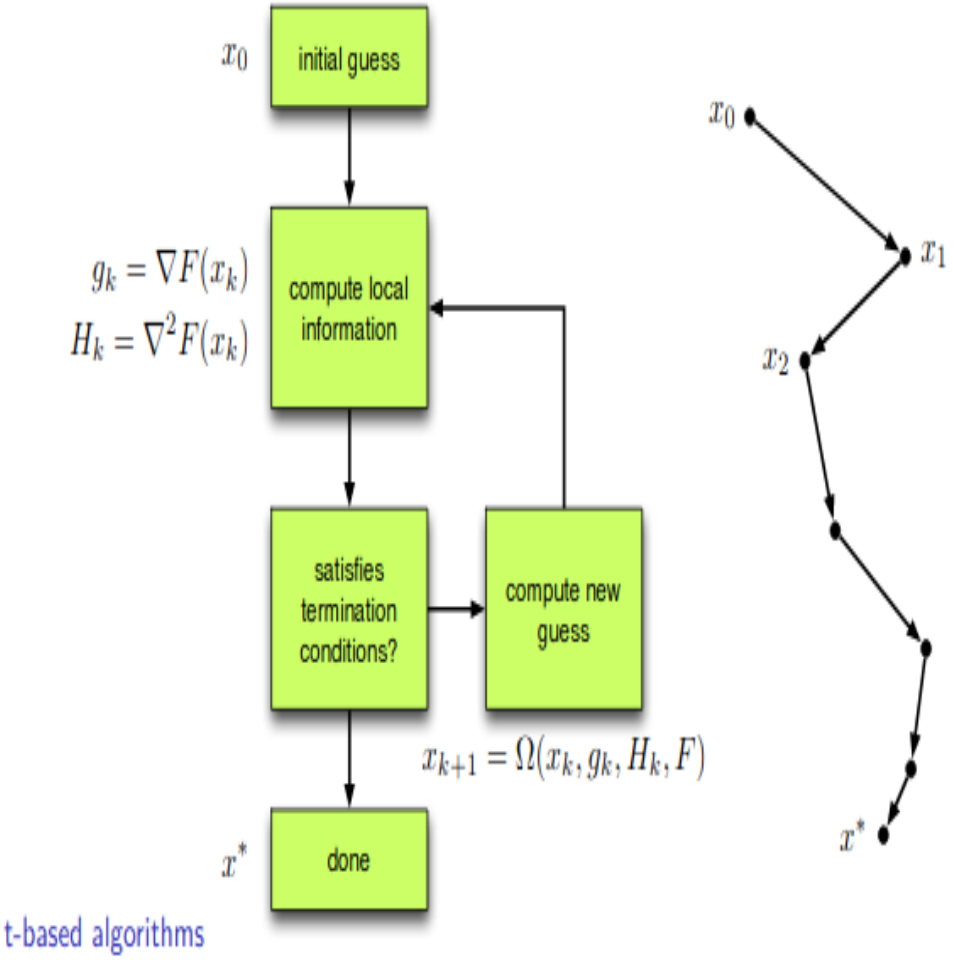
Choose direction p_k and step length α_k

(This often involves computing local information, e.g., $\nabla f(x_k)$, $\nabla^2 f(x_k)$)

$x_{k+1} = x_k + \alpha_k p_k$.

$k \leftarrow k + 1$

end while



t-based algorithms

There is two different classes of gradient-based algorithms.

- Line-search methods
  - compute p_k be a descent direction
  - compute a_k to produce a sufficient decrease in the objective function

Let's see here for how such a line search looks like in practice for the Newton-CG algorithm.

- Trust-region methods
  - determine a maximum allowable step length (trust-region radius) δk
  - compute step k with $||p_k|| \leq \Delta$ using a model $m(p) \approx f(x_k + p)$

As an example implementation, see here for the `scipy.optimize_trustregion.py` implementation.

Derivative-Free Methods

Benefits

- often better at finding a global minimum if function not convex
- robust with respect to noise in criterion function
- amenable to parallelization

Drawbacks

- extremely slow convergence for high-dimensional problems

There are two different classes of derivative-free algorithms.

- heuristic, inspired by nature
  - basin-hopping
  - evolutionary algorithms
- direct search
  - directional
  - simplicial

Test function

$$f(x) = \frac{1}{2}\sum_{i=1}^{n} a_i \cdot (x_i - 1)^2 + b \cdot \left[n - \sum_{i=1}^{n} \cos(2\pi(x_i - 1)) \right],$$

where a_i and b provide the parameterization of the function.

Exercises

1. Implement this test function.
2. Visualize the shape of our test function for the one-dimensional case.
3. What is the role of the parameters a_1 and b?
4. What is the functions global minimum?

[4]:
```
??get_test_function
```

[5]:
```
??get_parameterization
```

We want to be able to use our test function for different configurations of the challenges introduced by noise and ill-conditioning.

[6]:
```
add_noise, add_illco, x0 = False, False, [4.5, -1.5]


def get_problem(dimension, add_noise, add_illco, seed=123):
    np.random.seed(seed)
```

(continues on next page)

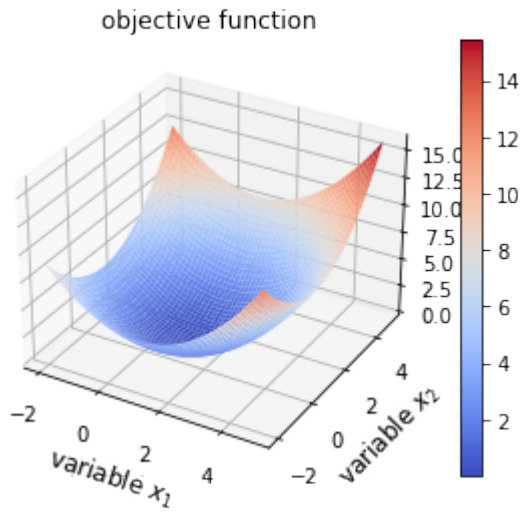
(continued from previous page)

```
    a, b = get_parameterization(dimension, add_noise, add_illco)
    get_test_function_p = partial(get_test_function, a=a, b=b)
    get_test_function_gradient_p = partial(get_test_function_gradient, a=a, b=b)
    return get_test_function_p, get_test_function_gradient_p


dimension = len(x0)
opt_test_function, opt_test_gradient = get_problem(dimension, add_noise, add_illco)
np.testing.assert_equal(opt_test_function([1, 1]), 0.0)
```

Let's see how the surface and contour plots look like under different scenarios.

[7]:
```
opt_test_function, _ = get_problem(dimension, add_noise, add_illco)
plot_surf(opt_test_function)
```

```
/Users/emilyschwab/Desktop/IAME_Work/ose-course-scientific-computing/labs/optimization/
↪optimization_plots.py:62: MatplotlibDeprecationWarning: Calling gca() with keyword␣
↪arguments was deprecated in Matplotlib 3.4. Starting two minor releases later, gca()␣
↪will take no keyword arguments. The gca() function should only be used to get the␣
↪current axes, or if no axes exist, create new axes with default keyword arguments. To␣
↪create a new axes with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection="3d")
```



Question

- How is the global minimum affected by the addition of noise and ill-conditioning?

Benchmarking exercise

Let's get our problem setting and initialize a container for our results. We will use the convenient interface to scipy.optimize.minimize. Its documentation also points you to research papers and textbooks where the details of the algorithms are discussed in more detail. We need to invest a little in the design of our setup first, but then we can run the benchmarking exercise with ease and even adding additional optimization algorithms is straightforward.

[8]:
```
ALGORITHMS = ["CG", "Newton-CG", "Nelder-Mead", "Diff-Evol"]
add_noise, add_illco, dimension = False, False, 2
```

[9]:
```
x0 = [4.5, -1.5]
opt_test_function, opt_test_gradient = get_problem(dimension, add_noise, add_illco)
df = pd.DataFrame(columns=["Iteration", "Distance"], index=ALGORITHMS)
df.index.name = "Method"
```

Let's fix what will stay unchanged throughout.

[10]:
```
call_optimizer = partial(
    opt.minimize,
    fun=opt_test_function,
    x0=x0,
    jac=opt_test_gradient,
    options={"disp": True, "return_all": True, "maxiter": 100000},
)
```

We prepared some functions to process results from the optimizer calls.

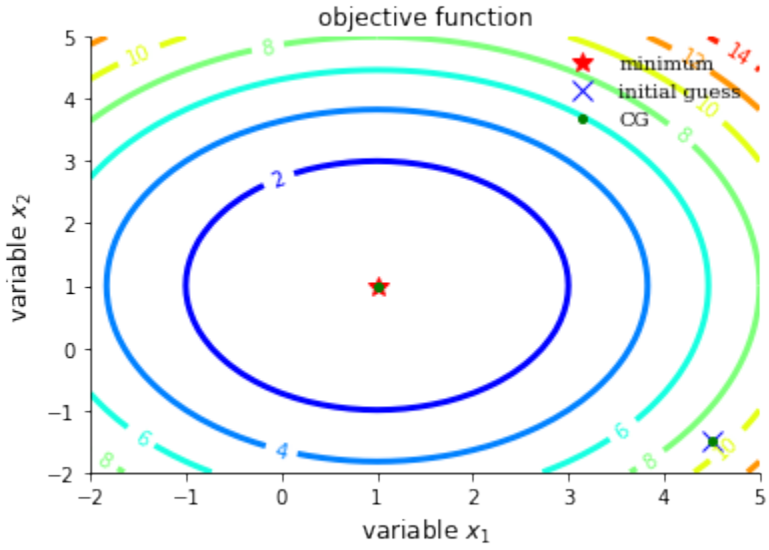
[11]:
```
??process_results
```

Conjugate gradient

[12]:
```
method = "CG"
res = call_optimizer(method=method)
initial_guess = [4.5, -1.5]
df = process_results(df, method, res)
plot_contour(opt_test_function, res["allvecs"], method, initial_guess)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 1
         Function evaluations: 3
         Gradient evaluations: 3
```

[12]:
```
<module 'matplotlib.pyplot' from '/Users/emilyschwab/miniconda3/envs/ose-course-
↪scientific-computing/lib/python3.8/site-packages/matplotlib/pyplot.py'>
```

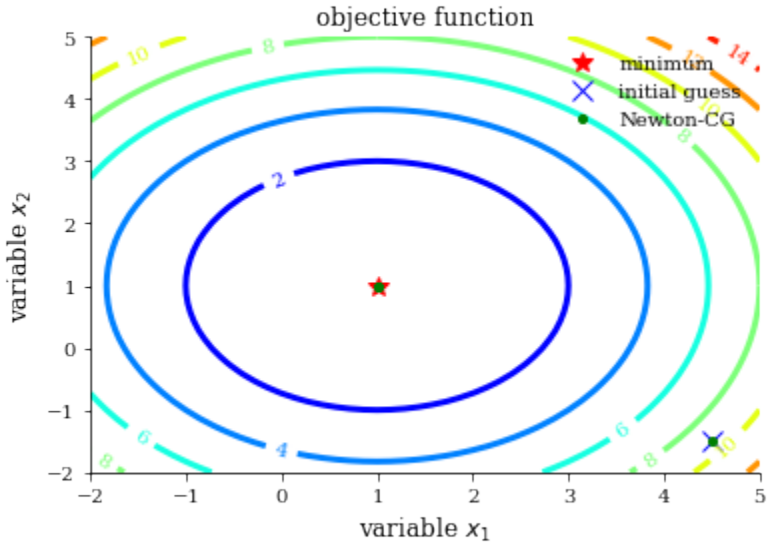


Newton-CG

[13]:
```
method = "Newton-CG"
res = call_optimizer(method=method)
initial_guess = [4.5, -1.5]
df = process_results(df, method, res)
plot_contour(opt_test_function, res["allvecs"], method, initial_guess)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 2
         Function evaluations: 2
         Gradient evaluations: 3
         Hessian evaluations: 0
```

[13]:
```
<module 'matplotlib.pyplot' from '/Users/emilyschwab/miniconda3/envs/ose-course-
↪scientific-computing/lib/python3.8/site-packages/matplotlib/pyplot.py'>
```



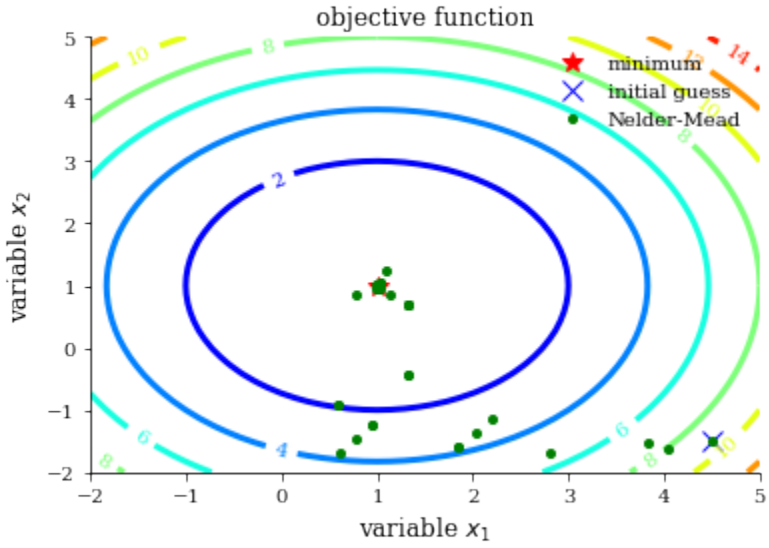
Nelder Mead

[14]:
```
method = "Nelder-Mead"
res = call_optimizer(method=method)
initial_guess = [4.5, -1.5]
df = process_results(df, method, res)
plot_contour(opt_test_function, res["allvecs"], method, initial_guess)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 51
         Function evaluations: 95
```

```
/Users/emilyschwab/miniconda3/envs/ose-course-scientific-computing/lib/python3.8/site-
↪packages/scipy/optimize/_minimize.py:522: RuntimeWarning: Method Nelder-Mead does not␣
↪use gradient information (jac).
  warn('Method %s does not use gradient information (jac).' % method,
```

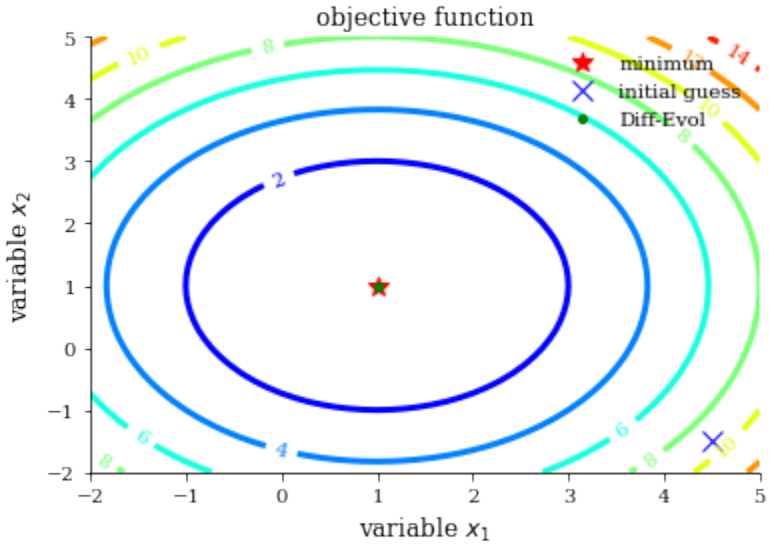
[14]:
```
<module 'matplotlib.pyplot' from '/Users/emilyschwab/miniconda3/envs/ose-course-
↪scientific-computing/lib/python3.8/site-packages/matplotlib/pyplot.py'>
```



Differential evolution

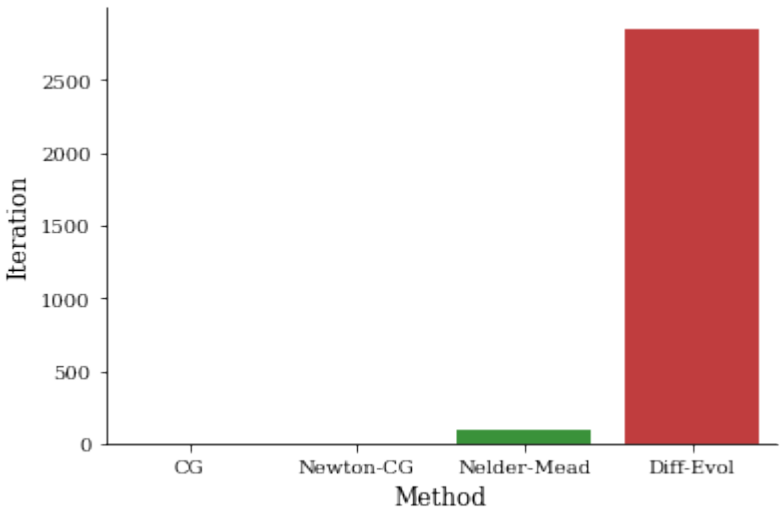
[15]:
```
??get_bounds
```

[16]:
```
method = "Diff-Evol"
res = opt.differential_evolution(opt_test_function, get_bounds(dimension))
initial_guess = [4.5, -1.5]
plot_contour(opt_test_function, res["x"], method, initial_guess)
df = process_results(df, method, res)
```

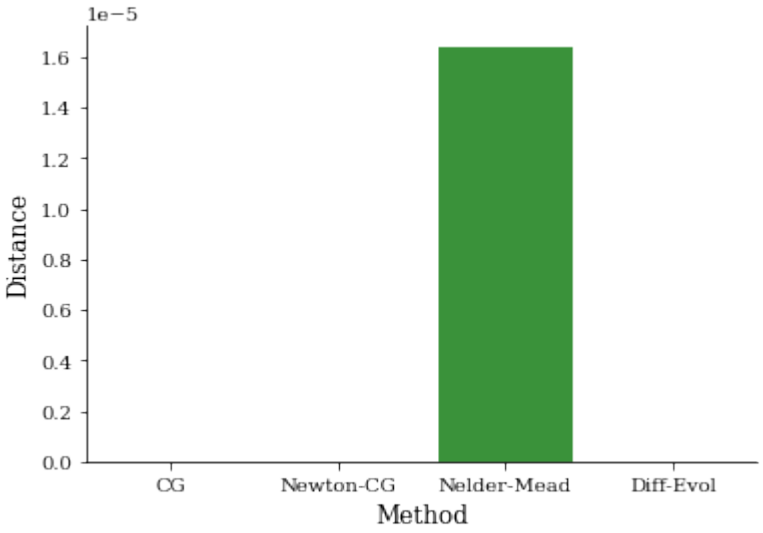


Summary

[17]:
```
_ = sns.barplot(x="Method", y="Iteration", data=df.reset_index())
```



[18]:
```
_ = sns.barplot(x="Method", y="Distance", data=df.reset_index())
```



Speeding up test function

We want to increase the dimensionality of our optimization problem going forward. Even in this easy setting, it is worth to re-write our objective function using `numpy` to ensure its speedy execution. A faster version is already available as part of the Python package temfpy. Below, we compare our test function to the temfpy version and assess their performance in regard to speed.

[19]:
```
??get_test_function
```

[20]:
```
??carlberg
```

It is very easy to introduce errors when speeding up your code as usually you face a trade-off between readability and performance. However, setting up a simple testing harness that simply compares the results between the slow, but readable, implementation and the fast one for numerous random test problems. For more automated, but random, testing see Hypothesis.

[21]:
```
def get_speed_test_problem():
    add_illco, add_noise = np.random.choice([True, False], size=2)
    dimension = np.random.randint(2, 100)

    a, b = get_parameterization(dimension, add_noise, add_illco)
    x0 = np.random.uniform(size=dimension)
    return x0, a, b
```

Now we are ready to put our fears at ease.

[22]:
```
for _ in range(1000):
    args = get_speed_test_problem()
    stats = get_test_function(*args), carlberg(*args)
    np.testing.assert_almost_equal(*stats)
```

Let's see whether this was worth the effort for a small and a large problem using the `%timeit` magic function.

[23]:
```
dimension, add_noise, add_illco = 100, True, True
x0 = np.random.uniform(size=dimension)
a, b = get_parameterization(dimension, add_noise, add_illco)
```

[24]:
```
%timeit carlberg(x0, a, b)
```

```
51.6 µs ± 20.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[25]:
```
%timeit get_test_function(x0, a, b)
```

```
618 µs ± 198 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In this particular setting, there is no need to increase the performance even further. However, as a next step, check out numba, for even more flexibility in speeding up your code.

Exercises

1. Repeat the exercise in the case of noise in the criterion function and try to summarize your findings.
2. What additional problems arise as the dimensionality of the problem for a 100-dimensional problem? Make sure to use the fast implementation of the test function.

Special cases

Nonlinear least squares and maximum likelihood estimation have special structure that can be exploited to improve the approximation of the inverse Hessian.

Nonlinear least squares

We will estimate the following nonlinear consumption function using data from Greene's textbook:

$$C = \alpha + \beta \times Y^{\gamma} + \epsilon$$

which is estimated with quarterly data on real consumption and disposable income for the U.S. economy from 1950 to 2000.

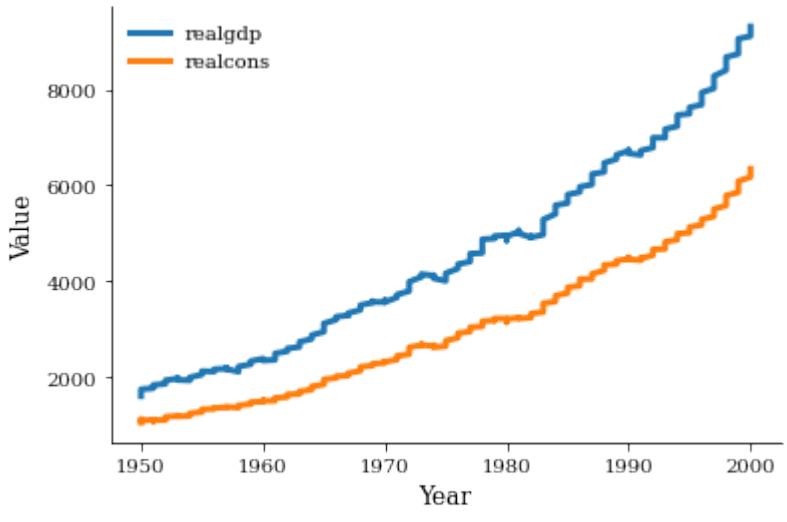
[26]:
```
df = pd.read_pickle("material/data-consumption-function.pkl")
df.head()
```

[26]:
```
          realgdp  realcons
Year qtr
1950 1     1610.5    1058.9
     2     1658.8    1075.9
     3     1723.0    1131.0
     4     1753.9    1097.6
1951 1     1773.5    1122.8
```

Let's confirm the basic relationship to get an idea of what to expect for the estimated parameters.

[27]:
```
fig, ax = plt.subplots()
x = df.index.get_level_values("Year")

for name in ["realgdp", "realcons"]:
    y = df[name]
    ax.plot(x, y, label=name)

ax.set_xlabel("Year")
ax.set_ylabel("Value")
ax.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x11dc95c70>
```



Now we set up the criterion function such that it fits the requirements.

[28]:
```
consumption = df["realcons"].values
income = df["realgdp"].values


def ssr(x, consumption, income):
    alpha, beta, gamma = x
    residuals = consumption - alpha - beta * income ** gamma
    return residuals


ssr_partial = partial(ssr, consumption=consumption, income=income)
rslt = sp.optimize.least_squares(ssr_partial, [0, 0, 1])["x"]
```

Exercise

- Evaluate the fit of the model.

Maximum likelihood estimation

Greene (2012) considers the following binary choice model.

$$P[Grade = 1] = F(\beta_0 + \beta_1 GPA + \beta_2 TUCE + \beta_3 PSI)$$

where F the cumulative distribution function for either the normal distribution (Probit) or the logistic distribution (Logit).

[29]:
```
df = pd.read_pickle("material/data-graduation-prediction.pkl")
df.head()
```

[29]:

OBS	GPA	TUCE	PSI	GRADE	INTERCEPT	GRADE
1	2.66	20	0	0	1	0
2	2.89	22	0	0	1	0
3	3.28	24	0	0	1	0
4	2.92	12	0	0	1	0
5	4.00	21	0	1	1	1

[30]:
```
def probit_model(beta, y, x):
    F = norm.cdf(x @ beta)
    fval = (y * np.log(F) + (1 - y) * np.log(1 - F)).sum()
    return -fval
```

[31]:
```
x, y = df[["INTERCEPT", "GPA", "TUCE", "PSI"]], df["GRADE"]
rslt = opt.minimize(probit_model, [0.0] * 4, args=(y, x))
```

Exercise

- Amend the code so that you can simply switch between estimating a Probit or Logit model.

Resources

- **Kevin T. Carlberg**: https://kevintcarlberg.net

Software

- **Ipopt**: https://coin-or.github.io/Ipopt
- **SNOPT (Sparse Nonlinear OPTimizer)**: https://ccom.ucsd.edu/~optimizers/solvers/snopt
- **Gurobi** https://www.gurobi.com
- **IBM CPLEX Optimizer** https://www.ibm.com/analytics/cplex-optimizer

Books

- Nocedal, J., & Wright, S. (2006). *Numerical optimization* . Springer Science & Business Media.
- Boyd, S., Boyd, S. P., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.
- Kochenderfer, M. J., & Wheeler, T. A. (2019). *Algorithms for optimization*. Mit Press.
- Fletcher, R. (2000). *Practical methods of optimization (2nd edn)*. Wiley.
- Nesterov, Y. (2018). *Lectures on convex optimization*. Springer Nature Switzerland.

Research

- Moré, J. J., & Wild, S. M. (2009). Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1), 172-191.
- Beiranvand, V., Hare, W., & Lucet, Y. (2017). Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18, 815–848.
- Bartz-Beielstein, T., et al. (2020). Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488.*.

1.5 Integration

We examine different strategies for the numerical integration of functions. We discuss rules based on Newton-Cotes quadrature formulas, Gaussian quadrature, and Monte Carlo methods in the uni-dimensional and multi-dimensional case. We conclude by comparing the performance of each approach under different scenarios.

1.5.1 Integration

[1]:
```
import numpy as np

from integration_algorithms import monte_carlo_quasi_two_dimensions
from integration_algorithms import quadrature_newton_trapezoid_one
from integration_algorithms import quadrature_newton_simpson_one
from integration_algorithms import quadrature_gauss_legendre_one
from integration_algorithms import quadrature_gauss_legendre_two
from integration_algorithms import monte_carlo_naive_one

from integration_plots import plot_naive_monte_carlo_randomness
from integration_plots import plot_naive_monte_carlo_error
from integration_plots import plot_gauss_legendre_weights
from integration_plots import plot_benchmarking_exercise
from integration_plots import plot_naive_monte_carlo
from integration_plots import plot_quasi_monte_carlo
from integration_plots import plot_starting_illustration
from integration_plots import plot_trapezoid_rule_illustration
from integration_plots import plot_simpsons_rule_illustration

from integration_problems import problem_kinked
from integration_problems import problem_smooth
```

Outline

1. Setup
2. Newton-Cotes rules
3. Gaussian formulas
4. Monte Carlo integration
5. Resources

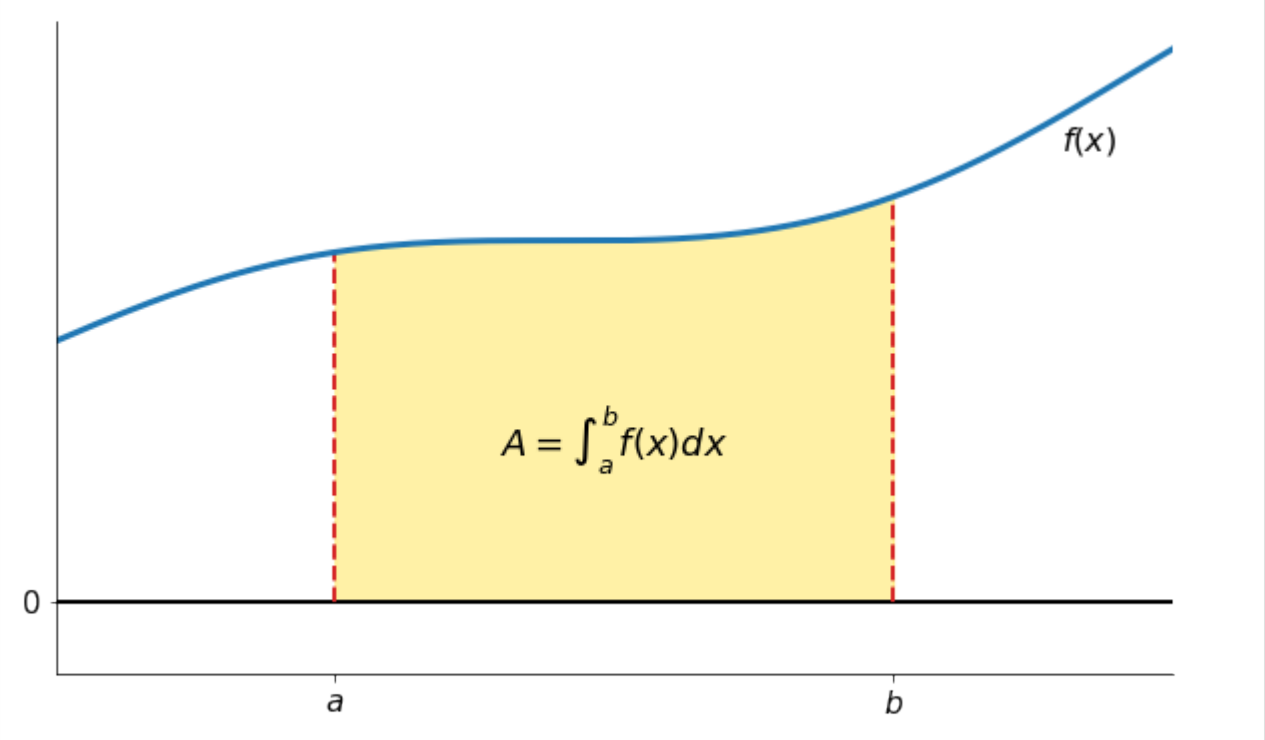
Setup

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

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

Let's look a visual representation of our problem.

[2]:
```
plot_starting_illustration()
```



Most numerical methods for computing this integral split up the original integral into a sum of several integrals, each covering a smaller part of the original integration interval $[a, b]$. This re-writing of the integral is based on a selection of integration points $x_i, i = 0, 1, ..., n$ that are distributed on the interval $[a, b]$. Integration points may, or may not, be evenly distributed.

Given the integration points, the original integral is re-written as a sum of integrals, each integral being computed over the sub-interval between two consecutive integration points. The integral from the beginning is thus expressed as:

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + ... + \int_{x_{n-1}}^{x_n} f(x)dx,$$

where $x_0 = a$ and $x_n = b$.

The different integration methods will differ in the way they approximate each integral on the right hand side. The fundamental idea is that each term is an integral over a small interval $[x_i, x_{i+1}]$, and over this small interval, it makes sense to approximate f by a simple shape

Newton-Cotes rules

Newton–Cotes quadrature rules are a group of formulas for numerical integration (also called quadrature) based on evaluating the integrand at equally spaced points. The integration points are then computed as

$$x_i = a + ih, i = 0, 1, ..., n,$$

where $h = (b - a)/n$

The closed Newton-Cotes formula of degree n is stated as

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

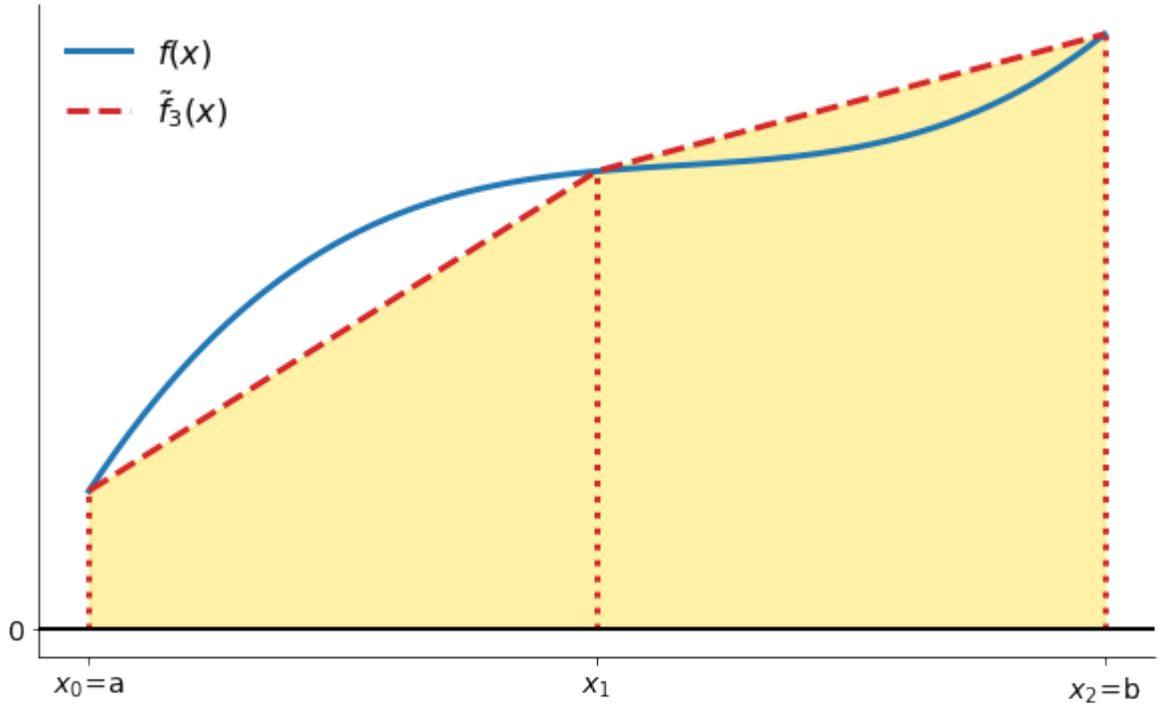
We will consider the first two degrees in more detail:

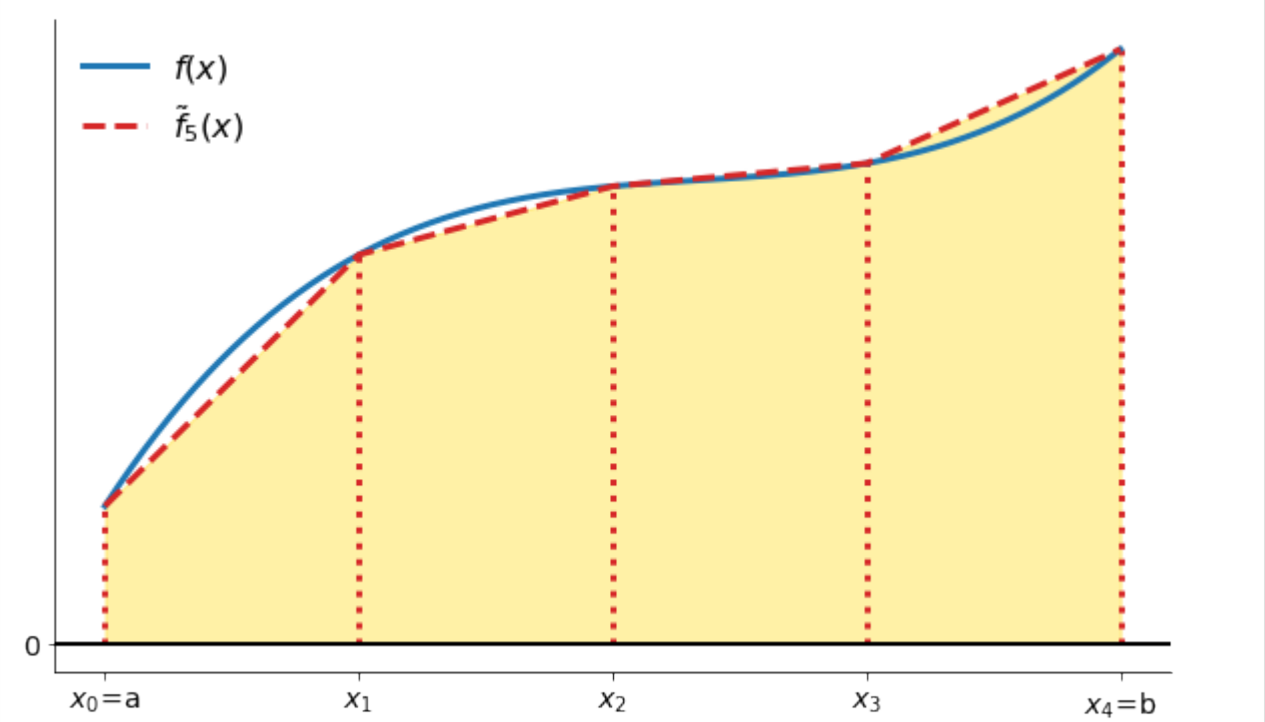
- Trapezoid rule
- Simpson's rule

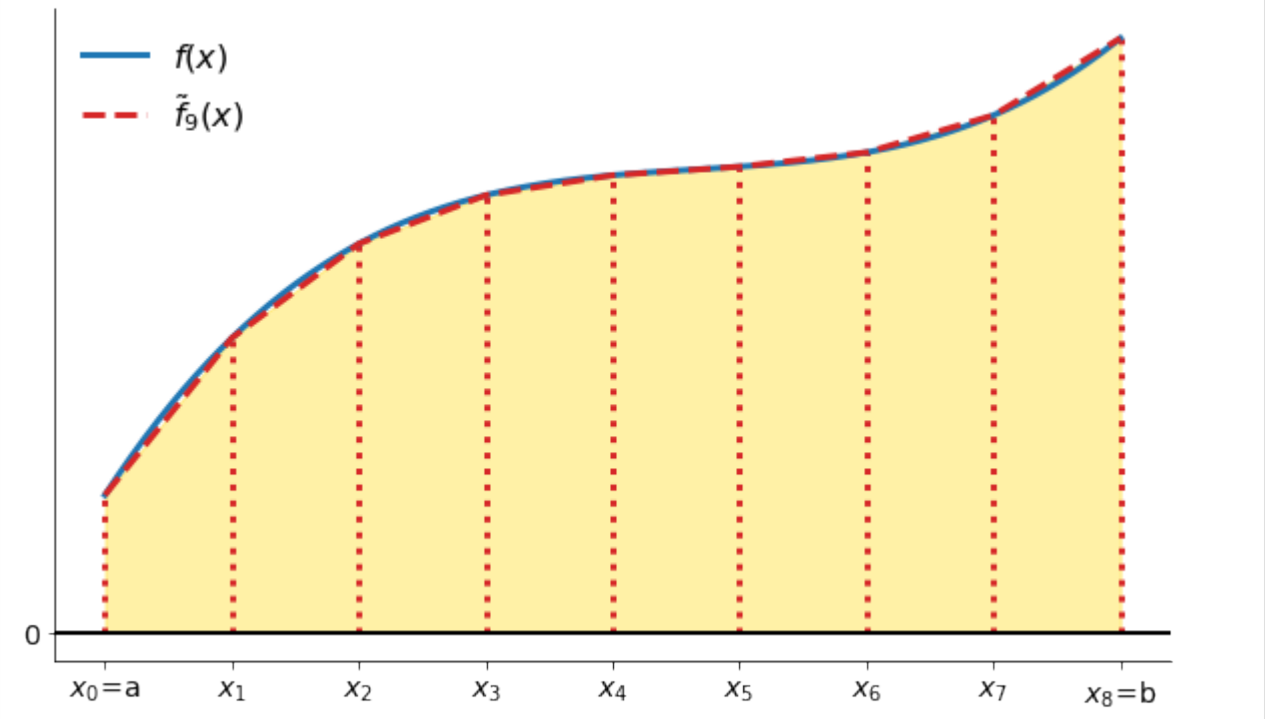
Trapezoid rule

The Trapezoid rule approximates the area under the function f with the area under a piecewise linear approximation to f.

[3]:
```
plot_trapezoid_rule_illustration()
```







More formally:

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + ... + \int_{x_{n-1}}^{x_n} f(x)dx$$
$$\approx h\frac{f(x_0) - f(x_1)}{2} + h\frac{f(x_1) - f(x_2)}{2} + ... + h\frac{f(x_{n-1}) - f(x_n)}{2},$$

which we can further simplify to:

$$\int_a^b f(x)dx \approx 0.5hf(x_0) + h\sum_{i=1}^{n-1} f(x_i) + 0.5hf(x_n).$$

The weights are then the following:

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

[4]:
```
??quadrature_newton_trapezoid_one
```

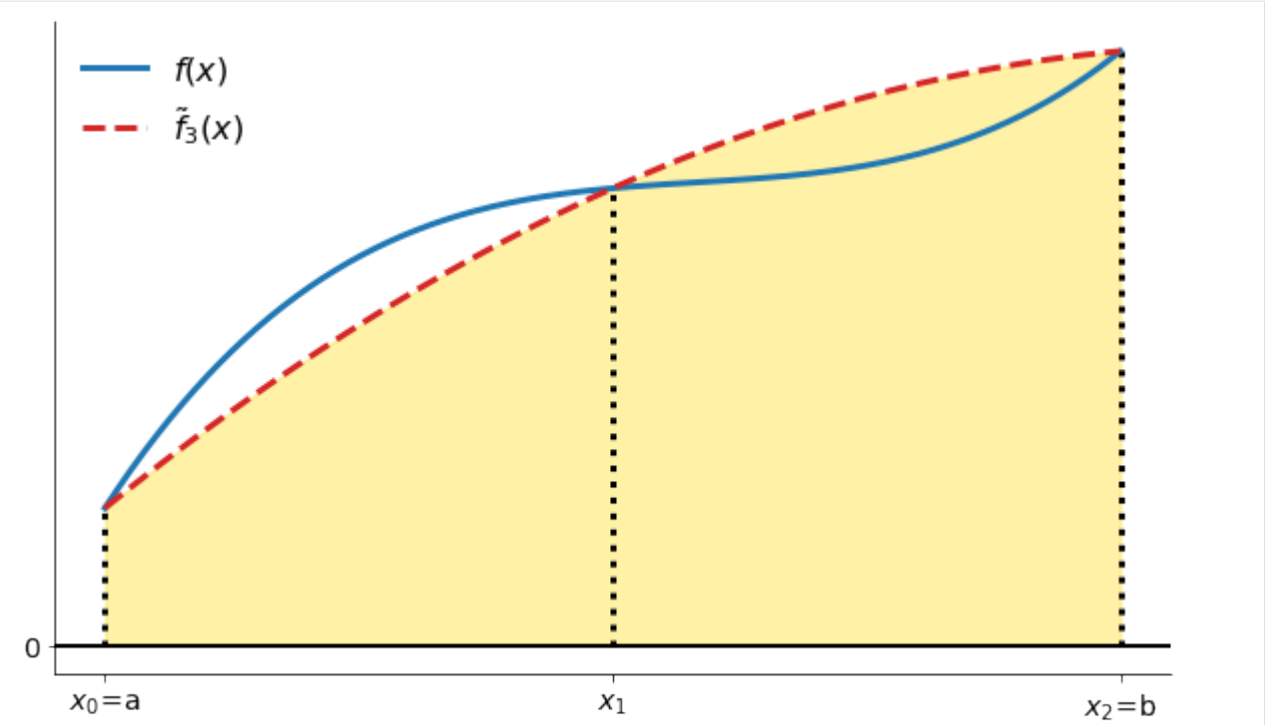
```
Signature: quadrature_newton_trapezoid_one(f, a, b, n)
Source:
def quadrature_newton_trapezoid_one(f, a, b, n):
    """Return quadrature newton trapezoid example."""
    xvals = np.linspace(a, b, n + 1)
    fvals = np.tile(np.nan, n + 1)
    h = xvals[1] - xvals[0]

    weights = np.tile(h, n + 1)
    weights[0] = weights[-1] = 0.5 * h

    for i, xval in enumerate(xvals):
        fvals[i] = f(xval)

    return np.sum(weights * fvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

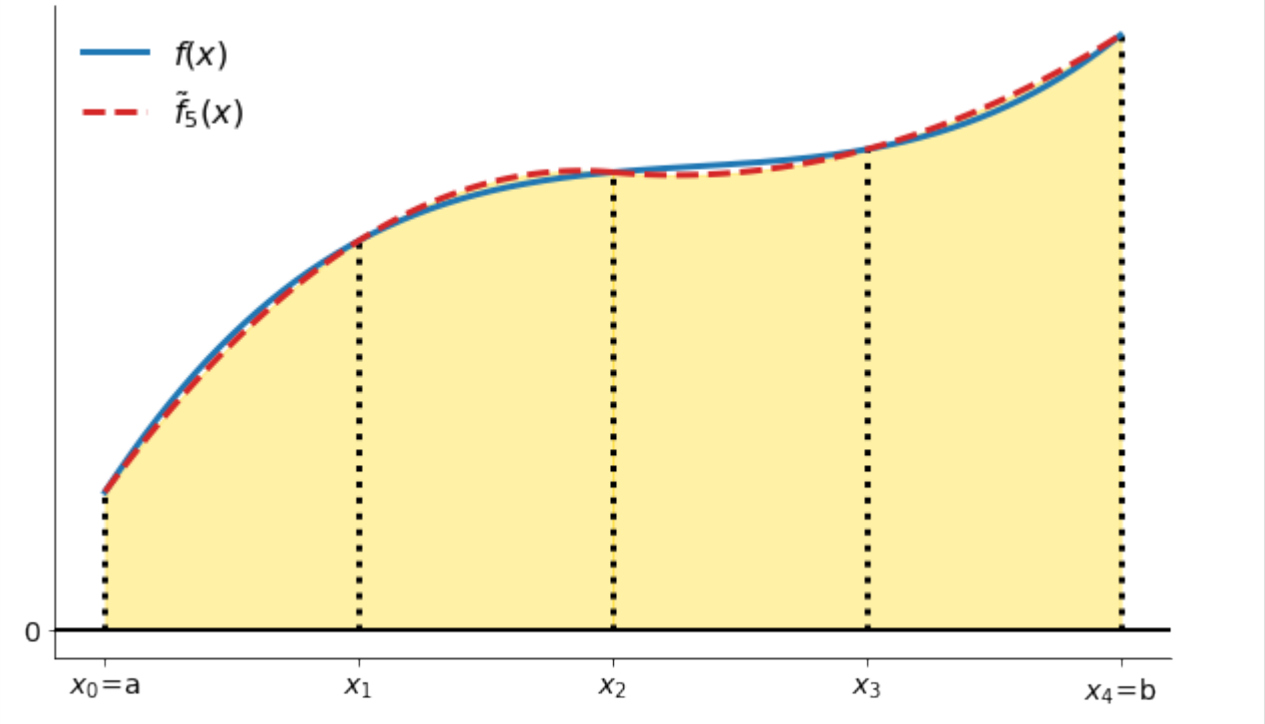
[5]:
```
integrand = quadrature_newton_trapezoid_one(np.exp, 0, 1, 1000)
np.testing.assert_almost_equal(integrand, np.exp(1) - 1)
```

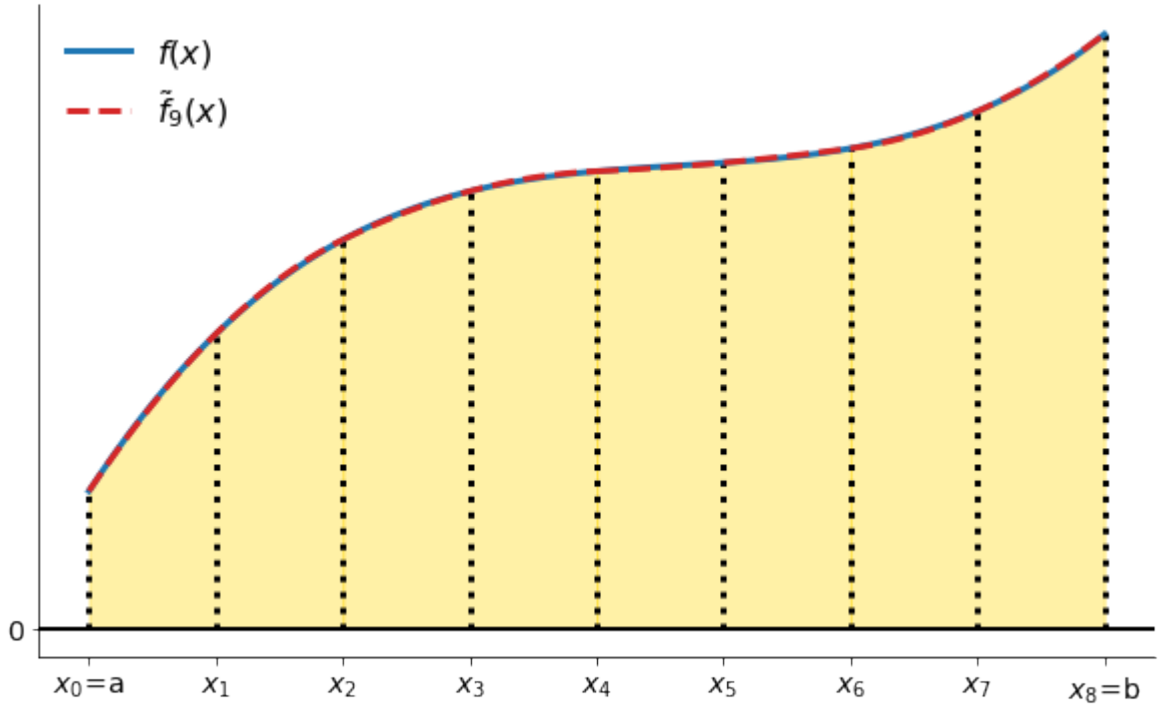
Simpson's rule

Simpson's rule approximates the area under a function f with the area under a piecewise quadratic approximation to f.

[6]:
```
plot_simpsons_rule_illustration()
```


f(x)
$\tilde{f}_3(x)$
0
$x_0=a$
x_1
$x_2=b$


f(x)
$\tilde{f}_5(x)$
0
$x_0=a$
x_1
x_2
x_3
$x_4=b$



The weights are the following:

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

[7]:
```
??quadrature_newton_simpson_one
```

```
Signature: quadrature_newton_simpson_one(f, a, b, n)
Source:
def quadrature_newton_simpson_one(f, a, b, n):
    """Return quadrature newton simpson example."""
    if n % 2 == 0:
        raise Warning("n must be an odd integer. Increasing by 1")
        n += 1

    xvals = np.linspace(a, b, n)
    fvals = np.tile(np.nan, n)

    h = xvals[1] - xvals[0]

    weights = np.tile(np.nan, n)
    weights[0::2] = 2 * h / 3
    weights[1::2] = 4 * h / 3
    weights[0] = weights[-1] = h / 3

    for i, xval in enumerate(xvals):
        fvals[i] = f(xval)
```

(continues on next page)

(continued from previous page)

```
    return np.sum(weights * fvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

[8]:
```
integrand = quadrature_newton_simpson_one(np.exp, 0, 1, 1001)
np.testing.assert_almost_equal(integrand, np.exp(1) - 1)
```

Gaussian formulas

Gaussian quadrature formulas employ a very 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 ω_i are chosen to exactly integrate polynomials of degree $2n - 1$ or less.

Gaussian quadrature approximations of the form:

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

for any nonnegative weighing function $w(x)$. Versions of $w(x)$ lead to include Gauss-Legendre, Gauss-Chebyshev, Gauss-Hermite and Gauss-Laguerre.

We start from the special domain between $[-1, 1]$, again the formula for our approximate solution looks very similar.

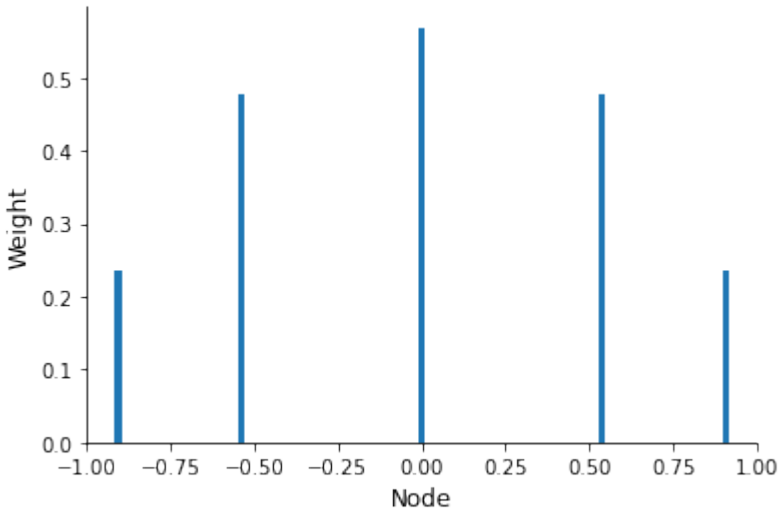
$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

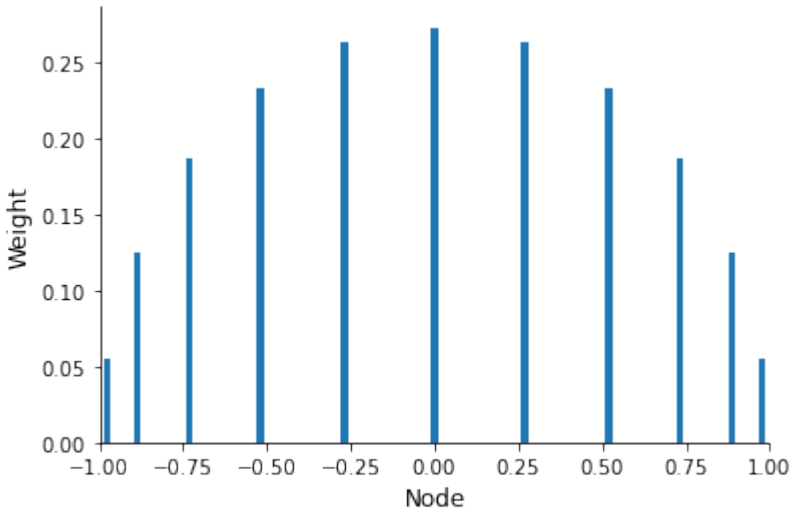
So, more generally using a change of variables:

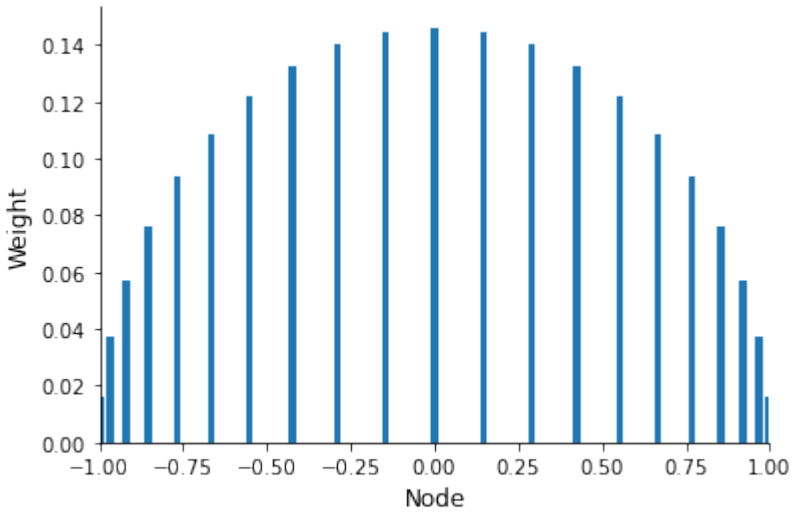
$$\int_a^b f(x) = \approx \frac{b-a}{2} \sum_{i=1}^{n} w_i f\left(\frac{(x_i+1)(b-a)}{2} + a\right)$$

Unlike in the case of the Newton-Cotes approach, Gaussian nodes are not uniformly spaced and do not include the integration limits.

[9]:
```
[plot_gauss_legendre_weights(deg) for deg in [5, 11, 21]]
```


Weight
0.0
0.1
0.2
0.3
0.4
0.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Node


Weight
0.00
0.05
0.10
0.15
0.20
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Node


Weight
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Node

[9]:
```
[None, None, None]
```

Question

- What are the properties of the weights?

[10]:
```
??quadrature_gauss_legendre_one
```

```
Signature: quadrature_gauss_legendre_one(f, a, b, n)
Source:
def quadrature_gauss_legendre_one(f, a, b, n):
    """Return quadrature gauss legendre example."""
    xvals, weights = np.polynomial.legendre.leggauss(n)
    xval_trans = (b - a) * (xvals + 1.0) / 2.0 + a

    fvals = np.tile(np.nan, n)
    for i, xval in enumerate(xval_trans):
        fvals[i] = ((b - a) / 2.0) * f(xval)

    return np.sum(weights * fvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

[11]:
```
integrand = quadrature_gauss_legendre_one(np.exp, 0, 1, 1000)
np.testing.assert_almost_equal(integrand, np.exp(1) - 1)
```

Benchmarking

Exercise

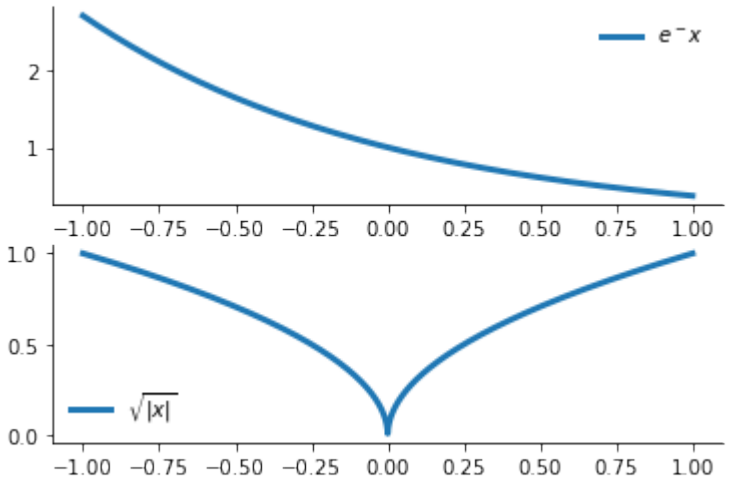
- Compare the performance of our integration routines on the following two integrals.

We study the following two cases:

$$f(x) = \int_{-1}^{1} e^{-x} dx$$

$$f(x) = \int_{-1}^{1} \sqrt{|x|} dx.$$

[12]:
```
plot_benchmarking_exercise()
```



How can we extend the ideas so far to multidimensional integration?

$$\int_x \int_y f(x, y) dx dy.$$

The product rule approximates the integral with the following sum:

$$\sum_{i_x=1}^{m} \sum_{i_y=1}^{m} \omega_{i_x} \omega_{i_y} f(x_1 y_1).$$

A difficulty of this approach is the curse of dimensionality as the number of nodes inreases exponentially with the number of dimensions.

[13]:
```
??quadrature_gauss_legendre_two
```

```
Signature: quadrature_gauss_legendre_two(f, a=-1, b=1, n=10)
Source:
def quadrature_gauss_legendre_two(f, a=-1, b=1, n=10):
    """Return quadrature gauss legendre example."""
    n_dim = int(np.sqrt(n))

    xvals, weight_uni = np.polynomial.legendre.leggauss(n_dim)
    xvals_transformed = (b - a) * (xvals + 1.0) / 2.0 + a

    weights = np.tile(np.nan, n_dim ** 2)
    fvals = np.tile(np.nan, n_dim ** 2)

    counter = 0
    for i, x in enumerate(xvals_transformed):
        for j, y in enumerate(xvals_transformed):
            weights[counter] = weight_uni[i] * weight_uni[j]
            fvals[counter] = f([x, y])
            counter += 1

    return ((b - a) / 2) ** 2 * np.sum(weights * np.array(fvals))
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

Monte Carlo integration

Naive Monte Carlo integration uses random sampling of a function to numerically compute an estimate of its integral. We can approximate this integral by averaging N samples of the function f drawn from a uniform distribution between a and b.

$$\int_a^b f(x)dx \approx (b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

Why?

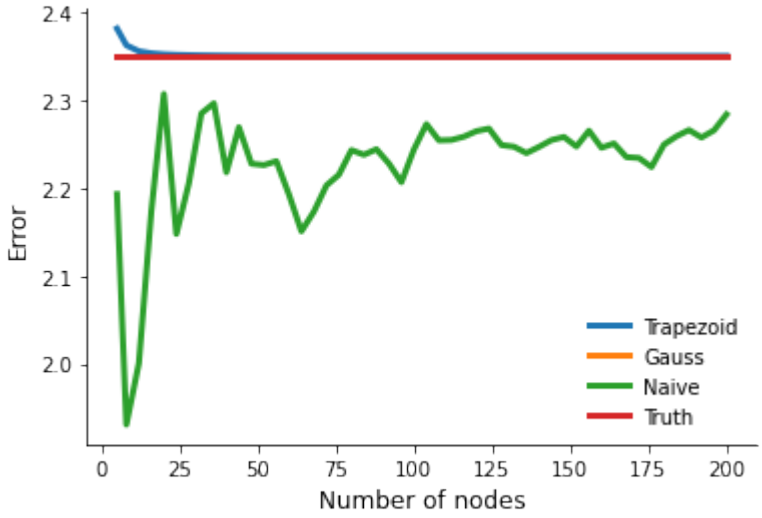
$$\begin{aligned} E\left[(b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right] &= (b-a)\frac{1}{N}\sum_{i=1}^{N} E\left[f(x_i)\right] \\ &= (b-a)\frac{1}{N}\sum_{i=1}^{N}\left[\int_a^b f(x)\frac{1}{b-a}dx\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)dx \\ &= \int_a^b f(x)dx. \end{aligned}$$

[14]:
```
??monte_carlo_naive_one
```

```
Signature: monte_carlo_naive_one(f, a=0, b=1, n=10, seed=123)
Source:
def monte_carlo_naive_one(f, a=0, b=1, n=10, seed=123):
    """Return naive monte carlo example."""
    np.random.seed(seed)
    xvals = np.random.uniform(size=n)
    fvals = np.tile(np.nan, n)
    weights = np.tile(1 / n, n)

    scale = b - a
    for i, xval in enumerate(xvals):
        fvals[i] = f(a + xval * (b - a))

    return scale * np.sum(weights * fvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

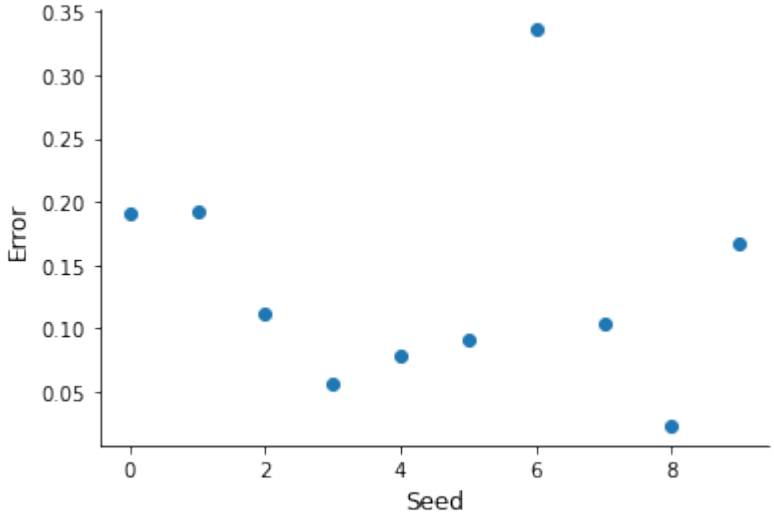
Monte Carlo integration only converges very slowly. Let's consider our smooth benchmarking example from earlier and compare the naive Monte Carlo approach to the quadrature methods.

[15]:
```
plot_naive_monte_carlo_error(max_nodes=200)
```



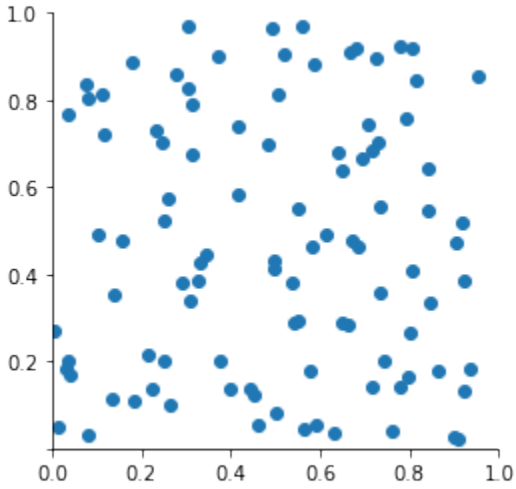
Naive Monte Carlo integration differs from our earlier methods as the approximation itself is a random variable. How about the level of randomness at a given number $(n = 50)$ of integration nodes?

[16]:
```
plot_naive_monte_carlo_randomness()
```



However, Monte Carlo integration is particular useful when tackling multidimensional integrals. Let's consider the computation of a two dimensional integral going forward.

[17]:
```
plot_naive_monte_carlo(100)
```

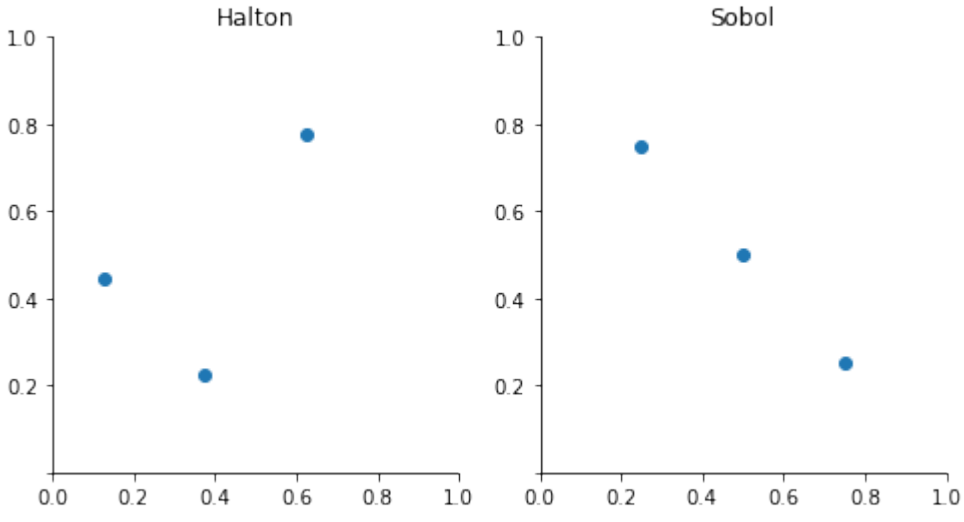


Quasi-Monte Carlo employ deterministic sequences of nodes x_i with the property that

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

Deterministic sequences of nodes chosen to fill space in a regular manner typically provide more accurate integration approximations than pseudo-random sequences. These sequences are readily available in the chaospy package. The documentation of the package also includes a tutorial on Monte Carlo integration.

[18]:
```
plot_quasi_monte_carlo(3)
```



Otherwise, all looks pretty similar to our earlier implementation. Note that the default rule actually takes us back to the naive Monte Carlo method.

[19]:
```
??monte_carlo_quasi_two_dimensions
```

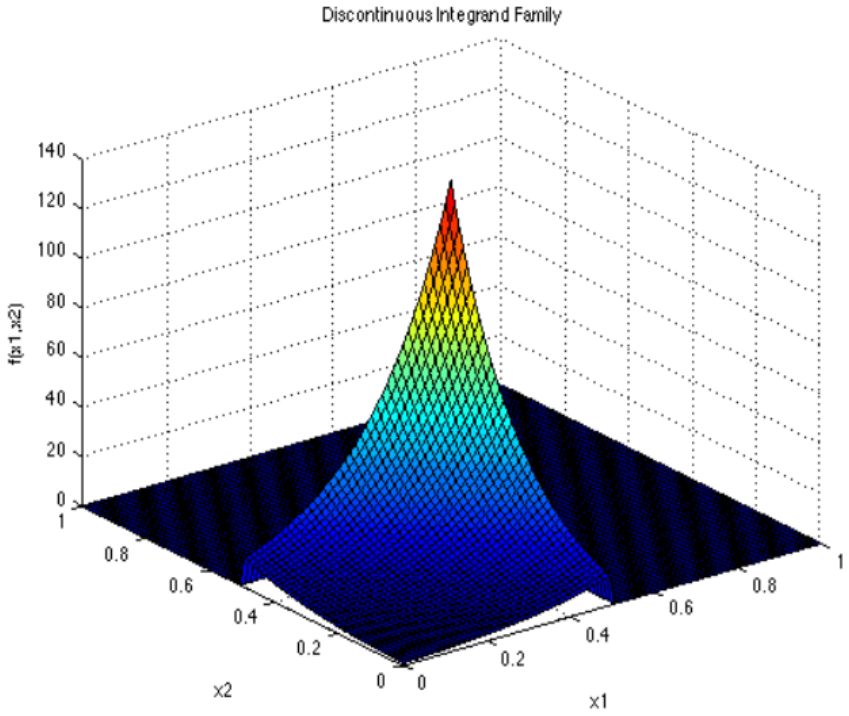
```
Signature: monte_carlo_quasi_two_dimensions(f, a=0, b=1, n=10, rule='random')
Source:
def monte_carlo_quasi_two_dimensions(f, a=0, b=1, n=10, rule="random"):
    """Return Monte Carlo example (two-dimensional).
```

(continues on next page)

(continued from previous page)

```
    Corresponds to naive Monthe Carlo for `rule='random'`. Restricted to same
    integration domain for both variables.
    """
    distribution = cp.J(cp.Uniform(a, b), cp.Uniform(a, b))
    samples = distribution.sample(n, rule=rule).T
    volume = (b - a) ** 2

    fvals = np.tile(np.nan, n)
    weights = np.tile(1 / n, n)

    for i, xval in enumerate(samples):
        fvals[i] = f(xval)

    return volume * np.sum(weights * fvals)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/lectures/integration/integration_algorithms.py
Type:      function
```

Let's consider the following test function from Genz (1984).



$$f(\mathbf{x}) = \begin{cases} 0, & \text{if } x_1 > u_1 \text{ or } x_2 > u_2 \\ \exp\left(\sum_{i=1}^{d} a_i x_i\right), & \text{otherwise} \end{cases}$$

Exercises

1. What are the properties of the function?
2. Compare the performance naive and quasi Monte Carlo integration routines for $u = (0.5, 0.5)$ and $a = (5, 5)$ over the two-dimensional unit cube.
3. How does Gauss-Legendre quadrature perform?

Applying the integration routines to functions with other input arguments can be achieved with some small modifications. Here we pass in our benchmarking function without any default arguments to a modified naive Monte Carlo integration routine.

[20]:
```
def genz_discontinuous_no_defaults(x, u, a):
    if x[0] > u[0] or x[1] > u[1]:
        return 0
    else:
        return np.exp((a * x).sum())


def monte_carlo_naive_with_args(f, args=(), a=0, b=1, n=10, seed=128):
```

(continues on next page)

(continued from previous page)

```
    np.random.seed(seed)
    xvals = np.random.uniform(low=a, high=b, size=2 * n).reshape(n, 2)
    volume = (b - a) ** 2

    fvals = np.tile(np.nan, n)
    weights = np.tile(1 / n, n)

    for i, xval in enumerate(xvals):
        # Here is the main modification that passes in the arguments by position now.
        fvals[i] = f(xval, *args)

    return volume * np.sum(weights * fvals)


u, a = (0.5, 0.5), (5, 5)
rslt = monte_carlo_naive_with_args(genz_discontinuous_no_defaults, args=(u, a))
```

Resources

Research

- https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1870703
- https://www.sciencedirect.com/science/article/abs/pii/S0304407607002552
- https://arxiv.org/abs/1908.04110
- Philip J. Davis and Philip Rabinowitz, Methods of Numerical Integration.
- https://www.sciencedirect.com/science/article/pii/S0021999185712090
- https://ins.uni-bonn.de/media/public/publication-media/sparse_grids_nutshell.pdf?pk=639

Other resources

- http://hplgit.github.io/prog4comp/doc/pub/p4c-sphinx-Python/._pylight004.html
- http://people.duke.edu/~ccc14/cspy/15C_MonteCarloIntegration.html
- https://www.math.ubc.ca/~pwalls/math-python/integration/integrals/
- https://guido.vonrudorff.de/wp-content/uploads/2020/05/NumericalIntegration.pdf
- https://readthedocs.org/projects/mec-cs101-integrals/downloads/pdf/latest/

References

- Genz, A. (1984, September). Testing multidimensional integration routines. In Proc. of international conference on Tools, methods and languages for scientific and engineering computation (pp. 81-94). Elsevier North-Holland, Inc..

1.5.2 Functions

This module contains the algorithms for the integration lab.

`labs.integration.integration_algorithms.`**`monte_carlo_naive_one`**(*f*, *a=0*, *b=1*, *n=10*, *seed=123*)
Return naive monte carlo example.

`labs.integration.integration_algorithms.`**`monte_carlo_naive_two_dimensions`**(*f*, *a=0*, *b=1*, *n=10*, *seed=128*)
Return naive monte carlo example (two-dimensional).

Restricted to same integration domain for both variables.

`labs.integration.integration_algorithms.`**`monte_carlo_quasi_two_dimensions`**(*f*, *a=0*, *b=1*, *n=10*, *rule='random'*)
Return Monte Carlo example (two-dimensional).

Corresponds to naive Monthe Carlo for *rule='random'*. Restricted to same integration domain for both variables.

`labs.integration.integration_algorithms.`**`quadrature_gauss_legendre_one`**(*f*, *a*, *b*, *n*)
Return quadrature gauss legendre example.

`labs.integration.integration_algorithms.`**`quadrature_gauss_legendre_two`**(*f*, *a=- 1*, *b=1*, *n=10*)
Return quadrature gauss legendre example.

`labs.integration.integration_algorithms.`**`quadrature_newton_simpson_one`**(*f*, *a*, *b*, *n*)
Return quadrature newton simpson example.

`labs.integration.integration_algorithms.`**`quadrature_newton_trapezoid_one`**(*f*, *a*, *b*, *n*)
Return quadrature newton trapezoid example.

1.6 Approximation

We study the function approximation using polynomials. We combine different strategies for the interpolation nodes and basis functions to study how they interact to determine the approximation's overall quality. We use this as an opportunity to iteratively develop a function that allows to combine the different ingredients to set up an interpolator. Finally, we extend the ideas to the case of multivariate interpolation.

1.6.1 Approximation

Outline

1. Setup
2. Polynomial interpolation
3. Resources

Setup

In many computational economics applications, we need to replace an analytically intractable function $f : R^n \to R$ with a numerically tractable approximation $\hat{f}$. In some applications, f can be evaluated at any point of its domain, but with difficulty, and we wish to replace it with an approximation $\hat{f}$ that is easier to work with.

We study interpolation, a general strategy for forming a tractable approximation to a function that can be evaluated at any point of its domain. Consider a real-valued function f defined on an interval of the real line that can be evaluated at any point of its domain.

Generally, we will approximate f using a function $\hat{f}$ that is a finite linear combination of n known basis functions $\phi_1, \phi_2, ..., \phi_n$ of our choosing:

$$f(x) \approx \hat{f}(x) \equiv \sum_{j=1}^{n} c_j \phi_j(x).$$

We will fix the n basis coefficients $c_1, c_2, ..., c_n$ by requiring $\hat{f}$ to interpolate, that is, agree with f , at n interpolation nodes $x_1, x_2, ..., x_n$ of our choosing.

The most readily recognizable basis is the monomial basis:

$$\begin{aligned}
\phi_0(x) &= 1 \\
\phi_1(x) &= x \\
\phi_2(x) &= x^2 \\
&\cdot \\
\phi_n(x) &= x^n.
\end{aligned}$$

This can be used to construct the polynomial approximations:

$$f(x) \approx \hat{f}(x) \equiv c_0 + c_1 x + c_2 x^2 + ...c_n x^n$$

There are other basis functions with more desirable properties and there are many different ways to choose the interpolation nodes.

Regardless of how the n basis functions and nodes are chosen, computing the basis coefficients reduces to solving a linear equation.

$$\sum_{j=1}^{n} c_j \phi_j(x) = f(x), \qquad i = 1, ..., n$$

Interpolation schemes differ only in how the basis functions ϕ_j and interpolation nodes x_j are chosen.

[1]:
```
from functools import partial
from temfpy.interpolation import runge
import warnings

from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
import numpy as np

from approximation_algorithms import get_interpolator_flexible_basis_flexible_nodes
from approximation_algorithms import get_interpolator_monomial_flexible_nodes
from approximation_algorithms import get_interpolator_monomial_uniform
from approximation_algorithms import get_interpolator_runge_baseline
```

(continues on next page)

(continued from previous page)

```
from approximation_auxiliary import compute_interpolation_error
from approximation_auxiliary import get_chebyshev_nodes
from approximation_auxiliary import get_uniform_nodes

from approximation_plots import plot_two_dimensional_problem
from approximation_plots import plot_reciprocal_exponential
from approximation_plots import plot_runge_different_nodes
from approximation_plots import plot_runge_function_cubic
from approximation_plots import plot_two_dimensional_grid
from approximation_plots import plot_approximation_nodes
from approximation_plots import plot_basis_functions
from approximation_plots import plot_runge_multiple
from approximation_plots import plot_runge

from approximation_problems import problem_reciprocal_exponential
from approximation_problems import problem_two_dimensions
```

Polynomial interpolation

A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponentiation of variables.

The Weierstrass Theorem asserts that any continuous real-valued function can be approximated to an arbitrary degree of accuracy over a bounded interval by a polynomial.

Specifically, if f is continuous on $[a, b]$ and $\epsilon > 0$, then there exists a polynomial p such that

$$\max_{x \in [a,b]} |f(x) - p(x)| < \epsilon$$

- How to find a polynomial that provides a desired degree of accuracy?
- What degree of the polynomial is required?

Naive polynomial interpolation

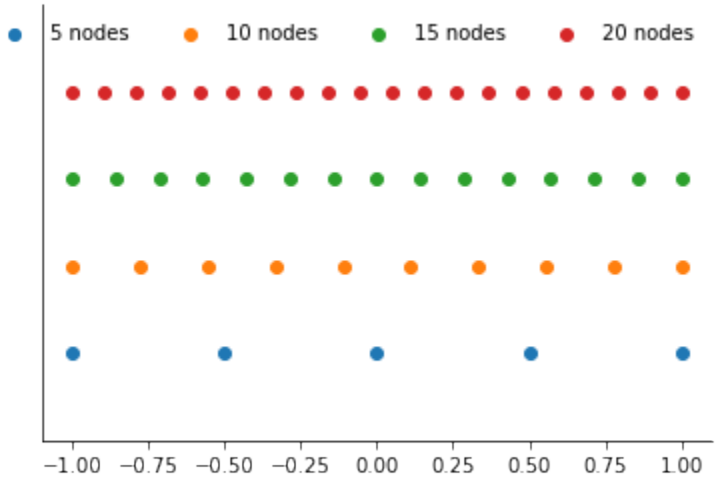
Let's start with a basic setup, where we use a uniform grid and monomial basis functions.

$$\hat{f}(x) \equiv \sum_{j=0}^{n} c_j x^j$$

[2]:
```
??get_uniform_nodes
```

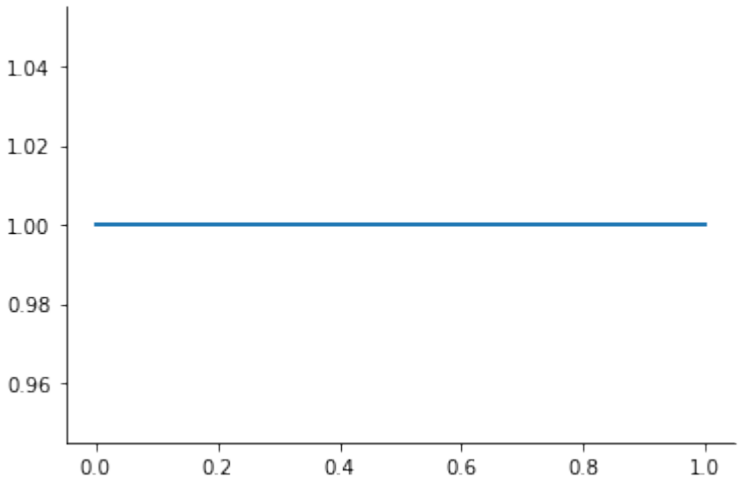
```
Signature: get_uniform_nodes(n, a=-1, b=1)
Source:
def get_uniform_nodes(n, a=-1, b=1):
    """Return uniform nodes."""
    return np.linspace(a, b, num=n)
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_auxiliary.py
Type:      function
```

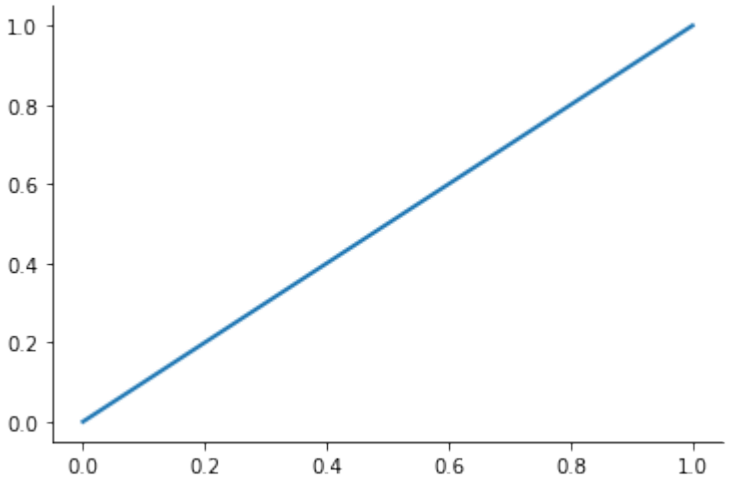
[3]:
```
plot_approximation_nodes([5, 10, 15, 20], nodes="uniform")
```

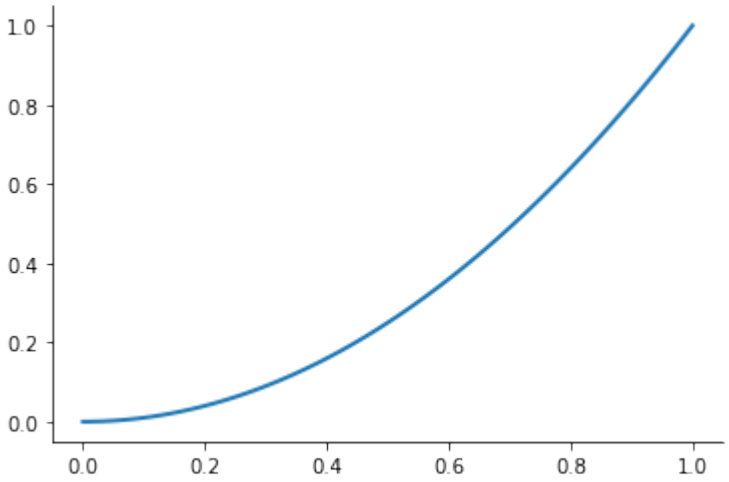


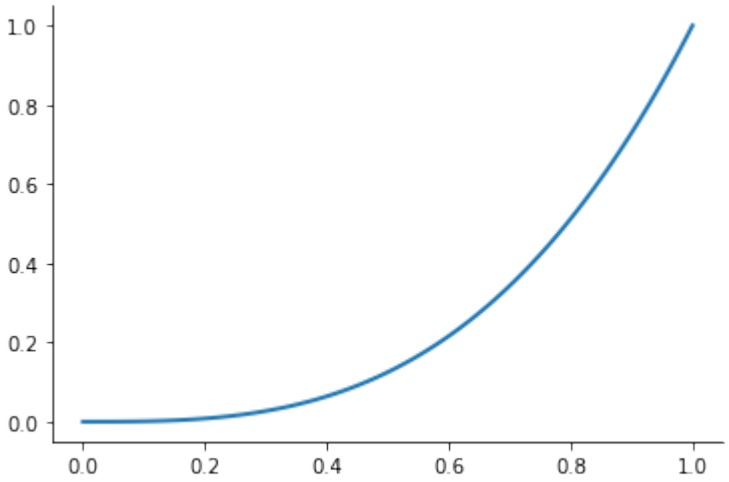
Now we can get a look at the interpolation nodes.

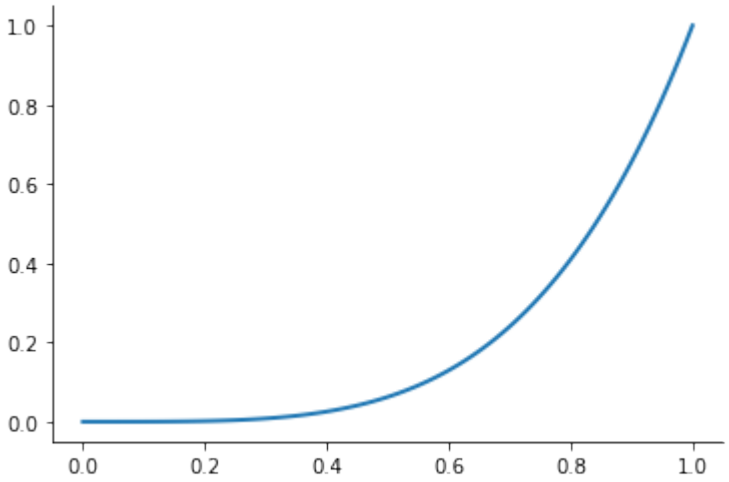
[4]:
```
plot_basis_functions("monomial")
```

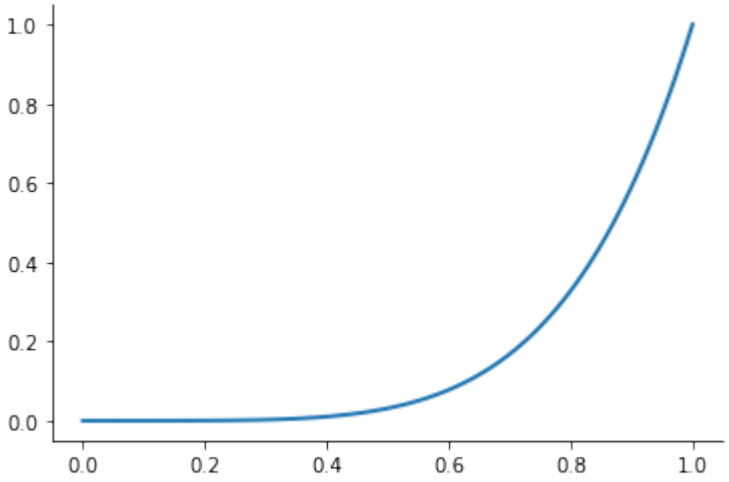





1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

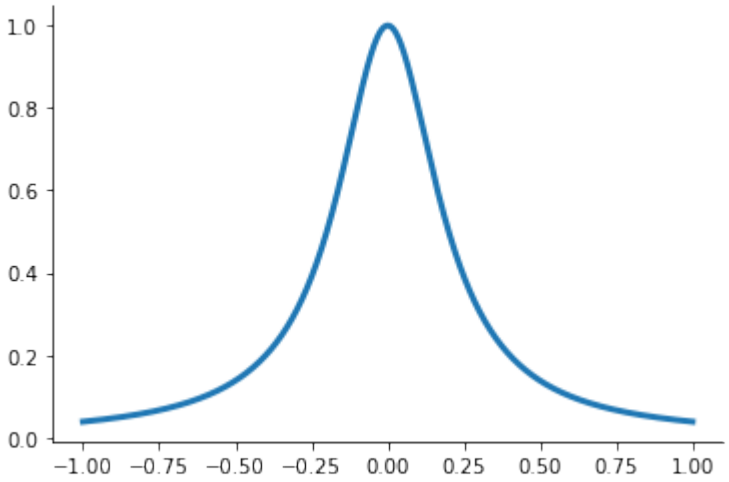

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



Let's look at the performance of this approach for the Runge function for $x \in [0, 1]$.

$$f(x) = \frac{1}{(1 + 25x^2)}$$

[5]:
```
plot_runge()
```



Due to its frequent use, `numpy` does offer a convenience class to work with polynomials. See here for its documentation.

[6]:
```
from numpy.polynomial import Polynomial as P  # noqa E402
from numpy.polynomial import Chebyshev as C  # noqa E402
```

We will use the attached methods to develop a flexible interpolation set in an iterative fashion.'

[7]:
```
??get_interpolator_runge_baseline
```

```
Signature: get_interpolator_runge_baseline(func)
Source:
def get_interpolator_runge_baseline(func):
    """Return interpolator runge function (baseline)."""
```

(continues on next page)

(continued from previous page)

```
    xnodes = np.linspace(-1, 1, 5)
    poly = np.polynomial.Polynomial.fit(xnodes, func(xnodes), 5)
    return poly
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_algorithms.py
Type:      function
```

[8]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    interpolant = get_interpolator_runge_baseline(runge)
    xvalues = np.linspace(-1, 1, 10000)
    yfit = interpolant(xvalues)
```

Question

- Why the warnings?

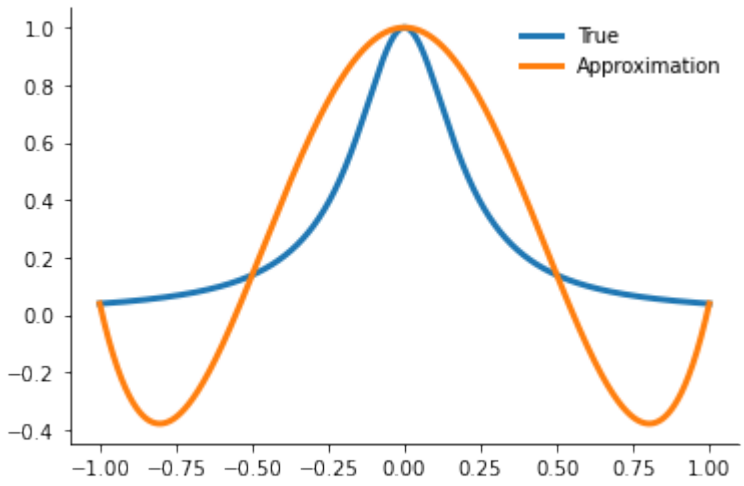
Since we have a good understanding what is causing the warning, we can simply turn it of going forward. A documentation that shows how to deal with more fine-grained filters is available here.

[9]:
```
warnings.simplefilter("ignore")
```

Now we are read to plot it against the true function.

[10]:
```
fig, ax = plt.subplots()
ax.plot(xvalues, runge(xvalues), label="True")
ax.plot(xvalues, yfit, label="Approximation")
ax.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7fc295f08f70>
```



We evaluate the error in our approximation by the the following statistic.

[11]:
```
??compute_interpolation_error
```

```
Signature: compute_interpolation_error(error)
Source:
def compute_interpolation_error(error):
    """Compute interpolation error."""
    return np.log10(np.linalg.norm(error, np.inf))
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_auxiliary.py
Type:      function
```

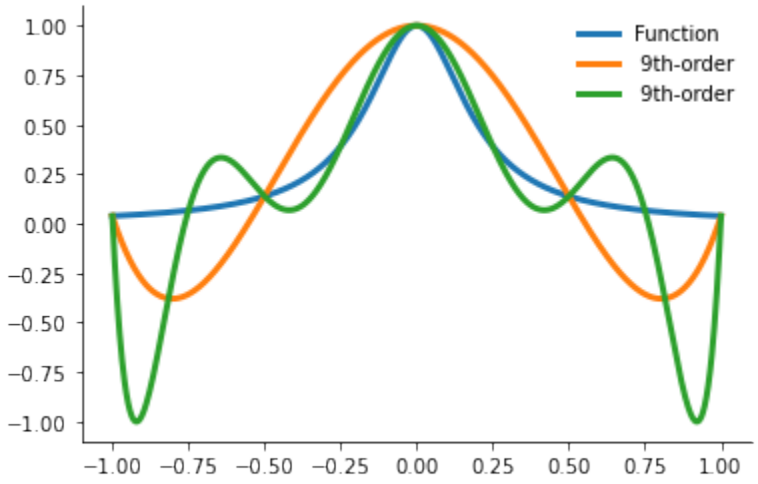
[12]:
```
compute_interpolation_error(yfit - runge(xvalues))
```

[12]:
```
-0.35817192020499683
```

Exercises

1. Generalize the function to allow to approximate the function with a polynomial of generic degree.
2. How does the quality of the approximation change as we increase the number of interpolation points?

[13]:
```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    plot_runge_multiple()
```



What can be done? First we explore a different way to choose the the nodes.

Theory asserts that the best way to approximate a continuous function with a polynomial over a bounded interval $[a, b]$ is to interpolate it at so called Chebychev nodes:

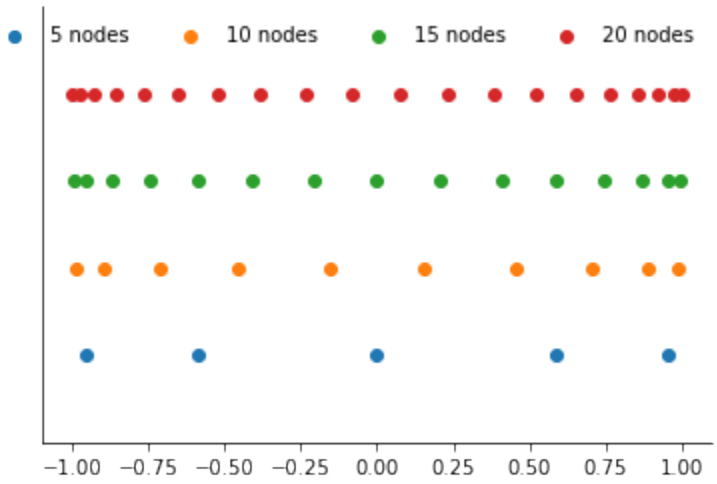
$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{n-i+0.5}{n}\pi\right)$$

[14]:
```
??get_chebyshev_nodes
```

```
Signature: get_chebyshev_nodes(n, a=-1, b=1)
Source:
def get_chebyshev_nodes(n, a=-1, b=1):
    """Return Chebyshev nodes."""
    nodes = np.tile(np.nan, n)

    for i in range(1, n + 1):
        nodes[i - 1] = (a + b) / 2 + ((b - a) / 2) * np.cos(((n - i + 0.5) / n) * np.pi)

    return nodes
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_auxiliary.py
Type:      function
```

Let's look at a visual representation.

[15]:
```
plot_approximation_nodes([5, 10, 15, 20], nodes="chebychev")
```



The Chebychev nodes are not evenly spaced and do not include the endpoints of the approximation interval. They are more closely spaced near the endpoints of the approximation interval and less so near the center.

If f is continuous ...

- Rivlin's Theorem asserts that Chebychev-node polynomial interpolation is nearly optimal, that is, it affords an approximation error that is very close to the lowest error attainable with another polynomial of the same degree.
- Jackson's Theorem asserts that Chebychev-node polynomial interpolation is consistent, that is, the approximation error vanishes as the degree of the polynomial increases.

[16]:
```
??get_interpolator_monomial_flexible_nodes
```

```
Signature:
get_interpolator_monomial_flexible_nodes(
    func,
    degree,
    nodes='uniform',
    a=-1,
    b=1,
```

(continues on next page)

(continued from previous page)

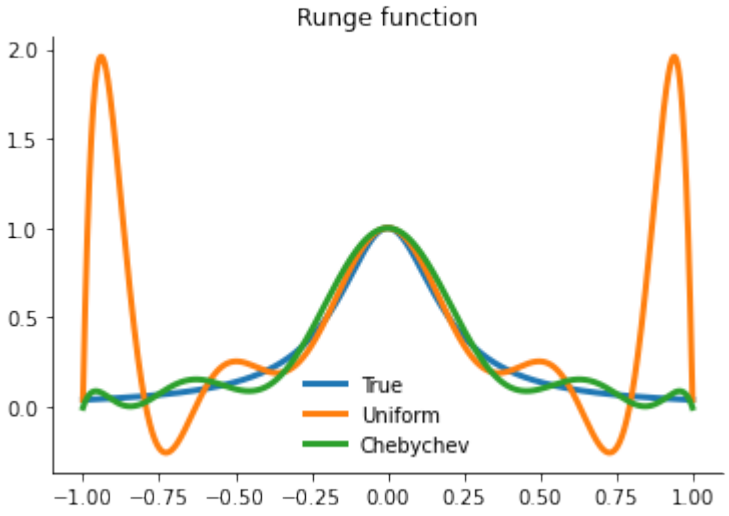
```
)
Source:
def get_interpolator_monomial_flexible_nodes(func, degree, nodes="uniform", a=-1, b=1):
    """Return monomial function (flexible nodes)."""
    if nodes == "uniform":
        get_nodes = get_uniform_nodes
    elif nodes == "chebychev":
        get_nodes = get_chebyshev_nodes

    xnodes = get_nodes(degree, a, b)
    poly = np.polynomial.Polynomial.fit(xnodes, func(xnodes), degree)

    return poly
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_algorithms.py
Type:      function
```

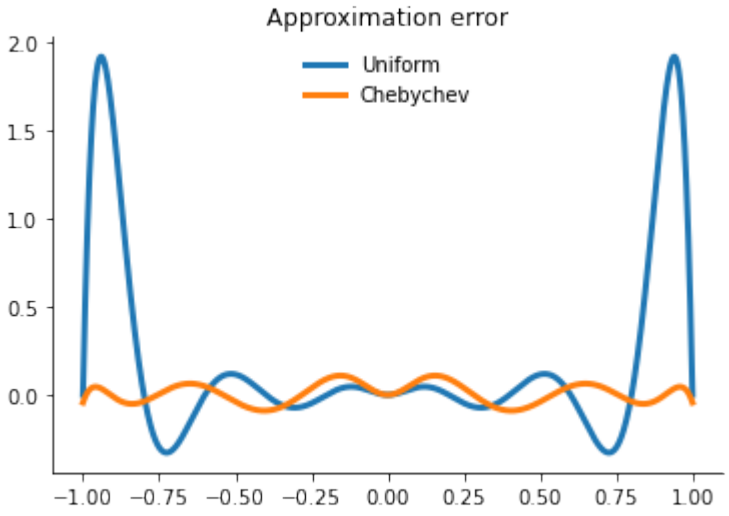
[17]:
```
intertp = get_interpolator_monomial_flexible_nodes(runge, 11, nodes="chebychev")
intertp(np.linspace(-1, 1, 10))
```

[17]:
```
array([-0.00532688,  0.04177453,  0.12887422,  0.20826539,  0.85508579,
        0.85508579,  0.20826539,  0.12887422,  0.04177453, -0.00532688])
```

Let's compare the performance of the two approaches.

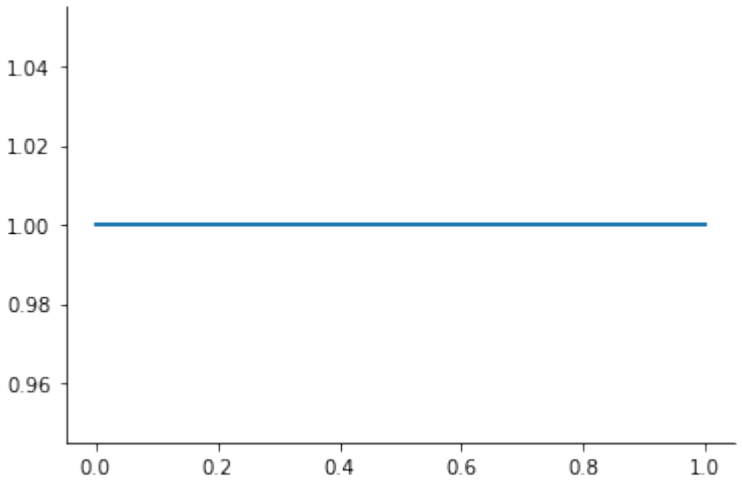
[18]:
```
plot_runge_different_nodes()
```

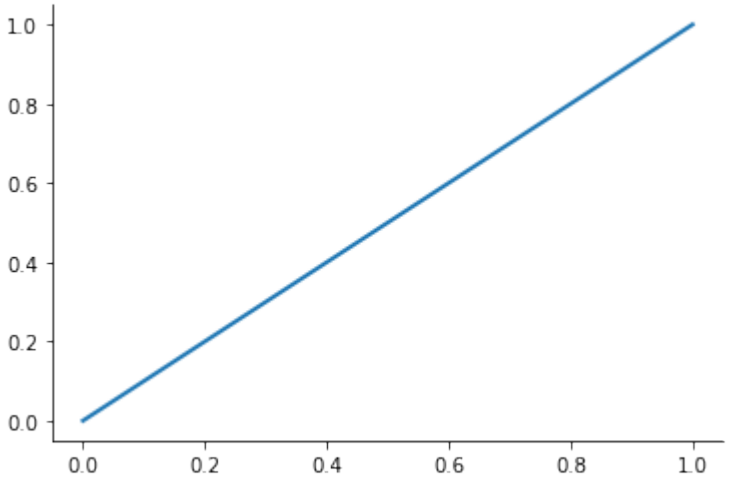


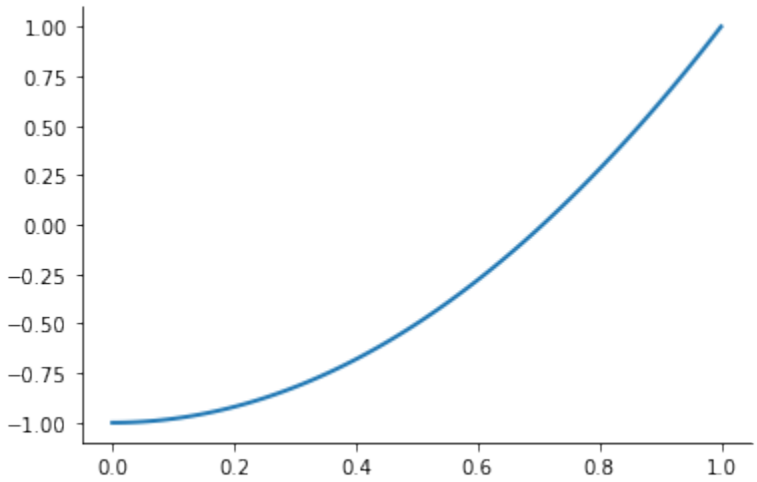


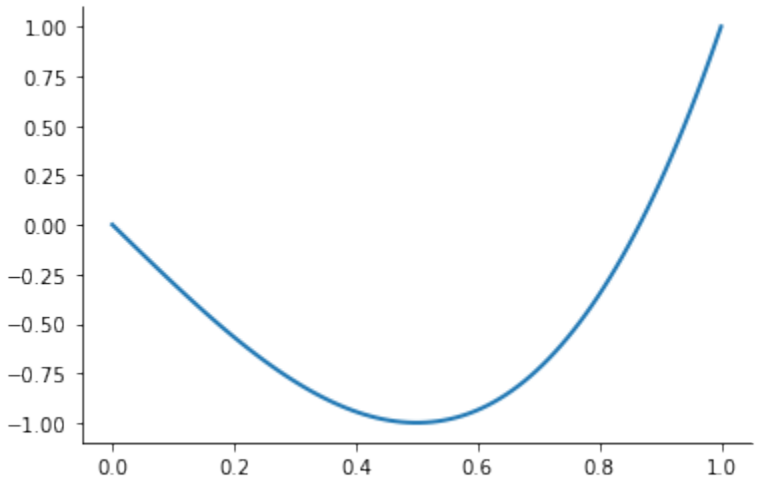
However, merely interpolating at the Chebychev nodes does not eliminate ill-conditioning. Ill-conditioning stems from the choice of basis functions, not the choice of interpolation nodes. Fortunately, there is alternative to the monomial basis that is ideal for expressing Chebychev-node polynomial interpolants. The optimal basis for expressing Chebychev-node polynomial interpolants is called the Chebychev polynomial basis.

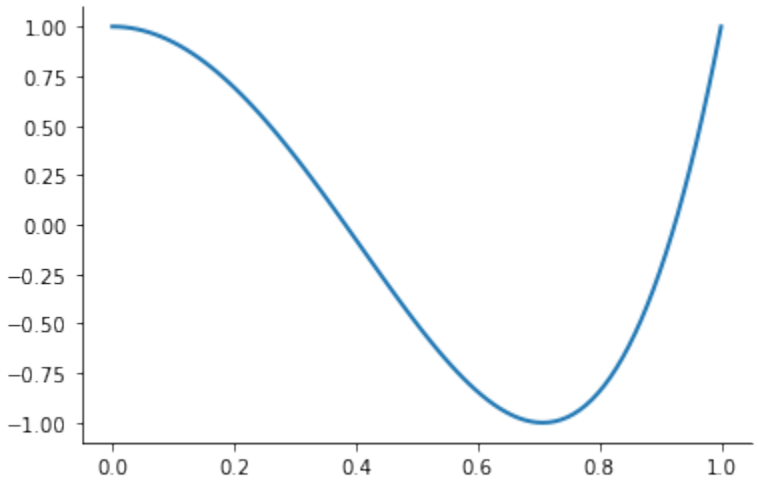
[19]:
```
plot_basis_functions("chebychev")
```

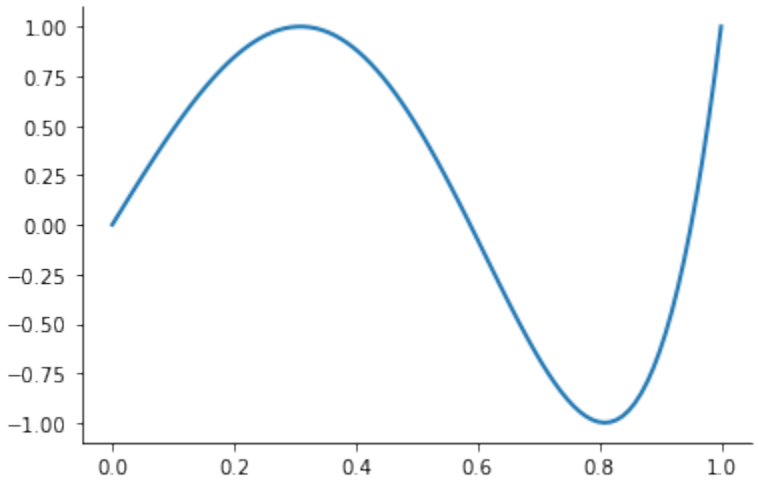



1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0





Combining the Chebychev basis polynomials and Chebychev interpolation nodes yields an extremely well-conditioned interpolation equation and allows to approximate any continuous function to high precision. Let's put it all together now.

[20]:
```
??get_interpolator_flexible_basis_flexible_nodes
```

```
Signature:
get_interpolator_flexible_basis_flexible_nodes(
    func,
    degree,
    basis='monomial',
    nodes='uniform',
    a=-1,
    b=1,
)
Source:
def get_interpolator_flexible_basis_flexible_nodes(
    func, degree, basis="monomial", nodes="uniform", a=-1, b=1
):
    """Return interpolator (flexible basis, flexible nodes)."""
    if nodes == "uniform":
```

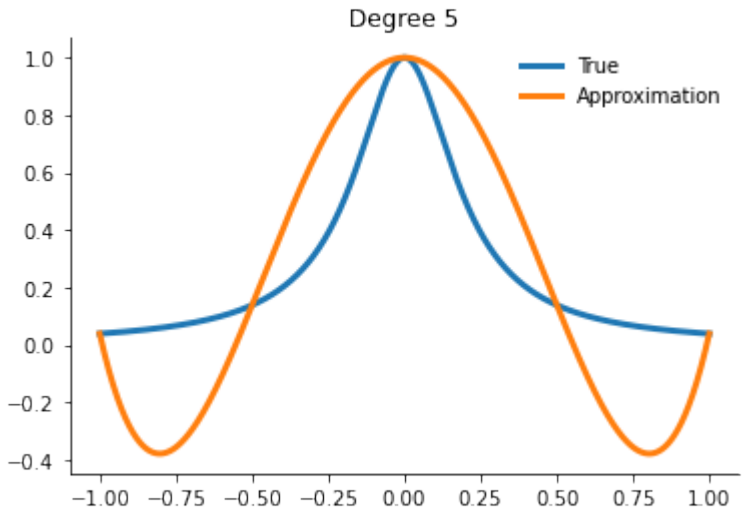
(continues on next page)

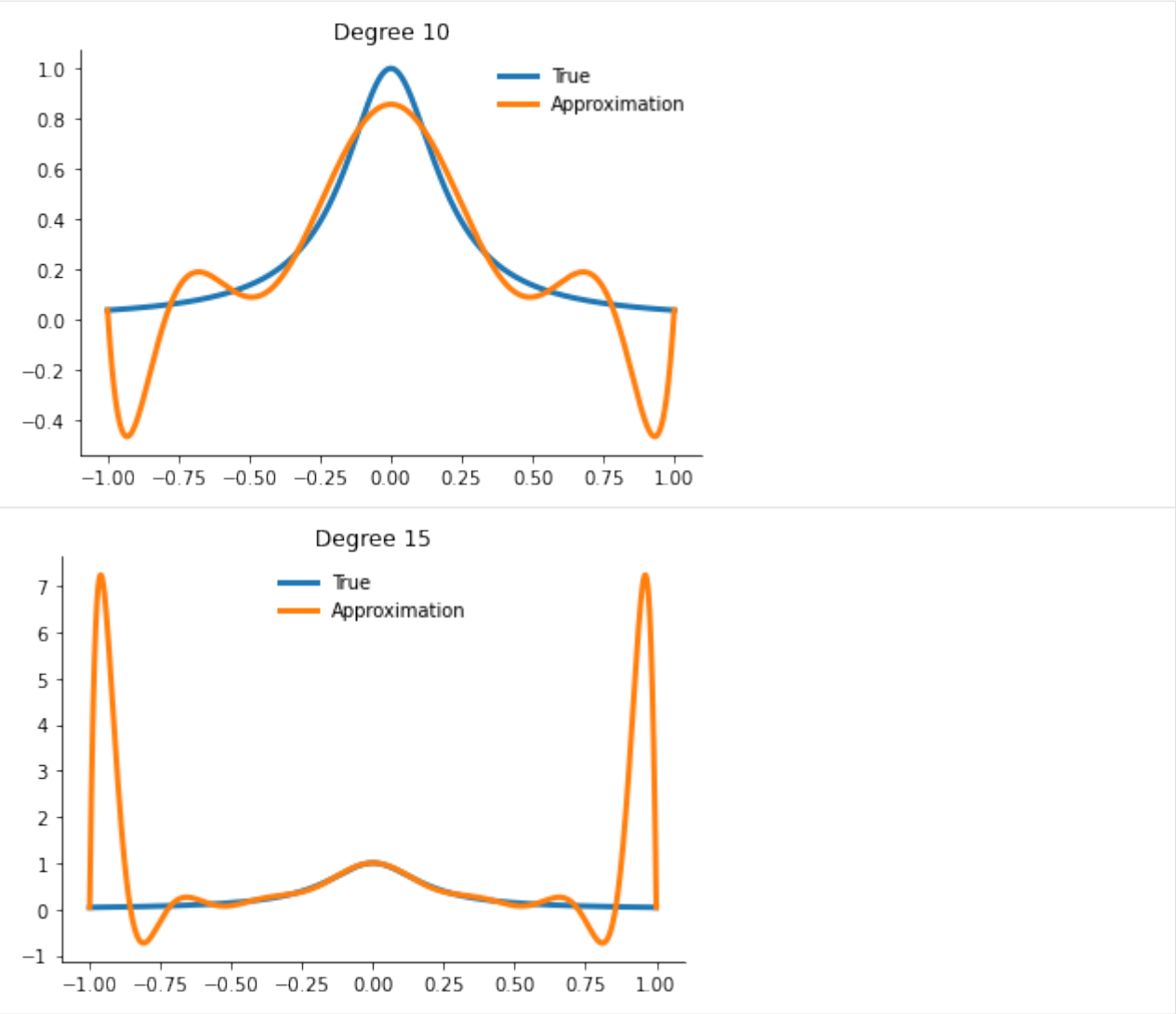
(continued from previous page)

```
        get_nodes = get_uniform_nodes
    elif nodes == "chebychev":
        get_nodes = get_chebyshev_nodes

    if basis == "monomial":
        fit = np.polynomial.Polynomial.fit
    elif basis == "chebychev":
        fit = np.polynomial.Chebyshev.fit

    xnodes = get_nodes(degree, a, b)
    poly = fit(xnodes, func(xnodes), degree)

    return poly
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_algorithms.py
Type:      function
```

How well can we actually do now?

[21]:
```
for degree in [5, 10, 15]:
    interp = get_interpolator_flexible_basis_flexible_nodes(
        runge, degree, nodes="uniform", basis="monomial"
    )
    xvalues = np.linspace(-1, 1, 10000)
    yfit = interp(xvalues)

    fig, ax = plt.subplots()
    ax.plot(xvalues, runge(xvalues), label="True")
    ax.plot(xvalues, yfit, label="Approximation")
    ax.legend()
    ax.set_title(f"Degree {degree}")
```

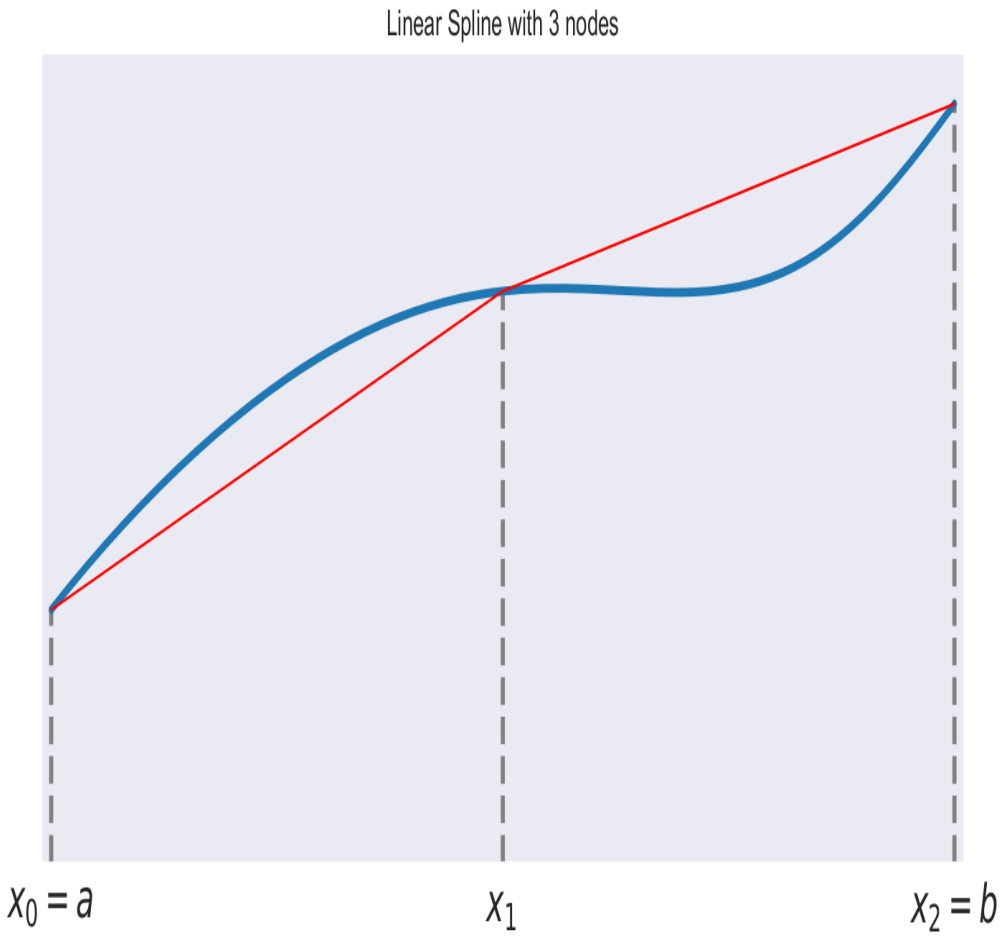


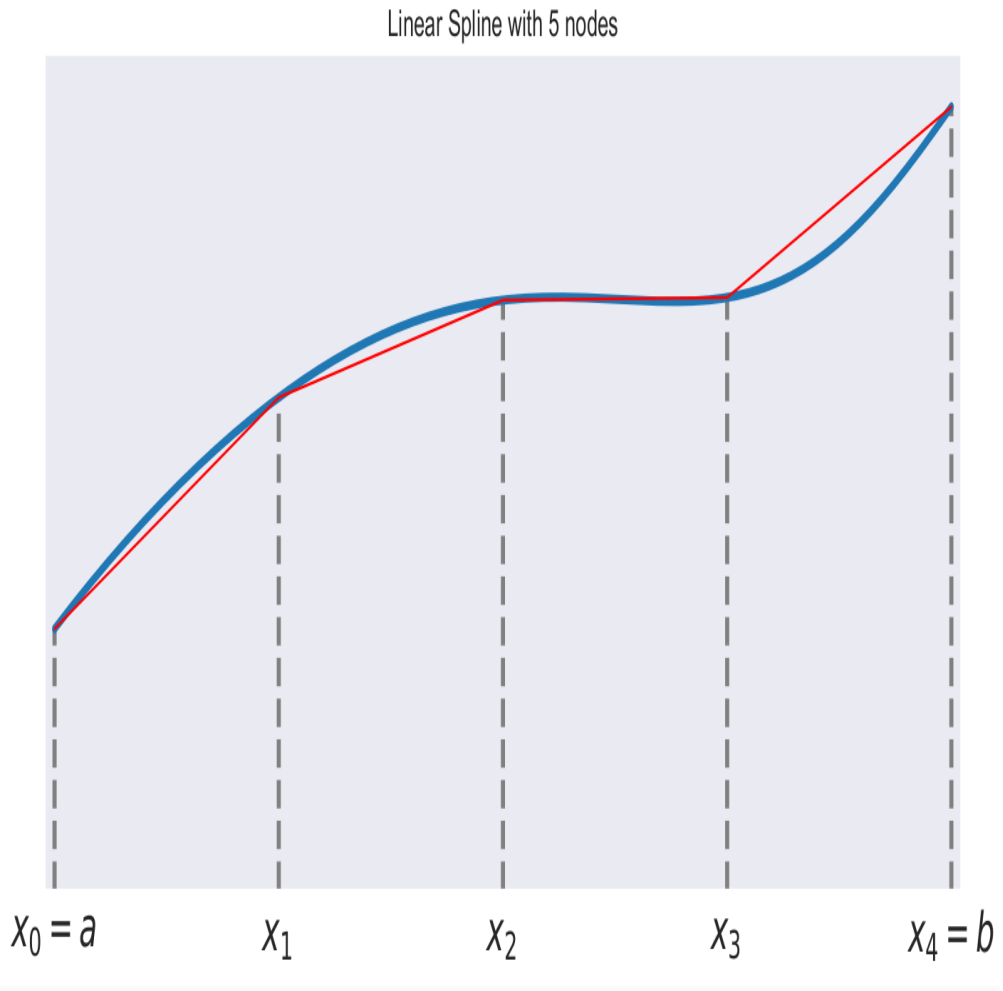


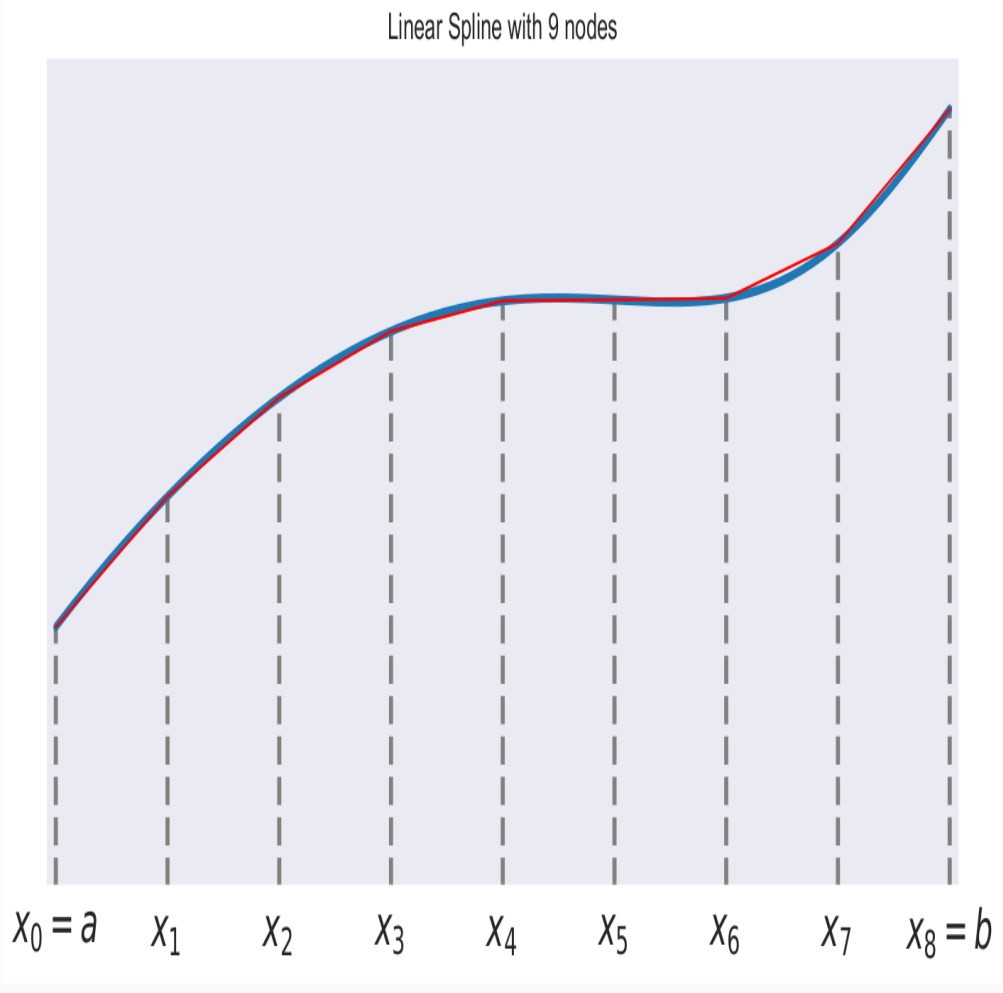
Spline interpolation

Piecewise polynomial splines, or simply splines for short, are a rich, flexible class of functions that may be used instead of high degree polynomials to approximate a real-valued function over a bounded interval. Generally, an order k spline consists of a series of k^{th} degree polynomial segments spliced together so as to preserve continuity of derivatives of order $k-1$ or less

- A first-order or **linear spline** is a series of line segments spliced together to form a continuous function.
- A third-order or **cubic spline** is a series of cubic polynomials segments spliced together to form a twice continuously differentiable function.


Linear Spline with 3 nodes
$x_0 = a$
x_1
$x_2 = b$


Linear Spline with 5 nodes
$x_0 = a$
x_1
x_2
x_3
$x_4 = b$



A linear spline with n + 1 evenly-spaced interpolation nodes $x_0, x_1, ..., x_n$ on the interval $[a, b]$ may be written as a linear combination of the $n + 1$ basis functions:

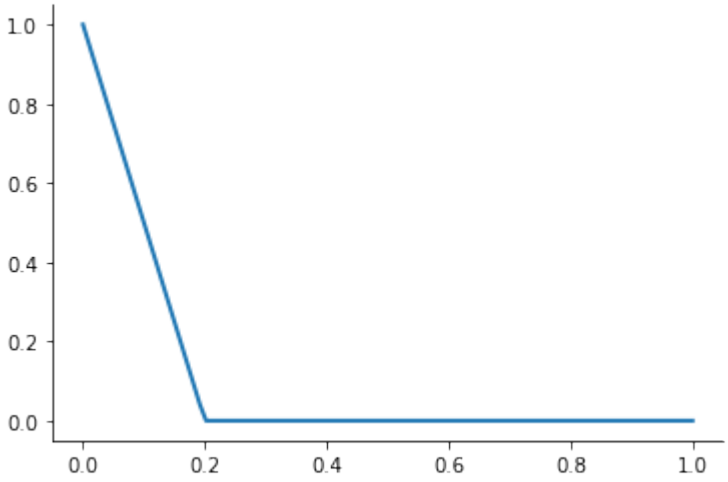
$$\phi_j(x) = \begin{cases} 1 - \frac{|x - x_j|}{h} & |x - x_j| \leq h \\ 0 \end{cases}$$

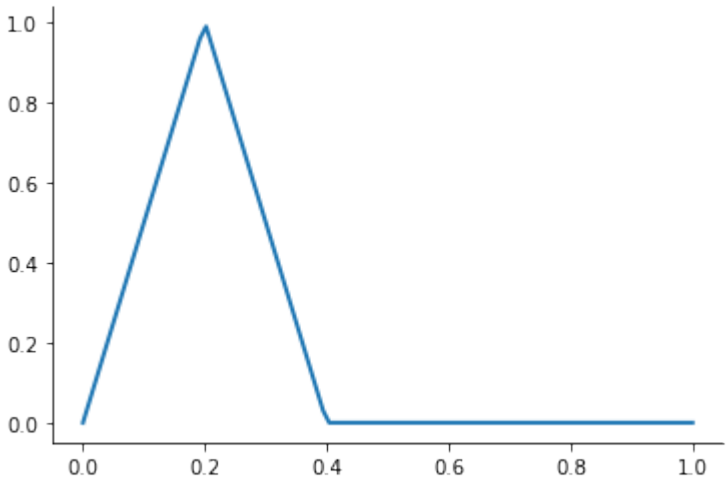
where $h = (b - a)/n$ is the distance between the nodes.

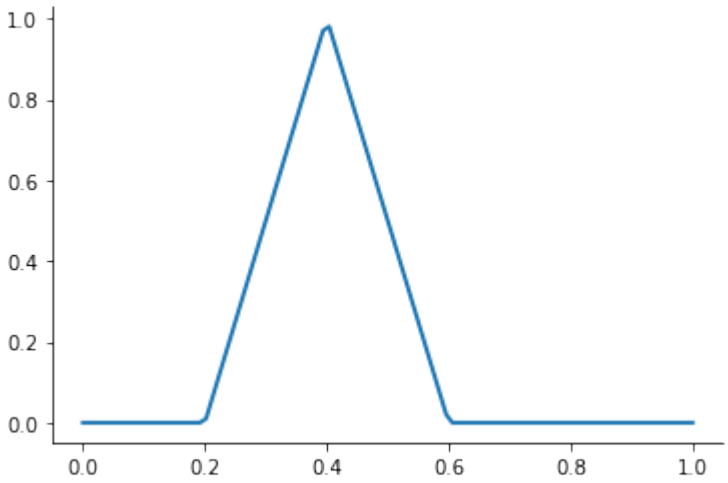
The linear spline approximant of f takes thus the form:

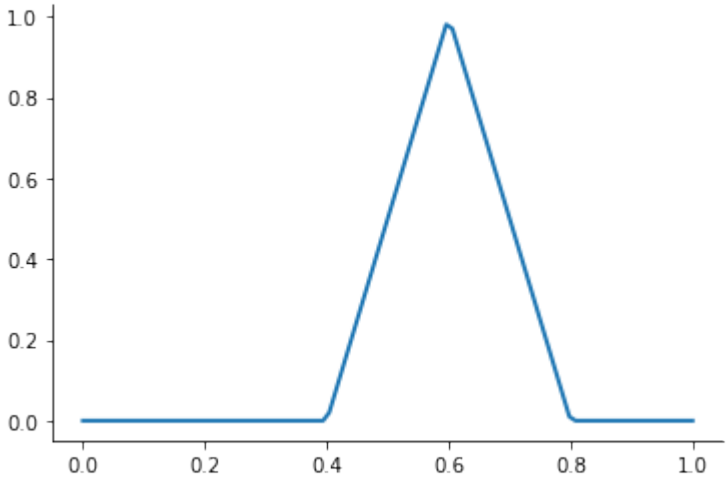
$$\hat{f}(x) = \sum_{j=1}^{n} f(x_j)\phi_j(x)$$

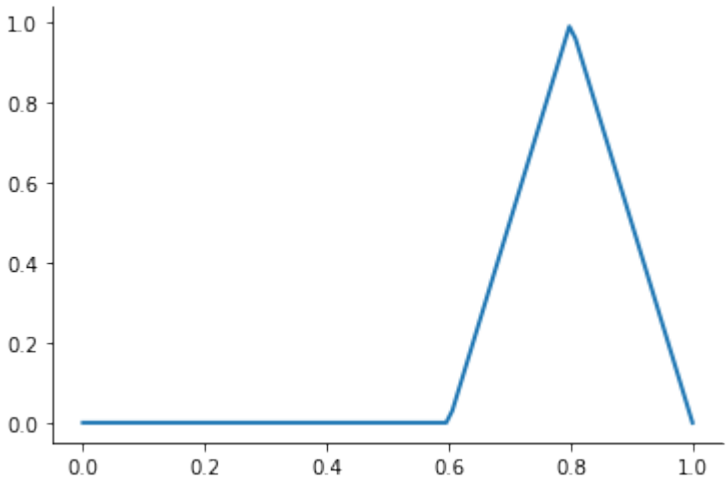
[22]:
```
plot_basis_functions("linear")
```

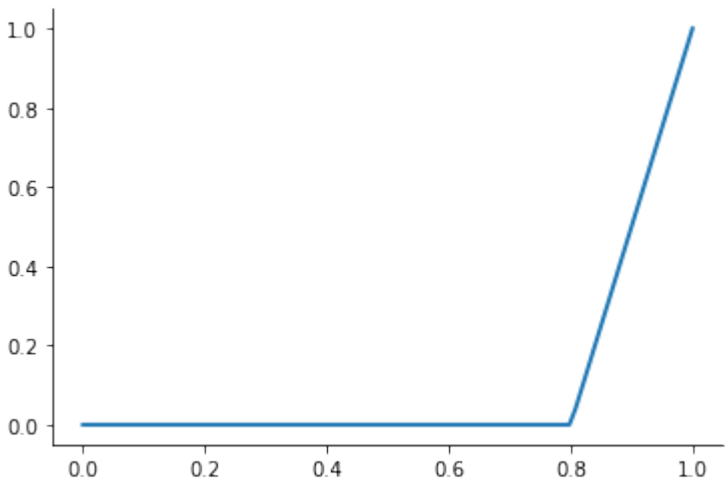

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0







This kind of interpolation procedure is frequently used in practice and readily available in `scipy`. The `interp1` function is documented here.

[23]:
```
x_fit = get_uniform_nodes(10, -1, 1)
```

(continues on next page)

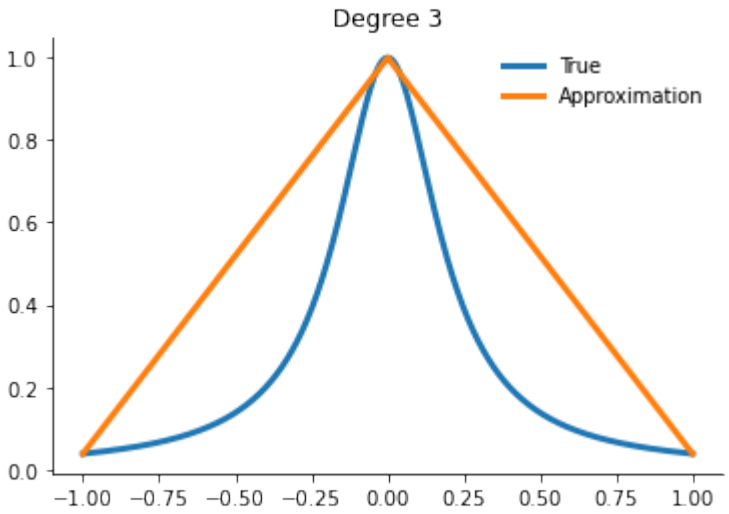
(continued from previous page)

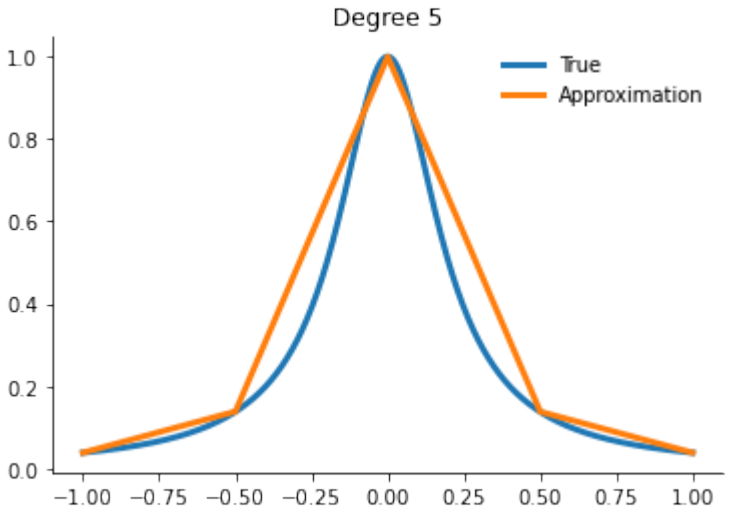
```
f_inter = interp1d(x_fit, runge(x_fit))
f_inter(0.5)
```

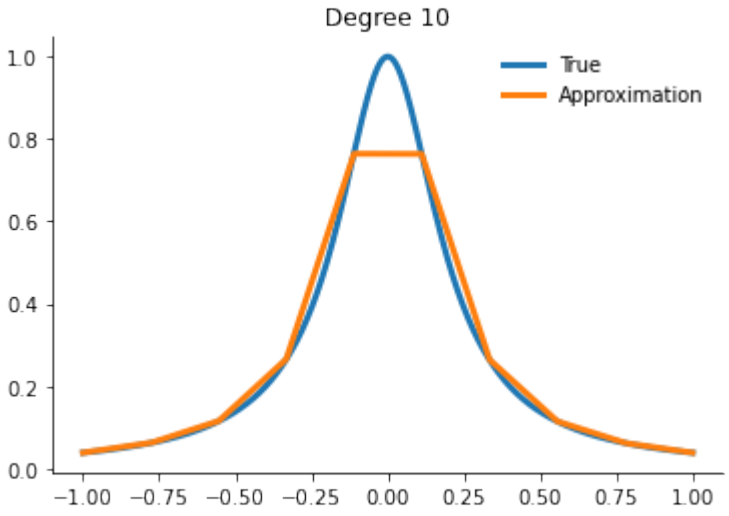
[23]:
```
array(0.15222463)
```

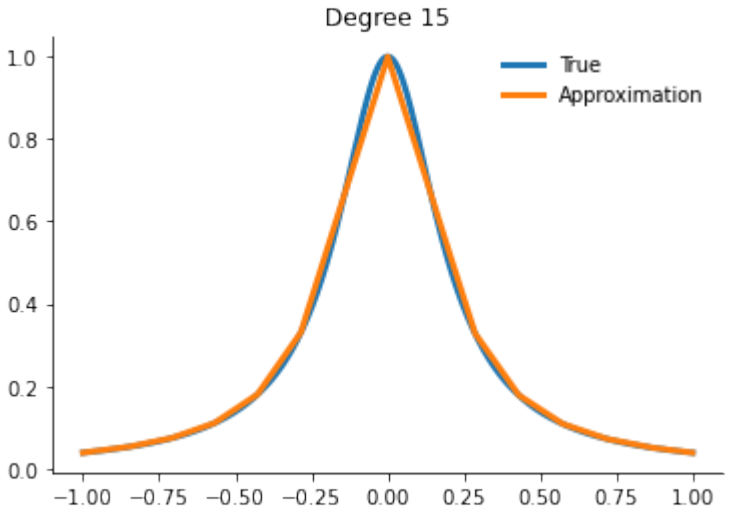
Let's get a feel for this approach using our earlier test function.

[24]:
```
x_eval = get_uniform_nodes(10000, -1, 1)

for degree in [3, 5, 10, 15]:
    x_fit = get_uniform_nodes(degree, -1, 1)

    interp = interp1d(x_fit, runge(x_fit))
    yfit = interp(xvalues)

    fig, ax = plt.subplots()
    ax.plot(xvalues, runge(xvalues), label="True")
    ax.plot(xvalues, yfit, label="Approximation")
    ax.legend()
    ax.set_title(f"Degree {degree}")
```




Degree 5
True
Approximation
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

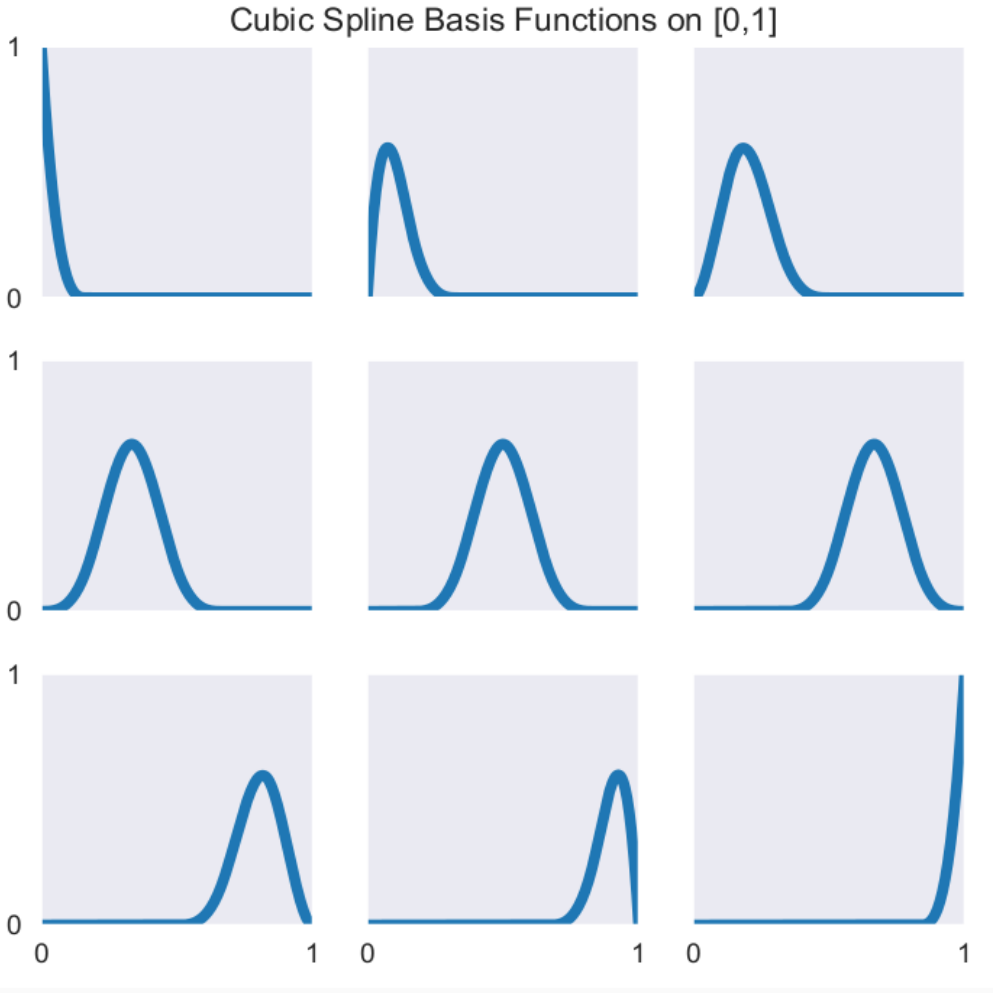

Degree 10
True
Approximation
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


Degree 15
True
Approximation
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

Question

- How about other ways to place the interpolation nodes?

Another widely-used specification relies on cubic splines. Here are the corresponding basis functions.



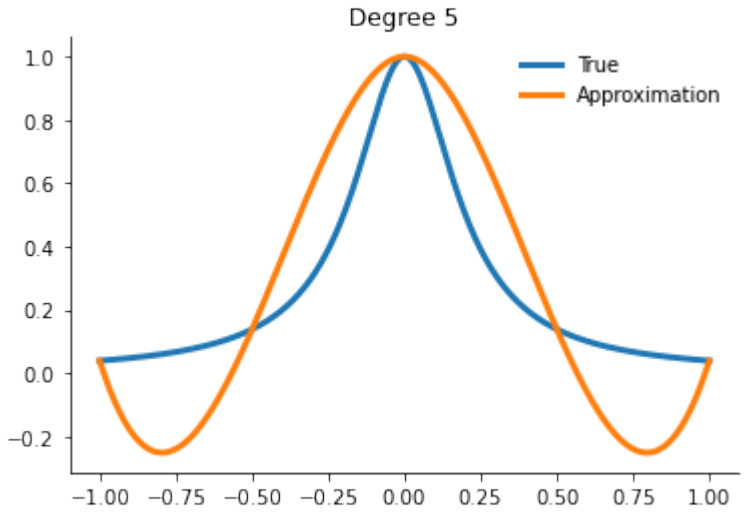
It is directly integrated into the `interp1` function.

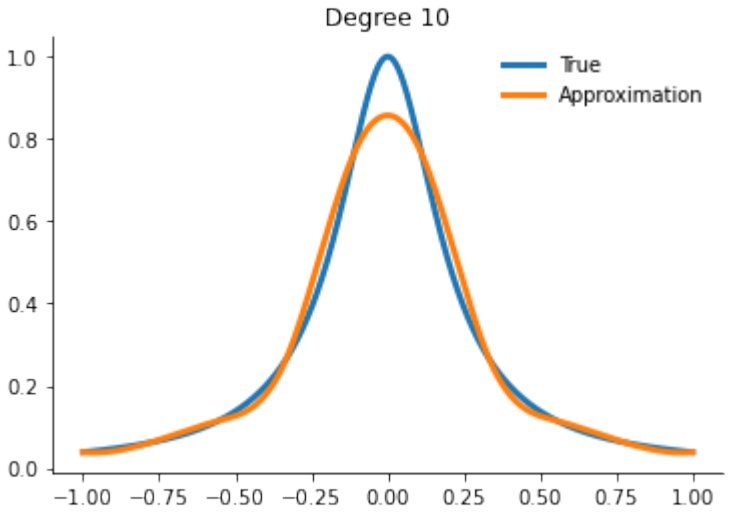
[25]:
```
x_fit = get_uniform_nodes(10, -1, 1)
f_inter = interp1d(x_fit, runge(x_fit), kind="cubic")
f_inter(0.5)
```

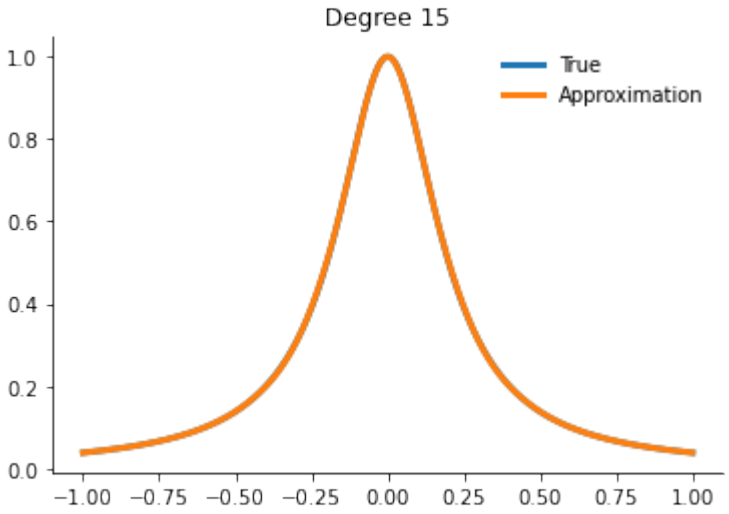
[25]:
```
array(0.12635205)
```

How about approximating Runge's function.

[26]:
```
plot_runge_function_cubic()
```







Let's take stock of our interpolation toolkit by running a final benchmarking exercise and then try to extract some

Exercises

Let's consider two test functions: problem_reciprocal_exponential, problem_kinked.

1. Visualize both over the range from -1 to 1. What is the key differences in their properties?
2. Set up a function that allows you to flexibly interpolate using either Chebychev polynomials (monomial basis, Chebychev nodes) or linear and cubic splines.
3. Compare the performance for the following degrees: 10, 20, 30.

We collect some rules-of-thumb:

- Chebychev-node polynomial interpolation dominates spline function interpolation whenever the function is smooth.
- Spline interpolation may perform better than polynomial interpolation if the underlying function exhibits a high degree of curvature or a derivative discontinuity.

Multidimensional interpolation

Univariate interpolation methods can be extended to higher dimensions by applying tensor product principles. We consider the problem of interpolating a bivariate real-valued function f over an interval:

$$I = \{(x, y)|a_x \leq x \leq b_x, a_y \leq y \leq b_y\}$$

Let $\phi_{x_1}, \phi_{x_2}, ..., \phi_{x_n}$ x and $x_1, x_2, ..., x_{n_x}$ be n_x univariate basis functions and n_x interpolation nodes for the interval $[a_x, b_x]$ and let $\phi_{y_1}, \phi_{y_2}, ..., \phi_{y_n}$ x and $y_1, y_2, ..., y_{n_y}$ be n_x univariate basis functions and n_y interpolation nodes for the interval $[a_y, b_y]$.

Then an $n = n_x n_y$ bivariate function basis defined on I may be obtained by forming the tensor product of the univariate basis functions: $\phi_{ij}(x, y) = \phi_i^x(x)\phi_j^y(y)$ for $i = 1, 2, ..., n_x$ and $j = 1, 2, ..., n_y$. Similarly, a grid of $n = n_x n_y$ interpolation nodes for I may be obtained by forming the Cartesian product of the univariate interpolation nodes

$$\{(x_i, y_j)|i = 1, 2, ..., n_x; j = 1, 2, ..., n_y\}.$$

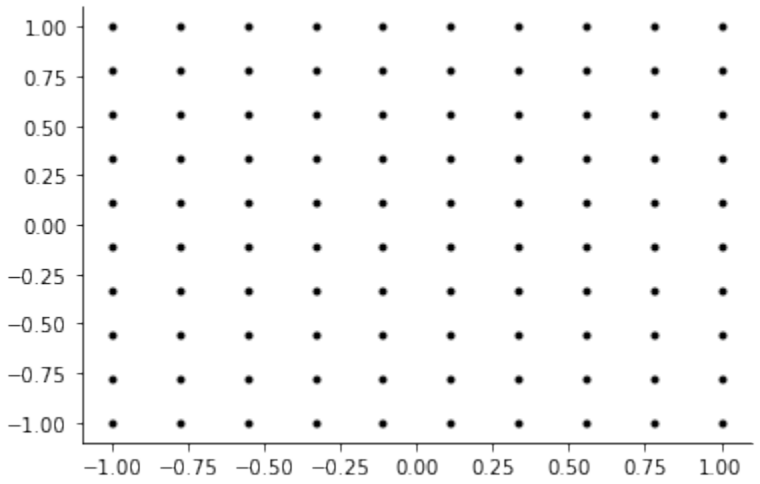
Typically, multivariate tensor product interpolation schemes inherit the favorable qualities of their univariate parents. An approximant for f then takes the form:

$$\hat{f}(x_1, x_2) = \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} c_{ij}\phi_{ij}(x_i, y_j)$$

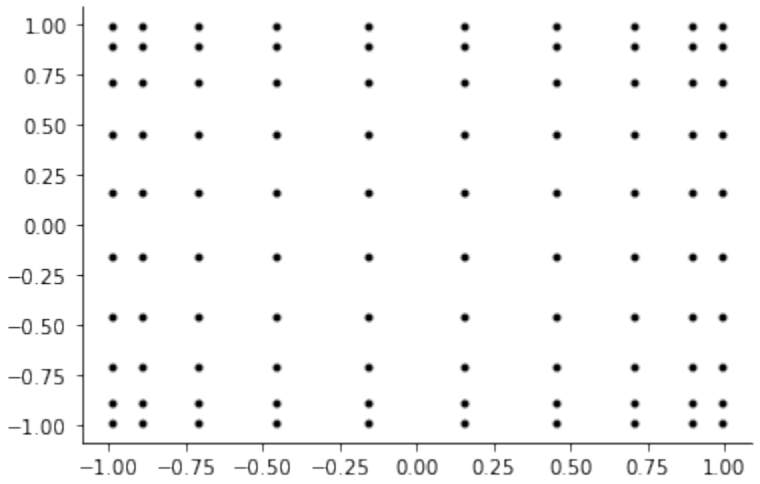
However, this straightforward extension to the multivariate setting suffers from the **curse of dimensionality**. For example, the number of interpolation nodes increases exponentially in the number of dimensions.

As an aside, we now move to the multidimensional setting where we often have to apply the same operation across multidimensional arrays and `numpy` provides some suitable capabilities to do this very fast if one makes an effort in understanding its broadcasting rules.

[27]:
```
plot_two_dimensional_grid("uniform")
```



[28]:
```
plot_two_dimensional_grid("chebychev")
```



Let's see how we can transfer the ideas to polynomial interpolation to the two-dimensional setting.

$$f(x, y) = \frac{\cos(x)}{\sin(y)}$$

[29]:
```
??plot_two_dimensional_problem
```

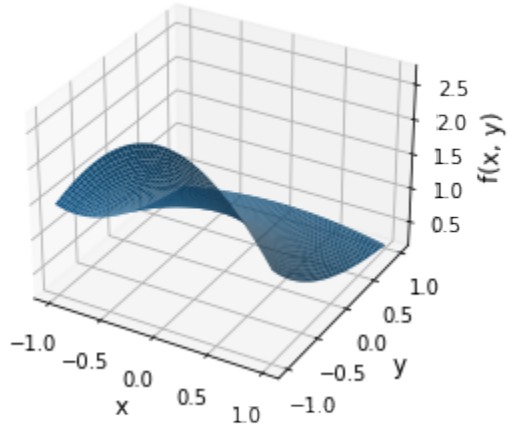
```
Signature: plot_two_dimensional_problem()
Source:
def plot_two_dimensional_problem():
    """Plot two-dimensional problem."""
    x_fit = get_uniform_nodes(50)
    y_fit = get_uniform_nodes(50)
    X_fit, Y_fit = np.meshgrid(x_fit, y_fit)
    Z_fit = problem_two_dimensions(X_fit, Y_fit)

    fig = plt.figure()
    ax = fig.gca(projection="3d")
```

(continues on next page)

(continued from previous page)

```
    ax.plot_surface(X_fit, Y_fit, Z_fit)
    ax.set_ylabel("y")
    ax.set_xlabel("x")
    ax.set_zlabel("f(x, y)")
File:      ~/external-storage/sciebo/office/OpenSourceEconomics/teaching/scientific-
↪computing/course/labs/approximation/approximation_plots.py
Type:      function
```

[30]:
```
plot_two_dimensional_problem()
```



Now, let's fit a two-dimensional polynomial approximation. We will have to rely on the `scikit-learn` library.

[31]:
```
from sklearn.preprocessing import PolynomialFeatures  # noqa: E402
from sklearn.linear_model import LinearRegression  # noqa: E402
import sklearn  # noqa: E402
```

We first need to set up an approximating model using some of its provided functionality. One of the functions at the core of this workflow is `np.meshgrid` which takes a bit of getting used to. Let's check out its documentation first and so some explorations.

[32]:
```
x_fit, y_fit = get_chebyshev_nodes(100), get_chebyshev_nodes(100)
```

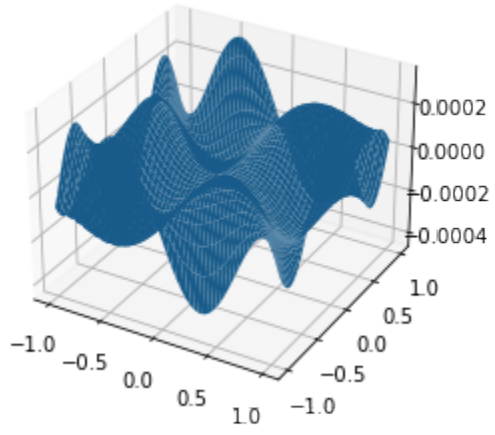
We now combine the univariate interpolation nodes into a two-dimensional grid, adjust it to meet the structure expected by `scikit-learn`, expand it to contain all polynomials (including interactions), and fit a linear regression model.

[33]:
```
X_fit, Y_fit = np.meshgrid(x_fit, y_fit)
grid_fit = np.array(np.meshgrid(x_fit, y_fit)).T.reshape(-1, 2)
y = [problem_two_dimensions(*point) for point in grid_fit]

poly = PolynomialFeatures(degree=6)
X_poly = poly.fit_transform(grid_fit)
clf = LinearRegression().fit(X_poly, y)
```

How well are we doing? As usual, we will simply compare the true and approximated values of the function over a fine grid.

[34]:
```
x_eval = get_uniform_nodes(100)
y_eval = get_uniform_nodes(100)

Z_eval = np.tile(np.nan, (100, 100))
Z_true = np.tile(np.nan, (100, 100))

for i, x in enumerate(x_eval):
    for j, y in enumerate(y_eval):
        point = [x, y]
        Z_eval[i, j] = clf.predict(poly.fit_transform([point]))[0]
        Z_true[i, j] = problem_two_dimensions(*point)

fig = plt.figure()
ax = fig.gca(projection="3d")
ax.plot_surface(*np.meshgrid(x_eval, y_eval), Z_eval - Z_true)
```

[34]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fc295c55640>
```



Resources

- https://relate.cs.illinois.edu/course/cs450-f18/file-version/a7a1965adf0479d36f1a34889afe55e2ec61a532/demos/upload/07-interpolation/Chebyshev%20interpolation.html
- https://www.unioviedo.es/compnum/labs/PYTHON/Interpolation.html
- https://www.johndcook.com/blog/2017/11/06/chebyshev-interpolation/
- https://numpy.org/devdocs/reference/routines.polynomials.html

1.6.2 Functions

This module contains the algorithms for the approximation lab.

labs.approximation.approximation_algorithms.**get_interpolator**(*name*, *degree*, *func*)
Return interpolator.

labs.approximation.approximation_algorithms.**get_interpolator_flexible_basis_flexible_nodes**(*func*, *degree*, *basis='monomial'*, *nodes='uniform'*, *a=-1*, *b=1*)
Return interpolator (flexible basis, flexible nodes).

labs.approximation.approximation_algorithms.**get_interpolator_monomial_flexible_nodes**(*func*, *degree*, *nodes='uniform'*, *a=-1*, *b=1*)
Return monomial function (flexible nodes).

labs.approximation.approximation_algorithms.**get_interpolator_monomial_uniform**(*func*, *degree*, *a=-1*, *b=1*)
Return interpolator monomial function (uniform).

labs.approximation.approximation_algorithms.**get_interpolator_runge_baseline**(*func*)
Return interpolator runge function (baseline).

CHAPTER

TWO

PROJECTS

All information regarding your course project is collected in the OSE course projects documentation.

CHAPTER

THREE

PARTNERS

Our course equips students with the required skills in statistics, technology, and communication to use data for decision-making. Our partnerships with the private and public sector connect students directly with employment opportunities that match their interests and skill set. All information regarding your partners is collected in the OSE course projects documentation.

CHAPTER

FOUR

ORGANIZATION

We start on October 12th, 2021, and meet on Tuesdays (2:00-3:30 PM) and Wednesdays (8:30-10:00 AM).

Lecturer: Philipp Eisenhauer

Teaching Assistants: Emily Schwab and Carolina Alvarez

We will conduct all course communications using the bonn-econ-teaching Zulip chat, so please be sure to join us there. To join the Zulip organization, please click on the button below.

By default, you will already be subscribed to Q&A and tech support streams (=chatrooms). You need to **subscribe manually to the scientific computing course stream** to receive all the messages for this course. There I also post the link to the online lectures using ZOOM. Find out how to subscribe to a stream on Zulip here.

4.1 Lecture Plan

We first meet to discuss the Open Source Economics initiative, the basic ideas behind the course and introduce participants to the basic tools used throughout.

Date	Topic
12/10/2021	Course introduction, Tooling
13/10/2021	Tooling

We then acquire the basic numerical skills that are needed in any implementation of computational economic models.

Date	Topic
19/10/2021	Linear equations
20/10/2021	Linear equations
25/10/2021	Deadline for registration with the examination office on BASIS
26/10/2021	Nonlinear equations
27/10/2021	Nonlinear equations
02/11/2021	Finite-dimensional optimization
03/11/2021	Finite-dimensional optimization
09/11/2021	Numerical integration and differentiation
10/11/2021	Numerical integration and differentiation
16/11/2021	German Reproducibility Day
17/11/2021	Guest lecture by Nuvolos and CI tutorial
23/11/2021	Function approximation
24/11/2021	Function approximation

We are now ready to study a dynamic model of human capital accumulation where all elements of our numerical toolbox are put together to study the mechanisms that determine individual investment decisions and to assess the impact of alternative human capital policies.

Date	Topic
30/11/2021	Eckstein-Keane-Wolpin models
01/12/2021	*Dies Akademicus (no classes)*
07/12/2021	Guest lecture by Ken Judd
08/12/2021	Keane & Wolpin (1997)
14/12/2021	Optimization using estimagic by Janos Gabler (OSE, University of Bonn)
15/12/2021	Research presentation: Uncertainty quantification
21/12/2021	Project clinic
22/12/2021	Project clinic

We then reconvene at the beginning of the new year to discuss some selected issues in more detail. We are fortunate to host guest lectures from the Hasenauer Lab on Bayesian parameter estimation, the Fraunhofer Institute for Applied Information Technology (FIT) who report on their use of microsimulation models in policy consulting, as well as the data science teams from Deutsche Bank, CommaSoft, and the AXA Data Innovation Lab. The goal of these lectures is to emphasize the transdisciplinary nature of research using computational models across domains and areas of expertise.

Date	Topic
11/01/2022	Guest lecture by AXA Data Innovation Lab
12/01/2022	Guest lecture by Fraunhofer FIT
18/01/2022	Guest lecture by CommaSoft
19/01/2022	Guest lecture by Manuel Huth
21/01/2022	**Project deadline**
25/01/2022	Guest lecture by Deutsche Bank
26/01/2022	Demo-Day

CHAPTER

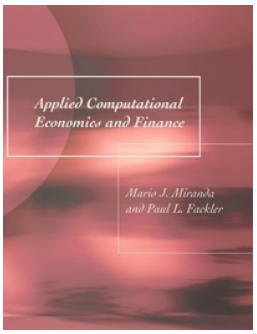
FIVE

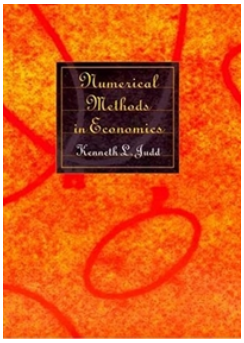
BIBLIOGRAPHY

CHAPTER

SIX

TEXTBOOKS





We use the book Applied computational economics and finance by Mario Miranda and Paul Fackler throughout the course. A special thanks to Randall Romero Aguilar who has also built a course around this book and maintains a Python implementation of the CompEcon toolbox. Many of our code examples are building on his implementation there. In addition, we will also consult Numerical methods in economics by Ken Judd for some of the more advanced material.

CHAPTER

SEVEN

REVIEWS

- **Athey, S., Imbens, G. (2019)**. Machine learning methods that economists should know about, *Annual Review of Economics*, 11(1): 685-725.
- **Judd, K.L. (1997)**. Computational economics and economic theory: Substitutes or complements?, *Journal of Economic Dynamics and Control*, 21(6): 907-942.

CHAPTER

EIGHT

POWERED BY

We gratefully acknowledge funding by the Federal Ministry of Education and Research (BMBF) and the Ministry of Culture and Science of the State of North Rhine-Westphalia (MKW) as part of the Excellence Strategy of the federal and state governments.

BIBLIOGRAPHY

[MF04] Mario J Miranda and Paul L Fackler. *Applied computational economics and finance*. MIT Press, 2004.

[RA20] Randall Romero-Aguilar. A Python version of Miranda and Fackler's CompEcon toolbox. 2020. URL: https://github.com/randall-romero/CompEcon.

[Fos19] John T Foster. *Numerical methods and computer programming*. e-book, 2019. URL: https://johnfoster.pge.utexas.edu/numerical-methods-book.

[Jud98] Kenneth L Judd. *Numerical methods in economics*. MIT Press, 1998.

[Joh18] Robert Johansson. *Numerical Pyython: scientific computing and data science applications with Numpy, SciPy and Matplotlib*. Apress, 2018.

[PFTV86] William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 1986.

PYTHON MODULE INDEX

INDEX

S