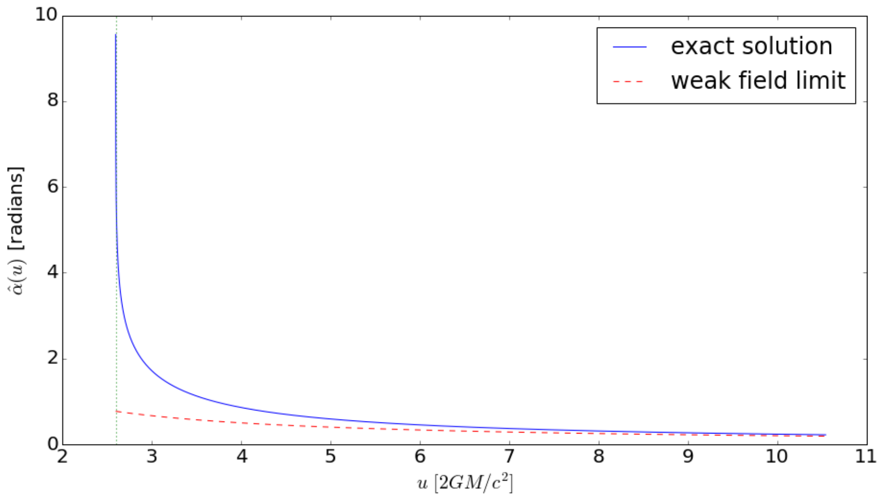

Figure 1.2.2: *Deflection angle by a compact lens as a function of the photon impact parameter. Shown are the exact solution of the geodesic equations for a Schwarzschild metric (solid line) and the solution in the weak field approximation (dashed line). The dotted vertical line shows the impact parameter, $u = 3\sqrt{3}GM/c^2$, for which the exact solution diverges, indicating that the photon keeps looping around the lens.*

1.3 Deflection by an ensemble of point masses

The deflection angle in Eq. 1.41 depends linearly on the mass M. This result was obtained by linearizing the equations of general relativity in the weak field limit. Under these circumstances, the superposition principle holds and the deflection angle of an array of lenses can be calculated as the sum of all contributions by each single lens.

Suppose we have a sparse distribution of N point masses on a plane, whose positions and masses are $\vec{\xi}_i$ and M_i, $1 \leq i \leq N$. The deflection angle of a light ray crossing the plane at $\vec{\xi}$ will be:

$$\hat{\vec{\alpha}}(\vec{\xi}) = \sum_i \hat{\vec{\alpha}}_i(\vec{\xi} - \vec{\xi}_i) = \frac{4G}{c^2} \sum_i M_i \frac{\vec{\xi} - \vec{\xi}_i}{|\vec{\xi} - \vec{\xi}_i|^2} \ . \tag{1.50}$$

Note that the formula above is similar to that we would use to compute the gravitational force between point masses on the plane. While the force depends on the inverse squared distance, the deflection angle scales as ξ^{-1}. In the case of many lenses, the computation of the deflection angle using Eq. 1.50 can become very computationally expensive, as it costs $O(N^2)$. However, as it is usually done to solve numerically N-body problems, algorithms employing meshes or hierarchies (such as the so-called tree algorithms (Barnes and Hut, 1986)) can significantly reduce the cost of calculations (e.g. to $O(N \log N)$). For some application of these algorithms in the computation of the deflection angles, we refer the reader to the works of Aubert, Amara, and Metcalf (2007) and Meneghetti et al. (2010).

1.4 Deflection by an extended mass distribution

We now consider more realistic lens models, i.e. three dimensional distributions of matter. Even in the case of lensing by galaxy clusters, the physical size of the lens is generally much smaller than the distances between observer, lens and source. The deflection therefore arises along a very short section of the light path. This justifies the usage of the *thin screen approximation*: the lens is approximated by a planar distribution of matter, the lens plane.

Within this approximation, the lensing matter distribution is fully described by its surface density,

$$\Sigma(\vec{\xi}) = \int \rho(\vec{\xi}, z)\, \mathrm{d}z\,, \tag{1.51}$$

where $\vec{\xi}$ is a two-dimensional vector on the lens plane and ρ is the three-dimensional density.

As long as the thin screen approximation holds, the total deflection angle is obtained by summing the contribution of all the mass elements $\Sigma(\vec{\xi})\mathrm{d}^2\xi$:

$$\vec{\hat{\alpha}}(\vec{\xi}) = \frac{4G}{c^2} \int \frac{(\vec{\xi} - \vec{\xi}')\Sigma(\vec{\xi}')}{|\vec{\xi} - \vec{\xi}'|^2}\, \mathrm{d}^2\xi'\,. \tag{1.52}$$

This equation shows that the calculation of the deflection angle is formally a convolution of the surface density $\Sigma(\vec{\xi})$ with the kernel function

$$\vec{K}(\vec{\xi}) \propto \frac{\vec{\xi}}{|\vec{\xi}|^2}\,. \tag{1.53}$$

This enables the calculation of the deflection angle field in the Fourier space as the product of the Fourier transforms of Σ and K:

$$\tilde{\hat{\alpha}}_i(\vec{k}) \propto \tilde{\Sigma}(\vec{k})\tilde{K}_i(\vec{k})\,, \tag{1.54}$$

where $\vec{k}$ is the conjugate variable to $\vec{\xi}$ and the tilde denotes the Fourier Transforms. The subscript $i \in [1,2]$ indicates the two components along the two axes on the lens plane (remember that $\hat{\alpha}$ is a vector!). This calculation can be implemented efficiently using the *Fast-Fourier-Transform (FFT)* algorithm (Cooley and Tukey, 1965). Note that this assumes that the integration extends to an infinite domain, while gravitational lenses have finite mass distributions. FFT algorithms implement this feature assuming periodic conditions on the boundaries of the integration domain.

1.5 Python applications

1.5.1 Deflection by a black-hole

In our first python application, we write a script to produce Fig. 1.2.2. A brief python tutorial can be found in Appendix A.

We want to implement the formula in Eq. 1.46. We also need to remind that

$$u^2 = \frac{C(R_m)}{A(R_m)}$$

We will compare the resulting deflection angle to

$$\hat{\alpha} = \frac{4GM}{c^2 u} \tag{1.55}$$

which is the result we obtained in the weak-field limit.

We start by importing some useful packages:

```
from scipy import special as sy  # need special functions for incomplete \\
# elliptic integrals of the first kind
import numpy as np # efficient vector and matrix operations
import matplotlib.pyplot as plt # a MATLAB-like plotting framework
%matplotlib inline # only needed in jupyter notebooks
```

Note that we import the module special from scipy in order to compute the elliptic integral of the first kind appearing in Eq. 1.46. See https://docs.scipy.org/doc/scipy/reference/special.html.

Our goal is to produce a graph. Let's setup the fonts and the character size

```
font = {'family' : 'normal',
        'weight' : 'normal',
        'size'   : 20}

import matplotlib
matplotlib.rc('font', **font)
```

The task can be completed in several ways. Here we chose to build a class for point black-holes:

```
class point_bh:

    def __init__(self,M):
        self.M=M

    # functions which define the metric.
    def A(self,r):
        return(1.0-2.0*self.M/r)

    def B(self,r):
        return (self.A(r)**(-1))

    def C(self,r):
        return(r**2)

    # compute u from rm
    def u(self,r):
        u=np.sqrt(self.C(r)/self.A(r))
        return(u)

    # functions concurring to the deflection angle calculation
    def ss(self,r):
        return(np.sqrt((r-2.0*self.M)*(r+6.0*self.M)))

    def mm(self,r,s):
        return((s-r+6.0*self.M)/2/s)

    def phif(self,r,s):
        return(np.arcsin(np.sqrt(2.0*s/(3.0*r-6.0*self.M+s))))
    # the deflection angle
    def defAngle(self,r):
        s=self.ss(r)
        m=self.mm(r,s)
        phi=self.phif(r,s)
        F=sy.ellipkinc(phi, m) # using the ellipkinc function
                               # from scipy.special
        return(-np.pi+4.0*np.sqrt(r/s)*F)
```

The class contains several methods which will be used to compute the deflection angle. For example, we implement the functions $A(R)$, $B(R)$, and $C(R)$. These will be used to convert the minimal distance R_m to u. We also implement the functions s, m, φ, which depend on the mass of the black-hole and on the minimal distance R_m. Finally, we implement the function `defAngle`, which enables to compute the deflection angle using Eq. 1.46. This function uses the method `elipkinc` from `scipy.special` to compute the incomplete elliptic integral of the first kind, $F(\varphi, m)$. Note that φ and m can be passed as `numpy` arrays, i.e. `elipkinc` can return values for a number of couples (φ, m).

Following the same approach, we build another class which deals with point lenses in the weak field limit, i.e. it implements Eq. 1.55:

```
class point_mass:

    def __init__(self,M):
        self.M=M

    # the classical formula
    def defAngle(self,u):
        return(4.0*self.M/u)
```

We can now use the two classes above to build two objects, namely a black-hole lens (employing the exact solution for the deflection angle) and a point mass lens, for which we will adopt the weak-field limit. In both cases, the mass of the lens is fixed to $3M_\odot$. For a mass of this size, the Schwarzchild radius is $R_s \sim 9$km:

```
bh=point_bh(3.0)
pm=point_mass(3.0)
```

We now use the `linspace` method from `numpy` to initialize an vector of minimal distances R_m, which we will use to compute $\hat{\alpha}$. We use the method `u(r)` of `point_bh` to convert R_m into an array of impact parameters u:

```
r=np.linspace(3.0/2.0,10,1000)*2.0*bh.M
u=bh.u(r)/2.0/bh.M
```

The deflection angle as a function of u or R_m can be computed in the cases of the exact solution and in the weak field limit using the method `defAngle` applied to `bh` and `pm`:

```
a=bh.defAngle(r)
b=pm.defAngle(u*2.0*bh.M)
```

Note that u is in units of the Schwarzchild radius and that we have set $G/c^2 = 1$.

Finally, we can produce a nice figure displaying the results of the calculation. We use `matplotlib.pyplot` to do this:

```
# initialize figure and axes
# (single plot, 15" by 8" in size)
fig,ax=plt.subplots(1,1,figsize=(15,8))
# plot the exact solution in ax
ax.plot(u,a,'-',label='exact solution')
# plot the solution in the weak field limit
ax.plot(u,b,'--',label='weak field limit',color='red')
# set the labels for the x and the y axes
ax.set_xlabel(r'$u$ $[2GM/c^2]$')
ax.set_ylabel(r'$\hat\alpha(u)$ [radians]')
# add the legend
ax.legend()
```

We also want to show the vertical asymptote at $u_{lim} = 3/2\sqrt{3}GM/c^2$:

```
# plot a vertical dotted line at u=3\sqrt(3)M
x=[np.min(u),np.min(u)]
y=[0,10]
ax.plot(x,y,':')
```

To conclude, we save the figure in a .png file:

```
# save figure in png format
fig.savefig('bhalpha.png')
```

1.5.2 Deflection by an extended mass distribution

In this application, we implement the calculation of the deflection angle field by an extended lens. A two-dimensional map of the lens surface-density is provided by the fits file `kappa_gl.fits` (see the data folder in the github repository). The map was obtained by projecting the mass distribution of a dark matter halo obtained from N-body simulations on a lens plane. To be precise, this is the surface density divided by a constant which depends on the lens and source redshifts (we will talk about this constant in the next lectures). Let's denote this quantity as κ. Accounting for this normalization, the calculation we want to implement is

$$\vec{\alpha}(\vec{x}) = \frac{1}{\pi}\int \kappa(\vec{x}')\frac{\vec{x}-\vec{x}'}{|\vec{x}-\vec{x}'|^2}d^2x' \ .$$

This is a convolution, which can be written in the Fourier Space as

$$\tilde{\vec{\alpha}}(\vec{k}) = 2\pi\tilde{\kappa}(\vec{k})\tilde{\vec{K}}(\vec{k})$$

where $\tilde{\vec{K}}(\vec{k})$ is the Fourier Transform of

$$\vec{K}(\vec{x}) = \frac{1}{\pi}\frac{\vec{x}}{|\vec{x}|^2}$$

We use the `numpy.fft` module:

```
import numpy as np
import numpy.fft as fftengine
```

We define a class called deflector, where the deflector object is initialized by reading the fits file containing the surface density map of the lens. To deal with the fits files, we need to use the `astropy.io.fits` module.

The class contains some methods to

- build the kernel $K(\vec{x})$;
- compute the deflection angle map by convolving the convergence with the kernel;
- perform the so-called "zero-padding";
- crop the zero-padded maps.

```
import astropy.io.fits as pyfits


class deflector(object):

    # initialize the deflector using a surface density (covergence) map
    # the boolean variable pad indicates whether zero-padding is used
    # or not
    def __init__(self,filekappa,pad=False):
        kappa,header=pyfits.getdata(filekappa,header=True)
        self.kappa=kappa
        self.nx=kappa.shape[0]
        self.ny=kappa.shape[1]
        self.pad=pad
        if (pad):
            self.kpad()
        self.kx,self.ky=self.kernel()

    # implement the kernel function K
    def kernel(self):
        x=np.linspace(-0.5,0.5,self.kappa.shape[0])
        y=np.linspace(-0.5,0.5,self.kappa.shape[1])
        kx,ky=np.meshgrid(x,y)
        norm=(kx**2+ky**2+1e-12)
        kx=kx/norm/np.pi
        ky=ky/norm/np.pi
        return(kx,ky)

    # compute the deflection angle maps by convolving
    # the surface density with the kernel function
    def angles(self):
        # FFT of the surface density and of the two components of the kernel
        density_ft = fftengine.fftn(self.kappa,axes=(0,1))
        kernelx_ft = fftengine.fftn(self.kx,axes=(0,1),
                                    s=self.kappa.shape)
        kernely_ft = fftengine.fftn(self.ky,axes=(0,1),
                                    s=self.kappa.shape)
        # perform the convolution in Fourier space and transform the result
        # back in real space. Note that a shift needs to be applied using
        # fftshift
        alphax = 2.0/(self.kappa.shape[0])/(np.pi)**2*\
                fftengine.fftshift(fftengine.ifftn(2.0*\
                np.pi*density_ft*kernelx_ft))
        alphay = 2.0/(self.kappa.shape[0])/(np.pi)**2*\
                fftengine.fftshift(fftengine.ifftn(2.0*\
                np.pi*density_ft*kernely_ft))
        return(alphax.real,alphay.real)

    # returns the surface-density (convergence) of the deflector
    def kmap(self):
        return(self.kappa)

    # performs zero-padding
    def kpad(self):
        # add zeros around the original array
        def padwithzeros(vector, pad_width, iaxis, kwargs):
            vector[:pad_width[0]] = 0
            vector[-pad_width[1]:] = 0
            return vector
        # use the pad method from numpy.lib to add zeros (padwithzeros)
        # in a frame with thickness self.kappa.shape[0]
        self.kappa=np.lib.pad(self.kappa, self.kappa.shape[0],
                              padwithzeros)

    # crop the maps to remove zero-padded areas and get back to the
    # original region.
    def mapCrop(self,mappa):
        xmin=0.5*(self.kappa.shape[0]-self.nx)
        ymin=0.5*(self.kappa.shape[1]-self.ny)
        xmax=xmin+self.nx
        ymax=ymin+self.ny
        mappa=mappa[xmin:xmax,ymin:ymax]
        return(mappa)
```

We can now build a deflector and use it to compute the deflection angles employing the method `angles`:

```
df=deflector('data/kappa_gl.fits')
angx_nopad,angy_nopad=df.angles()
kappa=df.kmap()

import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm, PowerNorm, SymLogNorm
%matplotlib inline

fig,ax = plt.subplots(1,3,figsize=(16,8))
```

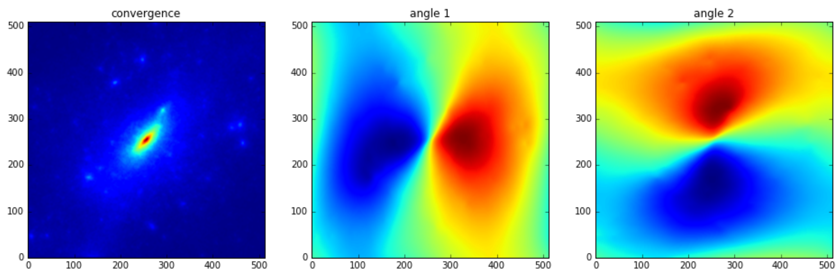

Figure 1.5.1: *Left panel: the surface density (convergence) map of the lens. Middle and right panels: maps of the two components of the deflection angles.*

```
ax[0].imshow(kappa,origin="lower")
ax[0].set_title('convergence')
ax[1].imshow(angx_nopad,origin="lower")
ax[1].set_title('angle 1')
ax[2].imshow(angy_nopad,origin="lower")
ax[2].set_title('angle 2')
```

Note that at this point we have not yet used the zero-padding trick. FFT assumes periodic boundaries conditions, meaning that the lens mass distribution is replicated outside the boundaries. Given that the region around the lens considered in this example is relatively small, we expect that the deflection angles will be biased near the borders. The three panels in Fig. 1.5.1 show the maps of the convergence and of the two components of the deflection angles obtained with this setting.

Zero-padding consists of placing zeros all around the convergence map. By doing so, we double the size of the original map, but we expect to increase the accuracy of the calculations near the borders, beacause the periodic conditions are better reproduced in this setting. We activate zero-padding by just setting the variable `pad=True` when initializing the deflector. Fig. 1.5.2 shows the zero-padded convergence map and the two new maps of the deflection angle components.

```
df=deflector('data/kappa_gl.fits',True)
angx,angy=df.angles()
kappa=df.kmap()

fig,ax = plt.subplots(1,3,figsize=(16,8))
angx,angy=df.angles()
ax[0].imshow(kappa,origin="lower")
ax[0].set_title('convergence')
ax[1].imshow(angx,origin="lower")
ax[1].set_title('angle 1')
ax[2].imshow(angy,origin="lower")
ax[2].set_title('angle 2')
```

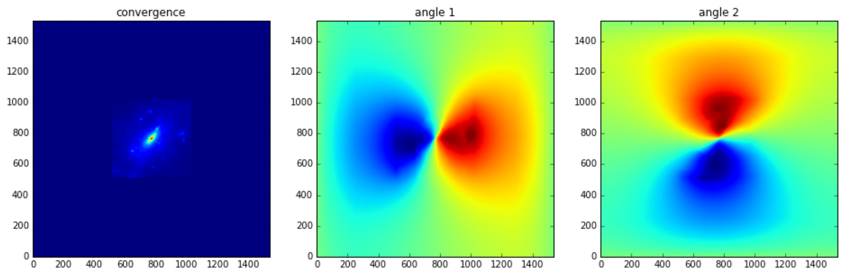

Figure 1.5.2: *The figure shows the same maps as in Fig. 1.5.1, but with zero-padding. Indeed, as shown in the left panel, the lens is surrounded by a frame of zeros, and the deflection angle maps are computed on an area which has double the size of the maps in Fig. 1.5.1.*

We are not interested in this large area, thus we can get rid of the values outside the lens convergence map by cropping the deflection angle maps. The results are shown in Fig. 1.5.3 and compared to the previous ones. In fact, significant differences are visible along the borders.

```
angx=df.mapCrop(angx)
angy=df.mapCrop(angy)

fig,ax = plt.subplots(2,2,figsize=(16,16))
ax[0,0].imshow(angx,origin="lower")
ax[0,0].set_title('angle 1')
ax[0,1].imshow(angy,origin="lower")
ax[0,1].set_title('angle 2')
ax[1,0].imshow(angx_nopad,origin="lower")
ax[1,0].set_title('angle 1 - no zero pad')
ax[1,1].imshow(angy_nopad,origin="lower")
ax[1,1].set_title('angle 2 - no zero pad')
```

1.6 Problems

Problem 1.1 — Write a python script to produce a figure displaying $\hat{\alpha}(R_m)$ with R_m in the range 9-1000 km for two lenses with mass $M = 3M_\odot$ and $M = 10M_\odot$..
Problem 1.2 — Define a class for an ensemble of point masses. The class should be initialized with two numpy arrays containing the masses and the positions of the lenses. Use the thin screen approximation and write the method to compute the deflection angle at a certain location on the lens plane..

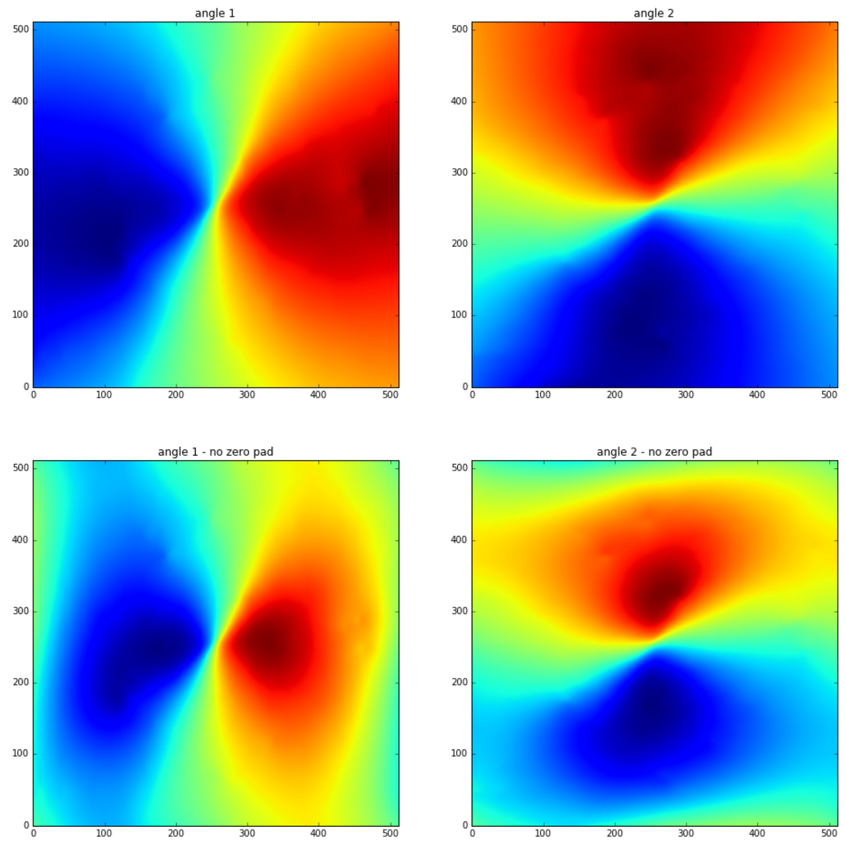

Figure 1.5.3: *The upper panels show the same two maps displayed in the middle and right panels of Fig. 1.5.2, which have been cropped to match the original size of the input convergence map. The bottom panels show the maps obtained without padding, for comparison.*