Introduction to IPython, NumPy and SciPy for COMP28512

IPython, NumPy and SciPy are useful way for you to perform the exercises in the labs. In particular IPython Notebooks allow you to perform labs and write reports simulataneously.

IPython Notebooks

Preparing the environment

1. On university computers the IPython binary is often located in `/opt/anaconda/bin`. **To use IPython this directory will need to be on your path**, we recommend modifying your `bashrc` to make sure this is always the case.
2. To listen to audio from IPython you need to download the file `comp28512_utils.py` from Moodle or from https://gist.github.com/mundya/006e6f365ea1abbe6921 (https://gist.github.com/mundya/006e6f365ea1abbe6921) and place it in the directory you are working in.

Launching IPython in Notebook mode

First go to the directory in which you are going to work, then execute `ipython notebook`. This should fire up your browser with a directory listing: click "New Notebook" to create a new notebook to work in. Change the filename by clicking on the text at the top of the page that says "Untitled0" or similar.

Notebooks contain "cells", these may contain Python code to execute or Markdown (http://daringfireball.net/projects/markdown/basics) formatted text.

- To finish a cell and progress to the next, type `Shift` + `Enter`.
- To make a cell a Markdown formatted, type `Ctrl+M` followed by `M`

Entering code in a cell allows it to be executed, variable definitions are retained between cells.

In [1]:
```
greeting = "Hello, Manchester"
```

In [2]:
```
print(greeting)
```

```
Hello, Manchester
```

Magic Commands

"Magic" commands begin with a `%` and are used to control the IPython environment. For example, `%reset` is used to delete all the variables we have defined.

In [3]:
```
%reset
```

In [4]:
```
try:
    print(greeting)
except Exception as e:
    print e
```

```
name 'greeting' is not defined
```


We will also use the `%matplotlib` magic to display graphs in the notebook.

Displaying equations

Markdown cells can display equations written using LaTeX commands. For example writing `$A\sin(2\pi f t + \phi)$` in a Markdown cell results in $A\sin(2\pi f t + \phi)$.

System commands

System commands and executables can be run in code cells by prefixing them with `!`.

In [5]:
```
!ls -ll *.ipynb
```

```
-rw------- 1 andrew users 1252677 Jan 24 16:23 lab1-intro.ipynb
-rw------- 1 andrew users   18295 Jan 24 17:32 lab1.ipynb
-rw------- 1 andrew users 3532874 Jan 29 20:54 lab-intro.ipynb
```

NumPy

Regardless of whether you are using IPython Notebooks we will be using NumPy (https://numpy.org) to work with arrays of samples. NumPy is a library which provides memory-efficient arrays and bindings to fast linear algebra implementations (BLAS).

In [6]:
```
import numpy as np  # This saves us a bit of typing
```

NumPy Arrays

We will be using NumPy arrays extensively in the labs.

To create a row vector:

In [7]:
```
a = np.array([1.0, 0.5, 0.333])
print(a)
```

```
[ 1.     0.5    0.333]
```

Or a column vector:

In [8]:
```
print np.array([[1.0], [0.5], [0.333]])
```

```
[[ 1.   ]
 [ 0.5  ]
 [ 0.333]]
```

Scalar multiplication works as we'd expect:

In [9]:
```
2 * a
```

Out[9]:
```
array([ 2.   ,  1.   ,  0.666])
```

As does division:

In [10]:
```
a / 3.0
```


Out[10]:
```
array([ 0.33333333,  0.16666667,  0.111     ])
```

Scalar addition and subtraction apply to the whole array:

In [11]:
```
print "     a =", a
print "a + 1 =", a + 1
print "a - 2 =", a - 2
```

```
    a = [ 1.     0.5    0.333]
a + 1 = [ 2.     1.5    1.333]
a - 2 = [-1.    -1.5   -1.667]
```

And array addition works like vector addition:

In [12]:
```
c = np.array([1, 2, 3])
d = np.array([3, 2, 1])

print c, "+", d, "=", c + d
```

```
[1 2 3] + [3 2 1] = [4 4 4]
```

We can get the size (number of elements of the array) by using `len` or `size`:

In [13]:
```
print len(a), "or", a.size
```

```
3 or 3
```

The "shape" of an array is the number of rows, columns, etc. (as arrays can be any dimension). For vectors this should be trivial:

In [14]:
```
print a.shape
```

```
(3,)
```

In [15]:
```
b = np.array([[1.0]] * 3)  # 3-dimensional column vector
print b
print "b contains", b.size, "elements -- shaped", b.shape, "meaning 3 rows and 1 column"
```

```
[[ 1.]
 [ 1.]
 [ 1.]]
b contains 3 elements -- shaped (3, 1) meaning 3 rows and 1 column
```

Useful NumPy functions and constants

We'll often need to use:

- `np.sin` - Sine
- `np.cos` - Cosine
- `np.round` - Rounds to the nearest integer
- `np.pi` - π
- `np.arange(a, b)` - The integers in the range $[a, b)$.

For example, we can calculate the value of $\sin\theta$ for $\theta = \{0, \frac{1}{2}\,\pi, \pi, \frac{3}{2}\,\pi\}$:

In [16]:
```
theta = np.arange(0, 4) * np.pi / 2.0
print "     theta =", theta
print "sin(theta) =", np.sin(theta), "~= [0 1 0 -1]"
```

```
     theta = [ 0.          1.57079633  3.14159265  4.71238898]
sin(theta) = [  0.00000000e+00   1.00000000e+00   1.22464680e-16  -1
.00000000e+00] ~= [0 1 0 -1]
```

Plotting

To plot results we'll use pyplot, a part of matplotlib (http://matplotlib.org).

In IPython notebooks we need to run the magic command `%matplotlib inline` to get plots to show in the notebook.

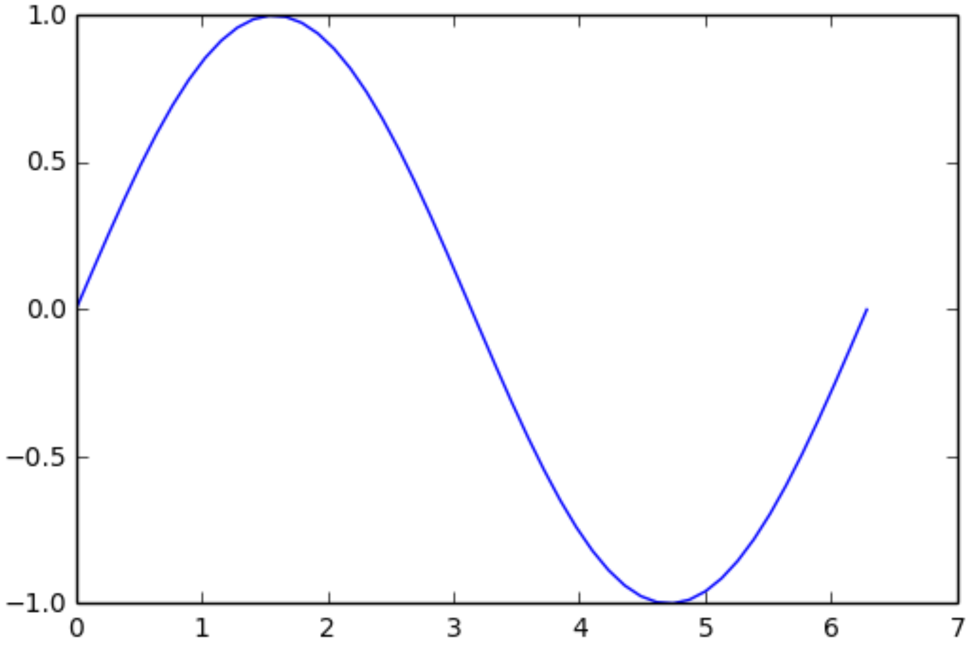
In [17]:
```
from matplotlib import pyplot as plt
%matplotlib inline

# The following lines make for better looking graphs but are not nec
essary
import matplotlib as mpl
mpl.rc("savefig", dpi=100)
```

Plotting is just a case of calling `plt.plot`, though often we will want to be a bit more advanced and will need to specify more details.

For example, plotting $\sin\theta$ for some samples of θ.

In [18]:
```
theta = np.linspace(0, 2*np.pi)
sintheta = np.sin(theta)

plt.plot(theta, sintheta)
```

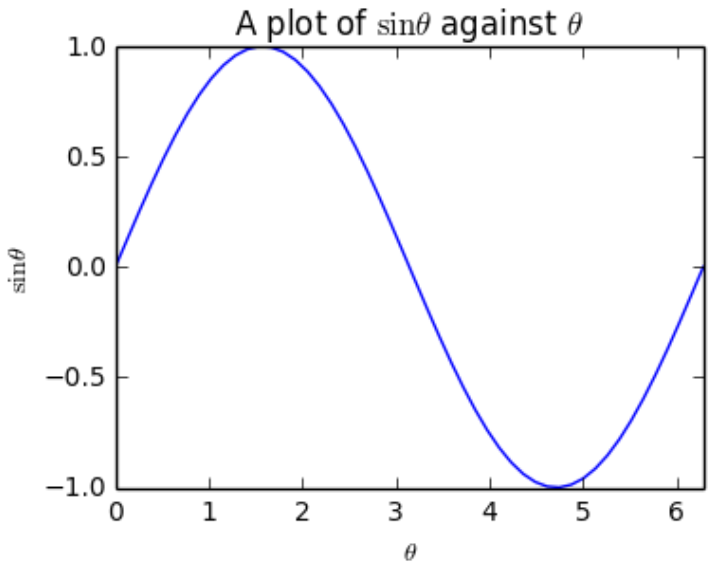
Out[18]: `[<matplotlib.lines.Line2D at 0x7f58ca863a90>]`



Obviously this graph is missing axis labels and a title. To add these we create a figure using an alternative method:

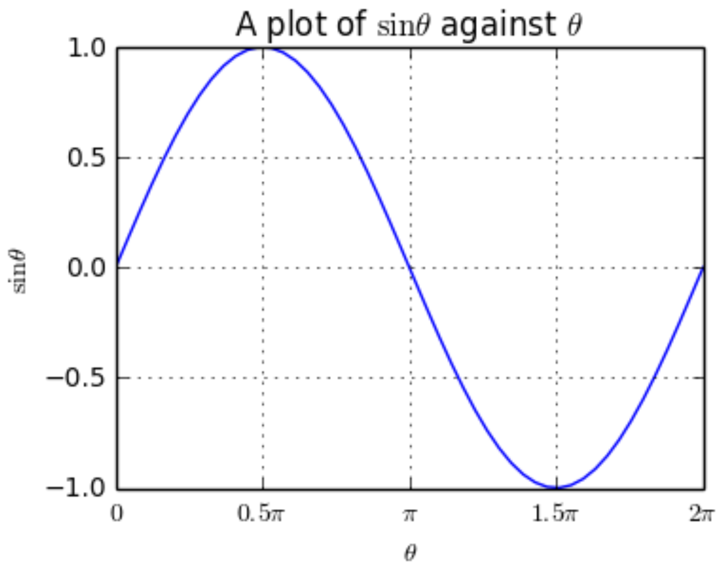
In [19]:
```
fig, ax = plt.subplots(1, figsize=(4, 3))  # We only want 1 subplot

# Plot the graph as before
ax.plot(theta, sintheta)

# Now we add axis labels and a title
ax.set_xlabel(r"$\theta$")  # Again we can use LaTeX formatting
ax.set_ylabel(r"$\sin\theta$")
ax.set_title(r"A plot of $\sin\theta$ against $\theta$")

# And set the limit of the x-axis
ax.set_xlim(0, 2*np.pi)
```

Out[19]:
```
(0, 6.283185307179586)
```



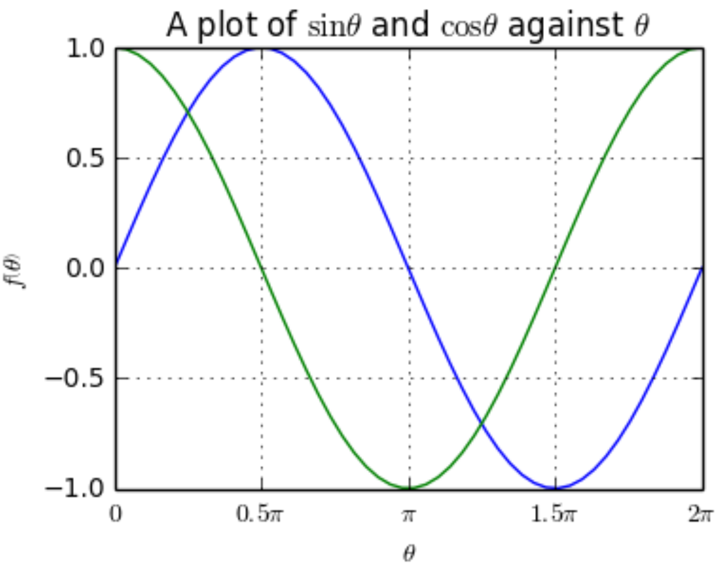
The x-axis ticks aren't in particularly meaningful places, we can also modify those.

In [20]:
```
fig, ax = plt.subplots(1, figsize=(4, 3))  # We only want 1 subplot

# Plot the graph as before
ax.plot(theta, sintheta)

# Now we add axis labels and a title
ax.set_xlabel(r"$\theta$")  # Again we can use LaTeX formatting
ax.set_ylabel(r"$\sin\theta$")
ax.set_title(r"A plot of $\sin\theta$ against $\theta$")

# And set the limit of the x-axis
ax.set_xlim(0, 2*np.pi)

# Set the position of the x-ticks
ax.set_xticks(np.arange(0, 5) * np.pi / 2.0)

# Set the x-tick labels
ax.set_xticklabels([r'$0$', r'$0.5\pi$', r'$\pi$', r'$1.5\pi$', r'$2
\pi$'])

# And turn the grid on
ax.grid(True)
```



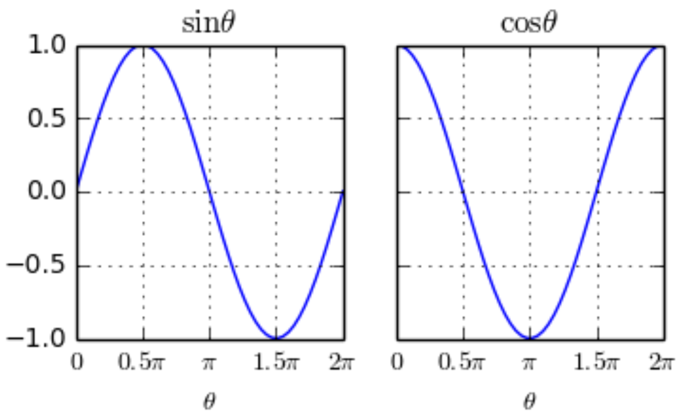
We can plot multiple things on the same axes:

In [21]:
```
fig, ax = plt.subplots(1, figsize=(4, 3))  # We only want 1 subplot

# Plot the graph as before, this time adding cos of theta
ax.plot(theta, sintheta)
ax.plot(theta, np.cos(theta))

# Now we add axis labels and a title
ax.set_xlabel(r"$\theta$")
ax.set_ylabel(r"$f\left(\theta\right)$")
ax.set_title(r"A plot of $\sin\theta$ and $\cos\theta$ against $\the
ta$")

# And set the limit of the x-axis
ax.set_xlim(0, 2*np.pi)

# Set the position of the x-ticks
ax.set_xticks(np.arange(0, 5) * np.pi / 2.0)

# Set the x-tick labels
ax.set_xticklabels([r'$0$', r'$0.5\pi$', r'$\pi$', r'$1.5\pi$', r'$2
\pi$'])

# And turn the grid on
ax.grid()
```



And we can have multiple plots in the same figure:

In [22]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(4, 2), sharey=True)  #
 We want axes on 1 row in 2 columns

ax1.plot(theta, sintheta)
ax2.plot(theta, np.cos(theta))

ax1.set_xlabel(r"$\theta$")
ax1.set_title(r"$\sin\theta$")
ax1.set_ylim(-1, 1)

ax2.set_xlabel(r"$\theta$")
ax2.set_title(r"$\cos\theta$")

for ax in (ax1, ax2):
    ax.set_xlim(0, 2*np.pi)
    ax.set_xticks(np.arange(0, 5) * np.pi / 2.0)
    ax.set_xticklabels([r'$0$', r'$0.5\pi$', r'$\pi$', r'$1.5\pi$',
r'$2\pi$'])
    ax.grid(True)
```



Generating samples of signals

To convert a *continuous-time signal* like $\sin(2\pi f t)$ to a *discrete-time signal* we replace t with nt_{sample} where n is the index of the sample and t_{sample} is the *sampling period*.

If F_s is the sampling frequency, then $t_{\text{sample}} = \frac{1}{F_s}$. The discrete time-sample is then: $\sin\left[2\pi f n t_{\text{sample}}\right]$

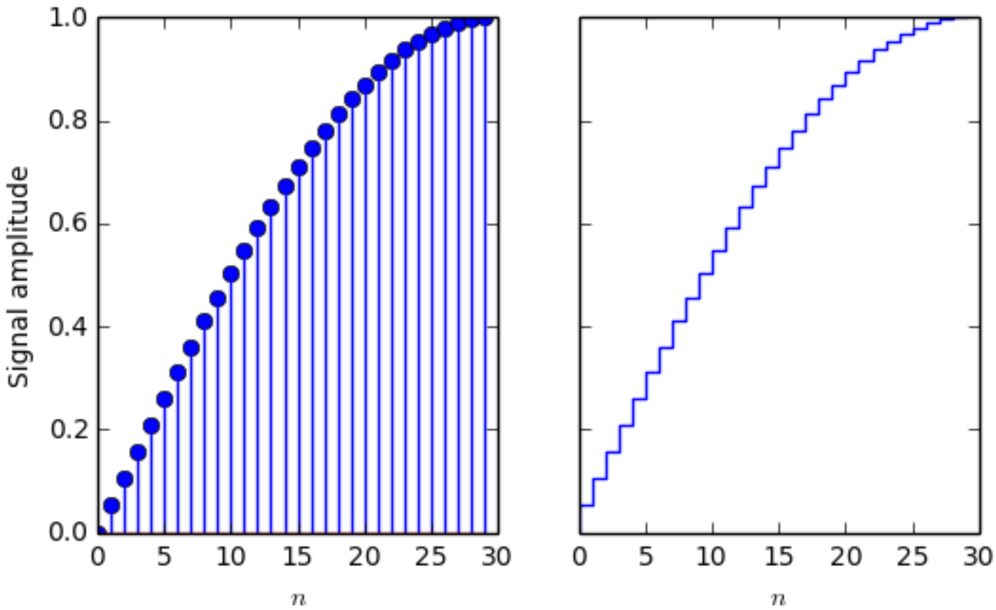
Using NumPy we generate this as:

In [23]:
```
fs = 44.1*10**3  # Sampling frequency (44.1kHz)
ts = 1.0 / fs  # Sampling period
duration = 3.0  # 3 seconds of sample

f = 369.994  # F# in the stave

# Generate the sample indices
n = np.arange(0, duration * fs)

# Generate the sample
sample = np.sin(2*np.pi*f*n*ts)  # sin[2 pi f n t_sample]
```

We can plot this as if it were continuous, or as a stem or stair plot. We'll plot the first 30 samples.

In [24]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(6, 3.5))

n_samples = 30
ax1.stem(n[0:n_samples], sample[0:n_samples])
ax1.set_xlabel("$n$")
ax1.set_ylabel("Signal amplitude")

ax2.step(n[0:n_samples], sample[0:n_samples])
ax2.set_xlabel("$n$")
```

Out[24]: <matplotlib.text.Text at 0x7f58ca79b350>



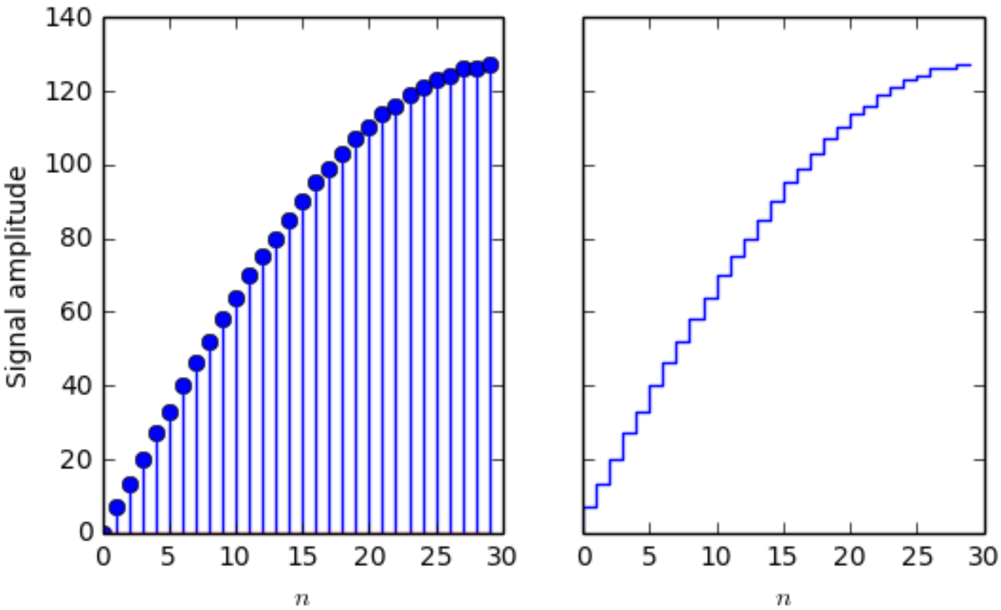
Quantising signals

We can quantise signals by scaling and then rounding, e.g.:

In [25]:
```
sample_8bit = np.int8(np.round(sample * (2**7 - 1)))  # 2**7 - 1 bec
ause 1 bit is used for the sign
```

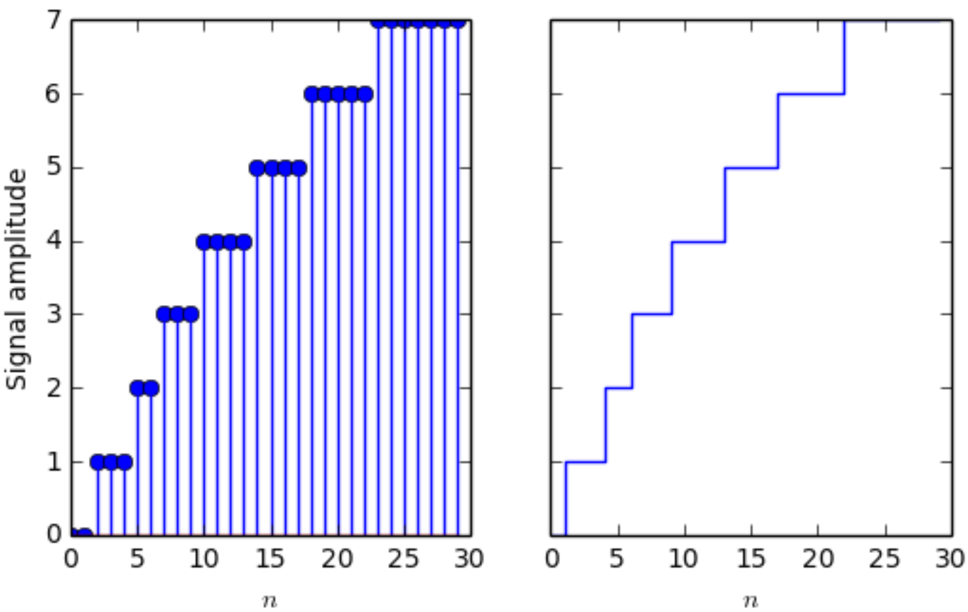
In [26]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(6, 3.5))

n_samples = 30
ax1.stem(n[0:n_samples], sample_8bit[0:n_samples])
ax1.set_xlabel("$n$")
ax1.set_ylabel("Signal amplitude")

ax2.step(n[0:n_samples], sample_8bit[0:n_samples])
ax2.set_xlabel("$n$")
```

Out[26]: <matplotlib.text.Text at 0x7f58ca355350>



In [27]:
```
sample_4bit = np.int8(np.round(sample * (2**3 - 1)))

fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(6, 3.5))

n_samples = 30
ax1.stem(n[0:n_samples], sample_4bit[0:n_samples])
ax1.set_xlabel("$n$")
ax1.set_ylabel("Signal amplitude")

ax2.step(n[0:n_samples], sample_4bit[0:n_samples])
ax2.set_xlabel("$n$")
```

Out[27]: <matplotlib.text.Text at 0x7f58ca159950>



Listening to samples (IPython notebooks only, Firefox and Chrome)

IPython notebooks can embed audio samples directly:

In [28]:
```
from comp28512_utils import Audio
```

In [29]:
```
Audio(sample_8bit, rate=fs, filename="sample_8bit2.wav")  # Provide
the samples and the rate
```

```
Data written to sample_8bit2.wav.
```

Out[29]:

Saving samples to file

SciPy provides methods for saving audio samples to wavefiles:

In [30]:
```
from scipy.io import wavfile
```

In [31]:
```
wavfile.write("sample_8bit.wav", rate=fs, data=sample_8bit)
```

In [32]:
```
!ls *.wav
```

```
7f3784cd8670.wav  7f3787b19620.wav  7f378820c0d0.wav  7f73b340bf80.w
av      7f73b67d0bc0.wav  7fa84e7ac5d0.wav  7fa876a80f30.wav  HQ-mus
ic-mono.wav  HQ-speech-mono.wav  mono_HQ_music_A.wav  sample_8bit2.w
av  sample_8bit.wav
```

We can also read files:

In [33]:
```
(speech_rate, speech) = wavfile.read("HQ-speech-mono.wav")
```

In [34]:
```
print "Speech sampled at {:.2f}kHz".format(speech_rate * 10**-3)
Audio(speech, rate=speech_rate)
```

```
Speech sampled at 44.10kHz
Data written to 7f58ca824cb0.wav.
```

Out[34]:

Preparing Notebooks for submission

We will be accepting IPython notebooks as submissions for your reports. To prepare a notebook for submission please:

- **ensure your name is placed under the title!**
- ensure the file has a sensible name (set this by clicking on the text to the left of the IPython Notebook logo)
- clear all cell output by clicking "Cell -> All Output -> Clear"

You may then submit the notebook.