APPLIED MACHINE LEARNING
FOR
ENGINEERS

Aamir Ahmad
Andrea Beck
André Mielke
Fabian Schimpf

IAG, IFR, ISD
-
University of Stuttgart

Version 0.2α - February 15, 2021

Preface

This LaTeX-script was automatically generated from the Jupyter notebooks used in the wintersemester 2020/2021 course Applied Machine Learning for Engineers with *Pandoc* and *XeLaTeX*. This was the first time the course took place so naturally, many errors and inconsistencies plague the content of this document. E.g., the interactive portions of the notebooks are not translated into images, and some tables look wonky. Take everything with a grain of salt and be cautious. There's no guarantee for anything in this document being correct. It will be continuously improved. Suggestions are always welcome.

- *André*

Copyright Notice

Contents

1 Python Introduction and Jupyter

Python's design goals are easy accessibility, easy development processes, and generality. Unlike other programming languages used in science and engineering, like C++ or Fortran, it is not a compiled language. The processes of compilation and linking is completely omitted from the development cycle when running pure Python. Python is an interpreted language. The interpreter is also called Python and can be used on almost any operating system.

There are two ways to use Python. In a shell, you can start the interpreter by invoking the python interpreter by simply typing "python" and hitting return. On Windows, the interpreter is often in "C:\pythonxx", where "xx" is the version number. The easiest solution to get python for Windows systems is to install "anaconda", which includes the "conda" package manager for python. To quit your session, press ctrl+d. This mode is usually only good for quick testing or small calculations, since nothing is saved and to reuse code you'd have to program functions or classes. The second way is to write a script. This is simple text in a textfile, which can later be executed by invoking the Python interpreter and providing the filename as an argument. The most comfortable way, and this is highly recommended for developing, is using an integrated development environmet (IDE). These programs offer rich features and comfortable shortcuts, as well as good error recognition and solution suggestions. Big names for Python are Spyder and pyCharm.

1.1 Jupyter

In this course, you won't need to install Python on your system. Instead, we will work with Jupyter notebooks, such as the one you're looking at right now. Jupyter notebooks are successors to IPython (interactive Python) and were initially developed to support JUlia, PYThon, and R, hence the name. Since then, support for a multitude of kernels was added and it's difficult to find a widely adopted language that isn't supported. Jupyter notebooks aim to provide an easy-to-use interface for programming tasks in a presentable fashion. The notebook is subdivided into cells, of which you will find two kinds: markdown cells, such as this one, or code cells, such as the one below:

[1]:
```
print("This is a code cell")
```

```
This is a code cell
```

You can change the type of a cell under "Cell" -> "Type". Markdown cells allow, well, usage of markdown, which enables rich text formatting options. See for example this markdown cheat sheet: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet To edit the text of a markdown cell, double-click on it. To submit your changes and also to execute code in a code cell click on it, then hit "shift" + "enter". Try this with the following cell.

[3]:
```
print(5*100/5.9)
```

```
84.7457627118644
```

The output is shown directly below the code cell. You can also directly input LaTeX-code (mainly math-related code) by either enclosing it in `$dollar signs\$` or starting an environment like `\begin{equation} \end{equation}`, just like you're probably used to.

That's pretty much all you need to know to work with the notebooks. For more info and a guided tour, click on "Help", then "User Interface Tour". For shortcuts, see "Help", then "Keyboard

Shortcuts".

1.2 Operators and strings

Python can be used as a calculator. Standard operations include +,-,*,/,**,//,%. Let's see what these do:

[3]:
```
3+7
5+8
```

[3]:
```
13
```

[5]:
```
print("Addition: with '+': \t\t3 + 11 = ", 3 + 11)
print("Subtraction with '-': \t\t7 - 11 = ", 7 - 11)
print("Multiplication with '*': \t78.2 * 99.3 = ", 78.2 * 99.3)
print("Division with '/': \t\t38 / 14 = ", 38/14)
print("Exponentiation with **: \t2**10 = ", 2**10)
```

```
Addition: with '+':             3 + 11 =  14
Subtraction with '-':           7 - 11 =  -4
Multiplication with '*':        78.2 * 99.3 =  7765.26
Division with '/':              38 / 14 =  2.7142857142857144
Exponentiation with **:         2**10 =  1024
```

Notice a few things here. We used the "print"-function, which prints all kinds of things and tries to make the output as nice as possible. The first thing printed here are `strings`, which are characters enclosed in quotes. Strings can also be manipulated by some of the operations above:

[3]:
```
print("first part" + "second part")
print(3*"three times")
print("number is " + str(3))    # concatenation only works with strings
```

```
first partsecond part
three timesthree timesthree times
number is 3
```

The `\t` are tab characters, which help align output nicely. To concatenate numbers to a string, you need to use the `str()` function on it first.

We missed two of the operators mentioned above. What do they do?

[1]:
```
print("??: \t67 // 11 = ", 67 // 11)
print("??: \t67  % 11 = ", 67  % 11)
```

```
??:     67 // 11 =  6
??:     67  % 11 =  1
```

A convenient way to add or subtract things is using the following abbreviated operators:

[2]:
```
a = 17
```

```
a *= 7 # a = a  + 7
print(a)

a -= 7
print(a)
```

```
119
112
```

1.3 Modules

To extend the base functionality of Python, extension modules can be imported. Many useful modules are delivered with the standard package, but many of those we will use have to be installed from external sources, like for example pyTorch, TensorFlow, numpy, scikit-learn and so on, which we will use at a later point. The preferred way to do this is to use a package manager. Starting with native modules, `math` offers many useful tools:

[3]:
```
import math

print(math.sin(math.pi))
print(math.cos(math.pi))
print(math.exp(0.7))
```

```
1.2246467991473532e-16
-1.0
2.0137527074704766
```

Sometimes directly importing modules clutters the workspace. If you only need a small subset of the functions, classes, or constants provided by a module, you can restrict the import to those:

[2]:
```
from math import sin, cos, exp, pi

print(sin(pi))
print(cos(pi))
print(exp(0.7))
```

```
1.2246467991473532e-16
-1.0
2.0137527074704766
```

Much nicer. In many cases, you do need a lot of different functions from a module and sometimes modules have long names. The workaround to this is to give an imported module a shorter alias. This is the preferred method of importing modules. It also helps avoiding masking issues, where two functions from different packages have the same name and in the end you don't know which of them is used in your script. An example would be the sine functions from `math` and `numpy`.

[4]:
```
import math as m
import numpy as np
```

```
print(m.sin(0))
print(np.sin(0))
```

```
0.0
0.0
```

1.4 Variables

We already used `pi` above from the `math` module as a variable. Variables are easily initiated in Python, since Python is a dynamically typed language, meaning you don't have to tell it what kind of variable you want to save in memory. It will deduce that from the value you're assigning to the variable:

[4]:
```
import math as m

a = 78.2
i = 15
s = "characters"
f = m.sin

print(type(a))
print(type(i))
print(type(s))
print(type(f))
```

```
<class 'float'>
<class 'int'>
<class 'str'>
<class 'builtin_function_or_method'>
```

This does come with detriments. E.g., you can never be sure what type a variable gets after assignment, so you should make sure that if you need a floating point variable, that you initiate it with a floating point number. This is just a safety measure, usually Python does the thinking here for the programmer. Speaking of floating point variables, perhaps you noticed the sine of π wasn't exactly zero in the example from last lesson. This is caused by machine precision, or the finite representation of numbers in computers. Some languages do not tell the truth regarding this limitation. See for example what Excel/LibreOffice Calc or similar tell you, when you ask `=IF(0,1 + 0,1 + 0,1 = 0,3;"TRUE";"FALSE")`. Compare this to the answer that Python gives:

[7]:
```
print(0.1 + 0.1 + 0.1 == 0.3)
```

```
False
```

This causes some surprising effects, for example cancellation, where subtracting numbers close in value might lead to significant loss of precision. The following example is taken from https://math.stackexchange.com/questions/1920525/why-is-catastrophic-cancellation-called-so. Take for example the polynomial $x^2 - 1000.0x + 1.0 = 0.0$. The midnight formula gives $x_{1/2} = \frac{1000.0 \pm 999.99}{2}$

[4]:
```
# roots:
x1 = (1000 + 999.99)/2
x2 = (1000 - 999.99)/2

print("Analytical roots:\t", x1, "and", x2)

# While the first is almost correct, the second one isn't at all
x1c = 1000
x2c = 0.0010005

print("x1 error:\t\t", 100*(x1 - x1c)/x1c, "%")
print("x2 error:\t\t", 100*(x2 - x2c)/x2c, "%")
```

```
Analytical roots:        999.995 and 0.004999999999954525
x1 error:                -0.0004999999999995453 %
x2 error:                399.7501249370767 %
```

For more information regarding floating point arithmetic and its limitations and implications, see this exhaustive document: https://docs.oracle.com/cd/E19422-01/819-3693/ncg_goldberg.html

The `==` here is a comparison operator. These work quite intuitively. Same applies to gate operators (`^` is `xor`):

[5]:
```
a = 14
b = True

print(11 < 13)
print(11 > 13)
print(11 == 13)
print(11 != a)
print(b)
print(not b)
print(11 < 13 and b)
print(11 < 13 and not b)
print(b or not b)
print((not b) ^ (not b))
```

```
True
False
False
True
True
False
True
False
True
False
```

1.5 Data structures

Many useful data structures are implemented in Python. The ones we will use most often are `lists` and `dicts`. Lists are pretty much self-explanatory, they contain a set of arbitrary things, even other lists, or even themselves:

[20]:
```
a = 7

testlist = []
print(type(testlist))

testlist.append("apple")
testlist.append(15)
testlist.append(a)
testlist.append(testlist)

print(testlist)
print(testlist[-2])
```

```
<class 'list'>
['apple', 15, 7, […]]
7
```

Square brackets access a certain element in a list. You can also access list elements using negative numbers, in which case they'll start referencing elements starting from the last element in a list. You can also use slicings, choosing a range of elements from a list. We will see how these work after getting to know one of the most beautiful concepts in Python; **list comprehensions**:

[25]:
```
import math as m

n = 15

squares = [x**2 for x in range(n)]

list_of_lists = [[x**2, m.sin(y)] for x in range(3) for y in range(3)]

print(squares)
print(list_of_lists)

print("Slicing: ", squares[1:3])
print("Negative elements: ", squares[-1])
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
[[0, 0.0], [0, 0.8414709848078965], [0, 0.9092974268256817], [1, 0.0], [1,
0.8414709848078965], [1, 0.9092974268256817], [4, 0.0], [4, 0.8414709848078965],
[4, 0.9092974268256817]]
Slicing:  [1, 4]
Negative elements:  196
```

`dicts` are an extension of lists to pairs of `keys` and `values`. The notation for them is similar to the `json` format:

[7]:
```
d = {"first": "some text", \
     "second": a, \
     "third": squares}

print(d["first"])
print(d["second"])
print(d["third"])
```

```
some text
7
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

You can print all available keys and values:

[10]:
```
print(d.keys())
print(d.values())
```

```
dict_keys(['first', 'second', 'third'])
dict_values(['some text', 7, [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144,
169, 196]])
```

1.6 Loops

There are two main ways to do loops. Important to notice here is *indentation*. Instead of brackets or other means to mark blocks of code, Python simply discerns these by indentation. Each level of indentation consists of **4 spaces**. This serves readability since you'll instantly see what block certain code belongs to.

[1]:
```
a, b = 0, 1

while b < 10:
    print(b)
    a, b = b, a + b
```

```
1
1
2
3
5
8
```

While it's possible to do multiple assignments per row, this should be avoided for readability.

The second way to do loops is using the powerful "for" contruct. A basic example uses the `range` generator to generate a list of integers to loop over:

[1]:
```
f = range(10, 20)
```

```
print(type(f))
print(*f)
print(*range(0, 5, 2))
print(*range(7, 1, -2))

# our squares list from last lesson
squares = [x**2 for x in range(15)]

for i in range(0, 5):
    print(i, ":", squares[i])
```

```
<class 'range'>
10 11 12 13 14 15 16 17 18 19
0 2 4
7 5 3
0 : 0
1 : 1
2 : 4
3 : 9
4 : 16
```

`for` iterates over all elements of a list:

[3]:
```
for element in squares:
    print(element, end=", ")
```

```
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
```

1.7 Control Structures

It's often necessary to check if a certain condition has been met. This is done using if, elif and else structures:

[3]:
```
a = 15000000

if a <= 20:
    print(a)
elif a > 5000001:
    print("a is really large")
else:
    print("a must be smaller than 21: ", a)
```

```
a is really large
```

Try to avoid loops like this:

[1]:
```
b = 11

while True:
    print(b)
    b -= 1
    if b < 10:
        break
```

```
11
10
```

Why is this a bad idea?

1.8 Functions

Usually, you need a certain functionality more often than once. This functionality can then be defined as a function. Let's use the Fibonacci numbers as an example:

[3]:
```
def fibonacci(n):
    """Print the Fibonacci series up to argument n."""
    a, b = 0, 1

    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

fibonacci(5)
fibonacci(81)
```

```
0 1 1 2 3
0 1 1 2 3 5 8 13 21 34 55
```

There's no check to see if the input `n` is smaller than zero. A better approach would be:

[2]:
```
def fibonacci(n):
    """Print the Fibonacci series up to argument n."""
    if n > 0:
        a, b = 0, 1
        while a < n:
            print(a, end=' ')
            a, b = b, a + b
        print()
    else:
        print("Error: n must be larger than 0")

fibonacci(10)
fibonacci(-10)
```

```
0 1 1 2 3 5 8
Error: n must be larger than 0
```

Functions may also return some values:

[4]:
```
def multiply(a, b):
    return a * b

print(multiply(5, 10))

result = multiply(7, 7)
print(result)
```

```
50
49
```

As an example, let's combine a few techniques here:

[1]:
```
def fibonacciList(n):
    """Get the Fibonacci series up to argument n in a list."""
    if n > 0:
        result = []     # an empty list
        a, b = 0, 1

        while a < n:
            result.append(a)
            a, b = b, a + b

        return result
    else:
        print("Error: n must be larger than 0")
        return []

print(fibonacciList(20))

for fibonacciNumber in fibonacciList(20):
    print(fibonacciNumber, end=', ')
```

```
[0, 1, 1, 2, 3, 5, 8, 13]
0, 1, 1, 2, 3, 5, 8, 13,
```

1.9 File Handling

Although most of the libraries we will use in the course have inbuilt functionality to save and load files, it is sometimes useful to being able to handle files manually.

In Python, `open()` will open a file, whether it is for reading or for writing, and calling `close()` on the variable referencing the file (this is called a *file handler*) will close it again. It's important to make sure that a file is closed after usage to avoid stale file handlers and memory hogging. `open()` has many arguments it can take, we'll just look at the most important ones here. One way to open a file for writing is the following, where the argument `"w"` is provided to indicate that the file should be *overwritten*:

[10]:
```
textlist = ["some", "text in form of", "a", "list"]

fh = open("test.txt", "w")

fh.write("testfile\n")
fh.write("some more text\n")
fh.writelines(textlist)

fh.close()
```

Since `test.txt` did not exist before executing the above cell, it was created. The `write` statements write the provided strings into the file, *without* a newline at the end. `writelines` write a sequence (here, a list) of strings into a file line by line. Let's see what happened by opening the file in read mode with `"r"`:

[11]:
```
filereader = open("test.txt", "r")

print(filereader.read())

filereader.close()
```

```
testfile
some more text
sometext in form ofalist
```

`read` gets every line separately. We can also use `readlines` to get a list of all the lines in the file:

[12]:
```
filereader = open("test.txt", "r")

print(filereader.readlines())

filereader.close()
```

```
['testfile\n', 'some more text\n', 'sometext in form ofalist']
```

For appending to a file instead of overwriting it, the option “a” has to be given for `open()`:

[15]:
```
fh = open("test.txt", "a")

fh.write("A new line\n")
fh.write("without changing the rest\n")

fh.close()
```

Let's check the result:

[16]:
```
filereader = open("test.txt", "r")

print(filereader.read())
```

```
filereader.close()
```

```
testfile
some more text
sometext in form ofalistA new line
without changing the rest
A new line
without changing the rest
```

Sometimes it's useful to iterate oover all the lines in a file:

[19]:
```
filereader = open("test.txt", "r")

for line in filereader:
    print(line, end='' )

filereader.close()
```

```
testfile
some more text
sometext in form ofalistA new line
without changing the rest
A new line
without changing the rest
```

Sometimes it's not clear whether a file already exists and overwriting it would yield catastrophic data loss. For creating new files without overwriting existing files, the option "x" is useful. In this case, an error is thrown since "test.txt" already exist:

[20]:
```
fh = open("test.txt", "x")

fh.write("New file\n")
fh.write("but only if it didn't exist before\n")
fh.writelines(textlist)

fh.close()
```

```
      ␣
↪---------------------------------------------------------------------------

       FileExistsError                           Traceback (most recent call␣
↪last)

       <ipython-input-20-32c15536a01c> in <module>
   ----> 1 fh = open("test.txt", "x")
         2
         3 fh.write("New file\n")
      4 fh.write("but only if it didn't exist before\n")
      5 fh.writelines(textlist)


    FileExistsError: [Errno 17] File exists: 'test.txt'
```

1.10 Classes

Although you don't need to use object-oriented programming in this course, it is good to have a rough understanding of what classes do and how to interact with them in order to understand what some of the modules we will use are doing in the background.

Classes encapsulate functionality. They have attributes and methods. Attributes are what the function "has" or "knows" (variables), methods are what a class can do (functions). Think for example of example of a car. A car has a number of things it "has", like the number of seats, horsepower, … and it also can do something, like accelerating, decelerating, use the brakes, steer, …

Let's use a much simpler example here. A Window has a status that indicates whether it is opened or closed, and it can open, or close. As a class, this looks like the following:

```
[1]: class Window:                # class names are Capitalized! This is convention.
         def __init__(self):
             self.open = False

         def openUp(self):
             if self.open:
                 print("Window already open.")
             else:
                 self.open = True

         def closeDown(self):
             if not self.open:
                 print("Window already closed.")
             else:
                 self.open = False

         def checkWindowStatus(self):
             if self.open:
                 print("Window is open.")
             else:
                 print("Window is closed.")
```

This is like a blueprint for an *instance* of a class, called an **object**. Objects are instantiated from classes similarly to initializing variables:

```
[2]: # here, we instantiate a Window in the variable "window"
     window = Window()
```

```
print(type(window))

# here, we use the methods of the object
window.openUp()
window.openUp()
window.checkWindowStatus()
window.closeDown()
window.checkWindowStatus()
```

```
<class '__main__.Window'>
Window already open.
Window is open.
Window is closed.
```

Classes can also be instantiated with arguments:

[2]:
```
class Window:
    def __init__(self, opened):   # __init__ function must be implemented
        self.open = opened

    def openUp(self):
        if self.open:
            print("Window already open.")
        else:
            self.open = True

    def closeDown(self):
        if not self.open:
            print("Window already closed.")
        else:
            self.open = False

    def checkWindowStatus(self):
        if self.open:
            print("Window is open.")
        else:
            print("Window is closed.")


window = Window(False)
window.checkWindowStatus()
print(window.open)
```

```
Window is closed.
False
```

This uses the inbuilt special function `__init__`, which is called when an instance of a class is created. This function *must always* be implemented, even if you don't initialize your instance with arguments.

You probably noticed the extensive use of `self`. This *must* be the first argument of every method you implement in a class. The reason is that a class is only a blueprint. When you later create an instance called, say, `w1`, and use a method like `w1.closeDown()` in your code, the `self` will contain your object name. This way, Python knows that when you call `w1.closeDown()`, it need to perform all the actions on attributes produced by that method on those attributes belonging to `w1`. This does take some getting used to, so a little bit of practicing with your own ideas for classes goes a long way.

This is hopefully all you need to know to follow along with the lecture and the exercises. Of course, this is in no way a comprehensive Python course and we missed out many of the features and comfortable properties Python offers. Since people proficient in the language are currently highly sought for, it is probably wise to accustom oneself with it further. We will learn everything else we need as we move on.

2 Linear Algebra and Data

2.1 NumPy

NumPy is probably the most widely used module for Python everywhere. It offers a lot of functionality for linear algebra, manipulating vectors, matrices, and some other features. We will take a quick look at the single most useful data structure in Python - numpy arrays - and learn everything else alongside the rest of this lecture. Since NumPy is a module, we need to import it before we can use it. The basic data structure NumPy works with is called a `numpy array`. You can use it pretty much like normal lists in pure Python, but they offer much more functionality. Here are a few things to know:

[2]:
```
import numpy as np

vector = np.array([1, 3, 4])

matrix = np.array([[3, 4, 9],
                   [9, 6, 7],
                   [8, 3, 7]])

print(vector)
print()
print(matrix)
```

```
[1 3 4]

[[3 4 9]
 [9 6 7]
 [8 3 7]]
```

One thing to be super careful about is the type of numpy array that is created. In the above cases, both arrays are saved as integers:

[3]:
```
print(vector.dtype)
print(matrix.dtype)
```

```
int64
int64
```

Numpy determines this from the values the array was instanciated with. To make sure that it has a certain type, you can force it with the `dtype` argument:

[4]:
```
vector = np.array([1, 3, 4], dtype=np.float32)

matrix = np.array([[3.0, 4.0, 9.0],
                   [9.0, 6.0, 7.0],
                   [8.0, 3.0, 7.0]])

print(vector.dtype)
print(matrix.dtype)
```

```
float32
float64
```

NumPy arrays can be used like lists, so slicing and negative indexing work the same.

Sometimes you need to adress a column of a matrix, which you can do like this:

[5]:
```
print(matrix[:,1])
```

```
[4. 6. 3.]
```

2.2 Linear Algebra

Good knowledge of linear algebra is essential in understanding some results and techniques in machine learning settings. In most cases, the input to ML algorithms is expressed as a vector of some values. These values can have any meaning, and the vector doesn't need to have consistent units. Often, a part of the problem is how to express the data you work with as a vector, e.g. image or video data, which in general is qualitative data. We will talk about a few approaches to this problem later in the course. The objects we will be working with are mostly one of the following: * Scalars: single numbers. Examples will be the learning rate $\alpha \in \mathbb{R}$, or the number of hyperparameters $n \in \mathbb{N}$. (*Side note: there are also pseudo-scalars*) * Vectors: arrays of numbers with (in most cases) no meaning whatsoever. Example is describing a pendulum by its position and velocity, which fully determines its state * Matrices: 2d arrays of numbers, also carrying (in most cases) no physical meaning. Examples are weight matrices $\mathbf{W} \in \mathbb{R}^{m \times n}$ * Higher order matrices: nd arrays of numbers. Examples are RGB images, or sensor data from multiple sensors, or video data with time as another degree of freedom

In ML, all of these are called **tensors** (reflected in the name `TensorFlow`) although mathematically, tensors are well-defined geometric quantities and (in most cases) different from what we are using.

Tasks that often arise in ML contexts include: * Turning matrices into vectors and vice versa * Stack physical quantities into a vector (state variables/degrees of freedom) * Unrolling time-series data * Map vectors of different dimension onto each other with non-square matrices

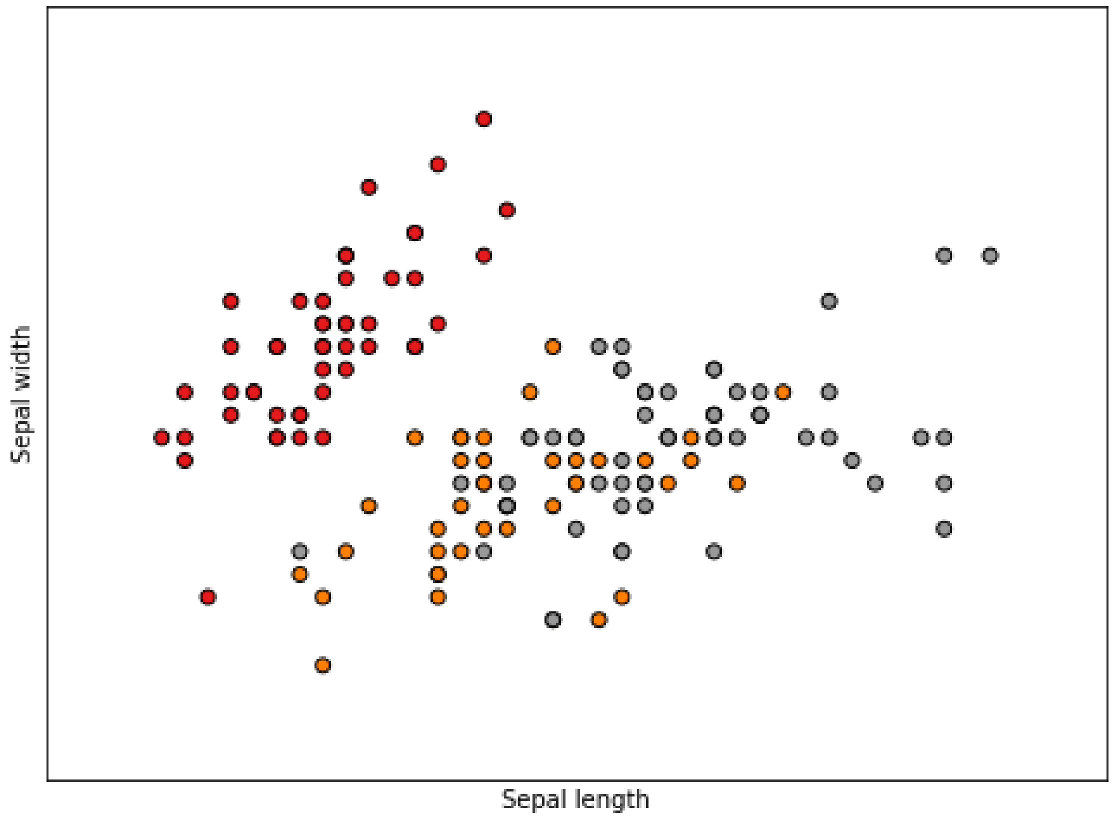
The latter is for example needed for neural networks. Basically, they learn a **mapping** from input data to output data. The neat thing here especially for engineering is that this mapping is in many cases **reversible**.

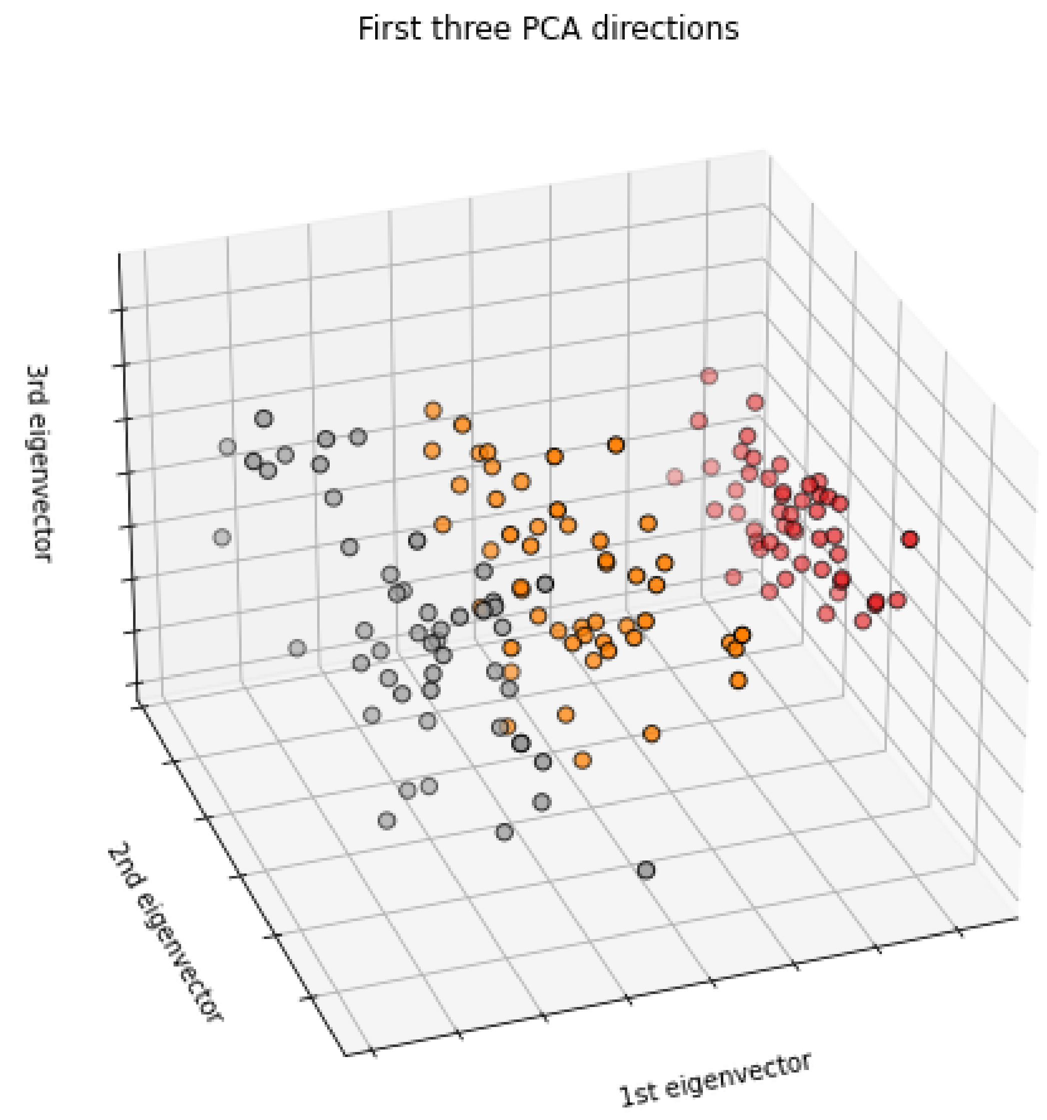
Turning matrices into vectors is necessary for example when processing **image data**. As an example, grayscale images are saved in a computer as matrices of grayscale values, with entries going from 0 to 255 or 0 to 1. To vectorize all the values necessary to describe a, say, 40x40 pixel grayscale image, you need a vector $\mathbf{v}$ of dimension $\dim(\mathbf{v}) = 40 \cdot 40 = 1600$. The dimension of this vector grows quickly with increasing image size. RGB images need three values per pixel, so an image of 40x40 pixels in RGB would need a vector of dimension 4800. Transparency increases this further. Hence, we need algorithms that perform well in these high-dimensional settings.

Representing everything as a vector gives geometrical meaning to data. See for example Anderson's Iris dataset, consisting of lengths and widths of petals and sepals of Iris flowers (you don't need to understand the code here, taken from the scikit-learn tutorial):

[1]:
```
%matplotlib notebook

# Code source: Gaël Varoquaux
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :2]  # we only take the first two features.
y = iris.target

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1,
            edgecolor='k')
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

# To getter a better understanding of interaction of the dimensions
# plot the first three PCA dimensions
fig = plt.figure(1, figsize=(8, 6))
```

```
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=y,
           cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st eigenvector")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd eigenvector")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd eigenvector")
ax.w_zaxis.set_ticklabels([])

plt.show()
```





The dataset itself is 4-dimensional, so in the above example, a reduced version of that dataset was plotted, so it could be represented in 2d and 3d (try rotating this image. It will be slow, but should be possible).

The different colors in the plot correspond to different species of Iris flower. Obviously, geometrical “closeness” of data points here suggests membership to the same species. Often, we want to find a mapping of data such that similar samples of the dataset will be close in a geometrical sense. We will investigate this closer in the lectures about model order reduction and clustering methods.

2.3 Array operations

NumPy offers several handy functions to manipulate arrays or to get some information about them. It is often necessary to get the dimensions of an array:

[2]:
```
import numpy # necessary because of a bug
import numpy as np

vector = np.array([1, 3, 4], dtype=np.float)

matrix = numpy.array([[3.0, 4.0, 9.0],
                      [9.0, 6.0, 7.0],
                      [8.0, 3.0, 7.0]])

print(vector.shape)
print(matrix.shape)
```

```
(3,)
(3, 3)
```

Let's look at a few of the basic operations on NumPy arrays.

For adding, arrays must have the same dimensions:

[4]:
```
A = np.array([[5, 3, 1], [3.0, 2, 7], [1, 3, 0]])
B = np.random.rand(3, 3)
C = np.random.rand(3, 4)

print(A + B)
print(A + C)
```

```
[[5.98790245 3.49984241 1.39192664]
 [3.11040607 2.32102182 7.495645  ]
 [1.37623334 3.06696395 0.62703563]]


       ␣
 ↪---------------------------------------------------------------------------


        ValueError                                Traceback (most recent call␣
 ↪last)


        <ipython-input-4-0e6835c35d69> in <module>
          4
          5 print(A + B)
    ----> 6 print(A + C)


        ValueError: operands could not be broadcast together with shapes (3,3)␣
 ↪(3,4)
```

The error here is desired.

Regarding addition, sometimes `broadcasting` is a handy shortcut. This looks a bit weird notation-wise, as if you would add a vector to a matrix. What actually happens is that the vector is duplicated and put into a matrix such that it can be added to $\mathbf{A}$:

[3]:
```
b = np.ones(3)

print(b)

print(A + b)
print(A + np.ones([3, 3]))
```

```
[1. 1. 1.]
[[6. 4. 2.]
 [4. 3. 8.]
 [2. 4. 1.]]
[[6. 4. 2.]
 [4. 3. 8.]
 [2. 4. 1.]]
```

There are several multiplications that can combine matrices. * Dot/Matrix product $\mathbf{C} = \mathbf{AB} = \sum_k A_{ik}B_{kj} = C_{ij}$ or $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$ or $\mathbf{v}^T\mathbf{w}$ for vectors * Hadamard product $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$, $A_{ij}B_{ij} = C_{ij}$, elementwise multiplication, all matrices must have same dimension * (Frobenius) scalar product $\mathbf{A} : \mathbf{B} = \sum_{ij} A_{ij}B_{ij}$ or $\mathbf{v} : \mathbf{w} = \sum_i v_i w_i$ or $\mathbf{v}^T\mathbf{w}$ for vectors

There are different ways to perform these using NumPy:

[4]:
```
v = numpy.random.rand(3)
w = numpy.random.rand(3)
A = numpy.random.rand(3, 5)
B = numpy.random.rand(5, 4)
C = numpy.random.rand(5, 4)

print("### Matrix product\n")
print(np.matmul(A, B))
print(np.dot(A, B))
print(A @ B)
print(v @ w)
print("\n\n")


print("### Hadamard product\n")
print(np.multiply(B, C))
print(np.multiply(v, w))
print("\n\n")


print("### (Frobenius) scalar product\n")
print(np.sum(np.multiply(B, C)))
print(np.trace(B.T @ C))
print(v.T @ w)
print(np.sum(np.multiply(v, w)))
print("\n\n")
```

```
### Matrix product

[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
0.8412912756661889



### Hadamard product

[[0.17681447 0.78257756 0.27551326 0.11236577]
 [0.07064154 0.16313143 0.5803325  0.02800163]
 [0.13442409 0.46555704 0.62498262 0.3243592 ]
 [0.26878177 0.17939867 0.27650369 0.73702217]
 [0.08314432 0.07775577 0.11926288 0.02877473]]
[0.58947782 0.07911717 0.17269629]



### (Frobenius) scalar product

5.50934512220024
5.50934512220024
0.8412912756661889
0.8412912756661889
```

`A.T` gives the **transpose** of $\mathbf{A}$, $\mathbf{A}^\mathrm{T}$.

We'll also need the **inverse**, which only exists for non-singular square matrices, defined as the Matrix $\mathbf{A}^{-1}$ that satisfies $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$:

[5]:
```
A = np.random.rand(3, 3)

A_inv = np.linalg.inv(A)
```

```
Identity = np.eye(3)

print(Identity)
print(A @ A_inv)
print(A_inv @ A)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[ 1.00000000e+00 -2.27076691e-16  7.71880226e-18]
 [-6.24902931e-17  1.00000000e+00 -2.73750163e-16]
 [-2.88072382e-16  1.59104102e-16  1.00000000e+00]]
[[ 1.00000000e+00 -8.20568143e-16  1.08072375e-16]
 [ 4.47426168e-16  1.00000000e+00 -3.34969881e-16]
 [-1.16315276e-16 -9.81591787e-17  1.00000000e+00]]
```

Note again the numerical zeros, as mentioned in the Python introduction.

In some cases, defining the **pseudoinverse** of a matrix can be useful. The pseudoinverse $\mathbf{A}^+$ is defined for non-square matrices $\mathbf{A}$ in a way that generalizes the concept of inverses of square matrices. There is no unique generalization. The most commonly used pseudoinverse is the Moore-Penrose inverse, which is defined such that it solves the least-squares problem $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{x} = \mathbf{A}^+\mathbf{b}$. You can calculate it via numpy:

[6]:
```
A = np.random.rand(5, 3)

A_plus = np.linalg.pinv(A)

print(A.shape)
print(A_plus.shape)
print(A_plus @ A)
print(A @ A_plus)
```

```
(5, 3)
(3, 5)
[[ 1.00000000e+00  1.11664759e-16  1.05540083e-16]
 [-1.78234610e-16  1.00000000e+00  4.93066658e-18]
 [-8.81678620e-17  2.32567099e-16  1.00000000e+00]]
[[ 0.47608845  0.00462968  0.48928445 -0.09398551  0.03426745]
 [ 0.00462968  0.9933698   0.0119251   0.07391415 -0.03097246]
 [ 0.48928445  0.0119251   0.50298549 -0.0924462   0.04362692]
 [-0.09398551  0.07391415 -0.0924462   0.17254859  0.346313  ]
 [ 0.03426745 -0.03097246  0.04362692  0.346313    0.85500767]]
```

As you can see, only the first matrix multiplication yields an identity matrix. The pseudoinverse here is a **left inverse**. We will see this distinction again when discussing singular value decomposition.

2.4 Linear combinations

A **linear combination w** of a set $V = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ of n vectors $\mathbf{v}_i$ is their weighted sum

$$\mathbf{w} = \sum_i \alpha_i \mathbf{v}_i \tag{1}$$

In FEM, the solution to a partial differential equation (PDE) for some field u (say, displacements) is calculated as the linear combination of the shape functions N^I at each node I, where the coefficients $\hat{u}$ are the degrees of freedom calculated during the analysis:

$$u = \sum_i \hat{u}_i N_i^I \tag{2}$$

In **block matrix notation**, a linear combination of two vectors $\mathbf{w} = 3\mathbf{v}_1 + 2\mathbf{v}_2$ with $\mathbf{v}_1 = \begin{bmatrix}1\\2\end{bmatrix}$ and $\mathbf{v}_2 = \begin{bmatrix}4\\2\end{bmatrix}$ can be written as

$$\mathbf{w} = \begin{bmatrix}\mathbf{v}_1 & \mathbf{v}_2\end{bmatrix}\begin{bmatrix}3\\2\end{bmatrix} \tag{3}$$

$$= \begin{bmatrix}1 & 4\\2 & 2\end{bmatrix}\begin{bmatrix}3\\2\end{bmatrix} \tag{4}$$

So matrix multiplication can be interpreted as a linear combination of the matrix columns.

2.5 Span

The **span** of a set of vectors is the set of all the vectors that can be built by a linear combination of the given vectors. Say we have $\mathbf{e}_1 = [1; 0]$ and $\mathbf{e}_2 = [0; 1]$:

[4]:
```
%matplotlib notebook

import numpy # necessary due to a bug
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)

#fig.patch.set_visible(False)
#ax.axis('off')

ax.set_xlim([-0.1, 2.0])
ax.set_ylim([-0.1, 2.0])

e1 = np.array([1, 0])
e2 = np.array([0, 1])
```

```
v = np.array([e1, e2]) @ np.array([0.7, 1.5])

V = np.array([e1,e2,v])
origin = np.array([0, 0])


ax.arrow(*origin, *e1, head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *e2, head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)

ax.arrow(*origin, *v, head_width=0.05, head_length=0.1, \
         fc='darkorange', ec='darkorange', length_includes_head=True)


plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

We can obviously express every 2d vector as a linear combination of $\mathbf{e}_1$ and $\mathbf{e}_2$, so the **span** of these two vectors is the whole of $\mathbb{R}^2$. This extends naturally to higher dimensions. The span of the vectors that make up the columns of a matrix is called its **column space**. $\mathbf{Ax} = \mathbf{b}$ has a solution only, if $\mathbf{b}$ lives in the column space of $\mathbf{A}$. The solution $\mathbf{x}$ then also lives in the column space of $\mathbf{A}$.

2.6 Linear independence

Let's look at the set of vectors V from above again. If none of the vectors in this set can be written as a linear combination of the other vectors, the set is called **linear independent**. Expressed mathematically, this means

$$\sum_i \alpha_i v_i = 0 \implies \alpha_i = 0 \quad \forall\, i, \tag{5}$$

so the only way this weighted sum can vanish is to make all coefficients α_i zero.

2.7 Matrix operations

We will often transform high-dimensional vectors into even higher-dimensional vectors and back, so it's imperative to think about how to get a smaller set of representative numbers that give us some information about what is going on. Examples discussed here include

norms: assign single values to matrices (see chapter about norms).

trace: the sum of the diagonal elements of a matrix $\operatorname{tr}(\mathbf{A}) = \sum_i A_{ii}$. This extends to non-square matrices, still summing over diagonal elements, ignoring every other element. Some useful properties: * $\operatorname{tr}(\mathbf{A}+\mathbf{B}) = \operatorname{tr}(\mathbf{A}) + \operatorname{tr}(\mathbf{B})$ * $\operatorname{tr}(\mathbf{AB}) = \operatorname{tr}(\mathbf{BA})$ * $\operatorname{tr}(\mathbf{A}) = \operatorname{tr}\left(\mathbf{A}^{\mathrm{T}}\right)$

[1]:
```
import numpy # necessary due to a bug
import numpy as np

A = np.array([[1, 3, 7],
              [4, 7, 11],
              [12, 6, 9]])
B = np.array([[1, 3, 7],
              [4, 7, 11],
              [12, 6, 9],
              [4, 7, 11]])
C = np.random.rand(4, 3)

print("dim(A): ", A.shape)
print("dim(B): ", B.shape)
print("dim(C): ", C.shape)
print()
print("tr(A) = ", np.trace(A))
print("tr(B) = ", np.trace(B))
print("tr(B^T) = ", np.trace(B.T))
print()
print("tr(BC^T) = ", np.trace(B @ C.T))
print("tr(CB^T) = ", np.trace(C @ B.T))
```

```
dim(A):  (3, 3)
dim(B):  (4, 3)
dim(C):  (4, 3)

tr(A) =  17
tr(B) =  17
tr(B^T) =  17

tr(BC^T) =  58.070677983665064
tr(CB^T) =  58.070677983665064
```

determinant: represents the volume of the parallelepiped/parallelotope (the nd equivalent of a 2d rhombus/diamond shape) spanned by the column vectors of a matrix. We won't go into the details of how to calculate it here. Some useful properties are * $\det\left(\mathbf{I}\right) = 1$, $\det\left(\mathbf{A}^T\right) = \det\left(\mathbf{A}\right)$ * $\det\left(\mathbf{A}^{-1}\right) = \frac{1}{\det(\mathbf{A})}$ * $\det\left(\alpha\mathbf{A}\right) = \alpha^n \det\left(\mathbf{A}\right)$ for $n \times n$-matrices $\mathbf{A}$ * $\det\left(\mathbf{AB}\right) = \det\left(\mathbf{A}\right)\det\left(\mathbf{B}\right)$ for square matrices $\mathbf{A}$ and $\mathbf{B}$ of equal dimension * There's also a "pseudo - triangle inequality": $\det\left(\mathbf{A}+\mathbf{B}\right) \geq \det\left(\mathbf{A}\right) + \det\left(\mathbf{B}\right)$.

Below is an example of how the determinant calculates volume (area in 2d) of a rhombus spanned by vectors. You don't need to understand the code here.

[2]:
```
%matplotlib notebook

import matplotlib.pyplot as plt
```

```
# plot a rhombus in 2d and calculate its volume
import matplotlib.patches as patches

fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-1.0, 1.0)

# vectors spanning the rhombus
v1 = np.array([0.5, 0.5])
v2 = np.array([1.0, 0.25])
origin = np.array([0, 0])

# we need the coordinates of the endpoints
ax.add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], color="darkorange"))

# vectors spanning the rhombus
ax.arrow(*origin, *v1, head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *v2, head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)

# diagonals, used for geometrical calculation of the area
ax.arrow(*origin, *(v1+v2), fc='k', ec='lightgray', ls='--')
ax.arrow(*v2, *(v1-v2), fc='k', ec='lightgray', ls='--')


plt.show()

# diagonals for plotting
d1 = np.linalg.norm(v1+v2, ord=2)
d2 = np.linalg.norm(v1-v2, ord=2)

# there are two equivalent ways to calculate area geometrically
# 1. using the cross product
print("Area using the cross product: ", np.cross(v1, v2))

# 2. using A = |v_1| |v_2| sin(\theta)
av1 = np.linalg.norm(v1)
av2 = np.linalg.norm(v2)
  = np.arccos(np.dot(v1, v2) / (av1*av2))
print("Area using the absolute form of the cross product: ", av1 * av2 * np.
 ↪sin( ))

# matrix containing the vectors as columns
V = np.array([v1, v2])
print("Area using the determinant of V: ", np.linalg.det(V))
```

```
#print("Why is the determinant negative here?")

B = np.random.rand(3, 4)

# print("det(B) = ", np.linalg.det(B)) # does this work? why or why not?
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


Area using the cross product:  -0.375
Area using the absolute form of the cross product:  0.37500000000000017
Area using the determinant of V:  -0.375
Why is the determinant negative here?
```

inverse: if $\det(\mathbf{A}) \neq 0$, the inverse $\mathbf{A}^{-1}$ of a square matrix $\mathbf{A}$ exists such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. A nonvanishing determinant indicates that the columns of the matrix are linear independent, or that the column space of the $n \times n$ matrix $\mathbf{A}$ spans the whole $\mathbb{R}^n$.

Intuitively, a matrix with a column space that isn't linear independent will transform a vector into a vector inside its column space. Since the dimension of this column space is lower than the dimension of the vector that is multiplied by the matrix, all components perpendicular to the column space will become 0. So, after applying the matrix, we lose all information about how the vector was embedded in the original vector space. Hence, the transformation is **not reversible**.

Let's talk about a few examples where this happens in FEM. The resulting deformations of a body, when the stiffness matrix is singular (and boundary conditions are set correctly) are called **zero modes**. A simple example of zero modes are **rigid body motions**, where your model just translates away or rotates indefinitely. How would you reverse such a motion? Since the motion is regular and unperturbed, you cannot derive the starting point or starting configuration of this kind of motion by examining the solution. This information is lost. Further examples are **locking** and **hourglassing**, which both cause internal deformations that do not cost **energy**. We will talk about zero modes/mode collapse in ML algorithms later.

[3]:
```
# extremely rarely, this will produce a singular matrix
A = np.random.rand(3, 3)
A_inv = np.linalg.inv(A)

print(A @ A_inv)
print(A_inv @ A)
```

```
[[ 1.00000000e+00 -2.35699587e-17 -5.94954288e-16]
 [-2.66047701e-16  1.00000000e+00 -1.09499673e-16]
 [ 2.02447976e-16  1.42945072e-16  1.00000000e+00]]
[[ 1.00000000e+00  4.49112168e-16  6.42398285e-16]
 [-3.29809212e-16  1.00000000e+00 -9.65556270e-16]
 [-6.71502601e-16 -2.15384020e-16  1.00000000e+00]]
```

2.8 Norms

Often, we need a way to determine the "size" of some quantity. That's what norms are useful for. From a mathematical perspective, a norm must satisfy the following properties: * $f(x) = 0 \implies x = 0$ (The only tensor of length 0 is the zero tensor) * $f(x+y) \leq f(x) + f(y)$ (triangle inequality) * $f(\alpha x) = |\alpha|\, f(x)$ (pseudo-linearity)

The most common norms all stem from the `p-norm`: $||\mathbf{v}||_p = (v_1^p + v_2^p + \cdots + v_n^p)^{\frac{1}{p}}$. For example: * $p = 1$: absolute sum norm * $p = 2$: Euclidean norm * $p \to \infty$: infinity/max norm

Let's implement the p-norm and see what it yields for different values of p:

```
[1]: import numpy as np

     v = np.array([1.0, 3.0, 7.0, 14.0])

     def pnorm(p, vector):
         return np.sum(vector**p)**(1/p)

     for p in range(1, 10):
         print(str(p), "norm: ", pnorm(p, v))

     print()
     print("max: ", max(v))
```

```
1 norm:  25.0
2 norm:  15.968719422671311
3 norm:  14.604477265746313
4 norm:  14.220935686711067
5 norm:  14.087665515874415
6 norm:  14.036446523396748
7 norm:  14.015614172122747
8 norm:  14.00683203817856
9 norm:  14.003037039787143

max:  14.0
```

We see why the infinity norm is also called the max norm.

Extending these norms to matrices is straightforward by applying the definition to each matrix element, but there are more possibilities for matrix norms.

One important norm is the Frobenius norm defined as $||\mathbf{A}||_\text{F} = \left(\sum_{ij} A_{ij}^2\right)^{\frac{1}{2}}$. Note that this is the natural extension of the 2-norm of vectors.

Norms also give rise to a notion of distance. To quantify how close two vectors are to each other, a norm can be applied to the difference between them $||\Delta\mathbf{v}|| = ||\mathbf{v}_2 - \mathbf{v}_1||$. Let's say you have images of hulls of planes that you want to evaluate for crack detection. Representing the images as vectors should in the end yield the result that images of cracks are "closer" to each other than images of unharmed areas.

2.9 Quadratic forms

Quadratic forms are, in a certain sense, "weighted" versions of the notion of length we introduced with the Euclidean norm. Sometimes, some components are more "important" than others, so a scaling makes sense to give them more "weight" or "importance" relative to other components. The generalized version of the scalar product as a quadratic form is

$$\mathbf{x}^T\mathbf{A}\mathbf{x} = \sum_{ij} x_i A_{ij} x_j = x_1 A_{11} x_1 + x_1 A_{12} x_2 + \cdots \tag{6}$$

Some properties follow from the numerical values of the eigenvalues of a matrix: * **positive definite**: $\lambda_i > 0\ \forall i$ * **positive semi-definite**: $\lambda_i \geq 0\ \forall i$ * **negative definite**: $\lambda_i < 0\ \forall i$ * **negative semi-definite**: $\lambda_i \leq 0\ \forall i$

For a positive definite matrix $\mathbf{A}$, $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ is true for all $\mathbf{x} \neq 0$. For a positive semi-definite matrix, the $>$ changes to $\geq$, so there exist non-zero vectors with zero weighted length. Think for example of the identity matrix, but set one row to all zeros. All vectors with only a component in that row will be mapped to zero. Remember here what we learned about the column space of a matrix and its span. Such a vector would live in a space exactly **perpendicular** to the column space of such a modified identity matrix. Similar corollaries follow for negative (semi-)definiteness.

This has various applications. The matrix $\mathbf{A}$ is called a **metric** under certain conditions, and gives rise to a notion of distance on curved manifolds. This has use cases for example in relativity or curved finite elements, such as shells.

2.10 Data Visualization with Matplotlib

Or rather, its `pyplot` interface. `matplotlib` is a collection of tools for 2D (and restricted 3D) visualization under Python. It's 2D capabilities are quite excessive. See for example the gallery of examples here. There's also a whole lot of code available on the web if you're searching for a specific issue you have and it's likely that someone had your exact problem before.

Matplotlib has a module for visualizations in Jupyter notebooks which can be activated with the magic command `%matplotlib notebook`. We'll start with the normal mode of visualization here, which can be activated with `%matplotlib inline`:

[2]:
```python
%matplotlib inline
import matplotlib.pyplot as plt
```

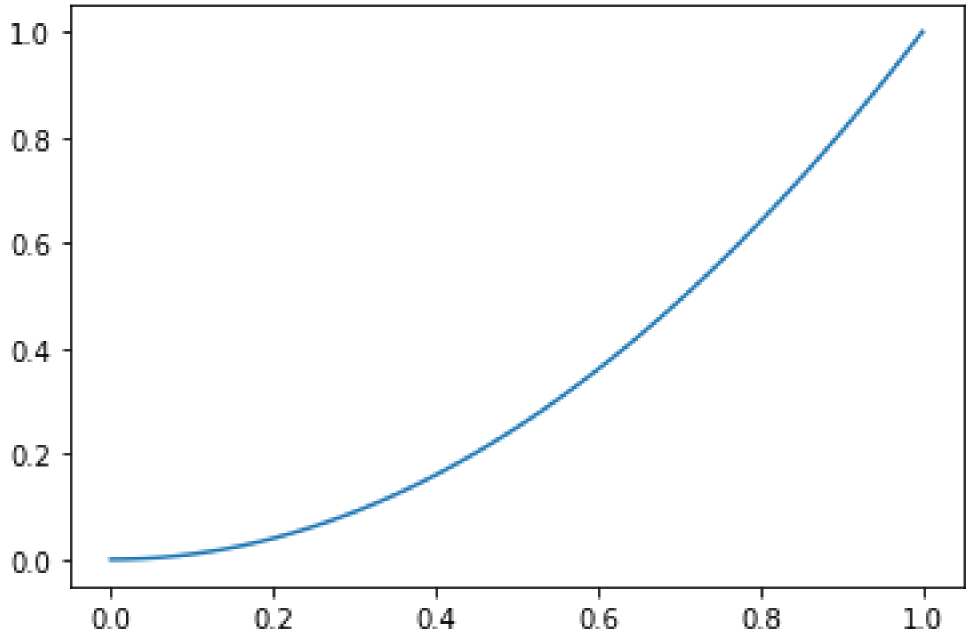
Much more often than not, functions from `numpy` are needed to generate the data necessary for plotting.

[3]:
```python
import numpy as np
```

Let's look at a simple example: $f(x) = x^2$. The way this is done in Python is similar to how matlab works. You need an array of plot points, which are taken as input for the function to plat, and an array of outputs, which are the y-values for the plot:

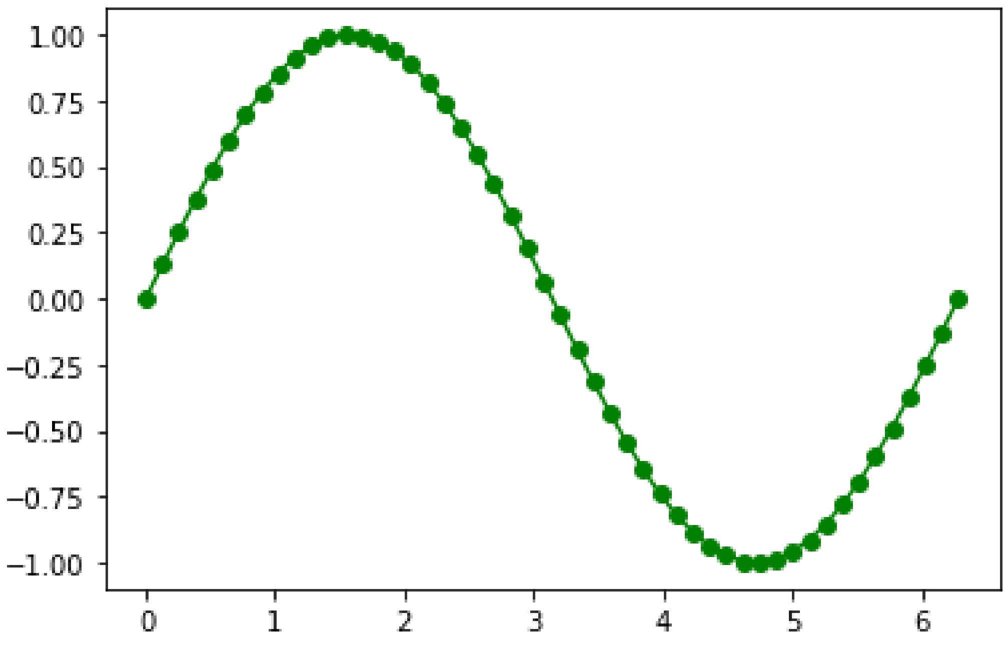
[4]:
```python
# create 50 points in the specified range
x = np.linspace(0, 1, 50)
```

```
# create a plot with those 50 points
plt.plot(x, x**2)

plt.show()  # plt.show() can be omitted in interactive mode (%matplotlib␣
↪notebook)
```



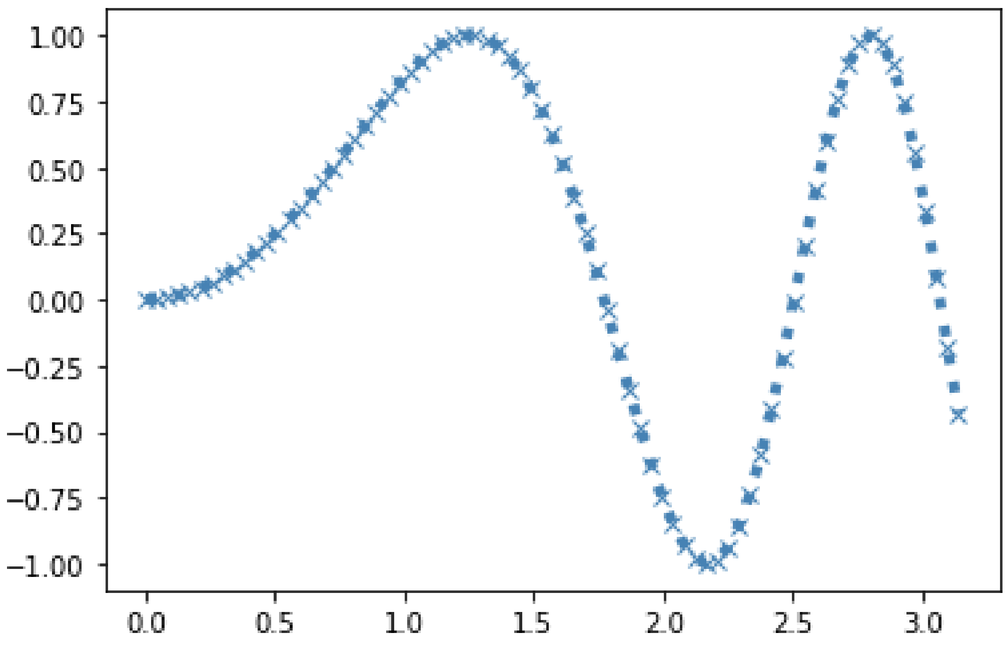
The line style can be changed significantly by providing more parameters (a comprehensive list can be found in the documentation). A simple way to do so are *format strings*:

[12]:
```
x = np.linspace(0, 2*np.pi, 50)
plt.plot(x, np.sin(x), 'go-')

plt.show()
```



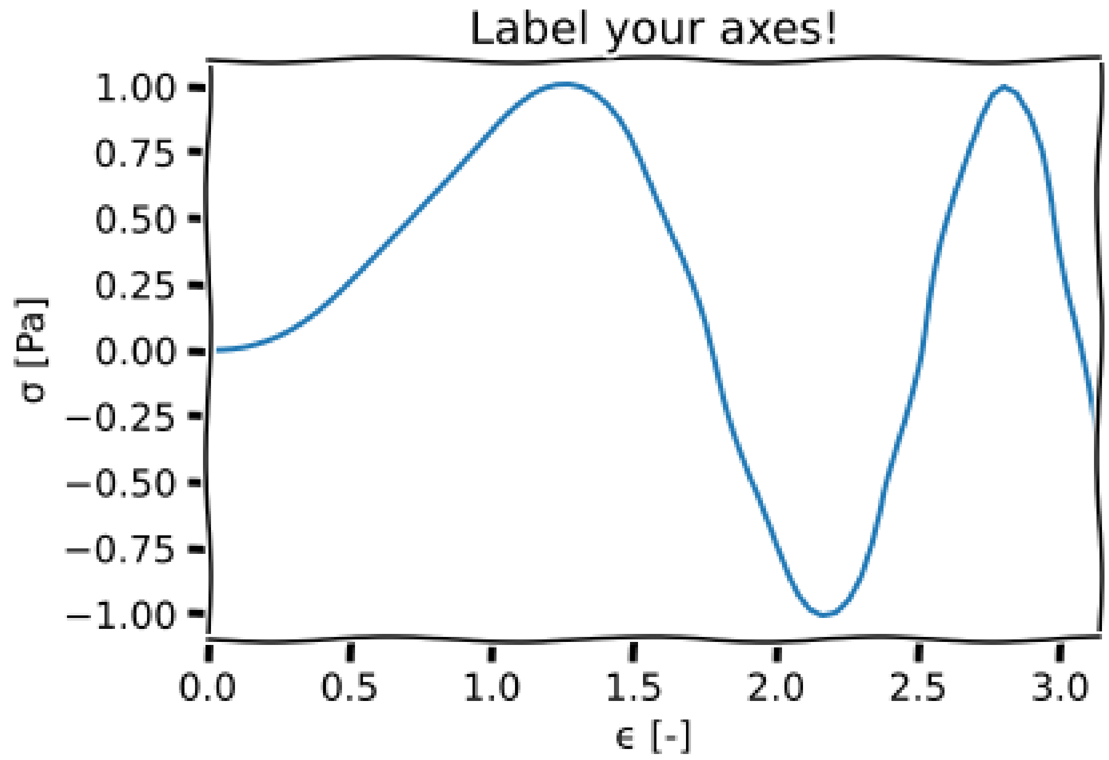
These format strings are shortcuts for other parameters which are commonly used:

[17]:
```
x = np.linspace(0, np.pi, 75)
plt.plot(x, np.sin(x**2), linewidth=4, color="steelblue", marker="x",␣
 ↪markersize=6, linestyle=":")

plt.show()
```



There's one important thing missing here (obligatory xkcd):

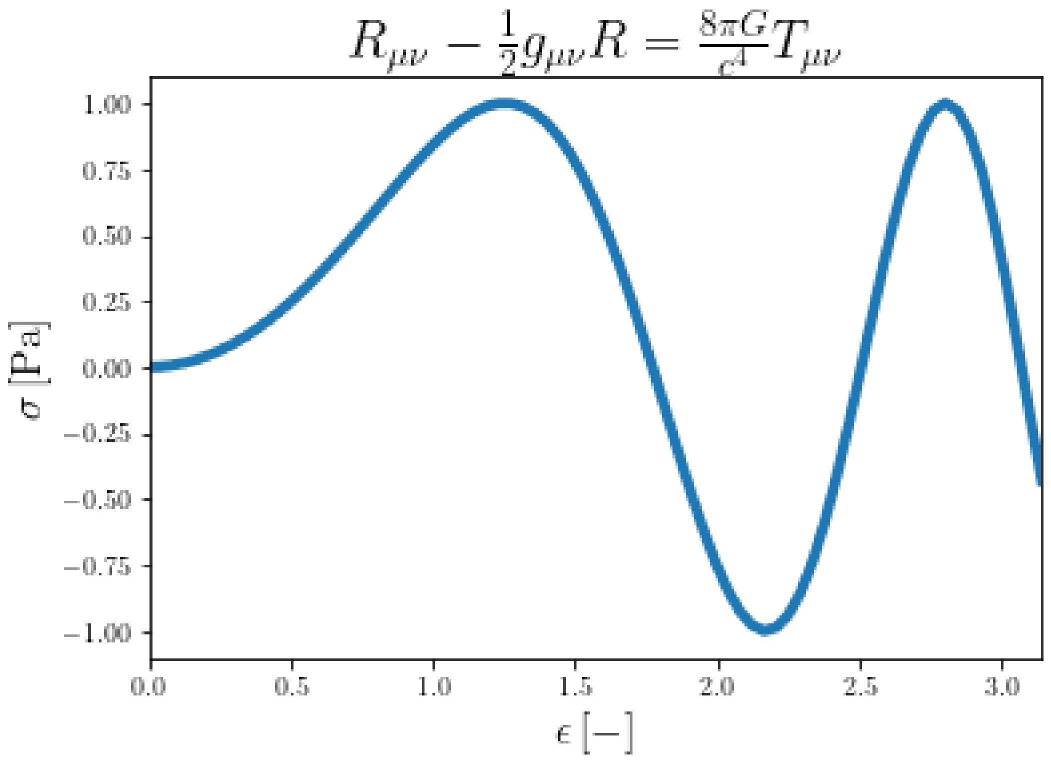
[18]:
```python
with plt.xkcd():
    plt.plot(x, np.sin(x**2))

    # set a title for your plot
    plt.title('Label your axes!')

    # label axes
    plt.xlabel('  [-]')
    plt.ylabel('  [Pa]')

    # set custom plot ranges
    plt.xlim(0, np.pi)
    plt.ylim(-1.1, 1.1)
```



Matplotlib also supports LaTeX rendering with the following global options set (a ltex distribution must be installed on your system):

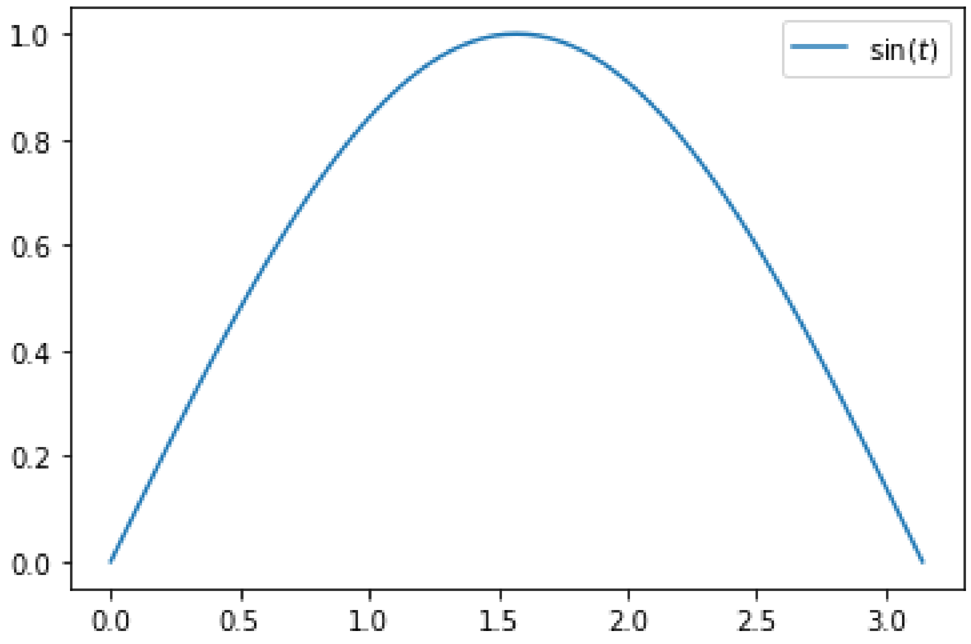
[27]:
```
from matplotlib import rc
# you can also set different fonts here:
#rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
rc('text', usetex=True)
```

[76]:
```
plt.plot(x, np.sin(x**2), linewidth=4)

# set a title for your plot
plt.title(r'$R_{\mu\nu} -\frac 1 2 g_{\mu\nu}R = \frac{8\pi␣
 ↪G}{c^4}T_{\mu\nu}$', fontsize=20)

# label axes
plt.xlabel(r'$\epsilon \, [\mathrm{-}]$', fontsize=16)
plt.ylabel(r'$\sigma \, [\mathrm{Pa}]$', fontsize=16)

# set custom plot ranges
plt.xlim(0, np.pi)
plt.ylim(-1.1, 1.1)

plt.show()
```



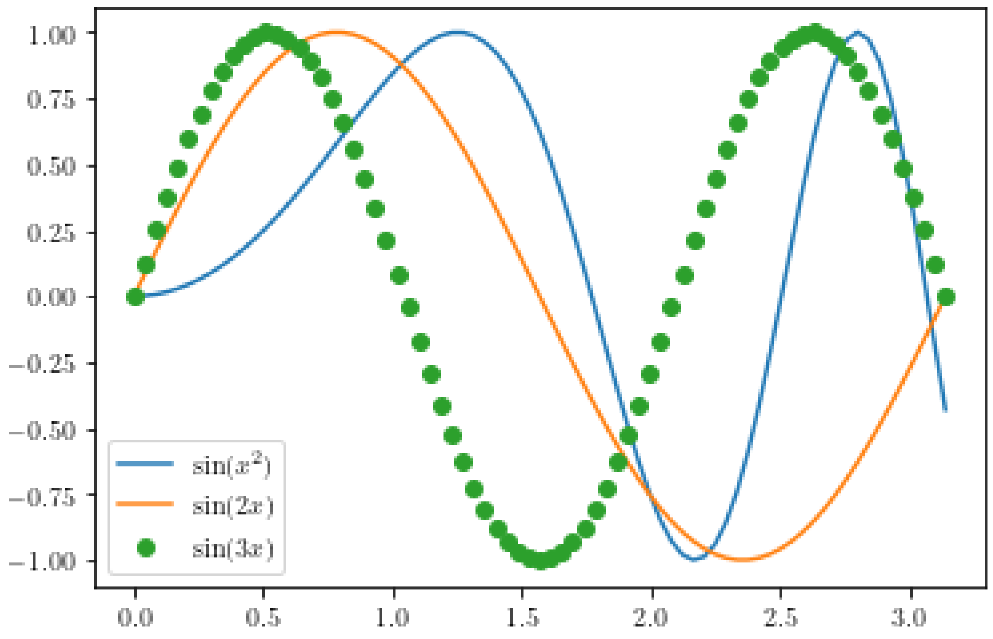
You can also place a legend in your plot:

[25]:
```
plt.plot(x, np.sin(x), label=r'$\sin(t)$')
plt.legend()
#plt.legend(loc='lower left')
#plt.legend(loc='best')

plt.show()
```



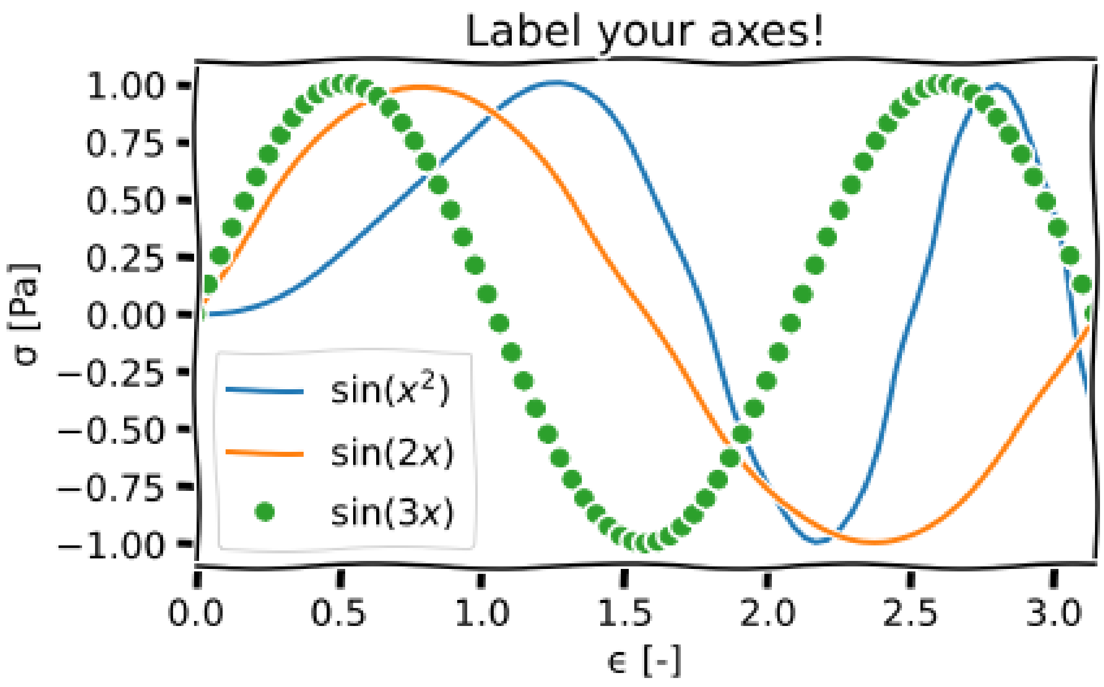
Legends usually only make sense when multiple curves are presented in a single plot:

[29]:
```
rc('text', usetex=True)

plt.plot(x, np.sin(x**2), label=r'$\mathrm{sin}(x^2)$')
plt.plot(x, np.sin(2*x), label=r'$\mathrm{sin}(2x)$')
plt.plot(x, np.sin(3*x), 'o', label=r'$\mathrm{sin}(3x)$')

plt.legend()

plt.show()
```



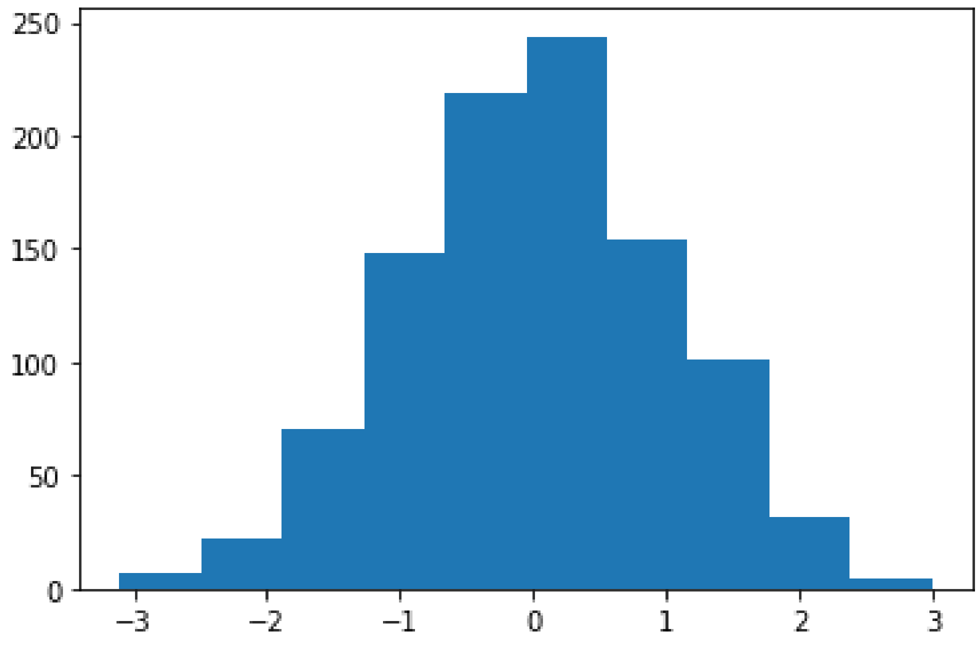
Saving a plot is easy. The file type is determined by the file suffix:

[31]:
```
rc('text', usetex=False)

with plt.xkcd():
    plt.plot(x, np.sin(x**2), label=r'$\mathrm{sin}(x^2)$')
    plt.plot(x, np.sin(2*x), label=r'$\mathrm{sin}(2x)$')
    plt.plot(x, np.sin(3*x), 'o', label=r'$\mathrm{sin}(3x)$')

    # set a title for your plot
    plt.title('Label your axes!')

    # label axes
    plt.xlabel('  [-]')
    plt.ylabel('  [Pa]')

    # set custom plot ranges
    plt.xlim(0, np.pi)
    plt.ylim(-1.1, 1.1)

    # plot legend
    plt.legend()

    # sometimes the following is necessary, otherwise label might be truncated
    plt.tight_layout()
    plt.savefig('weird_plot.pdf')
```



Sometimes *histograms* can yield some insight into statistical data:

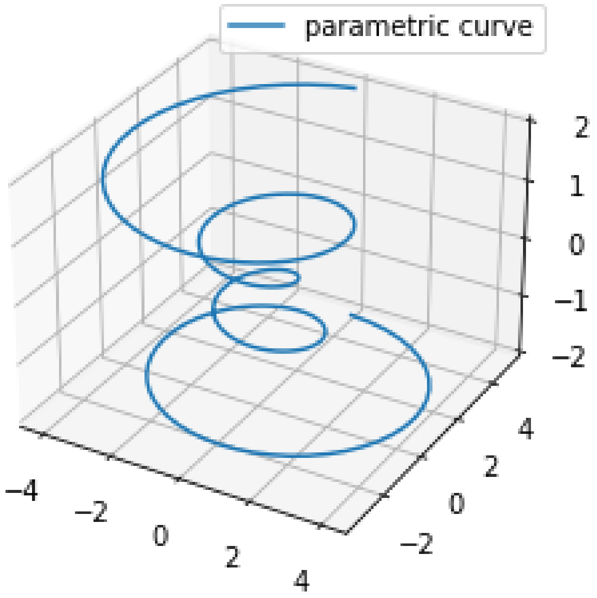
[103]:
```
# random data in this case
x = np.random.normal(0, 1, 1000)
plt.hist(x)

plt.show()
```



2.11 3D Plotting

Matplotlib's 3D capabilities are fairly restricted. It's not true 3D rendering you're seeing, but projections. This causes some objects not to be in the foreground, although they should clearly be in the foreground, among other problems. It's still quite useful in many situations. This is also a situation where it makes sense to use matplotlib's interactive mode, since it allows us to manipulate the 3D plot:

```
[2]: %matplotlib notebook
     from mpl_toolkits.mplot3d import Axes3D

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

       = np.linspace(-4 * np.pi, 4 * np.pi, 200)
     z = np.linspace(-2, 2, 200)

     r = z**2 + 1
     x = r * np.sin( )
     y = r * np.cos( )

     ax.plot(x, y, z, label='parametric curve')
     ax.legend()

     plt.show()
```



You can find this example among other interesting things in the documentation.

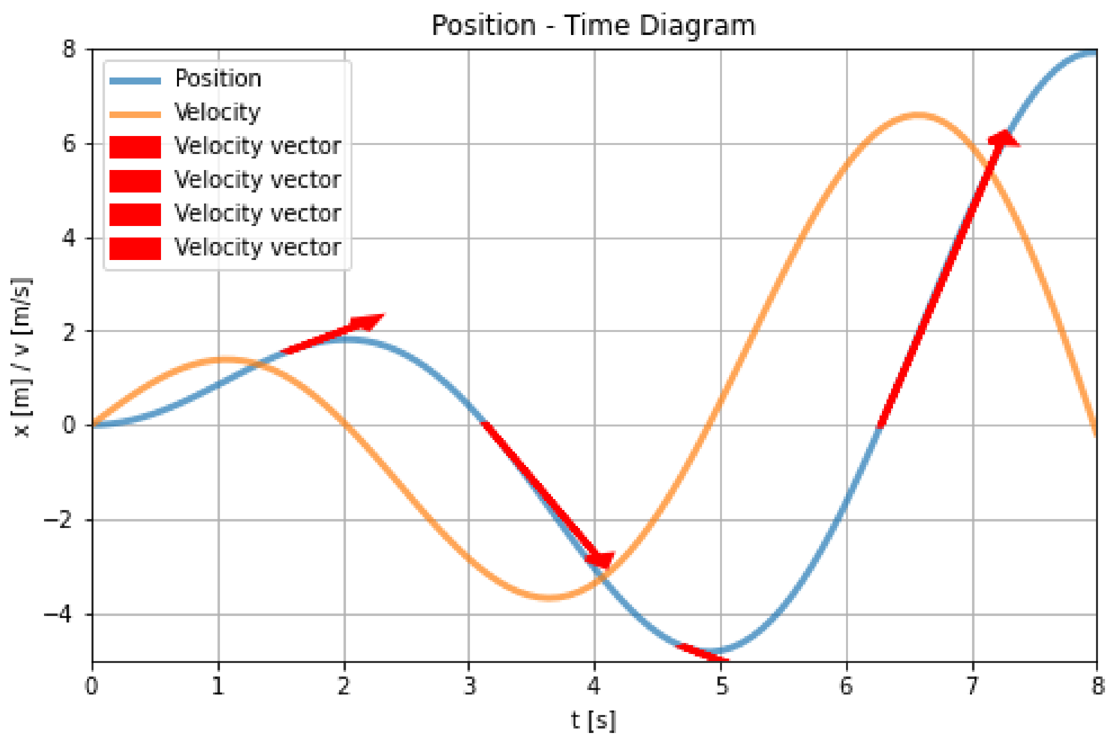
2.12 Derivatives

Most optimization techniques involve taking some derivative, so we'll shortly review different derivatives and their implications and interpretations in this section.

Often, a derivative is described as measuring how much something changes when there is a small change in another quantity. See for example this image:

[1]:
```
%matplotlib notebook
import sympy as sy
import numpy as np
from sympy.functions import sin,cos
from math import factorial
import matplotlib.pyplot as plt

# outsource plotting arrows in a function for efficiency
def plot_arrow(origin, slope):
    # need to cast to float, since sympy outputs a special type
    slope = float(slope)
      = np.arctan(slope)
    vec = [abs(slope)*np.cos( ), abs(slope)*np.sin( )]
    plt.arrow(*origin, *vec, head_width=0.2, head_length=0.2, \
              fc='red', ec='red', lw=3, length_includes_head=True, \
              zorder=10, label="Velocity vector")
```

```
# sympy needs all symbols that will be used declared
x = sy.Symbol('x')
# sympy functions must consist of sympy-compatible expressions
f = sin(x)*x
# this gets the analytical first derivative of a sympy-function
f_prime = f.diff(x,1)

# set up the sampling space for the curves
x_lims = [0, 8]
x_range = np.linspace(x_lims[0], x_lims[1], 100)

# values for the curves
y1 = np.array([f.subs(x,x_val) for x_val in x_range])
y2 = np.array([f_prime.subs(x,x_val) for x_val in x_range])

# plot the curves
plt.figure(figsize=([8,5]))
plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)
plt.plot(x_range,y2,label='Velocity', lw=3, alpha=0.7)

# plot a velocity arrow at multiples of  /2
for i in range(1, 5):
    plot_arrow([i*np.pi/2,f.subs(x,i*np.pi/2).evalf()], f_prime.subs(x,i*np.pi/
 ↪2).evalf())

# some plot options
plt.xlim(x_lims)
plt.ylim([-5,8])
plt.xlabel('t [s]')
plt.ylabel('x [m] / v [m/s]')
plt.legend()
plt.grid(True)
plt.title('Position - Time Diagram')
plt.show()
```



A more general definition would be that a derivative (of first order) is a **linear approximation** of a function at a specific point. This geometrical interpretation is often lost since for many functions, the derivative is only calculated as a number. See for example the following graph, where at $x = 3$ the derivative is plotted as a **tangent** to the curve. Use the slider to zoom closer to the point, where the tangent is attached.

[4]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

def plot_tangent(origin, slope, length):
    slope = float(slope)
      = np.arctan(slope)
    #vec = [abs(slope)*np.cos( ), abs(slope)*np.sin( )]
    vec = [length*np.cos( ), length*np.sin( )]
    origin = [origin[i] - vec[i] for i in range(len(origin))]
    vec = [2*i for i in vec]
    plt.arrow(*origin, *vec, head_width=0, head_length=0, \
              fc='red', ec='red', lw=3, length_includes_head=True, \
              zorder=10, label="Velocity vector")

def plot_zoomable_function(factor, x_val):
    x = sy.Symbol('x')
    f = sin(x)*x
    f_prime = f.diff(x,1)

    x_lims = [max(0,(1-factor)*x_val), min((1+factor)*x_val,8)]
    x_range = np.linspace(x_lims[0], x_lims[1], 50)
    y1 = [f.subs(x,x_v) for x_v in x_range]

    plt.figure()
    plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)

    plot_tangent([x_val,f.subs(x,x_val).evalf()], f_prime.subs(x,x_val).
 ↪evalf(), 0.5*(x_lims[1] - x_lims[0]))

    plt.xlim(x_lims)
    plt.ylim([-5,8])
    plt.xlabel('t [s]')
    plt.ylabel('x [m]')
    plt.legend()
    plt.grid(True)
    plt.title('Position - Time Diagram')
    plt.show()

interact(plot_zoomable_function, \
         factor=widgets.FloatSlider(min=0.1, max=2, value=1,␣
 ↪continuous_update=False), \
         x_val=widgets.FloatSlider(min=2, max=7, value=3,␣
 ↪continuous_update=False));
```

```
interactive(children=(FloatSlider(value=1.0, continuous_update=False, description='factor', ma
```

When zooming in you will see that the curve resembles that tangent line ever more closely. This is what is meant with “the (first-order) derivative approximates functions linearly”. This is where the notion of **manifold** stems from. Manifolds are in general *curved* n-dimensional point sets that resemble flat $\mathbb{R}^n$ when you zoom in close enough. Our earth for example, when viewed from space, is roughly a ball (oblate spheroid, actually looks more like a potato). Yet for us small beings viewing earth from the surface, it looks rather flat.

The derivative of a function $f(x)$ is defined as

$$f'(x=p) = \frac{\partial f}{\partial x}(x=p) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}(x=p)\ , \tag{7}$$

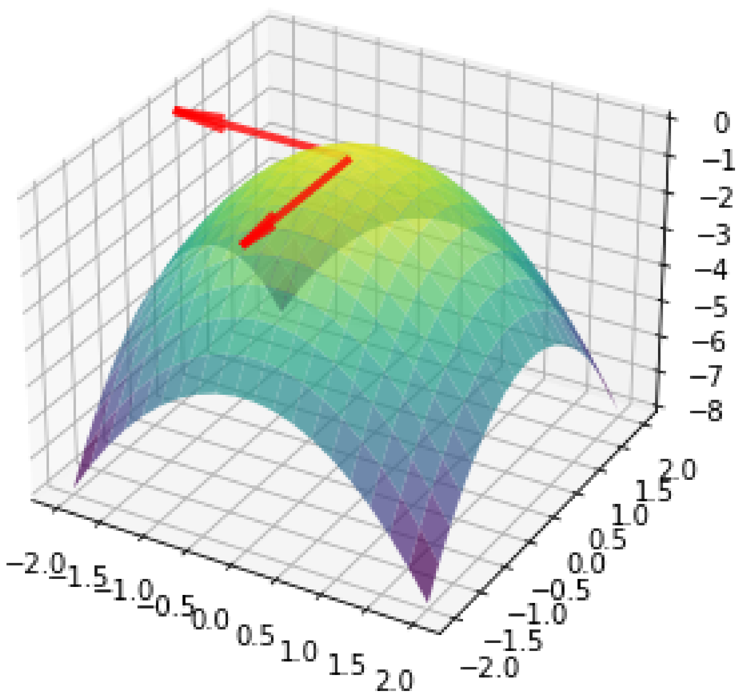
where $\frac{\partial}{\partial x}$ denotes the **partial derivative** with respect to x. For multivariate functions $f(\mathbf{x})$ in $\mathbb{R}^n$, there are n partial derivatives $\frac{\partial}{\partial x_i}$, each for one of the n coordinates x_i of f.

$$\frac{\partial f}{\partial x_i}(\mathbf{x}=\mathbf{p}) = \lim_{h\to 0} \frac{f(x_1,x_2,...,x_i+h,x_{i+1},\dots,x_n)-f(x_1,x_2,\dots,x_n)}{h}(\mathbf{x}=\mathbf{p})\ , \tag{8}$$

Using the language introduced above, the partial derivatives of a function *span* the **tangential space** at that point, or, in other words, they are the **basis vectors** of that tangential space. Any derivative is a *linear combination* of these basis vectors. In 3d, this could look like the following graph (The surface is plotted with transparency, because matplotlib's 3d capabilities are quite restricted):

[2]:
```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')

# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")

# create all function points at once
z = func(x_m, y_m)

# this will plot the surface of the action on the vectors
ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)

# plot partial derivative in x-direction
ax.quiver(
        0, 0, 0, \
        float(f_x.subs(x, 1).evalf()), 0, 0, \
        color = 'red', alpha = .8, lw = 3, zorder=1000
    )

# plot partial derivative in y-direction
ax.quiver(
        0, 0, 0, \
        0, float(f_y.subs(y, 1).evalf()), 0, \
        color = 'red', alpha = .8, lw = 3, zorder=1000
    )

# you might need to adjust the axes limits
#ax.set_xlim(-2, 2)
#ax.set_ylim(-2, 2)
#ax.set_zlim( 0, 8)
```

```
[2]: <mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x7fd00e537940>
```



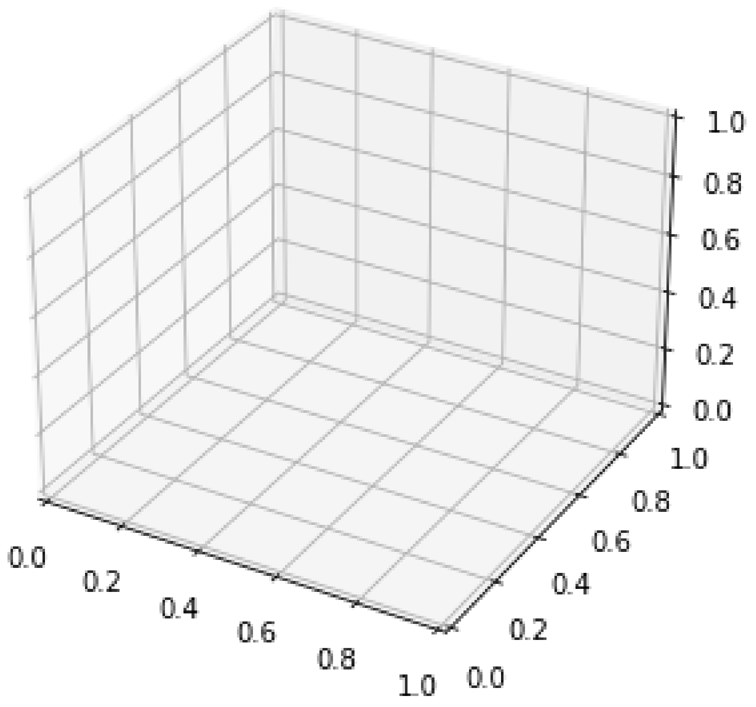
2.13 Gradient

The gradient of a function is the multivariate generalization of the partial derivatives.

$$\operatorname{grad}_{\mathbf{x}} f(\mathbf{x}) = \nabla_{\mathbf{x}} f(\mathbf{x}) = \frac{\partial f}{\partial x_1}\hat{e}_1 + \frac{\partial f}{\partial x_2}\hat{e}_2 + \cdots = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \end{bmatrix} \tag{9}$$

containing the partial derivatives in the rows. Often it's abbreviated to simply ∇f, where $\nabla = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots\right]^T$ is called “nabla”.

[1]:
```
%matplotlib notebook
import sympy as sy
import numpy as np
from sympy.functions import sin,cos
from math import factorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2  # try different factors, especially a + instead of -
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')


# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)
n = np.zeros(x_m.shape)
o = np.ones(x_m.shape)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")
func_x = sy.lambdify([x, y], f_x, "numpy")
func_y = sy.lambdify([x, y], f_y, "numpy")

z = func(x_m, y_m)

def plot_surface(surface, contours, cgrad, gradient):
    ax.cla()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_zlim(np.amin(z), np.amax(z))
    if surface:
        ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)
    if contours:
        ax.contour(x_m, y_m, z, zdir='z', offset=0, cmap='viridis')
    if cgrad:
        ax.quiver(x_m, y_m, n, func_x(x_m, y_m), func_y(x_m, y_m), n, \
                  length=0.05, normalize=False, alpha=0.7)
    if gradient:
        ax.quiver(x_m, y_m, func(x_m, y_m),
                  func_x(x_m, y_m), func_y(x_m, y_m), n, \
                  length=0.05, normalize=False, alpha=0.7)


interact(plot_surface, surface=True, contours=False, cgrad=False, gradient=True)
```



```
interactive(children=(Checkbox(value=True, description='surface'), Checkbox(value=False, descr
```

```
[1]: <function __main__.plot_surface(surface, contours, cgrad, gradient)>
```

It's often useful to plot **contour lines** of some function, which are curves of constant value, projected to a plane. The gradient is always perpendicular to the contour lines. This makes intuitive sense, after accepting the fact that the gradient always points in the direction of *steepest change*. We can easily see this after introducing the **directional derivative**

$$\mathrm{grad}_{\mathbf{v}} f(\mathbf{x}) = \nabla_{\mathbf{v}} f(\mathbf{x}) = \mathbf{v} \cdot \nabla f(\mathbf{x}) \tag{10}$$

Since this is a scalar product, this can be written as

$$\nabla_{\mathbf{v}} f = |\mathbf{v}|\,|\nabla f|\cos(\theta) \tag{11}$$

This product becomes maximal, when $\mathbf{v}$ and ∇f point in the same direction (are *colinear*), such that $\theta = 0$. Hence, the gradient always shows in the direction of *steepest ascent*. For a minimum, the vectors would have to point in opposite directions, such that $\theta = \pi$.

Let's look at an example:

[3]:
```
# quadratic int grid, to make things easier
grid_size = 10
X = np.arange(0, grid_size, 1)
Y = np.arange(0, grid_size, 1)
X, Y = np.meshgrid(X, Y)

# list of x-coordinates and y-coordinates in separate lists
carriers = np.array([[2, 7, 4], [2, 7, 6]])
charges  = np.array([ -1, 1, -1])

plt.figure(figsize=(8, 8))

def plot_elec_field(n_carriers):
    plt.cla()
    Ex = np.zeros((grid_size, grid_size))
    Ey = np.zeros((grid_size, grid_size))

    for carrier in range(n_carriers):
        for i in range(grid_size):
            for j in range(grid_size):
                Ex[i, j] += charges[carrier] * (j - carriers[1][carrier]) /\
                            ((i - carriers[0][carrier])**2 + (j -␣
 ↪carriers[1][carrier])**2)**1.5
                Ey[i, j] += charges[carrier] * (i - carriers[0][carrier]) /\
                            ((i - carriers[0][carrier])**2 + (j -␣
 ↪carriers[1][carrier])**2)**1.5

    E = np.hypot(Ex, Ey)
    Ex /= E
    Ey /= E

    # plot charges
    plt.plot(*carriers[:,:n_carriers], 'bo', markersize=10)

    # plot field
    plt.quiver(X, Y, Ex, Ey, E, pivot='mid')

    plt.axis('equal')
    plt.axis('off')
```

```
interact(plot_elec_field, n_carriers=(1, 3))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(IntSlider(value=2, description='n_carriers', max=3, min=1), Output()), _
```

```
[3]: <function __main__.plot_elec_field(n_carriers)>
```

For vector functions $\mathbf{f}(\mathbf{x})$, the gradient $\nabla\mathbf{f}(\mathbf{x})$ is often called the **Jacobian matrix**:

$$J_{\mathbf{f}}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \tag{12}$$

which is an $n \times m$ matrix. The determinant of this matrix is called the **Jacobian determinant** and both are used in transforming between coordinate system, e.g. from euclidean coordinate systems to polar coordinate system.

2.14 Hessian

The Hessian of a function f if the gradient of the gradient, measuring how much the gradient vector changes with changing spacial position. The Hessian is symmetric (see Schwarz's theorem, compare to **stiffness matrix**), so it has real *eigenvalues* and *eigenvectors*. For scalar multivariate $f(\mathbf{x})$, the Hessian is defined as

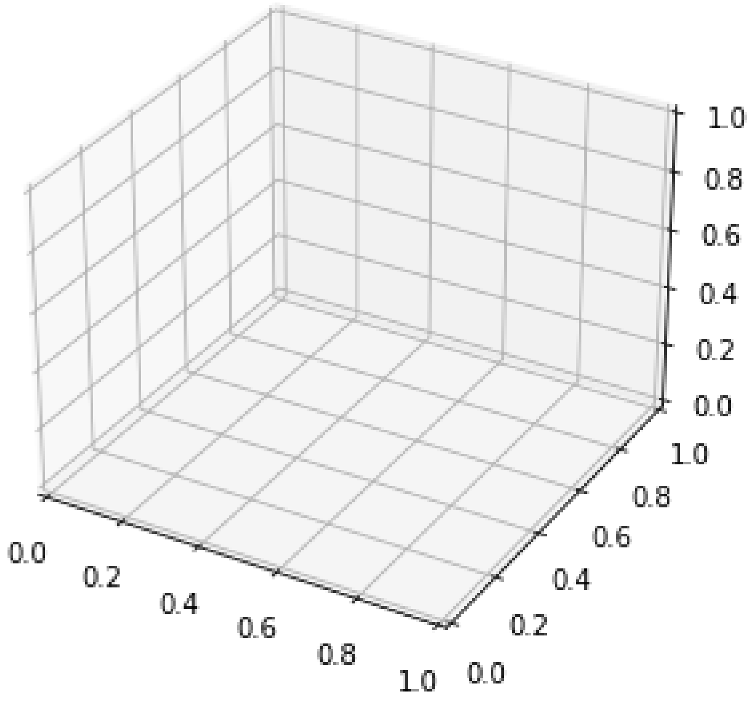
$$H_f(\mathbf{x}) = \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix} \tag{13}$$

which is an nxn-matrix. Each *column* contains the *gradient* of one component of the *gradient* of the original function, so each column tells us by how much that component of the gradient varies.

As in the 1d case, the Hessian as a second order derivative quantifies **curvature** of $f(\mathbf{x})$, with the only intricacy that curvature now depends on direction. A deep study of curvature is done in *differential geometry* and leads to several insights, that we cannot discuss in this short course.

Consider a function of the form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{x}^T\mathbf{b} + c$. The gradient of this function is $\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$. Its Hessian then is $H_f(\mathbf{x}) = \mathbf{A}$. Recall what we've learned about *quadratic forms*. In the following code, *eigenvalues* and *eigenvectors* are set and a corresponding quadratic form built up using a corollary of the **spectral theorem**: $\mathbf{A} = \sum_{i=1}^{2} \lambda_i \mathbf{v}_i \otimes \mathbf{v}_i$, where λ_i are the eigenvalues, $\mathbf{v}_i$ are the eigenvectors of $\mathbf{A}$ and $\otimes$ denotes the dyadic product.

[4]:
```
fig = plt.figure(figsize=(8,5))

ax = fig.add_subplot(111, projection='3d')

mesh_points = np.linspace(-2,2,20)

x, y = np.meshgrid(mesh_points, mesh_points)

vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T

def plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2):
    ax.cla()
      = np.array([eigval1, eigval2])
    v = np.array([[1, 0], [0, 1]])

    A = np.array(sum([ [i] * np.outer(v[i], v[i]) for i in range( .shape[0])]))

    q = np.array([vec.T @ A @ vec for vec in vecs])

    ax.set_xlim(np.amin(x), np.amax(x)+2)
    ax.set_ylim(np.amin(y), np.amax(y)+2)
    ax.set_zlim(np.amin(q)-2, np.amax(q))

    q = q.reshape(20,20)

    if plot_surface:
        ax.plot_surface(x, y, q, cmap='viridis', alpha=0.8)

    if plot_projs:
        ax.contourf(x, y, q, zdir='x', offset=np.amax(x)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='y', offset=np.amax(y)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='z', offset=np.amin(q)-2, cmap='viridis')

    if plot_eigv:
        for vi in v:
            ax.quiver(0, 0, 0, *vi, 0, color = 'red', lw = 3, zorder=1000)



interact(plot_hessian, \
         plot_surface=True, \
         plot_projs=True, \
         plot_eigv=True, \
         eigval1=widgets.IntSlider(min=-2, max=2, value=1,␣
 ↪continuous_update=False), \
        eigval2=widgets.IntSlider(min=-2, max=2, value=-1,␣
↪continuous_update=False))
```



```
interactive(children=(Checkbox(value=True, description='plot_surface'), Checkbox(value=True, de
```

```
[4]: <function __main__.plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1,
     eigval2)>
```

2.15 Taylor Series

Recall that derivatives approximate functions at a specific point. The first derivative gives a linear approximation, the second derivative a quadratic approximation and equivalently for higher orders. Summing up these derivatives of a function $f(\mathbf{x})$ in a specific way gives the **Taylor series** of that function at a specific point $\mathbf{a}$:

$$T_f^n(\mathbf{a}) = \sum_{i=1}^{n} \frac{f^{(n)}(\mathbf{a})}{n!}(\mathbf{x}-\mathbf{a})^n = f(\mathbf{a}) + (\mathbf{x}-\mathbf{a})^T \nabla f + \frac{1}{2}(\mathbf{x}-\mathbf{a})^T H_f(\mathbf{a})(\mathbf{x}-\mathbf{a}) + \dots \tag{14}$$

where $f^{(n)}(\mathbf{a})$ ist the n-th derivative of f, evaluated at $\mathbf{x} = \mathbf{a}$. For “sufficiently nice” functions, this series converges to the function f itself in the limit $n \to \infty$.

The Taylor series approximates a function with **polynomials**. See a few examples below:

[7]:
```
from sympy.functions import sin,cos
from math import factorial

plt.figure()

x = sy.Symbol('x')

f = x*sin(x)
func = sy.lambdify(x, f, "numpy")


# Taylor expansion at x0
def taylor(function,x0,n):
    p = 0
    for i in range(n):
        p += function.diff(x, i).subs(x, x0) / factorial(i) * (x - x0)**(i)

    return p


def plot_taylor(x0, order):
    plt.cla()

    x_lims = [x0 - 5, x0 + 5]

    x1 = np.linspace(x_lims[0], x_lims[1], 100)
    y1 = []

    plt.plot(x1,func(x1),label='sin(x)', lw=3)
    plt.scatter(x0,float(f.subs(x,x0).evalf()), lw=8, color="darkorange",␣
 ↪marker="+")

    for i in range(1,order+2):
        f_curr = taylor(f,x0,i)
        print('Taylor expansion at x0, order ' + str(i-1) + ": ", f_curr)
        for k in x1:
            y1.append(f_curr.subs(x,k))
        plt.plot(x1,y1,label='order '+str(i-1), linestyle="--")
        y1 = []

    plt.xlim(x_lims)
    plt.ylim([-5,8])
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.title('Taylor series approximation')
```

```
interact(plot_taylor, \
         x0=widgets.FloatSlider(min=2, max=7, value=3.1415927,␣
 ↪continuous_update=False), \
         order=widgets.IntSlider(min=0, max=8, value=3,␣
 ↪continuous_update=False));
```

```
<Figure size 432x288 with 0 Axes>
```

```
interactive(children=(FloatSlider(value=3.1415927, continuous_update=False, description='x0', 
```

Using polynomials is obviously easier to handle than more complicated function types. Very often, a **linearization** is enough. Linearization is a Taylor series up to order 1, so a linear approximation to some function. Still, the second derivative can give us critical information about the problem. We'll see this in a later lesson today.

The Taylor series for multivariate vector functions $\mathbf{f}(\mathbf{x})$ is "simply" the Taylor series of each component.

2.16 Pandas

We've seen how NumPy can handle data and manipulate it. There are some detriments though. E.g., NumPy only stores arrays of numbers with a fixed data type (unless using structured arrays, but they're not nice to handle). If you have labeled data, say, sequential sensor data and timestamps (incuding dates), you'd have to either create two arrays or find a format that can turn floats into timestamps again. You also have to keep track of what the columns represent. This is tedious and error-prone. The resulting data won't be easy to read by humans or scattered among various arrays.

`pandas` solves this problem and tries to enforce a better data handling strategy. It works similarly to relational databases and allows similar operations on data. The basic data structure used by `pandas` is a *data frame*. It makes use of quite a few NumPy features, so in many cases we need to import both:

[2]:
```
import numpy as np
import pandas as pd
```

Pandas supports different kinds of data structures, all managed as data frames. A simple examle is a `series`, where Pandas automatically creates running integer indices:

[5]:
```
ser = pd.Series([1, 3, 5, np.nan, 6, 8])

print(ser)
```

```
0    1.0
1    3.0
2    5.0
3    NaN
```

```
4    6.0
5    8.0
dtype: float64
```

The `np.nan` is pandas standard way of representing *missing data*, so it's even possible to keep track of where, e.g., sensor data couldn't be taken or some data was lost. We also see that printing a dataframe outputs a table. This is great for quickly understanding the type of data. What helps even more is a description of data columns:

[108]:
```
# pandas can automatically create a range of dates
dates = pd.date_range('20130101', periods=6)

# create a data frame with random number data, date indices and column labels
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=["Value 1",␣
↪"Value 2", "Value 3", "Value 4"])

print(df)
df
```

```
             Value 1   Value 2   Value 3   Value 4
2013-01-01 -0.643929 -0.941704  1.814025  0.863829
2013-01-02 -1.572498 -0.422332 -1.180399  0.381553
2013-01-03  2.024738  0.514611 -0.330865  0.123765
2013-01-04  0.927695  0.740022 -1.002011  0.095836
2013-01-05 -0.106089 -1.574821 -0.866819  0.418917
2013-01-06  1.095029  1.025670 -0.267690  0.129252
```

[108]:
```
             Value 1   Value 2   Value 3   Value 4
2013-01-01 -0.643929 -0.941704  1.814025  0.863829
2013-01-02 -1.572498 -0.422332 -1.180399  0.381553
2013-01-03  2.024738  0.514611 -0.330865  0.123765
2013-01-04  0.927695  0.740022 -1.002011  0.095836
2013-01-05 -0.106089 -1.574821 -0.866819  0.418917
2013-01-06  1.095029  1.025670 -0.267690  0.129252
```

So far, the data always had the same data type. As mentioned, this is not necessary. The following code converts a Python `dict` to a dataframe:

[110]:
```
df2 = pd.DataFrame({'val 1': 1.,
                    'val 2': pd.Timestamp('20130102'),
                    'val 3': pd.Series(1, index=list(range(4)),␣
↪dtype='float32'),
                    'val 4': np.array([3] * 4, dtype='int32'),
                    'val 5': pd.Categorical(["test", "train", "test", "train"]),
                    'val 6': 'foo'})



print(df2.dtypes)
```

```
df2
```

```
val 1           float64
val 2    datetime64[ns]
val 3           float32
val 4             int32
val 5          category
val 6            object
dtype: object
```

```
[110]:    val 1      val 2  val 3  val 4  val 5 val 6
       0    1.0 2013-01-02    1.0      3   test   foo
       1    1.0 2013-01-02    1.0      3  train   foo
       2    1.0 2013-01-02    1.0      3   test   foo
       3    1.0 2013-01-02    1.0      3  train   foo
```

There are many options to address elements in a data frame:

```
[43]: print(df2["val 5"])
      print()
      print(df2["val 5"][2])
      print()

      print(df2.iloc[:,4])
      print()
      print(df2.iloc[2,4])
```

```
0     test
1    train
2     test
3    train
Name: val 5, dtype: category
Categories (2, object): ['test', 'train']

test

0     test
1    train
2     test
3    train
Name: val 5, dtype: category
Categories (2, object): ['test', 'train']

test
```

Data can be converted to numpy arrays:

```
[114]: df.to_numpy()
```

```
[114]: array([[-0.64392853, -0.94170415,  1.81402525,  0.86382945],
              [-1.57249811, -0.42233239, -1.18039864,  0.38155329],
              [ 2.02473785,  0.51461143, -0.33086461,  0.12376472],
              [ 0.9276945 ,  0.74002206, -1.00201072,  0.09583597],
              [-0.10608895, -1.57482072, -0.86681926,  0.41891713],
              [ 1.09502893,  1.02567042, -0.26768965,  0.12925243]])
```

To get a quick statistical summary of the data, `describe()` can be called:

```
[115]: df.describe()
```

```
[115]:         Value 1   Value 2   Value 3   Value 4
       count  6.000000  6.000000  6.000000  6.000000
       mean   0.287491 -0.109759 -0.305626  0.335525
       std    1.308588  1.033190  1.100919  0.294286
       min   -1.572498 -1.574821 -1.180399  0.095836
       25%   -0.509469 -0.811861 -0.968213  0.125137
       50%    0.410803  0.046140 -0.598842  0.255403
       75%    1.053195  0.683669 -0.283483  0.409576
       max    2.024738  1.025670  1.814025  0.863829
```

The transpose works just like in NumPy:

```
[48]: print(df2.T)
```

```
                         0                    1                    2  \
val 1                    1                    1                    1
val 2  2013-01-02 00:00:00  2013-01-02 00:00:00  2013-01-02 00:00:00
val 3                    1                    1                    1
val 4                    3                    3                    3
val 5                 test                train                 test
val 6                  foo                  foo                  foo

                         3
val 1                    1
val 2  2013-01-02 00:00:00
val 3                    1
val 4                    3
val 5                train
val 6                  foo
```

There are **head** and **tail** commands like on UNIXoid OSs, to get a quick glimpse of what data is saved. The former prints a few of the beginning rows, the latter a few of the last lines:

```
[52]: df2.head()
      #df2.tail()
```

```
[52]:    val 1      val 2  val 3  val 4  val 5 val 6
      0    1.0 2013-01-02    1.0      3   test   foo
      1    1.0 2013-01-02    1.0      3  train   foo
      2    1.0 2013-01-02    1.0      3   test   foo
      3    1.0 2013-01-02    1.0      3  train   foo
```

A pretty usefule feature that's also implemented in NumPy is *boolean indexing*. Comparative statements like the following create *index masks*, which can be used to address non-continuous elements of a data frame:

```
[57]: df < 1
```

```
[57]:             Value 1  Value 2  Value 3  Value 4
      2013-01-01     True     True     True     True
      2013-01-02     True     True    False     True
      2013-01-03     True    False    False     True
      2013-01-04    False     True     True     True
      2013-01-05     True     True     True    False
      2013-01-06     True     True     True     True
```

```
[61]: new_df = df[df < 1]
      new_df
```

```
[61]:              Value 1   Value 2   Value 3   Value 4
      2013-01-01 -0.245117 -0.393222  0.624931  0.391709
      2013-01-02 -0.281136 -0.776862       NaN -1.910095
      2013-01-03 -1.553736       NaN       NaN -0.561621
      2013-01-04       NaN -1.244856  0.034360  0.248727
      2013-01-05 -0.164869  0.672306 -0.070947       NaN
      2013-01-06 -1.097641 -0.892463 -0.877066 -1.536823
```

When a row contains missing data, it might not be useful for further processing at all. Pandas makes it easy to quickly get rid of all rows with missing data, or to change the missing data to some other value. This is part of a process called *data cleaning*. PyJanitor is a part of the Pandas ecosystem and automates a few of the common data cleaning tasks. We don't have time to go into details here unfortunately, but it's worth checking out all the capabilities offered here for larger projects.

```
[62]: new_df.dropna(how='any')
```

```
[62]:              Value 1   Value 2   Value 3   Value 4
      2013-01-01 -0.245117 -0.393222  0.624931  0.391709
      2013-01-06 -1.097641 -0.892463 -0.877066 -1.536823
```

```
[64]: new_df.fillna(value=0)
```

```
[64]:              Value 1   Value 2   Value 3   Value 4
      2013-01-01 -0.245117 -0.393222  0.624931  0.391709
```

```
2013-01-02 -0.281136 -0.776862  0.000000 -1.910095
2013-01-03 -1.553736  0.000000  0.000000 -0.561621
2013-01-04  0.000000 -1.244856  0.034360  0.248727
2013-01-05 -0.164869  0.672306 -0.070947  0.000000
2013-01-06 -1.097641 -0.892463 -0.877066 -1.536823
```

You can get the the index mask of all missing values:

[66]:
```
pd.isna(new_df)
```

[66]:
```
            Value 1  Value 2  Value 3  Value 4
2013-01-01    False    False    False    False
2013-01-02    False    False     True    False
2013-01-03    False     True     True    False
2013-01-04     True    False    False    False
2013-01-05    False    False    False     True
2013-01-06    False    False    False    False
```

It's possible and sometimes a great help to apply functions to the data.

[117]:
```
df.apply(np.sin)
```

[117]:
```
             Value 1   Value 2   Value 3   Value 4
2013-01-01 -0.600342 -0.808562  0.970565  0.760335
2013-01-02 -0.999999 -0.409889 -0.924758  0.372363
2013-01-03  0.898726  0.492197 -0.324861  0.123449
2013-01-04  0.800240  0.674304 -0.842556  0.095689
2013-01-05 -0.105890 -0.999992 -0.762274  0.406771
2013-01-06  0.888941  0.855062 -0.264504  0.128893
```

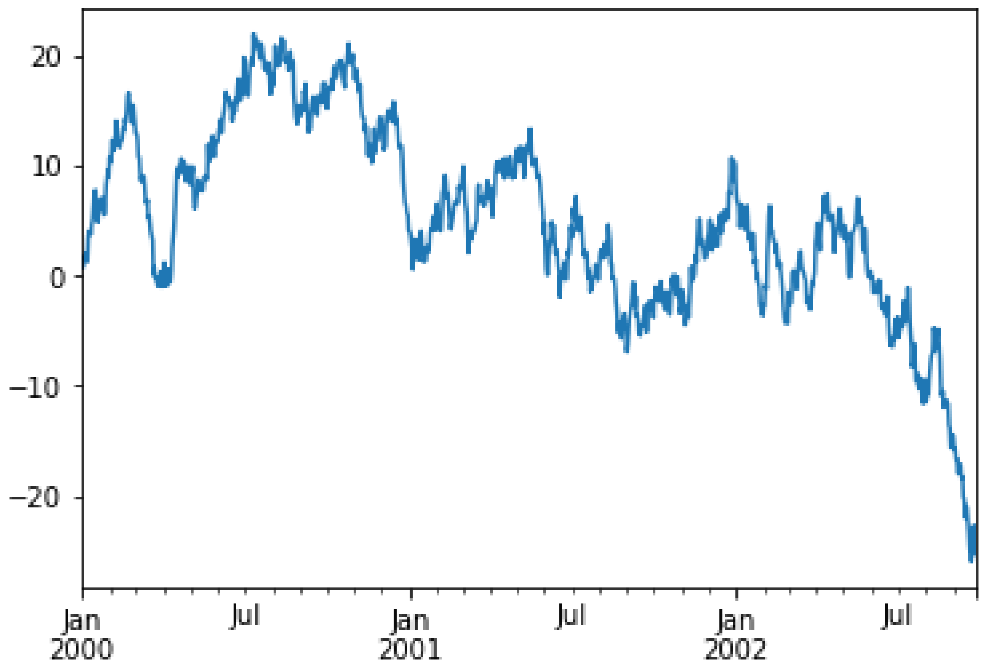
2.16.1 Plotting

Pandas can use different rendering engines as backends for plotting. The standard backend is matplotlib, which we already saw earlier today. Plotting is greatly simplified with pandas and many useful visualizations are already implemented and ready to use. If some things are missing, Pandas can be enhanced by a number of libraries.

Plotting something is as simple as calling the `plot()` method of a data frame:

[82]:
```
# create a random dataset with dates as indices
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',␣
↪periods=1000))

# get the cumulative sum along the columns to get a nicer plot
ts = ts.cumsum()

ts.plot()
```

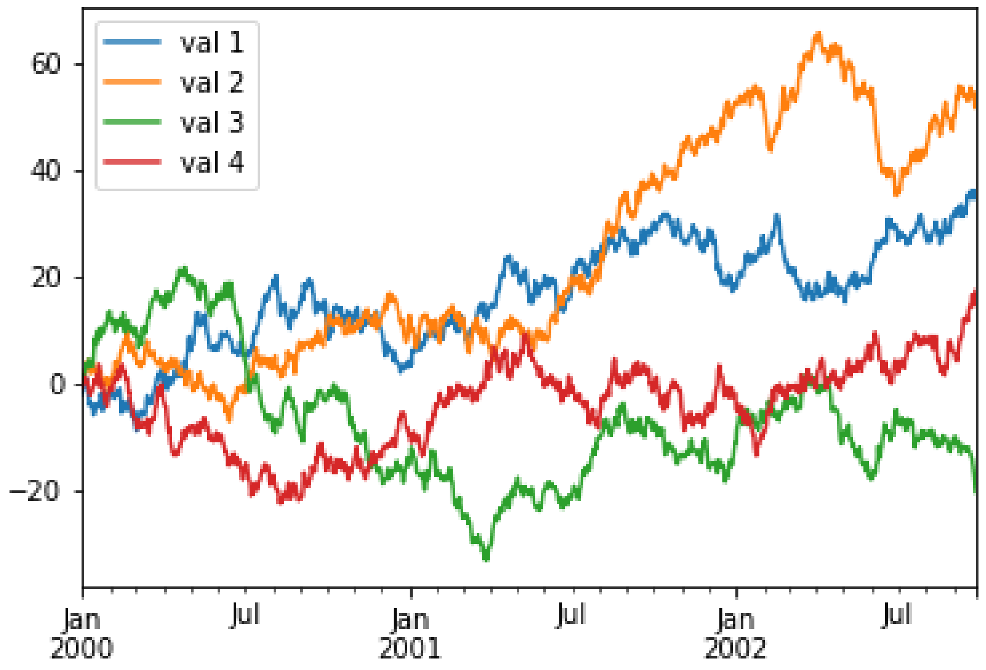
```
[82]: <AxesSubplot:>
```



Pandas can directly plot different curves for all columns with labels, but to handle plot details, pyplot has to be imported first:

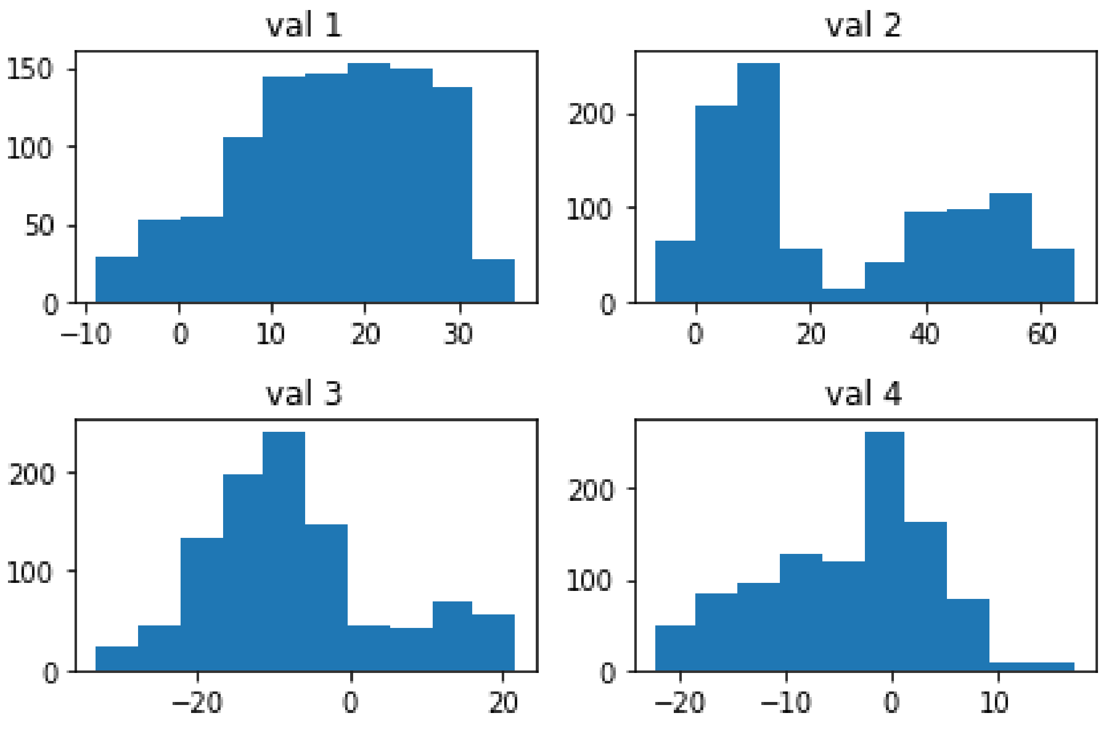
```
[97]: # create a random dataset with 4 dimensions and dates as indices
ts = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=["val 1",␣
 ↪"val 2", "val 3", "val 4"])

# get the cumulative sum along the columns to get a nicer plot
ts = ts.cumsum()

ts.plot()
```

```
[97]: <AxesSubplot:>
```



Histograms are also easy to draw. Here, we need to call the `tight_layout()` function from pyplot. To do so, it must be imported directly:

[118]:
```
import matplotlib.pyplot as plt

ts.hist(grid=False)

plt.tight_layout()
```



2.16.2 Saving and Loading Data

Tabular-structured data can be exported as, e.g., a **c**omma-**s**eparated **v**alues file:

```
[119]: ts.to_csv('ts.csv')
```

To read a csv file from disk:

```
[107]: ts_file = pd.read_csv('ts.csv')

       ts_file
```

```
[107]:     Unnamed: 0      val 1      val 2      val 3      val 4
       0    2000-01-01  -1.969445  -0.356405   1.026101  -0.762619
       1    2000-01-02  -0.606335   1.367543   1.555996  -0.232897
       2    2000-01-03   0.777804   1.326308   1.024626   0.017047
       3    2000-01-04   0.655275   2.141760   1.549831   0.905668
       4    2000-01-05   0.544901   2.625159   2.487415   1.339163
       ..          …          …          …          …          …
       995  2002-09-22  36.063779  54.502244 -14.120248  16.600449
       996  2002-09-23  34.513519  53.477599 -15.755223  14.546741
       997  2002-09-24  35.042919  54.429797 -16.938643  14.938983
       998  2002-09-25  36.122264  51.712579 -18.280598  16.089976
       999  2002-09-26  35.800122  52.101227 -20.261030  17.299086
```

```
[1000 rows x 5 columns]
```

This was only a very quick overview of the basic capabilities of Pandas. There's much more than a part of a lecture can cover. The documentation is easy to digest and exhaustive. There's barely anything we'll need in this course that Pandas can't do.

3 Dimensionality Reduction

3.1 The Curse of Dimensionality

We talked about how to identify and generate *features* for machine learning algorithms, which are the inputs for the methods and are an extremely important part of the complete machine learning pipeline. Apart from simply using available data $x_1, x_2, \ldots, x_N$ from, say, measurements in an experiment, we can also construct new features as combinations like $x_1 \cdot x_2$ or functions of the original features like $\sin(x_1)$. Including these in the analysis can be beneficial for reducing convergence time or even finding a solution at all. A simple example can be tested on the TensorFlow Playground. When changing the activation to linear (otherwise it will almost always find a suitable mapping) for the XOR problem, only using x_1 and x_2 as the features will not converge at all. Additionally using $\sin(x_1)$ and $\sin(x_2)$ creates at least a partial classification of the whole dataset (do you know why this classification does not work? Recall the arguments from last lecture). The number of features is generally referred to as the *dimension* of a problem.

A problem that will keep coming up in machine learning settings is how many data points to generate (or measure) for training, or, making the most out of the available data points. This is related to how many features a dataset is comprised of. The more features (dimensions) are in a dataset, the "exponentially more" samples are needed to accurately probe or *sample* that space. This is an instance of **the curse of dimensionality**, where adding a linear number of dimensions creates demand for an exponential increase in some resource(s), in this case the number of samples. You can play around with the number of samples used for probing 1D, 2D and 3D space below:

```
[4]: %matplotlib notebook
     #from __future__ import print_function
     import numpy as np
     import matplotlib.pyplot as plt
     from mpl_toolkits.mplot3d import Axes3D
     from ipywidgets import interact, interactive, fixed, interact_manual
     import ipywidgets as widgets


     samples = np.random.rand(501)

     def plot_samples(data_size=2):
         fig = plt.figure(figsize=(12,6))
         ax0 = fig.add_subplot(131)
         ax1 = fig.add_subplot(132)
         ax2 = fig.add_subplot(133, projection='3d')
```

```
    ax0.cla()
    ax1.cla()
    ax2.cla()

    ax0.set_title("1D")
    ax1.set_title("2D")
    ax2.set_title("3D")

    ax0.set_xlim([0,1])
    ax0.set_ylim([0,1])
    ax1.set_xlim([0,1])
    ax1.set_ylim([0,1])
    ax2.set_xlim([0,1])
    ax2.set_ylim([0,1])
    ax2.set_zlim([0,1])

    ax0.hlines(0.5, 0, 1, color='gray', alpha=0.5, lw=4)
    ax0.scatter(samples[:data_size], data_size*[0.5], lw=0.1)

    ax1.scatter(samples[:data_size], samples[-data_size:])

    ax2.scatter(samples[:data_size], samples[-data_size:],␣
 ↪samples[-10-data_size:-10])


    plt.tight_layout()

interact(plot_samples, data_size=(2, 500))
```

```
interactive(children=(IntSlider(value=2, description='data_size', max=500, min=2), Output()),
```

```
[4]: <function __main__.plot_samples(data_size=2)>
```

Roughly speaking, when we decide that 10 samples are sufficient for describing the 1D space above, the 2D space would need around 100 samples. For the 3D space, already around 1000 datapoints are needed. This is just a coarse estimate, but here, we'd need 10^d samples for d dimensions ($= d$ features) in the dataset. Usually, we're not in the position to get arbitrary amounts of data to probe these spaces, so including more features must be carefully planned with respect to the available data. The situation in most cases is that we're given a fixed number of data points, so adding more dimensions to the problem will make the dataset more *sparse* regarding the problem.

A solution, or at least a way to ease this problem, is **dimensionality reduction** or **model order reduction**. Theoretically, adding more features for an analysis should increase the predictive capabilities of machine learning models, but adding features also requires taking a lot more data into the training phase. The idea of dimensionality reduction is to represent the available data by fewer features than originally present.

The naive approach would be to simply discard some features and forget about them. This approach is called *feature elimination* and makes sense when it's very obvious that certain features do not play any role in the problem at hand, e.g., temperature for temperature-independet phenomena, or when it doesn't influence the dataset at all. An analytical way to do so is to perform a **principal component analysis**, which we'll come to later today.

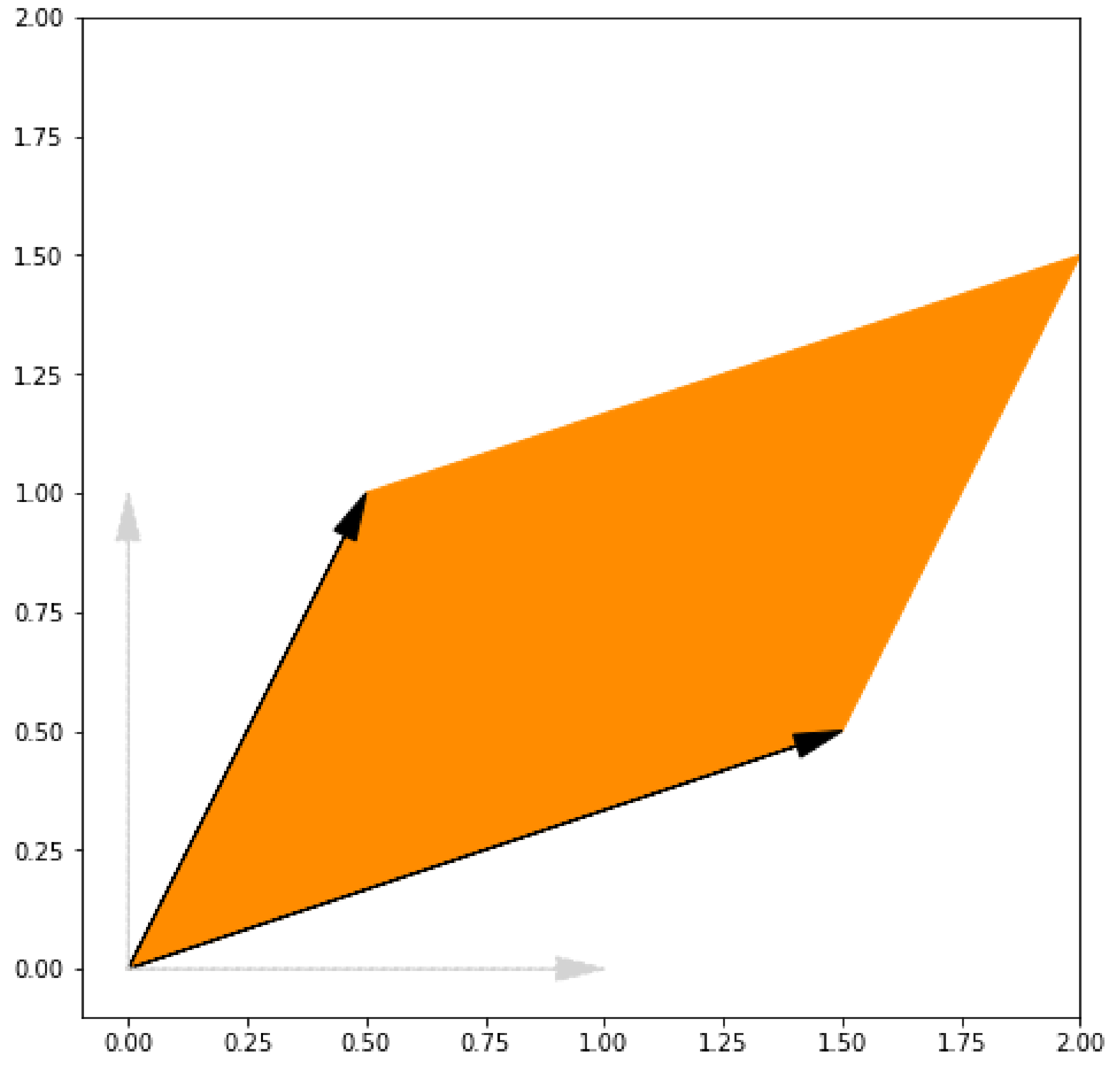
3.2 Eigendecomposition

Before moving on to principal component analysis, we should take a quick gander at eigendecomposition and its generalization to non-square matrices, as it will make things much easier to understand from a linear algebra perspective.

All *normal* matrices $\mathbf{A}$, which means they satisfy the identity $\mathbf{A}\mathbf{A}^T = \mathbf{A}^T\mathbf{A}$, can be eigendecomposed. This includes orthogonal, symmetric, and skew-symmetric matrices. The physical interpretation is that every such matrix can be thought of as a combination of a rotation and stretching, sometimes also a reflection (when the determinant is negative). See for example what the matrix $\mathbf{A} = \begin{bmatrix} 1.5 & 0.5 \\ 0.5 & 1.0 \end{bmatrix}$ does to a square:

[1]:
```python
import numpy as np
import matplotlib.patches as patches
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-0.1, 2.0)

# vectors
e1 = np.array([0.0, 1.0])
e2 = np.array([1.0, 0.0])
origin = np.array([0, 0])

# matrix is defined here
A = np.array([[1.5, 0.5],
              [0.5, 1.0]])

print("Is A normal?", "Yes." if np.all(A.T @ A == A @ A.T) else "No")

# matrix applied to starting vectors
v1 = A @ e1
v2 = A @ e2

# we need the coordinates of the endpoints
ax.add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], color="darkorange"))

ax.arrow(*origin, *(v1), head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *(v2), head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)

ax.arrow(*origin, *e1, head_width=0.05, head_length=0.1, \
         fc='lightgray', ec='lightgray', ls='--', length_includes_head=True)
ax.arrow(*origin, *e2, head_width=0.05, head_length=0.1, \
         fc='lightgray', ec='lightgray', ls='--', length_includes_head=True)


plt.show()
```

```
Is A normal? Yes.
```



Feel free to try different matrices and see how changing components changes the resulting shape.

Eigenvectors are vectors that only get *stretched* under the action of a matrix. The factor λ by which that eigenvector is stretched is called its corresponding **eigenvalue**:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

So after applying the matrix, the resulting vector is *colinear* to the original vector.

The eigenvectors of a matrix span an orthogonal system, in which the matrix $\mathbf{A}$ is *diagonal* (often denoted as $\mathbf{D}$). The eigenvector matrix $\mathbf{V} = [\mathbf{v}_1\mathbf{v}_2 \dots \mathbf{v}_n]$, containing all eigenvectors $\mathbf{v}_i$ in its columns, can be used to express this mathematically:

$$\mathbf{D} = \mathbf{V}^{-1}\mathbf{A}\mathbf{V}$$

or, equivalently:

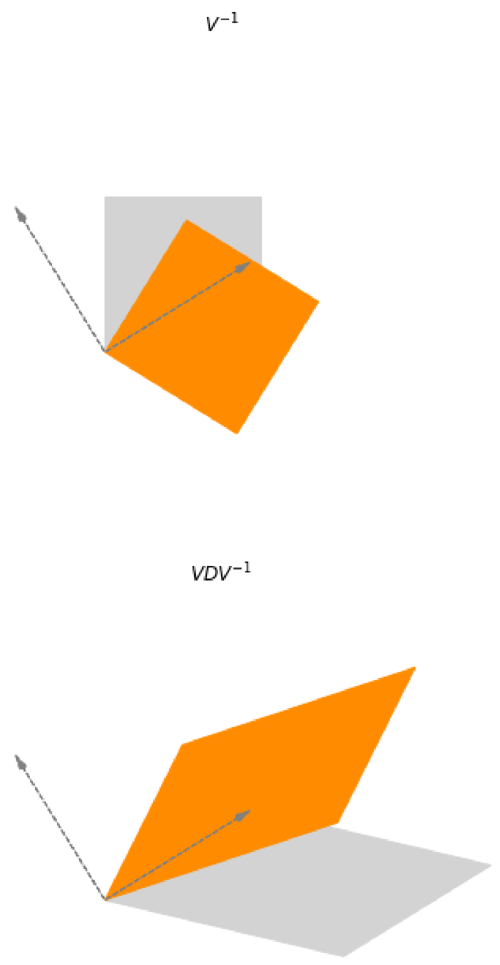
$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$$

The diagonal matrix $\mathbf{D}$ now contains all the eigenvalues of $\mathbf{A}$. What happens if we apply this decomposed form sequentially to a vector? Let's try this with a more complicated matrix:

[8]:
```
fig, axes = plt.subplots(2, 2, figsize=(10,10))
axes = axes.flatten()

titles = [r"$V^{-1}$",
          r"$ D V^{-1}$",
          r"$ V D V^{-1}$",
          r"$A$"]

for i,ax in enumerate(axes):
    ax.set_aspect('equal')
    ax.set_xlim(-1.0, 2.5)
    ax.set_ylim(-1.0, 2.0)
    ax.set_title(label=titles[i])


# vectors
e1 = np.array([0.0, 1.0])
e2 = np.array([1.0, 0.0])
origin = np.array([0.0, 0.0])

# try different matrices to see their effects
# diagonal matrices don't rotate, they only stretch, the stretch factors are␣
 ↪the eigenvalues
#A = np.array([[1.9, 0.0],
#               [0.0, 1.4]])

# rotation matrices do not stretch, they only rotate
# since the eigenvalues are imaginary here, the rotation
# happens in dimensions we do not plot here, hence we only see
# a line where perpendicular vectors should be
#  = np.pi / 2.0
#A = np.array([[np.cos( ), np.sin( )],
#               [-np.sin( ), np.cos( )]])

# combining both
A = np.array([[1.5, 0.5],
              [0.5, 1]])


# this saves eigenvalues into   and normalized eigenvectors as ROWS into V
 , V = np.linalg.eig(A)

# since we need the eigenvectors in the COLUMNS:
#V = V.T

V_inv = np.linalg.inv(V)
D = np.diag( )

print("Eigenvalues: \n",  )
print("Eigenvectors: \n", V)

# plot original square
axes[0].add_patch(patches.Polygon(xy=[origin, e1, e1+e2, e2],␣
 ↪color="lightgray"))

# apply first rotation
v1 = V_inv @ e1
v2 = V_inv @ e2
axes[0].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="darkorange"))

# apply stretch
axes[1].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="lightgray"))
v1 = D @ v1
v2 = D @ v2
axes[1].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="darkorange"))

# apply second rotation
axes[2].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="lightgray"))
v1 = V @ v1
v2 = V @ v2
```

```
axes[2].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="darkorange"))

# compare to action of original matrix
axes[3].add_patch(patches.Polygon(xy=[origin, e1, e1+e2, e2],␣
 ↪color="lightgray"))
v1 = A @ e1
v2 = A @ e2
axes[3].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2],␣
 ↪color="darkorange"))

# plot eigenvectors everywhere
for ax in axes:
    ax.arrow(*origin, *V[:,0], head_width=0.05, head_length=0.1, fc='gray',␣
 ↪ec='gray', ls='--')
    ax.arrow(*origin, *V[:,1], head_width=0.05, head_length=0.1, fc='gray',␣
 ↪ec='gray', ls='--')
    ax.axis("off")


plt.tight_layout()
plt.show()
```

```
Eigenvalues:
 [1.80901699 0.69098301]
Eigenvectors:
 [[ 0.85065081 -0.52573111]
 [ 0.52573111  0.85065081]]
```



So indeed, the vectors are first rotated by this decomposition, then stretched along the original coordinate axes, then rotated back.

Real symmetric matrices have real eigenvalues and eigenvectors. In case $\mathbf{A}$ is symmetric, the eigenvector matrix $\mathbf{V}$ satisfies $\mathbf{V}^T = \mathbf{V}^{-1}$, so $\mathbf{V}$ is orthogonal and hence, a **rotation matrix**. In this case, $\mathbf{V}$ is sometimes denoted $\mathbf{Q}$ or $\mathbf{R}$.

The eigendecomposition of a matrix might not be unique. An example of a matrix without a unique eigendecomposition is the identity matrix. The reason is that every vector is an eigenvector of the identity matrix, such that $\mathbf{V}$ is arbitrary.

If we take the eigendecomposition and explicitly write down each term vectorwise (recall that matrix multiplication multilies the rows of the first matrix with the columns of the second matrix) we get

$$\begin{aligned}
A &= \lambda_1 v_1 v_1^{\mathrm{T}} + \lambda_2 v_2 v_2^{\mathrm{T}} + \cdots \\
&= \lambda_1 \begin{bmatrix} | \\ v_1 \\ | \end{bmatrix} \begin{bmatrix} -- & v_1 & -- \end{bmatrix} + \lambda_2 \begin{bmatrix} | \\ v_2 \\ | \end{bmatrix} \begin{bmatrix} -- & v_2 & -- \end{bmatrix} + \cdots \\
&= \sum_i \lambda_i v_i v_i^{\mathrm{T}}
\end{aligned}$$

which is one formulation of the **spectral theorem**, found in virtually all areas of research, especially numerical analysis.

3.3 Singular Value Decomposition

Singular value decomposition is a generalization of the eigendecomposition of normal square matrices into rotational and stretching parts to *non-square* matrices. The matrices in the decomposition now also have different dimensions. Assuming $\mathbf{A} \in \mathbb{R}^{m \times n}$, we get

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}}$$

where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and orthogonal, $\mathbf{S} \in \mathbb{R}^{m \times n}$ and diagonal, and $\mathbf{V} \in \mathbb{R}^{n \times n}$ and orthogonal. So $\mathbf{U}$ and $\mathbf{V}$ are rotations, while $\mathbf{S}$ purely expresses stretching. All non-diagonal elements of $\mathbf{S}$ are zero.

The columns of $\mathbf{U}$ are eigenvectors of $\mathbf{A}\mathbf{A}^{\mathrm{T}}$, called *left-singular vectors*. They are real, since $\mathbf{U}$ is orthogonal. $\mathbf{V}$ contains the eigenvectors of $\mathbf{A}^T\mathbf{A}$, called the *right-singular eigenvectors*. The diagonal of $\mathbf{S}$ consists of the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$, called the *singular values*:

$$\operatorname{diag}(\mathbf{S}) = \sqrt{\lambda\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)}$$

The interpretation of what is happening with this decomposition is very similar to what's happening in an eigendecomposition. $\mathbf{V}^T$ rotates into the eigenspace of $\mathbf{A}$, $\mathbf{S}$ applies stretch factors, $\mathbf{U}$ rotates the system back. We'll visualize this behavior in the exercises today.

If we spell out the vectorwise multiplications like we did for the eigendecomposition, we get:

$$\begin{aligned}
A &= \sigma_1 u_1 v_1^{\mathrm{T}} + \lambda_2 u_2 v_2^{\mathrm{T}} + \cdots \\
&= \sigma_1 \begin{bmatrix} | \\ u_1 \\ | \end{bmatrix} \begin{bmatrix} -- & v_1 & -- \end{bmatrix} + \lambda_2 \begin{bmatrix} | \\ u_2 \\ | \end{bmatrix} \begin{bmatrix} -- & v_2 & -- \end{bmatrix} + \cdots \\
&= \sum_i^m \sigma_i u_i v_i^{\mathrm{T}}
\end{aligned}$$

where this time the sum only goes up to the first dimension of the original matrix $\mathbf{A}$ (can you see why?). This is especially interesting when $n \gg m$, which would be the case when samples are sparse and features are many in a data matrix.

There are several interesting applications. Very often, SVD is used for data compression, where dimensions of the system with small singular values are completely omitted, especially when the SVD of a *covariance matrix* is performed. This can reduce the order of a model significantly. In control theory, SVD is used for developing robust control. It can also be used as a tool to diagonalize matrices, or for *image/data compression*. The idea for all of these is to take the decomposition $\mathbf{U}, \mathbf{S}, \mathbf{V}$ and *truncate* their axis such that only a specific number of singular values is used. For $\mathbf{U}$, this would be the second axis, for $\mathbf{S}$ both axes, and for $\mathbf{V}$ the first axis. For the spectral decomposition above, this would be equivalent to cutting off all terms after some index.

Let's see what happens when we use SVD on an image. How many singular values do you need to be able to interpret the image? What's at the bottom right of the image?

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from skimage import data, img_as_float
from skimage.color import rgb2gray

grayscale = True

# taking an astronaut image from scikit-image as an example
image = img_as_float(data.astronaut())
if grayscale:
    image = rgb2gray(image)

# save this for later reshaping back
shape_original = image.shape

# get a 2d matrix of the image, so we can do SVD on it
# this is necessary because it contains 3 channels for all colors
if grayscale:
    image_2d = image
else:
    image_2d = image.reshape(shape_original[0], 3*shape_original[1])


# the SVD
U, S, V = np.linalg.svd(image_2d, full_matrices=False)

def compress_rgb_image(U,S,V, shape_original, cutoff=10):
    compressed_image = U[:,:cutoff] @ np.diag(S[:cutoff]) @ V[:cutoff,:]
    print(U[:,:cutoff].shape, np.diag(S[:cutoff]).shape, V[:cutoff,:].shape)

    # get back original shape
    image = compressed_image.reshape(shape_original)

    fig, axes = plt.subplots(1, 2, figsize=(12,7))
    axes[0].plot(S[:cutoff], lw=4)
    axes[0].set_title(label="all singular values used")
    axes[1].imshow((image * 255).astype(np.uint8), cmap='gray')

    plt.tight_layout()


interact(compress_rgb_image,
         U=fixed(U),
         S=fixed(S),
         V=fixed(V),
         shape_original=fixed(shape_original),
         cutoff=widgets.IntSlider(min=0, max=150, value=2,␣
 ↪continuous_update=False));
```

```
interactive(children=(IntSlider(value=2, continuous_update=False, description='cutoff', max=150
```

We see that not all singular values are necessary to convey meaning in the image. Depending on the level of detail desired, even a few suffice to recognize the astronaut, the US flag and the space ship. These kinds of data compression can be useful to reduce the order of a model in general, not only regarding ML techniques.

One area where this can be a great way to compress information is computational fluid dynamics. Consider a 2D flow field, the dynamics of which are calculated over some time range $[t_0, \ldots, t_m]$, discretized into m time slices, reminiscent of frames from a video. Each 2D slice can be unpacked into a vector, where each entry would be the value of some field quantity at a node (or a pixel for image data) in the system. These slices are the m samples in the data matrix, and the nodal values are the n features. An SVD performed on this data matrix will find m “eigenmodes” of the flow field and express the original flow fields as linear combinations of these eigenmodes. The eigenmodes are the row vectors of $\mathbf{U}$, the coefficients are the corresponding singular values and entries from $\mathbf{V}$. Truncating this description would yield flow fields with lower resolution as approximations to the real flow field.

3.4 Principal Component Analysis

We briefly mentioned a few naive approaches to manipulating the number of features used for describing a problem. Another approach would be to combine features by linear (or later even nonlinear transformations) in a more methodical way, that respects the *correlations* in a dataset. The idea is visualized in the graph below, where two features x_1 and x_2 are correlated:

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

x = np.random.rand(40)
y = -0.3 + 0.5*x
```

```
y_test = -0.3 + 0.5*x + 0.2*np.random.rand(40)-0.1
p1 = np.array([0.0, -0.3])
p2 = np.array([1.0,  0.2])


points = np.c_[x,y_test]

d=np.cross(p2-p1,points-p1)/np.linalg.norm(p2-p1)

dires = np.array([-d[i]*np.array([1/np.sqrt(2), -1/np.sqrt(2)]) for i in␣
 ↪range(len(d))])
pps = points-dires

def plot_corr(line, dists):
    fig = plt.figure(figsize=(12,12))
    ax = fig.add_subplot(111)

    #ax.set_xlim([])
    ax.set_ylim([-0.4, 0.4])

    ax.set_xlabel(r"$x_1$", fontsize=20)
    ax.set_ylabel(r"$x_2$", fontsize=20)

    ax.scatter(x, y_test)
    if line:
        ax.plot(x, y, lw=4, zorder=-20)
    if dists:
        for i in range(d.shape[0]):
            ax.plot([points[i,0], pps[i,0]], [points[i,1], pps[i,1]], \
                    color='darkorange', lw=3, alpha=0.7, zorder=-10)
        ax.scatter(pps[:,0], pps[:,1], marker='x')
    #plt.axes().set_aspect("auto")#1./plt.axes().get_data_ratio())
    ax.set_aspect("auto")

interact(plot_corr, line=False, dists=False)
```

```
interactive(children=(Checkbox(value=False, description='line'), Checkbox(value=False, descrip
```

```
[3]: <function __main__.plot_corr(line, dists)>
```

It is clear that there is a linear trend in the data (the features are correlated) and that we can find a new set of axes (activate line), where the distances of all the points to the new x'-axis are minimized, as in the image above when you activate dists.

These kinds of images are correlation plots. Sometimes manually deciding which features to eliminate or combine is possible by examining these correlation plots, but for hundreds of features this isn't really feasible. A technique that performs this combination of features in a way that reduces the overall number of features is principal component analysis (PCA). It tells us which features are the most important and provides the new axes like in the image above. PCA strives to maximize variance, while minimizing covariance of a dataset, so it finds rotated axes that best represent the dataset.

In the example above, the distances of the points to the new x'-axis are very small, so to reduce the dimension of the problem, instead of providing the coordinates of each point for both axes, we could provide the projection of the points onto it and ignore the distance in y'-direction, since it is very small.

Mathematically, PCA solves the following problem: For a given dataset X with N samples x_i comprised of F features ($X \in \mathbb{R}^{N\times F}$), find an $K \times F$-matrix $\mathbf{R}$ such that

$$\mathbf{x}'_i = \mathbf{R}^{\mathrm{T}}\mathbf{x}_i$$

where $\mathbf{x}'_i \in \mathbb{R}^K$ and $K \leq N$. Let's look at another example:

[2]:
```
x = np.random.rand(100)
y = -0.3 + 0.5*x
y_test = -0.3 + 0.5*x + 0.5*np.random.rand(100)-0.25
p1 = np.array([0.0, -0.3])
p2 = np.array([1.0,  0.2])

points = np.c_[x,y_test]

d=np.cross(p2-p1,points-p1)/np.linalg.norm(p2-p1)

dires = np.array([-d[i]*np.array([0.701, -0.701]) for i in range(len(d))])
pps = points-dires

def plot_corr(line, dists, new_axes):
    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111)
    #plt.cla()

    ax.set_xlabel(r"$x_1$", fontsize=16)
    ax.set_ylabel(r"$x_2$", fontsize=16)

    ax.set_xlim([-0.3, 1.3])
    ax.set_ylim([-0.5, 0.6])

    ax.scatter(x, y_test)

    if line:
        ax.plot(x, y, lw=4, zorder=-20)
    if dists:
        for i in range(d.shape[0]):
            ax.plot([points[i,0], pps[i,0]], [points[i,1], pps[i,1]], \
                        color='darkorange', lw=3, alpha=0.7, zorder=-10)
            ax.scatter(pps[:,0], pps[:,1], marker='x')
    if new_axes:
        ax.arrow(0, -0.3, 1, 0.5, head_width=0.05, head_length=0.1, \
         fc='red', ec='red', length_includes_head=True, zorder=1000)
        ax.arrow(0, -0.3, -0.701*0.1, 0.701*0.1, head_width=0.02, head_length=0.
↪05, \
         fc='red', ec='red', length_includes_head=True, zorder=1000)
        #plt.quiver(0, -0.3, [1, -0.701], [0.5, 0.701], color = 'red', lw = 3,␣
↪zorder=1000)


interact(plot_corr, line=False, dists=False, new_axes=False)
```

```
interactive(children=(Checkbox(value=False, description='line'), Checkbox(value=False, descrip
```

```
[2]: <function __main__.plot_corr(line, dists, new_axes)>
```

A lot of the information in the dataset is captured along the rotated x'-axis, which is the axis along which the variance is maximal. The y'-axis is orthogonal to this axis, maximizing the variance "perpendicular" to the first axis. If we project the points onto the x'-axis, we can retain the most information about the original dataset. To accurately do all of this, we need the directions of the axes x' and y', as well as their "lengths" λ_1 and λ_2. The length of an axis is the variance of the data along this direction. In this sense, PCA fits a hyperellipsoid to the data.

3.4.1 The Algorithm

To perform a PCA, the data has to be zero-centered first, which means that the mean of each feature over the whole dataset is calculated and subtracted from each respective column of every sample. The mean of feature f is calculated as

$$\mu_f = \frac{1}{N} \sum_i^N X_{if}$$

μ is a vector containing the feature means as columns. The zero-centered sample matrix is then

$$\bar{X} = X - U^{\mathrm{T}}$$

where $U = \begin{bmatrix} | & | & \cdots & | \\ \mu & \mu & \cdots & \mu \\ | & | & \cdots & | \end{bmatrix}$. The next step is to calculate the covariance matrix C of the data:

$$C = \frac{1}{N} \bar{X}^{\mathrm{T}} \bar{X}$$

Recall that the $F \times F$ covariance matrix contains the variance of a variable on the diagonal, and the covariances on the off-diagonal elements. When we're trying to maximize variance (or, equivalently, minimize covariance), we're looking for a diagonal matrix. This is the last step; diagonalize the covariance matrix (which is possible, since C is symmetric). This is done by finding the eigenvectors composing the matrix V (more specifically, the *right* eigenvectors), such that

$$C = VDV^{\mathrm{T}}$$

where D is a diagonal matrix. The eigenvectors v_i are the **principal components**. In this eigensystem, the entries λ_i of D are the **variance** of each data point in that system. In general, the eigenvalues and eigenvectors aren't sorted. After sorting them in decreasing order of magnitude of the eigenvalues, the reduction process is performed by truncating the last $F - K$ rows of V and D, such that only the entries with the lowest variances are discarded:

$$R = V[0 : K]$$

This is the reduction matrix R from the problem description. Multiplying the full dataset by this reduction matrix gives the new dataset

$$X' = \bar{X}R \in \mathbb{R}^{N \times K}$$

with a reduced number K of features. The first row of this dataset contains the projection of the first sample in the original dataset onto the first principal component, and so on. To get the embedding of the original data into this lower-dimensional subspace, the means U have to be added again to complete the transformation.

3.4.2 PCA by SVD

In practice, calculating and diagonalizing C is expensive and a much easier method is to use *singular value decomposition* on the data matrix itself:

$$X = USV^{\mathrm{T}}$$

such that

$$C = \frac{1}{N}X^{\mathrm{T}}X = \frac{1}{N}VS^{\mathrm{T}}U^{\mathrm{T}}USV^{\mathrm{T}} = \frac{1}{N}VS^2V^{\mathrm{T}}$$

Comparing this to the result from above, we see that the right-singular eigenvectors in V are the principal directions, and the variances are $\lambda_i = \frac{s_i^2}{N}$. You can see this by multiplying the equation above with V from the right. The columns of US are the principal components. Truncating all matrices like we saw in the last lesson to the first K entries to compute

$$X' = U[0 : K]S[0 : K]V[0 : K]^{\mathrm{T}}$$

gives the *reduced* representation of the data, which now only consists of the K most important feature combinations regarding the variance they explain for the original dataset.

3.4.3 Summary

PCA is extremely common in dimensionality reduction. It can be used as a way to visualize a very high-dimensional dataset in 2D or 3D. See for example the 1D, 2D, and 3D PCA representation of *Andersen's Iris dataset* we've briefly seen in lecture 01, but is also used as a preprocessing step for machine learning algorithms. As we've seen in the last lesson about SVD, it can be applied to images to retain only the most important information about an image to reduce the data size and to be able to use much smaller deep learning models. It improves the performance of many machine learning (or classical) algorithms, since it minimizes correlations between the new axes and, if desired, removes unwanted features which are most likely noise in the dataset. The newly found axes are still orthogonal. In image-processing, this transformation is sometimes also called the *Karhunen-Loève transformation.*

PCA can be generalized as principal curve analysis or principal surface analysis, and very commonly *kernel-PCA* which is a nonlinear version of PCA using the kernel-trick.

3.5 Kernel-PCA

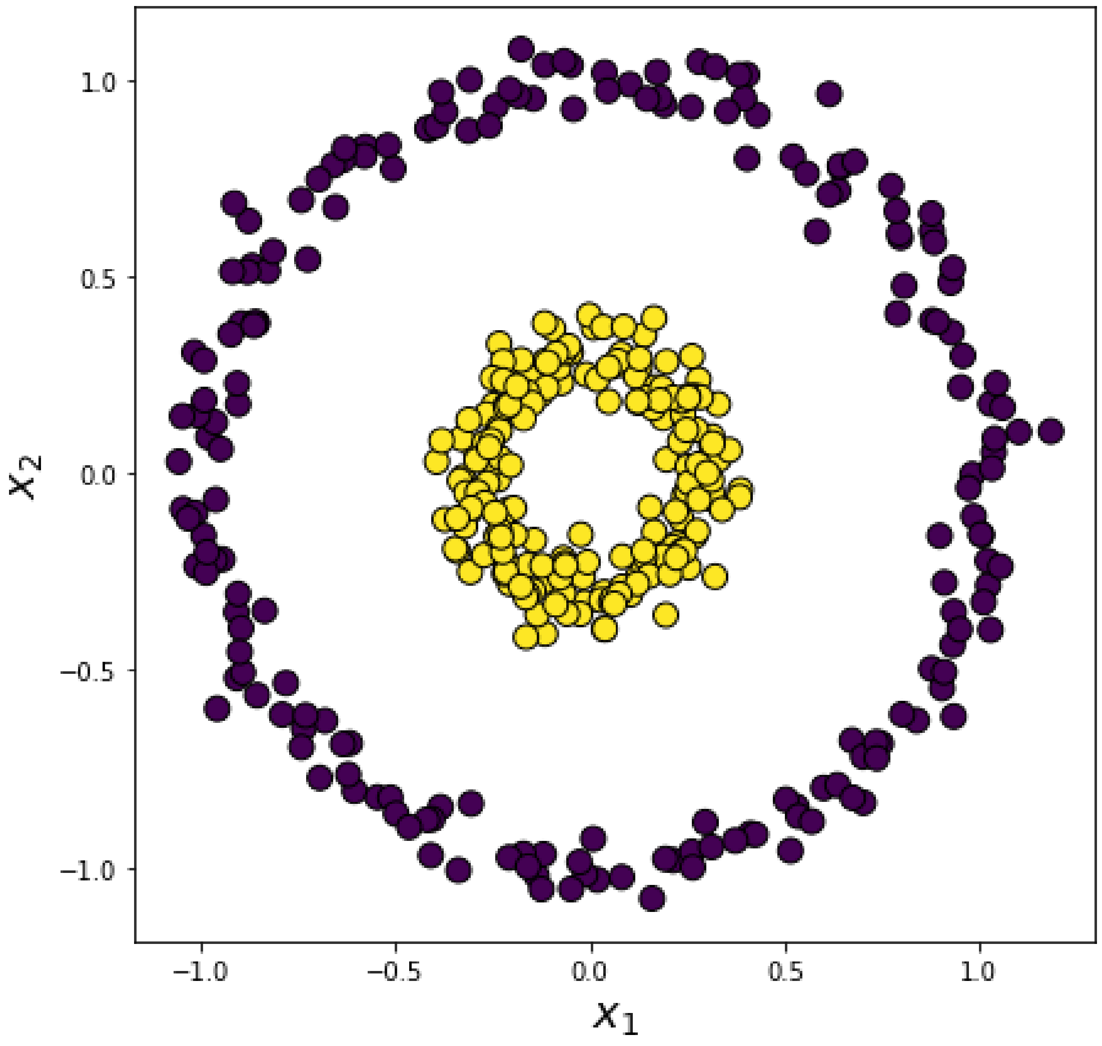
In the last lecture we saw how to apply the kernel trick to create a nonlinear formulation of support vector machines. The idea was to replace scalar products of the features with the kernel function of those features, which resulted in the product being calculated in a higher-dimensional space without us having to actually calculate the data points in that higher space. The kernel trick can also be used to create a nonlinear version of principal component analysis.

The idea of PCA is to diagonalize the data covariance matrix $\mathbf{C} = \frac{1}{N}\mathbf{X}\mathbf{X}^\mathrm{T} = \frac{1}{N}\sum_i^N x_i x_i^\mathrm{T}$, where $\mathbf{X}$ is the matrix containing the N data points for a problem and x_i are the feature vectors for each sample. With the kernel trick, the higher-dimensional covariance matrix becomes $\mathbf{C}' = \frac{1}{N}\sum_i^N \phi(x_i)\phi^\mathrm{T}(x_i)$ (see Quan Wang's paper about active shape modes and facial recognition for more info).

Equipped with this nonlinear version of PCA, we can apply it to highly nonlinear datasets, like the encircled rings from last lecture:

[1]:
```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=400, factor=.3, noise=.05)

plt.figure(figsize=(7,7))

plt.xlabel(r"$x_1$", fontsize=18)
plt.ylabel(r"$x_2$", fontsize=18)

plt.scatter(X[:,0], X[:,1], c=y, s=100, ec='k')
```

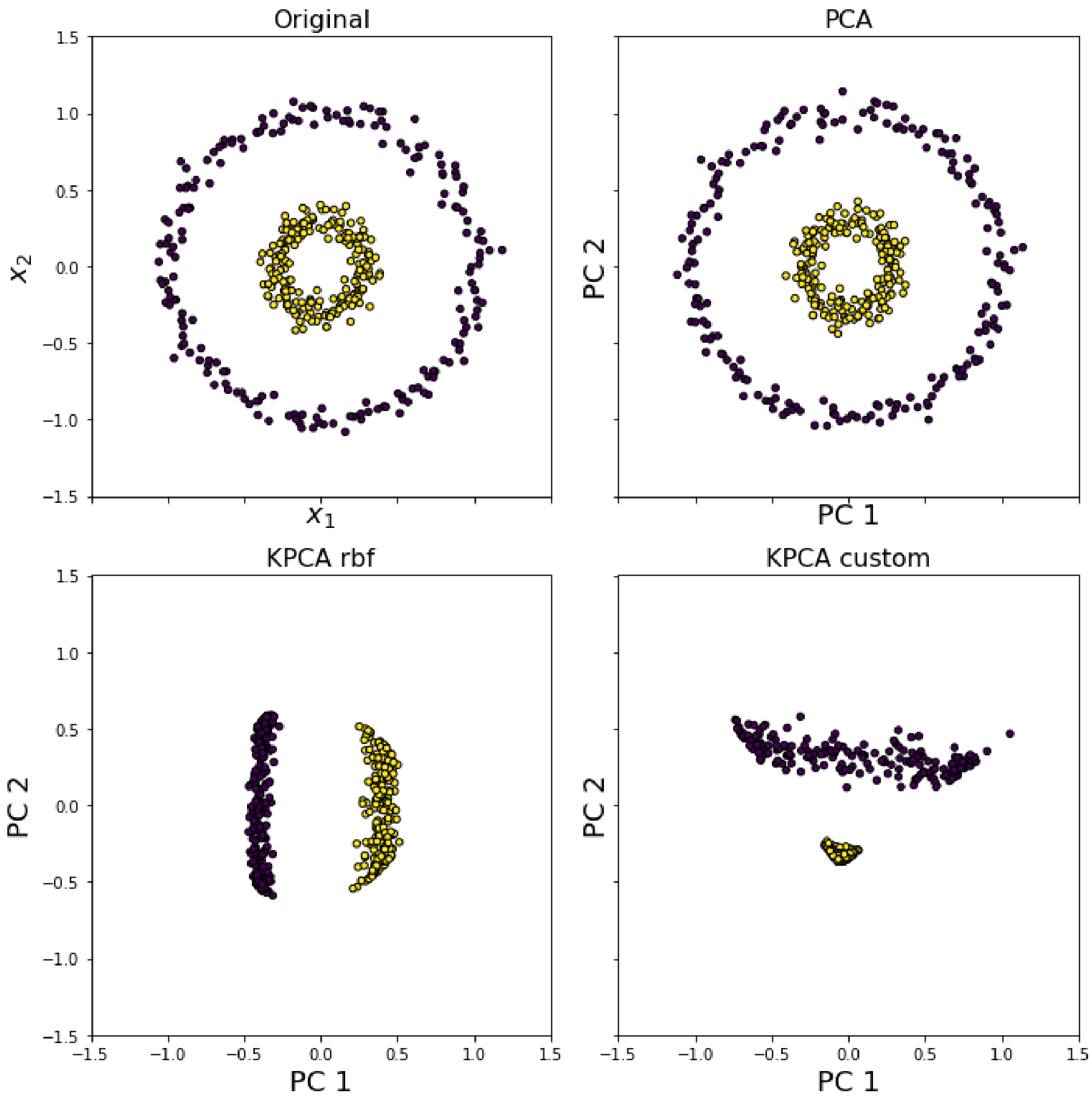
[1]:
```
<matplotlib.collections.PathCollection at 0x7f028746c710>
```



These are clearly not linearly separable. Let's see what happens if we try different kernel methods:

[6]:
```
import numpy as np
from sklearn.decomposition import PCA, KernelPCA

def custom_kernel(X, Y):
    # this function returns the Gram matrix of X and Y,
    # the entries G_ij are the scalar products <x_i,y_i>
    # try out different integer powers to see what happens
    # why do only certain values work?
    return (X**2).dot((Y**2).T)

# linear PCA
pca = PCA()
X_pca = pca.fit_transform(X)
```

```
# kernel-PCA with radial basis functions
rbf = KernelPCA(kernel="rbf", gamma=2)
X_rbf = rbf.fit_transform(X)

# custom kernel
custom = KernelPCA(kernel="precomputed")
X_custom = custom.fit_transform(custom_kernel(X,X))

# put values in lists for easier plotting
Xs = [X, X_pca, X_rbf, X_custom]
titles = [r"Original",
          r"PCA",
          r"KPCA rbf",
          r"KPCA custom"]


fig, axes = plt.subplots(2, 2, figsize=(10,10), sharex=True, sharey=True)
axes = axes.flatten()

for i,ax in enumerate(axes):
    ax.set_aspect('equal')
    ax.set_xlim(-1.5, 1.5)
    ax.set_ylim(-1.5, 1.5)
    ax.set_xlabel(r"PC 1", fontsize=18)
    ax.set_ylabel(r"PC 2", fontsize=18)
    ax.set_title(label=titles[i], fontsize=16)

    ax.scatter(Xs[i][:,0], Xs[i][:,1], c=y, s=20, edgecolor='k')


axes[0].set_xlabel(r"$x_1$", fontsize=18)
axes[0].set_ylabel(r"$x_2$", fontsize=18)

plt.tight_layout()
```



There's no chance to separate the data with linear PCA, but kernel-PCA finds ways if good kernels are provided (try this with the polynomial kernel. Does it also work?). In the last example, a custom kernel is provided to the algorithm, which sometimes yields the best results if the dataset and its topology is known well.

For dimensionality reduction, we'd actually choose fewer features than our dataset is originally comprised of. Imagine projecting the data graphs in the lower row onto an axis. Clearly, the points belonging to different classes in the original set are now well separated and identifiable by simply investigated a single dimension.

3.6 Generative PCA

Most machine learning algorithms can be viewed from (at least) two perspectives: linear algebra, and stochastics, although both are closely related. The implications are quite different though.

In the former lessons, we introduced PCA as a projective technique that finds principal directions and projects the original samples onto a lower-dimensional subspace in a way that maximizes the variance captured by the projected samples. These projections form the **latent space** of the dataset, which is a usually lower-dimensional representation of the data, and from which the original data can be reconstructed approximately. Latent spaces are encountered in many other machine learning algorithms, such as autoencoders or generative adversarial networks. The idea is always the same:

The latent space of a problem encodes usually not interpretable, but meaningful information about a higher-dimensional problem.

This property has powerful implications. It enables a probabilistic perspective of the algorithms which turn from projection and transformation algorithms into modelling the probability distribution underlying the actual dataset, from which many things can be learned about the process behind the data. For example, the latent space vectors might be Gauss-distributed, such that outliers in the original dataset produce latent vectors that are far outside of the Gaussian-distributed latent space, enabling *anomaly detection*.

We could also perform Bayesian model comparison, by making use of the marginal likelihood (see lectures 02 and 03), but we'll omit this here. The (probably) more interesting part is that probability distributions can be sampled and hence, be used to *generate* new data points that are, in a certain sense, *similar* to the original samples. Let's take a step back first. We saw that we can construct the latent space representation of a sample x_i with the reduction matrix R like so:

$$x_i' = z_= R^{\mathrm{T}} x_i$$

Usually, latent space vectors are denoted as z. We can also reconstruct the original sample by aplying R:

$$\tilde{x}_i = RR^{\mathrm{T}} x_i$$

where $\tilde{x}_i$ is the approximate reconstruction of x_i, since by projecting onto the principal directions, we lost the information about the directions with lower variances.

(*side note: PCA can be expressed as an optimization problem by demanding that the reconstruction $\tilde{X}$ of the data matrix should be as close as possible to X:*

$$\mathrm{PCA}(X) = \arg\min_{\tilde{X}} ||\tilde{X} - X||_{\mathrm{F}}$$

under the constraint rank$(\tilde{X}) = K$, where K is the number of singular values included in the reduction (which is equivalent to $\dim(Z)$*. F denotes the Frobenius norm. The Eckard-Young theorem proves that the truncated SVD is the optimal solution to this problem.*)

What we can do now is to take a latent representation z of some sample and change its components slightly. This will yield new data points that are in a meaningful way similar to the original samples, but not contained in the original dataset.

The probabilistic way to see this is that PCA models the probability to observe the sample x_i, given its latent representation z, the reduction matrix R, mean vector μ and covariance matrix Σ:

$$p(x|z, R, \mu, \Sigma) = \mathcal{N}(x|Rz + \mu, \lambda \mathbb{K})$$

where $\mathcal{N}$ is the normal distribution and $\lambda \mathbb{K}$ the covariance matrix in the latent space, which is diagonal and consisting of the eigenvalues of the data covariance matrix that survived the truncation.

We will explore how to use this generative method in the exercises today for creating and manipulating airfoil designs that will look like this:

[1]:
```
import pickle
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

with open('af_pca.pkl', 'rb') as f:
    af_pca = pickle.load(f)

with open('components.pkl', 'rb') as f:
    components = pickle.load(f)

latent_dim = 10

def plot_generated(**kwargs):
    fig = plt.figure(figsize=(7,7))

    input_vector = [weight for weight in kwargs.values()]

    generated = af_pca.inverse_transform(input_vector)

    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.15, 0.2])

    fig.patch.set_visible(False)
    plt.axis('off')

    plt.plot(generated.reshape(2,101)[0], generated.reshape(2,101)[1], lw=6,␣
 ↪color="black")

base_sample = 8
component_sliders = [widgets.FloatSlider(
        value=components[base_sample][i],
        min=min(components[:,i]),
        max=max(components[:,i]),
        step=(max(components[:,i] - min(components[:,i]))/10),
    ) for i in range(latent_dim)]

kwargs = {'c' + str(i) : slider for i,slider in enumerate(component_sliders)}
```

```
interact(plot_generated, **kwargs)
```

```
interactive(children=(FloatSlider(value=-0.03723766920148769, description='c0', max=0.19187495
```

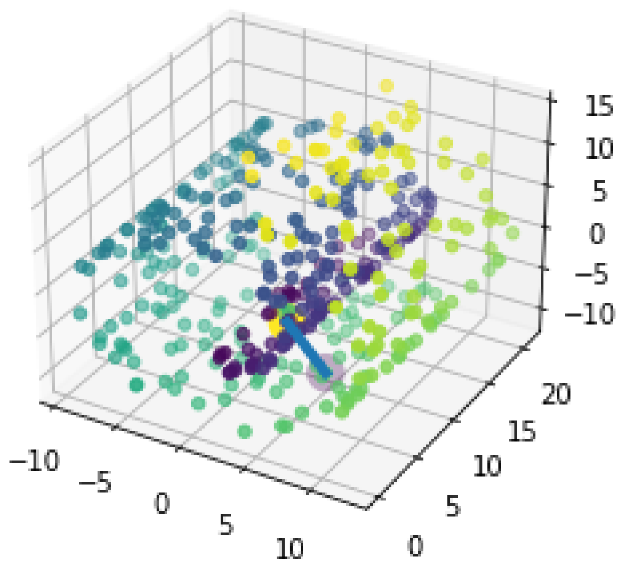
```
[1]: <function __main__.plot_generated(**kwargs)>
```

3.7 t-distributed Stochastic Neighborhood Embedding

Linear and kernel-techniques are limited in what information they can convey about highly convoluted high-dimensional data by the underlying idea that they preserve *euclidean distance* in the high-dimensional space. In convoluted cases, this can be quite misleading. See for example the swiss roll dataset:

```
[2]: %matplotlib notebook
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll, make_blobs
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

X, y = make_swiss_roll(n_samples=400, random_state=0)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:,0], X[:,1], X[:,2], c=y)

p1, p2 = 93, 332
ax.scatter(X[[p1,p2],0], X[[p1,p2],1], X[[p1,p2],2], lw=10, c=[0,1], zorder=500)
ax.plot(X[[p1,p2],0], X[[p1,p2],1], X[[p1,p2],2], lw=4, zorder=500)
```

```
[2]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f14dec58d68>]
```



The marked points have a small euclidean distance in the 3-dimensional space above (indicated by the blue line), but only respecting the swiss roll itself, they are very far apart. The swiss roll is an example of a **manifold** (recall lecture 01), which has various competing, but equivalent definitions. For us it should suffice to say that that at each point's immediate neighborhood, it looks like an $\mathcal{R}^n$. In this case, when you zoom in close enough, the data will look like an $\mathcal{R}^2$. This is a 2-dimensional manifold *embedded* into a 3-dimensional space.

Euclidean distance is not a faithful measure of distance on such a manifold and hence, all methods based on this measure are bound to fail, or at least yield suboptimal results. Enter **manifold learning** techniques. There are various competing algorithms, and depending on the circumstances, some work better or faster than others. Here, we'll concentrate solely on **t-distributed Stochastic Neighborhood Embedding**, which is an immensely popular technique for nonlinear dimensionality reduction, but not straight-forward to interpret in the end. The underlying idea of all manifold learning techniques is that the data dimensions are excessive, and the true distribution is low-dimensional, but "wrapped" in high-dimensional space.

The bird's eye perspective on t-SNE is that it calculates a "similarity" measure in the high-dimensional space, another one in a low-dimensional space, and an objective function which tries to maximize the similarity. In detail, the following things happen:

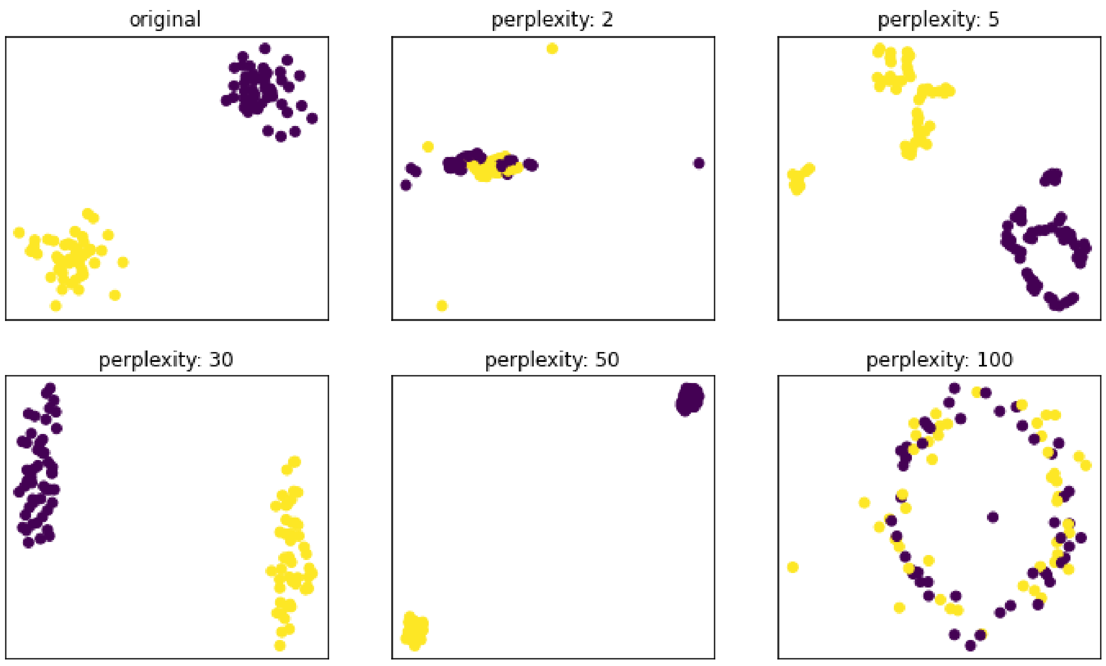
- The algorithm overlays a *Gaussian distribution* over each point x_i, then calculates the probabilities for some points x_j in the vicinity of x_i with $p(x_j|x_i) = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-||x_i - x_k||^2/2\sigma_i^2)}$
- Compute the similarity $p_{ij} = \frac{p(x_j|x_i)+p(x_i|x_j)}{2N}$ with $p_{ii} = 0$, $p_{ij} = p_{ji}$, and $\sum_j p_{ij} = 1$. The interpretation of this similarity is the probability that a point x_i would accept x_j as being its neighbor under the assumption that this probability is proportional to a Gaussian distribution centered at x_i.
- Very roughly speaking, the standard deviation σ, which gives an estimate of the radius in which points are sufficiently similar to the center point, is related to **perplexity**. This is an

important hyperparameter of t-SNE
- The similarity of the lower-dimensional points y_i is modeled by a *Student t-distribution* (sometimes called Cauchy distribution) $q_{ij} = \frac{(1+||y_i-y_j||^2)^{-1}}{\sum_k \sum_{l\neq k}(1+||y_k-y_l||^2)^{-1}}$
- Minimize the **Kullback-Leibler divergence** $\mathrm{KL}\left(P||Q\right) = \sum_{i\neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$, which measures the difference between two probability distributions.
- Minimizing the Kullback-Leibler divergence (also called relative entropy) is an optimization task, usually performed via gradient descent, and yields the low-dimensional representations y_i.

The Kullback-Leibler divergence is the *objective* or *loss function* to t-sne. *Perplexity* is a hyperparameter that balances local and global aspects of the data, meaning it provides an upper bound for how far points are allowed to deviate from the assumed underlying manifold to still count as belonging to the local neighborhood of a point.

t-SNE plots aren't straightforward to interpret due to their probabilistic nature, so let's look at a few examples:

[2]:
```
%matplotlib inline
from sklearn.manifold import TSNE

X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)

params = [[2, 1000],
          [5, 1000],
          [30, 1000],
          [50, 1000],
          [100, 1000]]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p[0], n_iter=p[1])
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p[0]))
    ax.set_xticks([])
    ax.set_yticks([])
```



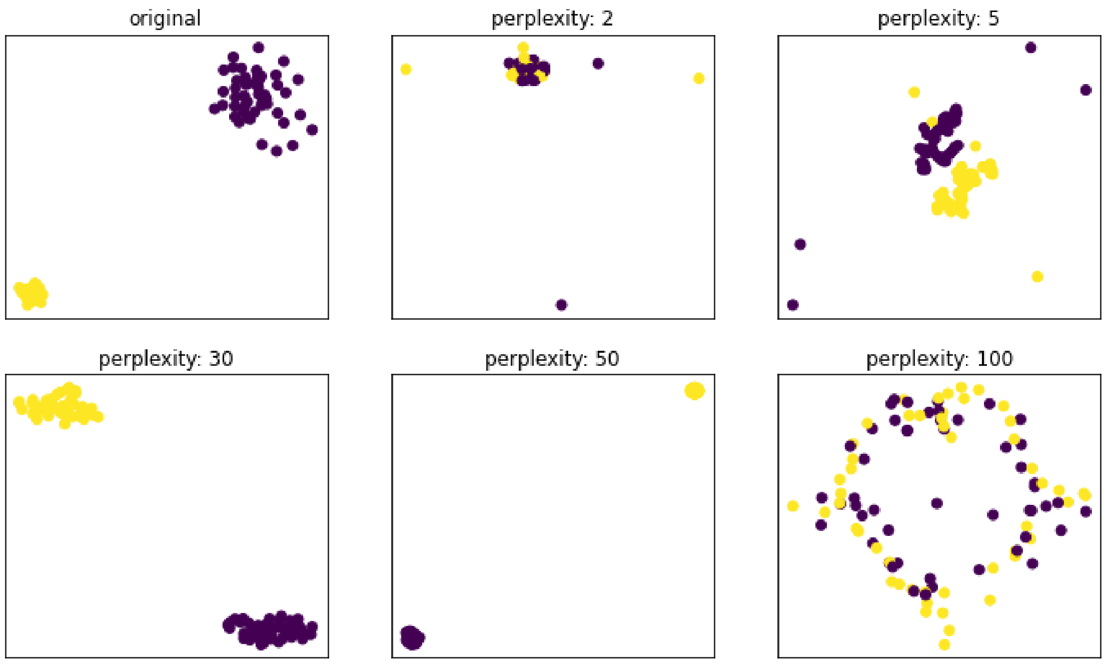
The flattened cluster you're seeing in the fourth image often mean that the algorithm was stopped too early, so increasing the number of iterations makes sense. We see that perplexity heavily controls the results. When it's too low, the neighborhood of points is too small and many clusters appear in the result. When it's too high, too many points belong to clusters, such that no good separation is found. In the original paper, the authors suggest using values between 5 and 50 (smaller than the number of points, obviously) and testing multiple times which values make most sense.

A perplexity of 50 seems to work well. Unfortunately, the sciki-learn implementation does not allow iteration counts below 250, so we cannot visualize what happens before that.

In the plot below, the same reductions are performed for two clusters that differ in size (standard deviation):

[7]:
```
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1,␣
 ↪cluster_std=[1, 0.2])

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])
```

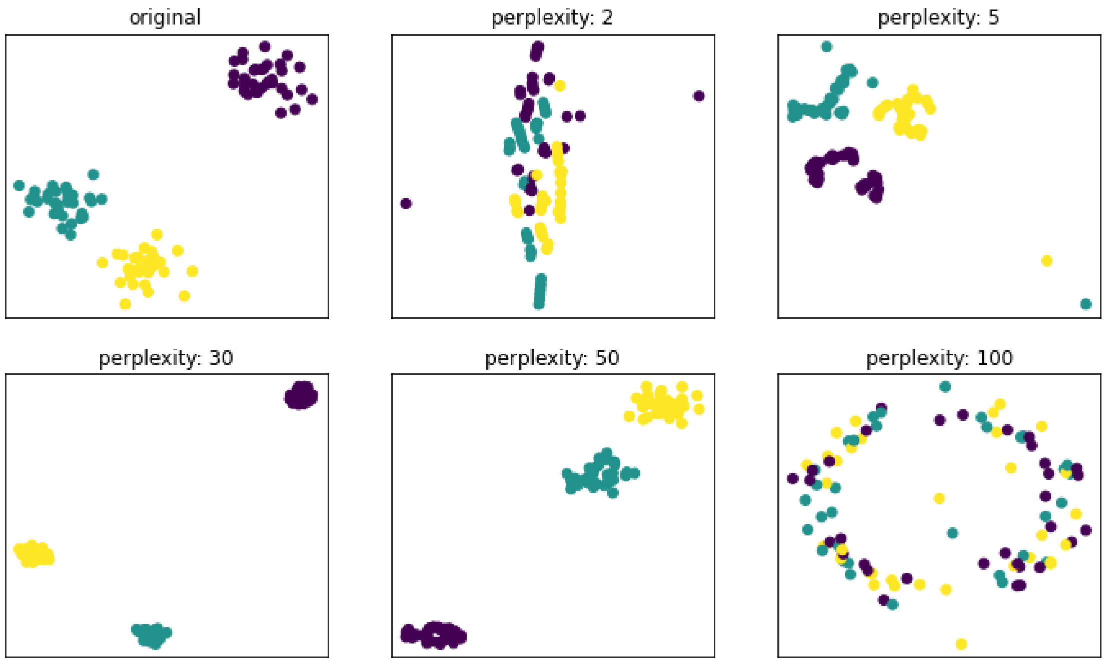
```
for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



The cluster size in the resulting t-SNE plot does not reflect the original cluster sizes. They look the same in the plots that converged. This is because the notion of similarity in t-SNE is a local one. As such, it expands dense clusters and contracts wide ones.

Let's see what happens to the distance of clusters themselves:

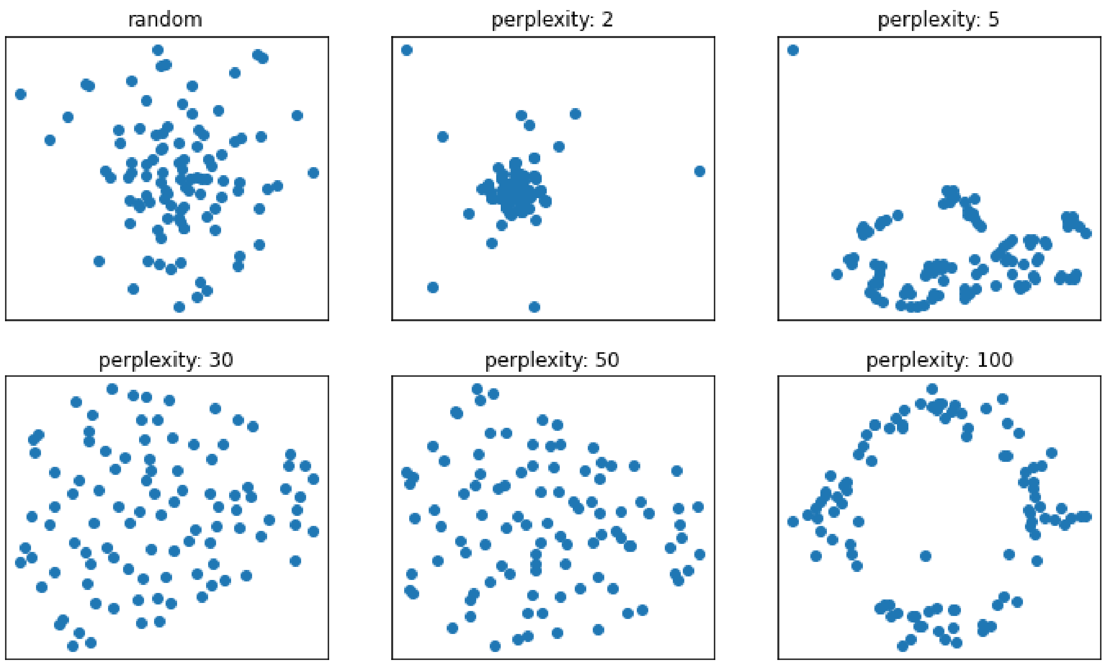
```
[8]: X, y = make_blobs(n_samples=100, centers=3, n_features=2, random_state=1)#,␣
      ↪cluster_std=[1, 0.2])

     params = [2, 5, 30, 50, 100]

     fig = plt.figure(figsize=(12,7))
     ax = fig.add_subplot(231)
     ax.scatter(X[:,0], X[:,1], c=y)
     ax.set_title("original")
     ax.set_xticks([])
     ax.set_yticks([])

     for i,p in enumerate(params):
         tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
         tsne_results = tsne.fit_transform(X)

         ax = fig.add_subplot(230+i+2)

         ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

         ax.set_title("perplexity: " + str(p))
         ax.set_xticks([])
         ax.set_yticks([])
```



In some of the plots, the global relative position of the clusters is reflected, while in others, it is not and instead they are more or less equidistant. t-SNE can, inprinciple, depict the relative position of clusters, but this necessitates fine-tuning the perplexity (the correct value depends on the number of points in each cluster). In general, the relative position of clusters in the reduced plots are not reliable predictors of the relative positions of the original clusters.

One reason for why it's necessary for t-SNE plots to do them multiple times with different hyper-parameters is that they sometimes portray random data as having clusters:

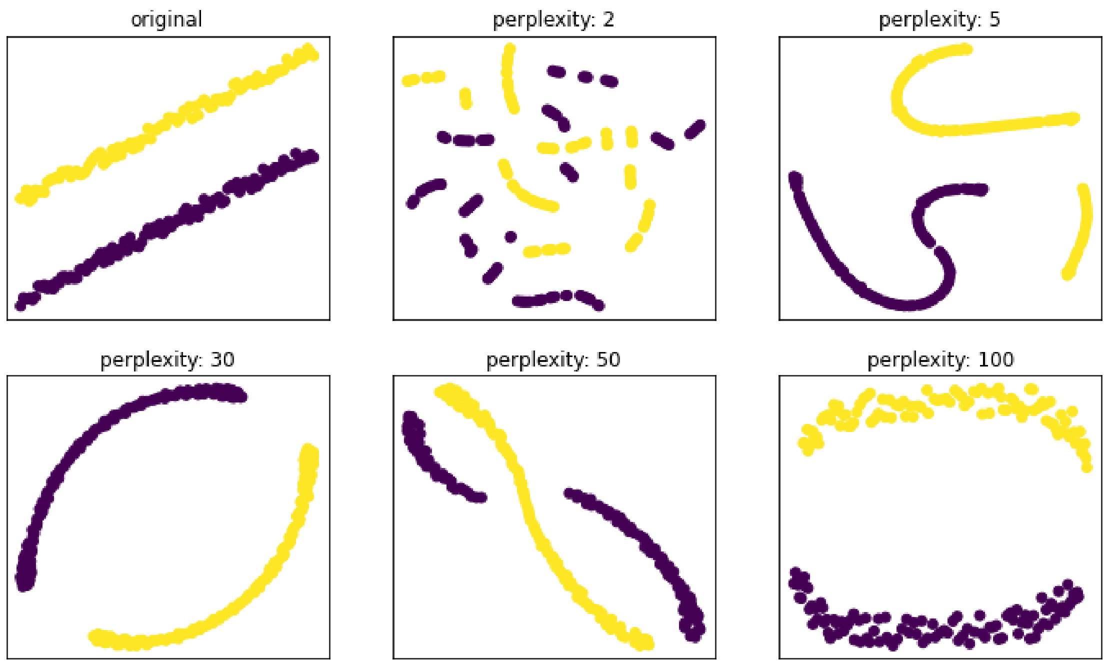
[9]:
```
from sklearn.datasets import make_gaussian_quantiles

X, Y = make_gaussian_quantiles(n_features=2)

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1])

ax.set_title("random")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1])

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



The very low and very high perplexity plots show some structure, where clearly none can be. This can be deceiving when dismissing the step to perform multiple plots.

Let's see what happens to parallel lines:

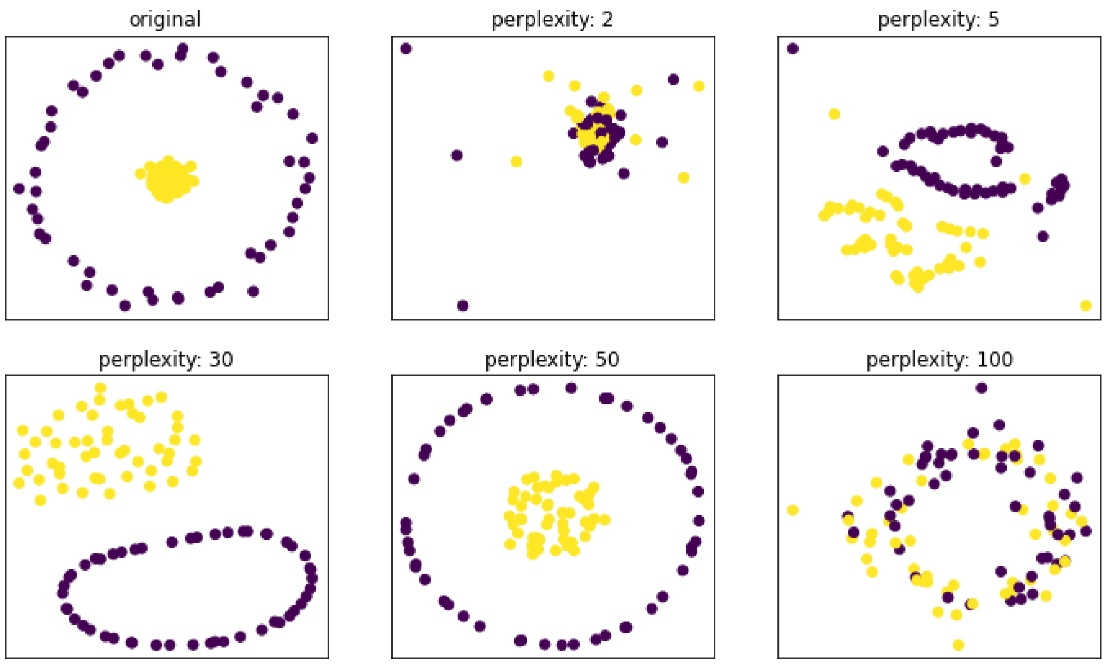
[20]:
```
import numpy as np

n_points = 100
x = np.linspace(0, 10, n_points)

y1 = 0.3*x + 0.3*np.random.random(n_points)
y2 = 0.3*x + 2 + 0.3*np.random.random(n_points)

X = np.zeros([2*n_points, 2])
X[:n_points,0] = x
X[n_points:,0] = x
X[:n_points,1] = y1
X[n_points:,1] = y2

y = np.zeros(2*n_points)
y[n_points:] = 1

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=500)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



Here, we needed to increase the number of iterations to get good results. The curves are depicted almost faithfully in most cases, but slightly curved outwards. The reason is that the regions in the middle of the lines are mode densely populated with samples than the ends. As explained above, t-SNE is sensitive to dense and wide distributions of points. This is reflected in the examples above.

The last important property is topology preservation. See for example the enclosed rings from before:

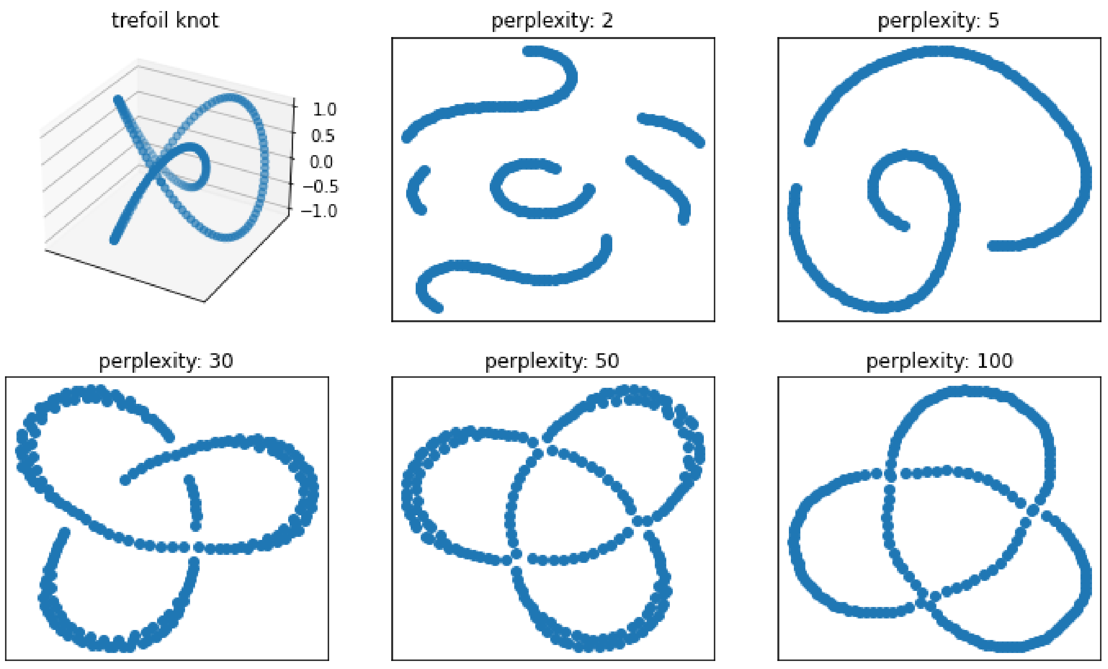
[22]:
```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=100, factor=.1, noise=.05)

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=500)
    tsne_results = tsne.fit_transform(X)
    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



Again, preserving the topology is highly dependent on fine-tuning the perplexity. This is also apparent when reducing the trefoil knot:

[4]:
```
%matplotlib notebook
from sklearn.datasets import make_circles

  = np.linspace(0, 2*np.pi, 200)
x = np.sin( )+2*np.sin(2* )
y = np.cos( )-2*np.cos(2* )
z = -np.sin(3* )

X = np.c_[x,y,z]

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231, projection='3d')
```

```
ax.scatter(X[:,0], X[:,1], X[:,2])

ax.set_title("trefoil knot")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1])

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```

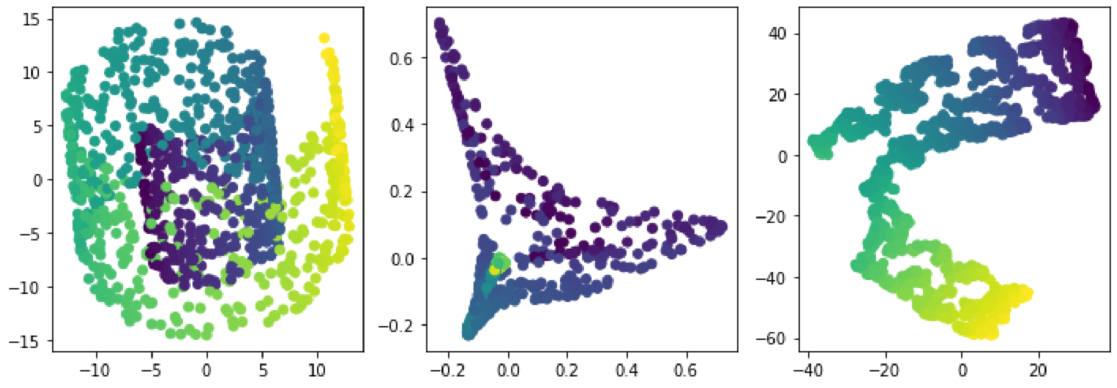


With lower perplexities, t-SNE cuts the continuous line at some places to produce severed strings. The actual topology is preserved with higher perplexities and appropriate numbers of iterations.

What's left is to perform t-SNE on the swiss roll and compare the result to PCA and kernel-PCA:

[26]:
```
%matplotlib inline
from sklearn.decomposition import PCA, KernelPCA
```

```
X_swiss, y_swiss = make_swiss_roll(n_samples=1000, random_state=0)

techniques = [PCA(n_components=2),
              KernelPCA(n_components=2, kernel='rbf', gamma=0.1),
              TSNE(n_components=2, perplexity=30, n_iter=5000, init='pca')]

results = [red.fit_transform(X_swiss) for red in techniques]

fig, axes = plt.subplots(1,len(results), figsize=(12,4))

for i,ax in enumerate(axes):
    ax.scatter(results[i][:,0], results[i][:,1], c=y_swiss)
```



The configuration here doesn't quite manage to completely unfold the swiss roll, but it comes close and detects the connections of the data points quite well. There are other, more advanced techniques that easily unroll the dataset, which all make use of the manifold property of the dataset. Linear PCA only projects the swiss roll onto a 2D space, kernel-PCA finds an embedding that slightly respects the position of points, but only t-SNE find an embedding that almost faithfully represents the manifold.

4 NN intuition

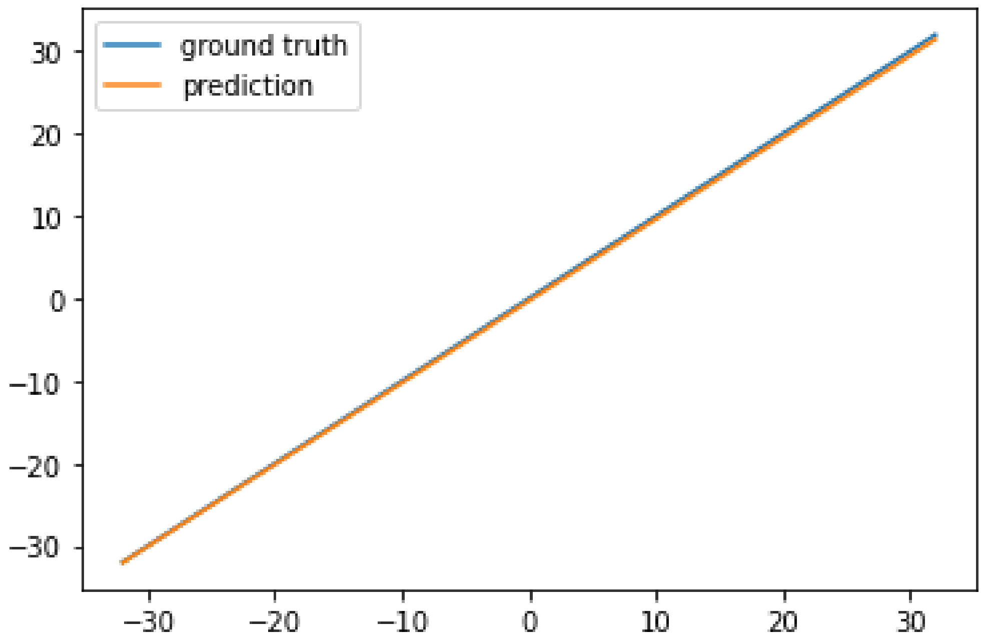
4.1 Linear Function

[5]:
```
%matplotlib inline
import matplotlib
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pylab
import tensorflow as tf
```

```
from tensorflow import keras
from tensorflow.keras import layers
xmax, n = 32, 2000

x = np.linspace(-xmax, xmax, n)
x = np.array(x)
y = np.zeros(x.size)
y = x #np.multiply(x,x)
print(y)
plt.plot(x,y, label="ground truth")

model = keras.Sequential(
    [
        layers.Dense(1, activation="linear", name="layer1"),
        layers.Dense(1, name="layer2"),
       # layers.Dense(1, name="layer3"),
       # layers.Dense(10, activation="relu", name="layer2"),
        #layers.Dense(20, activation="relu", name="layer3"),
       # layers.Dense(20, activation="relu", name="layer4"),
        #layers.Dense(12, activation="relu", name="layer5"),
      #  layers.Dense(1, name="layer6"),
    ]
)


def lr_scheduler(epoch, lr):
    decay_rate = 0.99
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr

callbacks = [
    keras.callbacks.LearningRateScheduler(lr_scheduler, verbose=0)
]

model.compile(loss='mean_squared_error', optimizer='adam',␣
 ↪metrics=['mean_squared_error'])
model.fit(x.reshape((x.shape[0],-1)), y, epochs=80, verbose=0,␣
 ↪validation_split=0.2,callbacks=callbacks)

plt.plot(x,model.predict(x), label="prediction")
plt.legend()
```

```
[-32.         -31.96798399 -31.93596798 …  31.93596798  31.96798399
  32.        ]
```

[5]: <matplotlib.legend.Legend at 0x7fd6105415f8>



[6]:
```
print(model.summary())
print(model.get_weights())
0.56854355*1.7588779
```

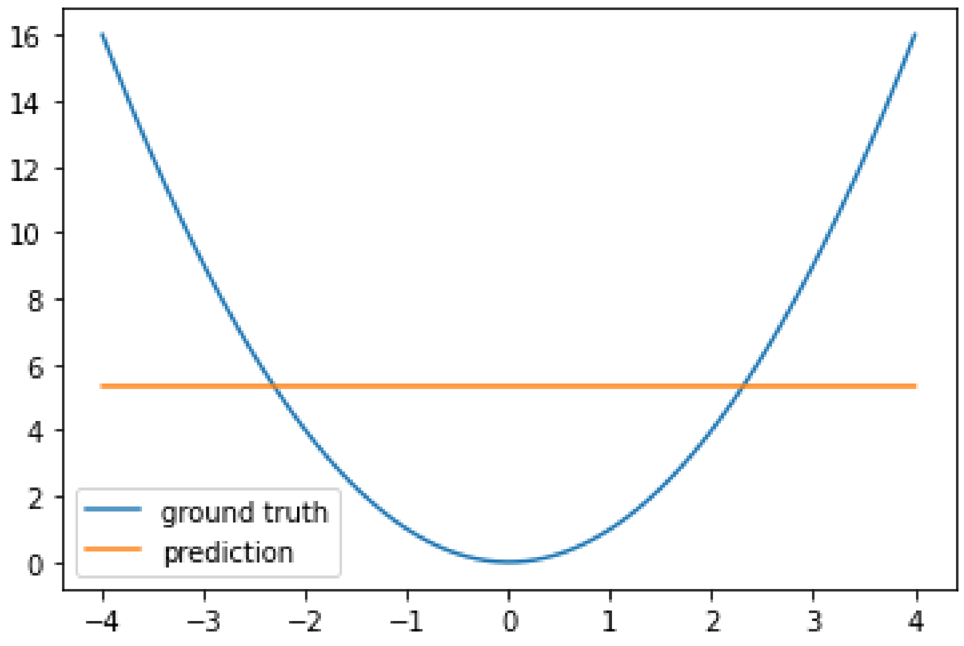
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
layer1 (Dense)               multiple                  2
_________________________________________________________________
layer2 (Dense)               multiple                  2
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
None
[array([[-1.3947442]], dtype=float32), array([-0.70992666], dtype=float32),
array([[-0.7114324]], dtype=float32), array([-0.8033232], dtype=float32)]
```

[6]: 0.9999986852825451

4.2 Quadratic function

[9]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
x=np.linspace(-4,4,2000)
x = np.array(x).reshape((-1, 1))
y = np.multiply(x,x)
#print(y)
model = LinearRegression()
model.fit(x, y)
print('intercept:', model.intercept_)
print('slope:', model.coef_)

plt.plot(x,y, label="ground truth")
plt.plot(x,model.predict(x), label="prediction")

plt.legend()


nx=20
A=np.ones((nx,2))
help=np.linspace(-4.0, 4.0, num=nx)
A[:,0]=help
b=np.transpose(np.column_stack(np.square(np.linspace(-4.0, 4.0, num=nx))))
AB=np.matmul(A.T,b)
AA=np.matmul(A.T,A)
invAA=np.linalg.inv(AA)
#print('intercept',np.matmul(invAA,AB)[1], "slope:",np.matmul(invAA,AB)[0])
```

```
intercept: [5.33866933]
slope: [[-3.95390781e-16]]
```



[10]:
```
%matplotlib inline
import matplotlib
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pylab
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.utils import shuffle
xmax, n = 4, 2000


x = np.linspace(-xmax, xmax, n)
x = np.array(x)
y = np.zeros(x.size)
print(x.shape, y.shape)
y = np.multiply(x,x)

x, y = shuffle(x,y)
model = keras.Sequential(
    [
        layers.Dense(1, activation="linear",name="layer1"),
         layers.Dense(5, activation="relu", name="layer2"),
        layers.Dense(5, activation="relu", name="layer3"),
        #layers.Dense(1, activation="relu", name="layer2"),
        #layers.Dense(20, activation="relu", name="layer3"),
       # layers.Dense(20, activation="relu", name="layer4"),
        #layers.Dense(12, activation="relu", name="layer5"),
        layers.Dense(1, name="layer6"),
    ]
)


def lr_scheduler(epoch, lr):
    if epoch == 0:
        lr=0.05
    decay_rate = 0.99
    decay_step = 10
    if epoch % decay_step == 0 and epoch:
        return lr * decay_rate
    return lr

callbacks = [
    keras.callbacks.LearningRateScheduler(lr_scheduler, verbose=0)
]
#model.compile(loss='mean_squared_error', optimizer='adam',␣
 ↪metrics=['mean_squared_error'])

model.compile(loss='mean_squared_error', optimizer='adam',␣
 ↪metrics=['mean_squared_error'])
model.fit(x.reshape((x.shape[0],-1)), y, epochs=50, verbose=0,␣
 ↪validation_split=0.2,callbacks=callbacks)
```

```
(2000,) (2000,)
```

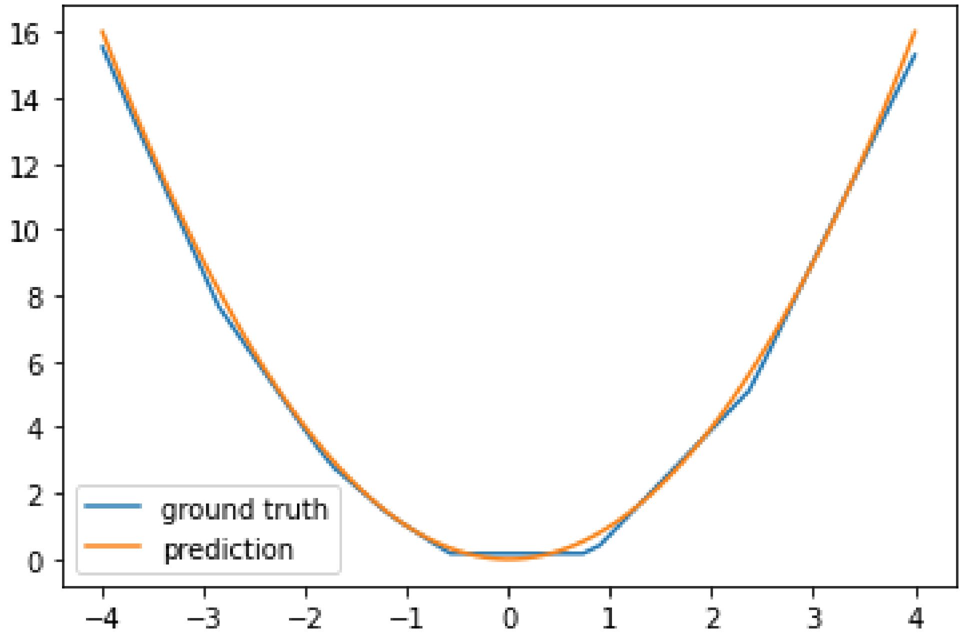
```
[10]: <tensorflow.python.keras.callbacks.History at 0x7fd5299ebb70>
```

```
[13]: model.summary()
weights = model.get_weights()
#print(weights)

print("intercept",weights[1])
print("slope",weights[0])

xin=np.linspace(-4,4,50)
xin = np.array(xin).reshape((-1, 1))
yout=np.zeros(xin.shape)
for i in range(0,xin.size):
    yout[i]=model.predict(xin[i])
#print("xin",xin)
#print('yout',yout)
```

```
plt.plot(xin,yout,label="ground truth")
x = np.linspace(-xmax, xmax, n)
x = np.array(x)
y = np.zeros(x.size)
y = np.multiply(x,x)
plt.plot(x,y, label="prediction")
plt.legend()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
layer1 (Dense)               multiple                  2
_________________________________________________________________
layer2 (Dense)               multiple                  10
_________________________________________________________________
layer3 (Dense)               multiple                  30
_________________________________________________________________
layer6 (Dense)               multiple                  6
=================================================================
Total params: 48
Trainable params: 48
Non-trainable params: 0
_________________________________________________________________
intercept [0.22977716]
slope [[-1.3242841]]
```

```
[13]: <matplotlib.legend.Legend at 0x7fd529538f60>
```


ground truth
prediction