

Astrophysical Applications of Gravitational Lensing

XXIV Canary Islands Winter School of Astrophysics

Edited by
E. Mediavilla, J. A. Muñoz

Contents

Tutorial on Inverse Ray Shooting
Jorge Jiménez-Vicente

page 1

Tutorial on Inverse Ray Shooting

Jorge Jiménez-Vicente

Departamento de Física Teórica y del Cosmos

Universidad de Granada

Av. Fuentenueva s/n, 18071 Granada, SPAIN

Abstract

Magnification maps are an essential tool in microlensing studies. Their calculation is based on very simple principles and it is therefore quite straightforward to implement. This tutorial is intended to show how this calculations are done by using a basic ray shooting procedure. The tutorial assumes some basic knowledge of any programming language, but no previous knowledge of the specific language used here, or experience with gravitational lensing computations is needed. The very basics of gravitational lensing are also implicitly assumed at some points. From the computational point of view, the tutorial covers from the simplest ray shooting program to generate images of an object through a simple lens system to the production of magnification maps for quasar microlensing. Source size effects and how to deal with them are also discussed in short. We will finish by also briefly discussing the main improvements that have been introduced into this technique to make calculations faster.

6.1 Introductory remarks

It is particularly difficult to put into written text what was intended as a couple of practical sessions on Inverse Ray Shooting techniques during the Winter School. Unfortunately, there is not much choice but to illustrate the ideas with some code. At this point, a choice has to be made on whether to use pseudocode or to choose a given programming language. I have chosen the second option here with the hope that the reader may actually use the code snippets presented here straight away, and be able to produce some useful programs from it. During the lectures, the Python programming language was used for the tutorial. I chose it for a couple of reasons that are enumerated below. I have therefore also used Python here

for presenting the techniques introduced in this tutorial. Nevertheless, the reader is not expected to know any Python in advance, as I shall introduce all the required information on Python syntax on section 6.2. Therefore, the reader should find no problem in following the explanations and/or in translating the code to his/her favourite programming language. Finally, this lectures were intended as practical sessions, and as such, the focus is put into producing some useful code and to understand the key steps in that process. Performance will be dealt with at some point in order introduce the necessary Python ingredients to make the programs fast enough to produce results in a reasonable time. But while performance is not something I have been particularly worried about for this tutorial, it is indeed important for real calculations. Therefore, although the reader should not expect to find here a thorough description of the many advanced techniques developed towards improving performance in these kind of calculations, some of the most successful and widely used techniques are briefly discussed at the very end of this tutorial.

6.2 A Short Introduction to Python, Numpy and Matplotlib

As mentioned above, Python will be used throughout this tutorial as the programming language to illustrate the Inverse Ray Shooting techniques in gravitational lensing. The main reasons to choose python for this purpose are:

(i) **It is free and easily available for most operative systems.**

Python is available in nearly any operative system running on nearly any hardware from supercomputers to mobile phones and gaming consoles. In particular, it is available in Linux/Unix in most (all?) distributions/flavours, Windows and Mac OS/OS X. This is certainly a very important aspect. Anybody can get Python and install it in virtually any computer for free in just a few minutes. This is indeed most convenient for a tutorial like this.

(ii) **It is an interpreted language.**

This is also a very handy property of Python (or more precisely of its standard implementation[†]). Therefore, after making any change to the code, it can be executed immediately without the need of recompiling or alike.

[†] In fact, Python is NOT interpreted. In the standard implementation, source code is indeed compiled to bytecode which is executed in a virtual machine. From the user perspective this processes are completely hidden so that in fact, it feels like it is indeed an interpreted language.

(iii) **It is easy to read/write and therefore easy to learn.**

Indeed, in many aspects there is not much difference between pseudocode and Python code. Being a dynamic typed language, the interpreter takes care of data types at run time, and the user should not take care of type declaration but to concentrate on other things. Memory management is also something the user may forget about for most purposes. It is therefore very well suited for teaching purposes.

(iv) **It is versatile and powerful (specially when combined with existing libraries).**

Python standard library is by itself a powerful tool with endless possibilities[†], but on top of it, there is now plenty of external libraries which make programmer's life even easier. Being a general purpose programming language, there is plenty of libraries for many specific purposes. Here, we will make use of some of them very handy in scientific computing: Numpy to operate in an efficient way with arrays, Scipy for some scientific calculations and/or image processing, matplotlib for plotting and images and pyfits for dealing with fits files.

If I were interested in convincing you in using Python, I could give you a few more reasons, like that it is really fun to use, or that it is a good investment in the mid term for a scientist in general and for an astronomer in particular. I could even try to impress you by telling you that Python is behind sites as well known as YouTube, and that it is widely used in scientific institutions as prestigious as NASA and the CERN. But that is not the case, so I shall stop the marketing here. You will find plenty of information on the official webpage: <http://www.python.org>.[‡]

Python can be used for procedural/functional programming as well as for object oriented programming. Although I may occasionally use some built-in object oriented methods, I shall follow a fully procedural/functional paradigm here. The code presented here will probably not be very *pythonic* and will violate most of the guiding principles in *The Zen of Python*. I hope the *Benevolent Dictator For Life* will forgive me for this. Not so sure about his acolytes.

[†] Python enthusiasts refer to this fact with the sentence “batteries included”

[‡] The code in this tutorial has been written to work with Python 2.6 or 2.7. In 2008 Python 3.0 was released which is not backwards compatible with the 2.x. Although versions of all the necessary libraries for this tutorial are available for Python 3.1 and 3.2, I have no experience with it and cannot recommend its use (nor discourage it).

6.2.1 Python in a (pine) nut shell

Python can be used in essentially two different ways: interactively or by using scripts. The Python interpreter can be invoked by typing *python* at the command line prompt. Once started, we can start to type commands. IPython is a fancy shell for interactive Python which is very convenient.

Alternatively, a script can be created with an editor. Usually, Python programs are indicated by a *.py* extension. A script named *script.py* can be executed as an argument of the python command with *python script.py*. As Python scripts can themselves have arguments, these should follow the script name. There are some useful development environments for Python, IDLE being widely used.

We will very quickly pass here over the most basic rudiments of Python to be able to follow the rest of the tutorial. The reader interested in more details is forwarded to *The Python Tutorial* at <http://docs.python.org/2/tutorial/index.html> or to the one written by Shipman (2011). Both are excellent places to look for more detailed information.

6.2.1.1 Variables and Data Types

We have already stated that Python is dynamically typed, meaning that variable names do not have an associated data type. Inside Python, objects do indeed have a well defined type, but we can refer to them by any name at any time. Types are linked to the objects, not to their names. Therefore, a variable name *a* can be used to label an integer and, within the same session or program, be reused to label a string, float, or any other object. This is extremely handy, as we do not usually have to care much about variable declaration, but it also brings some risks. If we mistype a variable name when trying to operate on some object, Python will not complain but will do whatever we asked it to do if that is possible. This type of error is sometimes not easy to debug, so one has to be particularly careful.

Creating variables and/or assigning them values in Python is, therefore, straightforward[†]:

```
a = 1 # a is an integer
a = 1.3 # a is now a float
a = 'Hallo!' # a is a string
print a[3] # We can refer to a character within a string
#But we cannot change it -> a[3]='k' is forbidden !!. Strings are not mutable.
#In Python indices start from 0 for the first element. C-like
```

There are also three different groups of objects: lists, dictionaries and tuples:

[†] The *#* character is used in Python to comment the rest of the line.


```

a = [1,3,4,'Hallo'] # a is a list. Elements in a list do not need to be of the same
                    type
a[0] = 7 # This is the first element. Lists are mutable
a[-1] = 'Bye' # Negative indices start from the end. This is the last element
print a[1:-1] # This is [3,4]. This is called a slice
a = [[1,2],[3,4]] # Elements of list can also be lists. But a list of lists is not
                  an array!!
a = {'mass':3.2,'vel':3.2e5,3:4} # Dictionary. Unordered named list.
a['mass'] = 6.7 # Elements of dictionaries are referred by keyword, not by position
                Dictionaries can be changed. They are mutable
a = (1,3,5,'siesta') # A tuple. Similar to a list but non mutable
# a[0]=5 This is not permitted if a is a tuple.

```

6.2.1.2 Flow Control, Functions and Input/Output

Probably the most distinctive hallmark of Python is the way in which code blocks are indicated. In Python code blocks do not use braces, parenthesis or brackets[†] but are indicated by the level of indentation. That means that in Python, indentation is much more than just a nice way to show code structure, but it is the code structure itself. Although it may result strange at first, one gets used to it quite quickly. Although not mandatory, it is customary to use four spaces per indentation level. In general, it is not safe to mix tabs and spaces, so try to ALWAYS stick to four spaces per indentation level to avoid problems.

Among the different structures to organize program flow, we will present here only three: **for** and **while** loops and **if ... else** blocks. A for loop looks something like:

```

for i in range(2,5): # variable i runs from 2 to 5 (excluding 5)
    print i, "bottles of wine are too much"

```

Explicit loops are not particularly fast in Python, and this becomes more pronounced when loops are nested. This is the reason why some more efficient alternatives like comprehension lists, or libraries like Numpy to deal with arrays, are preferred performance-wise. Loops can also be made with a **while** and a condition check:

```

i=0
while( i < 10 ):
    print i
    i += 1

```

We can check for conditions with **if ... elif ... else** structures:

```

if (a[i] == 7): # Next block is executed if condition is fulfilled
    print a[i], "is equal to seven"
    a[i] += 1
elif (a[i] == 5): # If this condition is fulfilled then
    print a[i], "is five"
    a[i] -= 1
else: # If none of the above is fulfilled
    print a[i], "is not 5 or 7"

```

[†] Try from `__future__` import `braces` within Python and see what happens

There are some more control flow statements but we will not use them here. These flow control structures can be combined at will to produce the desired outcome.

Python has around 80 built-in functions that can be used straight away, like `abs()`, `help()`, `raw_input()` ... etc. But the standard library that comes with Python is plenty of many more functions, and endless libraries are available in Python. Functions are usually organized into modules that contain several related functions. And, for namespace sanity, modules and submodules are often organized into packages. Modules can be imported into a Python program with the **import** statement, and functions within that module can then be used in that program. For example, we can import and use functions in the `random` module from the standard library with:

```
import random # Here the module is imported
print random.random() # Print a float in [0.0,1.0). Look at the syntax module.func
```

The reason to refer to the function in this way, with the module name in front is to avoid namespace mixing. We could have another `random()` function in another module and we could use both while still preventing confusion. We can use an alias for the module with the `as` clause:

```
import random as r # Import the random module with alias r
print r.random()
```

Indeed, we could avoid writing at all the module name if we import the module with:

```
from random import * # Names in random imported to current namespace. BEWARE!!
print random()
```

But this is indeed a dangerous practice due to namespace mixing and potential confusion and it is therefore not advisable. We could even import just one or a few names/functions into the current namespace with:

```
from random import random # Import random() to current namespace. Somewhat safer
but still BEWARE!!
print random()
```

Of course, we can do much more than using already made functions, and we can create our own functions/modules. A function is created with the **def** clause:

```
def factorial( n ): # def function ( arg1, arg2, arg3, ... )
    if n < 1: # Base case. Notice the indentation of the whole function code
        return 1
    else:
        return n * factorial( n - 1 ) # Python functions can be recursive !!
```

Several functions can be defined on the same file as a module that can be imported like any other module.

A Python script can also have command line arguments. These can be accessed via the variable `argv` of module `sys`. We may have something like:

```
import sys
nargs=len(sys.argv)-1    # Number of arguments. argv[0] is the script name.
for i in sys.argv[1:]:   # Loop over arguments (excluding script name)
    print i
```

Of course, there are more complex ways to parse arguments, but this should be enough for our needs here.

Files in Python are objects which can be created with the `open()` function. The first argument of `open()` is the file name. A second argument in `open()` can be used to set if the file is open for reading ('r'), writing ('w'), appending ('a'), etc... If not set, the file is open for reading. We can read a line with `readline()`. Indeed, files are iterable objects, so we can read a file with:

```
for i in open(filename): # A file is iterable. Loop over lines.
    print i              # i is a string containing of a line in file, not a
                        number.
```

Or we can write to it with:

```
f=open(filename,'w') # BEWARE, opening in 'w' mode will erase the file contents.
f.write(string)      # Write string to file
f.write(str(float))  # Write a float. Convert to string first with str() !!
```

Again there are more sophisticated ways to deal with input/output, but this is the minimum required.

6.2.2 Numpy and Matplotlib

We have already said that the standart Python implementation is not particularly fast, and this weakness becomes patent in scientific computing, when calculations with large arrays and/or nested loops are needed. Fortunately, there is a library that eases this deficiencies and makes array manipulation very natural and fast. It is Numpy. Numpy can be imported the usual way, but in order to avoid namespace mixing, we will invariably import it with `import numpy as np` Arrays are numpy objects which can be created with numpy functions from scratch or from pre-existent python objects. Let's have a look at an example to see how natural it is to work with arrays within Numpy:

```
import numpy as np    # Import numpy module
z=np.zeros(5)         # z is an array of 5 elements filled with zeros
o=np.ones((3,3))      # o is a 3x3 array filled with ones
l=np.linspace(3,5,21)  # Array with 21 elements from 3 to 5 (included)
a=[1,2,3,4,5]         # a is a python list
a=np.array(a)         # Convert a list into an array
c=a+b                 # c,a and b are arrays.
c=3*a                 # Multiply every element by 3
c=a*b                 # Per element multiplication
```

```

c=np.exp(a)           # Apply function to every element
b=np.ones(3)
c=o+b                 # b is broadcasted to right size before addition
y,x=mgrid[0:3,0:3]    # x and y are 3x3 arrays with column and row values
r=np.sqrt((x-1)**2+(y-1)**2) # Array with distance to pixel 1,1

```

In Numpy, arrays are stored in row order like in C (unlike FORTRAN or IDL which store arrays in column order). This means that as we move linearly through memory in an array, the rightmost index changes the fastest. This is an important fact to be able to use the memory cache efficiently in many cases. We must remember that arrays do multiply on a per basis element, not like matrices and/or vectors. If we need that behaviour, we should use `matrix` objects. Arrays can be converted into matrices with the command `np.matrix(arr)` (where `arr` is assumed to be an array). We will not use matrices in this tutorial.

To end this section, we will briefly speak about how to make simple plots in python. It is indeed very easy thanks to the `matplotlib` library. Let's illustrate it with a simple example:

```

import numpy as np
import matplotlib.pyplot as plt # Import the pyplot module from matplotlib
x=np.linspace(-5,5,101)        # Array of 101 elements from -5 to 5
y=np.exp(-x**2/2.)             # Gaussian function
plt.plot(x,y)                  # Plot x vx y
plt.plot(x,y,'g')              # Same using green + symbols
plt.show()                     # Show the plot

```

The plot is shown in an interactive window that allows some interaction, including zooming in and out, and even saving the plot into a file in different formats (gif, eps, pdf, etc). The plot function in `matplotlib` is plenty of options which we will not cover here. The interested reader can have a look at the official `matplotlib` documentation at <http://matplotlib.org/contents.html>. We can also make subplots in a page with the `subplot` command. The line `plt.subplot(nrows,ncols,nplot)` creates `nrows×ncols` subplots with `nplot` being the active plotting axes. If `nrows`, `ncols` and `nplot` are all smaller than 10, we can skip the commas. Thus `plt.subplot(221)` sets the upper left corner as the active plotting axes in a group of four subplots set in two rows and two columns. We can also plot images with the `plt.imshow(array, vmin=min, vmax=max)` command. We recall the reader to use the `plt.show()` command to show all the plots unless interactive mode is being used.

6.3 Inverse Ray Shooting Basics

The gravitational lens equation can be expressed in dimensionless units as (cf. Schneider, Ehlers & Falco, 1999):

$$\mathbf{y} = \mathbf{x} - \boldsymbol{\alpha}(\mathbf{x}) \quad (6.1)$$

where \mathbf{y} and \mathbf{x} are the (2D vector) coordinates at the source and lens plane respectively, and $\boldsymbol{\alpha}(\mathbf{x})$ is (also a 2D vector) called the scaled or reduced deflection angle. Solving this equation means to be able to invert it and to obtain the position of the image/s \mathbf{x}_i corresponding to a source point \mathbf{y} . This equation can be interpreted as describing the trajectory of a ray of light going *backwards* from location \mathbf{x} at the lens plane, to location \mathbf{y} at the source plane. For this reason it is sometimes referred to as the ray-trace equation of gravitational lensing. As light rays are invariant under time reversal, tracing them forward (as real photons are travelling) or backwards makes no difference from the mathematical point of view. We will assume here the *thin lens* approximation in which the deflection of light takes place mostly in a very narrow region of space which is very small compared to the distances between lens and source (D_{ls}), lens and observer (D_l) and, of course, source and observer (D_s).

This equation is, in general, highly non-linear due to the dependence of $\boldsymbol{\alpha}$ on the position on the lens plane. Indeed, it can only be solved analytically in a few simple cases. As we know already, in some circumstances, this equation can even produce multiple images of a source (so that there are several values of \mathbf{x} that are a solution to eq. 6.1 for a given value of \mathbf{y} .) When the lens and/or source are complex, equation 6.1 can only be inverted by numerical methods. Here, our ultimate goal is to be able to calculate magnification maps for quasar microlensing, but in the meantime, we will also apply the procedure to simpler cases.

As we have indicated above, the interpretation of the lens equation as a ray-trace equation is very natural. Therefore the technique of shooting rays from the observer plane backwards, deflecting them at the lens plane, and tracing them to the source plane is a very simple but effective way to solve the lens equation for a given region of the image plane. This technique is called Inverse (or Backwards) Ray Shooting (IRS) and was introduced by Schneider & Weiss (1986, 1987) and Kayser, Refsdal & Stabell (1986) to calculate magnification maps. Here, we will introduce this technique from the most simple case of a point lens (even the trivial case of no lens will be explored to set up the main structure of the code) to the complex case of quasar microlensing.

The underlying idea is, therefore, to study a certain region of the image/observer plane by tracing backwards the trajectories of a set of light rays and looking what is in the source plane at the position of the ray hit. The trajectory between observer and lens plane is a trivial straight line, as there is no deflection. At each point in the lens plane, the rays are deflected an amount given by the scaled deflection angle. From the lens plane to the source plane, the trajectory is again a straight line and therefore there is no further change in the coordinates. The convergence/divergence of light rays between lens and source planes is already implicitly taken into account by the different angular scales used at the lens and source plane in eq. 6.1.

The algorithm goes as follows:

- (i) Divide the source and image plane regions under study into cells/pixels. Let $2x_l$ and $2y_l$ be the size of the regions at the lens and source planes that we are going to consider. We divide both planes into n_x and n_y cells (or pixels) respectively. Therefore, the size of a cell is $x_s = 2x_l/(n_x - 1)$ and $y_s = 2y_l/(n_y - 1)$ in the lens and source plane respectively[†]. The input parameters are therefore the size and resolution of the regions at the source and image plane.
- (ii) Set the surface brightness at the source plane according to source model. We should know the surface brightness at the source plane. To start with something easy, let's define a gaussian circular source of sigma *rad* located at pixel (x_1, y_1) . Let's first write a function `gcirc(ny,rad,x1,y1)` that returns an array of $n_y \times n_y$ with such a source. We can create a module `source.py` like this:

```
import numpy as np
def gcirc(ny,rad,x1=0.0,y1=0.0): # Default position is 0,0
    x,y=np.mgrid[0:ny,0:ny]
    r2=(x-x1-ny/2)**2+(y-y1-ny/2)**2
    a=np.exp(-r2*0.5/rad**2)
    return a/a.sum()
```

The input parameters here are the location, size and structure of the source.

- (iii) Trace rays back by deflecting them at the lens plane according to the lens equation 6.1. Now, we just need to trace back rays from the center of each of our cells at the lens plane and see where these rays hit at the source plane.

[†] We are implicitly assuming here that $\pm x_l$ and $\pm y_l$ are indeed the coordinates of the center of the extreme cells and not the coordinates of the cell's edges. The real size of the regions in the lens and source plane are indeed $2x_l + x_s$ and $2y_l + y_s$. The reason to do it like this is that when pixels are numbered with i from 0 to $n_x - 1$, coordinates of cell centers defined as $x_1 = -x_l + i * x_s$ run from $-x_l$ to x_l . Same for coordinates in the source plane.

- (iv) Find the cell/pixel at the source plane where the ray hits. If the ray hits at a spot in the source plane where there is a bright source, then the value of that cell in the image/lens plane is set equal to the surface brightness of the source at that position. Even if there is a single source, the bending of light rays by the effect of the lens may end up producing several images of it. This would happen if rays starting from different positions at the image plane end up hitting the same spot of the source plane.

The last two steps in this algorithm involve conversion from pixels to coordinates (at the lens plane) and back to pixels (at the source plane after deflection). To start with the easiest case, let's just explore the case with no lens and therefore no deflection of the light rays. The reader may think that there is no interest in such a case, but this way we will focus only on the general setup of the program, independently of the lensing calculations. The input parameters are the location and structure (mass, shape, model) of the lens (which in this first example will be set to a no lens model).

The code should look something like this:

irs0.py

```
import numpy as np
import matplotlib.pyplot as plt
import source as s

nx=401 # Number of pixels in image plane
ny=401 # Number of pixels in source plane
xl=2.  # Half size of image plane covered (in "Einstein" radii)
yl=2.  # Half size of source plane covered (in Einstein radii)

xs=2.*xl/(nx-1) # pixel size on the image map
ys=2.*yl/(ny-1) # pixel size on the source map

#Source parameters
xpos=0.0          # Source position. X coordinate
ypos=1.0          # Source position. Y coordinate
rad=0.1           # Radius of source
ipos=int(round(xpos/ys)) # Convert source parameters to pixels
jpos=int(round(-ypos/ys))
rpix=int(round(rad/ys))
a=s.gcirc(ny, rpix, jpos, ipos) # This is a circular gaussian source
b=np.zeros((nx,nx))           # This is the image plane

#This is the main loop over pixles at the image plane
for j1 in range(nx):
    for j2 in range(nx):
        x1=-xl+j2*xs # Convert pix to coords on image.
        x2=-xl+j1*xs
        y1=x1-0.0 # Deflect X coordinate
        y2=x2-0.0 # Deflect Y coordinate
        i2=int(round((y1+yl)/ys)) # Convert coordinates to pixels
```

```

i1=int(round((y2+yl)/ys))
# If deflected ray hits a pixel within source then set image
# to brightness on that pixel
if ((i1 >= 0) and (i1 < ny) and (i2 >= 0) and (i2 < ny)):
    b[j1,j2]=a[i1,i2]
# Plot stuff
plt.subplot(121)
plt.imshow(a,extent=(-yl,yl,-yl,yl))
plt.subplot(122)
plt.imshow(b,extent=(-xl,xl,-xl,xl))
plt.show()

```

It is an interesting exercise to play around a bit with the values of the input parameters, in particular the size of the source and of the source and image regions and the resolution in these planes to see what is the effect. For example, it makes no much sense having a resolution at the image plane much higher than at the source plane. We can also see that if the source is small and the resolution at the image plane is small, we may miss the image or have a very unrealistic image. Thus, source and image plane should both be conveniently sampled.

6.3.1 Image/s for a simple lens: The Point Source Lens

The next step in our path towards producing magnification maps is the study of the case of a point source lens of mass m_l located at $\mathbf{x}_d = (x_d, y_d)$. In this case, we should set the deflection of the coordinates as $\alpha = m_l(\mathbf{x} - \mathbf{x}_d)/|\mathbf{x} - \mathbf{x}_d|^2$ where $\mathbf{x} = (x_1, x_2)$ are the coordinates of the ray at the source plane. As we did with the source, we can create a function that calculates this deflection for such a ray in a module called `lens.py`. The function will look something like[†]:

```

def Point(x1,x2,x1l,x2l,ml): # Point lens of mass ml at x1l,x2l
    x1ml=(x1-x1l) # Distance along x axis of ray to lens position
    x2ml=(x2-x2l) # Distance along y axis of ray to lens position
    d=x1ml*x1ml+x2ml*x2ml+1.0e-12 # Distance between ray and lens squared
    y1=x1-ml*(x1-x1l)/d # Lens equation for x coordinate
    y2=x2-ml*(x2-x2l)/d # Lens equation for y coordinate
    return (y1,y2)

```

We see that the function returns a tuple (y_1, y_2) with the coordinates of the deflected ray at the source plane. Then, after importing the module with `import lens as l` at the beginning of our previous program, we can set the lens parameters x_d, y_d, m_l , and we can then deflect rays by calling this function as `y1,y2=l.Point(x1,x2,xd,yd,ml)`. If we locate our source at point (0,1) and the

[†] The 10^{-12} added to the distance squared is a very inelegant yet simple way to avoid division by zero when lens and source are aligned. Sorry about that!

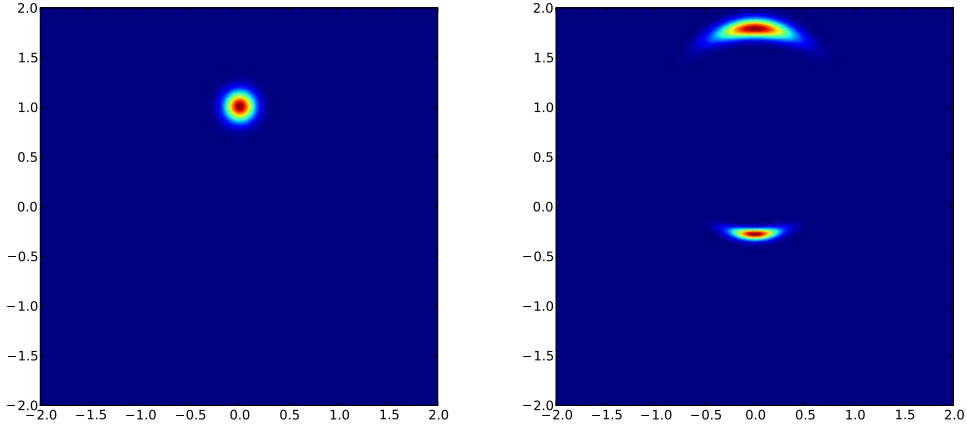


Fig. 6.1. Source (left) and Image (right) planes. Point lens of mass=1 located at (0,0.5). Source is a circular gaussian source at $x=0$, $y=1$.

lens of mass $ml = 1$ at position (0,0.5), the result is shown in fig. 6.1 We can see that two images appear with a separation $d \approx 2$. Indeed, that is exactly the theoretical result. If source and lens are aligned, the image is a nice Einstein ring of radius $\theta_E = 1$. The reader is encouraged to play around a bit with the input parameters, in particular with the relative positions of lens and source. The magnification can be calculated as the ratio of the flux in both planes (taking into account the pixel size on each plane.)[‡]

6.4 Playing around with lenses and sources

Once we are familiar with the simplest case, it is time to explore other popular lens models. We will explore some well known models which are described in chapter 8 of Schneider, Ehlers & Falco (1999). The procedure is very similar to what we have done with the point source lens, by creating a function that calculates the deflection of a given ray at location (x_1, x_2) in the lens plane. As an example, we will explore three well known cases:

- (i) **The binary lens:** It is a straightforward extension of the previous case by just adding a new lens of mass m_{2l} at position (x_{2d}, y_{2d}) . The deflection angle for a ray at location $\mathbf{x} = (x_1, x_2)$ is given by:

$$\boldsymbol{\alpha} = m_{1l} \frac{(\mathbf{x} - \mathbf{x}_{1d})}{|\mathbf{x} - \mathbf{x}_{1d}|^2} + m_{2l} \frac{(\mathbf{x} - \mathbf{x}_{2d})}{|\mathbf{x} - \mathbf{x}_{2d}|^2} \quad (6.2)$$

[‡] Recall that this is a magnification *averaged* over the whole source. This should not be confused with the value of the magnification of a point source lens at the position of the source.

The binary lens is a very well studied case (Scheider & Weiss, 1987) and widely applied, for example, in modelling microlensing events. The two relevant parameters are the mass ratio of the lenses and their separation.

- (ii) **The Chang-Refsdal lens:** It is a simple extension of the point lens that include the effect of a smooth background gravitational field by including a quadrupolar term in the deflection angle caused by the shear due to the background gravitational field of a nearby source. It was introduced by Chang & Refsdal (1979, 1984) to address the lensing of a star in the background potential of a galaxy. The effect of the background potential is included via its convergence κ and shear γ so that the deflection angle (in the coordinate system in which one of the axis runs along the direction of the shear) is:

$$\boldsymbol{\alpha} = \begin{pmatrix} \kappa + \gamma & 0 \\ 0 & \kappa - \gamma \end{pmatrix} \mathbf{x} + m_l \frac{(\mathbf{x} - \mathbf{x}_d)}{|\mathbf{x} - \mathbf{x}_d|^2} \quad (6.3)$$

- (iii) **The Singular Isothermal Sphere (SIS):** The SIS is a simple model which is often used to mimic the lensing properties of galaxies and/or cluster of galaxies. It is characterized by its Einstein Radius $\theta_E = 4\pi \left(\frac{\sigma_v^2}{c^2} \right) \frac{D_{ls}}{D_s}$ (where σ_v is the one-dimensional velocity dispersion). The deflection angle for this lens system is given by:

$$\boldsymbol{\alpha} = \theta_E \frac{(\mathbf{x} - \mathbf{x}_d)}{|\mathbf{x} - \mathbf{x}_d|} \quad (6.4)$$

The singularity at the origin can be removed by introducing a finite core that changes the lensing properties very close to the lens position. That model is called the Non Singular Isothermal Sphere.

Including these lenses in our `lens.py` module is quite straightforward following the example of the point source lens. The module would look something like:

`lens.py`

```
import numpy as np

def Point(x1,x2,x1l,x2l,ml):
    x1ml=(x1-x1l)
    x2ml=(x2-x2l)
    d=x1ml*x1ml+x2ml*x2ml+1.0e-12 # Add a tiny number to avoid division by zero
    y1=x1-ml*(x1-x1l)/d          # Lens equation for x coordinate
    y2=x2-ml*(x2-x2l)/d
    return (y1,y2)
```

```

def TwoPoints(x1,x2,x1l1,x2l1,x1l2,x2l2,m1,m2):
    x1ml1=(x1-x1l1)
    x2ml1=(x2-x2l1)
    x1ml2=(x1-x1l2)
    x2ml2=(x2-x2l2)
    d1=x1ml1*x1ml1+x2ml1*x2ml1+1.0e-12 # Add a tiny number to avoid division by
    zero
    d2=x1ml2*x1ml2+x2ml2*x2ml2+1.0e-12
    y1=x1-m1*(x1-x1l1)/d1-m2*(x1-x1l2)/d2 # Lens equation for x coordinate
    y2=x2-m1*(x2-x2l1)/d1-m2*(x2-x2l2)/d2
    return (y1,y2)

def ChangRefsdal(x1,x2,x1l,x2l,m,k,g):
    x1ml=(x1-x1l)
    x2ml=(x2-x2l)
    d=x1ml*x1ml+x2ml*x2ml+1.0e-12 # Add a tiny number to avoid division by zero
    y1=x1*(1.0-k-g)-m*(x1-x1l)/d # Lens equation for x coordinate
    y2=x2*(1.0-k+g)-m*(x2-x2l)/d
    return (y1,y2)

def SIS(x1,x2, x1l,x2l,k):
    x1ml=(x1-x1l)
    x2ml=(x2-x2l)
    d=np.sqrt(x1ml*x1ml+x2ml*x2ml+1.0e-12) # Add a tiny number to avoid division by
    zero
    y1=x1-k*(x1-x1l)/d # Lens equation for x coordinate
    y2=x2-k*(x2-x2l)/d
    return (y1,y2)

```

Any other lens models (like more than two point lenses, Singular & Non-Singular isothermal spheres or ellipsoids with or without external shear, etc) can be included in a similar way. We can now test our former IRS code by playing with these new lenses. For example, we can try to reproduce the configurations shown in Schneider & Weiss (1986). Figure 6.2 shows a situation similar to that of Fig 6b in Schneider & Weiss (1986).

The reader is encouraged to try these lens models or any other. Change the input parameters for the lens and/or source and try to reproduce the known theoretical results about number of images, parity, etc. It is also instructive to also use different sources, like the image of a real galaxy, or a field of galaxies. This can be read from a fits file using the `getdata` function from the `pyfits` library of python. We can include the following function in our `source.py` module to read a fits image as source:

```

from pyfits import getdata
def fitsim(filename):
    a=getdata(filename) # Read the file
    if (len(a.shape) > 2): a=a[0] # Take first plane if there are many.
    return (1.0*a)/a.sum() # Return the normalized image

```

Then, the source can be read from our IRS code with:

```

a=s.fitsim('Edgeon2.fits') # Read file 'Edgeon2.fits'

```

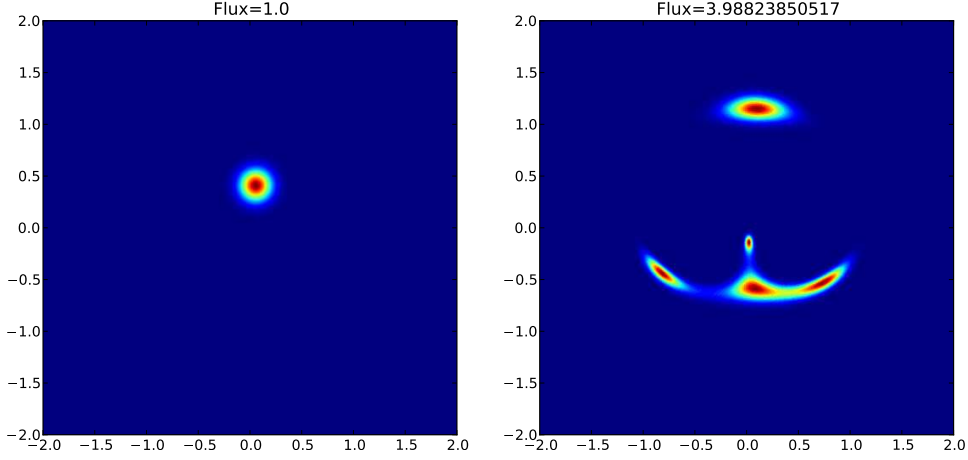


Fig. 6.2. Source (left) and Image (right) planes. Binary point lens of mass=0.5 located at (0.5,0.0) and (-0.5,0.0). Source is a circular gaussian source at $x=0.05$, $y=0.4$.

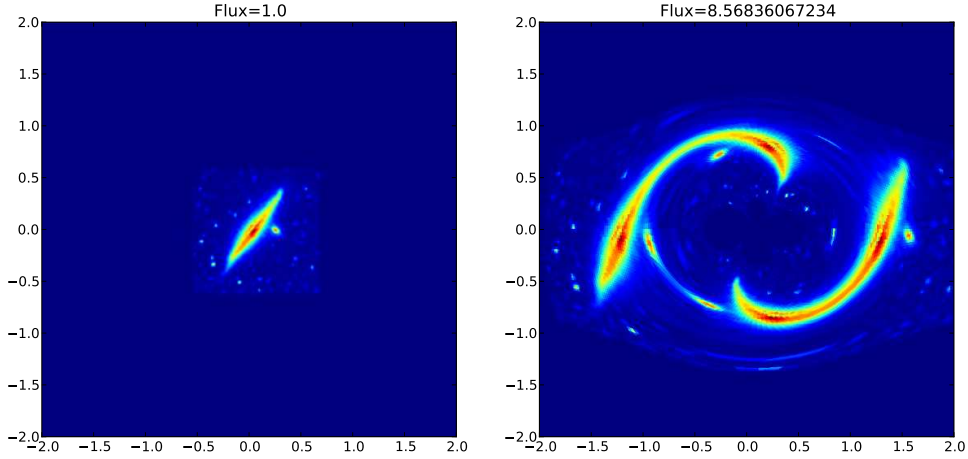


Fig. 6.3. Source (left) and Image (right) planes. SIS lens of $\theta_E = 1$ and $\gamma = 0.2$ located at (0.1,0).

```
ny=a[0].size
```

```
# Number of pix in souce plane set to image size
```

An example of the result of our code for a edge-on galaxy lensed by a SIS with some shear is shown in Figure 6.3.

This python code is useful and may allow to get some insight into some lens systems, but we have to accept that it is not very efficient. There are two nested loops (over rows and columns of the image plane) and, if there are many lenses, there would still be a third one over lenses. Python

is, in general, not very efficient in running loops, and consequently, nested loops are even worse. As the image plane becomes larger and/or we include more lenses, this code becomes very inefficient and, consequently, slow. Fortunately, improving its performance is both, easy and elegant, by using numpy arrays. Instead of running over individual pixels, we can deflect the whole bunch of rays at once by operating on an entire array of coordinates. This way, the nested loop will be dealt with internally by numpy which is much more efficient (as numpy is already compiled to machine code). And best news is that, as we were careful enough in writing our functions for the deflection of the different lens systems to use numpy functions when necessary, these functions, without any modification, are still as useful when called with full arrays of coordinates as they were with a single ray. We then just need to make some manipulation to create an array with the column and row numbers of the pixels, and to remove the loop over pixels by operating on full numpy arrays. The code will now look something like this:

irs1.py

```
import numpy as np
import matplotlib.pyplot as plt
import source as s
import lens as l
nx=801 # Number of pixels in image plane
ny=401 # Number of pixels in source plane
xl=2. # Size of image plane covered (in "Einstein" radii)
yl=2. # Size of source plane covered (in Einstein radii)
#Lens parameters
xlens=0.0
ylens=0.0
mlens=1.0
xs=2.*xl/(nx-1) # pixel size on the image map
ys=2.*yl/(ny-1) # pixel size on the source map
#Source parameters
xpos=0.0
ypos=0.0
rad=0.10
ipos=int(round(xpos/ys)) # Convert source parameters to pixels
jpos=int(round(-ypos/ys))
rpix=int(round(rad/ys))
a=s.gcirc(ny, rpix, jpos, ipos) # This is the source plane
b=np.zeros((nx,nx)) # This is the image plane
#In this version, the main loop is implicit
#We use operations on numpy arrays instead which is faster.
j1,j2=np.mgrid[0:nx,0:nx]
x1=-xl+j2*xs # Pix to coord on image x
x2=-xl+j1*xs # Pix to coord on image y
y1,y2=l.SIS(x1,x2, xlens+0.1,ylens,1.2) # This line calculates the deflection
i2=np.round((y1+yl)/ys)
i1=np.round((y2+yl)/ys)
# If deflected ray hits a pixel within source then set image
# to brightness on that pixel
```

```

ind= (i1 >= 0) & (i1 < ny) & (i2 >= 0) & (i2 < ny)  #Now this is an array
                                                    #which is True if the ray
                                                    #hits the source plane.

i1n=i1[ind]
i2n=i2[ind]
j1n=j1[ind]
j2n=j2[ind]
for i in xrange(np.size(i1n)):  # Loop over pixels that hit the source plane
    b[j1n[i],j2n[i]]=a[i1n[i],i2n[i]]
# Plot stuff including Fluxes in both planes
fig=plt.figure(1)
ax=plt.subplot(121)
ax.imshow(a,extent=(-yl,yl,-yl,yl))
fa=np.sum(a)  #Flux on source plane
ax.set_title('Flux='+str(fa))  #Set title for subplot 1
ax=plt.subplot(122)
ax.imshow(b,extent=(-xl,xl,-xl,xl))
fb=np.sum(b)*(xs**2)/(ys**2)  # Flux on image pl. (taking into account pix size)
ax.set_title('Flux='+str(fb))  # Set title for subplot 2
plt.show()

```

6.5 Magnification Maps and Light Curves

Up to now we have been able to produce images of many different types of sources which are gravitationally lensed by a broad group of lens models. Even if this is instructive and may have some usefulness, we would like to go further and to be able to produce magnification maps for our lens systems. A magnification map is a map that shows the amount of the magnification produced by a certain lens system in a region of the source plane. If, as is usually the case, lens(es) and source move with respect to each other at a certain velocity, the source suffers different magnifications at different times, which are observed as variations in the brightness of the source. A curve of the brightness of a source versus time is called a *light curve*, and contains plenty of information on the lens and/or source systems. Magnification maps are therefore needed to be able to interpret this kind of observations and to compare them with different possible models. They allow to determine many properties of the lens system (mass, structure) and/or of the lens (size, etc.) In order to calculate magnification maps, we will make use of the fact that gravitational lensing does not change the surface brightness of the images, and therefore the magnification is just the ratio between the subtended solid angles of image(s) and source, which is given by the inverse of the determinant of the A matrix (cf. Schneider, Ehlers & Falco, 1999, sect

5.2):

$$\mu(\mathbf{x}) = \frac{d\omega}{d\omega^*} = \frac{1}{\det A(\mathbf{x})} \quad (6.5)$$

where

$$A(\mathbf{x}) = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbb{1} - \frac{\partial \boldsymbol{\alpha}}{\partial \mathbf{x}} \quad (6.6)$$

As we are using angular coordinates in the lens and source plane, the solid angle is just an area on those planes. To estimate that ratio of areas we make use of our Inverse Ray Shooting technique. We shot a certain (fixed) number of rays per unit area (e.g. per pixel) at the image plane (N_r). Now, we collect those rays in cells of the same area at the image plane. If we collect $N_s(x_1, x_2)$ rays in cell at coordinates $\mathbf{x} = (x_1, x_2)$, then the magnification can be written as:

$$\mu(\mathbf{x}) = \frac{dS}{dS^*} = N_s(\mathbf{x})/N_r \quad (6.7)$$

We must take care with the units in calculating the right size of the shooting and collecting cells to fulfill the second equality of equation 6.7. Otherwise we could miss a normalization constant, although this constant can be easily calculated (by checking the unlensed case). Let's see how the method works with a simple example. If we shot rays at the center of every cell in the image plane and plot the location of the hits at the source plane in the case of a binary lens, we would find something like what is shown in Figure 6.4 (in which rows of rays are marked with the same colour to ease visualization of shape deformation).

We can see that, not only the squares originally formed by contiguous rays get *deformed* into irregular quadrilateral shapes, but very weird things happen in some regions where a large ammount of rays bunch together forming funny figures. Those are regions of high magnification according to equation 6.7. Indeed, along those regions of high concentration of rays, the lens transformation is no longer invertible (as those locations correspond to caustics which are the transformed locations of the critical curves where $\det A(\mathbf{x}) = 0$) and squares are transformed into crossed quadrilaterals. Therefore, a couple of important lessons can be learned from this example:

- Caustics, which are regions of very high magnification (virtually infinite), have a high concentration of rays and therefore very good statistics for calculating the magnification as described in equation 6.7 via Inverse Ray Shooting. Still, critical cells at the lens plane and caustics at the source plane are the places with the most peculiar topological properties of the

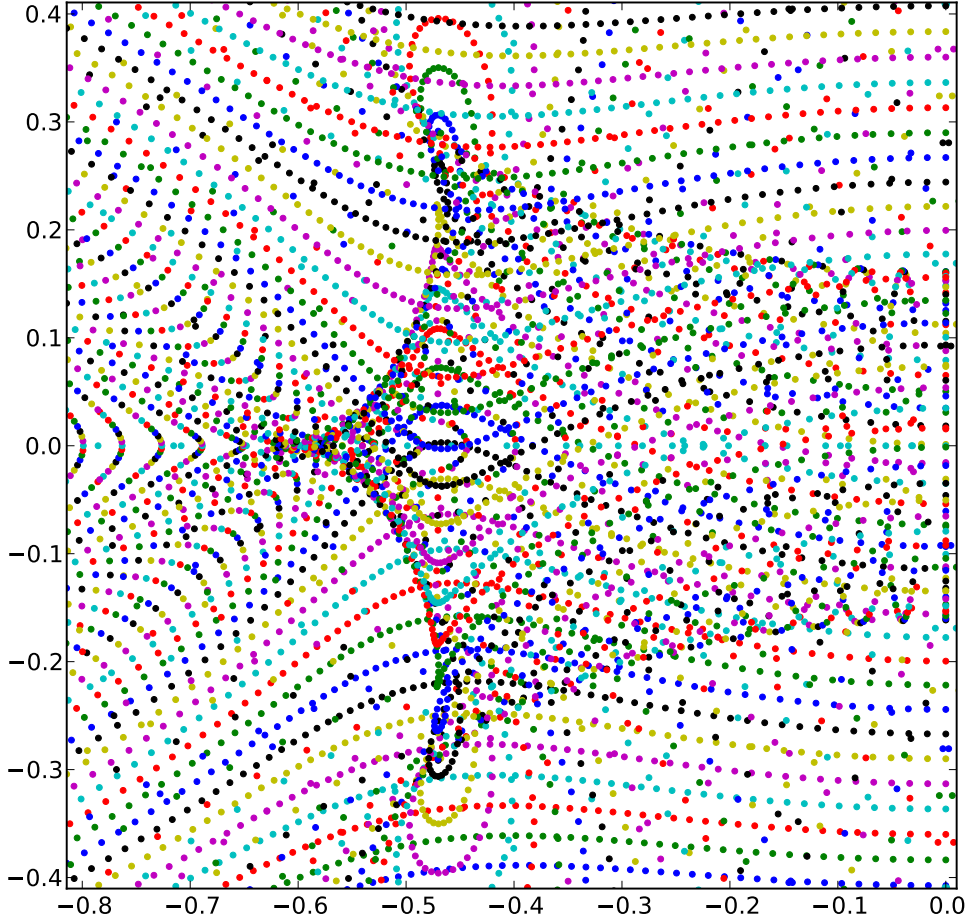


Fig. 6.4. Ray hits in the source plane for a regular grid of rays at the lens plane. One ray is shot at the center of every pixel of the lens plane. The lens is a binary lens of equal masses $m = 0.5$ with a separation of 1.5

lens transformation, and particular care must be taken with them for certain purposes.

- Low magnification regions receive few ray hits and, consequently, have poor statistics in calculating the magnification. The need for a very fine grid of rays to calculate accurate magnification maps comes mainly from the need to have good statistics in these low magnification regions. We will see that this is an important fact from which we will be able to profit soon.

With this experience, we have enough tools to write an IRS code to

calculate magnification maps for a simple lens model. The outline of the code could be this:

- (i) Prepare a uniform grid of rays at the image plane to be shot backwards towards the source plane. The denser the grid, the better the statistics and the more accurate the final magnification map. Obviously, a compromise must be set between accuracy and computation time. The relevant parameter here (apart from the already mentioned size of the image region and the number/size of pixels) is then the number of rays shot per (unlensed) pixel at the image plane: N_r . Some IRS codes prefer to shoot the rays at random (still keeping an average of N_r per pixel) instead of on a regular grid. By using a regular grid, we get an accuracy that improves as $N_r^{3/4}$ (cf. Kayser, Refsdal & Stabell, 1986, Mediavilla et al. 2001) which is better than what would be expected by Poissonian statistics.
- (ii) Loop over the grid of rays and deflect them according to the lens equation. As the lens equation is linear in the deflection angle, if there are several deflectors, loop over the deflectors and add the deflection produced by each of them. Calculate the coordinates of the ray hits at the source plane. It is worth noting here that this step is computationally very demanding, as these loops may be huge if there are many rays to shoot or many deflectors. To get an order of magnitude, let's do some rough calculations. The number of steps in our loops will be:

$$N_{steps} = N_x^2 \times N_r \times N_* \quad (6.8)$$

where N_x is the number of pixels on a side of our (for simplicity, square) shooting region (which is determined by the size of our magnification map as we will see below), N_r is the number of rays per unlensed pixel, and N_* is the number of deflectors. A standard magnification map usually needs around 100 rays per pixel, and if we create a map of 1000×1000 pixels, we are shooting $\sim 10^8$ rays. If we are calculating a microlensing map with a thousand stars as deflectors, we have to calculate 10^{11} deflections !!! This is the number of steps in our nested loop which is indeed an important figure. No wonder the main efforts aimed at improving performance of an IRS algorithm are devoted to reduce this number as much as possible. We shall come back to this issue later on.

A last comment is needed here with respect to our Python implementation for this nested loop. If we write explicit Python loops,

which we know are not very efficient, this calculation would take too long. Again, we can speed calculations up by a careful use of numpy arrays. We could be tempted to shot again the whole bunch of rays at once and do the calculations in one single step while keeping only the loop over deflectors. But as we have already seen, the number of rays in a magnification map is usually pretty high ($\approx 10^8$), with a very high risk of running out of memory in most computers if we proceed this way. We therefore have to lower a bit our expectations and we are forced to take a compromise. A pretty good one can be achieved by throwing a whole row (or column) of rays at once[†]. We shall use this implementation in our Python code below.

- (iii) Set an array of size $N_y \times N_y$ as our magnification map. Originally this array's elements are all set to zero. Now, for each ray shot, we calculate on which pixel at the source plane the ray hits (if at all), and we add 1 to the value at that pixel. After accounting for all ray hits, we divide this array by the number of rays shot at each unlensed pixel at the image plane. The resulting array is the magnification map we were looking for. The size and resolution (pixel size) of this map is determined by the smallest and largest sizes of the sources we are interested in. The pixel size should be smaller than the smallest source (which should span a few pixels). Large maps are usually needed to deal with large sources and/or to have good statistics. Finally, be aware that the size of this map N_y , will also determine the size of the shooting region. We will comment a bit on this below.

The resulting code would be something like this:

magmap.py

```
import numpy as np
import lens as l
import matplotlib.pyplot as plt

ny=401
yl=2.

b=np.zeros((ny,ny))
raypix=15. # This is the number of rays per pixel in absence of lensing.
sqrpix=np.sqrt(raypix) # Rays per pixel square root (rays/pix in one dir)
sqrinpix=np.sqrt(1./raypix)
ys=2.*yl/(ny-1) # Pixel size on source plane
xs=ys/sqrpix # Side of the square area transported back by a ray.
xl=2.*yl # Size of the shooting region at the image plane
```

[†] A whole row of pixels which is composed of $\sqrt{N_r}$ rows of rays each is also a good choice, although things may get somewhat tricky if N_r is not a square number

```

# BEWARE. This may need to be larger for certain lens models !!!!
nx=np.round(2*xl/xs)+1 # Number of rays on a column/row at the image plane
yr=np.arange(0,nx)    # This is an array with pixels on y direction
y,x=np.mgrid[0.0:1.0,0:nx] # Grid with pixel coordinates for a row at the image
perc0=5.               # Percentage step for printing progress
perc=5.               # Initial value for perc
for i in yr:           # Loop over rows or rays
    if ((i*100/nx)>=perc): # Check if we have already completed perc.
        perc=perc+perc0   # Increase perc.
        print round(i*100/nx),"% " # Print progress
    x1=-xl+y*xs # Convert pixels to coordinates in the image plane
    x2=-xl+x*xs
    y1,y2=l.TwoPoints(x1,x2,-0.75,0.,0.75,0.,0.5,2.5) # Deflect rays.
    # We can set another lens model here easily.
    i1=(y1+y2)/ys # Convert coords to pixels at the source plane
    i2=(y2+y1)/ys
    i1=np.round(i1) # Make pixel coords integer numbers
    i2=np.round(i2)
    ind=(i1>=0) & (i1<ny) & (i2>=0) & (i2<ny) # Indices of rays falling into our
    source plane
    i1n=i1[ind] # Coordinates of pixels hitting our source plane
    i2n=i2[ind]
    for i in xrange(np.size(i1n)): # Loop over hits "on target"
        b[i2n[i],i1n[i]]+=1 # Increase magnification at those pixels
    y=y+1.0 # Increase the y coordinate of the pixel/rays
b=b/raypix # Normalize magnification with N_r
print np.mean(b) # Print mean magnification
plt.imshow(b,vmin=0,vmax=15) # Show image
plt.show()

```

The result of running this code is shown in Figure 6.5.

One aspect that must be carefully considered is the choice of the size shooting region at the image plane. The shooting region must be large enough as to contain most (all would be desirable but not always possible) rays hitting the magnification map. But it should be small enough to avoid wasting rays that never hit the map. Therefore, how big the shooting region must be depends on the particular lens model. If the shooting region is too small, the map will present *discontinuities* corresponding to regions where there are no ray hits (because there are not rays shot at the appropriate locations in the image plane). The effect of a small shooting region is shown in Figure 6.6 with the same values for x_l and y_l than Figure 6.5 but with a Cheng-Refsdal lens of $M = 0.5$ on top of a gravitational potential with $\kappa = 0.15$ and $\gamma = 0.5$.

We can see the vertical dark stripes at the edges of the map originated by lack of ray hits. Indeed, the shear concentrates the rays in the horizontal direction and expands them in the vertical direction (This is the case for backward rays. It is the opposite for real photons). As a result, to fill the full map we should take rays further appart along the horizontal to fill

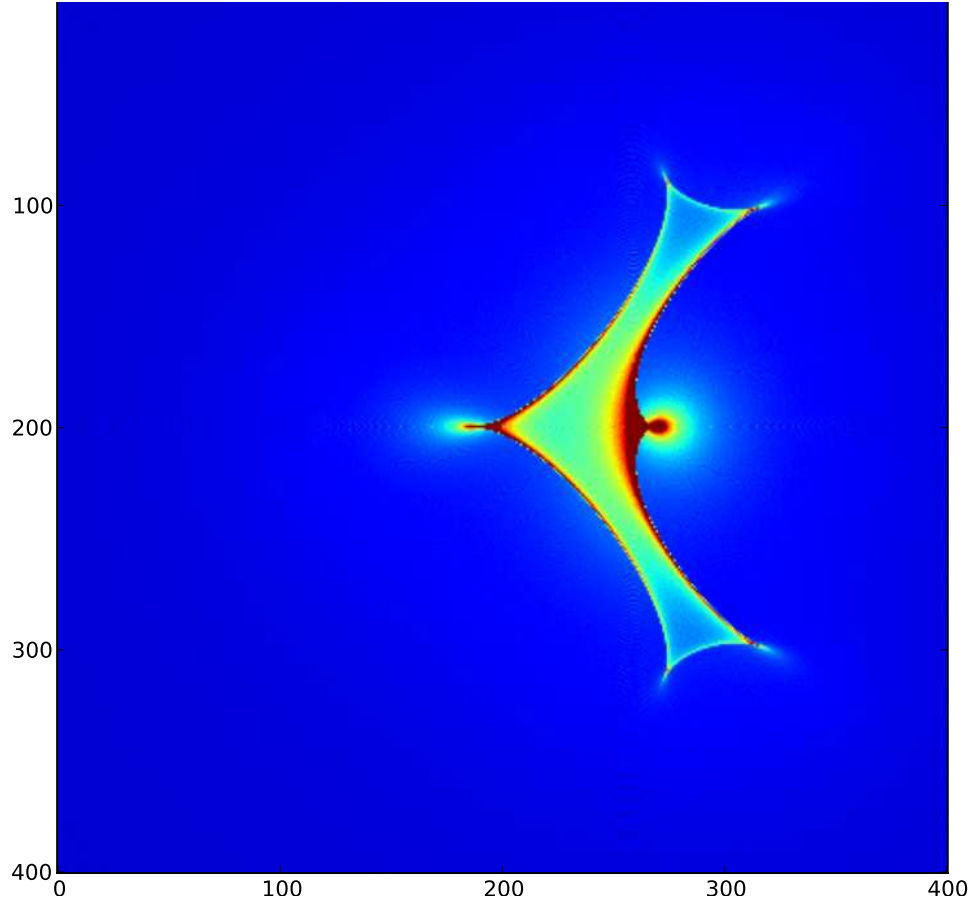


Fig. 6.5. Magnification map produced with the code above. Lens is a binary with a masses 0.5 and 2.5 located at $(-0.75, 0.0)$ and $(0.75, 0.0)$.

our magnification map. A factor $1/(1 - \gamma)$ in the horizontal direction is usually enough to account for this. In this case, the missing rays show off very clearly, but for some lens systems it may not be so obvious. We must therefore always choose a region large enough to prevent flux losses in our calculations. If the shooting region is too large we are just wasting time by shooting rays which are never going to be collected. In case of doubt, be generous. In general, the lemma to apply is: It is better to waste some rays than to lose flux.

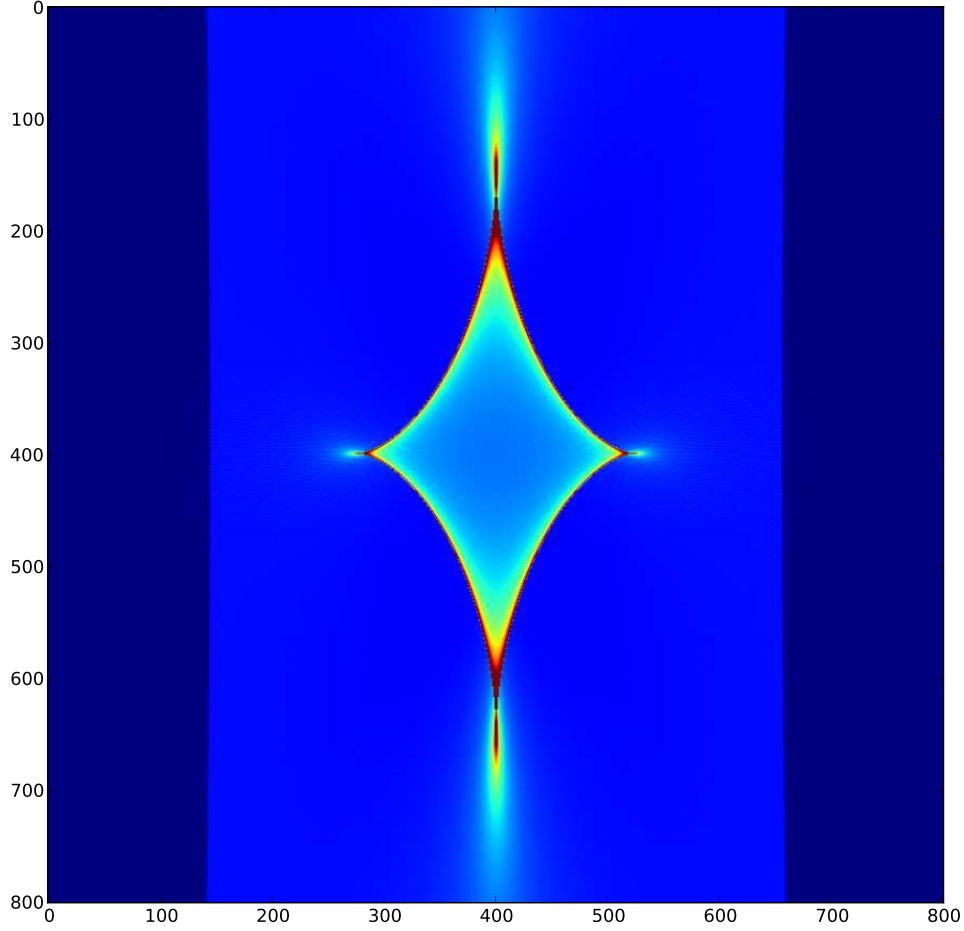


Fig. 6.6. Magnification map produced with the code above. Lens is a Chang-Refsdal Lens with a mass 0.5 and a background potential characterized by $\kappa = 0.15$ and $\gamma = 0.5$.

6.5.1 Case Study: Light curves for the Binary Lens

Magnification maps are, of course, not observable quantities. The source sits at a given location at the source plane and is affected by the magnification at that particular location[†]. But the lens-source system is rarely static, and therefore, the magnification with which we see the source changes with time depending on the relative motions of source and lens. The timescale for the change is $t_E = R_E/v_\perp$ and it is called lensing time or Einstein time (where R_E is the Einstein radius and v_\perp is the transverse velocity of the lens

[†] In fact, if the source has a finite size covering several pixels in our magnification map, it sees an *effective* magnification averaged over its extent

with respect to the source-observer line of sight.) In galactic microlensing the involved timescales are of order of tens of days, which are not too long, and it is therefore relatively easy to observe the change in brightness of the source as this relative motion makes the source *travel* through the magnification map. As we have already said, such a brightness vs time curve is what is called a *light curve*. A light curve is, therefore, a one dimensional *cut* through a magnification map. Usually, the cut is performed over a straight line, although in the most general case the path can be curved (e.g. due to parallax).

For quasar microlensing (lensing produced by stars in a galaxy lens that is itself producing multiple macro images of a background quasar) this timescale is of the order of years. But multiple lens systems have sharp caustics which present large magnification variations in very narrow regions. Therefore we can still expect to detect brightness variations on smaller timescales if a source crosses a caustic. To account for this fact, a second time scale called the crossing time is defined as the time it takes a source to travel its own radius (cf. Schneider, Kochanek & Wambsganss, 2006). The crossing time is of order of months for typical quasar microlensing systems.

The magnification map for single axisymmetric lenses are also axisymmetric, and therefore light curves for these systems are symmetric around the time of maximum magnification and, despite providing very valuable information, we could say that they are a bit *boring*. But we also know that a binary lens is enough to spring up a plethora of variety in magnification maps (cf. Schneider & Weiss, 1986). Therefore, light curves for binary lenses, being cuts through those maps along a straight line, combine the already rich variety in maps with the variety in paths over the map. The result is an extraordinary richness in light curves which make them indeed much more appealing. Light curves for binary lenses are widely used in galactic microlensing for modelling some peculiar events.

As an exercise we will try here to reproduce some of the light curves from binary lenses from a well known paper by the MACHO Collaboration (Alcock et al., 2000). The paper provides the lens parameters, so we can easily produce the magnification maps. Here, the angular scales will be given in units of the Einstein Radius for the total mass of the lens, so we can assume $M_T = M_1 + M_2 = 1$ (for a different total mass, lengths scale as $L \propto M_T^{1/2}$). The relevant parameters are therefore the mass ratio $\mu = M_1/M_2$ and lens separation a , which we can get from the paper. Thus, two masses $M_1 = 1/(1 + \mu)$ and $M_2 = \mu/(1 + \mu)$ located $(M_2 a, 0.0)$ and $(-M_1 a, 0.0)$ are used for the magnification map.

The path along the map is described by the angle with respect to the line between the lenses θ and the closest approach of the track to the center of mass of the system u_{min} . This is a straight line $y = mx + x$ with $m = \tan \theta$ and $n = u_{min}/\sqrt{m^2 + 1}$. Once the abscissae of the extremes of the track are chosen (for example as the edges of the magnification map), ordinates can be easily calculated from the straight line equation above. Some other parameters (the time of closest approach to center of mass t_0 , and a lensing time \hat{t}) are needed to put the x axis into an absolute time scale. In our case, being interested just in the procedure to produce the curves, we will keep it in angular units.

In order to be able to produce the profile, we will use a simple function to produce the cut through magnification map. Let's put it in a module called `aux.py`. The function, which we call `profile` receives as parameters the magnification map, and the abscissae and ordinates of the initial and end point of the track. The module would look something like this:

```
import numpy as np
from math import sqrt

def profile(c,x0,y0,x1,y1,method='nn'):    # Coords are in pixels
    num=int(round(sqrt((x1-x0)**2+(y1-y0)**2))) # Length of track in pixels
    xp, yp = np.linspace(xpp0, xpp1, num), np.linspace(ypp0, ypp1, num)
    # x and y coordinates of track
    zp =c[yp.astype(np.int), xp.astype(np.int)]
    return zp
```

Be aware that `profile` expects coordinates in pixels, not in R_E units, so we must convert coordinates into pixels first.

With our previous code to produce magnification maps and this function, it is easy to try to reproduce some of the cases in Alcock et al. (2000). Here, we show the result for events 98-SMC-1 in Figure 6.7 and 403-C in Figure 6.8.

The light curves are pretty noisy in the low magnification regions because we have only used one hundred rays per unlensed pixel in the calculations.

6.5.2 Quasar Microlensing Magnification Maps

Our next step is already to address the calculation of magnification maps for the case of quasar microlensing. In quasar microlensing we have a *macro* lens responsible for the multiple images of the quasar. From the image configuration we can calculate a *macro* model which provides values of the convergence κ and shear γ at the position of each image. But if a fraction α of the surface mass density originating the multiple images is in the form of compact objects, then microlensing can occur due to the line of sight to

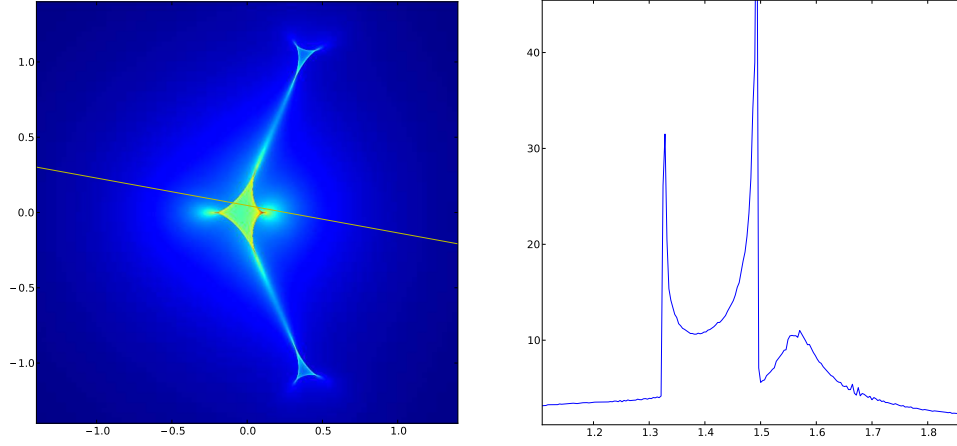


Fig. 6.7. Magnification map, source track and light curve for microlensing event MACHO 98-SMC-1

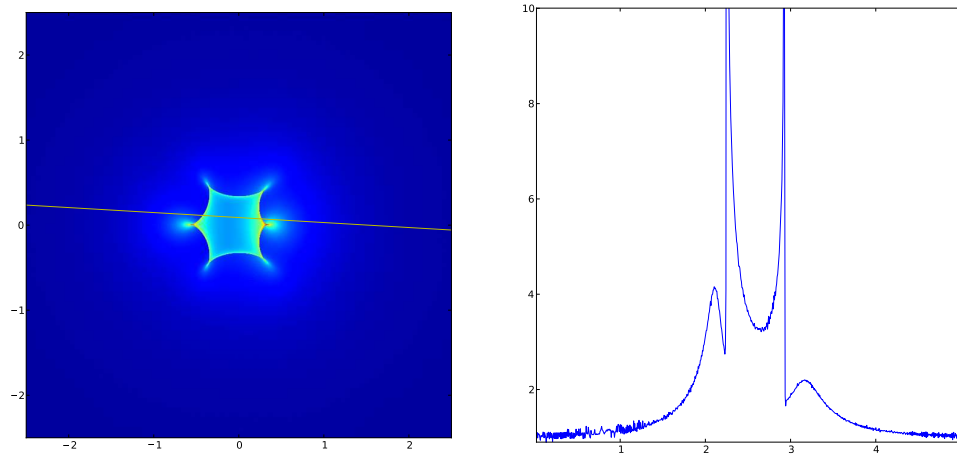


Fig. 6.8. Magnification map, source track and light curve for microlensing event MACHO 403-C

the source passing close to one of these objects. The total convergence κ is therefore split into a part coming from a smooth distribution of matter $\kappa_s = (1 - \alpha)\kappa$ and a part coming from compact objects (able to produce microlensing) $\kappa_* = \alpha\kappa$. This last fraction of matter is assumed here to be in form of point sources (although other lens models are also possible) and distributed uniformly.

The lens equation for quasar microlensing is then:

$$\mathbf{y} = \begin{pmatrix} 1 - \kappa_s - \gamma & 0 \\ 0 & 1 - \kappa_s + \gamma \end{pmatrix} \mathbf{x} + \sum_{i=1}^{N_*} m_i \frac{(\mathbf{x} - \mathbf{x}_i)}{|\mathbf{x} - \mathbf{x}_i|^2} \quad (6.9)$$

For simplicity we will assume here that all masses are equal to one solar mass $m_i = m = 1$, although different mass distributions can be considered as well. As we are making our calculations in units of the Einstein radius for a unitary mass, the effect of having a different average mass is just to rescale all length scales by $\sqrt{\langle M \rangle}$.

We will choose a rectangular shooting region at the lens plane with axis ratio $q = \frac{(1-\kappa+\gamma)}{(1-\kappa-\gamma)}$. The reason for this is that this is the axis ratio of the region into which a square at the source plane would be mapped by the lens if all the convergence would be in smooth matter. The size of the shooting region is made proportional to the size of the magnification map taking into account the stretch mentioned above. An extra factor 3/2 is included to account for the fact that the grainyness of the deflector may deflect rays further out into our magnification map. Therefore, if we want a magnification map with half-size y_l , the shooting region will have a size $1.5y_l/|(1-\kappa-\gamma)|$ and $1.5y_l/|(1-\kappa+\gamma)|$ along the horizontal and vertical axis respectively.

Another important detail to be considered is the size of the region where we are going to distribute the deflectors. The issue here is the *diffuse flux* that may end up hitting our map at the source plane due to deflectors outside the region where we are putting our stars. Katz, Balbus & Paczynski (1986) and Schneider & Weiss (1987) independently estimated that the minimum number of stars which need to be included in order to lose less than a fraction ϵ of the flux is given by:

$$N_* = \frac{3\kappa_*^2 \epsilon^{-1}}{(1-\kappa)^2 - \gamma^2} \quad (6.10)$$

Taking into account that for a square region of side L_x we have:

$$N_* = \kappa_* L_x^2 / \pi^\dagger. \quad (6.11)$$

This gives a minimum size for the region into which the deflectors are to be distributed of:

$$L_x^{min} = \left(\frac{\pi 3\kappa_* \epsilon^{-1}}{(1-\kappa)^2 - \gamma^2} \right)^{1/2} \quad (6.12)$$

[†] It is sometimes common to distribute the stars in a circle. In that case, if L_x is the radius of the circle, the π factor in the denominator disappears

This size is valid to be sure that there is no more than ϵ flux lost at the origin. But the shooting region may be quite large for large magnification maps (and/or large values of the mean magnification). Then, this size has to be increased accordingly (and of course, also the number of stars in order to fulfill eq. 6.11) by the size of the shooting region ($2x_l$):

$$L_x = L_x^{min} + \frac{3y_l}{\min(|1 \pm \kappa - \gamma|)} \quad (6.13)$$

where the *min* in the denominator of the second term chooses the sign in \pm which makes the second term larger.

For most practical applications κ_* is a very small number and (except, for example, for very small sizes of the magnification maps) the second member of equation 6.13 usually dominates and it is usually safe to neglect the first term. In any case, it is necessary to compare both terms before neglecting any of them. In our implementation Once this size is determined, the number of stars is calculated using equation 6.11.

We are now ready to write a piece of python code that calculates magnification maps for quasar microlensing. The algorithm will be structured the following way:

- (i) Set up the input parameters of the problem: Total convergence (κ) and shear (γ), fraction of convergence in form of stars α , rays per unlensed pixel (N_r), size of the magnification map in Einstein Radii ($2y_l$) and the resolution as the number of pixels of a side of the map (N_y). We may also set the value for the fraction of flux ϵ that we may leave out and that will set the minimum number of stars and the size of the region populated with deflectors. We then make some calculations to set up the size of the shooting region ($2x_{l1}, 2x_{l2}$) and of the region populated with the stars ($2x_{ls}$) according to the principles described above. Finally, in this preparatory section, we create the array that is to contain our final map.
- (ii) Next, we randomly distribute N_* the stars within the appropriate region.
- (iii) Loop over the grid of rays in a similar way as what we did in Section 6.5. Again, we take benefit of numpy arrays and throw one row of rays at a time in order to speed up the process.
- (iv) Deflect the row of rays according to the lens equation. This deflection contains an inner loop over all the deflectors which, for many interesting cases, is quite time consuming.
- (v) With the coordinates of the deflected rays at the source plane,

- calculate the coordinates of the pixel on which the rays have hit and, if it is within our region of interest, add 1 to the value of that pixel.
- (vi) When the loop over rows of rays is over, normalize the magnification map by dividing the array by the number of rays per unlensed pixel.
 - (vii) Display and/or save the results.

A python code that does just that is shown below:

qmic.py

```
import numpy as np                # Import needed modules
from math import pi
import matplotlib.pyplot as plt
from random import seed, uniform # Random number stuff
from time import time, clock, sleep # Timing stuff
from pyfits import writeto       # To be able to save output as fits
startt=time()                   # Time at start of execution

# ***** Model Parameters *****
kappa=0.59    # Total Convergence
gamma=0.61    # Shear
alpha=0.999   # Fraction of mass in form of microlenses
raypix=15.0   # Rays per pixel in absence of lensing
ny=1000       # Pixels in the magnification map
yl=10         # Half size of magnification map in Einstein Radii
eps=0.02      # Maximum fraction of flux lost
# **** Make some preliminary calculations ****
ks=kappa*alpha    # Convergence in microlenses
kc=kappa*(1.-alpha) # Convergence in smooth matter
ys=2.*yl/(ny-1)   # Pix size in the image plane
oos=1./ys         # Inverse of pixel size on image plane
sqrpix=np.sqrt(raypix) # Rays per pixel in one dimension
f1=1./abs(1.-kappa-gamma) # Exp. factor on horizontal axis
f2=1./abs(1.-kappa+gamma) # Exp. factor on vertical axis
fmax=max(f1,f2)    # Max Exp factor
xl1,xl2=1.5*yl*f1, 1.5*yl*f2 # Half Size of shooting region in x and y
xl=1.5*yl*fmax      # Longest Half side of shooting region
nsmin=3*ks**2/eps/abs((1.-kappa)**2-gamma**2) # Min number of stars
xmin=np.sqrt(pi*nsmin/ks)/2 # Min half side of star region
xls=xl+xmin         # Expand to account for shooting region
nx1=np.int16(np.round(1.5*ny*f1*sqrpix)) # Rays in shoot. reg. along x axis
nx2=np.int16(np.round(1.5*ny*f2*sqrpix)) # Rays in shoot. reg. along y axis
nx=max(nx1,nx2)     # Number of rays along longest side
xs=2.*xl1/(nx1-1)   # Pixel size on image plane
xn1=abs(ks*(2*xls)*(2*xls)/pi) # Number of microlenses
nl=int(xn1)         # Number of microlenses (int)
thmag=1./(1-kappa-gamma)/(1-kappa+gamma) # Theoretical value of magnification
print "*****"
print "Half Size of map in Einstein radii      =", yl    # Print some parameters
print "Number of pixels of magnification map   =", ny
print "Half size of shooting region           =", xl
print "Number of rays along the longest axis   =", nx
print "Half size of region with microlenses    =", xls
print "Total Convergence,                     k =", kappa
print "Shear,                                gamma =", gamma
print "Fraction of mass in microlenses, alpha =", alpha
print "Convergence in form of microlenses, ks =", ks
print "Number of microlenses                   =", nl
```

```

print "Rays per unlensed pixel,          raypix =", raypix
print "Theoretical Mean Magnification,    mu =", thmag
print "*****"
b=np.zeros((ny,ny))          # Initialize magnification map
# ***** Randomly distribute stars in region *****
x1l=np.zeros(nl)             # Initialize microlens positions to zero
x2l=np.zeros(nl)
seed(1.0)                    # Initialize random number generator
for i in range(nl):          # Generate positions of microlenses
    x1l[i]=uniform(-xls,xls)
    x2l[i]=uniform(-xls,xls)
# *****
perc0=5.                     # Percentage step to show progress
perc=5.                      # Initial Percentage
yr=np.arange(0,nx2)          # Array for looping over rows of rays
y,x=np.mgrid[0.0:1.0,0:nx1] # These are arrays with x and y coords
                                # of one row of rays in image plane
nrange=np.arange(nl)         # Array for looping over lenses
# ***** MAIN LOOP *****
for i in yr:                  # Main Loop (over rows of rays)
    if ((i*100/nx2)>=perc): # If perc is completed, then show progress
        perc=perc+perc0
        print round(i*100/nx2,"%", round(time()-startt,3), " secs"
        # print Completed fraction and elapsed execution time
        x2=-x12+y*xs        # Convert pixels to coordinates in the image plane
        x1=-x11+x*xs
        y2=x*0.0            # Initialize variables
        y1=x*0.0
        for ii in nrange:    # Loop over microlenses
            x1ml=x1-x1l[ii]
            x2ml=x2-x2l[ii]
            d=x1ml**2+x2ml**2 # Distance to lens ii squared
            y1=y1+x1ml/d      # Deflect x coordinate due to lens ii
            y2=y2+x2ml/d      # Deflect y coordinate due to lens ii
            del x1ml,x2ml,d
            y2=x2-y2-(kc-gamma)*x2 # Calculate total y deflection
            y1=x1-y1-(kc+gamma)*x1 # Calculate total x deflection
            i1=(y1+y1)*oosys      # Convert coordinates to pixels on source plane
            i2=(y2+y1)*oosys
            i1=np.round(i1)        # Make indices integer
            i2=np.round(i2)
            ind=(i1>=0) & (i1<ny) & (i2>=0) & (i2<ny) # Select indices of rays
                                                # falling onto our source plane
            i1n=i1[ind] # Array of x coordinates of rays within map
            i2n=i2[ind] # Array of y coordinates of rays within map
            for ii in xrange(np.size(i1n)): # Loop over rays hitting the source plane
                b[i2n[ii],i1n[ii]]+=1      # Increase map in one unit if ray hit
            y=y+1.0 # Move on to next row of rays
#*****
b=b/raypix # Normalize by rays per unlensed pixel
print "*****"
print "Measured mean magnification =",np.mean(b)
print "Theoretical magnification is =",thmag
print "*****"
if (thmag<0): # Vertical or horizontal flip in some cases
    if (gamma<0):
        b=np.flipud(b)
    else:
        b=np.fliplr(b)
#***** Display results *****
ax=plt.subplot(121) # Left plot

```

```

plt.plot(xl1,x2l,'+') # Plot positions of stars
rayboxx=[-xl1,-xl1,xl1,xl1,-xl1]
rayboxy=[-xl2,xl2,xl2,-xl2,-xl2]
plt.plot(rayboxx,rayboxy) # Show Shooting region
mapboxx=np.array([-yl,-yl,yl,yl,-yl])
mapboxy=np.array([-yl,yl,yl,-yl,-yl])
plt.plot(mapboxx*f1,mapboxy*f2,'r') # Show region mapped onto map
plt.xlim(-1.1*xls,1.1*xls)
plt.ylim(-1.1*xls,1.1*xls)
ax.set_aspect('equal') # Keep aspect ratio
plt.subplot(122) # Right plot
imshow=plt.imshow(b,origin='lower') # Display magnification map
#*****
print "Exec. time = ",round(time()-startt,3), ' seconds' # Print execution time
plt.show()
#***** Save result as fits file? *****
save=''
while (save not in ['y','n']): # Wait for input unless it is 'y' or 'n'
    save=raw_input("Save file (y/n)? ")
    if (save == 'y'): # If 'y'
        filename=''
        filename=raw_input("Filename = ").+'.fits'
        writeto(filename,b) # Write fits file

```

This program is somewhat longer than our previous examples, but still it is only 131 lines of code, of which nearly 20% are mostly *ornamental* (i.e. printing values and comments). It is pretty simple yet it is a fully standalone piece of software (assuming the needed libraries are installed, of course) capable of producing nice magnification maps for quasar microlensing. An example of a magnification map produced with this code can be found in Figure 6.9. On top, the lens plane is also shown with the position of the stars and a box marking the shooting region.

Figure 6.10 shows the magnification map for a real lensed quasar. It is a magnification map corresponding to the image D of the famous *Einstein Cross* lens system QSO 2237+0305. This lens system is peculiar in many ways, but in particular in that the lines of sight through the lens that produce the four images of the quasar cross the very innermost part of the lens galaxy and therefore we can assume that nearly 100% of the surface mass density is in form of stars. But for most multiply imaged quasars, this is usually not the case. Mediavilla et al. (2009) by studying a sample of 29 image pairs through 20 lens systems showed that, on average, around 5% of the mass is in form of stars while the rest is smooth (dark) matter.

6.6 Source Size Effects

Up to now, we are implicitly assuming that the source has the size of one pixel at the source plane. We know nothing about the magnification structure

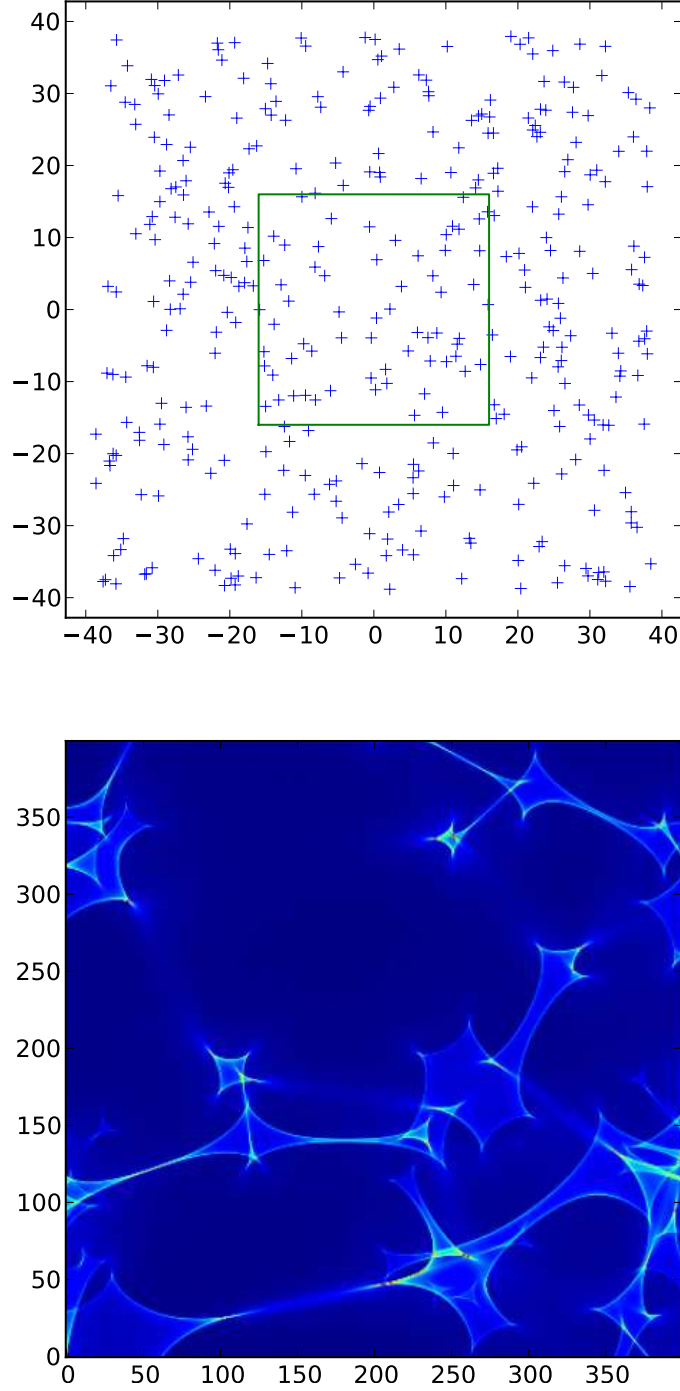


Fig. 6.9. Magnification map produced with the code above for $\kappa = 0.7$ and $\gamma = 0.0$. 28.6% of the lens mass is in form of stars, resulting in $\kappa_* = 0.2$. The size of the map is $6.4R_E$ and it has 400×400 pixels. On top, the lens plane with the positions of the 385 stars. The ray shooting region is marked by the green square. ϵ is set to 0.01.

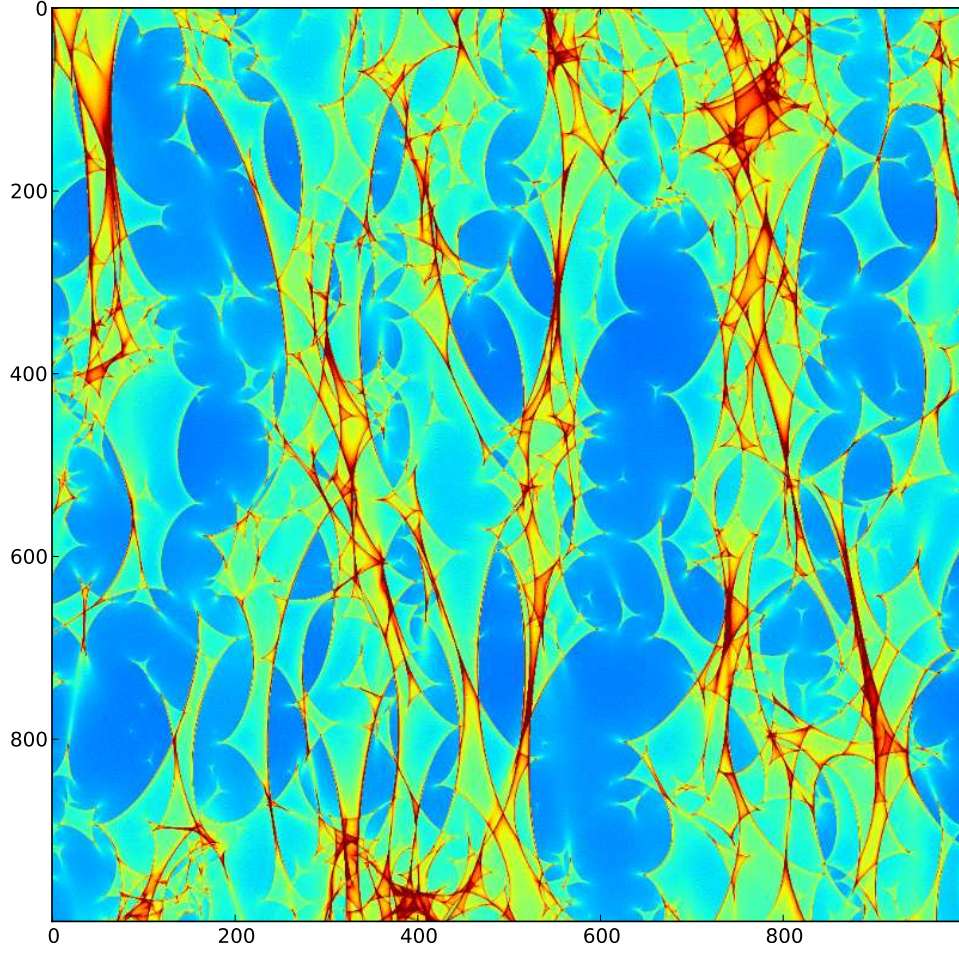


Fig. 6.10. Magnification map for image D of QSO 2237+0305. The map is $20R_E$ on a side. The parameters for the calculations are $\kappa = 0.59$, $\gamma = 0.61$ and 100% of the mass in form of stars. The calculation was done with 6553 stars.

at subpixel scales. The only solution is to increase the resolution of the magnification map. On the other side, if the source is larger than a pixel, the source is affected by an *effective* magnification which is the convolution of the source's brightness distribution with the magnification map at that location. If the source has a brightness distribution described by $S(\mathbf{y})$, then the magnification of a finite source M^s at location \mathbf{y} at the source plane would be given by:

$$M^s(\mathbf{y}) = \sum_{\mathbf{y}_0} S(\mathbf{y} - \mathbf{y}_0) M(\mathbf{y}_0) \quad (6.14)$$

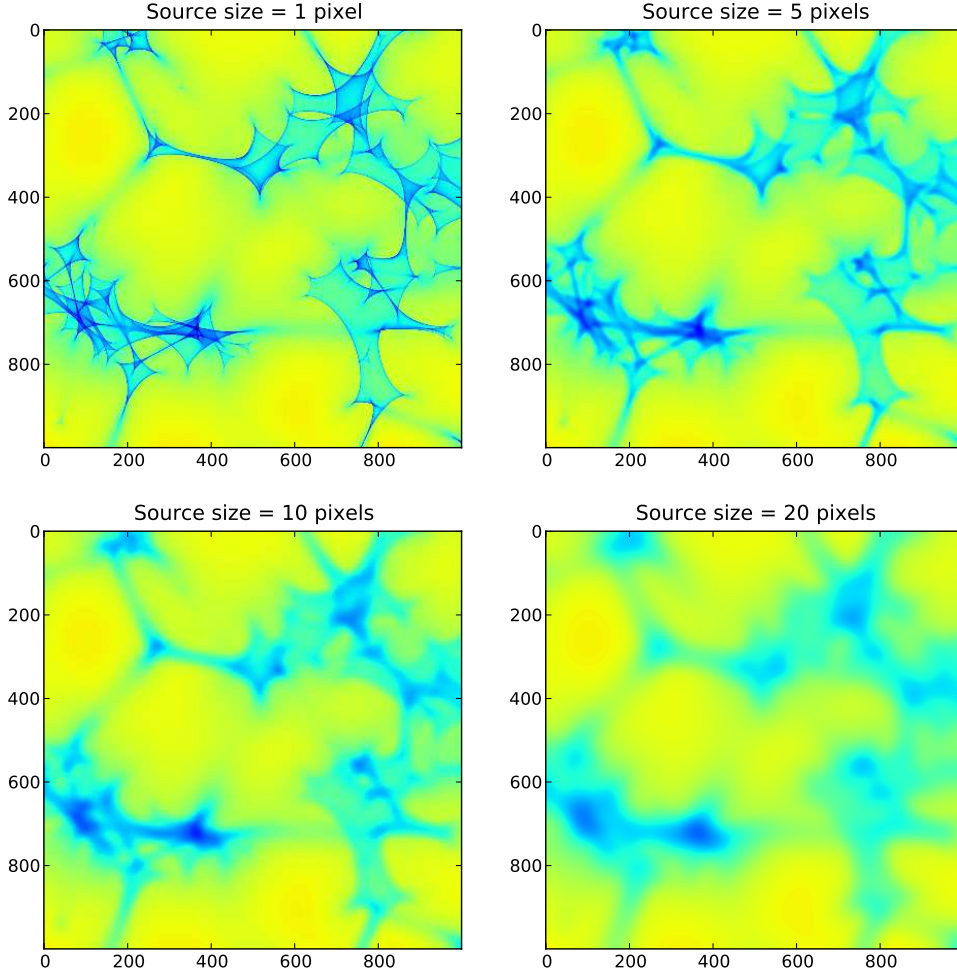


Fig. 6.11. Effect of source size on effective magnification. The original map (upper left) corresponds to $\kappa = 0.45$ and $\gamma = 0$ with 90% of matter in form of stars of $M = 1M_{\odot}$. The side of the map is $10 R_E$. For the other maps, the source has a gaussian profile with a σ of 5 (upper right), 10 (lower left) and 20 (lower right) pixels (corresponding to 0.005, 0.01 and 0.02 R_E respectively).

where $M(\mathbf{y}_0)$ is the magnification at pixel \mathbf{y}_0 , and the summation is carried over the whole magnification map (although only pixels within the extent of the source do actually contribute). Therefore, the effect of a finite source size (larger than the pixel size) is to smear the magnification map structures. The larger the size of the source, the larger the smearing effect. The effect of different source sizes is illustrated in Fig. 6.11.

We may wonder to what extent the effect of the size depends on the shape

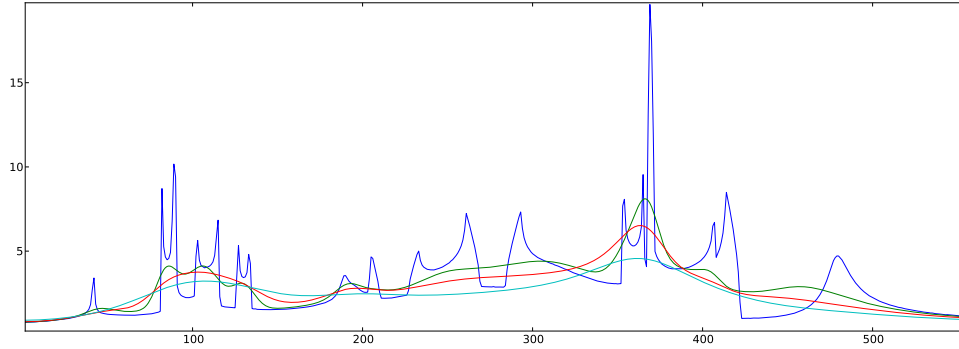


Fig. 6.12. Light curve for different sizes of the source. Blue, Green, Red and Cyan correspond to gaussian sources with a sigma of 1, 5, 10, and 20 pixels respectively shown in Fig. 6.11.

of the luminosity profile of the source. Mortonson, Schechter & Wambsganss (2005) have shown that, for circular sources, the radial profile of the source is quite irrelevant for statistical purposes. The only important parameter is the *half light radius* $r_{1/2}$ that contains half of the source luminosity. Once we know that the particular shape of the brightness profile of the source is not that important, we are free to choose the handiest one. It is quite common to use a circular gaussian of sigma r_s to model the source. The half light radius for such a source is $r_{1/2} = 1.18r_s$. Other profiles have also been used in the literature (e.g. uniform disks, etc.). Gaussians are very convenient to model the source, as convolution with a gaussian is a pretty common task, and it is very easy to find the convolution already implemented in most programming languages[†]. Python is no exception, and we can find such a function in the `ndimage` module of `scipy`. The function is called `gaussian_filter`. We can therefore add a function for gaussian convolution function `gaussconv` into `aux.py` to perform this task:

```
import scipy.ndimage # To go at the top of aux.py with the other imports
def gausscon(a,size): # a is the input map and size the sigma of the gaussian
    b=a*0.0           # Create the output array. Same size as input array
    scipy.ndimage.gaussian_filter(a, [size, size],output=b)
    return b
```

The effect of this blurring can be seen more clearly on the light curves. A cut through those maps can be seen in Figure 6.12:

We clearly see that sharp changes in magnification due to caustic crossings disappear for large sources, and light curves look pretty smooth for the larger

[†] 2D convolution with a gaussian kernel is a separable problem that can be split into two 1D convolutions. Implementations that benefit from this are much faster, particularly for large kernels

sizes. This different light curves for the same lens system but for different sizes are indeed to be expected. The standard model for accretion disks around black holes is that of Shakura and Sunyaev (1973) predicts that the temperature of the accretion disk is much higher at the inner parts than further out. Thus, it also predicts that the *effective* radius of the disk is wavelength dependent (with a power law $r \propto \lambda^{4/3}$) and we can expect to be able to observe something like the light curves in Figure 6.12 for different wavelengths. For example, the blue curve could correspond to X-Ray observations while the cyan one could correspond to the optical disk. This effect is known as microlensing *chromaticity*. A recent study of this effect using these techniques on HST data has been done by Muñoz et al. (2011).

To see what the effect of this blurring is on the statistics of the magnification maps, we may plot histograms of the magnification in the different maps shown in Figure 6.11. As astronomers use to express brightness in magnitudes, it is customary also to represent magnification histograms in magnitudes around the mean magnification of the map. The result is shown in Figure 6.13.

What we see is that the larger the size, the less probable it is to find large magnifications. The rule of thumb is therefore: *Small sources suffer more microlensing magnification than large sources*. Of course, the latter is to be understood in a statistical sense. But beware: Even things with very low probability may eventually happen!. So do not take the above as an infallible rule. In this particular case shown in Figure 6.13, this effect is more pronounced for negative magnifications. So, it is nearly 8 times more probable to find a magnification of -2 mags for a source of 1 pixel (blue curve) than it is to find it for sources of 10 pixels (green curve). On the contrary, lower magnifications below -1 are more probable for larger sizes.

This effect therefore allows to perform statistical studies based on the observed microlensing magnifications and to use this effect to put constraints on the size of the source. This has been done for example by Jiménez-Vicente et al. (2012) to estimate the size of the accretion disk for a sample of 19 lensed quasars.

6.7 Beyond Simple Inverse Ray Shooting: Treecodes and IPM

The Inverse Ray Shooting is a pretty simple algorithm with a very easy implementation as we have just seen. Nevertheless, for many practical applications it requires very long execution times. Quite often we need to address problems in which very compact and very large sources are involved

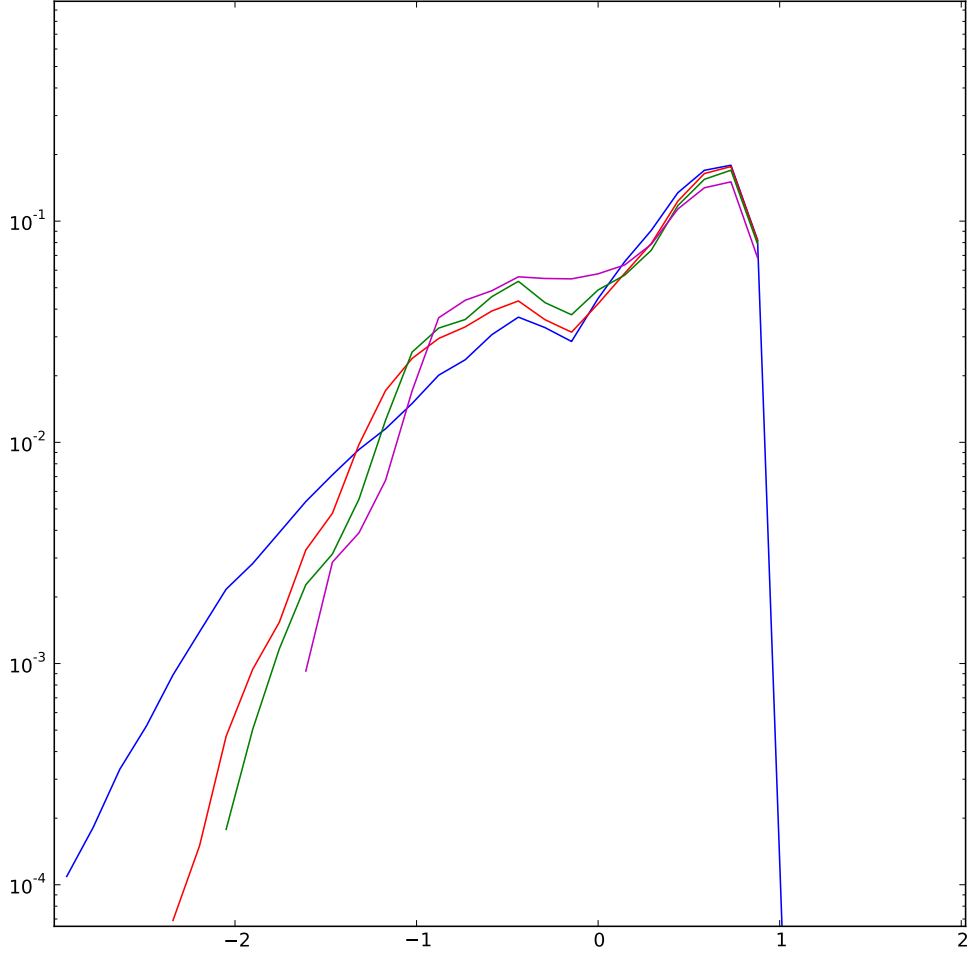


Fig. 6.13. Magnification histograms for different sizes of the source. Blue, Red, Green and Magenta correspond to gaussian sources with a sigma of 1, 5, 10, and 20 pixels respectively shown in Fig. 6.11.

at the same time. This means that we need to calculate magnification maps for very large regions at the source plane but, at the same time, with a very high resolution. The reason why this calculations take so long is easily found by looking at equation 6.8. As an example, let's calculate how many calculations are needed to calculate a map at moderately high resolution (i.e. $N_x = 2500$) and for a large region at the source plane (which means we will need to include many stars at the lens plane in our calculations). Let's take $N_* = 5 \times 10^5$. To have good accuracy in the map we need to shoot a few hundred rays per unlensed pixel. Let's take $N_r = 500$. If every deflection

need or order 10 operations, we have, for the total number of operations to deflect all rays:

$$N_{op} = 10 \times N_x^2 \times N_r \times N_* \approx 1.5 \times 10^{16} \text{ flop} \approx 15 P \text{ flop} \quad (6.15)$$

This is quite heavy indeed. A modern fast workstation is able to sustain around 10 Gflop per second. Such computer would need nearly three weeks to finish this tremendous task !!. Even a much more modest calculation with only $N_x = 1000$ pixels and $N_* = 5 \times 10^4$ stars would need nearly 7 hours !!.

Of course we can always use faster hardware. But not everyone may have access to it. And still, using it may sometimes be rather complicated. Nowadays, the use of GPUs for scientific computing has become increasingly popular. They provide very fast yet reasonably cheap hardware which, moreover, is perfectly suited for the calculation of microlensing magnification maps. This is so because GPUs benefit better from having many available cores when the problem is highly parallelizable. And it is difficult to find a problem more parallelizable than this one!. Indeed, every ray deflection is independent of any other, and even the deflection produced by a lens/cell is independent of the others. This makes the load balance very easy, and this problem very well suited for GPUs. A performance comparison of an implementation of a plain IRS algorithm into GPU, with an parallel implementation of the hierarchical treecode into a cluster of CPUs, and a standard implementation of the hierarchical treecode into a single CPU can be found in Bate et al. (2010).

How can we help this calculations to go faster?. A carefull look into the previous equation reveals that there is only two places where there is some hope, if any, to make improvements. We cannot change the resolution of the map, as this is set by the problem. We may try to optimize the number of operations needed to perform a single ray deflection. But, to be honest, even if we do the finest of the works, there is not much gain. But, what about the other two factors?. The reader may (rightly) say that the number of stars is also set by the problem, and that a high number of rays per unlensed pixel is needed in order to produce accurate maps. We may fine tune the latter number one a bit, but, again, to no avail. But some clever people have seen somewhat deeper that that, and realized that we can indeed save quite a large amount of time by reducing those two factors. The reduction of the number of lenses included in the calculations is the basis of the treecodes. The reduction of the number of rays per unlensed pixel needed to produce a highly accurate map is the path taken by the developers of the Inverse Polygon Mapping (IPM).

6.7.1 Treecodes

The key idea behind treecodes is to group particles together and treat their combined gravitational effect as if it comes from a single entity. This is possible because the effect of particles far away is weak (it diminishes like $1/r$), so that many particles can be grouped together and its combined effect taken into account as a single *pseudoparticle*. This way, we do not need to loop over all the lenses for every single ray.

The effect of many of those far away lenses varies quite smoothly with position, and we can then group together a bunch of lenses which are far away from a given point but close to each other and treat them as a single *pseudoparticle* which properties very much reproduce the gravitational effect of the whole set of individual lenses. This way, for a given location at the lens plane, we only really need to treat individually the closest lenses. For the rest, they can be grouped together into *pseudoparticles*. This way, the deflection angle is split into two parts:

$$\alpha = \sum_{i=1}^{N_*} \alpha_i = \sum_{i=1}^{N_*^{near}} \alpha_i + \sum_{i=1}^{N_*^{pseudo}} \alpha'_i = \alpha^{near} + \alpha^{pseudo} \quad (6.16)$$

Where N_*^{near} is the number of nearby lenses that are to be included individually, and N_*^{pseudo} is the number of *pseudolenses* (which, each account for the contribution of some amount of individual lenses). This way, the factor N_* in our equation is greatly reduced, as now, the effective number of stars is $N_* = N_*^{near} + N_*^{pseudo}$ which may be of order of a few hundred. No wonder, there is a big gain in execution speed by using this approach.

This idea of grouping together far away lenses to speed up computation of the magnification maps was already used in the seminal work by Schneider & Weiss (1987). This procedure was refined and improved by introducing a *hierarchical* treecode by Wambsganss (1991, 1999). *Hierarchical* Treecodes were introduced into n-body problems in the early 80's by several authors, the most popular and widely used being the algorithm introduced by Barnes & Hut (1986).

The idea of a hierarchical treecode is that of recursively dividing the space into smaller cells at different levels. The division ends when every cell has either one or zero particles. It is common that cells at a given level are half the (linear) size of the cells in the immediately inferior level. In 3D, if we start with a cubic volume (the *root* cell) and divide it into 8 cells (branches) of half the (linear) size, and we repeat the procedure recursively, this division of space is called an *octree*. In 2D, the equivalent division of space has 4 subcells per parent cell and the tree is called a *quadtree*. For each cell, we calculate

the properties of a *pseudoparticle* with the whole mass of the particles within the cell, and located at the center of mass. We can even calculate higher multipolar terms to account for finer details of the gravitational effect of the *pseudoparticle*. The idea behind hierarchical treecodes is that of using larger cells for far away regions, smaller cells for nearer regions, and individual particles for the nearest region. In principle, we would like to use cells as large as possible without introducing large errors. How to decide if a cell has the right size to stop going *down* the tree?[†] In the Barnes-Hut algorithm this is decided via the *cell-opening* parameter δ . This criterium establishes that a cell of size s is already small enough to be included in the calculation if it is at distance d and fulfills that $s/d < \delta$. Values of $\delta \lesssim 1$ usually give a good approximation for the force calculations.

This way, if we are calculating the forces between N particles, the algorithm reduces a problem from $O(N^2)$ computations to another with $O(N \log N)$ computations. This is indeed a huge improvement when N is a large number!. In our case, we want to calculate the deflection of $N_r \times N_x^2$ rays by N_* stars, so we convert a problem of $O(N_r \times N_x^2 \times N_*)$ into a problem of $O((N_r \times N_x^2) \log N_*)$. Again, a very substantial gain.

A thorough description of the algorithm is completely out of the scope of this tutorial and the reader can find all the details in the original publications by Wambsganss (1991, 1999). Nevertheless, I shall briefly describe here the most important details:

- The lens plane is recursively divided into a quadtree.
- For each cell, not only the center of mass, but multipoles up to order 6 are calculated.
- An *opening* parameter δ with values between 0.4 and 0.9 is used to decide the size of the cells to include in the calculations.
- Further speed-up is achieved by taking into account that the particle/cell structure to include for nearby rays does not change much. Therefore, there is no need to recalculate that structure for every ray, as it can be re-used for a certain amount of them.
- Still further speed-up is achieved by noticing that the deflection angle of cells far away do change little and very smoothly with position, so it can be calculated at a few positions and interpolated in between.
- The deflection is calculated by directly adding the effect of nearby lenses, and including only large cells/pseudoparticles at longer distances.

[†] We should actually say *up* the tree, as we are going from root to leaves. But it is customary to represent the tree upside-down and therefore goint deeper into the tree is going *down*.

This way, for problems with millions of particles, speed-ups up to a thousand can be achieved.

6.7.2 Inverse Polygon Mapping

Further improvement into the performance of the Inverse Ray Shooting algorithms was introduced by the Inverse Polygon Mapping (Mediavilla et al., 2006; Mediavilla et al., 2011).

The basic idea behind this algorithm is noticing that what we actually do to calculate the magnification by inverse ray shooting is to transport *backwards* test areas at the lens plane, and compare them (by collecting them) with reference areas (pixels) at the source plane. In doing this, we are naively assigning the whole area transported by a ray to the pixel at the source plane which is hit. But this is extremely inefficient, as ignores a large fraction of information on the lens mapping, and we can actually do much better than that. How? By really transporting backwards a test area, and by apportioning it among the pixels that are enclosed by it (if there are more than one). In fact, except for the critical curves, the lens mapping is a diffeomorphism (meaning the function and its inverse are differentiable), so we can expect it to be topologically *well behaved*. This way, we can use the *backwards* image of a cell in the source plane to calculate the local properties of the mapping, and with them, we can make a good estimate for the magnification. What makes this method so good is that the places where we need many rays to have a good estimate of the magnification are those with low magnifications. But those regions are away from the caustics and therefore the backwards mapping of test areas works best even for very large cells!. And what about cells containing a critical curve?. In principle, these have to be identified and treated separately. Although we will soon see that this is indeed very easily done.

Again, as with the hierarchical treecode, describing all the details of the algorithm here is not possible, and the reader is forwarded to the original papers (Mediavilla et al., 2006; Mediavilla et al., 2011), but we will sketch here the main points:

- (i) Tessellate the lens plane. While the regular grid is a natural choice for the IRS algorithm, any tessellation is possible. Still, square cells (maybe with an extra ray at cell center) are commonly used.
- (ii) Transform cell vertices with the lens equation
- (iii) Check for out of linearity (i.e. identify critical cells)

- (iv) Reprocess critical cells. This can be done by subdividing them into smaller (hopefully noncritical) cells, applying IRS to this cells etc.
- (v) Aportion the transformed area among the corresponding pixels at the source plane

As stated in Mediavilla et al. (2006), the IRS would correspond to steps (i) and (ii) and would be the zeroeth order approximation. IPM without linearity check would take steps (i), (ii) and (v) and would be first order. IPM with linearity check would use the 5 steps and would correspond to second order approximation. The good news is that first order IPM, without any linearity control (i.e. ignoring critical cells and treating them as if they were not critical) works so well, that it is usually enough. The gain with IPM comes from the fact that the cells that we are transforming back can be as large as the pixels we used with IRS at the image plane or even larger. That is equivalent to say that we are using $N_r \approx 1$ or even smaller!. In fact, values of N_r as small as 0.12 (i.e. transported cells are three pixels wide!) are able to produce magnification maps even more accurate than standard IRS with 350 rays per pixel in a tiny fraction of the computational time. Indeed, moving from $N_r = 500$ to $N_r = 0.2$ in equation 6.8 makes a huge difference. With this procedure, if done carefully, the extra time to set up the tiling, etc. is really minimal and accounts for only a tiny fraction of the total computational time. And last, but not least, the IPM and treecodes are not incompatible. On the contrary, they are complementary. Indeed, the IPM is prefectly suitable to be combined with a hierarchical treecode and therefore to combine the gains provided by the two methods. IPM allows to shoot many fewer rays, while hierarchical treecodes make the calculation for each ray much faster.

6.7.3 Summary

We have seen how the IRS can be improved in two different and complementary ways to speed up computations and make the calculation of magnification maps accesible for a standard desktop computer. A few last words may help the reader to decide which path to take if he/she needs magnification map. I shall try here to give a (far from precise) recipes:

- (i) For magnification maps of a few lenses, IPM is without doubt the way to go. This is particularly true if we need to produce many of these maps in a reasonable short amount of time. If only a handful maps are needed, maybe plain IRS is enough.
- (ii) If we need microlensing magnification maps that include many lenses,

then a hierarchical treecode is the best choice. How many lenses are “many” in this context?. Well, there is not a fixed number, but a few thousand lenses is a reasonable estimate for when treecodes start to pay off. Close to this number, probably the IPM is as competitive as a treecode, and well below that, IPM is certainly faster.

- (iii) If you need mass production of maps, maybe you should seriously consider getting a GPU and give it a try.

Hopefully very soon an IPM algorithm with a hierarchical treecode will be available and we will not need to make a choice between them.

Acknowledgments

I want to express my gratitude to the organisers of this Winter School, Evencio Mediavilla and José A. Muñoz, for organizing such a wonderful School and for inviting me to deliver these lectures. In fact, I am doubly indebted with Evencio, as he has taught me most of what I know about lensing in general and making microlensing maps in particular.

I also want to thank Simon Verley for introducing me to Python and for having the patience to answer my endless questions about it during my first steps with it.

References

- Alcock, C. et al. (2000), *ApJ*, **541**, 270
 Barnes, J., Hut, P. (1986) *Nature*, **324**, 446.
 Bate, N. F., Fluke, C. J., Barsdell, B. R., Garsden, H., & Lewis, G. F. (2010), *New Astron.*, **15**, 726
 Chang, K., Refsdal, S. (1979), *Nature*, **282**, 561
 Chang, K., Refsdal, S. (1984), *A&A*, **132**, 168
 Jiménez-Vicente, J., Mediavilla, E., Muñoz, J. A., Kochanek, C. S. (2012), *ApJ*, **751**, 106
 Katz, N., Balbus, S., Paczynski, B. (1986), *A&A*, **306**, 2
 Kayser, R., Refsdal, S., Stabell, R. (1986), *A&A*, **166**, 36
 Mediavilla, E., Muñoz, J. A., López, P., Mediavilla, T., Abajas, C., Gonzalez-Morcillo, C., Gil-Merino, R. (2006), *ApJ*, **653**, 942
 Mediavilla, E., Muñoz, J. A., Falco, E., Motta, V., Guerras, E., Canovas, H., Jean, C.m Oscoz, A.; Mosquera, A. (2009), *ApJ*, **706**, 1451
 Mediavilla, E., Mediavilla, T., Muñoz, J. A., Ariza, O., López, P., González-Morcillo, C., Jiménez-Vicente, J. (2011), *ApJ*, **741**, Iss. 1, art. id. 42
 Mortonson, M. J., Schechter, P. L., Wambsganss, J. (2005), *ApJ*, **628**, 594
 Muñoz, J. A., Mediavilla, E., Kochanek, C. S., Falco, E. E., Mosquera, A. M. (2011), *ApJ*, **742**, 67
 Schneider, P., Ehlers, J., Falco, E. E. (1999), *Gravitational Lenses*, Astronomy and Astrophysics Library, Springer

- Schneider, P., Kochanek, C., Wambsganss, J. (2006), *Gravitational Lensing: Strong, Weak and Micro*. Saas-Fee Advanced Course 33, Springer.
- Schneider, P., Weiss, A. (1986), *A&A*, **164**, 237
- Schneider, P., Weiss, A. (1987), *A&A*, **171**, 49
- Shakura, N. I., Sunyaev, R. A. (1973), *A&A*, **24**, 337
- Shipman, J. W. (2011), *A Python programming tutorial*,
<http://infohost.nmt.edu/tcc/help/pubs/lang/pytut1/>
- Wambsganss, J., (1990) Ph.D. Thesis, Munich available as MPA Report 550.
- Wambsganss, J., (1999) *Journal of Comp. and App. Math.*, **199**, 353