SNTD Documentation

Release 2.0.0

Justin Pierel

Jun 23, 2021

Getting Started

CHAPTER 1

Installation

SNTD works on Python 3.5+ and requires the following Python packages:

- numpy
- scipy
- astropy
- SNCosmo
- cython
- sklearn
- nestle

1.1 Install using pip

Using pip:

```
pip install sntd
```

If you plan to use SNTD to simulate the effects of microlensing, which uses the [Wambsganss 1999] microlens code, then you will need a fortran compiler. You can install gfortran here.

1.2 Common Installation Issues

1. You will need a C compiler (e.g. `gcc` or `clang`) to be installed for the installation to succeed due to SNCosmo.
2. Sometimes there is an issue with the install related to numpy, particularly if you are installing in a fresh build of python (e.g. a new virtual environment). If this happens, try installing numpy using pip, and then installing SNTD using pip.

3. On MacOS and a fresh python build, you may have an issue with importing sntd because of matplotlib and an error saying python was not installed as a framework. A fix for that issue is on stack overflow.
4. Sometimes you can get an error when using the microlensing component of SNTD involving the *microlens* executable. This involves differences between the machine the microlens executable was created on and yours. Just go to the microlens github repo and follow the instructions for downloading and installing the wambsganss microlens code here, for the *with FITS-file output* option. When you have the *microlens* executable, just replace the SNTD executable in *sntd/sntd/microlens* with your new executable, and reinstall.

1.3 Install latest development version

SNTD is being developed on github. To get the latest development version using `git`:

```
git clone git://github.com/sntd.git
cd sntd
```

then:

```
tox
python setup.py install
```

tox can be install with *pip*, and runs the unit test suite (optional).

1.4 Optional dependencies

Several additional packages are recommended for enabling optional functionality in SNCosmo. These packages are all pip installable.

- matplotlib for plotting functions.
- pyParz for microlensing uncertainty estimation and fitting large numbers of lensed SN.
- corner used for plotting joint and marginalized fitting posteriors
- iminuit used for finding the best model quickly when fitting a list of models

CHAPTER 2

Using Your Own Data

In order to fit your own data, you must turn your light curve into an astropy table. There is an example multiply-imaged SN example provided for reference. In this example, we have a doubly-imaged SN with image files (in the `sntd/data/examples` folder, what you see when running this code may be slightly different) `example_image_1.dat` and `example_image_2.dat`. The only optional column in these files is `image`, which sets the name of the key used to reference this SN image. If you do not provide `flux`/`fluxerr` but instead `magnitude`/`magerr` SNTD will attemp to translate to `flux`/`fluxerr`, but it's best to simply provide flux from the beginning to avoid conversion errors. First we can read in these tables:

```
ex_1,ex_2=sntd.load_example_data()
print(ex_1)
```

Out:

```
time          band        flux              fluxerr         zp zpsys  image
 ------------------ ----- ------------------ ------------------ --- ----- -------
                0.0 F110W   7.62584933633316 1.0755597651249478  25    AB image_1
  8.421052631578947 F110W  18.14416283987298 1.1095627479983372  25    AB image_1
 16.842105263157894 F110W  20.89546596937984 1.0789885092102118  25    AB image_1
 25.263157894736842 F110W  18.79112295341033 1.0989312899693982  25    AB image_1
  33.68421052631579 F110W  18.02156628265722  1.12828303196303  25    AB image_1
  42.10526315789473 F110W 12.085231206169226 1.1179040967533518  25    AB image_1
 50.526315789473685 F110W  6.962961065741445 1.0339735940234938  25    AB image_1
  58.94736842105263 F110W   5.62540442388384 1.0359032050500712  25    AB image_1
  67.36842105263158 F110W 4.0345924798153305  1.01660451026687  25    AB image_1
  75.78947368421052 F110W 2.6846046764855243 1.0526764468991552  25    AB image_1
                ...   ...                ...                ... ...   ...     ...
 50.526315789473685 F160W 10.672033110263037  1.05358904440353  25    AB image_1
  58.94736842105263 F160W 10.152052390226839 1.0434013477890232  25    AB image_1
  67.36842105263158 F160W 7.6435469190906415 1.0575259139938555  25    AB image_1
  75.78947368421052 F160W  9.001903536381562 1.1095984293220644  25    AB image_1
  84.21052631578947 F160W  4.136105731166709 1.1086683584388777  25    AB image_1
  92.63157894736841 F160W  3.361829966775289  1.074708117236386  25    AB image_1
 101.05263157894737 F160W 2.3313546542843953 1.0952338255910397  25    AB image_1
 109.47368421052632 F160W 0.6946360860451652  1.088838824485642  25    AB image_1
```

(continues on next page)

(continued from previous page)

```
 117.89473684210526 F160W 2.3779217294236012  1.042417668422105  25     AB image_1
  126.3157894736842 F160W  3.834499918918735 1.0138401599728672  25     AB image_1
 134.73684210526315 F160W  0.623370473979624 1.0693683341393225  25     AB image_1
 Length = 33 rows
```

Now, to turn these two data tables into an *MISN* object that will be fit, we use the *table_factory()* function:

```
new_MISN=sntd.table_factory([ex_1,ex_2],telescopename='HST',object_name='example_SN')
print(new_MISN)
```

Out:

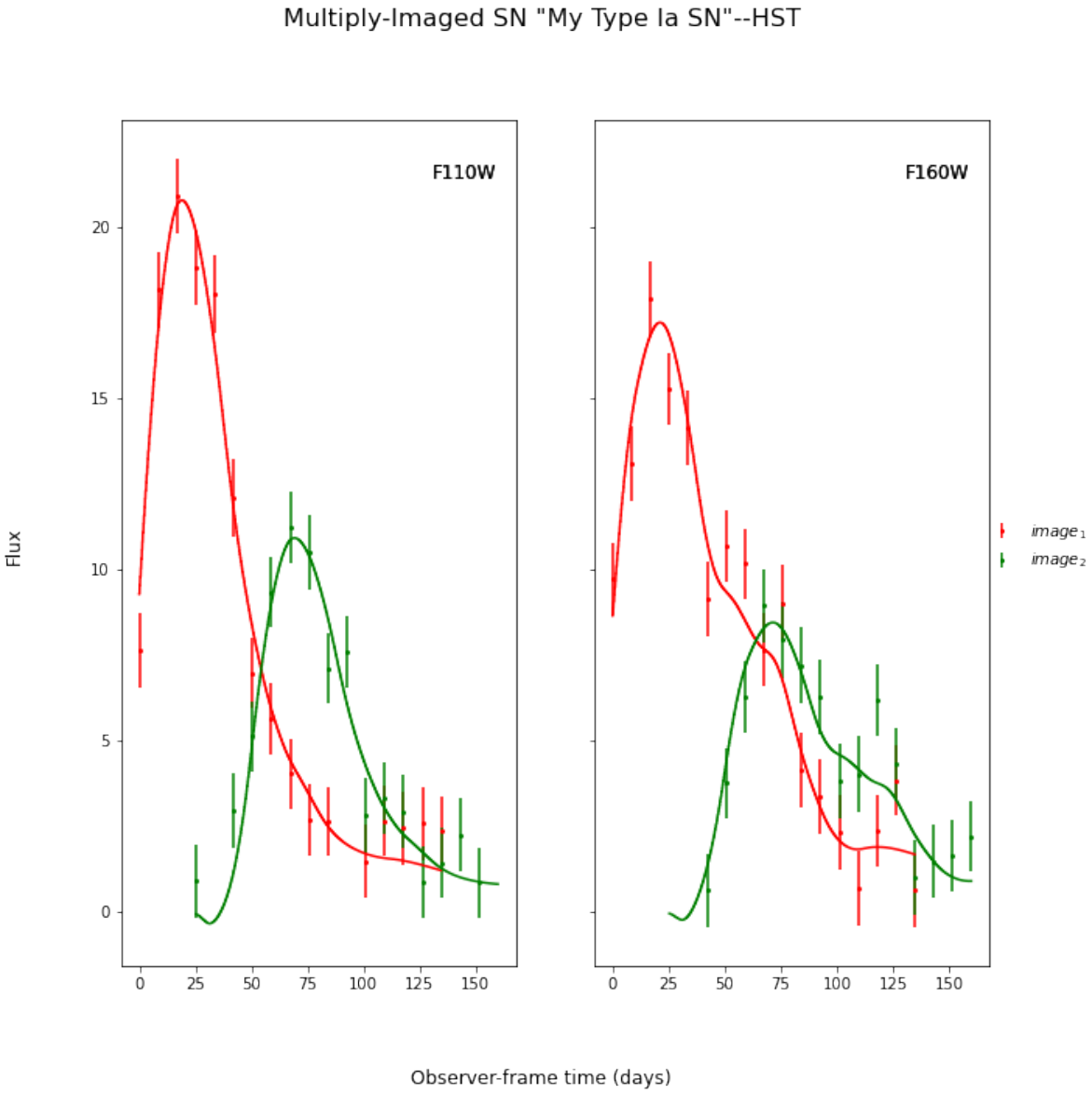
```
Telescope: HST
Object: example_SN
Number of bands: 2

------------------
Image: image_1:
Bands: {'F160W', 'F110W'}
Date Range: 0.00000->134.73684
Number of points: 33
------------------
Image: image_2:
Bands: {'F160W', 'F110W'}
Date Range: 25.26316->160.00000
Number of points: 30
------------------
```

And finally let's fit this SN, which is a Type Ia, with the SALT2 model (your exact time delay may be slightly different after fitting the example data). For reference, the true delay here is 50 days.

```
fitCurves=sntd.fit_data(new_MISN,snType='Ia', models='salt2-extended',bands=['F110W',
↪'F160W'],
                params=['x0','x1','t0','c'],constants={'z':1.4},
                bounds={'t0':(-25,25),'x1':(-2,2),'c':(-1,1)})
print(fitCurves.parallel.time_delays)
fitCurves.plot_object(showFit=True)
plt.show()
```

Out:

```
{'image_1': 0, 'image_2': 50.28975081972289}
```


Multiply-Imaged SN "My Type Ia SN"--HST
F110W
F160W
Flux
Observer-frame time (days)
image$_1$
image$_2$
0
5
10
15
20
0
25
50
75
100
125
150

CHAPTER 3

Uknown Supernova Type

You may not know that your SN is a Type Ia (as other examples here and in *Measure Time Delays*). In that case you have two more options. You could use the parameterized Bazin model:

```
fitCurves=sntd.fit_data(new_MISN,snType='Ia', models='Bazin',bands=['F160W'],
            params=['t0','B','amplitude','rise','fall'],refImage='image_1',cut_
↪time=None,
            bounds={'t0':(-20,20),'amplitude':(.1,100),'rise':(1,200),'fall':(1,200),
↪'B':(0,1),
                    'td':(-20,20),'mu':(.5,2)},
            fitOrder=['image_1','image_2'],fit_prior=None,minsnr=3,trial_fit=True,
            method='parallel',microlensing=None,modelcov=False,npoints=100,
                        maxiter=None)
fitCurves.plot_object(showFit=True)
print(fitCurves.parallel.time_delays)
```

Out:

```
{'image_1': 0, 'image_2': 49.40967718450336}
```

Another option is to fit multiple models from different SN types. SNTD will choose the "best" model using the Bayesian Evidence.

```
fitCurves=sntd.fit_data(new_MISN,snType='Ia', models=['salt2-extended','hsiao','snana-
↪2004gq',
    'snana-2004fe','snana-2004gv','snana-2007nc'],
    bands=['F110W','F140W'],cut_time=[-500,30],
    params=['x0','t0','x1','c','amplitude'],constants={'z':1.33},refImage='image_1',
    bounds={'t0':(-20,20),'x1':(-3,3),'c':(-1,1),'td':(-20,20),'mu':(.5,2)},
    fitOrder=['image_2','image_1'],trial_fit=True,minsnr=3,
    method='parallel',microlensing=None,modelcov=False,npoints=50,clip_data=True,
    maxiter=None)
```

CHAPTER 4

Batch Processing Time Delay Measurements

Parallel processing and batch processing is built into SNTD in order to fit a large number of (likely simulated) MISN. To access this feature, simply provide a list of MISN instead of a single *MISN* object, specifying whether you want to use multiprocessing (split the list across multiple cores) or batch processing (splitting the list into multiple jobs with sbatch). If you specify batch mode, you need to provide the partition and number of jobs you want to implement or the number of lensed SN you want to fit per node.

```
myMISN1 = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.33,z_lens=.53, bands=['F110W','F125W'],
                 zp=[26.8,26.2], cadence=5., epochs=35.,time_delays=[10., 70.],␣
↪magnifications=[7,3.5],
     objectName='My Type Ia SN',telescopename='HST')
myMISN2 = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.33,z_lens=.53, bands=['F110W','F125W'],
                 zp=[26.8,26.2], cadence=5., epochs=35.,time_delays=[10., 50.],␣
↪magnifications=[7,3.5],
     objectName='My Type Ia SN',telescopename='HST')
curve_list=[myMISN1,myMISN2]
fitCurves=sntd.fit_data(curve_list,snType='Ia', models='salt2-extended',bands=['F110W
↪','F125W'],
                  params=['x0','t0','x1','c'],constants={'z':1.3},refImage='image_1',
                  bounds={'t0':(-20,20),'x1':(-3,3),'c':(-1,1)},fitOrder=['image_2',
↪'image_1'],
                  method='parallel',npoints=1000,par_or_batch='batch', batch_
↪partition='myPartition',nbatch_jobs=2)

for curve in fitCurves:
  print(curve.parallel.time_delays)

fitCurves=sntd.fit_data(curve_list,snType='Ia', models='salt2-extended',bands=['F110W
↪','F125W'],
                  params=['x0','t0','x1','c'],constants={'z':1.3},refImage='image_1',
                  bounds={'t0':(-20,20),'x1':(-3,3),'c':(-1,1)},fitOrder=['image_2',
↪'image_1'],
                  method='parallel',npoints=1000,par_or_batch='parallel')
```

(continues on next page)

(continued from previous page)

```
for curve in fitCurves:
  print(curve.parallel.time_delays)
```

Out:

```
Submitted batch job 5784720
{'image_1': 0, 'image_2': 60.3528844834}
{'image_1': 0, 'image_2': 40.34982372733}
Fitting MISN number 1...
Fitting MISN number 2...
{'image_1': 0, 'image_2': 60.32583528844834}
{'image_1': 0, 'image_2': 40.22834982372733}
```

You can also use batch processing and multiprocssing (using N cores per node across M cores using Slurm):

```
fitCurves=sntd.fit_data(curve_list,snType='Ia', models='salt2-extended',bands=['F110W
↪','F125W'],
            params=['x0','t0','x1','c'],constants={'z':1.3},refImage='image_1',
            bounds={'t0':(-20,20),'x1':(-3,3),'c':(-1,1)},fitOrder=['image_2','image_1
↪'],n_cores_per_node=2,
            method='parallel',npoints=1000,par_or_batch='batch', batch_partition=
↪'myPartition',nbatch_jobs=1)
```

If you would like to run multiple methods in a row in batch mode, the recommended way is by providing a list of the methods to the *fit_data()* function. You can have it use the parallel fit as a prior on the subsequent fits by setting `fit_prior` to `True` instead of giving it a *MISN* object.

```
fitCurves_batch=sntd.fit_data(curve_list,snType='Ia', models='salt2-extended',bands=[
↪'F110W','F125W'],
                params=['x0','t0','x1','c'],constants={'z':1.3},refImage='image_1',
↪fit_prior=True,
                bounds={'t0':(-20,20),'x1':(-3,3),'c':(-1,1)},fitOrder=['image_2',
↪'image_1'],
                method=['parallel','series','color'],npoints=1000,par_or_batch=
↪'batch', batch_partition='myPartition',nbatch_jobs=2)
```

CHAPTER 5

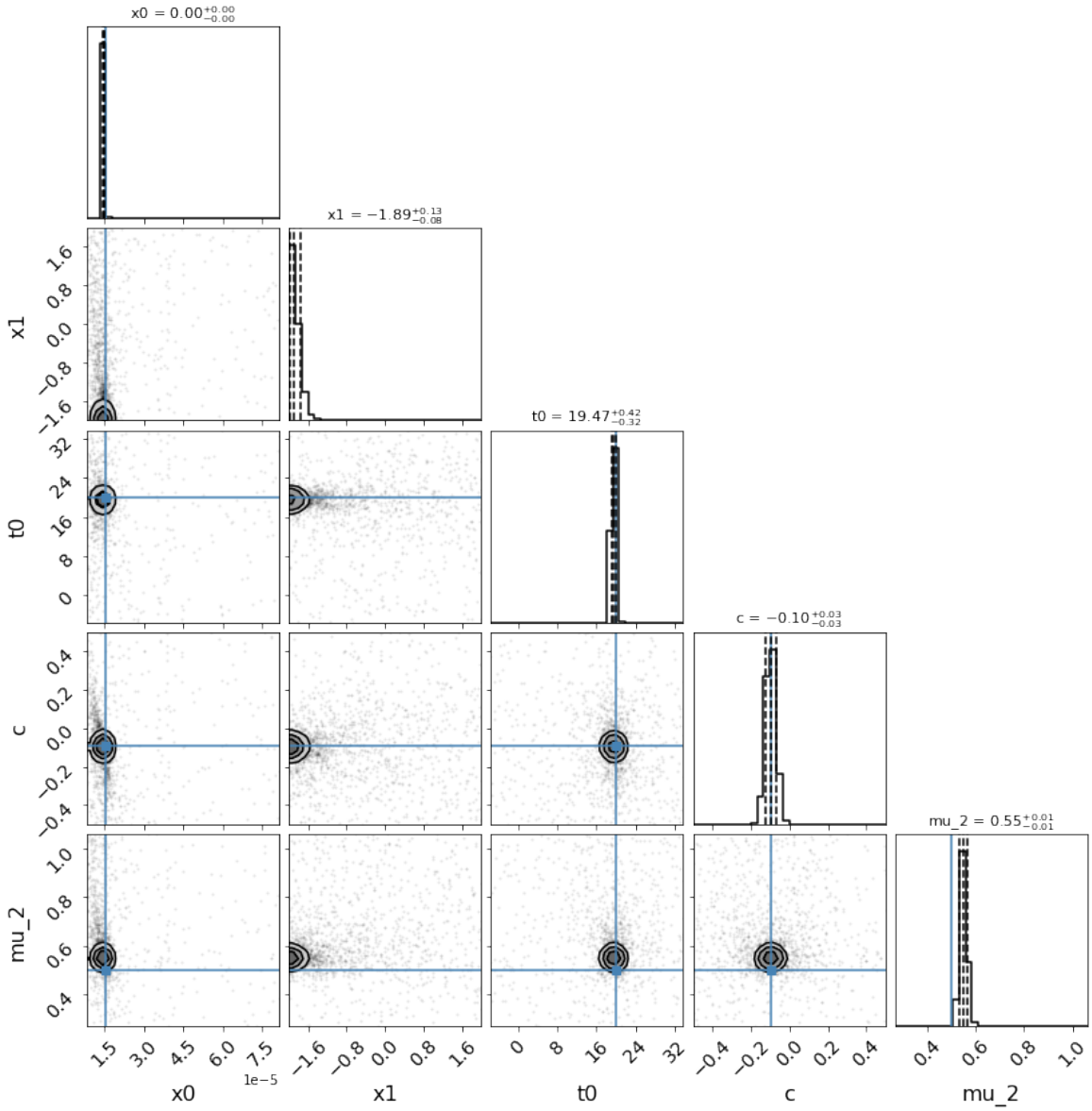
Fitting Model Params Only

You can also easily fix the time delays (for the series or color method) and/or magnifications (series method only) while fitting a light curve model:

```
myMISN = sntd.load_example_misn()
fitCurves=sntd.fit_data(myMISN,snType='Ia',models='salt2-extended',bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c'],constants={'z':1.4,'td':{'image_1
↪':0,'image_2':50}},
                        refImage='image_1',trial_fit=False,
                        bounds={'t0':(-20,20),'mu':(.5,2),'x1':(-2,2),'c':(-.5,.5)},
                        method='series',npoints=100)
print(fitCurves.series.time_delays)
print(fitCurves.series.magnifications)
fitCurves.plot_fit(method='series')
plt.show()
```

Out:

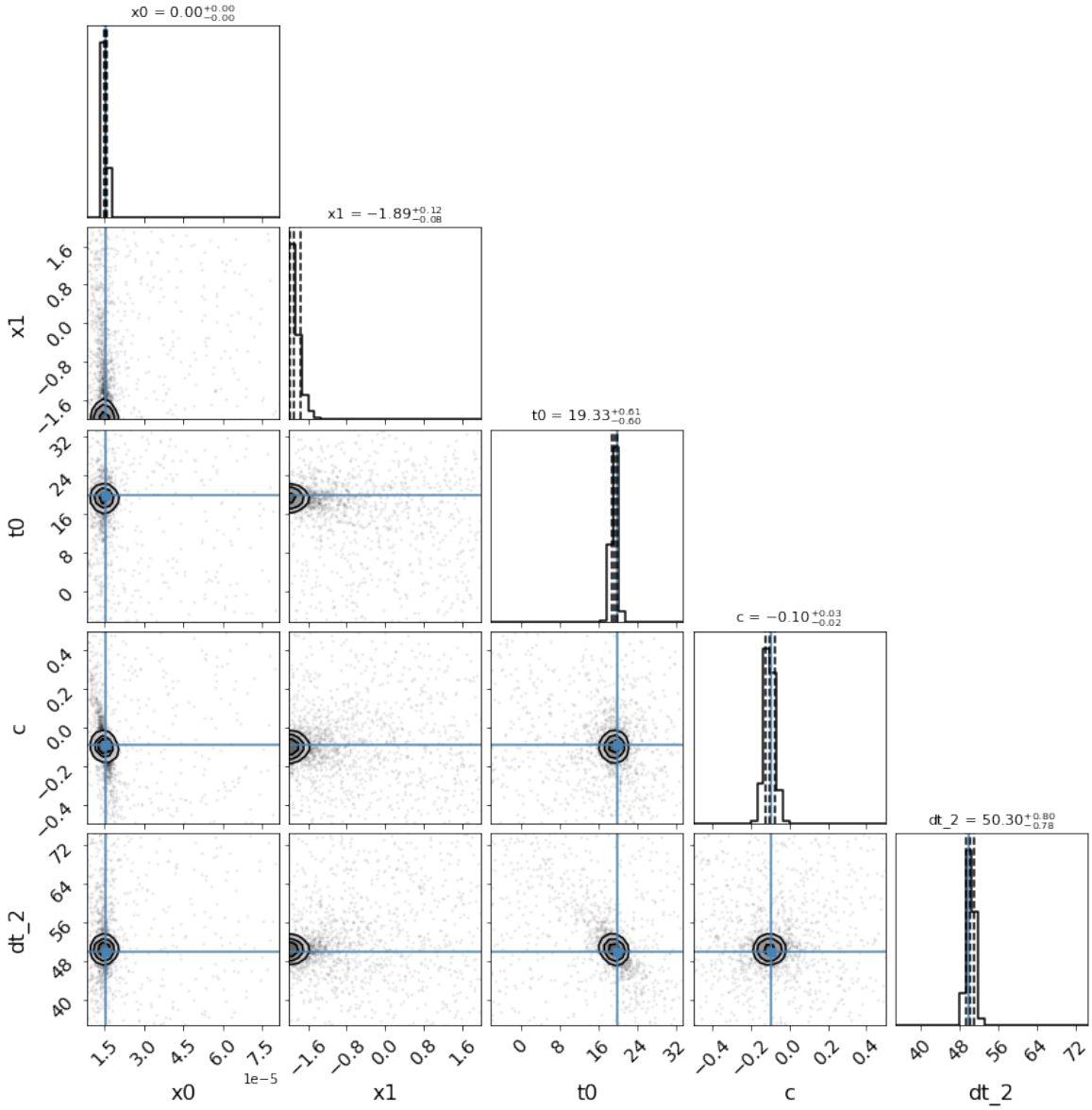
```
{'image_1': 0, 'image_2': 50}
{'image_1': 1, 'image_2': 0.5496620676889558}
```




```
myMISN = sntd.load_example_misn()
fitCurves=sntd.fit_data(myMISN,snType='Ia',models='salt2-extended',bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c'],constants={'z':1.4,'mu':{'image_1
↪':1,'image_2':0.5}},
                        refImage='image_1',trial_fit=False,
                        bounds={'t0':(-20,20),'mu':(.5,2),'x1':(-2,2),'c':(-.5,.5)},
                        method='series',npoints=100)
print(fitCurves.series.time_delays)
print(fitCurves.series.magnifications)
fitCurves.plot_fit(method='series')
plt.show()
```

Out:

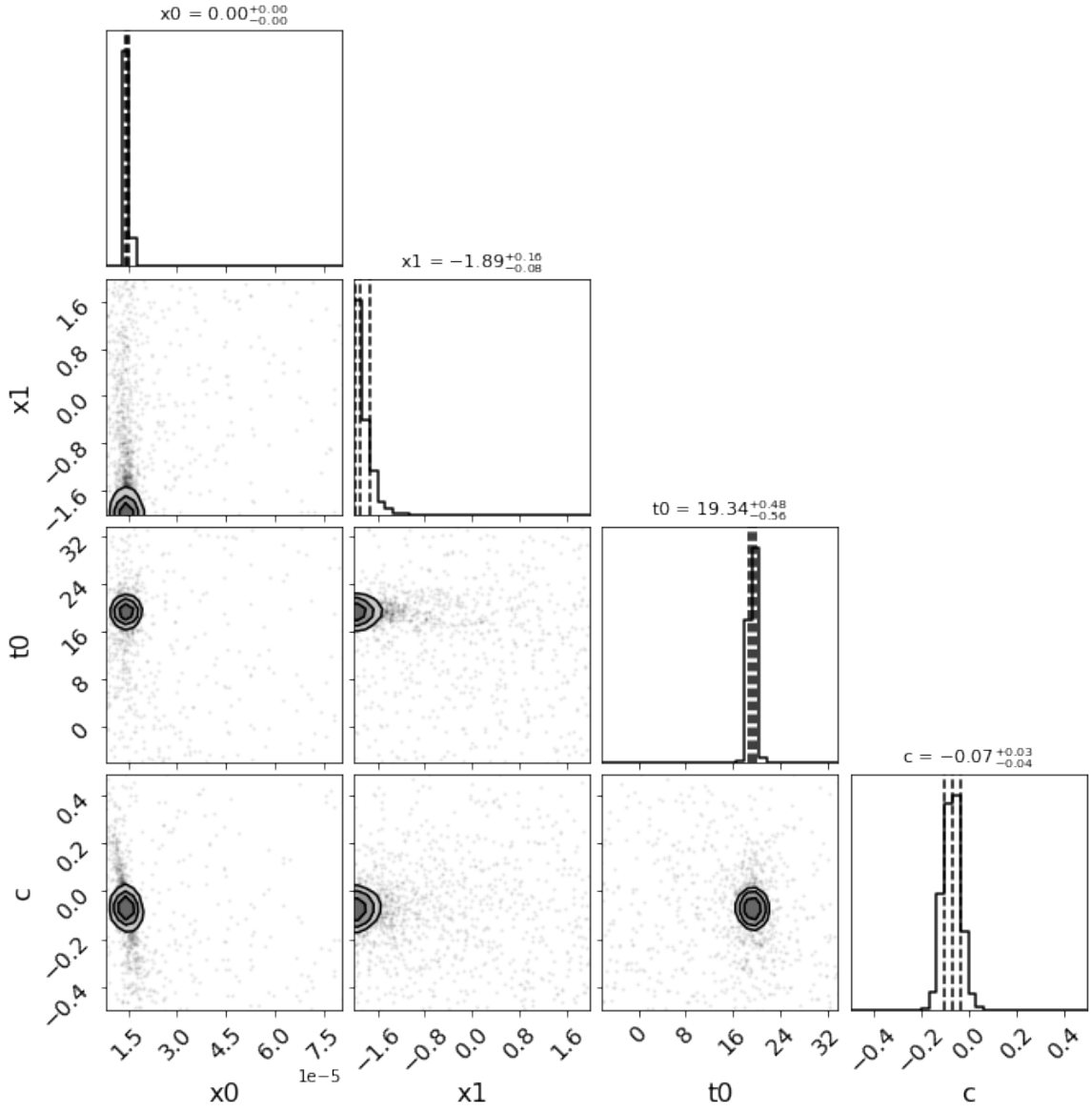
```
{'image_1': 0, 'image_2': 50.29975290189155}
{'image_1': 1, 'image_2': 0.5}
```




```
myMISN = sntd.load_example_misn()
fitCurves=sntd.fit_data(myMISN,snType='Ia',models='salt2-extended',bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c'],constants={'z':1.4,'td':{'image_1
↪':0,'image_2':50},'mu':{'image_1':1,'image_2':0.5}},
                        refImage='image_1',trial_fit=False,
                        bounds={'t0':(-20,20),'mu':(.5,2),'x1':(-2,2),'c':(-.5,.5)},
                        method='series',npoints=100)
print(fitCurves.series.time_delays)
print(fitCurves.series.magnifications)
fitCurves.plot_fit(method='series')
plt.show()
```

Out:

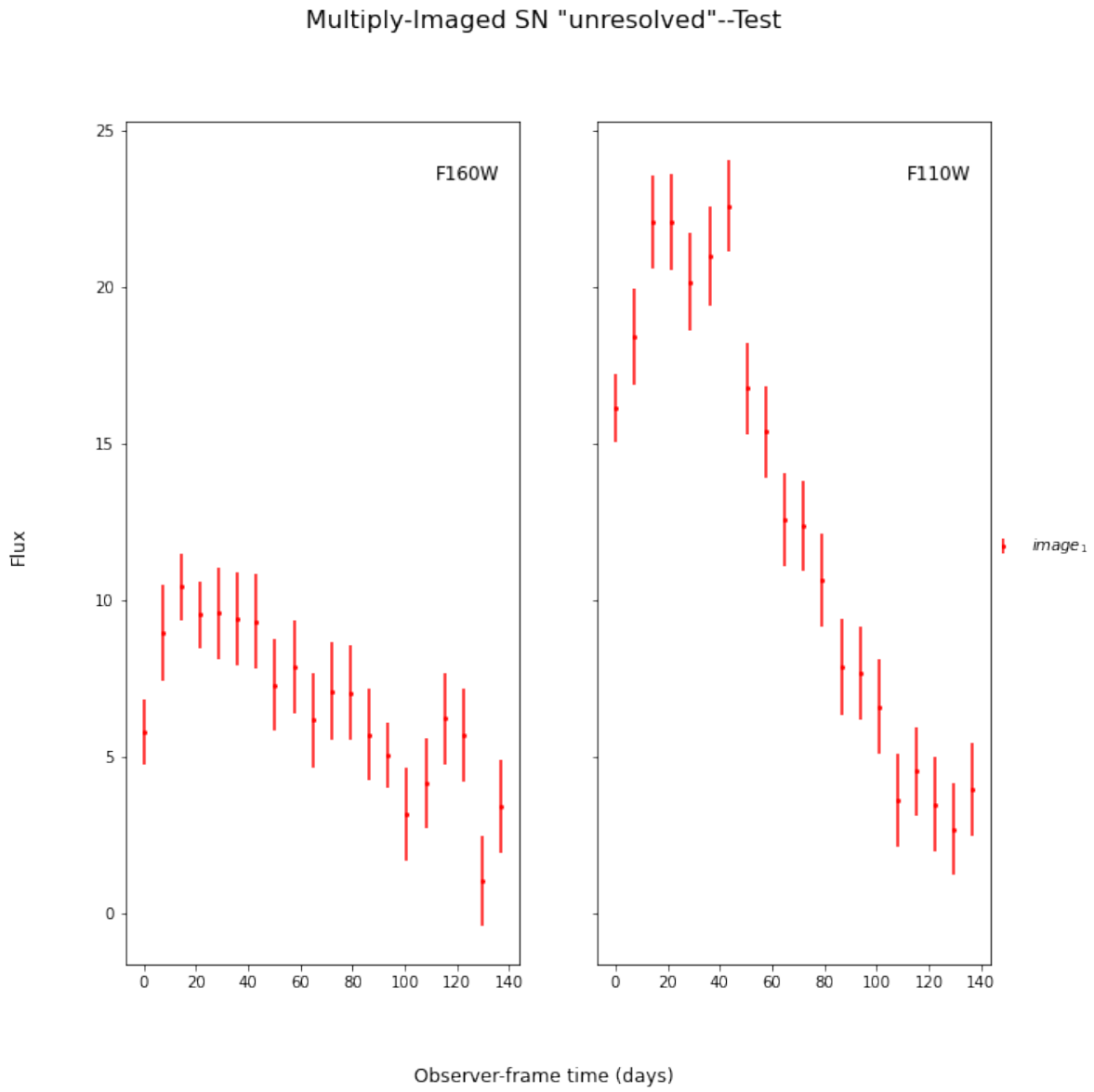
```
{'image_1': 0, 'image_2': 50}
{'image_1': 1, 'image_2': 0.5}
```



CHAPTER 6

Unresolved Lensed SN Models

SNTD now includes an option for fitting unresolved multiply-imaged light curves. Here is an example of an unresolved light curve of a doubly-imaged SN Ia:



This object has only a single image, whose table is the combined light curve shown above. Simply create SNCosmo models for each blended image, and combine them in the following way:

```
image_1 = sncosmo.Model('salt2-extended')
image_2 = sncosmo.Model('salt2-extended')
unresolved=sntd.unresolvedMISN([image_1,image_2])
```

You can set the time delays and magnifications with two simple functions:

```
unresolved.set_delays([20,55])
unresolved.set_magnifications([4,1])
print(unresolved.model_list[0].parameters)
print(unresolved.model_list[1].parameters)
```

Out:

```
[ 0. 20. 4. 0. 0. ]
[ 0. 55. 1. 0. 0. ]
```

You may now set parameters as you would for a normal model. SN parameters (i.e. z, x1, c, etc.) will set across all models, "lens" parameters (i.e. t0, x0/amplitude, etc.) will shift each model by that amount. For example (note the time delays above):

```
unresolved.set(z=1.4)
unresolved.set(t0=10)
print(unresolved.param_names)
print(unresolved.parameters)
print(unresolved.model_list[0].parameters)
print(unresolved.model_list[1].parameters)
```

Out:

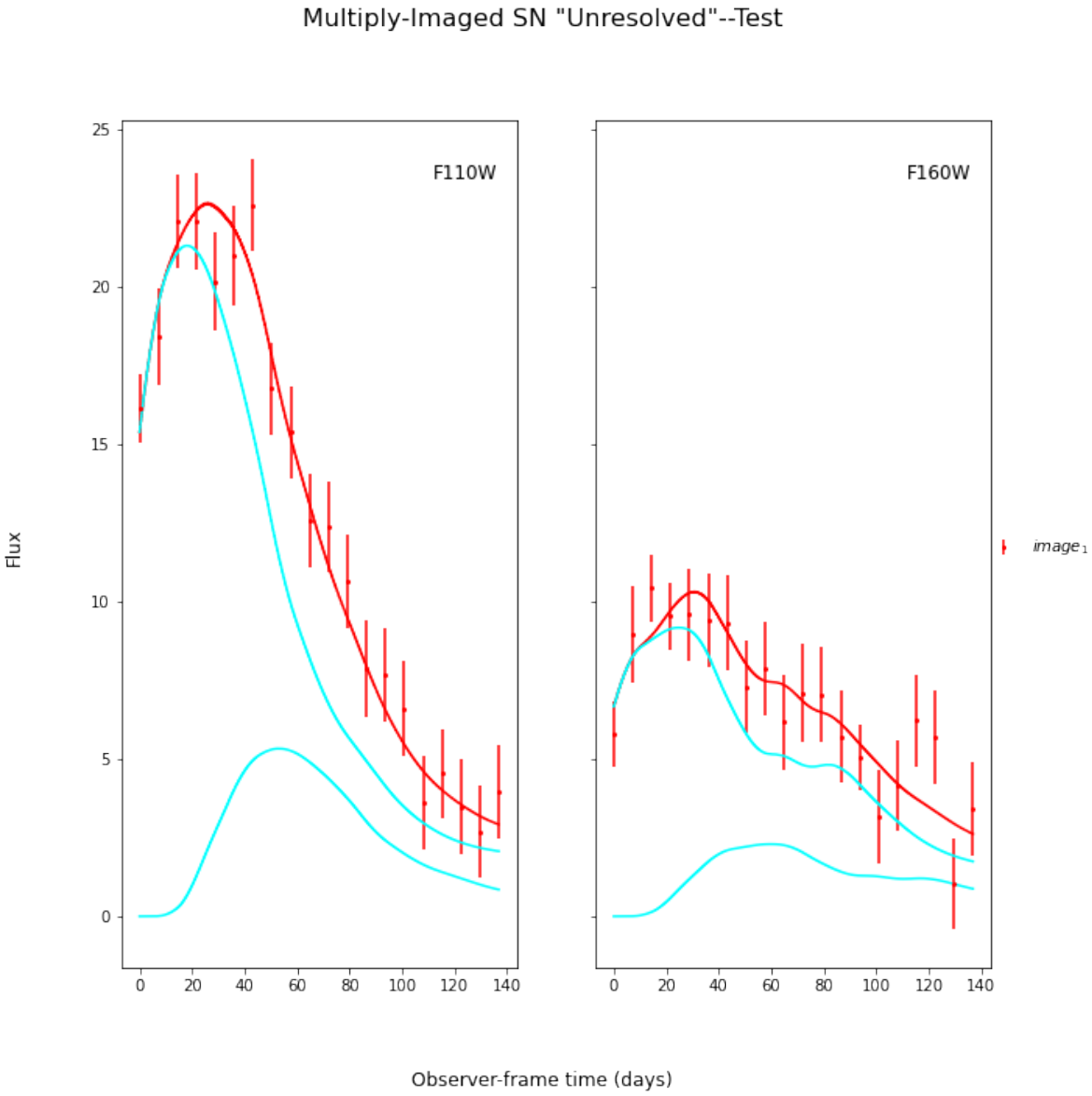
```
['z', 't0', 'x0', 'x1', 'c', 'dt_1', 'mu_1', 'dt_2', 'mu_2']
[ 1.4 10.   1.   0.   0. 20. 4. 55. 1. ]
[ 1.4 30.   4.   0.   0. ]
[ 1.4 65.   1.   0.   0. ]
```

Now this unresolved model can be simply used in the SNTD fitting methods as normal:

```
fitCurves=sntd.fit_data(combined_MISN,snType='Ia',models=unresolved,bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c'],constants={'z':1.4},
                        bounds={'t0':(-20,20),'x1':(-2,2),'c':(-.5,.5)},
                        method='parallel',npoints=100)

print(list(zip(fitCurves.images['image_1'].fits.model.param_names,
  fitCurves.images['image_1'].fits.model.parameters)))
fitCurves.plot_object(showFit=True,plot_unresolved=True)
plt.show()
```

Out:

```
[('z', 1.4), ('t0', -0.7620832162021888), ('x0', 7.298888590029448e-07), ('x1', 1.
↪4960457248192203), ('c', -0.08384340717738858), ('dt_1', 20.0), ('mu_1', 4.0), ('dt_
↪2', 55.0), ('mu_2', 1.0)]
```



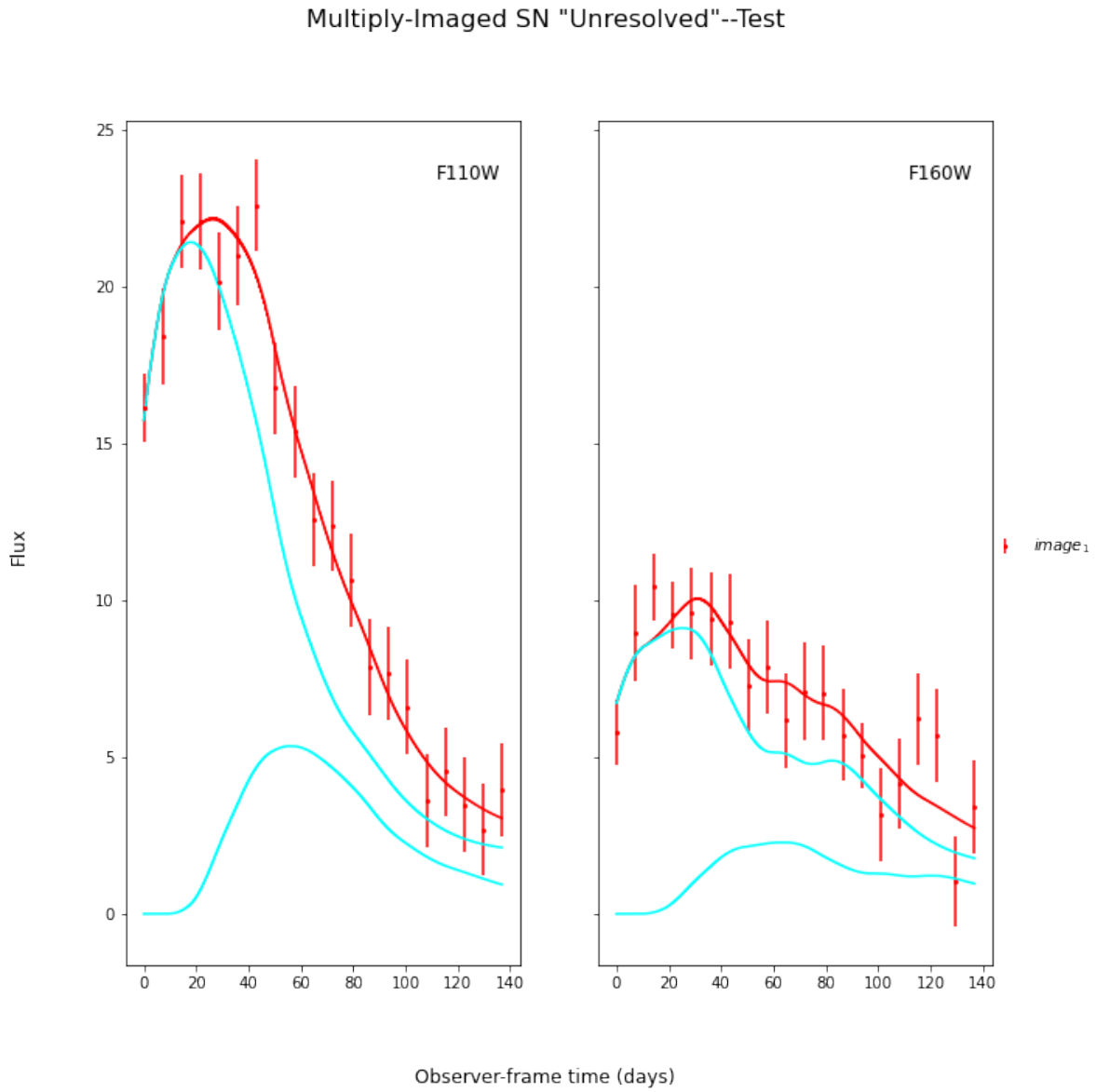
Here the true parameters were t0=0 (the input time delays were correct, and t0 is fitting for a global offset in the model), x1=-1.33, c=-.02, and x0 is somewhat arbitrary here as the magnifications were also correct based on the sim.

Additionally, we can attempt to fit for relative time delays/magnifications of the unresolved images.

```
fitCurves=sntd.fit_data(combined_MISN,snType='Ia',models=unresolved,bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c','dt_2'],
                        constants={'z':1.4,'mu_1':2,'mu_2':.5},
                        bounds={'t0':(-20,20),'dt_2':(20,40),'x1':(-2,2),'c':(-.5,.5)}
↪,
                        method='parallel',npoints=100)

print(list(zip(fitCurves.images['image_1'].fits.model.param_names,
  fitCurves.images['image_1'].fits.model.parameters)))
fitCurves.plot_object(showFit=True,plot_unresolved=True)
plt.show()
```

Out:

```
[('z', 1.4), ('t0', 19.203237085466867), ('x0', 7.349261421179482e-07), ('x1', 1.
↪682206180032783), ('c', -0.0909991294403899), ('dt_1', 0.0), ('mu_1', 4.0), ('dt_2',
↪ 38.205546125150704), ('mu_2', 1.0)]
```




```
fitCurves=sntd.fit_data(combined_MISN,snType='Ia',models=unresolved,bands=['F110W',
↪'F160W'],
                        params=['x0','x1','t0','c','mu_1',dt_2','mu_2'],
                        constants={'z':1.4},
                        bounds={'t0':(-20,20),'mu_1':(3.5,4.5),'mu_2':(.5,1.5),
                                'dt_2':(20,80),'x1':(-3,3),'c':(-.5,.5)},
                        method='parallel',npoints=100)

print(list(zip(fitCurves.images['image_1'].fits.model.param_names,
  fitCurves.images['image_1'].fits.model.parameters)))
```

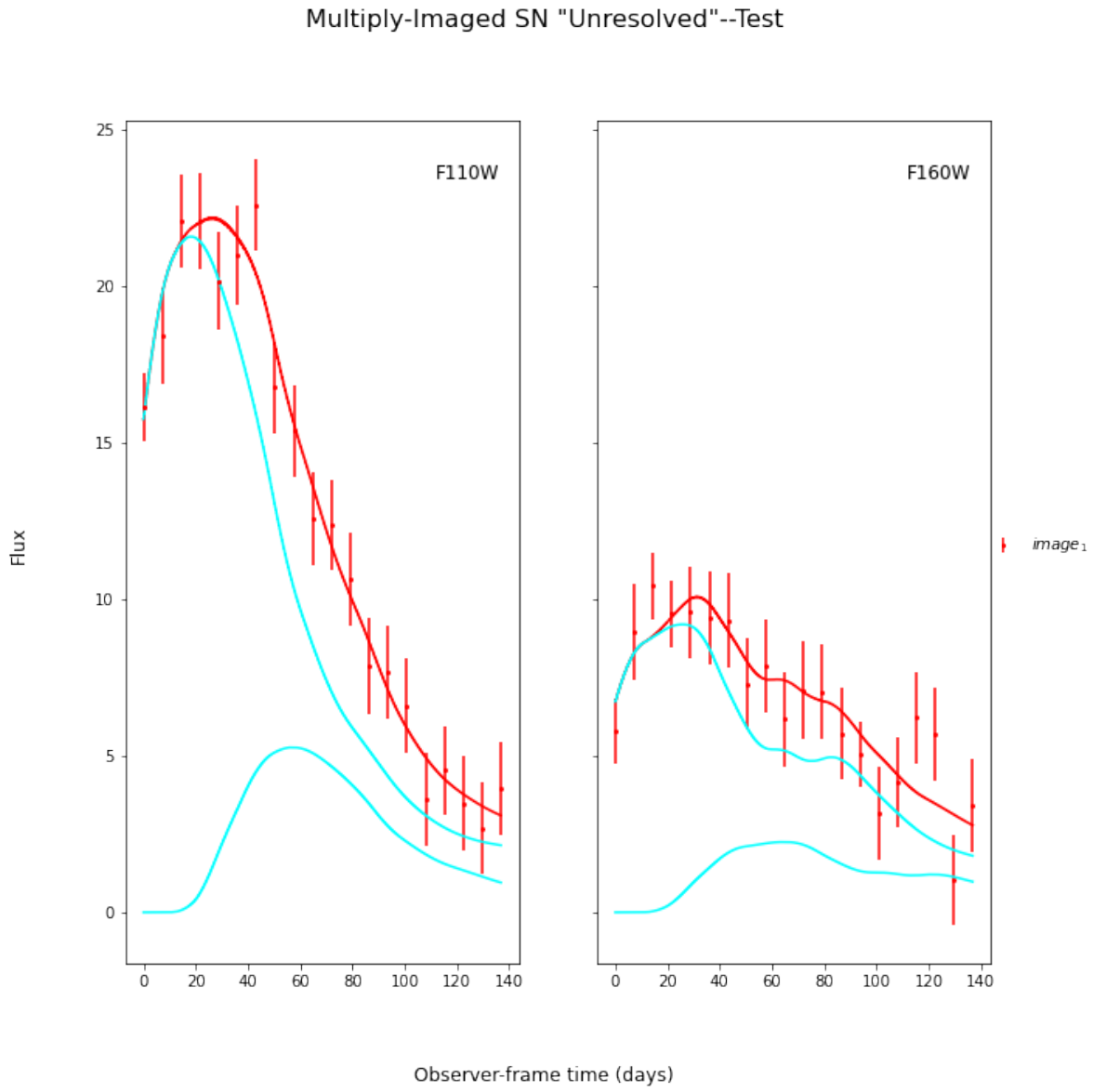
(continues on next page)

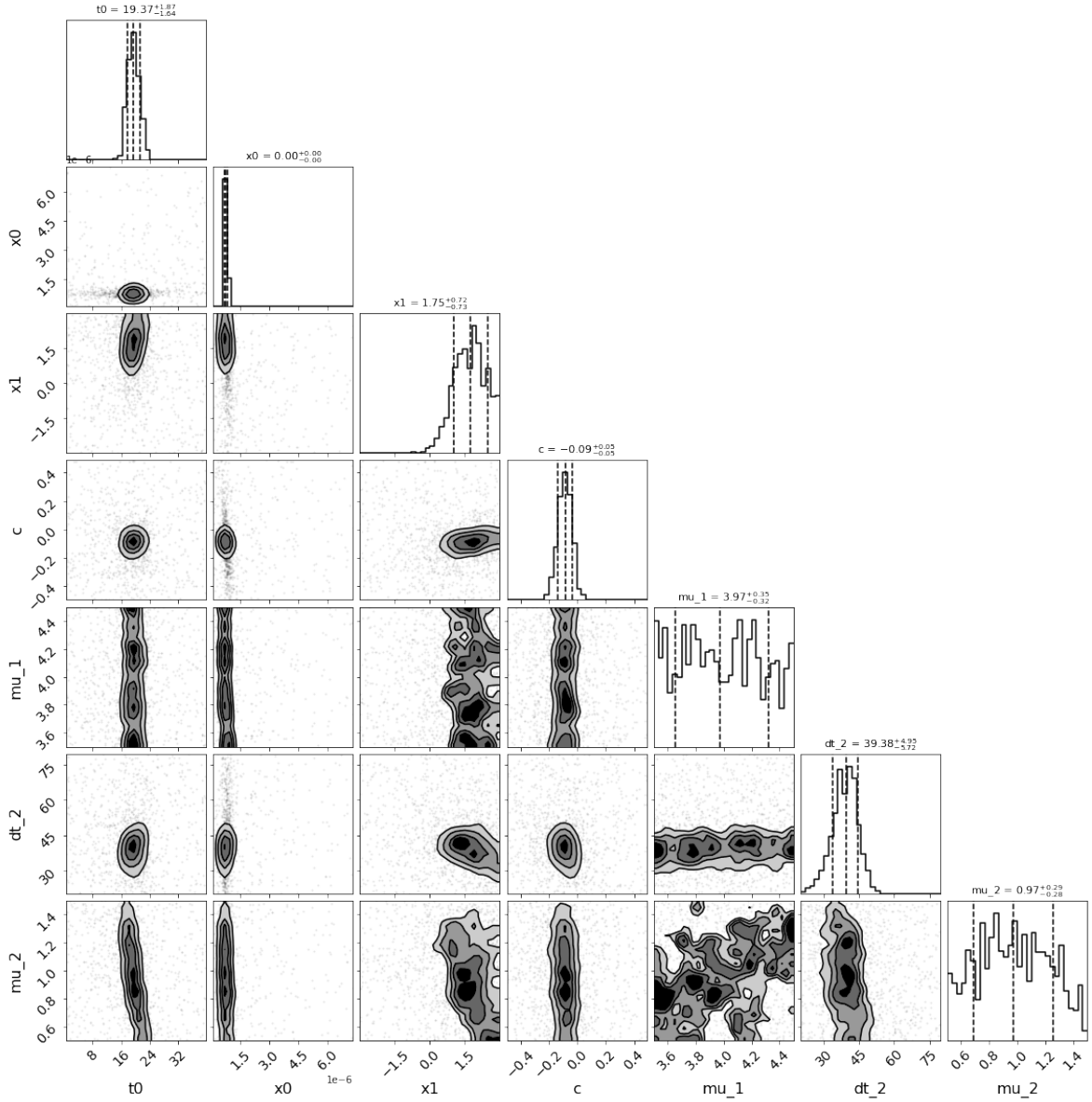
(continued from previous page)

```
fitCurves.plot_object(showFit=True,plot_unresolved=True)
fitCurves.plot_fit()
plt.show()
```

Out:

```
[('z', 1.4), ('t0', 19.49243682866722), ('x0', 7.43527262511094e-07), ('x1', 1.
↪7221753624460694), ('c', -0.08879485874943827), ('dt_1', 0.0), ('mu_1', 3.
↪980681598438305), ('dt_2', 38.91567004151568), ('mu_2', 0.9718061929858269)]
```





For these last two examples, there is a lot of parameter degeneracy but we still do reasonably well. The true values are t0=20,dt_2=35,mu_1=4,mu_2=1, c=-0.02., x1=1.33.

Note that you can add individual propagation effects to each images model at the outset (such as lens plane dust or microlensing models), or you could add a single propagation effect to the unresolved model that will impact the combined model (such as host galaxy dust):

```
dust = sncosmo.F99Dust()
image_1 = sncosmo.Model('salt2-extended',effects=[dust],
                                    effect_names=['lens_1'],
                                    effect_frames=['free'])
image_2 = sncosmo.Model('salt2-extended')
unresolved=sntd.unresolvedMISN([image_1,image_2])
print(list(zip(unresolved.param_names,unresolved.parameters)))
```

Out:

```
[('z', 0.0), ('t0', 0.0), ('x0', 1.0), ('x1', 0.0), ('c', 0.0), ('lens_1z', 0.0), (
↪'lens_1ebv', 0.0), ('lens_1r_v', 3.1), ('dt_1', 0.0), ('mu_1', 1.0), ('dt_2', 0.0),␣
↪('mu_2', 1.0)]
```

```
unresolved.set(lens_1ebv=.5)
print(unresolved.get('lens_1ebv'))
unresolved.add_effect(dust,'host','rest')
print(list(zip(unresolved.param_names,unresolved.parameters)))
unresolved.set(hostebv=.3)
print(unresolved.get('hostebv'))
print(unresolved.model_list[0].effect_names)
```

Out:

```
0.5
[('z', 0.0), ('t0', 0.0), ('x0', 1.0), ('x1', 0.0), ('c', 0.0), ('lens_1z', 0.0), (
↪'lens_1ebv', 0.5), ('lens_1r_v', 3.1), ('dt_1', 0.0), ('mu_1', 1.0), ('dt_2', 0.0),␣
↪('mu_2', 1.0), ('hostebv', 0.0), ('hostr_v', 3.1)]
0.3
['lens_1', 'host']
```

Note that the number of parameters is getting extremely long (and this is only 2 images!), so it's recommended to fix as many of these parameters by other means as possible (i.e., the dust parameters will be difficult to constrain simultaneously with the light curve and time delay parameters).

orphan

CHAPTER 7

Examples

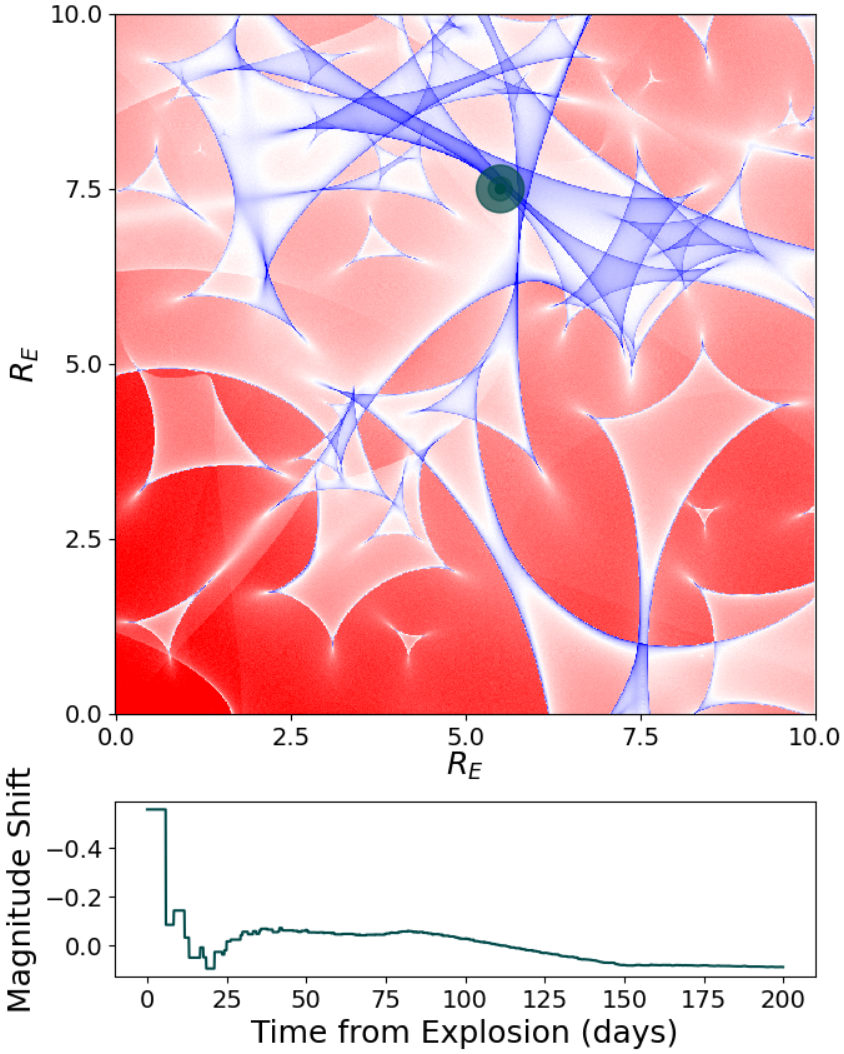
7.1 Microlensing Analysis

Simulate and fit for microlensing.

This notebook gives examples on creating microlensing for simulations, including microlensing in light curves, and fitting for a microlensing uncertainty.

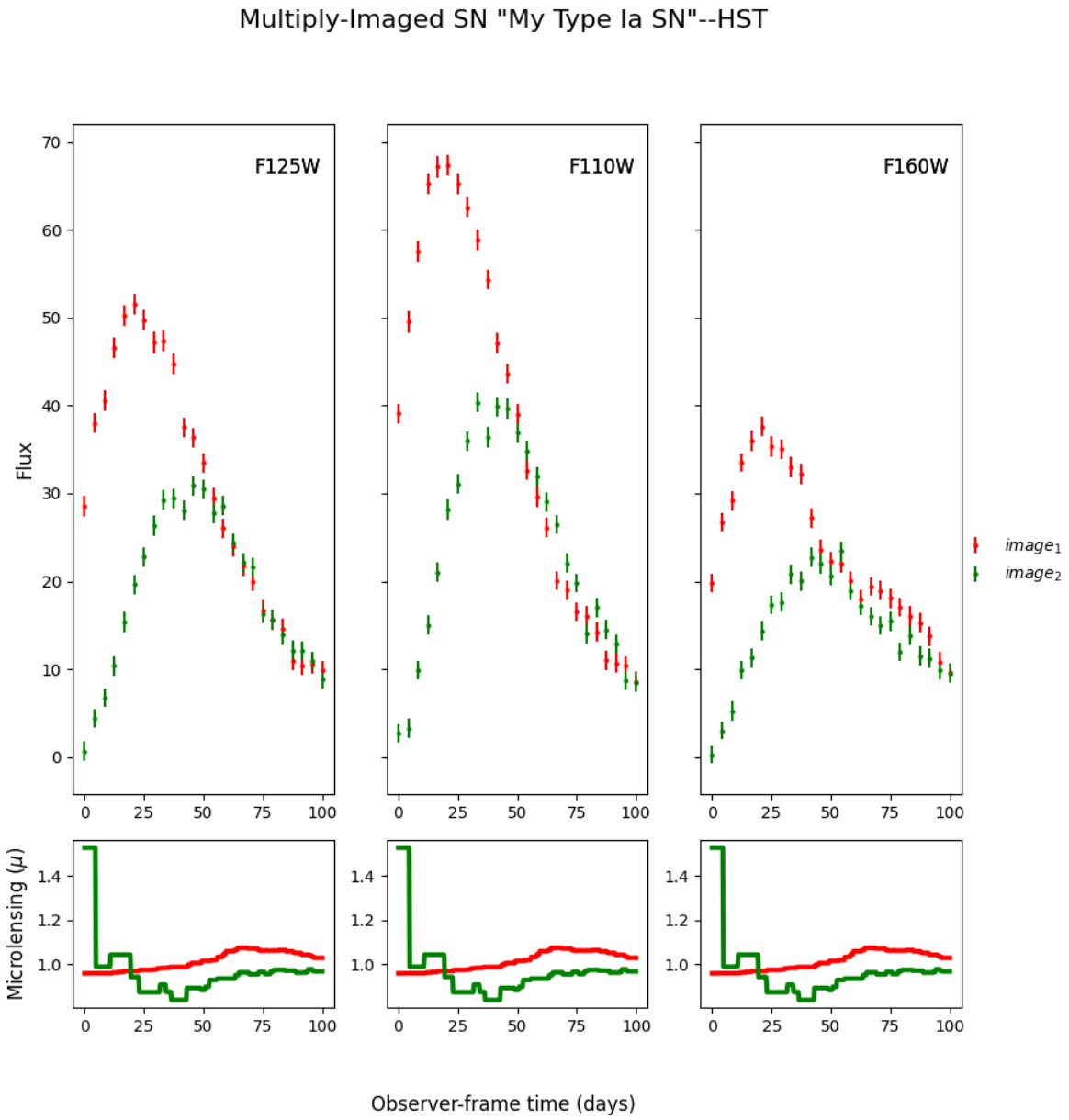
7.1.1 Run this notebook with Google Colab.

```
import sntd
import numpy as np
from sklearn.gaussian_process.kernels import RBF
np.random.seed(3)

myML=sntd.realizeMicro(nray=100,kappas=1,kappac=.3,gamma=.4)
time,dmag=sntd.microcaustic_field_to_curve(field=myML,time=np.arange(0,200,.1),zl=.5,
↪zs=1.5,plot=True,loc=[550,750])
```



Including Microlensing in Simulations Now we can take the simulated microcaustic and use it to include microlensing in a multiply-imaged supernova simulation. See the *Simulate Supernovae* example for more simulation details.

```
myMISN = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.5,z_lens=.5, bands=['F110W','F125W','F160W'],
        zp=[26.8,26.5,26.2], cadence=4., epochs=25.,time_delays=[20., 40.],␣
↪magnifications=[12,8],ml_loc=[[600,800],[550,750]],
        objectName='My Type Ia SN',telescopename='HST', microlensing_type=
↪'AchromaticMicrolensing',microlensing_params=myML)
myMISN.plot_object(showMicro=True)
```



Out:

```
<Figure size 1000x1000 with 6 Axes>
```

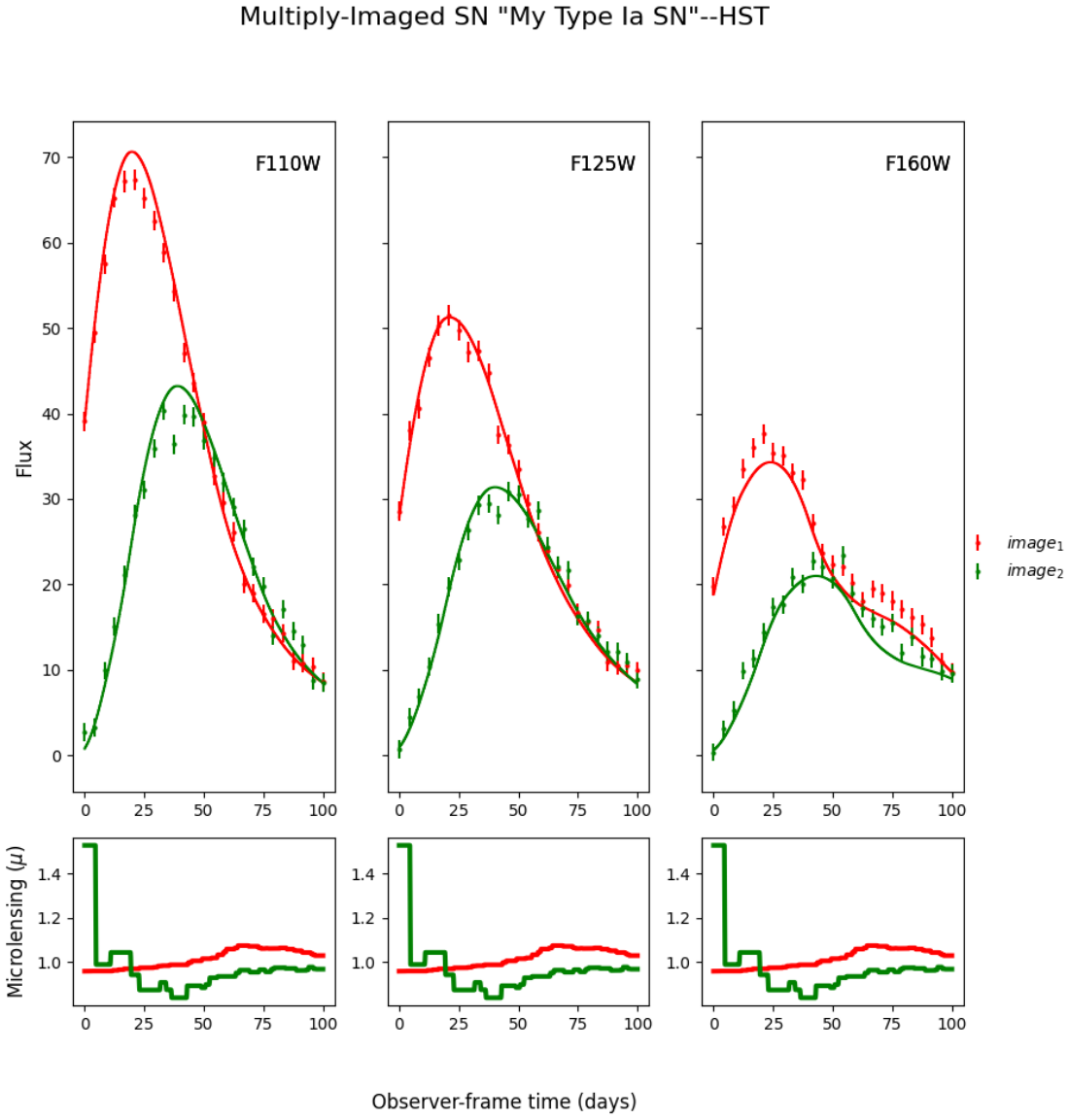
Measuring Addition Microlensing Uncertainty Now we can take the simulated light curve with microlensing and fit for an additional microlensing uncertainty term. See the *Measure Time Delays* example for fitting details. We start by assuming the correct shape/color parameters.

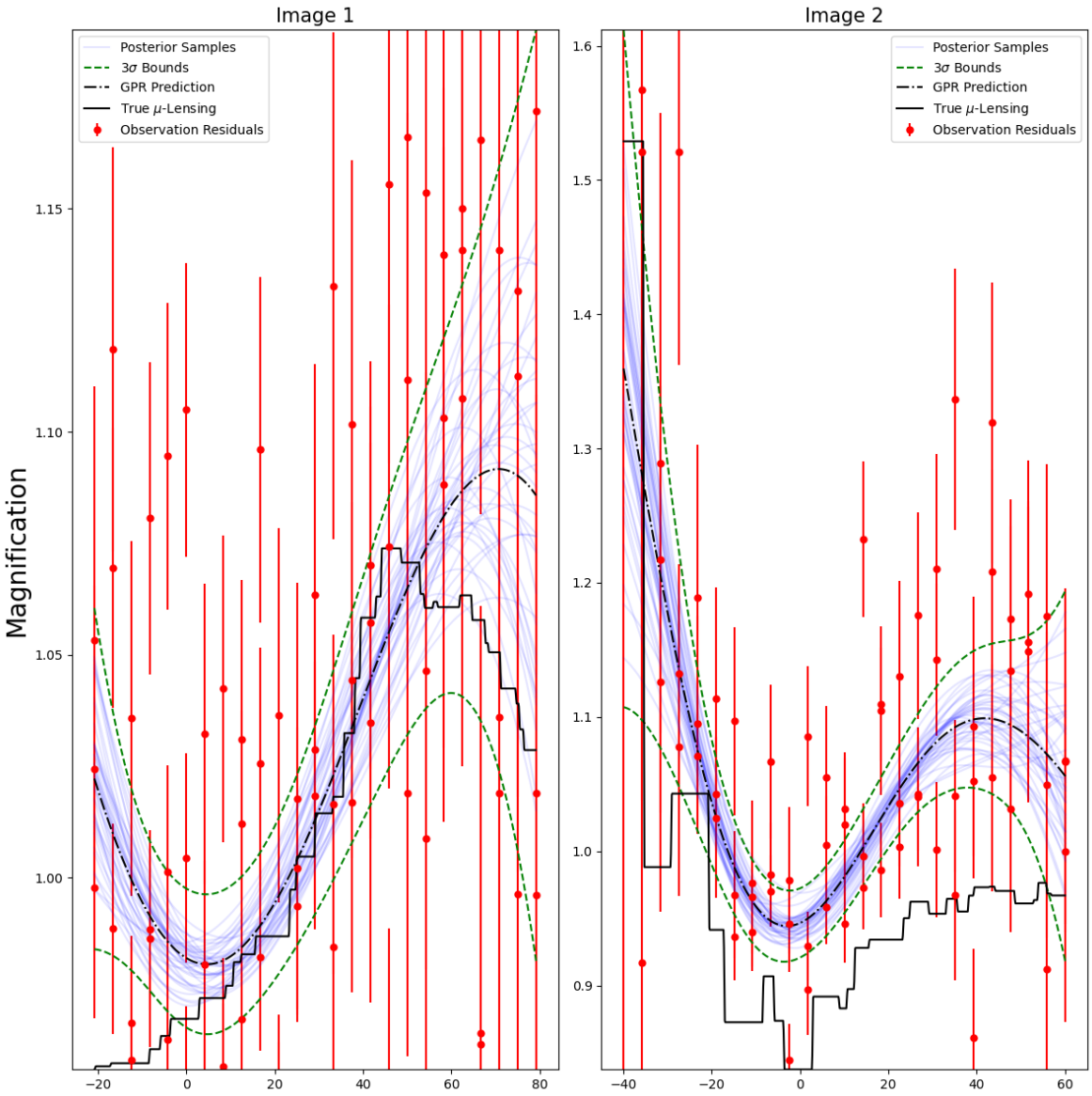
```
fitCurves = sntd.fit_data(myMISN, snType='Ia', models='salt2-extended', bands=['F110W
↪','F125W', 'F160W'],
                                  params=['x0', 't0'],
                                  constants={'z': 1.5,'x1':myMISN.images['image_1'].
↪simMeta['x1'],'c':myMISN.images['image_1'].simMeta['c']},
                                  bounds={'t0': (-40, 40),'c': (-1, 1), 'x1': (-2, 2),
↪ },
```

(continues on next page)

(continued from previous page)

```
                                    method='parallel', microlensing='achromatic',
                                    nMicroSamples=40, npoints=100, minsnr=5,
↪kernel=RBF(1.,(.0001,1000)))
print('Time Delays:',fitCurves.parallel.time_delays)
fitCurves.plot_object(showFit=True,showMicro=True)
for image in fitCurves.images.keys():
    print(image,'Microlensing Uncertainty:',fitCurves.images[image].param_quantiles[
↪'micro'],' Days')

fitCurves.plot_microlensing_fit(show_all_samples=True)
```





Out:

```
Time Delays: {'image_1': 0, 'image_2': 19.10361601481741}
image_1 Microlensing Uncertainty: 0.010268162214037983  Days
image_2 Microlensing Uncertainty: 0.028788219176668055  Days

(<Figure size 1200x1200 with 2 Axes>, [])
```

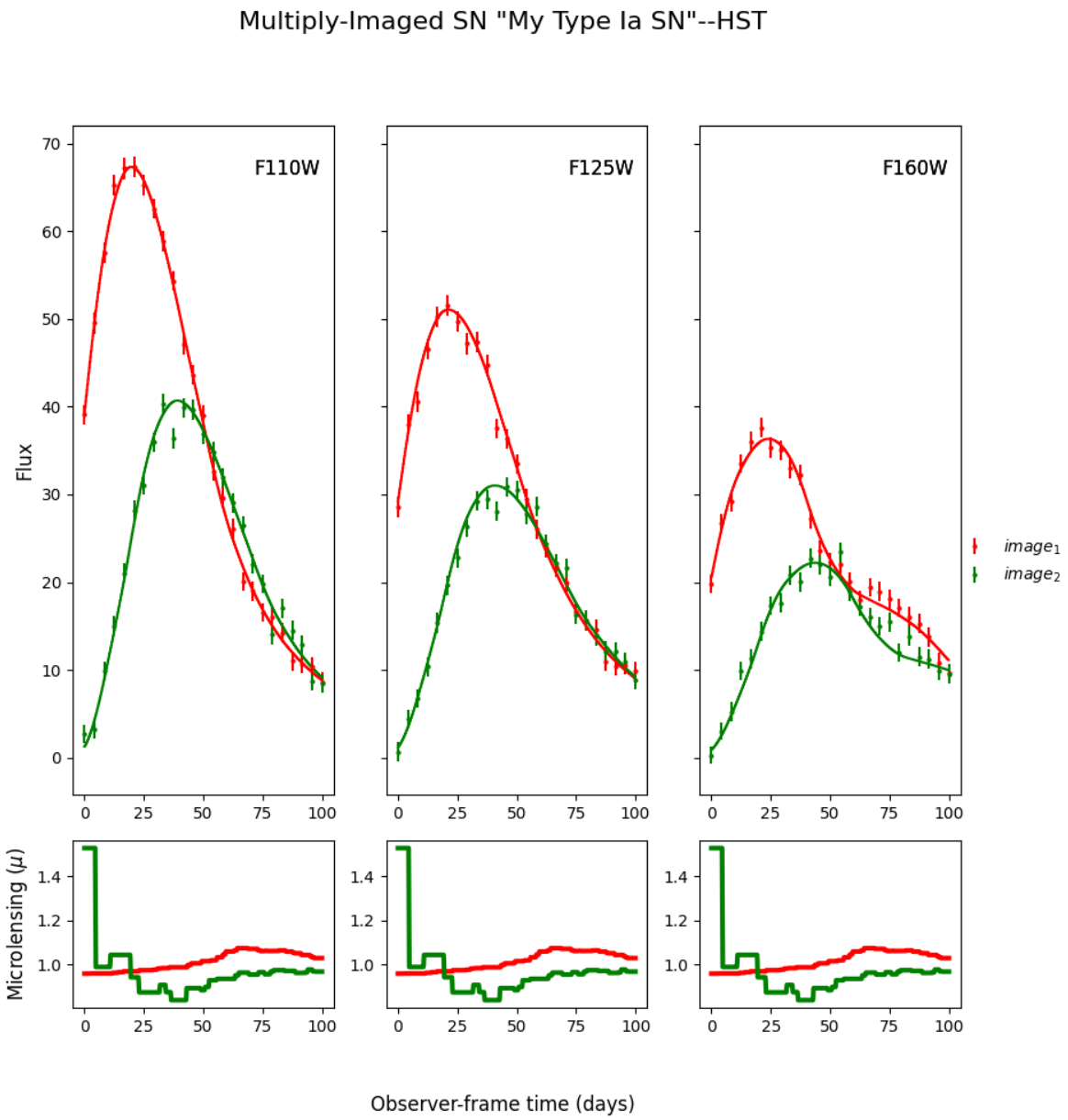
We see that this extra uncertainty is quite small here, and indeed when fitting for x1/c as well, the time delay measurement is very close to the true value of 20 days.

```
fitCurves = sntd.fit_data(myMISN, snType='Ia', models='salt2-extended', bands=['F110W
↪','F125W', 'F160W'],
                                  params=['x0', 't0','x1','c'],
                                  constants={'z': 1.5},
                                  bounds={'t0': (-40, 40),'c': (-1, 1), 'x1': (-2, 2)}
↪,
```

(continues on next page)

(continued from previous page)

```
                                    method='parallel', microlensing=None,
                                    npoints=100, minsnr=5)
print('Time Delays:',fitCurves.parallel.time_delays)
fitCurves.plot_object(showFit=True,showMicro=True)
```



Out:

```
Time Delays: {'image_1': 0, 'image_2': 19.55887430479142}

<Figure size 1000x1000 with 6 Axes>
```

Total running time of the script: (4 minutes 29.920 seconds)

7.2 Simulate Supernovae

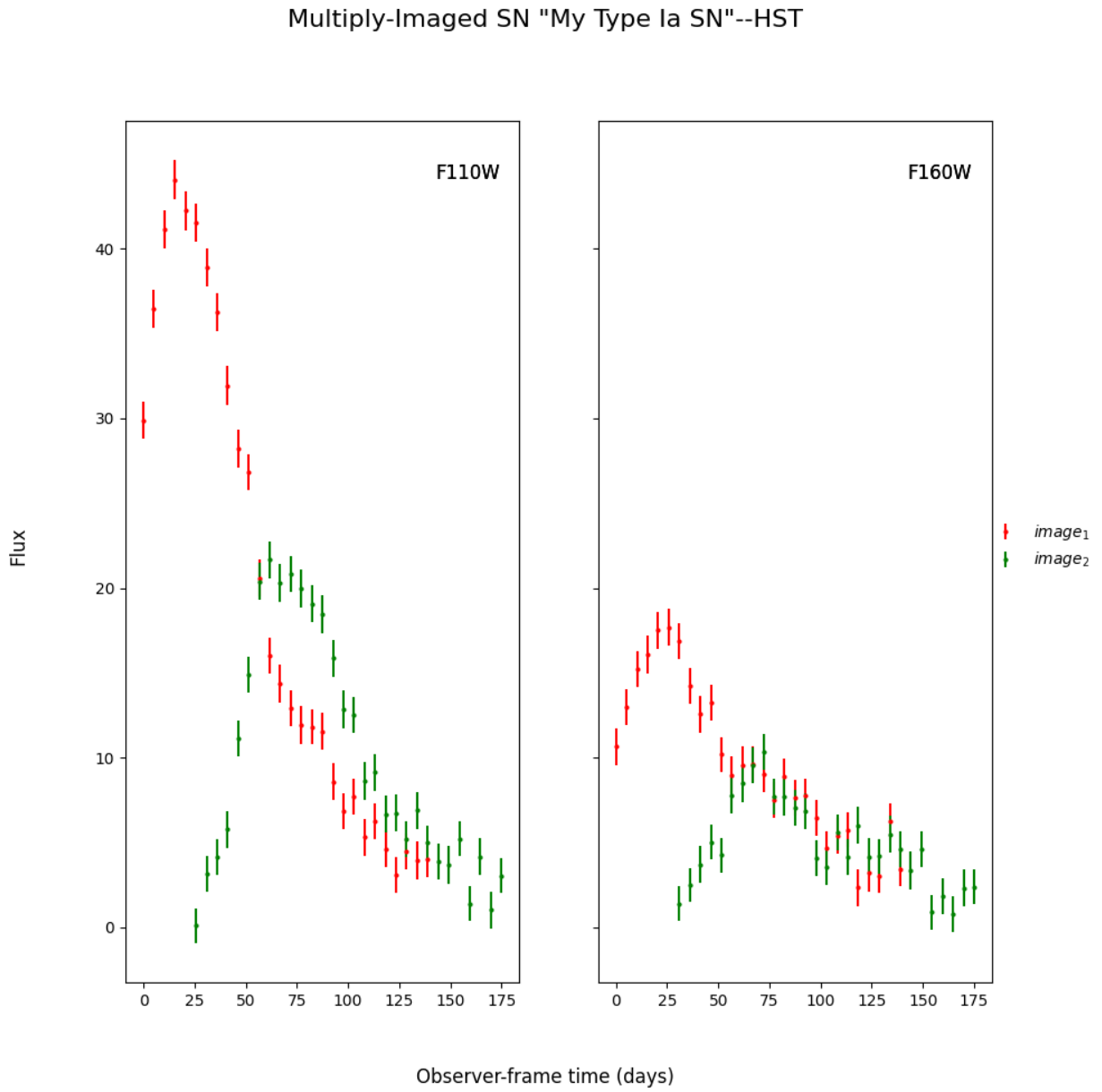
Simulating a multiply-imaged supernova.

Create a simulated multiply-imaged supernova that we can then fit, with no microlensing included in the simulation. Note that your final printed information will be different, as this is a randomly generated supernova. The function being used in these examples is *createMultiplyImagedSN()* .

7.2.1 Run this notebook with Google Colab.

No Microlensing

```
import sntd
import numpy as np

myMISN = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.4,z_lens=.53, bands=['F110W','F160W'],
             zp=[26.8,26.2], cadence=5., epochs=35.,time_delays=[20., 70.],␣
↪magnifications=[10,5],
 objectName='My Type Ia SN',telescopename='HST',av_host=False)
print(myMISN)
myMISN.plot_object()
```



Out:

```
Telescope: HST
Object: My Type Ia SN
Number of bands: 2

------------------
Image: image_1:
Bands: ['F110W', 'F160W']
Date Range: 0.00000->138.97059
Number of points: 56

Metadata:
   z:1.4
   t0:20.0
```

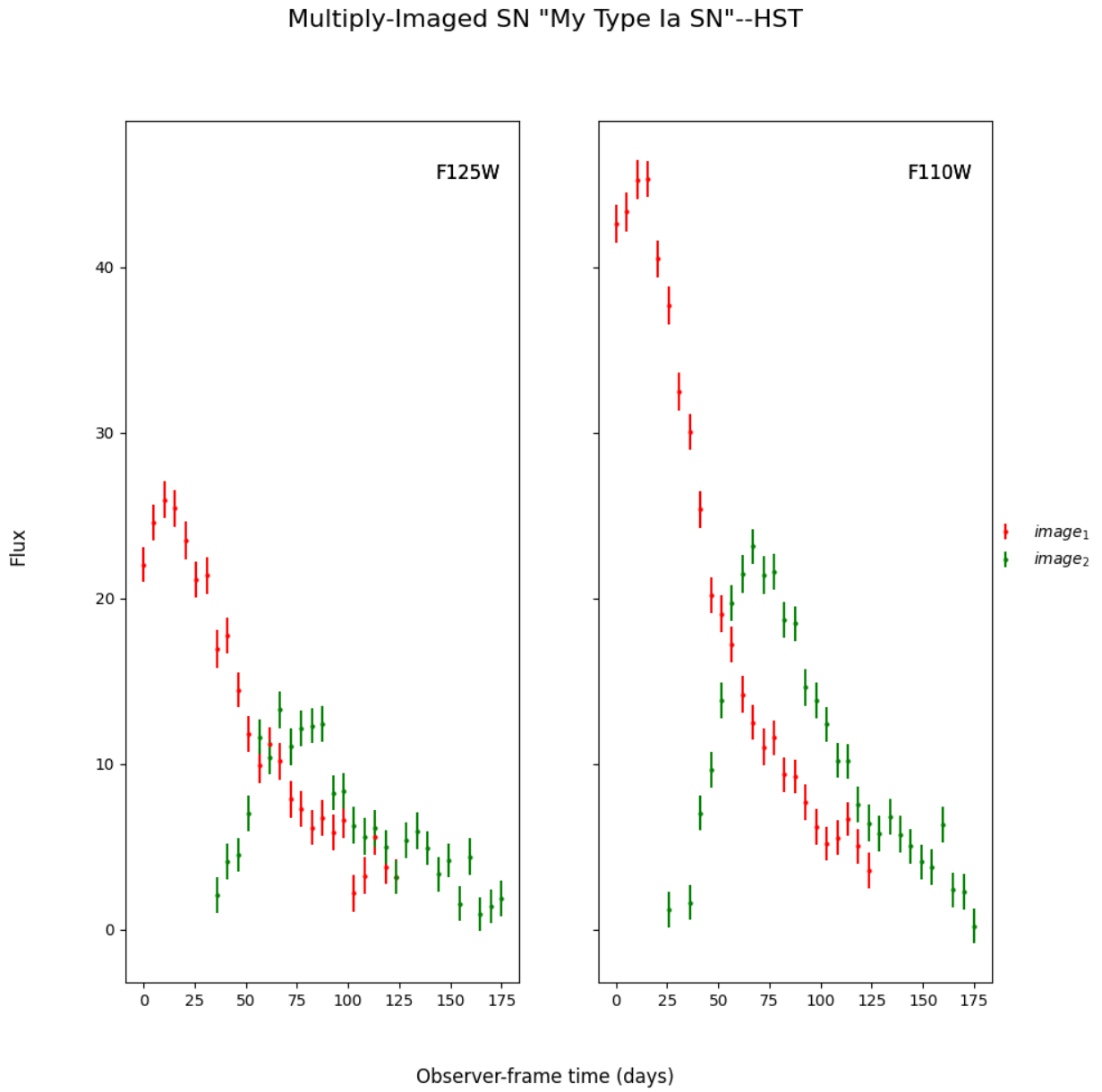
(continues on next page)

(continued from previous page)

```
   x0:5.9554678275562e-06
   x1:1.5139356714340149
   c:-0.17709989559397843
   sourcez:1.4
   hostebv:0
   lensebv:0
   lensz:0.53
   mu:10
   td:20.0
------------------
Image: image_2:
Bands: ['F110W', 'F160W']
Date Range: 25.73529->175.00000
Number of points: 59

Metadata:
   z:1.4
   t0:70.0
   x0:2.9777339137781e-06
   x1:1.5139356714340149
   c:-0.17709989559397843
   sourcez:1.4
   hostebv:0
   lensebv:0
   lensz:0.53
   mu:5
   td:70.0
------------------

<Figure size 1000x1000 with 2 Axes>
```

Specify the distributions you want to use for any model parameter by providing a function that returns the parameter in any way you want.

```
def x1_func():
    return(np.random.normal(1,.5))
def c_func():
    return(np.random.normal(-.05,.02))
param_funcs={'x1':x1_func,'c':c_func}
myMISN2 = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.33,z_lens=.53, bands=['F110W','F125W'],
              zp=[26.8,26.2], cadence=5., epochs=35.,time_delays=[10., 70.],␣
↪magnifications=[7,3.5],
              objectName='My Type Ia SN',telescopename='HST',sn_params=param_funcs)
print(myMISN2)
myMISN2.plot_object()
```



Out:

```
Telescope: HST
Object: My Type Ia SN
Number of bands: 2

------------------
Image: image_1:
Bands: ['F125W', 'F110W']
Date Range: 0.00000->123.52941
Number of points: 50

Metadata:
   z:1.33
   t0:10.0
```

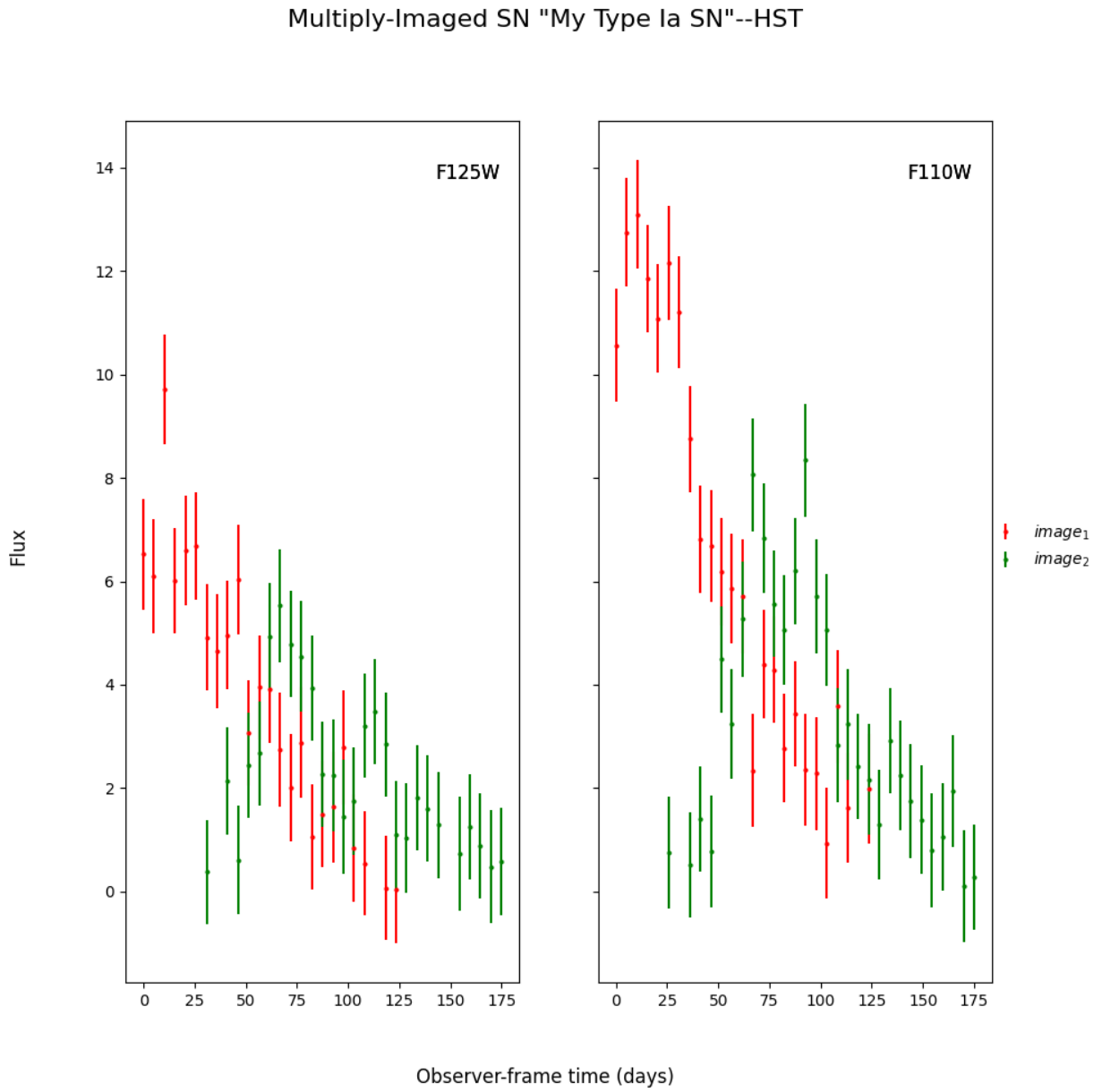
(continues on next page)

(continued from previous page)

```
   x0:8.380581686942374e-06
   x1:1.1026083422743245
   c:-0.03916533959272924
   sourcez:1.33
   hostebv:0.0967741935483871
   lensebv:0
   lensz:0.53
   mu:7
   td:10.0
------------------
Image: image_2:
Bands: ['F125W', 'F110W']
Date Range: 25.73529->175.00000
Number of points: 57

Metadata:
   z:1.33
   t0:70.0
   x0:4.190290843471187e-06
   x1:1.1026083422743245
   c:-0.03916533959272924
   sourcez:1.33
   hostebv:0.0967741935483871
   lensebv:0
   lensz:0.53
   mu:3.5
   td:70.0
------------------

<Figure size 1000x1000 with 2 Axes>
```

Specify the distributions you want to use for dust parameters by providing a function that returns the parameter in any way you want.

```
def hostav_func():
    return(np.random.normal(.5,.1))
def lensav_func():
    return(np.random.normal(.7,.2))
param_funcs={'host':hostav_func,'lens':lensav_func}
myMISN3 = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.33,z_lens=.53, bands=['F110W','F125W'],
              zp=[26.8,26.2], cadence=5., epochs=35.,time_delays=[10., 70.],␣
↪magnifications=[7,3.5],
              objectName='My Type Ia SN',telescopename='HST',av_dists=param_funcs)
print(myMISN3)
myMISN3.plot_object()
```



Out:

```
Telescope: HST
Object: My Type Ia SN
Number of bands: 2

------------------
Image: image_1:
Bands: ['F125W', 'F110W']
Date Range: 0.00000->123.52941
Number of points: 48

Metadata:
   z:1.33
   t0:10.0
```

(continues on next page)

(continued from previous page)

```
   x0:4.354187651106564e-06
   x1:1.6358939552410352
   c:-0.08458570217749427
   sourcez:1.33
   hostebv:0.15535413969553127
   lensebv:0.18363144965770012
   lensz:0.53
   mu:7
   td:10.0
------------------
Image: image_2:
Bands: ['F125W', 'F110W']
Date Range: 25.73529->175.00000
Number of points: 56

Metadata:
   z:1.33
   t0:70.0
   x0:2.177093825553282e-06
   x1:1.6358939552410352
   c:-0.08458570217749427
   sourcez:1.33
   hostebv:0.15535413969553127
   lensebv:0.18363144965770012
   lensz:0.53
   mu:3.5
   td:70.0
------------------

<Figure size 1000x1000 with 2 Axes>
```

Total running time of the script: (0 minutes 0.569 seconds)

7.3 Measure Time Delays

A series of examples demonstrating various fitting options/features with SNTD.

There are 3 methods built into SNTD to measure time delays (parallel, series, color). They are accessed by the same function: *fit_data()* . Here myMISN was generated in the *Simulate Supernovae* part of the documentation, using the *createMultiplyImagedSN()* function. The true delay for all of these fits is 50 days. You can batch process (with sbatch or multiprocessing) using any or all of these methods as well (see *Batch Processing Time Delay Measurements*)

7.3.1 Run this notebook with Google Colab.

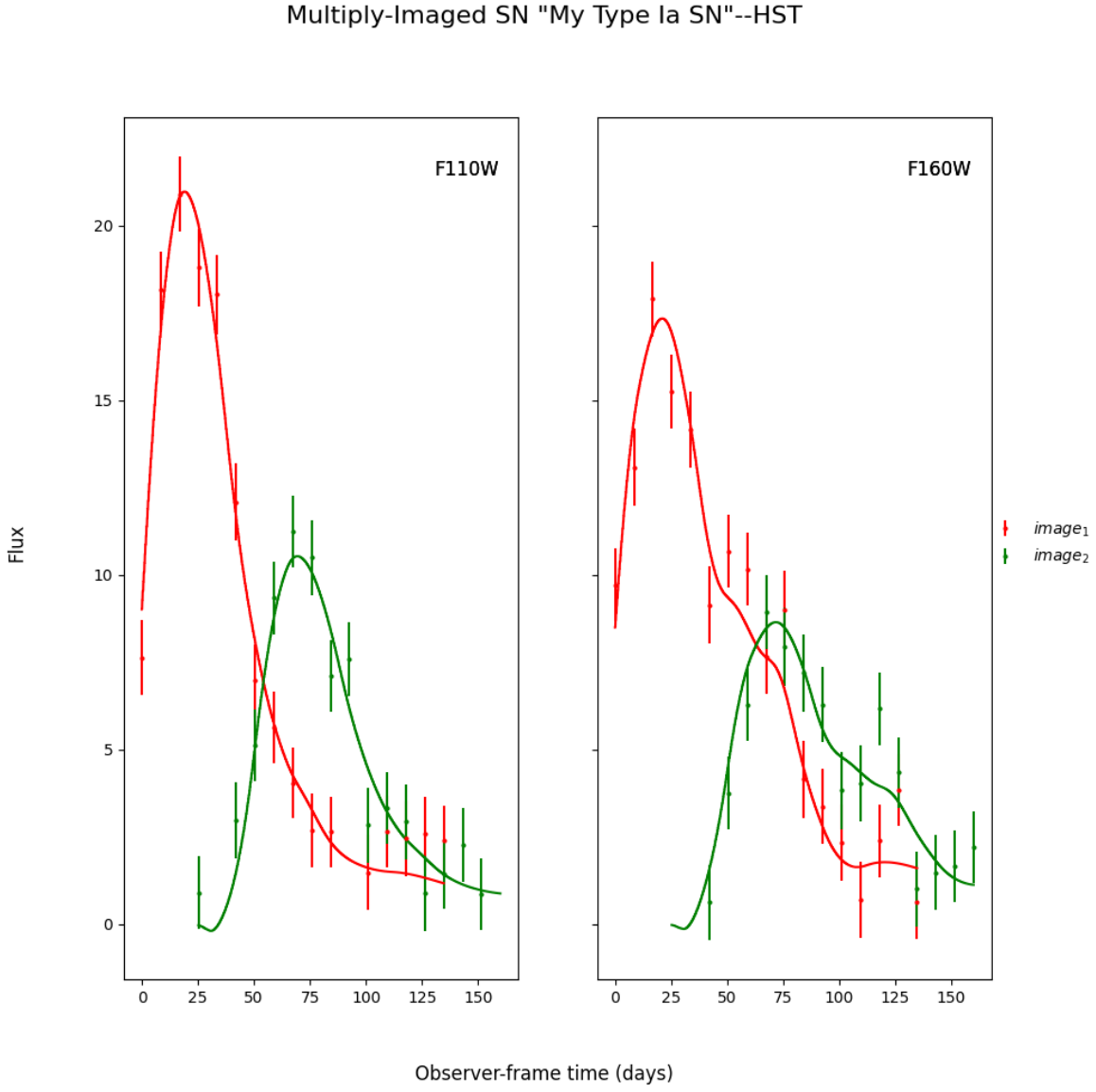
Parallel:

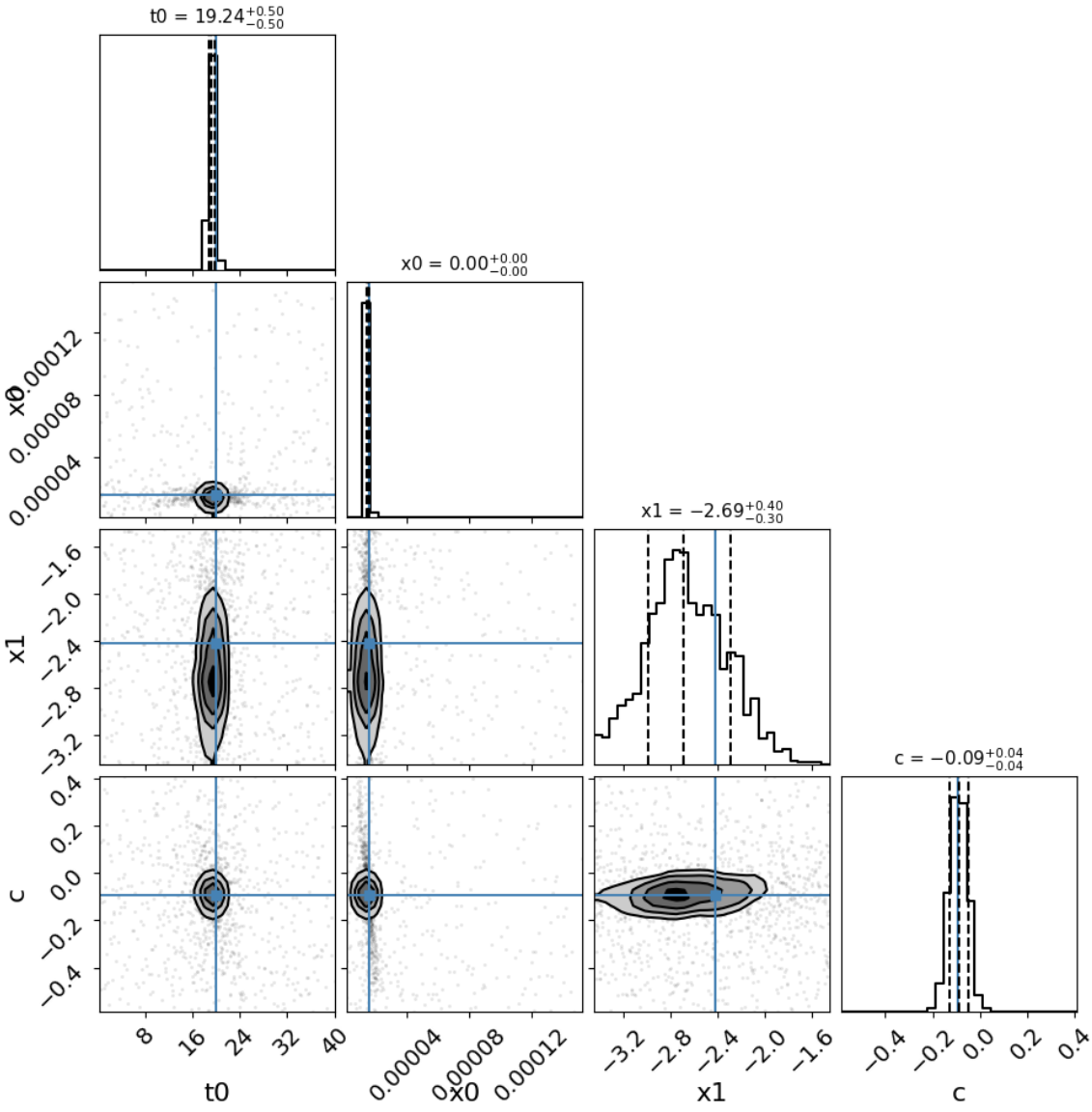
```
import sntd

myMISN=sntd.load_example_misn()

fitCurves=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=['F110W',
↪'F160W'],
```

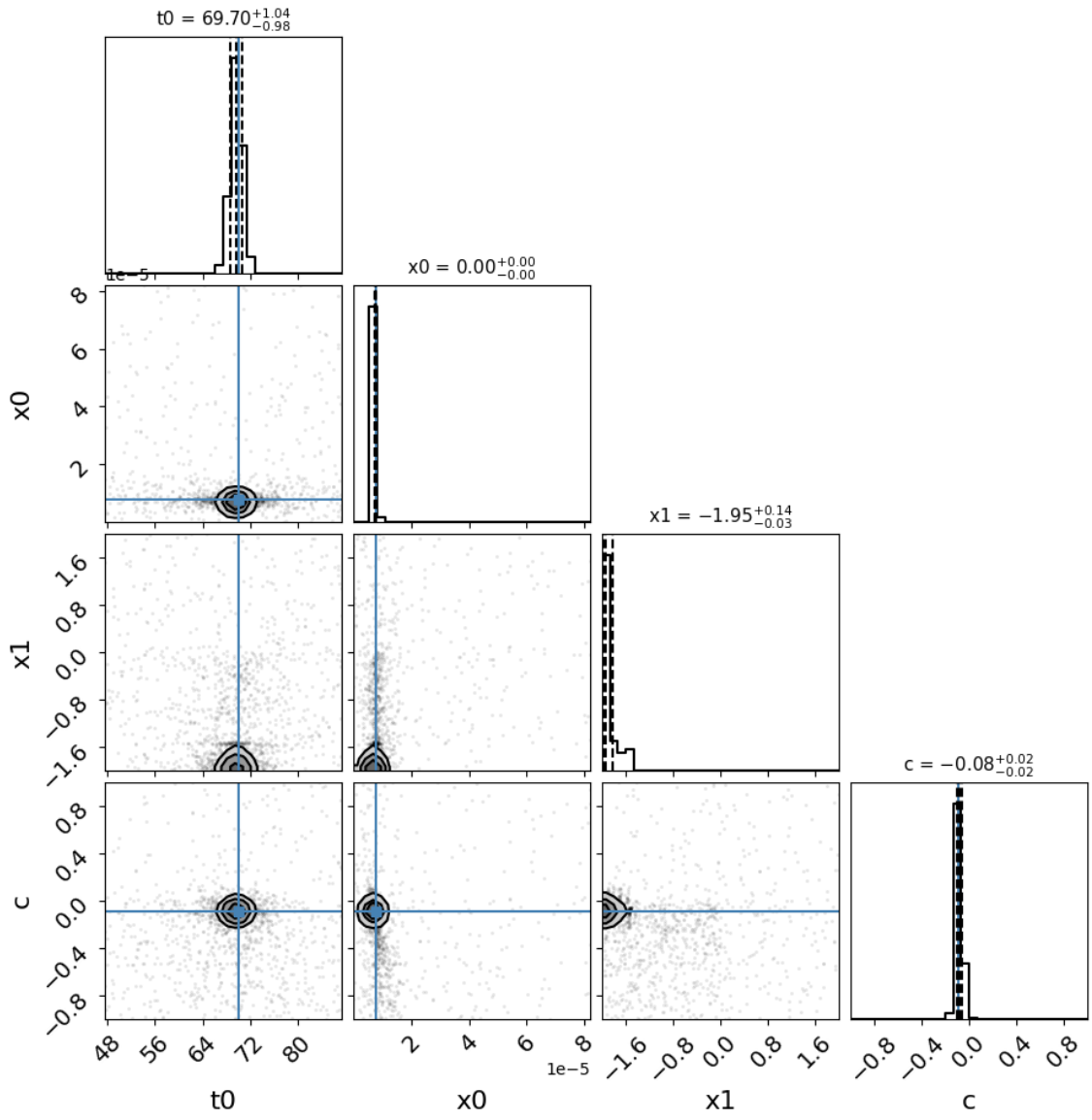
(continues on next page)

(continued from previous page)

```
                params=['x0','t0','x1','c'],constants={'z':1.4},refImage='image_1',
↪cut_time=[-30,40],
                bounds={'t0':(-20,20),'x1':(-2,2),'c':(-1,1),'mu':(.5,2)},fitOrder=[
↪'image_1','image_2'],
                method='parallel',microlensing=None,modelcov=False,npoints=100)
print(fitCurves.parallel.time_delays)
print(fitCurves.parallel.time_delay_errors)
print(fitCurves.parallel.magnifications)
print(fitCurves.parallel.magnification_errors)
fitCurves.plot_object(showFit=True,method='parallel')
fitCurves.plot_fit(method='parallel',par_image='image_1')
fitCurves.plot_fit(method='parallel',par_image='image_2')
```




t0 = 19.24+0.50−0.50
x0 = 0.00+0.00−0.00
x1 = −2.69+0.40−0.30
c = −0.09+0.04−0.04
t0
x0
x1
c



Out:

```
{'image_1': 0, 'image_2': 50.442299642467276}
{'image_1': array([0, 0]), 'image_2': array([-0.95826261,  0.98199374])}
{'image_1': 1, 'image_2': 0.4998899592804862}
{'image_1': array([0, 0]), 'image_2': array([-0.02284181,  0.02404375])}

<Figure size 970x970 with 16 Axes>
```

Note that the bounds for the 't0' parameter are not absolute, the actual peak time will be estimated (unless t0_guess is defined) and the defined bounds will be added to this value. Similarly for amplitude, where bounds are multiplicative

Other methods are called in a similar fashion, with a couple of extra arguments:

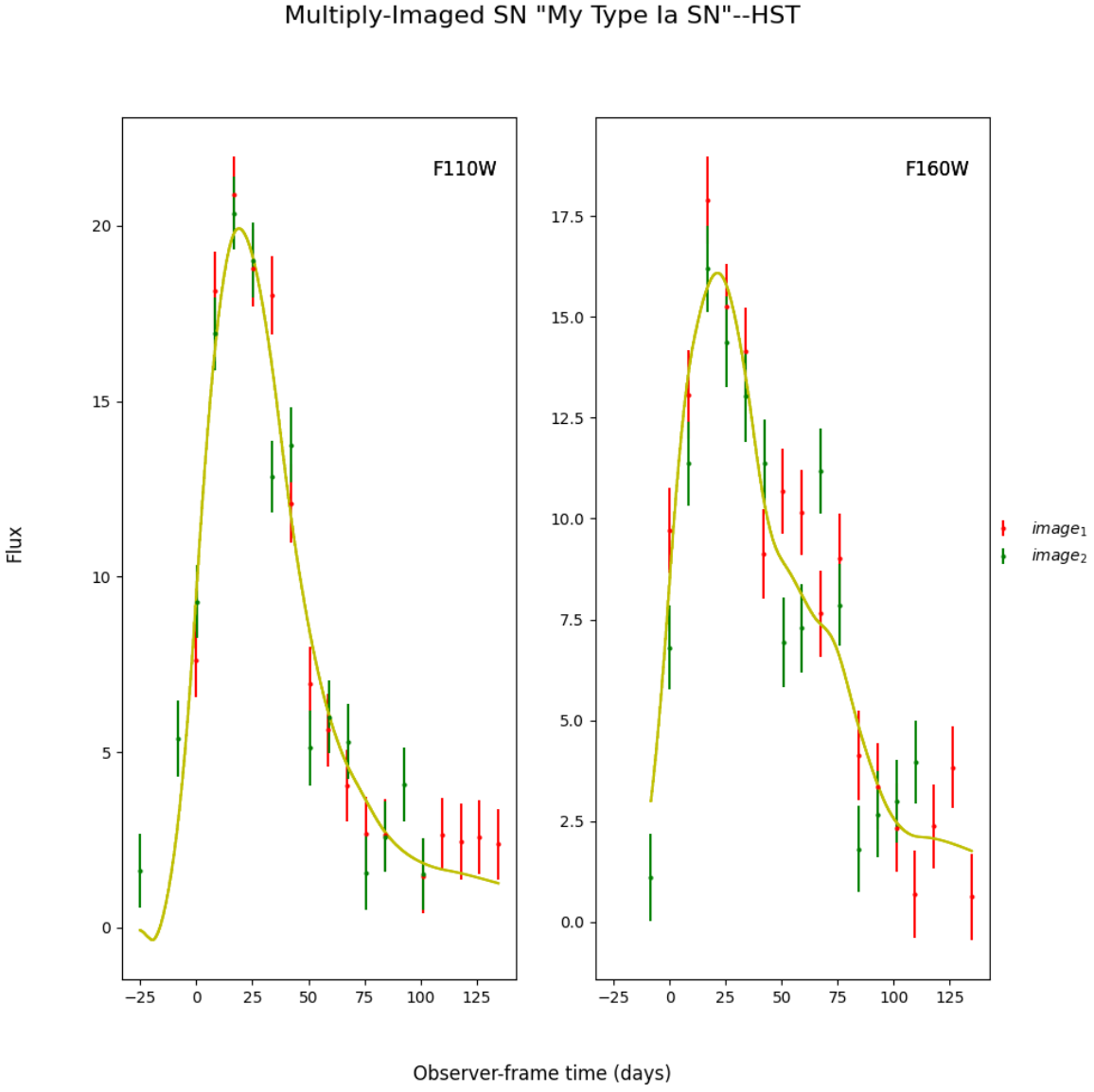
Series:

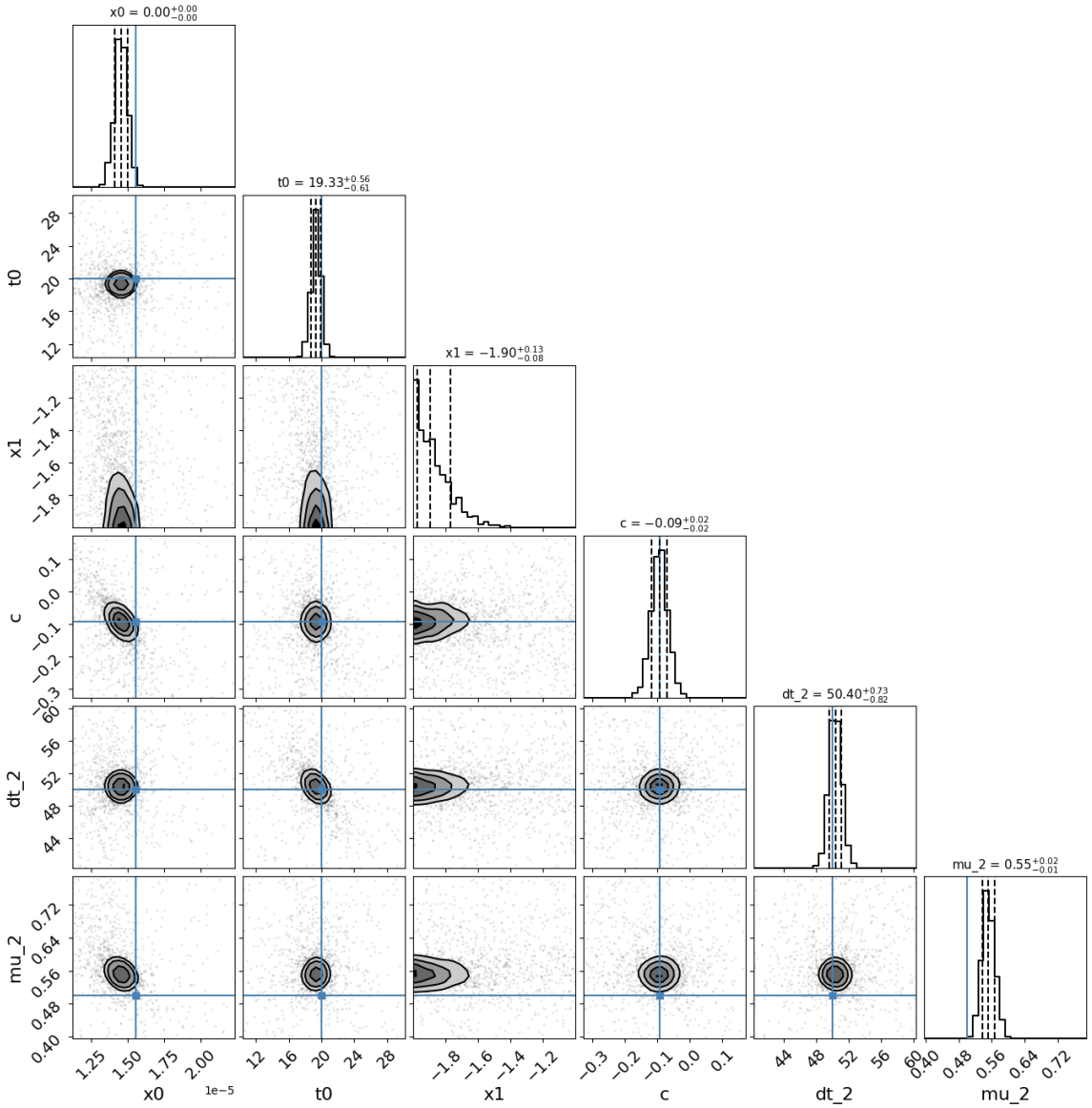
```
fitCurves=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=['F110W',
↪'F160W'],
```

(continues on next page)

(continued from previous page)

```
        params=['x0','t0','x1','c'],constants={'z':1.4},refImage='image_1',cut_time=[-
↪30,40],
        bounds={'t0':(-20,20),'td':(-20,20),'mu':(.5,2),'x1':(-2,2),'c':(-.5,.5)},
        method='series',npoints=100)


print(fitCurves.series.time_delays)
print(fitCurves.series.time_delay_errors)
print(fitCurves.series.magnifications)
print(fitCurves.series.magnification_errors)
fitCurves.plot_object(showFit=True,method='series')
fitCurves.plot_fit(method='series')
```





Out:

```
['x0', 't0', 'x1', 'c']
['x0', 't0', 'x1', 'c']
{'image_1': 0, 'image_2': 50.39797715147582}
{'image_1': array([0, 0]), 'image_2': array([-0.82216382,  0.72853045])}
{'image_1': 1, 'image_2': 0.5515560895917695}
{'image_1': 1, 'image_2': array([-0.01397073,  0.01642504])}

<Figure size 1390x1390 with 36 Axes>
```

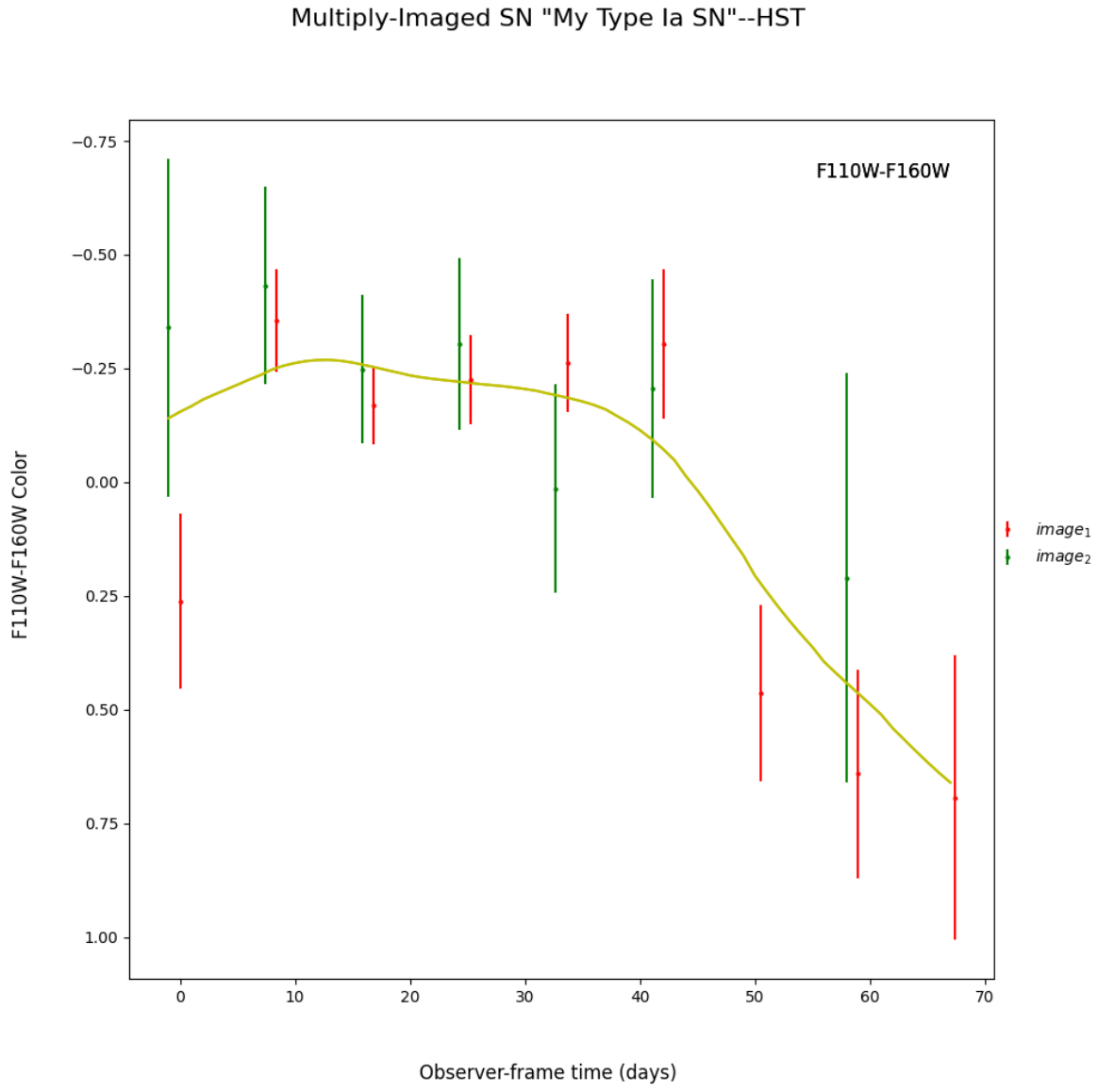
Color: By default, this will attempt to fit every combination of colors possible from the bands present in the data. You can define specific colors using the "fit_colors" argument.

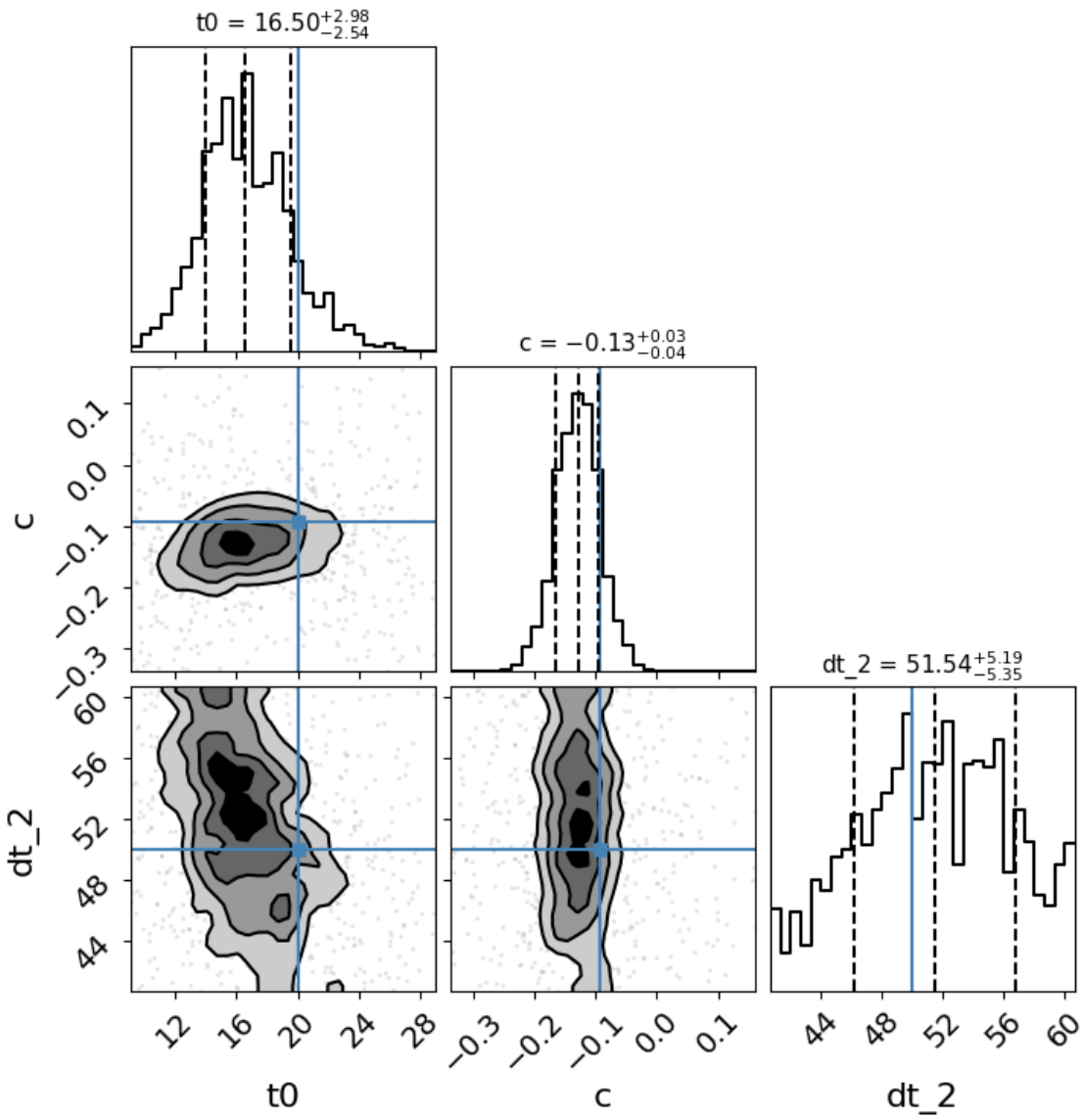
```
fitCurves=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=['F110W',
↪'F160W'],
                        params=['t0','c'],constants={'z':1.4,'x1':fitCurves.images['image_
↪1'].fits.model.get('x1')},refImage='image_1',
```

(continues on next page)

(continued from previous page)

```
                    color_param_ignore=['x1'],bounds={'t0':(-20,20),'td':(-20,20),'mu
↪':(.5,2),'c':(-.5,.5)},cut_time=[-30,40],
                    method='color',microlensing=None,modelcov=False,npoints=200,
↪maxiter=None,minsnr=3)

print(fitCurves.color.time_delays)
print(fitCurves.color.time_delay_errors)
fitCurves.plot_object(showFit=True,method='color')
fitCurves.plot_fit(method='color')
```





Out:

```
{'image_1': 0, 'image_2': 51.530224748944036}
{'image_1': array([0, 0]), 'image_2': array([-5.35065195,  5.19171365])}

<Figure size 760x760 with 9 Axes>
```

You can include your fit from the parallel method as a prior on light curve and time delay parameters in the series/color methods with the "fit_prior" command:

```
fitCurves_parallel=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=[
↪'F110W','F160W'],
                    params=['x0','t0','x1','c'],constants={'z':1.4},refImage='image_1
↪',
                    bounds={'t0':(-20,20),'x1':(-3,3),'c':(-.5,.5),'mu':(.5,2)},
↪fitOrder=['image_1','image_2'],cut_time=[-30,40],
                        method='parallel',microlensing=None,modelcov=False,
↪npoints=100,maxiter=None)
```

(continues on next page)

(continued from previous page)

```
fitCurves_color=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=[
↪'F110W','F160W'],cut_time=[-50,30],
                    params=['t0','c'],constants={'z':1.4,'x1':fitCurves.images['image_
↪1'].fits.model.get('x1')},refImage='image_1',
                    bounds={'t0':(-20,20),'td':(-20,20),'mu':(.5,2),'c':(-.5,.5)},fit_
↪prior=fitCurves_parallel,
                    method='color',microlensing=None,modelcov=False,npoints=200,
↪maxiter=None,minsnr=3)

print(fitCurves_parallel.parallel.time_delays)
print(fitCurves_parallel.parallel.time_delay_errors)
print(fitCurves_color.color.time_delays)
print(fitCurves_color.color.time_delay_errors)
```

Out:

```
{'image_1': 0, 'image_2': 50.096483213906325}
{'image_1': array([0, 0]), 'image_2': array([-0.99028586,  1.26368148])}
{'image_1': 0, 'image_2': 50.31747986433648}
{'image_1': array([0, 0]), 'image_2': array([-2.01481246,  2.28440021])}
```

Fitting Using Extra Propagation Effects

You might also want to include other propagation effects in your fitting model, and fit relevant parameters. This can be done by simply adding effects to an SNCosmo model, in the same way as if you were fitting a single SN with SNCosmo. First we can add some extreme dust in the source and lens frames (your final simulations may look slightly different as **c** is chosen randomly):

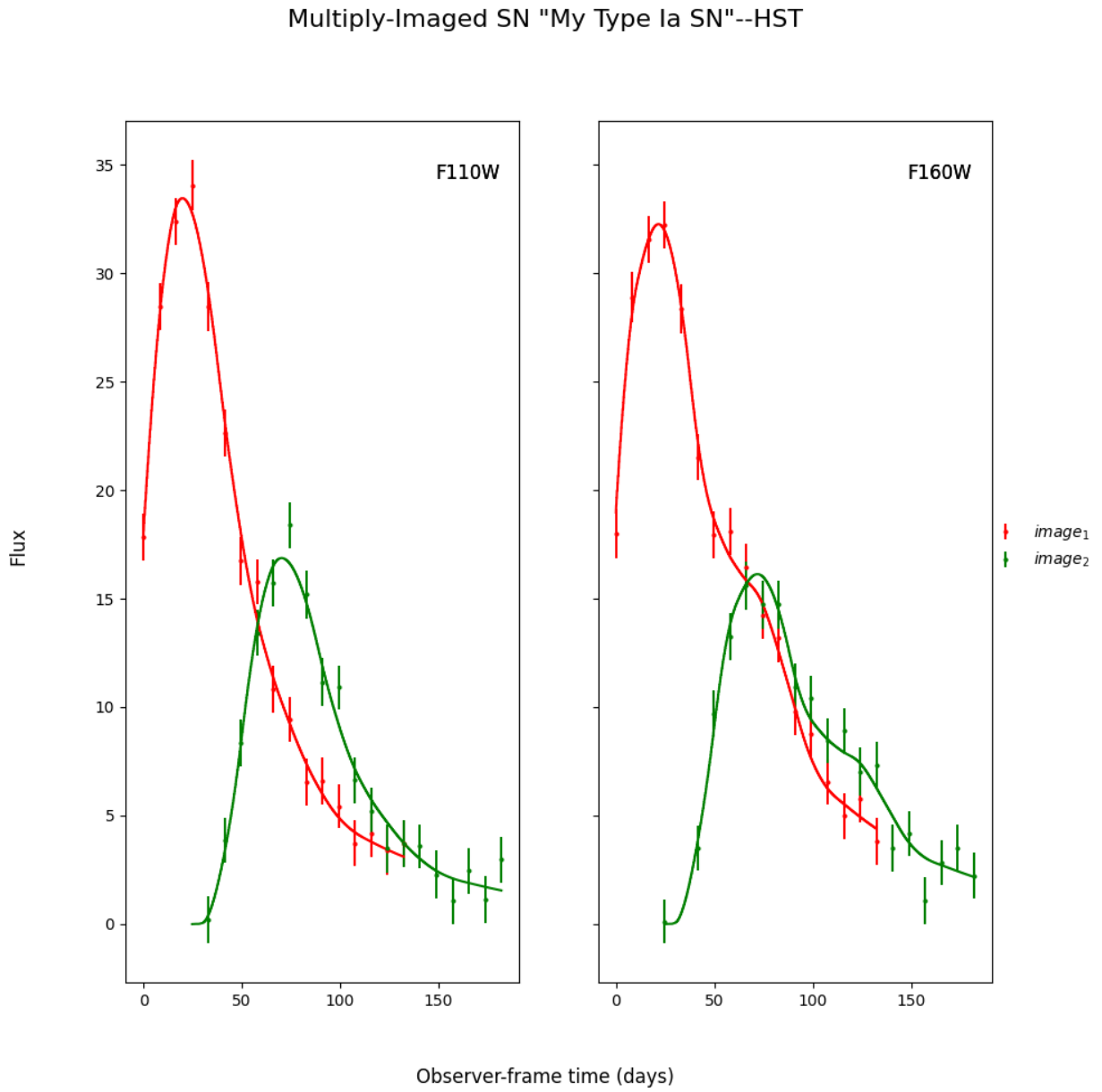
```
myMISN2 = sntd.createMultiplyImagedSN(sourcename='salt2-extended', snType='Ia',␣
↪redshift=1.4,z_lens=.53, bands=['F110W','F160W'],
              zp=[26.9,26.2], cadence=8., epochs=30.,time_delays=[20., 70.],␣
↪magnifications=[20,10],
              objectName='My Type Ia SN',telescopename='HST',av_lens=1.5,
              av_host=1)
print('lensebv:',myMISN2.images['image_1'].simMeta['lensebv'],
     'hostebv:',myMISN2.images['image_1'].simMeta['hostebv'],
     'c:',myMISN2.images['image_1'].simMeta['c'])
```

Out:

```
lensebv: 0.48387096774193544 hostebv: 0.3225806451612903 c: 0.1377735187090485
```

Okay, now we can fit the MISN first without taking these effects into account:

```
fitCurves_dust=sntd.fit_data(myMISN2,snType='Ia', models='salt2-extended',bands=[
↪'F110W','F160W'],
                                                    params=['x0','x1','t0','c'],
↪npoints=200,
                                                    constants={'z':1.4},minsnr=1,cut_
↪time=[-30,40],
                                                    bounds={'t0':(-15,15),'x1':(-3,
↪3),'c':(-1,1)})
print(fitCurves_dust.parallel.time_delays)
print(fitCurves_dust.parallel.time_delay_errors)
print('c:',fitCurves_dust.images['image_1'].fits.model.get('c'))
fitCurves_dust.plot_object(showFit=True)
```



Out:

```
{'image_1': 0, 'image_2': 50.412287698579206}
{'image_1': array([0, 0]), 'image_2': array([-0.74964906,  0.72983875])}
c: 0.8117704098202272

<Figure size 1000x1000 with 2 Axes>
```

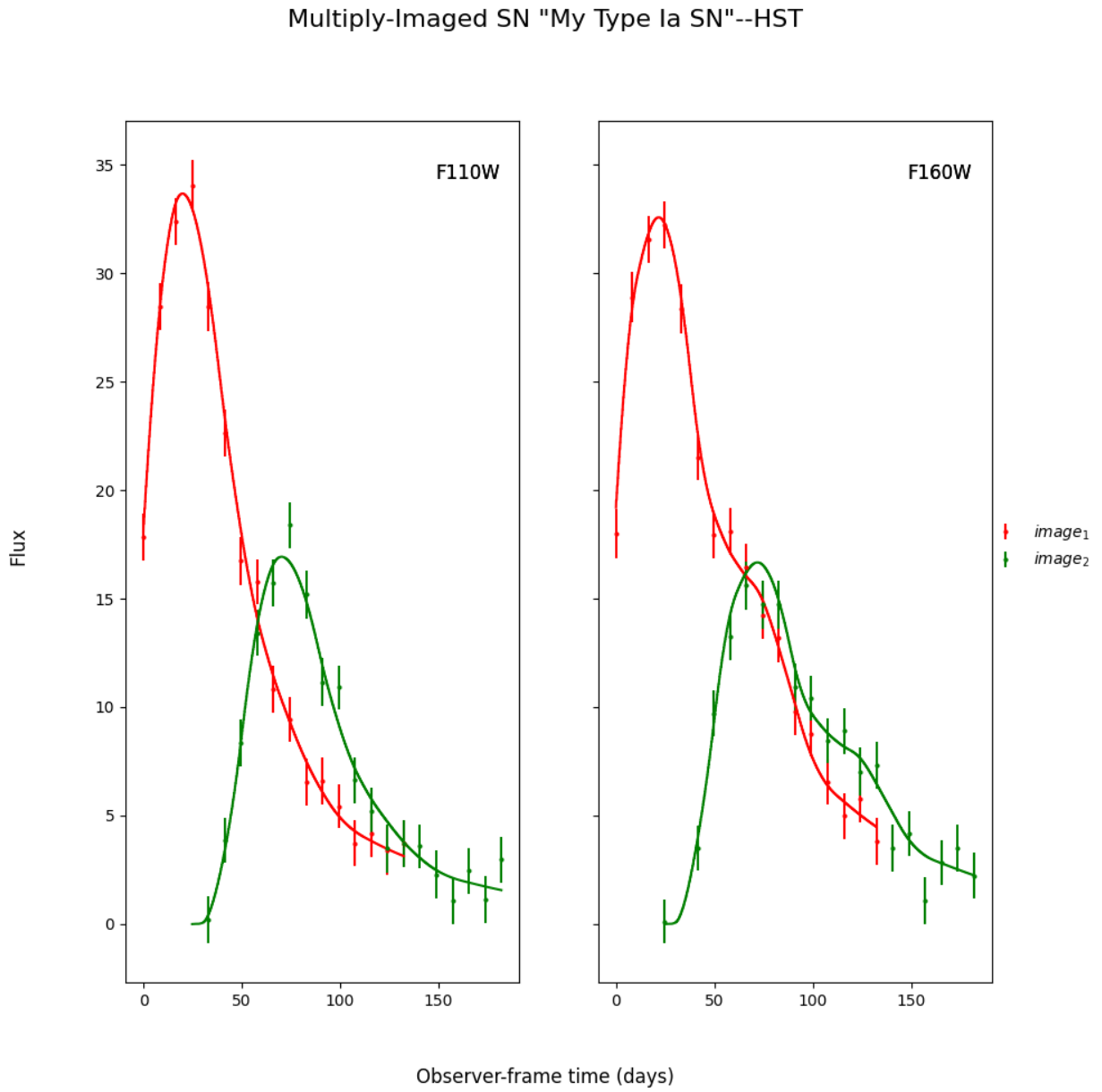
We can see that the fitter has done reasonably well, and the time delay is still accurate (True delay is 50 days). However, one issue is that the measured value for **c** is vastly different than the actual value as it attempts to compensate for extinction without a propagation effect. Now let's add in the propagation effects:

```
import sncosmo
dust = sncosmo.CCM89Dust()
```

(continues on next page)

(continued from previous page)

```
salt2_model=sncosmo.Model('salt2-extended',effects=[dust,dust],effect_names=['lens',
↪'host'],effect_frames=['free','rest'])
fitCurves_dust=sntd.fit_data(myMISN2,snType='Ia', models=salt2_model,bands=['F110W',
↪'F160W'],npoints=200,
                    params=['x0','x1','t0','c','lensebv','hostebv'],minsnr=1,cut_
↪time=[-30,40],
                    constants={'z':1.4,'lensr_v':3.1,'lensz':0.53,'hostr_v':3.1},
                    bounds={'t0':(-15,15),'x1':(-3,3),'c':(-.1,.1),'lensebv':(.2,1.),
↪'hostebv':(.2,1.)})

print(fitCurves_dust.parallel.time_delays)
print(fitCurves_dust.parallel.time_delay_errors)
print('c:',fitCurves_dust.images['image_1'].fits.model.get('c'),
      'lensebv:',fitCurves_dust.images['image_1'].fits.model.get('lensebv'),
      'hostebv:',fitCurves_dust.images['image_1'].fits.model.get('hostebv'))
fitCurves_dust.plot_object(showFit=True)
```



Out:

```
{'image_1': 0, 'image_2': 50.33052950846436}
{'image_1': array([0, 0]), 'image_2': array([-0.65493107,  0.68273663])}
c: -0.30174440537013747 lensebv: -0.6076719364812484 hostebv: 1.5424337700970974

<Figure size 1000x1000 with 2 Axes>
```

Now the measured value for **c** is much closer to reality, and the measured times of peak are somewhat more accurate.

Total running time of the script: (2 minutes 57.033 seconds)

7.4 Constrain Cosmology

Simulate cosmological constraints from a sample of lensed SN.

All of these cosmology tools are based on Coe & Moustakas 2009. and Dan Coe's Fisher matrix starter paper.

7.4.1 Run this notebook with Google Colab.

Creating a Survey

```
import sntd
import numpy as np
```

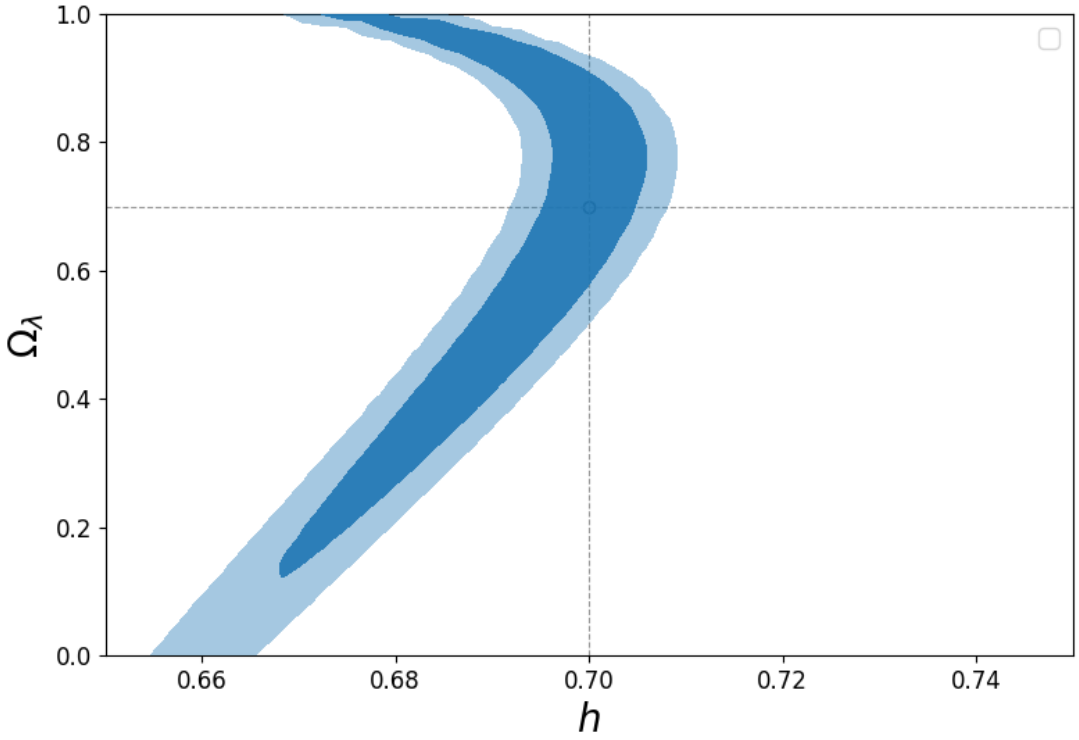
Start by defining your survey parameters. In this case we have a survey called "Test Survey" with 10 lenses with normally distributed lens and source redshifts, 5% lens model uncertainty and 2% time delay uncertainty.

```
np.random.seed(3)

my_survey=sntd.Survey(dTl=5,dTT=2,zl=np.random.normal(.5,.1,size=10),zs=np.random.
↪normal(1.6,.2,size=10),name='Test Survey')
```

Gridded Parameter Search

This will make a smooth contour plot for 2 parameters.

```
my_survey.survey_grid(vparam_names=['h','Ode0'],
                      bounds={'h':[.65,.75],'Ode0':[0,1]},npoints=50)

my_survey.plot_survey_contour(['h','Ode0'],math_labels=[r'$h$',r'$\Omega_\lambda$'],
↪confidence=[.68,.95],alphas=[.9,.4],show_legend=True)
```

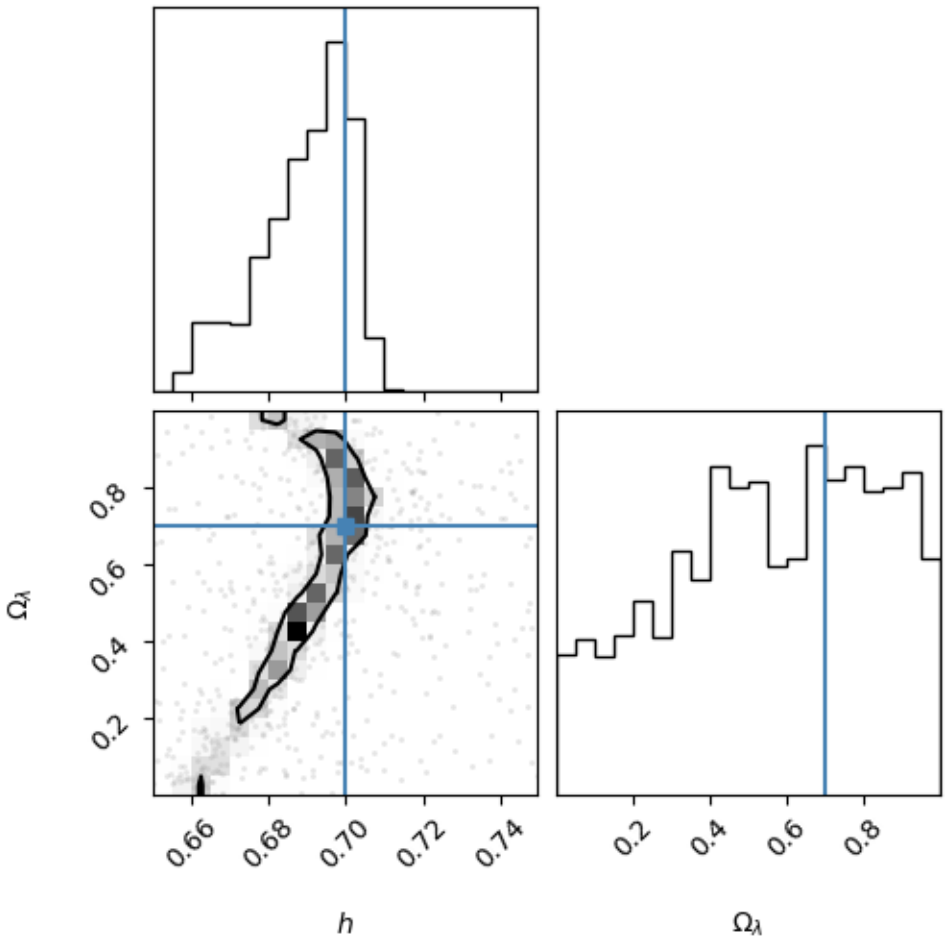


Out:

```
No handles with labels found to put in legend.

<AxesSubplot:xlabel='$h$', ylabel='$\\Omega_\\lambda$'>
```

MCMC-Like Parameter Search

```
my_survey.survey_nestle(vparam_names=['h','Ode0'],
                        bounds={'h':[.65,.75],'Ode0':[0,1]},npoints=200)

my_survey.plot_survey_contour(['h','Ode0'],math_labels=[r'$h$',r'$\Omega_\lambda$'],
↪filled=False)
```



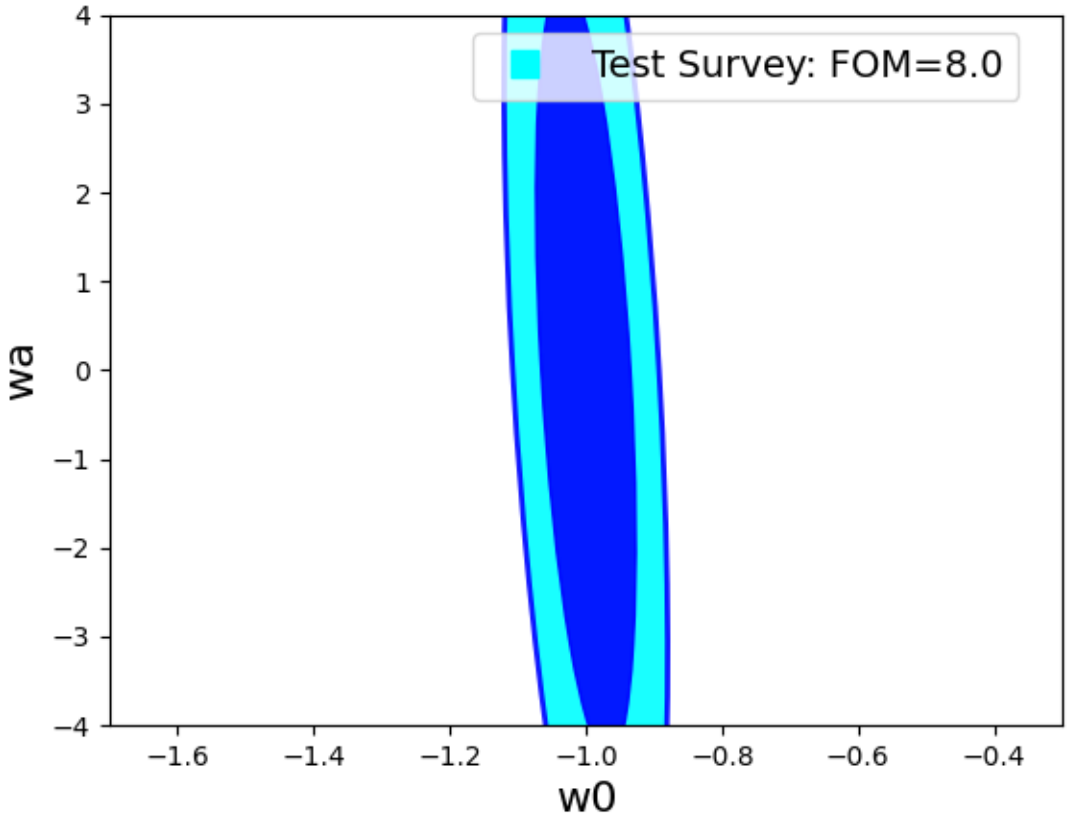
Fisher Matrix Analysis

This will make a 5x5 fisher matrix with the given parameters

```
my_survey.survey_fisher(['h','Ode0','Om0','w0','wa'])
```

Add a prior that assumes perfect knowledge of all other parameters

```
my_survey.fisher_matrix.prior('Om0',0.0001)
my_survey.fisher_matrix.prior('Ode0',0.0001)
my_survey.fisher_matrix.prior('h',0.0001)
my_survey.fisher_matrix.plot('w0','wa',x_limits=[-1.7,-.3],y_limits=[-4,4])
```



Out:

```
<AxesSubplot:xlabel='w0', ylabel='wa'>
```

Total running time of the script: (0 minutes 15.501 seconds)

7.5 API Documentation

7.5.1 Summary

sntd.fitting.fit_data([curves, snType, ...])	The main high-level fitting function.
sntd.simulation.createMultiplyImagedSN(...)	Generate a multiply-imaged SN light curve set, with user-specified time delays and magnifications.
sntd.survey_cosmo.Survey([N, dTL, dTT, zl, ...])	A Survey class that enables cosmology tests with assumptions of redshift distributions and time delay/lens model precision.
sntd.curve_io.image_lc([zpsys])	SNTD class that describes each image light curve of a MISN

Continued on next page

Table 1 – continued from previous page

sntd.curve_io.MISN([telescopename, object_name])	The main object for SNTD.
sntd.curve_io.table_factory(tables[, ...])	This function will create a new curve object using an astropy table or tables.
sntd.ml.realizeMicro([arand, debug, kappas, ...])	Creates a microcaustic realization based on Wambsganss 1990 microlens code.
sntd.ml.microcaustic_field_to_curve(field, ...)	Convolves an expanding photosphere (achromatic disc) with a microcaustic to generate a magnification curve.
sntd.models.unresolvedMISN(curve_models[, ...])	Wrapper class of SNCosmo.Model to provide support for unresolved lensed SN fitting.

7.5.2 Fitting

sntd.fitting.**fit_data**(*curves=None*, *snType='Ia'*, *bands=None*, *models=None*, *params=None*, *bounds={}*, *ignore=None*, *constants={}*, *ignore_models=[]*, *method='parallel'*, *t0_guess=None*, *effect_names=[]*, *effect_frames=[]*, *batch_init=None*, *cut_time=None*, *force_positive_param=[]*, *dust=None*, *microlensing=None*, *fitOrder=None*, *color_bands=None*, *color_param_ignore=[]*, *min_points_per_band=3*, *identify_micro=False*, *min_n_bands=1*, *max_n_bands=None*, *n_cores_per_node=1*, *npar_cores=4*, *max_batch_jobs=199*, *max_cadence=None*, *fit_colors=None*, *fit_prior=None*, *par_or_batch='parallel'*, *batch_partition=None*, *nbatch_jobs=None*, *batch_python_path=None*, *n_per_node=None*, *fast_model_selection=True*, *wait_for_batch=False*, *band_order=None*, *set_from_simMeta={}*, *guess_amplitude=True*, *trial_fit=False*, *clip_data=False*, *use_MLE=False*, *kernel='RBF'*, *refImage='image_1'*, *nMicroSamples=100*, *color_curve=None*, *warning_supress=True*, *micro_fit_bands='all'*, *verbose=True*, ***kwargs*)

The main high-level fitting function.

Parameters

- **curves** (*MISN*) – The MISN object containing the multiple images to fit.
- **snType** (*str*) – The supernova classification
- **bands** (list of Bandpass or str, or Bandpass or str) – The band(s) to be fit
- **models** (list of Model or str, or Model or str) – The model(s) to be used for fitting to the data
- **params** (list of str) – The parameters to be fit for the models inside of the parameter models
- **bounds** (dict) – A dictionary with parameters in params as keys and a tuple of bounds as values
- **ignore** (list of str) – List of parameters to ignore
- **constants** (dict) – Dictionary with parameters as keys and the constant value you want to set them to as values
- **ignore_models** (class:~*list*) – List of model names to ignore, usually used if you did not specify the "models" parameter and let all models for a given SN type be chosen, but you want to ignore one or more.

- **method** (str or list) – Needs to be 'parallel', 'series', or 'color', or a list containting one or more of these
- **t0_guess** (dict) – Dictionary with image names (i.e. 'image_1','image_2') as keys and a guess for time of peak as values
- **effect_names** (list of str) – List of effect names if model contains a PropagationEffect.
- **effect_frames** (list of str) – List of the frames (e.g. obs or rest) that correspond to the effects in effect_names
- **batch_init** (str) – A string to be pasted into the batch python file (e.g. extra imports or filters added to sncosmo.)
- **cut_time** (list) – The start and end (rest frame) phase that you want to fit in, default accept all phases.
- **force_positive_param** (list) – Optional list of parameters to always make positive.
- **dust** (sncosmo.PropagationEffect) – An sncosmo dust propagation effect to include in the model
- **microlensing** (*str*) – If None microlensing is ignored, otherwise should be str (e.g. achromatic, chromatic)
- **fitOrder** (list) – The order you want to fit the images if using parallel method (default chooses by npoints/SNR)
- **color_bands** (list) – If using multiple methods (in batch mode), the subset of bands to use for color fitting.
- **color_param_ignore** (list) – If using multiple methods, parameters you may want to fit for one method but not for color method (e.g. stretch)
- **min_points_per_band** (*int*) – Only accept bands to fit with this number of points fitting other criterion (e.g. minsnr)
- **identify_micro** (*bool*) – If True, function is run to attempt to identify bands where microlensing is least problematic.
- **min_n_bands** (*int*) – Checks the SN to make sure it has this number of bands (with min_points_per_band in each)
- **max_n_bands** (*int*) – The best n bands are chosen from the data.
- **n_cores_per_node** (*int*) – The number of cores to run parallelization on per node
- **npar_cores** (*int*) – The number of cores to devote to parallelization
- **max_batch_jobs** (*int*) – The maximum number of jobs allowed by your slurm task manager.
- **max_cadence** (*int*) – To clip each image of a MISN to this cadence
- **fit_colors** (*list*) – List of colors to use in color fitting (e.g. ['bessellb-besselv','bessellb-besselr'])
- **fit_prior** (*MISN* or bool) – if implementing parallel method alongside others and fit_prior is True, will use output of parallel as prior for series/color. If SNTD MISN object, used as prior for series or color.
- **par_or_batch** (*str*) – if providing a list of SNe, par means multiprocessing and batch means sbatch. Must supply other batch parameters if batch is chosen, so parallel is default.

- **batch_partition** (*str*) – The name of the partition for sbatch command
- **nbatch_jobs** (*int*) – number of jobs (10 jobs for 100 light curves is 10 light curves per job)
- **batch_python_path** (*str*) – path to python you want to use for batch mode (if different from current)
- **n_per_node** (*int*) – Number of SNe to fit per node (in series) in batch mode. If none, just distributes all SNe across the number of jobs you have by default.
- **fast_model_selection** (*bool*) – If you are providing a list of models and want the best fit, turning this on will make the fitter choose based on a simple minuit fit before moving to the full sntd fitting. If false, each model will be fitted with the full sntd fitting and the best will be chosen.
- **wait_for_batch** (*bool*) – if false, submits job in the background. If true, waits for job to finish (shows progress bar) and returns output.
- **band_order** (list) – If you want colors to be fit in a specific order (e.g. B-V instead of V-B depending on band order)
- **set_from_simMeta** (dict) – Dictionary where keys are model parameters and values are the corresponding key in the *MISN*.images.simMeta dictionary (e.g. {'z':'sim_redshift'} if you want to set the model redshift based on a simulated redshift in simMeta called 'sim_redshfit')
- **guess_amplitude** (*bool*) – If True, the amplitude parameter for the model is estimated, as well as its bounds
- **trial_fit** (*bool*) – If true, a simple minuit fit is performed to locate the parameter space for nestle fits, otherwise the full parameter range in bounds is used.
- **clip_data** (*bool*) – If true, criterion like minsnr and cut_time actually will remove data from the light curve, as opposed to simply not fitting those data points.
- **use_MLE** (*bool*) – If true, uses MLE as the parameter estimator instead of the median of the nested sampling samples
- **kernel** (*str*) – The kernel to use for microlensing GPR
- **refImage** (*str*) – The name of the image you want to be the reference image (i.e. image_1,image_2, etc.)
- **nMicroSamples** (*int*) – The number of pulls from the GPR posterior you want to use for microlensing uncertainty estimation
- **color_curve** (astropy.Table) – A color curve to define the relationship between bands for parameterized light curve model.
- **warning_supress** (*bool*) – Turns on or off warnings
- **micro_fit_bands** (*str or list of str*) – The band(s) to fit microlensing. All assumes achromatic, and will fit all bands together.
- **verbose** (*bool*) – Turns on/off the verbosity flag

Returns fitted_MISN – The same MISN that was passed to fit_data, but with new fits and time delay measurements included. List if list was provided.

Return type *MISN* or list

Examples

```
>>> fitCurves=sntd.fit_data(myMISN,snType='Ia', models='salt2-extended',bands=[
↪'F110W','F125W'],
    params=['x0','x1','t0','c'],constants={'z':1.33},bounds={'t0':(-15,15),'x1':(-
↪2,2),'c':(0,1)},
    method='parallel',microlensing=None)
```

7.5.3 Simulation

sntd.simulation.**createMultiplyImagedSN**(*sourcename, snType, redshift, z_lens=None, telescopename='telescope', objectName='object', time_delays=[10.0, 50.0], magnifications=[2.0, 1.0], sn_params={}, av_dists={}, numImages=2, cadence=5, epochs=30, clip_time=[-30, 150], bands=['F105W', 'F160W'], start_time=None, gain=200.0, skynoiseRange=(1, 1.1), timeArr=None, zpsys='ab', zp=None, hostr_v=3.1, lensr_v=3.1, microlensing_type=None, microlensing_params=[], ml_loc=[None, None], dust_model='CCM89Dust', av_host=0.3, av_lens=0, fix_luminosity=False, minsnr=0.0, scatter=True, snrFunc=None*)

Generate a multiply-imaged SN light curve set, with user-specified time delays and magnifications.

Parameters

- **sourcename** (Source or str) – The model for the spectral evolution of the source. If a string is given, it is used to retrieve a Source from the registry.
- **snType** (*str*) – The classification of the supernova
- **redshift** (*float*) – Redshift of the source
- **z_lens** (*float*) – Redshift of the lens
- **telescopename** (*str*) – The name of the telescope used for observations
- **objectName** (*str*) – The name of the simulated supernova
- **time_delays** (list of float) – The relative time delays for the multiple images of the supernova. Must be same length as numImages
- **magnifications** (list of float) – The relative magnifications for the multiple images of hte supernova. Must be same length as numImages
- **sn_params** (dict) – A dictionary with SN model parameters as keys, and some sort of pdf or other callable function that returns the model parameter.
- **av_dists** (dict) – A dictionary with 'host' or 'lens' as keys, and some sort of pdf or other callable function that returns the relevant av parameter.
- **numImages** (*int*) – The number of images to simulate
- **cadence** (*float*) – The cadence of the simulated observations (if timeArr is not defined)
- **epochs** (*int*) – The number of simulated observations (if timeArr is not defined)

- **clip_time** (list) – Rest frame phase start and end time, will clip output table to these values.
- **bands** (list of Bandpass or str) – The bandpass(es) used for simulated observations
- **start_time** (*float*) – Start time for the leading image. If None, start will be the first value in the time array, and the peak will be this $\pm$ the relative time delay for the leading image.
- **gain** (*float*) – Gain of the telescope "obtaining" the simulated observations (if snrFunc not defined)
- **skynoiseRange** (list of float) – The left and right bounds of sky noise used to define observational noise (if snrFunc not defined)
- **timeArr** (list of float) – A list of times that define the simulated observation epochs
- **zpsys** (str or MagSystem) – The zero-point system used to define the photometry
- **zp** (float or list of float) – The zero-point used to define the photometry, list if simulating multiple bandpasses. Then this list must be the same length as bands
- **hostr_v** (*float*) – The r_v parameter for the host.
- **lensr_v** (*float*) – The r_v parameter for the lens.
- **microlensing_type** (*str*) – If microlensing is to be included, defines whether it is "AchromaticSplineMicrolensing" or "AchromaticMicrolensing"
- **microlensing_params** (array or list of int) – If using AchromaticSplineMicrolensing, then this params list must give three values for [nanchor, sigmadm, nspl]. If using AchromaticMicrolensing, then this must be a microcaustic defined by a 2D numpy array
- **ml_loc** (list) – List containing tuple locations of SN on microlensing map (random if Nones)
- **dust_model** (*str*) – The dust model to be used for simulations, see sncosmo documentation for options
- **av_host** (*float*) – The A_V parameter for the simulated dust effect in the source plane
- **av_lens** (*float*) – The A_V parameter for the simulated dust effect in the lens plane
- **fix_luminosity** (*bool*) – Set the luminosity of every SN to be the peak of the distribution
- **minsnr** (*float*) – A minimum SNR threshold for observations when defining uncertainty
- **scatter** (*bool*) – Boolean that decides whether Gaussian scatter is applied to simulated observations
- **snrFunc** (interp1d or dict) – An interpolation function that defines the signal to noise ratio (SNR) as a function of magnitude in the AB system. Used to define the observations instead of telescope parameters like gain and skynoise. This can be a dictionary, so that it's different for each filter with filters as keys and interpolation functions as values.

Returns **MISN** – A MISN object containing each of the multiply-imaged SN light curves and the simulation parameters.

Return type MISN

Examples

```
>>> myMISN = sntd.createMultiplyImagedSN('salt2', 'Ia', 1.33,z_lens=.53, bands=[
↪'F110W'],
    zp=[26.8], cadence=5., epochs=35.,skynoiseRange=(.001,.005),gain=70. , time_
↪delays=[10., 78.],
    magnifications=[7,3.5], objectName='My Type Ia SN', telescopename='HST',
↪minsnr=5.0)
```

7.5.4 Cosmology

class sntd.survey_cosmo.**Survey**(*N=10, dTL=5, dTT=5, zl=0.3, zs=0.8, P=1, calc_ensemble_P=False, name='mySurvey', sys_dTL=0, **kwargs*)

A Survey class that enables cosmology tests with assumptions of redshift distributions and time delay/lens model precision.

Parameters

- **N** (*int*) – The number of discovered glSN (overruled by zl/zs below)
- **dTL** (*float*) – The percent precision on each lens model.
- **dTT** (*float*) – The percent precision on each time delay measurement
- **zl** (*float or list*) – Redshift(s) of the lens(es). If a float, assumes you want N identical SN.
- **zs** (*float or list*) – Redshift(s) of the source(s). If a float, assumes you want N identical SN.
- **P** (*float or list*) – The probability of each source/lens redshift combo, defaults to 1 (i.e. equal weight)
- **calc_ensemble_P** (*bool*) – If True, probabilities are calculated based on gaussian distributions
- **name** (*str*) – Name of your survey

dTc

Returns the combined lens/delay uncertainty assuming statistical uncertainties

plot_survey_contour(*params, math_labels=None, color='#1f77b4', filled=True, confidence=[0.68, 0.95], fom=False, ax=None, alphas=[0.9, 0.3], show_legend=False, show_nestle=True, use_H0=False, **kwargs*)

Plots the contours of a nestle or gridded survey

Parameters

- **params** (*list*) – 2 parameters to plot that have been included in a survey grid or nestle survey
- **math_labels** (*list*) – list of latex labels for params (for labeling plot)
- **color** (*str or tuple*) – color for plotting in matplotlib
- **filled** (*bool*) – filled vs. outlined (if false) contours

- **confidence** (*list or float*) – confidence interval(s) to plot
- **fom** (*bool*) – if True, FOM is calculated. MAJOR WARNING: This will be enormously biased if the full contour is not being drawn, set limits accordingly
- **ax** (*:class:'~plt.axes'*) – If you want the contours drawn onto existing axes
- **alphas** (*list or float*) – draws confidence intervals with this alpha, must match up with "confidence" parameter
- **show_legend** (*bool*) – If True, a legend is shown
- **show_nestle** (*bool*) – If both nestle and grid have been completed, choose nestle if true to show
- **use_H0** (*bool*) – If true and one of the params is 'h', multiply by 100

plot_survey_gradient(*params*, *math_labels=None*)

Plots the gradient survey grid, where one parameter is assumed to be systematically biased

Parameters

- **params** (*list*) – 2 parameters to plot that have been included in a survey grid
- **math_labels** (*list*) – list of latex labels for params (for labeling plot)

survey_fisher(*params*, *dx=1e-06*)

Create a fisher matrix using params, based on the overarching survey parameters.

Parameters

- **params** (*list*) – List of parameters names to be included in fisher matrix
- **dx** (*float*) – The dx used for calculating numerical derivatives

survey_grid(*vparam_names*, *bounds={}*, *npoints=100*, *grad_param=None*, *constants={}*, *grad_param_bounds=None*, *ngrad=10*, *grid_param1=None*, *grid_param2=None*, ***kwargs*)

Calculate cosmological contours by varying 2 parameters in a grid.

Parameters

- **vparam_names** (*list*) – The names of parameters to vary
- **bounds** (*dict*) – Dictionary with param names as keys and list/tuple/array of bounds as values
- **npoints** (*int*) – The number of sample points
- **grad_param** (*str*) – Parameter to assume we're to have measured wrong (see C&M 2009 Figure 4)
- **constants** (*dict*) – Constants that are not the defaults
- **grad_param_bounds** (*dict*) – Bounds for grad_param, same format as bounds
- **ngrad** (*int*) – Number of grid points to vary grad_param
- **grid_param1** (*iterable*) – Optional choice of grid param 1 list (default uniform based on bounds)
- **grid_param2** (*iterable*) – Optional choice of grid param 2 list (default uniform based on bounds)

Returns

- *Adds to class attribute "grid" (a dictionary), with a comma-separated list of*

- *the vparam_names as the key and grid values as the value.*

survey_nestle(*vparam_names*, *bounds*, *constants={}*, *npoints=100*, ***kwargs*)

Calculate cosmological contours in an MCMC-like fashion.

Parameters

- **vparam_names** (*list*) – The names of parameters to vary
- **bounds** (*dict*) – Dictionary with param names as keys and list/tuple/array of bounds as values
- **npoints** (*int*) – The number of sample points
- **grad_param** (*str*) – Parameter to assume we're to have measured wrong (see C&M 2009 Figure 4)
- **constants** (*dict*) – Constants that are not the defaults
- **grad_param_bounds** (*dict*) – Bounds for grad_param, same format as bounds
- **ngrad** (*int*) – Number of grid points to vary grad_param

Returns

- *Adds to class attribute "grid" (a dictionary), with a comma-separated list of*
- *the vparam_names as the key and grid values as the value.*

class sntd.survey_cosmo.**Fisher**(*inroot=''*, *xvar=''*, *yvar=''*, *fixes=''*, *margs=[]*, *data=[]*, *data_is_cov=False*, *params=[]*, *silent=False*, *name='my_fisher'*, *cosmo_truths={'Ode0': 0.7, 'Om0': 0.3, 'h': 0.7, 'w': 0, 'w0': 0, 'wa': -1}*)

Fisher class is more or less copied from Dan Coe's arxiv paper about Fisher matrices: https://arxiv.org/pdf/0906.4123.pdf

Only some minor adjustment from me.

Parameters

- **inroot** (*str*) – base directory to read fisher matrices from if using "load" function
- **xvar** (*str*) – can optionally set an x variable as the "default for other functions"
- **yvar** (*str*) – can optionally set a y variable as the "default for other functions"
- **fixes** (*str*) – comma-separated str of parameters to fix (if fix is called)
- **margs** (*list*) – list of parameters to marginalize over (if marg is called)
- **data** (*np.ndarray*) – an input fisher matrix (NxN where N is the length of your params)
- **data_is_cov** (*bool*) – If true, assumes that "data" is actually a covariance matrix
- **params** (*list*) – Parameter names
- **silent** (*bool*) – verbosity flag for some functions
- **name** (*str*) – Name to plot in legend for figures
- **cosmo_trues** (*dict*) – True parameters to plot as dashed lines in plots

addpar(*param*)

Add a parameter to the list of parameters

Parameters **param** (*str*) – The parameter to add

cov()

Covariance matrix, by definition the inverse of the Fisher matrix

dx (*xvar=''*)
Return uncertainty in parameter (if marginalizing over others)

dxdyp (*xvar=''*, *yvar=''*)
Return uncertainty in two parameters and their correlation

xvar: str x variable

yvar: str y variable

Returns

- **dx** (*float*) – x uncertainty
- **dy** (*float*) – y uncertainty
- **p** (*float*) – rho (correlation parameter)

fix (*fixes=[]*)
Fix parameters constant <==> Remove them from the Fisher matrix

Parameters **fixes** (*list*) – List of parameters to fix, otherwise uses fixes attribute

load (*fishdir=''*)
Loads existing Fisher matrix

Parameters **fishdir** (*str*) – The directory (default inroot)

marg (*margs=[]*)
Marginalize over variables: Remove them from the covariance matrix

Parameters **margs** (*list*) – List of parameters to marginalize over, otherwise uses margs attribute

merit (*param1*, *param2*)
Calculates the figure of merit for a pair of parameters.

Parameters

- **param1** (*str*) – Parameter 1 to use
- **param2** (*str*) – Parameter 2 to use

Returns **FOM** – The Figure of Merit

Return type float

pindex (*param*)
Index of parameter

Parameters **param** (*str*) – Parameter you want the index of

Returns **index** – The index of param

Return type int

plot (*param1*, *param2*, *x_limits*, *y_limits*, *math_label1=None*, *math_label2=None*, *bestfit1=None*, *bestfit2=None*, *alpha=0.9*, *color_list=None*, *print_merit=True*, *col_order=True*, *show_uncertainty=False*)
Plot contours from fisher matrix. This will plot all contours from matrices that have been added together make this matrix.

Parameters

- **param1** (*str*) – Parameter 1 to plot
- **param2** (*str*) – Parameter 2 to plot

- **xlimits** (list or tuple or ndarray) – The x parameter limits for plotting
- **ylimits** (list or tuple or ndarray) – The y parameter limits for plotting
- **math_label1** (*str*) – A latex label for axis 1
- **math_label2** (*str*) – A latex label for axis 2
- **bestfit1** (*float*) – The true/best fit value for parameter 1 (default self.cosmo_truths)
- **bestfit2** (*float*) – The true/best fit value for parameter 2 (default self.cosmo_truths)
- **alpha** (*float*) – The alpha for plotting
- **color_list** (*list*) – List of colors to use for plotting
- **print_merit** (*bool*) – If True, the figure of merit it calculated and added to the legend
- **show_uncertainty** (*bool*) – If true, the final uncertainty in each parameter is printed on the plot

pr()
Print contents

prior(*param*, *sig*)
Set a prior of sig on param

rename(*pdict1=None*)
Rename parameters given a dictionary of names & nicknames

Parameters **pdict1** (*dict*) – Dictionary containing old parameters as keys and new as vals

reorder(*params*)
Matrix with params in new order

Parameters **params** (*list*) – new list of parameters

transform(*params*, *M*)
Transform to new set of parameters using matrix provided (see Coe 2009 above)

Parameters

- **params** (*list*) – New list of parameters
- **M** (ndarray) – The new matrix

7.5.5 I/O

class sntd.curve_io.**image_lc**(*zpsys='AB'*)
SNTD class that describes each image light curve of a MISN

bands = None
str band names used

Type @type

chisq
A property that calculates the chisq based on your best fit model.

fits = None
newDict Contains fit information from fit_data

Type @type

meta = None
dict The metadata for the MISN object, intialized with an empty "info" key value pair. It's populated by added _metachar__ characters into the header of your data file.

Type @type

reduced_chisq
A property that calculates the reduced chisq based on your best fit model.

simMeta = None
dict A dictionary containing simulation metadata if this is a simulated curve object

Type @type

table = None
astropy.table.Table A table containing the data, used for SNCosmo functions, etc.

Type @type

zpsys = None
str The zero-point system for this curve object

Type @type

class sntd.curve_io.**MISN**(*telescopename='Unknown'*, *object_name='Unknown'*)
The main object for SNTD. This organizes a MISN, containing the multiple light curves in self.images, all the fits, etc.

add_image_lc(*myImage*, *key=None*)
Adds an image_lc object to the existing MISN (i.e. adds an image to a MISN)

Parameters

- **myImage** (image_lc) – The curve to add to self.
- **key** (*str*) – The key you want to save this as, default is 'image_1,image_2,etc.'

Returns self

Return type *sntd.curve_io.MISN*

bands = None
list The list of bands contained inside this MISN

Type @type

clip_data(*im*, *minsnr=-inf*, *mintime=-inf*, *maxtime=inf*, *peak=0*, *remove_bands=[]*, *max_cadence=None*, *rm_NaN=True*)
Clips the data of an image based on various properties.

Parameters

- **im** (*str*) – The image to clip
- **minsnr** (*float*) – Clip based on a minimum SNR
- **mintime** (*float*) – Clip based on a minimum time (observer frame relative to peak)
- **maxtime** (*float*) – Clip based on a maximum time (observer frame relative to peak)
- **peak** (*float*) – Used in conjunction with min/max time
- **remove_bands** (*list*) – List of bands to remove from the light curve
- **max_cadence** (*float*) – Clips data so that points are spread by at least max_cadence

- **rm_NaN** (*bool*) – If True, removed NaNs from the data.

color_chisq
A property that calculates the chisq based on your best fit color model.

color_reduced_chisq
A property that calculates the reduced chisq based on your best fit color model.

color_table(*band1s*, *band2s*, *time_delays=None*, *referenceImage='image_1'*, *ignore_images=[]*, *static=False*, *model=None*, *minsnr=0.0*)
Takes the multiple images in self.images and combines the data into a single color curve using defined time delays and magnifications or best (quick) guesses.

Parameters

- **band1s** (*str or list*) – The first band(s) for color curve(s)
- **band2s** (*str or list*) – The second band(s) for color curve(s)
- **time_delays** (dict) – Dictionary with image names as keys and relative time delays as values (e.g. {'image_1':0,'image_2':20}). Guessed if None.
- **referenceImage** (*str*) – The image you want to be the reference (e.g. image_1, image_2, etc.)
- **ignore_images** (list) – List of images you do not want to include in the color curve.
- **static** (*bool*) – Make the color curve, don't shift the data
- **model** (Model) – If you want to use an sncosmo Model (and the guess_t0_amplitude method) to guess time delays
- **minsnr** (*float*) – Cut data that don't meet this threshold before making the color curve.

Returns self

Return type *sntd.curve_io.MISN*

combine_curves(*time_delays=None*, *magnifications=None*, *referenceImage='image_1'*, *static=False*, *model=None*, *minsnr=0*)
Takes the multiple images in self.images and combines the data into a single light curve using defined time delays and magnifications or best (quick) guesses.

Parameters

- **time_delays** (dict) – Dictionary with image names as keys and relative time delays as values (e.g. {'image_1':0,'image_2':20}). Guessed if None.
- **magnifications** (dict) – Dictionary with image names as keys and relative magnifications as values (e.g. {'image_1':0,'image_2':20}). Guessed if None.
- **referenceImage** (*str*) – The image you want to be the reference (e.g. image_1, image_2, etc.)
- **ignore_images** (list) – List of images you do not want to include in the color curve.
- **static** (*bool*) – Make the color curve, don't shift the data
- **model** (Model) – If you want to use an sncosmo Model (and the guess_t0_amplitude method) to guess time delays
- **minsnr** (*float*) – Cut data that don't meet this threshold before making the color curve.

Returns self

Return type *sntd.curve_io.MISN*

meta = None
dict The metadata for the MISN object, intialized with an empty "info" key value pair. It's populated by added _metachar__ characters into the header of your data file.

Type @type

object = None
str Object of interest

Type @type

plot_fit(*method='parallel'*, *par_image=None*)
Makes a corner plot based on one of the fitting methods

Parameters **method** (*str*) – parallel, series, or color

Returns figure object

Return type figure

plot_microlensing_fit(*show_all_samples=False*)
Shows a plot of the best-fit microlensing curve from GPR.

Parameters **show_all_samples** (*bool*) – If True, show all GPR samples.

Returns figure object

Return type figure

plot_object(*bands='all'*, *savefig=False*, *plot3D=False*, *filename='mySN'*, *orientation='horizontal'*, *method='separate'*, *showModel=False*, *showFit=False*, *showMicro=False*, *plot_unresolved=False*, ***kwargs*)

Plot the multiply-imaged SN light curves and show/save to a file. Each subplot shows a single-band light curve, for all images of the SN.

Parameters

- **bands** (str or list of str) – 'all' = plot all bands; or provide a list of bands to plot
- **savefig** (*bool*) – boolean to save or not save plot
- **plot3D** (*bool*) – boolean to plot in 3D with plotly
- **filename** (*str*) – if savefig is True, this is the output filename
- **orientation** (*str*) – 'horizontal' = all subplots are in a single row 'vertical' = all subplots are in a single column
- **method** (*str*) – Plots the result of separate, series, or color curve method
- **showModel** (*bool*) – If true, the underlying model before microlensing is plotted as well
- **showFit** (*bool*) – If true and it exists, the best fit model from self.images['image'].fits.model is plotted
- **showMicro** (*bool*) – If true and it exists, the simulated microlensing is plotted as well
- **plot_unresolved** (*bool*) – If true the individual models of an unresolved fit will be plotted

Returns figure

Return type *~matplotlib.pyplot.figure*

quality_check(*min_n_bands=1*, *min_n_points_per_band=1*, *clip=False*, *method='parallel'*)
Checks the images of a SN to make sure they pass minimum thresholds for fitting.

Parameters

- **min_n_bands** (*int*) – The minimum number of bands needed to pass
- **min_n_points_per_band** (*int*) – The minimum number of bands in a given band to pass
- **clip** (*bool*) – If True, "bad" bands are clipped in place
- **method** (*str*) – Should be parallel, series, or color. Checks all images (parallel), or the series table (series), or the color table (color)

Returns self

Return type *sntd.curve_io.MISN*

series_chisq
A property that calculates the chisq based on your best fit series model.

series_reduced_chisq
A property that calculates the reduced chisq based on your best fit series model.

table = None
Table The astropy table containing all of the data in your data file

Type @type

telescopename = None
str Name of the telescope that the data were gathered from

Type @type

sntd.curve_io.**read_data**(*filename*, ***kwargs*)

Used to read a light curve or curve object in pickle format. Either way, it'll come out as a curve object.

Parameters **filename** (*str*) – Name of the file to be read (ascii or pickle)

Returns curve

Return type curve or MISN

sntd.curve_io.**write_data**(*curves*, *filename=None*, *protocol=-1*)

Used to write a MISN object to a pickle to be read later

Parameters

- **curves** (*~sntd.MISN*) –
- **filename** (*str*) – Name of output file
- **protocol** (*int*) – Pickling protocol

Returns

Return type None

sntd.curve_io.**table_factory**(*tables*, *telescopename='Unknown'*, *object_name='Unknown'*)
This function will create a new curve object using an astropy table or tables.

Parameters

- **tables** (*~astropy.table.Table* or `list`) – Astropy table with all of your data from your data file, or a list of such tables.
- **telescopename** (*str*) – Name of telescope for labeling purposes inside curve object
- **object_name** (*str*) – Name of object for labeling purposes inside curve object (e.g. SN2006jf, etc.)

Returns **curve**

Return type `curve`

7.5.6 Microlensing

`sntd.ml.`**realizeMicro**(*arand=0.25*, *debug=0*, *kappas=0.75*, *kappac=0.15*, *gamma=0.76*, *eps=0.6*, *nray=300*, *minmass=10*, *maxmass=10*, *power=-2.35*, *pixmax=5*, *pixminx=0*, *pixminy=0*, *pixdif=10*, *fracpixd=0.3*, *iwrite=0*, *verbose=False*)

Creates a microcaustic realization based on Wambsganss 1990 microlens code. All parameters are optional as they have defaults, see Wambsganss documentation for details on parameters.

`sntd.ml.`**microcaustic_field_to_curve**(*field*, *time*, *zl*, *zs*, *velocity=<Quantity 10000. km / s>*, *M=<Quantity 1.98840987e+30 kg>*, *width_in_einstein_radii=10*, *loc='Random'*, *plot=False*, *ax=None*, *showCurve=True*, *rescale=True*)

Convolves an expanding photosphere (achromatic disc) with a microcaustic to generate a magnification curve.

Parameters

- **field** (`numpy.ndarray`) – An opened fits file of a microcaustic, can be generated by realizeMicro
- **time** (`numpy.array`) – Time array you want for microlensing magnification curve, explosion time is 0
- **zl** (*float*) – redshift of the lens
- **zs** (*float*) – redshift of the source
- **velocity** (float* `astropy.units.Unit`) – The average velocity of the expanding photosphere
- **M** (float* `Unit`) – The mass of the deflector
- **width_in_einstein_radii** (*float*) – The width of your map in units of Einstein radii
- **loc** (*str or tuple*) – Random is defualt for location of the supernova, or pixel (x,y) coordiante can be specified
- **plot** (*bool*) – If true, plots the expanding photosphere on the microcaustic
- **ax** (*~matplotlib.pyplot.axis*) – An optional axis object to plot on. If you want to show the curve, this should be a list like this: [main_ax,lower_ax]
- **showCurve** (*bool*) – If true, the microlensing curve is plotted below the microcaustic
- **rescale** (*bool*) – If true, assumes image needs to be rescaled: (x-1024)/256

Returns

- **time** (`numpy.array`) – The time array for the magnification curve
- **dmag** (`numpy.array`) – The magnification curve.

class sntd.ml.**AchromaticMicrolensing**(*time*, *dmag*, *magformat='multiply'*, ***kwargs*)
An achromatic microlensing object, defined filter to filter.

propagate(*phase*, *wave*, *flux*)
Propagate the magnification onto the model's flux output.

class sntd.ml.**ChromaticFilterMicrolensing**(*times*, *dmags*, *bands*, *magformat='multiply'*, ***kwargs*)
A chromatic microlensing object, defined filter to filter.

propagate(*phase*, *wave*, *flux*)
Propagate the magnification onto the model's flux output.

7.5.7 Models

class sntd.models.**unresolvedMISN**(*curve_models*, *delays=None*, *magnifications=None*)
Wrapper class of SNCosmo.Model to provide support for unresolved lensed SN fitting.

add_effect(*effect*, *name*, *frame*)
Add a PropagationEffect to the model.

Parameters

- **effect** (*~sncosmo.PropagationEffect*) – Propagation effect.
- **name** (*str*) – Name of the effect.
- **frame** (*{'rest', 'obs', 'free'}*) –

set(***param_dict*)
Set parameters of the model by name.

set_delays(*delays*, *base_t0=0*)
Set the relative delays between blended image models.

Parameters **delays** (*list of float*) – A list of relative delays between models

set_magnifications(*magnifications*, *base_x0=1*)
Set the relative magnifications between blended image models.

Parameters **magnifications** (*list of float*) – A list of relative magnifications between models

7.6 Publications with SNTD

2. Projected Cosmological Constraints from Strongly Lensed Supernovae with the Roman Space Telescope: Pierel, J. D. R.; Rodney, S.; Vernardos, G.; Oguri, M.; Kessler, R.; Anguita, T.

1. Turning Gravitationally Lensed Supernovae into Cosmological Probes: Pierel, J. D. R.; Rodney, S.

7.7 Primary Contributors

Fig. 1: Justin Pierel

Fig. 2: Steve Rodney

7.7.1 Acknowledgements

Logo Design: Eleanor Pierel

CHAPTER 8

API Documentation

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S

T

U

W

Z