Optimization with SciPy

SciPy is a library that builds on NumPy and provides additional tools for scientific computing. Some of the tools provided are: numerical integration, optimization, interpolation, Fourier transforms, signal processing, linear algebra, spatial algorithms and statistics. The reference page for the current version is available here.

1 Install

On the terminal, execute:

```
pip install -U scipy
```

If you are having issues with the installation process check the scipy install page.

Scipy should now be available under the `scipy` namespace.

2 Statistics

The `scipy.stats` subpackage contains functions for: generating numbers from continuous distributions, from multivariate distributions and from discrete distributions, to compute summary and frequency statistics, statistical tests and density estimation.

2.1 Distribution of Random Variables

While NumPy provides several functions for random number generations from continuous and discrete distributions, SciPy extends on the list and also provides functions for the cdf, pdf, quantiles and so on.

In Scipy, the functions the statistical distributions are available under separate namespaces. For example, the functions for the normal distribution are available under `scipy.stats.norm`. The `scipy.stats.norm` class is a subclass of `rv_continuous`, which itself is an abstract class. This means that, `rv_continuous` defines the names of the methods and properties of a generic class, but it is the job of its subclasses to actually implement them. In this case, `scipy.stats.norm`, implements the methods of `rv_continuous` to the case of a normal distribution. Therefore, to get a sense of what functions are available for all continuous statistical distributions, you only need to read the documentation of `rv_continuous`. For discrete distributions, the abstract class is `rv_discrete`.

```
import matplotlib.pyplot as plt
import numpy as np
# Import logistic distribution
from scipy.stats import logistic
```

```
# Instantiate a logistic distribution
# scale (mean)
mu = 1
# scale (variance is proportional to scale)
s = 1
help(logistic)
l = logistic(loc=mu, scale=s)
# Freezes the distribution,
# so that the methods apply to a Logistic with
# mean mu and scale s.
# Histogram:
sample = l.rvs(size=(2000, 1))
fig, ax = plt.subplots(figsize=(10, 6))
ax.hist(sample, density=True, bins=50)  # normalized
# Density:
x = np.linspace(1.1*sample.min(), 1.1*sample.max(), 200)
ax.plot(x, l.pdf(x), 'k-', linewidth=2)
fig.show()
# Other methods:
# cumulative distribution
l.cdf(0)
# inverse cdf
l.ppf(0.5)
# Summary statistics:
l.mean()
l.var() == s**2*np.pi**2/3
# Alternatives:
logistic.cdf(0, loc=mu, scale=s)
logistic.ppf(0.5, loc=mu, scale=s)
```

3 Optimization

The sub-package `scipy.optimize` contains several functions for solving optimization problems.

Consider an optimization problem where we need to find the minimum value of a function $f : \mathbb{R} \mapsto \mathbb{R}$ on some subset of its domain:

$$\min_{x \in D} f(x)$$

We are interested in solving this problem. However, it is not clear that a solution to this problem exists. We know that if f is a continuous function and D is a compact set, then by the Weierstrass theorem $\exists x^* \in D : f(x^*) \leq f(x), \forall x \in D$. Under these conditions, we know that a solution to the problem exists, but we still need a way to find it. If the function f is not continuous, or D is not a compact set, then a solution may or may not exist.

Even though we cannot always guarantee that a solution exists, if a solution does exist, then it must satisfy a necessary condition. If x_0 is a point where f is minimized (or maximized) and f is differentiable at x_0, then $f'(x_0) = 0$ (see Fermat's theorem on stationary points). This necessary condition motivates many of the optimization algorithms.

The algorithms for finding the argument that minimizes a function are divided in two types: direct search methods and gradient-based methods. Direct search methods do not rely on the derivative of a function, and so can be applied to non-differentiable functions. These methods directly search for optimal points across the domain of a function. An example of a direct search method is the Golden-section search. Gradient-based methods use the derivative of a function to find a value x such that $f'(x) = 0$. In these algorithms, the gradient can be explicitly computed or approximated numerically. Examples of gradient-based algorithms are: Bisection method, Newton-Raphson method and Secant method.

These optimization methods apply to functions of a single variable. However, they are building blocks for optimization methods for functions of many variables. When it comes to optimization algorithms for functions of many variables, we can still divide the algorithms in the same two types. Examples of direct search methods for multivariate functions are: Powell's method and Nelder-Mead method. Some of the optimization methods that use the gradient are: Newton's method, DFP method, BFGS method and the Gradient Descent method.

The methods above apply to unconstrained optimization problems. However, in practice we usually have to deal with optimization problems that are subject to constraints:

$$\begin{aligned} \underset{x\in\mathbb{R}^d}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & f_i(x) \leq b_i, i = 1, \ldots, d \end{aligned}$$

For constrained problems, some of the optimization methods are: Penalty function method, Augmented Lagrangian method and Sequential quadratic programming. These methods either transform the constrained problem into an unconstrained problem with an added term that penalizes values that are outside the original constraints, or take extra care when looking through the domain to make sure the points satisfy the constrains.

Fortunately, we do not need to implement these algorithms, as most of them are already implemented in SciPy.

3.1 Finding the Roots of a Function

The function `bisect` implements the bisection algorithm. It requires a function and two endpoints of an interval. The function must have different signs on these two points. The idea is that if the function is continuous, then it must have changed signs in some point of the interval. The idea is to then take a third random point inside the interval and check whether there was a change in sign. Depending on the sign of that point, one part of the interval is thrown away, and search continues on the other part of the interval.

```python
import matplotlib.pyplot as plt
from scipy.optimize import bisect

fun = lambda x: x**3 - 2*x**2 + 3*x - 10;
# Visualize fun
fig, ax = plt.subplots()
x = np.linspace(-5, 5, 100)
ax.plot(x, fun(x))
fig.show()
# Find root
x_root = bisect(fun, -5, 5)
```

```
ax.plot(x_root, fun(x_root), 'k*', linewidth=3)
ax.text(x_root + 0.1, 0.1, 'Root')
```

The bisection method is a direct search method, and, while it works well for continuous functions, it can be quite slow depending on the function. An alternative is the Neton-Raphson method, which is a gradient-based method. This method works well for well-behaved functions, and achieves a higher speed than the bisection method. However, it can fail for certain types of functions. SciPy actually recommends the function `brentq` for root-finding, as it combines several ideas to speed-up the execution (hybrid method). The algorithm also requires the endpoints of the interval, a continuous function and that the function has different signs at the endpoints.

```
from scipy.optimize import brentq

x_root = brentq(fun, -5, 5)
# Visualize
fig, ax = plt.subplots()
x = np.linspace(-5, 5, 100)
ax.plot(x, fun(x))
ax.plot(x_root, fun(x_root), 'k*', linewidth=3)
ax.text(x_root + 0.1, 0.1, 'Root')
fig.show()
```

3.1.1 Fixed Point

A related problem is that of finding a fixed point. Remember, we say x is a fixed point of $f : \mathbb{R} \mapsto \mathbb{R}$ if $f(x) = x$. This problem can be cast as a root finding problem for the function $g(x) \equiv f(x) - x$. We can solve this type of problem with the function `fixed_point`. This function does not require endpoints of an interval, but an initial guess for the fixed point.

```
from scipy.optimize import fixed_point


# Visualize problem
fig, ax = plt.subplots()
x = np.linspace(-np.pi, np.pi, 50)
ax.plot(x, np.cos(x), label='$\cos(x)$')
ax.plot(x, x, label='$45^o$ line')
ax.legend()
fig.show()
# Find fixed point
x_fixed = fixed_point(np.cos, -np.pi)
ax.plot(x_fixed, np.cos(x_fixed), 'k*')
ax.text(x_fixed + 0.1, x_fixed + 0.1, 'Fixed Point')
```

3.2 Unconstrained Optimization

The function `minimize` can be used to solve unconstrained optimization problems for a function $f : \mathbb{R}^n \mapsto \mathbb{R}$. The `minimize` function implements the following algorithms for unconstrained optimization:

- Nelder-Mead (`'nelder-mead'`): direct search method, works well for well-behaved functions, but is slow since does not use the gradient
- Broyden-Fletcher-Goldfarb-Shanno (`'BFGS`): known as BFGS method, uses the gradient (if not supplied, the gradient is estimated numerically)
- Netwon-Conjugate-Gradient (`'Newton-CG'`): known as Netwon-CG, uses gradient and Hessian, finds descent step via quadratic approximation
- Trust-Region Newton-Conjugate-Gradient (`'trust-ncg'`): uses gradient and Hessian, but looks for the best descent step given a maximum step size
- Trust-Region Truncated Generalized Conjugate Gradient (`'trust-krylov'`): similar to the Trust-Region method above, but more efficiently
- Trust-Region Nearly Exact Algorithm (`trust-exact`): uses gradient and Hessian, adequate for medium-sized problems

According to SciPy, the methods `'Newton-CG'`, `'trust-ncg'` and `'trust-krylov'` are good for dealing with very large problems (thousands of variables). This is the case because instead of supplying the entire Hessian matrix, which can be slow to compute and take a lot of memory, we can supply the product of the Hessian matrix with an arbitrary vector. Let $H : \mathbb{R}^n \mapsto \mathbb{R}^{n \times n}$ denote the Hessian matrix, then we can either pass H to the `minimize`, or, we can pass $H \cdot p$, where p is a $n \times 1$ vector, implying that $H \cdot p : \mathbb{R}^n \mapsto \mathbb{R}^n$, diminishing the amount of memory required to store the result. The method `'trust-exact'` does not support the Hessian product, and takes the exact Hessian, but provides a good algorithm to solve medium-scale problems where the Hessian can be stored in memory.

Let's start by defining and visualizing a test function:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from scipy.optimize import minimize


# Function to be minimized
def rosenbrock(x, y, a, b):
    return (a - x)**2 + b*(y - x**2)**2


# Visualize problem
x_grid = np.linspace(-2, 2, 50)
y_grid = np.linspace(-1, 3, 50)
[x, y] = np.meshgrid(x_grid, y_grid)
z = rosenbrock(x, y, 1, 100)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(x, y, z, alpha=0.7, cmap=cm.gnuplot)
# Display global minimum
ax.scatter(1, 1, 0, color='black', marker='o', s=100)
# Rotate figure: 30 degrees on vertical, 50 degrees on horizontal
ax.view_init(30, 50)
```

```
# Add labels
ax.set(title="Unconstrained Minimization of Rosenbrock's Function")
fig.show()
```

Let's use the `minimize` function to minimize `rosenbrock` with the different algorithms. We need to redefine `rosenbrock` because `minimize` takes a function of a single argument, instead of a function of many arguments:

```
# Minimize rosenbrock
# minimize takes a function of a vector,
# so we need to rewrite rosenbrock
def fun(point): return rosenbrock(point[0], point[1], 1, 100)
```

We begin with Nelder-Mead, since it is the simplest algorithm to use and does not require any derivatives:

```
# Minimize with Nelder-Mead
# only requires the function itself and an initial guess
res = minimize(fun, np.array([-1, 3]), method='nelder-mead',
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

Next, we use the BFGS method. To use it, we need to define the Jacobian of `rosenbrcok`:

```
# Minimize with BFGS
# need to supply the gradient
def rosenbrock_der(x, y):
    return np.array([-400*(y - x**2)*x - 2*(1-x),
                     200*(y - x**2)])



def dfun(point):
    return rosenbrock_der(point[0], point[1])
```

There is a helper function named `check_grad` that we can use to check the gradient. It compares the theoretical gradient with its numerical counterpart. The function takes as an input the point where to evaluate the gradients, and it returns the difference between the approximation and the gradient we coded. If we coded everything correctly, this difference should be small.

```
from scipy.optimize import check_grad
difference = check_grad(fun, dfun, x0=[-3, 1])
print(f'Difference between theoretical and numerical gradient: {difference
```

We can now use the method BFGS with confidence in our Jacobian implementation:

```
res = minimize(fun, x0=np.array([-1, 3]),
               method='BFGS', jac=dfun,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

Next, we use the Newton-Conjugate Gradient method. To use it, we also need to supply the hessian:

```
# Minimize with Newton-Conjugate Gradient Algorithm
# requires the Hessian
```

```
def rosenbrock_hess(x, y):
    return np.array([[-400*(y - 3*x**2) + 2, -400*x],
                     [-400*x, 200]])


def ddfun(point):
    return rosenbrock_hess(point[0], point[1])


res = minimize(fun, x0=np.array([-1, 3]),
               method='Newton-CG', jac=dfun, hess=ddfun,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

The Newton-Conjugate Gradient method (and others) can accept the product of the Hessian with an arbitrary vector, instead of the full Hessian. This is useful for larger problems (functions of several variables), where the Hessian can be a big matrix and use a lot of memory.

```
# Minimize with Newton-Conjugate Gradient Algorithm
# use the Hessian product
def rosenbrock_hess_prod(x, y, p):
    return rosenbrock_hess(x, y)@p


def ddfun_prod(point, p):
    return rosenbrock_hess_prod(point[0], point[1], p)


res = minimize(fun, x0=np.array([-1, 3]),
               method='Newton-CG', jac=dfun, hessp=ddfun_prod,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

The Trust-Region Newton-Conjugate-Gradient method is similar to the previous method, but attempts to take bigger steps in the optimization procedure:

```
# Minimize with Trust-Region Newton-Conjugate-Gradient Algorithm
# with full Hessian
res = minimize(fun, x0=np.array([-1, 3]),
               method='trust-ncg', jac=dfun, hess=ddfun,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
# with Hessian product
res = minimize(fun, x0=np.array([-1, 3]),
               method='trust-ncg', jac=dfun, hessp=ddfun_prod,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

The Trust-Region Truncated method is an improvement of the previous method, but behaves similarly:

```
# Minimize with Trust-Region Truncated Generalized Conjugate Gradient Algo
# with full Hessian
```

```
res = minimize(fun, x0=np.array([-1, 3]),
               method='trust-krylov', jac=dfun, hess=ddfun,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
# with Hessian product
res = minimize(fun, x0=np.array([-1, 3]),
               method='trust-krylov', jac=dfun, hessp=ddfun_prod,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

The Trust-Region Nearly Exact Algorithm is similar to the previous two methods, but only accepts the entire Hessian as input. It is useful for medium-sized problems, in which the Hessian does not take too much memory.

```
# Minimize with Trust-Region Nearly Exact Algorithm
# with full Hessian
res = minimize(fun, x0=np.array([-1, 3]),
               method='trust-exact', jac=dfun, hess=ddfun,
               options={'disp': True})
print(f'Solution: {res.x}\nIterations: {res.nit}')
```

Let's compare the number of iterations and the solutions found by each of the methods:

```
# Compare methods
from functools import partial
getmin = partial(minimize, fun=fun, x0=np.array([-1, 3]),
                 jac=dfun, hess=ddfun)
methods = ['nelder-mead', 'BFGS', 'Newton-CG', 'trust-ncg',
           'trust-krylov', 'trust-exact']
print('|{:^15s}|{:^15s}|{:^15s}|'.format(
    'Method', 'Iterations', 'Solution'))
for method in methods:
    res = getmin(method=method)
    print((f'|{method:^15s}|{res.nit:^15d}|'
           f'{np.array2string(res.x, precision=2):^15s}|'))
```

Notice that `.format` is a string method that works in a similar way to f-Strings: it substitutes values wherever there are curly brackets. The `:` inside the curly brackets specify the formatting of the string. The number `15` means we want to put up to 15 whitespaces after the word. The letter `^` means we want the string to be centered, so that there is some whitespace before and after the string. The letter `s` indicates that what we are substituting in the curly brackets is a string.

3.3 Constrained Optimization

The function `minimize` also provides algorithms for solving constrained minimization problems. There are three algorithms implemented:

- Trust-Region Constrained Algorithm: `trust-constr`
- Sequential Least Squares Programming: `SLSQP`
- Constrained Optimization by Linear Approximation: `COBYLA`

We will discuss how to use the Trust-Region Constrained Algorithm.

3.3.1 Trust-Region Constrained Algorithm

The Trust-Region Constrained Algorithm can tackle constrained optimization problems of the form:

$$\begin{aligned} \underset{x\in\mathbb{R}^d}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & \begin{cases} c^l \leq c(x) \leq c^u \\ x^l \leq x \leq x^u \end{cases} \end{aligned}$$

Where c^l, c^u, x^l, x^u can be vectors. We can specify equality constraints by setting $c^l = c^u$, and one-sided constrains by setting c^l to `-np.inf` or c^u to `np.inf`.

Let's solve the following optimization problem:

$$\begin{aligned} \underset{(x,y)\in\mathbb{R}^2}{\text{minimize}} \quad & (1-x)^2 + 100(y-x^2)^2 \\ \text{subject to} \quad & \begin{cases} -1 \leq x + 2y \leq 1 \\ x^2 + y \leq 1 \\ 2x + y = 1 \\ -1 \leq x \leq 0.5 \\ -0.1 \leq y \leq 0.3 \end{cases} \end{aligned}$$

There are three constraints on transformations of (x, y), two of which are linear and one is non-linear, and two constraints on the values of (x, y).

To define the linear constraints, we use the object `LinearConstraint`:

```python
from scipy.optimize import LinearConstraint


# need to supply a matrix A, such that A @ [x, y] gives
# the linear constraints we care about
A = [[1, 0], [0, 1], [1, 2], [2, 1]]
lin_cons = LinearConstraint(A, [-1, -0.1, -1, 1], [0.5, 0.3, 1, 1])
```

For non-linear constraints we use the object `NonlinearConstraint`. It takes a function that specifies the non-linear constraints. We can also supply the gradient and Hessian matrix if it is possible to compute them, otherwise they are approximated numerically.

```python
from scipy.optimize import NonlinearConstraint


def nl_cons(point):
    return [point[0]**2 + point[1]]


def nl_cons_der(point):
    return [2*point[0], 1]


# WARNING: here the Hessian actually
# takes the inputs separately instead of
#  a single input! (don't ask me why)
def nl_cons_hess(x, y):
    return [[2, 0],
            [0, 0]]


# without supplying gradient and Hessian
nlin_cons = NonlinearConstraint(nl_cons, [-np.inf], [1])
# supplying gradient and Hessian
nlin_cons_full = NonlinearConstraint(nl_cons, [-np.inf], [1],
                                     jac=nl_cons_der,
                                     hess=nl_cons_hess)
```

We can now solve the optimization problem:

```
def rosenbrock(point):
    return (1-point[0])**2 + 100*(point[1]-point[0]**2)**2


def rosenbrock_der(point):
    return [-400*(point[1] - point[0]**2)*point[0] - 2*(1-point[0]),
            200*(point[1] - point[0]**2)]


def rosenbrock_hess(point):
    return [[-400*(point[1] - 3*point[0]**2) + 2, -400*point[0]],
            [-400*point[0], 200]]


x0 = np.array([0.2, 0.2])
res = minimize(rosenbrock, x0,
               method='trust-constr',
               jac=rosenbrock_der,
               hess=rosenbrock_hess,
               constraints=[lin_cons, nlin_cons_full],
               options={'verbose': 1})
print(f'Solution: {np.array2string(res.x, precision=2)}')
print(f'Iterations: {res.nit}')
```

4 Numerical Integration

The sub-package `scipy.integrate` implements methods for the numerical integration of functions. Consider the following integral:

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

Where $f : \mathbb{R} \mapsto \mathbb{R}$ and $a, b \in \mathbb{R} \cup \{\pm\infty\}$. To integrate such a function, we can use the function `quad`:

```
from scipy.integrate import quad
from scipy.stats import norm

help(quad)
```

```
res, _ = quad(np.sin, 0, np.pi)
print(res)
res, _ = quad(np.sin, 0, 2*np.pi)
print(res)
res, _ = quad(lambda x: np.exp(-x), 0, np.inf)
print(res)
res, _ = quad(norm.pdf, -np.inf, np.inf)
print(res)
```

There are other functions for evaluating double, triple and even bigger integrals. To see details, check the documentation for `dblquad`, `tplquad` and `nquad`.

5 Approximating Functions

Given a function $f : \mathbb{R} \mapsto \mathbb{R}$ we want to approximate it by a finite number n of basis functions $\{p_k\}_{k=1}^n$. We need to choose weights such that:

$$\begin{aligned} f(x) &\approx \beta_1 p_1(x) + \beta_2 p_2(x) + \cdots + \beta_n p_n(x) \\ &= \underbrace{\begin{pmatrix} p_1(x) & p_2(x) & \cdots & p_n(x) \end{pmatrix}}_{p(x)'} \underbrace{\begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix}}_{\beta} \\ &= p(x)'\beta \end{aligned}$$

If we have a set of points $\{x_i\}_{i=1}^d$ where we know the value of f, then we can write:

$$\underbrace{\begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_d) \end{pmatrix}}_{F} \approx \underbrace{\begin{pmatrix} p_1(x_1) & p_2(x_1) & \cdots & p_n(x_1) \\ p_1(x_2) & p_2(x_2) & \cdots & p_n(x_2) \\ & & \vdots & \\ p_1(x_d) & p_2(x_d) & \cdots & p_n(x_d) \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_d \end{pmatrix}}_{\beta} \tag{1}$$

If the number of basis functions and the number of points is the same, that is $n = d$, then there is an exact solution for β if P is invertible:

$$\beta = P^{-1}F$$

If we have more points than basis functions, then we can compute a least-squares approximation to f:

$$\hat{\beta} \equiv \underset{\beta \in \mathbb{R}^n}{\text{argmin}} \sum_{i=1}^{d} (f(x_i) - p(x_i)'\beta)^2$$

We know that the "OLS" solution is:

$$\hat{\beta} = (P'P)^{-}P'F$$

Notice the use of a generalized inverse (Moore-Penrose) in $(P'P)^{-}$. This is necessary because depending on choice of the basis function, we might have a column of zeros in

P (this happens with splines when there are more knots relative to points). Now, the function that approximates f is:

$$\hat{f}(x) \equiv p(x)'\hat{\beta}$$

This approximating function can be computed given a choice of basis functions $\{p_k\}_{k=1}^n$ and a collection of data points $\{x_i\}_{i=1}^d$.

As an aside, consider the case where the original function we wanted to estimate was a conditional mean, $f(x) = \mathbb{E}[Y|X = x]$. If we have i.i.d. observations of f and x, then the $\hat{f}$ above is exactly the non-parametric Sieves estimator for the conditional mean. In this case, there is theory justifying the consistency and asymptotic normality of $\hat{f}$.

Next, we discuss different choices for basis functions. We will focus on polynomials, since their properties are well understood and they are easy to work with.

5.1 Polynomials, Orthogonality and Orthonormality

Let $\mathcal{P}$ denote a family of polynomials parameterized by the degree of the polynomials:

$$\mathcal{P} = \{P_n : \mathbb{R} \mapsto \mathbb{R} : P_n \text{ is polynomial of degree n}\}_{n\in\mathbb{N}\cup\{0\}}$$

We say that a family of polynomials is <u>orthogonal</u> if any two of its members are orthogonal to each other under some inner product:

$$\exists w : \mathbb{R} \mapsto \mathbb{R} : \forall P_n, P_m \in \mathcal{P}, \int w(x)P_n(x)P_m(x)dx = 0 \text{ if } n \neq m$$

We say that a family of polynomials is <u>orthonormal</u> if it is orthogonal and:

$$\forall P_n \in \mathcal{P}, \int w(x)P_n(x)P_n(x)dx = 1$$

We are interested in using orthonormal polynomials as the basis functions in the functional approximation. The orthonormality property is important for the numerical stability, since polynomials of a high degree from a family of polynomials that is not orthonormal will have values that are big, leading to numerical errors very quickly.

A few observations before we begin. First, coding the different orthonormal polynomials by hand is surprisingly hard, since it is quite easy to make small mistakes but hard to debug. If you ever need to do so, the orthogonality and orthonomality conditions are useful to check if everything is correct. Second, the approximation of continuous functions over compacts is justified by Jackson's theorems. Third, better approximations require a higher number of basis functions. Fourth, to approximate more complex functions you need a higher number of $(x, f(x))$ observations.

5.2 Hermite Polynomials

The motivation for Hermite polynomials is to find a family of polynomials that is orthogonal on the real line. We can define Hermite polynomials recursively. Let H_k denote the Hermite polynomial of order k, then:

$$\begin{aligned}
&H_0(x) \equiv 1\\
&H_1(x) \equiv 2x\\
&H_k(x) \equiv 2xH_{k-1}(x) - 2(k-1)H_{k-2}(x)
\end{aligned}$$

The Hermite polynomials are orthogonal on the real line with respect to the weight function:

$$w(x) = \mathrm{e}^{-x^2}$$

Hermite polynomials are available in `scipy.special.hermite`:

```python
from scipy.special import hermite
from scipy.integrate import quad

# Verify orthogonality
def w(x): return np.exp(-x**2)
res, _ = quad(lambda x: w(x)*hermite(2)(x)*hermite(3)(x),
              -np.inf, np.inf)
print(res)
res, _ = quad(lambda x: w(x)*hermite(1)(x)*hermite(2)(x),
              -np.inf, np.inf)
print(res)
```

It is possible to show that:

$$\int_{-\infty}^{\infty} w(x)H_k(x)H_k(x)dx = 2^k k!\sqrt{\pi}$$

Thus, we can define Hermite polynomials that are orthonormal and with weight function $w(x) = 1$, by defining:

$$\begin{aligned}\tilde{H}_k(x) &= \sqrt{\frac{w(x)}{2^k k!\sqrt{\pi}}}H_k(x) \\ &= (2^k k!\sqrt{\pi})^{-\frac{1}{2}}\mathrm{e}^{-\frac{x^2}{2}}H_k(x)\end{aligned}$$

The family of polynomials $\left\{\tilde{H}_k : k \in \mathbb{N} \cup \{0\}\right\}$ is orthonormal.

```python
from scipy.special import factorial


# Hermite Polynomials
def orthoHermite(n):
    scale = (2**n*factorial(n)*np.pi**0.5)**-0.5
    return lambda x: scale*np.exp(-x**2/2)*hermite(n)(x)


# Verify Orthonormality
res, _ = quad(lambda x: orthoHermite(2)(x)*orthoHermite(3)(x),
              -np.inf, np.inf)
print(res)
res, _ = quad(lambda x: orthoHermite(3)(x)*orthoHermite(3)(x),
              -np.inf, np.inf)
print(res)


# Visualize polynomials
```

```
fig, ax = plt.subplots()
x = np.linspace(-5, 5, 200)
for n in range(5):
    ax.plot(x, orthoHermite(n)(x), label=f'$H_{n}$')
ax.legend()
ax.set(title="Orthonormal Hermite Polynomials")
fig.show()
```

Let's use the Orthonormal Hermite polynomials as basis functions to approximate some data points:

```
# Generate data
x_data = np.linspace(-5, 5, num=100)
y_data = 2.3 * np.sin(1.2 * x_data) + np.random.normal(size=100)


# Compute approximating function from basis of 4 polynomials
def poly(x): return np.array([orthoHermite(0)(x),
                              orthoHermite(1)(x),
                              orthoHermite(2)(x),
                              orthoHermite(3)(x)])


P = poly(x_data).T
beta = np.linalg.pinv(P.T @ P) @ P.T @ y_data


def fhat(x):
    return poly(x).T @ beta


# Visualize
fig, ax = plt.subplots()
ax.scatter(x_data, y_data,
           marker='o', color='black', label='Data')
ax.plot(x_data, fhat(x_data), color='green')
fig.show()
```

Notice that we are approximating a function f based on noisy observations. The algorithm is very fast and allows for quickly increasing the number of basis functions used in the approximation.

We can obtain a similar result using the `curve_fit` function from `scipy.optimize`. The function `curve_fit` takes a function as input, and uses non-linear least-squares to estimate its parameters.

```
from scipy.optimize import curve_fit


def p(x, *args):
    params = np.array(args)
    poly = np.array([orthoHermite(0)(x),
                     orthoHermite(1)(x),
                     orthoHermite(2)(x),
                     orthoHermite(3)(x)])
    return poly.T @ params


params, _ = curve_fit(p, x_data, y_data, p0=np.ones((4, 1)))
fig, ax = plt.subplots()
ax.scatter(x_data, y_data,
           marker='o', color='black', label='Data')
ax.plot(x_data, f(x_data, *params),
        color='blue', label='curve_fit')
ax.plot(x_data, fhat(x_data),
        color='green', label='sieves')
ax.legend()
fig.show()
```

We can improve the approximation by using a higher number of basis polynomials. Let's create a function that will create p with however many polynomials we want:

```
# Generate approximating function based on a specified
# number of basis polynomials
# With curve_fit
def approximator_generator(max_degree):
    def p(x, *args):
        params = np.array(args)
        poly = np.array([orthoHermite(n)(x) for n in range(max_degree)])
        return poly.T @ params
    return p
```

If we pass the number 10, then p will use 10 orthonormal Hermite polynomials. Let's use curve_fit and the Sieves approach to approximate the function.

```
p = approximator_generator(10)
params, _ = curve_fit(p, x_data, y_data, p0=np.ones(10))


# With sieves
def poly(x): return np.array([orthoHermite(n)(x) for n in range(10)]).T
P = poly(x_data)
beta = np.linalg.pinv(P.T @ P) @ P.T @ y_data
def fhat(x): return poly(x) @ beta


# Visualize both
fig, ax = plt.subplots()
ax.scatter(x_data, y_data,
           marker='o', color='black', label='Data')
ax.plot(x_data, p(x_data, *params),
        color='blue', label='curve_fit')
ax.plot(x_data, fhat(x_data),
        color='green', label='sieves')
ax.legend()
fig.show()
```

Compare different approximations as a function of the number of basis polynomials:

```python
fig, ax = plt.subplots()
ax.scatter(x_data, y_data,
           marker='o', color='black', label='Data')
for n in [5, 15, 30]:
    p = approximator_generator(n)
    params, _ = curve_fit(p, x_data, y_data, p0=np.ones(n))
    ax.plot(x_data, p(x_data, *params), label=f'{n} polynomials')
ax.legend()
fig.show()
```

Too few basis polynomials and the approximation is bad, too many and the approximation becomes highly variable. When creating approximating functions, you might want to hard-code the number of polynomials in the basis after finishing approximating the function, since this is faster than using something like `approximator_generator`.

5.3 Piecewise Polynomials (Splines)

Polynomials are useful to approximate continuous functions, however the approximations often oscillate wildly. These oscillations are more apparent at the tails of the function we are approximating.

Consider the approximation of a simple function $f(x) = \frac{1}{1+x^2}$ over the interval $[-5, 5]$ using polynomials. The function we want to approximate is infinitely differentiable, but the approximation by polynomials will incur in a wild oscillation near the tails of the function. This problem is known as Runge's example.

```python
def runge(x):
    return 1/(1 + 25*x**2)


# Use 15 points to approximate the function
x = np.linspace(-1, 1, 15)
y = runge(x)
p = approximator_generator(12)
params, _ = curve_fit(p, x, y, p0=np.ones(12))
fhat = lambda x: p(x, *params)

# Visualize problem
fig, ax = plt.subplots()
x = np.linspace(-1, 1, 200)
ax.plot(x, runge(x),
        color='black', label="Runge's function")
ax.plot(x, fhat(x),
        color='blue', label='Approximation')
ax.legend()
fig.show()
```

The idea of piecewise polynomials is that instead of using a single polynomial to approximate a function, we could use various different polynomials to approximate different parts of the function. We say that a function $s : [a, b] \mapsto \mathbb{R}$ is a piecewise polynomial of degree k if:

- The function s is continuous over $[a, b]$;

- There exists a finite number $n+1$ of points $\{\xi_i\}_{i=1}^{n+1}$, with $a = \xi_0 < \xi_1 < \cdots < \xi_n = b$, where s is a polynomial of degree at most k on each of the intervals $[\xi_{i-1}, \xi_i], i = 1, 2, \ldots, n$.

That is, s is a continuous function, but restricted to each interval it is also a polynomial of degree at most k. The points ξ_i are called knots. If the function s is differentiable up to its degree k minus 1, then we say it is a spline function of degree k. A spline is a piecewise polynomial that is as smooth as possible, without becoming a regular polynomial.

There are various functions that are splines. One of the simplest is the linear spline. Given a set of points $\{(x_1, y_1), \cdots, (x_m, y_m)\}$, where $x_1 = a$ and $x_m = b$, we can define s on each of the intervals $[x_i, x_{i+1}]$ by:

$$s(x) \equiv \frac{(x_{i+1} - x)y + (x - x_i)y_{i+1}}{x_{i+1} - x_i}, x \in [x_i, x_{i+1}]$$

The function s is defined as a sequence of lines connecting the intervals. Notice that the function is continuous, of degree $k = 1$, but not differentiable.

We will focus the discussion on a type of splines known as B-splines. The idea of B-splines is to write s as a linear combination of various polynomials of degree k, denoted by B_i^k:

$$s(x) = \sum_i \beta_i B_i^k(x) \text{ where } x \in [a, b]$$

Each of the B_i^k polynomials satisfy the property that they are nonzero only on a small interval, but are zero on the majority of $[a, b]$. The subscript i denotes the interval $[\xi_i, \xi_{i+1}]$ over which B_i^k is nonzero. The idea is to limit the impact of the polynomials on intervals far from where they are nonzero. An additional benefit is that B-splines incur in fewer calculations than other splines.

We can define the B_i^k polynomials by:

$$B_i^k(x) = \sum_{j=i}^{i+k+1} d_j (x - \xi_j)_+^k$$
$$d_j = \prod_{l=i, l \neq j}^{i+k+1} \frac{1}{\xi_l - \xi_j}$$

Where $(a)_+ \equiv \max(a, 0)$, and the terms $(x - \xi_j)_+^k$ are known as truncated power functions.

The subscript i indicates the interval $[\xi_i, \xi_{i+1}]$ over which the polynomial B_i^k is nonzero. However, the polynomial B_i^k is also nonzero over a slightly larger interval that extends after ξ_{i+1}. This is necessary so that the resulting function is continuous. The order of the polynomial k dictates how much longer the polynomial is extended. If the order is $k = 1$, then the polynomial extends over one more knot, that is, B_i^1 is nonzero on the interval $[\xi_i, \xi_{i+2}]$. If the order is $k = 2$, then the polynomial extends over two more knots, that is, B_i^2 is nonzero on the interval $[\xi_i, \xi_{i+3}]$. And so on for higher orders. The coefficients d_j are computed so that the polynomial B_i^k is zero outside the interval $[\xi_i, \xi_{i+k+1}]$.

The polynomials B_i^k are created for each of the knots ξ_i, for $i = 0, 1, 2, \cdots, n$. Notice that when we are at the final knots, we would need more points to the right of ξ_n to compute the coefficients d_j. In practice, we may not be able to generate these additional knots, so we need to truncate some of the coefficients to zero to generate the last few B_i^k.

The basis polynomials for B-Splines are available in `scipy.signal.bspline`.

```
from scipy.interpolate import BSpline
from scipy.signal import bspline
fig, ax = plt.subplots()
x = np.linspace(-3, 3, 200)
for n in range(5):
    ax.plot(x, bspline(x, n), label=f'Order {n}')
ax.set(title='Basis B-Splines')
ax.legend()
fig.show()
```

We can fit B-splines on data using the function `scipy.interpolate.splrep`, and then use `scipy.interpolate.BSpline` to create the approximating function:

```
from scipy.interpolate import splrep, BSpline


# Use 15 points to approximate the function
x = np.linspace(-1, 1, 15)
y = runge(x)
# Approximate using B-splines of 3rd degree (cubic splines)
# On each small interval, uses basis polynomials of up to 3rd
# degree to approximate the function.
(knots, coef, degree) = splrep(x, y, k=3)
#  Combine all polynomials using the coefficients to get
# the final approximation.
fhat = lambda x: BSpline(knots, coef, degree, False)(x)
# Visualize problem
fig, ax = plt.subplots()
x = np.linspace(-1, 1, 200)
ax.plot(x, runge(x),
        color='black', label="Runge's function")
ax.plot(x, fhat(x),
        color='green', label='B-Spline Approximation')
ax.legend()
fig.show()
```

5.4 Approximating Functions of Many Variables

We can use the same methodology to approximate a function f of many variables. In this case, $f : \mathbb{R}^n \mapsto \mathbb{R}$, so that x is now a vector of dimension n. Let's consider a basis of power functions to approximate f. Let $p_k(x)$ denote a power function of degree k for the vector x. It consists of all interaction terms between the elements of x, such that the sum of the exponents of each element is equal to k. For example:

$$\begin{aligned}
p_0(x) &= 1 \\
p_1(x) &= \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
p_2(x) &= \begin{pmatrix} x_1^2 \\ x_1x_2 \\ x_2^2 \end{pmatrix} \\
p_3(x) &= \begin{pmatrix} x_1^3 \\ x_1^2x_2 \\ x_1x_2^2 \\ x_2^3 \end{pmatrix}
\end{aligned}$$

Given a choice for the highest order k of the polynomials, we can define $p(x)$ as before. For example, if we choose $k = 2$:

$$p(x) = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1x_2 \\ x_2^2 \end{pmatrix}$$

Then, given a set of observations for x and $f(x)$, we can proceed as before to estimate $\hat{\beta}$ and approximate the function f.

5.5 Built-In Interpolating Functions

SciPy offers the function `scipy.interpolate.interp1d` to interpolate 1-dimensional data. It is similar to the `minimize` function, in the sense that we can choose a the input `kind` to get different types of interpolations.

```
from scipy.interpolate import interp1d

x = np.linspace(-1, 1, 15)
y = runge(x)

# Visualize different interpolations
fig, axes = plt.subplots(nrows=2)
x_grid = np.linspace(-1, 1, 200)
for ax in axes:
    ax.plot(x_grid, runge(x_grid), color='black',
            linewidth=2, label="Runge's Function")
for kind in ['linear', 'cubic']:
    fun = interp1d(x, y, kind=kind)
    axes[0].plot(x_grid, fun(x_grid), label=f'{kind} spline')
for kind in [5, 11]:
    fun = interp1d(x, y, kind=kind)
    axes[1].plot(x_grid, fun(x_grid),
                 label=f'{kind}th degree polynomial')
```

```
axes[0].legend()
axes[1].legend()
fig.show()
```

See `interp2d` for the interpolation of 2-dimensional functions. For the interpolation of n-dimensional functions, read the documentation of `interpn`.

6 Symbolic Math

The SymPy package provides tools for symbolic mathematics in Python. It can be installed with `pip`:

```
pip install sympy
```

Check its documentation page for usage details.

Due to time constraints, we will not explore this package, although if you have the time, it is very useful to learn how to use it. You can create scripts that show how to solve models step by step, or use SymPy as a checker for your calculations on paper.

7 Assignment

Problem 1 *(Conditional Maximum Likelihood Estimator)*

Consider the model:

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

where $\varepsilon \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$*. We can write the conditional density of* $Y|X$ *as the density of a normal (due to* ε*) with its mean shifted by* $\beta_0 + \beta_1 X$*:*

$$f_{\{Y|X\}}(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-(\beta_0+\beta_1 X))^2}{2\sigma^2}}$$

We can use this density to write the (conditional) likelihood given i.i.d. observations $\{Z_i \equiv (Y_i, X_i)\}_{i=1}^n$*:*

$$L(\beta_0, \beta_1, \sigma; Z) = \Pi_{i=1}^n f_{\{Y|X\}}(y; \beta_0, \beta_1, \sigma)$$

Thus, we can write the (conditional) maximum likelihood estimator for β_0*,* β_1 *and* σ *as:*

$$(\hat{\beta}_0, \hat{\beta}_1, \hat{\sigma}) = \underset{\beta_0,\beta_1,\sigma}{\operatorname{argmax}} \sum_{i=1}^{n} -\frac{1}{2} ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}(y_i - \beta_0 - \beta_1 x_i)^2$$

Given appropriate values for β_0 *and* β_1*, simulate* 1000 *observations for* X *from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.*

Problem 2 *(Continuation)*

Using the simulated data from the previous exercise, estimate the parameters with the `linreg_ols` *function. Now, estimate the parameters with MLE. You can use* `minimize` *to find the estimates. Compare the results.*

Problem 3 *(Continuation)*

Compute the gradient of the log-likelihood function. Use `minimize` *and the appropriate optimization method while supplying the gradient of the log-likelihood function to estimate the parameters via MLE.*

Problem 4 *(Continuation)*

We can also supply the Hessian to the `minimize` *function. Use* `minimize` *with the gradient and the Hessian of the log-likelihood function to estimate the parameters via MLE.*

Problem 5 *(Probit)*

The Probit model is used to analyze data where the dependent variable is binary ($Y \in \{0, 1\}$). In this context, we can write the conditional probability of Y as:

$$\begin{cases} \mathbb{P}(Y = 1|X;\theta) & = \Phi(X'\theta) \\ \mathbb{P}(Y = 0|X;\theta) & = 1 - \Phi(X'\theta) \end{cases}$$

where Φ is the cdf of the standard normal distribution, X is a vector of independent random variables and θ is a vector of unobserved parameters. It is possible to write the conditional probability of Y in a single equation, since Y can only take binary values:

$$\mathbb{P}(Y|X;\theta) = \Phi(X'\theta)^Y (1 - \Phi(X'\theta))^{1-Y}$$

Given i.i.d. observations $\{(y_i, x_i)\}_{i=1}^n$, we can write the (conditional) log-likelihood function for $Z \equiv (Y, X)$ as:

$$l(\theta; Z) = \sum_{i=1}^{n} y_i ln\left(\Phi(x_i'\theta)\right) + (1 - y_i) ln\left(1 - \Phi(x_i'\theta)\right)$$

Given appropriate values for θ, simulate 1000 *observations for X from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.*

Problem 6 *(Continuation)*

Estimate the parameters using MLE and `minimize`*.*

Problem 7 *(Logit)*

The Logit model is an alternative to the Probit model when it comes to analyzing binary data. It can be argued that the interpretation of the parameters is more direct in the case of the Logit model. We can write the conditional probability of Y as:

$$\begin{cases} \mathbb{P}(Y = 1|X;\theta) & = \Lambda(X'\theta) \\ \mathbb{P}(Y = 0|X;\theta) & = 1 - \Lambda(X'\theta) \end{cases}$$

where $\Lambda(v) = \frac{e^v}{1+e^v}$ is the cdf of the logistic distribution,X is a vector of independent random variables and θ is a vector of unobserved parameters. The above is equivalent to:

$$\mathbb{P}(Y|X;\theta) = \Lambda(X'\theta)^Y (1 - \Lambda(X'\theta))^{1-Y}$$

Given i.i.d. observations $\{(y_i, x_i)\}_{i=1}^n$, we can write the (conditional) log-likelihood function for $Z \equiv (Y, X)$ as:

$$l(\theta; Z) = \sum_{i=1}^{n} y_i ln\left(\Lambda(x_i'\theta)\right) + (1 - y_i) ln\left(1 - \Lambda(x_i'\theta)\right)$$

Given appropriate values for θ, simulate 1000 observations for X from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.

Problem 8 *(Continuation)*
Estimate the parameters using MLE and `minimize`.

Problem 9 *(Continuation) (Optional)*
Derive the gradient and Hessian of the log-likelihood in this case. (Hint: Look at Equation (8.1.6) in Hayashi's Econometrics) Estimate the parameters using MLE and `minimize`*, while supplying the gradient and Hessian.*

Problem 10 *(Dependent Observations)*
Consider a Gaussian AR(1) process:

$$y_t = \alpha + \beta y_{t-1} + \varepsilon_t$$

where the $\varepsilon_t \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$ and are i.i.d, and $|\beta| < 1$. Simulate 1000 observations given an appropriate initial value for y_0.
The log-likelihood in this case is given by:

$$l(\theta; Z) = \frac{1}{n}\sum_{t=1}^{n}\left[-\frac{1}{2}ln(2\pi) - \frac{1}{2}ln\left(\sigma^2\right) - \frac{1}{2\sigma^2}(y_t - \alpha - \beta y_{t-1})^2\right] + \\ + \frac{1}{n}\left[-\frac{1}{2}ln(2\pi) - \frac{1}{2}ln\left(\frac{\sigma^2}{1-\beta^2}\right) - \frac{(y_0 - \frac{\alpha}{1-\beta})^2}{2\frac{\sigma^2}{1-\beta^2}}\right]$$

Implement the log-likelihood and estimate the parameters via MLE.

Problem 11 *(Model Selection)*
We now consider the estimation of a linear regression via ordinary least-squares with the addition of a penalization term. The penalization term is a function of the magnitude of the parameters of the model. When we minimize the squared errors taking into account the penalization, some of the parameter estimates can be zero, leading to the estimation of a simpler model. For a good overview, read this page on Lasso.
Consider the model:

$$Y = \beta_0 + \sum_{i=20} \beta_i X_i + \varepsilon$$

where $\varepsilon \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$, and each of the X_i's are drawn from a normal distribution.
Fix the values of the β_i's, but let $\beta_1 = \beta_2 = 0$. Simulate the data for $\{(Y_i, X_{i,1}, \dots, X_{i,20})\}_{i=1}^n$, with $n = 1000$. Estimate the parameters via least-squares with `linreg_ols`.

Problem 12 *(Continuation)*
Estimate the parameters by minimizing the squared errors with the penalization term added:

$$\min_{\beta_0,\beta_1,\ldots,\beta_{20}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i,1} - \ldots - \beta_{20}x_{i,20})^2$$
$$\textit{subject to} \sum_{i=0}^{20} |\beta|_i \leq \lambda$$

for some $\lambda > 0$. Use the simulated data and the function `minimize` *in the estimation.*

Problem 13 *(Continuation)*
What happens to the parameters when n increases from 1000 *to* 10,000 *and to* 100,000*? Why?*

Problem 14 *(Continuation)*
What happens to the final squared-errors of the regression when λ increases? Why?

Problem 15 *(Continuation) (Optional)*
Consider the Lagrangian form of the problem:

$$\min_{\beta_0,\beta_1,\ldots,\beta_{20}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i,1} - \ldots - \beta_{20}x_{i,20})^2 + \lambda\sum_{i=0}^{20}|\beta_i|$$

The constrained problem was written as an unconstrained problem. Estimate the parameters and repeat the analysis from the previous two problems. Compare the results.

Problem 16 *(Growth Model)*
Consider a deterministic growth model, where an agent decides between consumption (c_t) and investment in capital (k_t), while maximizing his utility. We can write this problem as:

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$
$$\textit{subject to} \begin{cases} k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

Write the problem as a Bellman equation. You should obtain an equation similar to:

$$V(k) = \max_{k'} f(k,k') + \beta V(k')$$

Problem 17 *(Continuation)*
Consider $U(c;\sigma) = \frac{c^{1-\sigma}-1}{1-\sigma}$. Obtain the Euler equation for this problem in terms of the consumption c.

Problem 18 *(Continuation) (Optional)*
Use SymPy (Symbolic Mathematics Package for Python) toolbox to obtain the Euler equation for this problem in terms of the consumption c.

Problem 19 *(Continuation)*
Assume we are in a steady state ($c = c' = c^*$ and $k = k' = k^*$), then use the Euler equation to compute the steady state value of k.

Problem 20 *(Continuation)*
Solve the problem by Value Function Iteration. Consider $\sigma = 2$, $\beta = 0.95$, $\delta = 0.1$ and $\alpha = 0.33$. Use the steady state value of k to create a grid for the possible values of k, say 100 points between $0.25k^*$ and $1.75k^*$. Start with a guess for V over the grid, for example $V(k) = 0$ for all k in the grid. Use the SciPy minimization function `minimize` to solve for k. You may want to add the constraint that c should always be positive.

Problem 21 *(Continuation) Plot the value for different capital levels. Interpret.*

Problem 22 *(Continuation) Plot the optimal policy function (choice of k' given k). Plot a 45 degree line with the policy function. Interpret in terms of a steady state.*