Tutorial_2_part_a

May 27, 2021

1 Working with data structures

1.1 Junior mentor:

Y. Fabian Bautista

1.2 Objectives of the tutorial

1. Create and manipulate dictionaries
2. Use Pandas package to manipulate data sets
3. Plot data structures using matplotlib
4. Use SciPy library to fit a curve to a given data set, and compute the goodness of the fit.

1.2.1 Loading packages

[1]:
```
import matplotlib.pyplot as plt # For ploting
import numpy as np # For array and mathematical manipulations
```

2 Dictionaries

What are Dictionaries?

A dictionary consists of keys and values. It is helpful to compare a dictionary to a list. Instead of the numerical indexes such as a list, dictionaries have keys. These keys are the keys that are used to access values within a dictionary.

An example of a Dictionary Dict:

[2]:
```
# Create the dictionary

Dict = {"key1": 1, "key2": "2", "key3": np.array([3, 3, 8]), "key4": (4, 4,␣
↪'4'), ('key5'): 5, (0, 1): 6}
Dict
```

[2]:
```
{'key1': 1,
 'key2': '2',
 'key3': array([3, 3, 8]),
 'key4': (4, 4, '4'),
 'key5': 5,
```

```
 (0, 1): 6}
```

Notice for the keys we can use immutable objects such as strings or tuples .

2.0.1 Acces the keys of a dictionary COVER

We can ask for the list of keys in the dictionary using the atribute .keys(). This is pecially important we using datafiles whose content we know nothing about

[3]:
```
Dict.keys()
```

[3]:
```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5', (0, 1)])
```

2.0.2 Access to the value by the key COVER

[4]:
```
Dict["key3"]
```

[4]:
```
array([3, 3, 8])
```

[5]:
```
Dict[(0, 1)]
```

[5]:
```
6
```

2.0.3 We can also use the get method to acces he value of a given key HOMEWORK

[6]:
```
Dict.get("key3")
```

[6]:
```
array([3, 3, 8])
```

2.0.4 Why is get usefull? HOMEWORK

If we ask for the value of the Dictionary for a given key, and the key does not exist in the dictionary, then the get method avoids error messages, and simply returns non

[7]:
```
print(Dict.get("key7"))
```

```
None
```

[9]:
```
print(Dic["key7"])
```

```
            ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-9-0ae95d5ef788> in <module>
```

```
----> 1 print(Dic["key7"])


NameError: name 'Dic' is not defined
```

With the get method, if nothing comes out, we can assign an ouput value

[ ]:
```
value = 4
print(Dict.get("key7",value ))
```

the type of a dictionary is dict

[10]:
```
type(Dict)
```

[10]:
```
dict
```

2.0.5 Acces the values of a dicionary HOMEWORK

[11]:
```
Dict.values()
```

[11]:
```
dict_values([1, '2', array([3, 3, 8]), (4, 4, '4'), 5, 6])
```

Notice that the elements for each keys do not have to be of the same type.

[12]:
```
Dict['key3']
```

[12]:
```
array([3, 3, 8])
```

In the case that the data values are given by arrays, we can use the atribute .dtype , to know the type of data contained in a key argument.

[13]:
```
Dict['key3'].dtype
```

[13]:
```
dtype('int64')
```

Dictionary containing arrays can be sliced

[14]:
```
Dict['key3'][1:3]
```

[14]:
```
array([3, 8])
```

3 Add elements to a dictionary COVER

[15]:
```
Dict.keys()
```

[15]:
```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5', (0, 1)])
```

[16]:
```
Dict["list"] = [1,2,3]
```

[17]:
```
Dict.keys()
```

[17]:
```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5', (0, 1), 'list'])
```

[18]:
```
Dict
```

[18]:
```
{'key1': 1,
 'key2': '2',
 'key3': array([3, 3, 8]),
 'key4': (4, 4, '4'),
 'key5': 5,
 (0, 1): 6,
 'list': [1, 2, 3]}
```

3.0.1 Remove a key from a dictionary using the method pop HOMEWORK

[19]:
```
Dict.pop('list')
```

[19]:
```
[1, 2, 3]
```

[20]:
```
Dict.keys()
```

[20]:
```
dict_keys(['key1', 'key2', 'key3', 'key4', 'key5', (0, 1)])
```

3.0.2 Adding two dictionaries

[21]:
```
dict1 = {'Ten': 10, 'Twenty': 20, 'Thirty': 30}
dict2 = {'Thirty': 30, 'Fourty': 40, 'Fifty': 50}

dict3 = {**dict1, **dict2}
print(dict3)
```

```
{'Ten': 10, 'Twenty': 20, 'Thirty': 30, 'Fourty': 40, 'Fifty': 50}
```

4 Excercise HOMEWORK

Given the following dictionaries, and using the previous explanations, write short line codes to answer the following Questions. (See more exercices)

1) When was Plato born?

[22]:
```
dict={"name": "Plato", "country": "Ancient Greece", "born": -427, "teacher":␣
↪"Socrates", "student": "Aristotle"}
#Type your answer below.
answer_1= dict["born"]
```

```
print(answer_1)
```

```
-427
```

2) Change Plato's birth year from B.C. 427 to B.C. 428.

[23]:
```
#Type your answer below.
dict["born"] =-428

print(dict["born"])
```

```
-428
```

3) Add the keyword "work" to the dictionary, with the values "Apology", "Phaedo", "Republic", "Symposium" in a list

[24]:
```
#Type your answer below.
dict['work'] = ["Apology", "Phaedo", "Republic", "Symposium"]

print(dict)
```

```
{'name': 'Plato', 'country': 'Ancient Greece', 'born': -428, 'teacher':
'Socrates', 'student': 'Aristotle', 'work': ['Apology', 'Phaedo', 'Republic',
'Symposium']}
```

given the additional dictionary

[25]:
```
dict2 ={"son's name": "Lucas", "son's eyes": "green", "son's height": 32,␣
 ↪"son's weight": 25}
```

4)Add 2 inches to the son's height.

[26]:
```
#Type your answer below.
dict2["son's height"] +=2
print(dict2)
```

```
{"son's name": 'Lucas', "son's eyes": 'green', "son's height": 34, "son's
weight": 25}
```

5) Using .get() method print the value of "son's eyes".

[27]:
```
#Type your answer inside the print.
ans_e_c=dict2.get("son's eyes")

print (ans_e_c)
```

```
green
```

6) Merge dict and dict2 into the dictionary dic_merge

[28]:
```
#Type your answer inside the print.
dic_merge={**dict,**dict2}

print (dic_merge)
```

```
{'name': 'Plato', 'country': 'Ancient Greece', 'born': -428, 'teacher':
'Socrates', 'student': 'Aristotle', 'work': ['Apology', 'Phaedo', 'Republic',
'Symposium'], "son's name": 'Lucas', "son's eyes": 'green', "son's height": 34,
"son's weight": 25}
```

See methods on dictionaries, for additional methods

5 The Pandas library COVER

In this section we use Pandas library for data visualization of .csv files. Pandas is a very big library, useful for doing data analysis. Check pandas for more information. We will use the method pandas.DataFrame . See pandas.DataFrame for information on DataFrames

For a longer beginers introduction to pandas, watch the following lecture

Firts, let's import the library we will be using

[29]:
```
import pandas as pd
```

5.0.1 What are .csv files?

A Comma Separated Values (.csv) file, is a plain text file that contains a list of data, separated by commas. Let us show an DM example on how to load and manipulate a .csv file.

5.0.2 Galaxy rotation curve

Descritpion.....

In the following, we explore the csv files for different rotation curves....

Specifying the paths to the data files It is usefull tospecify the path for the data file in two steps.

1) Path to the folder containing the data files (IF THE DATA FILES ARE IN THE SAME FOLDER AS THE PYTHON NOTEBOOK, THIS STEP IS NOT NEEDED)

2) Name of the specific file to read

IMPORTANT, THE PATHS ARE STRING TYPE OBJECTS. When we have created the two path objects, we concatenate them using the + operation for strings (See tutorial 1)

[30]:
```
folder_path = '/home/fabian/Documents/Git files/python-tutorials/Tutorial 2/
↪Rotation curves/'
file_neme = 'RotationCurve_IC2574.csv'
full_path = folder_path + file_neme
```

[31]:
```
type(full_path)
```

[31]:
```
str
```

Creating Data frames To read the .csv file we use the method pandas.read_csv

[32]:
```
df_rot_curve =  pd.read_csv(full_path)
```

It creates a DataFrame which is a Two-dimensional, size-mutable, potentially heterogeneous tabular data. It consist of the following elements: DataFrame(data, index = , columns =). Columns corresponds to the headers of the Data file.

To acces the headers names we use the method columns '

[33]:
```
df_rot_curve.columns
```

[33]:
```
Index(['radius (kpc)', 'circ velocity (km/s)', 'circ velocity error (km/s)'],
dtype='object')
```

We can also acces the data type using the method dtypes

[34]:
```
df_rot_curve.dtypes
```

[34]:
```
radius (kpc)                  float64
circ velocity (km/s)          float64
circ velocity error (km/s)    float64
dtype: object
```

The method head(i) allows us to see only the first i elements of the data frame. The default value is i=5

[35]:
```
df_rot_curve.head()
```

[35]:

	radius (kpc)	circ velocity (km/s)	circ velocity error (km/s)
0	0.232710	3.33	1.36
1	0.465421	8.87	4.32
2	0.698131	11.89	4.51
3	0.930841	15.76	5.35
4	1.163550	18.62	5.54

The method tail(i) allows us to see only the last i elements of the data frame. The default value is i=10

[36]:
```
df_rot_curve.tail(3)
```

[36]:

	radius (kpc)	circ velocity (km/s)	circ velocity error (km/s)
52	12.3336	78.41	3.96
53	12.5664	78.38	4.55
54	12.7991	78.87	2.80

We can also slice the Data frame in a more general maner

[37]:
```
df_rot_curve[5:10]
```

[37]:

	radius (kpc)	circ velocity (km/s)	circ velocity error (km/s)
5	1.39626	20.15	6.81
6	1.62897	23.33	6.16
7	1.86168	23.72	4.92
8	2.09439	24.86	4.81
9	2.32710	27.63	4.28

To acces only to the data values in the Data frame, we use the atribute value

[38]:
```
df_rot_curve.values
```

[38]:
```
array([[ 0.23271 ,  3.33    ,  1.36    ],
       [ 0.465421,  8.87    ,  4.32    ],
       [ 0.698131, 11.89    ,  4.51    ],
       [ 0.930841, 15.76    ,  5.35    ],
       [ 1.16355 , 18.62    ,  5.54    ],
       [ 1.39626 , 20.15    ,  6.81    ],
       [ 1.62897 , 23.33    ,  6.16    ],
       [ 1.86168 , 23.72    ,  4.92    ],
       [ 2.09439 , 24.86    ,  4.81    ],
       [ 2.3271  , 27.63    ,  4.28    ],
       [ 2.55981 , 29.36    ,  3.69    ],
       [ 2.79252 , 33.38    ,  4.64    ],
       [ 3.02523 , 33.49    ,  4.4     ],
       [ 3.25794 , 34.81    ,  3.79    ],
       [ 3.49066 , 37.44    ,  4.03    ],
       [ 3.72337 , 37.95    ,  4.08    ],
       [ 3.95608 , 41.17    ,  4.33    ],
       [ 4.18879 , 42.97    ,  4.24    ],
       [ 4.4215  , 43.67    ,  3.57    ],
       [ 4.65421 , 44.86    ,  4.33    ],
       [ 4.88692 , 46.81    ,  4.84    ],
       [ 5.11963 , 49.89    ,  5.15    ],
       [ 5.35234 , 51.74    ,  5.33    ],
       [ 5.58505 , 52.91    ,  4.07    ],
       [ 5.81776 , 53.65    ,  3.5     ],
       [ 6.05047 , 55.38    ,  3.54    ],
       [ 6.28318 , 57.29    ,  3.81    ],
       [ 6.51589 , 58.99    ,  4.04    ],
       [ 6.7486  , 61.39    ,  4.07    ],
       [ 6.98131 , 64.56    ,  3.81    ],
       [ 7.21402 , 66.77    ,  4.16    ],
       [ 7.44673 , 68.09    ,  4.02    ],
       [ 7.67944 , 67.52    ,  4.03    ],
```

```
       [ 7.91215 , 69.26    ,  4.      ],
       [ 8.14486 , 69.45    ,  3.41    ],
       [ 8.37757 , 71.53    ,  3.99    ],
       [ 8.61028 , 71.89    ,  4.75    ],
       [ 8.84299 , 72.2     ,  4.84    ],
       [ 9.0757  , 72.72    ,  4.83    ],
       [ 9.30841 , 73.67    ,  4.84    ],
       [ 9.54112 , 73.74    ,  4.71    ],
       [ 9.77383 , 75.03    ,  5.3     ],
       [10.0065  , 76.39    ,  4.77    ],
       [10.2393  , 76.99    ,  4.45    ],
       [10.472   , 78.19    ,  5.31    ],
       [10.7047  , 75.13    ,  4.79    ],
       [10.9374  , 76.4     ,  4.51    ],
       [11.1701  , 79.99    ,  4.17    ],
       [11.4028  , 76.07    ,  4.06    ],
       [11.6355  , 76.04    ,  4.04    ],
       [11.8682  , 75.11    ,  3.55    ],
       [12.1009  , 73.76    ,  3.52    ],
       [12.3336  , 78.41    ,  3.96    ],
       [12.5664  , 78.38    ,  4.55    ],
       [12.7991  , 78.87    ,  2.8     ]])
```

Accesing specific elements of a Data frame

Accesing a full column Data frames are pretty much like dictionaries, therefore, we can acces different column elements using the same inputs methods used for dictionaries. Of course we have to know what are the keywords (headers) of the data frame

[39]:
```
df_rot_curve.columns
```

[39]:
```
Index(['radius (kpc)', 'circ velocity (km/s)', 'circ velocity error (km/s)'],
dtype='object')
```

[40]:
```
df_rot_curve['radius (kpc)']
```

[40]:
```
0       0.232710
1       0.465421
2       0.698131
3       0.930841
4       1.163550
5       1.396260
6       1.628970
7       1.861680
8       2.094390
9       2.327100
10      2.559810
```

```
11     2.792520
12     3.025230
13     3.257940
14     3.490660
15     3.723370
16     3.956080
17     4.188790
18     4.421500
19     4.654210
20     4.886920
21     5.119630
22     5.352340
23     5.585050
24     5.817760
25     6.050470
26     6.283180
27     6.515890
28     6.748600
29     6.981310
30     7.214020
31     7.446730
32     7.679440
33     7.912150
34     8.144860
35     8.377570
36     8.610280
37     8.842990
38     9.075700
39     9.308410
40     9.541120
41     9.773830
42    10.006500
43    10.239300
44    10.472000
45    10.704700
46    10.937400
47    11.170100
48    11.402800
49    11.635500
50    11.868200
51    12.100900
52    12.333600
53    12.566400
54    12.799100
Name: radius (kpc), dtype: float64
```

[41]:
```
#for accessing ith row we can use the iloc "i location " method
print(df_rot_curve.iloc[0])
```

```
radius (kpc)                  0.23271
circ velocity (km/s)          3.33000
circ velocity error (km/s)    1.36000
Name: 0, dtype: float64
```

6 Adding a column to the data frame

6.0.1 Adding a column for a systematic error

In some galaxies, the quoted errors are very small and have been underestimated. It is suggested to include a systematic error at the level of 5% for the last data point.

[42]:
```
# Recall the key arguments of the Data frame
df_rot_curve.columns
```

[42]:
```
Index(['radius (kpc)', 'circ velocity (km/s)', 'circ velocity error (km/s)'],
dtype='object')
```

[43]:
```
# Add a new column to the data frame
df_rot_curve['syst_error'] =  0.05*df_rot_curve[ 'circ velocity error (km/s)'].
↪iloc[-1]
```

[44]:
```
#Check that the column was properly added
df_rot_curve.head()
```

[44]:

	radius (kpc)	circ velocity (km/s)	circ velocity error (km/s)	syst_error
0	0.232710	3.33	1.36	0.14
1	0.465421	8.87	4.32	0.14
2	0.698131	11.89	4.51	0.14
3	0.930841	15.76	5.35	0.14
4	1.163550	18.62	5.54	0.14

7 Add the errors in quadrature

[45]:
```
df_rot_curve["total_error"] = np.
↪sqrt(df_rot_curve['syst_error']**2+df_rot_curve['circ velocity error (km/
↪s)']**2)
```

[46]:
```
df_rot_curve.head()
```

[46]:

	radius (kpc)	circ velocity (km/s)	circ velocity error (km/s)	syst_error \
0	0.232710	3.33	1.36	0.14
1	0.465421	8.87	4.32	0.14

```
2      0.698131               11.89                    4.51        0.14
3      0.930841               15.76                    5.35        0.14
4      1.163550               18.62                    5.54        0.14

   total_error
0     1.367187
1     4.322268
2     4.512172
3     5.351831
4     5.541769
```

We can also slice the column HOMEWORK

[47]:
```
df_rot_curve['radius (kpc)'][0:10]
```

[47]:
```
0    0.232710
1    0.465421
2    0.698131
3    0.930841
4    1.163550
5    1.396260
6    1.628970
7    1.861680
8    2.094390
9    2.327100
Name: radius (kpc), dtype: float64
```

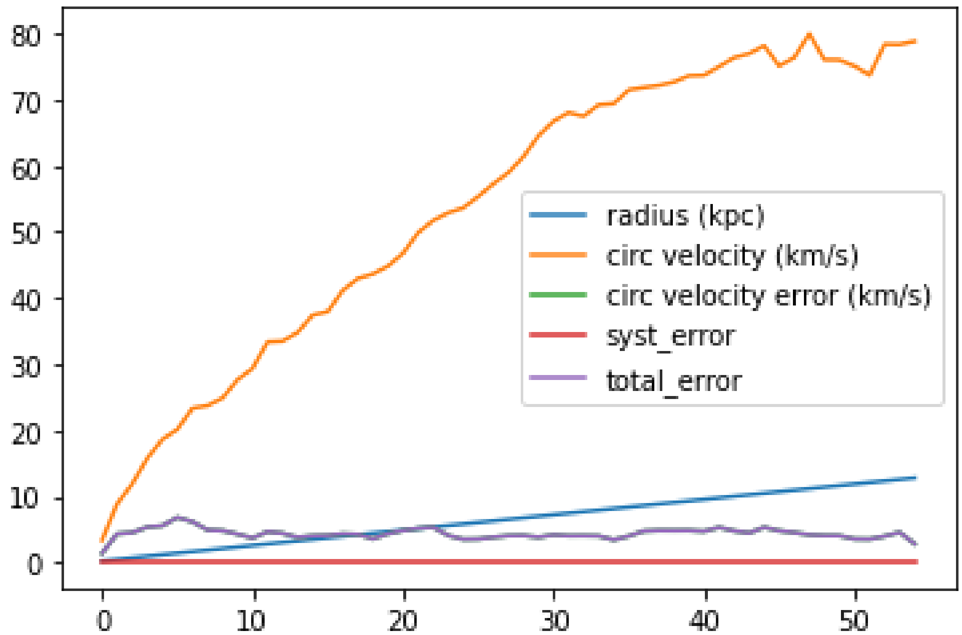
7.1 Ploting Data frames HOMEWORK

Pandas Library has the a ploting method called .plot() . It uses mathplot library to create the different plots.
(See pandas.DataFrame.plot)

We can crate a plot of all of the columns contained in the Data frame, however, it uses as x-axis the index-lablel in the Data frame

[48]:
```
df_rot_curve.plot()
```

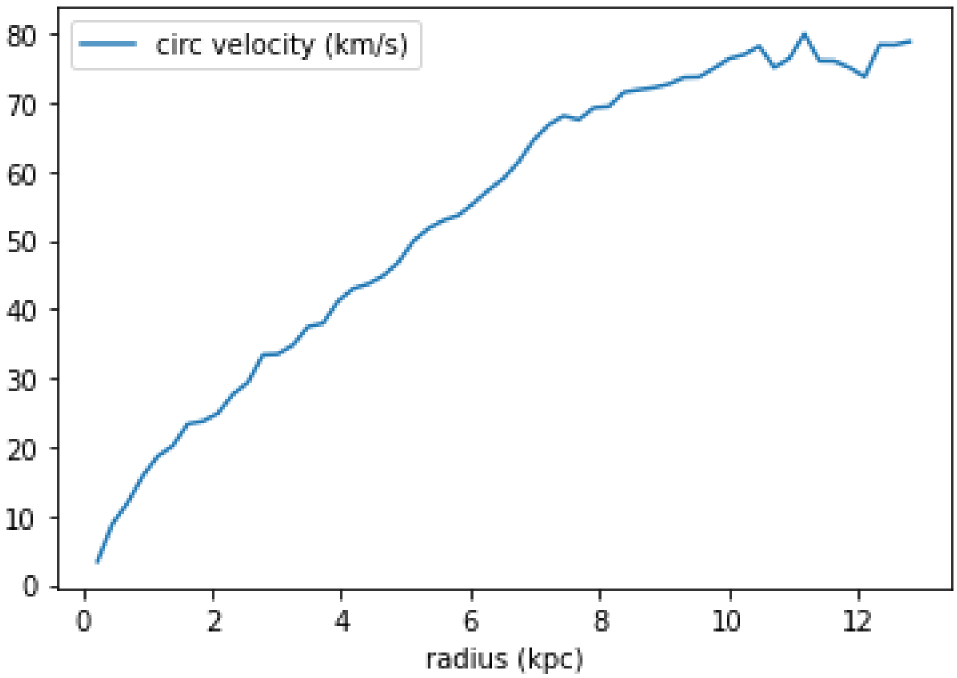
[48]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3d2b073c10>
```



To do more specific plots, for intance a plot of the velocity of rotation as a function of radius, we need to specify in angular paranthesis the keywords for the specific columns. The first keyword correspond to the values assigned to the x-axis

[49]:
```
df_rot_curve.plot('radius (kpc)','circ velocity (km/s)')
```

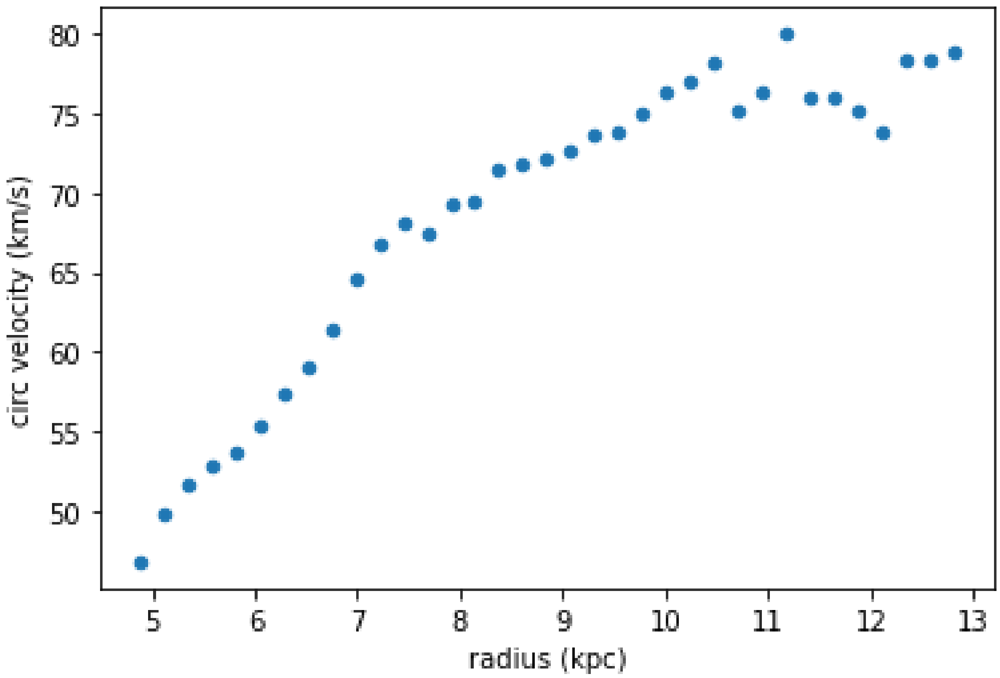
[49]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3d2a772810>



If we wanted to plot a specific slice of the Data frame, we just use our slicing comands before applying the plot method

[50]:
```
df_rot_curve[20:].plot.scatter('radius (kpc)','circ velocity (km/s)')
```

[50]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3d2a7041d0>



8 Fitting functions using SciPy library

The funtion fitting procces can be divided into 4 steps:

1) Collect the data

2) Define objective function (The fucntion that probably will describe well the data. It has free parameters that can be fixed from the fit)

3) Do the Fit

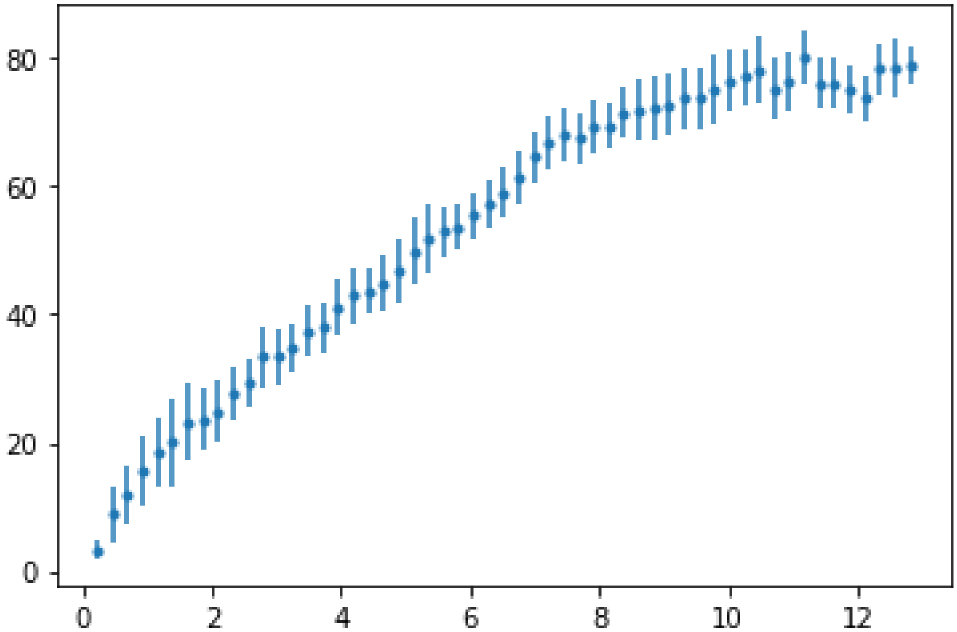
4) Compute the goodness of the fit

1) Collecting the data First we want to select the date (x,y), for which we will do the fitting. The data also have uncertanties in the measured velocity values, and we want our fit function to take into account those uncertaties

[51]:
```
# choose the input and output variables, as well as the uncertaties

r  = df_rot_curve['radius (kpc)']  # measured r vales
v = df_rot_curve['circ velocity (km/s)']  # measured velocity values
dv = df_rot_curve["circ velocity error (km/s)"] # Uncertaties in the measured␣
↪velocities
dvt = df_rot_curve["total_error"] # total error
```

```
[52]: # plot input vs output
plt.errorbar(r, v, yerr=dvt, fmt='.')
plt.show()
```




```
[53]: plt.errorbar?
```

2) Define the true objective function In this tutorial we will fit the rotation curve to a NFW profile , as well as to a Burkert profile. They are characterized by two parameters

$$\rho_{NFW} = \frac{\rho_s}{r/r_s(1 + r/r_s)^2}$$

and

$$\rho_{Burkert} = \rho_0 \frac{r_0^3}{(r + r_0)(r^2 + r_0^2)}$$

Notice that as $r \to 0$, the desity functions behave as

$$\rho_{NFW} \to \rho_s \frac{r_s}{r}$$

and

$$\rho_{Burkert} \to \rho_0$$

Let us see what the density profile look like

```
[54]: from density_profiles import rho_NFW, rho_burkert
```

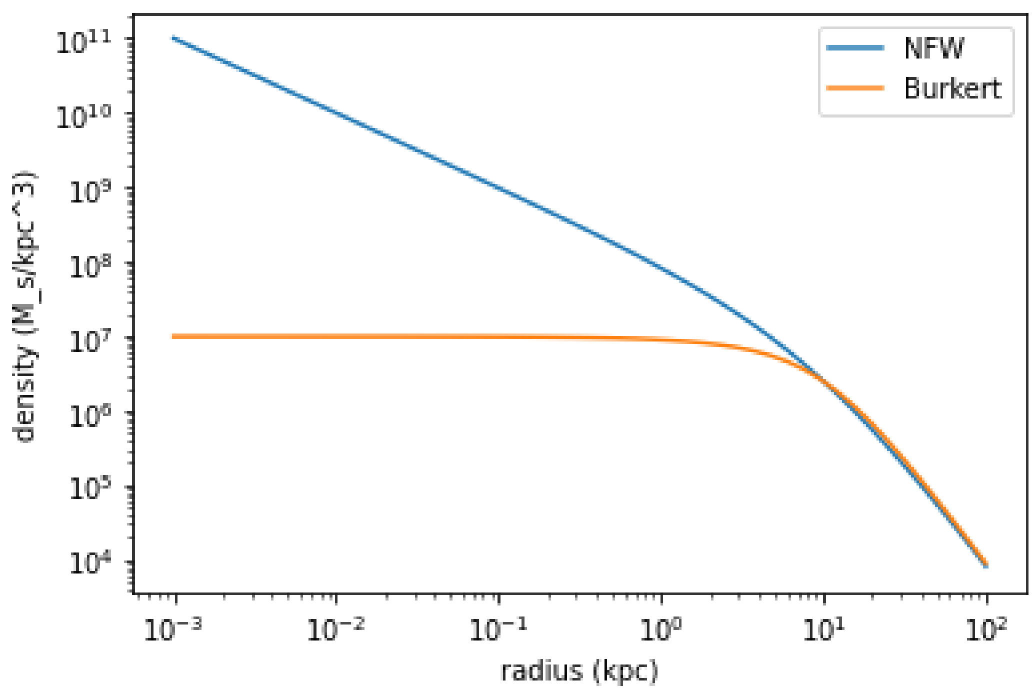
[55]:

```
rhoNFW= rho_NFW(10**7,10)
rhoburkert = rho_burkert(10**7,10)

rd = np.logspace(-3,2)
```

[56]:

```
plt.loglog(rd,rhoNFW(rd), label ="NFW")
plt.loglog(rd,rhoburkert(rd),label = "Burkert")
plt.xlabel('radius (kpc)')
plt.ylabel('density (M_s/kpc^3)')
plt.legend(loc="upper right")
plt.show()
```



The mass functions are given by the volume integral of the DM density function $\rho(r)$

$$M_{NFW}(r) = 4\pi\rho_s r_s^3 \left[\ln\left(1+\frac{r}{r_s}\right) - \frac{r}{r_s}\frac{1}{1+r/r_s}\right]$$

and

$$M_{Burkert}(r) = \pi\rho_0 r_0^3 \left[\ln\left((1+r^2/r_0^2)(1+r/r_0)^2\right) - 2\arctan(r/r_0)\right]$$

The velocity of rotation is then computed from

$$v = \sqrt{\frac{GM(r)}{r}}$$

[57]:
```
# Newton's constant
GN = 4.302e-6 # km^2/s^2*kpc/Msol

def M_NFW(r, r_s, rho_s):
    """NFW mass function"""
    xr = r/r_s
    return 4*np.pi*rho_s*r_s**3*( np.log(1+xr) - xr/(1+xr) )

def M_burkert(r, r_0, rho_0):
    """Burkert mass function"""
    xr = r/r_0
    return np.pi*rho_0*r_0**3*(-2 *np.arctan(xr) + np.log(1+xr**2) + 2*np.
↪log(1+xr) )
```

[58]:
```
M_NFW?
```

With the mass function, we can define the true objective function for the velocity as

[59]:
```
def objective(r, r_s, rho_s):

    """Velocity function for the NFW profile"""

    return np.sqrt(GN* M_NFW(r, r_s, rho_s)/r)


def objective_b(r, r_0, rho_0):

    """Velocity function for the Burkert profile"""

    return np.sqrt(GN* M_burkert(r, r_0, rho_0)/r)
```

8.0.1 Fit the curve

For that we use the curve_fit module in the scipy library

[60]:
```
from scipy.optimize import curve_fit
```

[61]:
```
curve_fit?
```

[62]:
```
# curve fit for NFW
par, c_matrix = curve_fit(objective, r, v, sigma = dvt,p0=np.array([10., 1e7]))
```

```
/home/fabian/anaconda3/lib/python3.7/site-packages/pandas/core/series.py:679:
RuntimeWarning: invalid value encountered in sqrt
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

[63]:
```
# curve fit Burkert
par_b, c_matrix_b = curve_fit(objective_b, r, v, sigma = dvt, p0=np.array([10.,␣
↪1e7]))
```

[64]:
```
# summarize the parameter values
r_s, rho_s = par
print(r_s, rho_s)
r_0, rho_0 = par_b
print(r_0, rho_0)
```

```
747.2453044088161 25679.68740293729
8.236653216336087 9843524.102512754
```

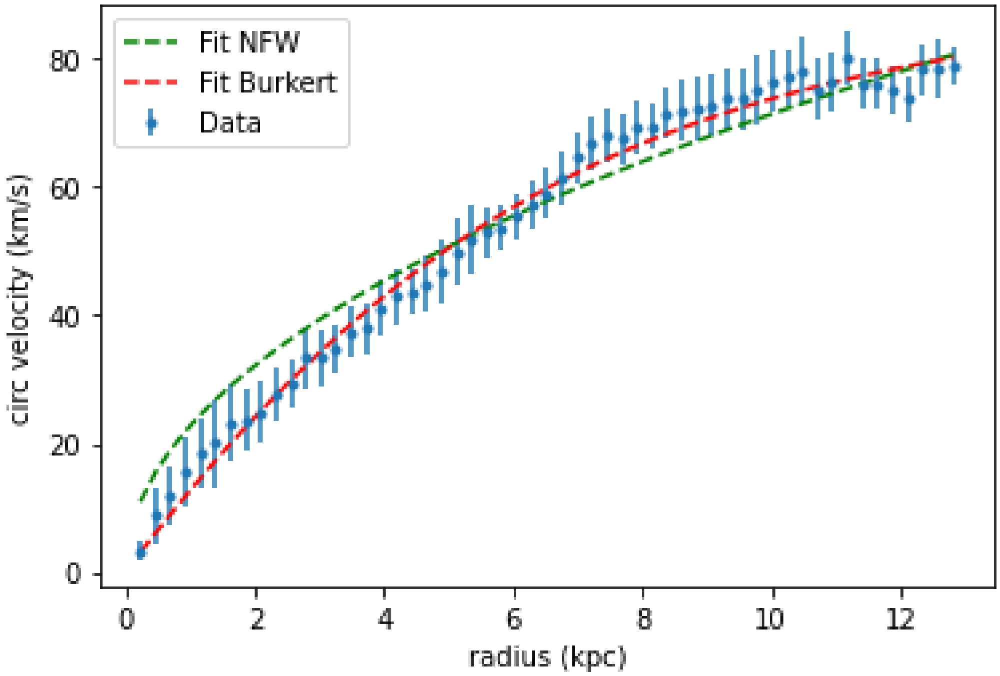
8.0.2 Plot of the fit function

[65]:
```
# Generate v_values from the fit

v_fit_b = objective_b(r, r_0, rho_0) # For Burkert
v_fit = objective(r, r_s, rho_s) # For NFW

#Do the plots

plt.errorbar(r, v, yerr=dvt, fmt='.',label = "Data")  # For the data
plt.plot(r, v_fit, "g", label="Fit NFW",linestyle = '--') # For  NFW
plt.plot(r, v_fit_b, "r", label="Fit Burkert",linestyle = '--')  # For Burkert
# Adding axis labels
plt.xlabel('radius (kpc)')
plt.ylabel('circ velocity (km/s)')
plt.legend(loc="upper left")
plt.show()
```



8.0.3 But, how good is the fitting function?

We can determine the goodness of a fit function by means of a chi_square analysis. That is, given the measured data $y_i \pm \sigma_{y_i}$, for some x_i range, and a fit function $f(x_i)$, the χ^2 function is defined by

$$\chi_n^2 = \sum_{i=1}^{n} \left(\frac{y_i - f(x_i)}{\sigma_{y_i}} \right)^2$$

It is well known that χ_n^2 typical values are

$$\chi_n^2 \approx \mathrm{dof},$$

where dof is the number of degrees of freedom, defined by

$$\mathrm{dof} = \mathrm{n} - \mathrm{n}_{\mathrm{parameters}}.$$

Then, $\chi_n^2 \gg n$ gives a bad fit. On the other hand, $\chi_n^2 \ll n$ indicates that the error bar in the data are wrong.

[66]:
```
def chi_s(y,y_fit,dy):
    """chi_square function"""
    return sum(((y-y_fit)/dy)**2)
```

[67]:
```
#Chi square for NFW
chi_s0 =  chi_s(v,v_fit,dvt)
print(chi_s0)
#Chi square for Burkert
chi_s_b =  chi_s(v,v_fit_b,dvt)
print(chi_s_b)
```

```
92.86707235446156
15.833933203105577
```

Then we can compare the value of χ^2 to the number of degrees of freedom

[68]:
```
def dof(y,par):

    return len(y)-len(par)
```

[69]:
```
# Degrees of freedom for NFW
dof0 = dof(v,par)
# Degrees od freedom for Burkert
dof_b = dof(v,par_b)
```

[70]:
```
print(chi_s0 / dof0)
print(chi_s_b / dof_b)
```

```
1.7522089123483313
0.29875345666236935
```

[71]:
```
#The ratio should be close to 1
```

We can quantify the difference further by checking the goodness of the fit. i.e. the probability that the fit function is the correct function to describe the data For this use the normalized incomplete gamma function distributions, which are included as special functions in scipy library

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

[72]:
```
import scipy.special as sf
```

[73]:
```
#Probabolity for NFW
print(sf.gammainc(dof0/2,chi_s0/2))

#probability for Burkert
print(sf.gammainc(chi_s_b/2,dof_b/2))
```

```
0.9994146140660786
0.9999932350105163
```

8.0.4 A good fit, but not the best! What went wrong? HOMEWORK

For not linear functions, it is then usefull to deffine an initual guess for the parameters when doing de fit using curve_fit . The default value for the initial guess is $p0 = [1.,]$.

The curve_fit function indeed computes the fit for which the χ^2 function is minimized. However, this function can have different local minima, we want to look for the Global minimum.

[74]:
```
curve_fit?
```

Let's try to do the fit with an initial guess

[75]:
```
# curve fit2
p0_guess = np.array([10, 5e7])
par_1, c_matrix_1 = curve_fit(objective, r, v, sigma = dv,p0=p0_guess)

# summarize the parameter values
r_s_1, rho_s_1 = par_1
print(r_s_1, rho_s_1)
```

```
/home/fabian/anaconda3/lib/python3.7/site-packages/pandas/core/series.py:679:
RuntimeWarning: invalid value encountered in sqrt
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

```
598.0422247888918 32205.5747716729
```

[76]:
```
v_fit1 = objective(r,*par_1)
```

[77]:
```
chi_s1 =  chi_s(v,v_fit1,dv)
dof1 = dof(v,par_1)
print(chi_s1)
```

```
93.84231725343906
```

We can again look the chi_square function, and compare it to the number of degrees of freefom.

[78]:
```
chi_s0 / dof0
```

[78]:
```
1.7522089123483313
```

compared to the case without initial guess

[79]:
```
chi_s1 / dof1
```

[79]:
```
1.7706097594988504
```

The ratio is closser to 1, but still need more precision

We can also look at the goodness of the fit

[80]:
```
sf.gammainc(dof0/2,chi_s0/2)
```

[80]: 0.9994146140660786

compare to the case without initial guess

[81]:
```
sf.gammainc(dof1/2,chi_s1/2)
```

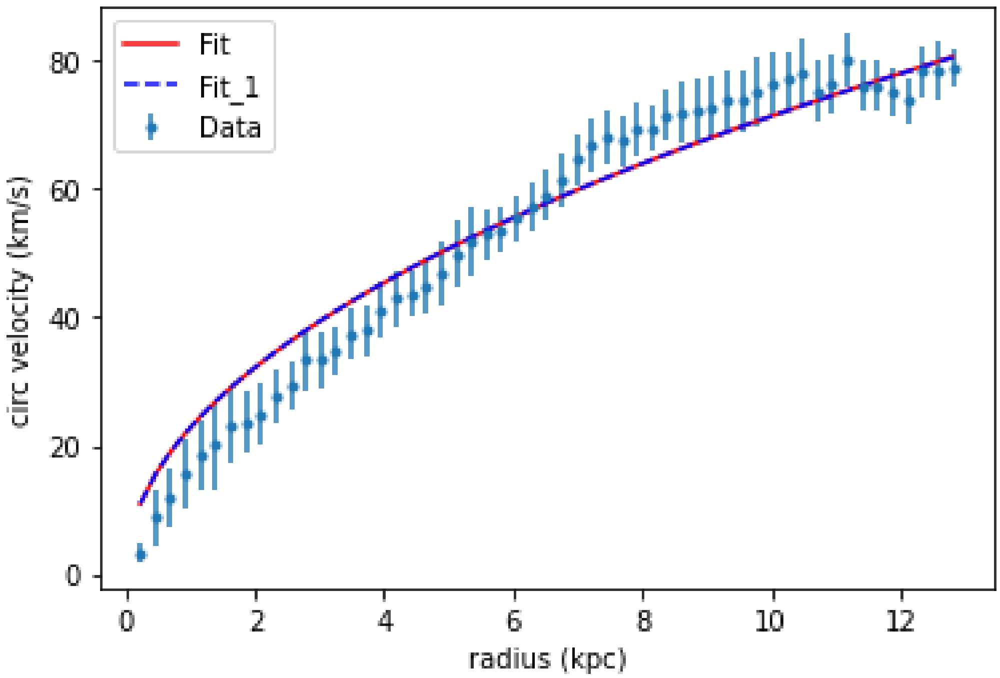
[81]: 0.999535776194912

still not too much difference

We can plot the results on top of each other, to se if something has changed

[82]:
```
plt.errorbar(r, v, yerr=dv, fmt='.',label = "Data")
plt.plot(r, v_fit, "r", label="Fit",linestyle = '-')
plt.plot(r, v_fit1, "b", label="Fit_1",linestyle = '--')

plt.xlabel('radius (kpc)')
plt.ylabel('circ velocity (km/s)')
plt.legend(loc="upper left")
plt.show()
```



Still not the best fit. What could have gone wrong?

9 Excercise COVER

The folder Rotation curves , contains different data sets for the rotation curve for several Dwarf galaxies. See readme for a description of the files.

In this excercise we will fit a rotation curve function for every single galaxy in the folder, and compare the resulting functions.

Let us divide the excercise into different steps:

1) Fit a Burkert velocity curve for the Galaxy IC 2574 So far in this tutorial we have worked only with the Galaxy IC 2574. We fitted a NFW rotation curve and found that the fit we obtained is not the best. We want to start investigating on how to get better fit functions. One possibility is to use a different velocity function to fit the data. In this part of the excercise you are asked to fit a Burkert velocity curve to the Galaxy IC 2574 rotation data.

- Before doing the fit, as explained in the readme file, we want to include a systematic error in the game. Add a new column to the data frame called, "total_error", which is the sum in quadrature of the systematic error, and the uncertanties in the measured velocities.
- Use the total_error as the uncertanties in v, to do the fit.

- Once the fit is done as explained in the turorial, you are required to compute the goodness of the fit, and compare it to the goodness of the fit for the NFW fit.

2) Fit rotation curves for the remaining galaxies Ignoring the contribution of the baryon to the rotation curve data (as we did for Galaxy IC 2574), fit a NFW and a Burkert velocity curve for the remaning galaxies in the folder Rotation curves . Compute the goodness of every fit, and compare them. For which Galaxies is better a NFW fit and for which a Burkert fit? Are these fits good enought to describe the measure data? If not, can you think of what are we missing?

3) Remove baryon contributions to the rotation curve As an extra challenge, as described in readme , remove the baryon contribution to the observed circular velocity data. Then, fit a NFW and a Burkert velocity curve for each galaxy. Compare the goodness of the NFW and Burkert among themselves, and also to the goodness of the fits, when the fit was done without removing the baryon contribution. For which Galaxies is better a NFW fit and for which a Burkert fit? Are these fits good enought to describe the measure data? If not, can you think of what are we missing?

Warning: Notice that the values or r points in the baryon data set are not the same as the r values in the observed circular velocity. Hint: Perhaps you might find usefull interpolating functions .

10 Solution

10.1 Preparing the data

Let us first load the data we are going to use

[83]:
```
folder_path = '/home/fabian/Documents/Git files/python-tutorials/Tutorial 2/
↪Rotation curves/'
file_IC2574_circular = 'RotationCurve_IC2574.csv'
```

```
file_IC2574_baryon = 'RotationCurve_baryons_IC2574.csv' # baryon contribution
full_IC2574_circular = folder_path + file_IC2574_circular
full_IC2574_baryon = folder_path +file_IC2574_baryon # Baryon contribution
```

Now let us create two data frames to unpack these data sets

[84]:
```
df_IC2574_circular = pd.read_csv(full_IC2574_circular)
df_IC2574_baryon = pd.read_csv(full_IC2574_baryon)
```

To have the data measured in the same r_grid, let us create an interpolating function for the stelar and gas velocities

[85]:
```
from scipy.interpolate import interp1d # For 1-D interpolations
```

[86]:
```
# Axis grids
r_baryon = df_IC2574_baryon["radius (kpc)"]
v_star = df_IC2574_baryon["stars circ velocity (km/s)"]
v_gas = df_IC2574_baryon['gas circ velocity (km/s)']
```

[87]:
```
# Interpolations
inter_stars_circ_velocity = interp1d(r_baryon,v_star,kind = 'quadratic')
inter_gas_circ_velocity = interp1d(r_baryon,v_gas,kind = 'quadratic')
```

Let us add two colums to the df_IC2574_circular data frame, with has the baryon velocities evaluated at the same position as the measured measured circular velocity

[88]:
```
df_IC2574_circular["stars circ velocity (km/s)"] = ␣
 ↪inter_stars_circ_velocity(df_IC2574_circular["radius (kpc)"])
```

[89]:
```
df_IC2574_circular["gas circ velocity (km/s)"] = ␣
 ↪inter_gas_circ_velocity(df_IC2574_circular["radius (kpc)"])
```

Let us now add a new column with the systematic error

[90]:
```
df_IC2574_circular['syst_error'] =  0.05*df_IC2574_circular[ 'circ velocity␣
 ↪error (km/s)'].iloc[-1]
```

Now add a column with the total error, as the sum in quadrature of the systematic error, and the error in the mearused velocity

[91]:
```
df_IC2574_circular["total_error"] = np.
 ↪sqrt(df_IC2574_circular['syst_error']**2+df_IC2574_circular['circ velocity␣
 ↪error (km/s)']**2)
```

[92]:
```
df_IC2574_circular.head()
```

[92]:
```
   radius (kpc)  circ velocity (km/s)  circ velocity error (km/s)  \
0      0.232710                  3.33                        1.36
```

```
1      0.465421                  8.87                           4.32
2      0.698131                 11.89                           4.51
3      0.930841                 15.76                           5.35
4      1.163550                 18.62                           5.54

   stars circ velocity (km/s)  gas circ velocity (km/s)  syst_error  \
0                    3.048860                  2.051611        0.14
1                    5.447722                  4.196565        0.14
2                    7.264598                  6.273282        0.14
3                    9.047033                  6.981577        0.14
4                   10.911456                  6.058803        0.14

   total_error
0     1.367187
1     4.322268
2     4.512172
3     5.351831
4     5.541769
```

Let us finally add a column with the DM velocity, which is given by

$$v_{DM} = \sqrt{v_c^2 - v_{star}^2 - v_{gas}^2}.$$

We have to be careful, since v_{DM}^2 could be negative, if that is the case, we choose to change sign of v_{gas}^2.

[93]:
```
vel_s = np.array([] )
v_s_c = df_IC2574_circular['circ velocity (km/s)']**2
v_s_star = df_IC2574_circular['stars circ velocity (km/s)']**2
v_s_gas = df_IC2574_circular['gas circ velocity (km/s)']**2

for i in range(len(df_IC2574_circular)):
    test = v_s_c[i] - v_s_star[i]-v_s_gas[i]

    if test <0:
        new_test = v_s_c[i] - v_s_star[i] + v_s_gas[i]
        vel_s = np.append(vel_s,new_test)

    else:
        vel_s = np.append(vel_s,test)
v_DM =  np.sqrt(vel_s)
```

[94]:
```
# Now add the new column
df_IC2574_circular["DM circ velocity (km/s)"] = v_DM
```

[95]:
```
df_IC2574_circular.head()
```

```
[95]:    radius (kpc)  circ velocity (km/s)  circ velocity error (km/s)  \
0      0.232710                  3.33                        1.36
1      0.465421                  8.87                        4.32
2      0.698131                 11.89                        4.51
3      0.930841                 15.76                        5.35
4      1.163550                 18.62                        5.54

   stars circc velocity (km/s)  gas circ velocity (km/s)  syst_error  \
0                  3.048860                  2.051611        0.14
1                  5.447722                  4.196565        0.14
2                  7.264598                  6.273282        0.14
3                  9.047033                  6.981577        0.14
4                 10.911456                  6.058803        0.14

   total_error  DM circ velocity (km/s)
0     1.367187                 2.449992
1     4.322268                 5.602506
2     4.512172                 7.017382
3     5.351831                10.852944
4     5.541769                13.817939
```

10.2 Defining the objective functions

```
[96]: def v_DM_NFW(r, r_s, rho_s):

          """ Dark matter velocity function for the NFW profile"""
          # Newton's constant
          GN = 4.302e-6 # km^2/s^2*kpc/Msol

          def M_NFW(r, r_s, rho_s):
              """NFW mass function"""
              xr = r/r_s
              return 4*np.pi*rho_s*r_s**3*( np.log(1+xr) - xr/(1+xr) )

          return np.sqrt(GN* M_NFW(r, r_s, rho_s)/r)

      def v_DM_Burkert(r, r_0, rho_0):
          """Dark Matter velocity function for the Burkert profile"""
          # Newton's constant
          GN = 4.302e-6 # km^2/s^2*kpc/Msol

          def M_burkert(r, r_0, rho_0):
              """Burkert mass function"""
              xr = r/r_0
              return np.pi*rho_0*r_0**3*(-2 *np.arctan(xr) + np.log(1+xr**2) + 2*np.
       ↪log(1+xr) )
```

```python
    return np.sqrt(GN* M_burkert(r, r_0, rho_0)/r)
```

10.3 Doing the fit

[97]:
```python
from scipy.optimize import curve_fit
```

[98]:
```python
r_val = df_IC2574_circular["radius (kpc)"]
dv_val =  df_IC2574_circular["total_error"]
```

[99]:
```python
# curve fit for NFW
par_NFW, c_matrix_NFW = curve_fit(v_DM_NFW, r_val, v_DM, sigma =dv_val,p0=np.
↪array([10., 1e7]))
```

```
/home/fabian/anaconda3/lib/python3.7/site-packages/pandas/core/series.py:679:
RuntimeWarning: invalid value encountered in sqrt
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

[100]:
```python
# curve fit for Burkert
par_Burkert, c_matrix_Burkert = curve_fit(v_DM_Burkert, r_val, v_DM, sigma␣
↪=dv_val,p0=np.array([10., 1e10]))
```

[101]:
```python
v_DM_NFW_fit = v_DM_NFW(r_val, par_NFW[0], par_NFW[1])
v_DM_Burkert_fit = v_DM_Burkert(r_val, par_Burkert[0], par_Burkert[1])
```

[102]:
```python
par_Burkert
```

[102]:
```
array([1.05927550e+01, 5.08979143e+06])
```

10.4 Goodness of the fit

[103]:
```python
def chi_s(y,y_fit,dy):
    """chi_square function"""
    return sum(((y-y_fit)/dy)**2)
def dof(y,par):

    return len(y)-len(par)
```

[104]:
```python
# For NFW
gf_NFW = chi_s(v_DM,v_DM_NFW_fit,dv_val)/dof(v_DM,par_NFW)
# For Burkert
gf_Burkert = chi_s(v_DM,v_DM_Burkert_fit,dv_val)/dof(v_DM,par_Burkert)
```

[105]:
```python
import scipy.special as sf
```

[106]:
```python
#Probabolity for NFW
print(sf.gammainc(dof(v_DM,par_NFW)/2,chi_s(v_DM,v_DM_NFW_fit,dv_val)/2))
```

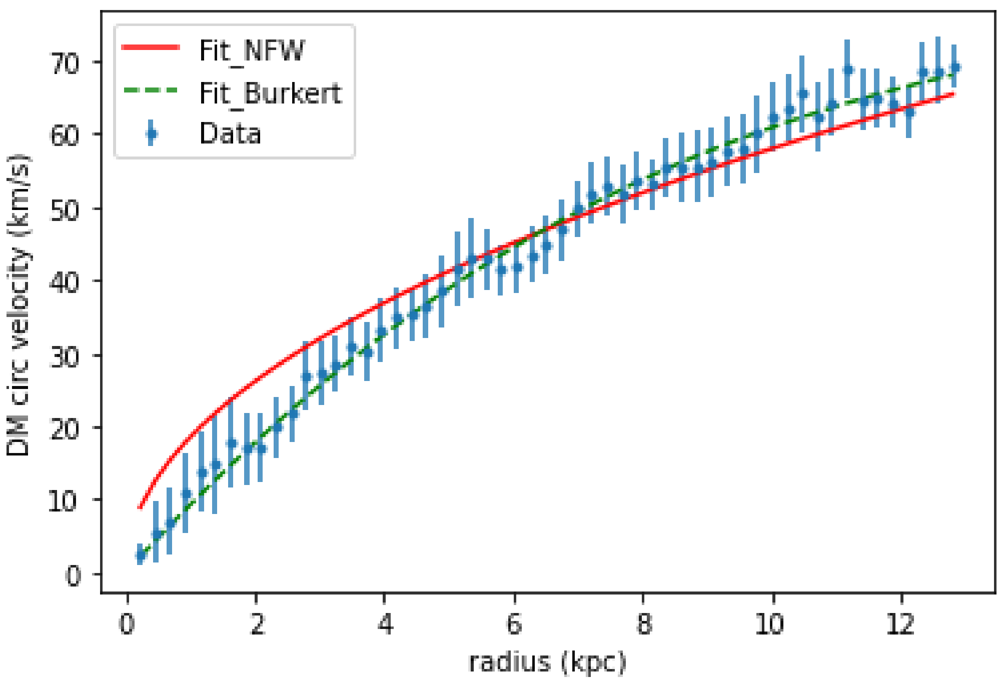
```
#probability for Burkert
print(sf.gammainc(chi_s(v_DM,v_DM_Burkert_fit,dv_val)/2,dof(v_DM,par_Burkert)/
 ↪2))
```

```
0.9733336688055315
0.9999999790100592
```

11 Plot of the results

[107]:
```
print("Goodness of the fit. NFW:", gf_NFW, ". For Burkert:",gf_Burkert)
plt.errorbar(r_val,v_DM , yerr=dv_val, fmt='.',label = "Data")
plt.plot(r_val, v_DM_NFW_fit, "r", label="Fit_NFW",linestyle = '-')
plt.plot(r_val, v_DM_Burkert_fit, "g", label="Fit_Burkert",linestyle = '--')
plt.xlabel('radius (kpc)')
plt.ylabel('DM circ velocity (km/s)')
plt.legend(loc="upper left")
plt.show()
```

```
Goodness of the fit. NFW: 1.4083615923568105 . For Burkert: 0.16340634132534845
```



[ ]:
```

```