SNANA Starter Kit

Oct 19, 2020

Contents

CHAPTER 1

Installing SNANA

Below, I outline the steps to installing SNANA. While SNANA will run on personal computers, please note that a lot of SNANA functionality is enabled by using a cluster environment with batch job submission.

For additional details, please see the SNANA installation guide: http://snana.uchicago.edu/doc/snana_install.pdf

1.1 Installing on a Mac

Installing on Linux is very similar, except for the *preliminaries* stage.

1.1.1 Preliminaries

If you haven't already, start by installing command line tools. In terminal, type:

```
xcode-select --install
```

Then, install homebrew by following the directions at https://brew.sh/. Once Homebrew is installed, use the following commands to install dependencies *GSL* and *cfitsio*:

```
brew update
brew install gsl
brew install cfitsio
```

SNANA can optionally use ROOT or HBOOK for plotting and some additional utilities. From the SNANA manual: go to http://root.cern.ch/ or pre-compiled libraries at https://root.cern.ch/content/release-53434. But, installing those is a bit harder so I won't try to explain that here.

1.1.2 Setting Environment Variables

SNANA requires the following environment variables to be set: `CFITSIO_DIR`, `GSL_DIR`, and `ROOT_DIR` (only if ROOT is installed). `SNANA_DIR` and `SNDATA_ROOT` must also be defined.

As standard practice I first put the following lines in my ~/.bash_profile file with a text editor:

```
if [ -f $HOME/.bashrc ]; then
      source $HOME/.bashrc
fi
```

Then, I set the environment variables by opening ~/.bashrc in a text editor and adding the following lines:

```
export SOFTDIR=/my/directory/path
export GSL_DIR=/usr/local/Cellar/gsl/2.4
export CFITSIO_DIR=/usr/local/Cellar/cfitsio/3.450_1
export SNDATA_ROOT=$SOFTDIR/SNDATA_ROOT
export SNANA_DIR=$SOFTDIR/SNANA
```

The exact path for GSL_DIR and CFITSIO_DIR will depend on your system. Usually Homebrew installs in /usr/local/Cellar and then you'll have to check out the subdirectory to see where the GSL and CFITSIO files were actually put. As the example above shows, it will depend which version gets installed.

Once your ~/.bashrc file has been saved, run:

```
source ~/.bashrc
```

or open up a new tab/window in terminal to set the environment variables.

1.1.3 Downloading SNANA

SNANA comes in two tarballs - the source code and the SNDATA_ROOT compilation of various data files. The SNANA source code can be cloned from git:

```
mkdir $SNANA_DIR
cd $SNANA_DIR
git clone https://github.com/RickKessler/SNANA.git
```

The SNDATA_ROOT tarball can be downloaded from the SNANA website: http://snana.uchicago.edu/. Or, at the command line:

```
mkdir $SNDATA_ROOT
cd $SNDATA_ROOT
wget http://snana.uchicago.edu/downloads/SNDATA_ROOT.tar.gz
tar -xvzf SNDATA_ROOT.tar.gz
```

1.1.4 Installing SNANA

Installing SNANA on a Mac requires two small edits to $SNANA_DIR/src/sntools.h. However, be warned that this could cause some ringing noise in spectral simulations due to a finite number of randoms. Unfortunately, I couldn't figure out another workaround here. The following two lines should be un-commented (remove the // at the beginning of the lines):

```
//#define  ONE_RANDOM_STREAM  // enable this for Mac (D.Jones, July 2020)
//#define  MACOS              // another MAC OS option, D.Jones, Sep 2020
```

Recently, there's one more hack needed in the Makefile. If your Mac has GCC version 10.x, SNANA will read the version incorrectly. That means you'll have to search for these lines:

```
ifeq ($(GCCVERSION10),0)
  EXTRA_FLAGS_FORTRAN  = $(EXTRA_FLAGS)
  EXTRA_FLAGS_C        = $(EXTRA_FLAGS)
else
  # for gcc v10, allow argument-mismatch errors (9.2020)
  EXTRA_FLAGS_FORTRAN = $(EXTRA_FLAGS) -fallow-argument-mismatch -fcommon
  EXTRA_FLAGS_C       = $(EXTRA_FLAGS) -fcommon
endif
```

and add the extra fortran flags to the first `EXTRA_FLAGS` line, like so:

```
ifeq ($(GCCVERSION10),0)
  EXTRA_FLAGS_FORTRAN  = $(EXTRA_FLAGS) -fallow-argument-mismatch -fcommon
  EXTRA_FLAGS_C        = $(EXTRA_FLAGS) -fcommon
else
  # for gcc v10, allow argument-mismatch errors (9.2020)
  EXTRA_FLAGS_FORTRAN = $(EXTRA_FLAGS) -fallow-argument-mismatch -fcommon
  EXTRA_FLAGS_C       = $(EXTRA_FLAGS) -fcommon
endif
```

After that, installing should be as easy as:

```
cd $SNANA_DIR/src
make
```

Once it finishes, open your `~/.bashrc` file again and add the following line at the bottom:

```
export PATH=$SNANA_DIR/bin:$PATH
```

and remember to type `source ~/.bashrc` afterwards. For the other exercises in this guide, check that `kcor.exe`, `snlc_sim.exe` and `snlc_fit.exe` have compiled correctly. Hopefully you can reproduce the following lines.

For `kcor.exe`:

```
> kcor.exe
SNANA_DIR   = /usr/local/SNANA
SNDATA_ROOT = /usr/local/SNDATA_ROOT

FATAL[rd_input]:
       Cannot open input file :
        'kcor.input'



 `|```````|`
 <| o\ /o |>
  | ' ; ' |
  |  ___  |     ABORT program on Fatal Error.
  | |' '| |
  | `---' |
  \_______/
```

For `snlc_sim.exe`:

```
> snlc_sim.exe
```

(continues on next page)

(continued from previous page)

```
************************************************************
 Begin execution of snlc_sim.exe
 Full command:

SNDATA_ROOT = /usr/local/SNDATA_ROOT
SNANA_DIR   = /usr/local/SNANA

###########################################################
   INIT_SNVAR: Init variables.
###########################################################

 HOST MACHINE =    ()
 SNDATA_ROOT = /usr/local/SNDATA_ROOT
 SNANA_DIR = /usr/local/SNANA    (v10_73j)
 Allocate 12.50 MB for CIDMASK array (to check duplicates)
 sizeof(INPUTS) =   1.001 MB
 sizeof(GENLC)  =   7.880 MB

FATAL[read_input]:
       Cannot open input file :
        'snlc_sim.input'



 `|```````|`
 <| o\ /o |>
  | ' ; ' |
  |  ___  |     ABORT program on Fatal Error.
  | |' '| |
  | `---' |
  \_______/
```

And finally, for `snlc_fit.exe`:

```
> snlc_fit.exe

###########################################################
   INIT_SNVAR: Init variables.
###########################################################

 HOST MACHINE =    ()
 SNDATA_ROOT = /usr/local/SNDATA_ROOT
 SNANA_DIR = /usr/local/SNANA    (v10_73j)
 Allocate 12.50 MB for CIDMASK array (to check duplicates)

###########################################################
   READ SNLCINP NAMELIST.
###########################################################

  Enter namelist filename (CR=snlc_fit.nml) ==>
```

You're done! Please report any issues with this guide using the SNANA_StarterKit GitHub page.

CHAPTER 2

Using Filter Functions to Make a KCOR file

One of the slightly peculiar things about SNANA is the so-called `KCOR` file. In spite of the name, this file usually does *not* contain k-corrections! Instead, it takes filter throughputs, combines them with user-defined filter shifts and zeropoint offsets, and creates an output FITS file that is used by the SNANA fitting and simulations programs.

2.1 The KCOR input file

For creating a new KCOR file, you need to create an INPUT file. I'll use the Pan-STARRS filter set to create a kcor file for fitting or simulating Pan-STARRS observations. Let's name this file `kcor_PS1.input`:

```
MAGSYSTEM:  AB
FILTSYSTEM: COUNT
FILTPATH: $SNDATA_ROOT/filters/PS1/Pantheon/PS1
FILTER: PS1-g   g_filt_tonry.txt   0.0
FILTER: PS1-r   r_filt_tonry.txt   0.0
FILTER: PS1-i   i_filt_tonry.txt   0.0
FILTER: PS1-z   z_filt_tonry.txt   0.0
FILTER: PS1-y   y_filt_tonry.txt   0.0

OUTFILE:  kcor_PS1.fits
```

The input file can get *a lot* more complicated, but this gives the basics. First, `magsystem` is set to `AB`, `VEGA`, or `BD17`, for example. For multiple magsystems, one simply has to split the filters into multiple blocks like so:

```
MAGSYSTEM:  AB
FILTSYSTEM: COUNT
FILTPATH: $SNDATA_ROOT/filters/PS1/Pantheon/PS1
FILTER: PS1-g   g_filt_tonry.txt   0.0
FILTER: PS1-r   r_filt_tonry.txt   0.0

MAGSYSTEM:  BD17
FILTSYSTEM: COUNT
FILTPATH: $SNDATA_ROOT/filters/PS1/Pantheon/CFA3_native
```

(continues on next page)

(continued from previous page)

```
FILTER:  CFA41-U/k            cfa4_U_p1.dat  9.724
FILTER:  CFA41-B/l            cfa4_B_p1.dat  9.88605-0.024
FILTER:  CFA41-V/m            cfa4_V_p1.dat  9.47432-0.0012
```

The final column here defines a zeropoint offset (these lines are specifically from the Pantheon sample, and the final term is the "Supercal" offset and is subtracted from the nominal zeropoint). Specifically, this number is the magnitude of BD17 in the Vega system, so these are effectively Vega magnitudes.

The `FILTER` lines give first a longer filter name and then after the `/`, a one-letter abbreviation for the filter. These have to be unique (unfortunately) so you might end up with counterintuitive names for your filters if you have a survey with many different filters. The next column is just the name of the filter file.

The last thing is the output file, which is fairly self-explanatory:

```
OUTFILE:  kcor_PS1.fits
```

2.2 Using `kcor.exe` to create the KCOR file

Finally, to create the file:

```
> kcor.exe kcor_PS1.input
```

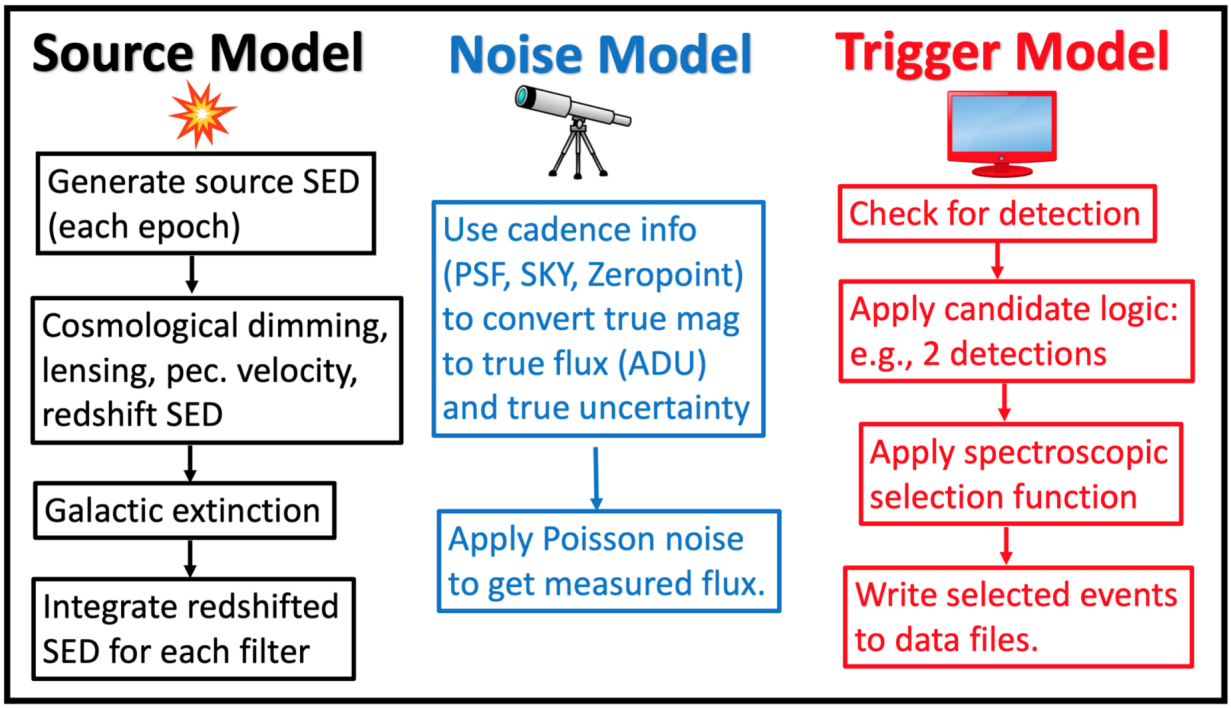
After running this, you should see the output FITS file. This file will be referenced in your simulation and LC fitting input files.

CHAPTER 3

Running a Simple Simulation

SNANA is an extremely powerful tool for survey simulations, and can simulate nearly every aspect of a survey, at least on the catalogue level (not pixel level) that I can think of.

From Kessler et al. (2019), here's a basic schematic of the SNANA simulation framework.



Building a SNANA simulation requires at minimum, four principal files: the KCOR file, the `SIMLIB` file, which contains the sequence of observations to simulate, the `SEARCHEFF` file that gives the efficiency of SN detection, and finally the sim-input file which is passed to the simulation program, *snlc_sim.exe*.

3.1 Writing a Simple SIMLIB File

The SIMLIB file defines the parameters of the survey observations. At minimum, this requires a list of MJDs, filters, sky noise, and image zeropoints. Here is a brief example for the Pan-STARRS medium deep survey:

```
TELESCOPE: PS1
SURVEY: PS1MD    FILTERS: griz

BEGIN LIBGEN  2016-3-28
# --------------------------------------------
LIBID: 1
RA: 0    DECL: 0 NOBS: 6 MWEBV: 0.024   PIXSIZE: 0.25
FIELD: 1
#                            CCD  CCD          PSF1 PSF2 PSF2/1
#     MJD      IDEXPT  FLT GAIN NOISE SKYSIG (pixels)  RATIO  ZPTAVG ZPTERR  MAG
S: 55074.600 0 g 1.0 0.0 6.8691 3.7960 0.0000 0.0000 30.2426 0.0034 -99
S: 55086.600 1 g 1.0 0.0 10.319 4.4315 0.0000 0.0000 30.1481 0.0034 -99
S: 55089.600 2 g 1.0 0.0 6.8671 2.8140 0.0000 0.0000 30.2175 0.0036 -99
S: 55095.500 3 g 1.0 0.0 7.2529 3.9464 0.0000 0.0000 30.3894 0.0041 -99
S: 55098.500 4 g 1.0 0.0 6.8112 3.3349 0.0000 0.0000 30.1146 0.0040 -99
S: 55101.600 5 g 1.0 0.0 7.4172 3.1407 0.0000 0.0000 30.1131 0.0037 -99
END_LIBID: 1
END_OF_SIMLIB: 1 ENTRIES
```

The basic syntax here is a short header, where the `TELESCOPE` and `SURVEY` keys shouldn't have any impact on the simulated outputs and `FILTERS` is just the list of one-letter filter names (specified in your KCOR file).

Next, there are a couple lines that specify different "libraries" with the `LIBID` key. In this case, `LIBID` or `FIELD` can serve to separate the simulation into distinct groups where simulated supernovae do not overlap.

The keys `RA`, `DEC`, `NOBS`, `MWEBV`, and `PIXSIZE` are right ascension, declination, number of observations, Milky Way E(B-V), and the pixel size of the detector. In particular, the pixel size of the detector is important for generating simulations with realistic noise properties.

Now, the observation lines:

```
#                            CCD  CCD          PSF1 PSF2 PSF2/1
#     MJD      IDEXPT  FLT GAIN NOISE SKYSIG (pixels)  RATIO  ZPTAVG ZPTERR  MAG
S: 55074.600 0 g 1.0 0.0 6.8691 3.7960 0.0000 0.0000 30.2426 0.0034 -99
```

In order, these fields are the observed MJD, an integer label for the observation number, the filter, the gain, the CCD noise (I think this is read noise?) and the RMS of the sky measurements. Next, three parameters of the PSF, which are spread in X, spread in Y, and the ratio of the two. Those second parameters can be set to 0 and SNANA will assume a spherically symmetric PSF.

`ZPTAVG` is the image zeropoint and the *ZPTERR* will be propagated to the observational uncertainties. `MAG` should be set to -99 or else SNANA will simulate *only* those specific magnitudes on a given date.

3.2 The Search-Efficiency File

The search-efficiency file defines the probability of detecting a source as a function of magnitude. This file is a essentially a two-column list for one or multiple filters that gives a magnitude and a probability of detection. Here's an example:

```
PHOTFLAG_DETECT: 4096
FILTER: g
SNR:  0.500  0.001
SNR:  1.500  0.004
SNR:  2.500  0.028
SNR:  3.500  0.189
SNR:  4.500  0.428
SNR:  5.500  0.644
SNR:  6.500  0.796
SNR:  7.500  0.841
SNR:  8.500  0.949
SNR:  9.500  0.861
SNR:  10.500  0.936
SNR:  11.500  0.956
SNR:  12.500  0.952
SNR:  13.500  0.920
SNR:  14.500  1.000
SNR:  15.500  1.000
SNR:  16.500  0.857
SNR:  17.500  0.933
SNR:  18.500  1.000
SNR:  19.500  1.000
FILTER: r
SNR:  0.500  0.001
SNR:  1.500  0.004
SNR:  2.500  0.025
SNR:  3.500  0.177
SNR:  4.500  0.423
SNR:  5.500  0.593
SNR:  6.500  0.693
SNR:  7.500  0.769
SNR:  8.500  0.878
SNR:  9.500  0.850
SNR:  10.500  0.900
SNR:  11.500  0.871
SNR:  12.500  0.849
SNR:  13.500  0.915
SNR:  14.500  0.969
SNR:  15.500  0.860
SNR:  16.500  0.889
SNR:  17.500  0.952
SNR:  18.500  0.864
SNR:  19.500  0.955
```

`PHOTFLAG_DETECT` is just a value that is added to the simulated data file to indicate whether or not a given epoch was high enough SNR to be "detected" in the simulation.

3.3 The SEARCHEFF_PIPELINE_LOGIC_FILE

This one is pretty simple. The pipeline logic file tells the simulation what combination of individual detections in various filters and epochs constitutes a "detection" for the purposes of the survey. For example, detections in two different filters or two separate detections.

Here are a couple examples. For a survey that requires two detections in any one of *griz*:

```
PS1MD: 2 g+r+i+z # require 1 epoch, any band
```

Here's SDSS:

```
SDSS: 3 gr+ri+gi  # require 3 epochs, each with detection in two bands.
```

3.4 The Sim-Input File

Finally, the input file! Starting with the basics:

```
GENVERSION: PS1MD          # simname
GENSOURCE:  RANDOM
GENMODEL:   SALT2.JLA-B14
GENPREEFIX: YSE_IA
RANSEED: 128473       # random number seed

SIMLIB_FILE: PS1MD_test.simlib # simlib file

KCOR_FILE:  kcor_PS1.fits
```

GENVERSION is the name of the simulation, GENSOURCE can be set to *RANDOM* or *GRID*. *GRID* simulations are a whole thing - let's stick to RANDOM (Monte Carlo) simulations. GENMODEL refers to one of the "model" directories in $SNDATA_ROOT/models.

SNANA has kind of a peculiar way of identifying models. The first word (before the period) specifies the model type while the portion after the period specifies the version. In this case, SALT2.B14 is SALT2.4 from Betoule et al. (2014), but extrapolated to slightly redder wavelengths than the original model. The model directory is $SNDATA_ROOT/models/SALT2/SALT2.JLA-B14 and some additional information is in the *SALT2.INFO* file in that directory. Some models also have README files, which are helpful :).

Onwards! There are three ways to figure out how many SNe to simulate (that I'm aware of). The first is NGEN_LC, which tells the simulation to run until the specified number of SNe have been detected and pass all cuts. The second is NGENTOT_LC, which tells the simulation to run until the specified number of SNe have been simulated *regardless of whether or not they pass cuts*. The third option is NGEN_SEASON which uses the catalog of observations, the SOLID_ANGLE key , and the SN rates to figure out how many SNe your survey is expected to observe:

```
NGEN_LC: 5000

APPLY_SEARCHEFF_OPT: 1

EXPOSURE_TIME_FILTER: g 1.0
EXPOSURE_TIME_FILTER: r 1.0
EXPOSURE_TIME_FILTER: i 1.0
EXPOSURE_TIME_FILTER: z 1.0

GENFILTERS: griz

GENSIGMA_SEARCH_PEAKMJD:  1.0 # sigma-smearing for  SEARCH_PEAKMJD (days)

GENRANGE_PEAKMJD:  55000 56000
SOLID_ANGLE: 0.192
```

APPLY_SEARCHEFF_OPT set to *1* keeps all "detected" SNe, but does not require that a SN has spectroscopic confirmation (APPLY_SEARCHEFF_OPT = 3) or a host galaxy redshift (APPLY_SEARCHEFF_OPT = 5) - those can

be specified in more complicated simulations. `APPLY_SEARCHEFF_OPT = 0` can be used to keep all SNe, detected or not.

`EXPOSURE_TIME_FILTER` here is set to 1.0, which just uses the nominal zeropoints, etc specified in the SIMLIB file. It can be scaled up or down to easily adjust simulations.

`GENSIGMA_SEARCH_PEAKMJD` just adds uncertainty to the time of maximum light reported in the generated file headers.

Next, referencing the search efficiency files:

```
SEARCHEFF_PIPELINE_FILE:  SEARCHEFF_PIPELINE_PS1.DAT
SEARCHEFF_PIPELINE_LOGIC_FILE:  SEARCHEFF_PIPELINE_LOGIC_PS1.DAT
```

Then, the redshift range to simulate. the redshift uncertainty to simulate, and the range of time around maximum light that is simulated for each light curve:

```
GENRANGE_REDSHIFT:  0.01    0.5
GENSIGMA_REDSHIFT:  0.000001
GENRANGE_TREST:   -20.0    80.0     # rest epoch relative to peak (days)
```

Next, SN rates. There are a lot of options here, but for low-z simulations a simple power law will suffice:

```
DNDZ: POWERLAW  2.6E-5  2.2 # rate=2.6E-5*(1+z)^1.5 /yr/Mpc^3
```

Here's an alternate example with two power laws from the manual:

```
# R0 Beta Zmin Zmax
DNDZ: POWERLAW2 2.2E-5 2.15 0.0 1.0 # rate = R0(1+z)^Beta
DNDZ: POWERLAW2 9.76E-5 0.0 1.0 2.0 # constant rate for z>1
```

And some extra things:

```
OPT_MWEBV: 1 # simulate galactic extinction from values in SIMLIB file

# smear flags: 0=off, 1=on
SMEARFLAG_FLUX:    1  # photo-stat smearing of signal, sky, etc ...
SMEARFLAG_ZEROPT:  1  # smear zero-point with zptsig
```

Apply some sample cuts:

```
APPLY_CUTWIN_OPT:     1
CUTWIN_NEPOCH:   5 -5.              # require 5 epochs (no S/N requirement)
CUTWIN_TRESTMIN: -20  10
CUTWIN_TRESTMAX:   9  40
CUTWIN_MWEBV:      0 .20
CUTWIN_SNRMAX:   5.0 griz 2 -20. 80.  # require 1 of griz with S/N > 5
```

`FORMAT_MASK: 32` specifies FITS format, while `FORMAT_MASK: 2` is ASCII:

```
FORMAT_MASK:  2 # terse format
```

Parameters of the SALT model:

```
# SALT shape and color parameters
GENMEAN_SALT2x1:     0.703
GENRANGE_SALT2x1:   -5.0  +4.0     # x1 (stretch) range
GENSIGMA_SALT2x1:    2.15  0.472      # bifurcated sigmas
```

(continues on next page)

(continued from previous page)

```
GENMEAN_SALT2c:      -0.04
GENRANGE_SALT2c:    -0.4    0.4      # color range
GENSIGMA_SALT2c:     0.033    0.125      # bifurcated sigmas
```

Nuisance parameters from the Tripp formula. Alpha is the correlation of shape parameter *x1* with distance, while Beta is the correlation of color parameter *c* with distance.:

```
# SALT2 alpha and beta
GENMEAN_SALT2ALPHA:   0.14
GENMEAN_SALT2BETA:    3.1
```

Cosmological parameters:

```
# cosmological params for lightcurve generation and redshift distribution
OMEGA_MATTER:  0.3
OMEGA_LAMBDA:  0.7
W0_LAMBDA:    -1.00
H0:            70.0
```

And last but not least, variables to "dump" to an output file for every SN (not just those detected). The first number is the number of variables:

```
SIMGEN_DUMPALL:  10  CID  Z  PEAKMJD S2c S2x1 SNRMAX MAGT0_r MAGT0_g MJD_TRIGGER␣
↪NON1A_INDEX
```

The *SIMGEN_DUMPALL* key will save the specified columns for both detected and undetected SNe, while *SIMGEN_DUMP* will save the information only for detected SNe. The full example file is available *here*.

3.4.1 Core-collapse and other non-Ia Simulations

For non-Ia simulations, only a couple things need to be changed in the input file:

```
GENMODEL:   NON1A
DNDZ: POWERLAW2 5E-5   4.5   0.0   0.8    # rate = R0(1+z)^Beta for z<0.8
DNDZ: POWERLAW2 5.44E-4  0.0   0.8   9.1  # rate = constant for z>0.8
DNDZ_PEC1A: POWERLAW  2.6E-5  2.2

INPUT_FILE_INCLUDE: $SNDATA_ROOT/snsed/NON1A/SIMGEN_INCLUDE_NON1A_J17-beforeAdjust.
↪INPUT
```

These lines change the generated model, the SN rates (including a new term for peculiar SNe Ia), and add an input file that defines the contribution of different CC SN templates.

3.5 Running the Simulation

Now that the files are assembled, running the simulations is easy:

```
snlc_sim.exe sim_PS1.input
```

The default output is placed in `$SNDATA_ROOT/SIM`. For your first simulation, you may need to execute the following command before running the simulation:

```
mkdir $SNDATA_ROOT/SIM
```

3.6 Examining the Output

3.6.1 Basic properties from the DUMP file

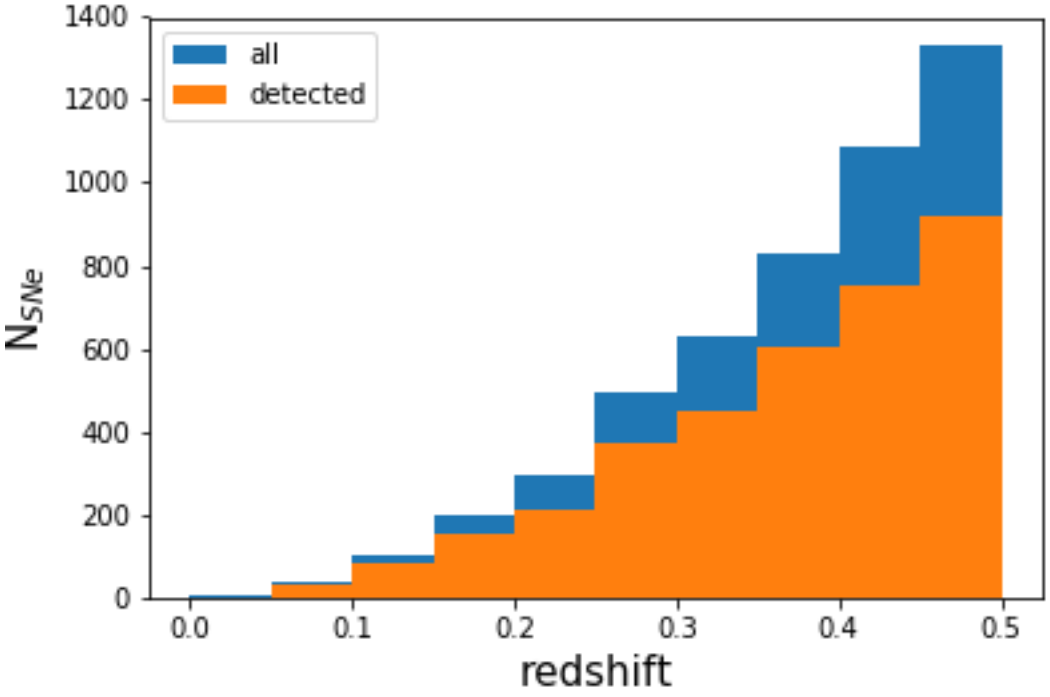
The simulated light curves are located in `$SNDATA_ROOT/SIM/<genversion/`. First, notice the `*.DUMP` file, which contains the variables specified in the `SIMGEN_DUMP` variable in your sim-input file. This is a good way to get some overview statistics for your sample.

For example, plotting the redshift and peak mag distribution of your simulation. Starting with some imports:

```
import numpy as np
import os
import matplotlib.pyplot as plt
from txtobj import txtobj

# read in the DUMP file
dmp = txtobj(os.path.expandvars('$SNDATA_ROOT/SIM/PS1MD/PS1MD.DUMP'),
↪fitresheader=True)
print(dmp.__dict__.keys())
```

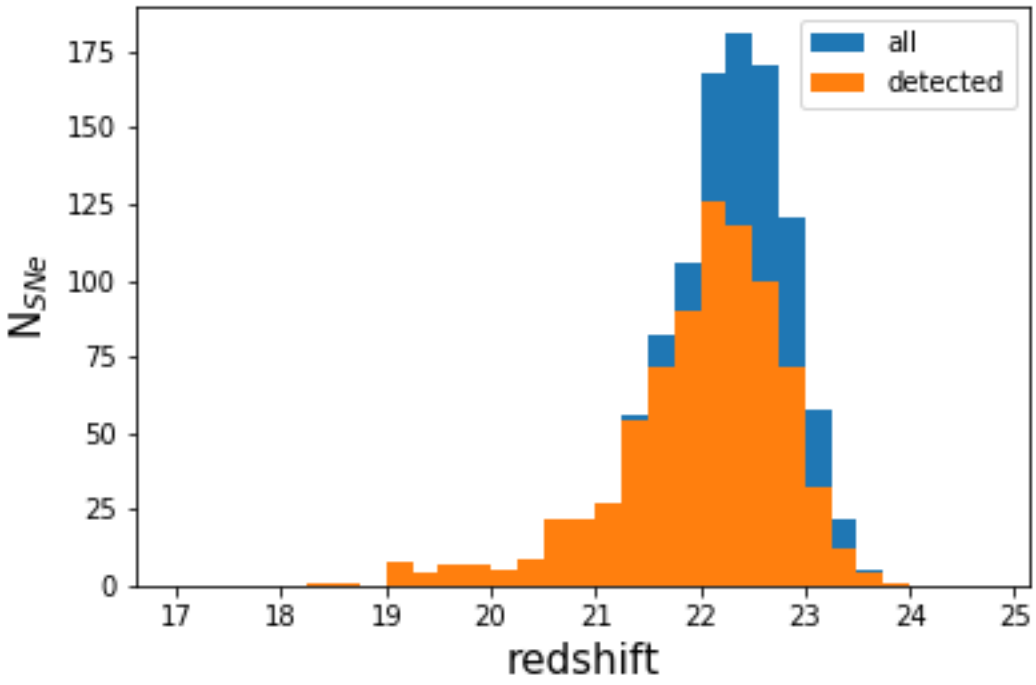
Plotting a redshift histogram for detected and undetected SNe:

```
bins = np.arange(0,0.55,0.05)
plt.hist(dmp.Z,label='all',bins=bins)
plt.hist(dmp.Z[dmp.MJD_TRIGGER != 1000000.000],bins=bins,label='detected')
plt.ylabel('N$_{SNe}$',fontsize=15)
plt.xlabel('redshift',fontsize=15)
plt.legend()
```



Plotting a peak magnitude distribution:

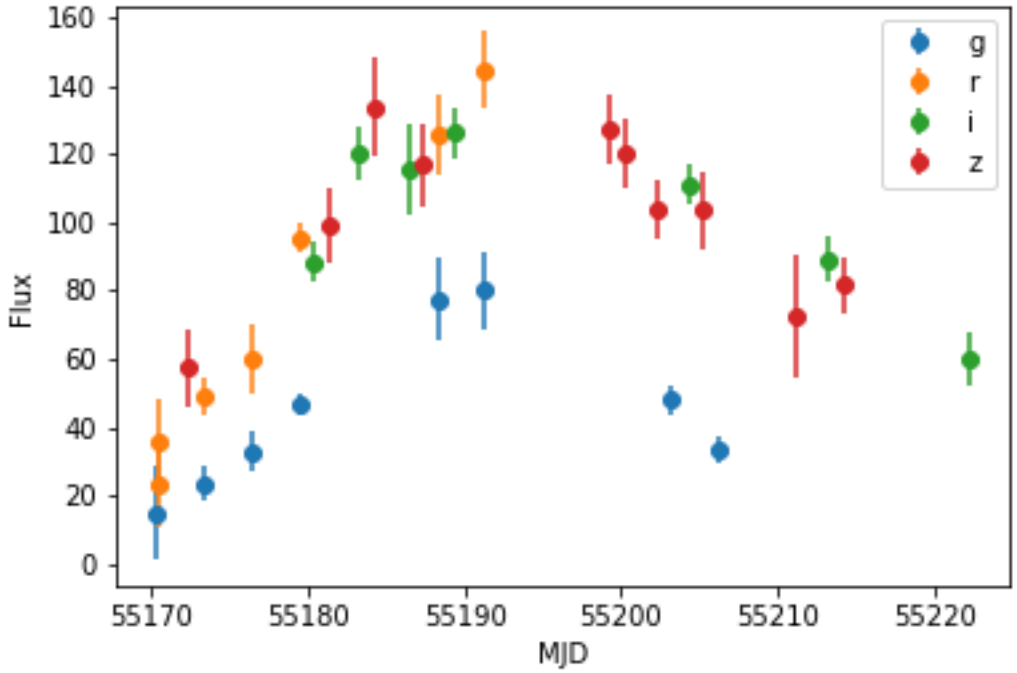
```
bins = np.arange(17,25,0.25)
plt.hist(dmp.MAGT0_r,label='all',bins=bins)
plt.hist(dmp.MAGT0_r[dmp.MJD_TRIGGER != 1000000.000],bins=bins,label='detected')
plt.ylabel('N$_{SNe}$',fontsize=15)
plt.xlabel('redshift',fontsize=15)
plt.legend()
```



3.6.2 The Simulated Light Curves

Let's take a look at some of the simulated ASCII files. These can be easily parsed with the *snana.py* script in the utils directory:

```
import numpy as np
import snana
import matplotlib.pyplot as plt
import os
import glob

lcfiles = glob.glob(os.path.expandvars('$SNDATA_ROOT/SIM/PS1MD/*DAT'))
sn = snana.SuperNova(lcfiles[10])
for f in sn.FILTERS:
    plt.errorbar(sn.MJD[sn.FLT == f],sn.FLUXCAL[sn.FLT == f],
                 yerr=sn.FLUXCALERR[sn.FLT == f],label=f,fmt='o')
plt.ylabel('Flux')
plt.xlabel('MJD')
plt.legend()
```



In FITS format, things are a little bit more annoying. Luckily, the LCs can be converted to a PANDAS dataframe thanks to some utilities from the PLAsTiCC team and Alexandre Boucaud (I modified them slightly).

As a reminder, use `FORMAT_MASK: 32` in your sim-input file to use FITS format and `FORMAT_MASK: 2` for ASCII format. A quick example for plotting a FITS-format light curve is below:

```
from sim_serializer import serialize
import matplotlib.pyplot as plt
datadict = serialize.main('PS1MD')
for key in datadict[list(datadict.keys())[20]].keys():
  if key == 'header': continue
  plt.errorbar(datadict[501][key]['mjd'],
               datadict[501][key]['fluxcal'],
               yerr=datadict[501][key]['fluxcalerr'],
               fmt='o',label=key)
plt.ylabel('flux')
plt.xlabel('mjd')
```

Please report any issues with this guide using the SNANA_StarterKit GitHub page.

CHAPTER 4

Fitting Light Curves (with SALT2)

Light curve fitting in SNANA uses the `snlc_fit.exe` program with a FORTRAN namelist file to provide paramers. It's easy enough to write, though a bit idiosyncratic (comments, for example, are exclamation marks).

I'm going to focus on fitting the SALT2 model here, which is the standard approach for most analyses. But SNANA can fit light curves using some of the other models in the `$SNDATA_ROOT/models` directory such as MLCS and SNooPy, for exampe.

4.1 Writing the Namelist (NML) File

The `snlc_fit.exe` namelist has three sections. First, the header section, which is only really used when running in batch mode (discussed briefly below). Next, the `SNLCINP` section, generally the SN light curve input parameters and the `FITINP` section, for the fitting input parameters.

The lines discussed below will be written to a file called `snfit_PS1.NML`.

4.1.1 The `SNLCINP` section

The SNLCINP section is bracketed by:

```
&SNLCINP

   ! a bunch of options here

&END
```

First, some options to find the input files, the filters, and which LC files to read in:

```
VERSION_PHOTOMETRY = 'PS1MD' ! name of your simulation or real data
KCOR_FILE          = '../kcor/kcor_PS1.fits' ! kcor file we made earlier
```

Basic outputs:

```
SNTABLE_LIST     = 'SNANA FITRES(text:key)' ! this is standard output format
TEXTFILE_PREFIX  = 'PS1MD' ! output filename (w/o the extension)
```

Milky Way reddening:

```
RV_MWCOLORLAW = 3.1
OPT_MWCOLORLAW = 99
OPT_MWEBV = 1
```

These lines give the value of R_V for the Milky Way, the Milky Way reddening colorlaw (99 = Fitzpatrick 1999), and `OPT_MWEBV = ` means `snlc_fit` will correct the light curves for MW reddening before fitting.

One `CUTWIN` option:

```
CUTWIN_REDSHIFT = 0.02, 1.0
```

The syntax here is that `CUTWIN_` followed by any variable can be used to make cuts (min,max) on the input light curves.

Then there are a few fit options that appear to not be placed in the `FITINP` section for reasons I don't totally understand:

```
NFIT_ITERATION = 3
INTERP_OPT     = 1
ABORT_ON_NOEPOCHS = F
LDMP_SNFAIL = T
```

Number of fitting iterations, the interpolation option (1 = linear interpolation), avoid aborting just because a SN cannot be fit, and report the reason for each SN that fails.

4.1.2 The `FITINP` section

Next, the fit options. Same as before, the section is denoted with:

```
&FITINP

   ! a bunch of options

&END
```

Then, identify the model and filter set:

```
FITMODEL_NAME  = 'SALT2.JLA-B14'
FILTLIST_FIT = 'griz'
```

This is SALT2.4, from Betoule et al. (2014). In a perfect world, you would know this from reading `$SNDATA_ROOT/models/SALT2/SALT2.JLA-B14/AAA_README`. However, in this case you'll have to just trust me.

Next, a prior on the time of maximum light and a window defining the range of rest-frame times (relative to peak) to include:

```
PRIOR_MJDSIG        = 25.0
TREST_REJECT  = -15.0, 45.0
```

And a few more things:

```
LFIXPAR_ALL      = F
OPT_COVAR = 1
OPT_COVAR_MWXTERR = 1
FUDGEALL_ITER1_MAXFRAC = 0.02
```

These just say, don't fix all the parameters, do use a model covariance matrix, do use Milky Way reddening uncertainty in the covariance matrix, and inflate the uncertainties by 2% for the first fitting iteration only.

That's it!

4.2 Running the Fit

As simple as:

```
snlc_fit.exe snfit_PS1.NML
```

4.3 Analyzing the Output

Recent versions of SNANA have cleaned up the output format a bit so that output files are easier to parse. The `FITRES` ASCII format gives a line starting with `VARNAMES` that identifies all the variables and subsequent lines starting with `SN` for the light curve parameters, metadata, and covariances.

Plotting light curves with their corresponding SALT2 fits however, is not (in my opinion) very intuitive. I usually resort to sncosmo and a custom script (below).

4.3.1 Plotting Light Curve Fits

I've included a clumsy python script, `txtobj.py` for reading FITRES files. But, it's easy enough to make your own, or use astropy utilities instead.

To read in the output FITRES file and generate a basic light curve plot like we had before in the simulations guide, first the basic imports:

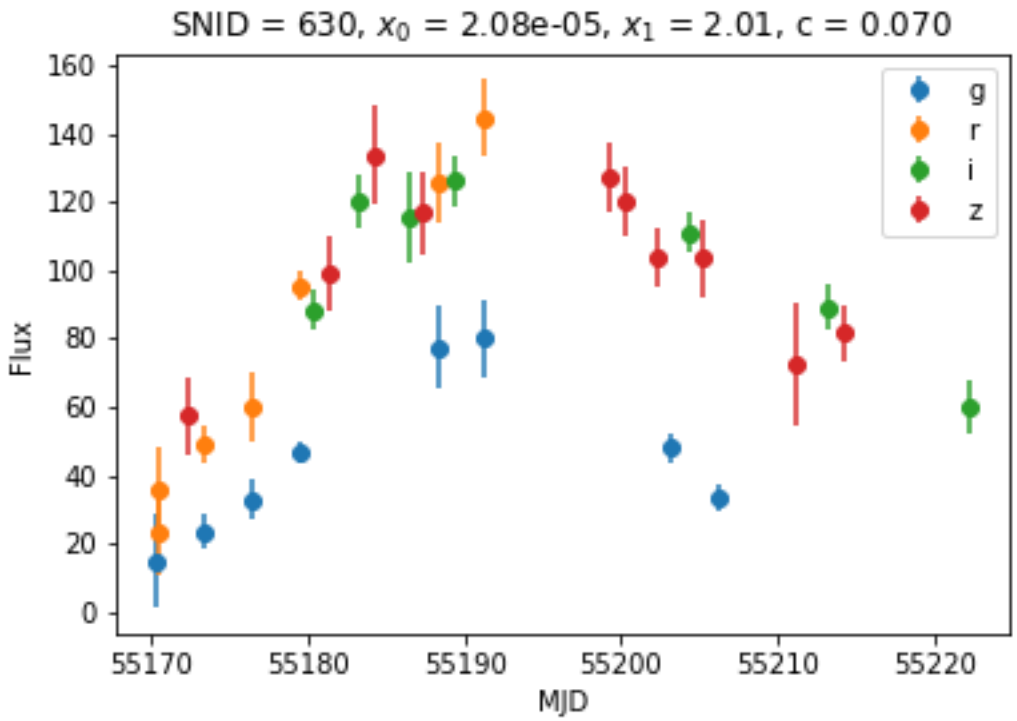
```
import glob
import os
import matplotlib.pyplot as plt
import numpy as np
from util import snana
from util.txtobj import txtobj
```

And then a few lines to read/plot stuff:

```
fr = txtobj('fitting/PS1MD.FITRES.TEXT',fitresheader=True)

lcfiles = glob.glob(os.path.expandvars('$SNDATA_ROOT/SIM/PS1MD/*DAT'))
sn = snana.SuperNova(lcfiles[10])
for f in sn.FILTERS:
    plt.errorbar(sn.MJD[sn.FLT == f],sn.FLUXCAL[sn.FLT == f],
                 yerr=sn.FLUXCALERR[sn.FLT == f],label=f,fmt='o')
plt.ylabel('Flux')
plt.xlabel('MJD')
plt.legend()
```

Now add the parameters from the fit:

```
iSN = fr.CID == sn.SNID
plt.title('SNID = %s, $x_0$ = %8.2e, $x_1$ = %.2f, c = %.3f'%(
          sn.SNID,fr.x0[iSN],fr.x1[iSN],fr.c[iSN]))
```



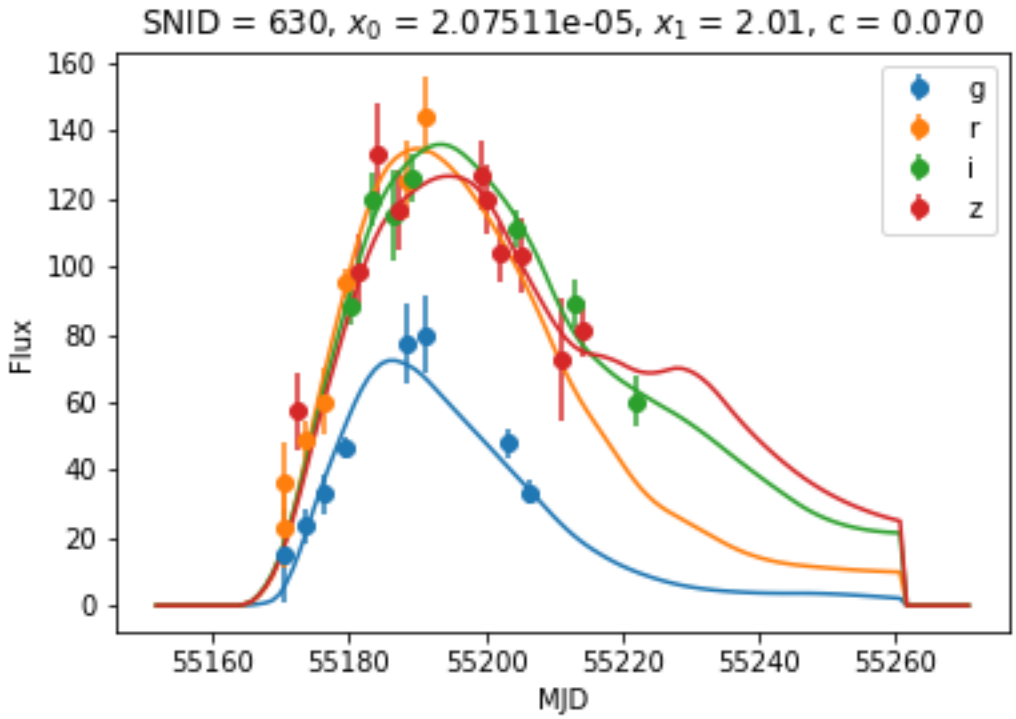
Now plot the SALT model (you'll need sncosmo for this example):

```
import sncosmo
from util import register

dust = sncosmo.F99Dust()
dust.set(ebv=float(sn.MWEBV.split()[0])) # pretty ugly but whatever

model = sncosmo.Model(source='salt2',effects=[dust],effect_names=['mwebv'],effect_
↪frames=['obs'])
# NOTE - SNANA SALT2 implementation has a 0.27 mag offset from sncosmo.
# this doesn't matter at all for most things like cosmology analyses,
# but certainly matters for making plots
model.set(z=fr.zHEL[iSN],t0=fr.PKMJD[iSN],
          x1=fr.x1[iSN],c=fr.c[iSN],x0=fr.x0[iSN]*10**(-0.4*(0.27)))

# register the filters to sncosmo using the kcor file.  sncosmo has
# pre-canned filter sets but it's best to be safe in a game where mmag
# offsets matter
mjd = np.arange(fr.PKMJD[iSN]-40,fr.PKMJD[iSN]+80,1)
register.from_kcor('kcor/kcor_PS1.fits')
for i,f in enumerate(sn.FILTERS):

    # careful about zpsys!
    salt2flux = model.bandflux(f, mjd, zp=27.5, zpsys='AB')
```

(continues on next page)

(continued from previous page)

```
    plt.plot(mjd,salt2flux,color='C%i'%i)
```



An important caveat - sncosmo won't check whether your filters are too red or blue for the SALT2 model. If your central filter wavelength in the SN rest frame is redder than 7000 Angstroms or bluer than 2800 Angstroms then SNANA did not include it in the fit!

4.3.2 Generating Distances and Making a Quick Hubble Diagram

More sophisticated ways of getting distances are discussed in the distances section, but for a small sample it's OK to use a simpler method. Namely, plugging light curve parameters into the Tripp formula and grabbing nuisance parameters from the latest and greatest cosmological analysis (at the time of writing, Scolnic et al. 2018).

The Tripp formula (Tripp 1999), to compute distance modulus μ

$$\mu = m_B - M + \alpha \times x_1 - \beta \times c$$

I'm excluding the mass step for these purposes, as well as things like bias corrections. Nuisance parameter M is approximately -19.36 for $H_0 = 70$ km s:math:^{-1} Mpc:math:^{-1}, while nuisance parameters α and $beta$ are xxx and yyy from the Pantheon analysis (non-bias corrected version).

This is simple in Python, but the errors have to take the covariances into account. I included a script `getmu.py` that does the work.

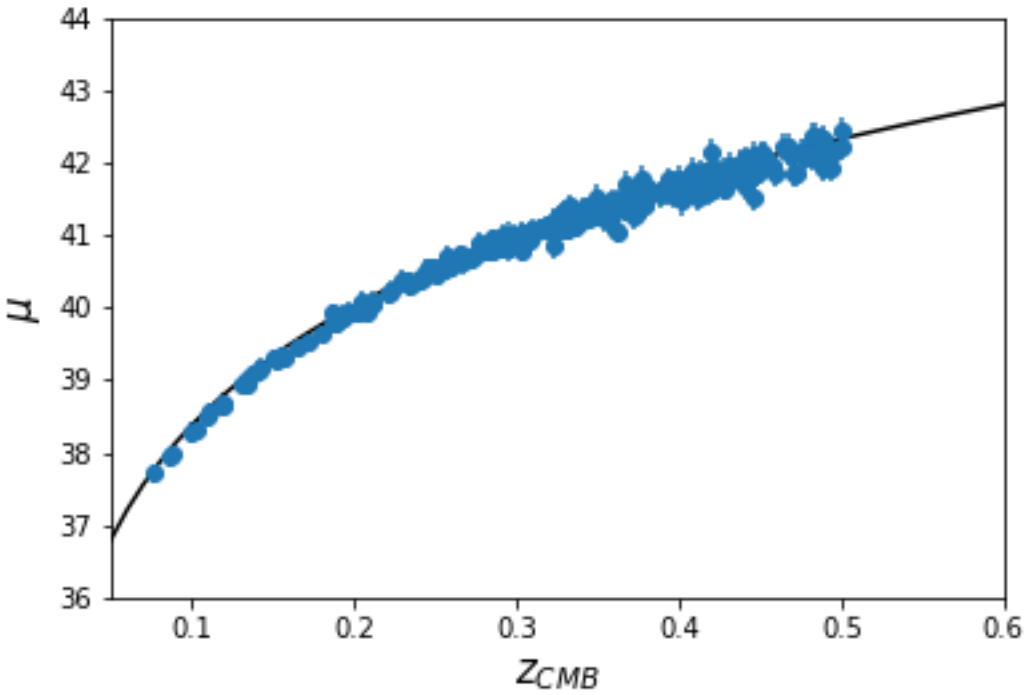
Making a Hubble diagram is pretty easy then:

```
# import our baseline cosmological params for comparison
from astropy.cosmology import Planck15 as cosmo
from util import getmu
```

(continues on next page)

(continued from previous page)

```
# gives the fitres object a "mu", "muerr" and "mures" attribute
fr = getmu.getmu(fr)
fr = getmu.mkcuts(fr)
iErr = fr.muerr < 0.2 # no crazy errors!

zrange = np.arange(0,1,0.01)
plt.plot(zrange,cosmo.distmod(zrange).value,color='k')
plt.errorbar(fr.zCMB[iErr],fr.mu[iErr],yerr=fr.muerr[iErr],fmt='o')
plt.xlabel('$z_{CMB}$',fontsize=15)
plt.ylabel('$\mu$',fontsize=15)
plt.xlim([0.05,0.6])
plt.ylim([36,44])
```



Note the SNe fall a little bit below the ΛCDM line. That's ok, it's just an artifact of H_0 and the SN absolute magnitude being degenerate. We marginalize over this global offset in cosmological analyses.

Done!

4.4 Running in Batch Mode

Check back later!

Please report any issues with this guide using the SNANA_StarterKit GitHub page.

CHAPTER 5

Measuring Distances from the Light Curve Fit Output

5.1 SALT2mu.exe

5.1.1 The BBC method

Please report any issues with this guide using the SNANA_StarterKit GitHub page.

CHAPTER 6

Measuring Approximate Cosmological Parameters (Quickly)

6.1 SALT2mu.exe

Please report any issues with this guide using the SNANA_StarterKit GitHub page.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search