Python Language Companion to

Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares

DRAFT

Jessica Leung and Dmytro Matsypura

January 10, 2020

Contents

Preface

This *Python Language Companion* is written as a supplement to the book *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares* written by *Stephen Boyd* and *Lieven Vandenberghe* (referred to here as VMLS). It is mostly a transliteration of the Julia Companion (by Stephen Boyd and Lieven Vandenberghe) to Python. This companion is meant to show how the ideas and methods in VMLS can be expressed and implemented in the programming language Python. We assume that the reader has installed Python, or is using an interactive Python shell online, and understands the basics of the language. We also assume that the reader has carefully read the relevant sections of VMLS. The organization of this companion follows the topics in VMLS.

You will see that mathematical notation and Python syntax are quite similar, but not the same. You must be careful never to confuse mathematical notation and Python syntax. In these notes, we write mathematical notation in standard mathematics font, e.g., $y = 2x$. Python code and expressions are written in a fixed-width typewriter font, e.g., `y = 2*x`. We encourage you to copy and paste our Python code into a Python console, interactive Python session or Jupyter Notebook, to test them out, and maybe modify them and rerun.

Python is a very powerful and efficient language, but in this companion, we only demonstrate a small and limited set of its features. We focus on numerical operations in Python, and hence we use the `numpy` package to facilitate our computations. The code written in this companion is based on Python 3.7. We also use a number of other standard packages such as `time` and `sklearn`. The next section, *Getting started with Python*, explains how to install Python on your computer.

We consider this companion to be a draft. We'll be updating this language companion occasionally, adding more examples and fixing typos as we become aware of them. So you may wish to check periodically whether a newer version is available.

We want to thank Stephen Boyd and Lieven Vandenberghe for reviewing this Python companion and providing insightful comments.

To cite this document: Leung J, and Matsypura D (2019) Python Language Companion to "Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares".

Jessica Leung	*Sydney, Australia*
Dmytro Matsypura	*Sydney, Australia*

Getting started with Python

Installing Python. We recommend beginners to install the Anaconda distribution of Python powered by Continuum Analytics because it comes as a convenient combo of many basic things you need to get started with Python. You can download the Anaconda distribution from `https://www.anaconda.com/download/`. You should choose the installer according to your operating system (Windows / Mac). This document assumes you are using Python 3.7 (Graphical Installer). You can then follow the instructions and install Anaconda with default settings.

You can confirm that Anaconda is installed and working by requesting the list of installed packages. To do so, you should type the command `conda list` into your terminal:

- For Windows users, you can search for `anaconda prompt` from the start menu. Enter `conda list` into the terminal (looks like a black box). Press Enter.
- For Mac users, you can press `command + space` and search for `terminal` in the spotlight search. Enter `conda list` into the terminal (looks like a white box). Press Enter.

If you see a list of packages showing in your terminal, Anaconda is correctly installed.

A significant advantage of using Anaconda distribution is that it includes a combination of commonly used packages and user-friendly applications. It also allows you to manage different environments and packages in a stable and convenient way. Popular packages such as `numpy` and `sklearn` are included in the distribution, and hence, in most cases, no additional packages will be required. On a few occasions, we use functions from other packages, but we will mention this in the text when we use these functions.

Installing Other Packages. If the package you need is not included in the Anaconda distribution, you can install that package manually. Majority of the packages can be added easily to your current Anaconda distribution with one command. For example, if you would like to install the `pandas` package, you should type the following into your terminal/anaconda prompt: `conda install pandas`

Sometimes you may encounter packages that require a few more steps during installation; in this case, you should follow the instructions in the package documentation.

A few words about Jupyter Notebook

We recommend using Jupyter Notebook, especially if you are a beginner. Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. You can edit and run your code in a web browser[1]. Apart from Python, Jupyter Notebook also recognizes other programming languages such as R and Julia. For these reasons, we are using Jupyter Notebook throughout this companion. However, you can also run the code in Python directly.

To launch Jupyter Notebook, enter `jupyter notebook` in the terminal. You will be guided to your home directory. You can now click into the folder where you would like to save your notebook. You can start a new notebook by clicking the `New` button in the top right corner. A drop-down menu will appear with the option of `Python 3` which you should click on. You can rename your notebook simply by clicking on the name of your notebook.

Try running the following piece of code:

```
In [ ]: print('Hello, world.')
```

You will see the phrase 'Hello world.' displayed as an output.

```
Hello, world.
```

You will find the `print` function very useful, especially when using Python outside of Jupyter Notebook, as Python does not usually display the values of variables. If you are unsure whether your code runs correctly, you can always `print` a variable to check its value.

[1]For Windows users, we recommend to set Google Chrome as default browser; for Mac users, we recommend Safari or Google Chrome.

However, Jupyter Notebook will display the value of a variable if you simply enter the variable name. For example:

```
In [ ]: x = 3.5
        x

Out[ ]: 3.5
```

We use both methods throughout this companion.

Remember: do not close the terminal while you are using Jupyter Notebook as this will kill your kernel, and you won't be able to run or save your code.

Before you run a cell in Jupyter Notebook, you will see an empty set of square brackets in the top left corner of a cell. After you run that cell, you will see a number appear inside the brackets. This number indicates the order of the cell that you have run. Sometimes instead of a number, you may see an asterisk (*) inside the brackets after you hit run. This indicates that the computer is still performing the calculation and needs more time to finish. Hitting the run button multiple times when you see an asterisk usually doesn't help, so please be patient.

Using Packages. To use a certain package, you simply need to include an `import` statement in your code. For example, if you want to use the package `numpy`:

```
In [ ]: import numpy
```

This will import the package `numpy` and you can call the package by the name `numpy` when you want to use it.
Alternatively, you can type the following:

```
In [ ]: import numpy as np
```

Then you can simply type `np` when you want to call the package. For every Jupyter Notebook, you only need to include and run the `import` statement once and you can use the imported package for the rest of the same notebook. In chapter 1, we include the `import numpy as np` statement in the code whenever we use the `numpy` package. After chapter 1, our code assumes that you have imported the `numpy` package and it can be called using `np`.

Part I.

Vectors

1. Vectors

1.1. Vectors

The simplest way to represent vectors in Python is using a *list* structure. A vector is constructed by giving the list of elements surrounded by square brackets, with the elements separated by commas. The assignment operator `=` is used to give a name to the list. The `len()` function returns the size (dimension).

```
In [ ]: x = [-1.1, 0.0, 3.6, -7.2]
        len(x)
```

```
Out[ ]: 4
```

Some common mistakes. Don't forget the commas between entries, and be sure to use square brackets. Otherwise, you'll get things that may or may not makes sense in Python, but are not vectors.

```
In [ ]: x = [-1.1  0.0  3.6  -7.2]
```

```
Out[ ]:   File "<ipython-input-1-18f108d7fe41>", line 1
            x = [-1.1  0.0  3.6  -7.2]
                         ^
        SyntaxError: invalid syntax
```

Here Python returns a `SyntaxError` suggesting the expression that you entered does not make sense in Python. (Depending on the kind of error that occurred, Python may return other types of error messages. Some of the error messages are quite self-explanatory, some are not. See Appendix A for other types of error messages from Python and possible solutions.)

```
In [ ]: y = (1,2)
```

Here `y` is a tuple consisting of two scalars. It is neither a list nor a vector.

Another common way to represent vectors in Python is to use a numpy array. To do so, we must first import the numpy package.

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        len(x)

Out[ ]: 4
```

We can initialize numpy arrays from Python list. Here, to call the package we put `np.` in front of the array structure. *Note*: we have parentheses outside the square brackets. A major advantage of numpy arrays is their ability to perform linear algebra operations which make intuitive sense when we are working with vectors.

Indexing. A specific element x_i is extracted with the expression `x[i]` where `i` is the index (which runs from 0 to $n-1$, for an n-vector).

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        x[2]

Out[ ]: 3.6
```

If we used array indexing on the left-hand side of an assignment statement, then the value of the corresponding element changes.

```
In [ ]: x[2] = 20.0
        print(x)

        [-1.1  0.  20.  -7.2]
```

-1 is a special index in Python. It is the index of the last element in an array.

```
In [ ]: x[-1]

Out[ ]: -7.2
```

Assignment versus copying. The behavior of an assignment statement `y = x` where `x` is an array may be surprising for those who use other languages like Matlab or Octave. The assignment expression gives a new name `y` to the *same* array that is already referenced by `x` instead of creating a copy of `x`.

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        y = x
        x[2] = 20.0
        print(y)
```

```
[-1.1  0.   20.  -7.2]
```

To create a new copy of array `x`, the method `copy` should be used.

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        y = x.copy()
        x[2] = 20.0
        print(y)
```

```
[-1.1  0.   3.6 -7.2]
```

Vector equality. Equality of vectors is checked using the relational operator `==` (double equal signs). The Python expression evaluates to `True` if the expression on the left and right-hand side of the relational operator is equal, and to `False` otherwise.

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        y = x.copy()
        x == y
```

```
Out[ ]: array([ True,  True,  True,  True])
```

```
In [ ]: import numpy as np
        x = np.array([-1.1, 0.0, 3.6, -7.2])
        y = x.copy()
        y[3] = 9.0
        x == y
```

```
Out[ ]: array([ True,  True,  True, False])
```

Here four evaluations (`True` or `False`) are shown because applying relational operator on numpy array performs element-wise comparison. However, if we apply the relational operator on list structures, the Python expression evaluates to `True` only when both sides have the same length and identical entries.

```
In [ ]: x = [-1.1, 0.0, 3.6, -7.2]
        y = x.copy()
        x == y

Out[ ]: True
```

```
In [ ]: x = [-1.1, 0.0, 3.6, -7.2]
        y = x.copy()
        y[3] = 9.0
        x == y

Out[ ]: False
```

Scalars versus 1-vectors. In the mathematical notations of VMLS, 1-vector is considered as a scalar. However, in Python, 1-vectors are not the same as scalars. For list structure, Python distinguishes 1-vector (list with only one element) `[2.4]` and the number `2.4`.

```
In [ ]: x = 2.4
        y = [2.4]
        x == y

Out[ ]: False
```

```
In [ ]: y[0] == 2.4

Out[ ]: True
```

In the last line, Python checks whether the first element of `y` is equal to the number `2.4`. For numpy arrays, Python compares the elements inside `np.array([2.4])` with the scalar. In our example, Python compares the first (and only) element of `y` to the number `2.4`. Hence, numpy 1-vectors behave like scalars.

```
In [ ]: import numpy as np
        x = 2.4
        y = np.array([2.4])
        x == y

Out[ ]: array([ True])
```

Block and stacked vectors. In Python, we can construct a block vector using the numpy function `concatenate()`. Remember you need an extra set of parentheses over the vectors that you want to concatenate. The use of numpy array or list structure does not create a huge difference here.

```
In [ ]: import numpy as np
        x = np.array([1, -2])
        y = np.array([1,1,0])
        z = np.concatenate((x,y))
        print(z)
```

```
[ 1 -2  1  1  0]
```

Some common mistakes. There are few Python operations that appear to be able to construct a block or stacked vector but do not. For example, `z = (x,y)` creates a tuple of two vectors; `z = [x,y]` creates an array of the two vectors. Both of these are valid Python expression but neither of them is the stacked vector.

Subvectors and slicing. In VMLS, the mathematical notation $r : s$ denotes the index range $r, r+1, \ldots, s$ and $x_{r:s}$ denotes the slice of the vector x from index r to s. In Python, you can extract a slice of a vector using an index range as the argument. Remember, Python indices start from 0 to $n-1$. For the expressing `x[a:b]` for array `x`, the slicing selects the element from index `a` to index `b-1`. In the code below, `x[1:4]` select element from index 1 to 3, which means the second to the fourth elements are selected.

```
In [ ]: import numpy as np
        x = np.array([1,8,3,2,1,9,7])
        y = x[1:4]
        print(y)
```

```
[8 3 2]
```

You can also use index ranges to assign a slice of a vector. In the code below, we reassigned index 3 to 5 in array `x`.

```
In [ ]: x[3:6] = [100,200,300]
        print(x)
```

```
[  1   8   3 100 200 300   7]
```

As a result, you can see the 4th, 5th and 6th elements in the array are reassigned.

You can also use slicing to select all elements in the array starting from a certain index

```
In [ ]: import numpy as np
        x = np.array([1,8,3,2,1,9,7])
        x[2:]
```

```
Out[ ]: array([3, 2, 1, 9, 7])
```

```
In [ ]: x[:-1]
```

```
Out[ ]: array([1, 8, 3, 2, 1, 9])
```

Here the first expression selects all elements in `x` starting from index 2 (the third element). The second expression selects all elements in `x` except the last element.

Python indexing into arrays Python slicing and subvectoring is much more general than the mathematical notation we use in VMLS. For example, one can use a number range with a third argument, that gives the increment between successive indexes. For example, the index range `1:5:2` is the list of numbers $8, 2$. The expression `x[1:4:2]` extracts the second (index 1) to the fifth (index 4) element with an increment of 2. Therefore, the second and fourth entries of `x` are extracted. You can also use an index range that runs backward. The Python expression `x[::-1]` gives the reversed vector, *i.e.*, the vector with the same coefficients but in opposite order.

Vector of first differences. Here we use slicing to create an $(n-1)$-vector d which is defined as the first difference vector $d_i = x_{i+1} - x_i$, for $i = 1, \ldots, n-1$, where x is an n-vector.

```
In [ ]: import numpy as np
        x = np.array([1,8,3,2,1,9,7])
        d = x[1:] - x[:-1]
        print(d)
```

```
[ 7 -5 -1 -1  8 -2]
```

Lists of vectors. An ordered list of n-vectors might be denoted in VMLS as $a_1, \ldots, a_k$ or $a^{(1)}, \ldots, a^{(k)}$ or just as a, b, c. There are several ways to represent lists of vectors in Python. If we give the elements of the list, separated by commas, and surrounded by square brackets, we form a one-dimensional arrays of vectors or a list of lists. If instead, we use parentheses as delimiters, we obtain a *tuple*.

```
In [ ]: x = [1,0]
        y = [1,-1]
        z = [0,1]
        list_of_vectors = [x,y,z]
        list_of_vectors[1] #Second element of list
```

```
Out[ ]: [1, -1]
```

```
In [ ]: list_of_vectors[1][0] #First entry in second element of list
```

```
Out[ ]: 1
```

```
In [ ]: tuple_of_vectors = (x,y,z)
        tuple_of_vectors[1] #Second element of list
```

```
Out[ ]: [1, -1]
```

```
In [ ]: tuple_of_vectors[1][0] #First entry in second element of list
```

```
Out[ ]: 1
```

Note that the difference between `[x,y,z]` (an array of arrays or a list of lists) and a stacked vector of x, y, and z, obtained by concatenation.

Zero vectors. We can create a zero vector of size n using `np.zeros(n)`. The expression `np.zeros(len(x))` creates a zero vector with the same dimension as vector `x`.

```
In [ ]: import numpy as np
        np.zeros(3)
```

```
Out[ ]: array([0., 0., 0.])
```

Unit vectors. We can create e_i, the ith unit vector of length n using index.

```
In [ ]: import numpy as np
        i = 2
        n = 4
        x = np.zeros(n)
        x[i] = 1
        print(x)
```

```
[0. 0. 1. 0.]
```

Ones vector. We can create a ones vector of size n using `np.ones(n)`. The expression `np.ones(len(x))` creates a ones vector with the same dimension as vector `x`.

```
In [ ]: import numpy as np
        np.ones(3)
```

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

Random vectors. Sometimes it is useful to generate random vectors to check our algorithm or test an identity. In Python, we generate a random vector of size n using `np.random.random(n)`.

```
In [ ]: np.random.random(2)
```

```
Out[ ]: array([0.79339885, 0.60803751])
```

Plotting. There are several external packages for creating plots in Python. One such package is `matplotlib`, which comes together in the Anaconda distribution. Otherwise, you can also add (install) it manually; see page vii. In particular, we use the module `pyplot` in `matplotlib`. Assuming the `matplotlib` package had been installed, you import it into Python using the command `import matplotlib.pyplot as plt`. (Now `plt` acts as an alias for `matplotlib.pyplot`.) After that, you can access the Python commands that create or manipulate plots.

For example, we can plot the temperature time series in Figure 1.3 in VMLS using the code below; the last line saves the plot in the file `temperature.pdf`. The result is shown in Figure 1.1.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        temps = [ 71, 71, 68, 69, 68, 69, 68, 74, 77, 82, 85, 86, 88, 86,
        ↪   85, 86, 84, 79, 77, 75, 73, 71, 70, 70, 69, 69, 69, 69, 67,
        ↪   68, 68, 73, 76, 77, 82, 84, 84, 81, 80, 78, 79, 78, 73, 72,
        ↪   70, 70, 68, 67 ]
        plt.plot(temps, '-bo')
        plt.savefig('temperature.pdf', format = 'pdf')
plt.show()
```

The syntax `-bo` indicates plotting with line (`-`) with circle marker (`o`) in blue (`b`). To show the plot in the interactive session or the notebook, we need to set the interactive output on with `plt.ion()` and then use the command `plt.show()`.

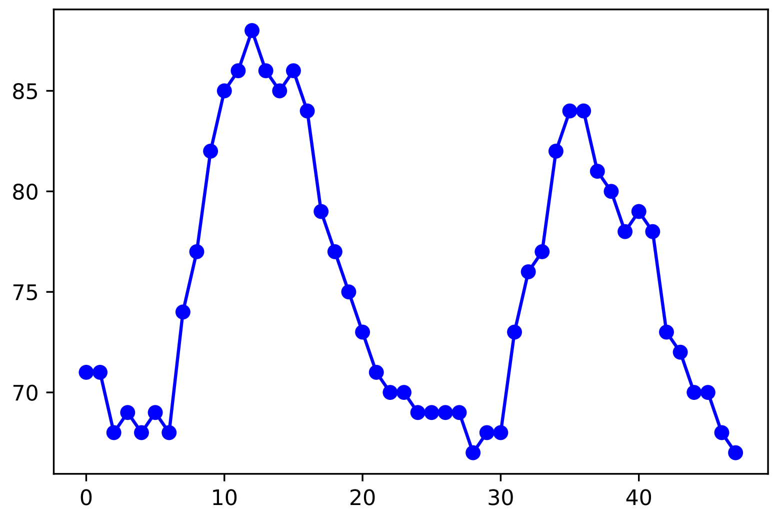


Figure 1.1.: Hourly temperature in downtown Los Angeles on August 5 and 6, 2015 (starting at 12:47AM, ending at 11:47PM).

1.2. Vector addition

Vector addition and subtraction. If `x` and `y` are numpy arrays of the same size, `x+y` and `x-y` give their sum and difference, respectively.

```
In [ ]: import numpy as np
        x = np.array([1,2,3])
        y = np.array([100,200,300])
        print('Sum of arrays:', x+y)
        print('Difference of arrays:', x-y)
```

```
Sum of arrays: [101 202 303]
Difference of arrays: [ -99 -198 -297]
```

Sometimes when we would like to print more than one value, we may add a piece of string in front of the value, followed by a comma. This allows us to distinguish between the values are we printing.

1.3. Scalar-vector multiplication

Scalar-vector multiplication and division. If `a` is a number and `x` is a numpy array (vector), you can express the scalar-vector product either as `a*x` or `x*a`.

```
In [ ]: import numpy as np
        x = np.array([1,2,3])
        print(2.2*x)
```

```
[2.2 4.4 6.6]
```

You can carry out scalar-vector division as `x/a`.

```
In [ ]: import numpy as np
        x = np.array([1,2,3])
        print(x/2.2)
```

```
[0.45454545 0.90909091 1.36363636]
```

Remark: For Python 2.x, integer division is used when you use the operator `/` on scalars. For example, `5/2` gives you 2. You can avoid this problem by adding decimals to the integer, *i.e.*, `5.0/2`. This gives you 2.5.

Scalar-vector addition. In Python, you can add a scalar `a` and a numpy array (vector) `x` using `x+a`. This means that the scalar is added to each element of the vector. This is, however, NOT a standard mathematical notation. In mathematical notations, we should denote this as, e.g. $x + a\mathbf{1}$, where x is an n-vector and a is a scalar.

```
In [ ]: import numpy as np
        x = np.array([1,2,3,4])
        print(x + 2)
```

```
[3 4 5 6]
```

Elementwise operations. In Python we can perform elementwise operations on numpy arrays. For numpy arrays of the same length `x` and `y`, the expressions `x * y`, `x / y` and `x ** y` give the resulting vectors of the same length as x and y and ith element x_iy_i,

x_i/y_i, and $x_i^{y_i}$ respectively. (In Python, scalar a to the power of b is expressed as `a**b`)

As an example of elementwise division, let's find the 3-vector of asset returns r from the (numpy arrays of) initial and final prices of assets (see page 22 in VMLS).

```
In [ ]: import numpy as np
        p_initial = np.array([22.15, 89.32, 56.77])
        p_final = np.array([23.05, 87.32, 53.13])
        r = (p_final - p_initial) / p_initial
        r
```

```
Out[ ]: array([ 0.04063205, -0.0223914 , -0.06411837])
```

Linear combination. You can form a linear combination is Python using scalar-vector multiplication and addition.

```
In [ ]: import numpy as np
        a = np.array([1,2])
        b = np.array([3,4])
        alpha = -0.5
        beta = 1.5
        c = alpha*a + beta*b
        print(c)
```

```
[4. 5.]
```

To illustrate some additional Python syntax, we create a function that takes a list of coefficients and a list off vectors as its argument (input), and returns the linear combination (output).

```
In [ ]: def lincomb(coef, vectors):
            n = len(vectors[0])
            comb = np.zeros(n)
            for i in range(len(vectors)):
                comb = comb + coef[i] * vectors[i]
            return comb

        lincomb([alpha, beta], [a,b])
```

```
Out[ ]: array([4., 5.])
```

A more compact definition of the function is as follows.

```
In [ ]: def lincomb(coef, vectors):
            return sum(coef[i]*vectors[i] for i in range(len(vectors)))
```

Checking properties. Let's check the distributive property

$$\beta(a+b) = \beta a + \beta b$$

which holds for any two n-vector a and b, and any scalar β. We'll do this for $n = 3$, and randomly generated a, b, and β. (This computation does not show that the property always holds; it only show that it holds for the specific vectors chosen. But it's good to be skeptical and check identities with random arguments.) We use the `lincomb` function we just defined.

```
In [ ]: import numpy as np
        a = np.random.random(3)
        b = np.random.random(3)
        beta = np.random.random()
        lhs = beta*(a+b)
        rhs = beta*a + beta*b
        print('a :', a)
        print('b :', b)
        print('beta :', beta)
        print('LHS :', lhs)
        print('RHS :', rhs)
```

```
a : [0.81037789 0.423708   0.76037206]
b : [0.45712264 0.73141297 0.46341656]
beta : 0.5799757967698047
LHS : [0.73511963 0.6699422  0.70976778]
RHS : [0.73511963 0.6699422  0.70976778]
```

Although the two vectors `lhs` and `rhs` are displayed as the same, they might not be exactly the same, due to very small round-off errors in floating point computations. When we check an identity using random numbers, we can expect that the left-hand and right-hand sides of the identity are not exactly the same, but very close to each other.

1.4. Inner product

Inner product. The inner product of n-vector x and y is denoted as $x^T y$. In Python the inner product of `x` and `y` can be found using `np.inner(x,y)`

In [ ]:
```
import numpy as np
x = np.array([-1,2,2])
y = np.array([1,0,-3])
print(np.inner(x,y))
```

```
-7
```

Alternatively, you can use the `@` operator to perform inner product on numpy arrays.

In [ ]:
```
import numpy as np
x = np.array([-1,2,2])
y = np.array([1,0,-3])
x @ y
```

Out[ ]:
```
-7
```

Net present value. As an example, the following code snippet finds the net present value (NPV) of a cash flow vector `c`, with per-period interest rate `r`.

In [ ]:
```
import numpy as np
c = np.array([0.1,0.1,0.1,1.1]) #cash flow vector
n = len(c)
r = 0.05 #5% per-period interest rate
d = np.array([(1+r)**-i for i in range(n)])
NPV = c @ d
print(NPV)
```

```
1.236162401468524
```

In the fifth line, to get the vector `d` we raise the scalar `1+r` element-wise to the powers given in the range `range(n)` which expands to `0, 1,2, ..., n-1`, using list comprehension.

Total school-age population. Suppose that the 100-vector x gives the age distribution of some population, with x_i the number of people of age $i-1$, for $i = 1, \ldots, 100$. The total number of people with age between 5 and 18 (inclusive) is given by

$$x_6 + x_7 + \cdots + x_{18} + x_{19}$$

We can express this as $s^T x$ where s is the vector with entries one for $i = 6, \ldots, 19$ and zero otherwise. In Python, this is expressed as

In [ ]:
```
s = np.concatenate([np.zeros(5), np.ones(14), np.zeros(81)])
school_age_pop = s @ x
```

Several other expressions can be used to evaluate this quantity, for example, the expression `sum(x[5 : 19 ])`, using the Python function `sum`, which gives the sum of entries of vector.

1.5. Complexity of vector computations

Floating point operations. For any two numbers a and b, we have $(a+b)(a-b) = a^2-b^2$. When a computer calculates the left-hand and right-hand side, for specific numbers a and b, they need not be exactly the same, due to very small floating point round-off errors. But they should be very nearly the same. Let's see an example of this.

In [ ]:
```
import numpy as np
a = np.random.random()
b = np.random.random()
lhs = (a+b) * (a-b)
rhs = a**2 - b**2
print(lhs - rhs)
```

```
4.336808689942018e-19
```

Here we see that the left-hand and right-hand sides are not exactly equal, but very very close.

Complexity. You can time a Python command using the `time` package. The timer is not very accurate for very small times, say, measured in microseconds (10^{-6} seconds). You should run the command more than once; it can be a lot faster on the second or subsequent runs.

In [ ]:
```
import numpy as np
import time
a = np.random.random(10**5)
```

```
b = np.random.random(10**5)
start = time.time()
a @ b
end = time.time()
print(end - start)
```

```
0.0006489753723144531
```

In [ ]:
```
start = time.time()
a @ b
end = time.time()
print(end - start)
```

```
0.0001862049102783203
```

The first inner product, of vectors of length 10^5, takes around 0.0006 seconds. This can be predicted by the complexity of the inner product, which is $2n-1$ flops. The computer on which the computations were done is capable of around 1 Gflop/s. These timings, and the estimate of computer speed, are very crude.

Sparse vectors. Functions for creating and manipulating sparse vectors are contained in the Python package `scipy.sparse`, so you need to import this package before you can use them. There are 7 classes of sparse structures in this package, each class is designed to be efficient with a specific kind of structure or operation. For more details, please refer to the `sparse` documentation.

Sparse vectors are stored as sparse matrices, i.e., only the nonzero elements are stored (we introduce matrices in chapter 6 of VMLS). In Python you can create a sparse vector from lists of the indices and values using the `sparse.coo_matrix` function. The `scipy.sparse.coo_matrix()` function create a sparse matrix from three arrays that specify the row indexes, column indexes and values of the nonzero elements. Since we are creating a row vector, it can be considered as a matrix with only 1 column and 100000 columns.

In [ ]:
```
from scipy import sparse
I = np.array([4,7,8,9])
J = np.array([0,0,0,0])
V = np.array([100,200,300,400])
A = sparse.coo_matrix((V,(I,J)),shape = (10000,1))
A
```

```
Out[ ]: <10000x1 sparse matrix of type '<class 'numpy.int64'>'
          with 4 stored elements in COOrdinate format>
```

We can perform mathematical operations on the sparse matrix `A`, then we can view the result using `A.todense()` method.

From this point onwards, in our code syntax, we assume you have imported `numpy` package using the command `import numpy as np`.

2. Linear functions

2.1. Linear functions

Functions in Python. Python provides several methods for defining functions. One simple way is to use lambda functions. A simple function given by an expression such as $f(x) = x_1 + x_2 - x_4^2$ can be defined in a single line.

```
In [ ]: f = lambda x: x[0] + x[1] - x[3]**2
        f([-1,0,1,2])

Out[ ]: -5
```

Since the function definition refers to the first, second and fourth elements of the argument `x`, these have to be well-defined when you call or evaluate `f(x)`; you will get an error if, for example, `x` has three elements or is a scalar.

Superposition. Suppose a is an n-vector. The function $f(x) = a^T x$ is linear, which means that for any n-vectors x and y, and any scalars α and β, the superposition equality

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

holds. Superposition says that evaluating f at a linear combination of two vectors is the same as forming the linear combination of f evaluated at the two vectors.

Let's define the inner product function f for a specific value of a, and then verify superposition in Python for specific values of x, y, α, and β. (This check does not show that the function is linear. It simply checks that superposition holds for these specific values.)

```
In [ ]: a = np.array([-2,0,1,-3])
        x = np.array([2,2,-1,1])
        y = np.array([0,1,-1,0])
        alpha = 1.5
```

```
beta = -3.7
LHS = np.inner(alpha*x + beta*y, a)
RHS = alpha*np.inner(x,a) + beta*np.inner(y,a)
print('LHS:', LHS)
print('RHS:', RHS)
```

```
LHS: -8.3
RHS: -8.3
```

For the function $f(x) = a^T x$, we have $f(e_3) = a_3$. Let's check that this holds in our example.

In [ ]:
```
a = np.array([-2,0,1,-3])
e3 = np.array([0,0,1,0])
print(e3 @ a)
```

```
1
```

Examples. Let's define the average function in Python and check its value of a specific vector. (Numpy also contains an average function, which can be called with `np.mean`.

In [ ]:
```
avg = lambda x: sum(x)/len(x)
x = [1,-3,2,-1]
avg(x)
```

Out[ ]:
```
-0.25
```

2.2. Taylor approximation

Taylor approximation. The (first-order) Taylor approximation of function $f : \mathbf{R}^n \to \mathbf{R}$, at the point z, is the affine function $\hat{f}(x)$ given by

$$\hat{f}(x) = f(z) + \nabla f(z)^T (x - z)$$

For x near z, $\hat{f}(x)$ is very close to $f(x)$. Let's try a numerical example (see page 36 of textbook) using Python.

In [ ]:
```
f = lambda x: x[0] + np.exp(x[1] - x[0])
grad_f = lambda z: np.array([1 - np.exp(z[1] - z[0]), np.exp(z[1]
↪ - z[0])])
z = np.array([1,2])
```

```
#Taylor approximation
f_hat = lambda x: f(z) + grad_f(z) @ (x - z)
f([1,2]), f_hat([1,2])
```

```
Out[ ]: (3.718281828459045, 3.718281828459045)
```

```
In [ ]: f([0.96, 1.98]), f_hat([0.96,1.98])
```

```
Out[ ]: (3.7331947639642977, 3.732647465028226)
```

```
In [ ]: f([1.10, 2.11]), f_hat([1.10, 2.11])
```

```
Out[ ]: (3.845601015016916, 3.845464646743635)
```

2.3. Regression model

Regression model. The regression model is the affine function of x given by $f(x) = x^T\beta + \nu$, where the n-vector β and the scalar ν are the parameters in the model. The regression model is used to guess or approximate a real or observed value of the number y that is associated with x (We'll see later how to find the parameters in a regression model using data).

Let's define the regression model for house sale price described on page 39 of VMLS, and compare its prediction to the true house sale price y for a few values of x.

```
In [ ]: # parameters in regression model
        beta = np.array([148.73, -18.85])
        v = 54.40
        y_hat = lambda x: x @ beta + v
        #Evaluate regression model prediction
        x = np.array([0.846, 1])
        y = 115
        y_hat(x), y
```

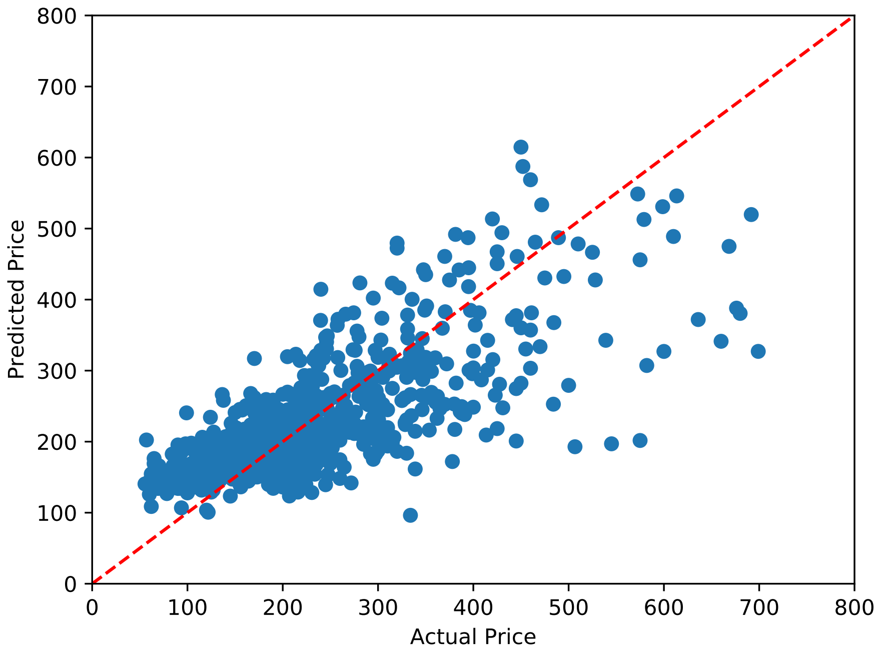
```
Out[ ]: (161.37557999999999, 115)
```

```
In [ ]: x = np.array([1.324, 2])
        y = 234.50
        y_hat(x), y
```

Out[ ]:
```
(213.61852000000002, 234.5)
```

Our first prediction is pretty bad; our second one is better. A scatter plot of predicted and actual house prices (Figure 2.4 of VMLS) can be generated as follows. We use the `house_sales_data` data set to obtain the vector `price, areas, beds` (see Appendix B). The data sets we used in this Python language companion can be found on: `https://github.com/jessica-wyleung/VMLS-py`. You can download the jupyter notebook from the repository and work on it directly or you can copy and paste the data set onto your own jupyter notebook.

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
D = house_sales_data()
price = D['price']
area = D['area']
beds = D['beds']
v = 54.4017
beta = np.array([147.7251, -18.8534])
predicted = v + beta[0]*area + beta[1]*beds
plt.scatter(price, predicted)
plt.plot((0,800),(0,800) ,ls='--', c = 'r')
plt.ylim(0,800)
plt.xlim(0,800)
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.show()
```



Predicted Price
Actual Price
0
100
200
300
400
500
600
700
800

3. Norm and distance

3.1. Norm

Norm. The norm $\|x\|$ can be computed in Python using `np.linalg.norm(x)`. (It can be evaluated in several other ways too.) The `np.linalg.norm` function is contained in the `numpy` package `linalg`.

In [ ]:
```
x = np.array([2,-1,2])
print(np.linalg.norm(x))
```

```
3.0
```

In [ ]:
```
print(np.sqrt(np.inner(x,x)))
```

```
3.0
```

In [ ]:
```
print((sum(x**2)**0.5)
```

```
3.0
```

Triangle inequality. Let's check the triangle inequality, $\|x + y\| \leq \|x\| + \|y\|$, for some specific values of x and y.

In [ ]:
```
x = np.random.random(10)
y = np.random.random(10)
LHS = np.linalg.norm(x+y)
RHS = np.linalg.norm(x) + np.linalg.norm(y)
print('LHS:', LHS)
print('RHS:', RHS)
```

```
LHS: 3.6110533105675784
RHS: 3.8023691306447676
```

Here we can see that the right-hand side is larger than the left-hand side.

RMS value. The RMS value of a vector x is $\mathbf{rms}(x) = \|x\|/\sqrt{n}$. In Python, this is expressed as `np.linalg.norm(x)/np.sqrt(len(x))`. Let's define a vector (which represents a signal, *i.e.* the value of some quantity at uniformly space time instances), and find its RMS value.

In [ ]:
```
rms = lambda x: (sum(x**2)**0.5)/(len(x)**0.5)
t = np.arange(0,1.01,0.01)
x = np.cos(8*t) - 2*np.sin(11*t)
print(sum(x)/len(x))
```

```
-0.04252943783238685
```

In [ ]:
```
print(rms(x))
```

```
1.0837556422598
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
plt.plot(t,x)
plt.plot(t, np.mean(x)*np.ones(len(x)))
plt.plot(t, (np.mean(x) + rms(x))*np.ones(len(x)), 'g')
plt.plot(t, (np.mean(x) - rms(x))*np.ones(len(x)), 'g')
plt.show()
```

The above code plots the signal, its average value, and two constant signals at $\mathbf{avg}(x) \pm \mathbf{rms}(x)$ (Figure 3.1).

Chebyshev inequality. The Chebyshev inequality states that the number of entries of an n-vector x that have absolute value at least a is no more than $\|x\|^2/a^2 = n\mathbf{rms}(x)^2/a^2$. If the number is, say, 12.15, we can conclude that no more than 12 entries have absolute value at least a, since the number of entries is an integer. So the Chebyshev bound can be improved to be $floor(\|x\|^2/a)$, where $floor(u)$ is the integer part of a positive number. Let's define a function with the Chebyshev bound, including the floor function improvement, and apply the bound to the signal found above, for a specific value of a.

In [ ]:
```
# Define Chebyshev bound function
import math
cheb_bound = lambda x,a: math.floor(sum(x**2)/a)
a = 1.5
print(cheb_bound(x,a))
```

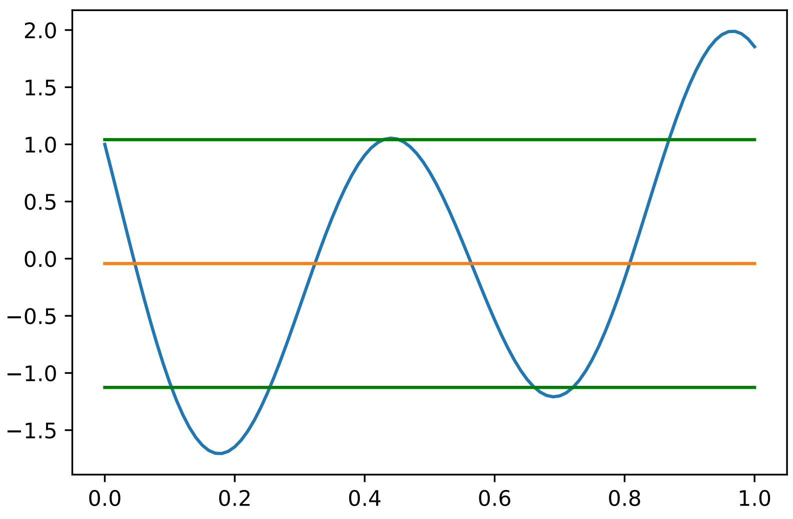


Figure 3.1.: A signal x. The horizontal lines show $\mathbf{avg}(x) + \mathbf{rms}(x)$, $\mathbf{avg}(x)$, and $\mathbf{avg}(x) - \mathbf{rms}(x)$.

```
79
```

In [ ]:
```
# Number of entries of x with |x_i| >= a
print(sum(abs(x) >= a))
```

```
20
```

In the last line, the expression `abs(x) >= a` creates an array with entries that are Boolean, *i.e.*, `true` or `false`, depending on whether the corresponding entry of `x` satisfies the inequality. When we sum the vector of Boolean, they are automatically converted to the numbers 1 and 0, respectively.

3.2. Distance

Distance. The distance between two vectors is $\mathbf{dist}(x, y) = \|x - y\|$. This is written in Python as `np.linalg.norm(x-y)`. Let's find the distance between the pairs of the three vectors u, v, and w from page 49 of VMLS.

In [ ]:
```
u = np.array([1.8, 2.0, -3.7, 4.7])
v = np.array([0.6, 2.1, 1.9, -1.4])
w = np.array([2.0, 1.9, -4.0, 4.6])
```

```
print(np.linalg.norm(u-v))
print(np.linalg.norm(u-w))
print(np.linalg.norm(v-w))
```

```
8.367795408588812
0.3872983346207417
8.532877591996735
```

We can see that u and w are much closer to each other than u and v, or v and w. Other expressions such as `np.sqrt(sum((a-b)**2))` and `sum((a-b)**2)**0.5` also give the distance between vector a and b.

Nearest neighbor. We define a function that calculates the nearest neighbour of a vector in a list of vectors, and try it on the points in Figure 3.3 of VMLS.

```
In [ ]: near_neigh = lambda x,z: z [np.argmin([np.linalg.norm(x-y) for y
        ↪  in z])]
        z = ([2,1], [7,2], [5.5,4], [4,8], [1,5], [9,6])
        x = np.array([5,6])
        print(near_neigh(x,z))
```

```
[5.5, 4]
```

```
In [ ]: print(near_neigh(np.array([3,3]),z))
```

```
[2, 1]
```

On the first line, the expression `[np.linalg.norm(x-y)for y in z]` uses a convenient construction in Python. Here `z` is a list of vectors, and the expression expands to an array with elements `np.linalg.norm(x-z[0])`, `np.linalg.norm(x-z[1])`,... The numpy function `np.argmin` applied to this array returns the index of the smallest element.

De-meaning a vector. We refer to the vector $x - \textbf{avg}(x)\mathbf{1}$ as the de-meaned version of x.

```
In [ ]: de_mean = lambda x: x - sum(x)/len(x)
        x = np.array([1,-2.2,3])
        print ('Average of x: ', np.mean(x))
        x_tilde = de_mean(x)
        print('x_tilde: ',x_tilde)
        print('Average of x_tilde: ',np.mean(x_tilde))
```

```
Average of x:  0.6
x_tilde:  [ 0.4 -2.8  2.4]
Average of x_tilde:  -1.4802973661668753e-16
```

(The mean of $\tilde{x}$ is very very close to zero.)

3.3. Standard deviation

Standard deviation. We can define a function that corresponding to the VMLS definition of the standard deviation of a vector, $\mathbf{std}(x) = \|x - \mathbf{avg}(x)\mathbf{1}\|/\sqrt{n}$, where n is the length of the vector.

```
In [ ]: x = np.random.random(100)
        stdev = lambda x: np.linalg.norm(x - sum(x)/len(x))/(len(x)**0.5)
        stdev(x)
```

```
Out[ ]: 0.30440692170248823
```

You can also use the `numpy` function `np.std(x)` to obtain the standard deviation of a vector.

Return and risk. We evaluate the mean return and risk (measured by standard deviation) of the four time series Figure 3.4 of VMLS.

```
In [ ]: a = np.ones(10)
        np.mean(a), np.std(a)
```

```
Out[ ]: (1.0, 0.0)
```

```
In [ ]: b = [ 5, 1, -2, 3, 6, 3, -1, 3, 4, 1 ]
        np.mean(b), np.std(b)
```

```
Out[ ]: (2.3, 2.4103941586387903)
```

```
In [ ]: c = [ 5, 7, -2, 2, -3, 1, -1, 2, 7, 8 ]
        np.mean(c), np.std(c)
```

```
Out[ ]: (2.6, 3.7735924528226414)
```

```
In [ ]: d = [ -1, -3, -4, -3, 7, -1, 0, 3, 9, 5 ]
        np.mean(d), np.std(d)
```

```
Out[ ]: (1.2, 4.308131845707604)
```

Standardizing a vector. If a vector x isn't constant (*i.e.*, at least two of its entries are different), we can standardize it, by subtracting its mean and dividing by its standard deviation. The resulting standardized vector has mean value zero and RMS value one. Its entries are called z-scores. We'll define a `standardize` function, and then check it with a random vector.

```
In [ ]: def standardize(x):
            x_tilde = x - np.mean(x)
            return x_tilde/np.std(x_tilde)
        x = np.random.random(100)
        np.mean(x), np.std(x)
```

```
Out[ ]: (0.568533078290501, 0.282467953801772)
```

```
In [ ]: z = standardize(x)
        np.mean(z), np.std(z)
```

```
Out[ ]: (1.3322676295501878e-17, 1.0)
```

The mean or average value of the standardized vector `z` is very nearly zero.

3.4. Angle

Angle. Let's define a function that computes the angle between two vectors. We will call it `ang`.

```
In [ ]: #Define angle function, which returns radians
        ang = lambda x,y : np.arccos(x @ y /
        ↪   (np.linalg.norm(x)*np.linalg.norm(y)))
        a = np.array([1,2,-1])
        b = np.array([2,0,-3])
        ang(a,b)
```

```
Out[ ]: 0.9689825515916383
```

```
In [ ]: #Get angle in degrees
        ang(a,b) * (360/(2*np.pi))
```

```
Out[ ]: 55.51861062801842
```

Correlation coefficient. The correlation coefficient between two vectors a and b (with nonzero standard deviation) is defined as

$$\rho = \frac{\tilde{a}^T\tilde{b}}{\|\tilde{a}\|\|\tilde{b}\|},$$

where $\tilde{a}$ and $\tilde{b}$ are the de-meaned versions of a and b, respectively. We can define our own function to compute the correlation. We calculate the correlation coefficients of the three pairs of vectors in Figure 3.8 in VMLS.

```
In [ ]: def corr_coef(a,b):
            a_tilde = a - sum(a)/len(a)
            b_tilde = b - sum(b)/len(b)
            denom =  (np.linalg.norm(a_tilde) * np.linalg.norm(b_tilde))
            return (a_tilde @ b_tilde) /denom
        a = np.array([4.4, 9.4, 15.4, 12.4, 10.4, 1.4, -4.6, -5.6, -0.6,
        ↪  7.4])
        b = np.array([6.2, 11.2, 14.2, 14.2, 8.2, 2.2, -3.8, -4.8, -1.8,
        ↪  4.2])
        corr_coef(a,b)
```

```
Out[ ]: 0.9678196342570432
```

```
In [ ]: a = np.array([4.1, 10.1, 15.1, 13.1, 7.1, 2.1, -2.9, -5.9, 0.1,
        ↪  7.1])
        b = np.array([5.5, -0.5, -4.5, -3.5, 1.5, 7.5, 13.5, 14.5, 11.5,
        ↪  4.5])
        corr_coef(a,b)
```

```
Out[ ]: -0.9875211120643734
```

```
In [ ]: a = np.array([-5.0, 0.0, 5.0, 8.0, 13.0, 11.0, 1.0, 6.0, 4.0,
        ↪  7.0])
        b = np.array([5.8, 0.8, 7.8, 9.8, 0.8, 11.8, 10.8, 5.8, -0.2,
        ↪  -3.2])
        corr_coef(a,b)
```

```
Out[ ]: 0.004020976661367021
```

The correlation coefficients of the three pairs of vectors are 96.8%, -98.8%, and 0.4%.

3.5. Complexity

Let's check that the time to compute the correlation coefficient of two n-vectors is approximately linear in n.

```
In [ ]: import time
        x = np.random.random(10**6)
        y = np.random.random(10**6)
        start = time.time()
        corr_coef(x,y)
        end = time.time()
        end - start
```

```
Out[ ]: 0.16412591934204102
```

```
In [ ]: x = np.random.random(10**7)
        y = np.random.random(10**7)
        start = time.time()
        corr_coef(x,y)
        end = time.time()
        end - start
```

```
Out[ ]: 1.6333978176116943
```

```
In [ ]: x = np.random.random(10**8)
        y = np.random.random(10**8)
        start = time.time()
        corr_coef(x,y)
        end = time.time()
        end - start
```

```
Out[ ]: 16.22579288482666
```

4. Clustering

4.1. Clustering

4.2. A clustering objective

In Python, we can store the list of vectors in a `numpy` list of N vectors. If we call this list `data`, we can access the ith entry (which is a vector) using `data[0]`. To specify the clusters or group membership, we can use a list of assignments called `grouping`, where `grouping[i]` is the number of group that vector `data[i]` is assigned to. (This is an integer between 1 and k.) (In VMLS, chapter 4, we describe the assignments using a vector c or the subsets G_j.) We can store k cluster representatives as a Python list called `centroids`, with `centroids[j]` the jth cluster representative. (In VMLS we describe the representatives as the vectors $z_1, \ldots, z_k$.)

Group assignment. We define a function to perform group assignment. With given initial value of centorids, we compute the distance between each centroid with each vector and assign the grouping according to the smallest distance. The function then returns a vector of groupings.

In [ ]:
```
def group_assignment(data,centroids):
    grouping_vec_c = np.zeros(len(data))
    for i in range(len(data)):
        dist = np.zeros(len(centroids))
        for j in range(len(centroids)):
            dist[j] = np.linalg.norm(data[i] - centroids[j])
        min_dist = min(dist)
        for j in range(len(centroids)):
            if min_dist == dist[j]:
                grouping_vec_c[i] = j+1
    return grouping_vec_c
```

Update centroid. We define a function to update the centroid after the group assignment, returning a new list of group centroids.

```
In [ ]: def update_centroid(data, grouping, centroids):
            new_centroids = [];
            for i in range(len(centroids)):
                cent = np.zeros(len(data[0]))
                count = 0
                for j in range(len(data)):
                    if grouping[j] == (i+1):
                        cent = cent+data[j]
                        count += 1
                group_average = cent/count
                new_centroids.append(group_average)
            return new_centroids
```

Clustering objective. Given the group assignment and the centroids with the data, we can compute the clustering objective as the square of the RMS value of the vector of distances.

```
In [ ]: def clustering_objective(data, grouping, centroids):
            J_obj = 0
            for i in range(len(data)):
                for j in range(len(centroids)):
                    if grouping[i] == (j+1):
                        J_obj += np.linalg.norm(data[i] - centroids[j])**2
            J_obj = J_obj/len(data)
            return J_obj
```

4.3. The k-means algorithm

We can define another function `Kmeans_alg` that uses the three functions defined in the above subsection iteratively.

```
In [ ]: def Kmeans_alg(data, centroids):
            iteration = 0
            J_obj_vector = []
            Stop = False
            while Stop == False:
                grouping = group_assignment(data, centroids)
```

```
        new_centroids = update_centroid(data, grouping, centroids)
        J_obj = clustering_objective(data, grouping,
        ↪  new_centroids)
        J_obj_vector.append(J_obj)
        iteration += 1
        if np.linalg.norm(np.array(new_centroids) -
        ↪  np.array(centroids)) < 1e-6:
            Stop = True
        else:
            centroids = new_centroids
    return new_centroids, grouping, J_obj_vector, iteration
```

Convergence. Here we use a `while` loop, which executes the statements inside the loop as long as the condition `Stop == False` is true. We terminate the algorithm when the improvement in the clustering objective becomes very small (`1e-6`).

Alternatively, we can use the `Kmeans` function in the `cluster` module of the `sklearn` package.

```
In [ ]: from sklearn.cluster import KMeans
        import numpy as np
        kmeans = KMeans(n_clusters=4, random_state=0).fit(data)
        labels = kmeans.labels_
        group_representative = kmeans.cluster_centers_
        J_clust = kmeans.inertia_
```

Here we try to apply the k-means algorithm on `data`, clustering the vectors into 4 groups. Note that the `sklearn.cluster.KMeans` function initialize the algorithms with random centroids and thus the initial values of centroids are not required as an argument but the random state to draw the random initialization is.

4.4. Examples

We apply the algorithm on a randomly generated set of $N = 300$ points, shown in Figure 4.1. These points were generated as follows.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
```

```
X = np.concatenate([[0.3*np.random.randn(2) for i in range(100)],
↪  [[1,1] + 0.3*np.random.randn(2) for i in range(100)], [[1,-1]
↪  + 0.3* np.random.randn(2) for i in range(100)]])
plt.scatter( X[:,0],X[:,1])
plt.xlim(-1.5,2.5)
plt.ylim(-2,2)
plt.show()
```

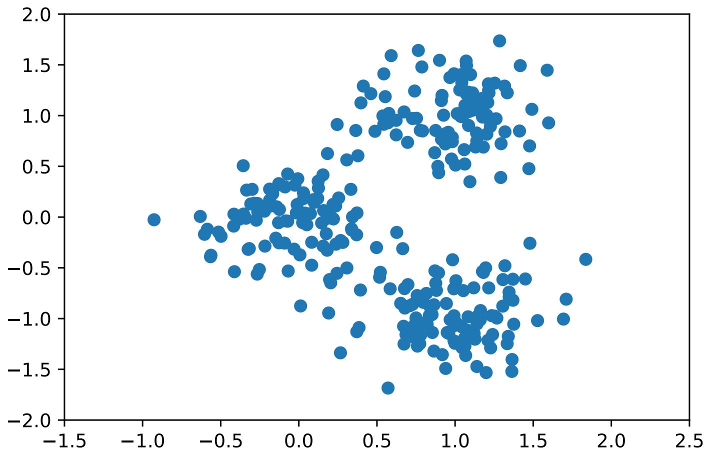


Figure 4.1.: 300 points in a plane.

On the first line, we import the `matplotlib` package for plotting. Then we generate three arrays of vectors. Each set consists of 100 vectors chosen randomly around one of the three points (0,0),(1,1), and (1,-1). The three arrays are concatenated using `np.concatenate()` to get an array of 300 points. Next, we apply the `KMeans` function and make a figure with the three clusters (Figure 4.2).

```
In [ ]: from sklearn.cluster import KMeans
        import numpy as np
        kmeans = KMeans(n_clusters=3, random_state=0).fit(X)
        labels = kmeans.labels_
        group_representative = kmeans.cluster_centers_
        J_clust = kmeans.inertia_

        grps = [[X[i,:] for i in range(300) if labels[i]==j] for j in
        ↪  range(3)]
```

```
plt.scatter([c[0] for c in grps[0]],[c[1] for c in grps[0]])
plt.scatter([c[0] for c in grps[1]],[c[1] for c in grps[1]])
plt.scatter([c[0] for c in grps[2]],[c[1] for c in grps[2]])
plt.xlim(-1.5,2.5)
plt.ylim(-2,2)
plt.show()
```

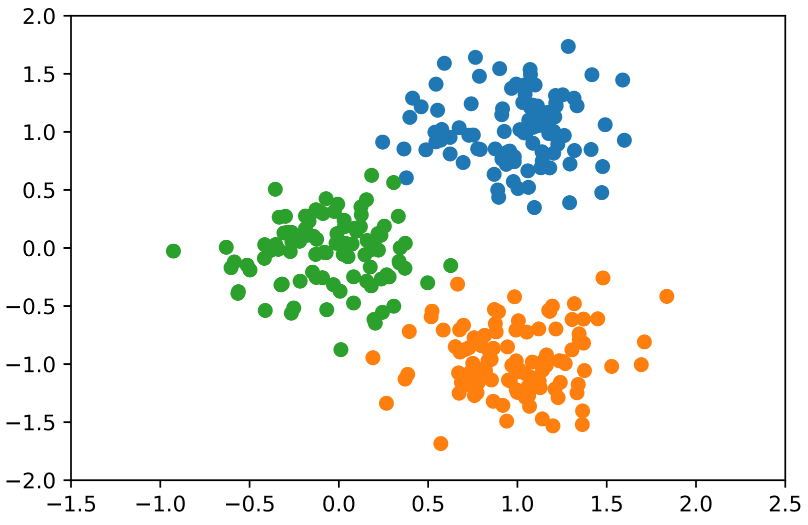


Figure 4.2.: Final clustering.

4.5. Applications

5. Linear independence

5.1. Linear independence

5.2. Basis

Cash flow replication. Let's consider cash flows over 3 periods, given by 3-vectors. We know from VMLS page 93 that the vectors

$$e_i = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad l_1 = \begin{bmatrix} 1 \\ -(1+r) \\ 0 \end{bmatrix}, \quad l_2 = \begin{bmatrix} 0 \\ 1 \\ -(1+r) \end{bmatrix}$$

form a basis, where r is the (positive) per-period interest rate. The first vector e_1 is a single payment of 1 in period (time) $t = 1$. The second vector l_1 is loan of \$ 1 in period $t = 1$, paid back in period $t = 1$ with interest r. The third vector l_2 is loan of \$ 1 in period $t = 2$, paid back in period $t = 3$ with interest r. Let's use this basis to replicate the cash flow $c = (1, 2, -3)$ as

$$c = \alpha_1 e_1 + \alpha_2 l_1 + \alpha_3 l_2 = \alpha_1 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ -(1+r) \\ 0 \end{bmatrix} + \alpha_3 \begin{bmatrix} 0 \\ 1 \\ -(1+r) \end{bmatrix}.$$

From the third component we have $c_3 = \alpha_3(-(1+r))$, so $\alpha_3 = -c_3/(1+r)$. From the second component we have

$$c_2 = \alpha_2(-(1+r)) + \alpha_3 = \alpha_2(-(1+r)) - c_3/(1+r),$$

so $\alpha_2 = -c_2/(1+r) - c_3/(1+r)^2$. Finally, from $c_1 = \alpha_1 + \alpha_2$, we have

$$\alpha_1 = c_1 + c_2/(1+r) + c_3/(1+r)^2,$$

which is the net present value (NPV) of the cash flow c.
Let's check this in Python using an interest rate of 5% per period, and the specific cash flow $c = (1, 2, -3)$.

```
In [ ]: import numpy as np
        r = 0.05
        e1 = np.array([1,0,0])
        l1 = np.array([1, -(1+r), 0])
        l2 = np.array([0,1,-(1+r)])
        c = np.array([1,2,-3])
        # Coefficients of expansion
        alpha3 = -c[2]/(1+r)
        alpha2 = -c[1]/(1+r) - c[2]/((1+r)**2)
        alpha1 = c[0] + c[1]/(1+r) + c[2]/((1+r)**2) #NPV of cash flow
        print(alpha1)
```

```
0.18367346938775508
```

```
In [ ]: print(alpha1*e1 + alpha2*l1 + alpha3*l2)
```

```
[ 1.,  2., -3.]
```

(Later in the course we'll introduce an automated and simple way to find the coefficients in the expansion of a vector in a basis.)

5.3. Orthonormal vectors

Expansion in an orthonormal basis. Let's check that the vectors

$$a_1 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \quad a_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad a_3 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix},$$

form an orthonormal basis, and check the expansion of $x = (1, 2, 3)$ in this basis,

$$x = (a_1^T x)a_1 + \ldots + (a_n^T x)a_n.$$

```
In [ ]: a1 = np.array([0,0,-1])
        a2 = np.array([1,1,0])/(2)**0.5
        a3 = np.array([1,-1,0])/(2)**0.5
        print('Norm of a1 :', (sum(a1**2))**0.5)
print('Norm of a2 :', (sum(a2**2))**0.5)
print('Norm of a3 :', (sum(a3**2))**0.5)
```

```
Norm of a1 : 1.0
Norm of a2 : 0.9999999999999999
Norm of a3 : 0.9999999999999999
```

```
In [ ]: print(a1 @ a2, a1 @ a3, a2 @ a3)
```

```
0.0, 0.0, 0.0
```

```
In [ ]: x = np.array([1,2,3])
        #Get coefficients of x in orthonormal basis
        beta1 = a1 @ x
        beta2 = a2 @ x
        beta3 = a3 @ x
        #Expansion of x in basis
        x_expansion = beta1*a1 + beta2*a2 + beta3*a3
        print(x_expansion)
```

```
[1. 2. 3.]
```

5.4. Gram-Schmidt algorithm

The following is a Python implementation of the Gram-Schmidt algorithm. It takes as input an array a which expands into [a[0], a[1], ..., a[k]], containing k vectors $a_1, \dots, a_k$. If the vectors are linearly independent, it returns an array q with orthonormal set of vectors computed by the Gram-Schmidt algorithm. If the vectors are linearly dependent and the Gram-Schmidt algorithm terminates early in the $i+1$th iteration, it returns the array q[0], ..., q[i-1] of length i.

```
In [ ]: import numpy as np
        def gram_schmidt(a):
            q = []
            for i in range(len(a)):
                #orthogonalization
                q_tilde = a[i]
                for j in range(len(q)):
                    q_tilde = q_tilde -  (q[j] @ a[i])*q[j]
                #Test for dependennce
        if np.sqrt(sum(q_tilde**2)) <= 1e-10:
            print('Vectors are linearly dependent.')
            print('GS algorithm terminates at iteration ', i+1)
            return q
        #Normalization
        else:
            q_tilde = q_tilde / np.sqrt(sum(q_tilde**2))
            q.append(q_tilde)
    print('Vectors are linearly independent.')
    return q
```

Here we initialize the output array as an empty array. In each iteration, we add the next vector to the array using the `append()` method.

Example. We apply the function to the example on page 100 of VMLS.

```
In [ ]: a = np.array([ [-1, 1, -1, 1], [-1, 3, -1, 3], [1, 3, 5, 7] ])
        q = gram_schmidt(a)
        print(q)
        #Test orthonormality
        print('Norm of q[0] :', (sum(q[0]**2))**0.5)
        print('Inner product of q[0] and q[1] :', q[0] @ q[1])
        print('Inner product of q[0] and q[2] :', q[0] @ q[2])
        print('Norm of q[1] :', (sum(q[1]**2))**0.5)
        print('Inner product of q[1] and q[2] :', q[1] @ q[2])
        print('Norm of q[2] :', (sum(q[2]**2))**0.5)
```

```
Vectors are linearly independent.
[array([-0.5,  0.5, -0.5,  0.5]), array([0.5, 0.5, 0.5, 0.5]),
↪  array([-0.5, -0.5,  0.5,  0.5])]
Norm of q[0] : 1.0
Inner product of q[0] and q[1] : 0.0
Inner product of q[0] and q[2] : 0.0
Norm of q[1] : 1.0
Inner product of q[1] and q[2] : 0.0
Norm of q[2] : 1.0
```

Example of early termination. If we replace a_3 with a linear combination of a_1 and a_2, the set becomes linearly dependent.

In [ ]:
```
b = np.array([a[0],a[1], 1.3*a[0] + 0.5*a[1]])
q = gram_schmidt(b)
print(q)
```

```
Vectors are linearly dependent.
GS algorithm terminates at iteration  3
[array([-0.5,  0.5, -0.5,  0.5]), array([0.5, 0.5, 0.5, 0.5])]
```

Here the algorithm terminated at iteration 3. That means, the third iteration is not completed and thus two orthonormal vectors are returned.

Example of independence-dimension inequality. We know that any three 2-vectors must be dependent. Let's use the Gram-Schmidt algorithm to verify this for three specific vectors.

In [ ]:
```
three_two_vectors = np.array([[1,1],[1,2],[-1,1]])
q = gram_schmidt(three_two_vectors)
print(q)
```

```
Vectors are linearly dependent.
GS algorithm terminates at iteration  3
[array([0.70710678, 0.70710678]), array([-0.70710678,
↪  0.70710678])]
```

Part II.

Matrices

6. Matrices

6.1. Matrices

Creating matrices from the entries. Matrices can be represented in Python using 2-dimensional numpy array or list structure (list of lists). For example, the 3×4 matrix

$$A = \begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & 4 & -0.1 & 1 \\ 4.1 & -1 & 0 & 1.7 \end{bmatrix}$$

is constructed in Python as

```
In [ ]: A = np.array([[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0,
         ↪ 1.7]])
        A.shape
```

```
Out[ ]: (3, 4)
```

```
In [ ]: m,n = A.shape
        print('m:', m)
        print('n:', n)
```

```
m: 3
n: 4
```

Here we use numpy `shape` property to get the current shape of an array. As an example, we create a function to check if a matrix is tall.

```
In [ ]: tall = lambda X: A.shape[0] > A.shape[1]
        tall(A)
```

```
Out[ ]: False
```

In the function definition, the number of rows and the number of columns are combined using the relational operator `>`, which gives a Boolean.

If matrix `A` is expressed as a list of lists, we can get the dimensions of the array using the `len()` function as follows.

```
In [ ]: A = [[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0, 1.7]]
        print('m:', len(A))
        print('n:', len(A[0]))
```

```
m: 3
n: 4
```

Indexing entries. We get the ith row jth column entry of a matrix `A` using `A[i-1, j-1]` since Python starts indexing from 0.

```
In [ ]: A[0,2] #Get the third element in the first row
```

```
Out[ ]: -2.3
```

We can also assign a new value to a matrix entry using similar bracket notation.

```
In [ ]: A[0,2] = 7.5 #Set the third element in the first row of A to 7.5
        A
```

```
Out[ ]: array([[ 0. ,  1. ,  7.5,  0.1],
               [ 1.3,  4. , -0.1,  0. ],
               [ 4.1, -1. ,  0. ,  1.7]])
```

Equality of matrices. `A == B` determines whether the matrices `A` and `B` are equal. For numpy arrays `A` and `B`, the expression `A == B` creates a matrix whose entries are Boolean, depending on whether the corresponding entries of `A` and `B` are the same. The expression `sum(A == B)` gives the number of entries of `A` and `B` that are equal.

```
In [ ]: A = np.array([[0,1,-2.3,0.1], [1.3, 4, -0.1, 0], [4.1, -1.0, 0,
        ↪  1.7]])
        B = A.copy()
        A == B
```

```
Out[ ]: array([[ True,  True,  True,  True],
               [ True,  True,  True,  True],
               [ True,  True,  True,  True]])
```

```
In [ ]: B[0,3] = 100  #re-assign one of the elements in B
        np.sum(A==B)
```

```
Out[ ]: 11
```

Alternatively, if `A` and `B` are list expressions (list of lists), the expression `A == B` gives a Boolean value that indicates whether the matrices are equivalent.

```
In [ ]: A = [[0,1,-2.3,0.1],[1.3, 4, -0.1, 0],[4.1, -1.0, 0, 1.7]]
        B = [[0,1,-2.3,100],[1.3, 4, -0.1, 0],[4.1, -1.0, 0, 1.7]]
        A == B
```

```
Out[ ]: False
```

Row and column vectors. In VMLS, n-vectors are the same as $n \times 1$ matrices. In Python, there is a subtle difference between a 1D array, a column vector and a row vector.

```
In [ ]: # a 3 vector
        a = np.array([-2.1, -3, 0])
        a.shape
```

```
Out[ ]: (3,)
```

```
In [ ]: # a 3-vector or a 3 by 1 matrix
        b = np.array([[-2.1],[-3],[0]])
        b.shape
```

```
Out[ ]: (3, 1)
```

```
In [ ]: # a 3-row vector or a 1 by 3 matrix
        c = np.array([[-2.1, -3, 0]])
        c.shape
```

```
Out[ ]: (1, 3)
```

You can see `a` has shape `(3,)`, meaning that it is a 1D array, while `b` has shape `(3, 1)`, meaning that it is a 3×1 matrix. This small subtlety generally won't affect you and `b` can be reshaped to look like `c` easily with the command `b.reshape(3)`.

Slicing and submatrices. Using colon notation you can extract a submatrix.

```
In [ ]: A = np.array([[-1, 0, 1, 0],[2, -3, 0, 1],[0, 4, -2, 1]])
        A[0:2, 2:4]
```

```
Out[ ]: array([[1, 0], [0, 1]])
```

This is different from the mathematical notation in VMLS due to the fact that Python starts index at 0. You can also assign a submatrix using slicing (index range) notation. A very useful shortcut is the index range : which refers to the whole index range for that index. This can be used to extract the rows and columns of a matrix.

```
In [ ]: # Third column of A
        A[:,2]
```

```
Out[ ]: array([ 1,  0, -2])
```

```
In [ ]: # Second row of A, returned as vector
        A[1,:]
```

```
Out[ ]: array([ 2, -3,  0,  1])
```

The numpy function `reshape()` gives a new $k \times l$ matrix. (We must have $mn = kl$, *i.e.*, the original and reshaped matrix must have the same number of entries. This is not standard mathematical notation, but they can be useful in Python.

```
In [ ]: A.reshape((6,2))
```

```
Out[ ]: array([[-1,  0],
               [ 1,  0],
               [ 2, -3],
               [ 0,  1],
               [ 0,  4],
               [-2,  1]])
```

```
In [ ]: A.reshape((3,2))
```

```
Out[ ]: ValueError: cannot reshape array of size 12 into shape (3,2)
```

Block matrices. Block matrices are constructed in Python very much as in the standard mathematical notation in VMLS. We can use the `np.block()` method to construct block matrices in Python.

```
In [ ]: B = np.array([0,2,3])            #1 by 3 matrix
        C = np.array([-1])               #1 by 1 matrix
        D = np.array([[2,2,1],[1,3,5]])  #2 by 3 matrix
        E = np.array([[4],[4]])           #2 by 1 matrix
        # Constrcut 3 by 4 block matrix
```

```
A = np.block([[B,C],[D,E]])
A
```

Out[ ]:
```
array([[ 0,  2,  3, -1],
       [ 2,  2,  1,  4],
       [ 1,  3,  5,  4]])
```

Column and row interpretation of a matrix. An $m \times n$ matrix A can be interpreted as a collection of n m-vectors (its columns) or a collection of m n-row-vectors (its rows). Column vectors can be converted into a matrix using the horizontal function `np.hstack()` and row vectors can be converted into a matrix using the vertical function `np.vstack()`.

In [ ]:
```
a = [1,2]
b = [4,5]
c = [7,8]
A = np.vstack([a,b,c])
B = np.hstack([a,b,c])
print('matrix A :',A)
print('dimensions of A :', A.shape)
print('matrix B :',B)
print('dimensions of B :', B.shape)
```

```
matrix A : [[1 2]
 [4 5]
 [7 8]]
dimensions of A : (3, 2)
matrix B : [1 2 4 5 7 8]
dimensions of B : (6,)
```

In [ ]:
```
a = [[1],[2]]
b = [[4],[5]]
c = [[7],[8]]
A = np.vstack([a,b,c])
B = np.hstack([a,b,c])
print('matrix A :',A)
print('dimensions of A :', A.shape)
print('matrix B :',B)
print('dimensions of B :', B.shape)
```

```
matrix A : [[1]
 [2]
 [4]
 [5]
 [7]
 [8]]
dimensions of A : (6, 1)
matrix B : [[1 4 7]
 [2 5 8]]
dimensions of B : (2, 3)
```

Another useful expression is `np.c_` and `np.r_`, which allow us to stack column (and row) vectors with other vectors (and matrices).

```
In [ ]: np.c_[-np.identity(3), np.zeros(3)]
```

```
Out[ ]: array([[-1., -0., -0.,  0.],
               [-0., -1., -0.,  0.],
               [-0., -0., -1.,  0.]])
```

```
In [ ]: np.r_[np.identity(3), np.zeros((1,3))]
```

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

6.2. Zero and identity matrices

Zero matrices. A zero matrix of size $m \times n$ is created using `np.zeros((m,n))`. Note the double brackets!

```
In [ ]: np.zeros((2,2))
```

```
Out[ ]: array([[0., 0.],
               [0., 0.]])
```

Identity matrices. Identity matrices in Python can be created in many ways, for example, by starting with a zero matrix and then setting the diagonal entries to one. You can also use the `np.identity(n)` function to create an identity matrix of dimension n.

```
In [ ]: np.identity(4)
```

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

Ones matrix. In VMLS we do not use special notation for a matrix with all entries equal to one. In Python, such a matrix is given by `np.ones((m,n))`

Diagonal matrices. In standard mathematical notation, $\textbf{diag}(1, 2, 3)$ is a diagonal 3×3 matrix with diagonal entries $1, 2, 3$. In Python, such a matrix can be created using `np.diag()`.

```
In [ ]: x = np.array([[0, 1, 2],
                      [3, 4, 5],
                      [6, 7, 8]])
        print(np.diag(x))
```

```
[0 4 8]
```

```
In [ ]: print(np.diag(np.diag(x)))
```

```
[[0 0 0]
 [0 4 0]
 [0 0 8]]
```

Here we can see that, when we apply the `np.diag()` function to a matrix, it extracts the diagonal elements as a vector (array). When we apply the `np.diag()` function to a vector (array), it constructs a diagonal matrix with diagonal entries given in the vector.

Random matrices. A random $m\times n$ matrix with entries distributed uniformly between 0 and 1 is created using `np.random.random((m,n))`. For entries that have a standard normal distribution, we can use `np.random.normal((m,n))`.

```
In [ ]: np.random.random((2,3))
```

```
Out[ ]: array([[0.13371842, 0.28868153, 0.6146294 ],
               [0.81106752, 0.10340807, 0.02324088]])
```

```
In [ ]: np.random.randn(3,2)
```

Out[ ]:
```
array([[ 1.35975208, -1.63292901],
       [-0.47912321, -0.4485537 ],
       [-1.1693047 , -0.05600474]])
```

Sparse matrices. Functions for creating and manipulating sparse matrices are contained in the `scipy.sparse` module, which must be installed and imported. Sparse matrices are stored in a special format that exploits the property that most of the elements are zero. The `scipy.sparse.coo_matrix()` function create a sparse matrix from three arrays that specify the row indexes, column indexes, and values of the nonzero elements. The following code creates a sparse matrix

$$A = \begin{bmatrix} -1.11 & 0 & 1.17 & 0 & 0 \\ 0.15 & -0.10 & 0 & 0 & 0 \\ 0 & 0 & -0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.13 & 0 \end{bmatrix}$$

In [ ]:
```
from scipy import sparse
I = np.array([ 0, 1, 1, 0, 2, 3 ]) # row indexes of nonzeros
J = np.array([ 0, 0, 1, 2, 2, 3 ]) # column indexes
V = np.array([ -1.11, 0.15, -0.10, 1.17, -0.30, 0.13 ]) # values
A = sparse.coo_matrix((V,(I,J)), shape=(4,5))
A
```

Out[ ]:
```
<4x5 sparse matrix of type '<class 'numpy.float64'>'
  with 6 stored elements in COOrdinate format>
```

In [ ]:
```
A.nnz
```

Out[ ]:
```
6
```

Sparse matrices can be converted to regular non-sparse matrices using the `todense()` method.

In [ ]:
```
A.todense()
```

Out[ ]:
```
matrix([[-1.11,  0.  ,  1.17,  0.  ,  0.  ],
        [ 0.15, -0.1 ,  0.  ,  0.  ,  0.  ],
        [ 0.  ,  0.  , -0.3 ,  0.  ,  0.  ],
        [ 0.  ,  0.  ,  0.  ,  0.13,  0.  ]])
```

A sparse $m \times n$ zero matrix is created with `sparse.coo_matrix((m, n))`. To create a sparse $n \times n$ identity matrix in Python, use `sparse.eye(n)`. We can also create a sparse diagonal matrix (with different offsets) using

```
In [ ]: diagonals = [[1, 2, 3, 4], [1, 2, 3], [1, 2]]
        B = sparse.diags(diagonals, offsets=[0,-1,2])
        B.todense()
```

```
Out[ ]: matrix([[1., 0., 1., 0.],
                [1., 2., 0., 2.],
                [0., 2., 3., 0.],
                [0., 0., 3., 4.]])
```

A useful function for creating a random sparse matrix is `sparse.rand()`. It generates a sparse matrix of a given shape and density with uniformly distributed values.

```
In [ ]: matrix = sparse.rand(3, 4, density=0.25, format='csr',
        ↪  random_state=42)
        matrix
```

```
Out[ ]: <3x4 sparse matrix of type '<class 'numpy.float64'>'
          with 3 stored elements in Compressed Sparse Row format>
```

6.3. Transpose, addition, and norm

Transpose. In VMLS we denote the transpose of an $m \times n$ matrix A as A^T. In Python, the transpose of `A` is given by `np.transpose(A)` or simply `A.T`.

```
In [ ]: H = np.array([[0,1,-2,1], [2,-1,3,0]])
        H.T
```

```
Out[ ]: array([[ 0,  2],
               [ 1, -1],
               [-2,  3],
               [ 1,  0]])
```

```
In [ ]: np.transpose(H)
```

```
Out[ ]: array([[ 0,  2],
               [ 1, -1],
               [-2,  3],
               [ 1,  0]])
```

Addition, subtraction, and scalar multiplication. In Python, addition and subtraction of matrices, and scalar-matrix multiplication, both follow standard and mathematical notation.

```
In [ ]: U = np.array([[0,4], [7,0], [3,1]])
        V = np.array([[1,2], [2,3], [0,4]])
        U + V
```

```
Out[ ]: array([[1, 6],
               [9, 3],
               [3, 5]])
```

```
In [ ]: 2.2*U
```

```
Out[ ]: array([[ 0. ,  8.8],
               [15.4,  0. ],
               [ 6.6,  2.2]])
```

(We can also multiply a matrix on the right by a scalar.) Python supports some operations that are not standard mathematical ones. For example, in Python you can add or subtract a constant from a matrix, which carries out the operation on each entry.

Elementwise operations. The syntax for elementwise vector operations described on page 11 carries over naturally to matrices.

Matrix norm. In VMLS we use $\|A\|$ to denote the norm of an $m \times n$ matrix,

$$\|A\| = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2 \right)^{\frac{1}{2}}$$

In standard mathematical notation, this is more often written as $\|A\|_F$, where F stands for the name Frobenius. In standard mathematical notation, $\|A\|$ usually refers to another norm of a matrix, which is beyond the scope of topics in VMLS. In Python, `np.linalg.norm(A)` gives the norm used in VMLS.

```
In [ ]: A = np.array([[2,3,-1], [0,-1,4]])
        np.linalg.norm(A)
```

```
Out[ ]: 5.5677643628300215
```

```
In [ ]: np.linalg.norm(A[:])
```

```
Out[ ]: 5.5677643628300215
```

Triangle inequality. Let's check that the triangle inequality $\|A + B\| \leq \|A\| + \|B\|$ holds, for two specific matrices.

```
In [ ]: A = np.array([[-1,0], [2,2]])
        B = np.array([[3,1], [-3,2]])
        print(np.linalg.norm(A + B))
        print(np.linalg.norm(A) + np.linalg.norm(B))
```

```
4.69041575982343
7.795831523312719
```

Alternatively, we can write our own code to find the norm of `A`:

```
In [ ]: A = np.array([[1,2,3], [4,5,6], [7,8,9], [10,11,12]])
        m,n = A.shape
        print(A.shape)
        sum_of_sq = 0
        for i in range(m):
            for j in range(n):
                sum_of_sq = sum_of_sq + A[i,j]**2
        matrix_norm = np.sqrt(sum_of_sq)
        print(matrix_norm)
        print(np.linalg.norm(A))
```

```
(4, 3)
25.495097567963924
25.495097567963924
```

6.4. Matrix-vector multiplication

In Python, matrix-vector multiplication has the natural syntax `y = A @ x`. Alternatively, we can use the numpy function `np.matmul(A,x)`.

```
In [ ]: A = np.array([[0,2,-1],[-2,1,1]])
        x = np.array([2,1,-1])
        A @ x
```

```
Out[ ]: array([ 3, -4])
```

```
In [ ]: np.matmul(A,x)

Out[ ]: array([ 3, -4])
```

Difference matrix. An $(n-1)\times n$ difference matrix (equation (6.5) of VMLS) can be constructed in several ways. A simple one is the following.

```
In [ ]: diff_mat = lambda n: np.c_[-np.identity(n-1), np.zeros(n-1)] +
        ↪  np.c_[np.zeros(n-1), np.identity(n-1)]
        D = diff_mat(4)
        x = np.array([-1,0,2,1])
        D @ x

Out[ ]: array([ 1.,  2., -1.])
```

Since a difference matrix contains many zeros, this is a good opportunity to use sparse matrices. Here we use the `sparse.hstack()` function to stack the sparse matrices.

```
In [ ]: diff_mat = lambda n: sparse.hstack([-sparse.eye(n-1),
        ↪  sparse.coo_matrix((n-1,1))])+
        ↪  sparse.hstack([sparse.coo_matrix((n-1,1)), sparse.eye(n-1)])
        D = diff_mat(4)
        D @ np.array([-1,0,2,1])

Out[ ]: array([ 1.,  2., -1.])
```

Running sum matrix. The running sum matrix (equation (6.6) in VMLS) is a lower triangular matrix, with elements on and below the diagonal equal to one.

```
In [ ]: def running_sum(n):
            import numpy as np
            S = np.zeros((n,n))
            for i in range(n):
                for j in range(i+1):
                    S[i,j] = 1
            return S
        running_sum(4)

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

```
In [ ]: running_sum(4) @ np.array([-1,1,2,0])
```

```
Out[ ]: array([-1.,  0.,  2.,  2.])
```

An alternative construction is `np.tril(np.ones((n,n)))`. This uses the function `np.tril`, which sets the elements of a matrix above the diagonal to zero.

Vandermonde matrix. An $m \times n$ Vandermonde matrix (equation (6.7) in VMLS) has entries t_i^{j-1} for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. We define a function that takes an m-vector with elements $t_1, \ldots, t_m$ and returns the corresponding $m \times n$ Vandermonde matrix.

```
In [ ]: def vandermonde(t,n):
            m = len(t)
            V = np.zeros((m,n))
            for i in range(m):
                for j in range(n):
                    V[i,j] = t[i]**(j)
            return V
        vandermonde(np.array([-1,0,0.5,1]),5)
```

```
Out[ ]: array([[ 1.    , -1.    ,  1.    , -1.    ,  1.    ],
               [ 1.    ,  0.    ,  0.    ,  0.    ,  0.    ],
               [ 1.    ,  0.5   ,  0.25  ,  0.125 ,  0.0625],
               [ 1.    ,  1.    ,  1.    ,  1.    ,  1.    ]])
```

An alternative shorter definition uses numpy `np.column_stack` function.

```
In [ ]: vandermonde = lambda t,n: np.column_stack([t**i for i in
        ↪   range(n)])
        vandermonde(np.array([-1,0,0.5,1]),5)
```

```
Out[ ]: array([[ 1.    , -1.    ,  1.    , -1.    ,  1.    ],
               [ 1.    ,  0.    ,  0.    ,  0.    ,  0.    ],
               [ 1.    ,  0.5   ,  0.25  ,  0.125 ,  0.0625],
               [ 1.    ,  1.    ,  1.    ,  1.    ,  1.    ]])
```

6.5. Complexity

Complexity of matrix-vector multiplication. The complexity of multiplying an $m \times n$ matrix by an n-vector is $2mn$ flops. This grows linearly with both m and n. Let's check this.

In [ ]:
```
import time
A = np.random.random((1000,10000))
x = np.random.random(10000)
start = time.time()
y = A @ x
end = time.time()
print(end - start)
```

```
0.004207134246826172
```

In [ ]:
```
A = np.random.random((5000,20000))
x = np.random.random(20000)
start = time.time()
y = A @ x
end = time.time()
print(end - start)
```

```
0.04157733917236328
```

In the second matrix-vector multiplication, m increases by a factor of 5 and n increases by a factor of 2, so the complexity should be increased by a factor of 10. As we can see, it is increased by a factor around 10.

The increase in efficiency obtained by sparse matrix computations is seen from matrix-vector multiplications with the difference matrix.

In [ ]:
```
n = 10**4;
Ds = sparse.hstack([-sparse.eye(n-1),
sparse.coo_matrix((n-1,1))]) +
↪  sparse.hstack([sparse.coo_matrix((n-1,1)), sparse.eye(n-1)])
D = np.column_stack([np.eye(n-1), np.zeros(n-1)]) +
↪  np.column_stack([np.zeros(n-1), np.eye(n-1)])
x = np.random.normal(size=n)
import time
start = time.time()
D @ x
end = time.time()
print(end - start)
```

```
0.05832314491271973
```

```
In [ ]: start = time.time()
        Ds @ x
        end = time.time()
        print(end - start)
```

```
0.00030207633972167097
```

7. Matrix examples

7.1. Geometric transformations

Let us create a rotation matrix, and use it to rotate a set of points $\pi/3$ radians (60 deg). The result is in Figure 7.1.

```
In [ ]: Rot = lambda theta: [[np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]]
        R = Rot(np.pi/3)
        R
```

```
Out[ ]: [[0.5000000000000001, -0.8660254037844386],
         [0.8660254037844386, 0.5000000000000001]]
```

```
In [ ]: #create a list of 2-D points
        points =
        ↪   np.array([[1,0],[1.5,0],[2,0],[1,0.25],[1.5,0.25],[1,0.5]])
        #Now rotate them
        rpoints = np.array([R @ p for p in points])
        #Show the two sets of points
        import matplotlib.pyplot as plt
        plt.ion()
        plt.scatter([c[0] for c in points], [c[1] for c in points])
        plt.scatter([c[0] for c in rpoints],[c[1] for c in rpoints])
        plt.show()
```

7.2. Selectors

Reverser matrix. The reverser matrix can be created from an identity matrix by reversing the order of its rows. The numpy function `np.flip()` can be used for this purpose.

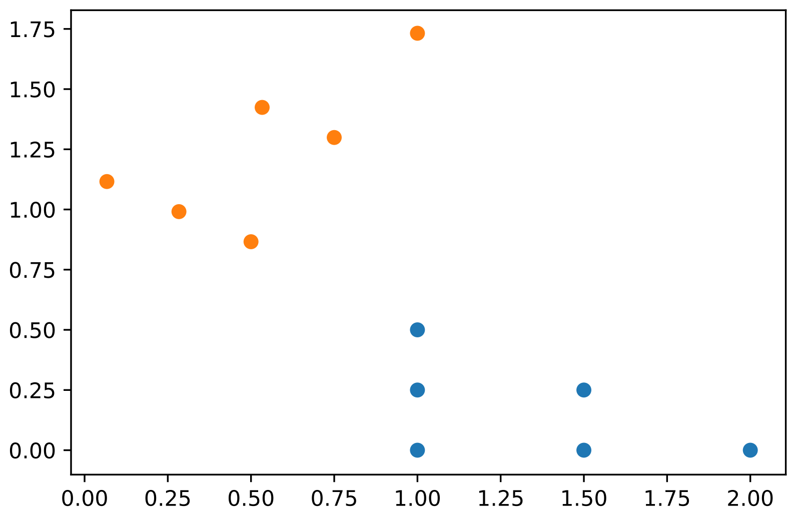


Figure 7.1.: Counterclockwise rotation by 60 degrees applied to six points.

In [ ]:
```python
reverser = lambda n: np.flip(np.eye(n),axis=0)
A = reverser(5)
A
```

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

Permutation matrix. Let's create a permutation matrix and use it to permute the entries of a vector. In Python, you can directly pass the permuted indexes to the vector.

In [ ]:
```python
A = np.array([[0,0,1], [1,0,0], [0,1,0]])
x = np.array([0.2, -1.7, 2.4])
A @ x # Permutes entries of x to [x[2], x[0], x[1]]
```

Out[ ]:
```
array([ 2.4,  0.2, -1.7])
```

In [ ]:
```python
x[[2,0,1]]
```

Out[ ]:
```
array([ 2.4,  0.2, -1.7])
```

7.3. Incidence matrix

Incidence matrix of a graph. We create the incidence matrix of the network shown in Figure 7.3 in VMLS.

```
In [ ]: A = np.array([[-1,-1,0,1,0], [1,0,-1,0,0], [0,0,1,-1,-1],
        ↪  [0,1,0,0,1]])
        xcirc = np.array([1,-1,1,0,1]) #A circulation
        A @ xcirc
```

```
Out[ ]: array([0, 0, 0, 0])
```

```
In [ ]: s = np.array([1,0,-1,0,]) # A source vector
        x = np.array([0.6,0.3,0.6,-0.1,-0.3]) #A flow vector
        A @ x + s #Total incoming flow at each node
```

```
Out[ ]: array([1.11022302e-16, 0.00000000e+00, 0.00000000e+00,
        ↪  0.00000000e+00])
```

Dirichlet energy. On page 135 of VMLS we compute the Dirichlet energy of two potential vectors associated with the graph of Figure 7.2 in VMLS.

```
In [ ]: A = np.array([[-1,-1,0,1,0], [1,0,-1,0,0], [0,0,1,-1,-1],
        ↪  [0,1,0,0,1]])
        vsmooth = np.array([1,2,2,1])
        np.linalg.norm(A.T @ vsmooth)**2 #Dirichlet energy of vsmooth
```

```
Out[ ]: 2.9999999999999996
```

```
In [ ]: vrough = np.array([1,-1, 2, -1])
        np.linalg.norm(A.T @ vrough)**2 # Dirichlet energy of vrough
```

```
Out[ ]: 27.0
```

7.4. Convolution

The numpy function `np.convolve()` can be used to compute the convolution of the vectors `a` and `b`. Let's use this to find the coefficients of the polynomial

$$p(x) = (1+x)(2-x+x^2)(1+x-2x^2) = 2+3x-3x^2-x^3+x^4-2x^5$$

```
In [ ]: a = np.array([1,1]) # coefficients of 1+x
        b = np.array([2,-1,1]) # coefficients of 2-x+x^2
        c = np.array([1,1,-2]) # coefficients of 1+x-2x^2
        d = np.convolve(np.convolve(a,b),c) # coefficients of product
        d

Out[ ]: array([ 2,  3, -3, -1,  1, -2])
```

Let's write a function that creates a Toeplitz matrix and check it against the `conv` function. We will also confirm that Python is using a very efficient method for computing the convolution.

To construct the Toeplitz matrix $T(b)$ defined in equation (7.3) of VMLS, we can first create a zero matrix of the correct dimensions ($(n+m-1)\times n$) and then add the coefficients b_i one by one. Single-index indexing comes in handy for this purpose. The single-index indexes of the elements b_i in the matrix$T(b)$ are $i, i+m+n, i+2(m+n), \ldots, i+(n-1)(m+n)$.

```
In [ ]: b = np.array([-1,2,3])
        a = np.array([-2,3,-1,1])
        def toeplitz(b,n):
            m = len(b)
            T = np.zeros((n+m-1,n))
            for j in range(n):
                T[j:j+m,j] = b
            return T
        Tb = toeplitz(b,len(a))
        Tb

Out[ ]: array([[-1.,  0.,  0.,  0.],
               [ 2., -1.,  0.,  0.],
               [ 3.,  2., -1.,  0.],
               [ 0.,  3.,  2., -1.],
               [ 0.,  0.,  3.,  2.],
               [ 0.,  0.,  0.,  3.]])

In [ ]: Tb @ a, np.convolve(b,a)

Out[ ]: (array([2.,-7., 1., 6.,-1., 3.]), array([2,-7, 1, 6,-1, 3]))
```

In [ ]:
```
import time
m = 2000
n = 2000
b = np.random.normal(size = n)
a = np.random.normal(size = m)
start = time.time()
ctoep = toeplitz(b,n) @ a
end = time.time()
print(end - start)
```

```
0.05798077583312988
```

In [ ]:
```
start = time.time()
cconv = np.convolve(a,b)
end = time.time()
print(end - start)
```

```
0.0011739730834960938
```

In [ ]:
```
np.linalg.norm(ctoep - cconv)
```

Out[ ]:
```
1.0371560230890881e-12
```

8. Linear equations

8.1. Linear and affine functions

Matrix-vector product function. Let's define an instance of the matrix-vector product function, and then numerically check that superposition holds.

```
In [ ]: A = np.array([[-0.1,2.8,-1.6],[2.3,-0.6,-3.6]]) #2 by 3 matrix A
        f = lambda x: A @ x
        #Let's check superposition
        x = np.array([1,2,3])
        y = np.array([-3,-1,2])
        alpha = 0.5
        beta = -1.6
        LHS = f(alpha*x + beta*y)
        print('LHS:', LHS)
        RHS = alpha*f(x) + beta*f(y)
        print('RHS:', RHS)
        print(np.linalg.norm(LHS - RHS))
```

```
LHS: [ 9.47 16.75]
RHS: [ 9.47 16.75]
1.7763568394002505e-15
```

```
In [ ]: f(np.array([0,1,0])) #Should be second column of A
```

```
Out[ ]: array([ 2.8, -0.6])
```

De-meaning matrix. Let's create a de-meaning matrix, and check that it works on a vector.

```
In [ ]: de_mean = lambda n: np.identity(n) - (1/n)
        x = np.array([0.2,2.3,1.0])
        de_mean(len(x)) @ x #De-mean using matrix multiplication
```

```
Out[ ]: array([-0.96666667,  1.13333333, -0.16666667])
```

```
In [ ]: x - sum(x)/len(x)
```

```
Out[ ]: array([-0.96666667,  1.13333333, -0.16666667])
```

Examples of functions that are not linear. The componentwise absolute value and the sort function are examples of nonlinear functions. These functions are easily computed by `abs` and `sorted`. By default, the `sorted` function sorts in increasing order, but this can be changed by adding an optional keyword argument.

```
In [ ]: f = lambda x: abs(x) #componentwise absolute value
        x = np.array([1,0])
        y = np.array([0,1])
        alpha = -1
        beta = 2
        f(alpha*x + beta*y)
```

```
Out[ ]: array([1, 2])
```

```
In [ ]: alpha*f(x) + beta*f(y)
```

```
Out[ ]: array([-1,  2])
```

```
In [ ]: f = lambda x: np.array(sorted(x, reverse = True))
        f(alpha*x + beta*y)
```

```
Out[ ]: array([ 2, -1])
```

```
In [ ]: alpha*f(x) + beta*f(y)
```

```
Out[ ]: array([1, 0])
```

8.2. Linear function models

Price elasticity of demand. Let's use a price elasticity of demand matrix to predict the demand for three products when the prices are changed a bit. Using this we can predict the change in total profit, given the manufacturing costs.

```
In [ ]: p = np.array([10, 20, 15]) #Current prices
        d = np.array([5.6, 1.5, 8.6]) #Current demand (say in thousands)
        c = np.array([6.5, 11.2, 9.8]) #Cost to manufacture
        profit = (p - c) @ d #Current total profit
```

```
print(profit)
```

```
77.51999999999998
```

In [ ]:
```
#Demand elesticity matrix
E =  np.array([[-0.3, 0.1, -0.1], [0.1, -0.5, 0.05], [-0.1, 0.05,
↪  -0.4]])
p_new = np.array([9,21,14]) #Proposed new prices
delta_p = (p_new - p)/p #Fractional change in prices
print(delta_p)
```

```
[-0.1         0.05       -0.06666667]
```

In [ ]:
```
delta_d = E @ delta_p # Predicted fractional change in demand
print(delta_d)
```

```
[ 0.04166667 -0.03833333  0.03916667]
```

In [ ]:
```
d_new = d * (1 + delta_d) # Predicted new demand
print(d_new)
```

```
[5.83333333 1.4425     8.93683333]
```

In [ ]:
```
profit_new = (p_new - c) @ d_new #Predicted new profit
print(profit_new)
```

```
66.25453333333333
```

If we trust the linear demand elasticity model, we should not make these price changes.

Taylor approximation. Consider the nonlinear function $f : \mathbf{R}^2 \to \mathbf{R}^2$ given by

$$f(x) = \begin{bmatrix} \|x-a\| \\ \|x-b\| \end{bmatrix} = \begin{bmatrix} \sqrt{(x_1-a_1)^2+(x_2-a_2)^2} \\ \sqrt{(x_1-b_1)^2+(x_2-b_2)^2} \end{bmatrix}.$$

The two components of f gives the distance of x to the points a and b. The function is differentiable, except when $x = a$ or $x = b$. Its derivative or Jacobian matrix is given by

$$Df(x) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1}(z) & \dfrac{\partial f_1}{\partial x_2}(z) \\ \dfrac{\partial f_2}{\partial x_1}(z) & \dfrac{\partial f_2}{\partial x_2}(z) \end{bmatrix} = \begin{bmatrix} \dfrac{z_1-a_1}{\|z-a\|} & \dfrac{z_2-a_2}{\|z-a\|} \\ \dfrac{z_1-b_1}{\|z-b\|} & \dfrac{z_2-b_2}{\|z-b\|} \end{bmatrix}.$$

Let's form the Taylor approximation of f for some specific values of a, b, and z, and then check it against the true value of f at a few points near z.

```
In [ ]: f = lambda x: np.array([np.linalg.norm(x-a),
                                np.linalg.norm(x-b)])
        Df = lambda z: np.array([(z-a)/np.linalg.norm(z-a),
                                 (z-b)/np.linalg.norm(z-b)])
        f_hat = lambda x: f(z) + Df(z)@(x - z)
        a = np.array([1,0])
        b = np.array([1,1])
        z = np.array([0,0])
        f(np.array([0.1,0.1]))
```

```
Out[ ]: array([0.90553851, 1.27279221])
```

```
In [ ]: f_hat(np.array([0.1,0.1]))
```

```
Out[ ]: array([0.9       , 1.27279221])
```

```
In [ ]: f(np.array([0.5,0.5]))
```

```
Out[ ]: array([0.70710678, 0.70710678])
```

```
In [ ]: f_hat(np.array([0.5,0.5]))
```

```
Out[ ]: array([0.5       , 0.70710678])
```

Regression model. We revisit the regression model for the house sales data in Section 2.3. The model is

$$\hat{y} = x^T\beta + \nu = \beta_1 x_1 + \beta_2 x_2 + \nu,$$

where $\hat{y}$ is the predicted house sales price, x_1 is the house area in 1000 square feet, and x_2 is the number of bedrooms.

In the following code we construct the 2×774 data matrix X and vector of outcomes y^{d}, for the $N = 774$ examples, in the data set. We then calculate the regression model predictions $\hat{y}^{\mathrm{d}}$, the prediction error r^{d}, and the RMS prediction errors.

```
In [ ]: # parameters in regression model
        beta = [148.73, -18.85]
        v = 54.40
        D = house_sales_data()
```

```
yd = D['price'] # vector of outcomes
N = len(yd)
X = np.vstack((D['area'], D['beds']))
X.shape
```

```
Out[ ]: (2, 774)
```

```
In [ ]: ydhat = beta @ X + v; # vector of predicted outcomes
        rd = yd - ydhat; # vector of predicted errors
        np.sqrt(sum(rd**2)/len(rd)) # RMS prediction error
```

```
Out[ ]: 74.84571862623025
```

```
In [ ]: # Compare with standard deviation of prices
        np.std(yd)
```

```
Out[ ]: 112.78216159756509
```

8.3. Systems of linear equations

Balancing chemical reactions. We verify the linear balancing equation on page 155 of VMLS, for the simple example of electrolysis of water.

```
In [ ]: R = np.array([2,1])
        P = np.array([[2,0], [0,2]])
        #Check balancing coefficients [2,2,1]
        coeff = np.array([2,2,1])
        coeff @ np.vstack((R, -P))
```

```
Out[ ]: array([0, 0])
```

9. Linear dynamical systems

9.1. Linear dynamical systems

Let's simulate a time-invariant linear dynamical system

$$x_{t+1} = Ax_t, \quad t = 1, \ldots, T,$$

with dynamics matrix

$$A = \begin{bmatrix} 0.97 & 0.10 & -0.05 \\ -0.30 & 0.99 & 0.05 \\ 0.01 & -0.04 & 0.96 \end{bmatrix}$$

and initial state $x_1 = (1, 0, -1)$. We store trajectory in the $n \times T$ matrix `state_traj`, with the ith column x_t. We plot the result in Figure 9.1.

In [ ]:
```
# initial state
x_1 = np.array([1,0,-1])
n = len(x_1)
T = 50
A = np.array([[0.97, 0.1, -0.05], [-0.3, 0.99, 0.05], [0.01,
↪   -0.04, 0.96]])
state_traj = np.column_stack([x_1, np.zeros((n,T-1))])
# dynamics recursion
for t in range(T-1):
    state_traj[:, t+1] = A @ state_traj[:,t]
import matplotlib.pyplot as plt
plt.ion()
plt.plot(np.arange(T), state_traj.T)
plt.legend(['(x_t)_1','(x_t)_2','(x_t)_3'])
plt.xlabel('t')
```

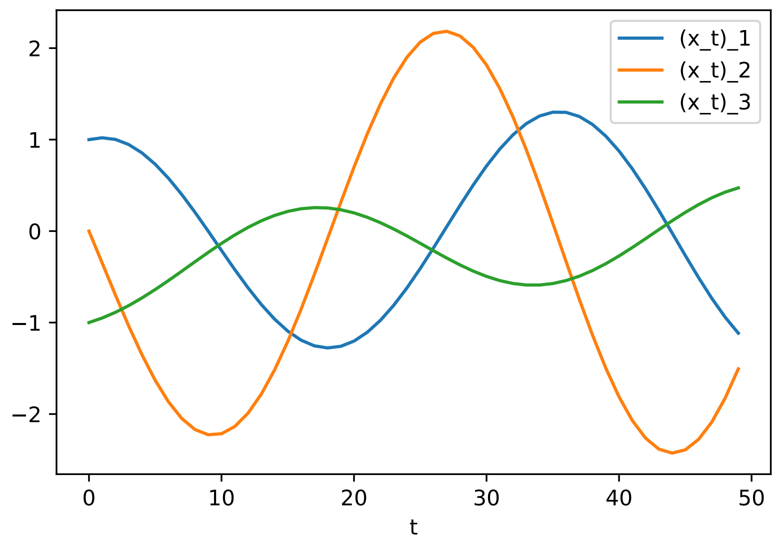


Figure 9.1.: Linear dynamical system simulation.

9.2. Population dynamics

We can create a population dynamics matrix with just one simple line of Python code. The following code predicts the 2020 population distribution in the US using the data of Section 9.2 of VMLS in the `population_data` dataset.

```
In [ ]: # Import 3 100-vector: population, birth_rate, death_rate
        D = population_data()
        b = D['birth_rate']
        d = D['death_rate']
        A = np.vstack([b, np.column_stack([np.diag(1 - d[:-1]),
        ↪  np.zeros((len(d) - 1))])])
        x = D['population']
        x = np.power(A,10) @ x
        import matplotlib.pyplot as plt
        plt.ion()
        plt.plot(x)
        plt.xlabel('Age')
        plt.ylabel('Population (millions)')
        plt.show()
```

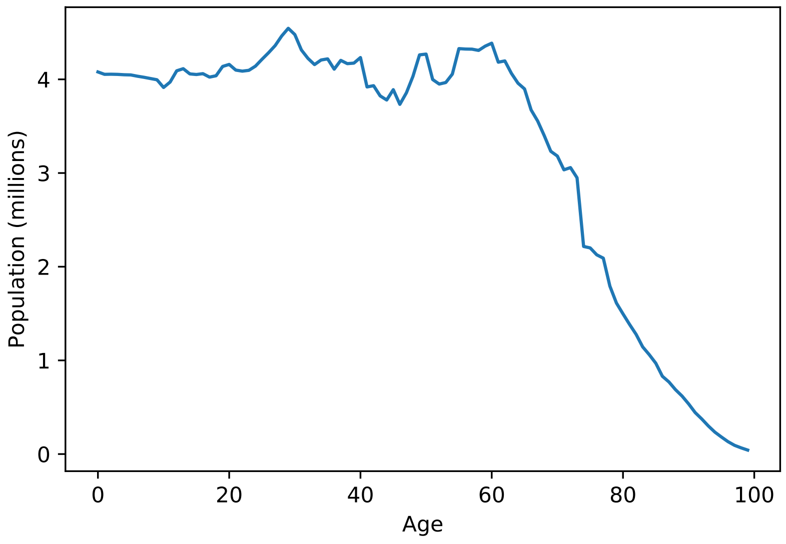


Figure 9.2.: Predicted age distribution in the US in 2020.

9.3. Epidemic dynamics

Let's implement the simulation of the epidemic dynamics from VMLS section 9.3.

```
In [ ]: T = 210
        A = np.array([[0.95,0.04,0,0], [0.05,0.85,0,0], [0,0.1,1,0],
        ↪  [0,0.01,0,1]])
        x_1 = np.array([1,0,0,0])
        # state trajectory
        state_traj = np.column_stack([x_1, np.zeros((4, T-1))])
        # dynamics recursion
        for t in range(T-1):
            state_traj[:, t+1] = A @ state_traj[:,t]
        import matplotlib.pyplot as plt
        plt.ion()
        plt.plot(np.arange(T), state_traj.T)
        plt.xlabel('Time t')
        plt.legend(['Susceptible', 'Infected', 'Recovered', 'Deceased'])
        plt.show()
```

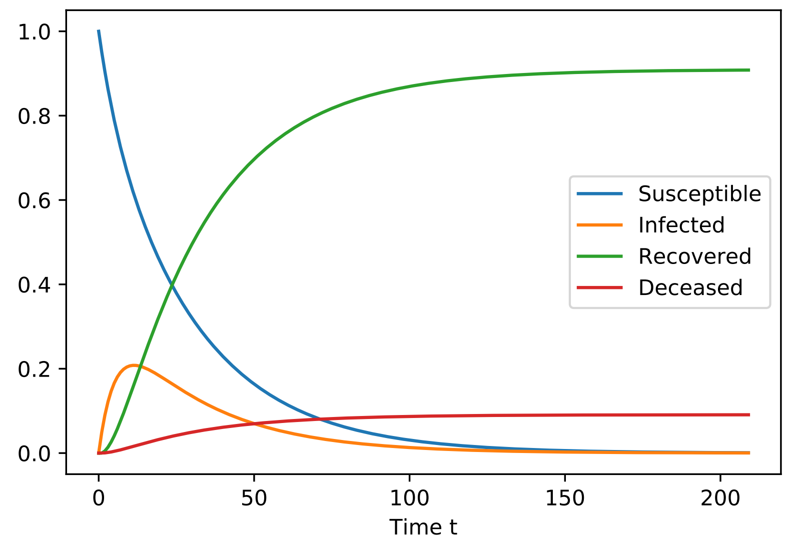


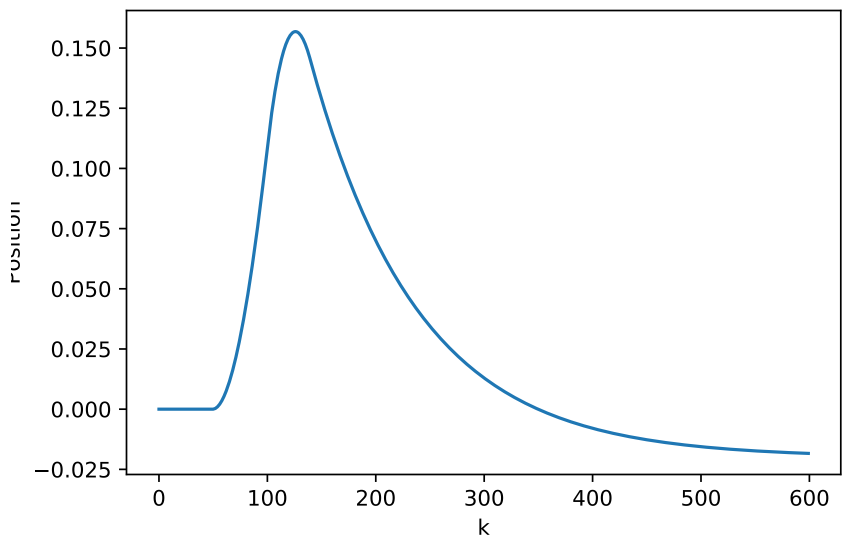
Figure 9.3.: Simulation of epidemic dynamics.

9.4. Motion of a mass

Let's simulate the discretized model of the motion of a mass in section 9.4 of VMLS.

```
In [ ]: h = 0.01
        m = 1
        eta = 1
        A = np.block([[1,h],[0, 1-h*eta/m]])
        B = np.vstack([0,h/m])
        x1 = np.array([0,0])
        K = 600 #simulate for K*h = 6 seconds
        f = np.zeros((K));
        f[49:99] = 1
        f[99:139] = -1.3
        X = np.column_stack([x1, np.zeros((2, K-1))])
        for k in range(K-1):
            X[:, k+1] = A @ X[:, k] + f[k]*B.T
        import matplotlib.pyplot as plt
        plt.ion()
        plt.plot(X[0,:])
        plt.xlabel('k')
        plt.ylabel('Position')
plt.show()
plt.plot(X[1,:])
plt.xlabel('k')
plt.ylabel('Velocity')
plt.show()
```

9.5. Supply chain dynamics



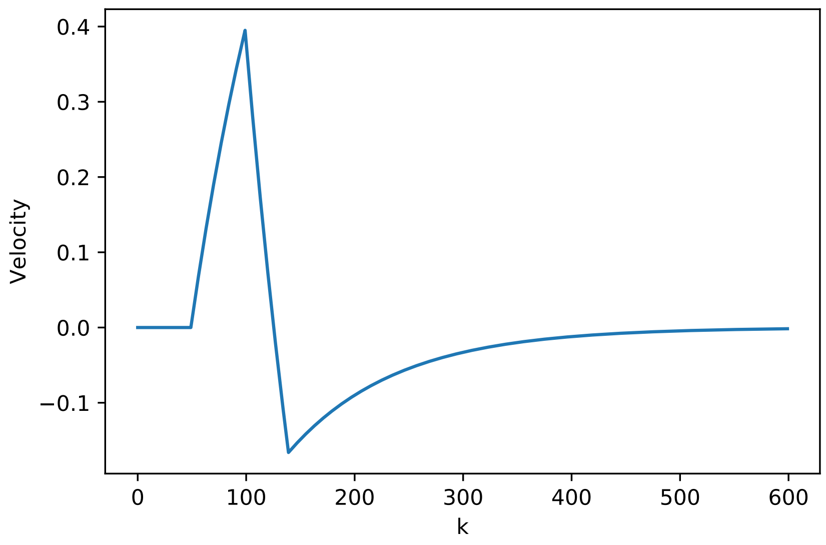


Figure 9.4.: Simulation of a mass moving along a line: position (top) and velocity (bottom).

Figure 9.5.

10. Matrix multiplication

10.1. Matrix-matrix multiplication

In Python the product of matrices `A` and `B` is obtained with `A @ B`. Alternatively, we can compute the matrix product using the function `np.matmul()`. We calculate the matrix product on page 177 of VMLS.

```
In [ ]: A = np.array([[-1.5, 3, 2], [1, -1,0]]) #2 by 3 matrix
        B = np.array([[-1,-1], [0,-2], [1,0]]) #3 by 2 matrix
        C = A @ B
        print(C)
```

```
[[ 3.5 -4.5]
```

```
In [ ]:  [-1.   1. ]]
         C = np.matmul(A,B)
        print(C)
```

```
[[ 3.5 -4.5]
 [-1.   1. ]]
```

Gram matrix. The Gram matrix of a matrix A is the matrix $G = A^T A$. It is a symmetric matrix and the i, j element $G_{i,j}$ is the inner product of columns i and j of A.

```
In [ ]: A = np.random.normal(size = (10,3))
        G = A.T @ A
        print(G)
```

```
[[ 8.73234113 -2.38507779  1.59667064]
 [-2.38507779  3.07242591  0.49804144]
 [ 1.59667064  0.49804144 15.65621955]]
```

```
In [ ]: #Gii is norm of column i, squared
        G[1,1]
```

```
Out[ ]: 3.072425913055046
```

```
In [ ]: np.linalg.norm(A[:,1])**2
```

```
Out[ ]: 3.072425913055046
```

```
In [ ]: #Gij is inner product of columns i and j
        G[0,2]
```

```
Out[ ]: 1.5966706354935412
```

```
In [ ]: A[:,0] @ A[:,2]
```

```
Out[ ]: 1.5966706354935412
```

Complexity of matrix triple product. Let's check the associative property, which states that $(AB)C = A(BC)$ for any $m \times n$ matrix A, any $n \times p$ matrix B, and any $p \times q$ matrix C. At the same time we will see that the left-hand and right-hand sides take very different amounts of time to compute.

```
In [ ]: import time
        m = 2000
        n = 50
        q = 2000
        p = 2000
        A = np.random.normal(size = (m,n))
        B = np.random.normal(size = (n,p))
        C = np.random.normal(size = (p,q))
        start = time.time()
        LHS = (A @ B) @ C
        end = time.time()
        print(end - start)
```

```
0.17589497566223145
```

```
In [ ]: start = time.time()
        LHS = (A @ B) @ C
        end = time.time()
        print(end - start)
```

```
0.23308205604553223
```

```
In [ ]: start = time.time()
        RHS = A @ (B @ C)
        end = time.time()
        print(end - start)
```

```
0.026722192764282227
```

```
In [ ]: start = time.time()
        RHS = A @ (B @ C)
        end = time.time()
        print(end - start)
```

```
0.025452852249145508
```

```
In [ ]: np.linalg.norm(LHS - RHS)
```

```
Out[ ]: 4.704725926477414e-10
```

```
In [ ]: start = time.time()
        D = A @ B @ C       #Evaluated as (A@B)@C or as A@(B@C)?
        end = time.time()
        print(end - start)
```

```
0.24454998970031738
```

From the above, we see that evaluating `(A@B)@C` takes around 10 times as much time as evaluating `A@(B@C)`, which is predicted from the complexities. In the last line, we deduce that `$A@B@C$` is evaluated left to right, as `$(A@B)@C$`. Note that for these particular matrices, this is the (much) slower order to multiply the matrices.

10.2. Composition of linear functions

Second difference matrix. We compute the second difference matrix on page 184 of VMLS.

```
In [ ]: D = lambda n: np.c_[-np.identity(n-1), np.zeros(n-1)] +
        ↪  np.c_[np.zeros(n-1), np.identity(n-1)]
        D(5)
```

```
Out[ ]: array([[-1.,  1.,  0.,  0.,  0.],
               [ 0., -1.,  1.,  0.,  0.],
               [ 0.,  0., -1.,  1.,  0.],
       [ 0.,  0.,  0., -1.,  1.]])
```

```
In [ ]: D(4)
```

```
Out[ ]: array([[-1.,  1.,  0.,  0.],
               [ 0., -1.,  1.,  0.],
               [ 0.,  0., -1.,  1.]])
```

```
In [ ]: Delta = D(4) @ D(5) #Second difference matrix
        print(Delta)
```

```
[[ 1. -2.  1.  0.  0.]
 [ 0.  1. -2.  1.  0.]
 [ 0.  0.  1. -2.  1.]]
```

10.3. Matrix power

The kth power of a square matrix A is denoted A^k. In Python, this power is formed using `np.linalg.matrix_power(A,k)`. Let's form the adjacency matrix of the directed graph on VMLS page 186. Then let's find out how many cycles of length 8 there are, starting from each node. (A cycle is a path that starts and stops at the same node.)

```
In [ ]: A = np.array([[0,1,0,0,1], [1,0,1,0,0], [0,0,1,1,1], [1,0,0,0,0],
        ↪  [0,0,0,1,0]])
        np.linalg.matrix_power(A,2)
```

```
Out[ ]: array([[1, 0, 1, 1, 0],
               [0, 1, 1, 1, 2],
               [1, 0, 1, 2, 1],
               [0, 1, 0, 0, 1],
               [1, 0, 0, 0, 0]])
```

```
In [ ]: np.linalg.matrix_power(A,8)
```

```
Out[ ]: array([[18, 11, 15, 20, 20],
               [25, 14, 21, 28, 26],
               [24, 14, 20, 27, 26],
               [11,  6,  9, 12, 11],
               [ 6,  4,  5,  7,  7]])
```

```
In [ ]: num_of_cycles = np.diag(np.linalg.matrix_power(A,8))
        print(num_of_cycles)
```

```
[18 14 20 12  7]
```

Population dynamics. Let's write the code for figure 10.2 in VMLS, which plots the contribution factor to the total US population in 2020 (ignoring immigration), for each age in 2010. The Python plot is in figure 10.1. We can see that, not surprisingly, $20-25$ years olds have the highest contributing factor, around 1.5. This means that on average, each $20-25$ year old in 2010 will be responsible for around 1.5 people in 2020. This takes into account any children they may have before then, and (as a very small effect) the few of them who will no longer be with us in 2020.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        D = population_data();
        b = D['birth_rate'];
        d = D['death_rate'];
        # Dynamics matrix for populaion dynamics
        A = np.vstack([b, np.column_stack([np.diag(1-d[:-1]),
        ↪  np.zeros((len(d)-1))])])
        # Contribution factor to total poulation in 2020
        # from each age in 2010
        cf = np.ones(100) @ np.linalg.matrix_power(A,10) # Contribution
        ↪  factor
        plt.plot(cf)
        plt.xlabel('Age')
        plt.ylabel('Factor')
```

10.4. QR factorization

In Python, we can write a `QR_factorization` function to perform the QR factorization of a matrix A using the `gram_schmidt` function on page 41.

Remarks. When we wrote the `gram_schmidt` function, we were checking whether the *rows* of the numpy array (matrix A) are linearly independent. However, in QR factorisation, we should perform the Gram-Schmidt on the *columns* of matrix A. Therefore, A^T should be the input to the `QR_factorization` instead of A.

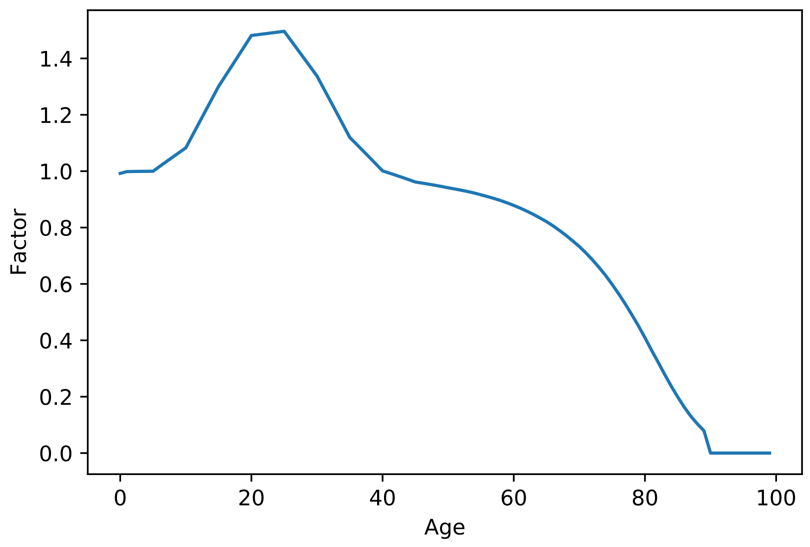


Figure 10.1.: Contribution factor per age in 2010 to the total population in 2020. The value of age $i-1$ is the ith component of the row vector $\mathbf{1}^T\mathbf{A}^10$.

```
In [ ]: def QR_factorization(A):
            Q_transpose = np.array(gram_schmidt(A.T))
            R = Q_transpose @ A
            Q = Q_transpose.T
            return Q, R
        Q, R = QR_factorization(A)
```

Alternatively, we can use the numpy function `np.linalg.qr(A)`.

```
In [ ]: Q, R = np.linalg.qr(A)
```

Here we would like to highlight a minor difference in the following examples with the VMLS definition. The R factor computed in the `QR_factorization` function may have negative elements on the diagonal, as opposed to only positive elements if we follow the definition used in VMLS and in `np.linalg.qr(A)`. The two definitions are equivalent, because R_{ii} is negative, one can change the sign of the ith row of R and the ith column of Q, to get an equivalent factorization with $R_{ii} > 0$. However, this step is not needed in practice, since negative elements on the diagonal do not pose any problem in applications of the QR factorization.

```
In [ ]: A = np.random.normal(size = (6,4))
        Q, R = np.linalg.qr(A)
        R
```

```
Out[ ]: array([[-2.96963114,  0.77296876,  1.99410713,  0.30491373],
               [ 0.        ,  0.73315602,  1.0283551 , -0.13470329],
               [ 0.        ,  0.        , -3.03153124,  0.60705777],
               [ 0.        ,  0.        ,  0.        , -1.0449406 ]])
```

```
In [ ]: q, r = QR_factorization(A)
        r
```

```
Out[ ]: array([[ 2.96963114e+00, -7.72968759e-01, -1.99410713e+00,
        ↪  -3.04913735e-01],
               [-7.77156117e-16,  7.33156025e-01,  1.02835510e+00,
               ↪  -1.34703286e-01],
               [-8.32667268e-17, -1.66533454e-16,  3.03153124e+00,
               ↪  -6.07057767e-01],
               [-2.22044605e-16, -1.11022302e-16,  8.88178420e-16,
               ↪  1.04494060e+00]])
```

```
In [ ]: np.linalg.norm(Q @ R - A)
```

```
Out[ ]: 1.3188305818502897e-15
```

```
In [ ]: Q.T @ Q
```

```
Out[ ]: array([[ 1.00000000e+00,  9.39209220e-17,  1.90941200e-16,
        ↪  -4.69424568e-17],
               [ 9.39209220e-17,  1.00000000e+00,  2.25326899e-16,
               ↪  2.04228665e-16],
               [ 1.90941200e-16,  2.25326899e-16,  1.00000000e+00,
               ↪  -4.57762562e-16],
               [-4.69424568e-17,  2.04228665e-16, -4.57762562e-16,
               ↪  1.00000000e+00]])
```

11. Matrix inverses

11.1. Left and right inverses

We'll see later how to find a left or right inverse, when one exists.

```
In [ ]: A = np.array([[-3,-4], [4,6], [1,1]])
        B = np.array([[-11,-10,16], [7,8,-11]])/9 #left inverse of A
        C = np.array([[0,-1,6], [0,1,-4]])/2 #Another left inverse of A
        #Let's check
        B @ A
```

```
Out[ ]: array([[ 1.0000000e+00,  0.0000000e+00],
               [-4.4408921e-16,  1.0000000e+00]])
```

```
In [ ]: C @ A
```

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

11.2. Inverse

If `A` is invertible, its inverse is given by `np.linalg.inv(A)`. You'll get an error if `A` is not invertible, or not square.

```
In [ ]: A = np.array([[1,-2,3], [0,2,2], [-4,-4, -4]])
        B = np.linalg.inv(A)
        B
```

```
Out[ ]: array([[-2.77555756e-17, -5.00000000e-01, -2.50000000e-01],
               [-2.00000000e-01,  2.00000000e-01, -5.00000000e-02],
               [ 2.00000000e-01,  3.00000000e-01,  5.00000000e-02]])
```

```
In [ ]: B @ A
```

```
Out[ ]: array([[ 1.00000000e+00, -2.22044605e-16, -2.22044605e-16],
               [ 0.00000000e+00,  1.00000000e+00,  8.32667268e-17],
               [ 0.00000000e+00,  5.55111512e-17,  1.00000000e+00]])
```

```
In [ ]: A @ B
```

```
Out[ ]: array([[ 1.00000000e+00,  0.00000000e+00, -1.38777878e-17],
               [ 5.55111512e-17,  1.00000000e+00,  1.38777878e-17],
               [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
```

Dual basis. The next example illustrates the dual basis provided by the rows of the inverse of $B = A^{-1}$. We calculate the expansion

$$x = (b_1^T x)a_1 + \cdots + (b_n^T x)a_n$$

for a 3×3 example (see page 205 of VMLS).

```
In [ ]: A = np.array([[1,0,1], [4,-3,-4], [1,-1,-2]])
        B = np.linalg.inv(A)
        x = np.array([0.2,-0.3,1.2])
        RHS = (B[0,:]@x) * A[:,0] + (B[1,:]@x) * A[:,1] + (B[2,:]@x) *
        ↪  A[:,2]
        print(RHS)
```

```
[ 0.2 -0.3  1.2]
```

Inverse via QR factorization. The inverse of a matrix A can be computed from its QR factorization $A = QR$ via the formula $A^{-1} = R^{-1}Q^T$.

```
In [ ]: A = np.random.normal(size = (3,3))
        np.linalg.inv(A)
```

```
Out[ ]: array([[ 0.83904201, -1.17279605,  1.02812262],
               [ 1.28411762, -0.30441464,  0.9310179 ],
               [ 0.06402464, -0.0154395 ,  1.51759022]])
```

```
In [ ]: Q,R = QR_factorization(A)
        np.linalg.inv(R) @ Q.T
```

```
Out[ ]: array([[ 0.83904201, -1.17279605,  1.02812262],
               [ 1.28411762, -0.30441464,  0.9310179 ],
       [ 0.06402464, -0.0154395 ,  1.51759022]])
```

11.3. Solving linear equations

Back substitution. Let's first implement back substitution (VMLS Algorithm 11.1) in Python, and check it.

```
In [ ]: def back_subst(R,b_tilde):
            n = R.shape[0]
            x = np.zeros(n)
            for i in reversed(range(n)):
                x[i] = b_tilde[i]
                for j in range(i+1,n):
                    x[i] = x[i] - R[i,j]*x[j]
                x[i] = x[i]/R[i,i]
            return x
        R = np.triu(np.random.random((4,4)))
        b = np.random.random(4)
        x = back_subst(R,b)
        np.linalg.norm(R @ x - b)
```

```
Out[ ]: 1.1102230246251565e-16
```

The function `np.triu()` gives the upper triangular part of a matrix, *i.e.*, it zeros out the entries below the diagonal.

Solving system of linear equations. Using the `gram_schmidt`, `QR_factorization` and `back_subst` functions that we have defined in the previous section, we can define our own function to solve a system of linear equations. This function implements the algorithm 12.1 in VMLS.

```
In [ ]: def solve_via_backsub(A,b):
            Q,R = QR_factorization(A)
            b_tilde = Q.T @ b
            x = back_subst(R,b_tilde)
            return x
```

This requires you to include the other functions in your code as well.

Alternatively, we can use the numpy function `np.linalg.solve(A,b)` to solve a set of linear equations

$$Ax = b.$$

This is faster than `x = np.linalg.inv(A)@ b`, which first computes the inverse of A and then multiplies it with b. However, this function computes the exact solution of a well-determined system.

```
In [ ]: import time
        n = 5000
        A = np.random.normal(size = (n,n))
        b = np.random.normal(size = n)
        start = time.time()
        x1 = np.linalg.solve(A,b)
        end = time.time()
        print(np.linalg.norm(b - A @ x1))
        print(end - start)
```

```
4.033331000615254e-09
1.2627429962158203
```

```
In [ ]: start = time.time()
        x2 = np.linalg.inv(A) @ b
        end = time.time()
        print(np.linalg.norm(b - A @ x2))
        print(end - start)
```

```
8.855382050136278e-10
4.3922741413116455
```

11.4. Pseudo-inverse

In Python the pseudo-inverse of a matrix `A` is obtained with `np.linalg.pinv()`. We compute the pseudo-inverse for the example of page 216 of VMLS using the `np.linalg.pinv()` function and via the formula $A^\dagger = R^{-1}Q^T$, where $A = QR$ is the QR factorization of A.

```
In [ ]: A = np.array([[-3,-4],[4,6],[1,1]])
        np.linalg.pinv(A)
        Q, R = np.linalg.qr(A)
        R
```

```
Out[ ]: array([[ 5.09901951,  7.256297  ],
               [ 0.        , -0.58834841]])
```

```
In [ ]: np.linalg.solve(R,Q.T)
```

```
Out[ ]: array([[-1.22222222, -1.11111111,  1.77777778],
               [ 0.77777778,  0.88888889, -1.22222222]])
```

Part III.

Least squares

12. Least squares

12.1. Least squares problem

We take the small least squares problem of Figure 12.1 in VMLS and check that $\|A\hat{x}-b\|$ is less than $\|Ax-b\|$ for some other value of x.

```
In [ ]: A = np.array([[2,0],[-1,1],[0,2]])
        b = np.array([1,0,-1])
        x_hat = np.array([1/3, -1/3])
        r_hat = A @ x_hat - b
        np.linalg.norm(r_hat)
```

```
Out[ ]: 0.816496580927726
```

```
In [ ]: x = np.array([1/2, -1/2]) #other value of x
        r = A @ x - b
        np.linalg.norm(r)
```

```
Out[ ]: 1.0
```

12.2. Solution

Least squares solution formula. Let's check the solution formulas (12.5) and (12.6) in VMLS,

$$\hat{x} = (A^TA)^{-1}A^Tb = A^\dagger b$$

for the small example of Figure 12.1 (where $\hat{x} = (1/3, -1/3)$).

```
In [ ]: np.linalg.inv(A.T @ A) @ A.T @ b
```

```
Out[ ]: array([ 0.33333333, -0.33333333])
```

```
In [ ]: np.linalg.pinv(A) @ b
```

```
Out[ ]: array([ 0.33333333, -0.33333333])
```

```
In [ ]: (A.T @ A) @ x_hat - A.T @ b #Check that normal equations hold
```

```
Out[ ]: array([-2.22044605e-16,  2.22044605e-16])
```

Orthogonality principle. Let's check the orthogonality principle for the same example.

```
In [ ]: z = np.array([-1.1,2.3])
        (A @ z).T @ r_hat
```

```
Out[ ]: 2.220446049250313e-16
```

```
In [ ]: z = np.array([5.3, -1.2])
        (A @ z).T @ r_hat
```

```
Out[ ]: -6.661338147750939e-16
```

12.3. Solving least squares problems

We can find the least square approximate solution using the `solve_via_backsub` we defined in section 11.3.

```
In [ ]: A = np.random.normal(size = (100,20))
        b = np.random.normal(size = 100)
        x1 = solve_via_backsub(A,b)
        x2 = np.linalg.inv(A.T @ A) @ (A.T @ b)
        x3 = np.linalg.pinv(A) @ b
        print(np.linalg.norm(x1-x2))
        print(np.linalg.norm(x2-x3))
        print(np.linalg.norm(x3-x1))
```

```
3.025314970522842e-16
7.545708143930438e-16
7.908978925534183e-16
```

Complexity. The complexity of solving the least squares problem with $m \times n$ matrix A is around $2mn^2$ flops. Let's check this in Python by solving a few least squares problems of different dimensions.

```
In [ ]: m = 2000
        n = 500
        A = np.random.normal(size = (m,n))
```

```
b = np.random.normal(size = m)
start = time.time()
x = solve_via_backsub(A,b)
end = time.time()
print(end - start)
```

```
2.00296688079834
```

```
In [ ]: m = 4000
        n = 500
        A = np.random.normal(size = (m,n))
        b = np.random.normal(size = m)
        start = time.time()
        x = solve_via_backsub(A,b)
        end = time.time()
        print(end - start)
```

```
3.680551290512085
```

```
In [ ]: m = 2000
        n = 1000
        A = np.random.normal(size = (m,n))
        b = np.random.normal(size = m)
        start = time.time()
        x = solve_via_backsub(A,b)
        end = time.time()
        print(end - start)
```

```
7.565088987350464
```

We can see that doubling m approximately doubles the computation time, and doubling n increases it by around a factor of four. The times above can be used to guess the speed of the computer on which it was carried out. For example, using the last problem solved, the number of flops is around $2mn^2 = 4 \times 10^9$, and it look around 0.4 seconds. This suggests a speed of around 10 Gflop per second.

12.4. Examples

Advertising purchases. We work out the solution of the optimal advertising purchase problem on page 234 of VMLS.

In [ ]:
```
n = 3
m = 10
R = np.array([[0.97,1.86,0.41], [1.23,2.18,0.53],
↪  [0.80,1.24,0.62], [1.29,0.98,0.51], [1.10,1.23,0.69],
↪  [0.67,0.34,0.54], [0.87,0.26,0.62], [1.10,0.16,0.48],
↪  [1.92,0.22,0.71], [1.29,0.12,0.62]])
v_des = 1e+3*np.ones(10)
s = solve_via_backsub(R,v_des)
print(s)
```

```
array([  62.07662454,   99.98500403, 1442.83746254])
```

In [ ]:
```
np.sqrt(sum((R @ s - v_des)**2)/len(v_des))
```

Out[ ]:
```
132.63819026326527
```

Illumination. The following code constructions and solves the illumination problem on page 234, and plots two histograms with the pixel intensity distributions (Figure (12.1)).

In [ ]:
```
import numpy as np
# number of lamps
n = 10
# x, y positions of lamps and height above floor
lamps = np.array([[4.1 ,20.4, 4],[14.1, 21.3, 3.5],[22.6, 17.1,
↪  6], [5.5 ,12.3, 4.0], [12.2, 9.7, 4.0], [15.3, 13.8, 6],
↪  [21.3, 10.5, 5.5], [3.9 ,3.3, 5.0], [13.1, 4.3, 5.0], [20.3,
↪  4.2, 4.5]])
N = 25 # grid size
m = N*N # number of pixels
# construct m x 2 matrix with coordinates of pixel centers
pixels = np.hstack([np.outer(np.arange(0.5,N,1),
↪  np.ones(N)).reshape(m,1), np.outer(np.ones(N),
↪  np.arange(0.5,N,1)).reshape(m,1)])
# The m x n matrix A maps lamp powers to pixel intensities.
# A[i,j] is inversely proportional to the squared distance of
# lamp j to pixel i.
A = np.zeros((m,n))
for i in range(m):
    for j in range(n):
        A[i,j] = 1.0 / (np.linalg.norm(np.hstack([pixels[i,:], 0])
        ↪  - lamps[j,:])**2)

A = (m/np.sum(A)) * A # scale elements of A
# Least squares solution
x = solve_via_backsub(A, np.ones(m))
rms_ls = (sum((A @ x - 1)**2)/m)**0.5
print(rms_ls)
```

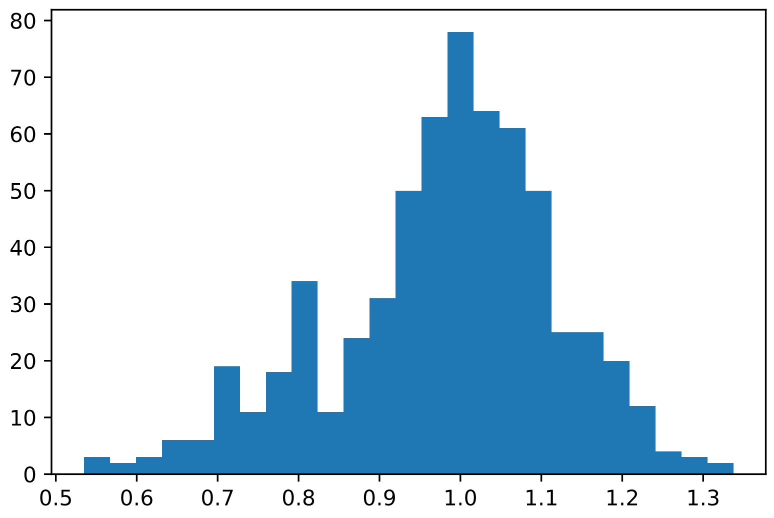
```
0.14039048134276055
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
plt.hist(A @ x, bins = 25)
plt.show()
```

In [ ]:
```
# Intensity if all lamp powers are one
rms_uniform = (sum((A @ np.ones(n) - 1)**2)/m)**0.5
print(rms_uniform)
```

```
0.24174131853807873
```

In [ ]:
```
plt.hist(A @ np.ones(n), bins = 25)
plt.show()
```



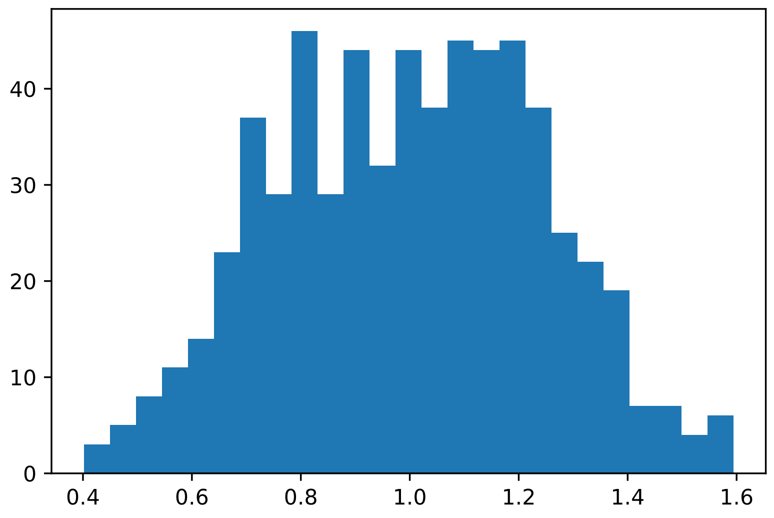


Figure 12.1.: Histogram of pixel illumination values using $p = \mathbf{1}$(top) and $\hat{p}$ (bottom). The target intensity value is one.

13. Least squares data fitting

13.1. Least squares data fitting

Straight-line fit. A straight-line fit to time series data gives an estimate of a trend line. In Figure 13.3 of VMLS we apply this to a time series of petroleum consumption. The figure is reproduced here as Figure 13.1.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        consumption = petroleum_consumption_data()
        n = len(consumption)
        A = np.column_stack((np.ones(n),np.arange(n)))
        x = solve_via_backsub(A,consumption)
        plt.scatter(np.arange(1980,2014), consumption)
        plt.plot(np.arange(1980,2014), A @ x, 'r')
        plt.show()
```

Estimation of trend and seasonal component. The next example is the least squares fit of a trend plus a periodic component to a time series. In VMLS this was illustrated with a time series of vehicle miles traveled in the US, per month, for 15 years (2000-2014). The following Python code replicates Figure 13.2 in VMLS. It imports the data via the function `vehicle_miles_data`, which creates a 15×12 matrix `vmt`, with the monthly values of each of the 15 years.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        vmt = vehicle_miles_data()
        m = 15*12
        A = np.column_stack((np.arange(m), np.vstack([np.identity(12) for
        ↪  i in range(15)])))
        b = np.reshape(vmt.T, m, 1)
        x = solve_via_backsub(A,b)
        plt.scatter(np.arange(m), b)
```

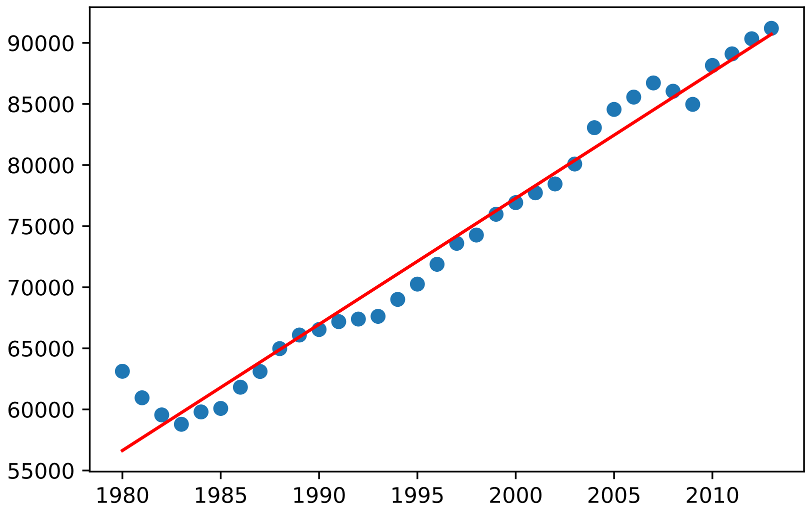


Figure 13.1.: World petroleum consumption between 1980 and 2013 (dots) and least squares straight-line fit (data from www.eia.gov).

```
plt.plot(np.arange(m), A @ x, 'r')
plt.show()
```

The matrix A in this example has size $m \times n$ where $m = 15 \times 12 = 180$ and $n = 13$. The first column has entries $0, 1, 2, \ldots, 179$. The remaining columns are formed by vertical stacking of 15 identity matrices of size 12×12. The Python expression `np.vstack([ np.identity(12)for i in range(15)])` creates an array of 15 identity matrices, and then stacks them vertically. The plot produced by the code is shown in Figure 13.2.

Polynomial fit. We now discuss the polynomial fitting problem on page 255 in VMLS and the results shown in Figure 13.4. We first generate a training set of 100 points and plot them (Figure 13.3).

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        #Generate training data in the interval [-1,1]
        m = 100
        t = -1 + 2*np.random.random(m)
        y = np.power(t,3) - t + 0.4 / (1 + 25*np.power(t,2)) +
        ↪  0.10*np.random.normal(size = m)
```

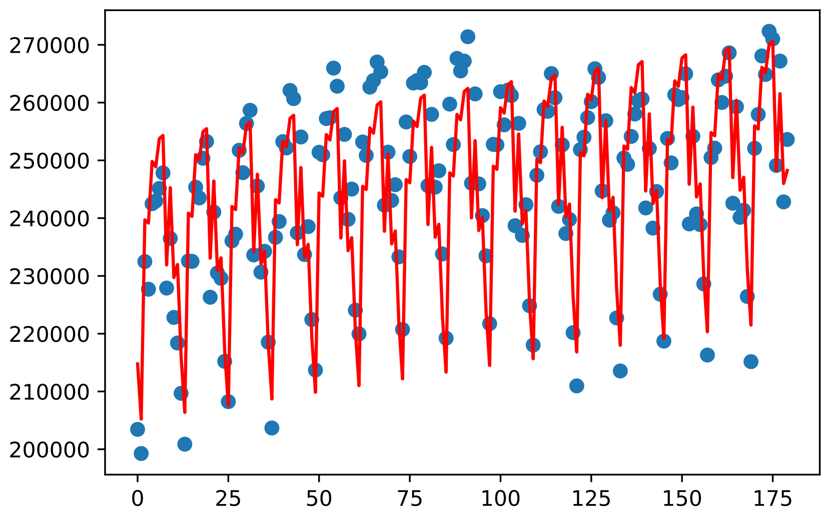


Figure 13.2.: The dots show vehicle miles traveled in the US, per month, in the period January 2000 - December 2014. The line shows the least squares fit of a linear trend and a seasonal component with a 12-month period.

```
plt.scatter(t,y)
plt.show()
```

Next we define a function that fits the polynomial coefficients using least squares. We apply the function to fit polynomials of degree $2, 6, 10, 15$ to our training set.

```
In [ ]: polyfit = lambda t,y,p: solve_via_backsub(vandermonde(t,p),y)
        theta2 = polyfit(t,y,3)
        theta6 = polyfit(t,y,7)
        theta10 = polyfit(t,y,11)
        theta15 = polyfit(t,y,16)
```

Finally, we plot the four polynomials. To simplify this, we first write a function that evaluates a polynomial at all points specified in a vector `x`. The plots are in Figure 13.4.

```
In [ ]: polyeval = lambda theta,x: vandermonde(x, len(theta)) @ theta
        t_plot = np.linspace(-1,1,num = 100)
        fig = plt.figure()
        plt.ylim(-0.7, 0.7)

        plt.subplot(2, 2, 1)
```

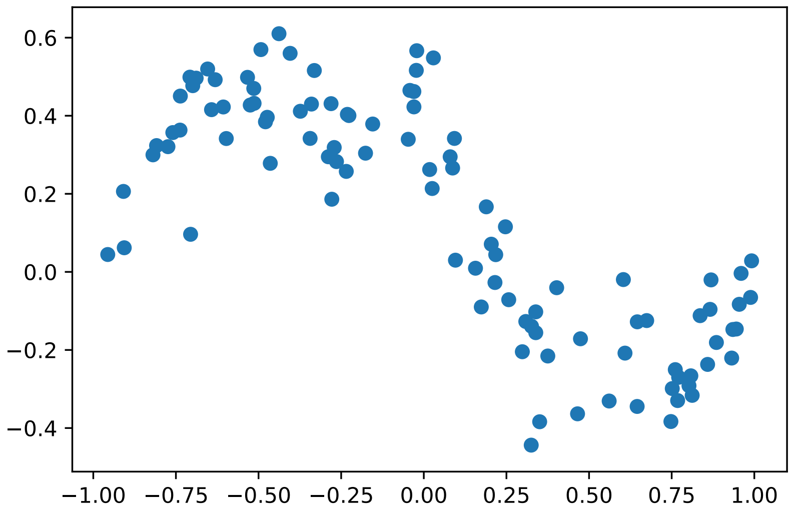


Figure 13.3.: Training set used in the polynomial fitting example.

```
plt.scatter(t,y)
plt.plot(t_plot, polyeval(theta2, t_plot),'r')

plt.subplot(2, 2, 2)
plt.scatter(t,y)
plt.plot(t_plot, polyeval(theta6, t_plot),'r')

plt.subplot(2, 2, 3)
plt.scatter(t,y)
plt.plot(t_plot, polyeval(theta10, t_plot),'r')

plt.subplot(2, 2, 4)
plt.scatter(t,y)
plt.plot(t_plot, polyeval(theta15, t_plot),'r')
plt.show()
```

Piecewise-linear fit. In the following code least squares is used to fit a piecewise-linear function to 100 points. It produces Figure 13.5, which is similar to Figure 13.8 in VMLS.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
```

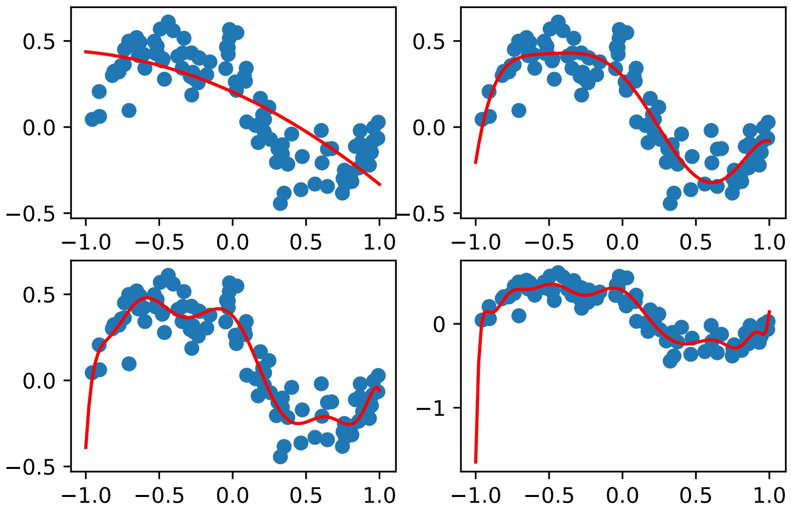


Figure 13.4.: Least squares polynomial fits of degree 1,6,10 and 15 to 100 points.

```
#Generate random data
m = 100
x = -2 + 4*np.random.random(m)
y = 1 + 2*(x-1) - 3*np.maximum(x+1,0) + 4*np.maximum(x-1, 0) +
↪  0.3*np.random.normal(size = m)
#least square fitting
theta = solve_via_backsub(np.column_stack((np.ones(m), x,
↪  np.maximum(x + 1, 0), np.maximum(x-1, 0))), y)
#plot result
t = np.array([-2.1, -1, 1, 2.1])
yhat = theta[0] + theta[1]*t + theta[2]*np.maximum(t+1,0) +
↪  theta[3]*np.maximum(t-1,0)
plt.scatter(x,y)
plt.plot(t, yhat, 'r')
plt.show()
```

House price regression. We calculate the simple regression model for predicting house sales price from area and number of bedrooms, using data of 774 house sales in Sacramento.

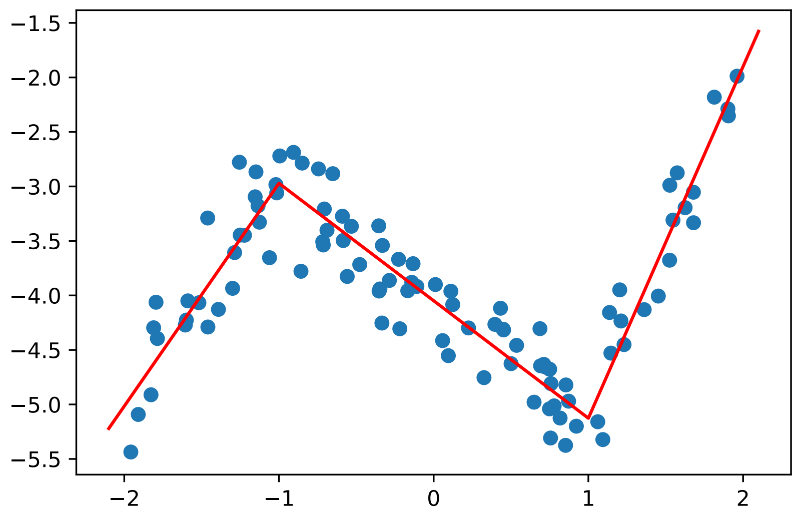


Figure 13.5.: Piecewise-linear fit to 100 points.

In [ ]:
```python
D = house_sales_data()
area = D["area"]
beds = D["beds"]
price = D["price"]
m = len(price)
A = np.column_stack((np.ones(m), area, beds))
x = solve_via_backsub(A, price)
rms_error = (sum((price - A @ x)**2)/len(price))**0.5
std_price = np.std(price)
print(rms_error)
print(std_price)
```

```
74.84571649590141
112.78216159756509
```

Auto-regressive time series model. In the following Python code, we fit an auto-regressive model to the temperature time series discussed on page 259 of VMLS. In Figure 13.6 we compare the first five days of the model predictions with the data.

In [ ]:
```python
#import time series of temperatures t
t = temperature_data()
```

```
N = len(t)
#Standard deviation
np.std(t)
```

Out[ ]:
```
3.0505592856293298
```

In [ ]:
```
#RMS error for simple predictor zhat_{t+1} = z_t
(sum((t[1:] - t[:-1])**2)/(N-1))**0.5
```

Out[ ]:
```
1.1602431638206123
```

In [ ]:
```
#RMS error for simple predictor zhat_{t+1} = z_{t-23}
(sum((t[24:] - t[:-24])**2)/(N-24))**0.5
```

Out[ ]:
```
1.733894140046875
```

In [ ]:
```
#Least squares fit of AR predictor with memory 8
M = 8
y = t[M:]
A = np.column_stack([t[i:i+N-M] for i in reversed(range(M))])
theta = solve_via_backsub(A,y)
ypred = A @ theta
#RMS error of LS AR fit
(sum((ypred-y)**2)/len(y))**0.5
```

Out[ ]:
```
1.0129632612687502
```

In [ ]:
```
#Plot first five days
import matplotlib.pyplot as plt
plt.ion()
Nplot = 24*5
plt.scatter(np.arange(Nplot), t[0:Nplot])
plt.plot(np.arange(M,Nplot), ypred[:Nplot-M], 'r')
plt.show()
```

13.2. Validation

Polynomial approximation. We return to the polynomial fitting example of page 106. We continue with the data vectors t and y in the code on page 106 as the training set, and generate a test set of 100 randomly chosen points generated by the same method

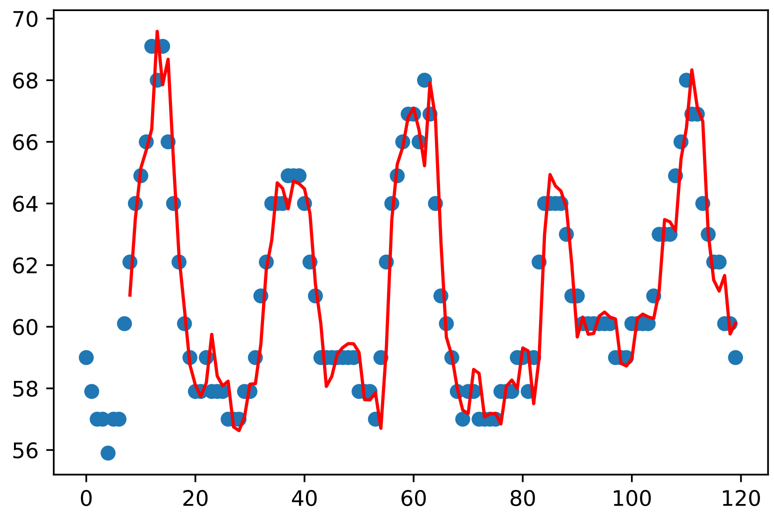


Figure 13.6.: Hourly temperature at Los Angeles International Airport between 12:53AM on May 1, 2016, and 11:53PM on May 5, 2016, shown as circles. The solid line is the prediction of an auto-regressive model with eight coefficients.

as used for the training set. We then fit polynomials of degree $0, \ldots, 20$ (i.e., with $p = 1, \ldots, 21$ coefficients) and compute the RMS error on the training set and the test set. This produces a figure similar to Figure 13.11 in VMLS, shown here as Figure 13.7.

In [ ]:
```
m = 100
# Generate the test set.
t = -1 + 2*np.random.random(m)
y = np.power(t,3) - t + 0.4 / (1 + 25*np.power(t,2)) +
↪  0.10*np.random.normal(size = m)
t_test = -1 + 2*np.random.random(m)
y_test = np.power(t_test,3) - t_test + 0.4/(1 +
↪  np.power(25*t_test,2)) + 0.10*np.random.normal(size = m)
error_train = np.zeros(21)
error_test = np.zeros(21)
for p in range(1,22):
    A = vandermonde(t,p)
    theta = solve_via_backsub(A,y)
    error_train[p-1] = np.linalg.norm(A @ theta -
    ↪  y)/np.linalg.norm(y)
    error_test[p-1] = np.linalg.norm(vandermonde(t_test,p) @ theta
    ↪  - y_test)/np.linalg.norm(y_test)

import matplotlib.pyplot as plt
plt.ion()
plt.plot(np.arange(21), error_train, 'b-o', label = 'Train')
plt.plot(np.arange(21), error_test, 'r-x', label = 'Test')
plt.xlabel('Degree')
plt.ylabel('Relative RMS error')
plt.show()
```

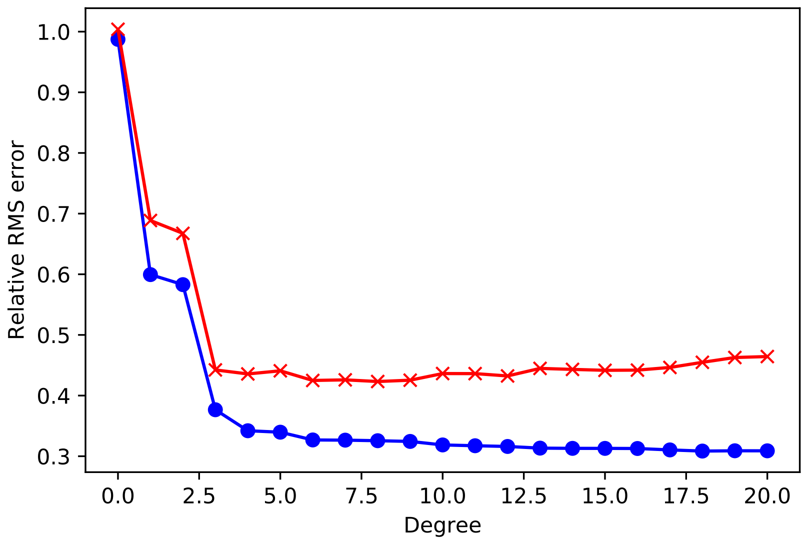


Figure 13.7.: RMS error versus polynomial degree for the fitting examples in figure 13.4.

House price regression model. On page 108 we used a data set of 774 house sales data to fit a simple regression model

$$\hat{y} = \nu + \beta_1 x_1 + \beta_2 x_2,$$

where $\hat{y}$ is the predicted sales price, x_1 is the area, and x_2 is the number of bedrooms. Here we apply cross-validation to assess the generalisation ability of the simple model. We use five folds, four of size 155 (`Nfold` in the code below) and one of size 154. To choose the five folds, we create a random permutation of the indices $1, \ldots, 774$. (We do this by calling the `np.random.permutation` function and the `np.array_split` function.) We choose the data points indexes by the first 155 elements in the permuted list as fold 1, the next 155 as fold 2, et cetera. The output of the following code outputs is similar to Table 13.1 in VMLS (with different numbers because of the random choice of folds).

```
In [ ]: D = house_sales_data()
        price = D['price']
        area = D['area']
        beds = D['beds']
        N = len(price)
        X = np.column_stack([np.ones(N), area, beds])
        I = np.array_split(np.random.permutation(N),5)
        nfold = np.floor(N/5)
        coeff = np.zeros((5,3))
        rms_train = np.zeros(5)
        rms_test = np.zeros(5)
        for k in range(5):
            Itest = I[k]
            Itrain = np.concatenate((np.delete(I,k)))
            Ntrain = len(Itrain)
            Ntest = len(Itest)
            theta = solve_via_backsub(X[Itrain], price[Itrain])
            coeff[k,:]  = theta
            rms_train[k] = (sum((X[Itrain] @ theta -
            ↪  price[Itrain])**2)/N)**0.5
            rms_test[k] = (sum((X[Itest] @ theta -
            ↪  price[Itest])**2)/N)**0.5

        # 3 coefficients for the five folds
        coeff
```

```
Out[ ]: array([[ 50.24831806, 153.00231387, -19.53167009],
               [ 52.07612052, 150.87670328, -19.25688475],
               [ 48.97172712, 147.21796832, -16.87045855],
               [ 59.47192274, 143.46388968, -17.6308993 ],
               [ 61.47688681, 149.73876065, -21.35382905]])
```

```
In [ ]: # RMS errors for five folds
        print(np.column_stack((rms_train, rms_test)))
```

```
[[67.81867582 31.75300852]
 [65.42979442 36.36228512]
 [67.67144708 32.02715856]
 [66.38289164 34.68626333]
 [67.26065478 32.87790231]]
```

Alternatively, one may consider using the `sklearn.model_selection.KFold` function in the `scikit-learn` package[1]. Please refer to the package documentation for the syntax of the `scikit-learn` package.

Validating time series predictions. In the next example, we return the AR model of hourly temperatures at LAX. We divide the time series in a training set of 24 days and a test set of 7 days. We fit the AR model to the training set and calculate the RMS prediction errors on the training and test sets. Figure 13.8 shows the model predictions and the first five days of the test set.

In [ ]:
```
# import time series of temperatures t
t = temperature_data()
N = len(t)
# use first 24 days as training set
Ntrain = 24*24
t_train = t[:Ntrain]
Ntest = N-Ntrain
t_test = t[Ntrain:]
# Least squares fit of AR predictor with memory 8
M = 8
m = Ntrain - M
y = t_train[M:M+m]
A = np.column_stack([t[i:i+m] for i in reversed(range(M))])
coeff = solve_via_backsub(A,y)
rms_train = (sum((A @ coeff-y)**2)/len(y))**0.5
print(rms_train)
```

```
1.0253577259862339
```

In [ ]:
```
ytest = t_test[M:]
mtest = len(ytest)
```

[1]The scikit-learn package is a comprehensive machine learning package in Python. It offers a wide range of simple and efficient tools for data mining and data analysis

```
ypred = np.column_stack([t_test[i:i+mtest] for i in
↪  reversed(range(M))]) @ coeff
rms_test = (sum((ypred-ytest)**2)/len(ytest))**0.5
print(rms_test)
```

```
0.9755113632201694
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
Nplot = 24*5
plt.scatter(np.arange(Nplot), t_test[np.arange(Nplot)])
plt.plot(np.arange(M,Nplot), ypred[np.arange(Nplot-M)],'r')
plt.show()
```

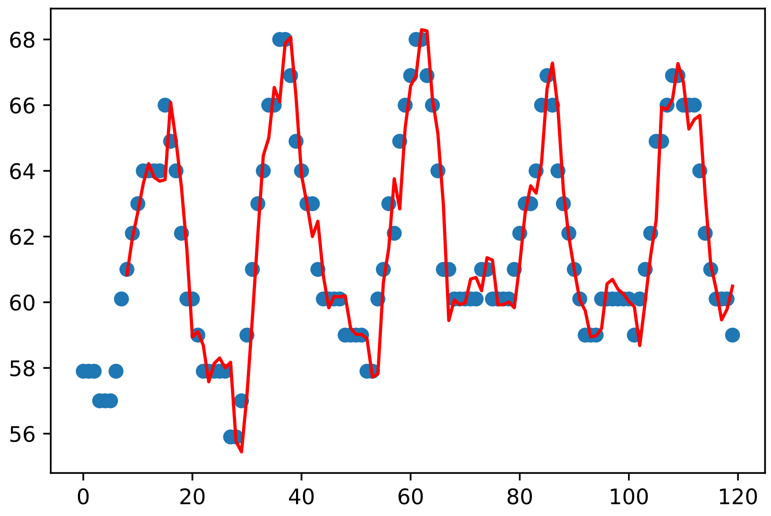


Figure 13.8.: Hourly temperature at Los Angeles International Airport between 12:53AM on May 26, 2016, and 11:53PM on May 29, 2016, shown as circles. The solid line is the prediction of an auto-regressive model with eight coefficients, developed using training data from May 1 to May 24.

13.3. Feature engineering

Next, we compute the more complicated house price regression model of section 13.3.5 of VMLS. The data are imported via the function `house_sales_data`, which returns a dictionary containing the following five vectors of length 774. Details of the data set can be found in Appendix B.

The code below computes the model and makes a scatter plot of actual and predicted prices (Figure 13.9). Note that the last three columns of the matrix `X` contain Boolean variables (`true` or `false`). We rely on the fact that Python treats this as integers 1 and 0.

In [ ]:
```
D = house_sales_data()
price = D['price']
area = D['area']
beds = D['beds']
condo = D['condo']
location = D['location']
N = len(price)
X = np.column_stack([np.ones(N), area, np.maximum(area-1.5,0),
↪  beds, condo, location==2, location==3, location==4])
theta = solve_via_backsub(X,price)
theta
```

Out[ ]:
```
array([ 115.61682367,  175.41314064,  -42.74776797,  -17.87835524,
        -19.04472565, -100.91050309, -108.79112222,
        ↪  -24.76524735])
```

In [ ]:
```
# RMS prediction error
(sum((X @ theta - price)**2)/N)**0.5
```

Out[ ]:
```
68.34428699036884
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
plt.scatter(price, X @ theta)
plt.plot([0,800],[0,800],'r--')
plt.ylim(0,800)
plt.xlim(0,800)
plt.xlabel('Actual price')
plt.ylabel('Predicted price')
plt.show()
```

We finish by a cross-validation of this method. We following the same approach as for the simple regression model on page 112, using five randomly chosen folds. The code shows the eight coefficients, and the RMS training and test errors for each fold.

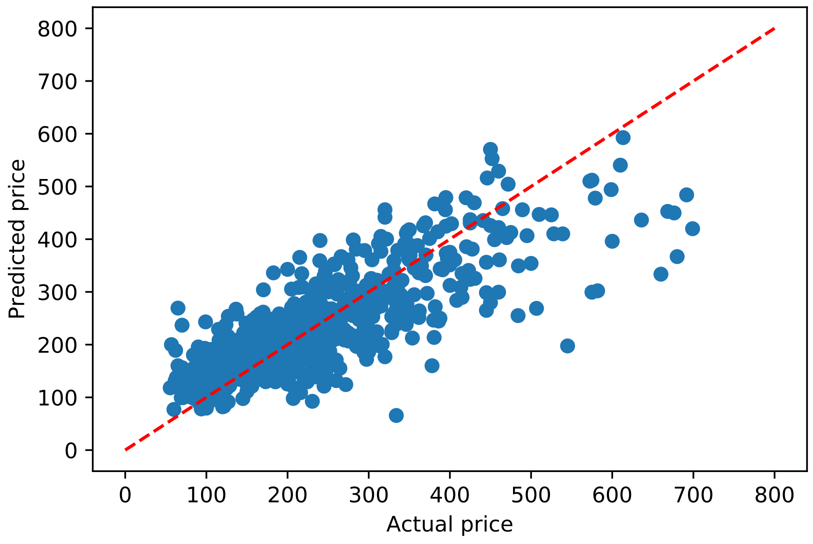


Figure 13.9.: Scatter plot of actual and predicted prices for a model with eight parameters.

In [ ]:
```
I = np.array_split(np.random.permutation(N),5)
nfold = np.floor(N/5)
# store 8 coefficients for the 5 models
models = np.zeros((5,8))
rms_train = np.zeros(5)
rms_test = np.zeros(5)
for k in range(5):
    Itest = I[k]
    Itrain = np.concatenate((np.delete(I,k)))
    Ntrain = len(Itrain)
    Ntest = len(Itest)
    theta = solve_via_backsub(X[Itrain], price[Itrain])
    models[k,:]  = theta
    rms_train[k] = (sum((X[Itrain] @ theta -
    ↪  price[Itrain])**2)/N)**0.5
    rms_test[k] = (sum((X[Itest] @ theta -
    ↪  price[Itest])**2)/N)**0.5

# display the 8 coefficients for each of the 5 folds
models
```

Out[ ]:
```
array([[ 102.3402802 ,  181.97752835,  -46.2691896 ,
↪  -20.68491222,
         -16.84462998,  -89.58550776,  -95.62153542,
          ↪  -13.20162517],
       [ 122.99137921,  165.21259743,  -37.15617807,
       ↪  -15.95828304,
         -23.12481482, -102.1544885 , -108.39142253,
          ↪  -21.40599881],
       [ 137.61858093,  181.79833222,  -47.39666676,
       ↪  -20.06847113,
         -19.71503497, -122.08026771, -132.99656753,
          ↪  -47.69513389],
       [ 109.96492671,  161.36369407,  -35.27910343,
       ↪  -9.40327285,
         -12.03343019, -101.12023823, -110.4278418 ,
          ↪  -33.38553974],
       [ 105.39525443,  187.93825777,  -48.73041524,
       ↪  -24.05769035,
         -24.38599371,  -88.87911655,  -95.726103  ,
          ↪  -6.69225326]])
```

In [ ]:
```python
# display training errors
print(rms_train)
# display testing errors
print(rms_test)
```

```
[61.42022599 62.33572416 62.17139464 60.7368078  58.27960703]
[30.13525437 28.21601462 28.8141013  31.78478227 36.0136247 ]
```

14. Least squares classification

14.1. Classification

Boolean values. Python has the Boolean values `true` and `false`. These are automatically converted to the numbers 1 and 0 when they are used in numerical expression. In VMLS we use the encoding (for classifiers) where True corresponds to +1 and False corresponds to −1. We can get our encoding from a Python Boolean value `b` using `2*b-1`.

```
In [ ]: tf2pm1 = lambda b: 2*b-1
        b = True
        tf2pm1(b)

Out[ ]: 1

In [ ]: b = np.array([True, False, True])
        tf2pm1(b)

Out[ ]: array([ 1, -1,  1])
```

Confusion matrix. Let's see how we would evaluate the prediction errors and confusion matrix, given a set of data `y` and predictions `yhat`, both stores as arrays (vectors) of Boolean values, of length `N`.

```
In [ ]: # Count errors and correct predictions
        Ntp = lambda y,yhat: sum((y == True)&(yhat == True))
        Nfn = lambda y,yhat: sum((y == True)&(yhat == False))
        Nfp = lambda y,yhat: sum((y == False)&(yhat == True))
        Ntn = lambda y,yhat: sum((y == False)&(yhat == False))
        error_rate = lambda y,yhat: (Nfn(y,yhat)+Nfp(y,yhat))/len(y)
        confusion_matrix = lambda y,yhat: np.block([[Ntp(y,yhat),
        ↪  Nfn(y,yhat)], [Nfp(y,yhat), Ntn(y,yhat)]])

        y = np.random.randint(2, size = 100)
```

```
yhat = np.random.randint(2, size = 100)
confusion_matrix(y,yhat)
```

```
Out[ ]: array([[23, 25],
               [31, 21]])
```

```
In [ ]: error_rate(y,yhat)
```

```
Out[ ]: 0.56
```

When we sum the Boolean vectors, they are converted to integers. In the last section of the code, we generate two random Boolean vectors, so we expect the error rate to be around 50%. In the code above, we compute the error rate from the numbers of false negatives and false positives.

Alternatively, one can compute the confusion matrix using the `confusion_matrix` function from the `sklearn` package. Note that the entries in the `sklearn` confusion matrix are reversed relative to the definition in VMLS.

```
In [ ]: from sklearn.metrics import confusion_matrix
        confusion_matrix(y,yhat)
```

```
Out[ ]: array([[21, 31],
               [25, 23]])
```

14.2. Least squares classifier

We can evaluate $f(\hat{x}) = \textbf{sign}(\tilde{f}(x))$ using `ftilde(x)>0`, which returns a Boolean value.

```
In [ ]: #Regression model
        ftilde = lambda x: x @ beta + v
        #Regression classifier
        fhat = lambda x: ftilde(x) > 0
```

Iris flower classification. The Iris data set contains 150 examples of three types of iris flowers. There are 50 examples of each class. For each example, four features are provided. The following code reads in a dictionary containing three 50×4 matrices `setosa, versicolor, virginica` with the examples for each class, and then computes a Boolean classifier that distinguishes *Iris Virginica* from the other two classes. We also use the `confusion_matrix` function and `error_rate` function defined above.

```
In [ ]: D = iris_data()
        # Create 150 by 4 data matrix
        iris = np.vstack([D['setosa'], D['versicolor'], D['virginica']])
        # y[k] is true (1) if virginica, false (-1) otherwise
        y = np.concatenate([np.zeros(100), np.ones(50)])
        A = np.column_stack([np.ones(150), iris])
        theta = solve_via_backsub(A, 2*y-1)
        theta

Out[ ]: array([-2.39056373, -0.09175217,  0.40553677,  0.00797582,
        ↪  1.10355865])

In [ ]: yhat = A @ theta > 0
        C = confusion_matrix(y,yhat)
        C

Out[ ]: array([[46,  4],
               [ 7, 93]])

In [ ]: error_rate(y,yhat)

Out[ ]: 0.07333333333333333

In [ ]: np.average(y != yhat)

Out[ ]: 0.07333333333333333
```

14.3. Multi-class classifiers

Multi-class error rate and confusion matrix. The overall error rate is easily evaluated `np.average(y!=yhat)`. We can form the $K \times K$ confusion matrix from a set of N true outcomes y and N predictions `yhat` (each with entries among $\{1, \ldots K\}$) by counting the number of times each pair of values occurs.

```
In [ ]: error_rate = lambda y,yhat: np.average(y != yhat)
        def confusion_matrix(y, yhat, K):
            C = np.zeros((K,K))
            for i in range(K):
                for j in range(K):
                    C[i,j] = sum((y == i+1) & (yhat == j+1))
            return C
```

```
# test for K = 4 on random vectors of length 100
K = 4
y = np.random.randint(1, K+1, size = 100)
yhat = np.random.randint(1, K+1, size = 100)
C = confusion_matrix(y,yhat,K)
print(C)
```

```
[[ 6.,  4.,  9., 11.],
 [ 4.,  3.,  8.,  4.],
 [ 9.,  8.,  7.,  5.],
 [ 3.,  9.,  5.,  5.]]
```

```
In [ ]: error_rate(y,yhat), 1 - sum(np.diag(C))/np.sum(C)
```

```
Out[ ]: (0.79, 0.79)
```

Least squares multi-class classifier. A K-class classifier (with regression model) can be expressed as

$$\hat{f}(x) = \underset{k=1,\ldots,K}{\operatorname{argmax}} \tilde{f}_k(x),$$

where $\tilde{f}_k(x) = x^T\theta_k$. The n-vector $\theta_1, \ldots, \theta_K$ are the coefficients or parameters in the model. We can express this in matrix-vector notation as

$$\hat{f}(x) = \operatorname{argmax}(x^T\Theta),$$

where $\Theta = [\theta_1 \ldots \theta_K]$ is the $n \times K$ matrix of model coefficients, and the argmax of a row vector has the obvious meaning.

Let's see how to express this in Python. In Python this numpy function `np.argmax(u)` finds the index of the largest entry in the row or column vector `u`, i.e., $\operatorname{argmax}_k u_k$. To extend this to matrices, we define a function `row_argmax` that returns a vector with, for each row, the index of the largest entry in that row.

```
In [ ]: row_argmax = lambda u: [np.argmax(u[i,:]) for i in range(len(u))]
        A = np.random.normal(size = (4,5))
        A
```

```
Out[ ]: array([[-0.46389523,  0.49276966, -2.91137672, -0.91803154,
        ↪   -0.74178812],
        [-0.65697642, -0.64776259, -0.84059464, -2.74870103,
        ↪  -0.31462641],
        [ 0.62005913,  0.70966526,  0.54062214, -0.50133417,
        ↪  0.00602328],
        [ 0.5946104 ,  0.88064277, -0.18461465, -1.44861368,
        ↪  -0.15476018]])
```

In [ ]:
```
row_argmax(A)
```

Out[ ]:
```
[1, 4, 1, 1]
```

If a data set with N examples is stored as an $n \times N$ data matrix `X`, and `Theta` is an $n \times K$ matrix with the coefficient vectors θ_k, as its columns, then we can now define a function

In [ ]:
```
fhat = lambda X, Theta: row_argmax(X @ Theta)
```

to find the N-vector of predictions.

Matrix least squares. Let's use least squares to find the coefficient matrix Θ for a multi-class classifier with n features and K classes, from a data set of N examples. We will assume the data is given as an $n \times N$ matrix X and an N-vector y^{cl} with entries in $\{1, \ldots, K\}$ that give the classes of the examples. The least squares objective can be expressed as a matrix norm squared,

$$\|X^T\Theta - Y\|^2,$$

where Y is the $N \times K$ vector with

$$Y_{ij} = \begin{cases} 1 & y_i^{\mathrm{cl}} = j \\ -1 & y_i^{\mathrm{cl}} \neq j \end{cases}$$

In other words, the rows of Y describe the classes using one-hot encoding, converted from $0/1$ to $-1/+1$ values. The least squares solution is given by $\hat{\Theta} = (X^T)^\dagger Y$.

Let's see how to express this in Python.

In [ ]:
```
def one_hot(ycl, K):
    N = len(ycl)
    Y = np.zeros((N,K))
    for j in range(K):
        Y[np.where(ycl == j),j] = 1
    return Y
K = 4
ycl = np.random.randint(K, size = 6)
Y = one_hot(ycl,K)
Y
```

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

```
In [ ]: 2*Y - 1
```

```
Out[ ]: array([[ 1., -1., -1., -1.],
               [-1., -1.,  1., -1.],
               [-1., -1.,  1., -1.],
               [-1., -1., -1.,  1.],
               [-1., -1.,  1., -1.],
               [-1.,  1., -1., -1.]])
```

Using the function we have defined, the matrix least squares multi-class classifier can be computed in a few lines.

```
In [ ]: def ls_multiclass(X, ycl, K):
            n, N = X.shape
            Theta = solve_via_backsub(X.T, 2*one_hot(ycl,K) - 1)
            yhat = 1 + row_argmax(X @ theta)
            return Theta, yhat
```

Iris flower classification. We compute a 3-class classifier for the iris flower data set. We split the data set of 150 examples in a training set of 120 (40 per class) and a test set of 30 (10 per class). The code calls the functions we defined above.

```
In [ ]: D = iris_data()
        setosa = np.array(D['setosa'])
        versicolor = np.array(D['versicolor'])
        virginica = np.array(D['virginica'])
```

```
# pick three random permutations of 1,...,50
I1 = np.random.permutation(50)
I2 = np.random.permutation(50)
I3 = np.random.permutation(50)
# training set is 40 randomly picked examples per class
Xtrain = np.vstack([setosa[I1[:40],:], versicolor[I2[:40],:],
↪  virginica[I3[:40],:]]).T
# add contant feature one
Xtrain = np.vstack([np.ones(120), Xtrain])
# the true labels for train set are a sequence of 1s, 2s and 3s
# since the examples in Xtrain are stacked in order
ytrain = np.hstack([np.ones(40), 2*np.ones(40), 3*np.ones(40)])
# test set is remaining 10 examples for each class
Xtest = np.vstack([setosa[I1[40:],:], versicolor[I2[40:],:],
↪  virginica[I3[40:],:]]).T
Xtest = np.vstack([np.ones(30), Xtest])
ytest = np.hstack([np.ones(10), 2*np.ones(10), 3*np.ones(10)])
Theta, yhat = ls_multiclass(Xtrain, ytrain, 3)
Ctrain = confusion_matrix(ytrain, yhat, 3)
print(Ctrain)
```

```
[[39.,  1.,  0.],
 [ 0., 28., 12.],
 [ 0.,  6., 34.]]
```

In [ ]:
```
error_train = error_rate(ytrain, yhat)
print(error_train)
```

```
0.15833333333333333
```

In [ ]:
```
yhat = row_argmax(Xtest.T @ Theta.T) + 1
Ctest = confusion_matrix(ytest, yhat, 3)
print(Ctest)
```

```
[[10.,  0.,  0.],
 [ 0.,  5.,  5.],
 [ 0.,  2.,  8.]]
```

In [ ]:
```
error_test = error_rate(ytest, yhat)
print(error_test)
```

```
0.23333333333333334
```

15. Multi-objective least squares

15.1. Multi-objective least squares

Let's write a function that solves the multi-objective least squares problems, with given positive weights. The data are a list (or array) of coefficient matrices (of possibly different heights) `As`, a matching list of (right-hand side) vectors `bs`, and the weights, given as an array or list, `lambdas`.

```
In [ ]: def mols_solve(As, bs, lambdas):
            k = len(lambdas)
            Atil = np.vstack([np.sqrt(lambdas[i])*As[i] for i in
            ↪  range(k)])
            btil = np.hstack([np.sqrt(lambdas[i])*bs[i] for i in
            ↪  range(k)])
            return solve_via_backsub(Atil,btil)
```

Simple example. We use the function `mols_solve` to work out a bi-criterion example similar to Figures 15.1, 15.2 and 15.3 in VMLS. We minimized the weighted sum objective

$$J_1 + \lambda J_2 = \|A_1x - b_1\|^2 + \lambda\|A_2x - b_2\|^2$$

for randomly chosen 10×5 matrices A_1, A_2 and 10-vectors b_1, b_2.

```
In [ ]: As = np.array([np.random.normal(size = (10,5)),
        np.random.normal(size = (10,5))])

        bs = np.vstack([np.random.normal(size = 10),
        np.random.normal(size = 10)])

        N = 200
        lambdas = np.power(10, np.linspace(-4, 4, 200))
        x = np.zeros((5,N))
        J1 = np.zeros(N)
```

```
J2 = np.zeros(N)
for k in range(N):
    x[:,k] = mols_solve(As, bs, [1, lambdas[k]])
    J1[k] = np.linalg.norm(As[0]@x[:,k] - bs[0])**2
    J2[k] = np.linalg.norm(As[1]@x[:,k] - bs[1])**2

import matplotlib.pyplot as plt
plt.ion()

# plot solution versus lambda
plt.plot(lambdas, x.T)
plt.xscale('log')
plt.xlabel('lambda')
plt.xlim((1e-4, 1e+4))
plt.legend(['y1','y2','y3','y4','y5'], loc='upper right')
plt.show()

# plot two objectives versus lambda
plt.plot(lambdas, J1)
plt.plot(lambdas, J2)
plt.xscale('log')
plt.xlabel('lambda')
plt.xlim((1e-4, 1e+4))
plt.legend(['J1', 'J2'])
plt.show()

# plot tradeoff curve
plt.plot(J1,J2)
plt.xlabel('J1')
plt.ylabel('J2')

# add (single objective) end points to trade-off curve
x1 = solve_via_backsub(As[0], bs[0])
x2 = solve_via_backsub(As[1], bs[1])
J1 = [np.linalg.norm(As[0]@x1 - bs[0])**2,
np.linalg.norm(As[0]@x2 - bs[0])**2]
J2 = [np.linalg.norm(As[1]@x1 - bs[1])**2,
np.linalg.norm(As[1]@x2 - bs[1])**2]
plt.scatter(J1, J2)
plt.show()
```

The expression `lambdas = np.power(10, np.linspace(-4,4,200))` generates 200 values of $\lambda \in [10^{-4}, 10^{4}]$, equally spaced on a logarithmic scale. Our code generates the three plots in Figures 15.1, 15.2 and 15.3.

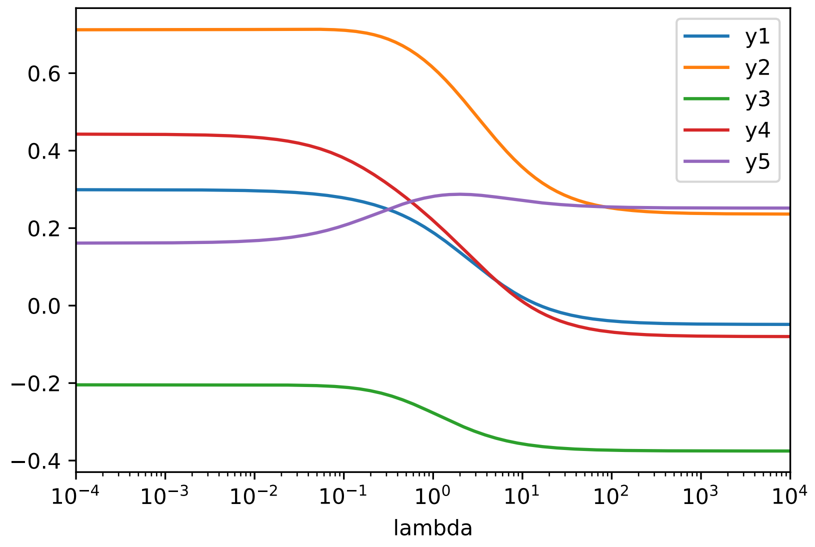


Figure 15.1.: Weighted-sum least squares solution $\hat{x}(\lambda)$ as a function of λ for a bi-criterion least squares problem with five variables.

15.2. Control

15.3. Estimation and inversion

Estimating a periodic time series. We consider the example of Figure 15.4 in VMLS. We start by loading the data, as a vector with hourly ozone levels, for a period of 14 days. Missing measurements have a value `NaN` (for Not a Number). The `matplotlib.pyplot` package skips those values (Figure 15.4).

In [ ]:
```
ozone = ozone_data() # a vector of length 14*24 = 336
k = 14
N = k*24
import matplotlib.pyplot as plt
plt.ion()
plt.plot(np.arange(N), ozone, 'o-')
plt.yscale('log')
```

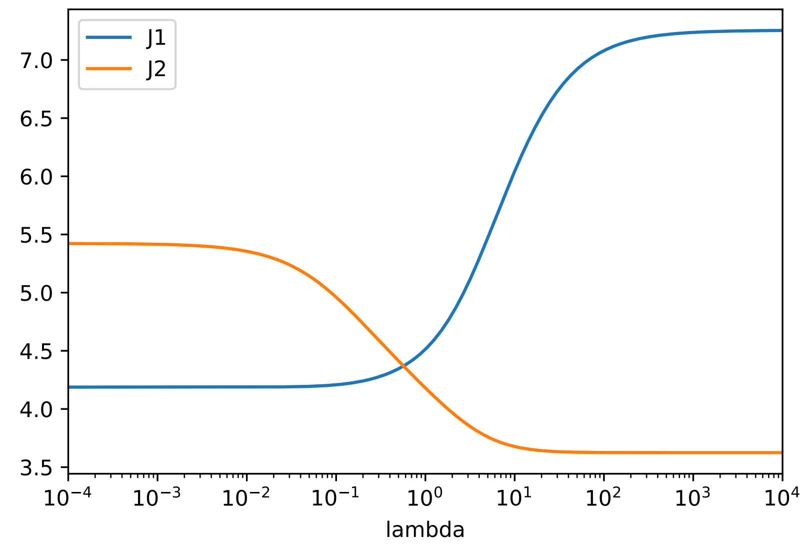


Figure 15.2.: Objective function $J_1 = \|A_1\hat{x}(\lambda)b_1\|^2$ (blue line) and $J_2 = \|A_2\hat{x}(\lambda)b_2\|^2$ (red line) as functions of λ for the bi-criterion problem in figure 15.1.

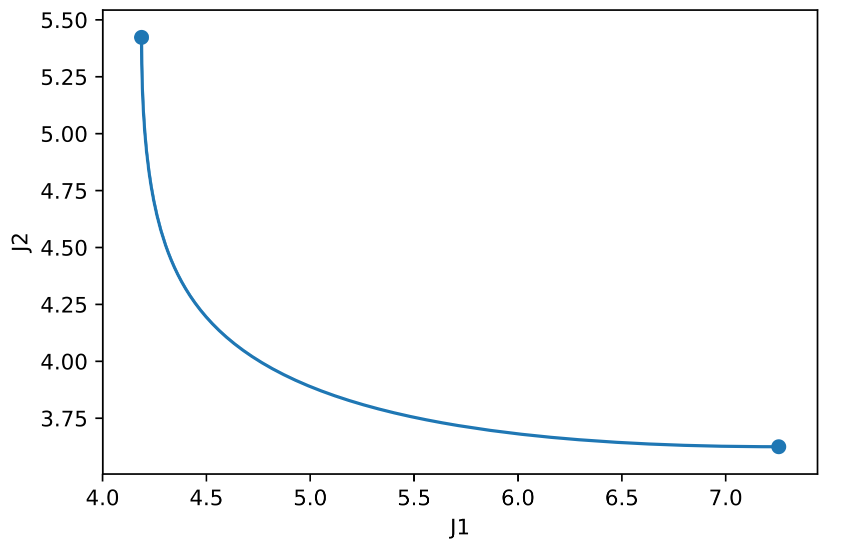


Figure 15.3.: Optimal trade-off curves for the bi-criterion least squares problem of figures 15.1 and 15.2.

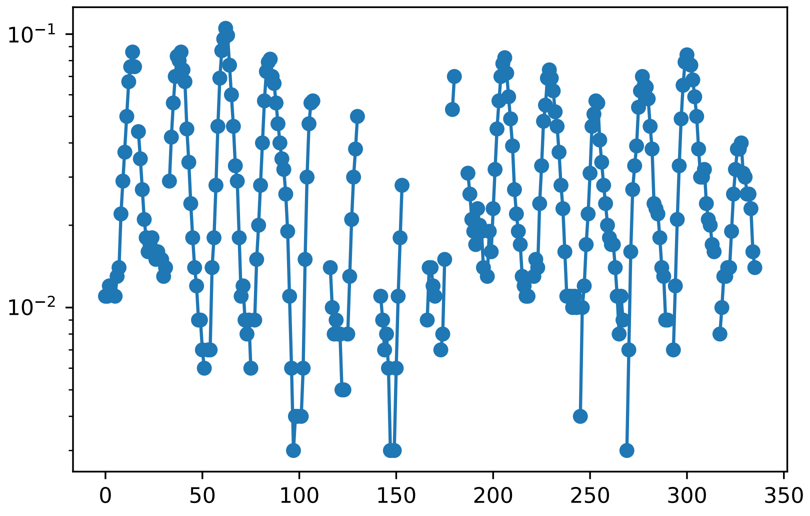


Figure 15.4.: Hourly ozone level at Azusa, California, during the first 14 days of July 2014 (California Environmental Protection Agency, Air Resources Board, www.arb.ca.gov). Measurements start at 12AM on July 1st, and end at 11PM on July 14. Note the large number of missing measurements. In particular, all 4AM measurements are missing.

Next, we make a periodic fit for the values $\lambda = 1$ and $\lambda = 100$. The numpy function **np.isnan** is used to find and discard the missing measurement. The results are shown in Figures 15.5 and 15.6.

```
In [ ]: A = np.vstack([np.eye(24) for i in range(k)])
        #periodic difference matrix
        D = -np.eye(24) + np.vstack((np.column_stack([np.zeros(23),
        ↪  np.eye(23)]), np.hstack([1, np.zeros(23)])))
        ind = [k for k in range(len(ozone)) if ~np.isnan(ozone[k])]
        As = [A[ind,:],D]
        bs = [np.log(ozone[ind]), np.zeros(24)]

        # solution for lambda = 1
        lambdas = np.array([1,1])
        n = len(lambdas)
        Atil = np.vstack([np.sqrt(lambdas[i])*As[i] for i in range(n)])
        btil = np.hstack([np.sqrt(lambdas[i])*bs[i] for i in range(n)])
        x = np.linalg.inv(Atil.T @ Atil) @ (Atil.T @ btil)
        plt.scatter(np.arange(N), ozone)
plt.plot(np.arange(N), np.hstack([np.exp(x) for i in
↪  range(k)]),'r')
plt.yscale('Log', basey=10)
plt.ylim(10**(-2.8),10**(-0.8))
plt.show()

# solution for lambda = 100
lambdas = np.array([1,100])
n = len(lambdas)
Atil = np.vstack([np.sqrt(lambdas[i])*As[i] for i in range(n)])
btil = np.hstack([np.sqrt(lambdas[i])*bs[i] for i in range(n)])
x = np.linalg.inv(Atil.T @ Atil) @ (Atil.T @ btil)
plt.scatter(np.arange(N), ozone)
plt.plot(np.arange(N), np.hstack([np.exp(x) for i in
↪  range(k)]),'r')
plt.yscale('Log', basey=10)
plt.ylim(10**(-2.8),10**(-0.8))
plt.show()
```

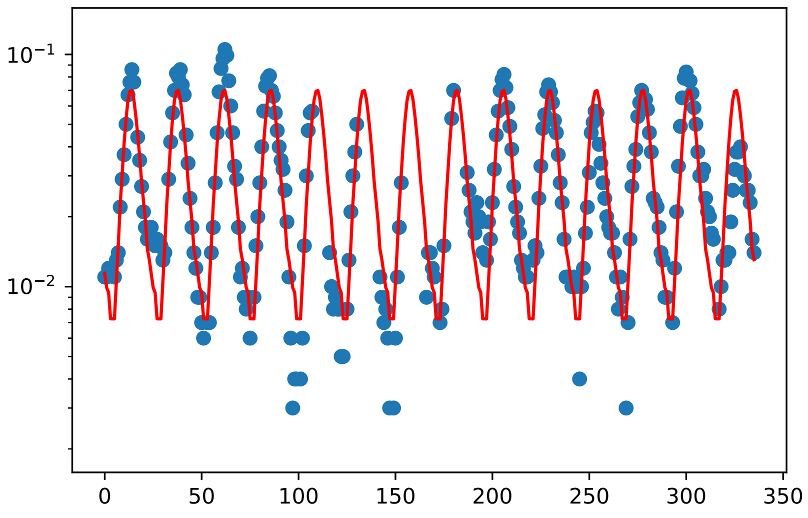


Figure 15.5.: Smooth periodic least squares fit to logarithmically transformed measurements, using $\lambda = 1$.

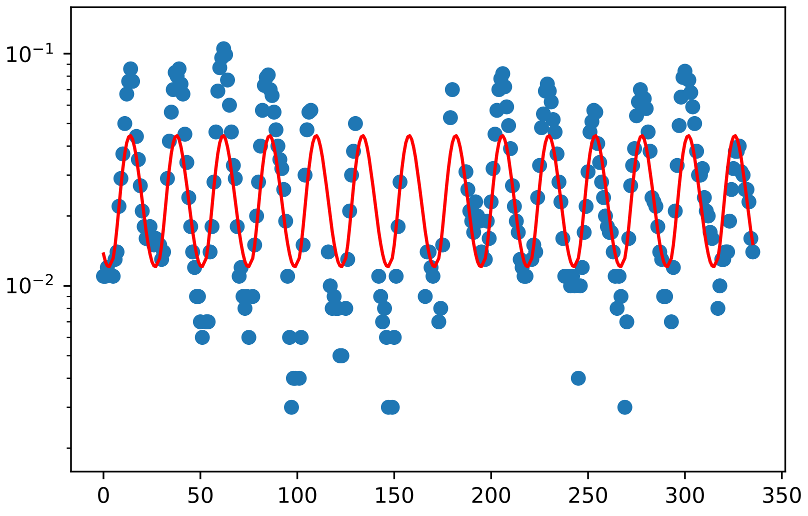


Figure 15.6.: Smooth periodic least squares fit to logarithmically transformed measurements, using $\lambda = 100$.

15.4. Regularized data fitting

Example. Next we consider the small regularised data fitting example of page 329 of VMLS. We fit a model

$$\hat{f}(x) = \sum_{k=1}^{5} \theta_k f_k(x)$$

with basis function $f_1(x) = 1$ and $f_{k+1}(x) = \sin(\omega_k x + \phi_k)$ for $k = 1, \ldots, 4$ to $N = 20$ data points. We use the values of ω_k, ϕ_k given in the text. We fit the model by solving a sequence of regularised least squares problem with objective

$$\sum_{i=1}^{N} \left(y^{(i)} - \sum_{k=1}^{5} \theta_k f_k(x^{(i)}) \right)^2 + \lambda \sum_{k=2}^{5} \theta_k^2,$$

The two plots are shown in Figures 15.7 and 15.8.

```
In [ ]: #Import data as vectors xtrain, ytrain, xtest, ytest
        D = regularized_fit_data()
        xtrain = D['xtrain']
        ytrain = D['ytrain']
        xtest = D['xtest']
        ytest = D['ytest']
```

```
N = len(ytrain)
Ntest = len(ytest)
p = 5
omega = np.array([13.69, 3.55, 23.25, 6.03])
phi = np.array([0.21, 0.02, -1.87, 1.72])
A = np.column_stack((np.ones(N), np.sin(np.outer(xtrain,omega) +
↪  np.outer(np.ones(N),phi))))
Atest = np.column_stack((np.ones(Ntest),
↪  np.sin(np.outer(xtest,omega) + np.outer(np.ones(Ntest),phi))))
npts = 100
lambdas = np.power(10, np.linspace(-6,6,npts))
err_train = np.zeros(npts)
err_test = np.zeros(npts)
thetas = np.zeros((p,npts))
As = [A, np.column_stack([np.zeros(p-1), np.eye(p-1)])]
bs = [ytrain, np.zeros(p-1)]

for k in range(npts):
    lam = [1,lambdas[k]]
    n = len(lam)
    Atil = np.vstack([np.sqrt(lam[i])*As[i] for i in range(n)])
    btil = np.hstack([np.sqrt(lam[i])*bs[i] for i in range(n)])
    theta = np.linalg.inv(Atil.T @ Atil) @ (Atil.T @ btil)
    err_train[k] = (sum((ytrain - A @ theta)**2)/len(ytrain))**0.5
    err_test[k] = (sum((ytest - Atest @
    ↪  theta)**2)/len(ytest))**0.5
    thetas[:,k] = theta

import matplotlib.pyplot as plt
plt.ion()
#plot RMS errors
plt.plot(lambdas, err_train.T)
plt.plot(lambdas, err_test.T)
plt.xscale('Log')
plt.xlabel('lambda')
plt.legend(['Train', 'Test'], loc='upper left')
plt.ylabel('RMS error')
plt.show()
#Plot coefficients
plt.plot(lambdas, thetas.T)
        plt.xscale('Log')
        plt.xlabel('lambda')
        plt.xlim((1e-6, 1e6))
        plt.legend(['y1', 'y2', 'y3', 'y4', 'y5'], loc='upper right')
        plt.show()
```

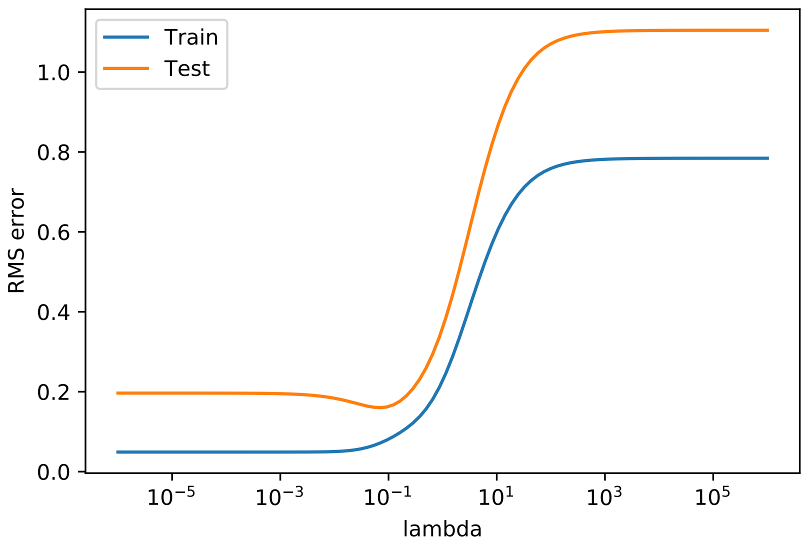


Figure 15.7.: RMS training and testing errors as a function of the regularization parameter λ.

15.5. Complexity

The kernel trick. Let's check the kernel trick, described in section 15.5.2 in VMLS to find $\hat{x}$, the minimizer of

$$\|Ax - b\|^2 + \lambda\|x - x^{\text{des}}\|^2,$$

where A is an $m \times n$ matrix and $\lambda > 0$. We will compute $\hat{x}$ in two ways. First, the naïve way, and then, using the kernel trick. We use the fact that if

$$\begin{bmatrix} A^T \\ \sqrt{\lambda} I \end{bmatrix} = QR,$$

then

$$(AA^T + \lambda I)^{-1} = (R^T Q^T QR)^{-1} = R^{-1}R^{-T}.$$

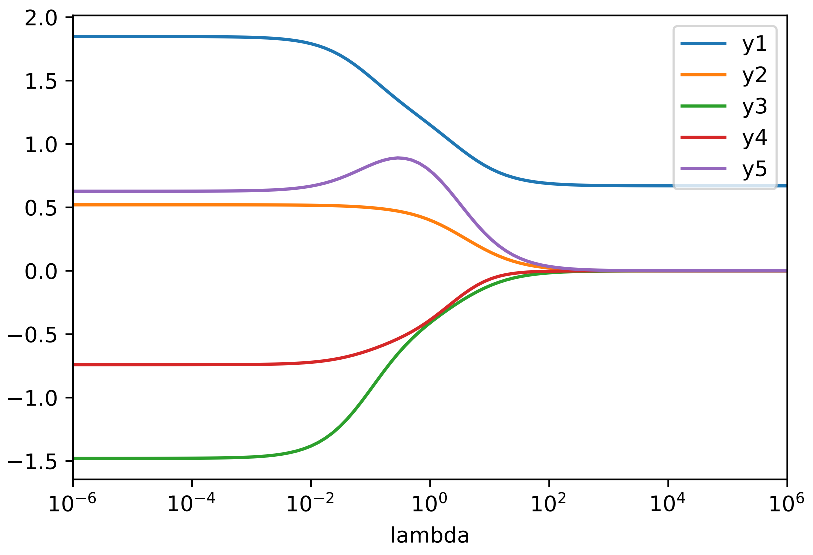


Figure 15.8.: The regularization path.

```
In [ ]: m = 100
        n = 5000
        A = np.random.normal(size = (m,n))
        b = np.random.normal(size = m)
        xdes = np.random.normal(size = n)
        lam = 2.0
        # Find x that minimizes ||Ax - b||^2 + lambda ||x||^2
        import time
        start = time.time()
        xhat1 = solve_via_backsub(np.vstack((A, np.sqrt(lam)*np.eye(n))),
        ↪   np.hstack((b.T, np.sqrt(lam)*xdes)))
        end = time.time()
        print('xhat1 time:', end - start)
```

```
xhat1 time: 419.9836058616638
```

```
In [ ]: # Now use kernel trick
        start = time.time()
        Q, R = QR_factorization(np.vstack((A.T, np.sqrt(lam)*np.eye(m))))
        xhat2 = A.T @ (solve_via_backsub(R, solve_via_backsub(R.T,(b-A @
        ↪   xdes))) ) + xdes
        end = time.time()
```

```
print('xhat2 time:', end - start)
```

```
xhat2 time: 0.22452521324157715
```

```
In [ ]: # compare solutions
        np.linalg.norm(xhat1 - xhat2)
```

```
Out[ ]: 5.361013146339246e-12
```

The naïve method requires the factorization of a 5100×5100 matrix. In the second method we factor a matrix of size 5100×100 and hence the second method is about 2000 times faster!

16. Constrained least squares

16.1. Constrained least squares problem

In the examples in this section, we use the `cls_solve` function, described later in section 16.3, to find the constrained least squares solution.

Piecewise polynomial. We fit a function $\hat{f} : \mathbf{R} \rightarrow \mathbf{R}$ to some given data, where $\hat{f}(x) = p(x)$ for $x \leq a$ and $\hat{f}(x) = q(x)$ for $x > a$, subject to $p(a) = q(a)$ and $p'(a) = q'(a)$, i.e. the two polynomials have matching value and slope at the knot point a. We have data points $x_1, \ldots, x_M \leq a$ and $x_{M+1}, \ldots, x_N > a$ and corresponding values $y_1, \ldots, y_N$. In the example we take $a = 0$, polynomials p and q of degree 3, and $N = 2M = 140$. The code creates a figure similar to Figure 16.1 of VMLS. We use the `vandermonde` function defined on page 58.

```
In [ ]: M = 70
        N = 2*M
        xleft = np.random.random(M) - 1
        xright = np.random.random(M)
        x = np.hstack([xleft, xright])
        y = np.power(x,3) - x + 0.4/(1+25*np.power(x,2)) +
        ↪  0.05*np.random.normal(size = N)
        n = 4
        A = np.vstack([np.hstack([vandermonde(xleft,n), np.zeros((M,n))]),
        ↪  np.hstack([np.zeros((M,n)), vandermonde(xright,n)])])
        b = y
        C = np.vstack((np.hstack([1,np.zeros(n-1), -1, np.zeros(n-1)]),
        ↪  np.hstack([0, 1, np.zeros(n-2), 0, -1, np.zeros(n-2)])))
        d = np.zeros(2)
        theta = cls_solve(A,b,C,d)

        import matplotlib.pyplot as plt
        plt.ion()
        # Evaluate and plot for 200 equidistant points on each side.
```

```
Npl = 200
xpl_left = np.linspace(-1, 0, Npl)
ypl_left = vandermonde(xpl_left, 4) @ theta[:n]
xpl_right = np.linspace(0, 1, Npl)
ypl_right = vandermonde(xpl_right, 4) @ theta[n:]
plt.scatter(x, y)
plt.plot(xpl_left, ypl_left, 'orange')
plt.plot(xpl_right, ypl_right, 'green')
plt.show()
```

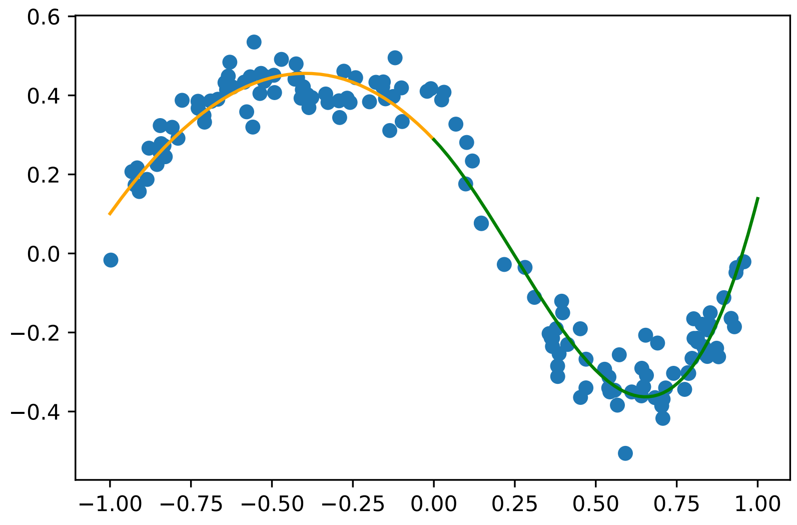


Figure 16.1.: Least squares fit of two cubic polynomials to 140 points, with continuity constraints $p(0) = q(0)$ and $p'(0) = q'(0)$.

Advertising budget. We continue the advertising example of page 98 and add a total budget constraint $\mathbf{1}^T s = 1284$.

```
In [ ]: n = 3
        m = 10
        R = np.array([[0.97,1.86,0.41], [1.23,2.18,0.53],
        ↪  [0.80,1.24,0.62], [1.29,0.98,0.51], [1.10,1.23,0.69],
        ↪  [0.67,0.34,0.54], [0.87,0.26,0.62], [1.10,0.16,0.48],
        ↪  [1.92,0.22,0.71],  [1.29,0.12,0.62]])
        cls_solve(R, 1e3*np.ones(m), np.ones((1,n)), np.array([1284]))
```

```
Out[ ]: array([315.16818459, 109.86643348, 858.96538193])
```

Minimum norm force sequence. We compute the smallest sequence of ten forces, each applied for one second to a unit frictionless mass originally at rest, that moves the mass position one with zero velocity (VMLS page 343).

```
In [ ]: A = np.eye(10)
        b = np.zeros(10)
        C = np.vstack([np.ones((1,10)), np.arange(9.5,0.4,-1).T])
        d = np.array([0,1])
        cls_solve(A, b, C, d)
```

```
Out[ ]: array([ 0.05454545,  0.04242424,  0.03030303,  0.01818182,
        ↪  0.00606061,
               -0.00606061, -0.01818182, -0.03030303, -0.04242424,
               ↪  -0.05454545])
```

16.2. Solution

Let's implement the function `cls_solve_kkt`, which finds the constrained least squares solution by forming the KKT system and solving it. We allow the `b` and `d` to be matrices, so one function call can solve multiple problems with the same A and C.

```
In [ ]: def cls_solve_kkt(A, b, C, d):
            m, n = A.shape
            p, n = C.shape
            #Gram matrix
            G = A.T @ A
            #KKT matrix
            KKT = np.vstack([np.hstack([2*G,C.T]), np.hstack([C,
            ↪  np.zeros((p,p))])])
            xzhat = solve_via_backsub(KKT, np.hstack([2*A.T @ b,d]))
            return xzhat[:n]

        A = np.random.normal(size = (10,5))
        b = np.random.normal(size = 10)
        C = np.random.normal(size = (2,5))
        d = np.random.normal(size = 2)
        x = cls_solve_kkt(A, b, C, d)
```

```
# Check that residual is small
print(C @ x - d)
```

```
[-1.36557432e-14  1.04916076e-14]
```

16.3. Solving constrained least squares problems

Solving constrained least squares via QR. Let's implement VMLS algorithm 16.1 and then check it against our method above, which forms and solves the KKT system.

```
In [ ]: def cls_solve(A, b, C, d):
            m, n = A.shape
            p, n = C.shape
            Q, R = QR_factorization(np.vstack([A,C]))
            Q1 = Q[:m,:]
            Q2 = Q[m:m+p+1,:]
            Qtil, Rtil = QR_factorization(Q2.T)
            w = solve_via_backsub(Rtil, (2*Qtil.T @ (Q1.T @ b) -
             ↪  2*solve_via_backsub(Rtil.T, d)))
            xhat = solve_via_backsub(R, (Q1.T @ b - Q2.T @ w/2))
            return xhat

        # compare with KKT method
        m = 10
        n = 5
        p = 2
        A = np.random.normal(size = (m,n))
        b = np.random.normal(size = m)
        C = np.random.normal(size = (p,n))
        d = np.random.normal(size = p)
        xKKT = cls_solve_kkt(A, b, C, d)
        xQR = cls_solve(A, b, C, d)
        # compare solutions
        print(np.linalg.norm(xKKT - xQR))
```

```
5.0005888865147664e-15
```

Sparse constrained least squares. Let's form and solve the system of linear equations in VMLS (16.11), and compare it to our basic method for constrained least squares.

This formulation will result in a sparse set of equations to solve if A and C are sparse. (The code below jsut checks that the two methods agree; it does not use sparsity. Unlike the earlier `cls_solve`, it assumes `b` and `d` are vectors.)

```
In [ ]: def cls_solve_sparse(A, b, C, d):
            m, n = A.shape
            p, n = C.shape
            bigA = np.vstack([np.hstack([np.zeros((n,n)),A.T,C.T]),
             ↪  np.hstack([A, -np.eye(m)/2,np.zeros((m,p))]),
             ↪  np.hstack([C, np.zeros((p,m)), np.zeros((p,p))])])
            xyzhat = solve_via_backsub(bigA,np.hstack([np.zeros(n),b,d]))
            xhat = xyzhat[:n]
            return xhat

        m = 100
        n = 50
        p = 10
        A = np.random.normal(size = (m,n))
        b = np.random.normal(size = m)
        C = np.random.normal(size = (p,n))
        d = np.random.normal(size = p)
        x1 = cls_solve(A, b, C, d)
        x2 = cls_solve_sparse(A, b, C, d)
        # compare solutions
        print(np.linalg.norm(x1 - x2))
```

```
6.115026165379654e-14
```

Solving least norm problem. In Python, the `solve_via_backsub` function that we defined previously and the numpy `np.linalg.solve` do not provide approximate solutions for under-determined set of equations, thus should not be used to find the least norm solution when the coefficient matrix is wide. Here we solve a least norm problem using several methods, to check that they agree.

```
In [ ]: p = 50
        n = 500
        C = np.random.normal(size=(p,n))
        d = np.random.normal(size=p)
        #solve via least norm analytical solution
        x1 = C.T @ np.linalg.inv(C @ C.T) @ d
```

```
#solve using cls_solve which uses KKT
x2 = cls_solve(np.eye(n),np.zeros(n),C,d)
#Using pseudo inverse
x3 = np.linalg.pinv(C) @ d
print(np.linalg.norm(x1-x2))
print(np.linalg.norm(x2-x3))
```

```
1.8636583822949544e-14
1.866643783671431e-14
```

17. Constrained least squares applications

17.1. Portfolio optimization

Compounded portfolio value. The cumulative value of a portfolio from a return time series vector r, starting from the traditional value of \$10000, is given by the value times series vector v, where

$$v_t = 10000(1 + r_1) \cdots (1 + r_{t-1}), \quad t = 1, \dots, T.$$

In other words, we form the *cumulative product* of the vector with entries $1 + r_t$. Numpy has a function that does this, `np.cumprod()`.

```
In [ ]: #Portfolio value of with re-investment, return time series r
        cum_value = lambda r: 10000*np.cumprod(1+r)
        T = 250 # One year's worth of trading days
        #Generate random returns sequence with
        #10% annualized return, 5% annualized risk
        mu = 0.10/250
        sigma = 0.05/np.sqrt(250)
        r = mu + sigma*np.random.normal(size = T)
        v = cum_value(r)
        # compute final value (compounded) and average return
        v[T-1], v[0]*(1+sum(r))
        # plot cumulative value over the year
        import matplotlib.pyplot as plt
        plt.ion()
        plt.plot(np.arange(T), v)
        plt.xlabel('t')
        plt.ylabel('v_t')
        plt.show()
```

The resulting figure for a particular choice of r is shown in Figure 17.1.

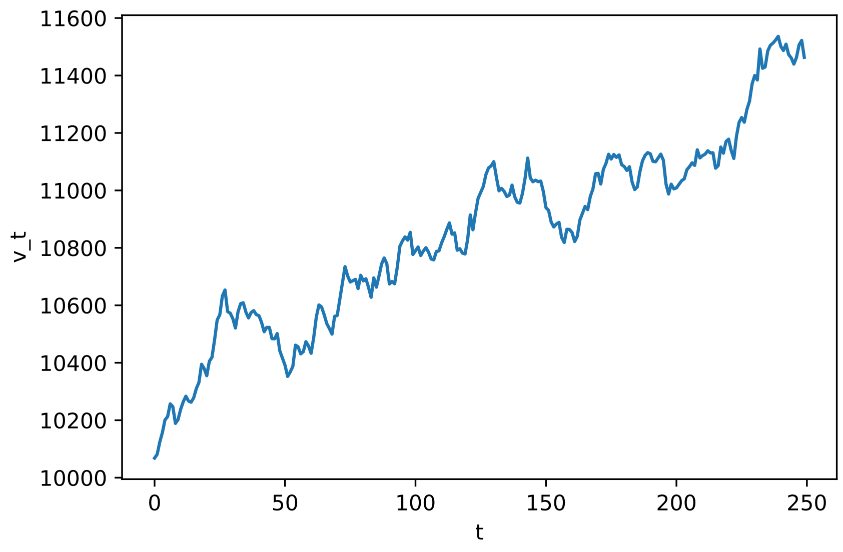


Figure 17.1.: Total portfolio value over time.

Portfolio optimization. We define a function `port_opt` that evaluates the solution (17.3) of the constrained least squares problem (17.2) in VMLS, and apply to the return data in VMLS Section 17.1.3.

```
In [ ]: def port_opt(R, rho):
            T, n = R.shape
            mu = np.sum(R, axis = 0).T/T
            KKT = np.vstack([np.column_stack([2*R.T @ R, np.ones(n), mu]),
            ↪  np.hstack([np.ones(n).T, 0 , 0]), np.hstack([mu.T, 0,
            ↪  0])])
            wz1z2 = solve_via_backsub(KKT, np.hstack([2*rho*T*mu, 1,
            ↪  rho]))
            w = wz1z2[:n]
            return w

        R, Rtest = portfolio_data()
        T, n = R.shape
        rho = 0.1/250 #Ask for 10% annual return
        w = port_opt(R, rho)
        r = R @ w #Portfolio return time series
        pf_return = 250*sum(r)/len(r)
        pf_risk = np.sqrt(250)*np.std(r)
```

```
print(pf_return)
print(pf_risk)
```

```
0.10000000001219751
0.08650183079228145
```

This produces the curve labeled '10%' in Figure 17.2. In the same figure we also include the plots for 20% and 40% annual return, and for the $1/n$ portfolio $w = (1/n)\mathbf{1}$.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        cum_value = lambda r: 10000*np.cumprod(1+r)
        # 10% annual return
        rho = 0.1/250 #Ask for 10% annual return
        w = port_opt(R, rho)
        r = R @ w #Portfolio return time series
        plt.plot(np.arange(T), cum_value(r), 'blue')
        # 20% annual return
        rho = 0.2/250
        w = port_opt(R, rho)
        r = R @ w #Portfolio return time series
        plt.plot(np.arange(T), cum_value(r), 'orange')
        # 40% annual return
        rho = 0.4/250
        w = port_opt(R, rho)
        r = R @ w #Portfolio return time series
        plt.plot(np.arange(T), cum_value(r), 'green')
        # Uniform portolio
        w = (1/n)*np.ones(n)
        r = R @ w #Portfolio return time series
        plt.plot(np.arange(T), cum_value(r), 'purple')
        plt.legend(['10%','20%','40%','1/n'])
        plt.show()
```

17.2. Linear quadratic control

We implement linear quadratic control as described in VMLS section 17.2, for a time-invariant system with matrices A, B, and C.

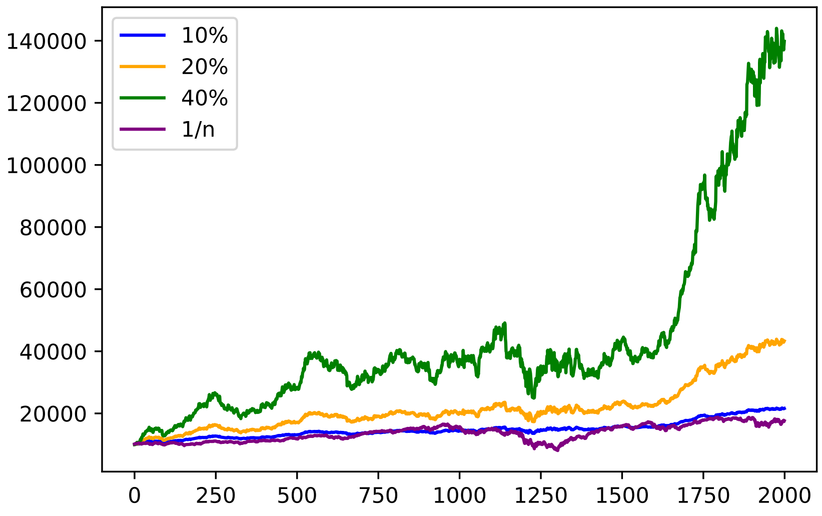


Figure 17.2.: Total value over time for four portfolio: the Pareto optimal portfolio with 10%, 20%, and 40% return, and the uniform portfolio. The total value is computed using the 2000×20 daily return matrix R.

Kronecker product. To create the big matrices $\tilde{A}$ and $\tilde{C}$, we need to define block diagonal matrices with the same matrix repeated a number of times along the diagonal. There are many ways to do this in Python. One of the simplest ways uses the `np.kron` function, for the Kronecker product of two matrices. The Kronecker product of an $m \times n$ matrix G and a $p \times q$ matrix H is defined as the $mp \times nq$ block matrix

$$\begin{bmatrix} G_{11}H & G_{12}H & \ldots & G_{1n}G \\ G_{21}H & G_{22}H & \ldots & G_{2n}G \\ \vdots & \vdots & & \vdots \\ G_{m1}H & G_{m2}H & \ldots & G_{mn}G \end{bmatrix}$$

It is computed in Python as `np.kron(G,H)`. If G is an $n \times n$ identity matrix, we obtain the block diagonal matrix with H repeated n times on the diagonal.

```
In [ ]: H = np.random.normal(size = (2,2))
        print(np.kron(np.eye(3),H))
```

```
[[-0.25527609 -0.09038866 -0.          -0.          -0.          -0.
↪  ]
 [ 0.12547218 -0.72681408  0.          -0.           0.          -0.
 ↪  ]
[-0.          -0.          -0.25527609 -0.09038866 -0.          -0.
 ↪  ]
 [ 0.          -0.           0.12547218 -0.72681408  0.          -0.
 ↪  ]
 [-0.          -0.          -0.          -0.          -0.25527609
 ↪  -0.09038866]
 [ 0.          -0.           0.          -0.           0.12547218
 ↪  -0.72681408]]
```

Alternatively, we can construct the matrix block by block.

```
In [ ]: H = np.random.normal(size = (2,2))
        G = np.eye(3)
        p, q = H.shape
        m, n = G.shape
        matrix = np.zeros((m*p,n*q))
        for k in range(n):
            matrix[p*k:p*(k+1),q*k:q*(k+1)] = H
        print(matrix)
```

```
[[-0.25527609 -0.09038866 -0.          -0.          -0.          -0.
 ↪  ]
 [ 0.12547218 -0.72681408  0.          -0.           0.          -0.
 ↪  ]
 [-0.          -0.          -0.25527609 -0.09038866 -0.          -0.
 ↪  ]
 [ 0.          -0.           0.12547218 -0.72681408  0.          -0.
 ↪  ]
 [-0.          -0.          -0.          -0.          -0.25527609
 ↪  -0.09038866]
 [ 0.          -0.           0.          -0.           0.12547218
 ↪  -0.72681408]]
```

Linear quadratic control example. We start by writing a function `lqr` that constructs and solves the constrained least squares problem for linear quadratic control. The function returns three arrays

```
x = [x[0], x[1], ..., x[T]],
u = [u[0], u[1], ..., u[T]],
y = [y[0], y[1], ..., y[T]].
```

The first two contain the optimal solution of the problem. The third array contains $y_t = Cx_t$.

We allow the input arguments `x_init` and `x_des` to be matrices, so we can solve the same problem for different pairs of initial and end states, with one function call. If the number of columns in `x_init` and `x_des` is q, then the entries of the three output sequences `x, u, y` are matrices with q columns. The ith columns are the solution for the initial and end states specified in the ith columns of `x_init` and `x_des`.

```
In [ ]: def lqr(A, B, C, x_init, x_des, T, rho):
            n = A.shape[0]
            m = B.shape[1]
            p = C.shape[0]
            q = x_init.shape[1]
            Atil = np.vstack([np.hstack([np.kron(np.eye(T), C),
            ↪   np.zeros((p*T,m*(T-1)))]), np.hstack([np.zeros((m*(T-1),
            ↪   n*T)), np.sqrt(rho)*np.eye(m*(T-1))])])
            btil = np.zeros((p*T+m*(T-1), q))

            # We'll construct Ctilde bit by bit
            Ctil11 =
            ↪   np.hstack([np.kron(np.eye(T-1),A),np.zeros((n*(T-1),n))])
            ↪   - np.hstack([np.zeros((n*(T-1),n)), np.eye(n*(T-1))])
            Ctil12 = np.kron(np.eye(T-1),B)
            Ctil21 = np.vstack([np.hstack([np.eye(n),
            ↪   np.zeros((n,n*(T-1)))]), np.hstack([np.zeros((n,n*(T-1))),
            ↪   np.eye(n)])])
            Ctil22 = np.zeros((2*n, m*(T-1)))

            Ctil = np.block([[Ctil11,Ctil12],[Ctil21,Ctil22]])
            dtil = np.vstack([np.zeros((n*(T-1),q)), x_init, x_des])
            z = cls_solve(Atil, btil.flatten(), Ctil, dtil.flatten())

            x = [z[i*n:(i+1)*n] for i in range(T)]
            u = [z[n*T+i*m: n*T+(i+1)*m] for i in range(T-1)]
            y = [C @ xt for xt in x]
            return x, u, y
```

Note that $q = 1$ in the above function as we are using `cls_solve`. We apply our `lqr` function to the example in section 17.2.1 of VMLS.

In [ ]:
```
A = np.array([[ 0.855, 1.161, 0.667], [0.015, 1.073, 0.053],
↪  [-0.084, 0.059, 1.022 ]])
B = np.array([[-0.076], [-0.139] ,[0.342 ]])
C = np.array([[ 0.218, -3.597, -1.683 ]])
n = 3
p = 1
m = 1
x_init = np.array([[0.496], [-0.745], [1.394]])
x_des = np.zeros((n,1))
```

We first plot the open-loop response in figure 17.3 (also shown in VMLS figure 17.4).

In [ ]:
```
T = 100
yol = np.zeros((T,1))
Xol = np.hstack([x_init, np.zeros((n,T-1))])
for k in range(T-1):
    Xol[:,k+1] = A @ Xol[:,k]
yol = C @ Xol
import matplotlib.pyplot as plt
plt.ion()
plt.plot(np.arange(T), yol.T)
plt.show()
```

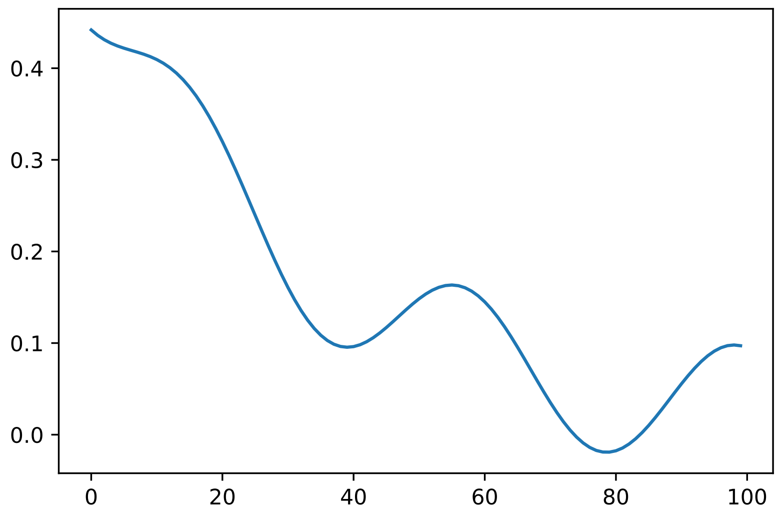


Figure 17.3.: Open-loop response $CA^{t-1}x^{init}$.

We then solve the linear quadratic control problem with $T = 100$ and $\rho = 0.2$. The result is shown in figure 17.4 (also shown in second row of VMLS figure 17.6).

In [ ]:
```
rho = 0.2
T = 100
x, u, y = lqr(A, B, C, x_init, x_des, T, rho)
J_input = np.linalg.norm(u)**2
J_output = np.linalg.norm(y)**2
print(J_input)
print(J_output)
```

```
0.7738942551160012
3.782998646333574
```

In [ ]:
```
plt.plot(np.arange(T-1), u)
plt.xlabel('t')
plt.ylabel('u_t')
plt.show()

plt.plot(np.arange(T), y)
plt.xlabel('t')
plt.ylabel('y_t')
plt.show()
```

Linear state feedback control. (This section will be added in the next iteration.)

17.3. Linear quadratic state estimation

The code for the linear quadratic estimation method is very similar to the one for linear quadratic control.

In [ ]:
```
def lqe(A, B, C, y, T, lam):
    n = A.shape[0]
    m = B.shape[1]
    p = C.shape[0]
    Atil = np.block([[np.kron(np.eye(T),C),
    ↪  np.zeros((T*p,m*(T-1)))], [np.zeros((m*(T-1), T*n)),
    ↪  np.sqrt(lam)*np.eye(m*(T-1))]])

    #We assume y is a p by T array, so we vectorize it
```

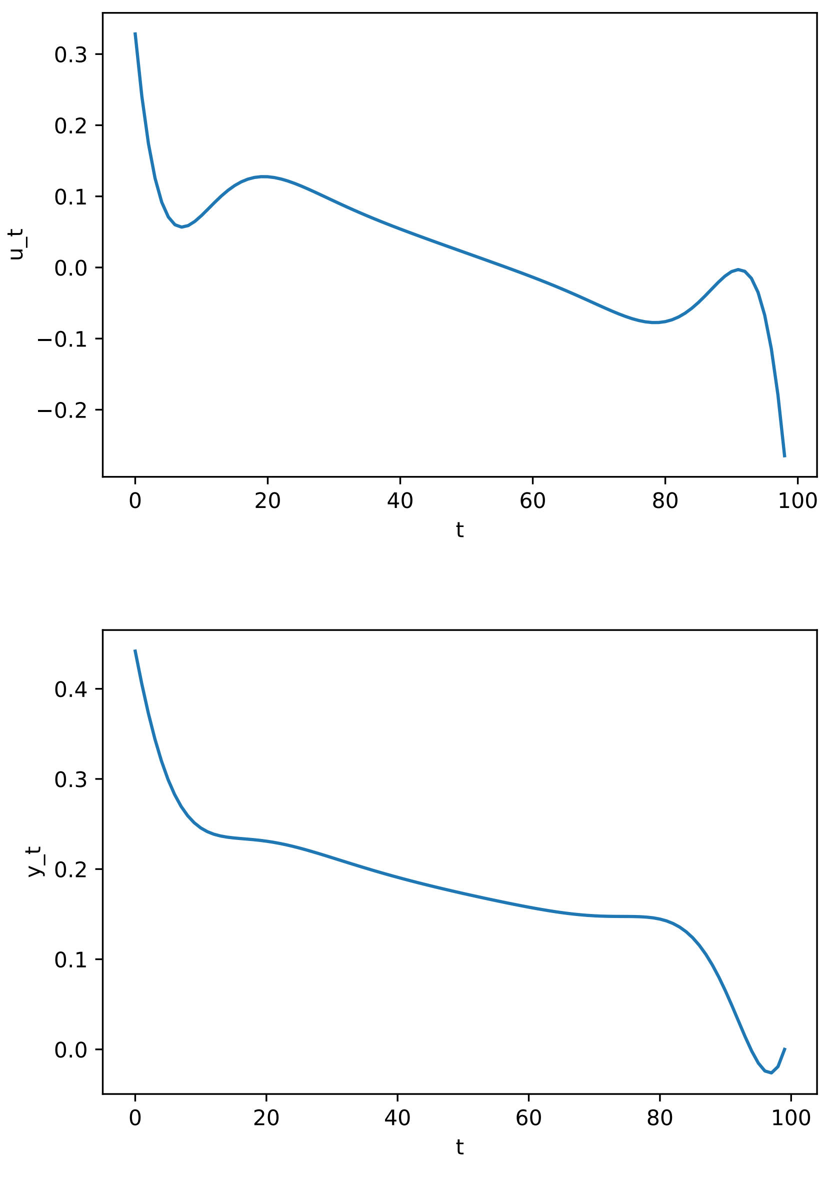


Figure 17.4.: Optimal input and output for $\rho = 0.2$.

```
    btil = np.block([np.hstack([i for i in
    ↪  y.T]),np.zeros((m*(T-1)))])
    Ctil = np.block([np.block([np.kron(np.eye(T-1),A),
    ↪  np.zeros((n*(T-1), n))]) + np.block([np.zeros((n*(T-1),
    ↪  n)), -np.eye(n*(T-1))]),np.kron(np.eye(T-1), B)])
    dtil = np.zeros(n*(T-1))

    z = cls_solve(Atil, btil, Ctil, dtil)
    x = [z[i*n:(i+1)*n] for i in range(T)]
    u = [z[n*T+i*m : n*T+(i+1)*m] for i in range(T-1)]
    y = [C @ xt for xt in x]
    return x, u, y
```

We use the system matrices in section 17.3.1 of VMLS. The output measurement data are read from the `estimation_data()` data set, which creates a 2×100 matrix `ymeas`. We compute the solution for $\lambda = 10^3$, shown in the lower-left plot of figure 17.8 of VMLS.

```
In [ ]: y = lq_estimation_data()
        A = np.block([[np.eye(2), np.eye(2)], [np.zeros((2,2)),
        ↪  np.eye(2)]])
        B = np.block([[np.zeros((2,2))], [np.eye(2)]])
        C = np.block([[np.eye(2), np.zeros((2,2))]])
        n = A.shape[0]
        m = B.shape[1]
        p = C.shape[0]
        T = 100
        lam = 1e+3
        x_hat, u_hat, y_hat = lqe(A,B,C,y,T,lam)

        import matplotlib.pyplot as plt
        plt.ion()
        plt.scatter(y[0,:], y[1,:])
        plt.plot(np.array([yt[0] for yt in y_hat]), np.array([yt[1] for yt
        ↪  in y_hat]),'r')
        plt.show()
```

The result is shown in Figure 17.5.

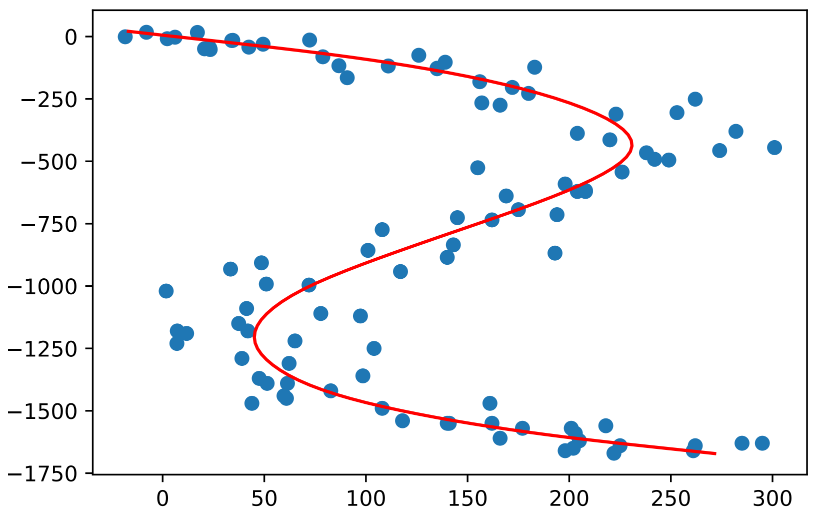


Figure 17.5.: The circles show 100 noisy measurements in 2-D. The solid line is the estimated trajectory $C\hat{x}_t$ for $\lambda = 1000$.

18. Nonlinear least squares

18.1. Nonlinear equations and least squares

18.2. Gauss-Newton algorithm

Basic Gauss-Newton algorithm. Let's first implement the basic Gauss-Newton method (algorithm 18.1 in VMLS) in Python. In Python, you can pass a function as an argument to another function, so we can pass `f` (the function) and also `Df` (the derivative or Jacobian matrix) to our Gauss-Newton algorithm.

```
In [ ]: def gauss_newton(f, Df, x1, kmax=10):
            x = x1
            for k in range(kmax):
                x = x - np.linalg.lstsq(Df(x),f(x))
            return x
```

Here we simply run the algorithm for a fixed number of iterations `kmax`, specified by an optimal keyword argument with default value 10. The code does not verify whether the final `x` is actually a solution, and it will break down when $Df(x^{(k)})$ has linearly dependent columns. This very simple implementation is only for illustrative purposes; the Levenberg-Marquardt algorithm described in the next section is better in every way.

Newton algorithm. The Gauss-Newton algorithm reduces to the Newton algorithm when the function maps n-vectors to n-vectors, so the function above is also an implementation of the Newton method for solving nonlinear equations. The only difference with the following function is the stopping condition. In Newton's method, one terminates when $\|f(x^{(k)})\|$ is sufficiently small.

```
In [ ]: def newton(f, Df, x1, kmax=20, tol=1e-6):
            x = x1
            fnorms = np.zeros((1,1))
            for k in range(kmax):
```

```
        fk = f(x)
        fnorms = np.vstack([fnorms, np.linalg.norm(fk)])
        if np.linalg.norm(fk) < tol:
            break
        if len(np.array([fk])) < 2:
            x = x - (fk/Df(x))
        else:
            x = x - solve_via_backsub(Df(x), fk)
    return x, fnorms
```

We add a second optional argument with the tolerance in the stopping condition. The default value is 10^{-6}. We also added a second output argument `fnorm`, with the sequence $\|f(x^{(k)})\|$, so we can examine the convergence in the following example. Since the `np.linalg.pinv` function only works when the input matrix has dimensions greater than 2×2, we split the case between scalars and matrices.

Newton algorithm for $n = 1$. Our first example is a scalar nonlinear equation $f(x) = 0$ with

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
In [ ]: Df = lambda x: 4/np.power((np.exp(x) + np.exp(-x)),2)
        f = lambda x: (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))
```

We first try with $x^{(1)} = 0.95$.

```
In [ ]: x, fnorms = newton(f, Df, 0.95)
        f(x)

Out[ ]: 4.34519743242004545e-07

In [ ]: fnorms

Out[ ]: array([[0.00000000e+00],
               [7.39783051e-01],
               [5.94166364e-01],
               [2.30111246e-01],
               [8.67002865e-03],
               [4.34519743e-07]])

In [ ]: plt.plot(fnorms,'o-')
        plt.xlabel('k')
plt.ylabel('|f|')
plt.show()
```

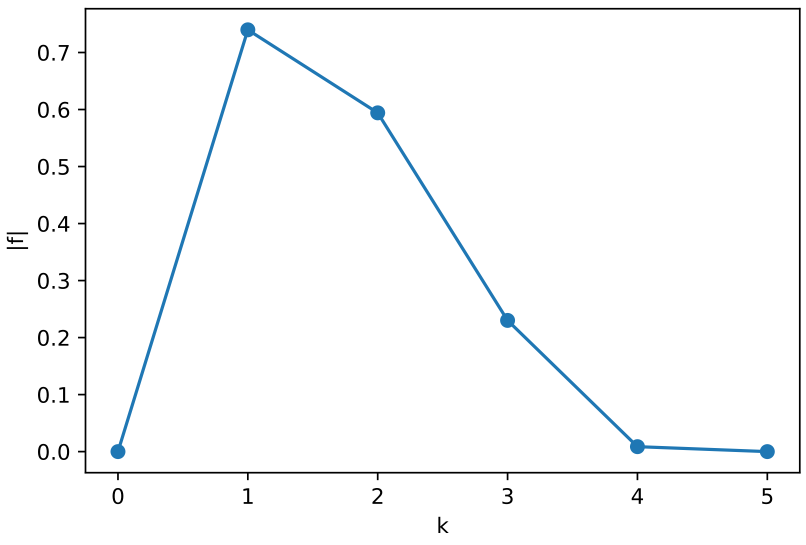


Figure 18.1.: The first iterations in the Newton algorithm for solving $f(x) = 0$ for starting point $x^{(1)} = 0.95$.

The method converges very quickly, as shown in figure 18.1. However it does not converge for a slightly larger starting point $x^{(1)} = 1.15$.

```
In [ ]: x, fnorms = newton(f, Df, 1.15)
        f(x)
```

```
Out[ ]: nan
```

```
In [ ]: fnorms
```

```
Out[ ]: array([[0.        ],
               [0.81775408],
               [0.86640565],
               [0.97355685],
               [1.        ],
               [       nan],
               [       nan],
               [       nan],
               [       nan],
               [       nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan],
       [        nan]])
```

18.3. Levenberg-Marquardt algorithm

The Gauss-Newton algorithm can fail if the derivative matrix does not have independent columns. It also does not guarantee that $\|f(x^{(k)})\|$ decreases in each iteration. Both of these shortcomings are addressed in the Levenberg-Marquardt algorithm. Below is a Python implementation of algorithm 18.3 in VMLS.

```
In [ ]: def levenberg_marquardt(f, Df, x1, lambda1, kmax=100, tol=1e-6):
            n = len(x1)
            x = x1
            lam = lambda1
            obj = np.zeros((0,1))
            res = np.zeros((0,1))
            for k in range(kmax):
                obj = np.vstack([obj, np.linalg.norm(f(x))**2])
                res = np.vstack([res, np.linalg.norm(2*Df(x).T @ f(x))])
                if np.linalg.norm(2*Df(x).T @ f(x)) < tol:
                    break
                xt = x - np.linalg.inv((Df(x).T @ Df(x)) +
                ↪  np.sqrt(lam)*np.eye(n)) @ (Df(x).T @ f(x))
                if np.linalg.norm(f(xt)) < np.linalg.norm(f(x)) :
                    lam = 0.8*lam
                    x = xt
                else:
                    lam = 2.0*lam
            return x, {'Objective':obj, 'Residual':res}
```

Here we use the second stopping criterion suggested on page 393 of VMLS, and checks whether the optimality condition is approximately satisfied. The default tolerance 10^{-6} can vary with the scale of the problem and the desired accuracy. Keep in mind that the optimality condition is a necessary condition and does not guarantee that the solution minimizes the nonlinear least squares objective $\|f(x)\|^2$. The code limits the number of iterations to $k^{\max}$, after which it is assumed that the algorithm fails to converge.

The function returns a dictionary with information about the sequence of iterations, including the value of $\|f(x^{(k)})\|^2$ and $\|Df(x^{(k)})^T f(x^{(k)})\|$ at each iteration.

Nonlinear equation. We apply the algorithm to the scalar function

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

with starting point $x^{(1)} = 1.15$.

```
In [ ]: Df = lambda x: 4/np.power((np.exp(x) + np.exp(-x)),2)
        f = lambda x: (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))
        x, history = levenberg_marquardt(f, Df, np.array([1.15]), 1.0)
        import matplotlib.pyplot as plt
        plt.ion()
        plt.plot(np.sqrt(history['Objective'][1:10]), 'o-')
        plt.xlabel('k')
        plt.ylabel('|f|')
        plt.show()
```

Note that we defined $x^{(1)}$ as the array `np.array([1.15])`, and use dot-operations in the definition of `f` and `Df` to ensure that these functions work with vector arguments. This is important because Python distinguishes scalars and 1-vectors. If we call the `levenberg-marquardt` function with a scalar argument `x1`, line 15 will raise an error, because Python does not accept subtractions of scalars and 1-vectors.

Equilibrium prices. We solve a nonlinear equation $f(p) = 0$ with two variables, where

$$f(p) = \exp(E^{\mathrm{s}} \log p + s^{\mathrm{nom}}) - \exp(E^{\mathrm{d}} \log p + d^{\mathrm{nom}}).$$

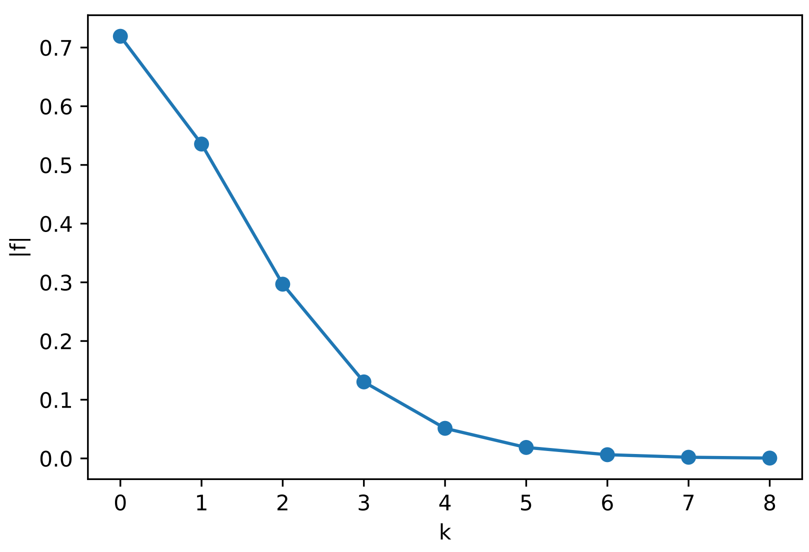


Figure 18.2.: Values of $|f(x^{(k)})|$ versus the iteration number k for the Levenberg-Marquardt algorithm applied to $f(x) = (exp(x) - exp(-x))/(exp(x) + exp(-x))$. The starting point is $x^{(1)} = 1.15$ and $\lambda^{(1)} = 1$.

Here exp and log are interpreted as elementwise vector operations. The problem parameters are $s^{\text{nom}} = (2.2, 0.3)$, $d^{\text{nom}} = (3.1, 2.2)$,

$$E^s = \begin{bmatrix} 0.5 & -0.3 \\ -0.15 & 0.8 \end{bmatrix}, \quad E^d = \begin{bmatrix} -0.5 & 0.2 \\ 0 & -0.5 \end{bmatrix}$$

In [ ]:
```
snom = np.array([2.2, 0.3]).reshape(2,1)
dnom = np.array([3.1, 2.2]).reshape(2,1)
Es = np.array([[0.5, -0.3], [-0.15,0.8]])
Ed = np.array([[-0.5,0.2], [0,-0.5]])
f = lambda p: (np.exp(Es @ np.log(p) + snom) - np.exp(Ed @
↪   np.log(p) + dnom))

def Df(p):
    S = np.exp(Es @ np.log(p) + snom)
    D = np.exp(Ed @ np.log(p) + dnom)
    result = np.block([[S.flatten()[0]*Es[0,0]/p[0],
    ↪  S.flatten()[0]*Es[0,1]/p[1]],
    ↪  [S.flatten()[1]*Es[1,0]/p[0],
    ↪  S.flatten()[1]*Es[1,1]/p[1]]]) -
    ↪  np.block([[D.flatten()[0]*Ed[0,0]/p[0],
    ↪  D.flatten()[0]*Ed[0,1]/p[1]],
    ↪  [D.flatten()[1]*Ed[1,0]/p[0],
    ↪  D.flatten()[1]*Ed[1,1]/p[1]]])
    return result

p, history = levenberg_marquardt(f, Df,
↪  np.array([3,9]).reshape(2,1), 1)
print(p)
```

```
[[5.64410859]
 [5.26575503]]
```

In [ ]:

```
import matplotlib.pyplot as plt
plt.ion()
plt.plot(history['Objective'][:10], 'o-')
plt.xlabel('k')
plt.ylabel('Objective')
plt.savefig('costfunction_k.pdf', format = 'pdf')
plt.show()
```

Location from range measurements. The next example is the location from range measurement on page 396 in VMLS. The positions of the $m = 5$ points a_i are given as rows in a 5×2 matrix `A`. The measurements are given in a 5-vector `rhos`. To simplify the code for the function `f(x)` and `Df(x)` we add a function `dist(x)` that computes the vector of distances $(\|x - a_1\|, \dots, \|x - a_m\|)$. The expression for the derivative is

$$Df(x) = \begin{bmatrix} \frac{x_1 - (a_1)_1}{\|x - a_1\|} & \frac{x_2 - (a_1)_2}{\|x - a_1\|} \\ \vdots & \vdots \\ \frac{x_1 - (a_m)_1}{\|x - a_m\|} & \frac{x_2 - (a_m)_2}{\|x - a_m\|} \end{bmatrix}.$$

They can be evaluated as the product of a diagonal matrix with diagonal entries $1/\|x - a_i\|$ and the 5×2 matrix with i, j entry $(x - a_i)_j$.

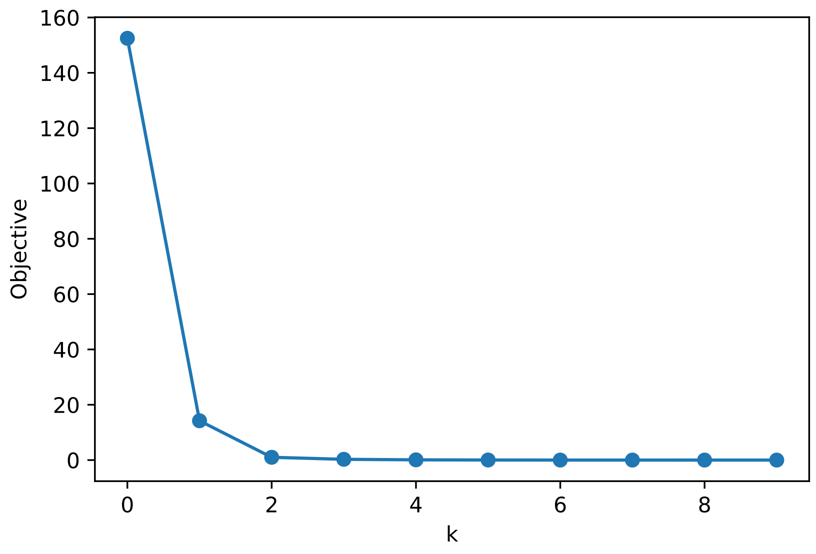


Figure 18.3.: Cost function $\|f(p^{(k)})\|^2$ versus iteration number k.

We run the Levenberg-Marquardt method for three starting points and $\lambda^{(1)} = 0.1$. The plot is shown in figure 18.4.

In [ ]:
```
# Five locations ai in a 5 by 2 matrix
A = np.array([[1.8, 2.5], [2.0, 1.7], [1.5, 1.5], [1.5, 2.0],
↪  [2.5, 1.5]])
# Vector of measured distances to five locations.
rhos = np.array([1.87288, 1.23950, 0.53672, 1.29273, 1.49353])
# dist(x) returns a 5-vector with the distances ||x-ai||.
dist = lambda x: np.sqrt(np.power((x[0] - A[:,0]), 2) +
↪  np.power((x[1] - A[:,1]), 2))
# f(x) returns the five residuals.
f = lambda x: dist(x).reshape(5,1) - rhos.reshape(5,1)
# Df(x) is the 5 by 2 derivative.
Df = lambda x: np.diag(1 / dist(x)) @ np.vstack([(x[0] -
↪  A[:,0]),(x[1] - A[:,1])]).T
# Solve with starting point (1.8, 3.5) and lambda = 0.1
x1, history1 = levenberg_marquardt(f, Df, np.array([1.8,
↪  3.5]).reshape(2,1), 0.1)
print(x1)
```

```
[[1.18248563]
 [0.82422915]]
```

In [ ]:
```
# Starting point (3.0,1.5)
x2, history2 = levenberg_marquardt(f, Df,
↪  np.array([3.0,1.5]).reshape(2,1), 0.1)
print(x2)
```

```
[[1.18248561]
 [0.82422909]]
```

In [ ]:
```
# Starting point (2.2,3.5)
x3, history3 = levenberg_marquardt(f, Df,
↪  np.array([2.2,3.5]).reshape(2,1), 0.1)
print(x3)
```

```
[[2.98526651]
 [2.12157657]]
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
plt.plot(history1['Objective'][:10],'bo-')
plt.plot(history2['Objective'][:10],'ro-')
plt.plot(history3['Objective'][:10],'go-')
plt.legend(['Starting point 1','Starting point 2','Starting point
↪  3'])
plt.xlabel('k')
plt.ylabel('Objective')
plt.show()
```

18.4. Nonlinear model fitting

Example. We fit a model

$$\hat{f}(x;\theta) = \theta_1 \exp(\theta_2 x)\cos(\theta_3 x + \theta_4)$$

to $N = 60$ data points. We first generate the data.

In [ ]:
```
# Use these parameters to generate data
theta_x = np.array([1, -0.2, 2*np.pi/5, np.pi/3])
# Choose 60 points x between 0 and 20
```

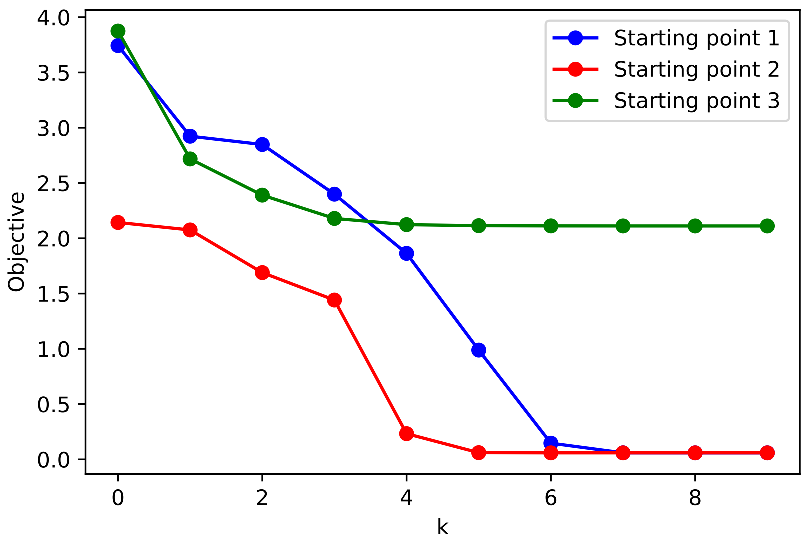


Figure 18.4.: Cost function $\|f(x^{(k)}\|^2$ versus iteration number k for the three starting points in the location from range measurements example.

```
M = 30
xd = np.vstack([5*np.random.random((M,1)),
↪  5+15*np.random.random((M,1))])
# Evaluate function at these points
yd = theta_x[0]*np.exp(theta_x[1]*xd)*np.cos(theta_x[2]*xd +
↪  theta_x[3])
# Create a random perturbation of yd
N = len(yd)
yd = np.multiply(yd.T, (1 + 0.2*np.random.normal(size = N)) +
↪  0.015*np.random.normal(size = N))

import matplotlib.pyplot as plt
plt.ion()
plt.scatter(xd, yd)
plt.show()
```

We now run our Levenberg-Marquardt code with starting point $\theta^{(1)} = (1, 0, 1, 0)$ and $\lambda^{(1)} = 1$. The 60 points and the fitted model are shown in figure 18.5.

```
In [ ]: f = lambda theta: (np.hstack(theta[0]*np.exp(theta[1]*xd) *
        ↪  np.cos(theta[2]*xd + theta[3])) - yd).flatten()
```

```
Df = lambda theta: np.hstack([np.exp(theta[1]*xd) *
↪  np.cos(theta[2]*xd + theta[3]),
↪  theta[0]*(xd*np.exp(theta[1]*xd) * np.cos(theta[2]*xd +
↪  theta[3])), - theta[0]*np.exp(theta[1]*xd) * xd *
↪  np.sin(theta[2]*xd + theta[3]), - theta[0] *
↪  np.exp(theta[1]*xd) * np.sin(theta[2]*xd + theta[3])])
theta1 = np.array([1,0,1,0])
theta, history = levenberg_marquardt(f, Df, theta1, 1.0)
x = np.linspace(0,20,500)
y = theta[0]*np.exp(theta[1]*x) * np.cos(theta[2]*x + theta[3])

import matplotlib.pyplot as plt
plt.ion()
plt.scatter(xd, yd)
plt.plot(x,y, 'r')
plt.show()
```

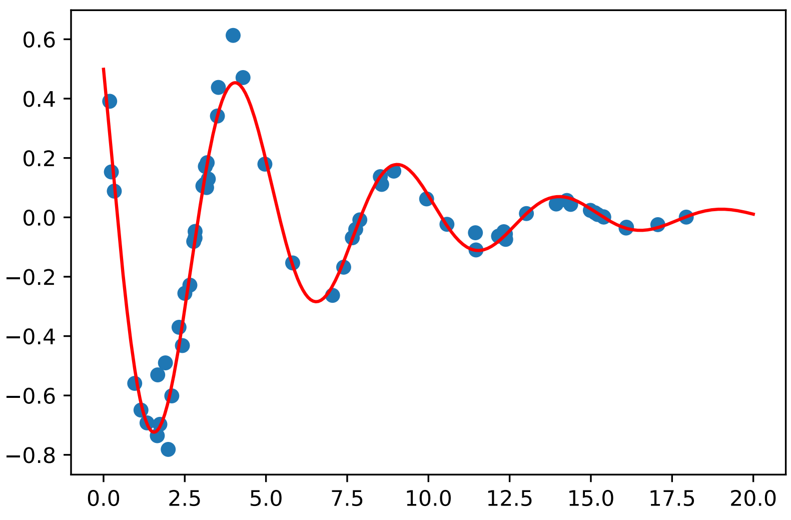


Figure 18.5.: Least squares fit of a function $\hat{f}(x;\theta) = \theta_1 \exp(\theta_2 x)\cos(\theta_3 x + \theta_4)$ to $N = 60$ points $(x^{(i)} y^{(i)})$.

Orthogonal distance regression. In figure 18.14 of VMLS we use orthogonal distance regression to fit a cubic polynomial

$$\hat{f}(x;\theta) = \theta_1 + \theta_2 x + \theta_3 x^2 + \theta_4 x^3$$

to $N = 25$ data points.

We first read in the data and compute the standard least squares fit.

```
In [ ]: xd, yd = orth_dist_reg_data() #2 vectors of length N = 25
        N = len(xd)
        p = 4
        theta_ls = solve_via_backsub(vandermonde(xd,p), yd)
```

The nonlinear least squares formulation on page 400 of VMLS has $p + N$ variables $\theta_1, \ldots, \theta_p, u^{(1)}, \ldots, u^{(N)}$. We will store them in that order in the nonlinear least squares vector variable. The objective is to minimize the squared norm of the $2N$-vector.

$$\begin{bmatrix} \hat{f}(u^{(1)}); \theta - y^{(1)} \\ \vdots \\ \hat{f}(u^{(N)}); \theta - y^{(N)} \\ u^{(1)} - x^{(1)} \\ \vdots \\ u^{(N)} - x^{(N)} \end{bmatrix}$$

```
In [ ]: def f(x):
            theta = x[:p]
            u = x[p:]
            f1 = vandermonde(u,p) @ theta - yd
            f2 = u - xd
            return np.vstack([f1,f2])

        def Df(x):
            theta = x[:p]
            u = x[p:]
            D11 = vandermonde(u,p)
            D12 = np.diag(np.hstack(theta[1] + 2*theta[2]*u +
            ↪   3*theta[3]*np.power(u,2)))
            D21 = np.zeros((N,p))
            D22 = np.eye(N)
            return np.block([[D11,D12], [D21,D22]])
```

We now call `levenberg_marquardt` function with these two functions. A natural choice for the initial points is to use the least squares solution for the variables θ and the data

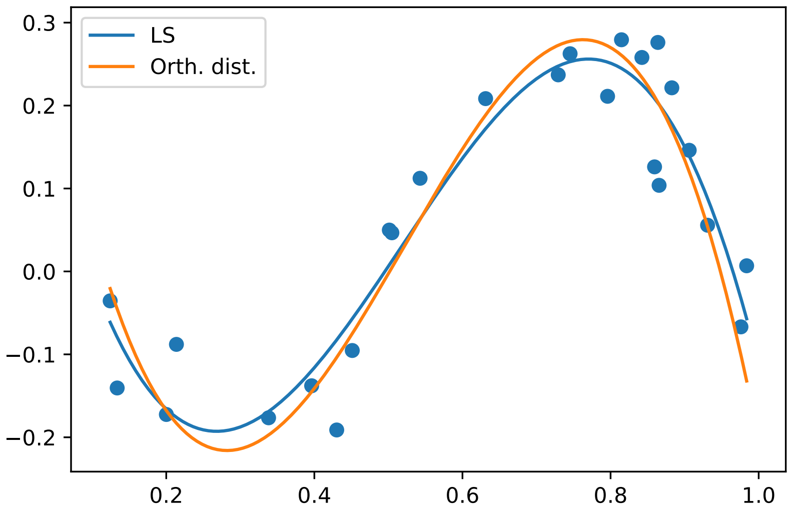


Figure 18.6.: Least squares and orthogonal distance regression fit of a cubic polynomial to 25 data points.

points $x^{(i)}$ for the variables $u^{(i)}$. We use $\lambda^{(1)} = 0.01$.

```
In [ ]: z = np.vstack([theta_ls.reshape(4,1), xd])
        sol, hist = levenberg_marquardt(f, Df, z, 0.01)
        theta_od = sol[:p]
```

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        plt.scatter(xd, yd)
        x = np.linspace(min(xd), max(xd), 500)
        y_ls = vandermonde(x,p) @ theta_ls
        y_od = vandermonde(x,p) @ theta_od
        plt.plot(x, y_ls)
        plt.plot(x, y_od)
        plt.legend(['LS', 'Orth. dist.'])
        plt.show()
```

Figure 18.6 shows the two fitted polynomials.

19. Constrained nonlinear least squares

19.1. Constrained nonlinear least squares

19.2. Penalty algorithm

Let's implement the penalty algorithm (algorithm 19.1 in VMLS).

```
In [ ]: def penalty_method(f, Df, g, Dg, x1, lambda1, kmax=100,
        ↪ feas_tol=1e-4, oc_tol=1e-4):
            x = x1
            mu = 1.0
            feas_res = [np.linalg.norm(g(x))]
            oc_res = [[np.linalg.norm(2*Df(x).T * f(x) +
            ↪ 2*mu*Dg(x).T*g(x))]]
            lm_iters = []
            for k in range(kmax):
                F = lambda x: np.hstack([f(x), np.sqrt(mu)*g(x)])
                DF = lambda x: np.block([[Df(x)],[ np.sqrt(mu)*Dg(x)]])
                x, hist = levenberg_marquardt(F, DF, x, lambda1,
                ↪ tol=oc_tol)
                feas_res.append(np.linalg.norm(g(x)))
                oc_res.append(hist['Residual'][-1])
                lm_iters.append(len(hist['Residual']))
                if np.linalg.norm(g(x)) < feas_tol:
                    break
                mu = 2*mu
            return x,
            ↪ {'lm_iters':lm_iters,'feas_res':feas_res,'oc_res':oc_res}
```

Here we call the function `levenberg_marquardt` of the previous chapter to minimize $\|F(x)\|^2$ where

$$F(x) = \begin{bmatrix} f(x) \\ \sqrt{\mu}g(x) \end{bmatrix}.$$

We evaluate two residuals. The ‘feasibility’ residual $\|g(x^{(k)})\|$ is the error in the constraint $g(x) = 0$. The ‘optimality condition’ residual is defined as

$$\|2Df(x^{(k)})^T f(x^{(k)}) + 2Dg(x^{(k)})^T z^{(k)}\|$$

where $z^{(k)} = 2\mu^{(k-1)} g(x^{(k)})$ and we take $\mu^{(0)} = \mu^{(1)}$. We obtain the optimality condition residual as the last residual in the Levenberg-Marquardt method. We return the final x, and a dictionary containing the two sequences of residuals and the number of iterations used in each call to the Levenberg-Marquardt algorithm

Examples. We apply the method to a problem with two variables

$$f(x_1, x_2) = \begin{bmatrix} x_1 + exp(-x_2) \\ x_1^2 + 2x_2 + 1 \end{bmatrix}, \quad g(x_1, x_2) = x_1^2 + x_1^3 + x_2 + x_2^2.$$

```
In [ ]: f = lambda x: np.array([x[0] + np.exp(-x[1]), x[0]**2 + 2*x[1] +
        ↪  1])
        Df = lambda x: np.array([[1 ,- np.exp(-x[1])],[2*x[0] , 2]])
        g = lambda x: np.array([x[0] + x[0]**3 + x[1] + x[1]**2])
        Dg = lambda x: np.array([1 + 3*x[0]**2, 1+ 2*x[1]])

        x, hist = penalty_method(f, Df, g, Dg, np.array([0.5, -0.5]), 1.0)
        print(x)
```

```
[-3.31855595e-05, -2.78464833e-05]
```

The following lines create a staircase plot with the residuals versus the cumulative number of Levenberg-Marquardt iterations as in VMLS figure 19.4. The result is in figure 19.1.

```
In [ ]: import matplotlib.pyplot as plt
        plt.ion()
        cum_lm_iters = np.cumsum(hist['lm_iters'])
        itr = np.hstack([0,np.hstack([[i,i] for i in cum_lm_iters])])
        feas_res = np.hstack([np.hstack([r,r] for r in
        ↪  hist['feas_res'][:-1]), hist['feas_res'][-1]])
        oc_res =  np.hstack([np.hstack([r[0],r[0]] for r in
        ↪  hist['oc_res'][:-1]), hist['oc_res'][-1]])
        plt.plot(itr, feas_res,'o-')
        plt.plot(itr, oc_res,'o-')
plt.yscale('log')
plt.legend(['Feasibility','Opt.cond.'])
plt.xlabel('Cumulative Levenberg-Marquardt iterations')
plt.ylabel('Residual')
plt.show()
```

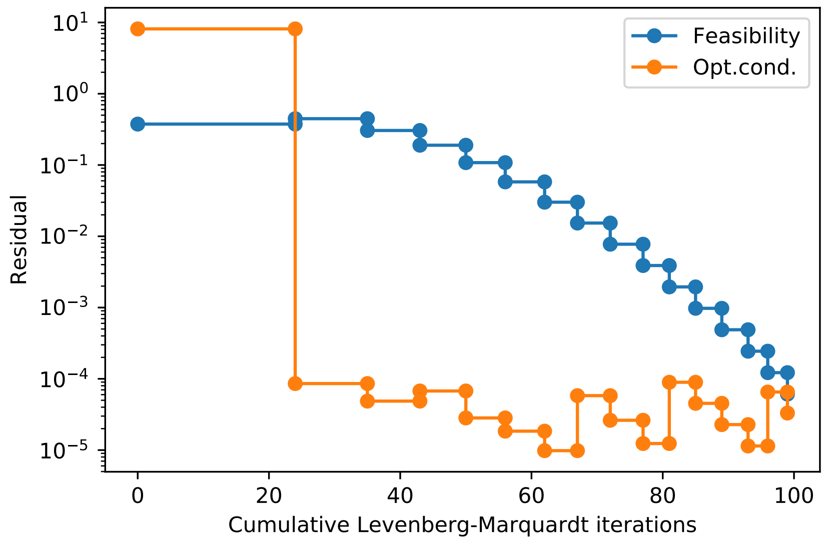


Figure 19.1.: Feasibility and optimality condition errors versus the cumulative number of Levenberg-Marquardt iterations in the penality algorithm.

19.3. Augmented Lagrangian algorithm

```
In [ ]: def aug_lag_method(f, Df, g, Dg, x1, lambda1, kmax=100,
        ↪ feas_tol=1e-4, oc_tol=1e-4):
            x = x1
            z = np.zeros(len(g(x)))
            mu = 1.0
            feas_res = [np.linalg.norm(g(x))]
            oc_res = [[np.linalg.norm(2*Df(x).T * f(x) +
            ↪ 2*mu*Dg(x).T*g(x))]]
            lm_iters = []
            for k in range(kmax):
```

```
        F = lambda x: np.hstack([f(x), np.sqrt(mu)*(g(x) +
        ↪  z/(2*mu))])
        DF = lambda x: np.block([[Df(x)], [ np.sqrt(mu)*Dg(x)]])
        x, hist = levenberg_marquardt(F, DF, x, lambda1,
        ↪  tol=oc_tol)
        z = z + 2*mu*g(x)
        feas_res.append(np.linalg.norm(g(x)))
        oc_res.append(hist['Residual'][-1])
        lm_iters.append(len(hist['Residual']))
        if np.linalg.norm(g(x)) < feas_tol:
            break
        if ~(np.linalg.norm(g(x)) < 0.25*feas_res[-2]):
            mu = 2*mu
    return x, z, {'lm_iters': lm_iters, 'feas_res': feas_res,
    ↪  'oc_res': oc_res}
```

Here the call to the Levenberg-Marquardt algorithm is to minimize $\|F(x)\|^2$ where

$$F(x) = \begin{bmatrix} f(x) \\ \sqrt{\mu^{(k)}}(g(x) + z^{(k)}/(2\mu^{(k)})) \end{bmatrix}.$$

We again record the feasibility residuals $\|g(x^{(k)})\|$ and the optimality conditions residuals

$$\|2Df(x^{(k)})^T f(x^{(k)}) + 2Dg(x^{(k)})^T z^{(k)}\|,$$

and return them in a dictionary.

Example. We continue the small example.

```
In [ ]: x, z, hist = aug_lag_method(f, Df, g, Dg, np.array([0.5, -0.5]),
        ↪  1.0)
        print(x)
        print(z)
```

```
[-1.79855232e-05, -1.49904382e-05]
[-1.99995875]
```

The following code shows the convergence figure 19.4 in VMLS. The plot is given in figure 19.2.

In [ ]:
```
import matplotlib.pyplot as plt
plt.ion()
cum_lm_iters = np.cumsum(hist['lm_iters'])
itr = np.hstack([0,np.hstack([[i,i] for i in cum_lm_iters])])
feas_res = np.hstack([np.hstack([r,r] for r in
↪  hist['feas_res'][:-1]), hist['feas_res'][-1]])
oc_res =  np.hstack([np.hstack([r[0],r[0]] for r in
↪  hist['oc_res'][:-1]), hist['oc_res'][-1]])

plt.plot(itr, feas_res,'o-')
plt.plot(itr, oc_res,'o-')
plt.yscale('log')
plt.legend(['Feasibility','Opt.cond.'])
plt.xlabel('Cumulative Levenberg-Marquardt iterations')
plt.ylabel('Residual')
plt.show()
```

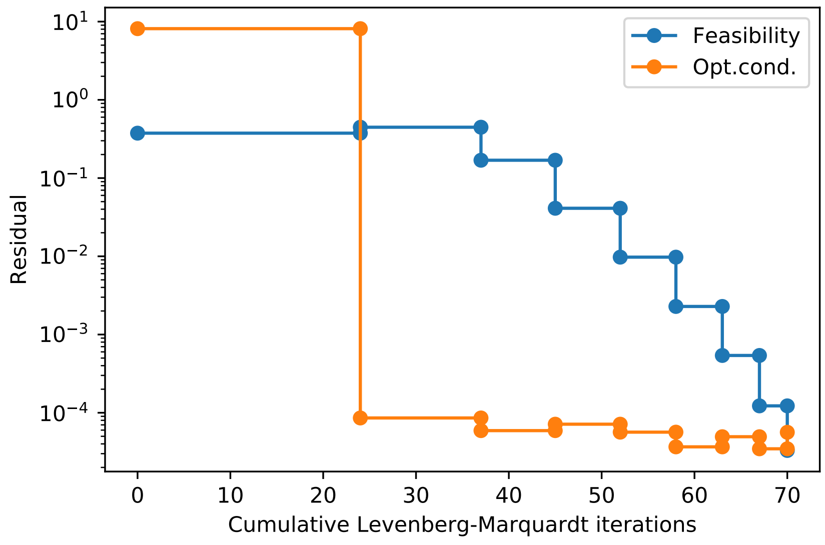


Figure 19.2.: Feasibility and optimality condition errors versus the cumulative number of Levenberg-Marquardt iterations in the augmented Lagrangian algorithm.

19.4. Nonlinear control

Appendix

A. Common coding errors

A.1. I am getting a Python error message.

It is natural to get error messages when you are coding, especially for beginners. Reading the error messages generated by Python can be very helpful in identifying the problem in your code syntax. Here are some common types of error messages in Python:

Attribute Error

- You are calling a method on the wrong type of object.

SyntaxError

- You have forgotten the quotation marks (`" "` or `' '`) around a string.
- You have forgotten to put a colon (:) at the end of a `def/if/for` line
- You have different numbers of open and close brackets

TypeError

- You are trying to use an operator on the wrong type of object
- You have used non-integer numbers when slicing a list
- You expect an object to have a value but the value is actually `None`
- You have called a method or function with the wrong number or type of arguments (input to a function).

Identification Error

- You have used a mixture of tabs and spaces
- You have not indented all lines in a block equally
- You have not indented when Python expects you to

NameError

- You have forgotten to define a variable
- You have misspelled a variable, function or method name
- You have forgotten to import a module or a package
- Your code uses a variable outside the scope (e.g. within the function) where it is defined
- Your code calls a function before it has been defined
- You are trying to print a single word and have forgotten the quotation marks

IOError

- You are trying to open a file that does not exist
- You are in the wrong directory

IndexError

- The index that you have given to a sequence is out of range

Key Error

- You are trying to look up a key that does not exist in a dictionary

ValueError

- You are trying to perform numerical operations on arrays with different dimensions
- You have given an argument with the right type but an inappropriate value to a function or to an operation

A.2. I am not getting an error message but my results are not as expected.

For loops

- **A list which should have a value for every iteration only has a single value.** You have defined the list inside the loop, you should move it outside.
- **A loop which uses the range function misses out the last value.** The range function is exclusive at the finish, you should increase it by one.
- **You are trying to loop over a collection of strings, but you are getting individual characters.** You are iterating over a string by mistake.
- **You are trying to write multiple lines to a file but only getting a single line.** You have opened the file inside the loop, you should move it outside.

If statements

- **Two numbers which should be equal are not.** You are comparing a number with a string representation of a number (e.g. `if 5 == '5'`)
- **A complex condition is not giving the expected result.** The order of precedence in the condition is ambiguous. Try adding some parentheses.

Others

- **A variable that should contain a value does not.** You are storing the return value of a function which changes the variables itself (e.g. sort).
- **I am reading a file but getting no input.** You have already read the contents of the file in the code, so the cursor is at the end.
- **A regular expression is not matching when I expect it to.** You have forgotten to use raw strings or escape backslash .
- **You are trying to print a value but getting a bunch of weird-looking string.** You are printing an object (e.g. a FileObject) when you want the result of calling a method on the object.

B. Data sets

The data sets we used in this Python language companion can be found on: `https://github.com/jessica-wyleung/VMLS-py`. You can download the jupyter noteobok from the repository and work on it directly or you can copy and paste the data set onto your own jupyter notebook.

`house_sales_data()`. Returns a dictionary `D` with the Sacramento house sales data. The 6 items in the dictionary are vectors of length 774, with data for 774 house sales.
`D["price"]`: selling price in 1000 dollars
`D["area"]`: area in 1000 square feet
`D["beds"]`: number of bedrooms
`D["baths"]`: number of bathrooms
`D["condo"]`: 1 if a condo, 0 otherwise
`D["location"]`: an integer between 1 and 4 indicating the location.

`population_data()`. Returns a dictionary `D` with the US population data. The items in the dictionary are three vectors of length 100.
`D["population"]`: 2010 population in millions for ages 0,. . . ,99
`D["birth_rate"]`: birth rate
`D["death_rate"]`: death rate.

`petroleum_consumption_data()`. Returns a 34-vector with the world annual petroleum consumption between 1980 and 2013, in thousand barrels/day.

`vehicle_miles_data()`. Returns a 15 matrix with the vehicle miles traveled in the US (in millions), per month, for the years 2000, . . . , 2014.

`temperature_data()`. Returns a vector of length $774 = 31 \times 24$ with the hourly temperature at LAX in May 2016.

`iris_data()`. Returns a dictionary D with the Iris flower data set. The items in the dictionary are:
`D["setosa"]`: a 50×4 matrix with 50 examples of Iris Setosa
`D["versicolor"]`: a 50×4 matrix with 50 examples of Iris Versicolor
`D["virginica"]`: a 50×4 matrix with 50 examples of Iris Virginica.

The columns give values for four features: sepal length in cm, sepal width in cm, petal length in cm, petal width in cm.

`ozone_data()`. Returns a vector of length $336 = 14 \times 24$ with the hourly ozone levels at Azusa, California, during the first 14 days of July 2014.

`regularized_fit_data()`. Returns a dictionary `D` with data for the regularized data fitting example. The items in the dictionary are:
`D["xtrain"]`: vector of length 10
`D["ytrain"]`: vector of length 10
`D["xtest"]`: vector of length 20
`D["ytest"]`: vector of length 20.

`portfolio_data()`. Returns a tuple `(R, Rtest)` with data for the portfolio optimization example. `R` is a 2000×20 matrix with daily returns over a period of 2000 days. The first 19 columns are returns for 19 stocks; the last column is for a risk-free asset. `Rtest` is a 500×20 matrix with daily returns over a different period of 500 days.

`lq_estimation_data()`. Returns a 2×100 matrix with the measurement data for the linear quadratic state estimation example.

`orth_dist_reg_data()`. Returns a tuple `(xd, yd)` with the data for the orthogonal distance regression example.