MC3 Documentation

Release 1.2.4

Patricio Cubillos

May 02, 2016

Contents

Author Patricio Cubillos and collaborators (see *Collaborators*)

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 May 02, 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 Collaborators

All of these people have made a direct or indirect contribution to `MCcubed`, and in many instances have been fundamental in the development of this package.

- Patricio Cubillos (UCF, IWF) patricio.cubillos[at]oeaw.ac.at
- Joseph Harrington (UCF)
- Nate Lust (UCF)
- AJ Foster (UCF)
- Madison Stemm (UCF)
- Kevin Stevenson (UCF)
- Chris Campo (UCF)
- Matt Hardin (UCF)
- Ryan Hardy (UCF)

3 Documentation

3.1 Getting Started

System Requirements

MC3 (version 1.2) is known to work 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, keep track of the folder where you are putting MC3:

```
topdir=`pwd`
git clone https://github.com/pcubillos/MCcubed
```

Compile

Compile the C-extensions of the package:

```
cd $topdir/MCcubed/
make
```

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

```
make clean
```

Example 1 (Interactive)

The following example (demo01) shows a basic MCMC run with MC3 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/")
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:
posterior, bestp = 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; for example:

```
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
  Multi-Core Markov-Chain Monte Carlo (MC3).
  Version 1.2.0.
  Copyright (c) 2015-2016 Patricio Cubillos and collaborators.
  MC3 is open-source software under the MIT license (see LICENSE).
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Start MCMC chains  (Fri Feb  5 10:45:17 2016)

[:          ]  10.0% completed  (Fri Feb  5 10:45:17 2016)
Out-of-bound Trials:
 [0 0 0]
Best Parameters:   (chisq=111.0541)
[ 3.79473869 -2.73050517  0.51636233]

...

[::::::::::] 100.0% completed  (Fri Feb  5 10:45:18 2016)
Out-of-bound Trials:
 [0 0 0]
Best Parameters:   (chisq=111.0449)
[ 3.77284276 -2.72330815  0.51634107]

Fin, MCMC Summary:
------------------
 Burned in iterations per chain:   100
```

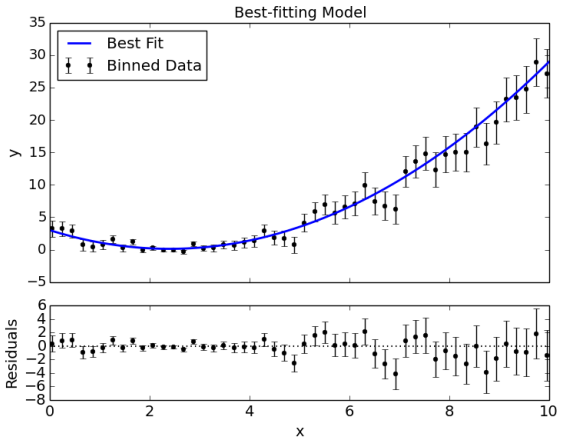
```
 Number of iterations per chain:  3000
 MCMC sample size:               29000
 Acceptance rate:   39.04%

 Best-fit params   Uncertainties        S/N       Sample Mean   Note
   3.7728428e+00   3.8407332e-01       9.82    3.7694995e+00
  -2.7233081e+00   2.1964109e-01      12.40   -2.7232216e+00
   5.1634107e-01   2.6891868e-02      19.20    5.1641806e-01

 Best-parameter's chi-squared:     111.0449
 Bayesian Information Criterion:   124.8604
 Reduced chi-squared:                1.1448
 Standard deviation of residuals:  2.93518
```

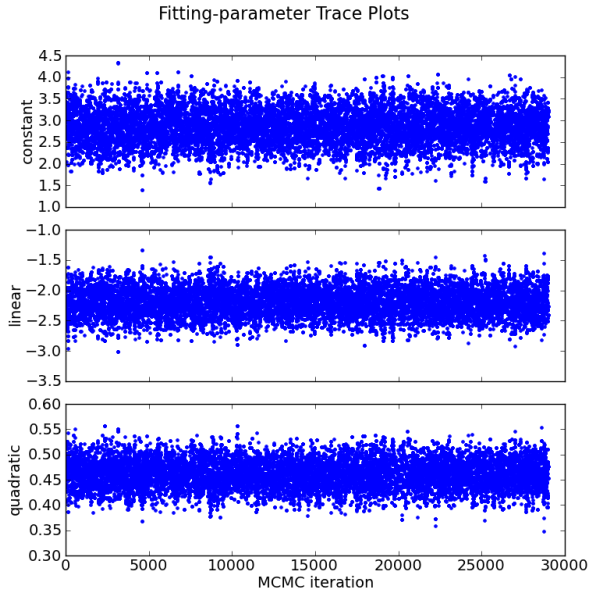
At the end of the MCMC run, `MC3` displays a summary of the MCMC sample, best-fitting parameters, uncertainties, mean values, and other statistics.

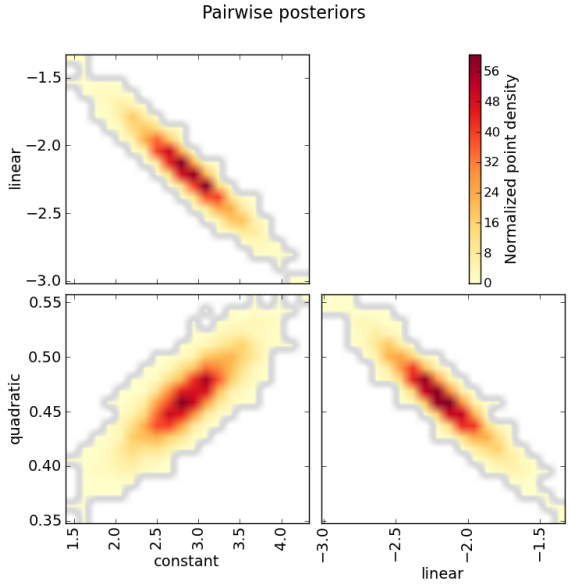
Note: More information will be displayed, depending on the MCMC configuration (see the *MCMC Tutorial*).

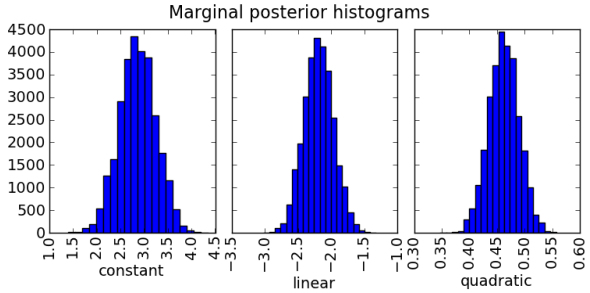
Additionally, the user has the option to generate several plots of the MCMC sample: the best-fitting model and data curves, parameter traces, and marginal and pair-wise posteriors (these plots can also be generated automatically with the MCMC run). The plots sub-package provides the plotting functions:

```
# Plot best-fitting model and binned data:
mc3.plots.modelfit(data, uncert, x, y, title="Best-fitting Model",
                   savefile="quad_bestfit.png")
# Plot trace plot:
parname = ["constant", "linear", "quadratic"]
mc3.plots.trace(posterior, title="Fitting-parameter Trace Plots",
                parname=parname, savefile="quad_trace.png")

# Plot pairwise posteriors:
mc3.plots.pairwise(posterior, title="Pairwise posteriors", parname=parname,
                   savefile="quad_pairwise.png")

# Plot marginal posterior histograms:
mc3.plots.histogram(posterior, title="Marginal posterior histograms",
                    parname=parname, savefile="quad_hist.png")
```









Note: These plots can also be automatically generated along with the MCMC run (see File Outputs).

Example 2 (Shell Run)

The following example (demo02) shows a basic 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/MCcubed/mccubed.py -c MCMC.cfg
```

Note: If you don't have MPI or dont want to use it, make the previous call as:

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

3.2 MCMC 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 (and strongly recommended), whether you are running the MCMC from the shell or from the interpreter, the arguments can be input through a configuration file.

Configuration Files

The MC3 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:
numit      = 1e5
burnin     = 100
nchains    = 10
walk       = demc
```

```
mpi        = True
# Fitting function:
func       = quad quadratic ../MCcubed/examples/models
# Model inputs:
params     = params.dat
indparams  = indp.npz
# The data and uncertainties:
data       = data.npz
```

MCMC-run Configuration

This 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.

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` arrays must have the same size of `params`.

Stepsize, Fixed, and Shared Paramerers

The `stepsize` argument (optional) is a 1D float ndarray, where each element correspond to one of the fitting parameters. The stepsize has multiple uses. When `walk='mrw'` (see *Random Walk* section), `stepsize` sets the standard deviation, σ, of the Gaussian proposal jump for the given parameter, (see Eq. (4)). When `walk='demc'`, `stepsize` sets the standard-deviation jump **only** of the initial jump (which is used to initialize the chains).

```
stepsize = np.array([  1.0,   0.5,   0.1])
```

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

If you want 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 whether 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])
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 a value of `priorlow` is 0.0 (default) for a given parameter, the MCMC will apply a uniform non-informative prior:

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

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):

$$p(\theta) = \frac{1}{\theta \ln(\theta_{\rm max}/\theta_{\rm min})}, \tag{2}$$

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:

$$p(\theta) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(\frac{-(\theta-\theta_p)^2}{2\sigma_p^2}\right), \tag{3}$$

where `prior` sets the prior value θ_p, and `priorlow` and `priorup` set 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.

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:

$$q(\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\theta-\theta_0)^2}{2\sigma^2}\right) \tag{4}$$

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

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. Additionaly, the main MCMC central hub will use one CPU. Thus, the total number of CPUs used is `nchains + 1`.

Normally, if you ask `MC3` to use more CPUs than the number of CPUs available, the code will be much much slower.

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-CPU runs, the MCMC algorithm 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 a previous MC3 Run

TBD

Inputs from Files

The `data`, `uncert`, `indparams`, `params`, `pmin`, `pmax`, `stepsize`, `prior`, `priorlow`, and `priorup` input arrays can be optionally be given as input file. Furthermore, multiple input arguments can be combined into a single file.

Data

The `data`, `uncert`, and `indparams` inputs can be provided as binary `numpy .npz` files. `data` and `uncert` can be stored together into a single file. An `indparams` input file contain the list of independent variables (must be a list, even if there is a single independent variable).

The `utils` sub-package of `MC3` provide utility functions to save and load these files. The `preamble.py` file in demo02 gives an example of how to create `data` and `indparams` input files:

```
# Import the necessary modules:
import sys
import numpy as np

# Import the modules from the MCcubed package:
sys.path.append("../MCcubed/")
import MCcubed as mc3
sys.path.append("../MCcubed/examples/models/")
from quadratic import quad


# Create a synthetic dataset using a quadratic polynomial curve:
x  = np.linspace(0.0, 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

# data.npz contains the data and uncertainty arrays:
```

```
mc3.utils.savebin([data, uncert], 'data.npz')
# indp.npz contains a list of variables:
mc3.utils.savebin([x], 'indp.npz')
```

Fitting Parameters

The `params`, `pmin`, `pmax`, `stepsize`, `prior`, `priorlow`, and `priorup` inputs can be provided as plain ASCII files. For simplycity all of these input arguments can be combined into a single file.

In the `params` file, each line correspond to one model parameter, whereas each column correspond to one of the input array arguments. This input file can hold as few or as many of these argument arrays, as long as they are provided in that exact order. 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
```

Alternatively, the `utils` sub-package of `MC3` provide utility functions to save and load these files:

```
params   = [ 10,   16, -1.8]
pmin     = [-10,  -20, -10]
pmax     = [ 60,   20,  10]
stepsize = [  1,  0.5,  0.1]

# Store ASCII arrays:
mc3.utils.saveascii([params, pmin, pmax, stepsize], 'params.txt')
```

Then, to run the MCMC simply provide the input file names to the `MC3` routine:

```
# To run MCMC, set the arguments to the file names:
data      = 'data.npz'
indparams = 'indp.npz'
params    = 'params.txt'
# Run MCMC:
posterior, bestp = 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)
```

References

3.3 Optimization Tutorial

The `MCcubed.fit` package provides the `modelfit` routine for model-fitting optimization through the least-squares Levenberg-Marquardt algorith.

`modelfit` is a wrapper of `scipy.optimize.leastsq` with additional features, including Gaussian-parameter priors, parameter boundaries, and sharing and fixing parameters. All `modelfit` arguments are identical to those of the MCMC.

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.

Modeling Function

The `func` argument (required) defines the parameterized modeling function. 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.

If func requires additional arguments, they can be provided through the `indparams` argument (see *Independent Parameters*). Eventually, the modeling function could be called with the following command:

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

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.

Independent Parameters

The `indparams` argument (optional) is a tuple (or list) that packs any additional arguments required by `func`. Even if `indparams` consists of a single variable, it must be defined as a list or tuple.

Stepsize: Fixed, and Shared Paramerers

The `stepsize` argument (optional) is a 1D float ndarray, where each element correspond to one of the fitting parameters. For optimization, `stepsize` determines the free, fixed, and shared parameters. If the stepsize is positive (irrelevant of the value), the parameter is a free fitting parameter.

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

To copy the value from another parameter (free or fixed), set the stepsize equal to the negative index of the sharing parameter.

Note: Consider that in this case, contrary to Python standards, the indexing starts counting from one instead of zero. Thus, for example, to share a value with that of the first parameter, set the parameter's stepsize to -1.

Parameter Boundaries

The `pmin` and `pmax` arguments (optional) are 1D float ndarrays that set the lower and upper boundaries explored by the minimizer for each fitting parameter (same size of `params`). The default values for each element of `pmin` and `pmax` are `-np.inf` and `+np.inf`, respectively.

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.

If a value of `priorlow` is 0.0 (default) for a given parameter, the MCMC will apply a uniform non-informative prior:

$$p(\theta) = \frac{1}{\theta_{\rm max} - \theta_{\rm min}}, \tag{5}$$

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

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

$$p(\theta) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(\frac{-(\theta-\theta_p)^2}{2\sigma_p^2}\right), \tag{6}$$

where `prior` sets the prior value θ_p, and `priorlow` and `priorup` set 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).

Outputs

`modelfit` returns four variables:

- `chisq` (float) is the best-fitting chi-square value.
- `bestparams` (1D float ndarray) is the array of best-fitting parameters, including fixed and shared parameters.
- **`bestmodel` (1D float ndarray) is the best-fitting model found, i.e.,** `func(bestparams, *indparams)`.
- `lsfit` (list) is `scipy.optimize.leastsq`'s full_output return.

Example

```
import sys
import MCcubed as mc3  # Add path to mc3 if necessary

# 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

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

# Get a modeling function (quadractic polynomial):
#sys.path.append("./MCcubed/models/")  # Set the appropriate path
import quadratic as q
func = q.quad
```

```
# indparams contains additional arguments of func (besides params):
indparams = [x]

params   = np.array([  1.0,   0.0,   0.3])
stepsize = np.array([  1.0,   1.0,   1.0])  # All model parameters free
pmin     = np.array([-10.0, -20.0, -10.0])  # Lower param boundaries
pmax     = np.array([ 40.0,  20.0,  10.0])  # Upper param boundaries
prior    = np.array([  0.0,   0.0,   0.0])
priorlow = np.array([  0.0,   0.0,   0.0])  # Flat priors
priorup  = np.array([  0.0,   0.0,   0.0])
# prior and priorup are irrelevant if priorlow == 0 (for a given parameter)

chisq, bestp, bestmodel, lsfit = mc3.fit.modelfit(params, q.quad,
                                 data, uncert, indparams=indparams,
                                 stepsize=stepsize, pmin=pmin, pmax=pmax,
                                 prior=prior, priorlow=priorlow, priorup=priorup)
```

3.4 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)

MC3 is open-source open-development software under the MIT *License*.

Thank you for using MC3!

5 Documentation for Previous Releases

- MC3 version 1.1.

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