Deflection of light by the Sun's gravitational field

Note that the deflection angle found here in the framework of general relativity is very similar to the result found in the Newtonian limit for a photon grazing the surface of the Sun. However, we find here an extra factor two.

The reason for the factor of 2 difference is that both the space and time coordinates are bent in the vicinity of massive objects — it is four-dimensional space–time which is bent by the Sun.

The famous eclipse expedition of 1919 to Sobral, Brazil, and the island of Principe, in the Gulf of Guinea, led by Eddington, Dyson, and Davidson was a turning point in the history of relativity: it confirmed that masses bend light by the amount that is predicted by General Relativity.

For further reading on the Eddington expedition, we refer the reader to Smith (2015).

1.2.2 Deflection of light in the strong field limit

For the vast majority of gravitational lenses in the universe, the weak field limit holds. However, compact objects such as neutron stars and black holes can also act as lenses. In these cases, the approximations introduced above break down, as photons travel through very strong gravitational fields. In the following, we briefly discuss the deflection angle of a static (i.e. non-rotating) compact lens.

For a general static, spherically symmetric metric in the form

$$ds^2 = A(R)dt^2 - B(R)dR^2 - C(R)(d\theta^2 + \sin^2\theta d\phi^2) \tag{1.42}$$

the analysis of the geodesic equations leads to the following expression for the deflection angle:

$$\hat{\alpha} = -\pi + \frac{2G}{c^2}\int_{R_m}^{\infty} u\sqrt{\frac{B(R)}{C(R)[C(R)/A(R) - u^2]}}dR\ , \tag{1.43}$$

where u is the impact parameter of the unperturbed photon and R_m is the minimal distance of the deflected photon from the lens (Bozza, 2010). It can be shown that

$$u^2 = \frac{C(R_m)}{A(R_m)}\ . \tag{1.44}$$

Note that, in the case of the Schwarzschild metric, $A(R) = 1 - 2GM/Rc^2$, $B(R) = A(R)^{-1}$, and $C(R) = R^2$.

In the weak field limit ($R \geq R_m \gg 2GM/c^2$, i.e. for impact parameters much larger than the lens Schwarzschild radius), Eq. 1.43 reduces to the well know equation

$$\hat{\alpha} = \frac{4GM}{c^2 u}\ . \tag{1.45}$$

The exact solution of Eq.1.43 was calculated by Darwin (1959) to be

$$\hat{\alpha} = -\pi + 4\frac{G}{c^2}\sqrt{R_m/s}F(\varphi, m)\ , \tag{1.46}$$

where $F(\phi, m)$ is the elliptic integral of the first kind, and

$$s = \sqrt{(R_m - 2M)(R_m + 6M)} \tag{1.47}$$

$$m = (s - R_m + 6M)/2s \tag{1.48}$$

$$\varphi = \arcsin\sqrt{2s/(3R_m - 6M + s)} \tag{1.49}$$

Fig.1.2.2 shows how the deflection angle varies as a function of the impact parameter of the photon. At large distances, Eq. 1.46 is well approximated by the solution in the weak field limit. For small impact parameters, the solutions in the strong and in the weak field limit differ significantly. In particular, the deflection angle in Eq. 1.46 diverges for $u = 3\sqrt{3}GM/c^2$ (or $R_m = 3GM/c^2$). Before reaching that point, the deflection angle exceeds 2π, meaning that the photon loops around the lens before leaving it.

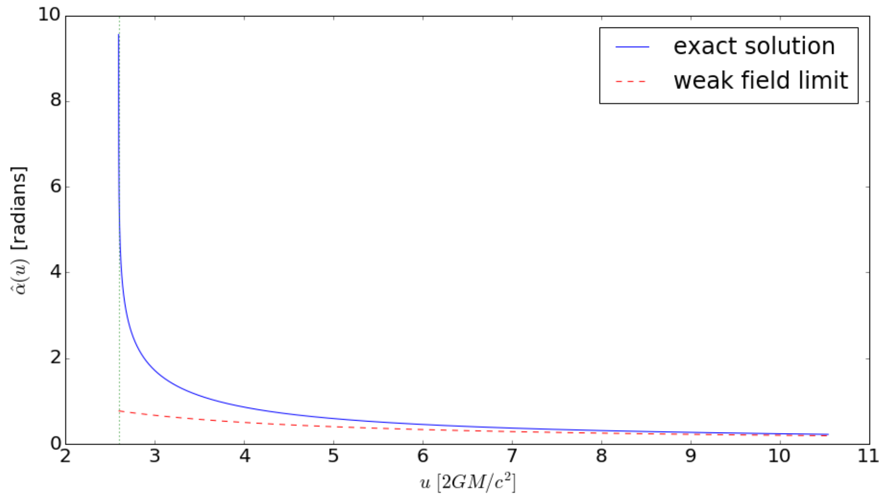

Figure 1.2.2: *Deflection angle by a compact lens as a function of the photon impact parameter. Shown are the exact solution of the geodesic equations for a Schwarzschild metric (solid line) and the solution in the weak field approximation (dashed line). The dotted vertical line shows the impact parameter,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

1.3 Python application

In our first python application, we write a script to produce Fig. 1.2.2.

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

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

We will compare the resulting deflection angle to

$$\hat{\alpha} = \frac{4GM}{c^2 u} \tag{1.50}$$

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

We start by importing some useful packages:

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

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

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

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

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

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

```
class point_bh:

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

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

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

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

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

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

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

    def phif(self,r,s):
        return(np.arcsin(np.sqrt(2.0*s/(3.0*r-6.0*self.M+s))))

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

The class contains several methods which will be used to compute the deflection angle. For example, we implement the functions $A(R)$, $B(R)$, and $C(R)$. These will be used to convert the minimal distance R_m to u. We also implement the functions s, m, φ, which depend on the mass of the black-hole and on the minimal distance R_m. Finally, we implement the function `defAngle`, which

enables to compute the deflection angle using Eq. 1.46. This function uses the method `elipkinc` from `scipy.special` to compute the incomplete elliptic integral of the first kind, $F(\varphi, m)$. Note that φ and m can be passed as `numpy` arrays, i.e. `elipkinc` can return values for a number of couples (φ, m).

Following the same approach, we build another class which deals with point lenses in the weak field limit, i.e. it implements Eq. 1.50:

```
class point_mass:

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

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

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

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

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

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

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

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

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

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

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

```
# add the legend
ax.legend()
```

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

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

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

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

1.4 Exercises

Exercise 1.2 — Write a python script to produce a figure displaying $\hat{\alpha}(R_m)$ with R_m in 9-1000 km for two lenses with mass $M = 3M_\odot$ and $M = 10M_\odot$. ■

A. Python tutorial

A.1 Installation

The codes discussed as part of these lectures have been developed and run using Anaconda python 2.7 by Continuum analytics. This is just one of the python distributions available for free and we expect that the codes proposed here should run without problems with any of them.
If the reader opts for the Anaconda distribution, she/he can download the installer, which is available for Windows, Mac OSX, and Linux platforms, from https://www.continuum.io/downloads.
Following the installation instructions, python should be ready for usage within few minutes.

A.2 Documentation

There are many resources online and books to learn how to program in python. The list below is just a starting point and does not want to be complete:

- the online official documentation can be found at this url: http://www.python.org/doc;
- several platforms for e-learning propose courses to learn python. For example, Codeacademy offers an excellent course, which can be completed in only 13 hours;
- Google also offers a python class online;
- a more extensive (and practical) guide to python is given by Learn python the hard way

A.3 Running python

Python can be run in several ways:

- from the interactive interpreter: launch “python" in a shell. Quit with Ctrl+D or type “exit()" when finished.
- create your own script with an extension “.py" and run it in a shell by typing “python <script name>.py”
- use an Interactive Development Environment (IDE). These are software which include an editor for coding and capabilities for executing the code. There are several options available (e.g. spyder, Rodeo, etc.)

- We recommend to become familiar with jupyter notebook, which is increasingly popular among python users for sharing code and ideas.

A.4 Your first python code

Try running the code:

```
# your first python code -- this is a comment
print ("Hello World!")
```

Congratulations! You have run your first python code!

A.5 Variables

Variables are names poinint to values or objects. Setting them in python is extremely easy, and you don't need to declare them before:

```
int_var = 4
float_var = 7.89778
boolean_var = True
string_var = "My name is Python"
obj_var=some_class_name(par1,par2)
```

A.6 Strings

String constants can be defined in three ways:

```
single_quotes = 'my name is Python'
double_quotes = "my name is Python"
triple_quotes = """my name is Python
and this is  a multiline  string.""" #This can contain line breaks!
```

Note that you can combine single and double quotes when you want to define strings which contain quotes themselves:

```
double_quotes1 = 'my name is "Python"'
double_quotes2 = "don't"
```

otherwise you have to use backslashes:

```
double_quotes3 = 'don\'t'
\end{python}

Strings can me sliced:
\begin{minted}[bgcolor=bg]{python}
my_name='Massimo Meneghetti'
name = my_name[:7]
surname = my_name[8:]
a_piece_of_my_name=my_name[4:7]
```

You can make many operations with strings. These are objects and have many methods. Check out this url to learn more: https://docs.python.org/2/library/stdtypes.html

Some examples:

- String concatenation:

```
back_to_my_full_name=name+" "+surname
```

- Convert to upper case

```
my_name_uppercase=back_to_my_full_name.upper()
```

The built-in function `str` converts numbers to strings:

```
my_int=2
my_float=2.0
str_int=str(my_int)
str_float=str(my_float)
```

Another way to include numbers in strings:

```
my_string1 = 'My integer is %d.' % my_int
my_string2 = 'My float is %f.' % my_float
my_string3 = 'My float is %3.1f (with only one decimal)' % my_float
```

With several variables, we need to use parentheses:

```
a = 2
b = 67
my_string4 = '%d + %d = %d' % (a, b, a+b)

a = 2
b = 67.3
my_string5 = '%d + %5.2f = %5.1f' % (a, b, a+b)
```

Not only you can convert numbers to string, but you can do the reverse operation:

```
s = '23'
i = int(s)
s = '23'
i = float(s)
```

Strip spaces at beginning and end of a string:

```
stripped = a_string.strip()
```

Replace a substring inside a string:

```
newstring = a_string.replace('abc', 'def')
```

Important note: a Python string is "immutable". In other words, it is a constant which cannot be changed in place. All string operations create a new string. This is strange for a C developer, but it is necessary for some properties of the language. In most cases this is not a problem.

A.7 Lists

A list is a dynamic array of any objects. It is declared with square brackets:

```
a_list = [1, 2, 3, 'abc', 'def']
```

Lists may contain lists:

```
another_list = [a_list, 'abc', a_list, [1, 2, 3]]
```

Note that a_{list} in this case is a pointer.

Access a specific element by index (index starts at zero):

```
elem = a_list[2]
elem2 = another_list[3][1]
```

It's easy to test if an item is in the list:

```
if 'abc' in a_list:
    print 'bingo!'
```

Extracting a part of a list is called slicing:

```
list2 = a_list[2:4] # returns a list with items 2 and 3 (not 4)
```

Other list operations like appending:

```
a_list.append('ghi')
a_list.remove('abc')
```

Other list operations: http://docs.python.org/lib/typesseq.html

A.8 Tuples

A tuple is similar to a list but it is a fixed-size, immutable array. This means that once a tuple has been created, its elements may not be changed, removed, appended or inserted.

It is declared using parentheses and comma-separated values:

```
a_tuple = (1, 2, 3, 'abc', 'def')
```

but parentheses are optional:

```
another_tuple = 1, 2, 3, 'abc', 'def'
```

Tip: a tuple containing only one item must be declared using a comma, else it is not considered as a tuple:

```
a_single_item_tuple = ('one value',)
```

R Tuples are not constant lists – this is a common misconception. Lists are intended to be homogeneous sequences, while tuples are hetereogeneous data structures.

In some sense, tuples may be regarded as simplified structures, in which position has semantic value [e.g. (name,surname,age,height,weight)]. For this reason they are immutable, contrary to lists.

A.9 Dictionaries

A Dictionary (or "dict") is a way to store data just like a list, but instead of using only numbers to get the data, you can use almost anything. This lets you treat a dict like it's a database for storing and organizing data.

Dictionaries are initialized using curl brackets:

```
person = {'name': 'Massimo', 'surname': 'Meneghetti'}
```

You can access the elements of the dictionary by using the entry keys:

```
person['name']
```

The keys can also be numbers:

```
person = {'name': 'Massimo', 'surname': 'Meneghetti', 1: 'new data'}
person[1]
```

A.10 Blocks and Indentation

Blocks of code are delimited using indentation, either spaces or tabs at the beginning of lines. This will become clearer in the next sections, when loops will be introduced.

Tip: NEVER mix tabs and spaces in a script, as this could generate bugs that are very difficult to be found.

A.11 IF / ELIF / ELSE

Here is an example of how to implement an IF/ELIF/ELSE loop:

```
if a == 3:
    print 'The value of a is:'
    print 'a=3'

if a == 'test':
    print 'The value of a is:'
    print 'a="test"'
    test_mode = True
else:
    print 'a!="test"'
    test_mode = False
    do_something_else()

if a == 1 or a == 2:
    pass # do nothing
elif a == 3 and b > 1:
    pass
elif a==3 and not b>1:
    pass
else:
    pass
```

A.12 While loops

```
a=1
while a<10:
    print a
    a += 1
```

A.13 For loops

```
for a in range(10):
    print a

my_list = [2, 4, 8, 16, 32]
for a in my_list:
    print a
```

A.14 Functions

Functions can be defined in python as follows:

```
def compute_sum(arg1,arg2):
        # implement function to calculate the sum of two numbers
        res=arg1+arg2
    return(res)
```

The function can be called by typing the function name. If the function returns a value or object, this is assigned to a variable as follows:

```
summa=compute_sum(3.0,7.0)
```

Otherwise, the function can just be called without setting it equal to any variable.

```
c=3

def change_global_c(val):
        # this function change the value of a global variable
    global c
    c=val

change_global_c(10)
```

A.15 Classes

Classes are a way to group a set of functions inside a container. These can be accessed using the . operator. The main purpose of classes is to define objects of a certain type and the corresponding methods. For example, we may want to define a class called 'square', containing the methods to compute the square properties, such as the perimeter and the area. The object is initialized by means of a "constructor":

```
class square:

    #the constructor:
    def __init__(self,side):
            self.side=side

           #area of the square:
    def area(self):
            return(self.side*self.side)

    #perimeter of the square:
    def perimeter(self):
            return(4.0*self.side)
```

We can then use the class to define a square object:

```
s=square(3.0) # a square with side length 3
print s.area()
print s.perimeter()
```

As in other languages (e.g. C++), python supports inheritance. A class can be used as an argument for another class. In this case the new class will inherit the methods of the parent class. For example:

```
class geometricalFigure(object):

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

    def getName(self):
        print 'this is a %s' % self.name

class square(geometrical_figure):

    #the constructor:
    def __init__(self,side):
        geometricalFigure.__init__(self,'square')
        self.side=side

    #area of the square:
    def area(self):
        return(self.side*self.side)

    #perimeter of the square:
    def perimeter(self):
        return(4.0*self.side)

class circle(geometrical_figure):

    #the constructor:
```

```
    def __init__(self,radius):
        geometricalFigure.__init__(self,'circle')
        self.radius=radius

    #area of the square:
    def area(self):
        return(3.141592653*self.radius**2)

    #perimeter of the square:
    def perimeter(self):
        return(2.0*self.radius*3.141592653)


s=square(3.0)
c=circle(3.0)
s.getName()
c.getName()
```

In the example above, `square` and `circle` are two examples of `geometricalFigure`. They have some specialized methods to compute the area and the perimeter, but both can access the method `getName`, which belongs to `geometricalFigure`, because they have inherited it from the parent class.

A.16 Modules

A module is a file containing Python definitions and statements (constants, functions, classes, etc). The file name is the module name with the suffix .py appended.

Modules can be imported in another script by using the `import` statement:

```
import modulename
```

The functions and statements contained in the module can be accessed using the . operator.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter).

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
from modulename import something
```

A.17 Importing packages

Packages can be added to your python distribution by using either the `pip` or `easy_install` utilities. Anaconda has its own utility for installing a (limited) set of supported packages, called `conda`. To learn more, check out https://packaging.python.org/installing/

Packages can be used by importing modules and classes in the code as discussed above.

Some packages that we will use a lot:

- numpy: fundamental package for scientific computing with Python (powerful N-dimensional array object, sophisticated functions, tools for integrating C/C++ and Fortran code, useful linear algebra, Fourier transform, and random number capabilities);

- scipy: provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization;
- matplotlib: a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms;
- astropy: a community effort to develop a single core package for Astronomy in Python and foster interoperability between Python astronomy packages.

Other packages will be introduced in the examples.