MC3 Documentation

Release 1.0

Patricio Cubillos

February 05, 2016

Contents

Author Patricio Cubillos and collaborators (see *Team members*)

Contact patricio.cubillos[at]oeaw.ac.at

Organizations University of Central Florida (UCF), Space Research Institute (IWF)

Web Site https://github.com/pcubillos/MCcubed

Date February 05, 2016

1 Features

MC3 is a powerful Bayesian-statistics tool that offers:

- Levenberg-Marquardt least-squares optimization.
- Markov-chain Monte Carlo (MCMC) posterior-distribution sampling following the:
  - Metropolis-Hastings algorithm with Gaussian proposal distribution, or
  - Differential-Evolution MCMC (recomended).

The following features are available when running MC3:

- Execution from the Shell prompt or interactively through the Python interpreter.
- Single- or multiple-CPU parallel computing.
- Uniform non-informative, Jeffreys non-informative, or Gaussian-informative priors.
- Gelman-Rubin convergence test.
- Share the same value among multiple parameters.
- Fix the value of parameters to constant values.
- Correlated-noise estimation with the Time-averaging or the Wavelet-based Likelihood estimation methods.

2 Team members

- Patricio Cubillos (UCF, IWF) patricio.cubillos[at]oeaw.ac.at
- Joseph Harrington (UCF)
- Nate Lust (UCF)
- AJ Foster (UCF)
- Madison Stemm (UCF)

3 License

4 Be Kind

Please cite this paper if you found MC3 useful for your research: Cubillos et al. 2016: On the Correlated Noise Analyses Applied to Exoplanet Light Curves, in preparation.

We welcome your feedback, but do not necessarily guarantee support. Please send feedback or inquiries to:

Patricio Cubillos (patricio.cubillos[at]oeaw.ac.at)

Thank you for using MC3!

5 Contents

5.1 Getting Started

System Requirements

MC3 (version 1.1) is known to work (at least) on Unix/Linux (Ubuntu) and OSX (10.9+) machines, with the following software:

- Python (version 2.7)
- Numpy (version 1.8.2+)
- Scipy (version 0.13.3+)
- Matplotlib (version 1.3.1+)
- mpi4py (version 1.3.1+)
- Message Passing Interface, MPI (MPICH preferred)

MC3 may work with previous versions of these software. However we do not guarantee nor provide support for that.

Install

To obtain the latest MCcubed code, clone the repository to your local machine with the following terminal commands. First, create a top-level directory to place the code:

```
mkdir MC3_demo/
cd MC3_demo/
topdir=`pwd`
```

Clone the repository to your working directory:

```
git clone https://github.com/pcubillos/MCcubed
```

Compile

Compile the C code:

```
cd $topdir/MCcubed/src/cfuncs
make
```

To remove the program binaries, execute (from the respective directories):

```
make clean
```

Example 1 (Interactive)

The following example (demo01) shows a basic MC3 MCMC run from the Python interpreter. This example fits a quadratic polynomial curve to a dataset. First create a folder to run the example (alternatively, run the example from any location, but adjust the paths of the Python script):

```
cd $topdir
mkdir run01
cd run01
```

Now start a Python interactive session. This script imports the necesary modules, creates a noisy dataset, and runs the MCMC:

```
import sys
import numpy as np
import matplotlib.pyplot as plt
sys.path.append("../MCcubed/src/")
import mccubed as mc3

# Get function to model (and sample):
sys.path.append("../MCcubed/examples/models/")
from quadratic import quad

# Create a synthetic dataset:
x = np.linspace(0, 10, 100)          # Independent model variable
p0 = 3, -2.4, 0.5                    # True-underlying model parameters
y = quad(p0, x)                      # Noiseless model
uncert = np.sqrt(np.abs(y))          # Data points uncertainty
error = np.random.normal(0, uncert)  # Noise for the data
data = y + error                     # Noisy data set

# Fit the quad polynomial coefficients:
params = np.array([ 20.0, -2.0, 0.1])  # Initial guess of fitting params.

# Run the MCMC:
allp, bp = mc3.mcmc(data, uncert, func=quad, indparams=[x],
                    params=params, numit=3e4, burnin=100)
```

Outputs

That's it, now let's see the results. `MC3` will print out to screen a progress report every 10% of the MCMC run, showing the time, number of times a parameter tried to go beyond the boundaries, the current best-fitting values, and corresponding χ^2, like this:

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
  Multi-Core Markov-Chain Monte Carlo (MC3).
  Version 1.1.20.
  Copyright (c) 2015-2016 Patricio Cubillos and collaborators.
  MC3 is open-source software under the MIT license (see LICENSE).
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Start MCMC chains  (Tue Jan  5 13:11:22 2016)

...

[::        ]  20.0% completed  (Tue Jan  5 13:11:22 2016)
Out-of-bound Trials:
 [0 0 0]
Best Parameters:   (chisq=87.5664)
[ 2.81119952 -2.33026943  0.48622898]

...
```

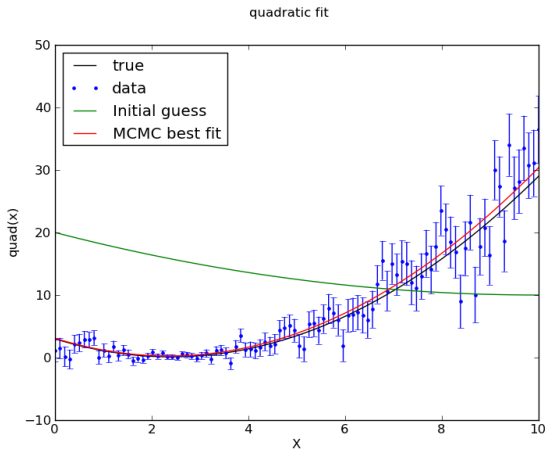
At the end of the MCMC run, `MC3` will display a summary of the MCMC sample, best-fitting parameters, uncertainties, mean values, and statistics:

```
Fin, MCMC Summary:
------------------
 Burned in iterations per chain:   100
 Number of iterations per chain:  3000
 MCMC sample size:               29000
 Acceptance rate:   39.39%

 Best-fit params    Uncertainties   Signal/Noise       Sample Mean
   2.8111995e+00    3.8625328e-01           7.28     2.8167688e+00
  -2.3302694e+00    2.2233506e-01          10.48    -2.3308174e+00
   4.8622898e-01    2.7225910e-02          17.86     4.8622772e-01

 Best-parameter's chi-squared:      87.5664
 Bayesian Information Criterion:   101.3819
 Reduced chi-squared:                0.9027
 Standard deviation of residuals:  2.5201
```

Note: More information will be displayed, depending on the MCMC configuration (see the *Tutorial*).

The user has the option to generate the best-fitting, trace, and posterior MCMC plots (these plots can also be generated automatically with the MCMC run):

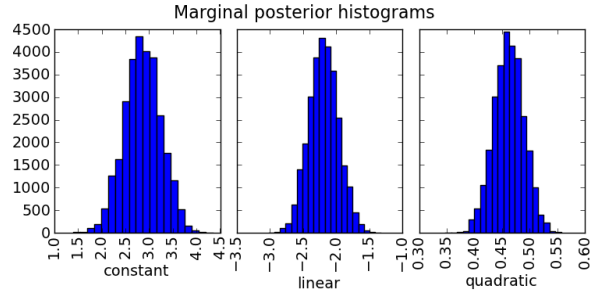
```
y0 = quad(params, x)  # Initial guess values
y1 = quad(bp,     x)  # MCMC best fitting values

plt.figure(10)
plt.clf()
plt.plot(x, y, "-k", label='true')
plt.errorbar(x, data, yerr=uncert, fmt=".b", label='data')
plt.plot(x, y0, "-g", label='Initial guess')
plt.plot(x, y1, "-r", label='MCMC best fit')
plt.legend(loc="best")
plt.xlabel("X")
plt.ylabel("quad(x)")
```

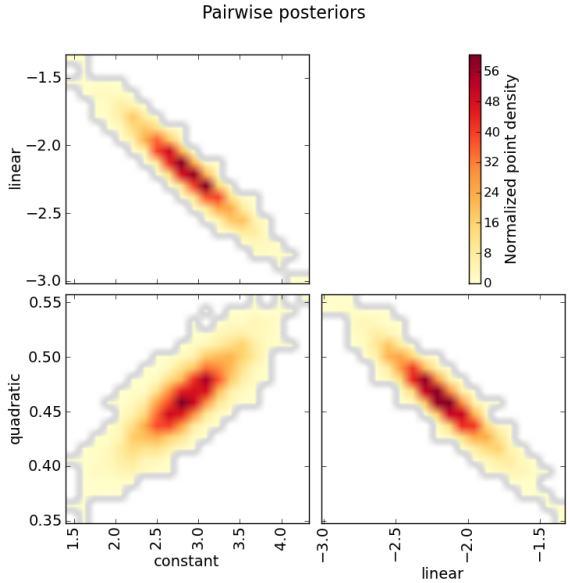


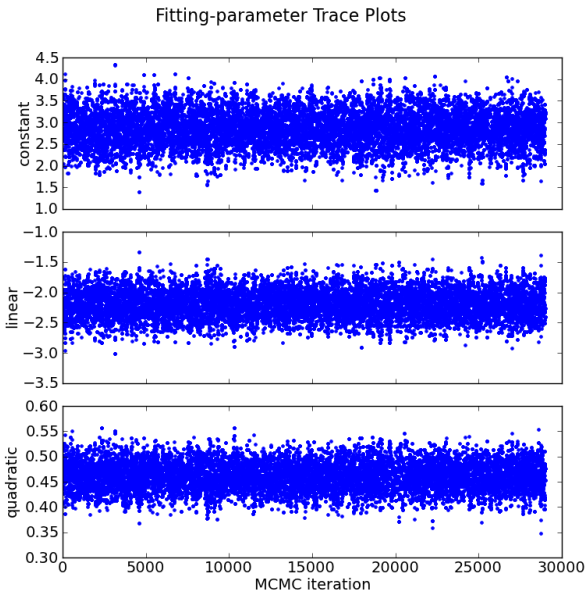
The mcplots module of `MC3` provides the functions to plot the parameter trace and posteriors:

```
# Import the mcplots module:
import mcplots as mp
# Plot trace plot:
parname = ["constant", "linear", "quadratic"]
```

```
mp.trace(allp, title="Fitting-parameter Trace Plots", parname=parname,
         savefile="quad_trace.png")

# Plot pairwise posteriors:
mp.pairwise(allp, title="Pairwise posteriors", parname=parname,
         savefile="quad_pairwise.png")

# Plot marginal posterior histograms:
mp.histogram(allp, title="Marginal posterior histograms", parname=parname,
         savefile="quad_hist.png")
```







Example 2 (Shell Run)

The following example (demo02) shows a basic MC3 MCMC run from the Shell prompt. To start, create a working directory to place the files and execute the program:

```
cd $topdir
mkdir run02
cd run02
```

Copy the demo files to run MC3 (configuration and data files):

```
cp $topdir/MCcubed/examples/demo02/* .
```

Call the MC3 executable, providing the configuration file as command-line argument:

```
mpirun $topdir/MCcubed/src/mccubed.py -c MCMC.cfg
```

5.2 Tutorial

This tutorial describes the available options when running an MCMC with MC3. As said before, the MCMC can be run from the shell prompt or through a function call in the Python interpreter.

Argument Inputs

When running from the shell, the arguments can be input as command-line arguments. To see all the available options, run:

```
./mccubed.py --help
```

When running from a Python interactive session, the arguments can be input as function arguments. To see the available options, run:

```
import mccubed as mc3
help(mc3.mcmc)
```

Additionally, wether you are running the MCMC from the shell or from the interpreter, the arguments can be input through a configuration file.

Configuration Files

MC3's configuration file follows the ConfigParser format. The following code block shows an example for an MC3 configuration file:

```
# Comment lines (like this one) are allowed and ignored
# Strings don't need quotation marks
[MCMC]
# DEMC general options:
nsamples  = 1e5
burnin    = 100
nchains   = 10
walk      = demc
mpi       = True
# Fitting function options:
func      = quad quadratic ../../examples/example01
params    = pars_ex02.dat
indparams = indp_ex02.dat
# The data and uncertainties:
data      = data_ex02.dat
```

MCMC-run Configuration

The following example describes the basic MCMC argument configuration. The following sub-sections make up a script meant to be run from the Python interpreter. The complete example script is located at tutorial01.

Data and Data Uncertainties

The `data` argument (required) defines the dataset to be fitted. This argument can be either a 1D float ndarray or the filename (a string) where the data array is located.

The `uncert` argument (required) defines the 1σ uncertainties of the `data` array. This argument can be either a 1D float ndarray (same length of `data`) or the filename where the data uncertainties are located.

```
# Create a synthetic dataset using a quadratic polynomial curve:
x  = np.linspace(0, 10, 100)          # Independent model variable
p0 = 3, -2.4, 0.5                     # True-underlying model parameters
y  = quad(p0, x)                      # Noiseless model
uncert = np.sqrt(np.abs(y))           # Data points uncertainty
error  = np.random.normal(0, uncert)  # Noise for the data
data   = y + error                    # Noisy data set
```

Note: See the *Data* Section below to find out how to set `data` and `uncert` as a filename.

Random Walk

The `walk` argument (optional) defines which random-walk algorithm will use the MCMC:

```
# Choose between: {'demc' or 'mrw'}:
walk    = 'demc'
```

If `walk = mrw`, MC3 will use the classical Metropolis-Hastings algorithm with Gaussian proposal distributions. I.e., in each iteration and for each parameter, θ, the MCMC will propose jumps, drawn from Gaussian distributions centered at the current value, θ_0, with a standard deviation, σ, given by the values in the `stepsize` argument:

$$q(\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\theta-\theta_0)^2}{2\sigma^2}\right) \tag{1}$$

If `walk = demc` (default value), MC3 will use Differential-Evolution MCMC algorithm (for further reading, see *[terBraak2006]*).

Modeling Function

The `func` argument (required) defines the parameterized modeling function. The user can set `func` either as a callable, e.g.:

```
# Define the modeling function as a callable:
sys.path.append("./../models/")
from quadratic import quad
func = quad
```

or as a tuple of strings pointing to the modeling function, e.g.:

```
# A three-elements tuple indicates the function name, the module
# name (without the '.py' extension), and the path to the module.
func = ("quad", "quadratic", "./../models/")

# Alternatively, if the module is already within the scope of the
# python-path, the user can set func with a two-elements tuple:
sys.path.append("./../models/")
func = ("quad", "quadratic")
```

Note: Important!

The only requirement for the modeling function is that its arguments follow the same structure of the callable in `scipy.optimize.leastsq`, i.e., the first argument contains the list of fitting parameters.

The `indparams` argument (optional) packs any additional argument that the modeling function may require:

```
# indparams contains additional arguments of func (if necessary). Each
# additional argument is an item in the indparams tuple:
indparams = [x]
```

Note: Even if there is only one additional argument to `func`, indparams must be defined as a tuple (as in the example above). Eventually, the modeling function could be called with the following command:

```
model = func(params, *indparams)
```

Fitting Parameters

The `params` argument (required) contains the initial-guess values for the model fitting parameters. The `params` argument must be a 1D float ndarray.

```
# Array of initial-guess values of fitting parameters:
params   = np.array([ 20.0,  -2.0,   0.1])
```

The `pmin` and `pmax` arguments (optional) set the lower and upper boundaries explored by the MCMC for each fitting parameter.

```
# Lower and upper boundaries for the MCMC exploration:
pmin     = np.array([-10.0, -20.0, -10.0])
pmax     = np.array([ 40.0,  20.0,  10.0])
```

If a proposed step falls outside the set boundaries, that iteration is automatically rejected. The default values for each element of `pmin` and `pmax` are `-np.inf` and `+np.inf`, respectively. The `pmin` and `pmax` arguments must have the same size of `params`.

Stepsize, Fixed, and Shared Paramerers

The `stepsize` argument (optional) provides multiple uses. The stepsize is a 1D float ndarray, where each element correspond to one of the fitting parameters. When `walk='mrw'`, `stepsize` sets the standard deviation, σ, of the Gaussian proposal jump for the given parameter, see Eq. (1). When `walk='demc'`, `stepsize` sets the standard-deviation jump **only** of the initial jump (which is used to initialize the chains).

```
# stepsize determines the standard deviation of the proposal Gaussian function:
# For Metropolis Random Walk, the Gaussian function draws the parameter
# proposals for each iteration.
# For Differential Evolution MCMC, the Gaussian function draws the
# starting values of the chains about the initial-guess values.
stepsize = np.array([  1.0,   0.5,   0.1])
```

If the user wants to fix a parameter at the given initial-guess value, set the stepsize of the given parameter to 0.

If the user wants to share the same value for multiple parameters along the MCMC exploration (multiple parametes will), set the stepsize of the parameter equal to the negative index of the sharing parameter, e.g.:

```
# If I want the second, third, and fourth model parameters to share the same value:
stepsize = np.array([1.0, 3.0, -2, -2])
```

Note: Clearly, in the given example it doesn't make sense to share parameter values. However, for an eclipe model for example, one may want to share the ingress and egress times.

Parameter Priors

The `prior`, `priorlow`, and `priorup` arguments (optional) set the prior probability distributions of the fitting parameters. Each of these arguments is a 1D float ndarray.

```
# priorlow defines wether to use uniform non-informative (priorlow = 0.0),
# Jeffreys non-informative (priorlow < 0.0), or Gaussian prior (priorlow > 0.0).
# prior and priorup are irrelevant if priorlow <= 0 (for a given parameter)
prior    = np.array([ 0.0,  0.0,   0.0]) # The prior value
priorlow = np.array([ 0.0,  0.0,   0.0])
priorup  = np.array([ 0.0,  0.0,   0.0])
```

MC3 supports three types of priors. If `priorlow` is 0.0 (default) for a given parameter, the MCMC will apply a uniform non-informative prior:

$$p(\theta) = \frac{1}{\theta_{\rm max} - \theta_{\rm min}},$$

Note: This is appropriate when there is no prior knowledge of the value of θ.

If `priorlow` is less than 0.0 for a given parameter, the MCMC will apply a Jeffreys non-informative prior (uniform probability per order of magnitude):

$$p(\theta) = \frac{1}{\theta \ln(\theta_{\rm max}/\theta_{\rm min})},$$

Note: This is valid only when the parameter takes positive values. This is a more appropriate prior than a uniform prior when θ can take values over several orders of magnitude. For more information, see *[Gregory2005]*, Sec. 3.7.1.

Note: Practical note!

In practice, I have seen better results when one fits $\log(\theta)$ rather than θ with a Jeffreys prior.

Lastly, if `priorlow` is greater than 0.0 for a given parameter, the MCMC will apply a Gaussian informative prior:

$$p(\theta) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(\frac{-(\theta-\theta_p)^2}{2\sigma_p^2}\right),$$

were `prior` sets the prior value θ_p, and `priorlow` and `priorup` setting the lower and upper 1σ prior uncertainties, σ_p, of the prior (depending if the proposed value θ is lower or higher than θ_p).

Note: Note that, even when the parameter boundaries are not known or when the parameter is unbound, this prior is suitable for use in the MCMC sampling, since the proposed and current state priors divide out in the Metropolis ratio.

MCMC Chains Configuration

The following arguments set the MCMC chains configuration:

```
mpi      = True # Multiple or single-CPU run
numit    = 3e4  # Number of MCMC samples to compute
nchains  = 10   # Number of parallel chains
burnin   = 100  # Number of burned-in samples per chain
thinning =   1  # Thinning factor for outputs
```

The `mpi` argument (optional, boolean, default=False) determines if MC3 will run in multiple or a single CPU.

Note: In a multi-core run, MC3 will assign one CPU to each chain.

The `numit` argument (optional, float, default=1e5) sets the total number of samples to compute.

The `nchains` argument (optional, integer, default=10) sets the number of parallel chains to use. The number of iterations run for each chain will be `numit/nchains`.

Note: Even for single-core runs, the MCMC exploration will use `nchains` parallel chains.

The `burnin` argument (optional, integer, default=0) sets the number of burned-in (removed) iterations at the beginning of each chain.

The `thinning` argument (optional, integer, default=1) sets the chains thinning factor (discarding all but every `thinning`-th sample).

Note: Thinning is often unnecessary for a DEMC run, since this algorithm reduces significatively the sampling autocorrelation.

Optimization

The `leastsq` argument (optional, boolean, default=False) is a flag that indicates MC3 to run a least-squares optimization before running the MCMC. MC3 implements the Levenberg-Marquardt algorithm via the `scipy.optimize.leastsq` function.

Note: The parameter boundaries, fixed and shared-values, and priors setup will apply for the minimization.

The `chisqscale` argument (optional, boolean, default=False) is a flag that indicates MC3 to scale the data uncertainties to force a reduced χ^2 equal to 1. The scaling applies by multiplying all uncertainties by a common scale factor.

```
leastsq    = True   # Least-squares minimization prior to the MCMC
chisqscale = False  # Scale the data uncertainties such red.chisq = 1
```

Gelman-Rubin Convergence Test

The `grtest` argument (optional, boolean, default=False) is a flag that indicates MC3 to run the Gelman-Rubin convergence test for the MCMC sample of fitting parameters. Values substantially larger than 1 indicate non-convergence. See *[GelmanRubin1992]* for further information.

The `grexit` argument (optional, boolean, default=False) is a flag that allows the MCMC to stop if the Gelman-Rubin test returns values below 1.01 for all parameter, two consecutive times.

```
grtest  = True   # Calculate the GR convergence test
grexit  = False  # Stop the MCMC after two successful GR
```

Note: The Gelman-Rubin test is computed every 10% of the MCMC exploration.

Wavelet-Likelihood MCMC

The `wlike` argument (optional, boolean, default=False) allows MC3 to implement the Wavelet-based method to estimate time-correlated noise. When using this method, the used must append the three additional fitting parameters $(\gamma, \sigma_r, \sigma_w)$ from Carter & Winn (2009) to the end of the `params` array. Likewise, add the correspoding values to the `pmin`, `pmax`, `stepsize`, `prior`, `priorlow`, and `priorup` arrays. For further information see *[CarterWinn2009]*.

```
wlike = False  # Use Carter & Winn's Wavelet-likelihood method.
```

File Outputs

The following arguments set the output files produced by MC3:

```
logfile    = 'MCMC.log'          # Save the MCMC screen outputs to file
savefile   = 'MCMC_sample.npy'   # Save the MCMC parameters sample to file
savemodel  = 'MCMC_models.npy'   # Save the MCMC evaluated models to file
plots      = True                # Generate best-fit, trace, and posterior plots
rms        = False               # Compute and plot the time-averaging test
```

The `logfile` argument (optional, string, default=None) sets the-text file name where to store MC3's screen output.

The `savefile` and `savemodel` arguments (optional, string, default=None) set the file names where to store the MCMC parameters sample and evaluated models. MC3 saves the files as three-dimensional `.npy` binary files, The first dimension corresponds to the chain index, the second dimension the fitting parameter or data point (for `savefile` and `savemodel`, respectively), and the third dimension the iteration number. The files can be read with the `numpy.load()` function.

The `plots` argument (optional, boolean, default=False) is a flag that indicates MC3 to generate and store the data (along with the best-fitting model) plot, the MCMC-chain trace plot for each parameter, and the marginalized and pair-wise posterior plots.

The `rms` argument (optional, boolean, default=False) is a flag that indicates MC3 to compute the time-averaging test for time-correlated noise and generate a rms-vs-binsize plot. For further information see *[Winn2008]*.

Returned Values

When run from a pyhton interactive session, MC3 will return two arrays: `posterior` a 2D array containing the burned-in, thinned MCMC sample of the parameters posterior distribution (with dimensions [nparameters, nsamples]); and `bestp`, a 1D array with the best-fitting parameters.

```
# Run the MCMC:
posterior, bestp = mc3.mcmc(data=data, uncert=uncert, func=func, indparams=indparams,
                params=params, pmin=pmin, pmax=pmax, stepsize=stepsize,
                prior=prior, priorlow=priorlow, priorup=priorup,
                leastsq=leastsq, chisqscale=chisqscale, mpi=mpi,
                numit=numit, nchains=nchains, walk=walk, burnin=burnin,
                grtest=grtest, grexit=grexit, wlike=wlike, logfile=logfile,
                plots=plots, savefile=savefile, savemodel=savemodel, rms=rms)
```

Resume an MC3 Run

TBD

Inputs from Files

TBD

As said in the help description, the data, uncert, indparams, params, # pmin, pmax, stepsize, prior, priorlow, and priorup arrays can be # read from a text file. In this case, set the argument to be the file name.

Each line in the 'indparams' file must contain one element of the indparams # list, the values separated by (one or more) empty spaces. # The other files must contain one array value per line (i.e., column-wise).

Furthermore, the 'data' file can also contain the uncert array (as a second # column, values separated by a empty space). # Likewise, the 'params' file can contain the pmin, pmax, stepsize, prior, # priorlow, and priorup arrays (as many or as few, provided that they are # written in columns in that precise order).

The array arguments of mc3.mcmc can be read from text/binary files. In this case, set the argument to the file name. The values of the `data`, `uncert`, and `indparams` must be stored in binary format, whereas the `params`, `pmin`, `pmax`, `stepsize`, `prior`, `priorlow`, and `priorup` arrays must be column-wise stored in plain ASCII format.

Furthermore, the `data` and the `uncert` arrays can be stored into a single file. Likewise, the `params`, `pmin`, `pmax`, `stepsize`, `prior`, `priorlow`, and `priorup` arrays can be stored into a single file, having white-space separated columns for each argument. The user can append as many or as few of these columns as long as they follow that exact order (e.g., you can't include the priors if you don't include the stepsize).

The `mcutils` module provides the functions `writedata` and `writebin` to easily create these files. Accordingly, the `read2array` and `readbin` function read these files. For example:

```
# Store binary array:
mu.writebin([data],          'data_ex01.dat')
# Store multiple arrays in binary format:
mu.writebin([data, uncert], 'data_ex01.dat')
mu.writedata(indparams, 'indp_ex01.dat')
# Store ASCII arrays:
mu.writedata([params, pmin, pmax, stepsize], 'pars_ex01.dat')

# To run MCMC, set the arguments to the file names:
data      = 'data_ex01.dat'
params    = 'pars_ex01.dat'
indparams = 'indp_ex01.dat'
# Run MCMC:
allp, bp = mc3.mcmc(data=data, func=func, indparams=indparams,
                    params=params,
                    numit=numit, nchains=nchains, walk=walk, grtest=grtest,
                    leastsq=leastsq, chisqscale=chisqscale,
                    burnin=burnin, plots=plots, savefile=savefile,
                    savemodel=savemodel, mpi=mpi)
```

Empty or comment lines are allowed (and ignored by the reader). A valid params file look like this:

```
#       params            pmin            pmax        stepsize
           10             -10              60               1
           16             -20              20             0.5
         -1.8             -10              10             0.1
```

If data, uncert, params, pmin, pmax, stepsize, prior, priorlow, or priorup are set as filenames, the file must contain one value per line.

Data

TDB

For simplicity, the data file can hold both data and uncert arrays. In this case, each line contains one value from each array per line, separated by an empty-space character.

Data can be in a file.

Fitting Parameters

TDB

Similarly, params can hold: params, pmin, pmax, stepsize, priorlow, and priorup. The file can hold as few or as many array as long as they are provided in that exact order.

5.3 License

References

[CarterWinn2009] Carter & Winn (2009): Parameter Estimation from Time-series Data with Correlated Errors: A Wavelet-based Method and its Application to Transit Light Curves

[GelmanRubin1992] Gelman & Rubin (1992): Inference from Iterative Simulation Using Multiple Sequences

[Gregory2005] Gregory (2005): Bayesian Logical Data Analysis for the Physical Sciences

[terBraak2006] ter Braak (2006): A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution

[Winn2008] Winn et al. (2008): The Transit Light Curve Project. IX. Evidence for a Smaller Radius of the Exoplanet XO-3b