Images as Matrices in Python

This isn't even the worst pun we could have included...

Setup

Before you begin these tasks, make sure you have done the following:

- Installed Python 3.6 or later (if you're reading this in Jupyter Notebook, this is taken care of!)
- Have the Python modules `numpy`, `scipy` and `matplotlib` installed
- Have downloaded the image `Teaching_Python.png`, which looks like:

  and know where it is stored (you can rightclick > Save Image As...). (It will be convenient for you to have the image stored in the same directory as this workbook, so that you don't need relative paths - the notebook assumes you have done this).

Background

Image dimensions are measured in pixels - the image above has dimensions 640×480 pixels (width-height). Each pixel has an associated **Red-Green-Blue (RBG)** triple associated with it, which stores the relative shades of red, green, and blue at that pixel - giving the image color. (By contrast, a greyscale image only has one value per pixel which quantifies how dark that pixel is). As such, any $w \times h$ RGB image can be thought of as a $h \times w \times 3$ image-matrix or array of floating-point numbers, which allows us to manipulate images on a computer. Note that the order of w and h is changed as row-index is stored first in a 2D-array on a computer.

The SVD

The **Singular Value Decomposition (SVD)** of a matrix is a powerful theoretical tool and to a more modest degree, a useful computational tool too. For the purposes of these exercises we will introduce the SVD soley for use in image manipulation, however be aware that you will likely meet this (if you haven't already) at some point in the future and it has much more general interpretations.

Let $A \in \mathbb{R}^{m \times n}$ (a real $m \times n$ matrix), and let $K = \min\{n, m\}$. There exist orthogonal matrices $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and numbers $\sigma_i \in \mathbb{R}, i \in \{1, \ldots, K\}$ such that A has the decomposition

$$A = U\Sigma V^\top$$
$$\Sigma = \operatorname{diag}\left(\sigma_1, \ldots, \sigma_k\right) \in \mathbb{R}^{m \times n}.$$

We call U the matrix of left-singular vectors and V the matrix of right-sigular vectors (these vectors span the image space and domain space of A, respectively) and the σ_i are the singular values. We normally order the σ_i such that $|\sigma_i| \geq |\sigma_{i+1}|$.

Note that Σ is not necessarily square - it can be "*tall*" (has extra rows of zeros) or "*long*" (extra columns of zeros). This decomposition is formally a theorem and has a proof which we are not going to detail, because it's long and would require some theoretical setup that we thought we'd spare you from.

Some interpretation in the context of image-matrices:

- The larger singular values can be thought of as defining "large-scale" features, whereas the smaller singular values define sharp features in the image.
- For an image array, we have a SVD for each of the fixed-RGB slices through the array. I.e: We have a "red SVD", "green SVD" and "blue SVD" for each image.
- Due to the existance of machine precision (the smallest positive number that can be stored on a computer), small singular values may fall near or below this value and a computer cannot reliably work with these values. This is a big problem when trying to invert large systems and is central to the field of *Inverse Problems*.
- If we know the SVD, we know how to construct the image. We can use the SVD to change the image for better or worse if we want.

Exercises

Use the empty code blocks below each task to type your solution. You can execute the current code block using the keyboard shortcut `Ctrl+Enter`, or using the toolbar at the top of the page.

In addition, please run the additional code block below to setup the Notebook. These are the only additional modules you will need to complete these exercises.

In [1]:

```
%matplotlib inline
#This means that matplotlib will display images in this window, rather than in a new one.

import numpy as np #To use functions or constants that are in the NumPy module, type np.function_name. EG: np.pi
is pi
import numpy.random as nprd #This imports random number generation functions
import scipy as sp #SciPy has lots of useful numerical methods, so we don't have to write them
import scipy.linalg as spla #This imports the sub-module containing linear algebra methods
import matplotlib.pyplot as plt #This will let us display images nicely

colours = ['red', 'green', 'blue'] #This is just so that we can label things conveniently
```

1: Setup

Read in the image `Teaching_Python` using `matplotlib`'s `imread` function, calling the output `pythonPic`. This is a $64 \times 64 \times 3$ image-array. Use `matplotlib.pyplot.imshow` and then `matplotlib.pyplot.show` to visualise the image.

NOTE: Read the above codeblock so that you know which names the various modules have been imported under!

Also store the dimensions in an array `dims` using the `np.shape` command.

`Teaching_Python` should look like the following image, possibly rescaled by `matplotlib`:

Solution (./02_python-ws-soln.html#-1%3A-Setup%0A)

2: Playing with Colours

Now that the image is imported, investigate the effect of removing one of the primary (RGB) colours from the image. As `pythonPic` is just an array, this amounts to creating a copy of the image and setting all the pixel-values for one RGB colour to be 0. When you plot the image, make sure it has a title which details which manipulation has taken place (`plt.title` is the function you want, and `colours` defined above may be useful).

Having done this, manipulate `Teaching_Python` so that:

- The top-left quarter of the image is unchanged
- The top-right quarter of the image has red removed
- The bottom-left quarter of the image has green removed
- The bottom-right quarter of the image has blue removed

  Solution (./02_python-ws-soln.html#-2%3A-Playing-with-Colours%0A)

3: (Optional) Colours are Fun

Repeat the previous task, but this time removing *all other* colours, rather than only a single one.

Solution (./02_python-ws-soln.html#-3%3A-%28Optional%29-Colours-are-Fun%0A)

4: The SVD and a Low-Rank Approximation

Write a function `LowRank(A, k)` which takes in a matrix `A` and integer `k`. The function should return the *low-rank approximation* to the matrix `A` by truncating the SVD of `A`; i.e. `LowRank` should calculate the SVD of `A`, then construct the matrix A_k given by:

$$\sigma_i' := 0 \text{ for } i > k,$$

$$\Sigma_k := \operatorname{diag}\left(\sigma_1, \ldots, \sigma_k, \sigma_{k+1}', \ldots, \sigma_k'\right), \text{ and}$$

$$A_k := U \Sigma_k V \in \mathrm{R}^{m \times n}$$

Compute the low-rank approximation to `pythonPic`, for $k = 2, 5, 7, 10, 20$, displaying each approximation. (Display `pythonPic` too, so that you can compare them.)

What are the reasons for wanting a low-rank approximation to an image rather than the full image?

Solution (./02_python-ws-soln.html#-4%3A-The-SVD-and-a-Low-Rank-Approximation%0A)

Image Noise Reduction (Greyscale)

Converting to Greyscale

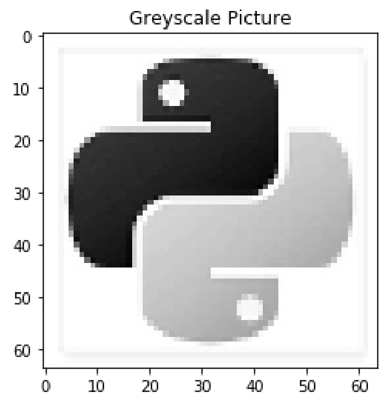
We are now going to look at one of the uses for the SVD and low-rank approximations - filtering out noise from an image. Noise can arise in an image (or indeed, any data type) a number of ways, errors in communication, imprecisions when first storing the information, etc.

To prevent things from getting too complicated we are going to use a greyscale image to do this, so please run the following codeblock. This codeblock will provide you with:

- A new image, `monoPic`, which is a greyscale version of `pythonPic`
- A function, `PlotGreyscale`, which you may use to plot the greyscale images properly. Calling `PlotGreyscale(image)` will just plot the image array `image`; if you give a second string argument such as by `PlotGreyscale(image, 'Snazzy title')`, your plot will also have the title '*Snazzy Title*'. An example call is given in the codeblock.

In [8]:

```
def RGB2Gray(rgb):

    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    grey = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return grey

def PlotGreyscale(img, titleStr=''):
    '''A function so that greyscale images plot correctly.
    Passing a second argument makes a title for the image.
    '''

    plt.title(titleStr)
    plt.imshow(img, cmap=plt.get_cmap('gray')) #Americanism...
    plt.show()

    return

monoPic = RGB2Gray(pythonPic)
PlotGreyscale(monoPic, 'Greyscale Picture')
```



5: Image Noise Generation

Rather than provide you with a deliberately bad image that we've added noise to earlier, we are going to add our own noise to `pythonPic` and then attempt to remove or reduce the noise to recover the original. Adding noise to the greyscale image is as simple as adding IID random numbers to every pixel, IE every pixel in the image has a random sample from $\mathcal{N}\left(0, \xi^2\right)$ added to it. Being the standard deviation of the random numbers, ξ can be thought of as the "*noise level*" - the larger ξ results in a noisier image.

Write a function `AddNoise(img, xi)` which performs the task described above. The input `img` is a greyscale image and `xi` is the standard deviation or noise level to be applied. `AddNoise` should return the noisey image.

TIP: The function `numpy.random.normal` may help. We have also already imported `numpy.random` as `nprd`.

Solution (./02_python-ws-soln.html#-5%3A-Image-Noise-Generation%0A)

6: Noise Reduction

By taking a low-rank SVD approximation to the image, we can remove some of the noisey features that we introduced earlier. Provided the noise level isn't so high as to destroy *all* the features of the original image; the larger features of the image should still be recognisable. As information about large features of an image is *encoded* in the largest singular values and vectors, a low-rank approximation to a noisey image should weed out some of the (small) features introduced by the noise.

Take $\xi = 0.1$, and use `AddNoise` to add this level of noise to `monoPic`, calling the resulting noisey image `fuzzyPic`. Starting at $k = 5$ and increasing in fives up to (and including) $k = 25$:

- Reconstruct the image in the k-low-rank approximations (calling it `resolvedPic`)
- Store the error in the L^2 norm of the low-rank image and the original image, in an array called `errNorms`. That is, obtain the quantity || `monoPic`-`resolvedPic` $||_2$.

TIP: `LowRank` from earlier may be useful, as well as `scipy.linalg.norm`.

Create a plot of k against `errNorms`. What do you notice about the change in image quality as k increases, and around which value of k is the image at it's "*best quality*"?

(Optional)

Experiment with different noise levels, and the effectiveness of this method of noise reduction. How do the conclusions of the previous task depend on ξ?

NOTE: $\xi \geq 1$ will result in this process breaking down, but feel free to experiment with it nonetheless!

Solution (./02_python-ws-soln.html#-6%3A-Noise-Reduction%0A)