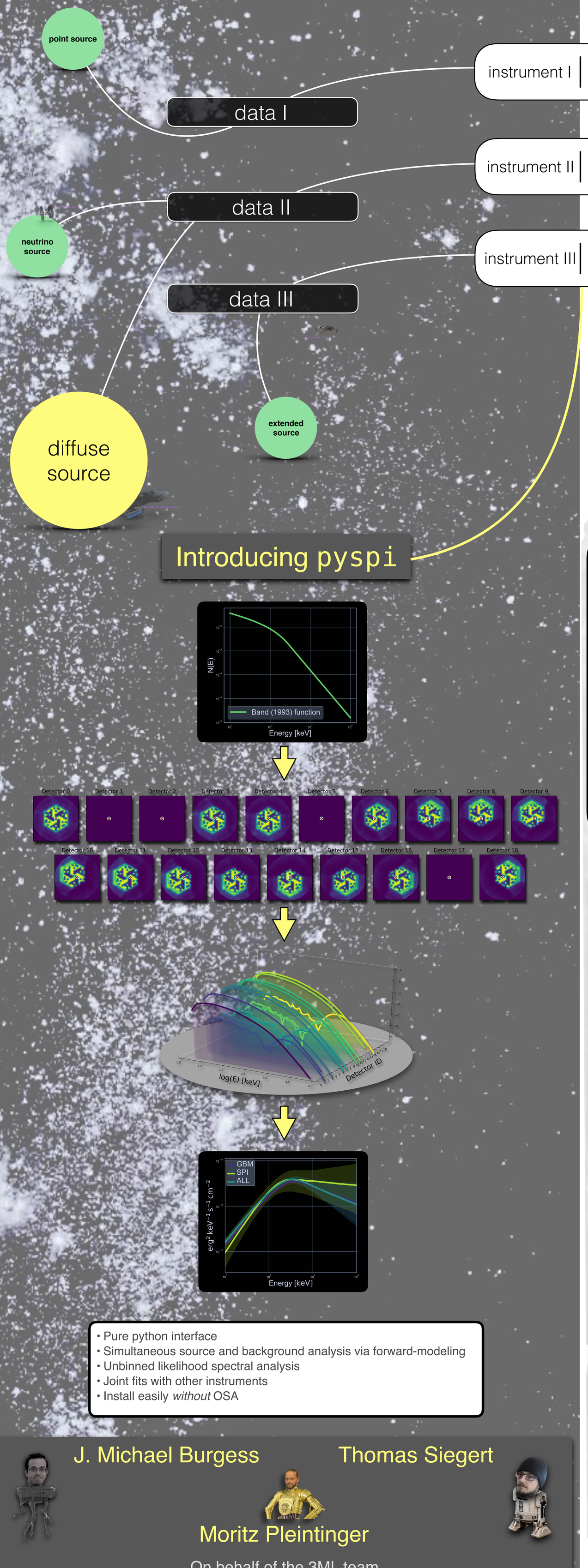
3ML: THE MULTI-MISSION MAXIMUM LIKELIHOOD FRAMEWORK



CONCEPT

- Multi-Mission, Multi-wavelength, Multi-species analysis
- Plugin based system (many provided but build your own)
- Maximum Likelihood and Bayesian analysis
- Time-Energy-Polarization dependent likelihoods
- Python based + C interface

MODELS

- 1,2,3 -D models
- Basic spectral models
- Extended source models (dark matter, etc.)
- All XSPEC models
- Custom models (on the fly!)

PLUGINS

- Generic XY data
- OGIP/FITS/XSPEC-style data
- Photometric Spectra
- Binned Likelihood Profiles
- Fermi-LAT
- HAWC
- VERITAS
- POLAR
- **SPI**

FITTING

- **Optimization**
  - minuit
  - ROOT
  - PyGMO
- **Bayesian**
  - emcee
  - multinest
  - polychord
  - Stan

- Pure python interface
- Simultaneous source and background analysis via forward-modeling
- Unbinned likelihood spectral analysis
- Joint fits with other instruments
- Install easily *without* OSA

J. Michael Burgess

Thomas Siegert

Moritz Pleintinger

On behalf of the 3ML team

Browse the capabilities and see more at threeml.readthedocs.io

Quickstart

In this simple example we will generate some simulated data, and fit them with 3ML.

Let's start by generating our dataset:

```
from threeML import *

# Let's generate some data with y = Powerlaw(x)

gen_function = Powerlaw()


# Generate a dataset using the power law, and a
# constant 30% error

x = np.logspace(0, 2, 50)

xyl_generator = XYLike.from_function("sim_data", function = gen_function,
                                     x = x,
                                     yerr = 0.3 * gen_function(x))

y = xyl_generator.y
y_err = xyl_generator.yerr
```

```
Configuration read from /home/giacomov/.threeML/threeML_config.yml
Plotter is MatPlotLib
Using Gaussian statistic (equivalent to chi^2) with the provided errors.
```

We can now fit it easily with 3ML:

```
fit_function = Powerlaw()

xyl = XYLike("data", x, y, y_err)

parameters, like_values = xyl.fit(fit_function)
```

```
Using Gaussian statistic (equivalent to chi^2) with the provided errors.
Best fit values:
```

parameter	result	unit
source.spectrum.main.Powerlaw.K	1.02 -0.08 +0.09	1 / (cm2 keV s)
source.spectrum.main.Powerlaw.index	-2.002 +/- 0.031	

Correlation matrix:

1.00	-0.86
-0.86	1.00

Values of -log(likelihood) at the minimum:

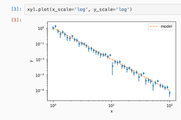
	-log(likelihood)
data	19.354004
total	19.354004

Values of statistical measures:

statistical measures	
AIC	42.963327
BIC	46.5320054

Plot data and model:

```
xyl.plot(x_scale='log', y_scale='log')
```



Compute the goodness of fit using Monte Carlo simulations (NOTE: if you repeat this exercise from the beginning many time, you should find that the quantity "gof" is a random number distributed uniformly between 0 and 1. That is the expected result if the model is a good representation of the data)

fast and easy fitting of data in python

GROND Example

Now we will look at GROND. We get the filter from the 3ML filter library.

(Just play with tab completion to see what is available)

```
grond = PhotometryLike('GROND',
                       filters=threeML_filter_library.ESO.GROND,
                       #g=(21.5,83,.23), # we exclude these filters
                       g=(21,.83),
                       r=(21,.83),
                       i=(21.6,.83),
                       z=(19.6,.83),
                       H=(18.6,.83),
                       K=(18,.83))
```

```
Using Gaussian statistic (equivalent to chi^2) with the provided errors.
```

```
grond.display_filters()
```

Model specification

Here we use XSPEC's dust extinction models for the milky way and the host

```
spec =  Powerlaw() * XS_zdust() * XS_zdust()

data_list = DataList(grond)

model = Model(PointSource('grb',0,0,spectral_shape=spec))

spec.piv_1 = 1E-2
spec.index_1.fix=False
spec.redshift_2 = 0.347
spec.redshift_2.fix = True

spec.e_bmv_2 = 5./2.93
spec.e_bmv_2.fix = True
spec.rv_2 = 2.93
spec.rv_2.fix = True


spec.method_2 = 3
spec.method_2.fix=True


spec.e_bmv_3 = .092/3.08
spec.e_bmv_3.fix = True
spec.rv_3= 3.08
spec.rv_3.fix=True
spec.redshift_3 = 0
spec.redshift_3.fix=True
spec.method_3 = 1
spec.method_3.fix=True

jl = JointLikelihood(model,data_list)
```

We compute $m_{\lambda,0}$ from astromodels photon fluxes. This is done by convolving the differential flux over the filter response:

$$\bar{F}(\mathbf{R}, f_\lambda) \equiv \int_0^\infty \frac{df_\lambda}{d\lambda}(\lambda) R(\lambda) a(\lambda) d\lambda$$

where we have converted the astromodels functions to wavelength properly.

Fit photometric optical spectra using the built in filter library

Making a custom plugin

Let's build a simple (and useless) plugin to see how the process works. First, we import the PluginPrototype class from 3ML.

```
from threeML import PluginPrototype
```

If we try to create a plugin without implementing all the needed memebers, we will run into an error.

```
class BadPlugin(PluginPrototype):
    pass

bad_plugin = BadPlugin('name',{})
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-5-8a1ffdacb3ff> in <module>()
----> 1 bad_plugin = BadPlugin("name",{})

TypeError: Can't instantiate abstract class BadPlugin with abstract methods get_log_like, set_model
```

So, let's instead build a proper plugin

```
from astromodels import Parameter
import collections

class GoodPlugin(PluginPrototype):

    def __init__(self, name):

        # create the hash for the nuisance parameters
        nuisance_parameters = collections.OrderedDict()

        # create a dummy parameter
        par = Parameter("dummy_%s" % name, 1.0, min_value=0.8, max_value=1.2, delta=0.05,
                        free=False, desc="A dummy parameter for %s"

        nuisance_parameters[par.name] = par

        # call the prototype constructor
        super(GoodPlugin, self).__init__(name,nuisance_parameters)


    def set_model(self, model):

        # attach the model to the object
        self._model = model

    def get_log_like(self):

        # this isn't going to be very usefull
        return -99.

    def inner_fit(self):

        return self.get_log_like()
```

```
good_plugin = GoodPlugin('name')
```

```
good_plugin.name
```

```
'name'
```

```
good_plugin.get_log_like()
```

```
-99.0
```

```
good_plugin.nuisance_parameters
```

```
OrderedDict([('dummy_name', Parameter dummy_name = 1.0 [)
                                (min_value = 0.8, max_value = 1.2, delta = 0.05, free = False)])
```

While our plugin is not very useful, we can now see how to build a plugin. Examine the source code of other plugins that are provided. Figure out how to interface your instrument's model evaluation and likelihood computation to python and place these methods inside your plugin class. We have taken care of the rest.

create a plugin for your instrument

Define the model

After you have loaded your data, you need to define a model for them. A model is a collection of one or more sources. A source represents an astrophysical reality, like a star, a galaxy, a molecular cloud... There are 3 kinds of sources: PointSource, ExtendedSource and ParticleSource. The latter is used only in special situations. The models are defined using the package astromodels. Here we will only go through the basics. You can find a lot more information here: astromodels.readthedocs.org

Point sources

A point source is characterized by a name, a position, and a spectrum. These are some examples:

```
# A point source with a power law spectrum

source1_sp = Powerlaw()
source1 = PointSource("source1", ra=23.5, dec=-22.7, spectral_shape=source1_sp)

# Another source with a log-parabolic spectrum plus a power law

source2_sp = Log_parabola() + Powerlaw()
source2 = PointSource("source2", ra=36.5, dec=-27.1, spectral_shape=source2_sp)

# A third source defined in terms of its Galactic latitude and longitude
source3_sp = Cutoff_powerlaw()
source3 = PointSource("source3", l=216.1, b=-74.56, spectral_shape=source3_sp)
```

Extended sources

An extended source is characterized by its spatial shape and its spectral shape:

```
# An extended source with a Gaussian shape centered on R.A., Dec = (30.5, -27.1)
# and a sigma of 2.0 degrees
ext1_spatial = Gaussian_on_sphere(lon0=30.5, lat0=-27.1, sigma=3.0)
ext1_spectral = Powerlaw()

ext1= ExtendedSource("ext1", ext1_spatial, ext1_spectral)

# An extended source with a 3D function
# (i.e., the function defines both the spatial and the spectral shape)
ext2_spatial = Continuous_injection_diffusion()
ext2 = ExtendedSource("ext2", ext2_spatial)
```

NOTE: not all plugins support extended sources. For example, the XYLike plugin we used above do not, as it is meant for data without spatial resolution.

Create the likelihood model

Now that we have defined our sources, we can create a model simply as:

```
model = Model(source1, source2, source3, ext1, ext2)

# We can see a summary of the model like this:
model.display()
```

Model summary:

	N
Point sources	3
Extended sources	2
Particle sources	0

Free parameters (19):

	value	min_value	max_value	unit
source1.spectrum.main.Powerlaw.K	1	1e-30	1000	cm-2 keV-1 s-1
source1.spectrum.main.Powerlaw.index	-2	-10	10	
source2.spectrum.main.composite.K_1	1	1e-30	100000	cm-2 keV-1 s-1
source2.spectrum.main.composite.alpha_1	-2	None	None	
source2.spectrum.main.composite.beta_1	-1	None	None	
source2.spectrum.main.composite.K_2	1	1e-30	1000	cm-2 keV-1 s-1
source2.spectrum.main.composite.index_2	-2	-10	10	
source3.spectrum.main.Cutoff_powerlaw.K	1	1e-30	1000	cm-2 keV-1 s-1
source3.spectrum.main.Cutoff_powerlaw.index	-2	-10	10	
source3.spectrum.main.Cutoff_powerlaw.xc	50	None	None	keV
ext1.Gaussian_on_sphere.lon0	30.5	0	360	deg
ext1.Gaussian_on_sphere.lat0	-27.1	-90	90	deg
ext1.Gaussian_on_sphere.sigma	3	0	20	deg
ext1.spectrum.main.Powerlaw.K	1	1e-30	1000	cm-2 keV-1 s-1
ext1.spectrum.main.Powerlaw.index	-2	-10	10	
ext2.Continuous_injection_diffusion.lon0	0	0	360	deg
ext2.Continuous_injection_diffusion.lat0	0	-90	90	deg

model point and extended sources simultaneously

Example joint fit between GBM and Swift BAT

One of the key features of 3ML is the abil ity to fit multi-messenger data properly. A simple example of this is the joint fitting of two instruments whose data obey different likelihoods. Here, we have GBM data which obey a Poisson-Gaussian profile likelihoog (PGSTAT in XSPEC lingo) and Swift BAT which data which are the result of a "fit" via a coded mask and hence obey a Gaussian (χ^2) likelihood.

```
%matplotlib notebook
from threeML import *
import os
```

```
Configuration read from /Users/jburgess/.threeML/threeML_config.yml
```

Plugin setup

We have data from the same time interval from Swift BAT and a GBM NAI and BGO detector. We have preprocessed GBM data to so that it is OGIP compliant. (Remember that we can handle the raw data with the TimeSeriesBuilder). Thus, we will use the OGIPLike plugin to read in each dataset, make energy selections and examine the raw count spectra.

Swift BAT

```
gbm_dir = "../../examples/gbm"
bat_dir = "../../examples/bat"

bat = OGIPLike('BAT',
               observation=os.path.join(bat_dir,'gbm_bat_joint_BAT.pha'),
               response=os.path.join(bat_dir,'gbm_bat_joint_BAT.rsp'))

bat.set_active_measurements('15-150')
bat.view_count_spectrum()
```

```
Auto-probed noise models:
- observation: gaussian
- background: None
Range 15-150 translates to channels 3-62
Now using 60 channels out of 80
```

```
WARNING RuntimeWarning: Minimum MC energy (10.0) is larger than minimum EBOUNDS energy (8
```


```
WARNING RuntimeWarning: invalid value encountered in sqrt
```

Fermi GBM

```
nai6 = OGIPLike('n6',
                os.path.join(gbm_dir,'gbm_bat_joint_NAI_06.pha'),
                os.path.join(gbm_dir,'gbm_bat_joint_NAI_06.bak'),
                os.path.join(gbm_dir,'gbm_bat_joint_NAI_06.rsp'),
                spectrum_number=1)


nai6.set_active_measurements('8-900')
nai6.view_count_spectrum()

bgo0 = OGIPLike('b0',
                os.path.join(gbm_dir,'gbm_bat_joint_BGO_00.pha'),
                os.path.join(gbm_dir,'gbm_bat_joint_BGO_00.bak'),
                os.path.join(gbm_dir,'gbm_bat_joint_BGO_00.rsp'),
                spectrum_number=1)

bgo0.set_active_measurements('250-30000')
bgo0.view_count_spectrum()
```

```
Auto-probed noise models:
- observation: poisson
- background: gaussian
```

read any OGIP compliant data

With effective are correction

Now let's add an effective area correction between the detectors to see if this fixes the problem. The effective area is a nuisance parameter that attempts to model systematic problems in a instruments calibration. It simply scales the counts of an instrument by a multiplicative factor. It cannot handle more complicated energy dependent

```
# turn on the effective area correction and set it's bounds
nai6.use_effective_area_correction(.2,1.8)
bgo0.use_effective_area_correction(.2,1.8)

model_eac = Model(PointSource('joint_fit_eac',0,0,spectral_shape=band))

jl_eac = JointLikelihood(model_eac, data_list)

jl_eac.fit();
```

```
Best fit values:
```

parameter	result	unit
joint_fit_eac.spectrum.main.Band.K	(2.980 +/- 0.12) x 10^-2	1 / (cm2 keV s)
joint_fit_eac.spectrum.main.Band.alpha	(-9.840 +/- 0.26) x 10^-1	
joint_fit_eac.spectrum.main.Band.xp	(3.310 +/- 0.20) x 10^2	keV
joint_fit_eac.spectrum.main.Band.beta	-2.360 +/- 0.15	
cons_n6	1.560 +/- 0.04	
cons_b0	1.410 +/- 0.10	

Correlation matrix:

1.00	0.95	-0.92	0.31	0.29	0.62
0.95	1.00	-0.82	0.26	0.26	0.52
-0.92	-0.82	1.00	-0.36	-0.45	-0.77
0.31	0.26	-0.36	1.00	0.04	-0.03
0.29	0.26	-0.45	0.04	1.00	0.47
0.62	0.52	-0.77	-0.03	0.47	1.00

Values of -log(likelihood) at the minimum:

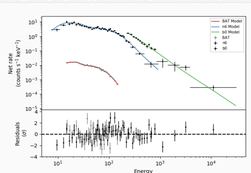
-log(likelihood)	
BAT	40.065347
b0	544.751052
n6	644.715453
total	1229.532012

Values of statistical measures:

statistical measures	
AIC	2471.548770
BIC	2493.326586

Now we have a much better fit to all data sets

```
display_spectrum_model_counts(jl_eac, step=False,min_rate=[.01,10.,10.],data_colors=['gre
```



fit data from different instruments with their appropriate likelihoods

MULTINEST

MULTINEST is a nested sampling algorithm that is designed to integrate the the posterior to obtain the marginal likelihood. For details on the algorithm see Feroz et al. (2009), Feroz et al. (2013), and for details on the input arguments for the python interface we implement, see the documentation of pymultinest. If you find these algorithms useful for your research, please cite the originals authors!

Let's resample the complex likelihood from above with MULTINEST using

```
res = bayes.sample_multinest(400)
```

```
Sampling

MULTINEST has its own convergence criteria... you will have to wait blindly for it to fin
If INS is enabled, one can monitor the likelihood in the terminal for completion informat
 analysing data from chains/fit-1.txt
Maximum a posteriori probability (MAP) point:
```

parameter	result	unit
test.spectrum.main.Complex.mu	(4.4 -0.9 +1.6) x 10	keV

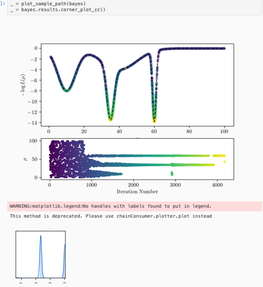
Values of -log(posterior) at the minimum:

-log(posterior)	
custom	13.838423
total	13.838423

Values of statistical measures:

statistical measures	
AIC	-29.676845
BIC	-27.676845
DIC	-44.681983
PDIC	-18.606368
log(Z)	4.457435

```
_ = plot_sample_path(bayes)
_ = bayes.results.corner_plot_cc()
```




```
WARNING:matplotlib.legend:No handles with labels found to put in legend.
```

This method is deprecated. Please use chainConsumer.plotter.plot instead

MULTINEST has fully sampled the likelihood and broken apart the modes. Cleary point-descriptors of the results will be inefficent, but we use the output files of MULTINEST to do multi-modal analysis. For details, consult the MULTINEST documentation.

use your favorite Bayesian samplers

Constructing plugins from TimeSeries

Many times we encounter event lists or sets of spectral histograms from which we would like to derive a single or set of plugins. For this purpose, we provide the **TimeSeriesBuilder** which provides a unified interface to time series data. Here we will demonstrate how to construct plugins from an different of data types.

Constructing time series objects from different data types

The **TimeSeriesBuilder** currently supports reading of the following data type: * A generic PHAII data file * GBM TTE/CSPEC/CTIME files * LAT LLE files

If you would like to build a time series from your own custom data, consider creating a TimeSeriesBuilder.from_your_data() class method.

GBM Data

Building plugins from GBM is achieved in the following fashion

```
cspec_file = get_path_of_data_file('datasets/glg_cspec_n3_bn080916009_v01.pha')
tte_file = get_path_of_data_file('datasets/glg_tte_n3_bn080916009_v01.fit.gz')
gbm_rsp = get_path_of_data_file('datasets/glg_cspec_n3_bn080916009_v00.rsp2')

gbm_cspec = TimeSeriesBuilder.from_gbm_cspec_or_ctime('nai3_cspec',
                                                      cspec_or_ctime_file=cspec_file,
                                                      rsp_file=gbm_rsp)

gbm_tte = TimeSeriesBuilder.from_gbm_tte('nai3_tte',
                                         tte_file=tte_file,
                                         rsp_file=gbm_rsp)
```

LAT LLE data

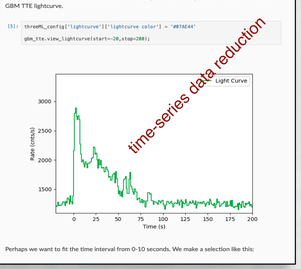
LAT LLE data is constructed in a similar fashion

```
lle_file = get_path_of_data_file('datasets/gll_lle_bn080916009_v10.fit')
ft2_file = get_path_of_data_file('datasets/gll_pt_bn080916009_v10.fit')
lle_rsp = get_path_of_data_file('datasets/gll_cspec_bn080916009_v10.rsp')

lat_lle = TimeSeriesBuilder.from_lat_lle('lat_lle',
                                         lle_file=lle_file,
                                         ft2_file=ft2_file,
                                         rsp_file=lle_rsp)
```

Viewing Lightcurves and selecting source intervals

All time series objects share the same commands to get you to a plugin. Let's have a look at the GBM TTE lightcurve.

```
threeML_config['lightcurve']['lightcurve color'] = '#87AE44'

gbm_tte.view_lightcurve(start=-20,stop=200);
```



time-series data reduction

Perhaps we want to fit the time interval from 0-10 seconds. We make a selection like this:

Read the docs

Get the code

Black holes

MPE

3ML
The Multi-Mission Maximum Likelihood framework