GRAVITATIONAL LENSING

6 – DERIVATION OF THE PROPERTIES OF A LENS

Massimo Meneghetti
AA 2017-2018

CALCULATION OF THE LENSING POTENTIAL (ALGORITHM)

The Laplacian of the lensing potential is twice the convergence:

$$\kappa(\theta) = \frac{1}{2}\triangle_\theta \hat{\Psi}$$

The solution of this problem (Poisson equation on the plane) can be found in Fourier space:

Laplace operator in Fourier space:

$$\tilde{\triangle}(\vec{k}) = -4\pi^2 k^2$$

Rewrite the Eq. above in Fourier space

$$-4\pi^2 k^2 \tilde{\Psi}(\vec{k}) = 2\tilde{\kappa}(\vec{k})$$

Find the Fourier transform of the potential from the Fourier transform of the convergence

$$\tilde{\Psi}(\vec{k}) = -\frac{\tilde{\kappa}(\vec{k})}{2\pi^2 k^2}$$

PYTHON IMPLEMENTATION (SEE ALSO NOTEBOOK 5_2018)

```
    def potential(self):
        # define an array of wavenumbers (two components k1,k2)
        k = np.array(np.meshgrid(fftengine.fftfreq(self.kappa.shape[0])\
                                 ,fftengine.fftfreq(self.kappa.shape[1])))
        pix=1 # pixel scale (now using pixel units)
        #Compute Laplace operator in Fourier space = -4*pi*l*l
        kk = k[0]**2 + k[1]**2
        kk[0,0] = 1.0
        #FFT of the convergence
        kappa_ft = fftengine.fftn(kappa)
        #compute the FT of the potential
        kappa_ft *= - pix**2 / (kk * (2.0*np.pi**2))
        kappa_ft[0,0] = 0.0
        potential=fftengine.ifftn(kappa_ft) #units should be rad**2
        return potential.real
```

PYTHON IMPLEMENTATION (SEE ALSO NOTEBOOK 5_2018)

Wave vectors from size of the convergence map

```
    def potential(self):
        # define an array of wavenumbers (two components k1,k2)
        k = np.array(np.meshgrid(fftengine.fftfreq(self.kappa.shape[0])\
                                 ,fftengine.fftfreq(self.kappa.shape[1])))
        pix=1 # pixel scale (now using pixel units)
        #Compute Laplace operator in Fourier space = -4*pi*l*l
        kk = k[0]**2 + k[1]**2
        kk[0,0] = 1.0
        #FFT of the convergence
        kappa_ft = fftengine.fftn(kappa)
        #compute the FT of the potential
        kappa_ft *= - pix**2 / (kk * (2.0*np.pi**2))
        kappa_ft[0,0] = 0.0
        potential=fftengine.ifftn(kappa_ft) #units should be rad**2
        return potential.real
```

PYTHON IMPLEMENTATION (SEE ALSO NOTEBOOK 5_2018)

Wave vectors from size of the convergence map

```
    def potential(self):
        # define an array of wavenumbers (two components k1,k2)
        k = np.array(np.meshgrid(fftengine.fftfreq(self.kappa.shape[0])\
                                 ,fftengine.fftfreq(self.kappa.shape[1])))
        pix=1 # pixel scale (now using pixel units)
        #Compute Laplace operator in Fourier space = -4*pi*l*l
        kk = k[0]**2 + k[1]**2
        kk[0,0] = 1.0
        #FFT of the convergence
        kappa_ft = fftengine.fftn(kappa)
        #compute the FT of the potential
        kappa_ft *= - pix**2 / (kk * (2.0*np.pi**2))
        kappa_ft[0,0] = 0.0
        potential=fftengine.ifftn(kappa_ft) #units should be rad**2
        return potential.real
```

FT of the convergence

PYTHON IMPLEMENTATION (SEE ALSO NOTEBOOK 5_2018)

Wave vectors from size of the convergence map

```
    def potential(self):
        # define an array of wavenumbers (two components k1,k2)
        k = np.array(np.meshgrid(fftengine.fftfreq(self.kappa.shape[0])\
                                 ,fftengine.fftfreq(self.kappa.shape[1])))
        pix=1 # pixel scale (now using pixel units)
        #Compute Laplace operator in Fourier space = -4*pi*l*l
        kk = k[0]**2 + k[1]**2
        kk[0,0] = 1.0
        #FFT of the convergence
        kappa_ft = fftengine.fftn(kappa)
        #compute the FT of the potential
        kappa_ft *= - pix**2 / (kk * (2.0*np.pi**2))
        kappa_ft[0,0] = 0.0
        potential=fftengine.ifftn(kappa_ft) #units should be rad**2
        return potential.real
```

FT of the convergence

FT of the Laplacian

PYTHON IMPLEMENTATION (SEE ALSO NOTEBOOK 5_2018)

Wave vectors from size of the convergence map

```
    def potential(self):
        # define an array of wavenumbers (two components k1,k2)
        k = np.array(np.meshgrid(fftengine.fftfreq(self.kappa.shape[0])\
                                 ,fftengine.fftfreq(self.kappa.shape[1])))
        pix=1 # pixel scale (now using pixel units)
        #Compute Laplace operator in Fourier space = -4*pi*l*l
        kk = k[0]**2 + k[1]**2
        kk[0,0] = 1.0
        #FFT of the convergence
        kappa_ft = fftengine.fftn(kappa)
        #compute the FT of the potential
        kappa_ft *= - pix**2 / (kk * (2.0*np.pi**2))
        kappa_ft[0,0] = 0.0
        potential=fftengine.ifftn(kappa_ft) #units should be rad**2
        return potential.real
```

FT of the convergence

FT of the Laplacian

IFT to get the potential

RESULT

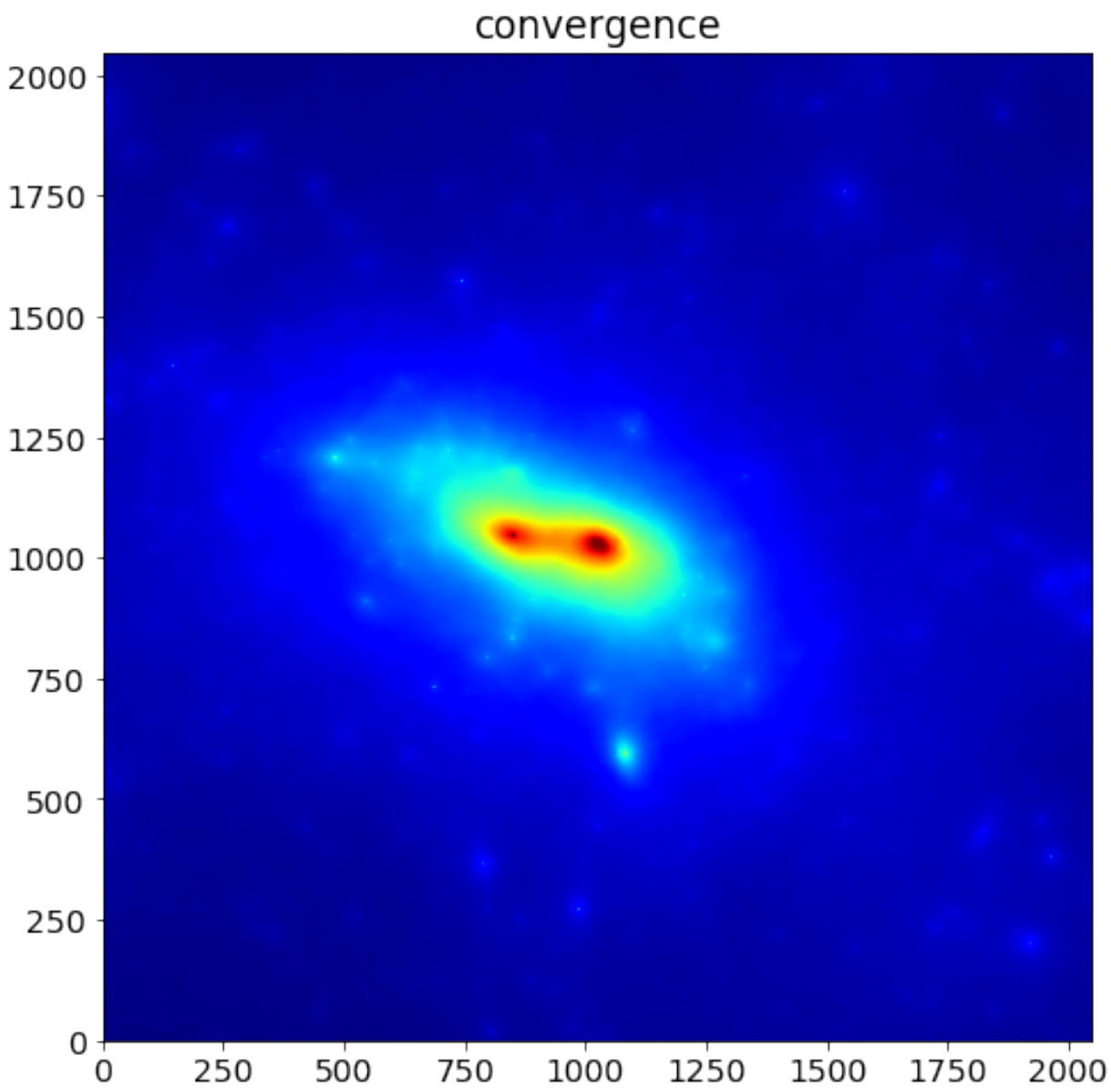
```
pot=df.potential() # compute the potential
kappa=df.mapCrop(kappa) # remove zero-padded region from
                        # convergence and potential maps
pot=df.mapCrop(pot)

# display the results
fig,ax = plt.subplots(1,2,figsize=(17,8))
ax[0].imshow(kappa,origin="lower")
ax[0].set_title('convergence')
ax[1].imshow(pot,origin="lower")
ax[1].set_title('potential')
```

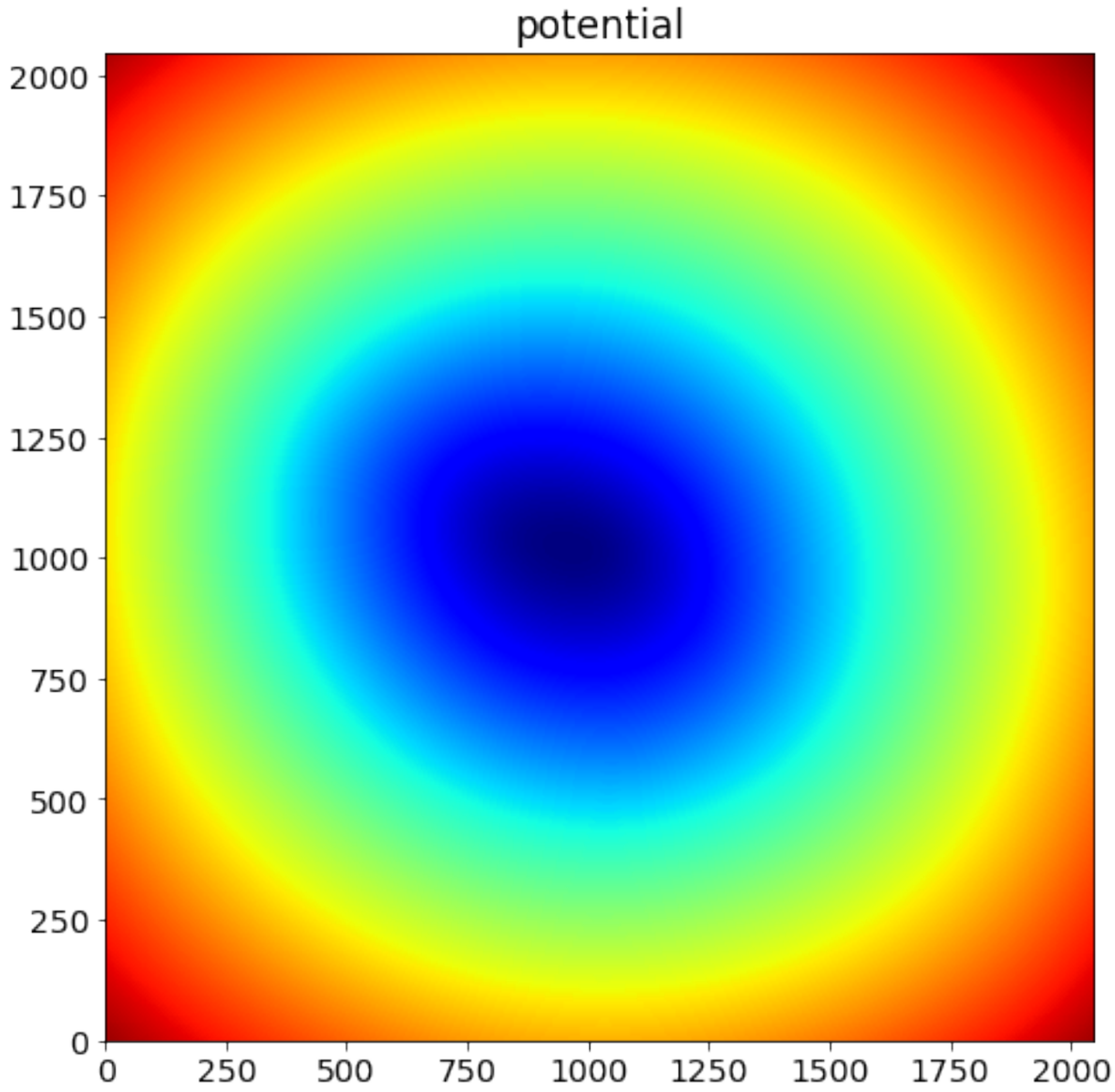
RESULT

Apply the potential method to the deflector df from the past lesson

```
pot=df.potential() # compute the potential
kappa=df.mapCrop(kappa) # remove zero-padded region from
                        # convergence and potential maps
pot=df.mapCrop(pot)

# display the results
fig,ax = plt.subplots(1,2,figsize=(17,8))
ax[0].imshow(kappa,origin="lower")
ax[0].set_title('convergence')
ax[1].imshow(pot,origin="lower")
ax[1].set_title('potential')
```

RESULT (NOTE THE DIFFERENT LEVEL OF SMOOTHNESS)

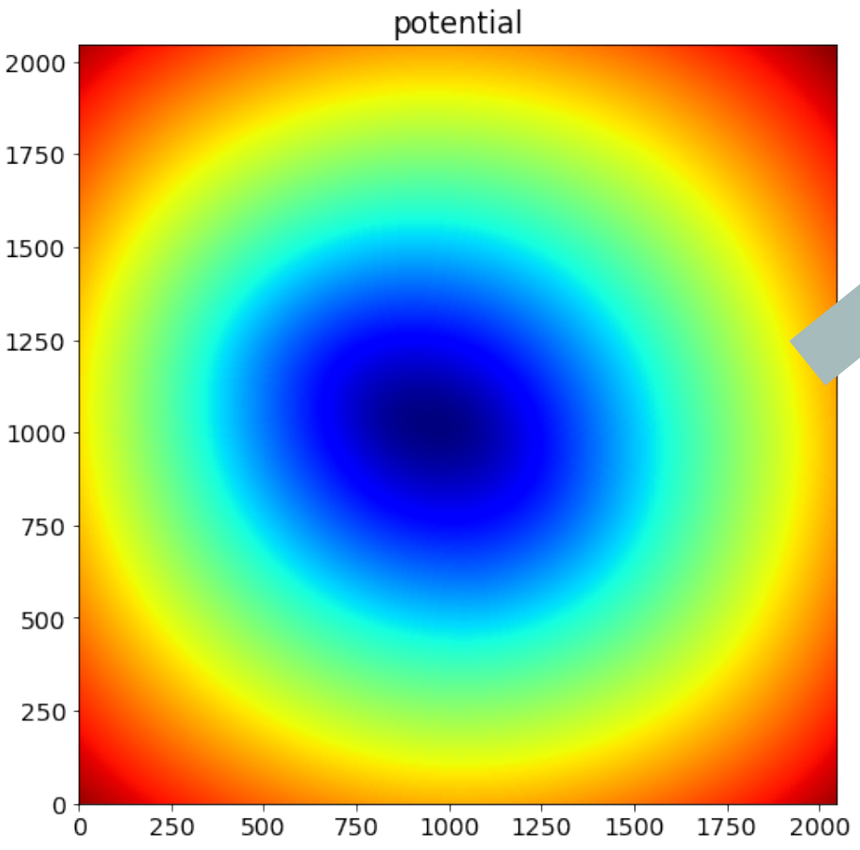


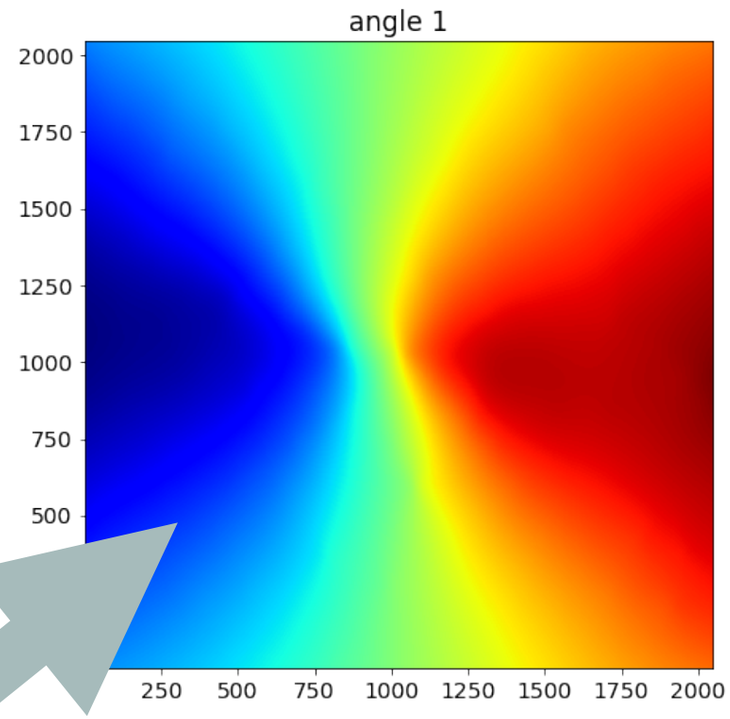


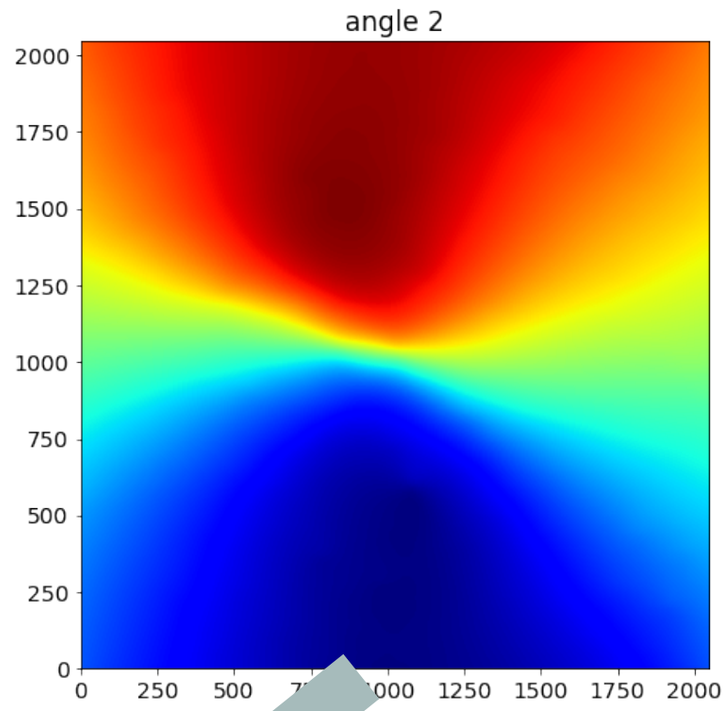
RESULT (NOTE THE DIFFERENT LEVEL OF SMOOTHNESS)

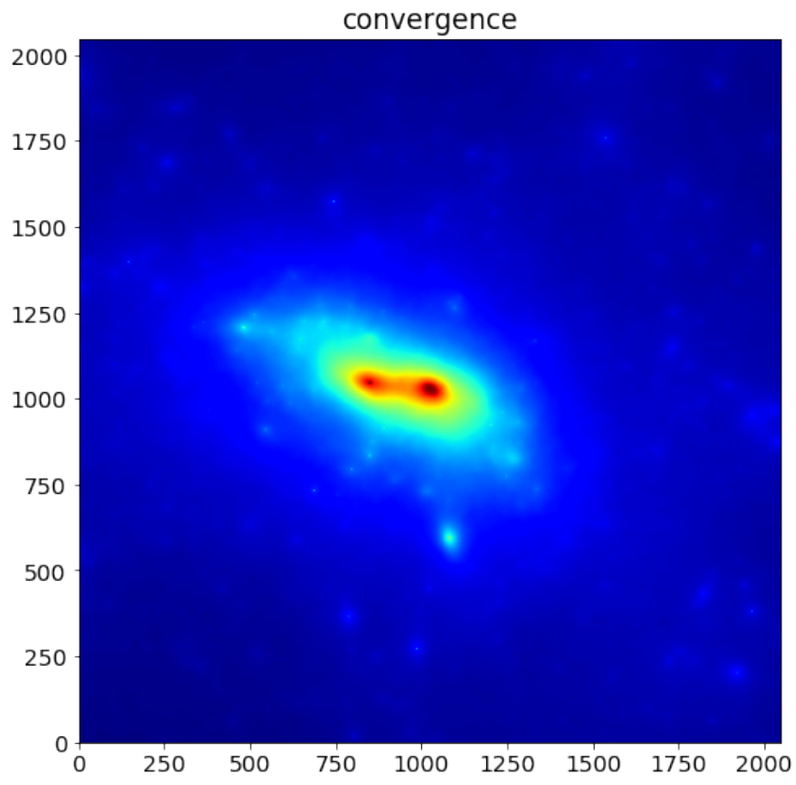
First derivative (gradient)

```
a2,a1=np.gradient(pot)
```







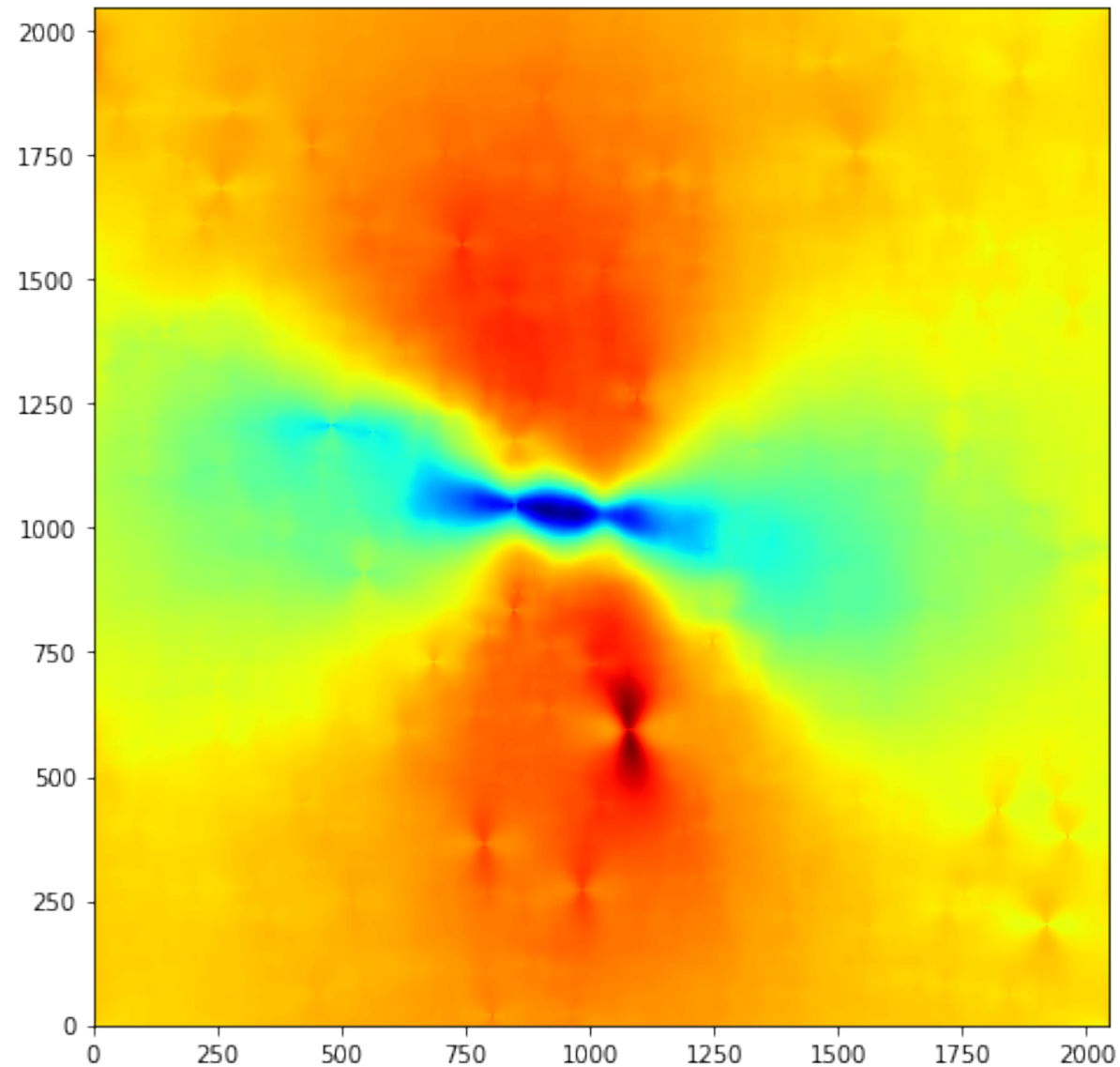


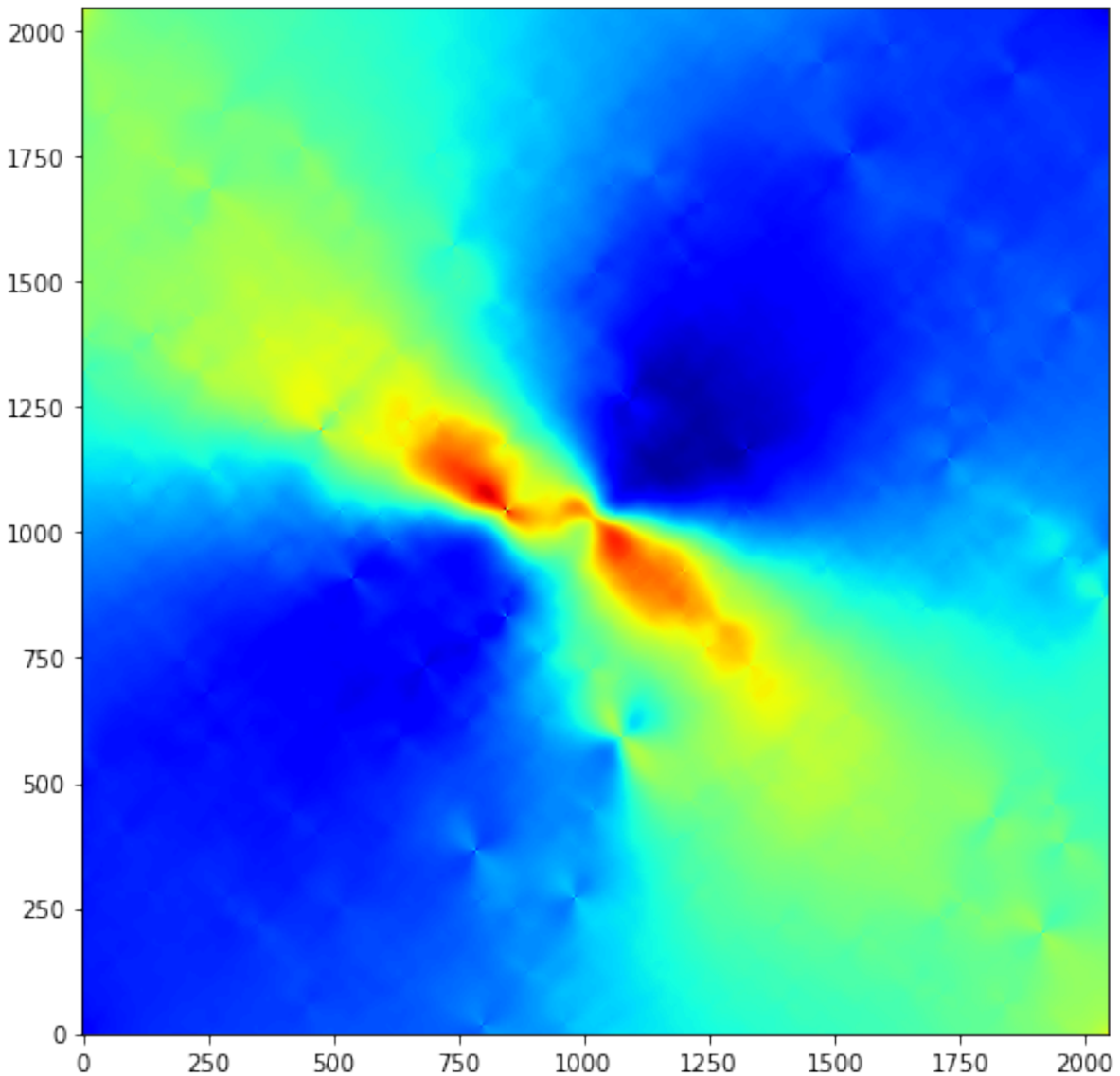
Second derivative (Laplacian)

```
a12,a11=np.gradient(a1)
a22,a21=np.gradient(a2)

ka=0.5*(a11+a22)
```

MAPS OF THE SHEAR COMMONENTS

```
gamma_1=0.5*(a11-a22)
gamma_2=a12
```





Note that patterns are rotated by 45 degrees

DIRECTION OF DISTORTIONS (FIRST ORDER)

```
pixel_step=gamma_1.shape[1]/64+1
x,y = np.meshgrid(np.arange(0,gamma_1.shape[1],pixel_step),
                  np.arange(0,gamma_1.shape[0],pixel_step))

phi=np.arctan2(gamma_2,gamma_1)/2.0
#phi[1-kappa+gamma<1-kappa-gamma]=phi[1-kappa+gamma<1-kappa-gamma]+np.pi/2
fig,ax=plt.subplots(1,2,figsize=(18,8))
ax[0].imshow(ka,origin='lower',vmax=3)
ax[1].imshow(ka,origin='lower',vmax=3)
gamma=np.sqrt(gamma_1**2+gamma_2**2)*5
# showing the intensity and the orientation of the shear
ax[0].quiver(y,x,gamma[x,y]*np.cos(phi[x,y]),gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[0].quiver(y,x,-gamma[x,y]*np.cos(phi[x,y]),-gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")

# showing only the orientation of the shear
fact=1.2
ax[1].quiver(y,x,fact*np.cos(phi[x,y]),fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[1].quiver(y,x,-fact*np.cos(phi[x,y]),-fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
```

DIRECTION OF DISTORTIONS (FIRST ORDER)

```
pixel_step=gamma_1.shape[1]/64+1
x,y = np.meshgrid(np.arange(0,gamma_1.shape[1],pixel_step),
                  np.arange(0,gamma_1.shape[0],pixel_step))

phi=np.arctan2(gamma_2,gamma_1)/2.0
#phi[1-kappa+gamma<1-kappa-gamma]=phi[1-kappa+gamma<1-kappa-gamma]+np.pi/2
fig,ax=plt.subplots(1,2,figsize=(18,8))
ax[0].imshow(ka,origin='lower',vmax=3)
ax[1].imshow(ka,origin='lower',vmax=3)
gamma=np.sqrt(gamma_1**2+gamma_2**2)*5
# showing the intensity and the orientation of the shear
ax[0].quiver(y,x,gamma[x,y]*np.cos(phi[x,y]),gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[0].quiver(y,x,-gamma[x,y]*np.cos(phi[x,y]),-gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")

# showing only the orientation of the shear
fact=1.2
ax[1].quiver(y,x,fact*np.cos(phi[x,y]),fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[1].quiver(y,x,-fact*np.cos(phi[x,y]),-fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
```

phase of the shear

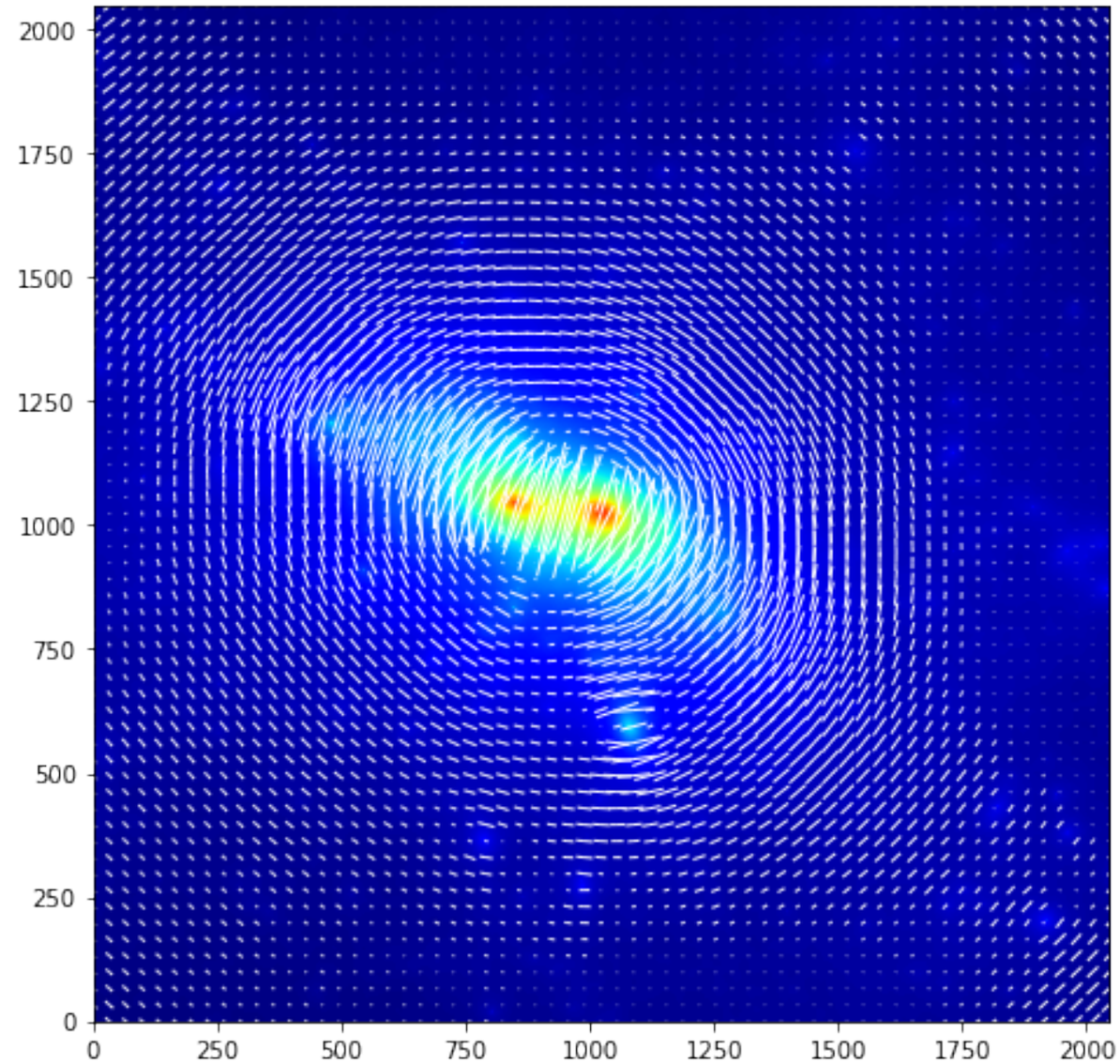
DIRECTION OF DISTORTIONS (FIRST ORDER)

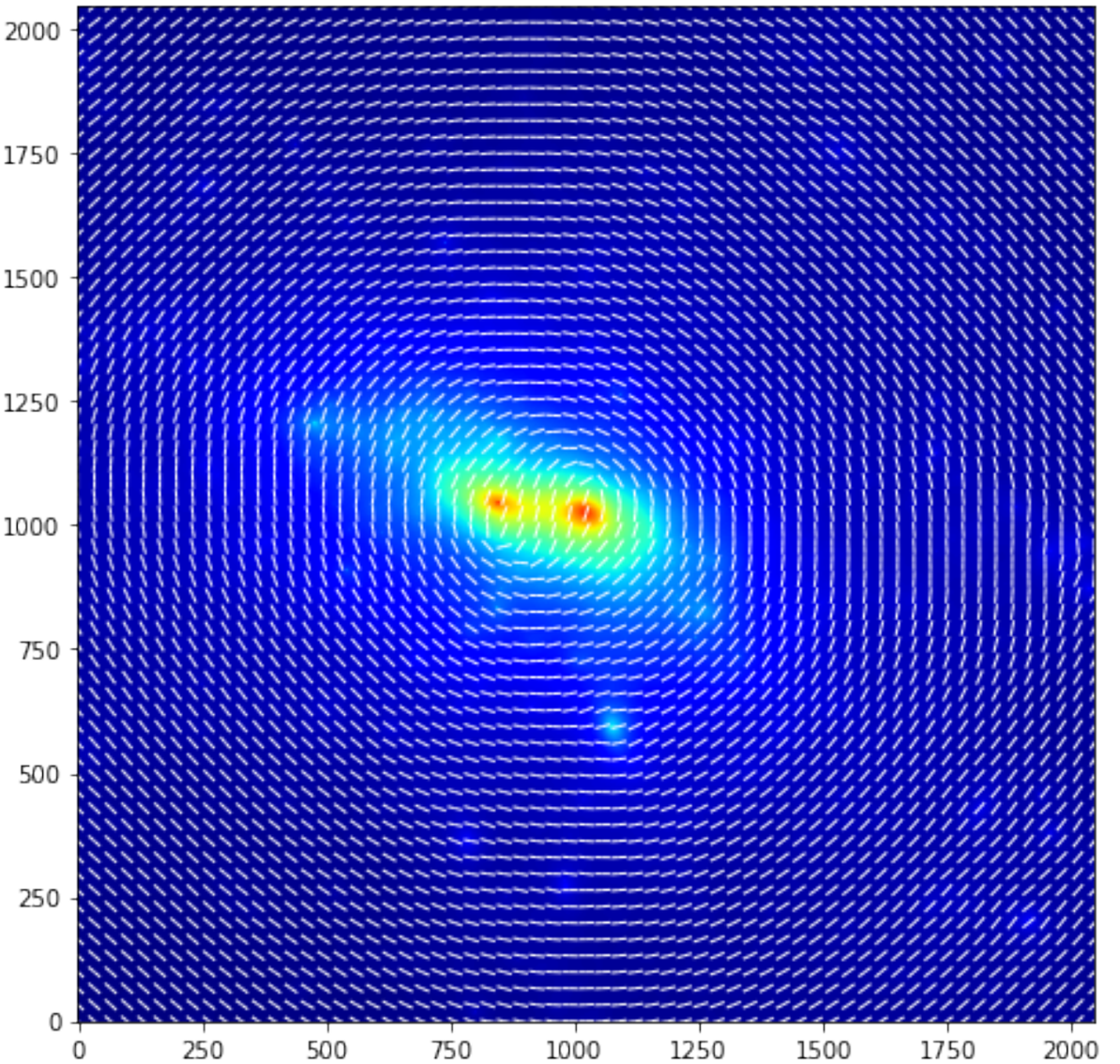
```
pixel_step=gamma_1.shape[1]/64+1
x,y = np.meshgrid(np.arange(0,gamma_1.shape[1],pixel_step),
                  np.arange(0,gamma_1.shape[0],pixel_step))

phi=np.arctan2(gamma_2,gamma_1)/2.0
#phi[1-kappa+gamma<1-kappa-gamma]=phi[1-kappa+gamma<1-kappa-gamma]+np.pi/2
fig,ax=plt.subplots(1,2,figsize=(18,8))
ax[0].imshow(ka,origin='lower',vmax=3)
ax[1].imshow(ka,origin='lower',vmax=3)
gamma=np.sqrt(gamma_1**2+gamma_2**2)*5
# showing the intensity and the orientation of the shear
ax[0].quiver(y,x,gamma[x,y]*np.cos(phi[x,y]),gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[0].quiver(y,x,-gamma[x,y]*np.cos(phi[x,y]),-gamma[x,y]*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")

# showing only the orientation of the shear
fact=1.2
ax[1].quiver(y,x,fact*np.cos(phi[x,y]),fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
ax[1].quiver(y,x,-fact*np.cos(phi[x,y]),-fact*np.sin(phi[x,y]),
             headwidth=0,units="height",scale=x.shape[0],color="white")
```

phase of the shear

shear as a stick: project along x,y

DIRECTION OF DISTORTIONS (FIRST ORDER)





shear traces mass!

shear is strong also in between mass clumps

CRITICAL LINES

$$\lambda_t = 1 - \kappa - \gamma = 0$$

$$\lambda_r = 1 - \kappa + \gamma = 0$$

```
from matplotlib.colors import LogNorm, PowerNorm, SymLogNorm

gamma=np.sqrt(gamma_1**2+gamma_2**2)
detA=(1.0-ka-gamma)*(1.0-ka+gamma)
lambdat=1.0-ka-gamma
lambdar=1.0-ka+gamma

fig,ax=plt.subplots(1,3,figsize=(28,8))
ax[0].imshow(lambdat,origin='lower')
ax[0].contour(lambdat,levels=[0.0])
ax[0].set_title('$\lambda_t$',fontsize=25)
ax[1].imshow(lambdar,origin='lower')
ax[1].contour(lambdar,levels=[0.0])
ax[1].set_title('$\lambda_r$',fontsize=25)
ax[2].imshow(detA,origin='lower',norm=SymLogNorm(0.3))
ax[2].contour(detA,levels=[0.0])
ax[2].set_title('$\det A$',fontsize=25)
fig.savefig('critlines_sim.png')
```

CRITICAL LINES

$$\lambda_t = 1 - \kappa - \gamma = 0$$

$$\lambda_r = 1 - \kappa + \gamma = 0$$

```
from matplotlib.colors import LogNorm, PowerNorm, SymLogNorm

gamma=np.sqrt(gamma_1**2+gamma_2**2)
detA=(1.0-ka-gamma)*(1.0-ka+gamma)
lambdat=1.0-ka-gamma
lambdar=1.0-ka+gamma

fig,ax=plt.subplots(1,3,figsize=(28,8))
ax[0].imshow(lambdat,origin='lower')
ax[0].contour(lambdat,levels=[0.0])
ax[0].set_title('$\lambda_t$',fontsize=25)
ax[1].imshow(lambdar,origin='lower')
ax[1].contour(lambdar,levels=[0.0])
ax[1].set_title('$\lambda_r$',fontsize=25)
ax[2].imshow(detA,origin='lower',norm=SymLogNorm(0.3))
ax[2].contour(detA,levels=[0.0])
ax[2].set_title('$\det A$',fontsize=25)
fig.savefig('critlines_sim.png')
```

Maps of detA and of the eigenvalues of A

CRITICAL LINES

$$\lambda_t = 1 - \kappa - \gamma = 0$$

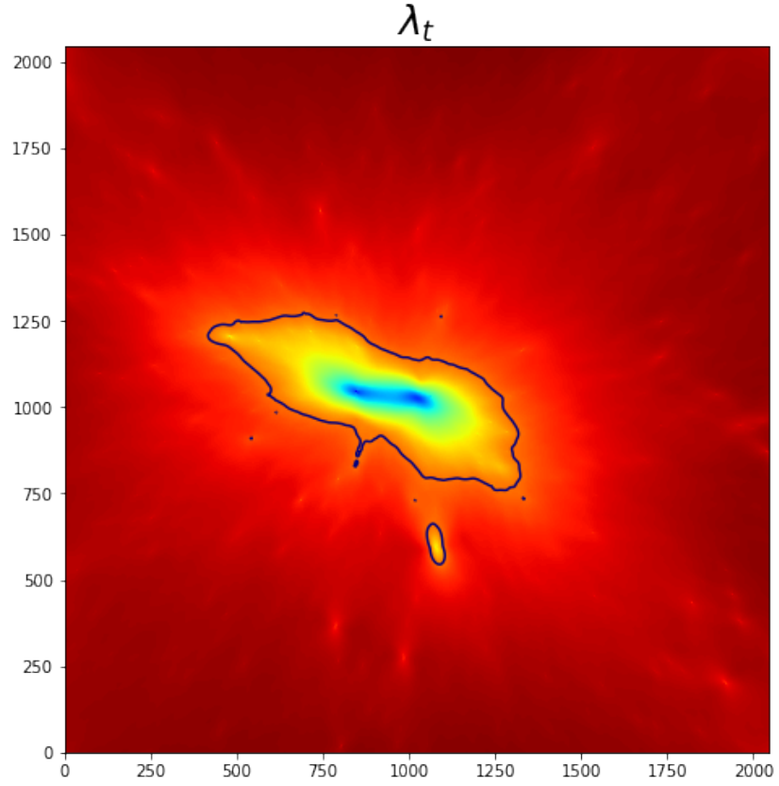
$$\lambda_r = 1 - \kappa + \gamma = 0$$

```
from matplotlib.colors import LogNorm, PowerNorm, SymLogNorm

gamma=np.sqrt(gamma_1**2+gamma_2**2)
detA=(1.0-ka-gamma)*(1.0-ka+gamma)
lambdat=1.0-ka-gamma
lambdar=1.0-ka+gamma

fig,ax=plt.subplots(1,3,figsize=(28,8))
ax[0].imshow(lambdat,origin='lower')
ax[0].contour(lambdat,levels=[0.0])
ax[0].set_title('$\lambda_t$',fontsize=25)
ax[1].imshow(lambdar,origin='lower')
ax[1].contour(lambdar,levels=[0.0])
ax[1].set_title('$\lambda_r$',fontsize=25)
ax[2].imshow(detA,origin='lower',norm=SymLogNorm(0.3))
ax[2].contour(detA,levels=[0.0])
ax[2].set_title('$\det A$',fontsize=25)
fig.savefig('critlines_sim.png')
```

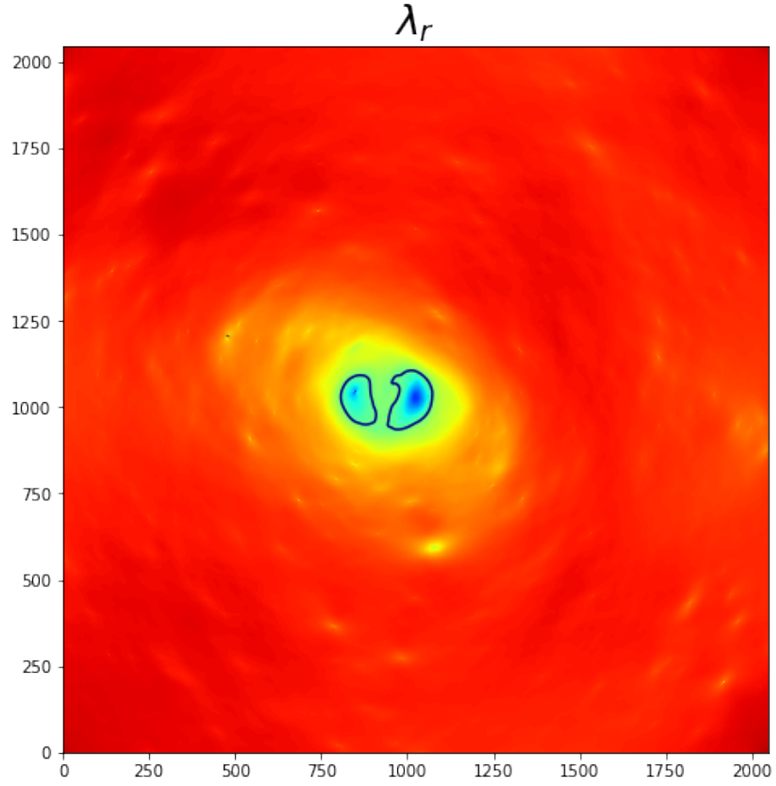
Maps of detA and of the eigenvalues of A

0-level contours

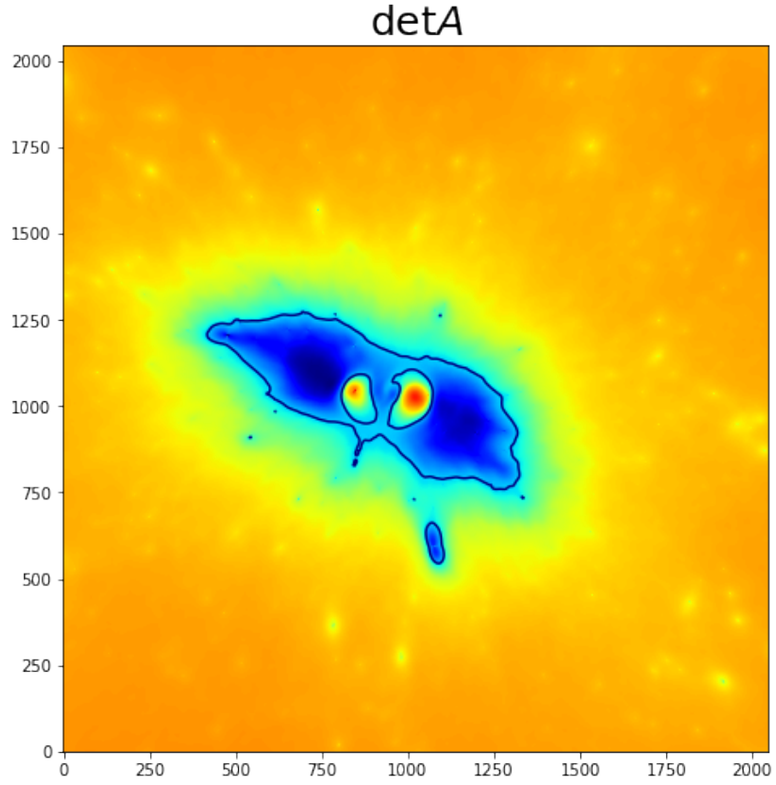
CRITICAL LINES



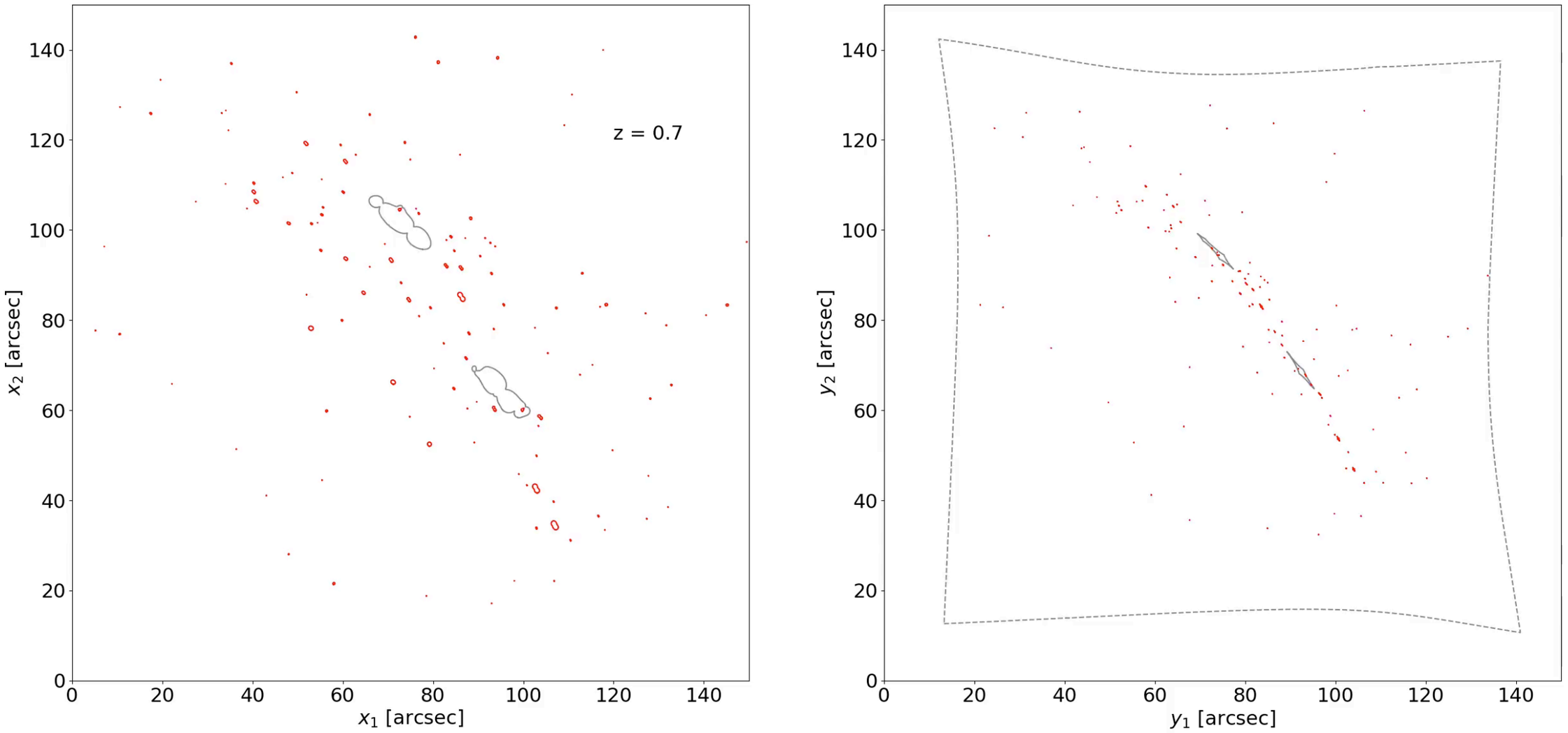
tangential



radial



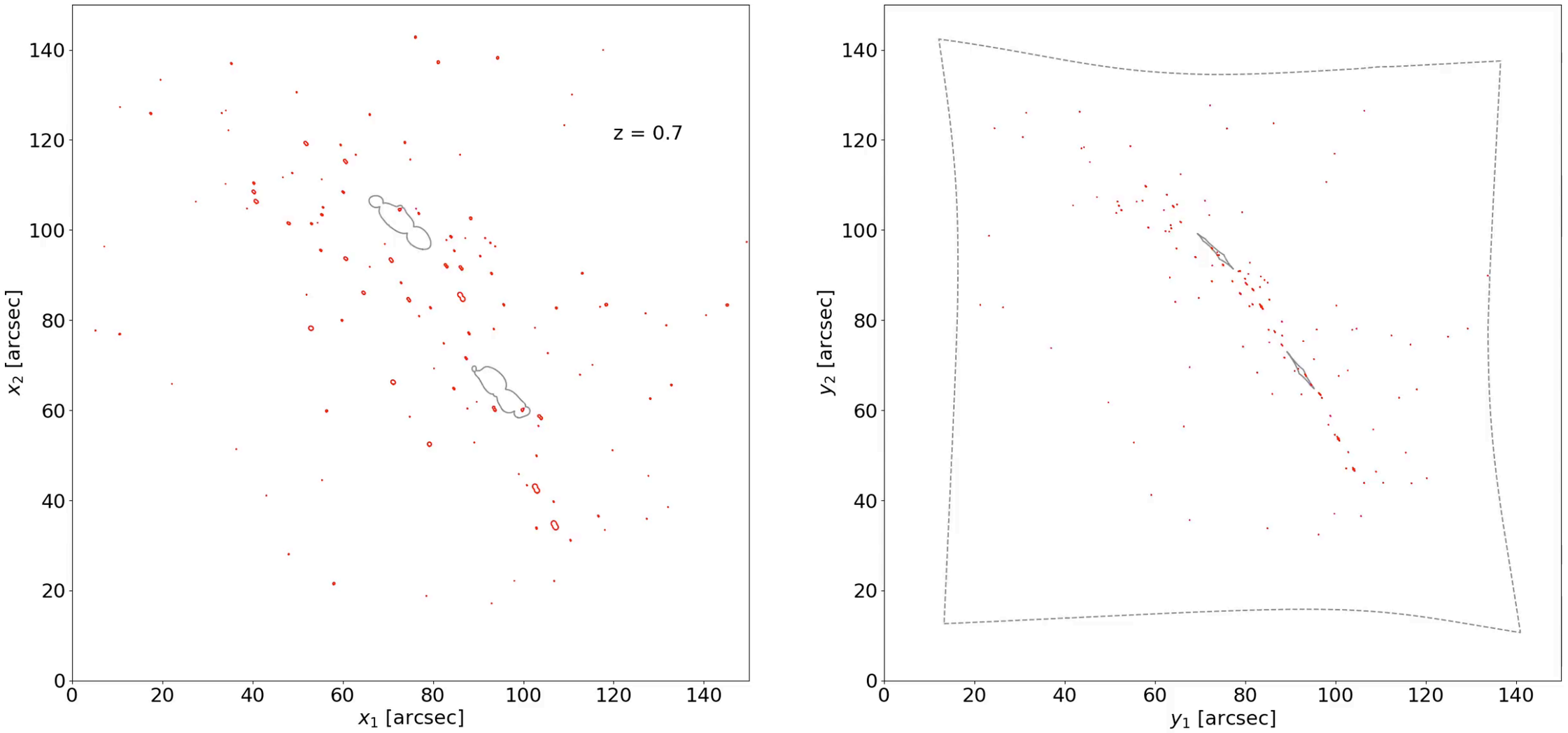
both



multiple images of two different sources at different redshift constrain the "family ratio"

$$\Xi(z_{\rm d}, z_{\rm s_1}, z_{\rm s_2}) = \frac{D_{\rm ds_1} D_{\rm s_2}}{D_{\rm s_1} D_{\rm ds_2}}$$

which in turn depends on cosmological parameters (Soucail et al. 2004; Jullo et al. 2010)



multiple images of two different sources at different redshift constrain the "family ratio"

$$\Xi(z_{\rm d}, z_{\rm s_1}, z_{\rm s_2}) = \frac{D_{\rm ds_1} D_{\rm s_2}}{D_{\rm s_1} D_{\rm ds_2}}$$

which in turn depends on cosmological parameters (Soucail et al. 2004; Jullo et al. 2010)

REDSHIFT DEPENDENCE

```
from astropy.cosmology import FlatLambdaCDM
cosmo = FlatLambdaCDM(H0=70, Om0=0.3)

zl=0.5
zs=np.linspace(zl,10.0,20)
dl=cosmo.angular_diameter_distance(zl)
ds=cosmo.angular_diameter_distance(zs)
dls=[]
for i in range(ds.size):
    dls.append(cosmo.angular_diameter_distance_z1z2(zl,zs[i]).value)

zs_norm=9.0
ds_norm=cosmo.angular_diameter_distance(zs_norm)
dls_norm=cosmo.angular_diameter_distance_z1z2(zl,zs_norm)

fig,ax=plt.subplots(1,2,figsize=(16,8))
ax[0].imshow(lambdat,origin='lower')
ax[1].imshow(lambdar,origin='lower')
for i in range(ds.size):
    kappa_new=ka*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    gamma_new=gamma*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    lambdat_new=(1.0-kappa_new-gamma_new)
    lambdar_new=(1.0-kappa_new+gamma_new)
    ax[0].contour(lambdat_new,levels=[0.0])
    ax[1].contour(lambdar_new,levels=[0.0])

ax[0].contour(lambdat,levels=[0.0],colors="yellow",linewidths=2)
ax[1].contour(lambdar,levels=[0.0],colors="magenta",linewidths=2)
```

REDSHIFT DEPENDENCE

cosmological model and angular diameter distances for each source redshift

```
from astropy.cosmology import FlatLambdaCDM
cosmo = FlatLambdaCDM(H0=70, Om0=0.3)

zl=0.5
zs=np.linspace(zl,10.0,20)
dl=cosmo.angular_diameter_distance(zl)
ds=cosmo.angular_diameter_distance(zs)
dls=[]
for i in range(ds.size):
    dls.append(cosmo.angular_diameter_distance_z1z2(zl,zs[i]).value)

zs_norm=9.0
ds_norm=cosmo.angular_diameter_distance(zs_norm)
dls_norm=cosmo.angular_diameter_distance_z1z2(zl,zs_norm)

fig,ax=plt.subplots(1,2,figsize=(16,8))
ax[0].imshow(lambdat,origin='lower')
ax[1].imshow(lambdar,origin='lower')
for i in range(ds.size):
    kappa_new=ka*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    gamma_new=gamma*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    lambdat_new=(1.0-kappa_new-gamma_new)
    lambdar_new=(1.0-kappa_new+gamma_new)
    ax[0].contour(lambdat_new,levels=[0.0])
    ax[1].contour(lambdar_new,levels=[0.0])

ax[0].contour(lambdat,levels=[0.0],colors="yellow",linewidths=2)
ax[1].contour(lambdar,levels=[0.0],colors="magenta",linewidths=2)
```

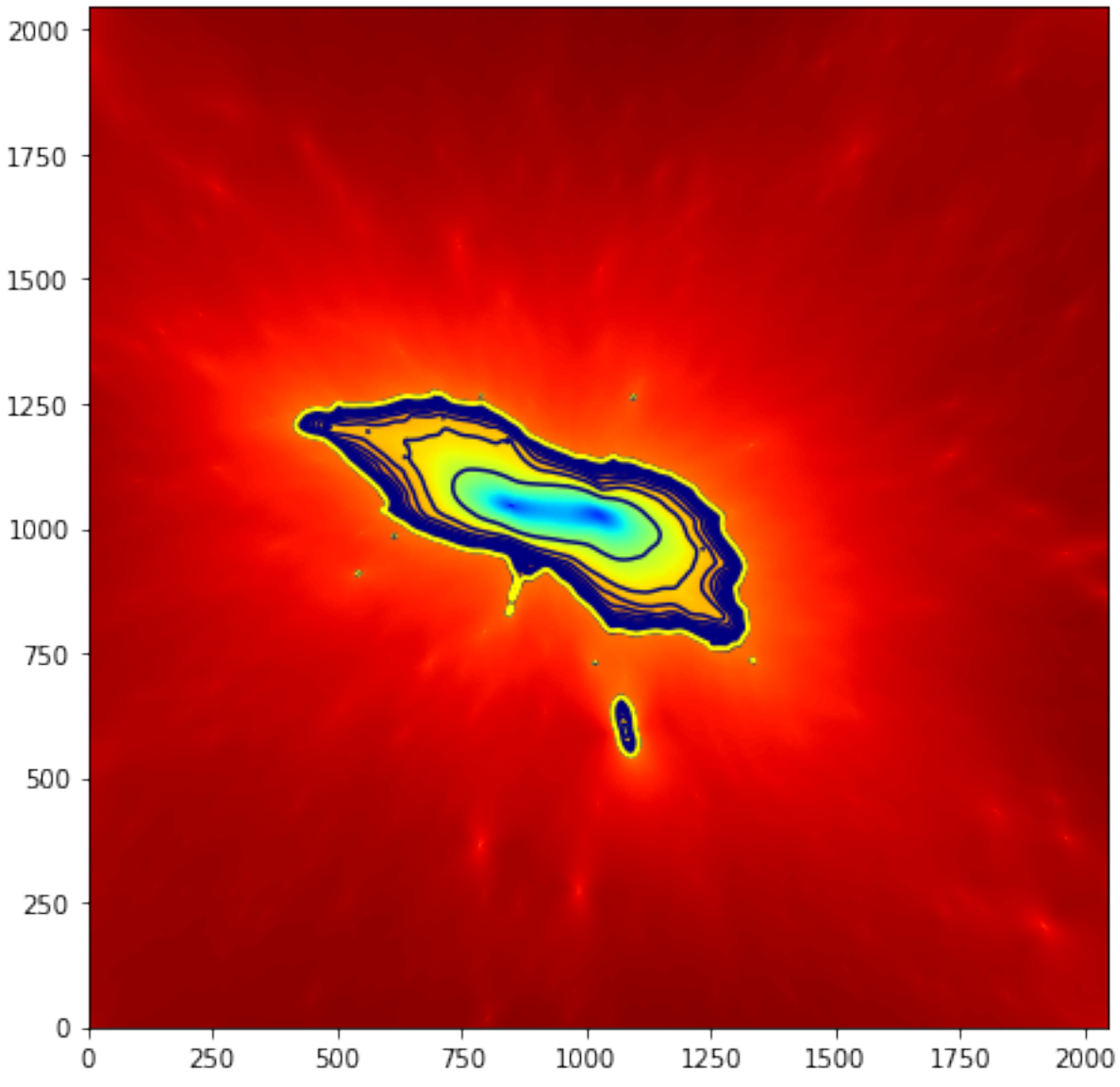
REDSHIFT DEPENDENCE

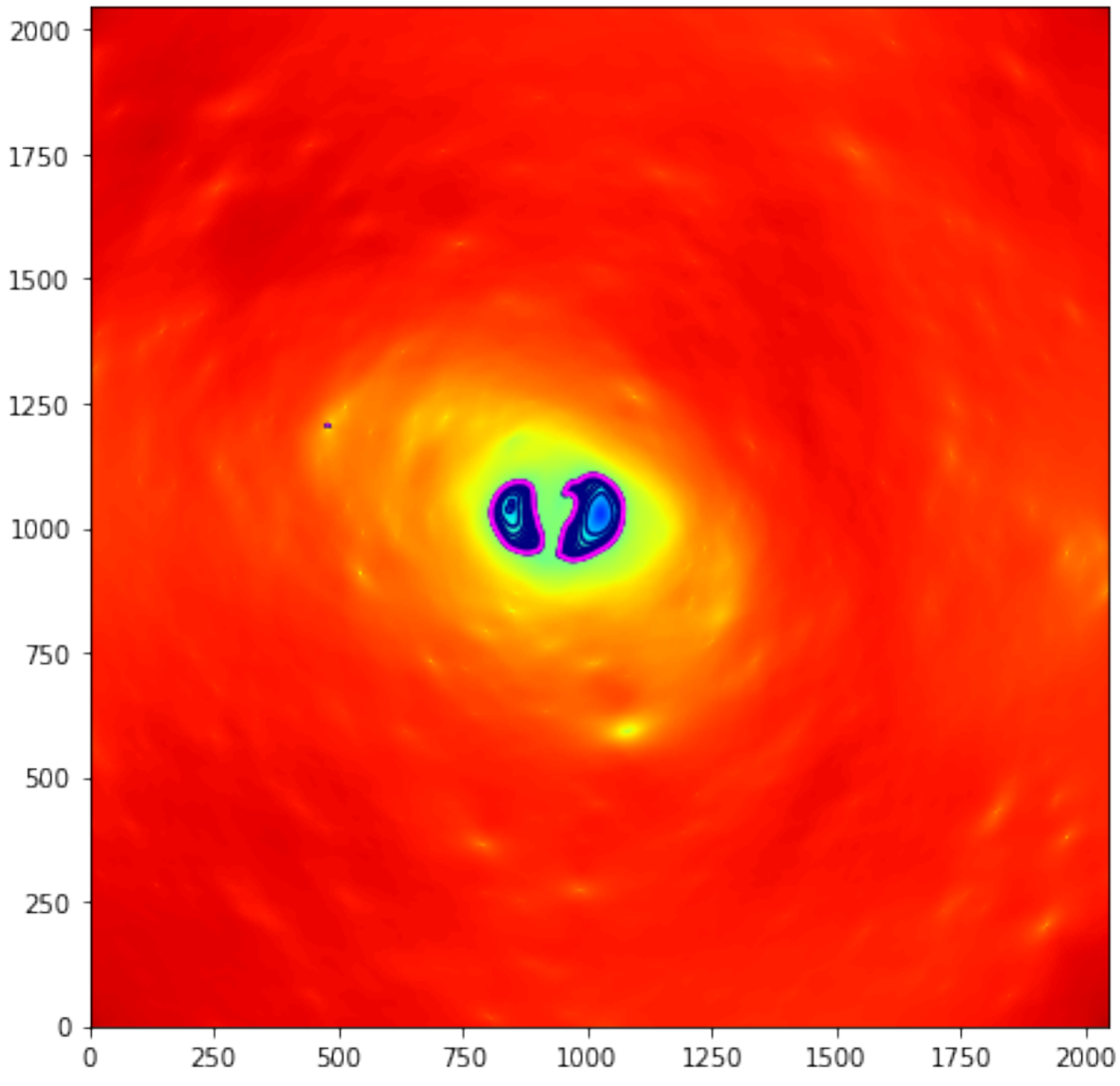
cosmological model and angular diameter distances for each source redshift

```
from astropy.cosmology import FlatLambdaCDM
cosmo = FlatLambdaCDM(H0=70, Om0=0.3)

zl=0.5
zs=np.linspace(zl,10.0,20)
dl=cosmo.angular_diameter_distance(zl)
ds=cosmo.angular_diameter_distance(zs)
dls=[]
for i in range(ds.size):
    dls.append(cosmo.angular_diameter_distance_z1z2(zl,zs[i]).value)

zs_norm=9.0
ds_norm=cosmo.angular_diameter_distance(zs_norm)
dls_norm=cosmo.angular_diameter_distance_z1z2(zl,zs_norm)

fig,ax=plt.subplots(1,2,figsize=(16,8))
ax[0].imshow(lambdat,origin='lower')
ax[1].imshow(lambdar,origin='lower')
for i in range(ds.size):
    kappa_new=ka*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    gamma_new=gamma*ds_norm.value/dls_norm.value*dls[i]/ds[i].value
    lambdat_new=(1.0-kappa_new-gamma_new)
    lambdar_new=(1.0-kappa_new+gamma_new)
    ax[0].contour(lambdat_new,levels=[0.0])
    ax[1].contour(lambdar_new,levels=[0.0])

ax[0].contour(lambdat,levels=[0.0],colors="yellow",linewidths=2)
ax[1].contour(lambdar,levels=[0.0],colors="magenta",linewidths=2)
```

Update convergence and shear for each redshift of the source

REDSHIFT DEPENDENCE





CAUSTICS

$$\vec{\beta}_c = \vec{\theta}_c - \vec{\alpha}(\vec{\theta}_c)$$

points on the caustics

points on the critical lines

deflection angle at the positions of the critical points

```
fig,ax=plt.subplots(1,2,figsize=(18,8))
# first we extract the level-0 contours of the map of detA
cs=ax[0].contour(detA,levels=[0.0])

# then, we take the path of each closed contour
contour=cs.collections[0]
p=contour.get_paths() # p contains the paths of each individual critical line

sizevs=np.empty(len(p),dtype=int)

from scipy.ndimage import map_coordinates

# if we found any contour, then we proceed
if (sizevs.size > 0):
    for j in range(len(p)):
        # for each path, we create two vectors containing the x1 and x2 coordinates of the vertices
        vs = contour.get_paths()[j].vertices
        sizevs[j]=len(vs)
        x1=[]
        x2=[]
        for i in range(len(vs)):
            xx1,xx2=vs[i]
            x1.append(float(xx1))
            x2.append(float(xx2))
        # these are the points we want to map back on the source plane.
        # To do that we need to evaluate the deflection angle at their positions
        # using scipy.ndimage.interpolate.map_coordinates we perform a bi-linear interpolation
        a_1=map_coordinates(a1, [[x2],[x1]],order=1)
        a_2=map_coordinates(a2, [[x2],[x1]],order=1)

        # now we can make the mapping using the lens equation:
        y1=x1-a_1[0]
        y2=x2-a_2[0]

        # plot the results!
        ax[0].plot(x1,x2,'-')
        ax[1].plot(y1,y2,'-')

ax[1].set_xlim([0,2048])
ax[1].set_ylim([0,2048])
```

```
fig,ax=plt.subplots(1,2,figsize=(18,8))
# first we extract the level-0 contours of the map of detA
cs=ax[0].contour(detA,levels=[0.0])

# then, we take the path of each closed contour
contour=cs.collections[0]
p=contour.get_paths() # p contains the paths of each individual critical line

sizevs=np.empty(len(p),dtype=int)

from scipy.ndimage import map_coordinates

# if we found any contour, then we proceed
if (sizevs.size > 0):
    for j in range(len(p)):
        # for each path, we create two vectors containing the x1 and x2 coordinates of the vertices
        vs = contour.get_paths()[j].vertices
        sizevs[j]=len(vs)
        x1=[]
        x2=[]
        for i in range(len(vs)):
            xx1,xx2=vs[i]
            x1.append(float(xx1))
            x2.append(float(xx2))
        # these are the points we want to map back on the source plane.
        # To do that we need to evaluate the deflection angle at their positions
        # using scipy.ndimage.interpolate.map_coordinates we perform a bi-linear interpolation
        a_1=map_coordinates(a1, [[x2],[x1]],order=1)
        a_2=map_coordinates(a2, [[x2],[x1]],order=1)

        # now we can make the mapping using the lens equation:
        y1=x1-a_1[0]
        y2=x2-a_2[0]

        # plot the results!
        ax[0].plot(x1,x2,'-')
        ax[1].plot(y1,y2,'-')

ax[1].set_xlim([0,2048])
ax[1].set_ylim([0,2048])
```

save the 0-level contours in an object and extract the path of each of them

```
fig,ax=plt.subplots(1,2,figsize=(18,8))
# first we extract the level-0 contours of the map of detA
cs=ax[0].contour(detA,levels=[0.0])

# then, we take the path of each closed contour
contour=cs.collections[0]
p=contour.get_paths() # p contains the paths of each individual critical line

sizevs=np.empty(len(p),dtype=int)

from scipy.ndimage import map_coordinates

# if we found any contour, then we proceed
if (sizevs.size > 0):
    for j in range(len(p)):
        # for each path, we create two vectors containing the x1 and x2 coordinates of the vertices
        vs = contour.get_paths()[j].vertices
        sizevs[j]=len(vs)
        x1=[]
        x2=[]
        for i in range(len(vs)):
            xx1,xx2=vs[i]
            x1.append(float(xx1))
            x2.append(float(xx2))
        # these are the points we want to map back on the source plane.
        # To do that we need to evaluate the deflection angle at their positions
        # using scipy.ndimage.interpolate.map_coordinates we perform a bi-linear interpolation
        a_1=map_coordinates(a1, [[x2],[x1]],order=1)
        a_2=map_coordinates(a2, [[x2],[x1]],order=1)

        # now we can make the mapping using the lens equation:
        y1=x1-a_1[0]
        y2=x2-a_2[0]

        # plot the results!
        ax[0].plot(x1,x2,'-')
        ax[1].plot(y1,y2,'-')

ax[1].set_xlim([0,2048])
ax[1].set_ylim([0,2048])
```

save the 0-level contours in an object and extract the path of each of them

make a list of the points in each path

```
fig,ax=plt.subplots(1,2,figsize=(18,8))
# first we extract the level-0 contours of the map of detA
cs=ax[0].contour(detA,levels=[0.0])

# then, we take the path of each closed contour
contour=cs.collections[0]
p=contour.get_paths() # p contains the paths of each individual critical line

sizevs=np.empty(len(p),dtype=int)

from scipy.ndimage import map_coordinates

# if we found any contour, then we proceed
if (sizevs.size > 0):
    for j in range(len(p)):
        # for each path, we create two vectors containing the x1 and x2 coordinates of the vertices
        vs = contour.get_paths()[j].vertices
        sizevs[j]=len(vs)
        x1=[]
        x2=[]
        for i in range(len(vs)):
            xx1,xx2=vs[i]
            x1.append(float(xx1))
            x2.append(float(xx2))
        # these are the points we want to map back on the source plane.
        # To do that we need to evaluate the deflection angle at their positions
        # using scipy.ndimage.interpolate.map_coordinates we perform a bi-linear interpolation
        a_1=map_coordinates(a1, [[x2],[x1]],order=1)
        a_2=map_coordinates(a2, [[x2],[x1]],order=1)

        # now we can make the mapping using the lens equation:
        y1=x1-a_1[0]
        y2=x2-a_2[0]

        # plot the results!
        ax[0].plot(x1,x2,'-')
        ax[1].plot(y1,y2,'-')

ax[1].set_xlim([0,2048])
ax[1].set_ylim([0,2048])
```

save the 0-level contours in an object and extract the path of each of them

make a list of the points in each path

calculate deflection angles at each critical point

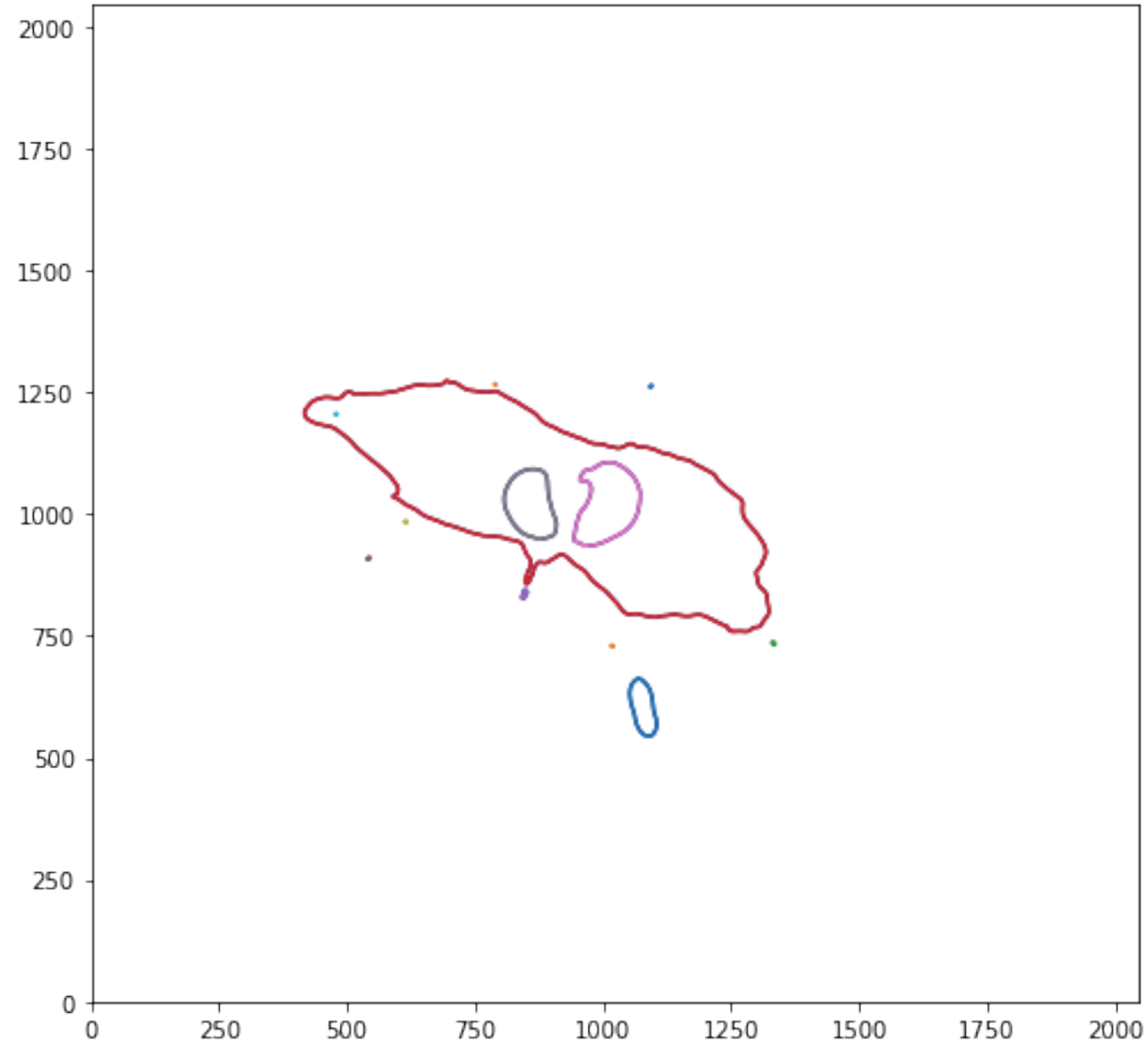
```
fig,ax=plt.subplots(1,2,figsize=(18,8))
# first we extract the level-0 contours of the map of detA
cs=ax[0].contour(detA,levels=[0.0])

# then, we take the path of each closed contour
contour=cs.collections[0]
p=contour.get_paths() # p contains the paths of each individual critical line

sizevs=np.empty(len(p),dtype=int)

from scipy.ndimage import map_coordinates

# if we found any contour, then we proceed
if (sizevs.size > 0):
    for j in range(len(p)):
        # for each path, we create two vectors containing the x1 and x2 coordinates of the vertices
        vs = contour.get_paths()[j].vertices
        sizevs[j]=len(vs)
        x1=[]
        x2=[]
        for i in range(len(vs)):
            xx1,xx2=vs[i]
            x1.append(float(xx1))
            x2.append(float(xx2))
        # these are the points we want to map back on the source plane.
        # To do that we need to evaluate the deflection angle at their positions
        # using scipy.ndimage.interpolate.map_coordinates we perform a bi-linear interpolation
        a_1=map_coordinates(a1, [[x2],[x1]],order=1)
        a_2=map_coordinates(a2, [[x2],[x1]],order=1)

        # now we can make the mapping using the lens equation:
        y1=x1-a_1[0]
        y2=x2-a_2[0]

        # plot the results!
        ax[0].plot(x1,x2,'-')
        ax[1].plot(y1,y2,'-')

ax[1].set_xlim([0,2048])
ax[1].set_ylim([0,2048])
```

save the 0-level contours in an object and extract the path of each of them

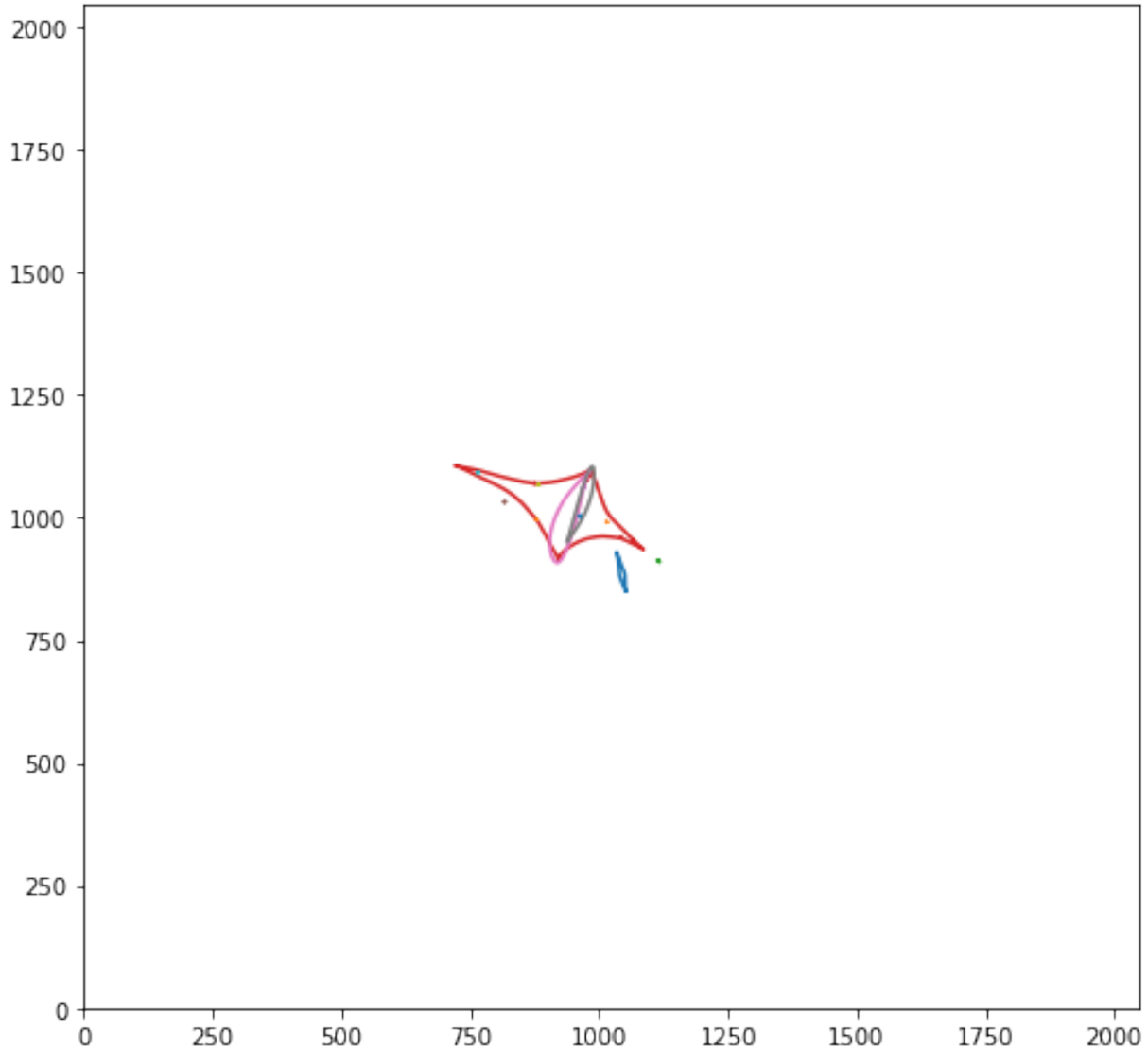
make a list of the points in each path

calculate deflection angles at each critical point

apply lens equation



critical lines



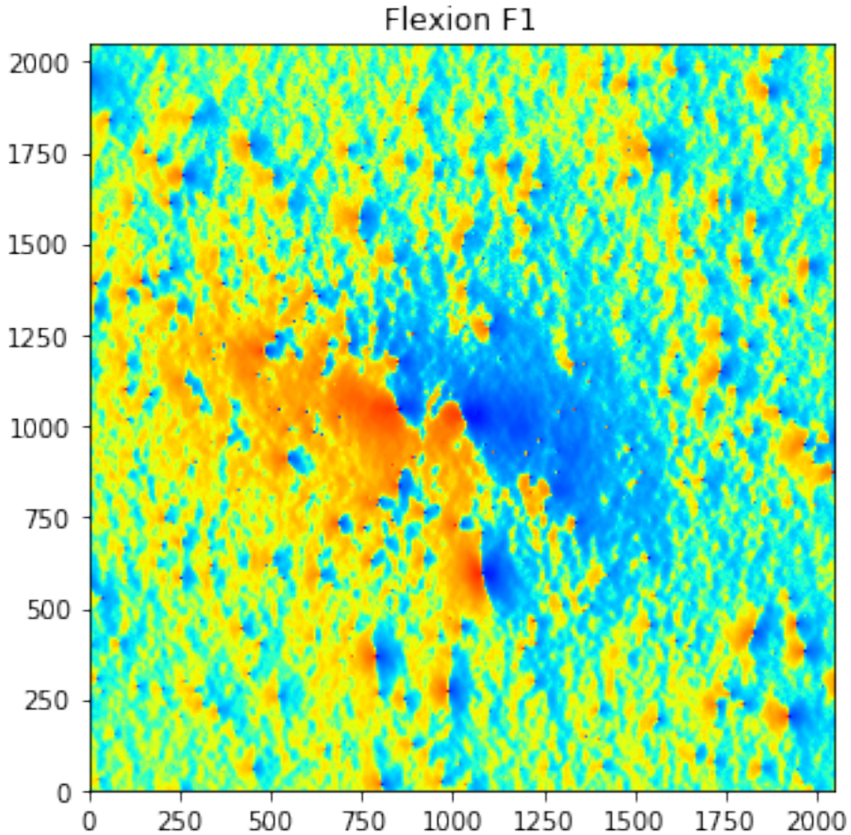
caustics

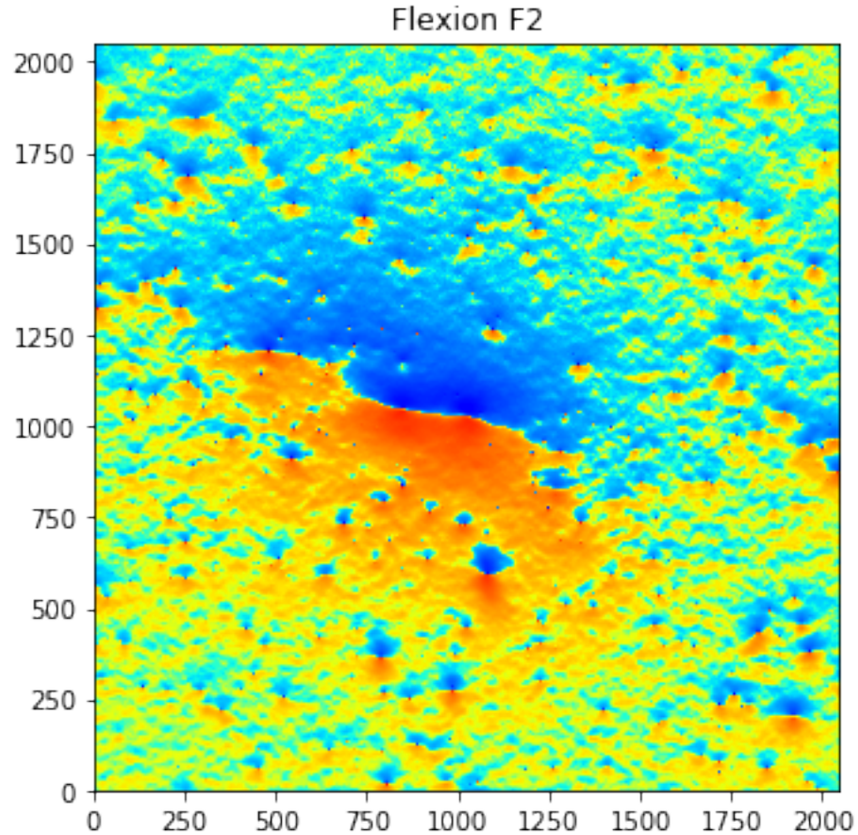
FLEXION

$$F = (\gamma_{1,1} + \gamma_{2,2}) + i(\gamma_{2,1} - \gamma_{1,2})$$

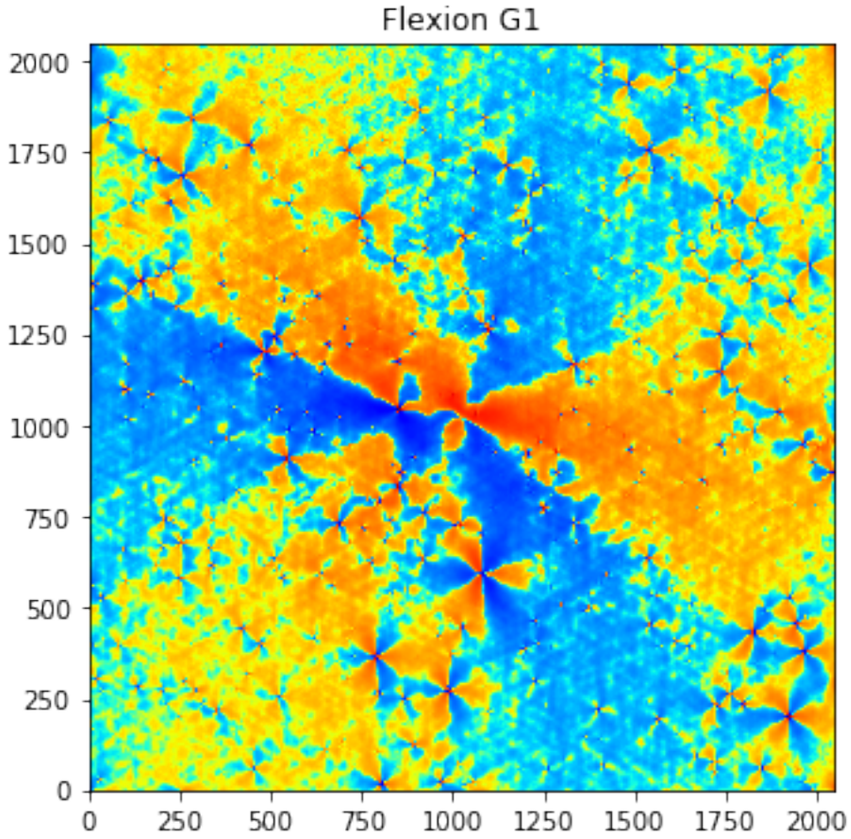
$$G = (\gamma_{1,1} - \gamma_{2,2}) + i(\gamma_{2,1} + \gamma_{1,2})$$

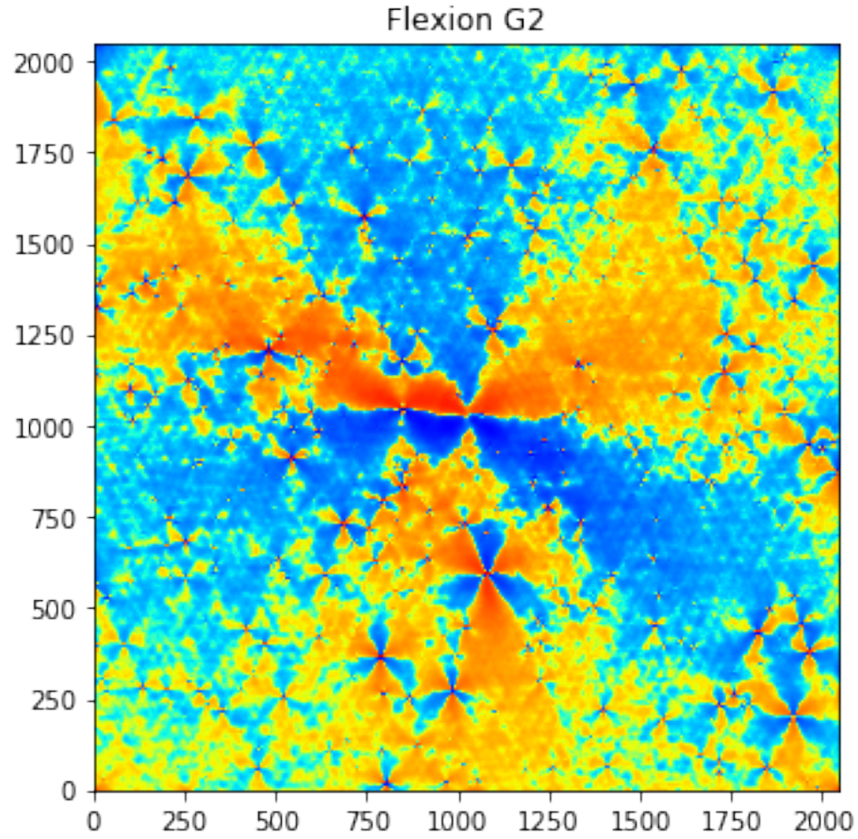
```
gamma12,gamma11=np.gradient(gamma_1)
gamma22,gamma21=np.gradient(gamma_2)
F1,F2=gamma11+gamma22,gamma21-gamma12
G1,G2=gamma11-gamma22,gamma21+gamma12
```


Flexion F1
2000
1750
1500
1250
1000
750
500
250
0
0
250
500
750
1000
1250
1500
1750
2000


Flexion F2
2000
1750
1500
1250
1000
750
500
250
0
0
250
500
750
1000
1250
1500
1750
2000

spin 1


Flexion G1
2000
1750
1500
1250
1000
750
500
250
0
0
250
500
750
1000
1250
1500
1750
2000


Flexion G2
2000
1750
1500
1250
1000
750
500
250
0
0
250
500
750
1000
1250
1500
1750
2000

spin 3

LENS DISTORTIONS UP TO 2ND ORDER

Lens equation including 2nd order terms

$$\beta_i = \frac{\partial \beta_i}{\partial \theta_j}\theta_j + \frac{1}{2}\frac{\partial \beta_i}{\partial \theta_j \partial \theta_k}\theta_j\theta_k$$

$$\beta_i = A_{ij}\theta_j + \frac{1}{2}D_{ijk}\theta_j\theta_k$$

Elements of D_{ijk}

$$D_{111} = -2\gamma_{11} - \gamma_{22} = -\frac{1}{2}(3F_1 + G_1)$$

$$D_{211} = D_{121} = D_{112} = -\gamma_{21} = -\frac{1}{2}(F_2 + G_2)$$

$$D_{122} = D_{212} = D_{221} = -\gamma_{22} = -\frac{1}{2}(F_1 - G_1)$$

$$D_{222} = 2\gamma_{12} - \gamma_{21} = -\frac{1}{2}(3F_2 - G_2)$$

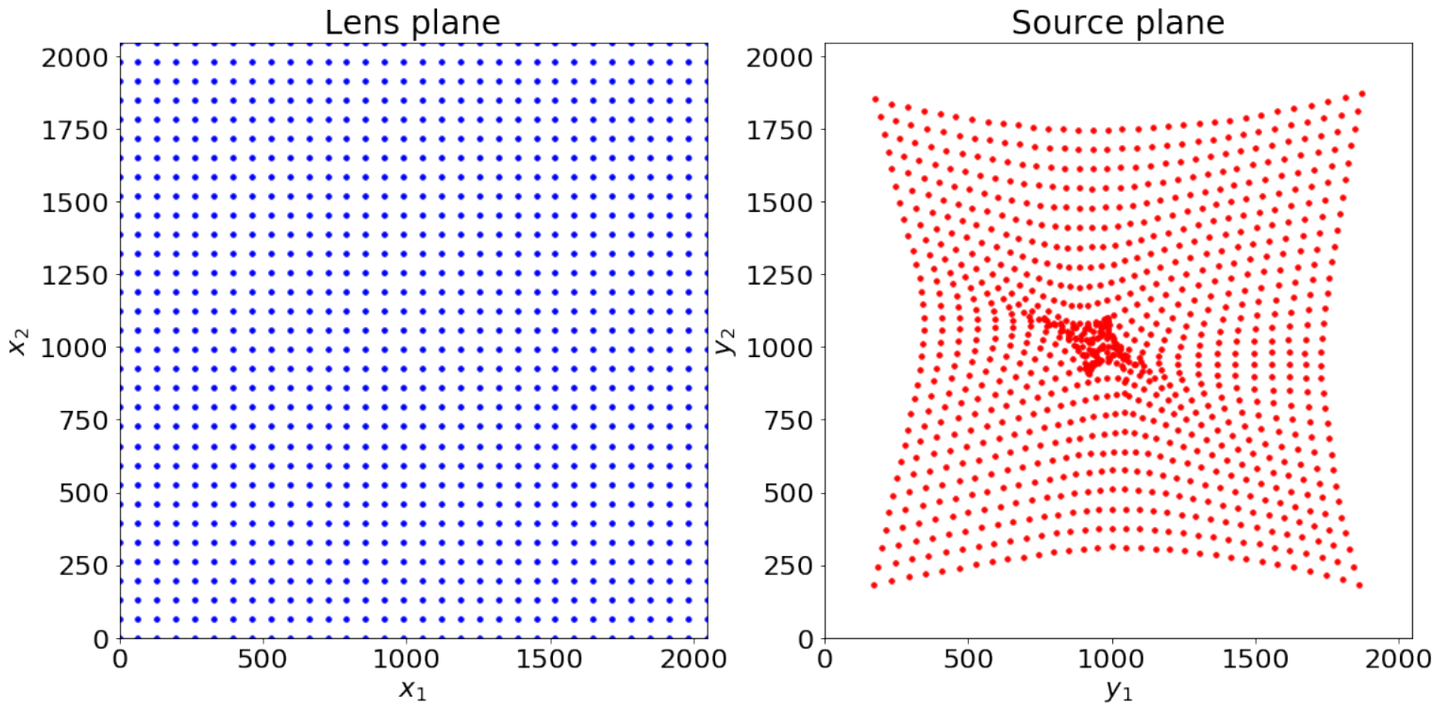
LENS DISTORTIONS UP TO 2ND ORDER

After some math:

$$\beta_1 = A_{11}\theta_1 + A_{12}\theta_2 + \frac{1}{2}D_{111}\theta_1^2 + D_{121}\theta_1\theta_2 + \frac{1}{2}D_{122}\theta_2^2$$

$$\beta_2 = A_{21}\theta_1 + A_{22}\theta_2 + \frac{1}{2}D_{211}\theta_1^2 + D_{212}\theta_1\theta_2 + \frac{1}{2}D_{222}\theta_2^2$$

These equations allow us to perform ray-tracing: we can apply them to the coordinates of the points on a grid covering the lens plane and find the arrival positions of the source plane (well…only to second order approximation!)



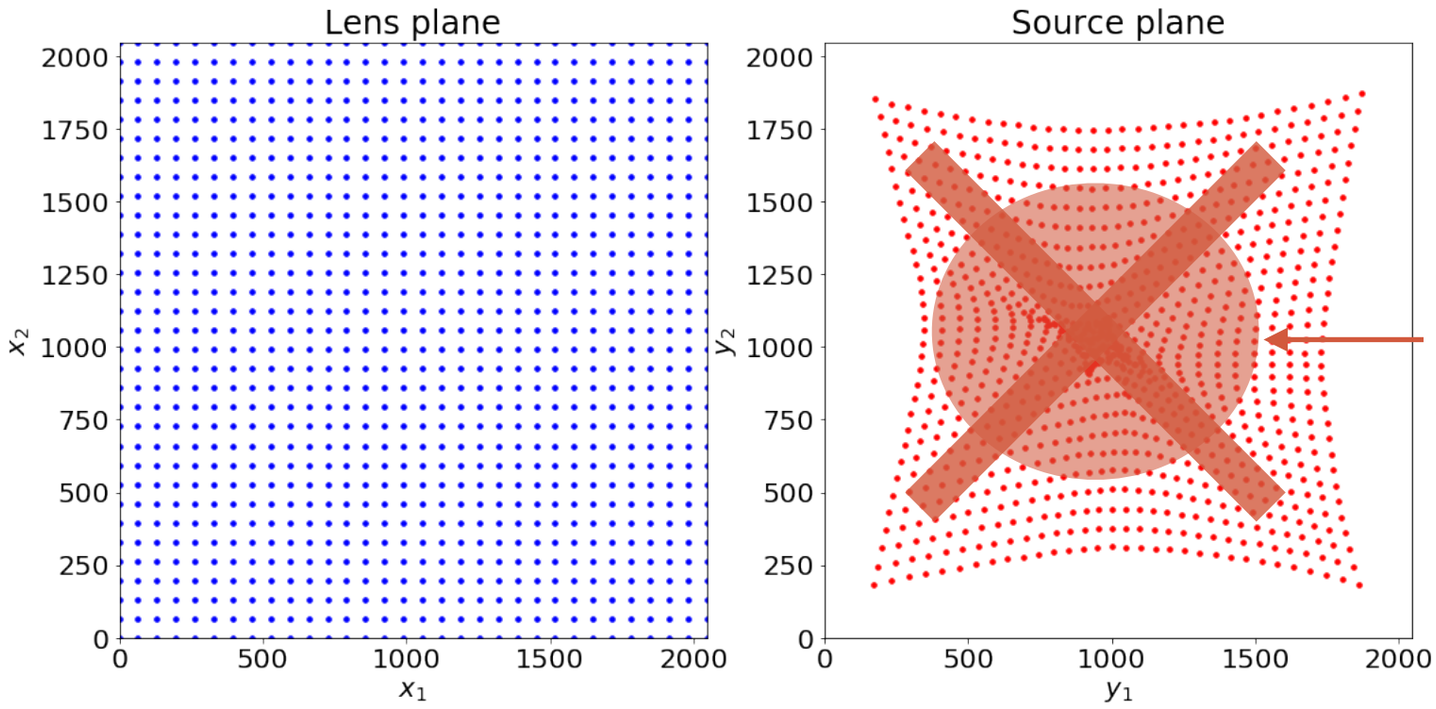
LENS DISTORTIONS UP TO 2ND ORDER

After some math:

$$\beta_1 = A_{11}\theta_1 + A_{12}\theta_2 + \frac{1}{2}D_{111}\theta_1^2 + D_{121}\theta_1\theta_2 + \frac{1}{2}D_{122}\theta_2^2$$

$$\beta_2 = A_{21}\theta_1 + A_{22}\theta_2 + \frac{1}{2}D_{211}\theta_1^2 + D_{212}\theta_1\theta_2 + \frac{1}{2}D_{222}\theta_2^2$$

These equations allow us to perform ray-tracing: we can apply them to the coordinates of the points on a grid covering the lens plane and find the arrival positions of the source plane (well…only to second order approximation!)

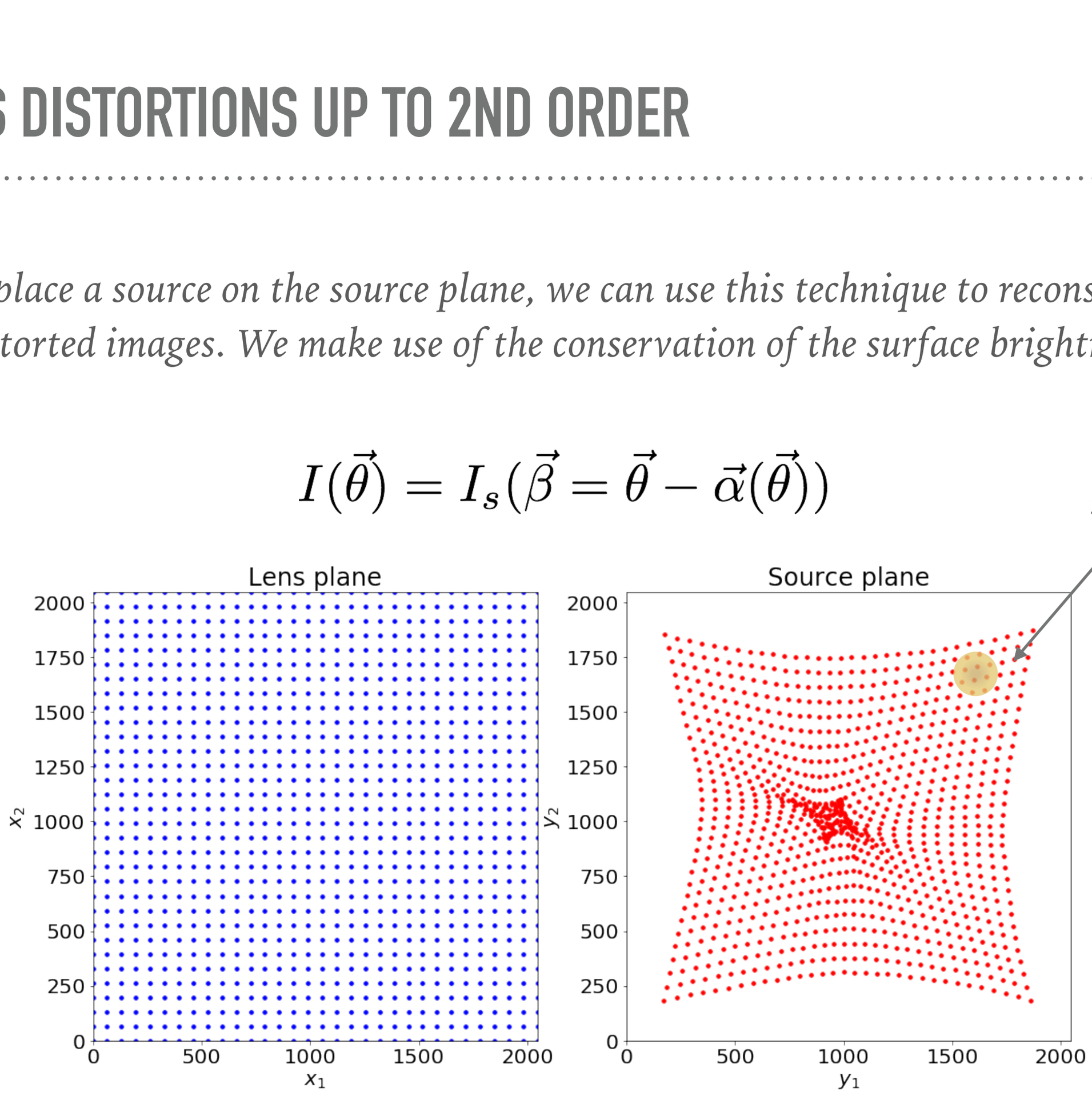


deflection angle varies too rapidly for these rays to apply 2nd order lens equation

LENS DISTORTIONS UP TO 2ND ORDER

If we place a source on the source plane, we can use this technique to reconstruct its distorted images. We make use of the conservation of the surface brightness:

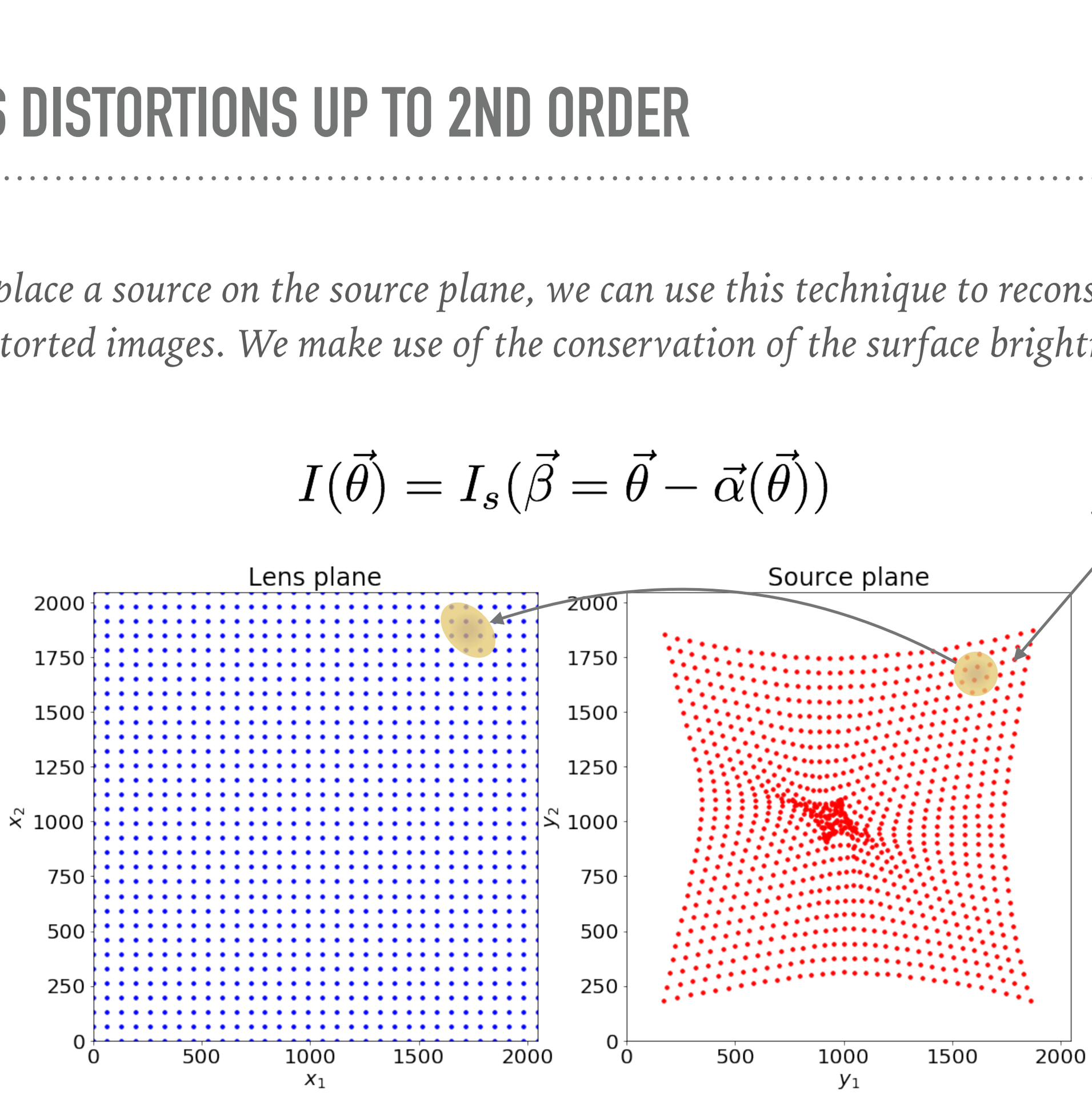
$$I(\vec{\theta}) = I_s(\vec{\beta} = \vec{\theta} - \vec{\alpha}(\vec{\theta}))$$



LENS DISTORTIONS UP TO 2ND ORDER

If we place a source on the source plane, we can use this technique to reconstruct its distorted images. We make use of the conservation of the surface brightness:

$$I(\vec{\theta}) = I_s(\vec{\beta} = \vec{\theta} - \vec{\alpha}(\vec{\theta}))$$



SHEAR AND FLEXION DISTORTIONS