Control and Optimization 2023/2024 (2nd semester)

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

Professors:

A. Pedro Aguiar (pedro.aguiar@fe.up.pt), **M. Rosário Pinho** (mrpinho@fe.up.pt)

FEUP, Feb. 2024

Notebook #01: Introduction to the course

1- What is the Colaboratory?

This section is a partially copy of the "Colab getting started" file. See the original at https://colab.research.google.com/

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

In [ ]:
```
seconds_in_a_day = 24 * 60 * 60
seconds_in_a_day
```

Out[ ]:
```
86400
```

To execute the code in the above cell, select it with a click and then either press the play button to the left of the code, or use the keyboard shortcut **"Command/Ctrl+Enter"**. To edit the code, just click the cell and start editing.

Variables that you define in one cell can later be used in other cells:

In [ ]:
```
seconds_in_a_week = 7 * seconds_in_a_day
seconds_in_a_week
```

Out[ ]:
```
604800
```

Colab notebooks allow you to combine **executable code** and **rich text** in a single document, along with **images**, **HTML**, **LaTeX** and more. When you create your own Colab notebooks, they are stored in your Google Drive account. You can easily share your Colab notebooks with co-workers or friends, allowing them to comment on your notebooks or even edit them. To learn more, see Overview of Colab. To create a new Colab notebook you can use the File menu above, or use the following link: create a new Colab notebook.

Colab notebooks are Jupyter notebooks that are hosted by Colab. To learn more about the Jupyter project, see jupyter.org.

2- Resources (Python)

Here goes a list of references where you can find more information about jupyter notebooks and Python.

- Jupyter notebook: https://jupyter.org
- The Markdown Guide: https://www.markdownguide.org
- Introduction to Python (a collection of notebooks): https://nbviewer.jupyter.org/github/ehmatthes/intro_programming/blob/master/noteb
- A Crash Course in Python for Scientists: https://nbviewer.jupyter.org/gist/rpmuller/5920182
- Scientific Computing with Python: https://nbviewer.jupyter.org/url/atwallab.cshl.edu/teaching/QBbootcamp3.ipynb
- For users who are more familiar with Matlab, check here a comparison of NumPy and analog functions in Matlab: https://numpy.org/devdocs/user/numpy-for-matlab-users.html

3- Python Essentials: a quickstart

This section provides a quick tour to some basic and useful functions in Python.

Useful libraries

Some interesting and useful Python libraries:

Numpy

NumPy is the fundamental package for scientific computing with Python.
NumPy user guide: https://numpy.org/doc/stable/user/index.html

Scipy

The SciPy library is a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.
Scipy user guide: https://scipy.github.io/devdocs/tutorial/index.html#user-guide

Matplotlib

Matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
Matplotlib user guide: https://matplotlib.org/stable/contents.html

Sympy

SymPy is a Python library for symbolic mathematics.
SymPy tutorial: https://docs.sympy.org/latest/tutorial/index.html#tutorial

Importing

To import a specific library `x`, simply type `import x`

To access the library function `f`, type `x.f`

If you want to change the library name to `y`, type `import x as y`

In [ ]:
```
import numpy as np

np.ones((3,1))
```

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

Data Types

Numbers

In [ ]:
```
x = 3
print(x, type(x))
```

Python Operators :

- Arithmetic operator
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators
- Membership operators

In [ ]:
```
print(x + 1)   # Addition
print(x - 1)   # Subtraction
print(x * 2)   # Multiplication
print(x ** 2)  # Exponentiation
```

In [ ]:
```
x += 1
print(x)
x *= 2
print(x)
```

In [ ]:
```
y = 2.5
print(type(y))
print(y, y + 1, y * 2, y ** 2)
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&` , `||` , etc.):

In [ ]:
```
t, f = True, False
print(type(t))
```

Now we let's look at the operations:

In [ ]:
```
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

Floats and Integers

For floating point division, use `/`

In [ ]:
```
1 / 4
```

To perform integer division, use `//` (integer division returns the floor)

In [ ]:
```
1 // 4
```

Strings

In [ ]:
```
print("Machine Learning")
```

Unlike MATLAB/C, doubles quotes and single quotes are the same thing. Both represent strings. `'+'` concatenates strings

In [ ]:
```
# This is a comment
"ABC" + 'DEF'
```

Lists

A list is a mutable array of data. That is we can change it after we create it. They can be created using square brackets []

Important functions:

- `'+'` appends lists.
- `len(x)` to get length

In [ ]:
```
x = [1, 2, 3] + ["ABC"]

print(x)
```

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

Sets

A set is an unordered collection of distinct elements.

- There is no duplicate elements.
- Support mathematical operations like union, intersection, difference, and symmetric difference.
- `set()` function (or `{}`) can be used to create sets. To create an empty set you have to use `set()`.

In [ ]:
```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True
print('fish' in animals)  # prints "False"
```

In [ ]:
```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

In [ ]:
```
animals.add('cat')       # Adding an element that is already in the set d
print(len(animals))
animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

In [ ]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

Tuples

A tuple is an unmutable list. They can be created using round brackets ().

They are usually used as inputs and outputs to functions

In [ ]:
```
t = ("A", "B", "C") + (1, 2, 3)
print(t)
```

Numpy Array

Numpy array is similar to a list with multidimensional support and more functions.

Creating a numpy array

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

print(A)
```

In [ ]:
```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

In [ ]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b)
```

In [ ]:
```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

Numpy also provides many functions to create arrays:

In [ ]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

In [ ]:
```
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

In [ ]:
```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

In [ ]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

In [ ]:
```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

Getting the shape of array

In [ ]:
```
A.shape
```

Elementwise operation

One major advantage of using numpy arrays is that arithmetic operations on numpy arrays correspond to elementwise operations.

In [ ]:
```
print(A + 2)
```

Matrix multiplication

You can use @ to do matrix multiplication

In [ ]:
```
print(A@A)
```

Alternatively, you can use np.matrix

In [ ]:
```
np.matrix(A) * np.matrix(A)
```

Slicing numpy arrays

Numpy uses pass-by-reference semantics so it creates views into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

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

We slice an array from a to b-1 with `[a:b]`

In [ ]:
```
y = x[0:4]
print(y)
```

Because slicing does not copy the array, changing `y` changes `x`

In [ ]:
```
y[0] = 7
print(x)
print(y)
```

To actually copy x, we should use .copy()

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

Useful Numpy function: r_

We use `r_` to create integer sequences in numpy arrays

`r_[0:N]` creates an array listing every integer from 0 to N-1

`r_[0:N:m]` creates an array listing every `m` th integer from 0 to N-1

In [ ]:
```
import numpy as np # by convention, import numpy as np
from numpy import r_ # import r_ function from numpy directly, so that we

print(np.r_[-5:5]) # every integer from -5 ... 4

print(np.r_[0:5:2]) # every other integer from 0 ... 4
```

Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

In [ ]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

In [ ]:
```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

In [ ]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

In [ ]:
```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
row_r3 = a[[1], :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
```

In [ ]:
```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

In [ ]:
```
a = np.array([[1,2], [3, 4], [5, 6]])
print(a)

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])   # indexes = (0,0), (1,1), (2, 0)

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

In [ ]:
```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

In [ ]:
```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

In [ ]:
```
# Create an array of indices
b = np.array([0, 2, 0, 1])

# note: np.arange(4) = [0 1 2 3]

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

In [ ]:
```
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

In [ ]:
```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

In [ ]:
```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

In [ ]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

In [ ]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

In [ ]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

In [ ]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

In [ ]:
```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Note that unlike MATLAB, `*` **is elementwise multiplication, not matrix multiplication**.

We instead use the **dot function** to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

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

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

You can also use the `@` operator which is equivalent to numpy's `dot` operator.

In [ ]:
```
print(v @ w)
```

In [ ]:
```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

In [ ]:
```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

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

print(np.sum(x))          # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

In [ ]:
```
print(x)
print("transpose\n", x.T)
```

In [ ]:
```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

Broadcasting

Broadcasting is a powerful mechanism that allows **numpy to work with arrays of different shapes when performing arithmetic operations**. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

In [ ]:
```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv` . We could implement this approach like this:

In [ ]:
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

In [ ]:
```
y = x + vv  # Add x and vv elementwise
print(y)
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

In [ ]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

In [ ]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

In [ ]:
```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

In [ ]:
```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

In [ ]:
```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

In [ ]:
```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

If statements

In [ ]:
```
x = int(input("Please enter an integer: "))

if x < 0:
  x = 0
  print('Negative changed to zero')
elif x == 0:
  print('Zero')
elif x == 1:
  print('Single')
else:
  print('More')
```

Loop statements

Python's for statement iterates over the items of any list or string in the order that they are appearing in the sequence.

You can loop over the elements of a list like this:

for

In [ ]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal, len(animal))
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function. This function returns a tupple containing a count (from start which defaults to 0) and the values.

In [ ]:
```
animals = ['cat', 'dog', 'monkey']

animals_enum = list(enumerate(animals))
print(animals_enum) # (idx_0, object_0), (idx_1, object_1), ...

animals_enum = list(enumerate(animals, start = 10))
print(animals_enum) # (idx_0 + Start, object_0), (idx_1 + Start, object_1

for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

In [ ]:
```
fruit  = ["apple", "banana", "orange"]

for i in range(len(fruit)):
  print(fruit[i], i)
```

while

The `while` statement repeates the execution as long as an expression is true:

In [ ]:
```
x = 0

while x < 5 :
  print(x)
  x+=1
```

break and continue Statements, and else Clauses on Loops

The `break` statement breaks out of the innermost enclosing for or while loop.

An `else` on a loop statements is executed when the loop terminates through **exhaustion** of the iterable (with for) or when the condition becomes false (with while).

It is **not executed** when the loop is terminated by a break statement.

In [ ]:
```
  for n in range(1, 5):

     for x in range(1, 3):
         print('n =', n, ' x=', x)

         if n > x:
             print(n, 'is higher than x=', x)
             break
     else:
         print('ending x =', x)
```

List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

This code can be written using List comprehensions:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

In [ ]:
```
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

Functions

Python functions are defined using the `def` keyword. For example:

In [ ]:
```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

We will often define functions to take optional keyword arguments, like this:

In [ ]:
```
def hello(name, loud=False):   #Default Argument Values
    if loud:
        print('HELLO, {}'.format(name.upper()))
    else:
        print('Hello, {}!'.format(name))

hello('Bob')
hello('Fred', loud=True)
```

You might object that hello is not a function but a procedure since it doesn't return a value.

- The `return` statement returns with a value from a function.
- `return` without an expression argument returns None. Falling off the end of a function also returns None.
- the method `result.append(a)` calls a method of the list object `result`

In [ ]:
```
def fib2(n):  # return Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)    # see below
        a, b = b, a+b
    return result

f100 = fib2(100)
print(f100)
```

Classes

The syntax for defining classes in Python is straightforward:

In [ ]:
```
class Greeter:
    """A simple example class"""
    i = 12345

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
          print('HELLO, {}'.format(self.name.upper()))
        else:
          print('Hello, {}!'.format(self.name))

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"

f = Greeter('John')
g.i = 100
print(g.i)            # is not shared because is not list or dictionary
print(f.i)

print(g.name)
print(f.name)
```

You need to pay attention to **mutable** objects such as **lists** and **dictionaries** because:

In [ ]:
```
class Dog:

    tricks = []             # mistaken use of a class variable.
                            # This will be like "static" in C++ (a member

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Correct design of the class should use an instance variable:

In [ ]:
```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

Plotting

We will use `matplotlib.pyplot` to plot signals and images.

By convention, we import `matplotlib.pyplot` as `plt`.

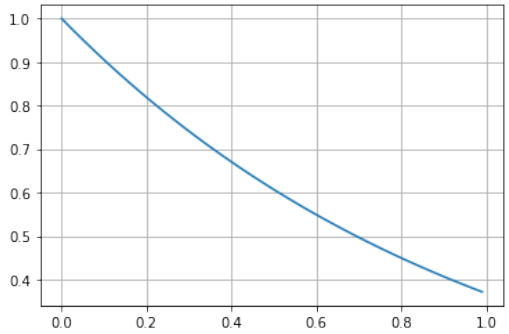
To display the plots inside the browser, we use the command `%matplotlib inline`

In [ ]:
```
import matplotlib.pyplot as plt # by convention, we import pyplot as plt

# plot in browser instead of opening new windows
%matplotlib inline
```

In [ ]:
```
# Generate signals
x = np.r_[:1:0.01] # if you don't specify a number before the colon, the
y1 = np.exp( -x )
y2 = np.sin( x*10.0 )/4.0 + 0.5
```

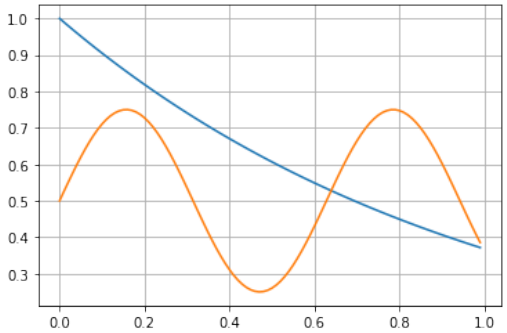
Plotting one signal

In [ ]:
```
plt.figure()
plt.plot( x, y1 )
plt.grid()
```



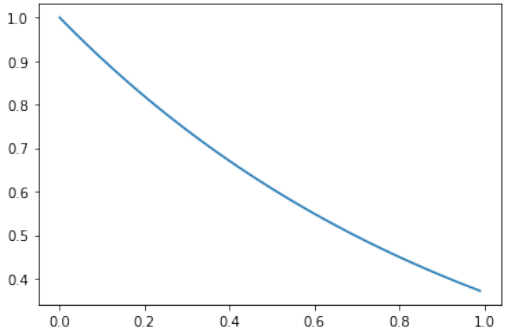
Plotting multiple signals in one figure

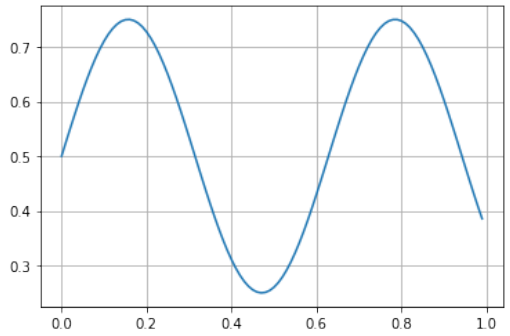
In [ ]:
```
plt.figure()
plt.plot( x, y1 )
plt.plot( x, y2 )
plt.grid()
```



Plotting multiple signals in different figures

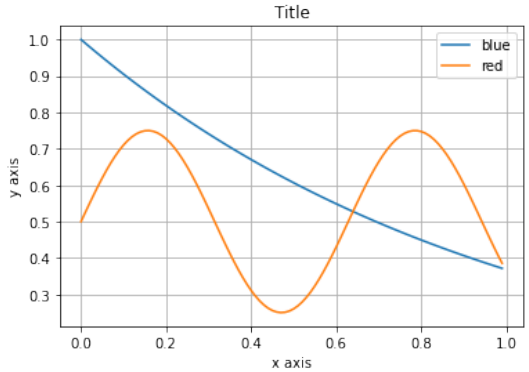
In [ ]:
```
plt.figure()
plt.plot( x, y1 )
plt.figure()
plt.plot( x, y2 )
plt.grid()
```





You can also add title and legends using `plt.title()` and `plt.legend()`

In [ ]:
```
fig = plt.figure()
plt.plot( x, y1 )
plt.plot( x, y2 )
plt.xlabel( "x axis" )
plt.ylabel( "y axis" )

plt.title( "Title" )

plt.legend( ("blue", "red") )
plt.grid()
```



Saving Figures to File

In [ ]:
```
fig.savefig('my_figure.png')
```

You can check the file `my_figure.png` in the current working directory:

In [ ]:
```
!ls
```

and display it again

In [ ]:
```
from IPython.display import Image
Image('my_figure.png')
```

Symbolic manipulation

We can create and manipulate expressions in Sympy. Consider this expression

$$(a+b)^3$$

that we would like to expand it. First, we create the symbolic variables a, b:

In [ ]:
```
import sympy as sym
a, b = sym.symbols('a, b')
```

and the expression...

In [ ]:
```
expr = (a + b) ** 3
expr
```

Out[ ]: $(a+b)^3$

Let us expand it

In [ ]:
```
expr.expand()
```

Out[ ]: $a^3 + 3a^2b + 3ab^2 + b^3$

Note that we can also get Sympy to produce the LaTeX code for future use:

In [ ]:
```
sym.latex(expr.expand())
```

Out[ ]:
```
'a^{3} + 3 a^{2} b + 3 a b^{2} + b^{3}'
```

4- Control and Optimization in Python

Here goes a very limited list of nice references/libraries.

- Python Control Systems Library: https://python-control.readthedocs.io
- ...

5- Exercises

Activity 1

Solving numerically Ordinary Differential Equations (ODEs)

Let's compare the **Euler method** with the fourth-order **Runge-Kutta** method for solving ODEs.

1.1.

Consider the following ODE

$$\dot{x} = f(t, x), \quad x(0) = x_0$$

with

$$f(t, x) = -x$$

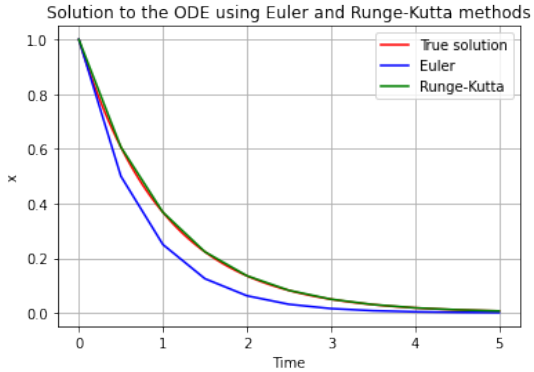
Obtain the numerical solution for $x_0 = 1$ from $t \in [0, 5]$ using the Euler method and the fourth-order Runge-Kutta method. Compare the results with the true solution using different time step sizes.

In [ ]:
```python
#To complete

import numpy as np
import matplotlib.pyplot as plt

# function to evaluate the derivative of the solution
def f(t, x):
#To complete    return ...

# the true solution to the problem
def true_solution(t0, x0, t):
#To complete    return ...


# Function to implement the Euler method
def euler(f, t0, x0, h, n):
    # t_curr: current time
    t_curr = t0
    # x: current value of the solution
    x = x0
    # t_array: list to store the time values
    t_array = [t0]
    # x_array: list to store the solution values
    x_array = [x0]
    for i in range(n):
        # x_new: updated value of the solution using the Euler method
 #To complete       x_new = ...
        # update the current time
        t_curr = t_curr + h
        # add the updated time and solution values to the lists
        t_array.append(t_curr)
        x_array.append(x_new)
        # update the current solution value
        x = x_new
    return t_array, x_array

# Function to implement the Runge-Kutta method
def runge_kutta(f, t0, x0, h, n):
    # t_curr: current time
    t_curr = t0
    # x: current value of the solution
    x = x0
    # t_array: list to store the time values
    t_array = [t0]
    # x_array: list to store the solution values
    x_array = [x0]
    for i in range(n):
        # k1, k2, k3, k4: intermediate values used in the Runge-Kutta met
        k1 = h * f(t_curr, x)
        k2 = h * f(t_curr + h/2, x + k1/2)
        k3 = h * f(t_curr + h/2, x + k2/2)
        k4 = h * f(t_curr + h, x + k3)
        # x_new: updated value of the solution using the Runge-Kutta meth
        x_new = x + (k1 + 2*k2 + 2*k3 + k4) / 6
        # update the current time
        t_curr = t_curr + h
        # add the updated time and solution values to the lists
        t_array.append(t_curr)
        x_array.append(x_new)
        # update the current solution value
        x = x_new
    return t_array, x_array


# initial time
t0 = 0
# initial value of the solution
x0 = 1
# time step
h = 0.5
# final time
tf = 5
# number of time steps
n = int((tf - t0) / h)

t_euler, x_euler = euler(f, t0, x0, h, n)
t_rk, x_rk = runge_kutta(f, t0, x0, h, n)

# array of 100 time values for the true solution
t_true = np.linspace(t0, tf, 100)
x_true = true_solution(t0, x0, t_true)

plt.plot(t_true, x_true, '-r', label='True solution')
plt.plot(t_euler, x_euler, '-b', label='Euler')
plt.plot(t_rk, x_rk, '-g', label='Runge-Kutta')
plt.legend(loc='best')
plt.grid()
plt.xlabel('Time')
plt.ylabel('x')
plt.title('Solution to the ODE using Euler and Runge-Kutta methods')
plt.show()
```

In [ ]:
```python
#Solution
```

```python
import numpy as np
import matplotlib.pyplot as plt

# function to evaluate the derivative of the solution
def f(t, x):
    return -x

# the true solution to the problem
def true_solution(t0, x0, t):
    return x0 * np.exp(-(t-t0))


# Function to implement the Euler method
def euler(f, t0, x0, h, n):
    # t_curr: current time
    t_curr = t0
    # x: current value of the solution
    x = x0
    # t_array: list to store the time values
    t_array = [t0]
    # x_array: list to store the solution values
    x_array = [x0]
    for i in range(n):
        # x_new: updated value of the solution using the Euler method
        x_new = x + h * f(t_curr, x)
        # update the current time
        t_curr = t_curr + h
        # add the updated time and solution values to the lists
        t_array.append(t_curr)
        x_array.append(x_new)
        # update the current solution value
        x = x_new
    return t_array, x_array

# Function to implement the Runge-Kutta method
def runge_kutta(f, t0, x0, h, n):
    # t_curr: current time
    t_curr = t0
    # x: current value of the solution
    x = x0
    # t_array: list to store the time values
    t_array = [t0]
    # x_array: list to store the solution values
    x_array = [x0]
    for i in range(n):
        # k1, k2, k3, k4: intermediate values used in the Runge-Kutta met
        k1 = h * f(t_curr, x)
        k2 = h * f(t_curr + h/2, x + k1/2)
        k3 = h * f(t_curr + h/2, x + k2/2)
        k4 = h * f(t_curr + h, x + k3)
        # x_new: updated value of the solution using the Runge-Kutta meth
        x_new = x + (k1 + 2*k2 + 2*k3 + k4) / 6
        # update the current time
        t_curr = t_curr + h
        # add the updated time and solution values to the lists
        t_array.append(t_curr)
        x_array.append(x_new)
        # update the current solution value
        x = x_new
    return t_array, x_array
```

```python
# initial time
t0 = 0
# initial value of the solution
x0 = 1
# time step
h = 0.5
# final time
tf = 5
# number of time steps
n = int((tf - t0) / h)

t_euler, x_euler = euler(f, t0, x0, h, n)
t_rk, x_rk = runge_kutta(f, t0, x0, h, n)

# array of 100 time values for the true solution
t_true = np.linspace(t0, tf, 100)
x_true = true_solution(t0, x0, t_true)

plt.plot(t_true, x_true, '-r', label='True solution')
plt.plot(t_euler, x_euler, '-b', label='Euler')
plt.plot(t_rk, x_rk, '-g', label='Runge-Kutta')
plt.legend(loc='best')
plt.grid()
plt.xlabel('Time')
plt.ylabel('x')
plt.title('Solution to the ODE using Euler and Runge-Kutta methods')
plt.show()
```



In [ ]:

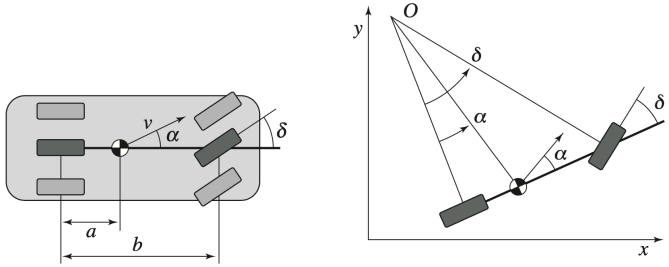
Activity 2

Vehicle steering

Consider the example at

https://python-control.readthedocs.io/en/0.9.3.post2/steering.html

that describes the vehicle steering dynamics shown in the figure below. The figure is taken from the book: *Åström, Karl Johan, and Richard M. Murray. Feedback systems: an introduction for scientists and engineers. 2nd Edition, Princeton university press, 2019.*



(a) Overhead view (b) Bicycle model

Figure 3.17: Vehicle steering dynamics. The left figure shows an overhead view of a vehicle with four wheels. The wheelbase is b and the center of mass at a distance a forward of the rear wheels. By approximating the motion of the front and rear pairs of wheels by a single front wheel and a single rear wheel, we obtain an abstraction called the *bicycle model*, shown on the right. The steering angle is δ and the velocity at the center of mass has the angle α relative the length axis of the vehicle. The position of the vehicle is given by (x, y) and the orientation (heading) by θ.

Let the model be given by a simple bicycle model governed by the following set of equations:

$$\dot{x} = v\cos(\alpha + \theta)$$
$$\dot{y} = v\sin(\alpha + \theta)$$
$$\dot{\theta} = \frac{v}{b}\tan\delta$$

where

$$\alpha = \tan^{-1}\left(\frac{a\tan\delta}{b}\right)$$

(x, y) is the position of the reference point of the vehicle in the plane and θ is the angle of the vehicle with respect to horizontal. The vehicle input is given by (v, δ) where v is the forward velocity of the vehicle and δ is the angle of the steering wheel. We take as parameters the wheelbase b and the offset a between the rear wheels and the reference point. The model includes saturation of the vehicle steering angle (`maxsteer`).

- System state: `x`, `y`, `theta`
- System input: `v`, `delta`
- System output: `x`, `y`
- System parameters: `wheelbase`, `refoffset`, `maxsteer`

Assuming no slipping of the wheels, the motion of the vehicle is given by a rotation around a point O that depends on the steering angle δ.

2.1. Implement the vehicle steering dynamics using the *Python Control Systems Library* and illustrate the dynamics of the system, with an input that correspond to driving down a curvy road.

In [ ]:
```
# Run this once to install the control systems library

!pip install control
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/co
lab-wheels/public/simple/
Collecting control
  Downloading control-0.9.3.post2-py3-none-any.whl (432 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 432.8/432.8 KB 11.4 MB/s eta
0:00:00
Requirement already satisfied: scipy>=1.3 in /usr/local/lib/python3.8/dis
t-packages (from control) (1.7.3)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.8/dis
t-packages (from control) (3.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-pac
kages (from control) (1.21.6)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 i
n /usr/local/lib/python3.8/dist-packages (from matplotlib->control) (3.0.
9)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pyth
hon3.8/dist-packages (from matplotlib->control) (2.8.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/d
ist-packages (from matplotlib->control) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python
3.8/dist-packages (from matplotlib->control) (1.4.4)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-
packages (from python-dateutil>=2.1->matplotlib->control) (1.15.0)
Installing collected packages: control
Successfully installed control-0.9.3.post2
```

In [ ]:

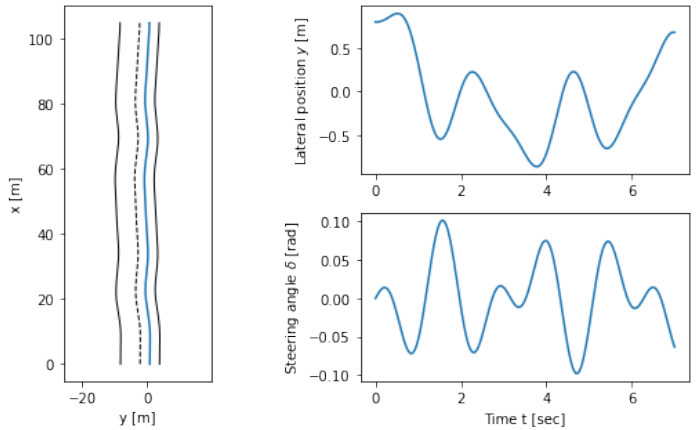
```
import numpy as np
import matplotlib.pyplot as plt
import control as ct
import control.optimal as opt
ct.use_fbs_defaults()
```

In [ ]:

```
# To complete

def vehicle_update(t, x, u, params):
    ''' Implement the system equation
        Include your code here...
    '''
    # Get the parameters for the model
    a = params.get('refoffset', 1.5)        # offset to vehicle reference
    b = params.get('wheelbase', 3.)         # vehicle wheelbase
    maxsteer = params.get('maxsteer', 0.5)  # max steering angle (rad)

    # Saturate the steering input
    delta = np.clip(u[1], -maxsteer, maxsteer)

    #alpha = ...

    # Return the derivative of the state
    return np.array([
        #,    # xdot = cos(theta + alpha) v
        #,    # ydot = sin(theta + alpha) v
        #      # thdot = v/b tan(delta)
    ])


def vehicle_output(t, x, u, params):
    return x[0:2]

# Default vehicle parameters (including nominal velocity)
vehicle_params={'refoffset': 1.5, 'wheelbase': 3, 'velocity': 15,
                'maxsteer': 0.5}

# Define the vehicle steering dynamics as an input/output system
vehicle = ct.NonlinearIOSystem(
    vehicle_update, vehicle_output, states=3, name='vehicle',
    inputs=('v', 'delta'), outputs=('x', 'y'), params=vehicle_params)
```

In [ ]:
```
# Solution

def vehicle_update(t, x, u, params):
    ''' Implement the system equation
        Include your code here...
    '''
    # Get the parameters for the model
    a = params.get('refoffset', 1.5)        # offset to vehicle reference
    b = params.get('wheelbase', 3.)         # vehicle wheelbase
    maxsteer = params.get('maxsteer', 0.5)  # max steering angle (rad)

    # Saturate the steering input
    delta = np.clip(u[1], -maxsteer, maxsteer)

    alpha = np.arctan2(a * np.tan(delta), b)

    # Return the derivative of the state
    return np.array([
        u[0] * np.cos(x[2] + alpha),    # xdot = cos(theta + alpha) v
        u[0] * np.sin(x[2] + alpha),    # ydot = sin(theta + alpha) v
        (u[0] / b) * np.tan(delta)      # thdot = v/l tan(delta)
    ])


def vehicle_output(t, x, u, params):
    return x[0:2]

# Default vehicle parameters (including nominal velocity)
vehicle_params={'refoffset': 1.5, 'wheelbase': 3, 'velocity': 15,
                'maxsteer': 0.5}

# Define the vehicle steering dynamics as an input/output system
vehicle = ct.NonlinearIOSystem(
    vehicle_update, vehicle_output, states=3, name='vehicle',
    inputs=('v', 'delta'), outputs=('x', 'y'), params=vehicle_params)
```

In [ ]:
```
# System parameters
wheelbase = vehicle_params['wheelbase']
v0 = vehicle_params['velocity']

# Control inputs
T_curvy = np.linspace(0, 7, 500)
v_curvy = v0*np.ones(T_curvy.shape)
delta_curvy = 0.1*np.sin(T_curvy)*np.cos(4*T_curvy) + 0.0025*np.sin(T_cur
u_curvy = [v_curvy, delta_curvy]
X0_curvy = [0, 0.8, 0]

# Simulate the system + estimator
t_curvy, y_curvy, x_curvy = ct.input_output_response(
    vehicle, T_curvy, u_curvy, X0_curvy, params=vehicle_params, return_x=

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the resulting trajectory (and some road boundaries)
plt.subplot(1, 4, 2)
plt.plot(y_curvy[1], y_curvy[0])
plt.plot(y_curvy[1] - 9/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1
plt.plot(y_curvy[1] - 3/np.cos(x_curvy[2]), y_curvy[0], 'k--', linewidth=
plt.plot(y_curvy[1] + 3/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1

plt.xlabel('y [m]')
plt.ylabel('x [m]');
plt.axis('Equal')

# Plot the lateral position
plt.subplot(2, 2, 2)
plt.plot(t_curvy, y_curvy[1])
plt.ylabel('Lateral position $y$ [m]')

# Plot the steering angle
plt.subplot(2, 2, 4)
plt.plot(t_curvy, delta_curvy)
plt.ylabel('Steering angle $\\delta$ [rad]')
plt.xlabel('Time t [sec]')
plt.tight_layout()
```



2.2. Do the same, but now using an Euler discretization with a sampling time of 0.01 seconds.

In [ ]:
```python
# To complete

def vehicle_step(x, u, params, t_sample):
  # Used for the implementation of an Euler discretization

  # Get the parameters for the model
  a = params.get('refoffset', 1.5)         # offset to vehicle reference p
  b = params.get('wheelbase', 3.)          # vehicle wheelbase
  maxsteer = params.get('maxsteer', 0.5)  # max steering angle (rad)

  # Saturate the steering input
  delta = np.clip(u[1], -maxsteer, maxsteer)
  #alpha = ...

  # Return the updated state (X_(i+1) as a function of X_i and u_i)
  return np.array([ #...,
                    #...,
                    #...
                    ])
```

In [ ]:
```python
# Solution

def vehicle_step(x, u, params, t_sample):
  # Used for the implementation of an Euler discretization

  # Get the parameters for the model
  a = params.get('refoffset', 1.5)         # offset to vehicle reference p
  b = params.get('wheelbase', 3.)          # vehicle wheelbase
  maxsteer = params.get('maxsteer', 0.5)  # max steering angle (rad)

  # Saturate the steering input
  delta = np.clip(u[1], -maxsteer, maxsteer)
  alpha = np.arctan2(a * np.tan(delta), b)
  #alpha = np.arctan(a*np.tan(delta)/b)

  # Return the updated state (X_(i+1) as a function of X_i and u_i)
  return np.array([ x[0] + u[0]*np.cos(alpha+x[2])*t_sample,
                    x[1] + u[0]*np.sin(alpha+x[2])*t_sample,
                    x[2] + u[0]/b*np.tan(delta)*t_sample
                    ])
```

In [ ]:
```python
sample_time = 0.01 #If you use 0.1, there is already a noticeable differe
time_end = 7

# System parameters
wheelbase = vehicle_params['wheelbase']
v0 = vehicle_params['velocity']

# Control inputs for the previously used function
T_curvy = np.linspace(0, time_end, 500)
v_curvy = v0*np.ones(T_curvy.shape)
delta_curvy = 0.1*np.sin(T_curvy)*np.cos(4*T_curvy) + 0.0025*np.sin(T_cur
u_curvy = [v_curvy, delta_curvy]
X0_curvy = [0, 0.8, 0]

# Simulate the system

t_signal = np.arange(0,time_end,sample_time) # time samples
v_signal = v0*np.ones(t_signal.shape)        # v adjusted to the number o
delta_signal = 0.1*np.sin(t_signal)*np.cos(4*t_signal) + 0.0025*np.sin(t_

x_signal = np.zeros_like(t_signal)
y_signal = np.zeros_like(t_signal)
theta_signal = np.zeros_like(t_signal)

#Initial conditions of our system
x_signal[0] = X0_curvy[0]
y_signal[0] = X0_curvy[1]
theta_signal[0] = X0_curvy[2]

for i in range(0,t_signal.shape[0]-1):
    new_x = vehicle_step((x_signal[i],y_signal[i],theta_signal[i]),(v_sig
    # Update the state
    x_signal[i+1] = new_x[0]
    y_signal[i+1] = new_x[1]
    theta_signal[i+1] = new_x[2]

#plt.axis('equal')
plt.plot(x_signal, y_signal, label='Path (Euler t=' + str(sample_time)+')
```

```
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.plot()
plt.title("Path using an Euler discretization")
plt.legend()
plt.show()

t_curvy, y_curvy, x_curvy = ct.input_output_response(
    vehicle, T_curvy, u_curvy, X0_curvy, params=vehicle_params, return_x=


plt.plot(x_signal, y_signal, label='Path (Euler t=' + str(sample_time)+')
plt.plot(y_curvy[0], y_curvy[1], label='Path (Original Function)')
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.plot()
plt.title("Comparison between both methods")
plt.legend()
plt.show()

# Configure matplotlib plots to be a bit bigger and optimize layout
plt.figure(figsize=[9, 4.5])

# Plot the resulting trajectory (and some road boundaries)
plt.subplot(1, 4, 2)
plt.plot(y_curvy[1], y_curvy[0])
plt.plot(y_curvy[1] - 9/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1
plt.plot(y_curvy[1] - 3/np.cos(x_curvy[2]), y_curvy[0], 'k--', linewidth=
plt.plot(y_curvy[1] + 3/np.cos(x_curvy[2]), y_curvy[0], 'k-', linewidth=1

plt.xlabel('y [m]')
plt.ylabel('x [m]')
plt.axis('Equal')

# Plot the lateral position
plt.subplot(2, 2, 2)
plt.plot(t_curvy, y_curvy[1])
plt.ylabel('Lateral position $y$ [m]')

# Plot the steering angle
plt.subplot(2, 2, 4)
plt.plot(t_curvy, delta_curvy)
plt.ylabel('Steering angle $\\delta$ [rad]')
plt.xlabel('Time t [sec]')
plt.tight_layout()
```