1. Introduction

This course assumes some familiarity with Python, Jupyter notebooks and python scientific packages such as Numpy. There are many great resources to learn Python, including within Jupyter environements. For example this (https://gitlab.erc.monash.edu.au/andrease/Python4Maths/tree/master/Intro-to-Python) is a great introduction that you can follow to refresh your memories if needed.

The course will mostly focus on image processing using the package scikit-image, which is 1) easy to install, 2) offers a huge choice of image processing functions and 3) has a simple syntax. Other tools that you may want to explore are OpenCV (https://opencv.org/) (focus on computer vision) and ITK (https://itkpythonpackage.readthedocs.io/en/latest/) (focus on medical image processing). Finally, it has recently become possible to "import" Fiji (ImageJ) (https://github.com/imagej/pyimagej) into Jupyter, which may be of interest if you rely on specific plugins that are not implemented in Python (this is however in very beta mode).

1.1 Installation

1.1.1 Running the course material remotely

To avoid loosing time at the beginning of the course with faulty installations, we provide every attendee access to a JupyterHub allowing to remotely run the notebooks (links will be provided in time). This possibility is only offered for the duration of the course. The notebooks can however be permanently accessed and executed through the mybinder (https://mybinder.org/) service that you can activate by clicking on the badge below that is also present on the repository. If you want to "full experience" you can also install all the necessary packages on your own computer (see below).

launch binder

(https://mybinder.org/v2/gh/guiwitz/PyImageCourse/master)

1.1.2 Local installation

Python and Jupyter can be installed on any operating system. Instead of manually installing all needed components, we highly recommend using the environment manager conda (https://conda.io/docs/user-guide/index.html) by installing either Anaconda or Miniconda (https://conda.io/docs/user-guide/install/index.html#) (follow instructions on the website). This will install Python, Python tools (e.g. pip), several important libraries (including e.g. Numpy) and finally the conda tool itself. For Mac/Linux users: Anaconda is quite big so we recommend installing Miniconda, and then installing additional packages that you need from the Terminal. For Windows users: Anaconda might be better for you as it installs a command prompt (Anaconda prompt) from which you can easily issue conda commands.

The point of using conda is that it lets you install various packages and even versions of Python within closed environments that don't interfere with each other. In such a way, once you have an environment that functions as intended, you don't have to fear messing it up when you need to install other tools for you next project.

Once conda is installed, you should create a conda environment for the course. We have automated this process and you can simply follow the instructions below:

- Clone or download (https://github.com/guiwitz/PyImageCourse/archive/master.zip) and unzip this repository.
- Open a terminal and `cd` to it.
- Create the conda environment by typing:

```
conda env create -f binder/environment.yml
```

- Activate the environment:

```
conda activate improc_env
```

- Several imaging datasets are used during the course. The download of these data is automated through the following command (the total size is 6Gb so make sure you have a good internet connection and enough disk space):

```
python installation/download_data.py
```

Note that if you need an additional package for that environment, you can still install it using conda or pip. **To make it accessible within the course environment don't forget to type:**

```
conda activate improc_env
```

before you conda or pip install anything. Alternatively you can type your instructions directly from a notebook e.g.:

```
! pip install mypackage
```

Whenever you close the terminal where notebooks are running, don't forget to first activate the environment before you want to run the notebooks next time:

```
conda activate improc_env
```

1.2 Some Python refresh

I give here a **very** short summary of basic Python, focusing on structures and operations that we will use during this lecture. So this is **not** an exhaustive Python introduction. There are many many operations that one can do on basic Python structures, however as we are mostly going to use Numpy arrays, those operations are **not** desribed here.

1.2.1 Variables and structures

There are multiple types of Python variables:

In [56]:
```
myint = 4
myfloat = 4.0
mystring ='Hello'
print(myint)
print(myfloat)
print(mystring)
```

```
4
4.0
Hello
```

The type of your variable can be found using type():

In [57]:
```
type(myint)
```

Out[57]: int

In [58]:
```
type(myfloat)
```

Out[58]: float

These variables can be assembled into various Python structures:

In [59]:
```
mylist = [7,5,9]
mydictionary = {'element1': 1, 'element2': 2}
print(mylist)
print(mydictionary)
```

```
[7, 5, 9]
{'element2': 2, 'element1': 1}
```

Elements of those structures can be accessed through **zero-based** indexing:

In [60]:
```
mylist[1]
```

Out[60]: 5

In [61]:
```
mydictionary['element2']
```

Out[61]: 2

One can append elements to a list:

In [62]:
```
mylist.append(1)
print(mylist)
```

```
[7, 5, 9, 1]
```

Measure its length:

In [63]:
```
len(mylist)
```

Out[63]: 4

Ask if some value exists in a list:

In [64]:
```
5 in mylist
```

Out[64]: True

In [65]:
```
4 in mylist
```

Out[65]: False

1.2.2 Basic operations

A lot of operations are included by default in Python. You can do arithmetic:

In [66]:
```
a = 2
b = 3
#addition
print(a+b)
#multiplication
print(a*b)
#powers
print(a**2)
```

```
5
6
4
```

Logical operations returning booleans (True/False)

In [67]:
```
a>b
```

Out[67]: False

In [68]:
```
a<b
```

Out[68]: True

In [69]:
```
a<b and 2*a<b
```

Out[69]: False

In [70]:
```
a<b and 1.4*a<b
```

Out[70]: True

In [71]:
```
a<b or 2*a<b
```

Out[71]: True

Operations on strings:

In [72]:
```
mystring = 'This is my string'
mystring
```

Out[72]: 'This is my string'

In [73]:
```
mystring+ ' and an additional string'
```

Out[73]:
```
'This is my string and an additional string'
```

In [74]:
```
mystring.split()
```

Out[74]:
```
['This', 'is', 'my', 'string']
```

1.2.2 Functions and methods

In Python one can get information or modify any object using either functions or methods. We have already seen a few examples above. For example when we asked for the length of a list we used the len() function:

In [75]:
```
len(mylist)
```

Out[75]:
```
4
```

Python variables also have so-called methods, which are functions associated with particular object types. Those methods are written as variable.method(). For example we have seen above how to append an element to a list:

In [76]:
```
mylist.append(20)
print(mylist)
```

```
[7, 5, 9, 1, 20]
```

The two examples above involve only one argument, but any number can be used. All Python objects, inculding those created by other packages like Numpy function on the same scheme.

There are two ways to ask for help on funtions and methods. First, if you want to know how a specific function is supposed to work you can simply type:

In [77]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

This shows you that you can pass any container to the function len() (list, dictionary *etc.*) and it tells you what comes out. We will see later some more advanced examples of help information.

Second, if you want to know what methods are associated with a particular object you can just type:

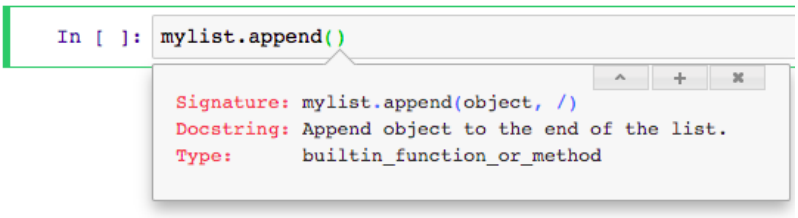
In [111]:
```
#¼dir(mylist)
```

This returns a list of all possible methods. At the moment, only consider those **not** starting with an underscore. If you need help on one of those methods, you can type

In [79]:
```
help(mylist.append)
```

```
Help on built-in function append:

append(...) method of builtins.list instance
    L.append(object) -> None -- append object to end
```

Finally, whenever writing a function you can place the cursor in the empty function parenthesis and hit Command+Shift which will open a window with the help information looking like this:



1.2.2 For, if

Loops and conditions are classical programming features. In python, one can write them in a very natural way. A for loop:

In [80]:
```
for i in [1,2,3,4]:
    print(i)
```

```
1
2
3
4
```

An if condition:

In [81]:
```
a=5
if a>6:
    print('large')
else:
    print('small')
```

```
small
```

A mix of those:

In [82]:
```
for i in [1,2,3,4]:
    if i>3:
        print(i)
```

```
4
```

Note that **indentation of blocks is crucial in Python.**

1.2.3. Mixing lists, for's and if's

A very useful feature of Python is the very simple way it allows one to create lists. For exampel to create a list containing squares of certain values, in a classical programming languange one would do something like:

In [83]:
```
my_initial_list = [1,2,3,4]

my_list_to_create = []#initialize list

for i in my_initial_list:
    my_list_to_create.append(i*i)
print(my_list_to_create)
```

```
[1, 4, 9, 16]
```

Python allows one to do that in one line through a comprehension list, which is basically a compressed for loop:

In [84]:
```
[i*i for i in my_initial_list]
```

Out[84]:
```
[1, 4, 9, 16]
```

In a lot of cases, the list that the for loop goes through is not an explicit list but another function, typically range() which generate either numbers from 0 to N (range(N)) or from M to N in steps of P (range(M,N,P)):

In [85]:
```
[i for i in range(10)]
```

Out[85]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [86]:
```
[i for i in range(0,10,2)]
```

Out[86]:
```
[0, 2, 4, 6, 8]
```

If statements can be introduced in comprehension lists:

In [87]:
```
[i for i in range(0,10,2) if i>3]
```

Out[87]:
```
[4, 6, 8]
```

In [88]:
```
[i if i>3 else 100 for i in range(0,10,2)]
```

Out[88]:
```
[100, 100, 4, 6, 8]
```

A last very useful trick offered by Python is the function enumerate. Often when traversing a list, one needs both the actual value and the index of that value:

In [89]:
```
for ind, val in enumerate([8,4,9]):
    print('index: '+str(ind))
    print('value: ' + str(val))
```

```
index: 0
value: 8
index: 1
value: 4
index: 2
value: 9
```

1.2.4 Using packages

Python comes with a default set of data structures and operations. For particular applications like matrix calculations (image processing) or visulaization, we are going to need additional resources. Those exist in the form of python packages, ensembles of functions and data structures whose defintiions can be simply imported in any Python program.

For example to do matrix operations, we are going to use Numpy, so we run:

In [90]:
```
import numpy
```

All functions of a package can be called by using the package name followed by a dot and a parenthesis `numpy.xxx()`. Most functions are used with an argument and either "act" on the argument e.g. to find the maximum in a list:

In [91]:
```
numpy.max([1,2])
```

Out[91]:
```
2
```

or use the arguments to create a new object e.g. a 4x3 matrix of zeros:

In [92]:
```
mymat = numpy.zeros((4,3))
```

In [93]:
```
mymat
```

Out[93]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

To avoid lengthy typing, package names are usually abbreviated by giving them another name when loading them:

In [94]:
```
import numpy as np
```

Within packages, some additional tools are grouped as submodules and are typically called e.g for numpy as `numpy.submodule_name.xxx()`. For example, generating random numbers can be done using the numpy.random submodule. An array of ten uniform random numbers can be for example generated using:

In [95]:
```
np.random.rand(10)
```

Out[95]:
```
array([0.00738174, 0.82510957, 0.59643586, 0.92919436, 0.46570716,
       0.92526076, 0.17081481, 0.03715798, 0.12744829, 0.35009797])
```

To avoid lengthy typing, specific functions can be directly imported, which allows one to call them without specifying their source module:

In [96]:
```
from numpy.random import rand

rand(10)
```

Out[96]:
```
array([0.81812159, 0.97452756, 0.4383594 , 0.91854004, 0.37517642,
       0.11077294, 0.66271078, 0.8482131 , 0.70100188, 0.44337187])
```

This should be used very cautiously, as it makes it more difficult to debgug code, once it is not clear anymore that a given function comes from a module.

1.3 Matplotlib

To quickly look at images, we are mostly going to use the package Matplotlib. We review here the bare minimum function calls needed to do a simple plot. First let's import the pyplot submodule:

```
In [97]: import matplotlib.pyplot as plt
```

1.3.1 Plotting images

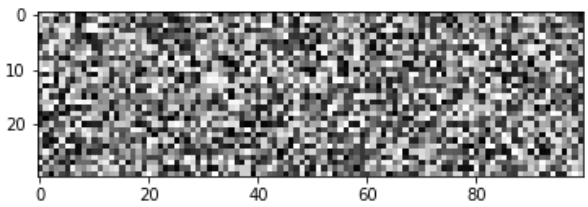
Using numpy we create a random 2D image of integers of 30x100 pixels (we will learn more about Numpy in the next chapters):

```
In [98]: image = numpy.random.randint(0,255,(30,100))
```

The variable image is a Numpy array, and we'll see in the next chapter what that exactly is. For the moment just consider it as a 2D image.

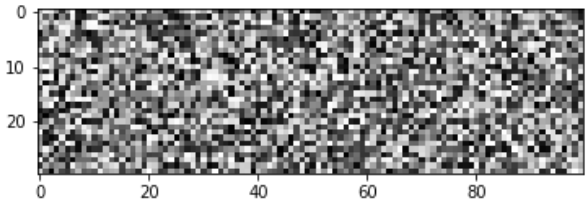
To show this image we are using the `plt.imshow()` command which takes an Numpy array as argument:

```
In [99]: plt.imshow(image)
Out[99]: <matplotlib.image.AxesImage at 0x7f7496c27160>
```



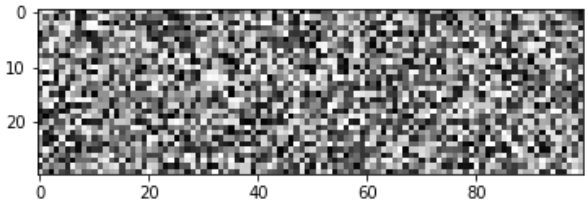
In order to suppress the matplotlib figure reference, you can end the line with `;`:

```
In [100]: plt.imshow(image);
```



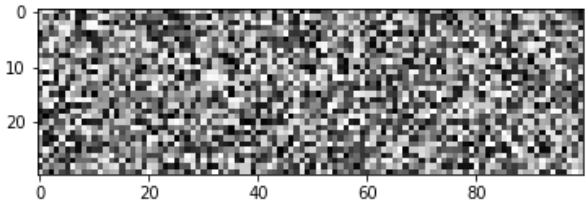
When plotting outside of an interactive environment like a notebook you will also have to use the show() command. If you use it in a notebook you won't have to use `;`:

In [101]:
```
plt.imshow(image)
plt.show()
```



The rows and number indices are indicates on the left and the bottom and **actually** correspond to pixel indices. The image is just a gray-scale image, and Matplotlib used its default lookup table (or color map) to color it (LUT in Fiji). We can change that by specifiy another LUT (you can find the list of LUTs here (https://matplotlib.org/examples/color/colormaps_reference.html) by using the argument cmap (color map):

In [102]:
```
plt.imshow(image, cmap = 'gray');
```



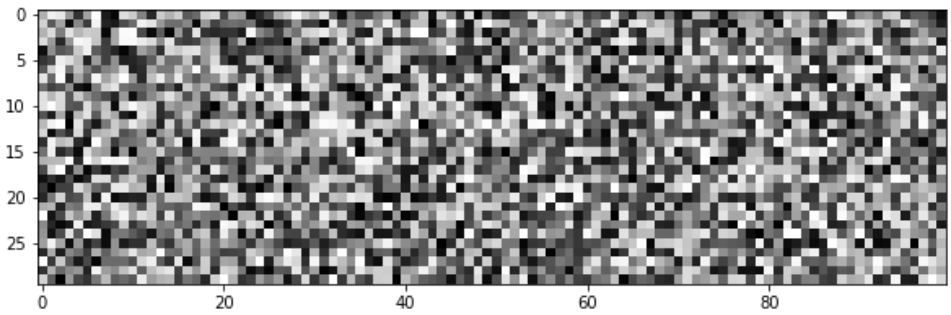
Note that you can change the default color map used by matplotlib using a command of the type plt.yourcolor, *e.g.* for gray scale:

In [103]:
```
plt.gray()
```

```
<Figure size 432x288 with 0 Axes>
```

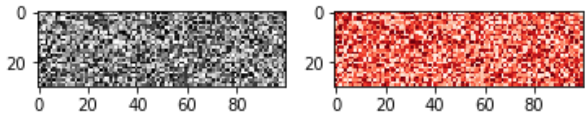
Sometimes we want to see a slightly larger image. To do that we have to add another line that specifies options for the figure.

In [104]:
```
plt.figure(figsize=(10,10))
plt.imshow(image);
```



Sometimes we want to show an array of figures to compare for example an original image and its segmentations. We use the subplot() function and pass three arguments: number of rows, number of columns and index of plot. We use it for each element and increment the plot index. There are multiple ways of creating complex figures and you can refer to the Matplotlib documentation for further information:

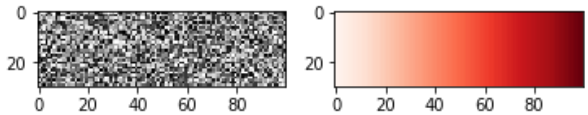
In [105]:
```
plt.subplot(1,2,1)
plt.imshow(image, cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image, cmap = 'Reds');
```



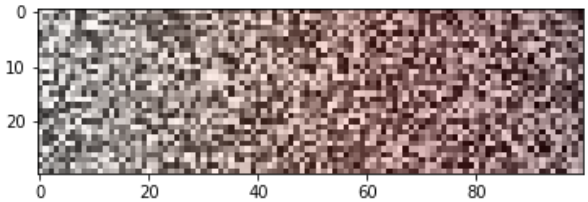
The imshow() function takes basically two types of data. Either single planes as above, or images with three planes. In the latter case, imshow() assumes that the image is in RGB format (Red, Green, Blue) and uses those colors.

Finally, one can superpose various plot elements on top of each other. One very useful option in the frame of this course, is the possibility to ovelay an image in transparency on top of another using the `alpha` argument. We create a gradient image and then superpose it:

In [106]:
```
image_grad = np.ones((30,100))*np.linspace(0, 1, 100)[None, :]

plt.subplot(1,2,1)
plt.imshow(image, cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image_grad, cmap = 'Reds');
```



In [107]:
```
plt.imshow(image, cmap = 'gray')
plt.imshow(image_grad, cmap = 'Reds', alpha = 0.2);
```



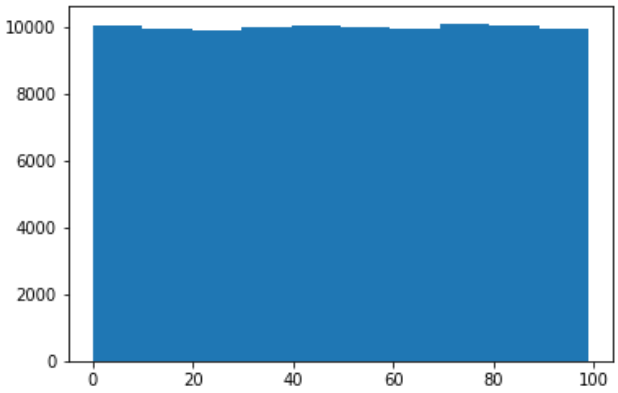
1.3.2 Plotting histograms

One thing that we are going to do very often is looking at histograms, typically of pixel values, for example to determine a threshold from background to signal. For that we can use the plt.hist() command.

If we have a list of numbers we can simply called the `plt.hist()` function on it (we will see more options later). We crate again a list of random numbers:

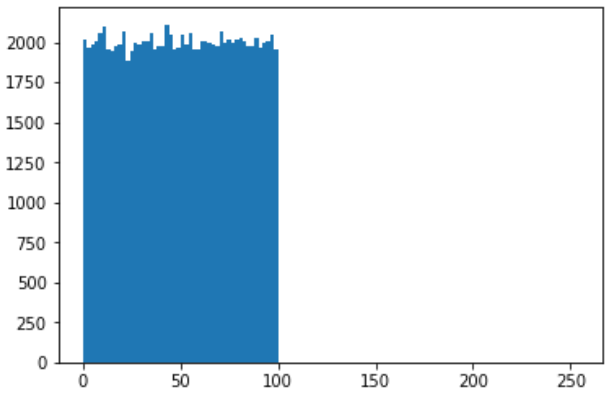
In [108]:
```
list_number = np.random.randint(0,100,100000)
```

In [109]:
```
plt.hist(list_number);
```



Once we have an idea of the distribution of values, we can refine the binning:

In [110]:
```
plt.hist(list_number, bins = np.arange(0,255,2));
```



2. Numpy with images

All images are essentially matrices with a variable number of dimensions where each element represents the value of one pixel. The different dimensions and the pixel values can have very different meanings depending on the type of image considered, but the structure is the same.

Python does not allow by default to gracefully handle multi-dimensional data. In particular it is not desgined to handle matrix operations. Numpy was developed to fill in this blank and offers a very similar framework as the one offered by Matlab. It is underlying a large number of packages and has become abolsutely essential to Python scientific programming. In particular it underlies the functions of scikit-image. The latter in turn forms the basis of other software like CellProfiler. It is thus essential to have a good understanding of Numpy to proceed.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lense of image processing in order to focus on the most useful features in the context of this course.

2.1 Exploring an image

Some test images are provided directly in skimage, so let us look at one (we'll deal with the details of image import later). First let us import the necessary packages.

In [1]:
```
import numpy as np
import skimage
import matplotlib.pyplot as plt
plt.gray();  # MZ: nsure it will use gray scale for the plotting
```

In [2]:
```
image = skimage.data.coins()

# submodule skimage.data => provide images
```

In [3]:
```
# MZ: added to have all outputs
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
a=5
a
b=2
b
# => will print 5 and 2 and not only 2
```

Out[3]: 2

2.1.1 Image size

The first thing we can do with the image is simply look at the output:

In [4]:
```
image # MZ: it is a numpy arrray
```

Out[4]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

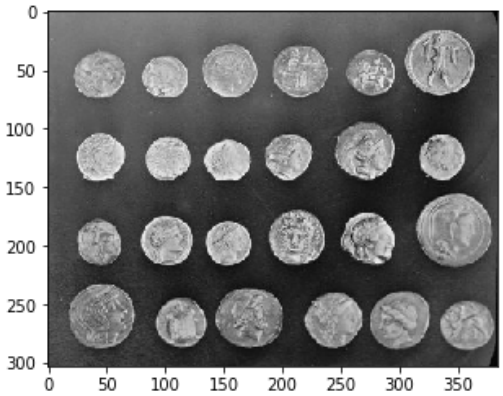
We see that Numpy tells us we have an `array` and we don't have a simple list of pixels, but a *list of lists* representing the fact that we are dealing with a two-dimensional object. Each list represents one row of pixels. Numpy smartly only shows us the first/last rows/columns. We can use the `.shape` method to check the size of the array:

In [5]:
```
image.shape  # MZ: give the dimension
```

Out[5]:
```
(303, 384)
```

This means that we have an image of 303 rows and 384 columns. We can also visualize the image using matplotlib:

In [6]:
```
plt.imshow(image);
```



In [7]:
```
%matplotlib inline
# %matplotlib notebook
# with notebook -> you can zoom, convenient for notebook
# MZ: magic lines for jupyter with %
```

2.1.2 Image type

In [8]:
```
image
```

Out[8]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

In the output above we see that we have one additional piece of information: the array has `dtype = uint8` , which means that the image is of type *unsigned integer 8 bit*. We can also get the type of an array by using:

In [9]:
```
image.dtype  # MZ: dtype is an attribute of "image" (// shape)
```

Out[9]:
```
dtype('uint8')
```

Standard formats we are going to see are 8bit (uint8), 16bit (uint16) and non-integers (usually float64). The type of the image pixels set what values they can take. For example 8bit means values from 0 to $2^8 - 1 = 256 - 1 = 255$. Just like for example in Fiji, one cane change the type of the image. If we know we are going to do operations requiring non-integers we can turn the pixels into floats trough the `.astype()` function.

In [10]:
```
# MZ:
# a bit more careful with types of images !
# if integer or not it really matters !
# numpy different from Python philosophy and dynamic typing
# be careful, e.g. if values > 255 -> can behave weird
```

In [11]:
```
image_float = image.astype(float)
```

Notice the '.':

In [12]:
```
image_float
```

Out[12]:
```
array([[ 47., 123., 133., ...,  14.,   3.,  12.],
       [ 93., 144., 145., ...,  12.,   7.,   7.],
       [126., 147., 143., ...,   2.,  13.,   3.],
       ...,
       [ 81.,  79.,  74., ...,   6.,   4.,   7.],
       [ 88.,  82.,  74., ...,   5.,   7.,   8.],
       [ 91.,  79.,  68., ...,   4.,  10.,   7.]])
```

In [13]:
```
image_float.dtype
```

Out[13]:
```
dtype('float64')
```

The importance of the image type goes slightly against Python's philosophy of dynamics typing (no need to specify a type when creating a variable), but a necessity when handling images. We are going to see now what types of operations we can do with arrays, and the importance of *types* is going to be more obvious.

2.2 Operations on arrays

2.2.1 Arithmetics on arrays

Numpy is written in a smart way such that it is able to handle operations between arrays of different sizes. In the simplest case, one can combine a scalar and an array, for example through an addition:

In [14]:
```
image
```

Out[14]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

In [15]:
```
image+10 # add 10 to each element of the array
# MZ:  advantage of using nupy ! will not work with list ! here it works
pixel-wise
```

Out[15]:
```
array([[ 57, 133, 143, ...,  24,  13,  22],
       [103, 154, 155, ...,  22,  17,  17],
       [136, 157, 153, ...,  12,  23,  13],
       ...,
       [ 91,  89,  84, ...,  16,  14,  17],
       [ 98,  92,  84, ...,  15,  17,  18],
       [101,  89,  78, ...,  14,  20,  17]], dtype=uint8)
```

Here Numpy automatically added the scalar 10 to **each** element of the array. Beyond the scalar case, operations between arrays of different sizes are also possible through a mechanism called broadcasting. This is an advanced (and sometimes confusing) features that we won't use in this course but about which you can read for example here (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html).

The only case we are going to consider here is operations between arrays of same size. For example we can multiply the image by itself. We use first the float version of the image:

In [16]:
```
image_sq = image_float*image_float
# MZ:
# does not perform matrix multiplication !, but multiply each pixel with
each pixel at the same position
# (will not perform like in linear algebra) (will have to use other nump
y functions)
```

In [17]:
```
image_sq
```

Out[17]:
```
array([[2.2090e+03, 1.5129e+04, 1.7689e+04, ..., 1.9600e+02, 9.0000e+00,
        1.4400e+02],
       [8.6490e+03, 2.0736e+04, 2.1025e+04, ..., 1.4400e+02, 4.9000e+01,
        4.9000e+01],
       [1.5876e+04, 2.1609e+04, 2.0449e+04, ..., 4.0000e+00, 1.6900e+02,
        9.0000e+00],
       ...,
       [6.5610e+03, 6.2410e+03, 5.4760e+03, ..., 3.6000e+01, 1.6000e+01,
        4.9000e+01],
       [7.7440e+03, 6.7240e+03, 5.4760e+03, ..., 2.5000e+01, 4.9000e+01,
        6.4000e+01],
       [8.2810e+03, 6.2410e+03, 4.6240e+03, ..., 1.6000e+01, 1.0000e+02,
        4.9000e+01]])
```

In [18]:
```
image_float
```

Out[18]:
```
array([[ 47., 123., 133., ...,  14.,   3.,  12.],
       [ 93., 144., 145., ...,  12.,   7.,   7.],
       [126., 147., 143., ...,   2.,  13.,   3.],
       ...,
       [ 81.,  79.,  74., ...,   6.,   4.,   7.],
       [ 88.,  82.,  74., ...,   5.,   7.,   8.],
       [ 91.,  79.,  68., ...,   4.,  10.,   7.]])
```

Looking at the first row we see $47^2 = 2209$ and $123^2 = 15129$ etc. which means that the multiplication operation has happened **pixel-wise**. Note that this is **NOT** a classical matrix multiplication. We can also see that the output has the same size as the original arrays:

In [19]:
```
image_sq.shape
```

Out[19]:
```
(303, 384)
```

In [20]:
```
image_float.shape
```

Out[20]:
```
(303, 384)
```

Let's see now what happens when we square the original 8bit image:

In [21]:
```
image*image
```

Out[21]:
```
array([[161,  25,  25, ..., 196,   9, 144],
       [201,   0,  33, ..., 144,  49,  49],
       [  4, 105, 225, ...,   4, 169,   9],
       ...,
       [161,  97, 100, ...,  36,  16,  49],
       [ 64,  68, 100, ...,  25,  49,  64],
       [ 89,  97,  16, ...,  16, 100,  49]], dtype=uint8)
```

We see that we don't get at all the expected result. Since we multiplied two 8bit images, Numpy assumes we want an 8bit output. And therefore the values are bound between 0-255. For example the first value is just the remainder of the modulo 256:

In [22]:
```
# MZ:
# what is above 255 get reassigned to a 0-255 value
# as numpy assumed that we have 8bit int !!!

# if you want > 255 values -> first make the matrix as float
```

In [23]:
```
2209%256
```

Out[23]:
```
161
```

The same thing happens e.g. if we add an integer scaler to the matrix:

In [24]:
```
print(image+230)
```

```
[[ 21  97 107 ... 244 233 242]
 [ 67 118 119 ... 242 237 237]
 [100 121 117 ... 232 243 233]
 ...
 [ 55  53  48 ... 236 234 237]
 [ 62  56  48 ... 235 237 238]
 [ 65  53  42 ... 234 240 237]]
```

Clearly something went wrong as we get values that are smaller than 230. Again any value "over-flowing" above 255 goes back to 0.

This problem can be alleviated in different ways. For example we can combine a integer array with a float scaler and Numpy will automatically give a result using the "most complex" type:

In [25]:
```
image_plus_float = image+230.0
```

In [26]:
```
print(image_plus_float)  # MZ: e.g. has removed 256: 277-256 = 21
```

```
[[277. 353. 363. ... 244. 233. 242.]
 [323. 374. 375. ... 242. 237. 237.]
 [356. 377. 373. ... 232. 243. 233.]
 ...
 [311. 309. 304. ... 236. 234. 237.]
 [318. 312. 304. ... 235. 237. 238.]
 [321. 309. 298. ... 234. 240. 237.]]
```

To be on the safe side we can also explicitely change the type when we know we might run into this kind of trouble. This can be done via the `.astype()` method:

In [27]:
```
# MZ:
# combine integer with float -> Python logic, use the most complex type
# will convert int to float and the output will be float
```

In [28]:
```
image_float = image.astype(float)
```

In [29]:
```
image_float.dtype
```

Out[29]:
```
dtype('float64')
```

Again, if we combine floats and integers the output is going to be a float:

In [30]:
```
image_float+230
```

Out[30]:
```
array([[277., 353., 363., ..., 244., 233., 242.],
       [323., 374., 375., ..., 242., 237., 237.],
       [356., 377., 373., ..., 232., 243., 233.],
       ...,
       [311., 309., 304., ..., 236., 234., 237.],
       [318., 312., 304., ..., 235., 237., 238.],
       [321., 309., 298., ..., 234., 240., 237.]])
```

2.2.2 Logical operations

A set of important operations when processing images are logical (or boolean) operations that allow to create masks for features to segment. Those have a very simple syntax in Numpy. For example, let's compare pixel intensities to some value *a*:

In [31]:
```
threshold = 100
```

In [32]:
```
image > threshold
```

Out[32]:
```
array([[False,  True,  True, ..., False, False, False],
       [False,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

We see that the result is again a pixel-wise comparison with a, generating in the end a boolean or logical matrix. We can directly assign this logical matrix to a variable and verify its shape and type and plot it:

In [33]:
```
image_threshold = image > threshold
```

In [34]:
```
image_threshold.shape
```

Out[34]:
```
(303, 384)
```

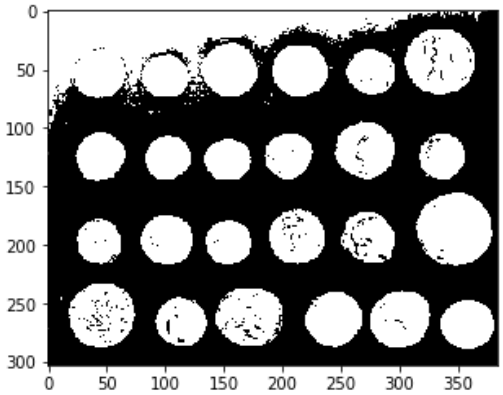
In [35]:
```
image_threshold.dtype
```

Out[35]:
```
dtype('bool')
```

In [36]:
```
image_threshold
```

Out[36]:
```
array([[False,  True,  True, ..., False, False, False],
       [False,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

In [37]:
```
plt.imshow(image_threshold);
```



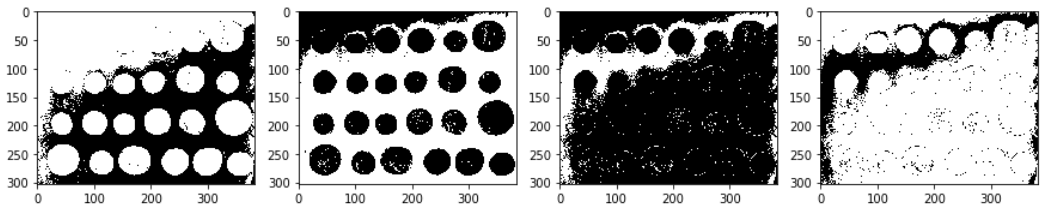
Of course other logical operator can be used (<, >, ==, !=) and the resulting boolean matrices combined:

In [38]:
```
threshold1 = 70
threshold2 = 100
image_threshold1 = image > threshold1
image_threshold2 = image < threshold2
```

In [39]:
```
# MZ
# logical: often use of masks
# e.g. you have a mask for dog and a mask for houses -> apply the masks
to the images using logicals
```

In [40]:
```
# MZ: here we deal with logical matrices
image_AND = image_threshold1 & image_threshold2  # MZ: True in the 2 mat
rices
image_XOR = image_threshold1 ^ image_threshold2  # MZ: what is True in 1
matrix but not in the other one
```

In [41]:
```
# MZ: multiple panels on matplot
plt.figure(figsize=(15,15)) # set the figure sizes
plt.subplot(1,4,1) # how many 1 row, 4 columns, and what is the 1st elem
ent
plt.imshow(image_threshold1)
plt.subplot(1,4,2) # in the subplot where 1 row and 4 columns, what shou
ld be the 2nd element
plt.imshow(image_threshold2)
plt.subplot(1,4,3)
plt.imshow(image_AND)
plt.subplot(1,4,4)
plt.imshow(image_XOR);
```



2.3 Numpy functions

To broadly summarize, one can say that Numpy offers three types of operations: 1. Creation of various types of arrays, 2. Pixel-wise modifications of arrays, 3. Operations changing array dimensions, 4. Combinations of arrays.

2.3.1 Array creation

Often we are going to create new arrays that later transform them. Functions creating arrays usually take arguments spcifying both the content of the array and its dimensions.

Some of the most useful functions create 1D arrays of ordered values. For example to create a sequence of numbers separated by a given step size:

In [42]:
```
np.arange(0,20,2)  # MZ: from where to where in step of what
```

Out[42]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Or to create an array with a given number of equidistant values:

In [43]:
```
np.linspace(0,20,5)
```

Out[43]:
```
array([ 0.,  5., 10., 15., 20.])
```

In higher dimensions, the simplest example is the creation of arrays full of ones or zeros. In that case one only has to specify the dimensions. For example to create a 3x5 array of zeros:

In [44]:
```
np.zeros((3,5))
```

Out[44]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Same for an array filled with ones:

In [45]:
```
np.ones((3,5))
```

Out[45]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

Until now we have only created one-dimensional lists of 2D arrays. However Numpy is designed to work with arrays of arbitrary dimensions. For example we can easily create a three-dimensional "ones-array" of dimension 5x8x4:

In [46]:
```
array3D = np.ones((2,6,5))
```

In [47]:
```
array3D
```

Out[47]:
```
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]])
```

In [48]:
```
array3D.shape
# MZ: you should decide which dimension is the channel/volume (usually t
he 1st or the last)

# MZ: numpy functions can easily deal with any dimension
# (e.g. it is easy to convert code written for 2D to code for 3D object
s)
```

Out[48]:
```
(2, 6, 5)
```

And all operations that we have seen until now and the following ones apply to such high-dimensional arrays exactly in the same way as before:

In [49]:
```
array3D*5
```

Out[49]:
```
array([[[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]],

       [[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]]])
```

We can also create more complex arrays. For example an array filled with numbers drawn from a normal distribution:

In [50]:
```
np.random.standard_normal((3,5))
```

Out[50]:
```
array([[ 0.51920188, -1.74490051,  0.19059971, -1.22782172, -0.08362917],
       [-1.91288875, -1.46339209, -0.29266003,  1.58959264,  1.39652976],
       [-2.23327794,  0.4977774 , -0.04227832,  0.97826304, -0.9933275
6]])
```

As mentioned before, some array-creating functions take additional arguments. For example we can draw samples from a gaussian distribution whose mean and variance we can specify.

In [51]:
```
np.random.normal(10, 2, (5,2))
# MZ: NB "Tab" for auto-completion; "Shift+Tab" to get the help for the
function
```

Out[51]:
```
array([[11.59504334, 10.84820206],
       [11.21592976,  9.46107067],
       [ 8.06999708, 10.02220069],
       [10.15008664, 11.81826128],
       [ 7.92993365, 11.43523018]])
```

2.3.2 Pixel-wise operations

Numpy has a large trove of functions to do all common mathematical operations matrix-wise. For example you can take the cosine of a matrix:

In [52]:
```
angles = np.random.random_sample(5)
angles
```

Out[52]:
```
array([0.94436116, 0.77710703, 0.8668537 , 0.68759525, 0.25572394])
```

In [53]:
```
np.cos(angles)
```

Out[53]:
```
array([0.58626054, 0.71294513, 0.64722816, 0.7727745 , 0.96748043])
```

Or to calculate exponential values:

In [54]:
```
np.exp(angles)
```

Out[54]:
```
array([2.57117028, 2.17517045, 2.37941273, 1.98892691, 1.29139618])
```

And many many more.

2.2.3 Operations changing dimensions

Some functions are accessible in the form of method, i.e. they are called using the dot notation. For example to find the maximum in an array:

In [55]:
```
angles.max() # MZ: return the max value inside the array
```

Out[55]:
```
0.9443611558667749
```

Alternatively there's also a maximum function:

In [56]:
```
np.max(angles)  # MZ: same as above but calling directly as a function
```

Out[56]: 0.9443611558667749

The `max` function like many others (min, mean, median etc.) can also be applied to a given axis. Let's imagine we have a 3D image (multiple planes) of 10x10x4 pixels:

In [ ]:
```
volume = np.random.random((10,10,4))
#volume
```

If we want to do a maximum projection along the third axis, we can specify:

In [58]:
```
projection = np.max(volume, axis = 2)
# MZ: specify an axis
# 0 1 2
# maximum along the 3 -> axis = 2
# creates a projection
```

In [59]:
```
projection.shape
```

Out[59]: (10, 10)

In [60]:
```
projection2 = np.max(volume, axis = 0)
projection2.shape
```

Out[60]: (10, 4)

In [61]:
```
projection3 = np.max(volume, axis = 1)
projection3.shape
```

Out[61]: (10, 4)

We see that we have indeed a new array with one dimension less because of the projection.

2.3.4 Combination of arrays

Finally arrays can be combined in multiple ways. For example if we want to assemble to images with the same size into a stack, we can use the stack function:

In [62]:
```
image1 = np.ones((4,4))
image2 = np.zeros((4,4))

stack = np.stack([image1, image2],axis = 2)
```

In [63]:
```
stack.shape
```

Out[63]: (4, 4, 2)

2.3 Slicing and indexing

Just like broadcasting, the selection of parts of arrays by slicing or indexing can become very sophisticated. We present here only the very basics to avoid confusion. There are often multiple ways to do slicing/indexing and we favor here easier to understant but sometimes less efficient solutions.

To simplify the visualisation, we use here a natural image included in the skimage package.

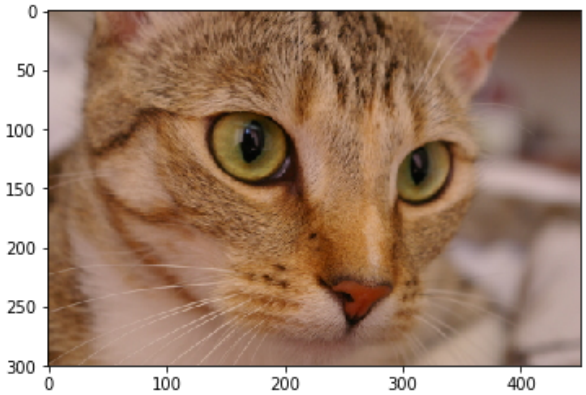
In [64]:
```
image = skimage.data.chelsea()
```

In [65]:
```
image.shape # MZ: 300x451 pixels and 3 planes: RGB
```

Out[65]: (300, 451, 3)

We see that the image has three dimensions, probably it's a stack of three images of size 300x400. Let us try to have a look at this image hoping that dimensions are handled gracefully:

In [66]:
```
plt.imshow(image); # MZ: if pass an image with 3 planes as last dim -> i
mplicitly assumes it is an RGB image
```



So we have an image of a cat with dimensions 300x400. The image being in natural colors, the three dimensions probably indicate an RGB (red, green, blue) format, and the plotting function just knows what to do in that case.

2.3.1 Array slicing

Let us now just look at one of the three planes composing the image. To do that, we are going the select a portion of the image array by slicing it. One can give:

- a single index e.g. 0 for the first element
- a range e.g. 0:10 for the first 10 elements
- take all elements using a semi-column :

What portion is selected has to be specified for each dimensions of an array. In our particular case, we want to select all rows, all columns and a single plane of the image:

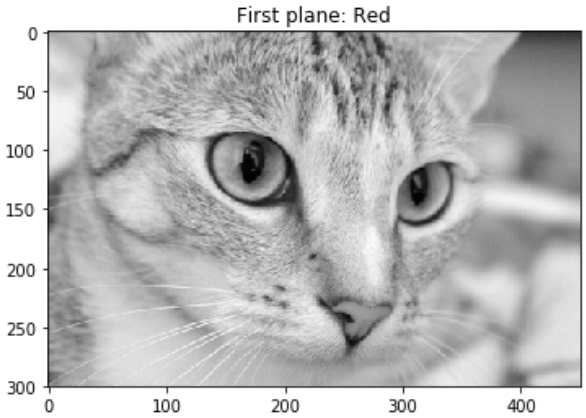
In [67]:
```
image.shape
```

Out[67]: (300, 451, 3)

In [68]:
```
image[:,:,1].shape # MZ: select only the 2nd plane
```

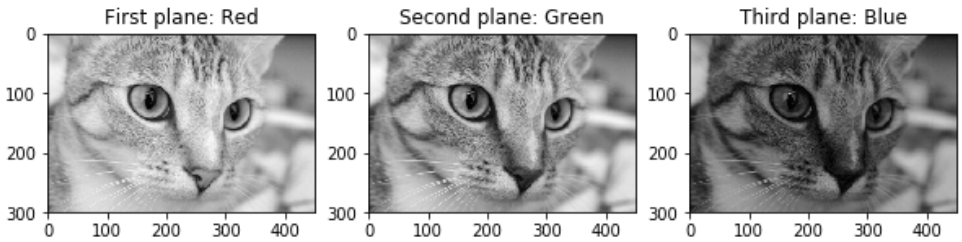
Out[68]: (300, 451)

In [69]:
```
plt.imshow(image[:,:,0],cmap='gray')
# MZ: cmap argument -> here redondant with plt.gray();
# different colormaps provided by matplotlib (map pixel-values to color
s)
plt.title('First plane: Red');
```



We see now the red layer of the image. We can do the same for the others by specifying planes 0, 1, and 2:

In [70]:
```
plt.figure(figsize=(10,10))
plt.subplot(1,3,1)
plt.imshow(image[:,:,0],cmap='gray')
plt.title('First plane: Red')
plt.subplot(1,3,2)
plt.imshow(image[:,:,1],cmap='gray')
plt.title('Second plane: Green')
plt.subplot(1,3,3)
plt.imshow(image[:,:,2],cmap='gray')
plt.title('Third plane: Blue');


# MZ:
# no physical meaning to the colormaps, you can put what ever you want a
s colors
# is only the rendering of the pixel values
```



Logically intensities are high for the red channel and low for the blue channel as the image has red/brown patterns. We can confirm that by measuring the mean of each plane. To do that we use the same function as above but apply it to a singel sliced plane:

In [71]:
```
image0 = image[:,:,0] # MZ: retain only 1st dim
```

In [72]:
```
np.mean(image0) # MZ: mean of all pixels
```

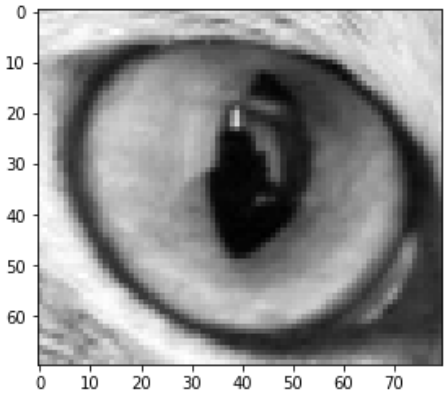
Out[72]:
```
147.67308943089432
```

and for all planes using a comprehension list:

```
In [73]: [np.mean(image[:,:,i]) for i in range(3)]  # MZ: calculat the mean of ev
         ery plane
Out[73]: [147.67308943089432, 111.44447893569844, 86.79785661492978]
```

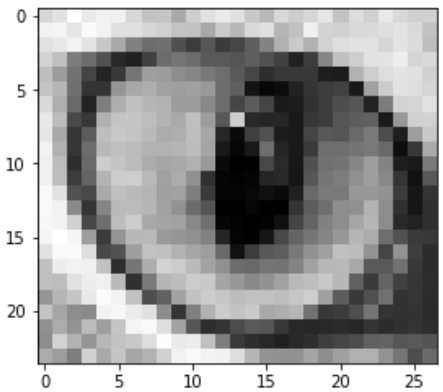
To look at some more details let us focus on a smaller portion of the image e.g. one of the cat's eyes. For that we are going to take a slice of the red image and store it in a new variable and display the selection. We consider pixel rows from 80 to 150 and columns from 130 to 210 of the first plane (0).

```
In [74]: image_red = image[80:150,130:210,0]
         plt.imshow(image_red,cmap='gray');
```



There are different ways to select parts of an array. For example one can select every n'th element by giving a step size. In the case of an image, this subsamples the data:

```
In [75]: image_subsample = image[80:150:3,130:210:3,0]
         plt.imshow(image_subsample,cmap='gray');
```



2.3.2 Array indexing

In addition to slicing an array, we can also select specific values out of it. There are many (https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html) different ways to achieve that, but we focus here on two main ones.

First, one might have a list of pixel positions and one wishes to get the values of those pixels. By passing two lists of the same size containing the rows and columns positions of those pixels, one can recover them:

In [76]:
```
row_position = [0,1,2,3]
col_position = [0,1,0,1]

print(image_red[0:5,0:5])
# MZ: pass the 2 lists -> assumes that you mean the pixels you want

image_red[row_position,col_position]
# MZ: output is just a list of pixels, not in 3 dim anymore ! output is
1D

# MZ => you can extract either with 3-dot notation or by passing a list
```

```
[[166 162 169 174 185]
 [183 192 185 183 173]
 [179 178 168 175 176]
 [187 184 187 189 185]
 [195 192 187 181 169]]
```

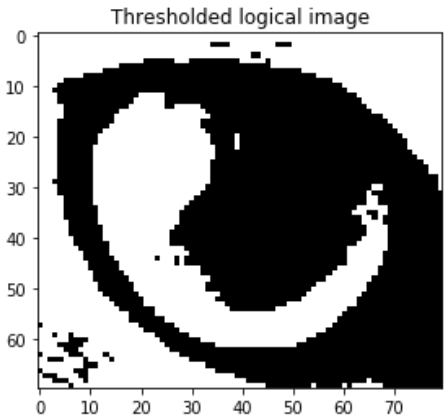
Out[76]:
```
array([166, 192, 179, 184], dtype=uint8)
```

Alternatively, one can pass a logical array of the same dimensions as the original array, and only the True pixels are selected. For example, let us create a logical array by picking values above a threshold:

In [77]:
```
threshold_image = image_red>120
```

Let's visualize it. Matplotlib handles logical arrays simply as a binary image:

In [78]:
```
plt.imshow(threshold_image)
plt.title('Thresholded logical image');
```



We can recover the value of all the "white" (True) pixels in the original image by **indexing one array with the other**:

In [79]:
```
selected_pixels = image_red[threshold_image]
# MZ:
# create a mask with logical array
# pass another image, of the same size, should be a boolean array and
# instead of passing explicit lists of rows/columns -> direct pass an ar
ray
# output is again a list
# useful e.g. for segmentation (create a mask where you have the cells o
nly to extract
# from other panes where you have light emission and average the light e
mission)
print(selected_pixels)
```

```
[166 162 169 ... 148 137 132]
```

And now ask how many pixels are above threshold and what their average value is.

In [80]:
```
len(selected_pixels)
```

Out[80]: 2585

In [81]:
```
np.mean(selected_pixels)
```

Out[81]: 153.59381044487426

In [82]:
```
threshold_image # MZ: mask is a boolean array 2D
```

Out[82]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True, False, False, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False]])
```

In [83]:
```
np.argwhere(threshold_image)
# MZ: 2 dim arrays -> gives where are the True values in x,y coordinates
```

Out[83]:
```
array([[ 0,  0],
       [ 0,  1],
       [ 0,  2],
       ...,
       [69, 65],
       [69, 66],
       [69, 67]])
```

In [ ]:
```
# MZ: to have all attributes and functions associated with an object
#dir(threshold_image)
```

In [ ]:
```
# MZ: same works for packages
#dir(np)
```

We now know that there are 2585 pixels above the threshold and that their mean is 153.6

In [86]:
```
# to plot with transprency: (e.g. to plot 1 fig on the top of another)
# imshow(alpha=0.5)
```

3. Image import/export

For the moment, we have only used images that were provided internally by skimage. We are however normally going to use data located in the file system. The module skimage.io deals with all in/out operations and supports a variety of different import mechanisms.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
```

3.1 Simple case

Most of the time the simples command imread() will do the job. One has just to specifiy the path of the file or a url. In general your path is going to look something like:

```
image = io.imread('/This/is/a/path/MyData/Klee.jpg')
```

In [5]:
```
file_path = 'Data/Klee.jpg'
print(file_path)
```

```
Data/Klee.jpg
```

Here we only use a relative path, knowing that the `Data` folder is in the same folder as the notebook. However you can also give a complete path. We can also check what's the complete path of the current file:

In [6]:
```
import os
print(os.path.realpath(file_path))
```

```
/home/marie/Documents/CAS_data_science/CAS_21.01.2020_Python_Image_Proces
sing/PyImageCourse-master/Data/Klee.jpg
```

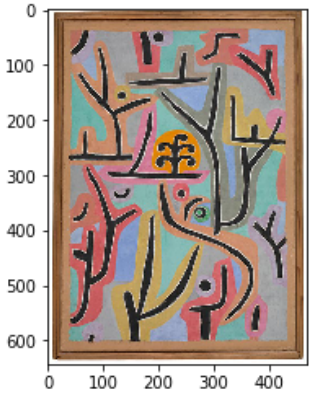
Now we can import the image:

In [7]:
```
image = io.imread(file_path)
```

In [8]:
```
image.shape
```

Out[8]: (643, 471, 3)

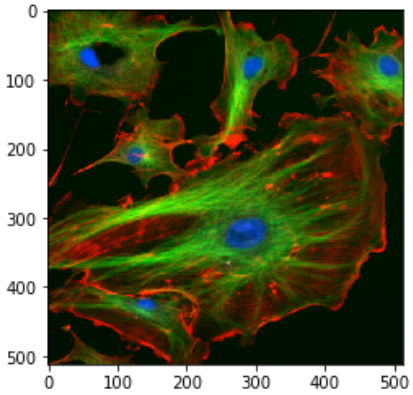
In [9]:
```
plt.imshow(image);
```



Now with a url:

In [18]:
```
image = io.imread('https://upload.wikimedia.org/wikipedia/commons/0/09/FluorescentCells.jpg')
```

In [19]:
```
plt.imshow(image)
plt.show()
```



3.2 Series of images (.tif)

Popular compressed formats such as jpg are usually used for natural images e.g. in facial recognition. The reason for that is that for those applications, in most situations one does not care about quantitative information and effects of information compression occurring in jpg are irrelevant. Also, those kind of data are rarely multi-dimensional (except for RGB).

In most other cases, the actual pixel intensity gives important information and one needs a format that preserves that information. Usually this is the .tif format or one of its many derivatives. One advantage is that the .tif format allows to save multiple images within a single file, a very useful feature for multi-dimensional acquisitions.

You might encounter different situations.

3.2.1 Series of separate images

In the first case, you would have multiple single .tif files within one folder. In that case, the file name usually contains indications about the content of the image, e.g a time point or a channel. The general way of dealing with this kind of situation is to use regular expressions, a powerful tool to parse information in text. This can be done in Python using the `re` module.

Here we will use an approach that identifies much simpler patterns.

Let's first see what files are contained within a folder of a microscopy experiment containing images acquired at two wavelengths using the `os` module:

In [10]:
```
import glob
import os
```

In [11]:
```
folder = 'Data/BBBC007_v1_images/A9'
```

Let's list all the files contained in the folder

In [13]:
```
files = os.listdir(folder) # MZ: list all files that are within the dire
ctory
print(files)
```

```
['A9 p10f.tif', 'A9 p5f.tif', 'A9 p9d.tif', 'A9 p7f.tif', 'A9 p7d.tif', '
A9 p10d.tif', 'A9 p9f.tif', 'A9 p5d.tif']
```

The two channels are defined by the last character before .tif. Using the wild-card sign *we can define a pattern to select only the 'd' channel:* d.tif. We complete that name with the correct path. Now we use the native Python module `glob` to parse the folder content using this pattern:

In [14]:
```
d_channel = glob.glob(folder+'/*d.tif')
d_channel
```

Out[14]:
```
['Data/BBBC007_v1_images/A9/A9 p9d.tif',
 'Data/BBBC007_v1_images/A9/A9 p7d.tif',
 'Data/BBBC007_v1_images/A9/A9 p10d.tif',
 'Data/BBBC007_v1_images/A9/A9 p5d.tif']
```

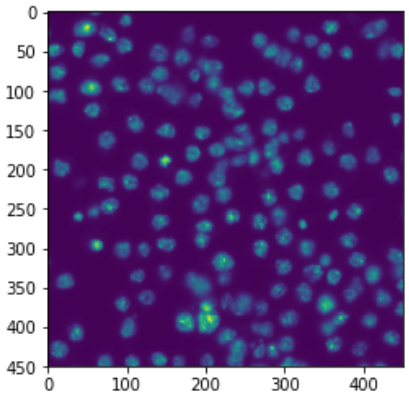
Then we use again the `imread()` function to import a specific file:

In [15]:
```
image1 = io.imread(d_channel[0])
```

In [16]:
```
image1.shape
```

Out[16]:
```
(450, 450)
```

In [17]:
```
plt.imshow(image1);
```



These two steps can in principle be done in one step using the `imread_collection()` function of skimage.

We can also import all images and put them in list if we have sufficient memory:

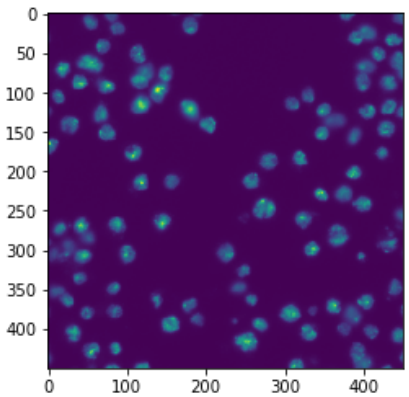
In [18]:
```
channel1_list = []
for x in d_channel:
    temp_im = io.imread(x)
    channel1_list.append(temp_im)
```

Let's see what we have in that list of images by plotting them:

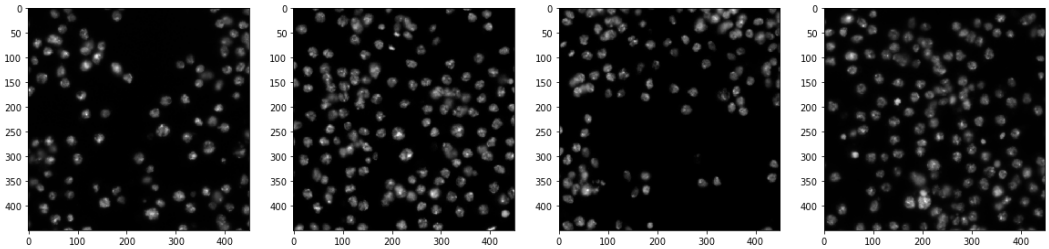
In [94]:
```
channel1_list[0].shape
```

Out[94]: (450, 450)

In [106]:
```
plt.imshow(channel1_list[0]);
```



In [105]:
```
num_plots = len(channel1_list)
plt.figure(figsize=(20,30))
for i in range(num_plots):
    plt.subplot(1,num_plots,i+1)
    plt.imshow(channel1_list[i],cmap = 'gray')
```



3.2.2 Multi-dimensional stacks

We now look at a more complex multi-dimensional case taken from a public dataset (J Cell Biol. 2010 Jan 11;188(1):49-68) that can be found here (http://flagella.crbs.ucsd.edu/images/30567).

We already provide it in the datafolder:

In [19]:
```
file = 'Data/30567/30567.tif'
# MZ: tif can contain many data and also can contain metadata -> very us
eful
```

In [21]:
```
image = io.imread(file)
```

The dataset is a time-lapse 3D confocal microscopy acquired in two channels, one showing the location of tubulin, the other of lamin (cell nuclei).

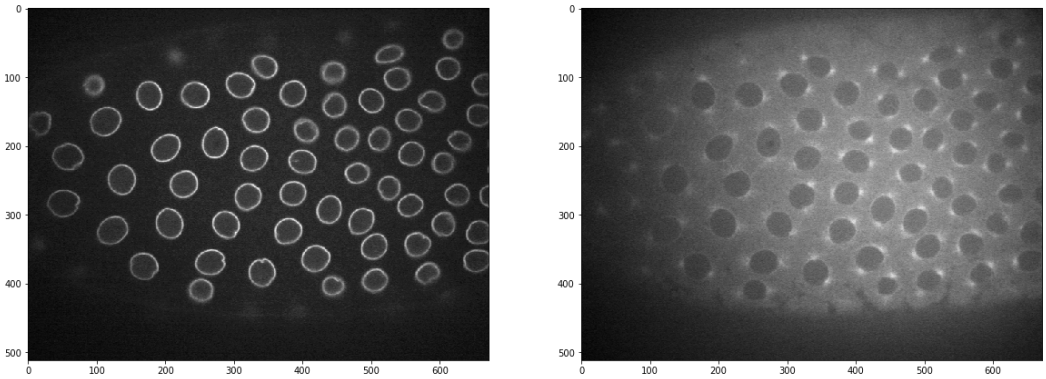
All .tif variants have the same basic structure: single image planes are stored in individual "sub-directories" within the file. Some meta information is stored with each plane, some is stored for the entire file. However, how the different dimensions are ordered within the file (e.g. all time-points of a given channel first, or alternatively all channels of a given time-point) can vary wildly. The simplest solution is therefore usually to just import the file, look at the size of various dimensions and plot a few images to figure out how the data are organized.

In [22]:
```
image.shape
```

Out[22]: (72, 2, 5, 512, 672)

We know we have two channels (dimension 2), and five planes (dimension 3). Usually the large numbers are the image dimension, and therefore 72 is probably the number of time-points. Using slicing, we look at the first time point, of both channels, of the first plane, and we indeed get an appropriate result:

In [23]:
```
plt.figure(figsize=(20,10))
plt.subplot(1,2,1)
plt.imshow(image[0,0,0,:,:],cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image[0,1,0,:,:],cmap = 'gray');
```



We can check that our indexing works by checking the dimensions of the sliced image:

In [24]:
```
# where are the metadata and how to access them -> data-specific
image[0,0,0,:,:].shape
```

Out[24]: (512, 672)

As we have seen in the Numpy chapter, we can do various operations on arrays. In particular we saw that we can do projections. Let's extract all planes of a given time point and channel:

In [109]:
```
stack = image[0,0,:,:,:]
stack.shape
```

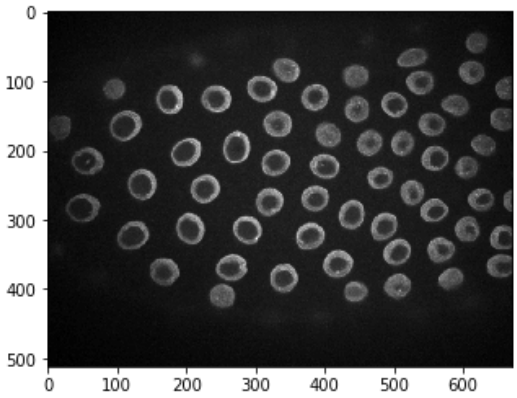
Out[109]: (5, 512, 672)

Here, to do a max projection, we now have to project all the planes along the **first** dimension, hence:

In [25]:
```
maxproj = np.max(image[0,0,:,:,:],axis = 0)
#MZ:  1st time point, 1st channel, but all the planes; take all max alon
g 1st dimension -> projection
# project on the 1st dimension (axis=0)
maxproj.shape
```

Out[25]: (512, 672)

In [26]:
```
plt.imshow(maxproj,cmap = 'gray')
plt.show()
```



skimage allows one to use specific import plug-ins for various applications (e.g. gdal for geographic data, FITS for astronomy etc.).

In particular it offers a lower-lever access to tif files through the tifffile module. This allows one for example to import only a subset of planes from the dataset if the latter is large.

In [27]:
```
# load only what you want (e.g. the 1st time point)
# so you don't need to load all the timepoints in memory

# tif -> most often used format for this kind of data
```

In [28]:
```
from skimage.external.tifffile import TiffFile

data = TiffFile(file)
```

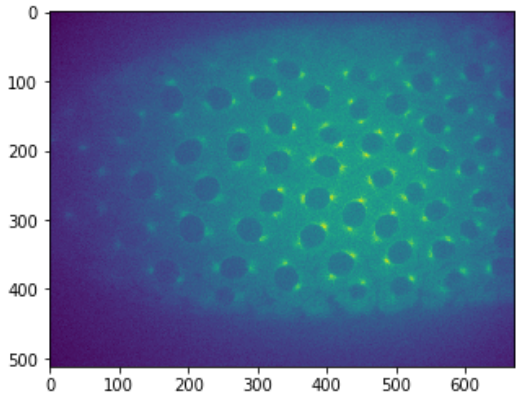
Now the file is open but not imported, and one can query information about it. For example some metadata:

In [29]:
```
data.info()
```

Out[29]:
```
'TIFF file: 30567.tif, 473 MiB, big endian, ome, 720 pages\n\nSeries 0: 7
2x2x5x512x672, uint16, TCZYX, 720 pages, not mem-mappable\n\nPage 0: 512x
672, uint16, 16 bit, minisblack, raw, ome|contiguous\n* 256 image_width
(1H) 672\n* 257 image_length (1H) 512\n* 258 bits_per_sample (1H) 16\n* 2
59 compression (1H) 1\n* 262 photometric (1H) 1\n* 270 image_description
(3320s) b\'<?xml version="1.0" encoding="UTF-8"?><!-- Wa\n* 273 strip_off
sets (86I) (182, 8246, 16310, 24374, 32438, 40502, 48566, 56630,\n* 277 s
amples_per_pixel (1H) 1\n* 278 rows_per_strip (1H) 6\n* 279 strip_byte_co
unts (86I) (8064, 8064, 8064, 8064, 8064, 8064, 8064, 8064, \n* 282 x_res
olution (2I) (1, 1)\n* 283 y_resolution (2I) (1, 1)\n* 296 resolution_uni
t (1H) 1\n* 305 software (17s) b\'LOCI Bio-Formats\''
```

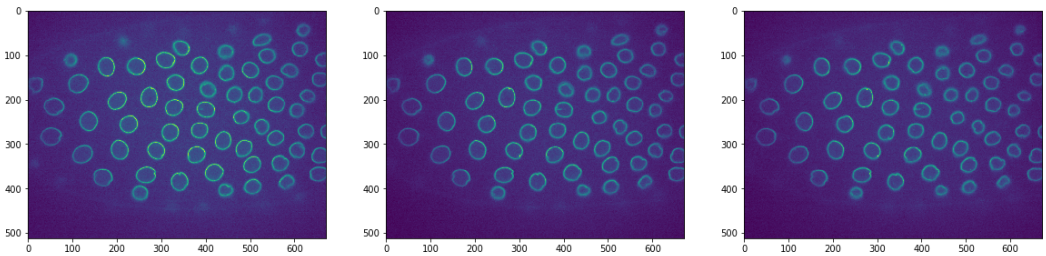
Some specific planes:

In [30]:
```
plt.imshow(data.pages[6].asarray())
plt.show()
```



In [31]:
```
image = [data.pages[x].asarray() for x in range(3)]
```

In [32]:
```
plt.figure(figsize=(20,10))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(image[i])
plt.show()
```



3.2.3 Alternative formats

While a large majority of image formats is somehow based on tif, instrument providers often make their own tif version by creating a proprietary format. This is for example the case of the Zeiss microscopes which create the .czi format.

In almost all cases, you can find an dedicated library that allows you to open your particular file. For example for czi, there is a specific package (https://pypi.org/project/czifile/).

More generally your research field might use some particular format. For example Geospatial data use the format GDAL, and for that there is of course a dedicated package (https://pypi.org/project/GDAL/).

Note that a lot of biology formats are well handled by the tifffile package. io.imread() tries to use the best plugin to open a format, but sometimes if fails. If you get an error using the default io.imread() you can try to specific what plugin should open the image, *.e.g*

```
image = io.imread(file, plugin='tifffile')
```

3.3 Exporting images

There are two ways to save images. Either as plain matrices, which can be written and re-loaded very fast, or as actual images.

Just like for loading, saving single planes is easy. Let us save a small region of one of the images above:

In [119]:
```
image[0].shape
```

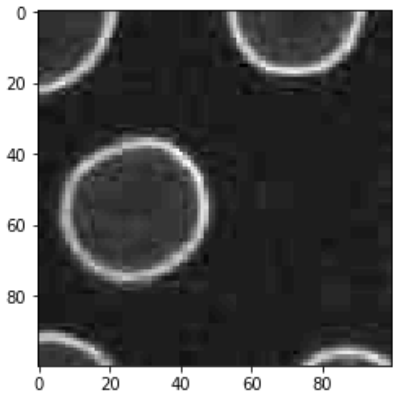
Out[119]:
```
(512, 672)
```

In [33]:
```
io.imsave('Data/region.tif',image[0][200:300,200:300])  # MZ: specify wh
at you want to save
io.imsave('Data/region.jpg',image[0][200:300,200:300])
```

```
/usr/local/lib/python3.5/dist-packages/skimage/util/dtype.py:141: UserWar
ning: Possible precision loss when converting from uint16 to uint8
  .format(dtypeobj_in, dtypeobj_out))
```

In [34]:
```
reload_im = io.imread('Data/region.jpg') # MZ: jpg not good for scientif
ic purposes
```

In [35]:
```
plt.imshow(reload_im,cmap='gray')
plt.show()
```



Saving multi-dimensional .tif files is a bit more complicated as one has of course to be careful with the dimension order. Here again the tifffile module allows to achieve that task. We won't go through the details, but here's an example of how to save a dataset with two time poins, 5 stacks, 3 channels into a file that can then be opened as a hyper-stack in Fiji:

In [36]:
```
from skimage.external.tifffile import TiffWriter

data = np.random.rand(2, 5, 3, 301, 219)#generate random images
data = (data*100).astype(np.uint8)#transform data in a reasonable 8bit r
ange

with TiffWriter('Data/multiD_set.tif', bigtiff=False, imagej=True) as ti
f:
    for i in range(data.shape[0]):
        tif.save(data[i])
```

3.4 Interactive plotting

Jupyter offers a solution to interact with various types of plots: ipywidget

In [125]:
```
from ipywidgets import interact, IntSlider
```

The interact() function takes as input a function and a value for that function. That function should plot or print some information. interact() then creates a widget, typically a slider, executes the plotting function and adjusts the ouptut when the slider is moving. For example:

In [126]:
```
def square(num=1):
    print(str(num)+' squared is: '+str(num*num))
```

In [127]:
```
square(3)
```

```
3 squared is: 9
```

In [128]:
```
interact(square, num=(0,20,1));
```

Depending on the values passed as arugments, interact() will create different widgets. E.g. with text:

In [129]:
```
def f(x):
    return x
interact(f, x='Hi there!');
```

An important note for our imaging topic: when moving the slider, the function is continuously updated. If the function does some computationally intensitve work, this might just overload the system. To avoid that, one can explicitly specifiy the slider type and its specificities:

In [130]:
```
def square(num=1):
    print(str(num)+' squared is: '+str(num*num))
interact(square, num = IntSlider(min=-10,max=30,step=1,value=10,continuo
us_update = False));
```

If we want to scroll through our image stack we can do just that. Let's first define a function that will plot the first plane of the channel 1 at all time points:

In [131]:
```
image = io.imread(file)
```

In [132]:
```
def plot_plane(t):
    plt.imshow(image[t,0,0,:,:])
    plt.show()
```

In [133]:
```
interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = False));
```

Of course we can do that for multiple dimensions:

In [134]:
```
def plot_plane(t,c,z):
    plt.imshow(image[t,c,z,:,:])
    plt.show()

interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = True),
         c = IntSlider(min=0,max=1,step=1,value=0,continuous_update = Tr
ue),
         z = IntSlider(min=0,max=4,step=1,value=0,continuous_update = Tr
ue));
```

And we can make it as fancy as we want:

In [135]:
```
def plot_plane(t,c,z):
    if c == 0:
        plt.imshow(image[t,c,z,:,:], cmap = 'Reds')
    else:
        plt.imshow(image[t,c,z,:,:], cmap = 'Blues')
    plt.show()

interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = True),
         c = IntSlider(min=0,max=1,step=1,value=0,continuous_update = Tr
ue),
         z = IntSlider(min=0,max=4,step=1,value=0,continuous_update = Tr
ue));
```

4. Basic Image processing: Filtering, scaling, thresholding

Almost all image processing pipelines start with some basic procedures like thresholding, scaling, or projecting a multi-dimensional image.

Let us import again all necessary packages:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
from skimage.external.tifffile import TiffFile
```

Most filtering functions will come out from the filters module of scikit-image:

In [2]:
```
import skimage.filters as skf
```

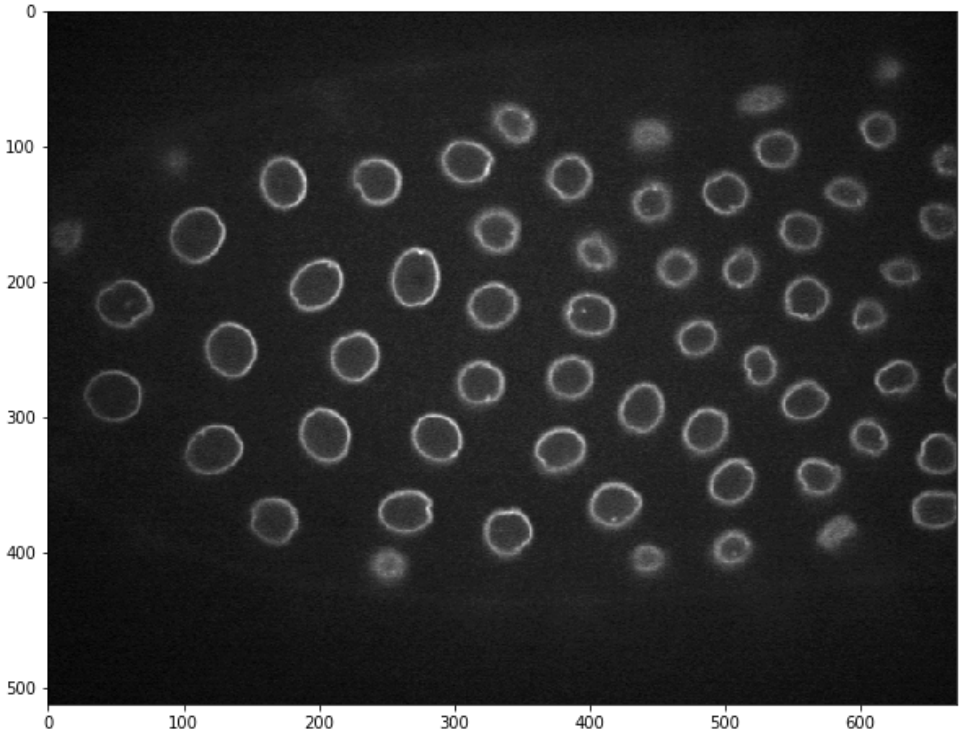
A specific region size/shape has often to be specified for filters. Those are defined in the morphology module:

In [3]:
```
import skimage.morphology as skm
```

Additionally, this module offers a set of binary operators essential to operate on the masks resulting from segmentation.

We will start working on a single plane of the dataset seen in chapter 3 (3-Image_import.ipynb)

In [4]:
```
#load image
data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()
#plot image
plt.figure(figsize=(10,10))
plt.imshow(image,cmap = 'gray');
```

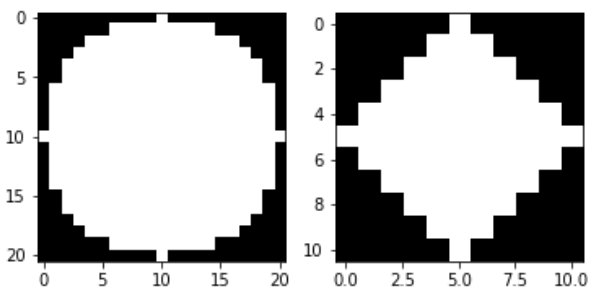


4.1 Filtering

A large set of filters are offered in scikit-image. Filtering is a local operation, where a value is calculated for each pixel and its surrounding region according to some function. For example a median filter of size 3, calculates for each pixel the median value of the 3x3 region around it.

Most filters take as input a specified region to consider for the calculation (e.g. 3x3 region). Those can be defined using the morphology module e.g.

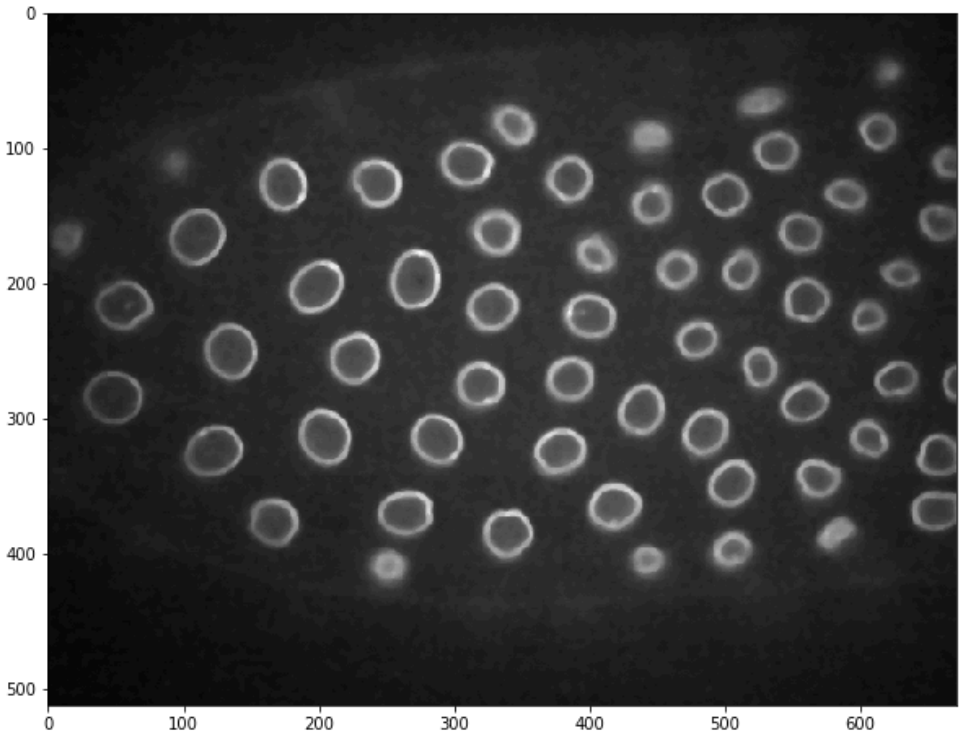
In [5]:
```
disk = skm.disk(10)
diamond  = skm.diamond(5)
plt.subplot(1,2,1)
plt.imshow(disk,cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(diamond,cmap = 'gray');
```



In [6]:
```
image_mean = skf.median(image,selem=skm.disk(3))  # MZ: selem = selectio
n element
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```

In [7]:
```
plt.figure(figsize=(10,10))
plt.imshow(image_mean,cmap = 'gray');
```



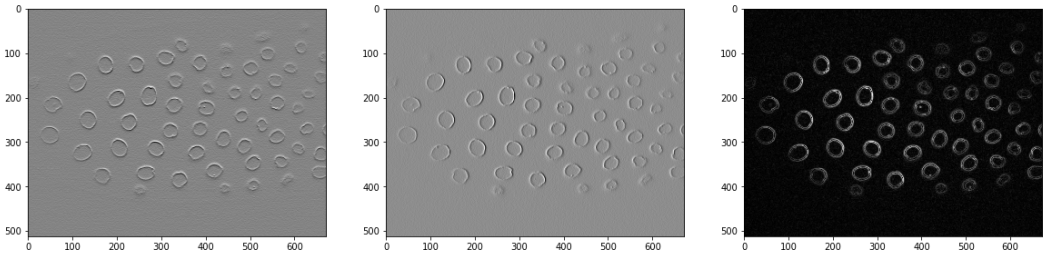
Similar filters can be defined for a large range of operations: sum, min, max, mean etc.

More specific filters are also provide in skimage. For example finding the gradient of intensity in an image can be done with a Sobel filter. Here for horizontal, vertical and their sum:

In [8]:
```
image_gradienth = skf.sobel_h(image) # MZ: sobel filter, applied horizon
tally
image_gradientv = skf.sobel_v(image) # MZ: same filter, applied vertical
ly
image_gradient = np.sqrt(image_gradientv**2+image_gradienth**2) # combin
e both
```

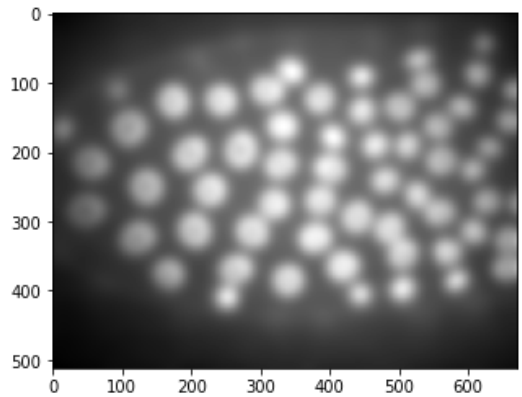
In [9]:
```
plt.figure(figsize=(20,10))
plt.subplot(1,3,1)
plt.imshow(image_gradienth,cmap = 'gray')
plt.subplot(1,3,2)
plt.imshow(image_gradientv,cmap = 'gray')
plt.subplot(1,3,3)
plt.imshow(image_gradient,cmap = 'gray')

# MZ: highlight the edges horizontally (1) and vertically (2)
# (3) combined, the edges are highlighted
```

Out[9]: <matplotlib.image.AxesImage at 0x7f7f96116a90>



Finally, some functions can be used to filter the image, and one can pass function parameters to the filter. For example to filter with a Gaussian of large standard deviation $\sigma = 10$:

In [10]:
```
image_gauss = skf.gaussian(image, sigma=10)#, preserve_range=True)
# MZ: Gaussian with really large sigma (e.g. highlight the nuclei)
# MZ: to just filter noise: use much smaller sigma
plt.imshow(image_gauss,cmap = 'gray');
# Gaussian filter automatically re-scales the image between the 0 and 1
```



A warning regarding filters: some filters can change the type and even the range of intensity of the image. Typically the gaussian filter used above rescales the image between 0 and 1:

In [11]:
```
print(image.dtype)
print(image.max())
print(image.min())
```

```
uint16
20303
2827
```

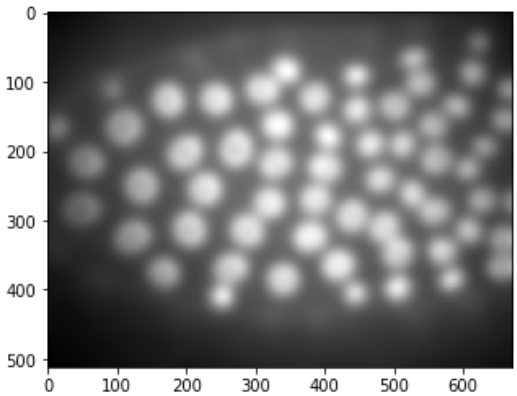
In [12]:
```
print(image_gauss.dtype)
print(image_gauss.max())
print(image_gauss.min())
```

```
float64
0.12531917375072713
0.054386287321711344
```

In many cases, one can specify whether the original range should be preserved:

In [13]:
```
image_gauss_preserve = skf.gaussian(image, sigma=10, preserve_range=True)
# MZ: use preserve_range, so that values are not re-scaled
plt.imshow(image_gauss_preserve,cmap = 'gray');
print(image_gauss_preserve.dtype)
print(image_gauss_preserve.max())
print(image_gauss_preserve.min())
```

```
float64
8212.792051753902
3564.2053396283527
```



4.2 Intensity re-scaling

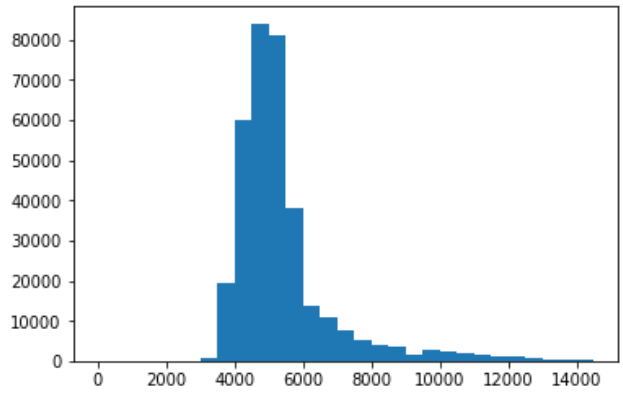
A very common operation to do in an image processing pipeline, is to rescale the intensity of images. The reason can be diverse: for example, one might want to remove an offset added to each pixel by the camera, or one might want to homogenize multiple images with slightly varying exposures.

The simplest thing to do is to rescale from min to max in the range 0-1. To create a histogram of the pixel values of an image, we first have to "flatten" the array, i.e. remove the dimensions, so that the plotting function doesn't believe we have a series of separate measurements.

In [14]:
```
np.ravel(image).shape
# MZ convert 2D to 1D array -> flatten to have 1 big list of pixels
# (needed to draw one single histogram for all values)
```

Out[14]: (344064,)

In [15]:
```
plt.hist(np.ravel(image), bins = np.arange(0,15000,500))
plt.show()
print("min val: "+ str(np.min(image)))
print("max val: "+ str(np.max(image)))
```




```
min val: 2827
max val: 20303
```

In [16]:
```
image_minmax = (image-image.min())/(image.max()-image.min())
image_minmax[image_minmax>1] = 1
```

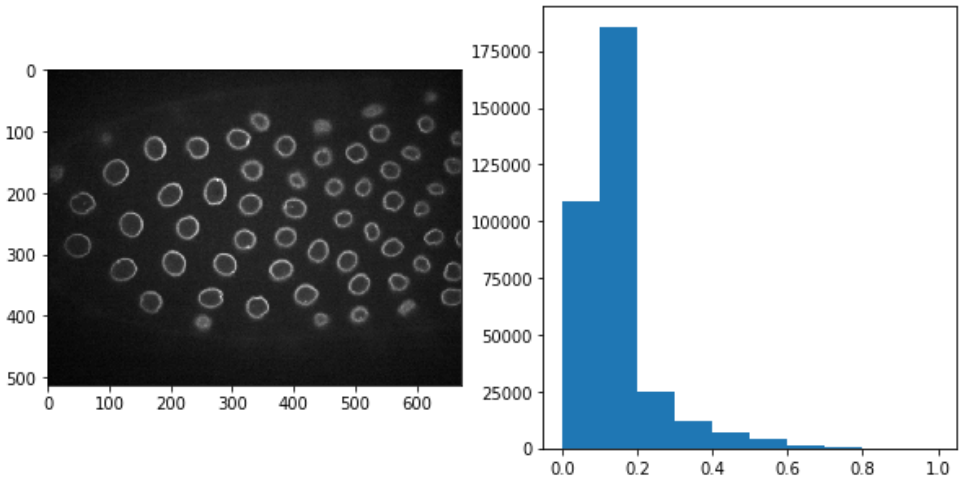
One problem that might emerge is that a few pixels might be affected by rare noise events that give them abnormal values. One way to remedy that is to use a small median filter in order to suppress those aberrant values:

In [17]:
```
image_median = skf.median(image,selem=np.ones((2,2)))
print("min val: "+ str(np.min(image_median)))
print("max val: "+ str(np.max(image_median)))

image_median_rescale = (image_median-image_median.min())/(image_median.m
ax()-image_median.min())
image_median_rescale[image_minmax>1] = 1
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```

```
min val: 3084
max val: 20046
```

In [18]:
```
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(image_median_rescale,cmap = 'gray')
plt.subplot(1,2,2)
plt.hist(np.ravel(image_median_rescale))#, bins = np.arange(0,15000,50
0))
plt.show()
```



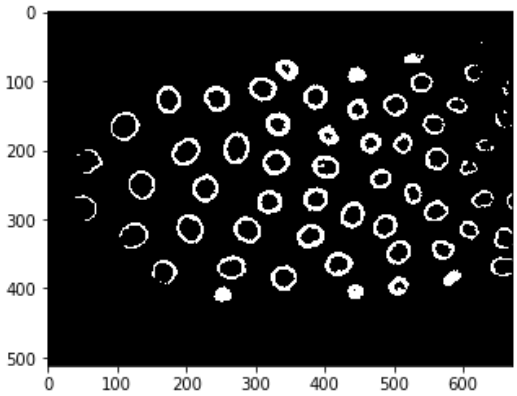
Note that the `skimage.exposure` module offers several functions to adjust the image intensity distribution.

4.3 Thresholding

Another common operation is to try isolating regions of an image based on their intensity by using an intensity threshold: one can create a maks object where all values larger than a threshold are 1 and the other 0. It is usually better to use a smoothed version of the image (e.g. median or gaussian filtering) to avoid including noisy pixels in the maks.

Let us imagine that we want to isolate the nuclei in our current image. To do that we can try to use their bright contour. Based on the intensity histogram, let's try to pick a threshold manually:

In [19]:
```
# MZ: thresholded image to only keep values above a given threshold

threshold_manual = 8000

#create a mask using a logical operation
image_threshold = image_median>threshold_manual  # MZ: create a boolean
array

plt.imshow(image_threshold, cmap ='gray')
plt.show()
```



Instead of picking manually the threshold, one can use one of the many automatic methods available in skimage,

In [20]:
```
image_otsu_threshold = skf.threshold_otsu(image_median)
```

In [21]:
```
image_otsu_threshold
```

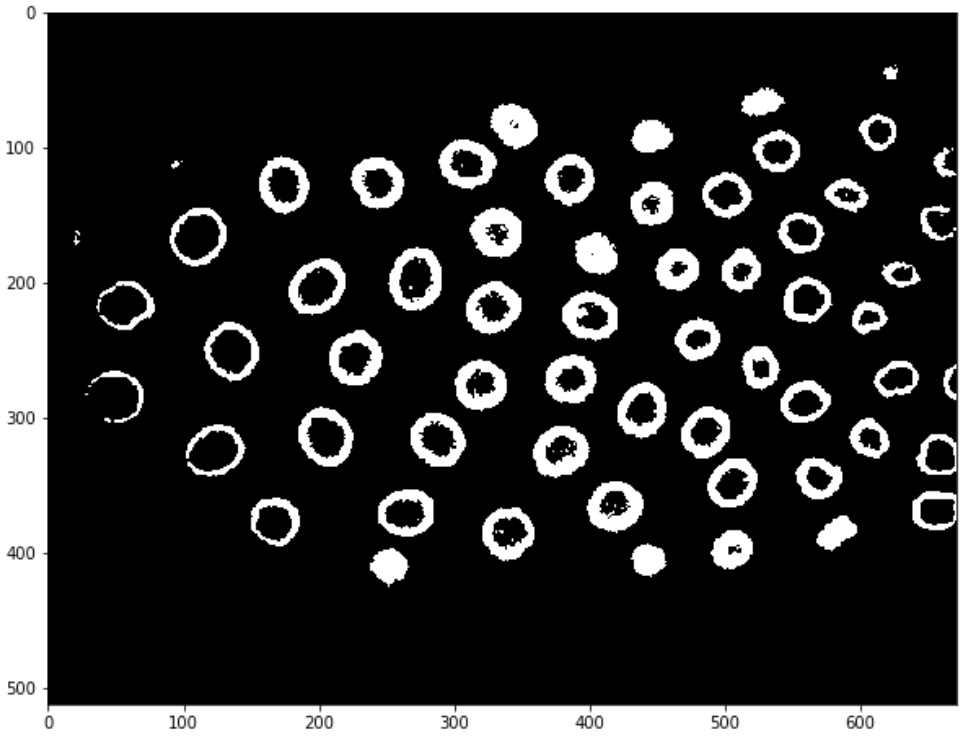
Out[21]: 7196

In [22]:
```
image_otsu_threshold = skf.threshold_otsu(image_median)
print(image_otsu_threshold)
image_li_threshold = skf.threshold_li(image_median)
print(image_li_threshold)
```

```
7196
6416.599708799512
```

Knowing that threshold value we can create a binary image setting all pixels higher than the threshold to 1.

In [23]:
```
image_otsu = image_median > image_otsu_threshold
plt.figure(figsize=(10,10))
plt.imshow(image_otsu, cmap = 'gray')
plt.show()
```



Since the illumination is uneven accross the image, all standard thresholding methods are going to fail in some region of the image. What we could try to do instead is using a local thresholding, by repeating a standard thresholding method in sub-regions of the image:

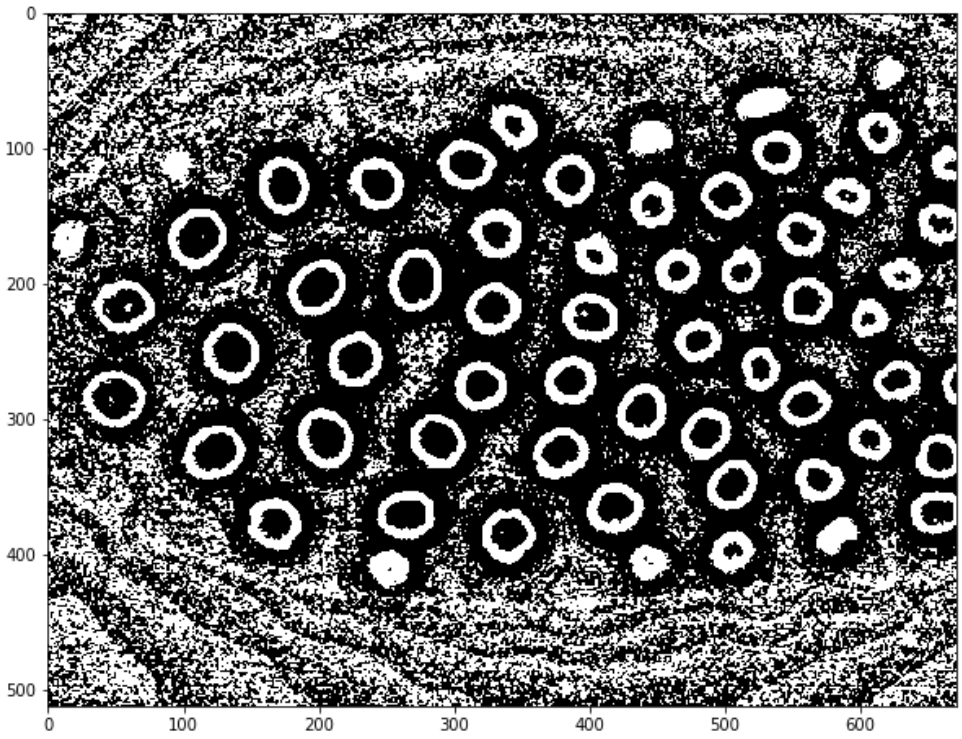
In [24]:
```
image_local_threshold = skf.threshold_local(image_median,block_size=51)
```

In [25]:
```
image_local_threshold.shape
```

Out[25]: (512, 672)

In [26]:
```
image_local_threshold = skf.threshold_local(image_median,block_size=51)
image_local = image_median > image_local_threshold
```

In [27]:
```
plt.figure(figsize=(10,10))
plt.imshow(image_local, cmap = 'gray')
plt.show()
```



We see that now each contour of the nuclei is recovered much better, however there is a lot of spurious background signal.

4.4 Note on higher-dimensional cases

Some functions of scikit-image are only designed for 2D images, and will generate an error message when used with 3D images. An alternative package to use in those cases is scipy and specifically scipy.ndimage and scipy.filtering

5. Binary operations, regions

Binary operations are an important class of functions to modify mask images (composed of 0's and 1's) and that are crucial when working segmenting images.

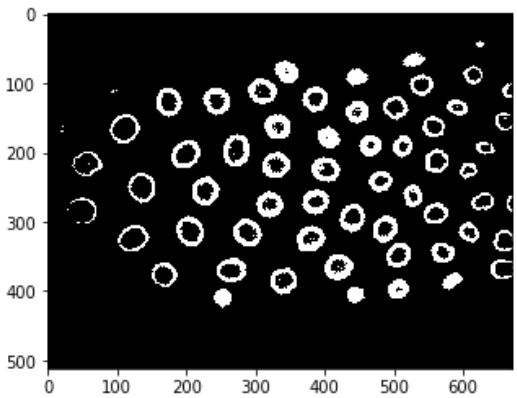
Let us first import the necessary modules:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray();
import skimage.io as io
from skimage.external.tifffile import TiffFile

import skimage.morphology as skm
import skimage.filters as skf
```

And we relaod the image from the last chapter and apply some thresholding to it:

In [2]:
```
#load image
data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()
image = skf.rank.median(image,selem=np.ones((2,2)))
image_otsu_threshold = skf.threshold_otsu(image)
image_otsu = image > image_otsu_threshold
plt.imshow(image_otsu);
```

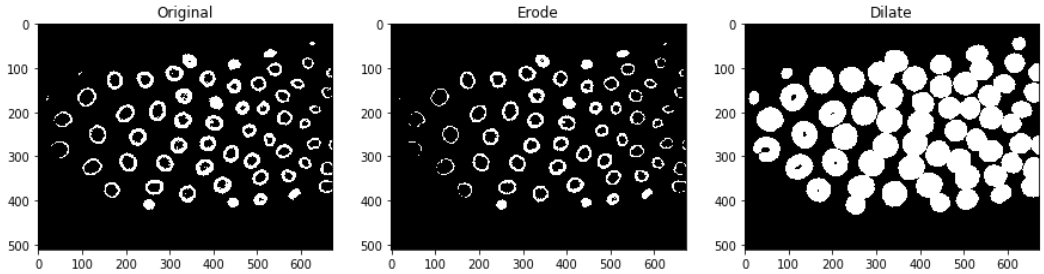
```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```



5.1 Binary operations

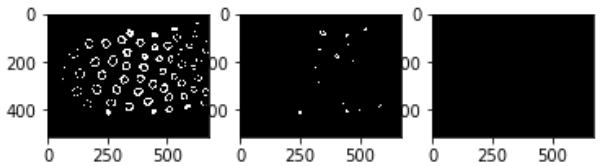
Binary operations assign to each pixel a value depending on it's neighborhood. For example we can erode or dilate the image using an area of radius 5. Erosion: If a white pixel has a black neighbor in its region it becomes black (erode). Dilation: any black pixel which as a white neighbour becomes white:

In [3]:
```
image_erode = skm.binary_erosion(image_otsu, selem = skm.disk(1))
image_dilate = skm.binary_dilation(image_otsu, selem = skm.disk(10))
plt.figure(figsize=(15,10))
plt.subplot(1,3,1)
plt.imshow(image_otsu,cmap = 'gray')
plt.title('Original')
plt.subplot(1,3,2)
plt.imshow(image_erode,cmap = 'gray')
plt.title('Erode')
plt.subplot(1,3,3)
plt.imshow(image_dilate,cmap = 'gray')
plt.title('Dilate');
```



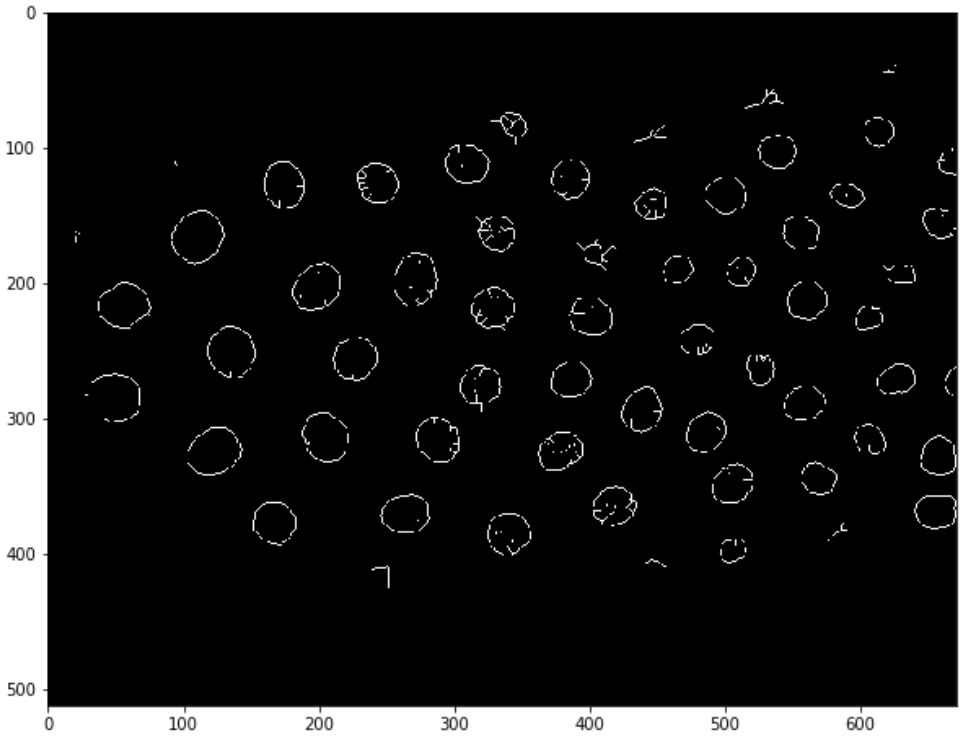
In [4]:
```
image_erode1 = skm.binary_erosion(image_otsu, selem = skm.disk(1))
image_erode1b = skm.binary_erosion(image_otsu, selem = skm.disk(5))
image_erode2 = skm.binary_erosion(image_otsu, selem = skm.disk(10))
plt.subplot(1,3,1)
plt.imshow(image_erode1,cmap = 'gray')
plt.subplot(1,3,2)
plt.imshow(image_erode1b,cmap = 'gray')
plt.subplot(1,3,3)
plt.imshow(image_erode2,cmap = 'gray')
```

Out[4]: `<matplotlib.image.AxesImage at 0x7f06ca6d8be0>`



If one is only interested in the path of those shapes, one can also thin them to the maximum:

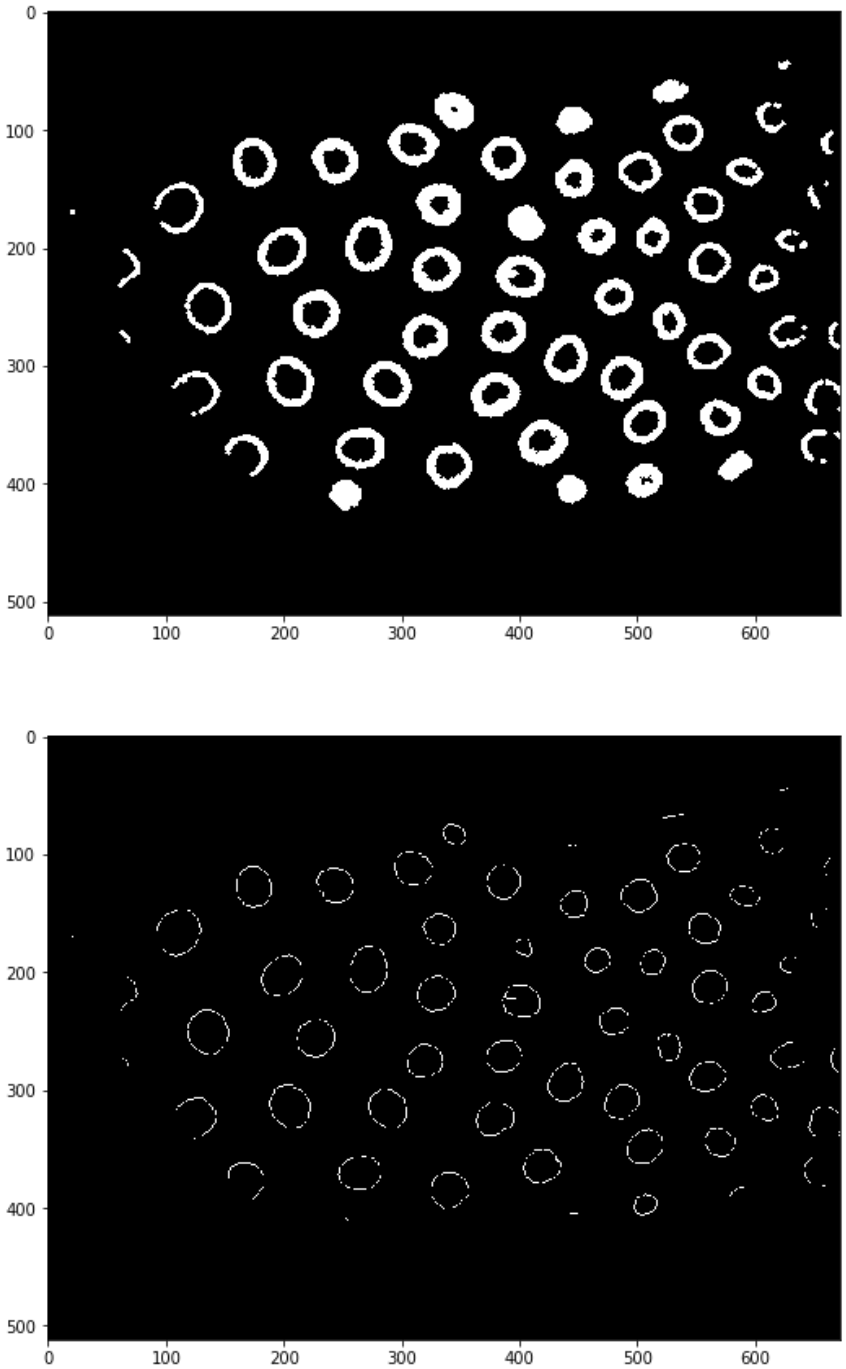
In [30]:
```
plt.figure(figsize=(10,10))
plt.imshow(skm.skeletonize(image_otsu));
```



Those operations can also be combined to "clean-up" an image. For example one can first erode the image to suppress isoltated pixels, and then dilate it again to restore larger structures to their original size. After that, the thinning operation gives a better result:

In [6]:
```
image_open = skm.binary_opening(image_otsu, selem = skm.disk(2))
image_thin = skm.skeletonize(image_open)
```

In [7]:
```
plt.figure(figsize=(15,15))
plt.subplot(2,1,1)
plt.imshow(image_open)
plt.subplot(2,1,2)
plt.imshow(image_thin);
```



The result of the segmentation is ok but we still have nuclei which are broken or not clean. Let's see if we can achieve a better result using another tool: region properties

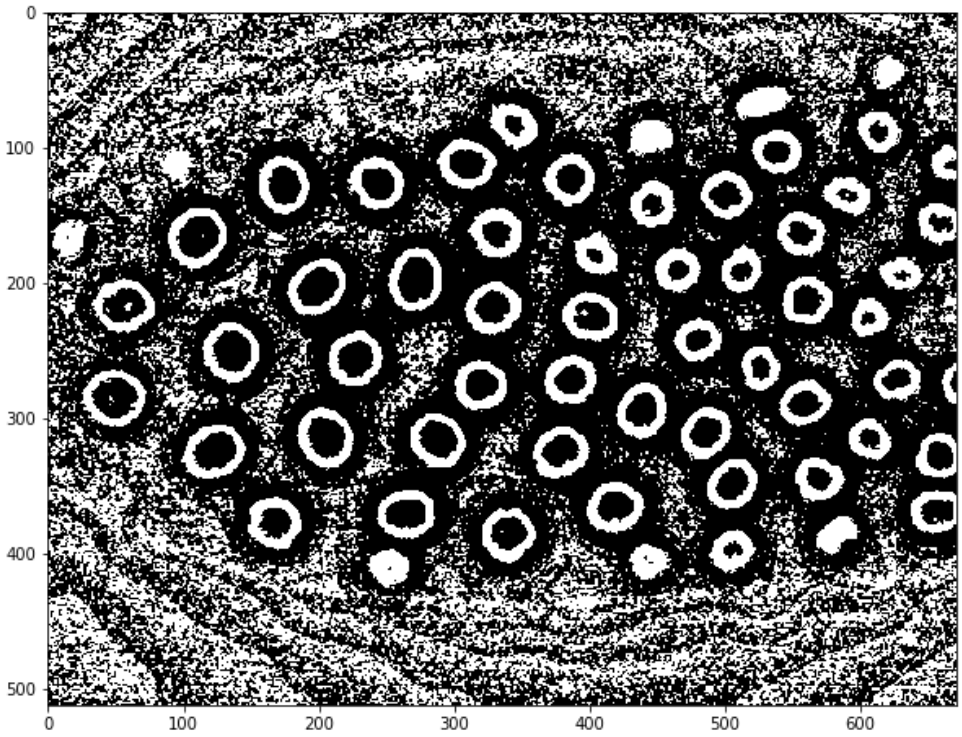
5.2 Region properties

In [8]:
```
from skimage.measure import label, regionprops
# MZ: labeling and region properties
# you have sth to segment (a mask); you want to mesure them individually
-> needs labeling
# 1 object for all pixels linked together, and label it (connected compo
nents)
```

When using binary masks, one can make use of functions that detect all objects (connected regions) in the image and calculate a list of properties for them. Using those properties, one can filer out unwanted objects more easily.

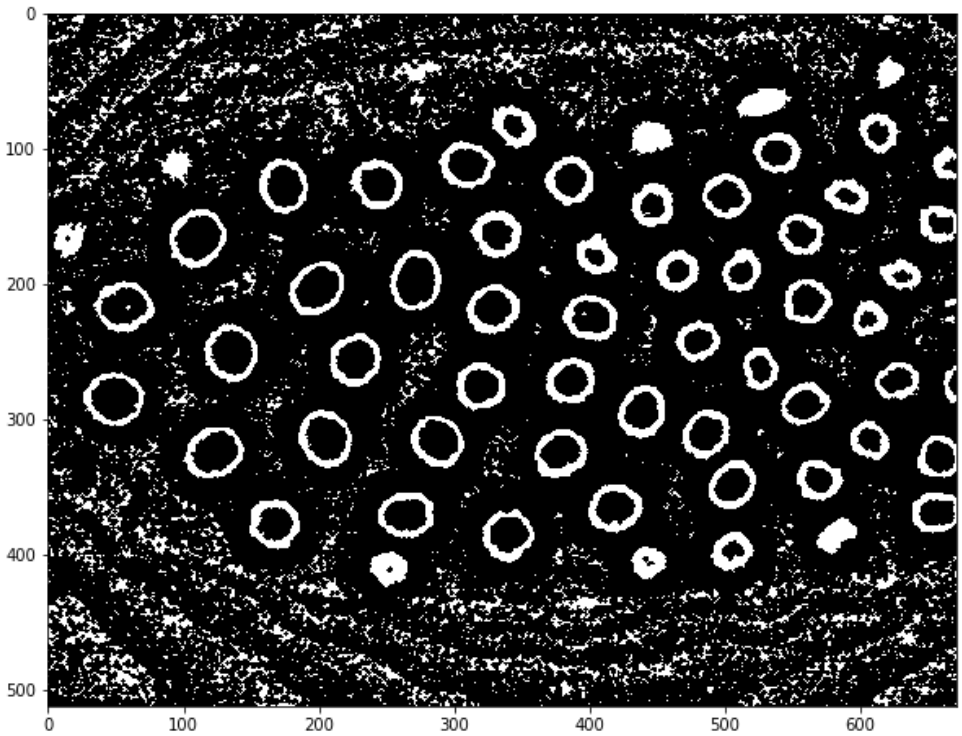
Thanks to this additional tool, we can now use the local thresholding method which preserved better all the nuclei but generated a lot of noise:

In [9]:
```
image_local_threshold = skf.threshold_local(image,block_size=51)
image_local = image > image_local_threshold

plt.figure(figsize=(10,10))
plt.imshow(image_local);
```



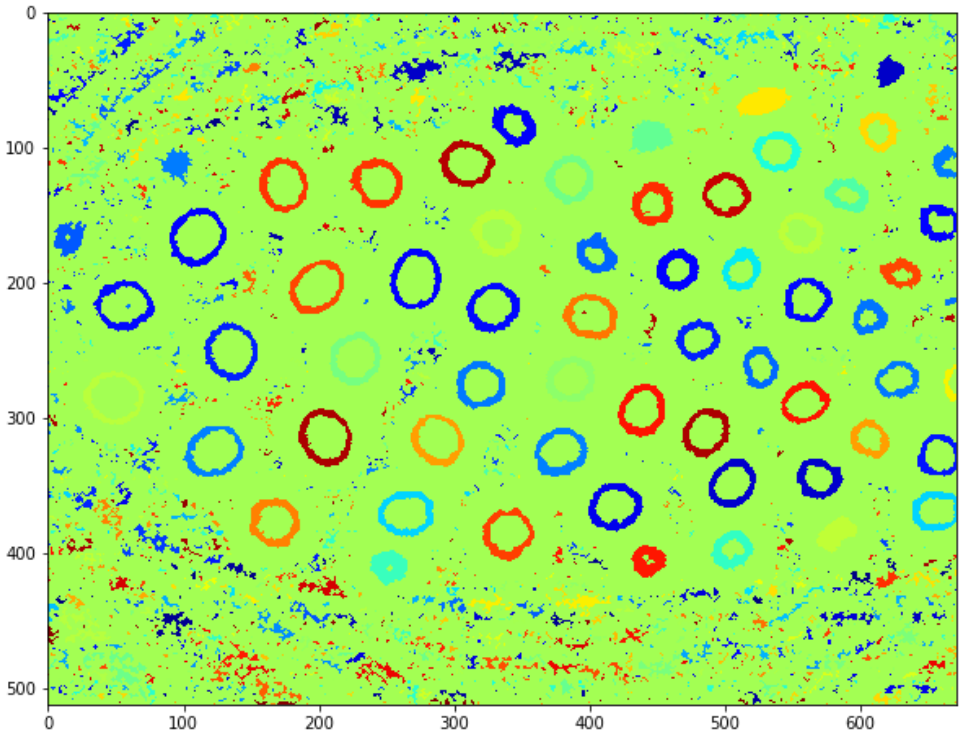
As the image id very noisy, there are a large number of small white regions, and applying the region functions on it will be very slow. So we first do some filtering and remove the smallest objects:

In [10]:
```
# MZ: still lot of noise !
# remove really small pixels with erosion
# to harsh erosion -> will remove also the patterns of interest, so only
soft erosion

image_local_eroded = skm.binary_erosion(image_local, selem= skm.disk(1))

plt.figure(figsize=(10,10))
plt.imshow(image_local_eroded);
```



To measure the properties of each region, we need a lablled image, i.e. an image in which each individual object is attributed a number. This is achieved usin the skimage.measure.label() function.

In [11]:
```
image_labeled = label(image_local_eroded)
# MZ: check all neighbors

#code snippet to make a random color scale
vals = np.linspace(0,1,256)
np.random.shuffle(vals)
cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

plt.figure(figsize=(10,10)) # MZ: to have bigger figure
plt.imshow(image_labeled,cmap = cmap);
```



In [12]:
```
image_labeled
# MZ: it is again an array
```

Out[12]:
```
array([[   0,    0,    0, ...,    0,    0,    0],
       [   0,    0,    0, ...,    0,    0,    0],
       [   0,    0,    0, ...,    0,    0,    0],
       ...,
       [   0,    0,    0, ...,    0,    0,    0],
       [   0,    0,    0, ...,    0,    0, 2894],
       [   0,    0,    0, ...,    0,    0,    0]])
```

In [13]:
```
image_labeled.max()
```

Out[13]:
```
2902
```

And now we can measure all the objects' properties

In [14]:
```
# MZ: now that we have regions -> we can use regionprops
# measure differences within each regions
# (we will get properties for each of the colored regions here above)
our_regions = regionprops(image_labeled)
len(our_regions)
```

Out[14]:
```
2902
```

We see that we have a list of 2902 regions. We can look at one of them more in detail and check what attributes exist:

In [15]:
```
# MZ: output is a list of structures, look at 1 element
our_regions[10]
```

Out[15]:
```
<skimage.measure._regionprops._RegionProperties at 0x7f06ca5b7b38>
```

In [29]:
```
# MZ: each region as a set of measurements associated with it
#dir(our_regions[10])
```

There are four types of information:

- geometric information on each shape (area, extent, perimeter, bounding box, etc.)
- vector information (pixel coordinates, centroid)
- region image information (average intensity, minimal intensity etc.)
- image-type information: the image enclosed in the bounding-box

Let us look at one region:

In [17]:
```
# MZ: a lot of other measurements (e.g. eccentricity, etc.)
our_regions[706].area
```

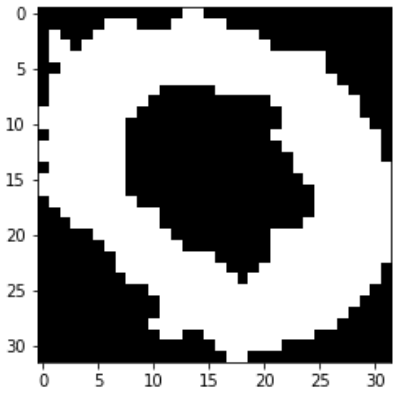
Out[17]:
```
526
```

In [18]:
```
# MZ: extract the image region that corresponds to the label
our_regions[706].image
```

Out[18]:
```
array([[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False,  True, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

In [19]:
```
print(our_regions[706].area)
print(our_regions[706].coords)

plt.imshow(our_regions[706].image);
```

```
526
[[ 69 342]
 [ 69 343]
 [ 70 335]
 ...
 [ 99 350]
 [100 346]
 [100 347]]
```

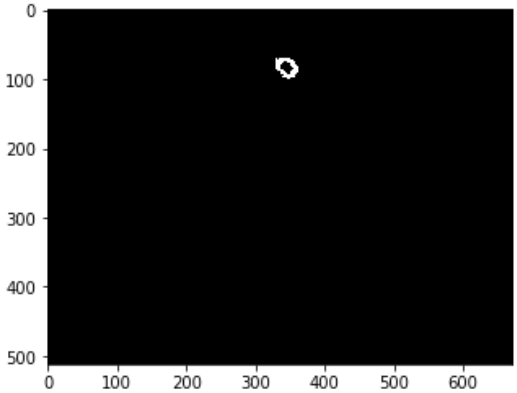


Using the coordinates information we can then for example recreate an image that contains only that region:

In [20]:
```
our_regions[706].coords
```

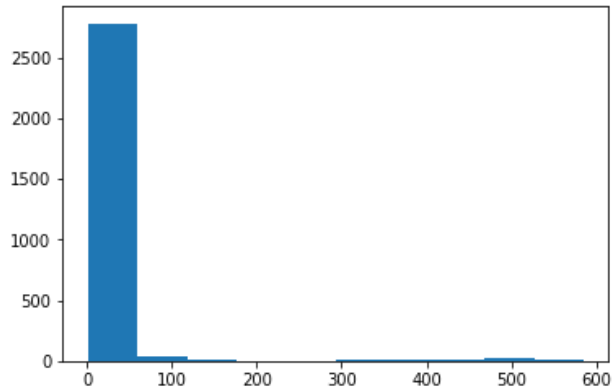
Out[20]:
```
array([[ 69, 342],
       [ 69, 343],
       [ 70, 335],
       ...,
       [ 99, 350],
       [100, 346],
       [100, 347]])
```

In [21]:
```
#create a zero image
newimage = np.zeros(image.shape)
#fill in using region coordinates
newimage[our_regions[706].coords[:,0],our_regions[706].coords[:,1]] = 1
#plot the result
plt.imshow(newimage);
```



In general, one has an idea about the properties of the objects that are interesting. For example, here we know that objects contain at least several tens of pixels. Let us recover all the areas and look at their distributions:

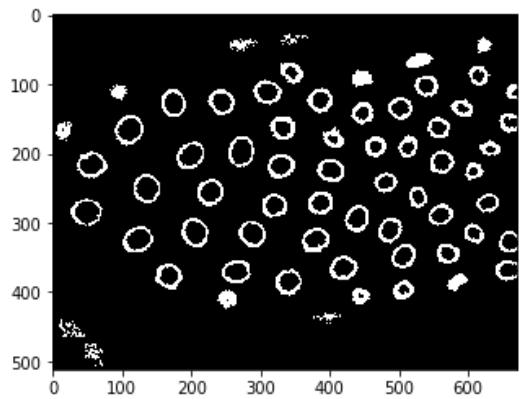
In [22]:
```
areas = [x.area for x in our_regions]
plt.hist(areas)
plt.show()
```



We see that we have a large majority of regions that are very small and that we can discard. Let's create a new image where we do that:

In [23]:
```
#create a zero image
newimage = np.zeros(image.shape)  # MZ: create 0-array, and then put 1 o
nlx where area > 200 (clean smaller stuff)
#fill in using region coordinates
for x in our_regions:
    if x.area>200:
        newimage[x.coords[:,0],x.coords[:,1]] = 1
#plot the result
plt.imshow(newimage)
# MZ: create a new image containing only the regions that have area > 20
0
```

Out[23]:
```
<matplotlib.image.AxesImage at 0x7f06ca59fd30>
```



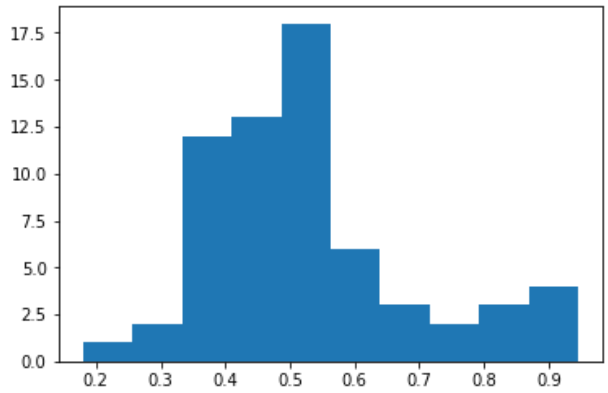
We see that we still have some spurious signal. We can measure again properties for the remaining regions and try to find another parameter for seleciton:

In [24]:
```
newimage_lab = label(newimage)
our_regions2 = regionprops(newimage_lab)
```

Most of our regions are circular, a property measures by the eccentricity. We can verifiy if we have outliers for that parameter:

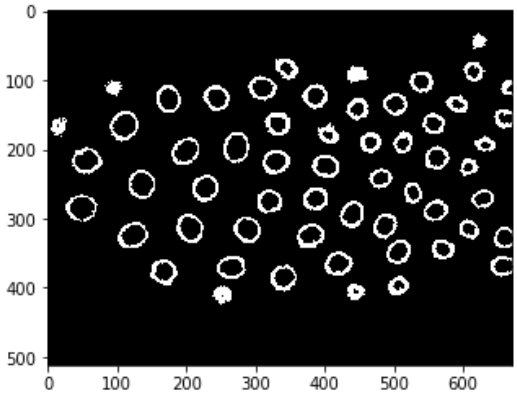
In [25]:
```
plt.hist([x.eccentricity for x in our_regions2]);
```

```
/usr/local/lib/python3.5/dist-packages/skimage/measure/_regionprops.py:25
0: UserWarning: regionprops and image moments (including moments, normali
zed moments, central moments, and inertia tensor) of 2D images will chang
e from xy coordinates to rc coordinates in version 0.16.
See http://scikit-image.org/docs/0.14.x/release_notes_and_installation.ht
ml#deprecations for details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
/usr/local/lib/python3.5/dist-packages/skimage/measure/_regionprops.py:26
0: UserWarning: regionprops and image moments (including moments, normali
zed moments, central moments, and inertia tensor) of 2D images will chang
e from xy coordinates to rc coordinates in version 0.16.
See http://scikit-image.org/docs/0.14.x/release_notes_and_installation.ht
ml#deprecations for details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
```



Let's discard regions that are too oblong (>0.8):

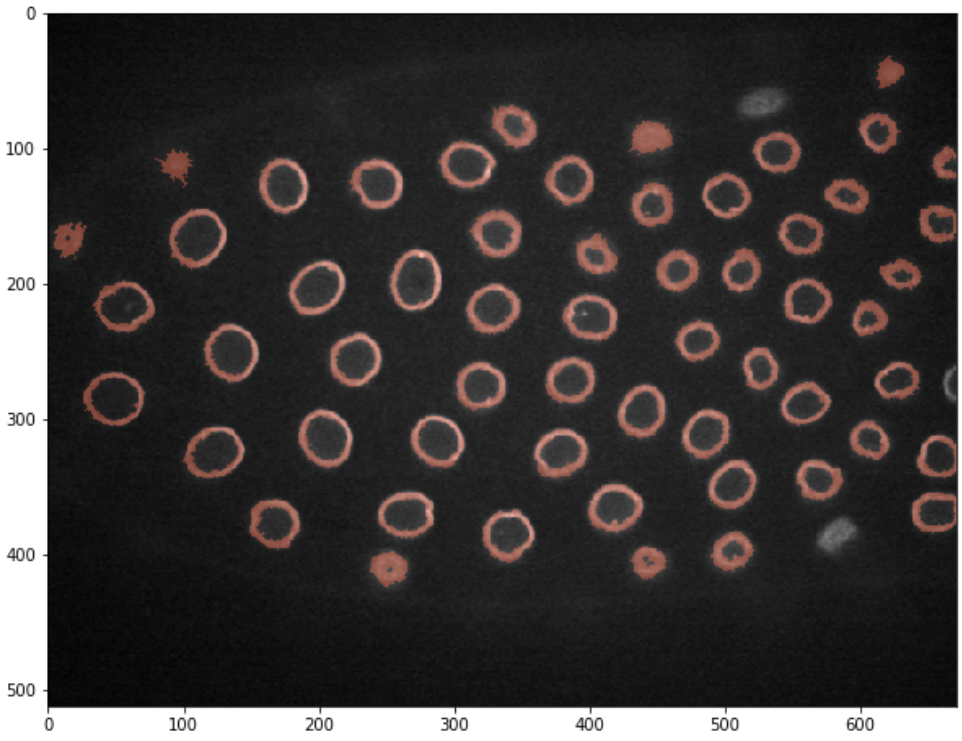
In [26]:
```
# MZ: now create a new image to clean up using eccentricity

#create a zero image
newimage = np.zeros(image.shape)

#fill in using region coordinates
for x in our_regions2:
    if x.eccentricity<0.8:
        newimage[x.coords[:,0],x.coords[:,1]] = 1

#plot the result
plt.imshow(newimage);
```



This is a success! We can verify how good the segmentation is by superposing it to the image. A trick to superpose a mask on top of an image without obscuring the image, is not set all 0 elements of the mask to `np.nan`.

In [27]:
```
newimage[newimage == 0] = np.nan
```

In [28]:
```
plt.figure(figsize=(10,10))
plt.imshow(image,cmap = 'gray')
plt.imshow(newimage,alpha = 0.4,cmap = 'Reds', vmin = 0, vmax = 2);
```



6. Applications: Satellite image

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
```

Looking at non-biology data

Most of this course focuses on biological data. To show the generality of the presented approaches, we show here short example based on satellite imagery.

Satellite imaging programs such as NASA's Landsat continuously image the earth and one can get retrieve data for free on several portals. We will deal here with images from a single region and use our basic image processing knowledge to do some vegetation analysis and image correction.

Let's first look at what a Landsat region data contains:

In [2]:
```
landsatfolder = 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/'
```

In [3]:
```
import glob
```

In [4]:
```
glob.glob(landsatfolder+'*tif')
```

Out[4]:
```
['Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band3_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band5_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band1_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band4_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_ipflag_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_cloud_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band6_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_cfmask_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band2_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_cfmask_conf_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band7_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_bqa_crop.tif']
```

The Landsat satellites acquires images in a series of wavelengths or "bands". Let us keep only those band files and sort them:

In [5]:
```
band_files = sorted(glob.glob(landsatfolder+'*band*tif'))
band_files
```

Out[5]:
```
['Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band1_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band2_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band3_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band4_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band5_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band6_crop.tif',
 'Data/geography/landsat/LC80340322016205-SC20170127160728/crop/LC80340322016205LGN00_sr_band7_crop.tif']
```

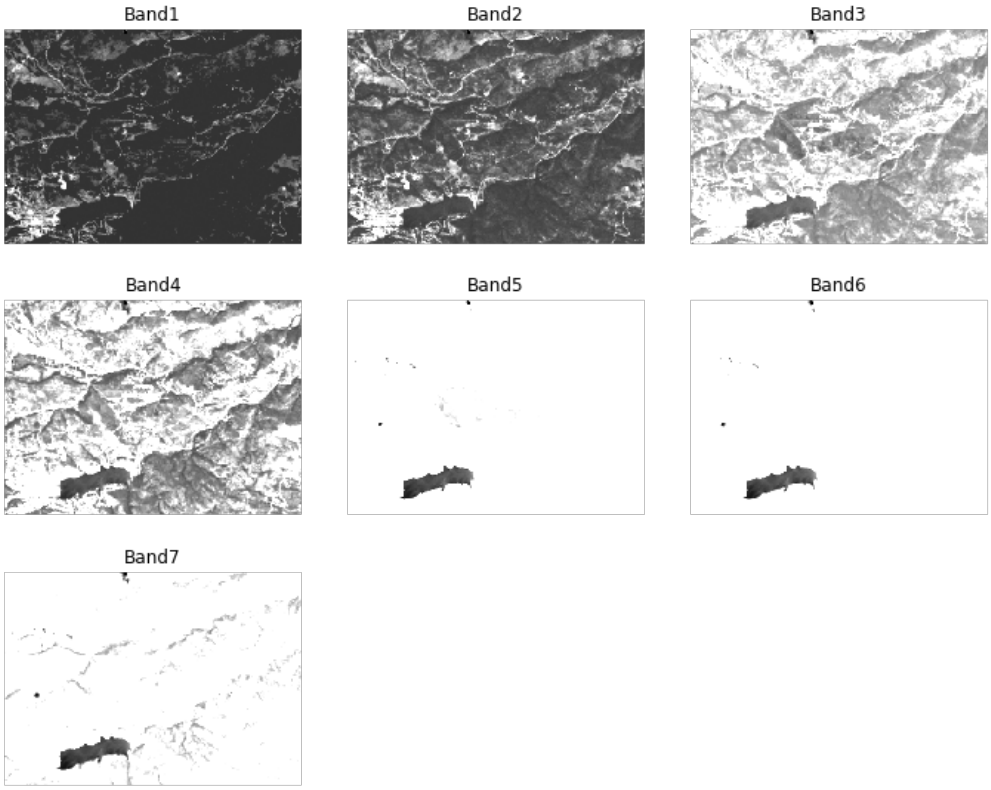
Now we can import all images and stack them into a Numpy array:

In [6]:
```
list_images = [io.imread(x) for x in band_files]
image_stack = np.stack(list_images)
image_stack.shape
```

Out[6]:
```
(7, 177, 246)
```

We see that we created an 3D array with the 7 different wavelength bands. Let's look at those:

In [7]:
```
fig, axarr = plt.subplots(3,3, figsize = (10,8))
for i in range(9):
    if i<7:
        axarr[int(i/3),np.mod(i,3)].imshow(image_stack[i,:,:],cmap = 'gr
ay', vmin=0, vmax = 500)
        axarr[int(i/3),np.mod(i,3)].set_title('Band'+str(i+1))
    axarr[int(i/3),np.mod(i,3)].axis('off')

fig.tight_layout(h_pad = 0, w_pad = 0)
```



From the Landsat information we know that bands 4,3 and 2 are RGB. So let's select those to create a natural image and try plotting it as RGB image:

In [8]:
```
image_stack.shape
```

Out[8]: (7, 177, 246)

In [9]:
```
image_RGB = image_stack[[3,2,1],:,:]
```

In [10]:
```
image_RGB.shape
```

Out[10]: (3, 177, 246)

In [13]:
```
# plt.imshow(image_RGB)
# plt.show()
# # TypeError: Invalid dimensions for image data
```

Oups, the dimensions are not correct:

In [14]:
```
image_RGB.shape
```

Out[14]:
```
(3, 177, 246)
```

We created a stack where the leading dimension are the different bands. However in the RGB format, the different colors are the last dimension! So we have to move the first axis to the end to be able to plot it:

In [15]:
```
np.moveaxis(image_RGB,0,2).shape
```

Out[15]:
```
(177, 246, 3)
```

In [41]:
```
# plt.imshow(np.moveaxis(image_RGB,0,2))
# plt.show()
# Clipping input data to the valid range for imshow with RGB data ([0..
1] for floats or [0..255] for integers).
```

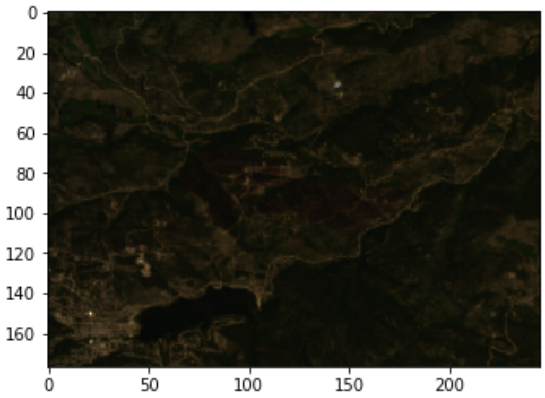
In [43]:
```
image_RGB
```

Out[43]:
```
array([[[535, 597, 576, ..., 242, 279, 281],
        [483, 547, 549, ..., 283, 321, 364],
        [436, 424, 432, ..., 324, 399, 481],
        ...,
        [667, 832, 854, ..., 425, 413, 433],
        [985, 745, 764, ..., 372, 385, 397],
        [455, 415, 352, ..., 388, 380, 384]],

       [[514, 537, 525, ..., 311, 338, 364],
        [488, 516, 510, ..., 327, 354, 407],
        [484, 490, 463, ..., 364, 411, 477],
        ...,
        [594, 727, 701, ..., 403, 403, 409],
        [738, 662, 710, ..., 364, 401, 425],
        [429, 354, 277, ..., 353, 375, 413]],

       [[263, 300, 292, ..., 141, 158, 156],
        [238, 268, 275, ..., 148, 172, 176],
        [203, 208, 209, ..., 163, 172, 188],
        ...,
        [303, 429, 392, ..., 183, 172, 189],
        [443, 314, 410, ..., 169, 170, 179],
        [230, 188, 118, ..., 162, 164, 166]]], dtype=int16)
```

Next problem: the values of the pixels are not between 0-1 (floats) or 0-255 (ints). So we have to correct for that. We could do it manually, but skimage has some function to help us where we can say what should be the output scale:

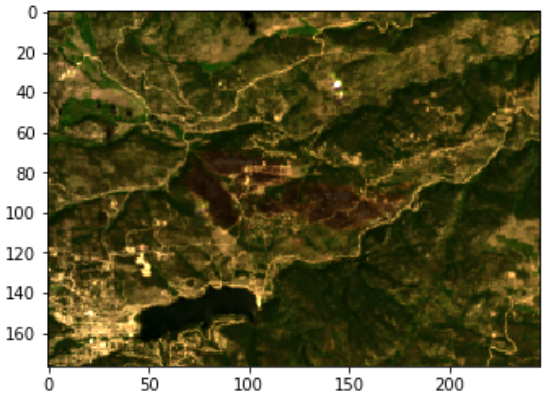
In [18]:
```
from skimage.exposure import rescale_intensity
```

In [19]:
```
plt.imshow(rescale_intensity(np.moveaxis(image_RGB,0,2), out_range = (0,
1)));
```



Now it starts looking like something reasonable. However the exposure is still not optimal. Let's clip values around the the dimmest and brightest pixels and pass that as an argument to the rescaling function:

In [20]:
```
v_min, v_max = np.percentile(image_RGB, (0.2, 99.8))
plt.imshow(rescale_intensity(np.moveaxis(image_RGB,0,2),in_range=(v_min,
v_max), out_range=(0,1)));
```



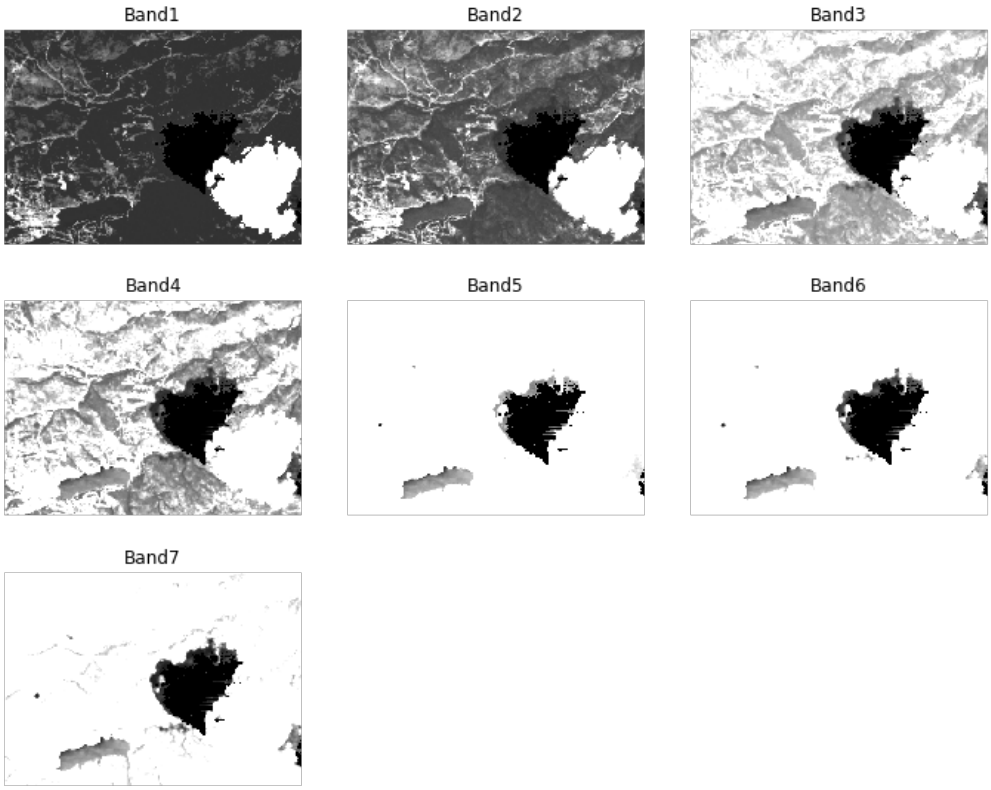
So that's much better. Note that we don't modify the image data. We just use the correcting functions within the plotting function. Indeed, we only want to improve the visual impression, not change the underlaying data.

Let us look at the images of the other day provided in the data for which we have the same bands:

In [21]:
```
landsatfolder = 'Data/geography/landsat/LC80340322016189-SC2017012809115
3/crop/'
band_files = sorted(glob.glob(landsatfolder+'*band*tif'))
band_files
```

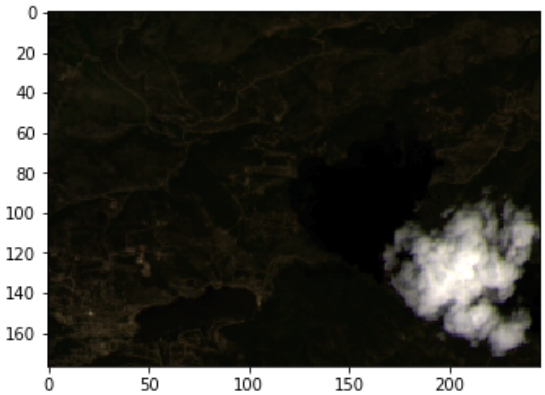
Out[21]:
```
['Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band1_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band2_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band3_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band4_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band5_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band6_crop.tif',
 'Data/geography/landsat/LC80340322016189-SC20170128091153/crop/LC8034032
2016189LGN00_sr_band7_crop.tif']
```

In [22]:
```
list_images = [io.imread(x) for x in band_files]
image_stack2 = np.stack(list_images)

fig, axarr = plt.subplots(3,3, figsize = (10,8))
for i in range(9):
    if i<7:
        axarr[int(i/3),np.mod(i,3)].imshow(image_stack2[i,:,:],cmap = 'g
ray', vmin=0, vmax = 500)
        axarr[int(i/3),np.mod(i,3)].set_title('Band'+str(i+1))
    axarr[int(i/3),np.mod(i,3)].axis('off')

fig.tight_layout(h_pad = 0, w_pad = 0)
```



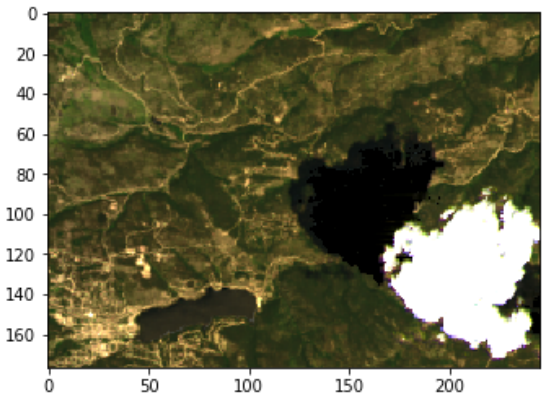
We see that there is a cloud in the image. In addition the cloud is casting a shadow. If our goal was to compare the evolution of the vegetation between these two days, we would somehow have to remove those areas from our dataset. Let's first try to plot our image in real colors:

In [23]:
```
image_RGB = image_stack2[[3,2,1],:,:]
v_min, v_max = np.percentile(image_RGB, (0.2, 99.8))
plt.imshow(rescale_intensity(np.moveaxis(image_RGB,0,2).astype(float),in
_range=(v_min, v_max),out_range=(0, 1)))
plt.show()
```



Because the cloud is so bright, the exposure in the rest of the image is really dim. We can manually clip the maximal values to be able to visualize our data:

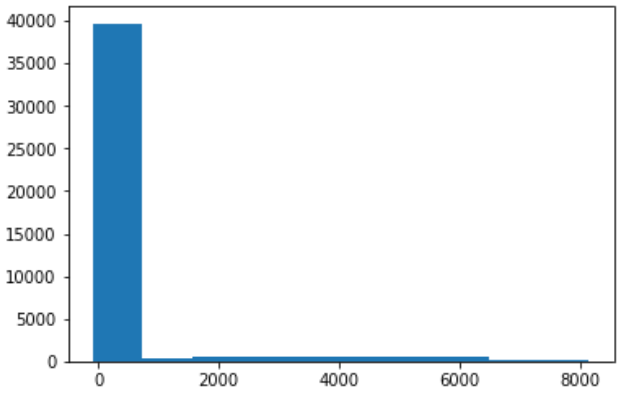
In [24]:
```
plt.imshow(rescale_intensity(np.moveaxis(image_RGB,0,2).astype(float),in
_range=(v_min, 0.2*v_max),out_range=(0, 1)))
plt.show()
```



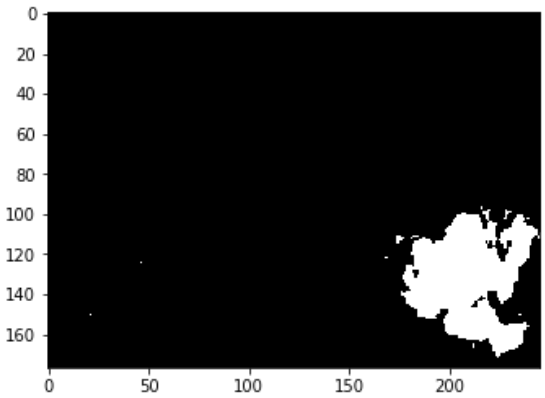
Now let us try to remove the cloud and it's shadow. Fortunately, we see that in band1 the clouds clearly appear as much brighter than the rest of the image. The histogram shows that most pixels are below ~1000. To avoid picking a value manually we can use the Otsu threshold and verify our mask

In [25]:
```
from skimage.filters import threshold_otsu
```

In [26]:
```
plt.hist(np.ravel(image_stack2[0,:,:]))#, bins = np.arange(0,20000,100))
plt.show()
```

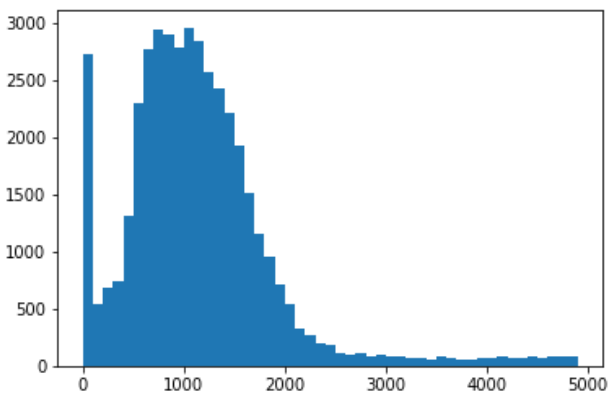


In [27]:
```
otsu_th = threshold_otsu(image_stack2[0,:,:])
plt.imshow(image_stack2[0,:,:]>otsu_th,cmap = 'gray')
plt.show()
```

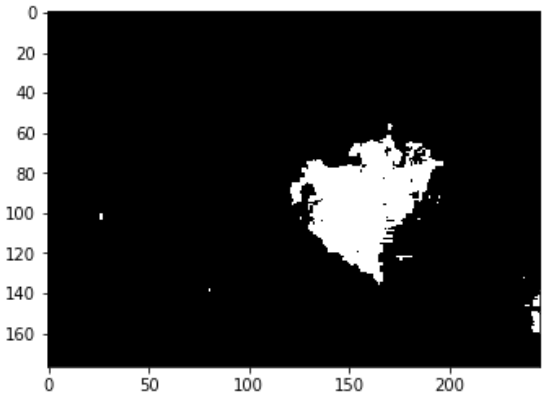


The shadow on the other side, appears as a clear dark region in band 7. The histogram shows clearly that we have a set of pixels that have been clipped in the lower range. If we create a maks just above, we get:

In [28]:
```
plt.hist(np.ravel(image_stack2[6,:,:]), bins = np.arange(0,5000,100))
plt.show()
```



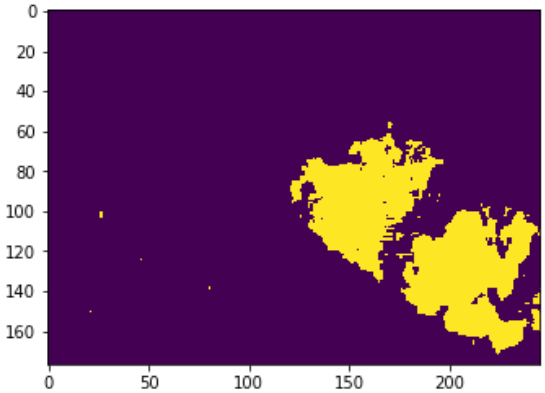
In [29]:
```
plt.imshow(image_stack2[6,:,:]<100,cmap = 'gray')
plt.show()
```



Now we have two masks that we can combine into one logical mask using Numpy logical operations

In [30]:
```
global_mask = (image_stack2[0,:,:]>otsu_th) | (image_stack2[6,:,:]<100)
```

In [31]:
```
plt.imshow(global_mask)
plt.show()
```

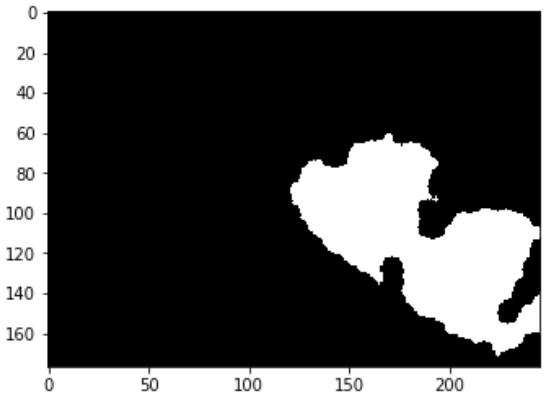


We do in addition one round of binary closing/opening to close holes in our maks and remove small pixels:

In [32]:
```
from skimage.morphology import binary_closing, disk, binary_opening
```

In [33]:
```
global_mask = binary_opening(binary_closing(global_mask, selem=disk(5)),
selem= disk(1))
```

In [34]:
```
plt.imshow(global_mask, cmap = 'gray')
plt.show()
```

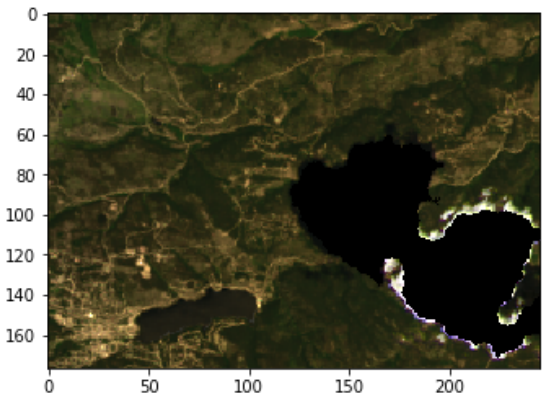


We can now apply the mask to our entire image stack, and use the fact that the 2D mask will be reproduced along the leading dimension of the stack

In [35]:
```
image_stack2_masked = image_stack2*~global_mask
```

Normally now we should be able to plot our RGB image without having to correct for the very bright cloud pixels:

In [36]:
```
image_RGB = image_stack2_masked[[3,2,1],:,:]
v_min, v_max = np.percentile(image_RGB, (0.2, 99.8))
plt.imshow(rescale_intensity(np.moveaxis(image_RGB,0,2).astype(float),in
_range=(v_min, v_max),out_range=(0, 1)));
```



Calculating the effect of fire

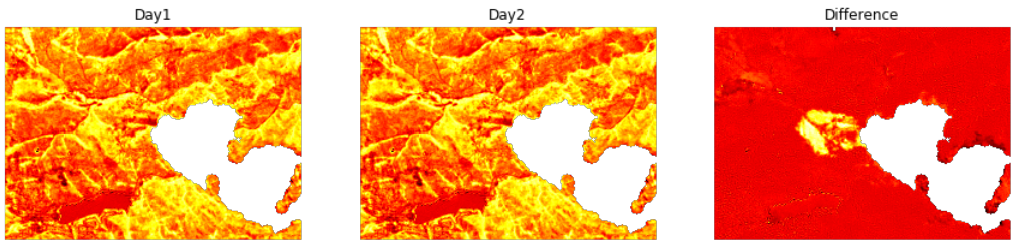
By comparing two channels reflecting vegetation areas and burned/earth areas, we can estimate where fire caused dammage. One typical value that is measured is Band5-Band7/(Band5+Band7)

In [37]:
```
burn_day1 = (image_stack2_masked[4]-image_stack2_masked[6])/(image_stack
2_masked[4]+image_stack2_masked[6])
burn_day2 = (image_stack[4]-image_stack[6])/(image_stack[4]+image_stack
[6])
difference = burn_day1-burn_day2

# MZ: to compare the 2 images to see where it has burnt
```

```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:1: RuntimeWa
rning: invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:2: RuntimeWa
rning: invalid value encountered in true_divide
```

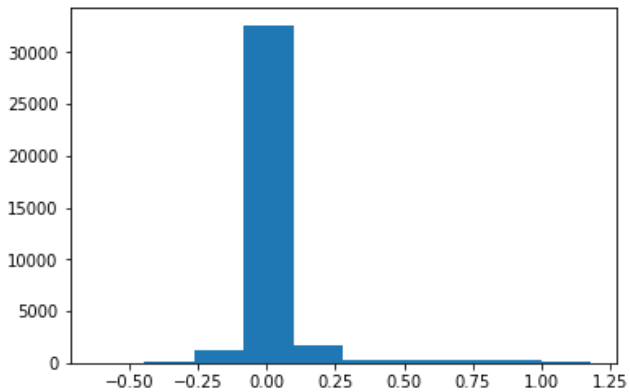
In [38]:
```
f, axarr = plt.subplots(1,3, figsize= (15,10))
axarr[0].imshow(burn_day1,cmap = 'hot')
axarr[0].axis('off')
axarr[0].set_title('Day1')
axarr[1].imshow(burn_day1,cmap = 'hot')
axarr[1].axis('off')
axarr[1].set_title('Day2')
axarr[2].imshow(difference,cmap = 'hot')
axarr[2].axis('off')
axarr[2].set_title('Difference')
```

Out[38]: Text(0.5, 1.0, 'Difference')



In [39]:
```
plt.hist(np.ravel(difference));
```

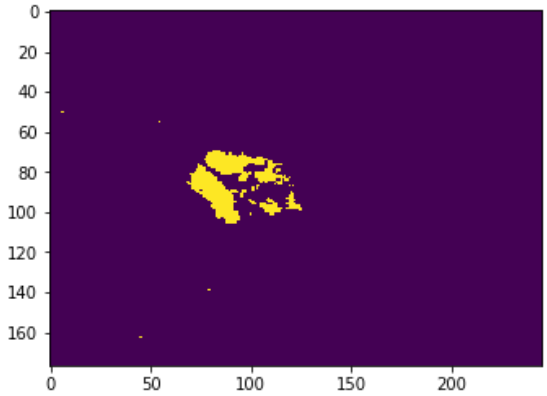
```
/usr/local/lib/python3.5/dist-packages/numpy/lib/histograms.py:754: Runti
meWarning: invalid value encountered in greater_equal
  keep = (tmp_a >= first_edge)
/usr/local/lib/python3.5/dist-packages/numpy/lib/histograms.py:755: Runti
meWarning: invalid value encountered in less_equal
  keep &= (tmp_a <= last_edge)
```



In [40]:
```
plt.imshow(difference>0.5)
```

```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:1: RuntimeWa
rning: invalid value encountered in greater
  """Entry point for launching an IPython kernel.
```

Out[40]:
```
<matplotlib.image.AxesImage at 0x7f65240aaef0>
```



7. Functions

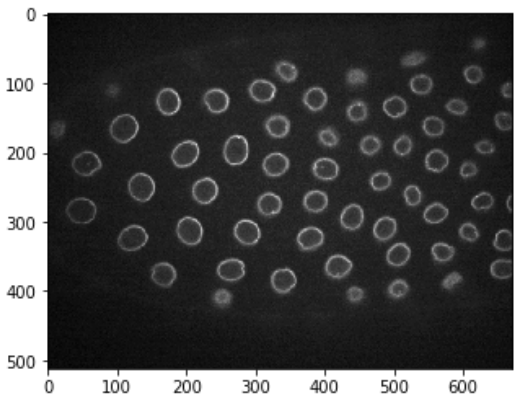
In the previous chapter we developped a small procedure to segment our image of nuclei. If you develop such a routine you are going to re-use it multiple times, so it makes sense to package it into a re-usable unit.

We will summarize here how to achieve that in this brief chapter.

In [1]:
```
#importing packages
import numpy as np
import matplotlib.pyplot as plt
plt.gray();

from skimage.external.tifffile import TiffFile

import skimage.morphology as skm
import skimage.filters as skf
```

In [2]:
```
#load the image to process
data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()
```

In [3]:
```
plt.imshow(image);
```



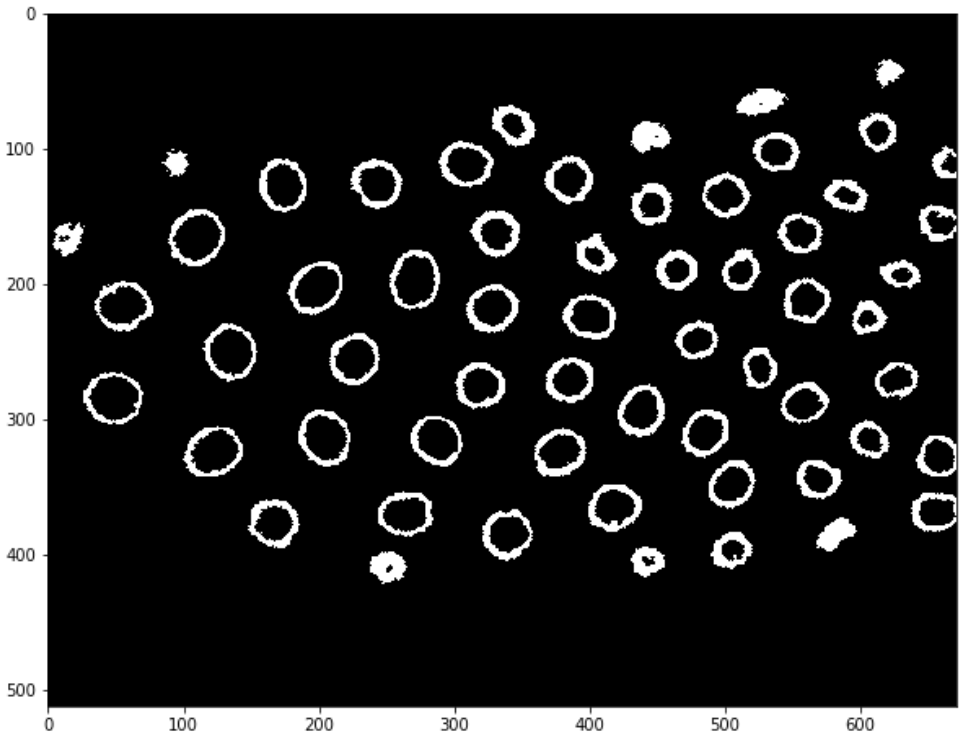
Let us summarize all the necessary steps within one code block

In [4]:
```
from skimage.measure import label, regionprops

#median filter
image_med = skf.rank.median(image,selem=np.ones((2,2)))
#otsu thresholding
image_local_threshold = skf.threshold_local(image_med,block_size=51)
image_local = image > image_local_threshold
#remove tiny features
image_local_eroded = skm.binary_erosion(image_local, selem= skm.disk(1))
#label image
image_labeled = label(image_local_eroded)
#analyze regions
our_regions = regionprops(image_labeled)
#create a new mask with constraints on the regions to keep
newimage = np.zeros(image.shape)
#fill in using region coordinates
for x in our_regions:
    if (x.area>200):# and (x.eccentricity<0.8):
        newimage[x.coords[:,0],x.coords[:,1]] = 1
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```

In [5]:
```
plt.figure(figsize=(10,10))
plt.imshow(newimage)
```

Out[5]:
```
<matplotlib.image.AxesImage at 0x7fcb880eeb38>
```



We can now make a function out of it. You can choose the "level" of your function depending on your needs. For example you could pass a filename and a plane index to the function and make it import your data, or you can pass directly an image.

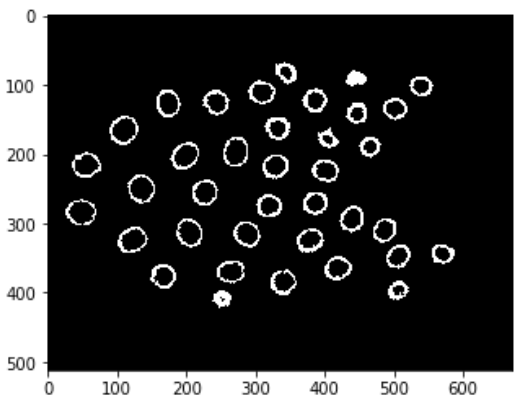
In addition to the image, you coud pass other arguments if you want to make your function more general. For example, you might not always want to filter objects of the same size or shape, and so you can set those as parameters:

In [6]:
```
from skimage.measure import label, regionprops

def detect_nuclei(image, size = 200, shape = 0.8):
    #median filter
    image_med = skf.rank.median(image,selem=np.ones((2,2)))
    #otsu thresholding
    image_local_threshold = skf.threshold_local(image_med,block_size=51)
    image_local = image > image_local_threshold
    #remove tiny features
    image_local_eroded = skm.binary_erosion(image_local, selem= skm.disk
(1))
    #label image
    image_labeled = label(image_local_eroded)
    #analyze regions
    our_regions = regionprops(image_labeled)
    #create a new mask with constraints on the regions to keep
    newimage = np.zeros(image.shape)
    #fill in using region coordinates
    for x in our_regions:
        if (x.area>size) and (x.eccentricity<shape):
            newimage[x.coords[:,0],x.coords[:,1]] = 1

    return newimage
```

And now we can test the function (which appears also now in autocompletion):

In [7]:
```
nuclei = detect_nuclei(image, size = 400)
plt.imshow(nuclei);
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
/usr/local/lib/python3.5/dist-packages/skimage/measure/_regionprops.py:25
0: UserWarning: regionprops and image moments (including moments, normali
zed moments, central moments, and inertia tensor) of 2D images will chang
e from xy coordinates to rc coordinates in version 0.16.
See http://scikit-image.org/docs/0.14.x/release_notes_and_installation.ht
ml#deprecations for details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
/usr/local/lib/python3.5/dist-packages/skimage/measure/_regionprops.py:26
0: UserWarning: regionprops and image moments (including moments, normali
zed moments, central moments, and inertia tensor) of 2D images will chang
e from xy coordinates to rc coordinates in version 0.16.
See http://scikit-image.org/docs/0.14.x/release_notes_and_installation.ht
ml#deprecations for details on how to avoid this message.
  warn(XY_TO_RC_DEPRECATION_MESSAGE)
```



In order to avoid cluttering your notebooks with function definitions and to be able to reuse your functions across multiple notebooks, I also strongly advise you to create your own module files. Those are .py files that group multipe functions and that can be called from any notebook.

Let's create one, call it my_module.py and copy our function in it. Now we can use the function like this:

In [8]:
```
import my_module
#or alternatively: from my_module import detect_nuclei
```

```
---------------------------------------------------------------------------
--
ImportError                               Traceback (most recent call las
t)
<ipython-input-8-a9447689b240> in <module>()
----> 1 import my_module
      2 #or alternatively: from my_module import detect_nuclei

ImportError: No module named 'my_module'
```

In [ ]:
```
nuclei2 = my_module.detect_nuclei(image)
```

We get an error because in that module, we use skimage functions that were not imported **in the module itself**. We have them in the notebook, but they are not accessible from there. We thus restart the kernel as re-loading a module doesn't work:

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray();
from skimage.external.tifffile import TiffFile

data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()

import my_module
nuclei2 = my_module.detect_nuclei(image)
```

In [ ]:
```
plt.imshow(nuclei2);
```

Your own modules are accessible if they are in the same folder as your notebook or on some path recognized by Python (on the PYTHONPATH). For more details see here (https://docs.python.org/3.3/tutorial/modules.html).

8. Pattern matching, local maxima

Sometimes threholding and binary operations are not appropriate tools to segment image features. This is particularly true when the object to be detected has as specific shape but a very variable intensity or if the image has low contrast. In that case it is useful to attempt to build a "model" of the object and look for similar shapes in the image. It is very similar in essence to convolution, however the operation is normalized so that after filtering every pixel is assigned a value between -1 (anti-correlation) to +1 perfect correlation. One can then look for local matching maxima to identify objects.

In [1]:
```
from skimage.feature import match_template, peak_local_max
import skimage.io as io
```

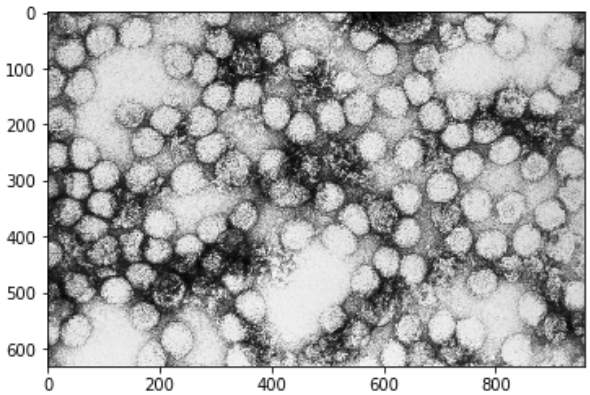
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray()
from skimage.external.tifffile import TiffFile
```

8.1 Virus on electron microscopy

Electron microscopy is a typical case where pixel intensity cannot be directly used for segmentation. For example in the following picture of a virus, even though we see the virus as white disks, many other regions are as bright.

In [3]:
```
#load the image to process
image = io.imread('http://res.publicdomainfiles.com/pdf_view/29/13512183
019720.jpg')
#image = io.imread('http://res.publicdomainfiles.com.s3.amazonaws.com/pd
f_alternate/29/13512183019720.tif?AWSAccessKeyId=AKIAJBE24BKMOLMJBBXA&Ex
pires=1579466193&Signature=uMi8UqvJbUX2mGkgZuEGAx6J6r4%3D')
```

In [4]:
```
plt.imshow(image);
```

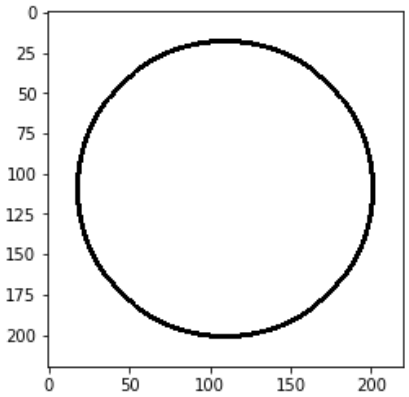


What is unique to the virus is the shape of the objects. So let's try to make a model of them to do template matching. Essentially a virus appears as a white disk surrounded by a thin dark line:

In [5]:
```
radius = 90

template = np.zeros((220,220))
center = [(template.shape[0]-1)/2,(template.shape[1]-1)/2]
Y, X = np.mgrid[0:template.shape[0],0:template.shape[1]]
dist_from_center = np.sqrt((X - center[0])**2 + (Y-center[1])**2)
template[dist_from_center<=radius] = 1
template[dist_from_center>radius+3] = 1

# MZ: identify all areas in the image that match the pattern of your int
erest
```

In [6]:
```
plt.imshow(template)
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x7fc3bea62d68>
```



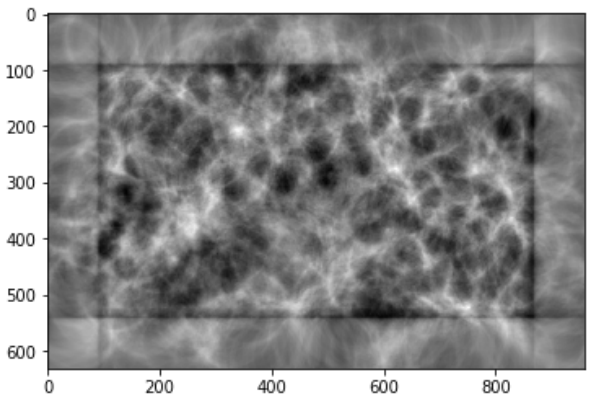
Now we do the template matching. Note that we specify the option `pad_input` to make sure the coordinates of the local maxima is not affected by boreder effects (try to turn it to `False` to see the effecf):

In [7]:
```
matched = match_template(image=image, template=template, pad_input=True)
```

And this is how the matched image looks like. Wherever there's a particle a local maximum appears.

In [8]:
```
plt.imshow(matched)
```

Out[8]:
```
<matplotlib.image.AxesImage at 0x7fc3bd1ae048>
```

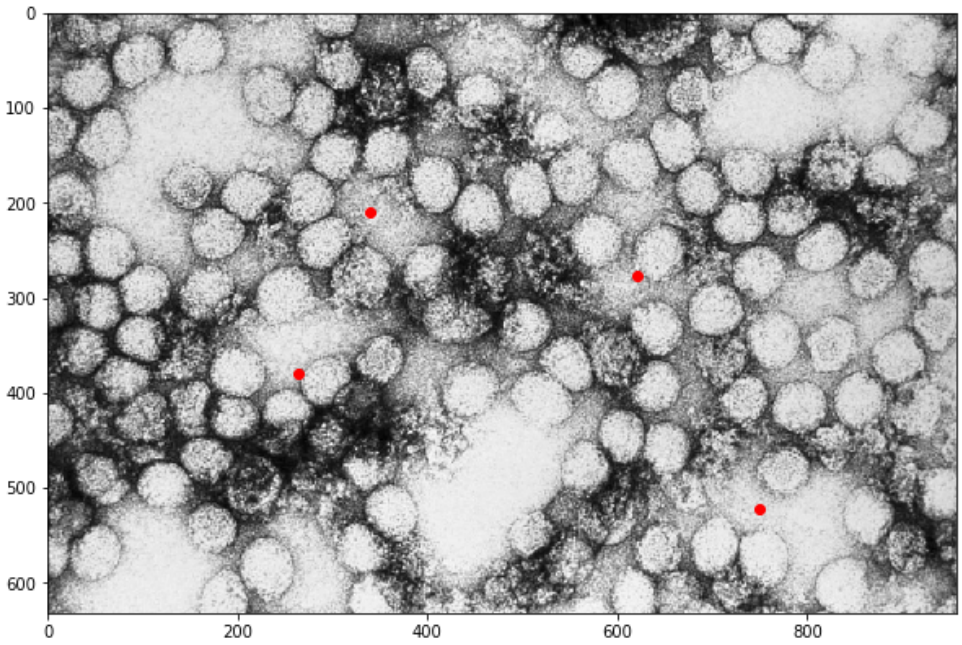


We can try to detect the local maxima to have the position of each particle. For that we use the scipy `peak_local_max` function. We specify that two maximia cannot be closer than 20 pixels (`min_distance`) and we also set a threshold on the quality of matching (`threshold_abs`). We also want to recover a list of indices rather than a binary mask of local maxima.

In [9]:
```
local_max_indices = peak_local_max(matched, min_distance=60,indices=Tru
e, threshold_abs=0.1)
```

Finally we can plot the result:

In [10]:
```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.plot(local_max_indices[:,1],local_max_indices[:,0],'ro')
plt.show()
```



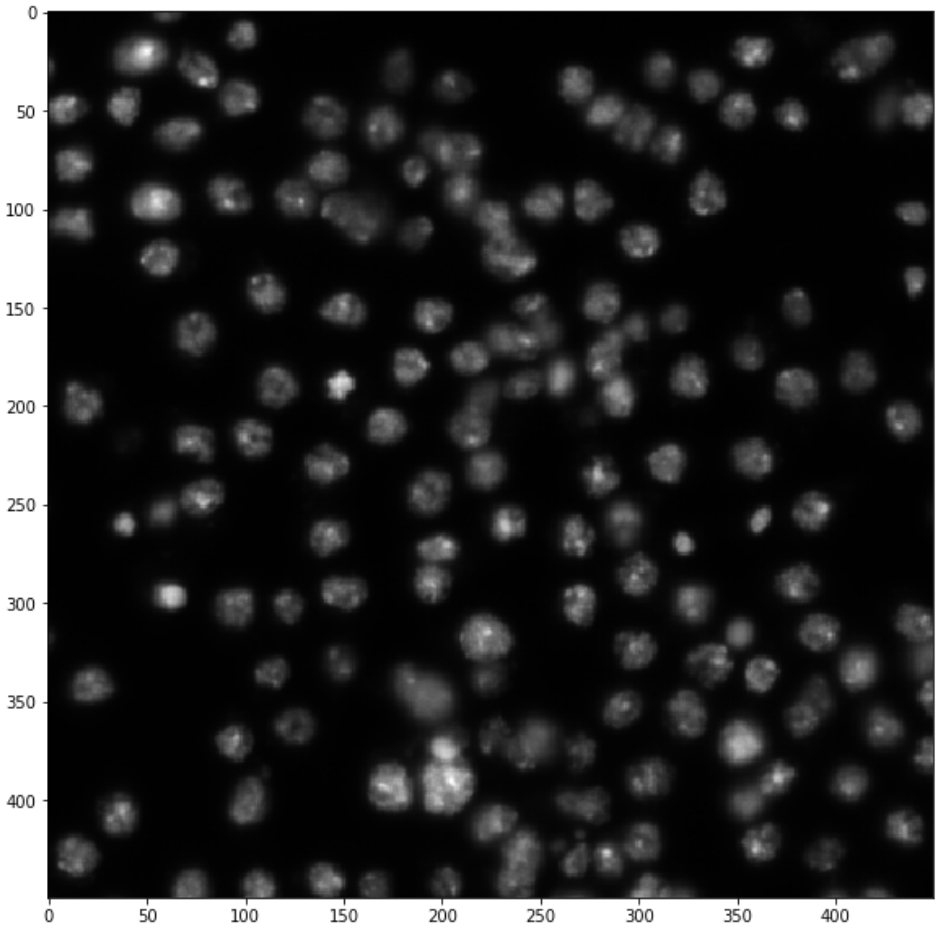
8.2 Fluorescence microscopy

In the following example we are looking at a nuclei imaged by fluorescence microscopy. Here, intensity can clearly be used for segmentation but is going to lead to merged objects when they are too close. To identify each nucleus in a first step before actual segmentation, we can again use template matching.

In [11]:
```
import skimage.io as io
```

In [12]:
```
image = io.imread('Data/BBBC007_v1_images/A9/A9 p9d.tif')
```

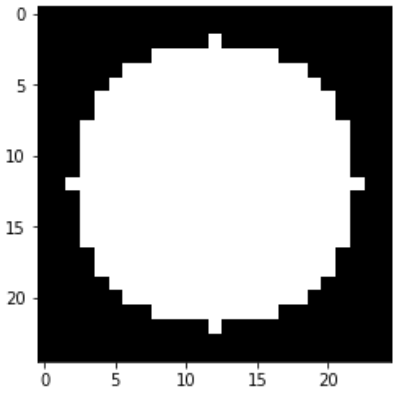
In [13]:
```
plt.figure(figsize=(10,10))
plt.imshow(image);
```



In this image, nuclei have radius of around 10 pixels. We can generate again a template:

In [14]:
```
radius = 10

template = np.zeros((25,25))
center = [(template.shape[0]-1)/2,(template.shape[1]-1)/2]
Y, X = np.mgrid[0:template.shape[0],0:template.shape[1]]
dist_from_center = np.sqrt((X - center[0])**2 + (Y-center[1])**2)
template[dist_from_center<=radius] = 1
```

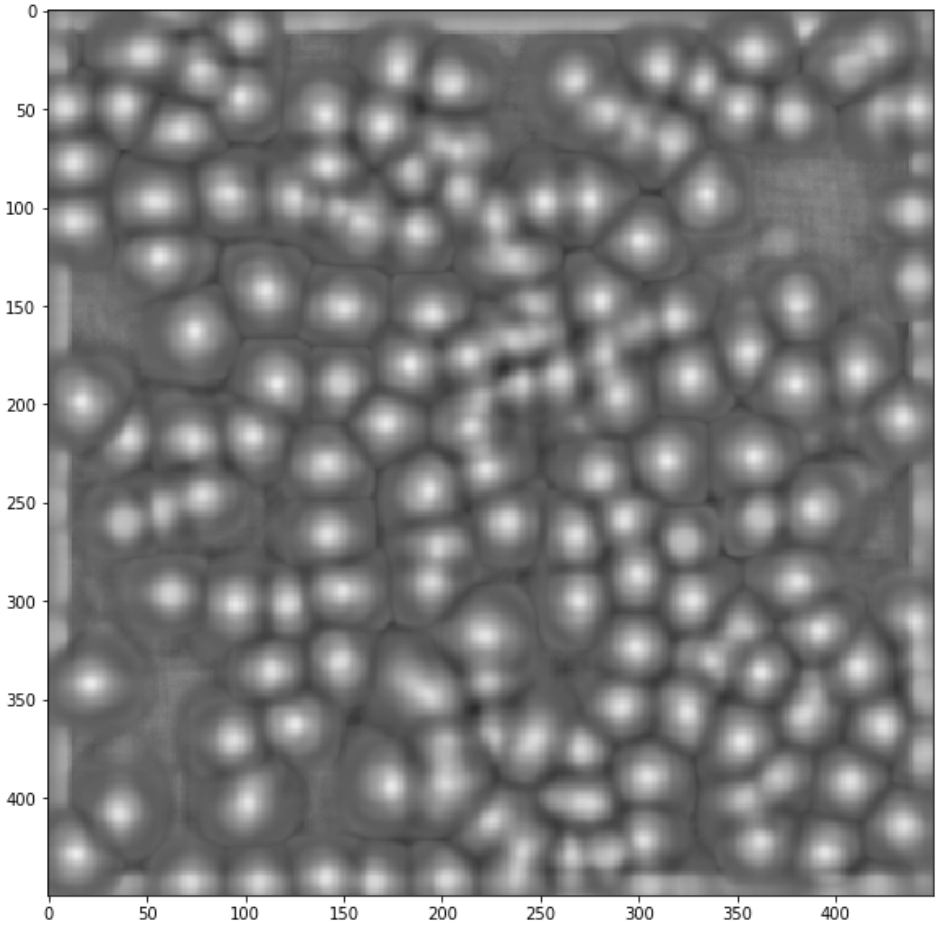
In [15]:
```
plt.imshow(template, cmap = 'gray')
plt.show()
```



In [16]:
```
matched = match_template(image=image, template=template, pad_input=True)
```

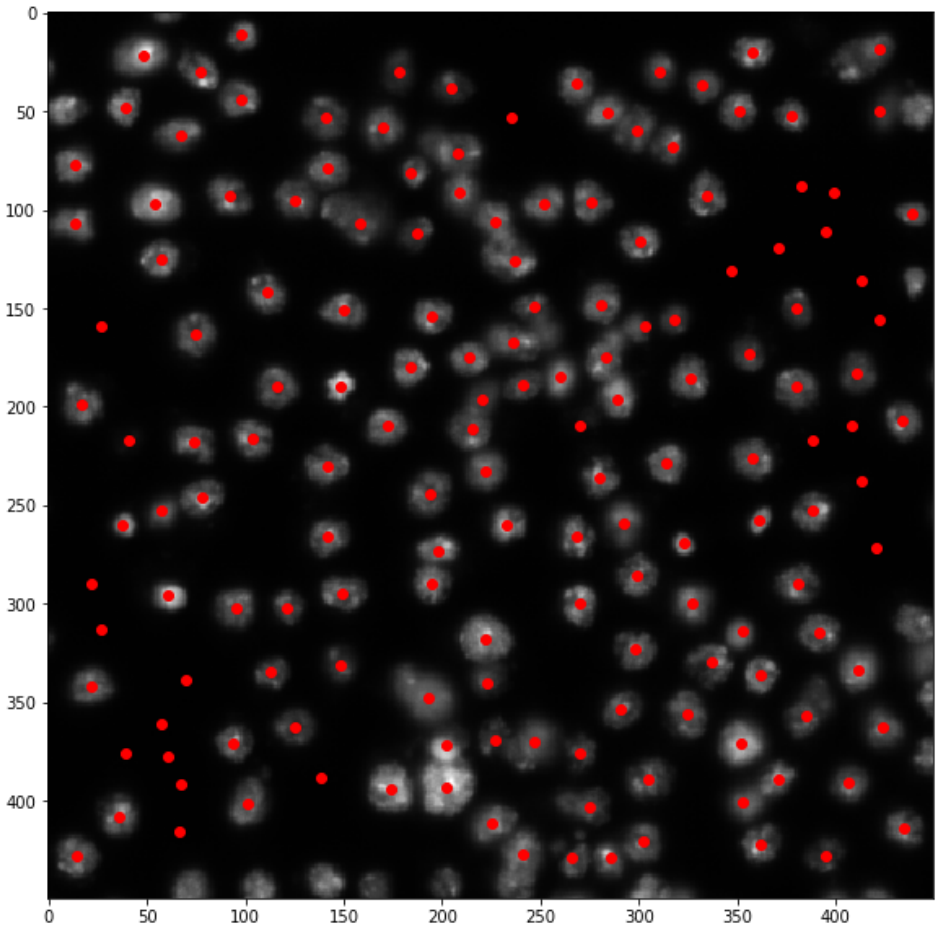
In [17]:
```
plt.figure(figsize=(10,10))
plt.imshow(matched, cmap = 'gray', vmin = -1, vmax = 1)
```

Out[17]:
```
<matplotlib.image.AxesImage at 0x7fc3bd16c9b0>
```



In [18]:
```
local_max = peak_local_max(matched, min_distance=10,indices=False)
local_max_indices = peak_local_max(matched, min_distance=10,indices=Tru
e)
```

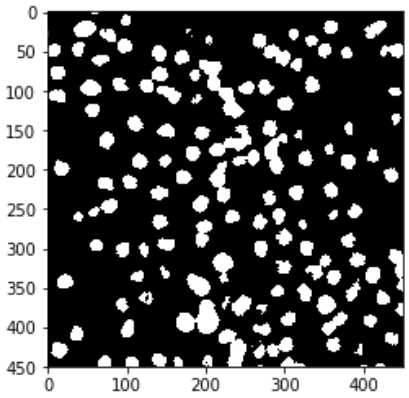
In [19]:
```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.plot(local_max_indices[:,1],local_max_indices[:,0],'ro');
```



We didn't set any threshold on what intensity local maxima should have, therefore we have a few detected cells that are clearly in the background. We could masks those using a rough threshold.

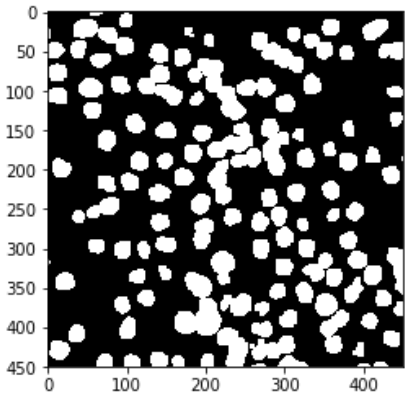
In [20]:
```
import skimage.filters
import skimage.morphology
```

In [21]:
```
otsu = skimage.filters.threshold_otsu(image)

otsu_mask = image>otsu

plt.imshow(otsu_mask);
```



We can dilate a bit all the regions to make sure we fill the holes and do not cut off dim cells

In [22]:
```
otsu_mask = skimage.morphology.binary_dilation(otsu_mask, np.ones((5,
5)))
plt.imshow(otsu_mask);
```



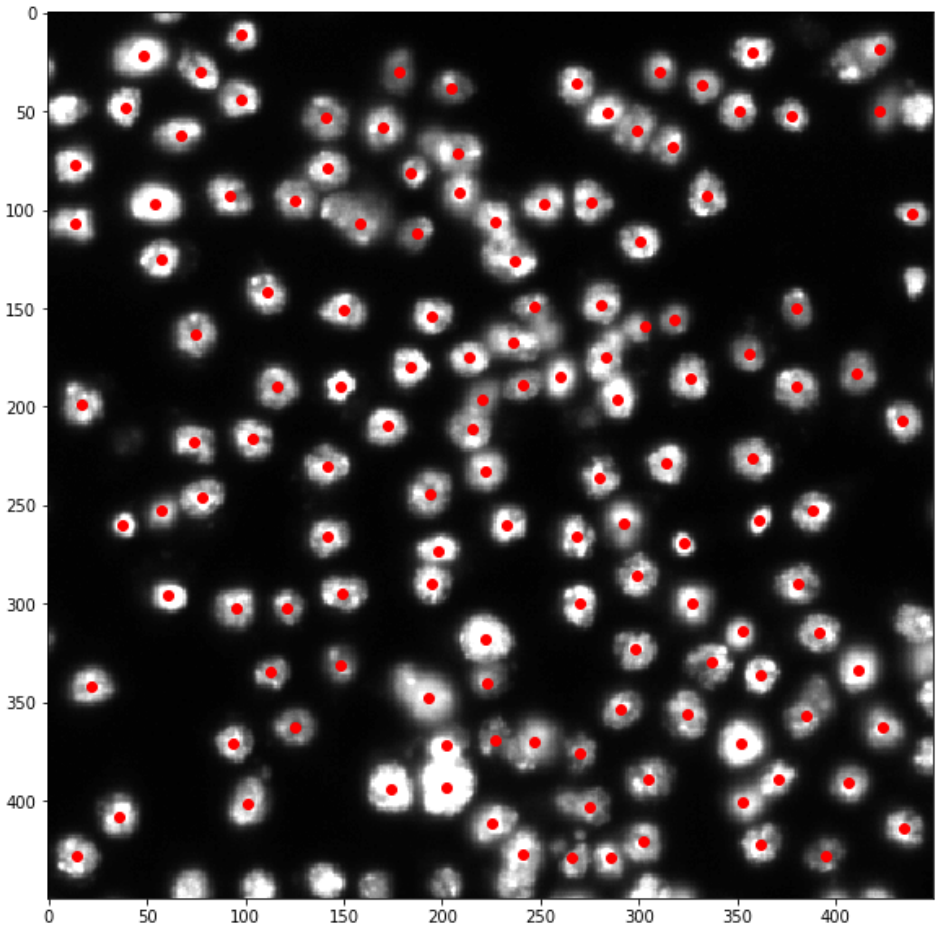
Now we can mask the image returned by the peak finder:

In [23]:
```
masked_peaks = local_max & otsu_mask
```

And recover the coordinates of the detected peaks:

In [24]:
```
peak_coords = np.argwhere(masked_peaks)
```

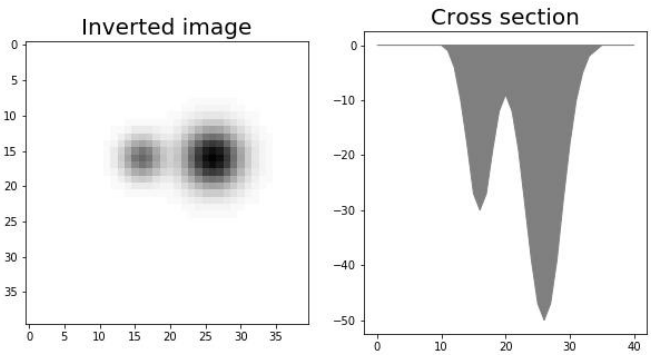
In [25]:
```
plt.figure(figsize=(10,10))
plt.imshow(image, cmap = 'gray',vmax = 100)
plt.plot(peak_coords[:,1],peak_coords[:,0],'ro');
```



In [26]:
```
# intensity is high, they touch each other -> would be complicated to do
without pattern matching
```

9. Watershed algorithm

In a number of cases, one is able to detect the positions of multiple objects on an image, but it might be difficult to segment them because they are close together or very irregular. This is where the wahtersehd algorithm is very practical. It takes as input an image, and a series of seeds and expands each region centered around a seed as if it was filling a topographic map.



In [1]:
```
from skimage.morphology import watershed
from skimage.measure import regionprops
```

In [2]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
plt.gray()
from skimage.external.tifffile import TiffFile
import skimage.io as io
from skimage.morphology import label

import course_functions
```

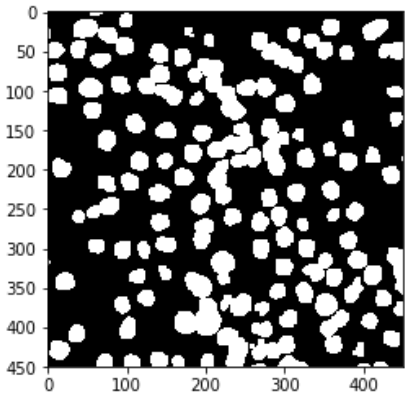
In [3]:
```
#load the image to process
image = io.imread('Data/BBBC007_v1_images/A9/A9 p9d.tif')
```

9.1 Create seeds

We can use the code of the last chapter to produce the seeds. We added the necessary code in our course module called course_functions

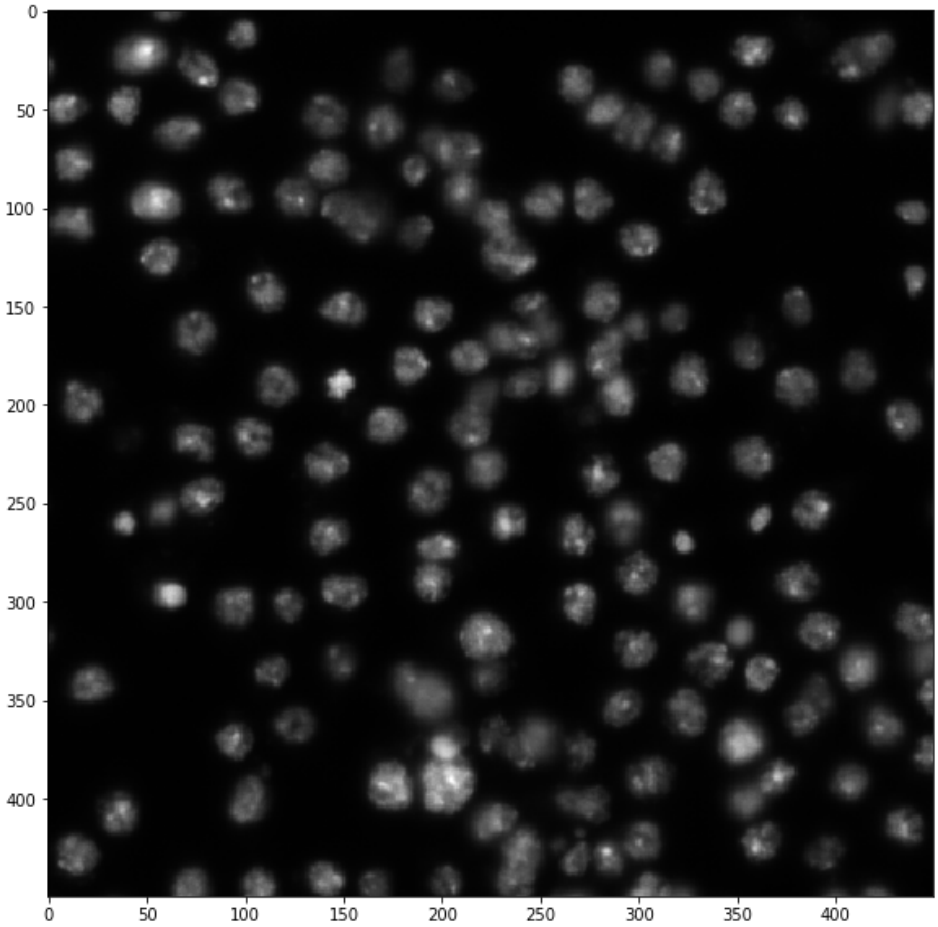
In [4]:
```
#generate template
template = course_functions.create_disk_template(10)
#generate seed map
seed_map, global_mask = course_functions.detect_nuclei_template(image, t
emplate)
```

In [5]:
```
plt.imshow(global_mask)
plt.show()
```

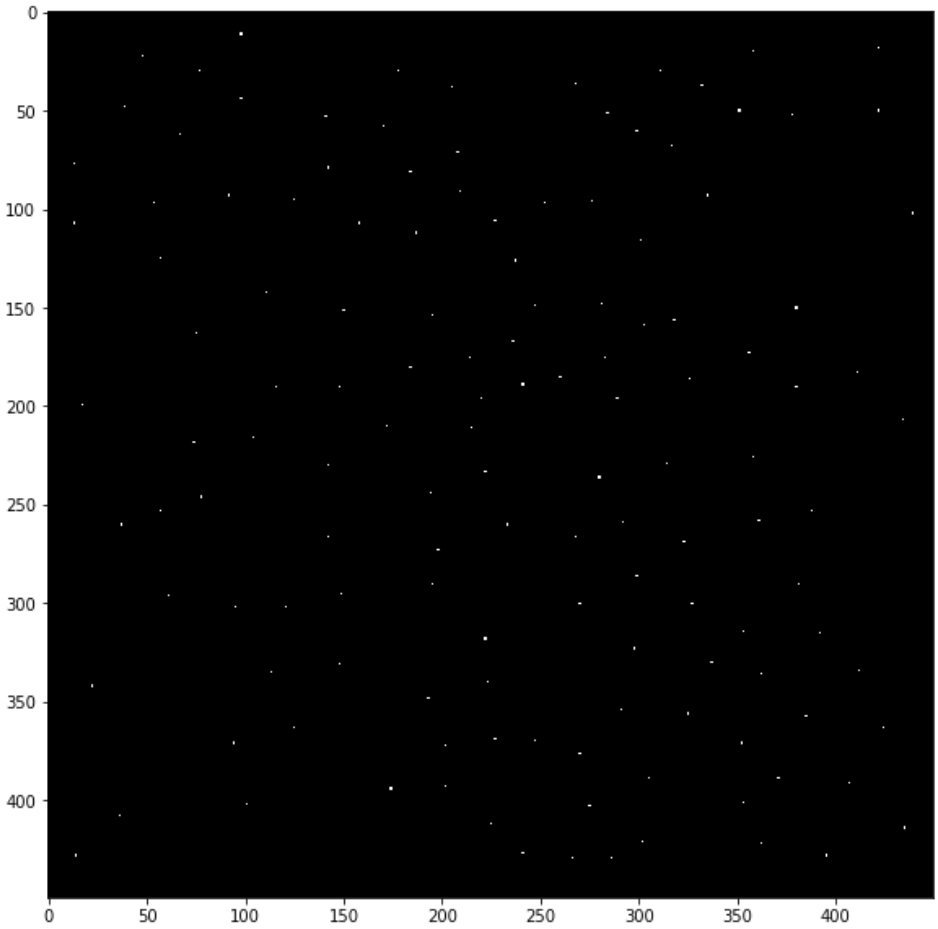


We need to create a labeled image, so that the watershed algorithm creates regions with different labels:

In [6]:
```
plt.figure(figsize=(10,10))
plt.imshow(image);
```



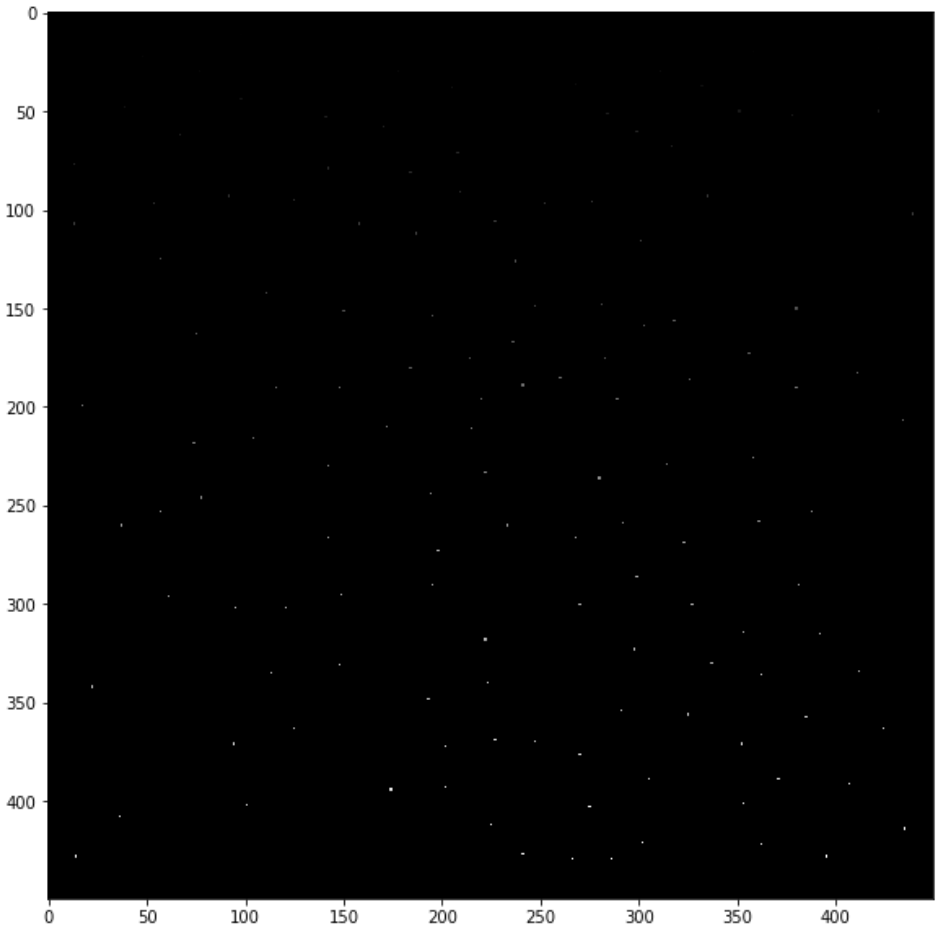
In [7]:
```
plt.figure(figsize=(10,10))
plt.imshow(seed_map);
```



In [8]:
```
seed_label = label(seed_map)
```

In [9]:
```
plt.figure(figsize=(10,10))
plt.imshow(seed_label)
```

Out[9]: <matplotlib.image.AxesImage at 0x7fb84e9f8ac8>



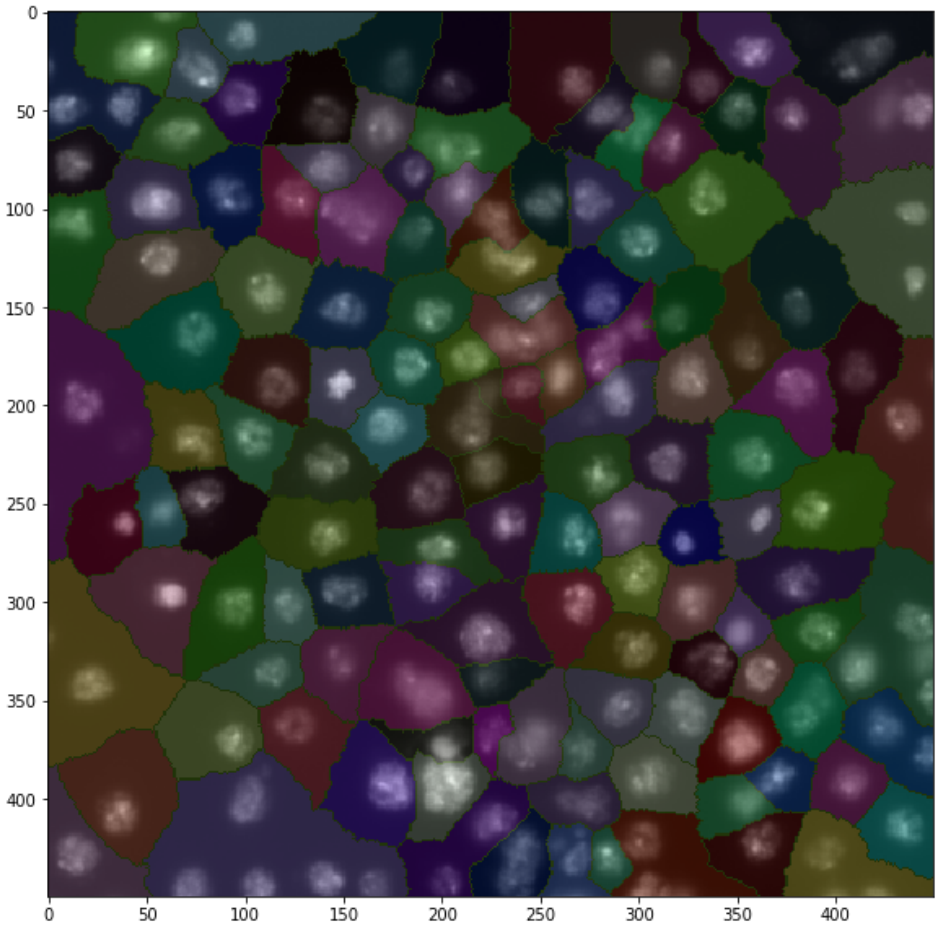
Now we can use the image and the labeled seed map to run the watershed algorithm. However, remember the analogy of filling a topographic map: our nuclei should be "deep" regions, so we need to invert the image. Finally we also require that a thin line separates regions (watershed_line option).

In [10]:
```
watershed_labels = watershed(image = -image, markers = seed_label, water
shed_line=True)
```

In [11]:
```
watershed_labels.max()
```

Out[11]: 136

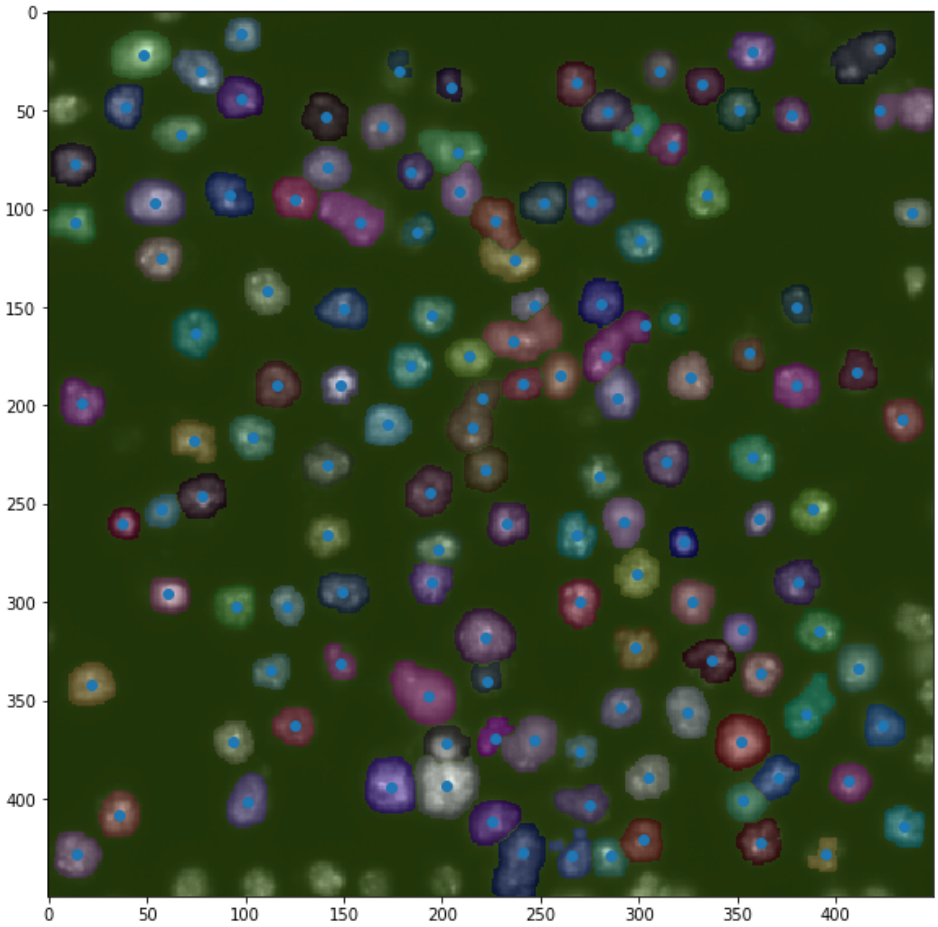
In [12]:
```
#create a random map
plt.figure(figsize = (10,10))

cmap = matplotlib.colors.ListedColormap ( np.random.rand ( 256,3))

plt.imshow(image)
plt.imshow(watershed_labels, cmap = cmap, alpha = 0.3);
```



The algorithm worked well and created regions around each nucleus. However we are only interested in the actual nuclei properties. So let's use our global masks to limit ourselves to those regions:

In [13]:
```
watershed_labels = watershed(image = -image, markers = seed_label, mask
= global_mask, watershed_line=True)
```

In [14]:
```
plt.figure(figsize = (10,10))
plt.imshow(image)
plt.imshow(watershed_labels, cmap = cmap, alpha = 0.3)
plt.plot(np.argwhere(seed_map)[:,1],np.argwhere(seed_map)[:,0],'o');
```

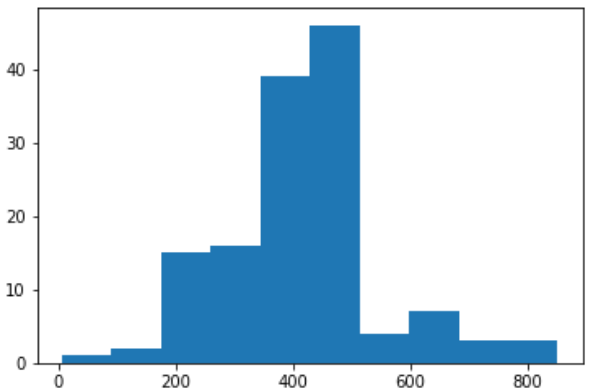


Finally, now that you have all the nuclei segmented you can proceed to do actual measurements e.g. by using the previously seen `regionprops` function.

In [15]:
```
myregions = regionprops(watershed_labels)
```

In [18]:
```
shape  = [x.area for x in myregions]
```

In [17]:
```
plt.hist(shape);
```



10. 3D case

Until now we have exclusively processed 2D images, eventhough the sometimes came from 3D acquisition. We are now going to look at an example of 3D processing where we are going to use the same tools as in 2D but in a 3D version.

Extending an image processing pipeline from 2D to 3D can be challenging for two reasons: first, computations can become very slow because of the amount of data changes usually roughly by an order of magnitude, and second, visualization of both original and processed data is more complicated.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray()
from ipywidgets import interact, IntSlider, fixed

import skimage.io as io
from skimage.transform import rescale, resize
from skimage.morphology import white_tophat
from skimage.feature import peak_local_max
from skimage.measure import regionprops, label
from skimage.filters import threshold_otsu, gaussian
import scipy.ndimage as ndi


#convenience functions
#create a segmentation image where background is NaN to use as overlay
def nan_image(image):
    image_nan = np.zeros(image.shape)
    image_nan[:] = np.nan
    for i in range(1,image.max()):
        image_nan[image==i]=i

#image plotting function used in concert with ipywidget interact. Plots
a single image.
def plot_plane(t,im, cmap):

    plt.figure(figsize=(10,10))
    plt.imshow(im[t,:,:],cmap = cmap)
    plt.show()

#image plotting function used in concert with ipywidget interact. Plots
two superposed images.
def plot_superpose(t, im1, im2, cmap):
    plt.figure(figsize=(10,10))
    plt.imshow(im1[t,:,:],cmap = 'gray')
    plt.imshow(im2[t,:,:],cmap = cmap, alpha = 0.3, vmin = 0, vmax = im
2.max())
    plt.show()

#Wrapping function to create an interactive view of an image stack for o
ne or a pair of stacks
def image_browser(image, image2 = None , color = True):
    if color == True:
        vals = np.linspace(0,1,int(image.max()))
        np.random.shuffle(vals)
        cmap = plt.cm.colors.ListedColormap(plt.cm.jet(vals))
    else:
        cmap = 'gray'

    if image2 is None:
        interact(plot_plane, t = IntSlider(min=0,max=image.shape[0],step
=1,value=0,
                                           continuous_update = False),im
= fixed(image), cmap = fixed(cmap));
    else:
        interact(plot_superpose, t = IntSlider(min=0,max=image.shape[0]-
1,step=1,value=0,
                                               continuous_update = Fals
e),im1 = fixed(image), im2 = fixed(image2),cmap = fixed(cmap));
```

In [2]:
```
from skimage.morphology import binary_closing, white_tophat, label, wate
rshed
from skimage.measure import regionprops, label
from skimage.feature import match_template, peak_local_max
```

We are going to look at a dataset of an embryo imaged in 3D in multiple wavelengths. We are first going to focus on one channel where the *nuclei* are marked. Then we will use that information to extract information from another channel where we will try to extract spot-like structures.

The goal here is to illustrate that most functions used before in 2D can be used in the same way in 3D, but with some new issues, especially around visualizations and computing time.

Let's load the first image and look at it along two projections:

In [3]:
```
image = io.imread('Data/BBBC032_v1_dataset/BMP4blastocystC3.tif')
```

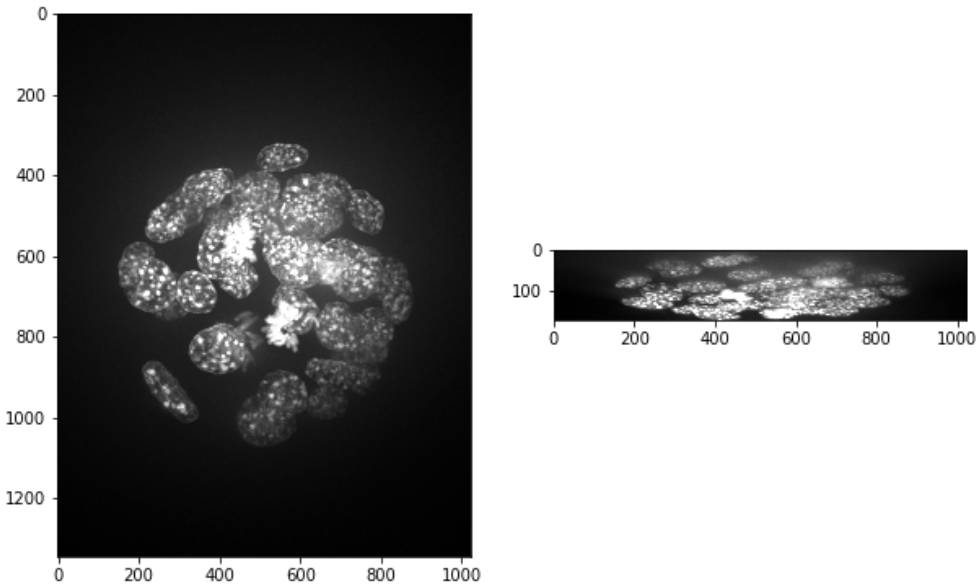
In [4]:
```
image.shape
```

Out[4]:
```
(172, 1344, 1024)
```

In [5]:
```
np.size(image)/10**6
```

Out[5]:
```
236.716032
```

In [6]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(image,axis = 0))
ax[1].imshow(np.max(image,axis = 1));
```



The image is really large, so any operation we are going to do on it will be very slow (e.g. a filter will have to visit every single one of the 230 millions pixels). As we just want to identify the *nuclei* we don't care about the details in the image, so a practical thing to do is to resample the imge. As the z dimension is larger than the xy (image on the right looks squished) we are going to use the opportunity to "stretch" the image during resampling:

In [7]:
```
image_resampled = rescale(image,(0.5,0.15,0.15), multichannel=False,pres
erve_range=True, anti_aliasing=True)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
```

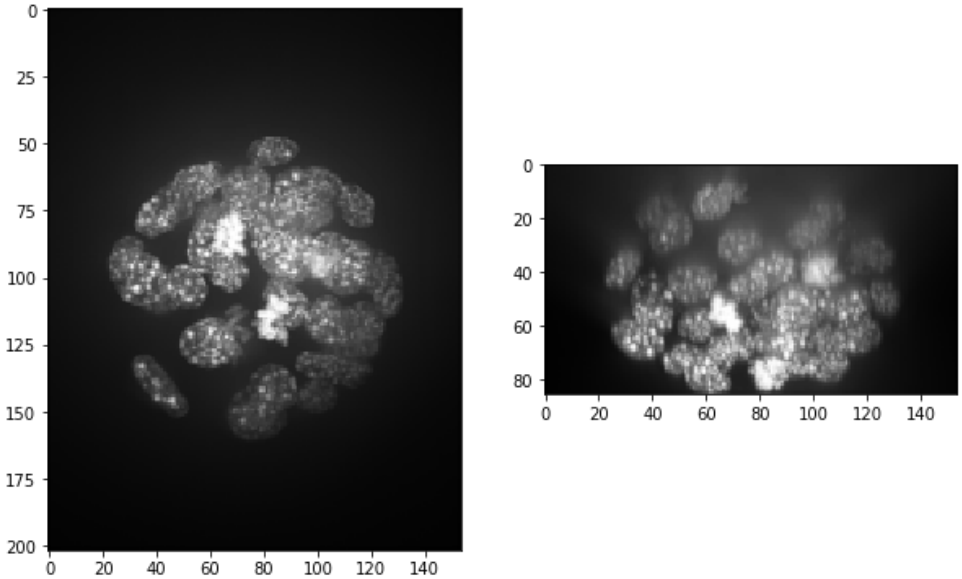
In [8]:
```
image_resampled.shape
```

Out[8]:
```
(86, 202, 154)
```

Let's look at the result;

In [9]:
```
#image_resampled = gaussian(image_resampled,sigma=(2,2,2))
```

In [10]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(image_resampled,axis = 0))
ax[1].imshow(np.max(image_resampled,axis = 1));
```

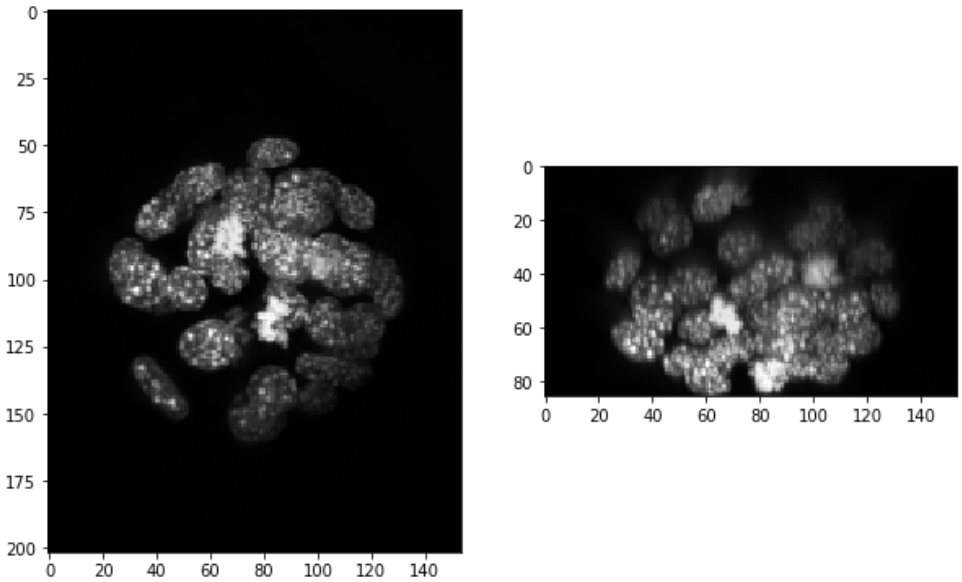


To remove some of the glare in the image we can use a top-hat filter, which keeps objects which are smaller than a structuring element and brighter than their surroundings. "Flat" low-illumination regions get therefore removed:

In [11]:
```
from skimage.morphology import binary_closing, white_tophat, label, wate
rshed, black_tophat
```

In [12]:
```
im_tophat = white_tophat(image_resampled,selem=np.ones((20,20,20)))
```

In [13]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(im_tophat,axis = 0))
ax[1].imshow(np.max(im_tophat,axis = 1));
```



We can have a look at what happens if we do a classical thresholding of the image, which works just like in 2D.

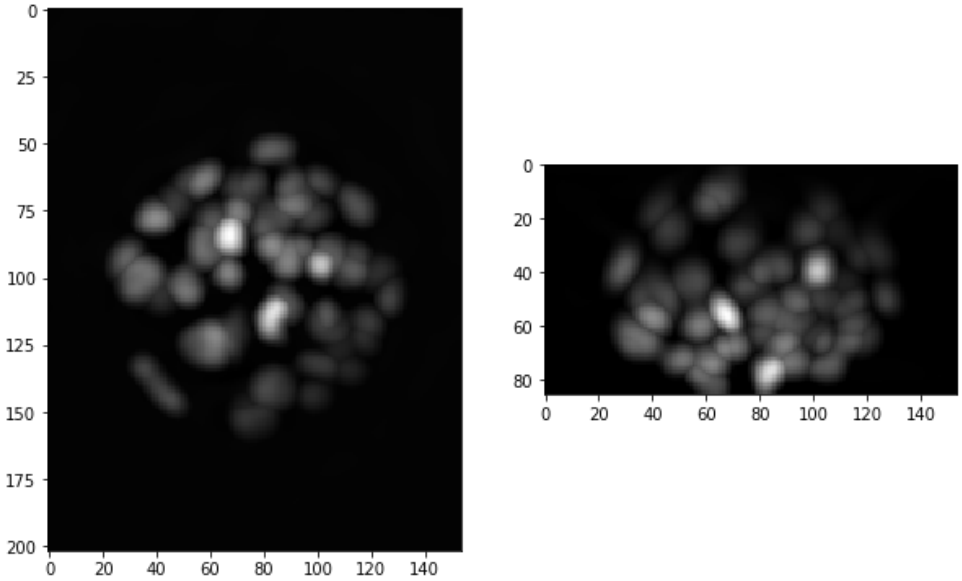
In [14]:
```
image_browser(im_tophat>threshold_otsu(im_tophat), color=False)
```

The result is poor because the *nucleus* signal is not homogeneous, *i.e.* each *nucleus* is made of sparse bright signals. To identify larger scale structures, we thus have to filter the image with a structuring element that has approximately the shape of the *nuclei*. A typical filter used to detect "blobs" is the LoG filter (Laplacian of a Gaussian).

The filter doesn't exist *per se* in scikit-image so we are going to use the one of scipy.

In [15]:
```
im_log = -ndi.filters.gaussian_laplace(im_tophat,(4,4,4))
```

In [16]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(im_log,axis = 0), cmap = 'gray')
ax[1].imshow(np.max(im_log,axis = 1), cmap = 'gray')
plt.show()
```



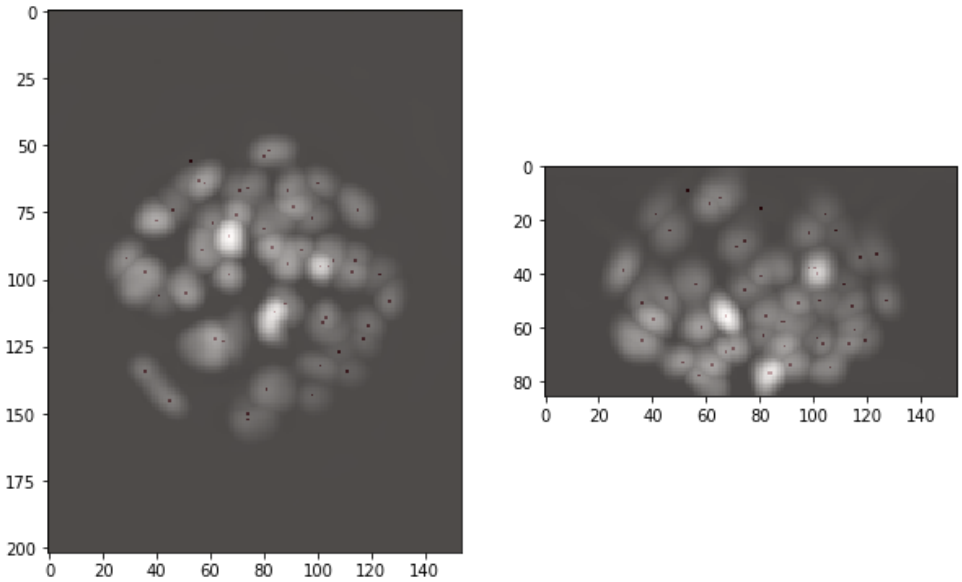
Now that we have more homogeneous regions, we can try again to use a classical thresholding, which should give a much better result.

In [17]:
```
image_browser(im_log>threshold_otsu(im_log), color = False)
```

We can now go back to some of the methods we have seen previously: we can find local maxima corresponding to single *nuclei*, define a global mask, and use the watershed algorithm for segmentation.

In [18]:
```
peak_image = peak_local_max(im_log, footprint=np.ones((10,10,10)), indic
es=False, threshold_abs= 1)
```

In [ ]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(im_log,axis = 0), cmap = 'gray')
ax[0].imshow(np.max(peak_image,axis = 0), cmap = 'Reds',alpha = 0.3)

ax[1].imshow(np.max(im_log,axis = 1), cmap = 'gray')
ax[1].imshow(np.max(peak_image,axis = 1), cmap = 'Reds',alpha = 0.3)

plt.show()
```



In [ ]:
```
mask = im_log>threshold_otsu(im_log)
im_label = label(peak_image)
im_water = watershed(image=-im_log,markers=im_label,mask = mask, compact
ness=0.01)
```

In [ ]:
```
image_browser(im_log, im_water, color = False)
```

In [ ]:
```
image_browser(image_resampled, im_water, color = False)
```

The result is rather crude but a good start for potential further processing. Note that we didn't segment the *nuclei per se* but their convolution with a LoG filter. We can also visualize the result in 3D. For that we use the ipyvolume package which allows one to represent 3D data in various ways. For example as isosurface (on a binary image, it just gives the surface of the objects):

In [ ]:
```
import ipyvolume.pylab as ipv
```

In [ ]:
```
ipv.figure()
ipv.plot_isosurface(im_water>0)
ipv.show()
```

But we can of course also show the volume data of our resampled image:

In [ ]:
```
ipv.figure()
ipv.volshow(im_tophat.astype(int).T)
ipv.style.background_color('black')
ipv.show()
```

```
/usr/local/lib/python3.5/dist-packages/ipyvolume/serialize.py:81: Runtime
Warning: invalid value encountered in true_divide
  gradient = gradient / np.sqrt(gradient[0]**2 + gradient[1]**2 + gradien
t[2]**2)
```

Detecting features within features

In [ ]:
```
image2 = io.imread('Data/BBBC032_v1_dataset/BMP4blastocystC1.tif')
```

In another wavelength, the collected signal appears as *puncti* in the image. We could for example now wish to know how many of those *puncti* appear in the *nuclei*. Here we cannot downscale the image as those small structures would otherwise disappear, so we use the fact that we know where nuclei are to just analyse those regions.

Let us first resize our segmentation map. Note that we use order = 0 (nearest neighbors) to preserve our labeling.

In [ ]:
```
im_nuclei_segm = resize(im_water, image.shape, order = 0, preserve_range
=True)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

In [ ]:
```
np.unique(im_nuclei_segm)
```

Let's recover all the single *nuclei* regions using regionproperties

In [ ]:
```
regions = regionprops(im_nuclei_segm.astype(int), image2)
```

In [ ]:
```
im_crop = image2#regions[10].intensity_image
```

In [ ]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(im_crop,axis = 0))
ax[1].imshow(np.max(im_crop,axis = 1))
```

The spots from those images have approximately a gaussian shape. So we can try to filter our image with an appropriately size 3D Gaussian to detect the spots:

In [ ]:
```
im_gauss = gaussian(im_crop, sigma = [1,1.5,1.5], preserve_range=True)
```

In [ ]:
```
fig, ax = plt.subplots(1,2,figsize = (10,10))
ax[0].imshow(np.max(im_gauss,axis = 0))
ax[1].imshow(np.max(im_gauss,axis = 1));
```

In [ ]:
```
peaks = peak_local_max(im_gauss,min_distance=4)
```

In [ ]:
```
fig, ax = plt.subplots(1,2,figsize = (20,10))
ax[0].imshow(np.max(im_gauss,axis = 0))
ax[0].plot(peaks[:,2], peaks[:,1],'ro',markersize = 0.1)
ax[1].imshow(np.max(im_gauss,axis = 1))
ax[1].plot(peaks[:,2], peaks[:,0],'ro',markersize = 0.1);
```

In [ ]:
```
plt.hist(im_gauss[peaks[:,0],peaks[:,1],peaks[:,2]],bins = np.arange(20
0,1600,1));
```

In [ ]:
```
plt.hist(im_gauss[peaks[:,0],peaks[:,1],peaks[:,2]],bins = np.arange(20
0,800,10));
```

In [ ]:
```
peak_val = im_gauss[peaks[:,0],peaks[:,1],peaks[:,2]]
```

In [ ]:
```
peaks_selected = peaks[peak_val>600,:]
```

In [ ]:
```
fig, ax = plt.subplots(1,2,figsize = (20,10))
ax[0].imshow(np.max(im_gauss,axis = 0))
ax[0].plot(peaks_selected[:,2], peaks_selected[:,1],'ro',markersize = 0.
1)
ax[1].imshow(np.max(im_gauss,axis = 1))
ax[1].plot(peaks_selected[:,2], peaks_selected[:,0],'ro',markersize = 0.
1);
```

In [ ]:
```
peak_crop = peaks_selected[peaks_selected[:,2]>400,:]
peak_crop = peak_crop[peak_crop[:,2]<600,:]
peak_crop = peak_crop[peak_crop[:,1]<800,:]
peak_crop = peak_crop[peak_crop[:,1]>600,:]


plt.figure(figsize = (20,10))
plt.imshow(np.max(im_gauss,axis = 0)[600:800,400:600])
plt.plot(peak_crop[:,2]-400, peak_crop[:,1]-600,'ro',markersize = 1);
```

11. Create a short complete analysis

Until now we have only seen pieces of code to do some specific segmentation of images. Typically however, one is going to have a complete analysis, including image processing and some further data analysis.

Here we are going to come back to an earlier dataset where *nuclei* appeared as circles. That dataset was a time-lapse, and we might be interested in knowing how those *nuclei* move over time. So we will have to analyze images at every time-point, find the position of the *nuclei*, track them and measure the distance traveled.

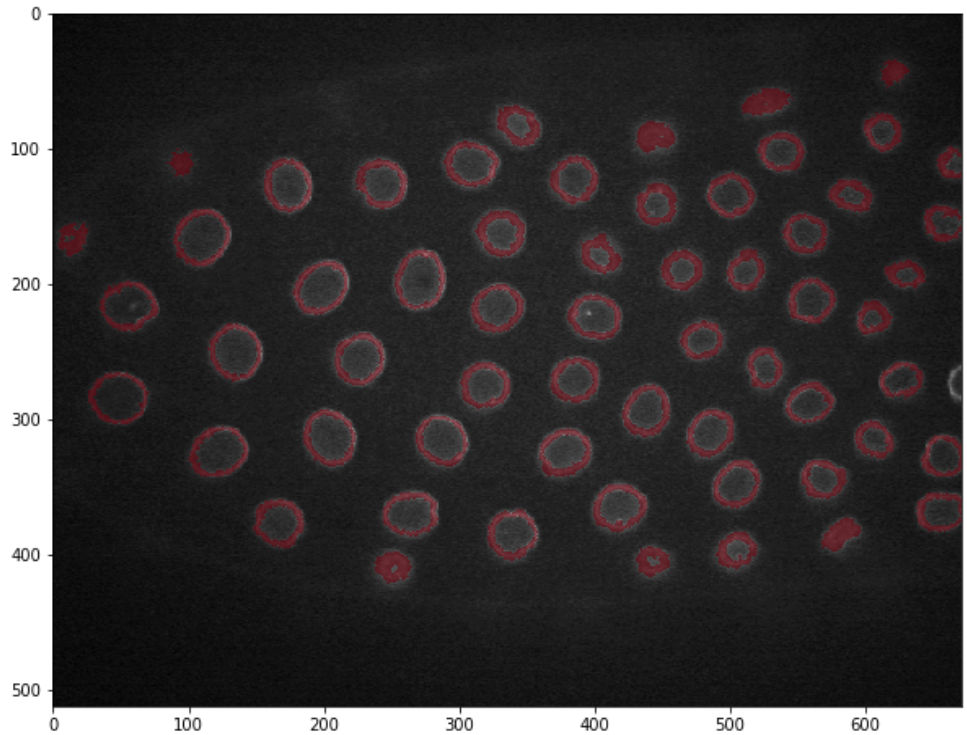
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray()
from skimage.external.tifffile import TiffFile
from skimage.measure import label, regionprops

#import your function
from course_functions import detect_nuclei
```

11.1 Remembering previous work

Let's remember what we did in previous chapters. We opened the tif dataset, selected a specific plane to look at and segmented the *nuclei*:

In [2]:
```
#load the image to process
data = TiffFile('Data/30567/30567.tif')
image = data.pages[3].asarray()
#create your mask
nuclei = detect_nuclei(image)
#create a nan-mask for overlay
nuclei_nan = nuclei.copy().astype(float)
nuclei_nan[nuclei == 0] = np.nan

#plot
plt.figure(figsize=(10,10))
plt.imshow(image, cmap = 'gray')
plt.imshow(nuclei_nan, cmap = 'Reds',vmin = 0,vmax = 1,alpha = 0.6)
plt.show()
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```



Let's also remember what was the format of that file (usually one would already know that or verify *e.g.* in Fiji)

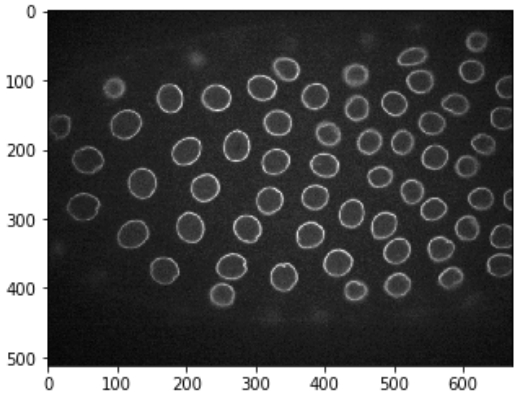
In [3]:
```
data.info()
```

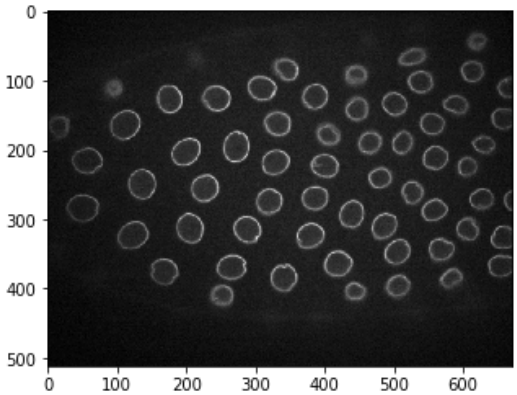
Out[3]:
```
'TIFF file: 30567.tif, 473 MiB, big endian, ome, 720 pages\n\nSeries 0: 7
2x2x5x512x672, uint16, TCZYX, 720 pages, not mem-mappable\n\nPage 0: 512x
672, uint16, 16 bit, minisblack, raw, ome|contiguous\n* 256 image_width
(1H) 672\n* 257 image_length (1H) 512\n* 258 bits_per_sample (1H) 16\n* 2
59 compression (1H) 1\n* 262 photometric (1H) 1\n* 270 image_description
(3320s) b\'<?xml version="1.0" encoding="UTF-8"?><!-- Wa\n* 273 strip_off
sets (86I) (182, 8246, 16310, 24374, 32438, 40502, 48566, 56630,\n* 277 s
amples_per_pixel (1H) 1\n* 278 rows_per_strip (1H) 6\n* 279 strip_byte_co
unts (86I) (8064, 8064, 8064, 8064, 8064, 8064, 8064, 8064, \n* 282 x_res
olution (2I) (1, 1)\n* 283 y_resolution (2I) (1, 1)\n* 296 resolution_uni
t (1H) 1\n* 305 software (17s) b\'LOCI Bio-Formats\''
```

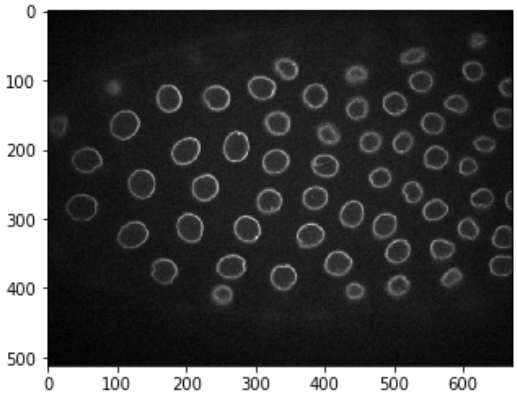
On the first line we see that we have 72 time points, 2 colors, 5 planes per color.

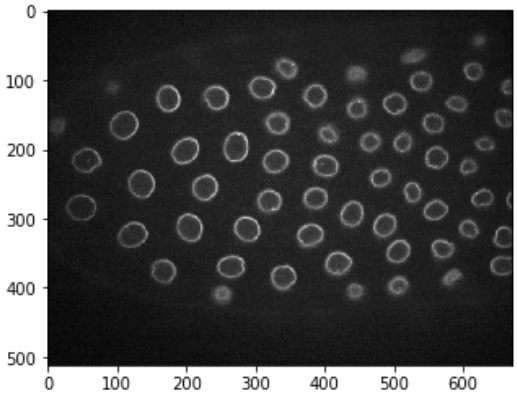
The *nuclei* are going to move a bit in Z (perpendicular to the image) over time, so it will be more accurate to segment a projection of the entire stack. So how do we get a complete stack at a given time point. Let's plot the first few images, to understand how they are stored.

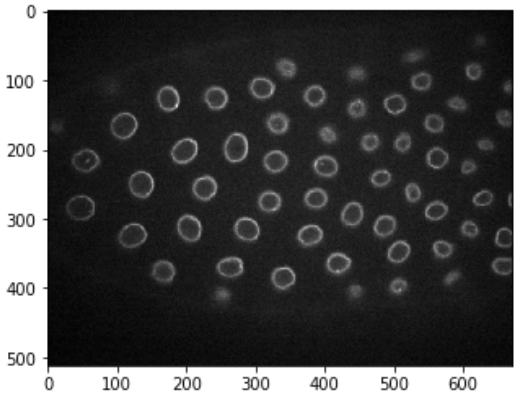
In [4]:
```
for i in range(15):
    plt.imshow(data.pages[i].asarray())
    plt.show()
```

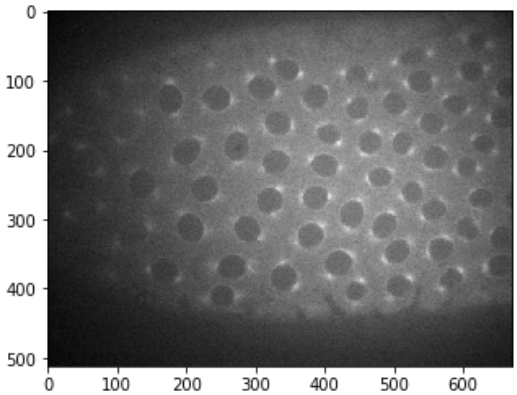

0
100
200
300
400
500
0 100 200 300 400 500 600

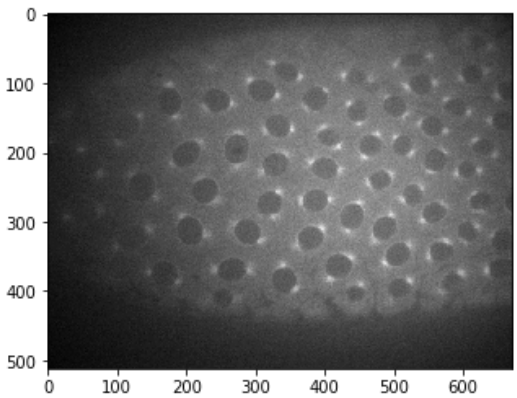

0
100
200
300
400
500
0 100 200 300 400 500 600

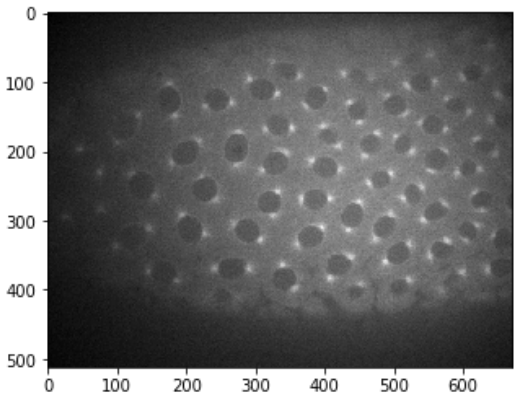

0
100
200
300
400
500
0 100 200 300 400 500 600

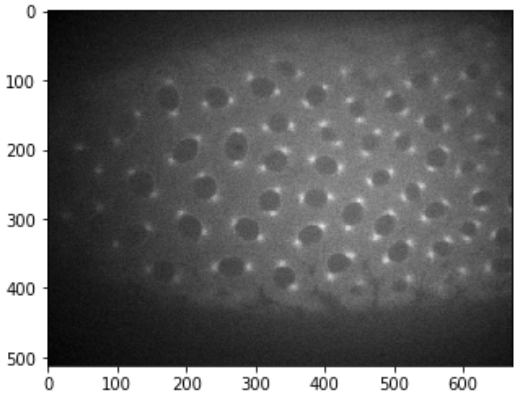

0
100
200
300
400
500
0 100 200 300 400 500 600

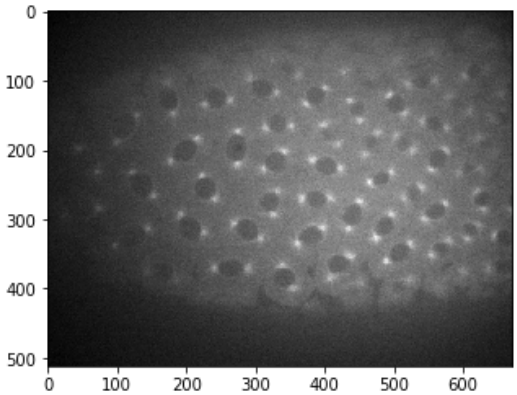

0
100
200
300
400
500
0 100 200 300 400 500 600

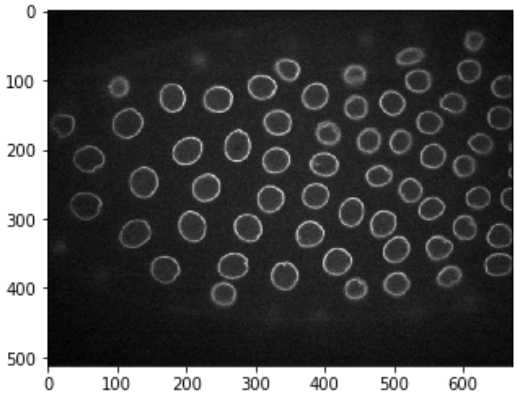

0
100
200
300
400
500
0 100 200 300 400 500 600

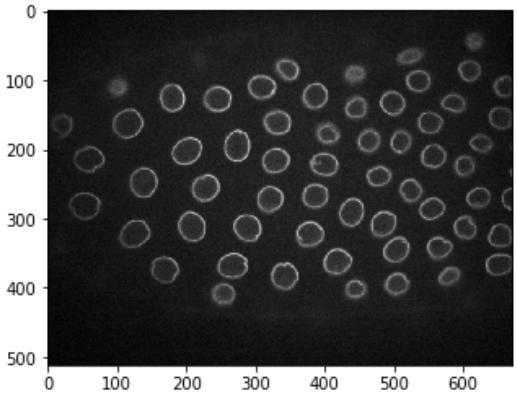

0
100
200
300
400
500
0 100 200 300 400 500 600

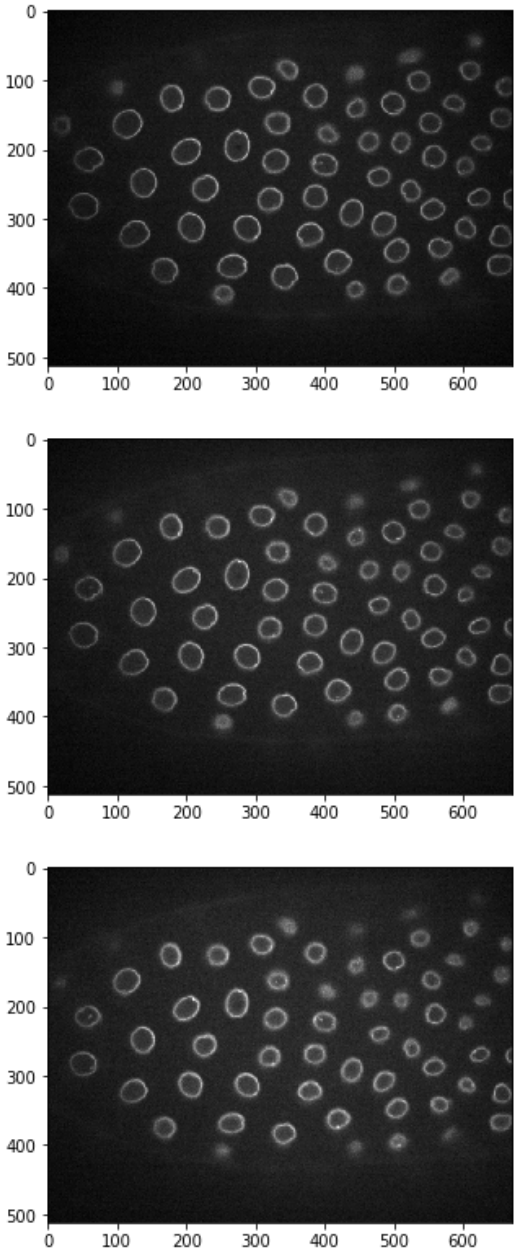

0
100
200
300
400
500
0 100 200 300 400 500 600


0
100
200
300
400
500
0 100 200 300 400 500 600


0
100
200
300
400
500
0 100 200 300 400 500 600


0
100
200
300
400
500
0 100 200 300 400 500 600

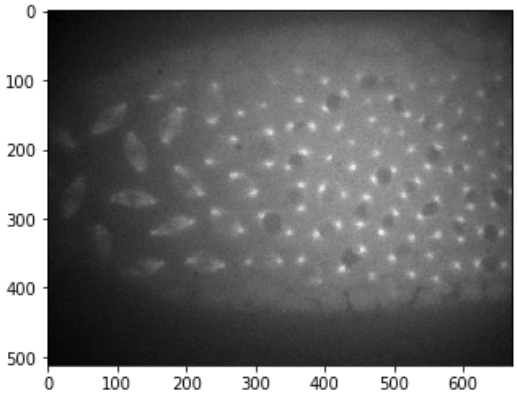

0
100
200
300
400
500
0 100 200 300 400 500 600



11.2 Processing a time-lapse

So it looks like we have all planes of colour 1 at time =0, then all planes of color 2 at time =0, then all planes of colour 1 at time = 1 etc... Therefore to get a full stack at a given time we have to use:

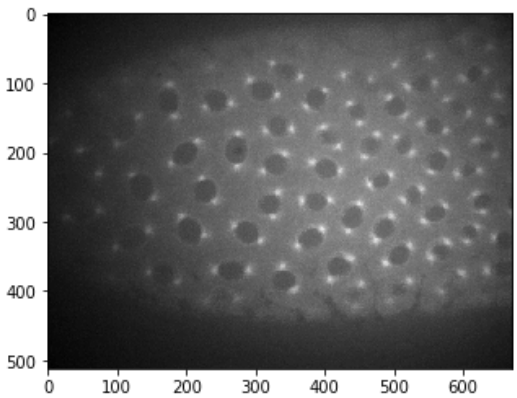
In [5]:
```
images_per_time = 10
time = 10
color = 1

image_stack = np.stack([x.asarray()
                        for x in data.pages[time*images_per_time+0+color
*5:time*images_per_time+5+color*5]])

plt.imshow(np.max(image_stack, axis = 0));
```



Let's make a little function out of that:

In [6]:
```
def get_stack(data, time, color, images_per_time):
    image_stack = np.stack([x.asarray()
                        for x in data.pages[time*images_per_time+0+color
*5:time*images_per_time+5+color*5]])
    return image_stack
```

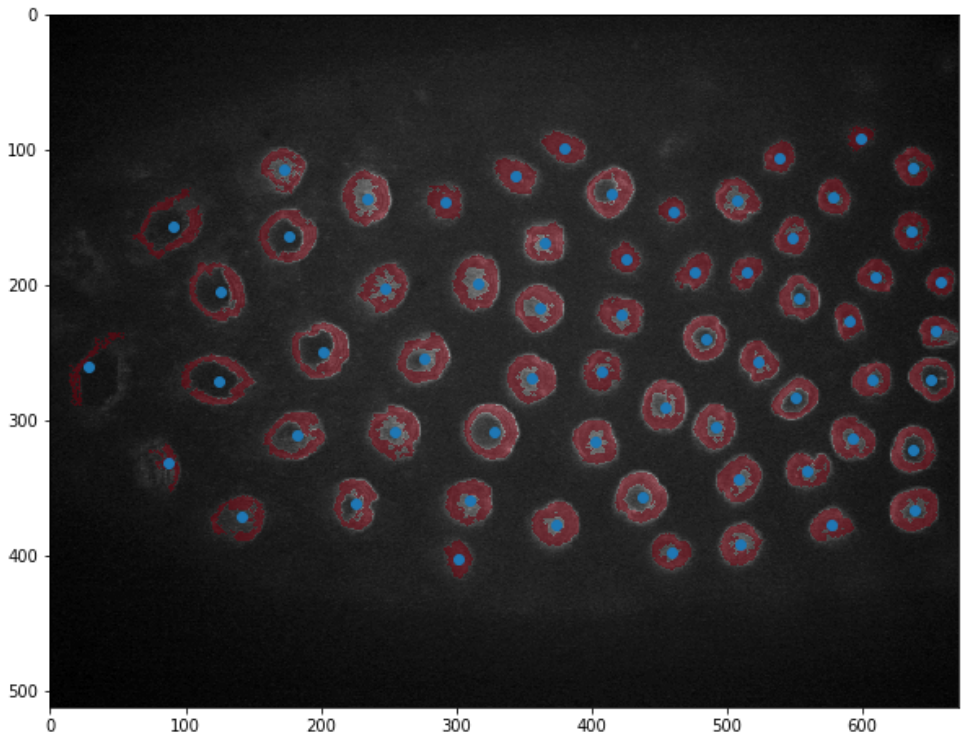
In [7]:
```
plt.imshow(np.max(get_stack(data, 0, 1, 10), axis = 0));
```



Now we can chose any time point and segment if using our two functions. In addition we can use the region properties to define the average position of each detected nucleus:

In [8]:
```
#choose a time
time = 10

#load the stack and segment it
image_stack = get_stack(data, time,0,10)
image = np.max(image_stack, axis = 0)
nuclei = nuclei = detect_nuclei(image)

#find position of nuclei
nuclei_label = label(nuclei)
regions = regionprops(nuclei_label)
centroids = np.array([x.centroid for x in regions])

#create a nan-mask for overlay
nuclei_nan = nuclei.copy().astype(float)
nuclei_nan[nuclei == 0] = np.nan

#plto the result
plt.figure(figsize=(10,10))
plt.imshow(image, cmap = 'gray')
plt.imshow(nuclei_nan, cmap = 'Reds',vmin = 0,vmax = 1,alpha = 0.6)
plt.plot(centroids[:,1], centroids[:,0],'o');
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```



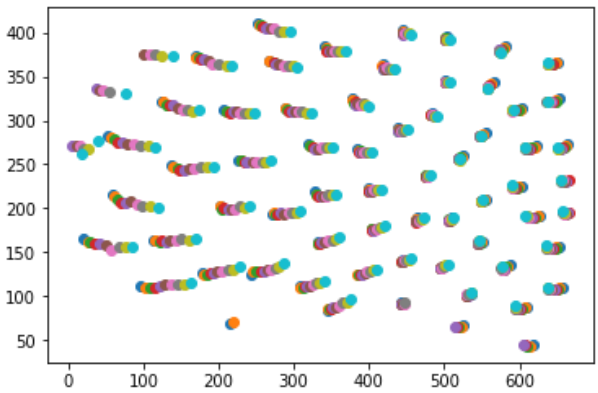
So now we can repeat the same operation for multiple time points and add the array with the coordinates to a list to keep them safe

In [9]:
```
centroids_time = []
for time in range(10):

    #load the stack and segment it
    image_stack = get_stack(data, time,0,10)
    image = np.max(image_stack, axis = 0)
    nuclei = nuclei = detect_nuclei(image)

    #find position of nuclei
    nuclei_label = label(nuclei)
    regions = regionprops(nuclei_label)
    centroids = np.array([x.centroid for x in regions])

    centroids_time.append(centroids)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```

Let's plot all those centroids for all time points

In [10]:
```
for x in centroids_time:
    plt.plot(x[:,1],x[:,0],'o')
```



We definitely see tracks corresponding to single nuclei here. How are we going to track them?

11.3 Tracking trajectories

The wonderful thing with Python, is that there are a lot of resources that one can just use. For example, if we Google "python tracking", one of the first hits if for the package trackpy which is originally designed to track diffusion particles but can be repurposed for anything.

Browsing through the documentation, we see that we need the function link_df. df stands for dataframe, which is a special data format offered by the package Pandas, and is very close to the R dataframe. Let's load those two modules:

In [11]:
```
import trackpy
import pandas as pd
```

And look for some help:

In [12]:
```
help(trackpy.link_df)
```

```
Help on function link in module trackpy.linking.linking:

link(f, search_range, pos_columns=None, t_column='frame', **kwargs)
    link(f, search_range, pos_columns=None, t_column='frame', memory=0,
        predictor=None, adaptive_stop=None, adaptive_step=0.95,
        neighbor_strategy=None, link_strategy=None, dist_func=None,
        to_eucl=None)

    Link a DataFrame of coordinates into trajectories.

    Parameters
    ----------
    f : DataFrame
        The DataFrame must include any number of column(s) for position a
nd a
        column of frame numbers. By default, 'x' and 'y' are expected for
        position, and 'frame' is expected for frame number. See below for
        options to use custom column names.
    search_range : float or tuple
        the maximum distance features can move between frames,
        optionally per dimension
    pos_columns : list of str, optional
        Default is ['y', 'x'], or ['z', 'y', 'x'] when 'z' is present in
f
    t_column : str, optional
        Default is 'frame'
    memory : integer, optional
        the maximum number of frames during which a feature can vanish,
        then reappear nearby, and be considered the same particle. 0 by d
efault.
    predictor : function, optional
        Improve performance by guessing where a particle will be in
        the next frame.
        For examples of how this works, see the "predict" module.
    adaptive_stop : float, optional
        If not None, when encountering an oversize subnet, retry by progr
essively
        reducing search_range until the subnet is solvable. If search_ran
ge
        becomes <= adaptive_stop, give up and raise a SubnetOversizeExcep
tion.
    adaptive_step : float, optional
        Reduce search_range by multiplying it by this factor.
    neighbor_strategy : {'KDTree', 'BTree'}
        algorithm used to identify nearby features. Default 'KDTree'.
    link_strategy : {'recursive', 'nonrecursive', 'numba', 'hybrid', 'dro
p', 'auto'}
        algorithm used to resolve subnetworks of nearby particles
        'auto' uses hybrid (numba+recursive) if available
        'drop' causes particles in subnetworks to go unlinked
    dist_func : function, optional
        a custom distance function that takes two 1D arrays of coordinate
s and
        returns a float. Must be used with the 'BTree' neighbor_strategy.
    to_eucl : function, optional
        function that transforms a N x ndim array of positions into coord
inates
        in Euclidean space. Useful for instance to link by Euclidean dist
ance
        starting from radial coordinates. If search_range is anisotropic,
this
        parameter cannot be used.

    Returns
    -------
    DataFrame with added column 'particle' containing trajectory labels.
    The t_column (by default: 'frame') will be coerced to integer.

    -    -
```

So we have a lot of options, but the most important thing is to get our data into a dataframe that has three columns, x,y and frame. How are we going to create such a dataframe ?

11.3.1 Pandas dataframe

In [13]:
```
help(pd.DataFrame)
```

```
Help on class DataFrame in module pandas.core.frame:

class DataFrame(pandas.core.generic.NDFrame)
 |  Two-dimensional size-mutable, potentially heterogeneous tabular data
 |  structure with labeled axes (rows and columns). Arithmetic operations
 |  align on both row and column labels. Can be thought of as a dict-like
 |  container for Series objects. The primary pandas data structure.
 |
 |  Parameters
 |  ----------
 |  data : ndarray (structured or homogeneous), Iterable, dict, or DataFr
ame
 |      Dict can contain Series, arrays, constants, or list-like objects
 |
 |      .. versionchanged :: 0.23.0
 |         If data is a dict, argument order is maintained for Python 3.6
 |         and later.
 |
 |  index : Index or array-like
 |      Index to use for resulting frame. Will default to RangeIndex if
 |      no indexing information part of input data and no index provided
 |  columns : Index or array-like
 |      Column labels to use for resulting frame. Will default to
 |      RangeIndex (0, 1, 2, ..., n) if no column labels are provided
 |  dtype : dtype, default None
 |      Data type to force. Only a single dtype is allowed. If None, infe
r
 |  copy : boolean, default False
 |      Copy data from inputs. Only affects DataFrame / 2d ndarray input
 |
 |  See Also
 |  --------
 |  DataFrame.from_records : Constructor from tuples, also record arrays.
 |  DataFrame.from_dict : From dicts of Series, arrays, or dicts.
 |  DataFrame.from_items : From sequence of (key, value) pairs
 |      pandas.read_csv, pandas.read_table, pandas.read_clipboard.
 |
 |  Examples
 |  --------
 |  Constructing DataFrame from a dictionary.
 |
 |  >>> d = {'col1': [1, 2], 'col2': [3, 4]}
 |  >>> df = pd.DataFrame(data=d)
 |  >>> df
 |     col1  col2
 |  0     1     3
 |  1     2     4
 |
 |  Notice that the inferred dtype is int64.
 |
 |  >>> df.dtypes
 |  col1    int64
 |  col2    int64
 |  dtype: object
 |
 |  To enforce a single dtype:
 |
 |  >>> df = pd.DataFrame(data=d, dtype=np.int8)
 |  >>> df.dtypes
 |  col1    int8
 |  col2    int8
 |  dtype: object
 |
 |  Constructing DataFrame from numpy ndarray:
 |
 |  >>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
 |  ...                    columns=['a', 'b', 'c'])
 |  >>> df2
```

Tons of information, but basically we can use as input a Numpy array. So let's just try to do that and see what comes out. Our list of coordinates arrays only contains x and y positions but no time. So first we will add a column to each array. Let's test on the first array:

In [34]:
```
first_array = centroids_time[0].copy()
#first_array
```

We now append a column to this array that contains the time of this frame:

In [35]:
```
time = 0
first_array = np.c_[first_array, time *np.ones(first_array.shape[0])]
#first_array
```

Let's do the same thing for all time points simply using a comprehension list:

In [33]:
```
centroids_time2 = [np.c_[x, ind *np.ones(x.shape[0])] for ind, x in enum
erate(centroids_time)]
#centroids_time2[6]
```

Now we can concatenate this list of arrays into one large array that we are then going to transform into a dataframe

In [17]:
```
centroids_time2 = np.concatenate(centroids_time2)
centroids_time2
```

Out[17]:
```
array([[ 44.60991736, 617.96859504,   0.        ],
       [ 66.87583893, 525.50503356,   0.        ],
       [ 69.8377193 , 214.86403509,   0.        ],
       ...,
       [392.24482109, 507.03578154,   9.        ],
       [397.68828452, 456.37656904,   9.        ],
       [401.73901099, 294.92582418,   9.        ]])
```

Let's simply pass that array to Pandas:

In [18]:
```
pd.DataFrame(centroids_time2)
```

Out[18]:

	0	1	2
0	44.609917	617.968595	0.0
1	66.875839	525.505034	0.0
2	69.837719	214.864035	0.0
3	84.217116	344.353407	0.0
4	87.518409	610.238586	0.0
5	92.680292	443.620438	0.0
6	102.700752	536.621053	0.0
7	111.597923	308.824926	0.0
8	110.965699	656.401055	0.0
9	111.904153	96.333866	0.0
10	124.475000	385.454167	0.0
11	126.619847	177.270229	0.0
12	125.789174	243.280627	0.0
13	133.640000	499.158182	0.0
14	135.221003	587.832288	0.0
15	140.683748	445.540264	0.0
16	155.810651	652.556213	0.0
17	163.572843	113.851485	0.0
18	161.836915	332.108723	0.0
19	162.773829	552.245557	0.0
20	166.139059	20.282209	0.0
21	177.107994	404.063114	0.0
22	189.304945	463.741758	0.0
23	189.364353	511.083596	0.0
24	193.846939	272.607143	0.0
25	192.450355	627.601064	0.0
26	203.456770	201.928222	0.0
27	210.922010	555.934142	0.0
28	215.804094	59.897661	0.0
29	218.667190	328.299843	0.0
...	...	...	...
591	261.488584	18.415525	9.0
592	256.252083	521.881250	9.0
593	277.380328	38.986885	9.0
594	264.311734	404.861646	9.0
595	269.465693	116.259854	9.0
596	268.803468	351.578035	9.0
597	270.057569	651.000000	9.0

Not too bad. The x, y and time columns of our arrays are now integrated into a dataframe.

We'd like now to change the headers of our dataframe. In the help we saw that there was on optional field called columns.
We can give the appropriate name there:

In [19]:
```
coords_dataframe = pd.DataFrame(centroids_time2, columns=('x','y','frame
'))
coords_dataframe
```

Out[19]:

	x	y	frame
0	44.609917	617.968595	0.0
1	66.875839	525.505034	0.0
2	69.837719	214.864035	0.0
3	84.217116	344.353407	0.0
4	87.518409	610.238586	0.0
5	92.680292	443.620438	0.0
6	102.700752	536.621053	0.0
7	111.597923	308.824926	0.0
8	110.965699	656.401055	0.0
9	111.904153	96.333866	0.0
10	124.475000	385.454167	0.0
11	126.619847	177.270229	0.0
12	125.789174	243.280627	0.0
13	133.640000	499.158182	0.0
14	135.221003	587.832288	0.0
15	140.683748	445.540264	0.0
16	155.810651	652.556213	0.0
17	163.572843	113.851485	0.0
18	161.836915	332.108723	0.0
19	162.773829	552.245557	0.0
20	166.139059	20.282209	0.0
21	177.107994	404.063114	0.0
22	189.304945	463.741758	0.0
23	189.364353	511.083596	0.0
24	193.846939	272.607143	0.0
25	192.450355	627.601064	0.0
26	203.456770	201.928222	0.0
27	210.922010	555.934142	0.0
28	215.804094	59.897661	0.0
29	218.667190	328.299843	0.0
...	...	...	...
591	261.488584	18.415525	9.0
592	256.252083	521.881250	9.0
593	277.380328	38.986885	9.0
594	264.311734	404.861646	9.0
595	269.465693	116.259854	9.0
596	268.803468	351.578035	9.0
597	270.057569	651.000000	9.0

That's it! We now have an appropriately formated dataframe to pass to our linking function, which required x,y and frame columns. Information can be retried from dataframes in similar ways as from Numpy arrays or Python dictionaries. For example, one can select a column (the head function limits the output):

```
In [20]: coords_dataframe['x'].head()
```

```
Out[20]: 0    44.609917
         1    66.875839
         2    69.837719
         3    84.217116
         4    87.518409
         Name: x, dtype: float64
```

One can access a specific row using its index:

```
In [21]: coords_dataframe.loc[0]
```

```
Out[21]: x         44.609917
         y        617.968595
         frame      0.000000
         Name: 0, dtype: float64
```

And one can use logical indexing. For example one can find all the lines corresponding to a given time frame, and extract them:

```
In [22]: coords_dataframe[coords_dataframe['frame']==0].head()
```

Out[22]:

	x	y	frame
0	44.609917	617.968595	0.0
1	66.875839	525.505034	0.0
2	69.837719	214.864035	0.0
3	84.217116	344.353407	0.0
4	87.518409	610.238586	0.0

A dataframe and its contents have also a series of methods attached to them. For example we can get the maximum value from a given columns like this:

```
In [23]: coords_dataframe['x'].max()
```

```
Out[23]: 409.8050595238095
```

Pandas and Numpy are very close, so of course we could also have used the Numpy function:

```
In [24]: np.max(coords_dataframe['x'])
```

```
Out[24]: 409.8050595238095
```

Using the Pandas package would be a course on itself as it is a very powerful tool to handle tabular data. We just showed some very basic features here so that what follows makes sense. Note that this is a situation that occurs often: you just need a few features of a package within a larger project, and have to figure out the basics of it. However, if you work with large tabular data, learning Pandas is highly recommended.

11.3.2 Tracking

There are multiple options in the tracking function. *E.g.* in how many frames a signal is allowed to disappear, how we calculate distances between objects *etc.* We are only going to give a value for the fields search_range which specifies in what neighborhood one is doing the tracking.

In [25]:
```
tracks = trackpy.link_df(coords_dataframe, search_range=20)
```

```
Frame 9: 63 trajectories present.
```

The output is a new dataframe. It contains the position (x,y,frame) of each particle, and to what track (particle) it belongs:

In [26]:
```
tracks.head()
```

Out[26]:

	x	y	frame	particle
0	44.609917	617.968595	0	0
33	248.584356	137.056748	0	1
34	255.506154	227.063077	0	2
35	260.481848	524.721122	0	3
36	268.189189	384.758347	0	4

We have seen before that we can use indexing. So let's do that to recover all the points forming for example the trajectory = 10

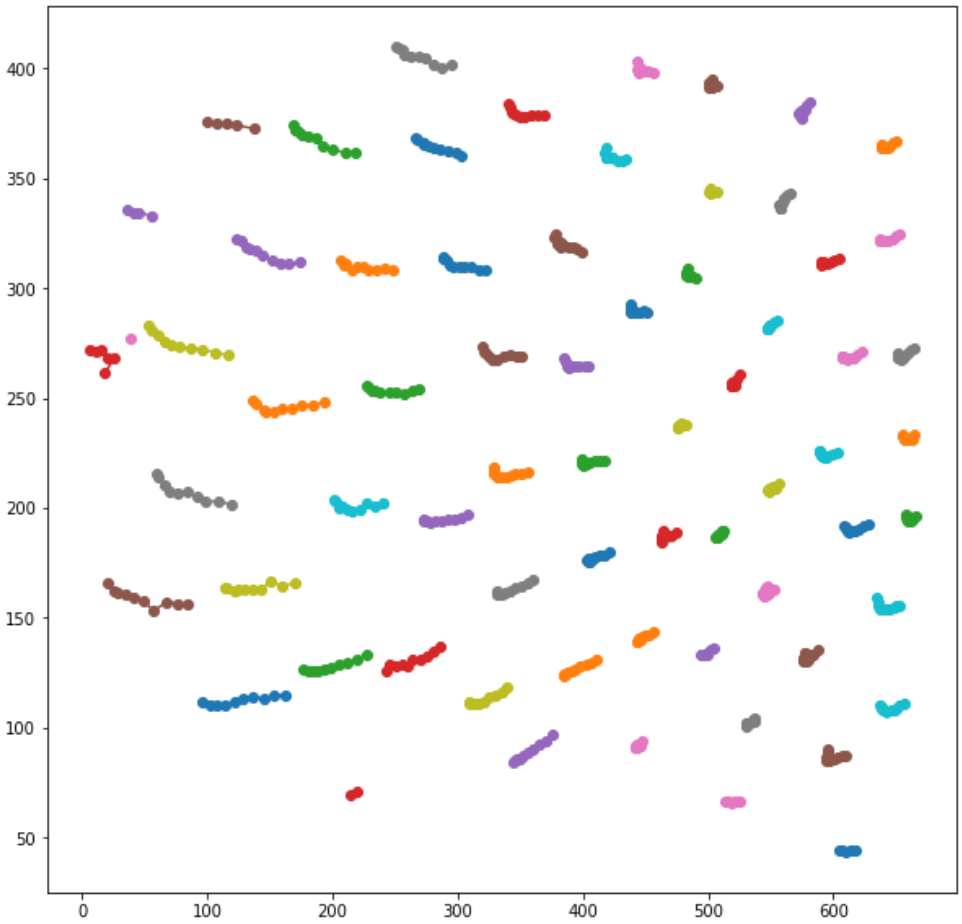
In [27]:
```
tracks[tracks['particle']==10]
```

Out[27]:

	x	y	frame	particle
42	292.320814	437.802817	0	10
103	290.185759	437.803406	1	10
163	288.868012	438.596273	2	10
225	288.651537	439.784773	3	10
288	288.668721	439.288136	4	10
350	289.728213	440.728213	5	10
413	288.701534	443.525802	6	10
476	288.875000	445.761765	7	10
538	289.774924	448.592145	8	10
600	289.171131	451.400298	9	10

We see that in that particular case, we have one point per frame and the successive points seem close together, so the tracking seems to have worked properly. We can recover all such trajectories and plot them on a single xy plot:

In [28]:
```
plt.figure(figsize=(10,10))
for particle_id in range(tracks['particle'].max()):
    plt.plot(tracks[tracks.particle==particle_id].y,tracks[tracks.partic
le==particle_id].x,'o-')
plt.show()
```



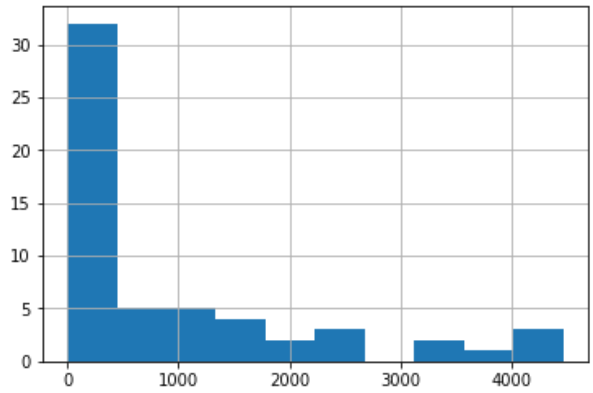
11.4 Analysing the data

Now that we have those tracks, we can finally do some quantification of the process. For example we can measure what is the largest distance traveled by each *nucleus*.

In [29]:
```
msd = trackpy.imsd(tracks,1,1)
```

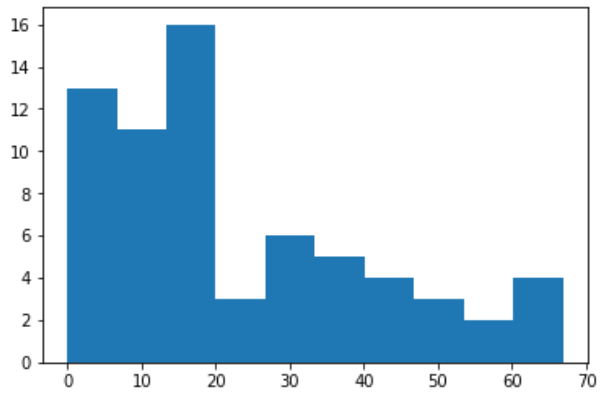
In [30]:
```
msd.loc[9].hist()
```

Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x7f011e0bf7f0>



In [31]:
```
distances = []
for particle_id in range(tracks['particle'].max()):
    #recover current track
    current_track = tracks[tracks.particle==particle_id]

    #find beginning and end of track
    min_time = np.min(current_track['frame'])
    max_time = np.max(current_track['frame'])

    #get positions at begin and end and measure distance
    x1 = current_track[current_track['frame']==min_time].iloc[0].x
    y1 = current_track[current_track['frame']==min_time].iloc[0].y
    x2 = current_track[current_track['frame']==max_time].iloc[0].x
    y2 = current_track[current_track['frame']==max_time].iloc[0].y

    distances.append(np.sqrt((x2-x1)**2+(y2-y1)**2))
```

In [32]:
```
plt.hist(distances)
plt.show()
```



As we could have guesses from looking at the displacement plot, we have two categories of *nucle*: those that move on the left of the image, and those that don't on the right.

12. Image registration

Image registration consists in aligning two images so that objects in them can be "aligned". This alignement can occur in space, as for example in the case of tomography where successive stacks are slightly shifted, or in time, for example when there is drift in a time-lapse acquisition.

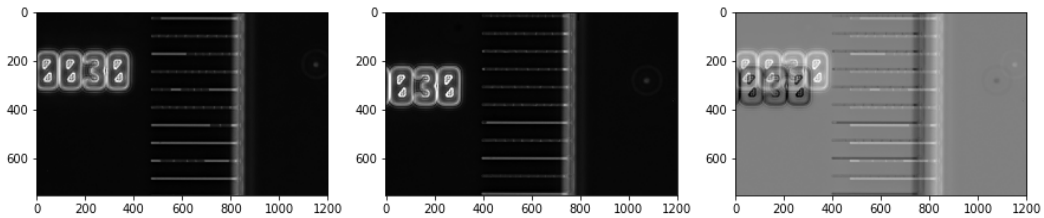
There are many methods, more or less complicated to do this registration, which can involve shits, rotation and deformation.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io

from skimage.feature import match_template
from skimage.filters import threshold_otsu

plt.gray()
```

In [2]:
```
image1 = io.imread('Data/channels/channels1.tif')
image2 = io.imread('Data/channels/channels2.tif')
```

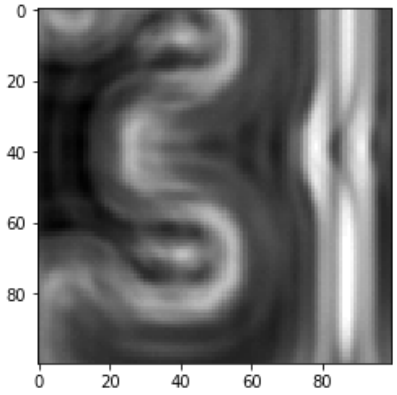
In [3]:
```
image1 = image1[0:750,:]
image2 = image2[0:750,:]
```

In [4]:
```
fig, ax = plt.subplots(1,3, figsize=(15,10))
ax[0].imshow(image1)
ax[1].imshow(image2)
ax[2].imshow(image1.astype(float)-image2.astype(float));
```



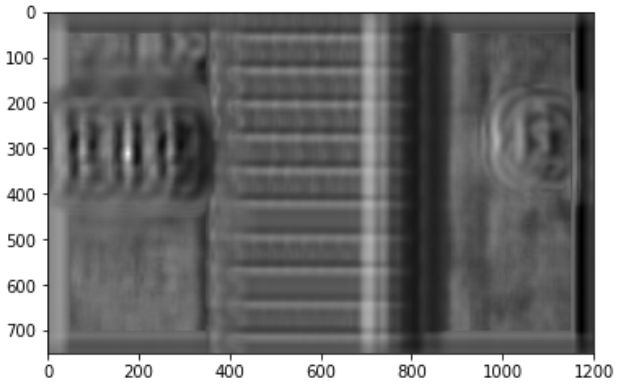
12.1 Simple approach

In [5]:
```
plt.imshow(image1[200:300,200:300]);
```



In [6]:
```
matched = match_template(image2, image1[200:300,200:300],pad_input=True)
```

In [7]:
```
plt.imshow(matched);
```



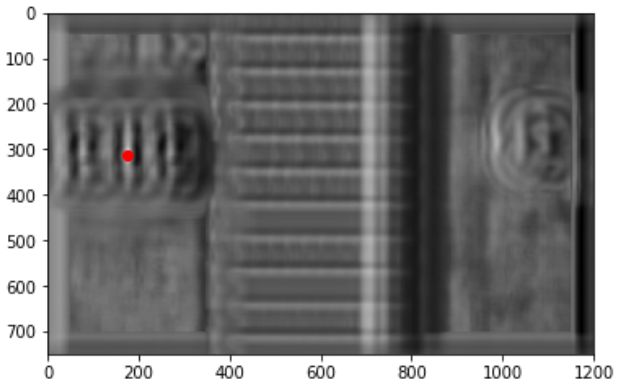
In [8]:
```
maxpos = np.argmax(matched)
maxpos
```

Out[8]: 374576

In [9]:
```
maxpos = np.unravel_index(maxpos, matched.shape)
maxpos
```

Out[9]: (312, 176)

In [10]:
```
plt.imshow(matched)
plt.plot([maxpos[1]],[maxpos[0]],'ro');
```

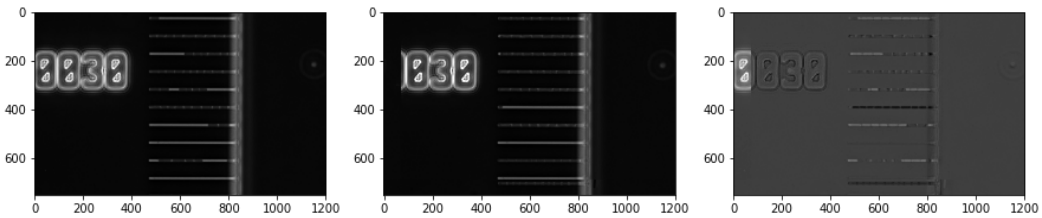


In [11]:
```
template_center = 250
maxpos = template_center - np.array(maxpos)
maxpos
```

Out[11]:
```
array([-62,  74])
```

In [12]:
```
image2_shift = np.roll(image2,shift = maxpos,axis = (0,1))
```

In [13]:
```
fig, ax = plt.subplots(1,3, figsize=(15,10))
ax[0].imshow(image1)
ax[1].imshow(image2_shift)
ax[2].imshow(image1.astype(float)-image2_shift.astype(float));
```



12.2 General approach

In [14]:
```
from skimage.feature import ORB, match_descriptors, plot_matches
from skimage.measure import ransac
from skimage.transform import AffineTransform
```

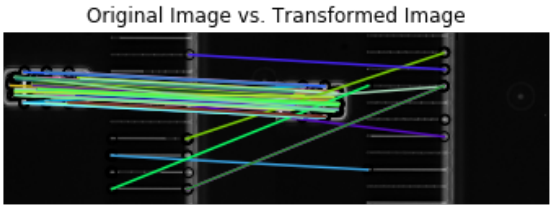
In [15]:
```
detector_extractor1 = ORB(n_keypoints=200)
detector_extractor2 = ORB(n_keypoints=200)
```

In [16]:
```
detector_extractor1.detect_and_extract(image1)
keypoints1 = detector_extractor1.keypoints
descriptors1 = detector_extractor1.descriptors


detector_extractor2.detect_and_extract(image2)
keypoints2 = detector_extractor2.keypoints
descriptors2 = detector_extractor2.descriptors
```

In [17]:
```
matches12 = match_descriptors(descriptors1, descriptors2, cross_check=True)
```

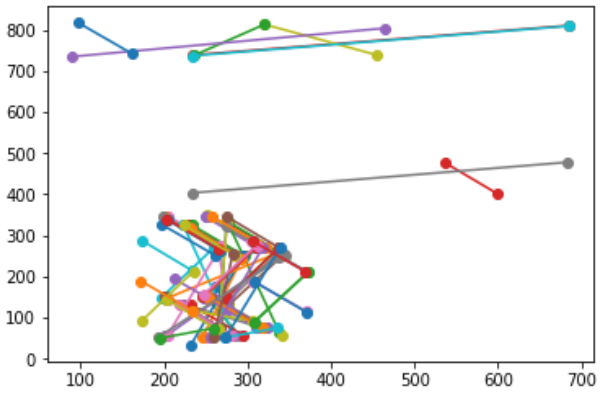
In [18]:
```
fig, ax = plt.subplots()

plt.gray()

plot_matches(ax, image1, image2, keypoints1, keypoints2, matches12)
ax.axis('off')
ax.set_title("Original Image vs. Transformed Image")
```

Out[18]: Text(0.5, 1.0, 'Original Image vs. Transformed Image')



In [19]:
```
coords1 = keypoints1[matches12[:, 0]]
coords2 = keypoints2[matches12[:, 1]]
```

In [20]:
```
for x1, x2 in zip(coords1, coords2):
    plt.plot([x1[0],x2[0]],[x1[1],x2[1]],'-o')
plt.show()
```



In [21]:
```
#coords1 = coords1[(coords1[:,0]<400)&(coords1[:,1]<400),:]
#coords2 = coords2[(coords2[:,0]<400)&(coords2[:,1]<400),:]
```

In [22]:
```
model = AffineTransform()
model.estimate(coords1, coords2)
```

Out[22]: True

In [23]:
```
print(model.scale, model.translation, model.rotation)
```

```
(3.9821515141194355, 1.461939042859838) [-427.31749233  -83.68571398] 0.1
2738167750904475
```

In [24]:
```
model_robust, inliers = ransac((coords1, coords2), AffineTransform, min_
samples=3, residual_threshold=0.1)
```

In [25]:
```
print(model_robust.scale, model_robust.translation, model_robust.rotatio
n)
```

```
(1.0000017322082522, 1.0018543351803184) [ 62.20768514 -74.84401361] -0.0
003090630277479058
```

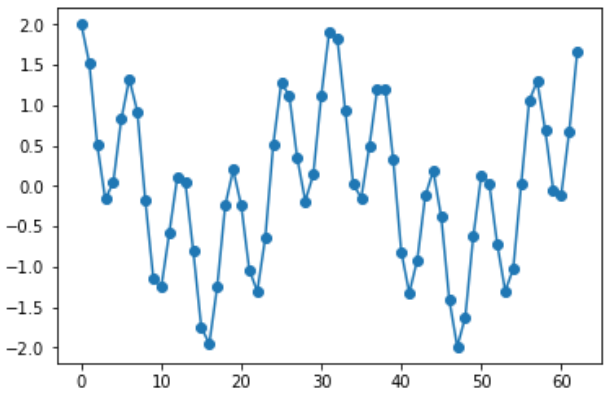
12.3 Fourier transform and rotation

If you have repetitive signal in your image, like here the channels, you can exploit it in your analysis. For example if your image is not clearly horizontal an you want to align it, you can use the power of Fourier transforms to do the job.

A fourier transform is a way to describe any signal as an infinite sum of periodic signals. Very roughly, each component of that sum has an amplitude given by the Fourier transform. In real world application one doesn't have continuous and infinite signals and one has to use an approximate Fourier transform called Fast Fourier Transform or FFT. Naturally this works not just in one dimension but also in two.

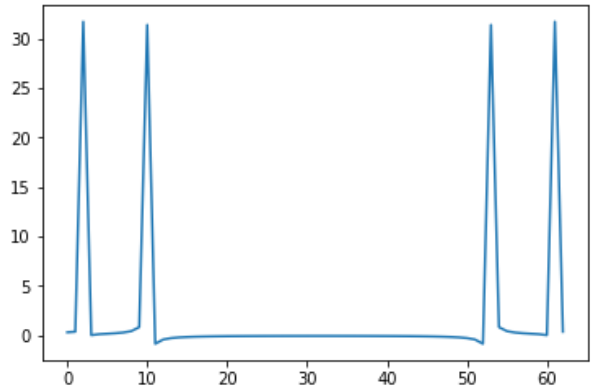
Quick 1d reminder:

In [26]:
```
#creaete x positions
x = np.arange(0,10*2*np.pi,1)

#pick frequencies
freq = 0.1
freq2 = 0.5

#generate signal
y = np.cos(2*freq*x) + np.cos(2*freq2*x)

#plot the result
plt.plot(x,y,'-o')
plt.show()
```



In [27]:
```
from scipy.signal import find_peaks

#fourier transform the signal
fourier = np.fft.fft(y)

#plot it
plt.plot(fourier)
plt.show()

#find peaks position and recover frequency
print(find_peaks(fourier[0:300])[0][0]/20)
print(find_peaks(fourier[0:300])[0][1]/20)
```

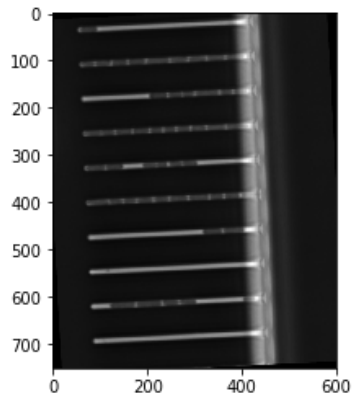
```
/usr/local/lib/python3.5/dist-packages/numpy/core/numeric.py:501: Complex
Warning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```




```
0.1
0.5
```

Now we look at the 2D case. For the purpose of the example, let's focus on the area that contains channels and create a slightly rotated version of one of the images and see how we can use Fourier transforms to correct it.

In [28]:
```
import skimage.transform

angle = 3
image_rotate = image1[:,400:1000]
image_rotate = skimage.transform.rotate(image_rotate,angle,cval=0)

plt.imshow(image_rotate)
plt.show()
```

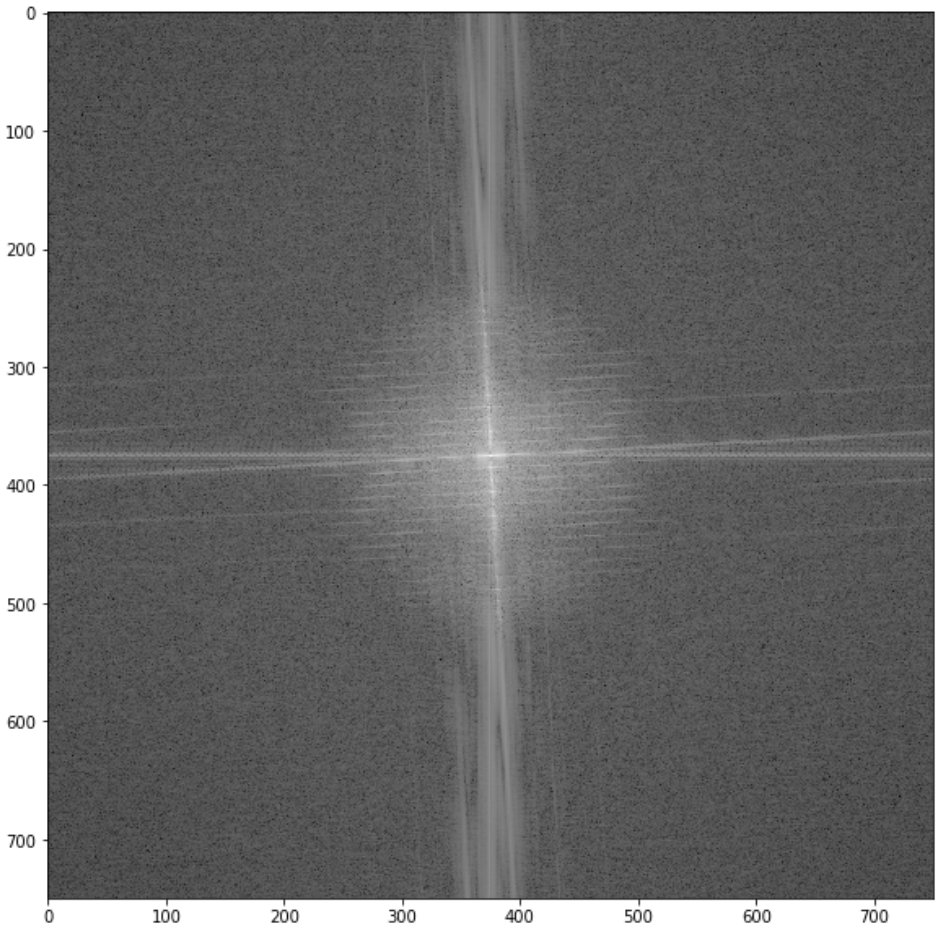


To avoid confusions with dimensions, let's make the image square by padding it.

In [29]:
```
image_pad = np.pad(image_rotate,((0,0),(image_rotate.shape[0]-image_rota
te.shape[1],0)),mode = 'constant',constant_values = 0)
```

Now we take the 2D transform of the signal and plot it. We also shift use a shifting function so that low frequency signal ends up in the middle of the image:

In [30]:
```
f0 = np.fft.fftshift(np.abs(np.fft.fft2(image_pad)))

plt.figure(figsize=(10,10))
plt.imshow(np.log(f0))
plt.show()
```



To find the rotation angle, we can now rotate the fourier transform using a range of angles, and project along the vertical axis. Once the cross visible in the middle of the image is aligned, its projection should show the maximal values:

In [31]:
```
allproj = []

#rotate the fourier transform and do a max projection
for i in np.arange(-10,10,1):
    basicim = skimage.transform.rotate(f0,i,cval=0)

    allproj.append(np.max(np.sum(basicim,axis=0)))

#find maximum angle
angle = np.arange(-10,10,1)[np.argmax(allproj)]
```

In [33]:
```
angle
```

Out[33]:
```
-3
```

13. Pixel classification

We have for the moment mostly seen methods that rely on pixel intensity and shapes of objects to segment features. When dealing with natural images (typical RGB images) one can however also exploit the fact that the channels taken together give information on the image structure. To illustrate this we are going to use a classical clustering method (Kmeans) found in the package scikit-learn. That package is the reference for anyone who wants to apply machine learning methods to their data. It is a nice pendant to scikit-image as it also has a simple syntax, a good documentation and many examples.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray
import sklearn.cluster
import skimage.io
```

We are going to deal again with a geography satellite image that can be loaded here:

In [2]:
```
image = skimage.io.imread('Data/geography/naip/m_3910505_nw_13_1_2015091
9/crop/m_3910505_nw_13_1_20150919_crop.tif')
```

```
/usr/local/lib/python3.5/dist-packages/skimage/external/tifffile/tifffil
e.py:2617: RuntimeWarning: py_decodelzw encountered unexpected end of str
eam
  strip = decompress(strip)
/usr/local/lib/python3.5/dist-packages/skimage/external/tifffile/tifffil
e.py:2552: UserWarning: unpack: buffer size must be a multiple of element
size
  warnings.warn("unpack: %s" % e)
```

Let's just keep the first three RGB channels (no clue what the fourth one is...)

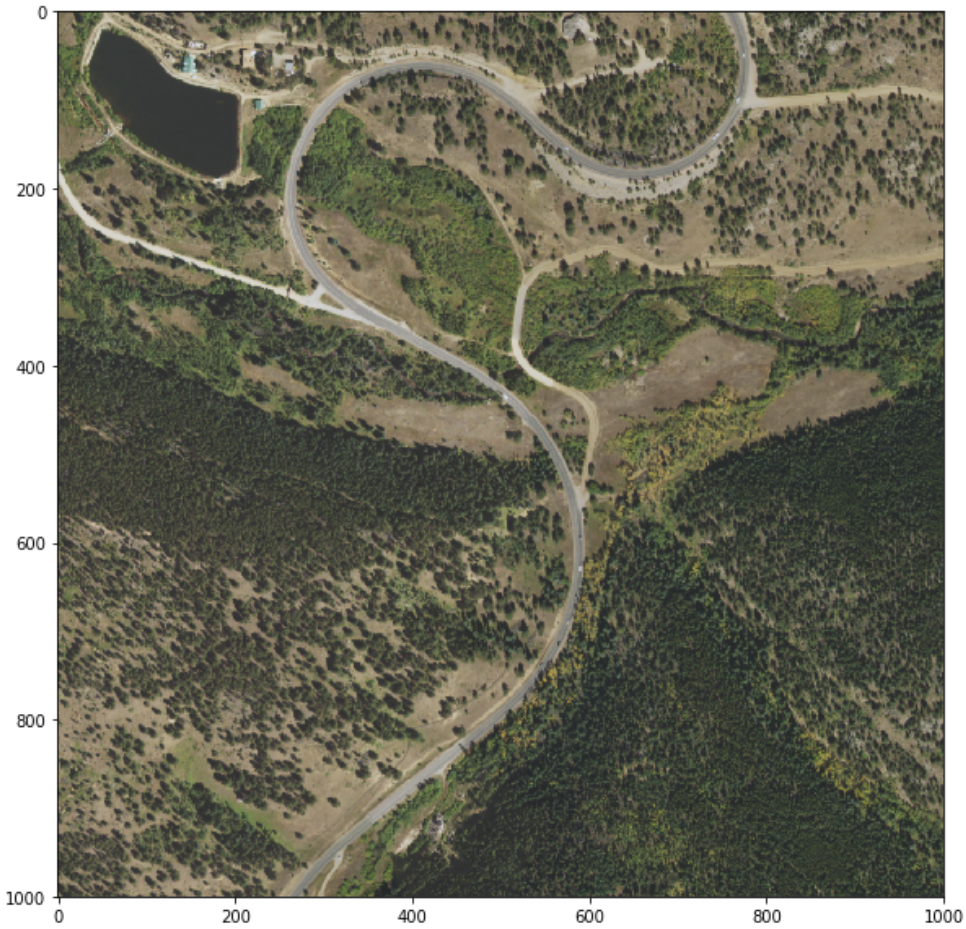
In [3]:
```
image = image[:,:,0:3]
```

In [4]:
```
plt.figure(figsize=(20,10))
plt.imshow(image);
```

The image is quite large, so let's focus on a smaller region first, to reduce computational time:

In [5]:
```
subim = image[0:1000,0:1000,:]
```

In [6]:
```
plt.figure(figsize=(20,10))
plt.imshow(subim)
plt.show()
```

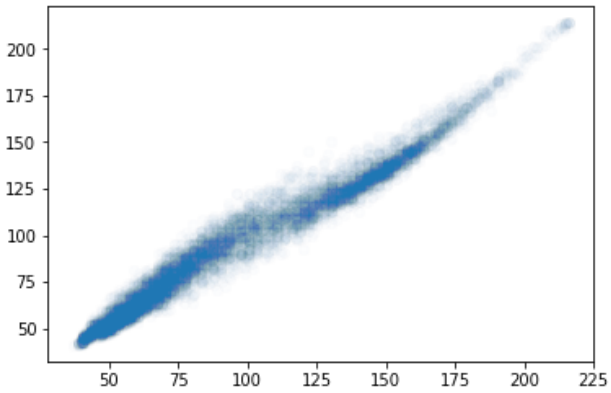


If we want to use a clustering approach, *i.e*. grouping pixels which have similar features, we have to reshape our image into an actual dataset where each pixel is a datapoint with three "properties", in this case RGB.

In [7]:
```
X = np.reshape(subim,(subim.shape[0]*subim.shape[1],3))
```

We can have a look at how this dataset looks like. Let's plot the first and second "features". We reduce the number of data points and make them transparent so that we don't saturate the plot:

In [8]:
```
plt.plot(X[::100,0],X[::100,1],'o',alpha = 0.01)
plt.show()
```



We see by eye that we have at least two categories, with two levels of Red/Green. Let's do some clustering just on these two components to better understand what happens for the image.

We are going to feed the Kmeans algorithm with a dataset containing the Red and Green features and say that we want two categories in the end. The algorithm is going to iteratively assign each pixel to one category, and is certain to converge. Of course there are other clustering methods that you can use in sklearn.

In [9]:
```
kmeans = sklearn.cluster.KMeans(n_clusters=2, random_state=0).fit(X[:,0:
2])
```

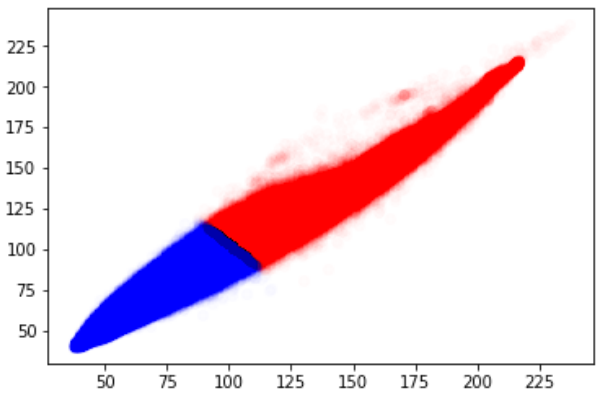
The labels of each element are stored in here:

In [10]:
```
kmeans.labels_
```

Out[10]:
```
array([0, 0, 0, ..., 1, 1, 1], dtype=int32)
```

Let's plot them by selecting them by label:

In [11]:
```
plt.plot(X[kmeans.labels_ == 0,0],X[kmeans.labels_ == 0,1],'ro',alpha =
0.01)
plt.plot(X[kmeans.labels_ == 1,0],X[kmeans.labels_ == 1,1],'bo',alpha =
0.01)
plt.show()
```

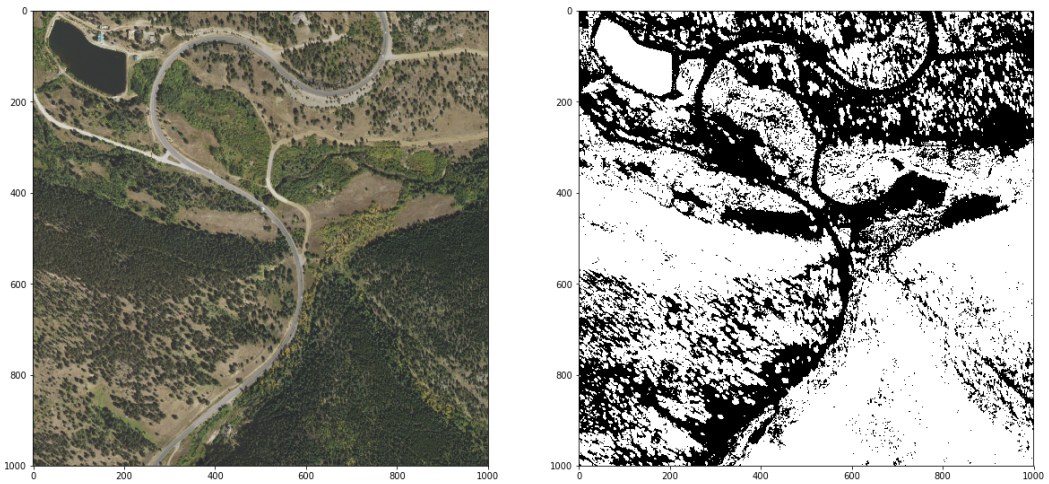


We see thats the algorith split the sample more or less at the expected position. Let's use now all the components and classify our pixels

In [12]:
```
kmeans = sklearn.cluster.KMeans(n_clusters=2, random_state=0).fit(X)
```

In [13]:
```
labels_im = np.reshape(kmeans.labels_,(1000,1000))
```

In [14]:
```
fig,ax = plt.subplots(1,2, figsize = (20,10))
ax[0].imshow(subim)
ax[1].imshow(labels_im,cmap = 'gray');
```

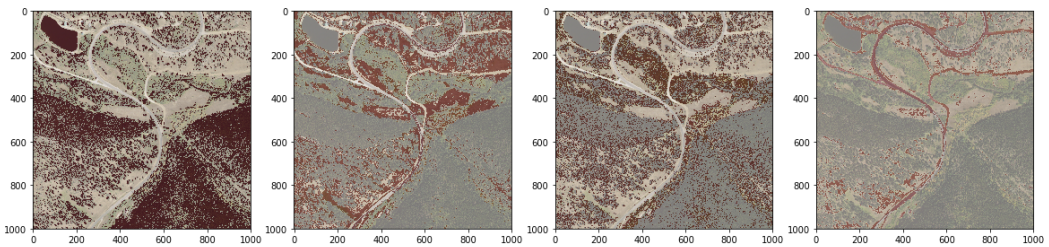


We see that we managed to plsit really well the data into forest and other types (roads, earth). Of course we couls use more categories. Maybe with four categories we could split roads, light forest, dark forest and earth. Let's do that and superpose each category to the original image.

In [15]:
```
kmeans = sklearn.cluster.KMeans(n_clusters=4, random_state=0).fit(X)
```

In [16]:
```
labels_im = np.reshape(kmeans.labels_,(1000,1000))
```

In [17]:
```
fig,ax = plt.subplots(1,4, figsize = (20,10))
for i in range(4):
    ax[i].imshow(subim)
    ax[i].imshow(labels_im==i,cmap = 'Reds', alpha = 0.4);
```



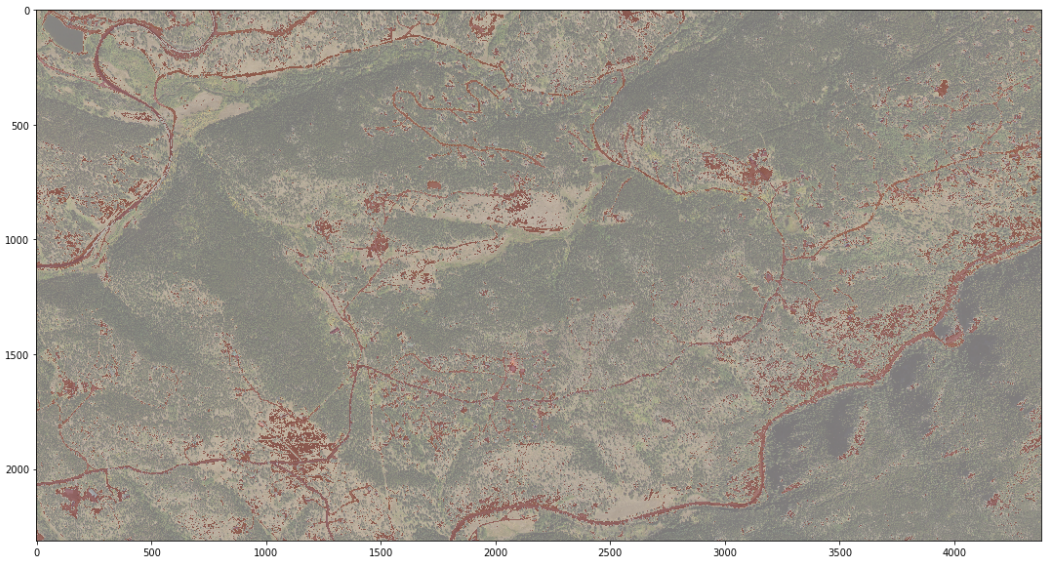
Of course this is a very crude approach, but we still managed to nicely recover different features on that image in only a few lines. The dataset for the entire image is huge and Kmeans clustering would be very time consuming. However we can just re-use the model we trained on the smaller image to classify all the pixels of the image:

In [18]:
```
X_large = np.reshape(image,(image.shape[0]*image.shape[1],3))
```

In [19]:
```
labels_large = kmeans.predict(X_large)
```

In [20]:
```
labels_im = np.reshape(labels_large,(image.shape[0],image.shape[1]))
```

In [21]:
```
plt.figure(figsize=(20,10))
plt.imshow(image)
plt.imshow(labels_im==3,cmap = 'Reds', alpha = 0.4);
```



14. Image classification by machine learning: Optical text recognition

There are different types of machine learning. In some cases, like in the pixel classification task, the algorithm does the classification on its own by trying to optimize groups according to a given rule (unsupervised). In other cases one has to feed the algorithm with a set of annotated examples to train it (supervised). Here we are going to train an ML algorithm to recognize digits. Therefore the first things that we need is a good set of annotated examples. Luckily, since this is a "popular" problem, one can find such datasets on-line. In general, this is not the case, and one has to manually create such a dataset. Then one can either decide on a set of features that the algorithm has to use for learning or let the algorithm define those itself. Here we look at the first case, and we will look at the second one in the following chapters.

Note that this notebooks does not present a complete OCR solution. The goal is rather to show the underlying principles of machine learning methods used for OCR.

In [1]:
```
import glob
import numpy as np
import matplotlib.pyplot as plt
plt.gray()
import pandas as pd
import skimage
import skimage.feature
import skimage.io
```

14.1 Exploring the dataset

We found a good dataset here (http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/) and downloaded it. Let's first have a look at it.

We have a folder with 62 sub-folders corresponding to digits and lower and upper-case characters:

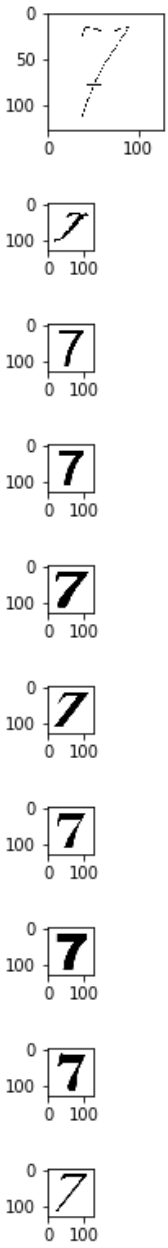
In [2]:
```
data_path = 'Data/Fonts/English/Fnt/'

samples = np.sort(glob.glob(data_path+'*'))
print(samples)
```

```
['Data/Fonts/English/Fnt/Sample001' 'Data/Fonts/English/Fnt/Sample002'
 'Data/Fonts/English/Fnt/Sample003' 'Data/Fonts/English/Fnt/Sample004'
 'Data/Fonts/English/Fnt/Sample005' 'Data/Fonts/English/Fnt/Sample006'
 'Data/Fonts/English/Fnt/Sample007' 'Data/Fonts/English/Fnt/Sample008'
 'Data/Fonts/English/Fnt/Sample009' 'Data/Fonts/English/Fnt/Sample010'
 'Data/Fonts/English/Fnt/Sample011' 'Data/Fonts/English/Fnt/Sample012'
 'Data/Fonts/English/Fnt/Sample013' 'Data/Fonts/English/Fnt/Sample014'
 'Data/Fonts/English/Fnt/Sample015' 'Data/Fonts/English/Fnt/Sample016'
 'Data/Fonts/English/Fnt/Sample017' 'Data/Fonts/English/Fnt/Sample018'
 'Data/Fonts/English/Fnt/Sample019' 'Data/Fonts/English/Fnt/Sample020'
 'Data/Fonts/English/Fnt/Sample021' 'Data/Fonts/English/Fnt/Sample022'
 'Data/Fonts/English/Fnt/Sample023' 'Data/Fonts/English/Fnt/Sample024'
 'Data/Fonts/English/Fnt/Sample025' 'Data/Fonts/English/Fnt/Sample026'
 'Data/Fonts/English/Fnt/Sample027' 'Data/Fonts/English/Fnt/Sample028'
 'Data/Fonts/English/Fnt/Sample029' 'Data/Fonts/English/Fnt/Sample030'
 'Data/Fonts/English/Fnt/Sample031' 'Data/Fonts/English/Fnt/Sample032'
 'Data/Fonts/English/Fnt/Sample033' 'Data/Fonts/English/Fnt/Sample034'
 'Data/Fonts/English/Fnt/Sample035' 'Data/Fonts/English/Fnt/Sample036'
 'Data/Fonts/English/Fnt/Sample037' 'Data/Fonts/English/Fnt/Sample038'
 'Data/Fonts/English/Fnt/Sample039' 'Data/Fonts/English/Fnt/Sample040'
 'Data/Fonts/English/Fnt/Sample041' 'Data/Fonts/English/Fnt/Sample042'
 'Data/Fonts/English/Fnt/Sample043' 'Data/Fonts/English/Fnt/Sample044'
 'Data/Fonts/English/Fnt/Sample045' 'Data/Fonts/English/Fnt/Sample046'
 'Data/Fonts/English/Fnt/Sample047' 'Data/Fonts/English/Fnt/Sample048'
 'Data/Fonts/English/Fnt/Sample049' 'Data/Fonts/English/Fnt/Sample050'
 'Data/Fonts/English/Fnt/Sample051' 'Data/Fonts/English/Fnt/Sample052'
 'Data/Fonts/English/Fnt/Sample053' 'Data/Fonts/English/Fnt/Sample054'
 'Data/Fonts/English/Fnt/Sample055' 'Data/Fonts/English/Fnt/Sample056'
 'Data/Fonts/English/Fnt/Sample057' 'Data/Fonts/English/Fnt/Sample058'
 'Data/Fonts/English/Fnt/Sample059' 'Data/Fonts/English/Fnt/Sample060'
 'Data/Fonts/English/Fnt/Sample061' 'Data/Fonts/English/Fnt/Sample062']
```

Let's check the contents by plotting the first 5 images of a folder:

In [3]:
```
files = glob.glob(samples[7]+'/*.png')
```

In [4]:
```
plt.figure(figsize=(15,15))
for i in range(10):
    plt.subplot(1,10,i+1)
    image = skimage.io.imread(files[i])
    plt.imshow(image)
    plt.show()
```



So we have samples of each character written with different fonts and types (italic, bold).

14.2 Classifying digits

We are first going to try to classify digits. Our goal is to be able to pass an image of the type shown above to our ML algorithm so that the latter can say what digit is present in that image.

First, we have to decide what information the algorithm should use to make that decision. The simplest thing to do is to just say that each pixel is a "feature", and thus to use a flattened version of each image as feature space.

So that the process is a bit faster we are going to rescale all the images to 32x32 pixels so that we have 32^2 features.

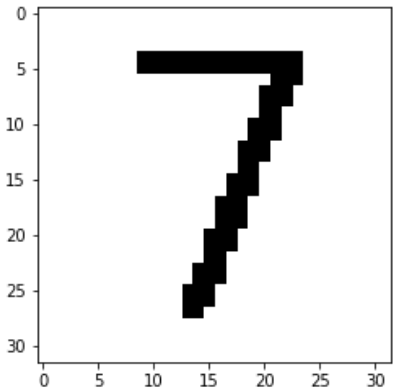
14.2.1 Loading and scaling images

For each digit, we load 50 images by randomly selecting them. We rescale them and reshape them in a single comprehension list. Let's see what happens for one digit:

In [5]:
```
data = [np.reshape((skimage.transform.rescale(skimage.io.imread(files
[x]),1/4,order = 1)>0.1).astype(np.uint8),32**2)
        for x in np.random.choice(np.arange(len(files)),10,replace=Fals
e)]
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:24: Us
erWarning: The default multichannel argument (None) is deprecated.  Pleas
e specify either True or False explicitly.  multichannel will default to
False starting with release 0.16.
  warn('The default multichannel argument (None) is deprecated.  Please '
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

In [6]:
```
plt.imshow(np.reshape(data[2],(32,32)),cmap = 'gray')
plt.show()
```



Now let's do this for all digits and aggregate all these data into `all_data`:

In [7]:
```
num_samples = 500

all_data = []
for ind, s in enumerate(samples[0:10]):
    files = glob.glob(s+'/*.png')

    data = np.array([np.reshape((skimage.transform.rescale(skimage.io.im
read(files[x]),1/4)>0.1).astype(np.uint8),32**2)
        for x in np.random.choice(np.arange(len(files)),num_samples,repl
ace=False)])

    all_data.append(data)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:24: Us
erWarning: The default multichannel argument (None) is deprecated.  Pleas
e specify either True or False explicitly.  multichannel will default to
False starting with release 0.16.
  warn('The default multichannel argument (None) is deprecated.  Please '
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

Now we concatenate all these data into one single matrix:

In [8]:
```
data = np.concatenate(all_data,axis = 0)
```

In [9]:
```
data.shape
```

Out[9]:
```
(5000, 1024)
```

14.2.2 Creating categories

We have 50 examples for 10 digits and each example has 1024 features. We also need to create an array that contains the information "what digit is present at each row of the `data` array. We have 500 times a list of each digit:

In [10]:
```
cats = [str(i) for i in range(len(all_data))]
category = np.concatenate([[cats[i] for j in range(num_samples)] for i i
n range(len(cats))])
```

In [11]:
```
category
```

Out[11]:
```
array(['0', '0', '0', ..., '9', '9', '9'], dtype='<U1')
```

14.2.3 Running the ML algorithm

Now we are ready to use our dataset of features and our corresponding list of categories to train a classifier. We are going to use here a Random Forest classifier implement in scikit-learn:

In [12]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.metrics import confusion_matrix
```

First we have to split the dataset into a training and a testing dataset. It is very important to test the classifier on data that have not been seen previously by it!

In [13]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(data, category, random_s
tate=0)
```

Now we can do the actual learning:

In [14]:
```
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
```

Out[14]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini
',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=None,
                       oob_score=False, random_state=None, verbose=0,
                       warm_start=False)
```

Finally we can verify the predictions on the test dataset. The predict function returns a list of the category to which each testing sample has been assigned.

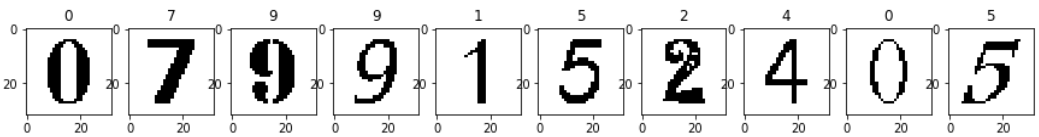
In [15]:
```
ypred = model.predict(Xtest)
```

In [16]:
```
ypred
```

Out[16]:
```
array(['0', '7', '9', ..., '0', '6', '4'], dtype='<U1')
```

We can look at a few examples:

In [17]:
```
fig, ax = plt.subplots(1, 10, figsize = (15,10))
for x in range(10):
    ax[x].imshow(np.reshape(Xtest[x],(32,32)),cmap='gray')
    ax[x].set_title(ypred[x])
plt.show()
```



In order to get a more comprehensive view, we can look at some statistics:

In [18]:
```
print(metrics.classification_report(ypred, ytest))
```

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       120
           1       0.97      0.94      0.95       129
           2       0.97      0.99      0.98       117
           3       0.89      0.99      0.94       106
           4       0.95      0.97      0.96       111
           5       0.97      0.93      0.95       144
           6       0.96      0.89      0.92       123
           7       0.98      0.95      0.97       133
           8       0.94      0.94      0.94       139
           9       0.92      0.95      0.93       128

   micro avg       0.95      0.95      0.95      1250
   macro avg       0.95      0.95      0.95      1250
weighted avg       0.95      0.95      0.95      1250
```

We see that our very simple features, basically the pixel positions, and 50 examples per class are sufficient to reach a very good result.

14.3 Using the classifier on "real" data

Let's try to segment a real-life case: an image of a digital screen:

In [19]:
```
#jpg = skimage.io.imread('/Users/gw18g940/Desktop/Test_data/ImageProcessingCourse/digit.jpg')
jpg = skimage.io.imread('Data/mz_digit.jpg')
```

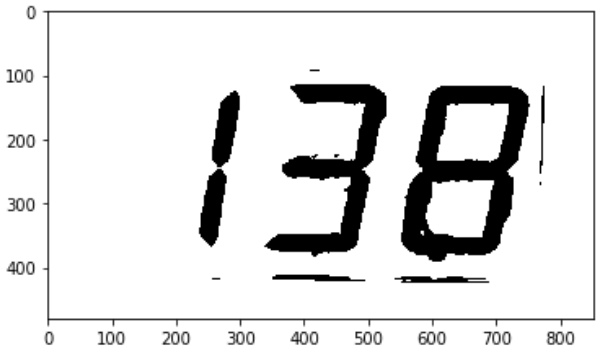
In [20]:
```
plt.imshow(jpg, cmap = 'gray')
plt.show()
```

14.3.1 Pre-processing

We trained our classifier on black and white pictures, so let's first convert the image and create a black and white version using a thresholder:

In [21]:
```
jpg = skimage.color.rgb2gray(jpg)
th = skimage.filters.threshold_li(jpg)
jpg_th = jpg<th
```

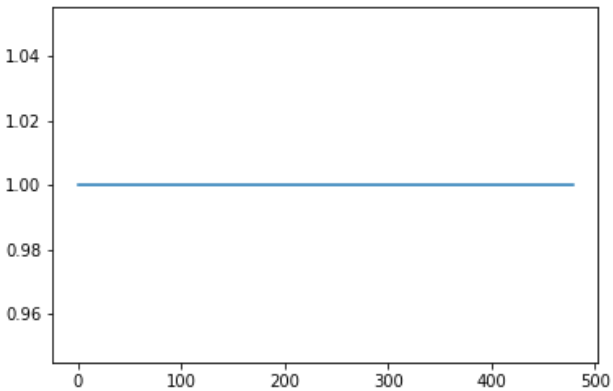
In [22]:
```
plt.imshow(jpg_th);
```



14.3.2 Identifying numbers

First we need to identify each single number present here in the second row. If we project the image along the horizontal direction, we clearly see an "empty" region. By detecting where the steps are, we can isolate the two lines of text:

In [23]:
```
plt.plot(np.max(jpg_th,axis = 1));
```



In [26]:
```
#create projection
proj = np.max(jpg_th,axis = 1)
#select "positive" regions and find their indices
regions = proj > 0.5
text_indices = np.arange(jpg.shape[0])[regions]
#find the steps and split the indices into two groups
splits = np.split(text_indices,np.where(np.diff(np.arange(jpg.shape
[0])[regions])>1)[0]+1)
```

In [31]:
```
plt.imshow(jpg[splits[1],:],cmap = 'gray')
plt.show()
```

To separate each digit we proceed in the same way by projecting along the vertical dimensions:

In [ ]:
```
#select line to process
line_ind = 1
proj2 = np.min(jpg[splits[line_ind],:],axis = 0)
regions = proj2 < 0.5
text_indices = np.arange(jpg.shape[1])[regions]
splits2 = np.split(text_indices,np.where(np.diff(np.arange(jpg.shape
[1])[regions])>1)[0]+1)
```

`splits2` contains all column indices for each digit:

In [30]:
```
characters = [jpg_th[splits[line_ind],x[0]:x[-1]] for x in splits2]
```

In [ ]:
```
[x.shape for x in characters]
```

In [ ]:
```
for ind, x in enumerate(characters):
    plt.subplot(1,10, ind+1)
    plt.imshow(x)
plt.show()
```

14.3.3 Rescaling

Since we rely on pixels positions as features, we have to make sure that the images we are passing to the classifier are similar to those used for training. Those had on average a height of 24 pixels. So let's rescale:

In [ ]:
```
im_re = (skimage.transform.rescale(characters[2],1/(characters[2].shape
[0]/24),
                                   preserve_range=True, order = 1, anti_a
liasing=False)>0.1).astype(np.uint8)
```

Additionally, the images are square and have 32 pixels. So let's pad our images. We do that by filling an empty image with our image at the middle. We also have to make sure that the intensity scale is correct:

In [ ]:
```
empty = np.zeros((32,32))
empty[int((32-im_re.shape[0])/2):int((32-im_re.shape[0])/2)+im_re.shape
[0],
     int((32-im_re.shape[1])/2):int((32-im_re.shape[1])/2)+im_re.shape
[1]] = im_re
empty = empty<0.5

to_pass = (1*empty).astype(np.uint8)
```

Finally we can pass this to the classifier:

In [ ]:
```
ypred = model.predict(np.reshape(to_pass,32**2)[np.newaxis,:])
fig,ax = plt.subplots()
plt.imshow(to_pass)
ax.set_title('Prediction: '+ ypred[0])
plt.show()
```

Let's do the same exercise for all digits:

In [ ]:
```
fig, ax = plt.subplots(1, 10, figsize = (15,10))
for x in range(10):
    final_size = 32

    im_re = (skimage.transform.rescale(characters[x],1/(characters[x].sh
ape[0]/24),
                                       preserve_range=True, order = 1, an
ti_aliasing=False)>0.1).astype(np.uint8)
    empty = np.zeros((32,32))
    empty[int((32-im_re.shape[0])/2):int((32-im_re.shape[0])/2)+im_re.sh
ape[0],
          int((32-im_re.shape[1])/2):int((32-im_re.shape[1])/2)+im_re.sha
pe[1]] = im_re
    to_pass = empty<0.5

    to_pass = (1*to_pass).astype(np.uint8)
    ypred = model.predict(np.reshape(to_pass,32**2)[np.newaxis,:])
    ax[x].imshow(to_pass)
    ax[x].set_title(ypred[0])
plt.show()
```

14.4 With all characters

In [ ]:
```
num_samples = 100

all_data = []
for ind, s in enumerate(samples[0:62]):
    files = glob.glob(s+'/*.png')

    data = np.array([np.reshape((skimage.transform.rescale(skimage.io.im
read(files[x]),1/4)>0.1).astype(np.uint8),32**2)
        for x in np.random.choice(np.arange(len(files)),num_samples,repl
ace=False)])

    all_data.append(data)
data = np.concatenate(all_data,axis = 0)

chars = 'abcdefghijklmnopqrstuvwxyz'
cats = [str(i) for i in range(10)]+[i for i in chars.upper()]+[i for i i
n chars]
category = np.concatenate([[cats[i] for j in range(num_samples)] for i i
n range(len(cats))])

Xtrain, Xtest, ytrain, ytest = train_test_split(data, category, random_s
tate=0)

model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
```

In [ ]:
```
ypred = model.predict(Xtest)
print(metrics.classification_report(ypred, ytest))
```

In [ ]:
```
mat = confusion_matrix(ytest, ypred)
fig, ax = plt.subplots(figsize=(10,10))
plt.imshow(mat.T,vmin = 0,vmax = 10)#, square=True, annot=True, fmt='d',
cbar=False)
plt.xticks(ticks=np.arange(62),labels=cats)
plt.yticks(ticks=np.arange(62),labels=cats)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

15. Deep learning

Deep learning methods are used more and more fequently for complex segmentation tasks. The basic idea of that approach is to let a system learn by itself what are the important features of the objects to segment by feeding it training examples.

Of course you will not learn all the details about deep learning in this single notebook. The goal here is simply to give a very brief overview of the steps involved. In particular the goal is to show that if you are provided with a trained network *e.g.* by a collaborator, using it to segment your data is very straightforward.

The example here uses Tensorflow and Keras. Tensorflow is Google's deep learning library that is widely used. Keras is a layer that sits on top of tools like Tensorflow and allows one to simplify the prototyping of a deep learning pipeline. It can also transparently be used with other "backends" like PyTorch, Facebook's deep learning library.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from skimage.external.tifffile import TiffFile
from skimage.measure import label, regionprops
from skimage.segmentation import watershed

#import your function
import sys, os
from course_functions import detect_nuclei

if not os.path.isdir('MyData/DL'):
    os.makedirs('MyData/DL')
```

15.1 Creating the training set

As a simple example, we are going to use the Zebra fish embryo *nuclei* that we have tried to segment before. Usually, one would create a training set by manually segmenting data or at least manually correcting them. Here we cheat and use our previous segmentation pipeline to create a learning dataset.

First we have to decide how large our training images are going to be. This is set by the type of computing resource used and the memory size.

In [3]:
```
imsize = 64
image_rows = 64
image_cols = 64
channels = 1
```

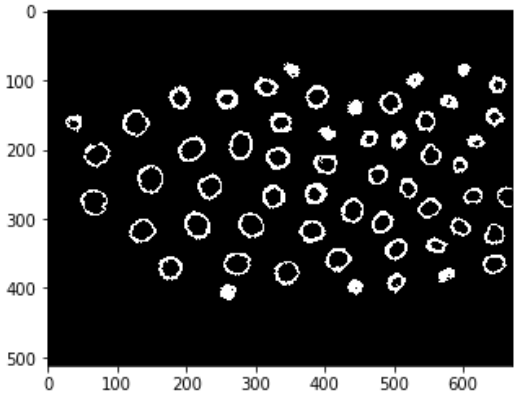
In [4]:
```
#load the image to process
data = TiffFile('Data/30567/30567.tif')
image = data.pages[0].asarray()
per_image = np.floor(np.array(image.shape)/imsize)
```

To create our training set, we are going to segment 5 images using our previous pipeline. Then we are going to cut the original image and its mask into 64x64 pieces. We exclude images which have no nuclei as they don't contain interesting information.

In [5]:
```
all_images = []
all_masks = []
for t in (3,13,23,33,43):
    image = data.pages[t].asarray()
    im_float = image.astype(np.float32)
    #create your mask
    nuclei = detect_nuclei(image)
    nuclei = nuclei.astype(np.uint8)

    for i in range(int(per_image[0])):
        for j in range(int(per_image[1])):
            if np.sum(nuclei[i*imsize:(i+1)*imsize,j*imsize:(j+1)*imsiz
e])>1:
                all_images.append(im_float[i*imsize:(i+1)*imsize,j*imsiz
e:(j+1)*imsize])
                all_masks.append(nuclei[i*imsize:(i+1)*imsize,j*imsiz
e:(j+1)*imsize])

plt.imshow(nuclei, cmap = 'gray')
```

```
/usr/local/lib/python3.5/dist-packages/skimage/filters/rank/generic.py:10
2: UserWarning: Bitdepth of 14 may result in bad rank filter performance
due to large number of bins.
  "performance due to large number of bins." % bitdepth)
```

Out[5]: <matplotlib.image.AxesImage at 0x7fbdbc1573c8>



Here we could split our dataset into a training and testing set. We have enough other data so we use all examples for training.

In [6]:
```
num_images = 5
total = len(all_masks)

num_train = int(0.99*total)
num_test = total-num_train
print(total)
print(num_train)
print(num_test)
```

```
283
280
3
```

Now we create empty arrays that are going to contain all our data. Note that this works only if the data are not too large or you have a computer with **a lot of RAM**. The alternative is to use a more complex approach using Python generators, which are going to *serve* images sequentially.

In [7]:
```
imgs = np.ndarray((num_train, image_rows, image_cols,channels), dtype=n
p.float64)
imgs_mask = np.ndarray((num_train, image_rows, image_cols), dtype=np.uin
t8)
imgs_test = np.ndarray((num_test, image_rows, image_cols,channels), dtyp
e=np.float64)
imgs_id = np.ndarray((num_test, ), dtype=np.int32)
imgs_weight = np.ndarray((num_train, image_rows, image_cols), dtype=np.u
int8)
imgs_weight[:]=1
```

Now we fill up our containers. Note that they have to be in special shapes to be fed correctly to the network. Also, in addition to our images and masks, we have so-called weights. This is an image that is going to assign more importance to certain regions. This is important for example if one category of pixels appears much less than another, like in our case nuclei vs. background.

Note also that we correct all images by normalizing them to avoid extreme values.

In [8]:
```
for counter in range(total):
    if counter<num_train:
        imgs[counter] = all_images[counter][..., np.newaxis]
        imgs_mask[counter] = all_masks[counter]
        imgs_weight[counter] = 10*all_masks[counter]+1
    else:
        imgs_test[counter-num_train] = all_images[counter][..., np.newax
is]
        imgs_id[counter-num_train] = counter-num_train

mean_val = np.mean(imgs)
imgs = imgs - mean_val
std_val = np.std(imgs)
imgs = imgs/std_val

np.save('MyData/DL/'+'imgs_train.npy', imgs)
np.save('MyData/DL/'+'imgs_mask_train.npy', imgs_mask.reshape((num_trai
n,image_rows*image_cols)))
np.save('MyData/DL/'+'imgs_test.npy', imgs_test)
np.save('MyData/DL/'+'imgs_id_test.npy', imgs_id)
np.save('MyData/DL/'+'imgs_weight_train.npy', imgs_weight.reshape((num_t
rain,image_rows*image_cols)))
```

15.2 Training the network

Now we can import our small deep learning module.

In [9]:
```
import deeplearning
```

```
Using TensorFlow backend.
```

And we can run the training of our network.

In [ ]:
```
image_rows = 64
image_cols = 64

deeplearning.nuclei_train('MyData/DL/', image_rows,image_cols, dims=1, b
atch_size = 10, epochs = 100, weights = None)
```

```
WARNING: Logging before flag parsing goes to stderr.
W0123 11:17:27.983967 140454236452608 deprecation_wrapper.py:119] From /u
sr/local/lib/python3.5/dist-packages/keras/backend/tensorflow_backend.py:
4070: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d
instead.

W0123 11:17:29.209715 140454236452608 deprecation.py:323] From /usr/local
/lib/python3.5/dist-packages/tensorflow/python/ops/math_grad.py:1250: add
_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops)
is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
W0123 11:17:33.899000 140454236452608 deprecation_wrapper.py:119] From /u
sr/local/lib/python3.5/dist-packages/keras/backend/tensorflow_backend.py:
422: The name tf.global_variables is deprecated. Please use tf.compat.v1.
global_variables instead.


Train on 224 samples, validate on 56 samples
Epoch 1/100
224/224 [==============================] - 37s 164ms/step - loss: 0.8377
- dice_coef: 0.2907 - val_loss: 0.4347 - val_dice_coef: 0.4183
Epoch 2/100
224/224 [==============================] - 27s 118ms/step - loss: 0.2871
- dice_coef: 0.6174 - val_loss: 0.1904 - val_dice_coef: 0.7066
Epoch 3/100
224/224 [==============================] - 30s 134ms/step - loss: 0.1857
- dice_coef: 0.7411 - val_loss: 0.1603 - val_dice_coef: 0.7344
Epoch 4/100
224/224 [==============================] - 28s 127ms/step - loss: 0.1449
- dice_coef: 0.7921 - val_loss: 0.1272 - val_dice_coef: 0.8245
Epoch 5/100
 40/224 [====>.........................] - ETA: 27s - loss: 0.1383 - dice
_coef: 0.8084
```

15.3 Using the trained network

Let's load an image that we did **not** use for training and select a 512x512 region.

In [ ]:
```
image = data.pages[143].asarray()[0:512,0:512]
im_float = image.astype(float)
```

Now we load again the network and say what the input size will be. Then **most importantly**, we use the weights that we just trained.

In [ ]:
```
model = deeplearning.get_unet(1,512,512)
model.load_weights('MyData/DL/weights.h5')
```

We correct now this single picture **with the same factors** used for the training set, so that it is in the same *state*.

In [ ]:
```
imgs_test = im_float.astype('float32')
imgs_test = imgs_test
imgs_test = imgs_test - mean_val
imgs_test = imgs_test/std_val
plt.imshow(imgs_test)
plt.show()
```

Finally we reshape it to fit into the network and use the predict() function to generate a prediction for each pixel to be foreground or background.

In [ ]:
```
imgs_test = imgs_test[np.newaxis,...,np.newaxis]
imgs_mask_test = model.predict(imgs_test, verbose=1)
imgs_mask_test = np.reshape(imgs_mask_test,imgs_test.shape)
```

Finally we can plot the resulting image, which has values from 0 to 1.

In [ ]:
```
plt.imshow(imgs_mask_test[0,:,:,0], vmin = 0, vmax = 1, cmap= 'gray')
plt.show()
```

We can now set a threshold for what should be considerd foreground to generate a mask, and compare to the previous segmentation.

In [ ]:
```
nuclei = detect_nuclei(image)

plt.figure(figsize=(10,10))
plt.subplot(1,2,1)
plt.imshow(imgs_mask_test[0,:,:,0]>0.9, cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(nuclei[0:512,0:512], cmap = 'gray')
plt.show()
```

16. Image classification using deep learning

In the previous notebooks, we have mostly focused on the segmentation task, i.e isolating structures in images. Another major image processing task is instead to classify entire images. For example when screening for skin caner, one is not necessarily in segmenting a tumor but rather saying whether a tumor is absent or present in an image.

Deep learning methods have been shown in the past years to be very efficient in this exercise, and many different networks have been designed. A lot of models can be found online, for example on Github. In addition, Keras, a very popular high-level package for machine learning, offers ready-to-use implementations of many popular networks. Those networks have already been trained on specific datasets, but of course one can re-train them to solve other classification tasks. Here we are going to see how to use these Keras implementations.

16.1 Importing the model

It is straightforward to import the needed model. Documentations can be found here (https://keras.io/applications/). Here we are using the VGG16 model (https://arxiv.org/abs/1409.1556) that has been trained on the ImageNet dataset, which classifies objects in 1000 categories.

In [1]:
```
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions

#from keras.applications.xception import Xception
#from keras.applications.xception import preprocess_input
#from keras.applications.xception import decode_predictions

import numpy as np
import skimage
import skimage.io
import skimage.transform
import matplotlib.pyplot as plt
```

```
Using TensorFlow backend.
```

Now we load the model, specifying the weights to be used. Those weights define all the filters that are used in the convolution steps as well as the actual weights that combine information from the output of different filters.

In [2]:
```
model = VGG16(weights='imagenet', include_top=True)
#model = Xception(weights='imagenet', include_top=True)
```

```
WARNING: Logging before flag parsing goes to stderr.
W0123 11:15:52.456051 140581987952384 deprecation_wrapper.py:119] From /u
sr/local/lib/python3.5/dist-packages/keras/backend/tensorflow_backend.py:
4070: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d
instead.


Downloading data from https://github.com/fchollet/deep-learning-models/re
leases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 91s 0us/step
```

We can have a look at the structure of the network:

In [3]:
```
model.summary()
```

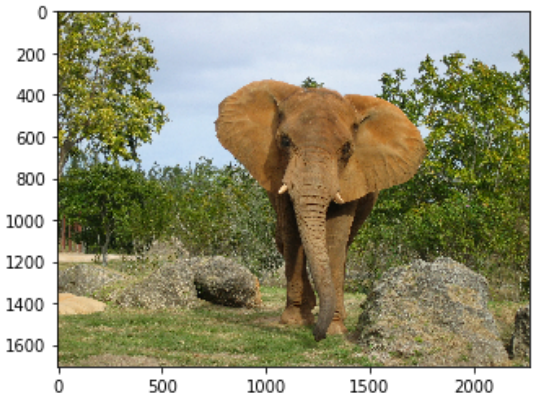
```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

16.2 Choosing and adjusting an image

Let's test the network on a simple image of an elephant:

In [4]:
```
image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commons/1/19/Afrikanische_Elefant%2C_Miami2.jpg')
```

In [5]:
```
plt.imshow(image)
plt.show()
```



Models are always expecting images of a certain size, and with intensities around a given values. This is taken care of here:

In [6]:
```
#adjust image size and dimensions
image_resize = skimage.transform.resize(image,(224,224),preserve_range=True)
x = np.expand_dims(image_resize, axis=0)

#adjust image intensities
x = preprocess_input(x)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: UserWarning: The default mode, 'constant', will be changed to 'reflect' in skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: UserWarning: Anti-aliasing will be enabled by default in skimage 0.15 to avoid aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

16.3 Prediction

Finally, we can pass that modified image to the network to give a prediction:

In [7]:
```
features = model.predict(x)
```

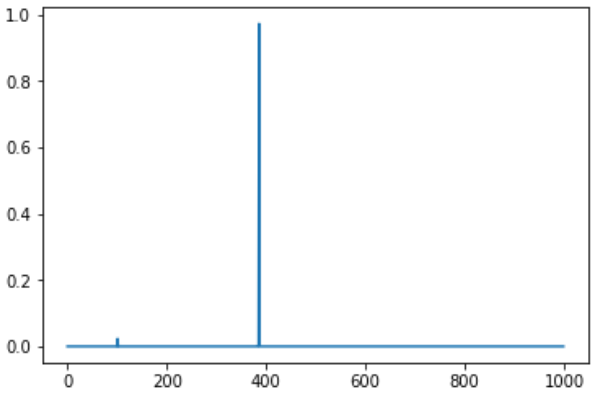
```
W0123 11:17:29.866466 140581987952384 deprecation_wrapper.py:119] From /usr/local/lib/python3.5/dist-packages/keras/backend/tensorflow_backend.py:422: The name tf.global_variables is deprecated. Please use tf.compat.v1.global_variables instead.
```

When we look at the dimensions of the output, we see that we have a vector of 1000 dimensions. Each dimensions corresponds to a category and the value represents the probability that the image contains that category. If we plot the vector we see that the image clearly belong to one category:

In [8]:
```
features.shape
```

Out[8]: (1, 1000)

In [9]:
```
plt.plot(features.T)
plt.show()
```



We can use the decond function, to know what this category index corresponds to:

In [10]:
```
decode_predictions(features, top=1000)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.
org/data/imagenet_class_index.json
40960/35363 [==================================] - 0s 1us/step
```

```
Out[10]: [[('n02504458', 'African_elephant', 0.97247916),
           ('n01871265', 'tusker', 0.02319269),
           ('n02504013', 'Indian_elephant', 0.004200729),
           ('n02437312', 'Arabian_camel', 9.9511075e-05),
           ('n02100583', 'vizsla', 6.808777e-06),
           ('n02099849', 'Chesapeake_Bay_retriever', 2.5692e-06),
           ('n03124170', 'cowboy_hat', 1.1540138e-06),
           ('n01704323', 'triceratops', 1.0709498e-06),
           ('n02389026', 'sorrel', 1.0571564e-06),
           ('n02422106', 'hartebeest', 8.660311e-07),
           ('n02096051', 'Airedale', 7.121821e-07),
           ('n02090379', 'redbone', 6.9493774e-07),
           ('n02087394', 'Rhodesian_ridgeback', 6.583336e-07),
           ('n04604644', 'worm_fence', 5.9125585e-07),
           ('n02092339', 'Weimaraner', 5.736652e-07),
           ('n03124043', 'cowboy_boot', 5.6223433e-07),
           ('n01688243', 'frilled_lizard', 4.8969315e-07),
           ('n03697007', 'lumbermill', 3.873958e-07),
           ('n03404251', 'fur_coat', 3.7333308e-07),
           ('n04350905', 'suit', 3.6049698e-07),
           ('n04259630', 'sombrero', 3.3618875e-07),
           ('n04399382', 'teddy', 3.2949515e-07),
           ('n07734744', 'mushroom', 3.0491432e-07),
           ('n07754684', 'jackfruit', 2.5534945e-07),
           ('n02408429', 'water_buffalo', 2.43335e-07),
           ('n04458633', 'totem_pole', 2.3871908e-07),
           ('n04597913', 'wooden_spoon', 2.3287092e-07),
           ('n11879895', 'rapeseed', 2.2912964e-07),
           ('n02963159', 'cardigan', 2.2274801e-07),
           ('n07802026', 'hay', 1.8962464e-07),
           ('n02088466', 'bloodhound', 1.8776879e-07),
           ('n02129165', 'lion', 1.8023877e-07),
           ('n02410509', 'bison', 1.5872558e-07),
           ('n02403003', 'ox', 1.5586099e-07),
           ('n02454379', 'armadillo', 1.5301437e-07),
           ('n03498962', 'hatchet', 1.4770363e-07),
           ('n04208210', 'shovel', 1.4289928e-07),
           ('n01518878', 'ostrich', 1.2654296e-07),
           ('n02412080', 'ram', 1.2329042e-07),
           ('n02109047', 'Great_Dane', 1.1550219e-07),
           ('n04417672', 'thatch', 1.0752834e-07),
           ('n03134739', 'croquet_ball', 1.0582936e-07),
           ('n03000684', 'chain_saw', 1.0351832e-07),
           ('n02906734', 'broom', 9.7853764e-08),
           ('n04099969', 'rocking_chair', 9.2880164e-08),
           ('n04562935', 'water_tower', 9.1811906e-08),
           ('n02489166', 'proboscis_monkey', 9.11181e-08),
           ('n02793495', 'barn', 8.838817e-08),
           ('n04371430', 'swimming_trunks', 8.648289e-08),
           ('n02113799', 'standard_poodle', 8.531001e-08),
           ('n04599235', 'wool', 8.296619e-08),
           ('n02843684', 'birdhouse', 8.085067e-08),
           ('n03776460', 'mobile_home', 7.9733795e-08),
           ('n02012849', 'crane', 7.9576395e-08),
           ('n02099429', 'curly-coated_retriever', 7.6460026e-08),
           ('n02397096', 'warthog', 7.4261834e-08),
           ('n01677366', 'common_iguana', 7.244132e-08),
           ('n02391049', 'zebra', 6.915432e-08),
           ('n02095570', 'Lakeland_terrier', 6.5524716e-08),
           ('n02093991', 'Irish_terrier', 6.52822e-08),
           ('n03743016', 'megalith', 6.347313e-08),
           ('n04532670', 'viaduct', 6.3057904e-08),
           ('n02422699', 'impala', 5.915353e-08),
           ('n02093647', 'Bedlington_terrier', 5.7262092e-08),
           ('n02099601', 'golden_retriever', 5.716737e-08),
           ('n03803284', 'muzzle', 5.6893036e-08),
           ('n03873416', 'paddle', 5.4728215e-08),
           ('n01695060', 'Komodo_dragon', 5.4479095e-08),
```

The three best categories are three categories of different elephants, but the best one is indeed the African one.

16.4 Image with multiple content

What happens if multiple objects are in an image like here a dog and a cat or a banana and strawberries?

In [11]:
```
#image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commons/0/07/Chien-lit_%26_Chat-en-lit.jpg')
image = skimage.io.imread('https://live.staticflickr.com/3652/3295428010_9284075e7b_b.jpg')
```

In [12]:
```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.show()
```

We preprocess the image and do the prediction:

In [13]:
```
model = VGG16(weights='imagenet', include_top=True)
#model = Xception(weights='imagenet', include_top=True)

image_resize = skimage.transform.resize(image,(224,224),preserve_range=True)
x = np.expand_dims(image_resize, axis=0)
x = preprocess_input(x)

features = model.predict(x)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: UserWarning: The default mode, 'constant', will be changed to 'reflect' in skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: UserWarning: Anti-aliasing will be enabled by default in skimage 0.15 to avoid aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

In [14]:
```
decode_predictions(features, top=1000)
```

```
Out[14]: [[('n07753592', 'banana', 0.5761249),
           ('n07745940', 'strawberry', 0.19425765),
           ('n07753275', 'pineapple', 0.05217132),
           ('n07614500', 'ice_cream', 0.030278875),
           ('n07749582', 'lemon', 0.015831826),
           ('n07760859', 'custard_apple', 0.012895143),
           ('n07753113', 'fig', 0.011580526),
           ('n07747607', 'orange', 0.009989414),
           ('n04476259', 'tray', 0.009962709),
           ('n07579787', 'plate', 0.0068863747),
           ('n07718472', 'cucumber', 0.006387197),
           ('n04332243', 'strainer', 0.0054645333),
           ('n07768694', 'pomegranate', 0.0054520713),
           ('n07836838', 'chocolate_sauce', 0.003862604),
           ('n07742313', 'Granny_Smith', 0.0028679618),
           ('n04597913', 'wooden_spoon', 0.0027667894),
           ('n03461385', 'grocery_store', 0.0026932321),
           ('n07716358', 'zucchini', 0.0026920456),
           ('n07613480', 'trifle', 0.0022338615),
           ('n07583066', 'guacamole', 0.00217574),
           ('n03089624', 'confectionery', 0.0013857629),
           ('n07932039', 'eggnog', 0.0013704551),
           ('n04204238', 'shopping_basket', 0.0013690287),
           ('n07717556', 'butternut_squash', 0.0013453267),
           ('n07718747', 'artichoke', 0.0012477095),
           ('n07930864', 'cup', 0.0011815256),
           ('n02909870', 'bucket', 0.0010817345),
           ('n03775546', 'mixing_bowl', 0.0010295234),
           ('n03944341', 'pinwheel', 0.0009666784),
           ('n03127925', 'crate', 0.0009592887),
           ('n07714990', 'broccoli', 0.00087834854),
           ('n03729826', 'matchstick', 0.0007504259),
           ('n02776631', 'bakery', 0.00071163604),
           ('n02971356', 'carton', 0.0006731103),
           ('n03482405', 'hamper', 0.00065947045),
           ('n07720875', 'bell_pepper', 0.00064582424),
           ('n03633091', 'ladle', 0.0006443229),
           ('n07892512', 'red_wine', 0.00063594204),
           ('n03445777', 'golf_ball', 0.00061149464),
           ('n03786901', 'mortar', 0.0006095598),
           ('n03908618', 'pencil_box', 0.00054616673),
           ('n03720891', 'maraca', 0.00052341406),
           ('n04399382', 'teddy', 0.0005185749),
           ('n12620546', 'hip', 0.00051782426),
           ('n07715103', 'cauliflower', 0.00050635735),
           ('n07871810', 'meat_loaf', 0.00050255464),
           ('n03047690', 'clog', 0.0004752425),
           ('n07693725', 'bagel', 0.0004202755),
           ('n07716906', 'spaghetti_squash', 0.000415875),
           ('n01945685', 'slug', 0.00041408345),
           ('n01734418', 'king_snake', 0.0004139101),
           ('n04270147', 'spatula', 0.0004090329),
           ('n03950228', 'pitcher', 0.00040275132),
           ('n07717410', 'acorn_squash', 0.00039611929),
           ('n02110341', 'dalmatian', 0.00039234685),
           ('n04263257', 'soup_bowl', 0.00039161675),
           ('n04259630', 'sombrero', 0.00038066693),
           ('n03991062', 'pot', 0.00036915473),
           ('n04133789', 'sandal', 0.00030882948),
           ('n07880968', 'burrito', 0.00030034475),
           ('n04141975', 'scale', 0.00029977187),
           ('n07734744', 'mushroom', 0.0002680286),
           ('n03133878', 'Crock_Pot', 0.00026488805),
           ('n04026417', 'purse', 0.0002394798),
           ('n03041632', 'cleaver', 0.00022614613),
           ('n03063599', 'coffee_mug', 0.00022026774),
           ('n02526121', 'eel', 0.00021577944),
           ('n04317175', 'stethoscope', 0.00021190982),
```

We end up with probabilitis split among multiple categories of a certain "style" like multiple dog breeds. One way to try improving on this, is to use this classifier to do an approximative segmentation by splitting the image into subregions.

We create overlapping patches and do the prediction on those:

In [15]:
```
patch = 400
step = 100
all_features =[]
for i in np.arange(0,image.shape[0]-patch-1,step):
    print(i)
    for j in np.arange(0,image.shape[1]-patch-1,step):
        subimage = image[i:i+patch,j:j+patch,:]
        image_resize = skimage.transform.resize(subimage,(224,224),prese
rve_range=True)
        x = np.expand_dims(image_resize, axis=0)
        x = preprocess_input(x)

        features = model.predict(x)
        all_features.append(features)
```

```
0
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "

100
200
```

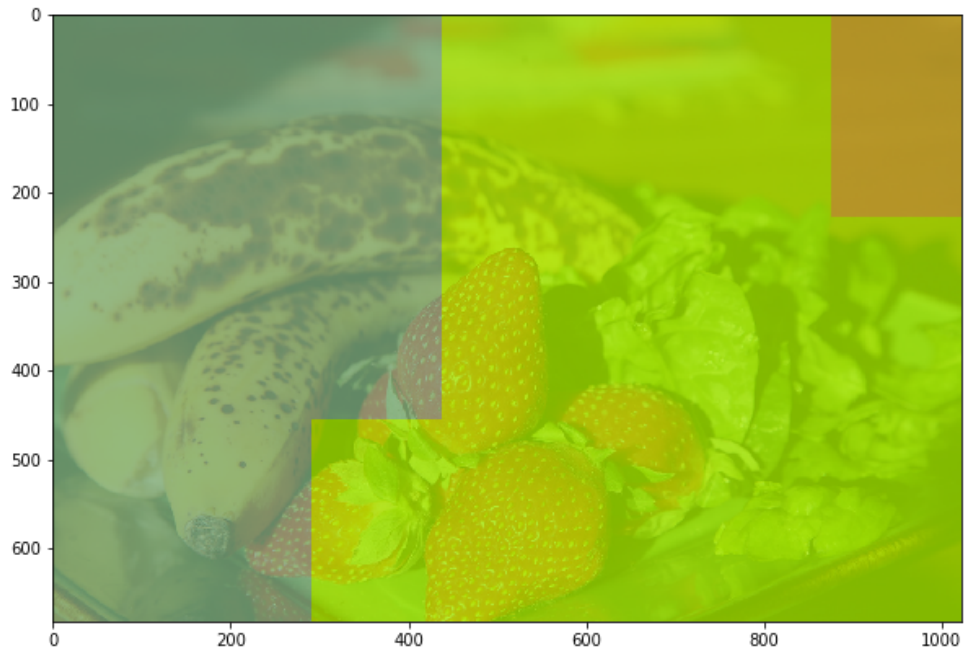
In [16]:
```
[decode_predictions(x, top=1000)[0][0] for x in all_features if decode_p
redictions(x, top=1000)[0][0][2]>0.3]
```

Out[16]:
```
[('n07753592', 'banana', 0.78769964),
 ('n07753592', 'banana', 0.47260636),
 ('n07753592', 'banana', 0.5114516),
 ('n07745940', 'strawberry', 0.68358153),
 ('n07745940', 'strawberry', 0.81029093),
 ('n07745940', 'strawberry', 0.92421764),
 ('n07753592', 'banana', 0.8903582),
 ('n07753592', 'banana', 0.7702213),
 ('n07753592', 'banana', 0.8909377),
 ('n07745940', 'strawberry', 0.9785351),
 ('n07745940', 'strawberry', 0.9900946),
 ('n07745940', 'strawberry', 0.98152024),
 ('n07745940', 'strawberry', 0.7820029),
 ('n07753592', 'banana', 0.98062783),
 ('n07753592', 'banana', 0.80547625),
 ('n07745940', 'strawberry', 0.69026893),
 ('n07745940', 'strawberry', 0.99909794),
 ('n07745940', 'strawberry', 0.99729496),
 ('n07745940', 'strawberry', 0.9918213),
 ('n07745940', 'strawberry', 0.4741534)]
```

We can now superpose those segmented features over the original image:

In [17]:
```
import matplotlib.colors
cmap = matplotlib.colors.ListedColormap ( np.random.rand ( 256,3))

reshaped = np.reshape([np.argmax(x) for x in all_features],
                      (len(np.arange(0,image.shape[0]-patch-1,step)),len
(np.arange(0,image.shape[1]-patch-1,step))))

plt.figure(figsize=(10,10))
plt.imshow(image)
plt.imshow(skimage.transform.resize(reshaped,(image.shape[0],image.shape
[1]), order=0,preserve_range=True),cmap = cmap,alpha = 0.8)
plt.show()
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```

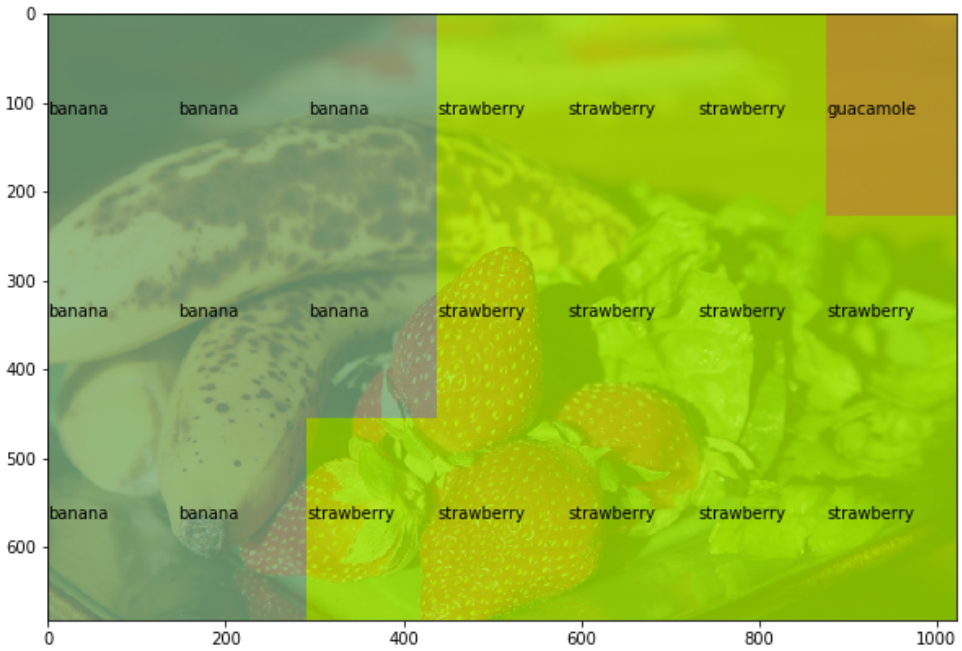


Let's create an array with the index names and plot them on top of the image:

In [18]:
```
names = np.reshape([decode_predictions(x, top=1000)[0][0][1] for x in al
l_features],
          (len(np.arange(0,image.shape[0]-patch-1,step)),len(np.arange
(0,image.shape[1]-patch-1,step))))
```

In [19]:
```
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.imshow(skimage.transform.resize(reshaped,(image.shape[0],image.shape
[1]), order=0,preserve_range=True),cmap = cmap,alpha = 0.8)
fact = image.shape[0]/reshaped.shape[0]
for x in range(names.shape[0]):
    for y in range(names.shape[1]):
        plt.text(x=(y)*image.shape[1]/reshaped.shape[1],y=(x+0.5)*image.
shape[0]/reshaped.shape[0],s = names[x,y])
plt.show()
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
```



17. Semantic segmentation: Github resources

Whenever one desires to try out some advanced technique not yet available as a nicely packaged tool like scikit-image, the best solution is to first search for open-source code that approximates what one wants to do. One of the main repositories of such code is Github (https://github.com/). As an examples, we will here do semantic segmentation, i.e. segmenting objects in an image.

In [1]:
```
import sys
import numpy as np
import skimage
import skimage.io
import skimage.transform
from matplotlib import pyplot as plt
```

17.1 Finding and exploring a repository

Let's have a look at this repository (https://github.com/bonlime/keras-deeplab-v3-plus).

17.2 Installing

We follow the instructions as given. We first check what version of tensorflow we have:

In [2]:
```
import tensorflow
```

In [3]:
```
tensorflow.__version__
```

Out[3]:
```
'1.14.0'
```

So we have to follow the second set of instructions. These are unix type commands that we would normally type in a terminal. As Jupyter support bash commands we can also do it right here:

In [4]:
```
%%bash
git clone https://github.com/bonlime/keras-deeplab-v3-plus/
cd keras-deeplab-v3-plus/
git checkout 714a6b7d1a069a07547c5c08282f1a706db92e20
```

```
fatal: destination path 'keras-deeplab-v3-plus' already exists and is not
an empty directory.
HEAD is now at 714a6b7... Merge branch 'master' of https://github.com/bon
lime/keras-deeplab-v3-plus
```

17.3 Making the package accessible

Since we only want to try out the package, we will simply add it's path to our current path. If we try multiple packages, this avoid over-crowding the conda environement with useless code. If we want to use it "in production" we can always install it later.

In [5]:
```
sys.path.append('keras-deeplab-v3-plus')
```

Now we can finally import the package:

In [6]:
```
from model import Deeplabv3
```

```
Using TensorFlow backend.
```

17.4 Using the network

We simply follow the instructions given in the repository to run the code. We only modify the image importation as we use a different package (skimage). As always there are some parameters set for pre-processing:

In [7]:
```
trained_image_width=512
mean_subtraction_value=127.5
```

Then we can pick the image of our choice:

In [8]:
```
image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/common
s/thumb/0/0c/Cow_female_black_white.jpg/1920px-Cow_female_black_white.jp
g')
#image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commo
ns/3/33/Chat-affut.JPG')
#image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commo
ns/1/18/TrailKitty.jpg')
image = image.astype('float')
```

And run the remaining of the proposed code:

In [9]:
```
# resize to max dimension of images from training dataset
w, h, _ = image.shape
ratio = float(trained_image_width) / np.max([w, h])
resized_image = skimage.transform.resize(image,(int(ratio * w),int(ratio
* h)))
#resized_image = np.array(Image.fromarray(image.astype('uint8')).resize
((int(ratio * h), int(ratio * w))))

# apply normalization for trained dataset images
resized_image = (resized_image / mean_subtraction_value) - 1.

# pad array to square image to match training images
pad_x = int(trained_image_width - resized_image.shape[0])
pad_y = int(trained_image_width - resized_image.shape[1])
resized_image = np.pad(resized_image, ((0, pad_x), (0, pad_y), (0, 0)),
mode='constant')

# make prediction
deeplab_model = Deeplabv3()
res = deeplab_model.predict(np.expand_dims(resized_image,0))
labels = np.argmax(res.squeeze(), -1)
```

```
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:105: U
serWarning: The default mode, 'constant', will be changed to 'reflect' in
skimage 0.15.
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/usr/local/lib/python3.5/dist-packages/skimage/transform/_warps.py:110: U
serWarning: Anti-aliasing will be enabled by default in skimage 0.15 to a
void aliasing artifacts when down-sampling images.
  warn("Anti-aliasing will be enabled by default in skimage 0.15 to "
WARNING: Logging before flag parsing goes to stderr.
W0123 11:16:21.968451 139872335447808 deprecation_wrapper.py:119] From /u
sr/local/lib/python3.5/dist-packages/keras/backend/tensorflow_backend.py:
4074: The name tf.nn.avg_pool is deprecated. Please use tf.nn.avg_pool2d
instead.


---------------------------------------------------------------------------
--
AttributeError                            Traceback (most recent call las
t)
<ipython-input-9-a13c11e8142d> in <module>()
     14
     15 # make prediction
---> 16 deeplab_model = Deeplabv3()
     17 res = deeplab_model.predict(np.expand_dims(resized_image,0))
     18 labels = np.argmax(res.squeeze(), -1)

~/Documents/CAS_data_science/CAS_21.01.2020_Python_Image_Processing/PyIma
geCourse-master/keras-deeplab-v3-plus/model.py in Deeplabv3(weights, inpu
t_tensor, input_shape, classes, backbone, OS, alpha)
    441     b4 = BatchNormalization(name='image_pooling_BN', epsilon=1e-
5)(b4)
    442     b4 = Activation('relu')(b4)
--> 443     b4 = BilinearUpsampling((int(np.ceil(input_shape[0] / OS)), i
nt(np.ceil(input_shape[1] / OS))))(b4)
    444
    445     # simple 1x1

/usr/local/lib/python3.5/dist-packages/keras/engine/base_layer.py in __ca
ll__(self, inputs, **kwargs)
    487             # Actually call the layer,
    488             # collecting output(s), mask(s), and shape(s).
--> 489             output = self.call(inputs, **kwargs)
    490             output_mask = self.compute_mask(inputs, previous_mas
k)
    491

~/Documents/CAS_data_science/CAS_21.01.2020_Python_Image_Processing/PyIma
geCourse-master/keras-deeplab-v3-plus/model.py in call(self, inputs)
     91     def call(self, inputs):
     92         if self.upsampling:
---> 93             return K.tf.image.resize_bilinear(inputs, (inputs.sha
pe[1] * self.upsampling[0],
     94                                                        inputs.sha
pe[2] * self.upsampling[1]),
     95                                                align_corners=True)

AttributeError: module 'keras.backend' has no attribute 'tf'
```

Since we padded and reshaped the image in the pre-processing step, we have now to correct the size of the output labels:

In [ ]:
```
if pad_x > 0:
    labels = labels[:-pad_x,:]
if pad_y > 0:
    labels = labels[:, :-pad_y]
labels = skimage.transform.resize(labels,(w, h),preserve_range=True, ord
er=0)
```

17.5 Checking the output

In [ ]:
```
plt.imshow(labels)
plt.show()
plt.imshow(image[:,:,0])
plt.show()
```

In [ ]:
```
class_names = np.array(['background','aeroplane', 'bicycle', 'bird', 'bo
at',
                        'bottle', 'bus', 'car', 'cat', 'chair',
                        'cow', 'diningtable', 'dog', 'horse',
                        'motorbike', 'person', 'pottedplant',
                        'sheep', 'sofa', 'train', 'tvmonitor'])
```

In [ ]:
```
class_names[np.unique(labels).astype(int)]
```

18. Application: DICOM

DICOM (Digital Imaging and Communications in Medicine) is the international standard to transmit, store, retrieve, print, process, and display medical imaging information. It is in particular widely used to store volumetric data from methods such as CT, MR, Ultrasound, etc.

This kind of specific image format is typically not supported by general packages such as scikit-image. However in most cases, independent dedicated packages exist. A simple Google search leads us to the pydicom (https://pydicom.github.io/pydicom/stable/getting_started.html) package.

In [1]:
```
import os
import matplotlib.pyplot as plt
plt.gray()
import pydicom
import numpy as np
import skimage
import ipyvolume as ipv
```

We will use an MRI dataset of a head available on the data sharing platform Zenodo. In this course, most data have been made directly available. To show the full procedure, we will here include the download step.

Install the missing package:

In [2]:
```
!pip install --user pydicom
```

```
Requirement already satisfied: pydicom in /usr/local/lib/python3.5/dist-p
ackages (1.4.1)
You are using pip version 19.0.3, however version 20.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' comman
d.
```

In [3]:
```
import pydicom
```

18.1. Download

The donwload address on Zenodo is:

In [4]:
```
data_address= 'https://zenodo.org/record/16956/files/DICOM.zip?download=
1'
```

Create a folder where to put the data:

In [5]:
```
#os.makedirs('MyData')
```

We can use the `urllib` native package to proceed with download which provides us with a zip file:

In [6]:
```
import urllib

urllib.request.urlretrieve(data_address, 'MyData/mri.zip')
```

Out[6]: ('MyData/mri.zip', <http.client.HTTPMessage at 0x7fb257782400>)

To automate the process we now also automatically unzip the file using the `zipfile` module:

In [7]:
```
import zipfile
```

In [8]:
```
with zipfile.ZipFile('MyData/mri.zip', 'r') as zip_ref:
    zip_ref.extractall('MyData/mri/')
```

18.2. Importing one slice

We define the general path to the folder containing slices:

In [9]:
```
path = 'MyData/mri/DICOM/ST000000/SE000002/'
```

Now we use the `pydicom` package to import a single slice using the `dcmread()` function:

In [10]:
```
single_slice = pydicom.dcmread(path+'MR000000')
```

A DICOM file does not just contain image data but a very extensive set of metadata. You can see these metadata by just printing the variable:

In [11]:
```
single_slice;
```

All that information is also available as attributes of the variable. For example you can get the patient's name:

In [12]:
```
single_slice.PatientName
```

Out[12]: 'LIONHEART^WILLIAM'

But also numerical values such as pixel spacing or position of slice in the stack:

In [13]:
```
single_slice.PixelSpacing
```

Out[13]: [0.8984375, 0.8984375]

In [14]:
```
single_slice.SliceLocation
```

Out[14]: "0.0"

18.3. Loading the complete stack

As we have already done previously, we have first to parse the folder content to gather the files belonging to the stack. Here we simply list the folder content:

In [15]:
```
file_list = os.listdir(path)
```

In [16]:
```
#file_list
```

We can now load each slice using a comprehension list. From the file sorting, we already see that we'll later have to reorder the slices.

In [17]:
```
slices = [pydicom.dcmread(path+x) for x in os.listdir(path)]
```

In principle we could reorder the file by names but this is going to depend on file name formatting. A more general solution is to reorganize based on the location of the file in the stack. Let's recover that position:

In [18]:
```
positions = [int(x.SliceLocation) for x in slices]
```

In [19]:
```
#positions
```

We then use `np.argsort()` function to get the indices of the ordered list:

In [20]:
```
import numpy as np
index_ordered = np.argsort(positions)
```

In [21]:
```
index_ordered
```

Out[21]:
```
array([21,  2,  1, 20,  3, 11, 13,  9, 29, 28, 22, 26, 18,  5, 23, 16, 31,
       15, 12, 10,  0, 19,  6,  4, 24, 14, 17,  8, 30,  7, 27, 25])
```

And finally use that ordered list to reorder the slices themselves:

In [22]:
```
reordered = []
slices_ordered = [slices[x] for x in index_ordered]
```

18.4. Visualization

Finally we can visualize our volume. First let's create an actual volume by stacking the planes:

In [23]:
```
volume = np.stack([x.pixel_array for x in slices_ordered])
```

In [24]:
```
volume.shape
```

Out[24]:
```
(32, 256, 256)
```

For the rendering, we'll see here two different solutions. The first one is ipyvolume, a leight-weight volume viewer purely based on browser technology. The syntax is very similar to matplotlib.

In [25]:
```
#import ipyvolume as ipv
```

In [26]:
```
ipv.figure()
ipv.volshow(volume)
ipv.show()
```

```
/usr/local/lib/python3.5/dist-packages/ipyvolume/serialize.py:81: Runtime
Warning: invalid value encountered in true_divide
  gradient = gradient / np.sqrt(gradient[0]**2 + gradient[1]**2 + gradien
t[2]**2)
```

As ipyvolume is fully browser-based, it's very easy to save an image as a web page. For example we can just type:

In [27]:
```
ipv.save('interactive_view.html')
```

```
/usr/local/lib/python3.5/dist-packages/ipyvolume/serialize.py:81: Runtime
Warning: invalid value encountered in true_divide
  gradient = gradient / np.sqrt(gradient[0]**2 + gradient[1]**2 + gradien
t[2]**2)
```

And this saves for us a full interactive version of the figure above. This can therefore be very useful for demonstration purposed e.g. to insert an image on a web-page.

Note that customizing the aspect of the view requires some work and that this package is not as mature as others.

An alternative solution is to use the ITK (Insight Toolkit), a very popular image processing tool suite in medical imaging (an interesting but more challenging alternative to scikit-image). ITK in particular offers a volume viewer compatible with Python and Jupyter:

In [29]:
```
import itkwidgets as itkw
import itk
```

We can just call the `view()` function:

In [30]:
```
itkw.view(volume)
```

We see that the head looks compressed because the acquisition is anisotropic (large depth dimension that width/height). Above we simply passed a Numpy array to the viewer. However we can also create a native ITK format to adjust parameters more easily:

In [31]:
```
image_from_array = itk.image_from_array(volume)
```

This object has now several new attributes and methods such as:

In [32]:
```
image_from_array.GetSpacing()
```

Out[32]:
```
itkVectorD3 ([1, 1, 1])
```

We can try to guess and adjust the spacing:

In [33]:
```
image_from_array.SetSpacing((1,1,10))
```

Or we can use the itk package to read the native spacing:

In [34]:
```
itk_slice = itk.imread(path+'MR000001')
spacing = itk_slice.GetSpacing()
spacing
```

Out[34]: itkVectorD3 ([0.898438, 0.898438, 6])

In [35]:
```
image_from_array.SetSpacing(spacing)
```

In [36]:
```
itkw.view(image_from_array)
```

18.5. Image processing

Finally, we can do the same image processing operations as we did before, just in 3D. For example a thresholding:

In [37]:
```
import skimage.filters
```

In [38]:
```
vol_thresh = volume>200
```

In [39]:
```
itkw.view(vol_thresh.astype(np.uint8))
```