Lab 1: Introduction¶

Background¶

This course includes a set of computer laboratories that explore topics related to signals and systems. These will be made available to you as the course progresses, and you are expected to engage with the content and produce and submit a set of results related to some defined tasks. Optional tasks will be included keep it interesting for students that are proficient with programming or are more comfortable with mathematics. Assistance will be provided during tutorial sessions if required, but it is really expected that you'll work in your own time and at your own pace: it's important right from the start for you to take responsibility for your own learning. You shouldn't be working in groups, but helping one another out on specific issues is fine.

The computer labs are based on the Python programming language, and require a kernel that includes all the standard numerical libraries. They have been created as worksheets using the *Jupyter notebook* system, which works in a browser and allows you to interleave text and code snippets that can be run interactively. However, there should be no problem with you implementing the required tasks in any other Python IDE if you prefer.

Jupyter notebook has been installed in the EBE green and red computer labs, via a package called *Anaconda*. If you want to install it on your own machine you can download it from https://www.continuum.io/downloads. You should probably choose the distribution based on the Python 3.6 kernel, since that's the one installed in the labs. The download is a few hundred megabytes, so if you rather want us to provide it locally on the campus then just make a request to the teaching assistant.

Getting started¶

The following procedure works in the EBE computer labs. There are two tasks to perform: fetching the notebook files, and running the notebook server. The initial setup takes a little work, but once done subsequent usage is simpler.

Instructions for how to run Jupyter on your own PC are similar, and there's lots of help on the web. There's a comprehesnive guide at https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/index.html.

Starting the Jupyter server¶

From the Windows start menu select "Run". In the box that pops up get a command prompt by typing `cmd` and pressing enter. Change to your home directory, which in my case is drive "F": to do this type `f:` and press enter. You could also use the drive letter of any other drive. You can list the files by typing `dir`.

The Jupyter notebook server serves files from the directory it's started in. Make a new folder, called for example "notebooks", using the command `mkdir notebooks`. Change into this folder using `cd notebooks`. Finally run `jupyter notebook`. A web browser should open up, showing a file view of the base directory.
You must keep the server running in the command prompt while using the notebooks. When complete you can shut it down by typing "control-c".

Accessing the notebooks¶

The notebooks are being actively developed and are hosted on github at address https://github.com/eee2047s-uct/notebooks.git. You can access them by downloading most recent versions of relevant files and putting them into the `notebooks` directory you made above. However, it will be difficult to keep track of updates using this approach.
We'll rather use git for version control. If you're on your own machine and don't have it then you'll need to install it. Instead of making the `notebooks` directory above we'll let git make it and keep track of it. In the command prompt use

```
git clone https://github.com/eee2047s-uct/notebooks.git
```

to do this. If successful, typing `dir` should show you a new "notebooks" directory. If a login prompt pops up then you typed the web address incorrectly.
Start a Jupyter notebook server in this new directory. You should see and be able to open the notebook file for this document `lab_intro.ipynb`.

Subsequent usage¶

Don't modify the files distributed in the git repository. Copy a file to a new filename before modifying it for your own purposes. That way you'll be able to pull updates from the repository without having to deal with conflicts. Your renamed files can still be in the same directory as the original without causing any problems.
If you've already created a local labs repository then you don't need to do it again. However, before working on a new task you should pull any updates that might have been made. In a command prompt change into "notebooks" directory. Then execute `git pull` to update local copies of labs.
If `git pull` gives an error then you've probably edited one of the distribution files. Move or rename the problem files and run again. If you want to just overwrite the existing ones, losing all local changes, use `git reset --hard` before repeating the `git pull` instruction.
Once updated, run `jupyter notebook` from the "notebooks" directory in the command prompt.

Basic Jupyter usage¶

A notebook is basically composed of cells. A cell can contain text ("markdown") or code, in this case Python. For any cell you can select the content type using the pulldown on the menubar. A cell can be executed by pressing shift-enter while it is in focus.
Below is a cell that creates two numpy arrays. The values in `xv` are initialised to be a particular function of `tv`. Select it and press shift-enter to run the code in the Python kernel attached to the

notebook. Change the number of points in `tv` and press shift-enter again to run, and note the change in output.

In [3]:

```
import numpy as np


tv = np.linspace(0, 10, 10);

xv = 0.1*tv**2 - np.cos(tv);

print(xv);
```

```
[-1.         -0.32020923  1.10014808  2.09278512  2.24005852  2.3396668
  3.51707674  5.97325259  8.76104957 10.83907153]
```

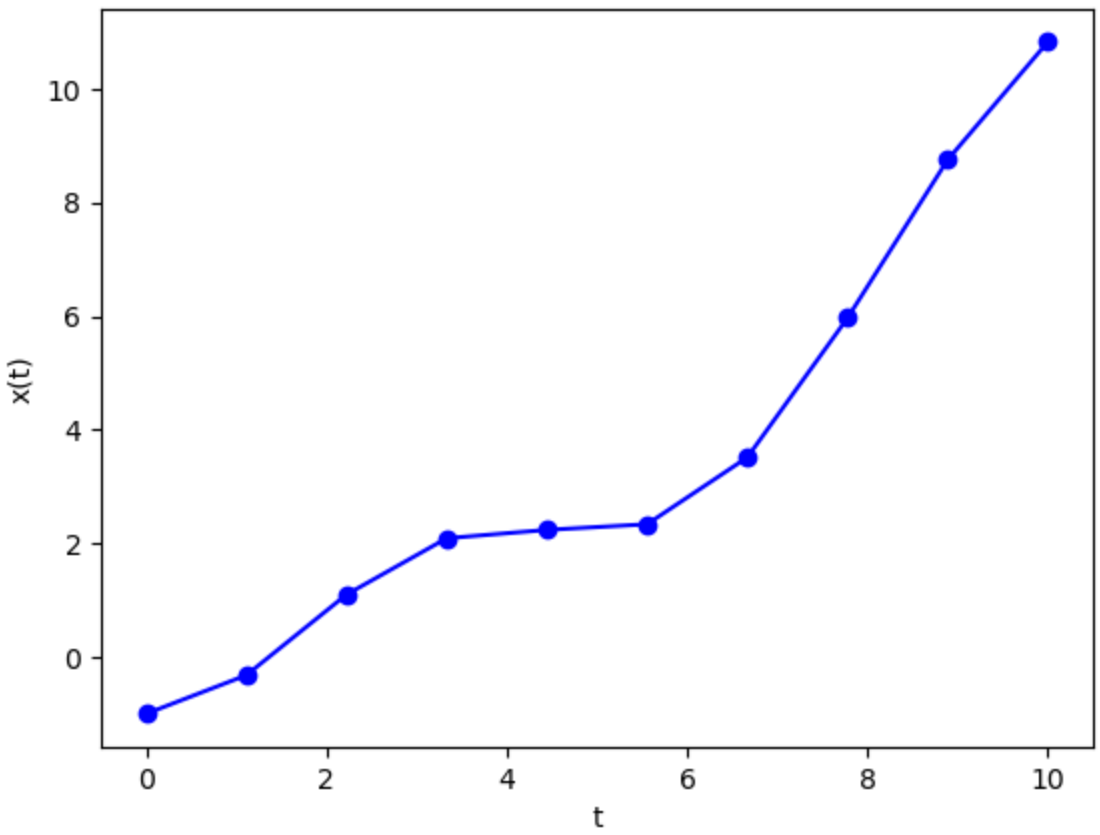
The Pythonn kernel is fully-featured for numeric computation, and includes most of the packages useful for scientific computing. The following code plots the vectors defined above using "matplotlib". The `%matplotlib notebook` line makes any plots appear in the notebook itself rather than in a separate window.

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib notebook


plt.plot(tv, xv, 'bo')

plt.xlabel('t');  plt.ylabel('x(t)');
```



When you execute the code above it runs it in the Python kernel for the notebook, which already has the variables `tv` and `xv` defined. This can cause problems: if you jump around the notebook running cells in some arbitrary order then the kernel will probably end up in a weird state. If this happens then you can use the menu item "Cell" -> "Run All" to reexecute all cells in the notebook in order. If things really go wrong then restart the kernel.

Executing a text or markdown cell has a different effect: the content is converted and rendered as rich text. The markdown language is documented all over the internet, and supports both HTML and LaTeX equations. Do a web search if you want details or introductions to these topics.

Gaining familiarity¶

One of the nice things about Python is that there's an active community of technical people that provide useful information and resources. If you don't know how to do something, then a simple web search can quickly help.

This section points you towards some resources that might help in getting you familiar with basic mathematical functionality. Take a look at what's available, or find any other resources you prefer. Youtube videos are also an option.

There's a basic numerical Python tutorial at http://cs231n.github.io/python-numpy-tutorial, and a Python notebook version at https://github.com/kuleshov/cs228-material/blob/master/tutorials/python/cs228-python-tutorial.ipynb. A copy of this notebook that has been updated for Python 3 is included in the "examples" directory of your "notebooks" folder. Load this notebook and work through the cells. Make changes to the contents, and press "shift-enter" to run and see the corresponding outputs. Pay particular attention to how to use numpy arrays, and how to plot using matplotlib.
Possibly the best place to find resources is the Scipy homege: https://www.scipy.org/getting-started.html.
You can also access function help in Jupyter notebook. If you remove the comment modifier "#" in the cell below and run it, for example, a help window for the matplotlib.pyplot.plot() function should pop up.

In [ ]:

```
import matplotlib.pyplot as plt
#?plt.plot()
```

If you don't want to use the notebooks for development then you can just use any other Python IDE. These worksheets usually have a `%run src/labX_preamble.py` command that enables saving functionality for code blocks. Any code cell that starts with the `%%writefileexec` directive saves the corresponding contents into the `src` directory in the `notebooks` folder, and you can pick up the raw python code from there.

Tasks¶

These tasks involve writing code, or modifying existing code, to meet the objectives described.

1. Create a plot of $x^2 \sin(\frac{1}{x^2}) + x$ on the interval $[−1,1]$ using 250 points. Remember to label the axes.

2. We often need to plot complex-valued signals, where for each value of t there is a real and an imaginary value, or a magnitude and a phase. We therefore need two sets of axes. Use the `subplot` functionality of `matplotlib.pyplot` to plot the real and imaginary parts of the signal $x(t) = e^{j \omega_0 t}$ in a single figure, but on two separate axes. Use a value of $\omega_0 = 2$ and make the plot range over $t=0$ to $t=10$.

3. Most of signal processing involves applying a recursive relationship to some given data. This course concentrates on continuous-time signals, but equivalent processes can be applied in the digital case. Consider the recursive equation $x[n] = -0.98 x[n-1]$. Create a numpy array with 100 elements for each of the values $x[0]$ to $x[99]$, and write code to populate it assuming the initial condition $x[0] = 10$. Use `stem` to make a plot of $x[n]$ versus n over the range calculated.

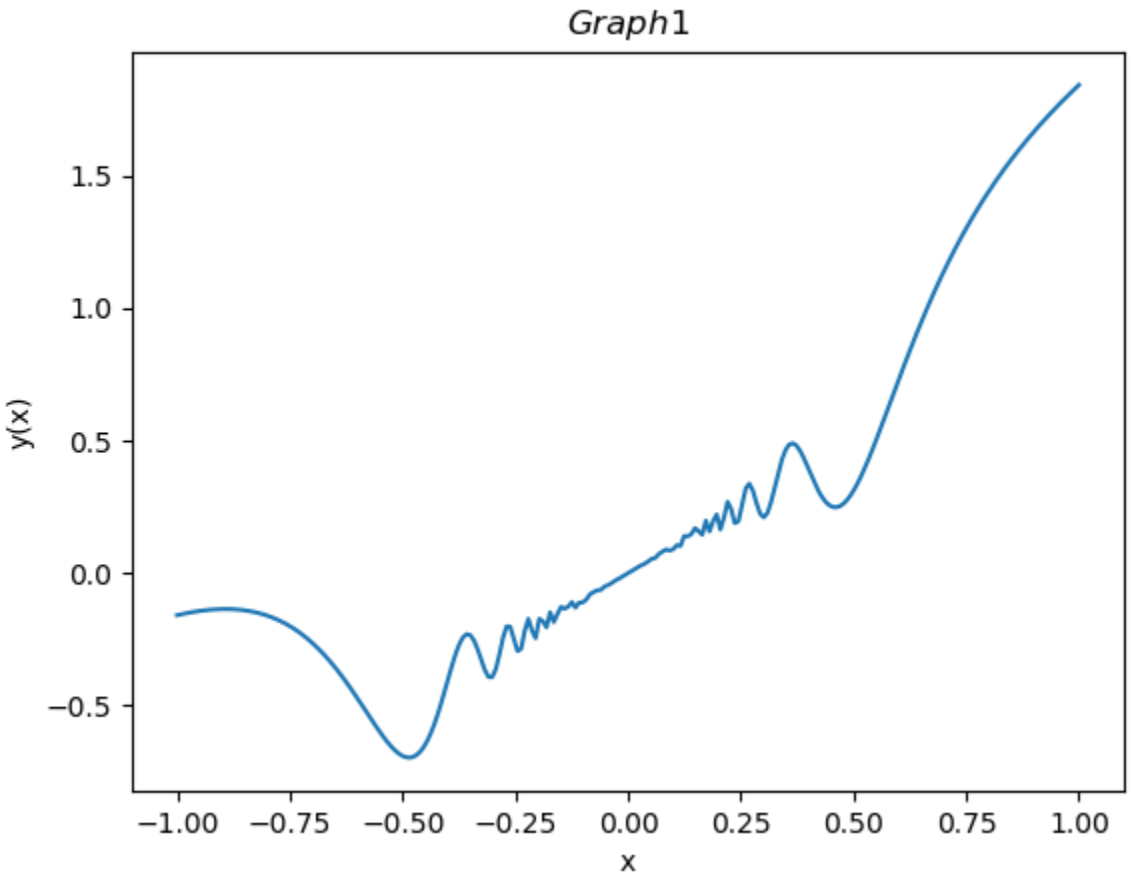
4. Create a surface plot of $\sin(x) \sin(y)$ on the domain $[-\pi,\pi] \times [-\pi,\pi]$.

In [10]:

```
# Question 1
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

x = np.linspace(-1, 1, 250);
y = ((x**2)*(np.sin(1/x**2)))+x;

plt.plot(x, y)
plt.xlabel('x');  plt.ylabel('y(x)');
plt.title('$Graph 1$')
```



Out[10]:

```
Text(0.5,1,'$Graph 1$')
```

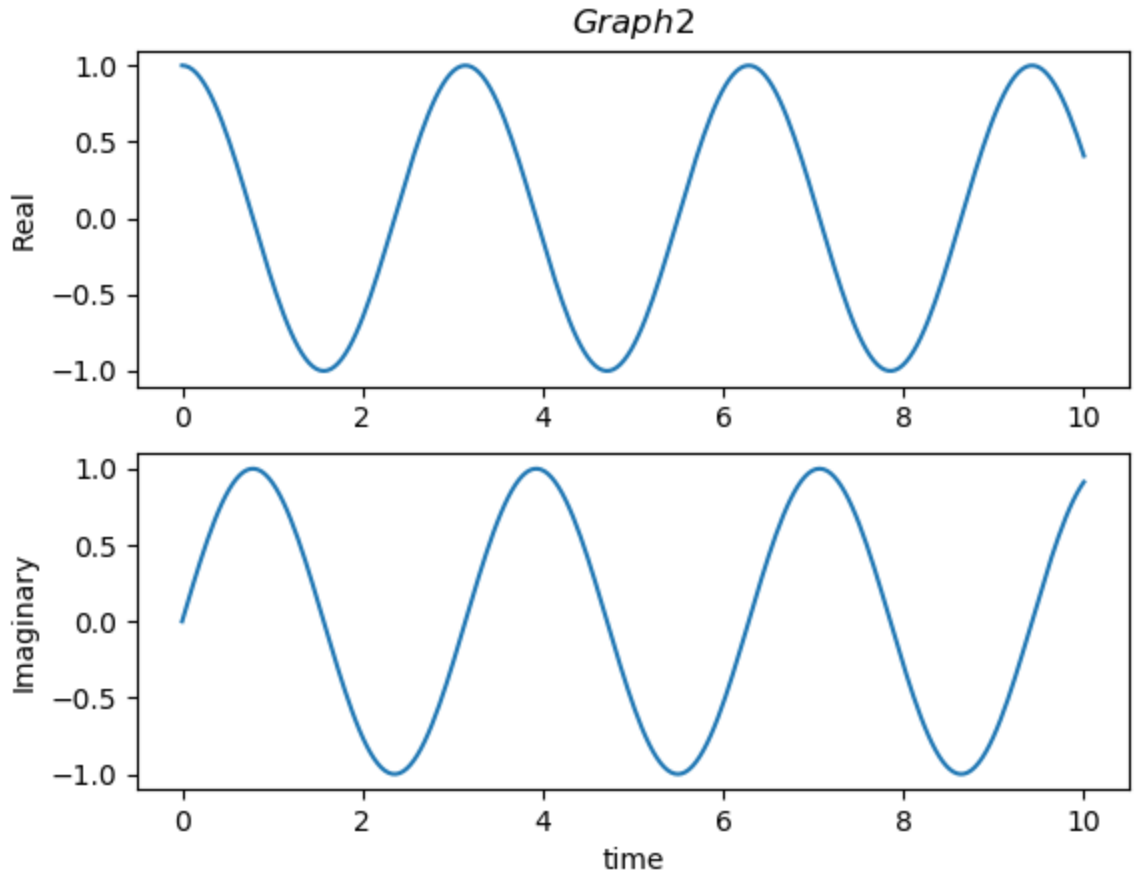
In [27]:

```
# Question 2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

x = np.linspace(0, 10, 250);
y = np.cos(2*x) + 1j*(np.sin(2*x));

plt.subplot(2, 1, 1)
plt.plot(x, y.real)
plt.ylabel('Real');
plt.title('$Graph 2$')

plt.subplot(2, 1, 2)
plt.plot(x, y.imag)
plt.xlabel('time');  plt.ylabel('Imaginary');
```



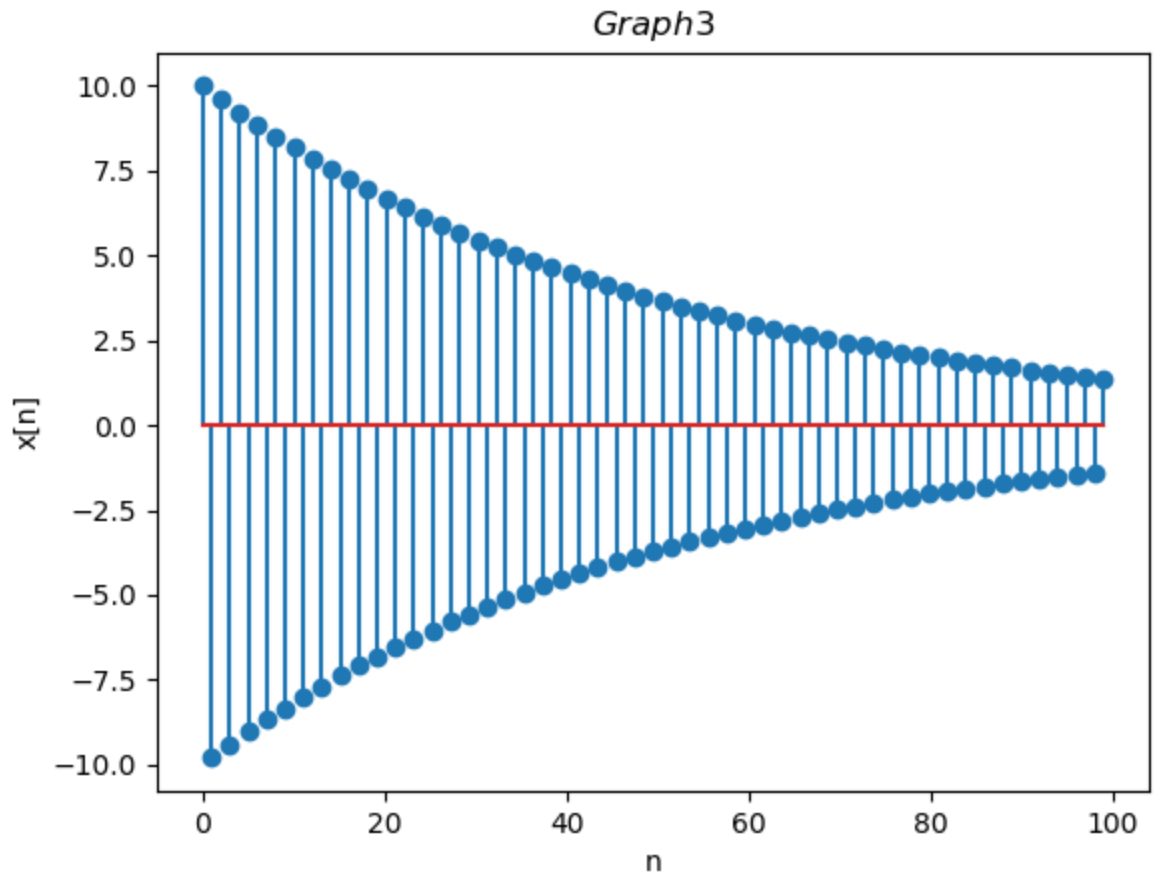
In [23]:

```
# Question 3
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

t = np.linspace(0, 99, 99)
x = np.empty(99);
x[0] = 10

for n in range(1,99):
    x[n]= -1*0.98*x[n-1]

plt.stem(t,x)
plt.xlabel('n');  plt.ylabel('x[n]');
plt.title('$Graph 3$')
```



Out[23]:

```
Text(0.5,1,'$Graph 3$')
```

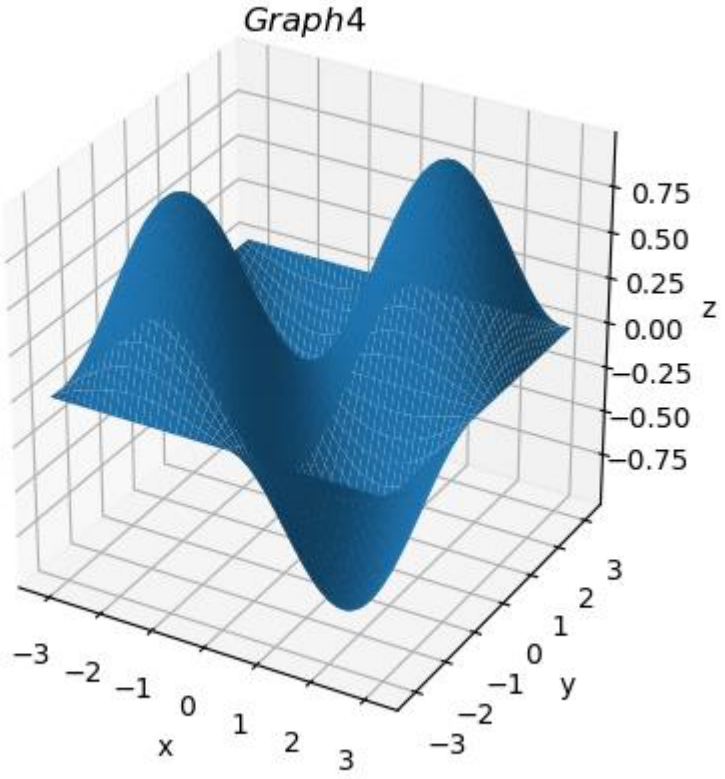
In [24]:

```
# Question 4
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(5,5))
ax = fig.gca(projection='3d')
x = np.linspace(-1*np.pi, np.pi, 1000);
y = np.linspace(-1*np.pi, np.pi, 1000);
x,y = np.meshgrid(x,y)
z = (np.sin(x)*np.sin(y));

ax.plot_surface(x, y, z)
```

```
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z');
plt.title('$Graph 4$')
```



Out[24]:

```
Text(0.5,0.92,'$Graph 4$')
```