University of Manchester
School of Computer Science
COMP28512: Mobile Systems
Semester 2: 2018-19
Laboratory Task 1
Introduction & 'Sound Sampling'

TAKE CARE: Your hearing is precious.

When using headphones, ensure that the volume is not too loud. When performing listening experiments, you may not know in advance how loud or distorted the sound will be. A programming error can produce results you do not anticipate. Therefore play the sound first before putting on the headphones. High volumes of distorted sound can be unpleasant and even damage your hearing.

1. Introduction

The aim of this course is to introduce some of the challenges of mobile system design including application development, sound and image coding, bandwidth preservation, error correction and power efficiency. This first laboratory task concentrates on speech and music signals as captured and represented in mobile systems.

For many years in the early development of digital mobile telephony and mobile music storage devices (MP3 players), the emphasis of much research was to find ways of digitally representing speech and music at the lowest possible bit-rate without incurring unacceptable loss of quality. The term 'bit-rate compression' was used. The need for bit-rate compression in mobile telephony was to make the most efficient use of the expensive and scarce radio spectrum available for transmissions to and from the users. The need for bit-rate compression in mobile music and speech storage devices was to allow users to make best possible use of the digital memory they had purchased for storing their recordings. Early MP3 players had only a few hundred megabytes of memory which was less than a single compact disc (approx 800 megabytes) could hold. Encoding good quality stereo music at 128000 bits per second (128 kb/s), using MP3 compression, allows about nine times as much music to be stored in a given amount of memory than would be possible using the bit-rate normally used for uncompressed compact disk (CD) recordings. Uncompressed CD recordings traditionally use a sampling frequency of 44.1 kHz with 16 bits per channel (stereo). The overall bit-rate for traditional CD recordings is therefore 176400 bytes per second which is 1411.2 kb/s (1411200 bits per second). Uncompressed CD quality recordings of speech at 1411.2 kb/s have a quality and naturalness that are immediately apparent and much better than normal telephone speech. MP3 recordings of speech can be excellent also. But the bit-rates of 1411.2 kb/s for uncompressed CD quality and 128 kb/s for MP3 are both too high for the radio transmissions of speech in mobile telephony. Note that b/s (small b) means bits per second and B/s (large) B is Bytes per second.

Traditional old fashioned wired telephone channels had long been digitised before mobile phones arrived, and they use a 64 kb/s standard, often referred to as 'ITU-G711', 'log-PCM', 'A-law' PCM or 'mu-law' PCM. This standard is based on the assumption that most of the information and intelligibility of speech is conveyed in the frequency-range 50 Hz to 4000 Hz.

The sound between 4 kHz and 20 kHz is therefore filtered out. Doing this to music would result in a dull and muffled sort of sound that would not sound natural and may not be enjoyable to listen to. However, doing this to speech does not impair its intelligibility too seriously, though there is some loss of naturalness. PCM stands for 'pulse code modulation' which is not really an appropriate name, but we have got used to it. Strictly, it just means transmitting signals as pulses, i.e. digital transmission, instead of old fashioned analogue transmission.

Restricting the bandwidth of speech to the frequency-range 50 Hz to 3400 Hz makes it 'narrow-band' speech. The sampling frequency can now be reduced from 44100 Hz to 8000 Hz, and this is clearly a good way of reducing the bit-rate. Further bit-rate reduction may be achieved by reducing the number of bits per sample from 16 to 8, though this requires the use of some processing referred to as logarithmic, A-law or mu-law compression to make the quality of the sound acceptable. Since telephone channels are monophonic rather than stereo, this leads to the 64 kb/s standard that has become universal in wired telephony. The loss of naturalness has long been tolerated by users, though there are numerous anecdotes detailing misunderstandings that have resulted from the inevitable loss of some intelligibility. The quality of the narrow-band speech obtained from the standard 64 kb/s log-PCM encoding technique under ideal conditions is referred to as 'toll' quality. It is the quality of telephone speech that we became used to before mobile phones were invented.

Even 64000 b/s is too high for mobile telephony and huge research effort had to be devoted to finding lower bit-rate speech coders. This effort resulted in forms of 'linear predictive' or LPC coders, the most famous of which are 'code-excited linear predictive' (CELP) coders. There are many of these in use, accommodating different bit-rates and bit-error conditions. Eight different narrow-band speech coding techniques are encapsulated in the 'adaptive multi-rate' (AMR) codec widely used in mobile telephony. The 'voice recorder' of an ANDROID phone encodes the voice in AMR form. Free software exists for converting between this and other formats such as 'wav' or 'mp3'.

The AMR speech coder encodes narrowband (50–3400 Hz) speech at bit-rates ranging from 4.75 kb/s to 12.2 kb/s. Toll quality speech is achieved at 7.4, 7.95, 10.2 and 12.2 kb/s. It is now widely used in mobile telephony and uses 'link adaptation' to select the bit-rate according to channel conditions. AMR is unlikely to give ideal results for music. The use of AMR coding for narrow-band speech reduces the bit-rate required by a factor of more than 5 when comparing with log-PCM. Compared with each channel of a CD quality sound recording, the reduction factor is more than 50.

The purpose of Task 1 will be to experiment with simple bit-rate compression techniques used for music and narrow-band speech in mobile systems. Before starting, an introduction to array processing in PYTHON, and some other techniques that will prove useful throughout this course, will be given.

2. Organisation

This is the first of five laboratory Tasks each lasting two weeks. For each Task, there will be an assessment after two weeks requiring both a report and a demonstration. The report should be a Jupyter notebook submitted to Blackboard. It should use a markdown cell to give a title and your name, and other markdown cells to give clear headings for each part of the Task and other text, including answers to questions in the lab-script. Python code should be arranged in cells

with brief comments as appropriate.
Give the notebook a title which begins with your own name: for example: JSmith_Task1.ipynb. The deadline for Task 1 reports is specified by Blackboard to be Tuesday of Week 3, 9:00 pm. This deadline applies strictly to Group F students but it will be extended to Thursday 9:00am for Group H students and to Friday 3:00pm for Group G students. No extensions beyond these deadlines are possible and lateness penalties will be applied automatically by the University.
For the demonstration, you must book a slot during the third lab session to demonstrate your working code and answer a few questions. It should take about five minutes
The marks given for the two parts of the assessment will be combined to produce a final mark for the Task. A breakdown of the allocation of marks, as percentages of the final coursework mark, is given in the table below. Each task will be marked out of 100 and then scaled to contribute to a final coursework mark. This mark will be combined 50/50 with the examination mark to produce the final mark for this module. All code and reports will be checked for originality and plagiarism.

Coursework Deliverable	Weeks	Allocation
Task 1: report & demo	1 - 2	20%
Task 2: report & demo	3 - 4	20%
Task 3: report & demo	5 - 6	20%
Task 4: report & demo	7 - 8	20%
Task 5: report & demo	9 - 10	20%

3. Computing facilities

The laboratory work requires the use of a PC running Linux or Windows with a standard sound card or equivalent chip-set. The sound-card has a microphone input socket and stereo output socket and performs analogue to digital conversion (ADC), digital to analogue conversion (DAC), amplification, filtering and buffering. Sound of bandwidth approximately 50 Hz to 20 kHz, as captured from a microphone, may be digitised with a sampling rate of 44.1 kHz and 16 bits per sample with uniform quantisation. Access to an Android mobile phone or tablet may be required at certain times for Tasks 4 and 5. The PCs in use will have available: Jupyter (previously called IPython) with Python version 2, MATLAB, Audacity, Android_Studio, an Android virtual machine and the facility for downloading Android software to a mobile device.

4. Function of the PC sound-card (or equivalent chip-set)

If a microphone is connected, a pre-amplifier amplifies the microphone voltage to a suitable level. When the sampling frequency (rate) is set to 44.1 kHz, a low-pass filter removes sound above about 20 kHz, and an analogue to digital converter performs the 44.1 kHz sampling and conversion (quantisation) to 16-bit binary numbers. A buffer accumulates suitably sized blocks of samples until it is appropriate to convey these to the processor.

The sound output or 'play-back' process requires binary numbers to be written to the sound-card in appropriately sized blocks to keep hardware buffers supplied with samples. When the sampling frequency is set to 44.1 kHz, uniformly quantised stereo 16-bit samples are converted to sound of bandwidth in the range 0 to 20 kHz.

PC sound cards are generally capable of using other sampling rates, not just 44.1 kHz, and

quantisation schemes different from uniform with 16 bits per sample. However, in these experiments the sound cards will mostly be required to use just this one digitisation scheme. If the sound is stereo, it can be converted to mono by averaging the two channels. To play back mono sound through stereo headphones copy the single channel to both a left and a right channel to form a stereo signal with two identical channels.

5. Headsets

Headsets should be connected to the built-in audio card of the PC. Microphones can be connected if needed. When not using a microphone, it is best to mute the microphone playback by opening up a ‘volume mixer’ panel, and adjusting the settings and volume levels. Use the Intel device, not the HDMI audio.

6. Setting up the required software platform

Create a working directory on your home directory (or P-drive) to contain all the exercises, documentation and required code. Any information not in this directory will be lost when you end a laboratory session.

7. Audacity

Audacity is an audio tool for recording, playing back, analysing and editing computer files containing digitised sound. The files are most commonly in a ‘binary’ format referred to as 'wav'. Such files cannot be read by text-editors or word-processors because the information is not represented as sequences of alphanumeric characters. It is more compactly represented as direct binary numbers.

To illustrate the difference, the two sample values 1000, -2000 would not be represented by a sequence of 8-bit numbers representing the ascii characters ‘1’, ‘0’, ‘0’, ‘0’, ‘-‘, ‘2’, ‘0’, ‘0’, ‘0’ . Instead just four 8-bit binary numbers would be stored: 11101000, 00000011, 00110000, 11111000. If you wish to check this, note that 1000 = hex 03E8 and -2000 = F830 in 16-bit two’s complement form. Note the change in byte order which is system specific. Some people refer to the first representation as ‘text’, ‘ascii’ or ‘formatted’ and the second representation as ‘binary’ or ‘unformatted’. The term ‘binary’ is a little misleading as all representations use binary numbers.

All 'wav' files incorporate headers which specify the sampling rate, the number of bits (word-length) per sample and other characteristics of the recording. When making a recording from a microphone, Audacity will ask the user to specify these characteristics before allowing it to be exported as a 'wav' file. When playing back the ‘wav’ sound recording, Audacity will be able to read the same information from the headers to know how to play it. The options for 'wav' file characteristics are numerous and possibly confusing. To begin with, it is best to restrict experimentation to ‘wav’ files with uniform quantisation rather than 'A-law', 'mu-law' or 'mp3'. Uniform quantisation, sometimes referred to as 'linear' quantisation', is the simplest option where voltages obtained from analogue sources are simply and directly converted to binary numbers. For example, 65 millivolts, may become '00000000 01000001' with byte order reversed. Remember that sound is pressure variation which a microphone converts to a voltage variation.

To run the Audacity tool, click on the icon or run it from a command line. The tool is largely intuitive and straightforward to use. The following paragraph gives a very brief guide to a few features you might use.

Load a previously recorded 'wav' file or record a segment of sound you wish to examine. Notice

whether the sound is stereo or mono, observe the sampling rate and the number of bits per sample. The sound duration may be anything from about one second to several hours, but initially it is best to examine quite short segments of between about three and ten seconds. Examine the time-domain waveforms produced by Audacity which are blue graphs of voltage against time.

Ten seconds of sound produce a lot of samples and the time-domain waveforms are of great complexity. Get accustomed to using the 'zoom' facility and try selecting and listening to very short segments. Zoom into a segment of very short duration and select “Frequency Analysis/plot spectrum” from the “Analyze” menu. Instead of 'time-domain waveforms' we now see 'frequency-domain' graphs or 'spectral graphs'. They indicate how the sound is composed of different frequency components. To view a single frequency spectrum, select the pull-down menu on the wave window and select “Spectrum”. Use 'Ctrl+1' and 'Ctrl+3' to zoom in and out. To compare two sound files add additional 'wav' files to the current project window through “Project” -> “Import Audio”. There is much more to explore, particularly the colourful display of a spectrogram which shows how the spectrum varies with time across the recording.

8. Assessing sound quality

For 'narrow-band' (300 to 3400 Hz bandwidth) monophonic telephone speech, it is common to use a 'MOS (mean opinion score)' as a measure of the quality of the sound. This used to require many volunteers to give subjective opinions (ratings) of the quality of the sound, according to the table below. The opinions of the users were averaged to give an 'MOS' score for the sound.

Rating	Sound quality	Level of distortion
5	Excellent	Imperceptible
4	Good	Just perceptible but not annoying
3	Fair	Perceptible & slightly annoying
2	Poor	Annoying but not objectionable
1	Bad	Very annoying & objectionable

PESQ stands for ‘Perceptual Evaluation of Speech Quality’, and offers a way of automatically assessing telephone speech quality. It was developed to model the subjective tests used to assess MOS scores and is standardized as ‘ITU-T recommendation P.862’. It is now a world standard for objective voice quality testing suitable for ‘narrowband’ telephone speech and, more recently, for speech of wider bandwidth. The evaluation software is made available by ITU-T to be used for understanding the PESQ Algorithm, evaluating its ability to perform its intended function and evaluating its computational complexity, with the limitation that none of these evaluations are used for external commercial use. Licenses are available for a wider range of uses (see www.itu.int/rec/T-REC-P.862-200102/en, and www.pesq.org).

The PESQ application may be run at command line in Windows or Linux. The executables (Windows and Linux) can be found on ‘Blackboard’. It takes a narrow-band speech file as a reference, and compares any degraded version of this file to the reference file to assess the degree of degradation. Both speech files must be monophonic, have the same sampling frequency (8000 or 16000 Hz) and use 16-bit integers to represent the samples with uniform quantisation. The reference file should contain narrow-band speech of good quality with no perceivable distortion. The use of 16-bit integers for the degraded file does not imply 16-bit

resolution. For example, if the resolution is only 8-bits per sample, the most significant or least significant 8 bits of each word can remain unused. It is best to leave the least sig 8 bits unused, i.e. set to zero.

Let the reference file be speechref.wav and the degraded version be speechdeg.wav, both sampled at 8000 Hz. At the command line, type:

pesqmain +8000 speechref.wav speechdeg.wav

The result will be an assessment of the sound quality of the degraded file assuming that the reference file is ideal. The assessment is intended to be a sort of MOS score for the degraded file. This software is not appropriate for music. It is appropriate only for speech sampled at 8000 Hz or 16000 Hz.

The simplest way to use the PESQ tool is as described above, at the command line. The user must then make a written note of the score produced. However, all scores are logged in a text file called 'pesq_results.txt' (in your working directory) therefore you can use this file as your record. Two PESQ numbers are produced: just use the first one. The PESQ application can be run from PYTHON or MATLAB by including the following operating system (OS) statement:

! linux_pesqmain +8000 speechref.wav speechdeg.wav /dev/null

(In Windows, omit 'linux_' and '/dev/null' and in Linux apply 'chmod a+x linux_pesqmain')
The linux_pesqmain exe file must be stored in the 'working directory'. The PESQ score can then be automatically extracted from the file pesq_results.txt that will be created in the same working directory. As results are accumulated, you may wish to delete the pesq_results.txt as follows at the start of your program:

! rm pesq_results.txt (or '! del pesq_results.txt' in Windows)

After running the PESQ tool, the scores generated can be printed from 'pesq_results.txt' or extracted automatically by parsing. The 'COMP28512_utils.py' library file, which is downloadable from the Blackboard site, has a function which does this (see later).

In your assessments of voice quality, the MOS or PESQ score can be augmented with specific detail using phrases such as 'muffled', 'lacking bass', 'intelligible', unintelligible', 'natural', 'unnatural' or 'machine-like', 'containing clicks and pops' and others you may think up. If there is distortion, you may note whether it appears to be correlated with the sound or largely independent of the sound.

9. Python and Jupyter

9.1. Introduction: For revision of 'basic Python', there are many documents and tutorials available on-line. A rapid revision is available on YouTube as "Python Programming" (Learn Python in one video) by Derek Banas. In this course, we need to use Python with a number of well known libraries: 'Numpy', 'Scipy', 'matplotlib' and others.

For NumPy see the user guide: http://docs.scipy.org/doc/numpy/user/ and the reference guide: http://docs.scipy.org/doc/numpy/reference/ or http://docs.scipy.org/doc/numpy/numpy-ref-1.10.1.pdf

For SciPy, see:

http://docs.scipy.org/doc/scipy/reference/ or http://docs.scipy.org/doc/scipy-0.17.0/scipy-ref-0.17.0.pdf

For matplotlib see: http://matplotlib.org/contents.html or http://matplotlib.org/Matplotlib.pdf

The pdf documents referred to above are mirrored on Blackboard. They are: numpy-ref-1.10.1.pdf, scipy-ref-0.17.0.pdf and Matplotlib.pdf. There is also documentation on the 'magic' (%.) statements of Jupyter.

A Python interpreter may be downloaded from www.python.org/downloads. There are two versions: 2 and 3. Version 3 is not totally compatible with the language and libraries of version 2. Therefore many scientific users have stayed with version 2. Python may be used with a basic general-purpose text editor such as 'gedit' in Linux or 'Notepad' in Windows, or it may be

used with a variety of integrated development environments (IDEs) such as IDLE or Spyder. Python text files are traditionally titled '*.py'.

In this course, we use a more sophisticated form of IDE called Jupyter. It was originally called IPython which stands for interactive Python. Windows users can obtain Python, Spyder, Jupyter and most of the libraries they need by downloading 'WinPython' from www.sourceforge.net. The version WinPython-64bit-2.7.10.3 was used to prepare these notes, though later versions are now available. Linux users can easily download Jupyter at home. Jupyter is available on laboratory machines.

Jupyter is a web-server based IDE which can be run on a single PC using a browser set to the 'local loop' address http://localhost:8888 which is ' http://127.0.0.1:8888 ' with port 8888. The Jupyter server could be elsewhere, and the client could be on a mobile phone or tablet. Instead of '*.py' text files, the server produces '*.ipynb' files which are called notebooks. Notebooks contain both Python code and HTML text referred to as 'mark-down'. Jupyter is therefore a combined software development and documentation environment.

Python with NumPy and other extension libraries is replacing MATLAB these days for representing, processing and plotting sampled signals. The 'list' and 'tuple' objects provided by basic Python when you type List1 = [1,2,3,'Hello'] or Tup1 = (1,2,3,'Hello') can be used as arrays of purely numerical information, but they are not efficient or fast enough for processing quite long arrays containing, for example, one second of speech sampled at 8 kHz (8000 array elements needed). Also, basic Python does not provide the numerical array processing functionality required for these laboratory tasks. NumPy is a library which provides this functionality. It is part of a larger library called 'SciPy' which also provides more 'numeric/scientific' functions that can be embedded in a Python program script. The 'matplotlib' library provides the means of drawing graphs in Python.

9.2. Command window: In Linux, launch Jupyter by typing 'startx' (if necessary) and bringing up a terminal window. Then type:

```
cd ~            (~ means 'home-directory)
mkdir "myWorkDir"
cd "myWorkDir"
anaconda2
jupyter-notebook
```

This procedure launches Jupyter and configures it to use the working directory 'myWorkDir', or whatever you choose to call it, in your 'home directory'. Hopefully you should not lose any data when you log out, but please check this before you write any serious code. When you exit from Jupyter in Linux, you may be required to enter '~C' and answer 'yes' before the session closes down completely. You may need to attend to your 'bash–profile' (gedit ~/.bash_profile) before you can run Jupyter, especially if you have changed it to run software for other courses.

In Windows, if you have downloaded a WinPython installer, execute it to create a folder containing the software you need. Open this folder and click on the 'Jupyter' icon. By default, WinPython 2.7.10.3 places '*.ipynb' notebooks in the folder 'Notebooks'. Any supporting software or 'wav' speech or music files will be found if they are placed in this folder.

The Jupyter server will link automatically to your default browser. You may need to change your default browser. There may be more than one version of Python and Jupyter available on your PC. Use an instance of version 2 rather than version 3. Jupyter notebooks appear not to be compatible with previous IPython versions.

9.3. Working directory: Check the current working directory at the command line. If you place any files here, Jupyter will see them. Create your own working directory and change to it as

suggested above. In the CS laboratories, if the working directory is not in your home directory or on your P-drive, you will lose its contents when you log-off.

9.4. Help: When you open a Jupyter notebook, there is a 'help' facility available in the menus.

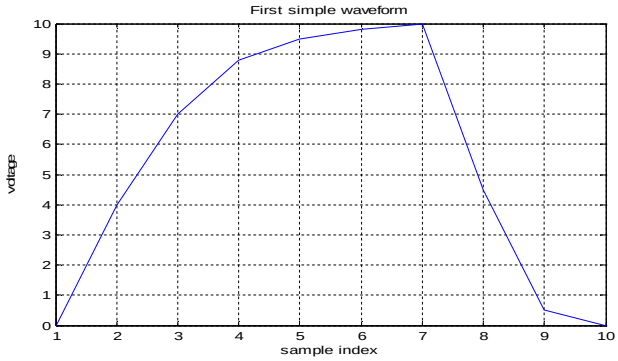
9.5. Running Python at command level: In Python, sequences of instructions may be stored as *.py files which are just script files in ascii form. We can call these programs. To create a script file using Python, simply open a text editor, enter the required statements then save the file as name.py. Typing 'python name.py' will then run the program.

9.6. Running Jupyter: To experiment with Jupyter, after typing 'Jupyter Notebook', observe the web-page that is displayed and notice that it allows you to create a new 'notebook' in the working directory. The notebook is initially 'untitled', but you can click on this name and change it. Now you see a single 'cell' labelled 'In [ ]'. Copy and past the following Python and Jupyter statements into this cell, making sure that the tag above it specifies 'code' and not 'markdown' or anything else. The first statement allows us to use 'NumPy' which we would like to refer to as 'np' for short. Other libraries are then specified. Statements beginning '%' are 'magic' statements for Jupyter only and give an error with basic Python. The code below creates an array 'y' using NumPy, and then plots its contents. In fact, y is created from a basic Python list [0. 4, 7 …] which is converted to a NumPy array using the NumPy function 'np.array'. To run the code in the cell, press the 'shift + enter' keys together, or click on the black triangular icon. Notice that 'In [ ]' changes to 'In [1]' and that you may get a syntax error (as happens frequently when you cut and paste from word-processing documents). In this case, you just need to hand-edit the quote marks. Run again, and 'In [1]' changes to 'In [2]'. The program in the cell should then run successfully to produce a message and a graph.

```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
print("Hello Manchester");
y = np.array([0, 4,  7,  8.8,   9.5,  9.8, 10,  4.5,  0.5,  0])
fig, axs = plt.subplots(1)
axs.plot(y)
axs.grid(True)                          # Turn on grid lines
axs.set_xlabel("Time / samples")  # Set x-axis label
axs.set_ylabel("Amplitude")         # Set y-axis label
axs.set_title("Simple graph from Section 9.6")
```

If the script is currently untitled, you will be prompted to give it a name before it will run. Enter a name, like 'myTest', and the program will be saved as file myTest.ipynb (a Jupyter notebook) in the 'working directory' and then it will run. This program illustrates some statements that you may find useful, especially graph plotting.

9.7 Arrays and graph plotting: The program just illustrated creates a sampled (or discrete time) signal whose samples are conveniently stored in an array y. If the samples are assumed to be taken at intervals of 1 second, the sampling rate will then be 1 sample per second, i.e. 1 Hz. For all arrays in Python, the first element has index zero.



NumPy has both arrays and matrices. We will use only arrays to begin with. Arrays do not always have to be declared in advance but they can effectively be 'declared' as follows:

A = np.zeros(100); declare an array with one row containing 100 zero elements

Going beyond the 100 declared elements will raise an exception. If array A is one-dimensional, its length (number of elements) is obtained by: 'L = A.size'

If array A is 2-dimensional, 'L = A.size' again gives the total number of elements and the statement: 'S = A.shape' gives both dimensions of A as a 'tuple' S. For example, if A = np.zeros([100,2]), then A.shape gives (100,2). Such a statement may be useful when dealing with stereo sound files which are read into 2-dimensional arrays, with right and left channel signals.

Arrays can be concatenated using the NumPy 'horizontal stack' function as follows:

A = np.hstack([np.ones(3), np.zeros(2)]) giving [1.0,1.0,1.0,0.0,0.0].

This works for row-vector arrays. Use np.vstack for column-vector arrays.

Also, parts of arrays can be extracted by slicing as follows:

A = np.array([1,2,3,4,5,6]) [1:3] gives the array A = [2.3] .

Numpy supports a much greater variety of numerical types than Python does. The following table shows which are available, and how to modify an array's data-type. By default, np.array gives us 64-bit floating point numbers (I think) but the following statement:

y = np.array([0, 4, 7, 18, 9, -9, 100, -70, 0, -1], int16)

gives us an array of signed 16-bit integers. See documentation in NumPy, as mirrored on Blackboard, for more detail about numeric types and converting between them. You can find out the data-type of a variable 'var' by typing 'var.dtype'. For more details about variable types, see http://docs.scipy.org/doc/numpy-dev/user/basics.types.html#id3

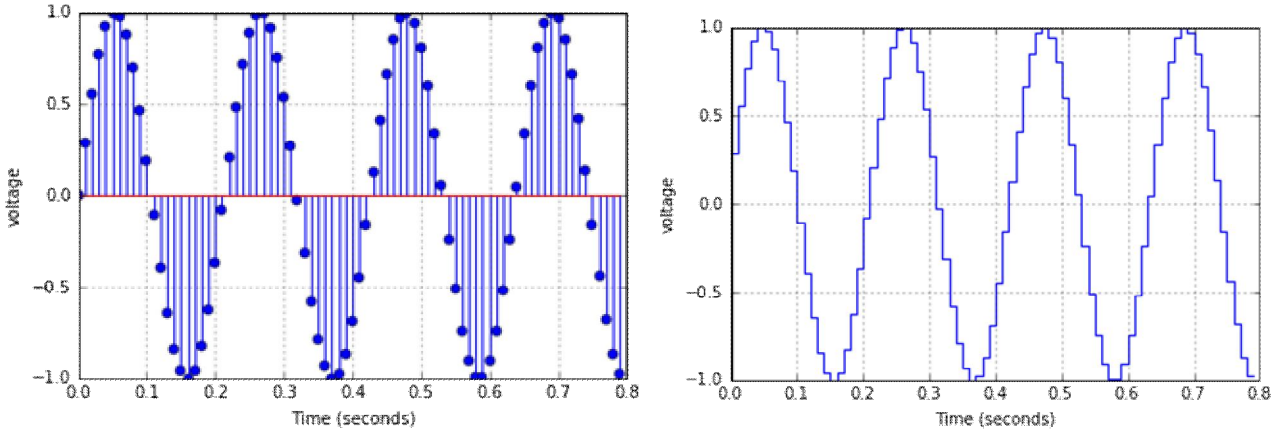
Data type	Description
bool	Boolean (True or False) stored as a byte
int	Default signed integer type (same as long; normally either int64 or int32)
int8	Byte (-128 to 127)
int16, int32, int64	Signed integer (16, 32 or 64 bits)
uint8, uint16, uint32, uint64	Unsigned integer (8, 16, 32 or 64 bits)
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float (float64)	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex64	Complex number, represented by two 32-bit floats (real & imaginary parts)
complex (complex128)	Complex number, represented by two 64-bit floats (real & imaginary parts)

9.8. Generating an array containing a sampled sine-wave: Now consider the following sequence of statements (Cell 1) which creates and plots a list y containing 80 samples of a sine-wave. The sine-wave has amplitude 1.0 volts. Its frequency is 4.7 cycles per second (Hz) and it is sampled at *Fs* =100 samples per second (Hz). The sampling points are therefore at intervals of *T* seconds where *T* = 1.0/*Fs* = 0.01 seconds. The sampling points are stored in the list x. The 'plot', 'stem' and 'step' functions provided by the matplotlib library allow graphs of y against x to be drawn in three different ways, and these functions work for NumPy arrays as well as lists and tuples. The 'stem' and 'step' plotting functions are like 'plot', but emphasise that a digital signal has values defined only at the sampling points.

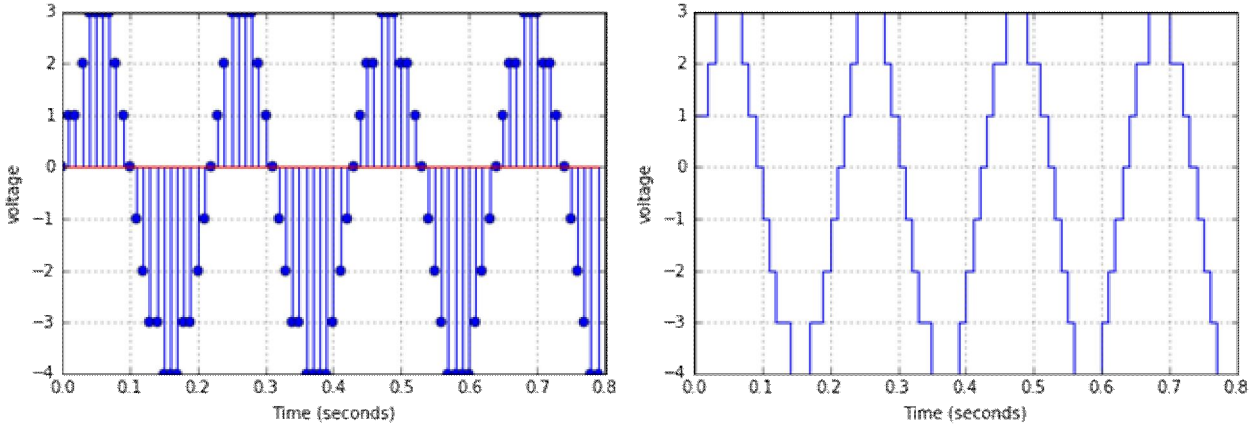
```
# Cell 1 for creating & plotting a sine-wave using lists.
import numpy
from matplotlib import pyplot
% matplotlib inline
Fs = 100.0        # Sampling frequency (Hz)
T = 1.0  / Fs     # Sampling period (seconds)
A = 1.0           # Amplitude of sine-wave
f = 4.7           # Frequency of sine-wave (Hz)
y = [0]*80 ; x = [0]*80     # make x & y both lists
for n in range(0,80):
    y[n]=  A * numpy.sin( 2 * numpy.pi * f * n * T )     # A sin(2πfnT)
    x[n]=  n*T
fig, ax0 = pyplot.subplots(1)
ax0.stem(x, y)
ax0.set_xlabel("Time (seconds)") ; ax0.set_ylabel("voltage")
ax0.grid(True)
fig, ax1 = pyplot.subplots(1)
ax1.step(x, y)
ax1.set_xlabel("Time (seconds)") ; ax1.set_ylabel("voltage")
```

```
# Cell 2 for quantising y with 3 bits per sample (See Subsection 9.10)
y=numpy.array(y) ; x=numpy.array(x)
quant_y = numpy.round (y * 3.5 - 0.5 )
quant_y = numpy.int16(quant_y)
fig, ax2 = pyplot.subplots(1)
ax2.stem(x, quant_y)
ax2.set_xlabel("Time (seconds)") ; ax2.set_ylabel("voltage")
ax2.grid(True)
fig, ax3 = pyplot.subplots(1)
ax3.step(x, quant_y)
ax3.set_xlabel("Time (seconds)") ; ax3.set_ylabel("voltage")
ax3.grid(True)
```

Note that x and y are 'lists' in Cell 1. In Cell 2, they are converted to NumPy arrays by inserting the statements: x=np.array(x) and y=np.array(y). The NumPy version of y is then quantised to 3 bits per sample which can represent integers in the range -4 to +3. The third statement in Cell 2 is quite clever and ensures that we use all 8 quantisation levels. Note that the sine-wave has been shifted down by subtracting 0.5 to achieve this. Producing quantised values only in the range ±3 would waste one level. To display the 80 samples in list y (or array y after conversion), just type 'print y'.



Graphs from Cell 1: Unquantised sine-wave plotted using 'stem' and "stairs'



Graphs from Cell 2: Sine-wave quantised with 3 bits/sample

9.9. For loops and array processing in Python: The use of 'for' loops and 'lists' as provided by introductory Python would be too slow for processing large arrays of samples. You can avoid 'for' loops when processing large arrays as follows:

```
n = np.arange(0, 10001)                # Makes n a Numpy array of integers 0, 1, 2, … 10000
y = A * np.sin( 2 * (np.pi) * f * n * T)    # y is  Numpy float array of 10001 sine-wave samples
```

Look at NumPy documentation (on Blackboard) for guidance. Notice the spelling of 'np.arange'. Also, be careful with casting, and when giving specific values to variables such as Fs (sampling frequency), and T (period between samples), which are not naturally integers, make sure they are floats by writing statements like 'Fs=100.0' rather than Fs=100. Note that np.pi is the Python representation of $\pi = 3.14159....$

9.10. Interrupting execution: When processing very large arrays of numbers, the cost of making mistakes can be to take up minutes of computing time, perhaps producing lots of print-out and maybe strange and boring sounds. Moving the curser to the command window and entering 'control' and 'C' together will, in principle, interrupt the processing. You might have to repeat the interrupt request a few times.

9.11 Conditional instructions (if, else, while, etc.): These are straightforward and best dealt with by typing 'help if' help while' etc.

9.12. Functions: Python provides many functions, and the user can easily develop his/her own. The following is an example:

```
def  sumof(x,y):
   # sumof: an example of a function
   z = x + y;
   return z
```

If this is saved in 'my_functs.py' and stored in the working directory, it can be imported by any other Python script as follows: 'from my_functs import sumof'.

9.13. Strings: Typing s = "Manchester" makes s a string. There are many operations in Python for manipulating such a string. You may not need any of these, but if you do, use 'help' to explain them and search for others.

9.14. Input-output: Simple ways to print messages and numbers on the screen, and to input numbers and text from the keyboard are provided by basic Python. For example:

```
print "Sampling frequency is ",Fs, " Hz"
N = input("Enter a number on the keyboard: ")
```

To read or write sound files in 'wav' format into or out from a NumPy 'int16' array y, use wavfile as follows:

```
from scipy.io import wavfile
wavfile.write("fileName.wav", Fs, y);
(sampling_freq, y) = wavfile.read("fileName.wav");
```

The write statement above produces, in the working directory, a wav-file "fileName.wav" with 16 bits per sample and a sampling rate *Fs* Hz. It may then be listened to and analysed using a media player or 'Audacity'. The samples stored in the NumPy 'int16' array y must lie in the range -2^{15} to $(2^{15} - 1)$. If any samples are outside this range before they are converted to NumPy 'int16' form they will be 'limited' or 'clipped' and this will produce non-linear distortion that may not sound very nice. If the array y has only one row, the output sound will be mono. If y has two rows, stereo sound will be produced; the first row feeding the left channel and the second row feeding the right channel.

The wavfile.read function delivers integers (samples) in the range -32768 to +32767, as stored in fileName.wav', to the 'int16' NumPy array y. The value of the sampling frequency is read from the file itself.

The 'COMP28612_utils.py' library, which may be downloaded into your working directory from Blackboard, contains a function for playing out sound from a wav file. To use this program insert the following statements:

```
import comp28512_utils as utils
utils.audio_from_file("fileName.wav")
```

To listen to the sound stored in 'int16' NumPy array y, use statement:

```
utils.Audio(y, rate=Fs,)
```

These library functions do not play out the sound straight away. Instead they create 'sound-bar' icons within your program (which we call a notebook) when you run it. Your notebook may generate lots of sound-bars. You will click on the sound-bars one-by-one to hear the sounds you have produced. It is essential that each 'sound-bar' icon is clearly labelled with a print message such as: "Sine-wave of frequency 250 Hz sampled at 44100 Hz".

9.15: Using Jupyter: Experiment with the development and editing facilities of Jupyter before starting any serious work. Note the 'Cell' tab with its drop-down menu containing 'Run all' and

'All Output'. Observe the effect of selecting 'Clear' from the 'All output' entry after you have been testing the code in some cells. Also, observe the effect of selecting the Kernel tab and 'restarting the Kernel'. When testing a complete program after editing some of the cells, you can start from scratch by clearing all output (Cell>All Output>Clear), restarting the kernel (Kernel>Restart) and then selecting Cell>run all.
When using Jupyter, instead of writing a program and then a separate report on what it does, and what conclusions can be drawn from it, you just write one 'ipynb' notebook. This one notebook contains all your text and the Python code which generates all the results.
The Python code can be split into cells which can be tested separately. Cells can be used for Python code or documentation referred to as 'markdown. Notice the 'Code'/'Markdown' selection that can be made for each cell. The text can within a markdown cell be HTML with titles, headings and body text. For example:
<h1> COMP28512 Laboratory Task 1: Student's name: Zhang San</h1>
 Part 1.1. Generating & listening to sine-waves
Don't forget to save your notebook. Note that control-L will turn on line numbering.

10. Quantisation

Many of the calculations within Python code will use floating point arithmetic typically using 32 or 64 bits for each variable. Such word-lengths would be too expensive for transmitting sound, image and other samples over radio and also for storing them in mobile phone memory. Therefore, we often have to convert these floating point numbers to integers and this is quantization. Also we often have to use further quantization to reduce the number of bits used for each integer to achieve further savings in bit-rate capacity.

10.1. Quantisation to 16-bit signed integers: Each sample of the sine-wave generated by Cell 1 in Section 9.8, or the code in Subsection 9.9, is within the range ±A and is represented by a floating point number. If the samples are to be stored in a 'wav' file, each sample must be converted to a 16-bit integer in the range -2^{15} to $(2^{15}-1)$. Note that $+2^{15}$ cannot be represented in signed 16-bit form. The statement:

```
quant_y = np.round( y * (2**15 – 1) )
```

produces integers in the range ±32767 when $-1 \le y \le 1$, but they are still represented as type 'floating point'. They can be converted to binary integers of type 'int16' as follows:

```
iquant_y = np.int16(quant_y)
```

The integer -2^{15} = -32768 will never occur with y in the range ±1.0. But this fact is often disregarded.

10.2. Quantisation using fewer bits per sample: If we decide to use only 8 bits per sample rather than 16, we can only represent integers in the range -2^7 to (2^7-1), which is -128 to 127. When $-1 \le y \le 1$, the statements:

```
quant_y = np.round( y * 127 )
iquant_y = np.int16(quant_y)
```

produce signed 16-bit integers in the range ±127. Then, only the least significant 8 bits need be transmitted or stored, and we can forget about the most significant 8 bits. (They will be all zeros for positive integers and all ones for negative integers.)
A lproblem occurs if you want to listen to 8-bit quantized sound represented in this way because it will be very quiet when sent to a 16-bit D to A converter. The problem is solved by multiplying (scaling up) each 8-bit quantised sample by 2^8 (= 256) to produce a 'listening version' of the array which is sent to the D to A converter or Audacity. This produces an integer array whose

elements lie between ±32512 and have all 8 least significant bits equal to zero. So the number of usable bits per sample remains the same.

10.3. Quantisation using very few bits per sample: The quantisation scheme in Subsection 10.2 would still work for a 3-bit quantiser. The statements:

```
quant_y = np.round( y * (2**2 -1))
iquant_y = np.int16(quant_y)
```

would produce signed 16-bit integers in the range ±3, allowing the most significant 13 bits to be disregarded. However, as mentioned in Section 9.8, a 3-bit quantiser can have integers in the range -4 to +3, and we are wasting one of them. This waste occurred also in Subsections 10.1 and 10.2, but it did not matter so much because there were so many quantization levels.. But with only 8, we must try to use them all. To achieve this for a 3-bit quantiser, use the statements:

```
quant_y = np.round( y * 3.5 – 0.5)          # where -1.0 ≤ y ≤ 1.0
iquant_y = np.int16(quant_y)
```

and remember to add 0.5 when converting a quantised signal back to floating point form. This process can be used for any order of quantiser, but is most important when there are very few bits per sample. To produce a 'listening version' of iquant_y, multiply by 2^{13}.

11. Changing the sampling rate of a digitized signal

11.1. Re-sampling and decimation: Bit-rate savings can be achieved by quantization and also by reducing the sampling rate used to represent a signal. Among the many useful functions provided by the Python SciPy libraries is a function for changing the sampling rate of a signal stored in an array. Invoking the function 'resample' as follows:

```
from scipy.signal import resample
yr = resample(y, (y.size)/2);
```

reduces the sampling frequency of the signal stored in y by a factor 2. Python also has a function called 'decimate' for reducing the sampling rate. The following statements reduce the sampling frequency of y by a factor 7:

```
from scipy.signal import decimate
yr = decimate(y, 7)
```

Both these functions take samples of sound obtained at one sampling frequency, Fs Hz say, and re-samples the sound to a different sampling frequency. Resample can increase as well as decrease the sampling frequency. The effect is as though the sound were played back through speakers, recaptured by a microphone and re-digitised using a different sampling frequency.

11.2. Aliasing distortion: You can reduce the sampling frequency very easily by just omitting samples. The following two lines of code reduce the sampling frequency from Fs to Fs/2 by omitting alternate samples:

```
n = np.arange(0 , y.size/2)
yr = y[n*2]
```

But with just these two statements, the down-sampled signal in array yr may be distorted by 'aliasing'. This is because y can have frequency components up to Fs/2 whereas yr can only have frequency components up to (Fs/s)/2 = Fs/4. The 'Sampling Theorem' tells us that the sampling frequency must be greater than twice the highest frequency component of a signal to avoid aliasing distortion. Therefore, any frequency components above Fs/4 in y cannot be represented correctly in array yr. Unfortunately, they do not just disappear. Instead they

become distorted components of yr and can sound very bad.

11.3 Avoiding aliasing distortion: To avoid the possibility of 'aliasing distortion', the 'resample' and 'decimate' functions have to filter out all components of y above Fs/4 before omitting any samples. Therefore, they have to do more than just omit samples.

11.4. Increasing the sampling rate: :Calling 'resample' as follows:

```
yr = resample(y, (y.size)*1.5)
```

increases the sampling frequency by multiplying it by a factor 1.5. Using 'resample' to increase the sampling frequency of a signal will not affect its bandwidth. If the sampling frequency is *Fs*, the same signal can be sampled at 2×*Fs* without having to modify its bandwidth.

11.5 Listening to the effect of decimation: Assume you wanted to reduce the sampling frequency by a factor of 7 and then listen to the resulting signal. An obvious approach is:

```
yr = decimate(y,7)   # Reduce sampling freq by factor 7 (with antialias filtering)
Audio(yr, rate=Fs/7)
```

However, this may fail if your sound equipment does not allow a sampling frequency of *Fs*/7, i.e. 6.3 kHz if *Fs* = 44.1 kHz. So instead, you can to use 'resample' to increase the sampling rate back to *Fs* as follows:

```
yr = resample(y, (y.size)/7)   # Reduce sampling freq by factor 7 (with filtering)
yy = resample(y , (yr.size) * 7.0)   # Increase sampling freq by factor 7
Audio(yy, rate=Fs)
```

The second 'resample' statement does not just reverse the effect of reducing the sampling rate. It does not just get you back to y. Any distortion produced by the first re-sampling process remains there.

12. Speech and music files.

Download the following files from Blackboard or the following web-site into your working directory: www.cs.man.ac.uk/~barry/mydocs\MyCOMP28512

1. 'HQ-speech44100-mono.wav' and 'NarrobandSpeech8k.wav',
2. 'HQ-music44100mono.wav' and 'SVivaldi44.1mono.wav'
3. 'comp28512_utils.py',
4. 'linux_pesqmain' or 'pesqmain.exe for Windows'
5. Task1examples.ipynb

HQ-speech44100-mono.wav is a high quality monophonic recording of about ten seconds of speech sampled at 44.1 kHz with 16 bit per sample uniform quantisation. This is not narrow-band (telephone quality) speech. Narrobandspeech8k.wav is a recording of narrowband speech sampled at 8000 Hz and uniformly quantised with 16 bits per sample. The second pair of wav files consists of high quality monophonic recordings of about 20 seconds of music sampled at 44.1 kHz with 16 bits per sample uniform quantisation. The recordings were originally stereophonic, but have been converted to mono to simplify your experimental work. The following code serves as a system test and also illustrates how quantisation may be achieved. It is contained in Test1examples.ipynb, or you can cut/paste it into a cell. Make sure you understand each section, and ask a demonstrator to listen to the output sound when you get it working.

```
%reset
import numpy as np
from scipy.io import wavfile
# Read in a wav file from working directory
(Fs, speech) = wavfile.read("HQ-speech44100-mono.wav")
# Display sampling frequency (Hz) as read from the wav file:
print "Fs = ", Fs
SM=max(abs(speech))          # Get maximum amplitude
speech = speech/float(SM)    # Scale maximum to 1 (note the float)
# Plot graph of the whole speech with 16 bits per sample:
from matplotlib import pyplot as plt
% matplotlib inline
fig,axs=plt.subplots(1)
axs.plot(speech)
# Plot 2000 samples of the speech with 16 bits per sample:
fig,axs=plt.subplots(1)
axs.plot(speech[20000:22000])
# Qspeech = np.round(speech * (2**4 -1))     #Quantise to +/-15 as float (suboptimal)
Qspeech = np.round(speech * (2**4 -0.5) - 0.5)  #Quantise to -16 to +15 as float (better)
Qspeech = np.int16(Qspeech)                      #Convert to int16 form
# Plot 2000 samples of quantised speech: int16 using 5 bits per sample:
fig,axs=plt.subplots(1)
axs.plot(Qspeech[20000:22000])
# Increase amplitude of quantised speech to approx 32000 & plot it all:
# scaledQs=Qspeech*(2**11)                    # (suboptimal)
scaledQs=np.int16( (Qspeech+0.5)*2**11 )  # (Better)
fig,axs=plt.subplots(1)
axs.plot(scaledQs)
# Plot 2000 samples of the scaled-up quantised speech (listening version):
fig,axs=plt.subplots(1)
axs.plot(scaledQs[20000:22000])
# Now write the listening version of 5-bit quantised speech to a wav file:
wavfile.write("quantisedSpeech5.wav",Fs,scaledQs)
# Listen to the quantised speech (from array):
from comp28512_utils import Audio
Audio(scaledQs,rate=Fs)
# Alternatively, listen to quantised speech as stored in wav file:
from comp28512_utils import audio_from_file
audio_from_file("quantisedSpeech5.wav")
```

13. Running PESQMain to assess narrow-band speech quality

Look at the code listed below (also in the Task1examples notebook). It takes, as a reference, the file NarrobandSpeech8k.wav which contains 16 bits per sample uniformly quantised narrowband (telephone) speech sampled at 8 kHz. It can compare this reference with another wav file containing uniformly quantised narrowband speech, again sampled at 8 kHz. The second file could be a quantised version of the first one. However, in the example given, we just compare the reference file with itself. The notebook makes an operating system call (!) to the linux or Windows version of 'PESQmain.exe' which compares the second speech file against the reference. Any distortion in the second wav file will be reflected by a PESQ-MOS score which is reduced from the maximum possible value of 4.5.

PESQMain produces displayed text (which we try to suppress using the '> /dev/null/' message) . Unfortunately the suppression seems not to work in Windows. This text is written

to a file called pesq_results.txt in your working directory. This may be printed out, or parsed to extract the required PESQ-MOS score.

The parsing may be done by a method 'get_pesq_scores' in the 'comp28512.utils.py' library which creates an array called pesq_results. Scores may be extracted as seen below. Note that PESQMain is only applicable to narrowband speech sampled at 8000 Hz or 16000 Hz. It cannot be applied to high quality (44100 Hz sampled) speech or music. The file pesq_results.txt accumulates all results obtained from PESQMain, but you may like to delete the text file from time to time to remove unnecessary data.

```
# Running PESQMain (in working directory with 'chmod a+x' set ) to obtain PESQ-MOS score:
# Windows & Linux versions (compares narrowband speech file with itself):
# ! pesqmain +8000 NarrobandSpeech8k.wav NarrobandSpeech8k.wav
! linux_pesqmain +8000 NarrobandSpeech8k.wav NarrobandSpeech8k.wav > /dev/null
from comp28512_utils import get_pesq_scores
# comp28512_utils must be in working directory
pesq_results = get_pesq_scores()
score = pesq_results["NarrobandSpeech8k.wav"] ["NarrobandSpeech8k.wav"]
```

14. Log-PCM encoded speech

In practice, telephone speech coding at 64000 bit/s requires the use of 'A-law' or 'mu-law' logarithmic companding. The simplest is mu-law which takes samples, x say, scaled to lie in the range ±1, and converts them to companded samples, y say, which are also in the range ±1. The companding (compression) is achieved by the following formula:

$$y = sign(x) \times \frac{\log_e(1+\mu \times |x|)}{\log_e(1+\mu)}$$

By convention, the value of μ (mu) is 255, |x| = abs(x), and the function sign(x) is 1 when x > 0, -1 when x <0 and zero when x =0.

Instead of quantizing and transmitting samples x directly, they are 'companded' first. This has the effect of making samples that are close to zero, larger than they really are, and they are then represented more accurately when they are quantized. Samples of x not close to zero are not given this advantage since they need it less. The mu-law equation is easily implemented by NumPy, and the array processing magic of NumPy allows the equation to be applied to arrays x and y just as easily as to individual values; with a considerable increase in processing speed.

Before applying it to speech, it is instructive to plot y against x over a range of about 200 values of x from -1 to 1. Observe how values of x close to zero are increased in value.

When speech is transmitted in compressed form, it must be uncompressed at the receiver. This is achieved by the following 'mu-law expansion' formula:

$$u = sign(y) \times \frac{(1+\mu)^{|y|} - 1}{\mu} \quad = \quad sign(y) \times \frac{\exp(\log_e(1+\mu) \times |y|) - 1}{\mu} \qquad \text{(use either)}$$

It is just the inverse of the mu-law compression formula, and can also be plotted as a graph of u (uncompanded value) against y.

15. Sound Sampling (Task 1)

As a ‘quick start’, run the code listed in Sections 9.6, 9.8, 12 and 13 using Jupyter. The code is provided on Blackboard as a Jupyter notebook called 'Task1examples.ipynb'. Just place it in your Work-Directory along with the other files listed in Section 12, select it in Jupyter, then select ‘Cell - Run All’. It is sometimes useful to select ‘Cell – All output – Clear’ before running. You can also run the notebook one cell at a time by selecting ‘Cell Run’.

Part 1.1. Generating and listening to sine-waves [3 marks]

Write a Python function called playSin(*F,Fs*). It must generate an ‘int16’ NumPy array containing a sine-wave of amplitude 30000 and frequency *F* (Hz). The sine-wave must be sampled at Fs Hz and last for 3 seconds. The array should be written to a clearly labelled 'sound-bar' ready to be played out as sound.

In another cell, with Fs = 44100 Hz call the function twice; with F = 220 Hz, and then with F = 440 Hz. The fundamental frequency of a musical instrument playing the note 'A' closest to ‘middle C’ is 220 Hz. Make sure that the sound-bar labels are clear.

Write another cell that calls the function for the following values of F: 1000 Hz, 2000 Hz, 4000 Hz, 8000 Hz, 16000 Hz, 18000 Hz and 20000 Hz. Note the frequency that is reached before you can no longer hear any output. DO NOT ADJUST THE VOLUME OF THE AMPLIFIER WHILE DOING THIS TEST.

Create a markdown cell with answers to the following questions:

1. Describe the sound produced by a sine-wave of frequency 220 Hz.
2. How does the sound differ from that produced by musical instruments and the human voice?
3. Describe any differences and similarities between the sound at *F*=220 to *F* = 440.
4. What is the highest frequency you could hear?
5. Could there be other factors that affect your answer to question 5?
6. Why is it best not to use ‘for’ loops for this software?

Part 1.2. Demonstration of aliasing for sine-waves [2 marks]

Create a new cell which calls the function playSin with Fs = 4000 Hz and *F* starting at 500 Hz, and then increasing to 1 kHz, 1.5kHz, 2 kHz, 2.5 kHz, 3 kHz, 3.5 kHz. When you listen to the sound-bars, you should be able to hear aliasing distortion. Answer the following questions in a mark-down cell:

1. Why does aliasing distortion occur in this experiment?
2. What is the effect of aliasing on each of the six sine-waves?
3. Explain what happens when F = 2000 Hz.

Part 1.3. Processing a music file to demonstrate aliasing [2 marks]

Create a new cell that reads in the violin music from ‘SVivaldi44.1mono.wav’ and produces a sound-bar that plays it out unmodified. Then reduce the sampling frequency by a factor of 11 without an antialiasing filter. You should hear two effects. Then repeat the down-sampling experiment using 'decimate' or ‘resample’, to reduce the aliasing distortion and listen again.

Record answers to the following questions:

1. How does your program know what is the original sampling frequency?
2. Could you hear any distortion in the original wav file?
3. Describe the two effects you heard with the sampling frequency reduced to about 4 kHz without antialiasing filtering.
4. What was the effect of the antialiasing filtering when you used 'resample' or 'decimate'?
5. How does the aliasing distortion affect musical notes?
6. Is an antialiasing filter always necessary before sampling music?

Part 1.4. Reducing the bit-rate of a speech or music file by reducing the sampling frequency [4 marks]

By how much can we reduce the bit-rate of a speech or music file by reducing the sampling frequency without severely affecting the quality of the sound? Experiment with different sampling rates for the high quality speech file 'HQ-speech44100-mono.wav' and the high quality music files 'HQ-music44100mono.wav'. Use the 'resample' or 'decimate' function to avoid aliasing distortion. Produce two sound-bars, to illustrate the sound produced by what you consider to be the minimum acceptable sampling rate for speech and the minimum acceptable sampling rate for music. Answer the following questions in both cases when there are 16 bits per sample with uniform quantisation.

1. What is the effect on speech of reducing the sampling rate?
2. What you consider to be the minimum acceptable sampling rate for speech that you would like to hear from the built in speaker of your mobile phone?
3. What is the effect on music of reducing the sampling rate ?
4. What you consider to be the minimum acceptable sampling rate for music that you would like to hear from your mobile phone when using headphones or a good quality speaker?

Part 1.5. Reducing the bit-rate for music by reducing number of bits per sample [3 marks]

Start by reading in the music file 'HQ_music44100_mono.wav' where the number of bits per sample is 16. Produce a sound-bar for this recording in its original form.. By the method introduced in Section 10 (Quantisation), produce a program which produces quantised versions of this file where the number of bits per sample (call this *NB*) may be specified. Listen to the quantised versions of the music and observe the degradation in quality that occurs as NB is decreased. Decide what you consider to be minimum acceptable value of *NB* for music sampled at 44.1 kHz and produce a labelled sound-bar to illustrate the quality of sound thus obtained. . Record your answers to the following questions:

1. What do you consider to be the minimum acceptable value of *NB* for music sampled at 44.1 kHz?
2. Describe the distortion that occurs as *NB is* decreased from 16 towards 3.
3. Does the nature of the distortion change when the number of bits per sample becomes three or less

Part 1.6. Telephone quality speech [3 marks]

Read in the recording of narrowband speech from *Narrobandspeech8k.wav* which uses 16 bits

per sample and is sampled at 8 kHz. Develop code to apply quantisation to reduce the number of bits per sample, NB, to 10, and to output this to a 'listening version' wav file. Produce a clearly labelled sound-bar for this wav-file using 'audio_from_file', and compare it with the original wav file using PESQMain. Extend the code to repeat this procedure for other values of *NB*: 8, 6, 4 and 3. When you run the code, record your own assessments of the quality of the degraded signals and note the PESQ score for each value of *NB*. Record answers to the following questions:

1. Can you hear any difference between the original 16 bit per sample version and your 8 bit version?
2. Taking '*Narrobandspeech8k.wav'* as the reference, what are the PESQ scores for (a) your 8 bit per sample version (b) a 4-bit per sample version and (c) any others you tested?
3. Compare your own assessments with the PESQ scores obtained for several values of NB.
4. Decide what you consider to be a reasonable number of bits (*NB*) per sample for telephone speech when the sampling rate is 8 kHz with uniform quantisation. Summarise your reasons in one sentence, and note whether your decision is significantly different from the PESQ assessment.
5. You have heard that land-line telephone calls use 64000 bits/second links. Based on your experiments today, do you consider that 8 bits per sample with uniform quantisation may be acceptable for telephone quality speech sampled at 8 kHz?
6. Mobile telephony cannot afford 64000 bits/second, and must use considerably less than 16,000 bits/second. How many bits per sample would be possible using 16000 bits/second with uniform quantisation of speech sampled at 8 kHz? Based on your experiments, do you believe that reasonable quality speech can be encoded in this way for mobile telephony?

Part 1.7 Log-PCM encoded speech [3 marks]

To achieve a bit-rate of 64000, speech must be companded (compressed), uniformly quantised to 8 bits per sample, and then un-companded (expanded) at the receiver. Develop and test a documented Jupyter program to implement these processes and thus demonstrate the advantage of mu-law encoding. Before applying it to speech, plot graphs of compressor output y against input x over a range of about 200 values of x from -1 to 1. Plot a corresponding graph for the expander. Then apply compression and expansion to the speech file '*Narrobandspeech8k.wav'* with samples initially scaled to ±1. Refer to Section 11 for details. Demonstrate the effect of using 8 bits per compressed sample and then reducing the number of bits per sample from 8 to 7 or 6. Record brief answers to the following questions:

1. What do we learn from the mu-law companding and expansion graphs?
2. Compare mu-Law PCM speech at 64000 bit/s with the result of uniform quantisation at the same bit-rate. Give PESQ scores and your own assessments for both.
3. If you have time, compare mu-Law PCM speech with NB=7, 6, etc. with result of uniform quantisation at the same bit-rate (56,000 bit/s, 48,000 bit/s, etc).

13. Submitting your report to Blackboard

Give the notebook a title which begins with your own name, e.g., JSmith_Task1.ipynb. Before

submitting it, check that it definitely runs correctly by clearing all cell output (Cell→All Output→Clear) and then selecting Cell→Run All. Finally, clear all output again (Cell→All Output→Clear) to minimise the size or the notebook. If your notebook uses the sound files and libraries mentioned in this laboratory script, the person who marks the report will have these in his or her own home directory. If, however, you use any additional material, you must attach this to your submission and mention it clearly in the documentation. The style and presentation of your notebook will be an important factor in the marking. It is very important that all the information your code produces (graphs, sound-bars, text etc.) is clearly labelled. If your notebook produces an error for a particular cell, it will run no further, and you may not get any marks for what follows. The deadline for Task 1 is the start of your laboratory session in Week 3.

--- ---

End of Document (Last modified by Barry 28/01/19) A final word: be careful about CASTING!

--- ---