Python4LSC - main regression code

Siobhan Tobin

February 23, 2018

Contents

Note: code line numbers in this document do
not match the line numbers in the actual code.
They are included here for easy in-text referencing
of specific lines, and to indicate in what order the code
generally appears.

1 Overview

The *python4LSC* suite of programs was developed in 2017 to automate much of the analysis of processed data $4\pi\beta - \gamma$ liquid scintillation measurements. The original data processing is done through the *LSC Converter* and *LSC Viewer* programs. The result is a csv file for each threshold voltage, which contains counts of single β and γ events, logical double β events, logical double $\beta - \gamma$ coincidences, triple β events, triple $\beta - \gamma$ coincidences, start and end times of measurements, real time and live time.

The *python4LSC* family consists of five basic functional programs and assorted variations:

1. the `pandas` *user* program, which is the starting point for LSC analysis, and one of only two programs the user 'should' edit;
2. the `pandas` *main* program, responsible for number crunching of measurements. It is called from the *user* program and outputs two xlsx files;
3. the `pandas` *regression* program, which fits functions to data and calculates extrapolated activity concentration values. It is called from the *user* program and outputs one xlsx file;
4. the *PlotStuff user* program, which is the starting point for making pictures once the above is complete. It is the second program that the user 'should' edit;
5. the *PlotStuff 1,2,3,4* programs, which are called from the *PlotStuff user* program. Each outputs three png files.

1.1 User program summary

We start with the *user* progam, which gives the following options:

- logical doubles or triples data to be used for calculations;
- the user can supply a manually-specified background xlsx file or have background count rates calculated for each threshold based on a background csv file for each threshold;
- the theoretical standard deviation can be used to weight points for regression, or the observed standard deviation can be used.

This program is where the location and names of the source and background data are specified.

1.2 Main program summary

The user script then calls the corresponding *main* program, which turns the measurement data into decay- and background-corrected count rates (N_β, N_γ, N_C for β, γ and

coincidence count rates respectively). Uncertainties are also calculated. This information is written to an xlsx file ('threshold data' file). These quantities are then used to calculate averaged quantities for each threshold for plotting, along with propagated uncertainties:

- $N_\beta N_\gamma / N_C$ and $N_\gamma / N_C - 1$, with the former on the vertical axis and the latter on the horizontal;
- N_β and $1 - N_C / N_\gamma$, with the former on the vertical axis and the latter on the horizontal.

These averaged quantities are saved in another xlsx file ('regression data file'). This is the end of the *main* program.

1.3 Regression program summary

The script calls the *regression* program. This reads in the data in the regression data file and fits linear and cubic functions to this data. Fits are made to $N_\beta N_\gamma / N_C$ and $N_\gamma / N_C - 1$ values, and also N_β and $1 - N_C / N_\gamma$ values, using both least squares regression and orthogonal distance regression. The chief difference between these methods is that orthogonal distance regression takes into account 'horizontal' as well as 'vertical' uncertainties, while least squares only deals with 'vertical' uncertainties. Orthogonal distance regression is also harder computationally, and uses the least squares fit as an initial estimate. The values can be weighted by either the theoretical standard deviation or the observed standard deviation.

All of the fit parameters are saved in an xlsx file across three worksheets: one for linear fit parameters, one for cubic fit parameters, and one as a summary showing activity concentrations calculated from intercepts. Estimated uncertainties from parameter covariance matrices are also included.

2 User program

The *user* program is designed so that the user can input all required information and run the programs of their choice in the same place.

First, all packages needed for *Python4LSC* programs are imported.

```
import os
import pandas as pd
import numpy as np
import datetime
from scipy.optimize import curve_fit
import scipy.odr.odrpack as odear
import sys
```

The constants to be provided are: reference date and time, half-life of the radioisotope, mass and dilution factor of the sample, the branch ratio correction factor, the source name, the resolving time, the dead time, and the gamma shift. While the last three are not strictly necessary for analysis, they are printed in the 'fit information' xlsx output file from the *regression* program, allowing the user to keep track of how different data processing affects the extrapolated activities. The source name is included in the name of all output files. **The program will overwrite any existing files with the same name – if in doubt, add a suffix to the source name.**

```
# set reference date yyyy,mm,dd,hh,MM,SS,fractions of second. leave out
# leading zeroes, e.g. June is 6 not 06
refdatetime=datetime.datetime(2017,10,11,12,0,0,0)
# Half life in seconds
halflifeseconds=12.7004*60*60
# Dilution factor
dilution=8.3028881
# Active solution mass in mg
mass=26.3194
# Branch ratio
branchratio=1
# What do you want the output files to be called?
# Source name is generally good
outputfilename='D3LS3-sb'
# Resolving time in ns
rt=200
# Dead time in us
dt=50
# Gamma shift
gs=-2750
```

Then the user selects the options they want – the *user* program can be re-run multiple times for the same data with different options selected.

```
# Doubles or triples? Select one at a time
doubles_data = 1 # 0=NO 1=YES
triples_data = 0 # 0=NO 1=YES
# Manually specified background?
specified_background = 0 # 0=NO 1=YES
# weighting of points? Select one at a time
```

```
35 theoretical_sd = 0 # 0=NO 1=YES
36 observed_sd = 1 # 0=NO 1=YES
```

Next is to point the program to where the files are located, both source data and background data. The directory, **which must be contained within the working directory of the *python4LSC* programs used**, is entered just after the constants:

```
37 # DIRECTORY, where are the csv files located?
38 rtdt1dir="csvs"
```

When it comes to selecting the files for source data and background data, there is the option of using the `speedyfilegrabber`. Only use this if each source measurement csv file has its own corresponding background csv. For example, my source files might be source1_30mV.csv, source1_50mV.csv, and source1_70mV.csv, with accompanying backgrounds bkg_30mV.csv, bkg_50mV.csv, and bkg_70mV.csv. Then I could specify the following prefixes:

```
39 """ DO YOU WANT TO USE THE SPEEDY FILE GRABBER? 0=NO, 1=YES """
40 speedyfilegrabber=1
41 bkgprefix=bkg #for using speedy file grabber section
42 dataprefix=source1 #for using speedy file grabber section
```

and the `speedyfilegrabber` would pick all files that start with `source1` from the directory `rtdt1dir` as specified above and match them with all files starting with `bkg`. If your files are not named in this way, or you wish to use the same background file for multiple source files, or you want to make up a specific background (i.e. the *specified background* option), then leave `speedyfilegrabber=0`. The `speedyfilegrabber` will come up with an error if there is a different number of csv files with the prefix `bkg` compared to the csv files with the prefix `source1`.

The very straightforward option is to just type all the file names in. These files should all be located in the directory `rtdt1dir`.

```
43 """ ONLY USE THIS SECTION IF NOT USING SPEEDY FILE GRABBER """
44 # FILES
45 thresh1A="Cu64-D3-LS3_30mV.csv"
46 thresh1B="Cu64-D3-LS3_50mV.csv"
47 thresh1C="Cu64-D3-LS3_70mV.csv"
48 thresh1D="Cu64-D3-LS3_90mV.csv"
49 thresh1E="Cu64-D3-LS3_110mV.csv"
50 thresh1F="Cu64-D3-LS3_130mV.csv"
51 thresh1G="Cu64-D3-LS3_150mV.csv"
52 thresh1H="Cu64-D3-LS3_180mV.csv"
53 thresh1I="Cu64-D3-LS3_210mV.csv"
54 thresh1J="Cu64-D3-LS3_240mV.csv"
55 thresh1K="Cu64-D3-LS3_270mV.csv"
56 thresh1L="Cu64-D3-LS3_300mV.csv"
57 thresh1M="Cu64-D3-LS3_30mV_REP.csv"
58 thresh1N="Cu64-D3-LS3_70mV_REP.csv"
59 thresh1O=0
60 thresh1P=0
61 thresh1Q=0
```

If you are specifying the background manually, then this needs to be in the form of a single xlsx file. This file should have eight columns (0 - 7 using the computer science convention of starting to count from 0), with the following labels:

- column 0: index
- column 1: threshold
- column 2: backLDr8 (or backABCr8 for triples)
- column 3: unc backLDr8 (or unc backABCr8)
- column 4: backXr8
- column 5: unc backXr8
- column 6: backLDXr8 (or backABCXr8)
- column 7: unc backLDXr8 (or unc backABCXr8)

An example is shown in Figure 1.

backgroundall

	A	B	C	D	E	F	G	H
1	index	threshold	backLDr8	unc backLDr8	backXr8	unc backXr8	backLDXr8	unc backLDXr8
2	0	30	9.44730607	0.03808702	0.4767	0.017	0.00438624	0.0015
3	1	50	8.84030589	0.07366598	0.4767	0.017	0.00422137	0.0015
4	2	70	8.69525083	0.04716398	0.4767	0.017	0.0040565	0.00145
5	3	90	8.47576886	0.05861043	0.4767	0.017	0.00389163	0.0014
6	4	110	8.21508676	0.05442836	0.4767	0.017	0.00372676	0.00135
7	5	130	7.47356944	0.05532396	0.4767	0.017	0.00356188	0.0013
8	6	150	7.04547747	0.03710385	0.4767	0.017	0.00339701	0.00125
9	7	180	6.02134622	0.04335214	0.4767	0.017	0.0031497	0.0012
10	8	210	5.27100291	0.03621254	0.4767	0.017	0.0029024	0.00115
11	9	240	4.67969224	0.03915761	0.4767	0.017	0.00265509	0.0011
12	10	270	4.37713758	0.03277393	0.4767	0.017	0.00240778	0.00105
13	11	300	4.24716654	0.03912661	0.4767	0.017	0.00216047	0.001
14	12	30	9.44730607	0.03808702	0.4767	0.017	0.00438624	0.0015
15	13	70	8.69525083	0.04716398	0.4767	0.017	0.0040565	0.00145

Figure 1: The index associated with each threshold in the background file matches the position of the source measurement taken at the same threshold in the list (lines 45-58).

```
62 """ If you are specifying backgrounds with a single excel spreadsheet """
63 # i.e. specfied_background = 1
64 """ enter the name of the spreadsheet here... """
65 # BACKGROUND EXCEL FILE
66 backgroundexcel="backgroundall.xlsx"
```

If you are not specifying a background manually, then you can type those file names in. As you can see with this example, the file names would not have worked with the `speedyfilegrabber`, because the user wants to use the 30mV and 70mV background csv files twice (`back1A` and `back1M`, `back1C` and `back1N`) to account for the repeat measurements `thresh1M` and `thresh1N`.

```
""" Or if you are just analysing csv files to get the backgrounds for
each threshold, put those files here... """
# i.e. specfied_background = 0
# BACKGROUND FILES
# back1A must be the background for thresh1A etc
back1A="1872-Cu64-bkg171016-30mV.csv"
back1B="1872-Cu64-bkg171016-50mV.csv"
back1C="1872-Cu64-bkg171016-70mV.csv"
back1D="1872-Cu64-bkg171016-90mV.csv"
back1E="1872-Cu64-bkg171016-110mV.csv"
back1F="1872-Cu64-bkg171016-130mV.csv"
back1G="1872-Cu64-bkg171016-150mV.csv"
back1H="1872-Cu64-bkg171016-180mV.csv"
back1I="1872-Cu64-bkg171016-210mV.csv"
back1J="1872-Cu64-bkg171016-240mV.csv"
back1K="1872-Cu64-bkg171016-270mV.csv"
back1L="1872-Cu64-bkg171016-300mV.csv"
back1M="1872-Cu64-bkg171016-30mV.csv"
back1N="1872-Cu64-bkg171016-70mV.csv"
back1O=0
back1P=0
back1Q=0
```

The remainder of the *user* program calls the *main* and *regression* programs as appropriate. The *main* program has four variations: *doubles_main*, *doubles_main_specifiedbackground*, *triples_main*, and *triples_main_specifiedbackground*. It is not necessary to use all four programs, usually one or two options will be appropriate for the data. So, only one or two of these main programs would need to be copied to the working directory. None of this code needs editing – select your main program options using the user input section above.

```
#~~~~~~~~~~~~~~~~~ NO MORE USER INPUT REQUIRED UNLESS PROMPTED ~~~~~~~~~~~~~~~

# Which main program to run???
if doubles_data==1:
    DorT='doubles'
    if specified_background==1:
        Sb='_SB'
        print("Calling␣pandas4LSCdoubles_main_specifiedbkg")
        import pandas4LSCdoubles_main_specifiedbkg
    else:
        Sb=''
        print("Calling␣pandas4LSCdoubles_main")
        import pandas4LSCdoubles_main

if triples_data==1:
    DorT='triples'
```

```
    if specified_background==1:
        Sb='_SB'
        print("Calling pandas4LSCtriples_main_specifiedbkg")
        import pandas4LSCtriples_main_specifiedbkg
    else:
        Sb=''
        print("Calling pandas4LSCtriples_main")
        import pandas4LSCtriples_main
```

There is only one regression program: *pandas4LSC_regression.py*. This should work with outputs of any of the four main programs. The regression program just needs to be told which data to use, and which standard deviation to use to weight the points for the regression. The user has made the choice of either the theoretical standard deviation, or the observed standard deviation, in the user input section above. The 'regression data file' will be found by the program using variables previously defined by the user. Again, you should not need to edit this part.

```
# Data for regression
filename="{0}_rt{1}dt{2}_RegData_{3}{4}.xlsx".format(outputfilename,rt,dt,
DorT,Sb)
regdf=pd.read_excel(filename,header=0,index_col=0)

# Which regression code to run???

if theoretical_sd==1:
    StandDev='TSD'
    print("Calling pandas4LSC_regression")
    import pandas4LSC_regression

if observed_sd==1:
    StandDev='OSD'
    print("Calling pandas4LSC_regression")
    import pandas4LSC_regression
```

Finally, the user program (and so the whole *python4LSC* process) wraps up with some print statements.

```
if specified_background==1:
    print("Background specified by user")
print("{0} data used. Points weighted by {1}".format(DorT,StandDev))
print("Finished python4LSC. See you later :) ")
```

3 Main program: basic number crunching

The vast majority of the number crunching of csv files is performed by a function `threshold`, so-called because it takes the multiple measurements taken at a single threshold value and outputs a variety of calculated quantities for each of these measurements. The `threshold` function differs slightly if it is in the *pandas4LSCtriples_main.py* program compared to the *pandas4LSCdoubles_main.py* program. Here we will step through each line of the `threshold` function as it appears in the doubles program.

There is also a difference in the way the `threshold` function deals with data from background measurements depending on whether you choose to manually input the background count rates into a spreadsheet (one spreadsheet for all thresholds, i.e. select *pandas4LSCdoubles_main_specifiedbackground.py*) or use data from background measurements at each threshold (each threshold has its own csv file and the *pandas4LSCdoubles_main.py* program is selected). This is explained further below.

In summary, the `threshold` function calculates decay- and background-corrected count rates, and products of these quantities, for each measurement at each threshold.

3.1 Dealing with the background

3.1.1 Backgrounds calculated from csv files

Background count rates are calculated from csv files for each measurement at each threshold. **Notice all the annoying spaces when referring to columns of the read-in csv file? The LSC Viewer writes the files with these spaces, so don't change them!**

```
1  def background(df):
2      df.head()
3      # background ABC Rate
4      df['ABCr8']=df['␣ABC']/df['␣Live']
5      # background X Rate
6      df['Xr8']=df['␣X']/df['␣Live']
7      # background ABCX Rate
8      df['ABCXr8']=df['␣ABCX']/df['␣Live']
9          # background LD Rate
10     df['LDr8']=df['␣LD']/df['␣Live']
11         # background LDX Rate
12     df['LDXr8']=df['␣LDX']/df['␣Live']
13     return df
```

For each threshold, mean background count rates can be calculated. Here, background γ rate is shown as an example.

```
14 def bckgrndX(group):
15     """ Mean X background """
16     d = group['Xr8']
17     c1 = group.count(axis=0)[0]
18     return d.sum()/c1
```

For high source count rates and efficiencies, the uncertainty in the background is usually negligible. Nevertheless the variance for each mean is calculated.

```
19 def bckgrndX_var(group):
20     """ variance X background """
21     d = group['Xr8']
22     m = bckgrndX(group)
23     c1 = group.count(axis=0)[0]
24     return ((d-m)**2).sum()/(c1-1)
```

3.1.2 User-specified background

A single spreadsheet, with a row for each threshold, may be used instead of multiple csv files for background correction. While there are no additional functions written to do this, several functions are edited to take an extra argument, namely the row number that corresponds to a particular threshold.

3.2 'Threshold' function

This is the real workhorse of the code. It takes in background and source data for a single threshold, and outputs a dataframe consisting of a single row containing calculated values for that threshold. Here we break it down, step by step.

The following convention is used:

- γ = gamma, β = ABC or LD, C = ABCX or LDX, depending on whether doubles or triples data is being used;
- C refers to raw counts, e.g. C_{LD} is the logical double counts;
- R refers to uncorrected rate, e.g. R_γ is the gamma rate;
- B refers to background rate, e.g. B_C is the background coincidence rate;
- N refers to corrected rate, e.g. N_C is the decay- and background-corrected coincidence count rate.

The source mass is m, the dilution factor is A, the half life in seconds is $t_{1/2}$, the reference datetime is R, the live time is τ, the finish time of a measurement is τ_f, the real time is $\tau_\mathcal{R}$, and the time difference to the reference date time is τ_δ. Furthermore, weights are related to uncertainties by:

$$\text{weight} = \frac{1}{\text{uncertainty}}.$$

For backgrounds calculated from csv files, the function starts:

```
25 def threshold(df,backdf):
26     backdf.head()
27     backdf=background(backdf)
28     df.head()
29     df['backLDr8']=bckgrndLD(backdf)
30     df['backXr8']=bckgrndX(backdf)
31     df['backLDXr8']=bckgrndLDX(backdf)
```

This is taking the background dataframe, and calculating background count rates for each measurement in it using the `background` function. Then mean background rates (calculated with the function `bckgrnd`) are added to the dataframe `df`, which is basically the large spreadsheet where we are storing all these numbers. Each mean background rate is placed in its corresponding column, e.g. the column `'backLDr8'` is defined by `df['backLDr8']=bckgrndLD(backdf)`.

For user-specified background, the start looks like this instead:

```
32 def threshold(df,backdf,i_index):
33     thresh_backdf=backdf.iloc[[i_index]]
34     df['backLDr8']=thresh_backdf['backLDr8'].sum()
35     df['backXr8']=thresh_backdf['backXr8'].sum()
36     df['backLDXr8']=thresh_backdf['backLDXr8'].sum()
```

The `threshold` function takes an extra argument (`i_index`), which is used to refer to a specific row in the dataframe referred to as `backdf`. This single row is then a new dataframe `thresh_backdf`. As you will see later, `backdf` is basically the background xlsx file. The background count rates are then input into the main dataframe `df`. You cannot define a dataframe column in terms of a single dataframe entry, hence why we sum each column in the background dataframe (remember there is only one entry in each column).

The next section of the `threshold` function deals with time differences. For each measurement at a threshold, the time difference between the measurement time and the reference time must be calculated. The reference time has already been defined as a `datetime` object in the user code. Specific points in time are `datetime` objects in Python, amounts of time to add and subtract are `timedeltas`. To account for the buffer between the time when the measurement 'starts' and when data actually starts being collected, the following equation is used to calculate time difference:

$$\tau_\delta = R - (\tau_f - \tau_\mathcal{R})$$

where τ_δ is time difference, R is the reference time, τ_f is the finish time, and $\tau_\mathcal{R}$ is the real time.

```
37     #calculate time difference between start time and reference time
38     df['Real␣as␣s']=pd.to_timedelta(df['␣Real'],unit='s')
39     df['timedif']=refdatetime-(df['␣Finished']-df['Real␣as␣s'])
40     #change to seconds
41     df['timedif(s)']=df['timedif']/pd.Timedelta(seconds=1)
```

As Excel and Python deal with times in different ways, and Excel has more limited time resolution, we work with seconds, and however many decimal places of a second. Line 41 above divides a column of `timedeltas` by a `timedelta` = 1 s to give a number of seconds in floating point.

Now we can calculate the decay factor, D:

$$D = \ln 2 \cdot \tau_\mathcal{R}/t_{1/2} \cdot \left(1 - \exp\left(-\ln 2 \cdot \tau_\mathcal{R}/t_{1/2}\right)\right)^{-1} \cdot 2^{-\tau_\delta/t_{1/2}}.$$

```
    #calculate decay factor
    df['Decay␣Factor']=(np.log(2)*df['␣Real']/halflifeseconds/
              (1-np.exp(-np.log(2)*df['␣Real']/halflifeseconds))*
              0.5**(df['timedif(s)']/halflifeseconds))
```

Using the decay factor and the background count rates, corrected count rates are calculated. Here, doubles data are used as an example.

$$N_\beta = \left(\frac{C_{LD}}{\tau} - B_{LD}\right) \cdot D$$

```
    # LD Rate, decay and background corrected
    df['LDr8']=(df['␣LD']/df['␣Live']-df['backLDr8'])*df['Decay␣Factor']
    # X Rate, decay and background corrected
    df['Xr8']=(df['␣X']/df['␣Live']-df['backXr8'])*df['Decay␣Factor']
    # LDX Rate, decay and background corrected
    df['LDXr8']=(df['␣LDX']/df['␣Live']-df['backLDXr8'])
  *df['Decay␣Factor']
```

With N_β, N_γ and N_C calculated, the `threshold` function can calculate the stuff we're really interested in.

The $\beta\gamma/C$ per unit mass is

$$\frac{N_\beta N_\gamma}{N_C} \cdot \frac{A}{m}$$

with uncertainty

$$u_{N_{\beta\gamma/C}/m} = \sqrt{\frac{1}{C_\beta} + \frac{1 - N_\beta/N_\gamma}{C_C}} \cdot \frac{N_\beta N_\gamma}{N_C} \cdot \frac{A}{m}$$

```
    #BetaGamma/Coinc /mass, dilution factor included
    df['BeGa/Co/m']= df['LDr8']*df['Xr8']*dilution/(df['LDXr8']*mass)
    df['uncBeGa/Co/m']=(np.sqrt((1/df['␣LD'])+
              (1-df['LDXr8']/df['Xr8'])/df['␣LDX'])*df['BeGa/Co/m'])
    df['weightsBGC']=1/df['uncBeGa/Co/m']**2
```

γ count rate / coincidence count rate -1 is $N_\gamma/N_C - 1$. The uncertainty is complicated, as it includes terms for the background counts.

$$u^2_{N_\gamma} = \frac{R^2_\gamma}{C_\gamma} \cdot \left(1 + B_\gamma \cdot \frac{2^{\tau_\delta/t_{1/2}}}{R_\gamma \cdot (1 - \ln(2)\tau)}\right)^2$$

$$u^2_{N_C} = \frac{R^2_C}{C_C} \cdot \left(1 + B_C \cdot \frac{2^{\tau_\delta/t_{1/2}}}{R_C \cdot (1 - \ln(2)\tau)}\right)^2$$

$$u^2_{B_\gamma} = \sigma^2_{B_\gamma}$$

$$u^2_{B_C} = \sigma^2_{B_C}$$

$$u_{N_\gamma/N_C - 1} = \sqrt{\frac{u^2_{N_\gamma} + u^2_{B_\gamma}}{N^2_C} + N^2_\gamma \frac{u^2_{N_C} + u^2_{B_C}}{N^4_C}}$$

```
    #Gamma/Coinc-1
    df['Ga/Co-1']=df['Xr8']/df['LDXr8']-1
    df['sigmaXsq']=(df['Xr8']**2/df['␣X']*(1+df['backXr8']*
    np.exp(np.log(2)*df['timedif(s)']/halflifeseconds)/(df['Xr8']
                        *(1-np.log(2)*df['␣Live'])))**2)
    df['sigmaLDXsq']=(df['LDXr8']**2/df['␣LDX']*(1+df['backLDXr8']
                        *np.exp(np.log(2)*df['timedif(s)']/halflifeseconds)
                        /(df['LDXr8']*(1-np.log(2)*df['␣Live'])))**2)
    df['sigmabackXsq']=bckgrndX_var(backdf)
    df['sigmabackLDXsq']=bckgrndLDX_var(backdf)
    df['uncGa/Co-1']=np.sqrt((df['sigmaXsq']+df['sigmabackXsq'])
                        /df['LDXr8']**2+(df['sigmaLDXsq']
                       +df['sigmabackLDXsq'])*df['Xr8']**2/df['LDXr8']**4)
    df['weightsGa/Co-1']=1/df['uncGa/Co-1']**2
```

The β count rate per unit mass is $N_\beta \cdot A/m$ with uncertainty

$$u_{N_\beta/m} = \sqrt{\frac{1}{C_\beta} + \frac{1 - N_\beta/N_\gamma}{C_C}} \cdot N_\beta \frac{A}{m}.$$

```
    #Beta/mass, dilution factor included
    df['Be/m']= df['LDr8']*dilution/mass
    df['uncBe/m']=(np.sqrt((1/df['␣LD'])+
              (1-df['LDXr8']/df['Xr8'])/df['␣LDX'])*df['Be/m'])
    df['weightsB']=1/df['uncBe/m']**2
```

$1 - \gamma$ count rate / coincidence count rate is $1 - N_\gamma/N_C$. The uncertainty is

```
    # 1-Gamma/Coinc
    df['1-Co/Ga']=1-df['LDXr8']/df['Xr8']
    df['unc1-Co/Ga']=np.sqrt((df['sigmaXsq']+df['sigmabackXsq'])
                *df['LDXr8']**2/df['Xr8']**4+(df['sigmaLDXsq']
                +df['sigmabackLDXsq'])*1/df['Xr8']**2)
    df['weights1-Co/Ga']=1/df['unc1-Co/Ga']**2
```

Some constants are also written into the dataframe to help the user identify which numbers were used in calculations after the fact.

```
    #Constants to include in dataframe
    df['Reference␣datetime']=refdatetime
    df['Source␣mass␣mg']=mass
    df['Source␣dilution␣factor']=dilution
    return df
```

In conclusion, the `threshold` function adds a lot of columns to the main calculation dataframe!

3.3 'Threshold data' output

The following function `dataget` is quite important as it handles the csv files and returns the output of the `threshold` function. First, it designates `dfpath` and `backdfpath` to the

file paths where the threshold data file and background data file are located. Then each of these csv files is read into the dataframes `thresh1df` and `backdf`. The headers are skipped, as are any dodgy instances where zero counts have been recorded. `parse_dates` and `infer_datetime_format` enable Python to extract date and time information in the correct way. These dataframes are then passed to the `threshold` function. The user is alerted to the analysis of each threshold csv file being completed through a print statement `'done filename'`.

```
88 def dataget(dirname,filename,backfilename):
89     dfpath="{0}/{1}".format(dirname,filename)
90     print()
91     backdfpath="{0}/{1}".format(dirname,backfilename)
92     #Treats zero counts as corrupted files with na_values=0
93     thresh1df = pd.read_csv(dfpath, sep=',', skiprows=5, na_values=0,
94         index_col=0,dayfirst=True, parse_dates=[20], infer_datetime_format=True)
95     back1df = pd.read_csv(backdfpath, sep=',', skiprows=5, na_values=0,
96          index_col=0,dayfirst=True, parse_dates=[20], infer_datetime_format=True)
97     threshold(thresh1df,back1df)
98     print('done␣{0}'.format(dfpath[:-5]))
99     return thresh1df
```

4 Main program: statistical functions

Once the `threshold` function has number crunched each measurement, these results are combined to obtain averaged values for each threshold. It is these numbers that are then used for regression. The statistical quantities required for data analysis are calculated with functions defined in the programs *pandas4LSCdoubles_main.py* and *pandas4LSCtriples_main.py*. These functions make use of `numpy` and `pandas`.

The following naming convention applies:

- `BGC` refers to a quantity calculated using 'beta gamma / coincidence' count rate values.
- `G_C_1` refers to a quantity calculated using 'gamma / coincidence - 1' count rate values.
- `B` refers to a quantity calculated using 'beta' count rate values.
- `1_C_G` refers to a quantity calculated using '1 - coincidence / gamma' count rate values.

In the following examples, `BGC` functions are shown.

4.1 Weighted mean

The weighted mean is defined as

$$\bar{x}_w = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i}$$

where $\bar{x}_w$ is the weighted mean of x, N is the size of the sample and w_i are the weights corresponding to different values x_i.

```
1 def wmBGC(group):
2     """ Weighted mean """
3     d = group['BeGa/Co/m']
4     w = group['weightsBGC']
5     return (d * w).sum() / w.sum()
```

4.2 Observed variance of the weighted mean

The observed variance of the weighted mean is given by:

$$s_w^2 = \frac{N' \sum_{i=1}^{N} w_i (x_i - \bar{x}_w)^2}{(N' - 1) \sum_{i=1}^{N} w_i}.$$

Here N' is simply the number of non-zero weights associated with the values x_i.

```
6  def obsvar_wmBGC(group):
7      """ Observed variance of weighted mean """
8      d = group['BeGa/Co/m']
9      w = group['weightsBGC']
10     wm1 = wmBGC(group)
11     c1 = group.count(axis=0)[0]
12     return (w*(d-wm1)**2).sum() *c1 / ((c1-1)*w.sum())
```

4.3 Observed standard deviation of the weighted mean

The observed standard deviation of the weighted mean is simply the square root of the observed variance of the weighted mean.

```
13 def obsstdev_wmBGC(group):
14     """ Observed standard deviation of weighted mean """
15     obsvar = obsvar_wmBGC(group)
16     return np.sqrt(obsvar)
```

4.4 Observed standard deviation of the weighted mean as a relative percentage

We can express the observed standard deviation of the weighted mean as a relative percentage, by dividing by the weighted mean and multiplying by 100.

```
17 def obsstdev_wm_percBGC(group):
18     """ Observed standard deviation of weighted mean, relative % """
19     wm1 = wmBGC(group)
20     obsvar = obsvar_wmBGC(group)
21     return np.sqrt(obsvar)/wm1*100
```

4.5 Theoretical standard deviation of the weighted mean

The theoretical standard deviation of the weighted mean is given by:

$$\sigma_w = \sqrt{\frac{1}{\sum w_i}}.$$

```
22 def theorstdev_wmBGC(group):
23     """ Theoretical standard deviation of weighted mean """
24     w = group['weightsBGC']
25     return np.sqrt(1/w.sum())
```

4.6 Theoretical standard deviation of the weighted mean as a relative percentage

We can express the theoretical standard deviation of the weighted mean as a relative percentage, by dividing by the weighted mean and multiplying by 100.

```
26 def theorstdev_wm_percBGC(group):
27     """ Theoretical standard deviation of weighted mean, relative % """
28     w = group['weightsBGC']
29     wm1 = wmBGC(group)
30     return np.sqrt(1/w.sum())/wm1*100
```

4.7 'Regression data' output

To put all these statistical quantities together, a function `statsget` is defined.

For the 'independent' variables (`G_C_1` and `1_C_G`) it returns the weighted mean, as well as the theoretical standard deviation, the observed variance, and the observed standard deviation of the weighted mean.

For the 'dependent' variables (`BGC` and `B`) it returns the weighted mean, along with the theoretical standard deviation, the theoretical standard deviation as a relative percentage, the observed variance, the observed standard deviation, and the observed standard deviation as a relative percentage of the weighted mean.

The two data sets (`G_C_1` + `BGC`, and `1_C_G` + `B`) are separated by a blank column.

```
31 def statsget(df):
32     return [(wmG_C_1(df), theorstdev_wmG_C_1(df),obsvar_wmG_C_1(df),
33              obsstdev_wmG_C_1(df),
34              wmBGC(df), theorstdev_wmBGC(df),
35              theorstdev_wm_percBGC(df), obsvar_wmBGC(df),
36              obsstdev_wmBGC(df), obsstdev_wm_percBGC(df),'␣',
37              wm1_C_G(df), theorstdev_wm1_C_G(df), obsvar_wm1_C_G(df),
38              obsstdev_wm1_C_G(df),
39              wmB(df), theorstdev_wmB(df), theorstdev_wm_percB(df),
40              obsvar_wmB(df), obsstdev_wmB(df), obsstdev_wm_percB(df))]
```

5 Speedy file grabber

The `speedyfilegrabber` is an option that works well if all data files have their own corresponding background csv files. A prefix for the data filenames, and another for the background filenames is nominated in the `user` code.

5.1 'Start string' function

A function `startstring` selects file names that start with a prefix `char`, adds them to a list, and returns this list.

```
def startstring(char, stringlist):
    newlist = []
    for string in stringlist:
        if string.startswith(char):
            newlist.append(string)
    newlist.sort()
    return newlist
```

5.2 Handling of paths and files

If the `speedyfilegrabber` option is turned on, then the code finds the path of the `cwd` (current working directory). It then creates a new path `path1` by adding the directory where all the files are (`rtdt1dir`) to the `cwd` path. `filenames1` lists all the files in this directory. The `startstring` function is called twice, once for the `bkgprefix` and again for the `dataprefix`. Now there are two lists of files (`bkgs1` and `files1`), which contain the background files starting with the background prefix, and the data files starting with the data prefix.

```
if speedyfilegrabber:
    path  = os.getcwd()
    path1 = "{0}/{1}".format(path,rtdt1dir)
    filenames1 = os.listdir(path1)

    bkgs1 = startstring(bkgprefix,filenames1)
    files1 = startstring(dataprefix, filenames1)


    print()
    print('Running␣pandas4LSC')
    print('Checking␣files')
```

If the lengths of these two lists are different, then there must be more background files than data files or visa versa. An error is called and then the code stops. If the lengths of the two lists are the same, then the code continues.

```
    LB=len(bkgs1)
    LD=len(files1)
    if LB == LD:
        pass
    else:
```

```
        print()
        print('!!!ATTENTION!!! Different number of background files
        and data files')
        print('Since there is a mismatch with background and data files')
        print('Take another look at your files and rename or delete as
        necessary, then run code again')
        print()
        sys.exit('')
```

To check if each data file is paired with the correct background file, the first element of `bkgs1` is printed next to the first element of `files1`, then the second element and so on.

```
    print("Check background files have corresponding threshold data files:")
    i=0
    if LD > LB:
        l=LB
    else:
        l=LD
    while i < l:
        print("{0} --- {1}".format(bkgs1[i],files1[i]))
        print()
        i=i+1
```

If the user is happy with the pairings, they type `Y`, and the code moves on to the analysis part. If the pairings show a mismatch, then they type `N`, and the code exits.

```
    g2g=input('Does every entry in the list match its background file?
    type Y or N:  ')

    if g2g=='N':
        print()
        print('Since there is a mismatch with background and data files')
        print('Take another look at your files and rename or delete as
        necessary, then run code again')
        print()
        sys.exit('')
    if g2g=='Y':
        print('Data analysis happening now!')
```

Finally if the `speedyfilegrabber` was not selected, all of this section of code is passed.

```
else:
    pass
```

6 Main program: putting it all together

Now that all the functions have been defined, let us move on to the main code.

As a throwback to an earlier version of the code that worked across several directories for different resolving times `rt` and deadtimes `dt`, `rtdtdirectories` is a tuple that contains the directory with the data files in it `rtdt1dir`. The `rt` and `dt` entered by the user are incorporated in the tuple `rtdtinfo`. If the `speedyfilegrabber` was run, then the background and data files are passed to `bckgrnds1` and `threshfiles1`.

```
rtdtdirectories=(rtdt1dir)
rtdtinfo=(rt,dt)

if speedyfilegrabber:
    bckgrnds1=bkgs1 #for speedy file grabber
    threshfiles1=files1 #for speedy file grabber
```

If the `speedyfilegrabber` was not used, `threshfiles1` is a tuple of all threshold data files specified in the *user* code. Similarly, `bckgrnds1` is a tuple of all background files.

```
else:
    threshfiles1=(thresh1A,thresh1B,thresh1C,thresh1D,thresh1E,thresh1F,
           thresh1G,
              thresh1H,thresh1I,thresh1J,thresh1K,thresh1L,thresh1M,thresh1N,
              thresh1O,thresh1P,thresh1Q)
    bckgrnds1=(back1A,back1B,back1C,back1D,back1E,back1F,back1G,back1H,
        back1I,
           back1J,back1K,back1L,back1M,back1N,back1O,back1P,back1Q)
```

Now for the construction of the dataframes. *pandas* loves dataframes! `threshcolumns` is a list of columns for the threshold data dataframe `threshdf`. Some of these are already a part of the incoming threshold data csv files (`' A', ' B', ' C',' AB', ' AC', ' BC', ' ABC', ' X', ' ABX', ' ACX', ' BCX', ' ABCX', ' Real', ' Live', ' LD', ' LDX', ' Started', ' Finished'`). Others we will populate through the rest of this code. `regcolumns` is a list of columns for the dataframe that will contain the values to use for regression calculations `regdf`.

```
threshcolumns= [
' A', ' B', ' C',' AB', ' AC', ' BC', ' ABC', ' X', ' ABX', ' ACX', ' BCX',
' ABCX', ' Real', ' Live', ' LD', ' LDX', ' Started',
' Finished', 'Source mass mg', 'Source dilution factor', 'Reference datetime',
'timedif(s)','Decay Factor', 'backLDr8','LDr8','backXr8', 'Xr8','backLDXr8',
 'LDXr8',
'Ga/Co-1','uncGa/Co-1','BeGa/Co/m', 'uncBeGa/Co/m','','1-Co/Ga',
'unc1-Co/Ga','Be/m','uncBe/m']

threshdf=pd.DataFrame(data=[], columns=(threshcolumns))

regcolumns=['Ga/Co-1 wm','Ga/Co-1 theor st dev of wm',
 'Ga/Co-1 obs var of wm',
'Ga/Co-1 obs st dev of wm', 'BeGa/Co wm',
'BeGa/Co theor st dev of wm','BeGa/Co theor % st dev of wm',
```

```
'BeGa/Co obs var of wm','BeGa/Co obs st dev of wm',
'BeGa/Co obs % st dev of wm','',
'1-Co/Ga wm','1-Co/Ga theor st dev of wm', '1-Co/Ga obs var of wm',
'1-Co/Ga obs st dev of wm',
'Be/m wm', 'Be/m theor st dev of wm','Be/m theor % st dev of wm',
'Be/m obs var of wm','Be/m obs st dev of wm', 'Be/m obs % st dev of wm']

regdf=pd.DataFrame(data=[],columns=(regcolumns))
```

Now for the number crunching! This section of code takes the first threshold data csv file in the list `threshfiles1` (`i=0`). It gets the data from this csv file, and the corresponding background data csv file using `dataget`. This data and associated calculated values fills up `thresh1df`. Then we do some average calculations these values usings `statsget`. The `statsget` output then goes to a new dataframe `reg1df`, which becomes the first line of the master `regdf` dataframe. Similarly, `thresh1df` is the first line of the master `threshdf` dataframe. Once `reg1df` is passed to `regdf`, and `thresh1df` is passed to `threshdf`, then `reg1df` and `thresh1df` are cleared. Then the next pair of csv files is looked at (`i = 1`), and so on until there are no more csv file pairs i.e. `threshfiles1[i]=0`.

```
dirname=rtdt1dir
i=0
while i < len(threshfiles1):
    if threshfiles1[i]:
        thresh1df=dataget(dirname,threshfiles1[i],bckgrnds1[i])
        stats1=statsget(thresh1df)
        #print(thresh1df)
        reg1df=pd.DataFrame(data=stats1,index=[i],columns=regcolumns)
        threshdf=threshdf.append(thresh1df)
        regdf=regdf.append(reg1df)
        thresh1df.drop(thresh1df.index, inplace=True)
        reg1df.drop(reg1df.index, inplace=True)
    i=i+1
```

Once every pair of threshold data and background csvs has been crunched, the dataframes `regdf` and `threshdf` can have their columns reordered (`regdf[regcolumns]` puts the columns in the order of the list `regcolumns`), their values sorted, and then be saved to an xlsx file with a name *outputfilename*-rt*resolving time*-dt*dead time*-RegData-*doubles or triples-specified background*.xlsx etc. That is the end of the *main* code.

```
regdf=regdf[regcolumns]
regdf=regdf.sort_values('Ga/Co-1 wm')
regdf.to_excel(
"{0}_rt{1}dt{2}_RegData_{3}{4}.xlsx".format(outputfilename,rt,dt,DorT,Sb))
regdf.drop(regdf.index, inplace=True)
threshdf=threshdf[threshcolumns]
threshdf.to_excel(
"{0}_rt{1}dt{2}_ThreshData_{3}{4}.xlsx".format(outputfilename,rt,dt,DorT,Sb))
threshdf.drop(threshdf.index, inplace=True)
print()
print('Finished with different thresholds in {0}'.format(dirname))
```

7 Regression program: regression functions

7.1 Least squares regression

It is necessary to import all names from the main working namespace:

```
from __main__ import *
```

Defining a linear function `f1` for least squares regression. The two regression methods handle mathematical functions slightly differently.

```
def f1(x, m, c):
    """ The linear function y= m*x + c """
    return m*x + c
```

This next function defines the fitting routine for linear least squares using the inbuilt *scipy* function `curve_fit`. Since all uncertainties handled are their true size (not merely sized correctly relative to each other), the option `absolute_sigma` is `True`. The two outputs of `curve_fit` are a matrix of optimal parameters `poptLW` and a covariance matrix for these parameters `pcovLW`. We are interested in the standard deviations of the parameters m and c: `stdevm` and `stdevc`. A relative uncertainty `relunc` is calculated as a percentage for the intercept, as well as the all-important activity concentration `actconc`. Theses values `fitparams` are plonked in the dataframe `transferdf` that has columns `fitcols`.

```
def fit(xdata, ydata, yunc, branchratio):
    poptLW, pcovLW = curve_fit(f1, xdata, ydata, p0=None, sigma=yunc,
    absolute_sigma=True)
    stdevm=np.sqrt(pcovLW[0][0])
    stdevc=np.sqrt(pcovLW[1][1])
    relunc=stdevc/poptLW[1]*100
    actconc=poptLW[1]/(1000*branchratio)
    fitparams=[(branchratio, poptLW[0],stdevm,poptLW[1],stdevc,actconc,
    relunc)]
    fitcols=['Branch ratio','gradient','unc gradient', 'intercept',
    'unc intercept','Activity conc', 'rel unc intercept %']
    transferdf=pd.DataFrame(data=fitparams,columns=fitcols)
    return transferdf
```

With the cubic least squares, it is basically the same, except the intercept is parameter number 2, rather than 1 (the cubic coefficient is parameter 0).

```
def f3(x, a, c, d):
    """ The cubic function y= a*x**3 + c*x + d """
    return a*x**3 + c*x + d
\begin{py}[name=one6]
def fit3(xdata, ydata, yunc, branchratio):
    poptLW, pcovLW = curve_fit(f3, xdata, ydata, p0=None, sigma=yunc,
    absolute_sigma=True)
    stdeva=np.sqrt(pcovLW[0][0])
    stdevc=np.sqrt(pcovLW[1][1])
    stdevd=np.sqrt(pcovLW[2][2])
    relunc=stdevd/poptLW[2]*100
    actconc=poptLW[2]/(1000*branchratio)
```

```
    fit3params=[(branchratio,poptLW[0],stdeva,poptLW[1],stdevc,
    poptLW[2],stdevd,actconc,relunc)]
    fit3cols=['Branch ratio','a','unc a', 'c', 'unc c', 'intercept',
    'unc intercept','Activity conc', 'rel unc intercept %']
    transfer3df=pd.DataFrame(data=fit3params,columns=fit3cols)
    return transfer3df
```

7.2 Orthogonal distance regression

Orthogonal distance regression (ODR) requires that you explicitly specify the parameters in a vector ahead of the independent variable when defining a mathematical function.

```
def f1ODR(p,x):
    """ The linear function y= m*x + c """
    m,c=p
    return m*x + c

def f3ODR(p,x):
    """ The cubic function y= a*x**3 + c*x + d """
    a,c,d=p
    return a*x**3 + c*x + d
```

I imported the ODR module as `odear` because I found it slightly funny. A model `lin_model` and a data instance `lin_reg_data` is created. Then ODR happens, with the initial parameter estimate `beta0` equal to the parameters returned by least squares fitting (`beta_lin` is defined so later in the module), the model is run, and the parameters read out into an array `linreg_info`. This array is then passed to a dataframe `linregdf`, which the function `fitODR` returns.

```
def fitODR(xdata, ydata, xunc, yunc, branchratio):
    lin_model=odear.Model(f1ODR)
    lin_reg_data=odear.RealData(xdata,ydata,sx=xunc,sy=yunc)
    lin_ODR=odear.ODR(lin_reg_data, lin_model, beta0=beta_lin)
    lin_out= lin_ODR.run()
    fitcols=['Branch ratio','gradient','unc gradient', 'intercept',
    'unc intercept','Activity conc', 'rel unc intercept %']
    linreg_info=[(branchratio, lin_out.beta[0],lin_out.sd_beta[0],lin_out.beta[1],
    lin_out.sd_beta[1],lin_out.beta[1]/(1000*branchratio),
    lin_out.sd_beta[1]/lin_out.beta[1]*100)]
    linregdf=pd.DataFrame(data=linreg_info,columns=(fitcols))
    return linregdf
```

Again, the cubic ODR routine is very similar to the linear one.

```
def fit3ODR(xdata, ydata, xunc, yunc, branchratio):
    cub_model=odear.Model(f3ODR)
    cub_reg_data=odear.RealData(xdata,ydata,sx=xunc,sy=yunc)
    cub_ODR=odear.ODR(cub_reg_data, cub_model, beta0=beta_cub)
    cub_out= cub_ODR.run()
    fit3cols=['Branch ratio','a','unc a', 'c', 'unc c', 'intercept',
    'unc intercept', 'Activity conc', 'rel unc intercept %']
    cubreg_info=[(branchratio, cub_out.beta[0],cub_out.sd_beta[0],
        cub_out.beta[1], cub_out.sd_beta[1],cub_out.beta[2],
```

```
66          cub_out.sd_beta[2],cub_out.beta[2]/(1000*branchratio),
67          cub_out.sd_beta[2]/cub_out.beta[2]*100)]
68      cubregdf=pd.DataFrame(data=cubreg_info,columns=(fit3cols))
69      return cubregdf
```

8 Regression program: putting it all together

The uncertainties associated with the data are in different columns of the main `regdf`, depending on the standard deviation selected by the user. If the observed standard deviation is selected (`OSD`), then the code points to the columns that contain observed standard deviations, and the same treatment is given for theoretical standard deviation (`TSD`). Once the data and uncertainties are read into the *regression* code, the `regdf` dataframe is cleared.

```
if StandDev=='OSD':
    xdataBGC = np.array(regdf)[:,0]
    ydataBGC = np.array(regdf)[:,4]
    xuncBGC = np.array(regdf)[:,3]
    yuncBGC = np.array(regdf)[:,8]

    xdataB = np.array(regdf)[:,11]
    ydataB = np.array(regdf)[:,15]
    xuncB = np.array(regdf)[:,14]
    yuncB = np.array(regdf)[:,19]

if StandDev=='TSD':
    xdataBGC = np.array(regdf)[:,0]
    ydataBGC = np.array(regdf)[:,4]
    xuncBGC = np.array(regdf)[:,1]
    yuncBGC = np.array(regdf)[:,5]

    xdataB = np.array(regdf)[:,11]
    ydataB = np.array(regdf)[:,15]
    xuncB = np.array(regdf)[:,12]
    yuncB = np.array(regdf)[:,16]

regdf.drop(regdf.index, inplace=True)
```

Dataframes `fitparamsdf` and `fit3paramsdf` are built to handle linear and cubic fit parameters, with columns specified by `fitparamscolumns` and `fit3paramscolumns` respectively. To compare both linear and cubic fits in one dataframe, `fitsummarydf` (columns `fitsummarycols`) is built.

```
fitparamscolumns=[
'Regression␣data', 'Regression␣method',
'gradient','unc␣gradient','intercept','unc␣intercept',
'Activity␣conc','rel␣unc␣intercept␣%']

fitparamsdf=pd.DataFrame(data=[],columns=(fitparamscolumns))

fit3paramscolumns=['Regression␣data', 'Regression␣method',
'a','unc␣a', 'c', 'unc␣c', 'intercept',
'unc␣intercept','Activity␣conc', 'rel␣unc␣intercept␣%']

fit3paramsdf=pd.DataFrame(data=[],columns=(fit3paramscolumns))

fitsummarycols=['Function', 'Regression␣data', 'Regression␣method',
'Activity␣conc', 'rel␣unc␣intercept␣%']
```

```
39
40 fitsummarydf=pd.DataFrame(data=[],columns=(fitsummarycols))
```

The `infodf` dataframe is where all information given by the user in the *user* code is summoned. It will be printed below the `fitsummarydf` in the final xlsx file for easy reference. Since these two dataframes will be combined, we make sure `infodf` has the same columns as `fitsummarydf`.

```
41 infodat=[('','','','',''),
42          ('Source␣name',outputfilename,'','',''),
43          ('Branch␣ratio',branchratio,'','',''),
44          ('Reference␣date␣time',refdatetime,'','',''),
45          ('Half␣life␣s',halflifeseconds,'','',''),
46          ('Dilution␣factor',dilution,'','',''),
47          ('Mass␣mg',mass,'','',''),
48          ('Resolving␣time␣ns',rt,'','',''),
49          ('Dead␣time␣us',dt,'','',''),
50          ('Gamma␣shift',gs,'','',''),
51          ('Standard␣deviation␣used␣as␣weights',StandDev,'','',''),
52          ('Background',Sb,'','',''),
53          ('Data',DorT,'','','')]
54
55 infodf=pd.DataFrame(data=infodat,columns=(fitsummarycols))
```

Now there are eight different regressions to be performed.

- Linear least squares on $\beta\gamma/C$ vs $\gamma/C - 1$
- Linear ODR on $\beta\gamma/C$ vs $\gamma/C - 1$
- Cubic least squares on $\beta\gamma/C$ vs $\gamma/C - 1$
- Cubic ODR on $\beta\gamma/C$ vs $\gamma/C - 1$
- Linear least squares on β vs $1 - C/\gamma$
- Linear ODR on β vs $1 - C/\gamma$
- Cubic least squares on β vs $1 - C/\gamma$
- Cubic ODR on β vs $1 - C/\gamma$

Here only the code for the first two is shown. The linear least squares function `fit` is called and the output saved to a dataframe `linregBGC_LS`, which is added to the master `fitparamsdf` dataframe.

```
56 # LINEAR regression. Data = BG/C vs G/C-1 Method = Least Squares
57 linregBGC_LS=fit(xdataBGC,ydataBGC,yuncBGC,branchratio)
58 BGC_LS=pd.DataFrame(data=[('Linear','BG/C␣vs␣G/C-1','Least␣Squares')],
59         columns=['Function','Regression␣data','Regression␣method'])
60 linregBGC_LS=pd.concat([linregBGC_LS,BGC_LS],axis=1)
61 fitparamsdf=fitparamsdf.append(linregBGC_LS)
```

For the linear ODR version, the parameters from the previous linear least squares fit are read from the `linregBGC_LS` dataframe and become the `beta_lin` array. As previously mentioned `beta_lin` is given to the ODR routine as `beta0`, an initial estimate of fitting parameters. Because the parameters from least squares and ODR should be similar, ODR doesn't take long in this case. The `fitODR` function is called, with the output destined for `linregBGC_ODR`, a dataframe that is appended to the master `fitparamsdf`. This section of code, along with the least squares section, are repeated another three times with slight differences for cubic fits and the β vs $1 - C/\gamma$ data.

```
# LINEAR regression. Data = BG/C vs G/C-1 Method = ODR
# Least squares becomes initial estimate for ODR
beta_lin=[np.array(linregBGC_LS)[0,3],np.array(linregBGC_LS)[0,5]]
linregBGC_ODR=fitODR(xdataBGC, ydataBGC, xuncBGC, yuncBGC,branchratio)
linBGC_ODR=pd.DataFrame(data=[('Linear','BG/C␣vs␣G/C-1',
        'Orthogonal␣Distance')],columns=['Function','Regression␣data',
        'Regression␣method'])
linregBGC_ODR=pd.concat([linregBGC_ODR,linBGC_ODR],axis=1)
fitparamsdf=fitparamsdf.append(linregBGC_ODR)
```

All dataframes are cleared at the end of the regression procedure.

```
# CLEAR DATA FRAMES
linregBGC_LS.drop(linregBGC_LS.index, inplace=True)
linregBGC_ODR.drop(linregBGC_ODR.index, inplace=True)
linregB_LS.drop(linregB_LS.index, inplace=True)
linregB_ODR.drop(linregB_ODR.index, inplace=True)
cubregBGC_LS.drop(cubregBGC_LS.index, inplace=True)
cubregBGC_ODR.drop(cubregBGC_ODR.index, inplace=True)
cubregB_LS.drop(cubregB_LS.index, inplace=True)
cubregB_ODR.drop(cubregB_ODR.index, inplace=True)
```

The three dataframes of fit parameters are combined into one, but with the `fitsummarycolumns` so only the most interesting parameter (the intercept) appears in the `fitsummarydf`. All other parameters and their associated uncertainties are still safe in `fitparamsdf` and `fit3paramsdf`. The linear and cubic functions are added to the linear fit dataframe `fitparamsdf` and the cubic fit dataframe `fit3paramsdf` respectively.

```
fitsummarydf=fitsummarydf.append(fit3paramsdf)
fitsummarydf=fitsummarydf.append(fitparamsdf)
fitsummarydf=fitsummarydf.append(infodf)

fit1df=pd.DataFrame(data=[('','','','','','','',''),
        ('y=m*x+c','','','','','','','')],columns=(fitparamscolumns))
fitparamsdf=fitparamsdf.append(fit1df)
fit3df=pd.DataFrame(data=[('','','','','','','','','','',''),
        ('y=a*x^3␣+␣c*x␣+␣d','','','','','','','','','')],columns
        =(fit3paramscolumns))
fit3paramsdf=fit3paramsdf.append(fit3df)
```

The three dataframes are tidied up and saved to an xlsx file with name *outputfile-name*-rt*resolving time*-dt*dead time*-RegData-*doubles or triples-specified background*.xlsx. Each dataframe appears as a separate sheet of the xlsx file. Finally the three dataframes are wiped clear.

```
91  fitparamsdf=fitparamsdf[fitparamscolumns]
92  fit3paramsdf=fit3paramsdf[fit3paramscolumns]
93  fitsummarydf=fitsummarydf[fitsummarycols]
94  fits_writer=pd.ExcelWriter("{0}_rt{1}dt{2}_AllFits_{3}{4}_{5}.xlsx".
95  format(outputfilename,rt,dt,DorT,Sb,StandDev))
96  fitsummarydf.to_excel(fits_writer,"Summary")
97  fitparamsdf.to_excel(fits_writer,"LinearFits")
98  fit3paramsdf.to_excel(fits_writer,"CubicFits")
99  fits_writer.save()
100 print("All␣fit␣info␣saved␣in␣{0}_rt{1}dt{2}_AllFits_{3}{4}_{5}.xlsx".
101 format(outputfilename,rt,dt,DorT,Sb,StandDev))
102 fitparamsdf.drop(fitparamsdf.index, inplace=True)
103 fit3paramsdf.drop(fit3paramsdf.index, inplace=True)
```