

C_svdphoto

February 27, 2023

1 Worksheet C

variationalform <https://variationalform.github.io/>

Just Enough: progress at pace <https://variationalform.github.io/>

<https://github.com/variationalform>

Simon Shaw <https://www.brunel.ac.uk/people/simon-shaw>.

This work is licensed under CC BY-SA 4.0 (Attribution-ShareAlike 4.0 International)

Visit <http://creativecommons.org/licenses/by-sa/4.0/> to see the terms.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

This worksheet is based on the material in the notebooks

- matrices: matrix concepts and algebra
- systems: systems of linear equations, under- and over-determined cases
- decomp: eigensystem decomposition and SVD.

We look at a specific example of using SVD to compress a photograph.

Note that while the ‘lecture’ notebooks are prefixed with 1_, 2_ and so on, to indicate the order in which they should be studied, the worksheets are prefixed with A_, B_, ...

```
[1]: # Two new imports here ... PIL and IPython
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import IPython.display
```

```
[2]: # Use a jpeg photo - ffc.jpg is about 6.2MB (use your own path/filename here)
IPython.display.Image(filename='./gfx/ffc.jpg', width = 150)
```

```
[2]:
```

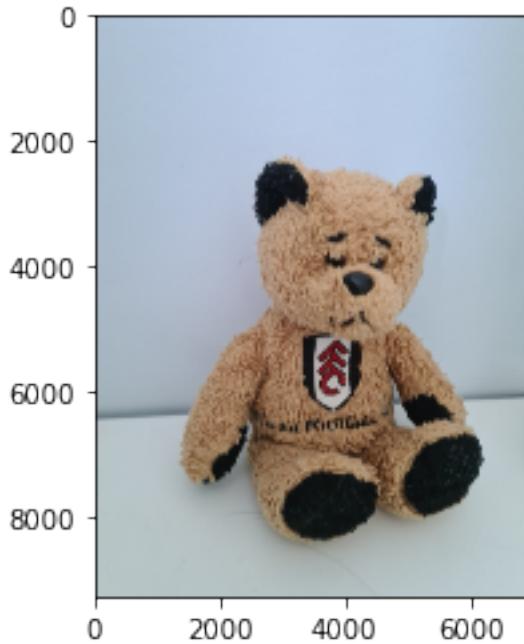


[3]: # that is just a display, so ...
load in the FFC bear - Roy - and visually check him.

```
img = Image.open('./gfx/ffc.jpg')

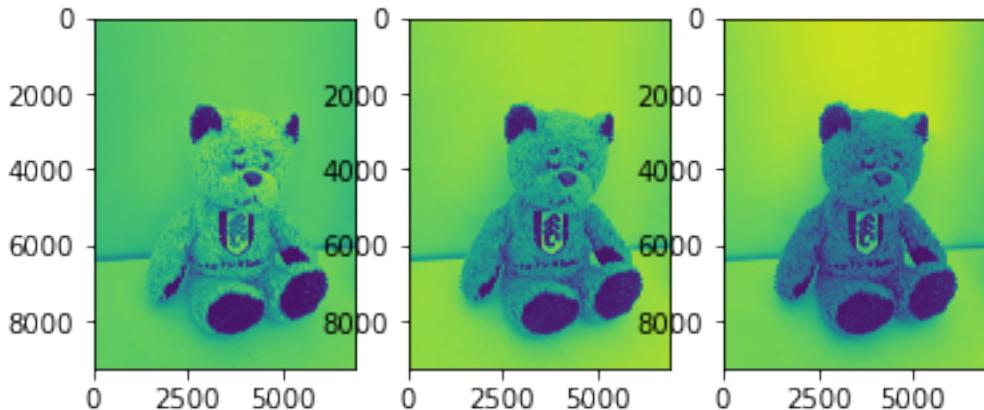
[4]: # convert him to a numpy array for processing as a matrix
a = np.asarray(img)
im_orig = Image.fromarray(a)
plt.imshow(im_orig)

[4]: <matplotlib.image.AxesImage at 0x7fcc60d9f710>
```



This image is made up of pixels where each pixel has a value for RED, GREEN and BLUE. We can get these *colour bands* and show them as follows...

```
[5]: # convert band 'bnd' to a numpy array and show them...
for bnd in range(3):
    plt.subplot(1, 3, 1+bnd)
    img_mat = np.array(list(img.getdata(bnd)), float)
    img_mat = np.matrix(img_mat)
    img_mat.shape = (img.size[1], img.size[0])
    plt.imshow(img_mat)
```



```
[6]: # get the red, green and blue bands as separate objects...
rband = img.getdata(band=0)
gband = img.getdata(band=1)
bband = img.getdata(band=2)

# and convert each to a numpy arrays for maths processing
imgr_mat = np.array(list(rband), float)
imgg_mat = np.array(list(gband), float)
imgb_mat = np.array(list(bband), float)
```

```
[7]: # each of these is about 64k elements
print('sizes = ', imgr_mat.size, imgg_mat.size, imgb_mat.size)
print('shapes = ', imgr_mat.shape, imgg_mat.shape, imgb_mat.shape)
```

```
sizes = 64144128 64144128 64144128
shapes = (64144128,) (64144128,) (64144128,)
```

```
[8]: # get image shape - we can assume they are all the same
imgr_mat.shape = imgg_mat.shape = imgb_mat.shape = (img.size[1], img.size[0])
print('imgr_mat.shape = ', imgr_mat.shape)
print('imgg_mat.shape = ', imgg_mat.shape)
print('imgb_mat.shape = ', imgb_mat.shape)

# convert these 1D-arrays to matrices
imgr_mat1D = np.matrix(imgr_mat)
imgg_mat1D = np.matrix(imgg_mat)
imgb_mat1D = np.matrix(imgb_mat)
print(type(imgb_mat))
```

```
imgr_mat.shape = (9248, 6936)
imgg_mat.shape = (9248, 6936)
imgb_mat.shape = (9248, 6936)
```

```
<class 'numpy.ndarray'>
```

- The chained assignment above is OK but you should be aware that there are pitfalls... Take a look here for example.
- <https://stackoverflow.com/questions/7601823/how-do-chained-assignments-work>
- It depends on what is being chained... Here they are just values and not objects...

```
[9]: print(imgb_mat.shape is imgg_mat.shape)
```

```
False
```

```
[10]: # each of these is about 64 million elements: 9248 by 6936
print('sizes = ', imgr_mat1D.size, imgg_mat1D.size, imgb_mat1D.size)
print('shapes = ', imgr_mat1D.shape, imgg_mat1D.shape, imgb_mat1D.shape)
print('check: 9248 x 6936 = ', 9248 * 6936)
```

```
sizes = 64144128 64144128 64144128
shapes = (9248, 6936) (9248, 6936) (9248, 6936)
check: 9248 x 6936 = 64144128
```

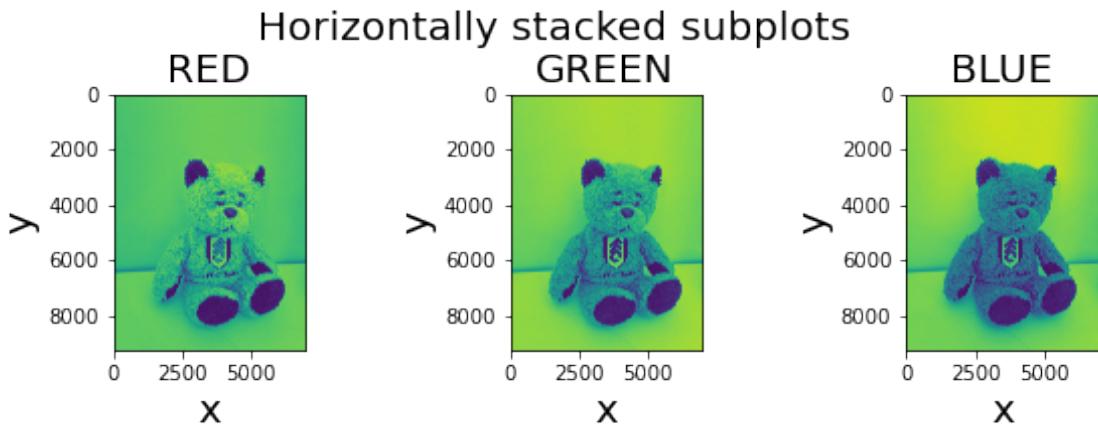
Let's look at these matrices - the following code introduces subplot

```
[11]: fontsize=20
fig=plt.figure(figsize=(8, 3))
fig.suptitle('Horizontally stacked subplots', fontsize=fontsize)

axs=fig.add_subplot(1,3,1); axs.imshow(imgr_mat)
axs.set_xlabel('x', fontsize=fontsize); axs.set_ylabel('y', fontsize=fontsize)
axs.set_title('RED', fontsize=fontsize)

axs=fig.add_subplot(1,3,2); axs.imshow(imgg_mat)
axs.set_xlabel('x', fontsize=fontsize); axs.set_ylabel('y', fontsize=fontsize)
axs.set_title('GREEN', fontsize=fontsize)

axs=fig.add_subplot(1,3,3); axs.imshow(imgb_mat)
axs.set_xlabel('x', fontsize=fontsize); axs.set_ylabel('y', fontsize=fontsize)
axs.set_title('BLUE', fontsize=fontsize)
# use fractions of fontsize
plt.tight_layout(pad=0.3, w_pad=2.5, h_pad=0.3); plt.show()
```



Let's look at how much memory these photo layers occupy...

```
[12]: print('Each matrix contains ', imgr_mat.size, ' elements')
print('Each element occupies ', imgr_mat.itemsize, ' bytes')
print('So each matrix occupies ', imgr_mat.size * imgg_mat.itemsize, ' bytes in '
     'memory')
print('This is ', 3*imgr_mat.size * imgg_mat.itemsize, ' bytes for all three')

print('This is {:.e} bytes for all three'.format(3*imgr_mat.size * imgg_mat.
     itemsize))
```

```
Each matrix contains 64144128 elements
Each element occupies 8 bytes
So each matrix occupies 513153024 bytes in memory
This is 1539459072 bytes for all three
This is 1.539459e+09 bytes for all three
```

1.2 Compression...

Now that we know about the **Singular Value Decomposition**, we can hope to compress these objects using **mathematics**.

First get the SVD's of the R, G and B layers... (takes a while)

```
[13]: Ur, Sr, VTr = np.linalg.svd(imgr_mat)
Ug, Sg, VTg = np.linalg.svd(imgg_mat)
Ub, Sb, VTb = np.linalg.svd(imgb_mat)
```

```
[14]: print(f'RED: shapes of Ur, Sr, VTr = {Ur.shape}, {Sr.shape}, {VTr.shape}')
print(f'GREEN: shapes of Ug, Sg, VTg = {Ug.shape}, {Sg.shape}, {VTg.shape}')
print(f'BLUE: shapes of Ub, Sb, VTb = {Ub.shape}, {Sb.shape}, {VTb.shape}')
```

```
RED: shapes of Ur, Sr, VTr = (9248, 9248), (6936,), (6936, 6936)
```

```
GREEN: shapes of Ug, Sg, VTg = (9248, 9248), (6936,), (6936, 6936)
BLUE: shapes of Ub, Sb, VTb = (9248, 9248), (6936,), (6936, 6936)
```

```
[15]: # choose the number of components to use in the reconstruction
nc = 5 # 1387
rec_imgr = np.matrix(Ur[:, :nc]) * np.diag(Sr[:nc]) * np.matrix(VTr[:nc, :])
rec_imgg = np.matrix(Ug[:, :nc]) * np.diag(Sg[:nc]) * np.matrix(VTg[:nc, :])
rec_imgb = np.matrix(Ub[:, :nc]) * np.diag(Sb[:nc]) * np.matrix(VTb[:nc, :])

img_all = np.array([rec_imgr, rec_imgg, rec_imgb]).T
img_all = np.swapaxes(img_all, 0, 1)

PIL_image = Image.fromarray(np.uint8(img_all)).convert('RGB')
# uncomment this to spawn an external viewer
#PIL_image.show()
# save the reconstruction
PIL_image.save("ffc_recon.jpg")
```

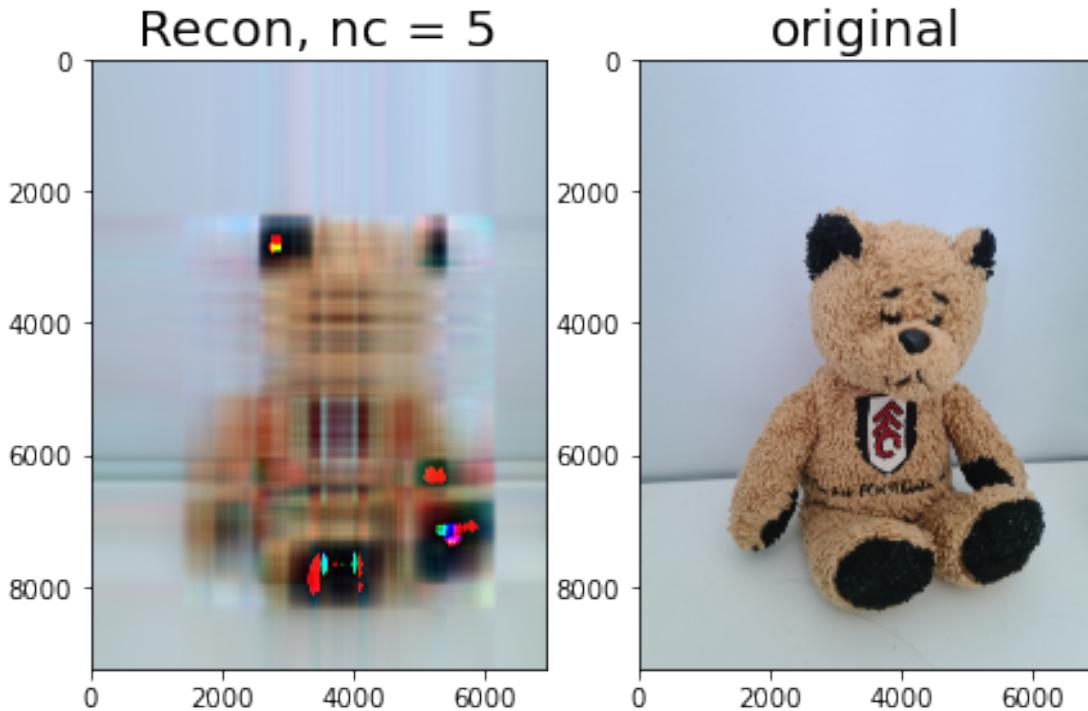
```
[16]: fig=plt.figure(figsize=(7, 5))
fig.suptitle('Comparison', fontsize=30)

plt.subplot(1,2,1)
ax = plt.gca()
im = Image.fromarray(np.uint8(img_all)).convert('RGB')
ax.imshow(im)
ax.set_title(f'Recon, nc = {nc}', fontsize=20)

plt.subplot(1,2,2)
ax = plt.gca()
ax.imshow(im_orig)
ax.set_title('original', fontsize=20)
```

```
[16]: Text(0.5, 1.0, 'original')
```

Comparison



1.3 Review

We have seen a few examples now of how the SVD can play a very important role in Data Science. It is able to take data in matrix-form and distill it to its *essence*.

2 Exercises

1. Create logarithmic scree plots for each of the colour bands - what value of `nc` do these suggest?
2. Try to create non-log scree plots for each of the colour bands - what happens?
3. Calculate the percentage in memory saving resulting from the chosen value of `nc`
4. Look at each ‘photo’ corresponding to each singular component. Can you see the original coming through?
5. Overlay the first few modes for each colour. What can you see now?

[17]: `# Start solution 1 here`

[18]: `# Start solution 3 here`

[19]: `# Start solution 3 here`

[20]: `# Start solution 4 here`

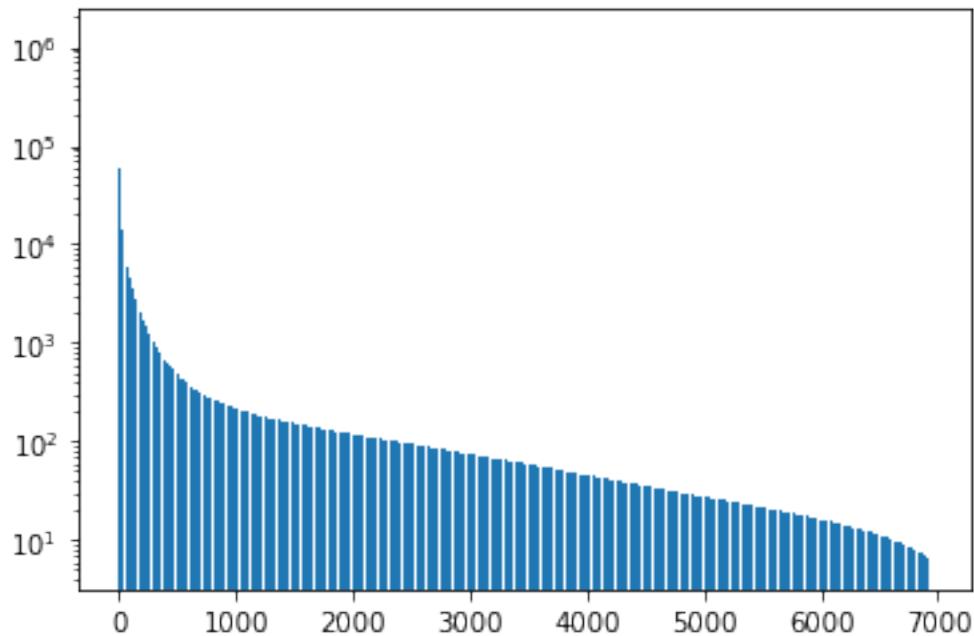
```
[21]: # Start solution 5 here
```

3 Outline Solutions

1. For the scree plots - here are the red ones.

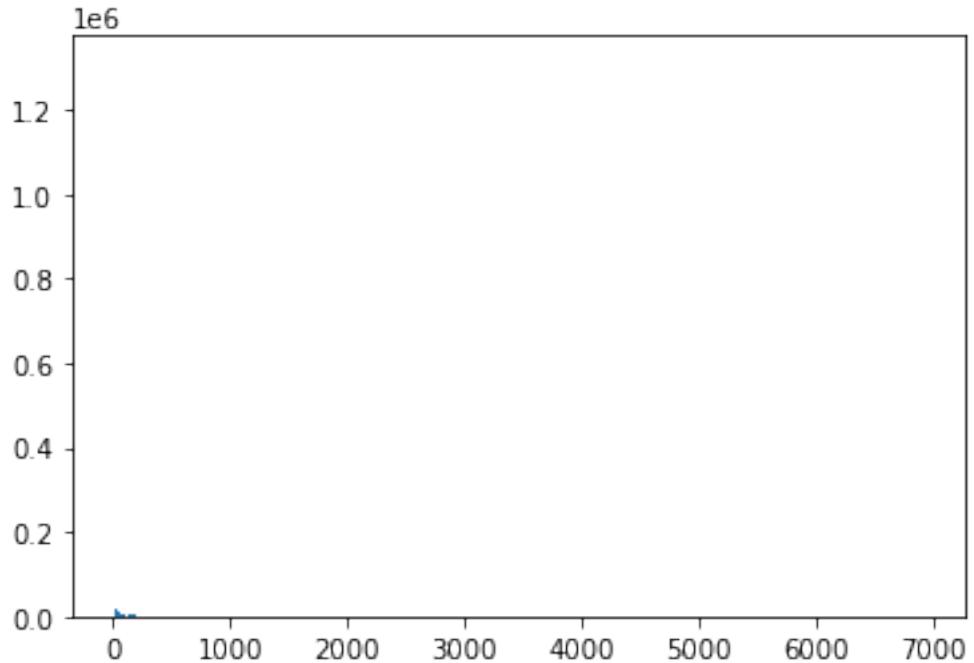
```
[22]: plt.bar(range(Sr.shape[0]),Sr, log=True)
```

```
[22]: <BarContainer object of 6936 artists>
```



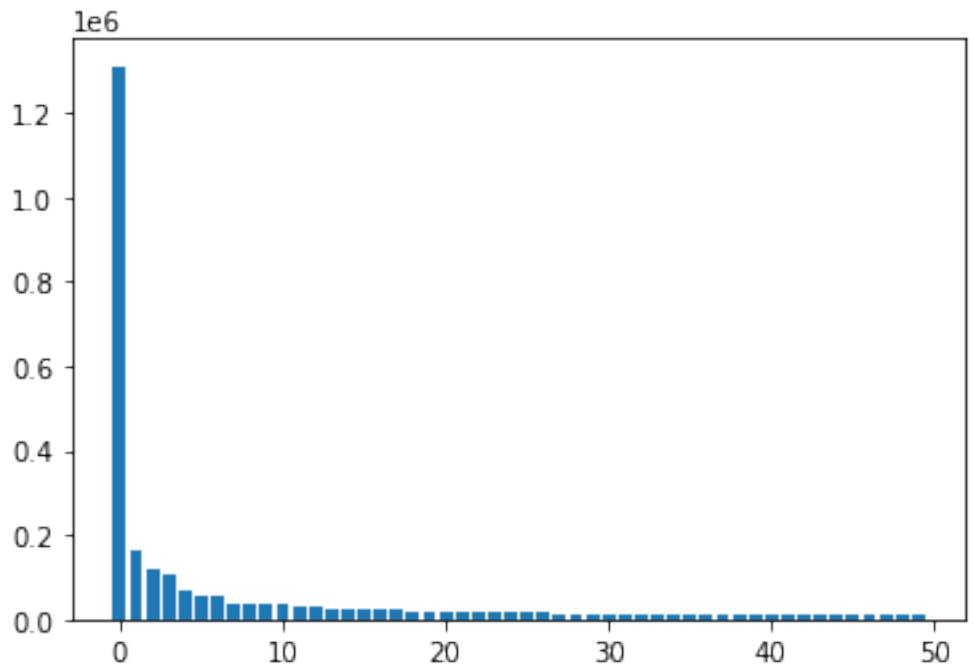
```
[23]: # this fails - there are too many and each bar is too thin to show
# plt.figure(figsize=(12, 12)) # you can try a bigger plot, doesn't help
plt.bar(range(Sr.shape[0]),Sr)
```

```
[23]: <BarContainer object of 6936 artists>
```



```
[24]: plt.bar(range(50),Sr[:50])
```

```
[24]: <BarContainer object of 50 artists>
```



2. The memory ratio is the new size of the first `nc` components divided by the size of the original

```
[25]: print('Each matrix contains ', img_r_mat.size, ' elements')
print('Each element occupies ', img_r_mat.itemsize, ' bytes')
print('So each matrix occupies ', img_r_mat.size * img_g_mat.itemsize, ' bytes in_
→memory')

num_bytes_orig = 3*img_r_mat.size * img_g_mat.itemsize
print(f'This is {num_bytes_orig} bytes for all three')
```

```
Each matrix contains 64144128 elements
Each element occupies 8 bytes
So each matrix occupies 513153024 bytes in memory
This is 1539459072 bytes for all three
```

For `nc` components, for each colour, we had this:

```
rec_imgr = np.matrix(Ur[:, :nc]) * np.diag(Sr[:nc]) * np.matrix(VTr[:nc, :])
```

So, for this colour we need `nc` columns in `Ur`, `nc` scalars in `Sr` and `nc` rows in `VTr`. And the same for each of the other colours.

```
[26]: print(f'RED: shapes of Ur, Sr, VTr = {Ur.shape}, {Sr.shape}, {VTr.shape}')
num_bytes_recon = 3*nc*(Ur.shape[1] + 1 + VTr.shape[0])*Ur.itemsize
print(f'This is {num_bytes_recon} bytes for all three')
```

```
RED: shapes of Ur, Sr, VTr = (9248, 9248), (6936,), (6936, 6936)
This is 1942200 bytes for all three
```

```
[27]: # The percentage memory savings ratio is then...
print(f'percentage savings in bytes = {100*num_bytes_recon/num_bytes_orig} %')
```

```
percentage savings in bytes = 0.12616119748326768 %
```

3. This is a complicated bit of code. There is a lot of value in understanding it

The graphics are unlikely to fit on the slide - use the notebook

```
[ ]: # cell toggle scrolling from menu to prevent small window and scroll bar
fontsize=20; fig=plt.figure(figsize=(15, 15))
fig.suptitle('First Few Modes', fontsize=fontsize)

c = 0; max_m = 5; step_m = 2
for m in range(0,max_m*step_m, step_m):
    n = m+1
    c += 1
    mode_imgr = np.matrix(Ur[:, m:n]) * np.diag(Sr[m:n]) * np.matrix(VTr[m:n, :])
    axs=fig.add_subplot(max_m,3,c)
    axs.imshow(mode_imgr)
    axs.set_title('red, mode = '+str(m), fontsize=fontsize)
    c += 1
```

```

mode_imgg = np.matrix(Ug[:, m:n]) * np.diag(Sg[m:n]) * np.matrix(VTg[m:n, :])
axs=fig.add_subplot(max_m,3,c)
axs.imshow(mode_imgg)
axs.set_title('green, mode = '+str(m), fontsize=fontsize)
c += 1
mode_imgb = np.matrix(Ub[:, m:n]) * np.diag(Sb[m:n]) * np.matrix(VTb[m:n, :])
axs=fig.add_subplot(max_m,3,c)
axs.imshow(mode_imgb)

axs.set_title('blue, mode = '+str(m), fontsize=fontsize)

```

4. This code is also of value.

The graphics are unlikely to fit on the slide - use the notebook

```
[ ]: # cell toggle scrolling from menu to prevent small window and scroll bar
fontsize=20
fig=plt.figure(figsize=(15,7))
fig.suptitle('First Few Modes Combined', fontsize=fontsize)

m = 0; n = max_m*step_m
c = 1
mode_imgr = np.matrix(Ur[:, m:n]) * np.diag(Sr[m:n]) * np.matrix(VTr[m:n, :])
axs=fig.add_subplot(1,3,c)
axs.imshow(mode_imgr)
axs.set_title('red, modes up to = '+str(n), fontsize=fontsize)
c += 1
mode_imgg = np.matrix(Ug[:, m:n]) * np.diag(Sg[m:n]) * np.matrix(VTg[m:n, :])
axs=fig.add_subplot(1,3,c)
axs.imshow(mode_imgg)
axs.set_title('green, modes up to = '+str(n), fontsize=fontsize)
c += 1
mode_imgb = np.matrix(Ub[:, m:n]) * np.diag(Sb[m:n]) * np.matrix(VTb[m:n, :])
axs=fig.add_subplot(1,3,c)
axs.imshow(mode_imgb)
axs.set_title('blue, modes up to = '+str(n), fontsize=fontsize)
```

3.1 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
 - jupyter nbconvert --to slides C_svdphoto.ipynb
- Set OUTPUTTING=1 below

- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set OUTPUTTING=0 below
- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
 - git add, commit and push
- rebuild binder

Some of this originated from

<https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-notebook>

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

```
[ ]: %%bash
NBROOTNAME='C_svdphoto'
OUTPUTTING=1

if [ $OUTPUTTING -eq 1 ]; then
    jupyter nbconvert --to html $NBROOTNAME.ipynb
    cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
    mv -f $NBROOTNAME.html ./formats/html/

    jupyter nbconvert --to pdf $NBROOTNAME.ipynb
    cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
    mv -f $NBROOTNAME.pdf ./formats/pdf/

    jupyter nbconvert --to script $NBROOTNAME.ipynb
    cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
    mv -f $NBROOTNAME.py ./formats/py/
else
    echo 'Not Generating html, pdf and py output versions'
fi
```