Contents:
CHAPTER 1

Introduction

This documentation gives details of the code in project TransientDict, informally called Magnetron. Magnetron is work in progress; it supplies code to decompose complex time series into a number of simple shapes (currently exponentials). Use at your own peril, but do let me know if it breaks, or make a pull request if it breaks and you’ve fixed it (even better!).

Two caveats:

1. Most of this documentation concerns the branch parameterclass, which is not master, but what I’ve spent most of my development efforts on, and which seems to be fairly stable at this point.

2. There was a second caveat, but I’ve forgotten it now. It will come back to me.
CHAPTER 2

To Do

This I need to do, with no guarantee of completeness. Feel free to add to it.

- create log files for running BurstModel.find_spikes. Needs to save at the very least autocorrelation times, and overall parameters like the number of ensemble walkers and iterations for the MCMC run, and whether the rise time and skewness parameter are fixed between model runs.
- make new implementation in branch parameterclass compatible with the code the others wrote in master.
You can find *Magnetron* as a git repository at its bitbucket address. Use `:: git clone https://dhuppenkothen@bitbucket.org/dhuppenkothen/transientdict.git` to clone the repository to your local disc, or set up your ssh client.
Magnetron requires a number of packages, as listed below:

- numpy
- scipy
- emcee
- acor
- triangle
- (dnest if you want to play around on branch dnest.)

If it breaks because of a failed import, I’ve probably forgotten something.
Installing Magnetron

There’s currently no way to install *Magnetron* as such; it isn’t a full-fledged *python* module, just some scripts. This may be implemented in the future. Use at your own peril.
Usage of the scripts in *Magnetron* depends strongly on the use case. Here, we focus on the usage of the code at the highest level. For details on the individual classes consult *The Magnetron API*.

### 6.1 Defining a Model

There are two classes in `burstmodel.py` that are used to define and work with models:

- **BurstDict** defines the actual model. It can be invoked with an arbitrary number of model model components like this:

```python
import numpy as np
import burstmodel
import word

## make some fake Poisson-distributed data
times = np.arange(1000)/1000.0
counts = np.ones(len(times))*20.0
poisson_counts = np.array([np.random.poisson(c) for c in counts])

## now create a BurstDict instance with a single model shape of
## type word.TwoExp:
bd = burstmodel.BurstDict(times, poisson_counts, word.TwoExp)
```

Defining a model with more than one components is simple: replace the `word.TwoExp` call by a list of calls of `word.TwoExp` (or another model defined in word, if such a thing existed):

```python
## use three model components
nwords = 3

## define a list with the model components:
wordlist = [word.TwoExp for w in xrange(nwords)]

## now define model:
bd = burstmodel.BurstDict(times, poisson_counts, wordlist)
```

Making a model light curve requires the appropriate `parameters.TwoExpParameters` object:

```python
import parameters

## define parameters
t0 = 0.2
```
log_scale = -4.0
log_amp = 10.0
log_skew = 2.0

## make TwoExpParameters object, set log=True for scale, skew and amp in log units:
theta = parameters.TwoExpParameters(t0=t0, scale=log_scale, amp=log_amp,
                                          skew=log_skew, log=True)

or, for a model with several components, a parameters.TwoExpCombined object:

nwords = 3
## first component: [t0, log_scale, log_amp, log_skew]:
c1 = [0.1, -5.0, 10.0, 1.0]
## second component, same form:
c2 = [0.3, -6.0, 12.0, 0.0]
## third component, same form:
c3 = [0.7, -4.0, 8.0, -1.0]

## whole (flat) parameter list: [c1, c2, c3, log(background)]
bkg = 3.0
theta_list = c1
theta_list.extend(c2)
theta_list.extend(c3)
theta_list.append(bkg)

## make parameter object, each component has its own scale and skew, 
## thus scale_locked and skew_locked are False:
theta_combined = parameters.TwoExpCombined(theta_list, nwords, scale_locked=False,
                                           skew_locked=False, log=True, bkg=True)

To make a model of the background only, simply make an instance of BurstDict with an empty list, and similarly an instance of TwoExpCombined with a background parameter only:

bkg = 4.0
bd = burstmodel.BurstDict(times, poisson_counts, [])
theta = parameters.TwoExpCombined([bkg], 0, log=True, bkg=True)

We can now make a model light curve with our parameter object (simple, 1-component case):

## nbins sets the number of points per time bin to average over
model_counts = bd.model_means(theta, nbins=10)

If we are trying to simulate, we can also make a realisation of a light curve by running the model light curve through a Poisson distribution:

poisson_model_counts = bd.poissonify(theta)

or we can plot the model to file:

bd.plot_model(theta, plotname="myplot")

If you have a second parameter set that you want to overplot, e.g. the posterior maximum from an MCMC run, you can do it like this:

bd.plot_model(theta, postmax=other_theta, plotname="myplot")

but you are advised to plot results from an MCMC run using the BurstModel `method plot_results.

• BurstModel is a high-level class that can run MCMC on a model and do stuff with the results. Invoking it is simple:
import numpy as np
import burstmodel
import word
import parameters

## make some Poisson data
times = np.arange(1000)/1000.0
counts = np.ones(len(times))*20.0
poisson_counts = np.array([np.random.poisson(c) for c in counts])

## make an instance of BurstModel
bm = burstmodel.BurstModel(times, poisson_counts)

See sections below for more details on what to do once you’ve got a burst model defined.

### 6.2 Defining a Posterior Probability Density Function

Sometimes, for applications outside the ones served by the classes and methods in burstmodel.py, it may be convenient to just define the log-posterior probability density function, for use with other code.

Before delving into the details of what one can do with BurstModel, a short description of the relevant class in burstmodel.py, WordPosterior. This class takes the data as input, as well as an instance of BurstDict that defines the model:

```python
import numpy as np
import word
import parameters
import burstmodel

## make some fake data
times = np.array(1000)/1000.0
counts = np.ones(len(times))*20.0
poisson_counts = np.array([np.random.poisson(c) for c in counts])

## define the BurstDict instance
nwords = 3
wordlist = [word.TwoExp for w in xrange(nwords)]
bd = burstmodel.BurstDict(times, poisson_counts, wordlist)

## define the log posterior:
lpost = burstmodel.WordPosterior(times, poisson_counts, bd,
    scale_locked=False, skew_locked=False,
    log=True, bkg=True)
```

One can now simply call lpost on a list of parameters:

```python
nwords = 3
## first component: [t0, log_scale, log_amp, log_skew]:
c1 = [0.1, -5.0, 10.0, 1.0]
## second component, same form:
c2 = [0.3, -6.0, 12.0, 0.0]
## third component, same form:
c3 = [0.7, -4.0, 8.0, -1.0]

## whole (flat) parameter list: [c1, c2, c3, log(background)]
bkg = 3.0
theta_list = [c1, c2, c3, bkg]
```
theta_list = np.array(theta_list).flatten()

## call log-posterior:
log_post_prob = lpost(theta_list)

Note that \texttt{lpost} can take either a simple list of parameters as input, or an object of type \texttt{TwoExpParameters} or \texttt{TwoExpCombined}:

## make parameter object, each component has its own scale and skew, 
## thus \texttt{scale\_locked} and \texttt{skew\_locked} are False:
theta_combined = parameters.TwoExpCombined(theta_list, nwords, scale_locked=False, skew_locked=False, log=True, bkg=True)

log_post_prob = lpost(theta_combined)

It is also possible to call the prior and the log-likelihood (for Poisson data) independently, again either with a list of parameters or an object as defined in \texttt{parameters.py}:

log_prior_prob = lpost.logprior(theta_combined)
log_likelihood = lpost.loglike(theta_combined)

### 6.3 Running MCMC on an Individual Model

To run MCMC, define a \texttt{BurstDict} instance as above, a list of initial parameters for the model, and then call \texttt{BurstModel.mcmc} on it:

nwords = 3
## times and poisson_counts are lists with the data time stamps and counts/bin
bd = burstmodel.BurstDict(times, poisson_counts, \[word.TwoExp for w in xrange(nwords)\])

## first component: \([t0, log\_scale, log\_amp, log\_skew]\):
c1 = [0.1, -5.0, 10.0, 1.0]
## second component, same form:
c2 = [0.3, -6.0, 12.0, 0.0]
## third component, same form:
c3 = [0.7, -4.0, 8.0, -1.0]

## whole (flat) parameter list: \([c1, c2, c3, log(background)]\)
bkg = 3.0
theta_list = [c1, c2, c3, bkg]
theta_list = np.array(theta_list).flatten()

## define BurstModel object:
bm = burstmodel.BurstModel(times, poisson_counts)

## define number of ensemble walkers:
nwalker = 500
## define number of burn-in iterations:
burnin = 200
## define number of actual iterations after burn-in:
niter = 200

## now run MCMC:
sampler = bm.mcmc(bd, theta_list, nwalker=nwalker, niter=niter, 
burnin=burnin, scale\_locked=False, skew\_locked=False, 
log=True, bkg=True, plot=True, plotname="myplot")
Magnetron Documentation, Release 0.0.0

BurstModel.mcmc returns an instance of type emcee.EnsembleSampler. If plot=True, a triangle-plot with the posterior probability distributions derived from the MCMC run will be saved to plotname.png. It is possible, of course, to run BurstModel.plot_mcmc on sampler.flatchain manually afterwards, but the implementation in BurstModel.mcmc takes care of meaningfully labelling the axes in the triangle plot.

6.4 Making Sense of an MCMC Run

There are several things to do with the output of an MCMC run. Most of these use sampler.flatchain, and in the following, I will define

```python
## times and poisson_counts are lists of time stamps and counts/bin
bm = burstmodel.BurstModel(times, poisson_counts)

## sampler is an emcee.EnsembleSampler object as returned by BurstModel.mcmc
sample = sampler.flatchain

The methods currently implemented in BurstModel to make use of the output of an MCMC run are:

- **plot_mcmc**: takes sample, a name for the plot and a list of labels for the axes and saves a triangle plot:

  ```python
  plotlabels=['t0', "log(scale)", "log(amplitude)", "log(skewness)"
  ```

  ```python
  ## note that the number of labels for the plot must be the same as the
  ## number of parameters in sample:
  assert len(plotlabels) == np.min(np.shape(samples)), "Incorrect number of plot labels"

  ## now we can create the triangle plot
  bm.plot_mcmc(sample, "myplot", plotlabels)
  ```

- **find_postmax**: find the posterior maximum and 0.05, 0.5 and 0.95 quantiles. Unlike most of the other methods, this requires the actual emcee.EnsembleSampler object, because it requires its attribute flatlnprobability:

  ```python
  ## number of model components:
  nwords = 3
  quants, postmax = bm.find_postmax(sampler, nwords, scale_locked=False, skew_locked=False, log=True, bkg=True)
  ```

  Quantiles are computed by BurstModel._quantiles (a direct call to that method allows direct specification of the quantile range), and returned as a list of parameters.TwoExpParameters or parameters.TwoExpCombined objects: [lower quantile, median, upper quantile]. Similarly, postmax will be an object of the same parameter class as the quantiles.

- **plot_chains**: plot a time series of the Markov chains for each parameter in the model, for diagnostic purposes (e.g. to see whether the chain has converged):

  ```python
  bm.plot_chains(sample, niter, "mymodel")
  ```

  One needs to know the number of iterations in the MCMC run, because otherwise splitting up the flat sample by walkers doesn’t work (although for a converged chain, this shouldn’t matter). The method produces a splurge of n plots, called mymodel_p[i]_chains.png, where i is the ith parameter.

- **plot_results**: Plot the data as a count rate instead of counts/bin, overplot the posterior maximum (if postmax is not None), as well as the 0.05 and 0.95 quantiles and median for each time step in the list of time stamps, derived from nsamples model light curves from nsamples randomly chosen parameter sets from sample:
bm.plot_results(sample, postmax=theta_postmax, nsamples=1000, scale_locked=False, skew_locked=False, bkg=True, log=True, bin=True, nbins=10, namestr="mymodel")

postmax needs to be an object of a parameter class (for example as returned by find_postmax). If bin=True, then the light curve will be binned to a new time resolution, which must be an integer multiple of the original time resolution (the multiple is set in nbins). The method automatically deduces the number of model components, and saves the resulting plot in a file mymodel_k[nwords]_lc.png.

6.5 Running MCMC for a Sequence of Models with Increasing Number of Model Components

BurstModel implements a method that takes a lot of the above, and runs it over an iteratively larger number of model components (starting with only background). This is easy and straightforward to run, but takes quite a long time, depending on the number of model parameters and the number of ensemble walkers/iterations.

The method is called like this:

```python
## define burst model, times and poisson_counts are lists with data time
## stamps and counts/bin, respectively
bm = burstmodel.BurstModel(times, poisson_counts)

## maximum number of model components to consider:
nmax = 10
## number of ensemble walkers:
nwalker = 500
## number of burn-in iterations:
burnin=200
## number of iterations after burn-in:
niter=200

## run find_spikes to run a number of models
all_means, all_err, all_postmax, all_quants, all_theta_init
    = bm.find_spikes(nmax=nmax, nwalker=nwalker, niter=niter, burnin=burnin, scale_locked=False, skew_locked=False, namestr="mymodel")
```

For each model from 0 to nmax, it does an MCMC run, computes posterior maxima and quantiles, and produces plots for the light curve with models, posterior distributions of parameters and Markov chains, as described above.

The method determines the initial parameter set for each model component automatically: the initial parameters for the model components already present in the previous model (with one fewer component) is set to the posterior maximum of the previous MCMC run. The initial peak position of a new model component is located at the greatest discrepancy between the data and the posterior maximum model light curve of the previous model with one fewer model component. If the rise time (=scale) is the same for all model components, the initial guess for the new model will be based on the posterior maximum of the previous model. Otherwise it is set to 1/10 of the duration of the time series. Similarly, if the skewness parameter is the same for all model components, the initial guess for the new model will be based on the posterior maximum of the old model. Otherwise, the skewness is set to 0. The initial guess for the amplitude of the new component is directly derived from the difference between the data and the posterior maximum model light curve of the previous model.

find_spikes returns a number of lists of parameter-type objects for each model used, returning (in order): posterior means, standard deviation from posterior means, posterior maxima, posterior quantiles (0.05, 0.5, 0.95) and initial parameter guesses for each MCMC run.
find_spikes also saves the MCMC samples, posterior maximum, associated log-probabilities, posterior means, standard deviations, quantiles and initial parameter choices for the MCMC run in a dictionary, which is written to disc in a python pickle file of name mymodel_k[i]_posterior.dat, where \( i \) is the number of components in the model.
The primary objective is to decompose a time series into a number of simple shapes. This is done by searching for a likely position of a peak, then defining a model with a single peak + a background parameter, and running MCMC via emcee. Note that the branch dnest also supports Diffusive Nested Sampling, if the relevant code is installed. The results of the MCMC run are stored in a python pickle file as well as a number of diagnostic plot. Iteratively, a new model will be produced with another model component added at the most likely location (the highest outlier of the data-previous model residuals). Again MCMC provides an approximation of the posterior distribution of the parameters. This procedure is repeated up to the maximum number of model components defined (10 by default).

7.1 Tests

Simple tests are implemented in parameter_tests.py. The functions in this script test the basic functionality of the classes defined in parameters.py, word.py and burstdict.py. These tests can be run all together or individually from within python or from the command line like this:

$ python parameter_tests.py --help


Various tests for the classes defined in parameters.py, word.py and
burstdict.py

optional arguments:
-h, --help            show this help message and exit
-p, --parameters     Run parameter class tests
-w, --word           Run word class tests
-d, --burstdict      Run burstdict class tests
-a, --all            Run all tests at once!
--post               Run tests on class WordPosterior with new parameter
                        implementation
-m, --model          Run tests on class BurstModel with new parameter
                        implementation
-l, --longrun        When running BurstModel tests, do you want to perform a
                        long MCMC run?

The tests for class BurstModel support a --longrun option; for many quick checks on whether the code breaks or basic functionality is there, a full MCMC run would take too much time, thus by default the number of ensemble walkers and iterations used is low. When -l or --longrun is set, a longer MCMC run will be performed.
7.2 Modeling Data with Model Shapes

Running the whole procedure on a single or multiple time series proceeds via the script `samescalesameskew.py`. This script can be invoked from the command line with a multitude of options:

```bash
$ python samescalesameskew.py --help
usage: samescalesameskew.py [-h] (-a | -s) [-w NWALKER] [-i NITER]
                          [--instrument INSTRUMENT] [--lock-scale]
                          [--lock-skew] [-f FILENAME] [-d DIR]
```

Model magnetar bursts with spikes!

Optional arguments:
- `-h, --help` show this help message and exit
- `-a, --all` run on all files in the directory
- `-s, --single` run on a single file
- `-w NWALKER, --nwalker NWALKER` Number of emcee walkers
- `-i NITER, --niter NITER` number of emcee iterations
- `--instrument INSTRUMENT` Instrument data was taken with
- `--lock-scale` If true, scale will be the same for all words
- `--lock-skew` If true, skew will be the same for all words

Single file:
- `-f FILENAME, --filename FILENAME` file with data

All bursts:
- `-d DIR, --dir DIR` directory with data files

There’s a main switch `--single` versus `--all`, which tells the script whether to run on a single data file (which then needs to be specified via the `-f` option), or on all files in a directory (which needs then to be specified with the `-d` or `--dir` option).

Data files must be in ASCII format and have at least two columns, where the first two will be read out. The first column must include the time stamps of the data points, the second column the counts per bin. Unbinned data is currently not supported.

Despite the name of the script, whether the model considers one rise time and/or skewness parameter per model component, or one rise time and/or skewness parameter for all model components simultaneously can be set with the keywords `--lock-scale` (for the rise time) and `--lock-skew` for the skewness parameter. Note that these are True/False arguments: inclusion of the argument on the command line will automatically set this True, absence of it on the command line will set it False.

Arguments `--nwalker` and `--niter` set the number of ensemble walkers and iterations for the MCMC run, respectively. At this point, one cannot change this between models considered (this would need to be implemented separately).

The `--instrument` argument currently does nothing; at the moment we only consider data recorded with Fermi/GBM. If other data types are used, this could potentially be useful in the future to read in data in a consistent manner.

Outputs are saved in a number of files, all of which have a common root. As we currently only look at Fermi/GBM data, the root for the output filenames are taken from the input filename, minus `_data.dat` at the end.

For each model, the script saves a python pickle file under `fileroot_k[n]_posterior.dat` (where `[n]` is the number of components in the model) with a dictionary with the following keywords:

- `means`: posterior means of the parameters, in a `parameters.TwoExpCombined` object
• **max**: posterior maximum of the parameters, in a `parameters.TwoExpCombined` object

• **sampler**: list of parameter sets, as given in `s.flatchain`, where `s` is an object of type `emcee.EnsembleSampler`.

• **lnprob**: log posterior probability of the parameter sets stored in `sampler`

• **err**: standard deviation for each parameter as computed from the samples in `sampler`

• **quants**: list with 0.05, 0.5 and 0.95 quantiles for each parameter.

• **init**: initial parameter set used as a starting point for the MCMC run

• **niter**: number of iterations in MCMC run; this is a recent addition and not yet present in every data file

Three types of plots are saved:

1. A triangle plot of the posterior parameter distributions, under `fileroot_k[n]_posterior.png`

2. the original time series with the model of the posterior maximum overplotted in blue, and models for the 0.05, 0.5 and 0.95 quantiles derived from 1000 randomly chosen parameter sets overplotted in red (bands), in `fileroot_k[n]_lc.png`

3. time series of the actual Markov chains for each parameter in `fileroot_k[n]_p[j]_chains.png`. `j` is the jth parameter; I could put the actual parameter names, but I’m currently too lazy to do this for purely diagnostic plots (also, with a bit of knowledge of the code, it’s easy to read off which is which)

4. for all models considered, a plot of the posterior quantiles of each parameter versus the number of components in the model, grouped by parameter type. Produces four plots for the `word.TwoExp` model currently used in all analyses: `fileroot_t0.png` for the peak positions of each component, `fileroot_log_scale.png` for the logarithm of the rise times, `fileroot_log_amp.png` for the logarithm of the component amplitudes, and `fileroot_log_skew.png` for the logarithm of the skewness parameter.

Below a few examples on how to run the script.

1. Run on a single time series data file, with no common parameters between model components; emcee will use 500 ensemble walkers and evolve the Markov chains for 100 iterations (after a standard 200 iterations of burning in):

   ```
   $ python samescalesameskew.py -s -w 500 -i 100 -f "mydata.dat"
   ```

2. Run on all data files in directory `./data/`, with the rise times linked between model components:

   ```
   $ python samescalesameskew.py -a -w 500 -i 100 --lock-scale -d "/data/"
   ```

3. Run on all data files in current directory, with rise times and skewness parameter linked between model components:

   ```
   $ python samescalesameskew.py -a -w 500 -i 100 --lock-scale --lock-skew -d "/"
   ```

### 7.3 Extracting Information from Many Bursts

Making inferences over many bursts can be difficult. By default, the code run by `samescalesameskew.py` produces some output in the form of the MCMC samples for the parameters, as well as diagnostic plots. It is possible to re-make these plots from the saved posterior distributions, change details of these plots, and gather quantities like the posterior maxima and quantiles into one data file for analysis across a whole ensemble of time series.

The easiest way to do this is by fiddling with `plot_parameters.py`. This re-makes most of the plots returned by `samescalesameskew.py`, but plotting can be commented out if only a file with the combined results of the MCMC runs for many bursts is required.

Note that this scripts is currently set up to deal exclusively with Fermi/GBM data, which comes out of my pipeline in files with `BurstID_BurstStartTime_data.dat`-format.
plot_parameters.py can be called from the command line like this:

```bash
$ python plot_parameters.py --help
```

```
```

Model magnetar bursts with spikes!

optional arguments:

- `-h`, `--help` show this help message and exit
- `--scale-locked` Scale the same for all words?
- `--skew-locked` Skew the same for all words?
- `-d DATA_DIR`, `--dir DATA_DIR` Directory where the data files are located
- `-b BID`, `-bid BID` Pick specific burst ID to run on
- `-n NSAMPLES`, `--nsamples NSAMPLES` Number of samples to be used in average light curve.
- `-i NITER`, `--niter NITER` Number of iterations in MCMC run

Again, one must specify whether rise time and skewness parameter are the same for each model component. This requires knowledge of whatever arguments were used when running the analysis itself. By default, the script takes the entire contents of directory specified with `-d` or `--dir` (default is `./`), but it is possible to specify a Fermi/GBM BurstID with `-b` or `--bid` to run on. For the quantiles overplotted on the output time series plots of the data and models, one may specify how many samples to use in the computation of the quantiles via `-n` or `--nsamples`; a larger number translates into longer compute times. If the number specified with this argument is greater than the number of samples in the files storing the MCMC samples, it is automatically re-set to that number.

The argument `-i`, `--niter` is a recent addition. Previously, I did not save the number of iterations per MCMC run anywhere, which makes computing the MCMC time series for diagnostics *a posteriori* quite difficult. For those files without `niter` keyword in `fileroot_posterior.dat`, `--niter` must be set explicitly, or the code throws an exception.

This script returns some of the same plots as `samescalesameskew.py`:

- `fileroot_k[n]lc.png`
- `fileroot_k[n]p[j]chains.png`
- `fileroot_t0.png`, `fileroot_log_scale.png`, `fileroot_log_amp.png`, `fileroot_log_skew.png`,

as well as a python pickle file with a dictionary storing quantities for all models and time series files in the directory considered:

- `t0_max`, `t0_cl`, `t0_m`, `t0_cu`: posterior maximum, 0.05, 0.5 and 0.95 quantiles for the peak time
- `scale_max`, `scale_cl`, `scale_m`, `scale_cu`: posterior maximum, 0.05, 0.5 and 0.95 quantiles for the log rise time
- `amp_max`, `amp_cl`, `amp_m`, `amp_cu`: posterior maximum, 0.05, 0.5 and 0.95 quantiles for the log amplitude
- `skew_max`, `skew_cl`, `skew_m`, `skew_cu`: posterior maximum, 0.05, 0.5 and 0.95 quantiles for the log skewness parameter

The latter can be used for further ensemble analysis.
The Magnetron API

Placeholder for shiny new documentation, yet to come.
CHAPTER 9

Indices and tables

- genindex
- modindex
- search