MC3 Documentation

Release 2.3.25

Patricio Cubillos

Aug 11, 2019

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 Aug 11, 2019

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,
  - Differential-Evolution MCMC (DEMC), or
  - DEMCzs (Snooker).

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.

Note: `MC3` works in both Python2.7 and Python3!

---

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)
- Tom Loredo (Cornell)
- Kevin Stevenson (UCF)
- Chris Campo (UCF)
- Matt Hardin (UCF)
- Ryan Hardy (UCF)
- Monika Lendl (IWF)
- Ryan Challener (UCF)
- Michael Himes (UCF)

3 Documentation

3.1 Getting Started

3.1.1 System Requirements

MC3 (version 2.2) is known to work on Unix/Linux (Ubuntu) and OSX (10.9+) machines, with the following software:

- Python (version 2.7+ or 3.4+)
- Numpy (version 1.8.2+)
- Scipy (version 0.17.1+)
- Matplotlib (version 1.3.1+)

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

3.1.2 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
```

3.1.3 Compile

To compile the C-extensions of the package run:

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

To compile the documentation of the package, run:

```
cd $topdir/MCcubed/docs
make latexpdf
```

A pdf version of this documentation will be available at `$topdir/MCcubed/docs/latex/MC3.pdf`. To remove the program binaries, run:

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

3.1.4 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

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, 1000)          # 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([10.0, -2.0, 0.1])  # Initial guess of fitting params.
stepsize = np.array([0.03, 0.03, 0.05])

# Run the MCMC:
bestp, CRlo, CRhi, stdp, posterior, Zchain = mc3.mcmc(data, uncert,
    func=quad, indparams=[x], params=params, stepsize=stepsize,
    nsamples=1e5, burnin=1000)
```

The code will return the best-fitting values (`bestp`), the lower and upper boundaries of the 68%-credible region (`CRlo` and `CRhi`, with respect to `bestp`), the standard deviation of the marginal posteriors (`stdp`), the posterior sample (`posterior`), and the chain index for each posterior sample (`Zchain`).

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 2.3.20.
  Copyright (c) 2015-2018 Patricio Cubillos and collaborators.
  MC3 is open-source software under the MIT license (see LICENSE).
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

Yippee Ki Yay Monte Carlo!
Start MCMC chains  (Sun Nov  4 16:20:40 2018)

[:         ]  10.0% completed  (Sun Nov  4 16:20:42 2018)
Out-of-bound Trials:
[0 0 0]
Best Parameters: (chisq=1024.2992)
[ 3.0603825  -2.42108869  0.50075726]

...

[::::::::::] 100.0% completed  (Sun Nov  4 16:20:47 2018)
Out-of-bound Trials:
[0 0 0]
Best Parameters: (chisq=1024.2772)
[ 3.0679888  -2.4229654   0.50064008]

Fin, MCMC Summary:
------------------
  Total number of samples:            100002
  Number of parallel chains:               7
  Average iterations per chain:        14286
  Burned-in iterations per chain:       1000
  Thinning factor:                         1
  MCMC sample size (thinned, burned):  93002
  Acceptance rate:   26.76%

Param name     Best fit   Lo HPD CR   Hi HPD CR        Mean     Std dev       S/N
---------- ---------------------------------- -------------------- ----------
Param 1      3.0577e+00 -1.2951e-01  1.1875e-01  3.0555e+00 1.2384e-01      24.7
Param 2     -2.4055e+00 -6.7695e-02  7.5366e-02 -2.4033e+00 7.1281e-02      33.7
Param 3      4.9933e-01 -8.9207e-03  8.5756e-03  4.9902e-01 8.7305e-03      57.2

  Best-parameter's chi-squared:     1024.2772
  Bayesian Information Criterion:   1045.0004
  Reduced chi-squared:                 1.0274
  Standard deviation of residuals:  2.78898
```

At the end of the MCMC run, `MC3` displays a summary of the MCMC sample, best-fitting parameters, credible-region boundaries, posterior mean and standard deviation, among other statistics.

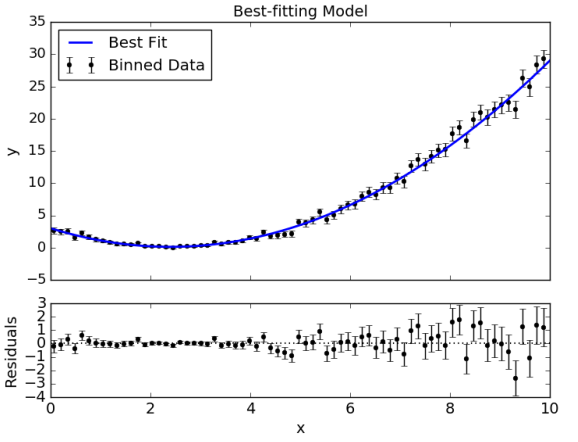
Note: More information will be displayed, depending on the MCMC configuration (see the *MCMC Tutorial*).

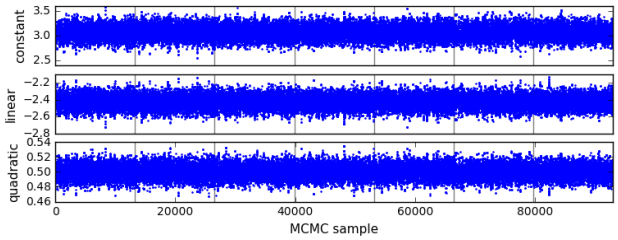
Additionally, the user has the option to generate several plots of the MCMC sample: the best-fitting model and data curves, parameter traces, and marginal and pair-wise posteriors (these plots can also be generated automatically with the MCMC run by setting `plots=True`). The plots sub-package provides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

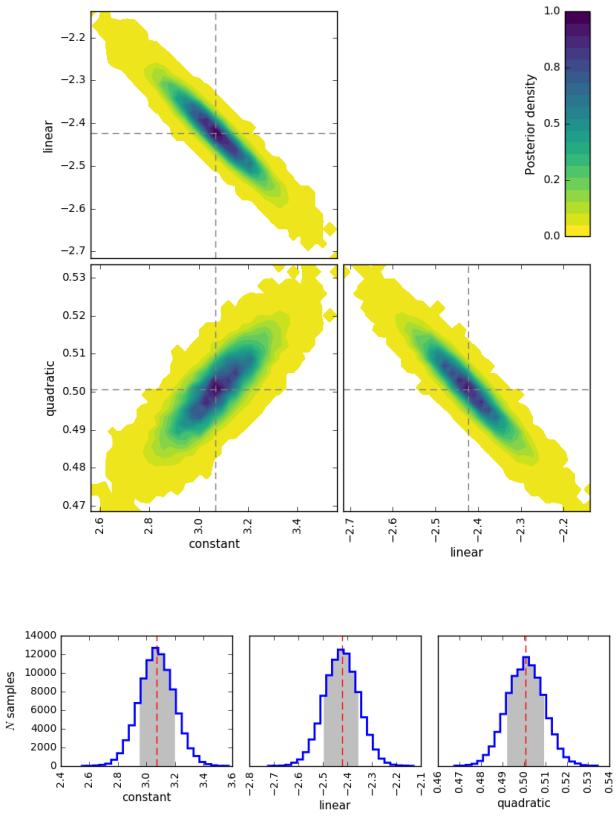
```
# Plot best-fitting model and binned data:
mc3.plots.modelfit(data, uncert, x, y, savefile="quad_bestfit.png")
# Plot trace plot:
pnames = ["constant", "linear", "quadratic"]
mc3.plots.trace(posterior, Zchain, pnames=pnames, savefile="quad_trace.png")

# Plot pairwise posteriors:
mc3.plots.pairwise(posterior, pnames=pnames, bestp=bestp,
   savefile="quad_pairwise.png")

# Plot marginal posterior histograms (with 68% highest-posterior-density credible␣
↪regions):
mc3.plots.histogram(posterior, pnames=pnames, bestp=bestp, percentile=0.683,
    savefile="quad_hist.png")
```







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

3.1.5 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 (configuration and data files) to the run folder:

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

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

```
$topdir/MCcubed/mc3.py -c MCMC.cfg
```

3.1.6 Troubleshooting

There may be an error with the most recent version of the `multiprocessing` module (version 2.6.2.1). If the MCMC breaks with an "AttributeError: __exit__" error message pointing to a `multiprocessing` module, try installing a previous version of it with this shell command:

```
pip install --upgrade 'multiprocessing<2.6.2'
```

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.

3.2.1 Argument Inputs

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

```
./mc3.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.

3.2.2 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:
nsamples  = 1e5
burnin    = 1000
nchains   = 7
walk      = snooker
# Fitting function:
func      = quad quadratic ../MCcubed/examples/models
# Model inputs:
params    = params.dat
indparams = indp.npz
# The data and uncertainties:
data      = data.npz
```

3.2.3 MCMC Run

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.

Input Data

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:
import sys
import numpy as np
sys.path.append("../MCcubed/examples/models/")
from quadratic import quad

x  = np.linspace(0, 10, 1000)          # 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("../MCcubed/examples/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", "../MCcubed/examples/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("../MCcubed/examples/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([ 10.0,  -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 Parameters

The `stepsize` argument (required) is a 1D float ndarray, where each element correspond to one of the fitting parameters.

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

The stepsize has a dual purpose: (1) detemines the free, fixed, and shared parameters; and (2) determines the step size of proposal jumps.

To fix a parameter at the given initial-guess value, set the stepsize of the given parameter to 0. To share the same value for multiple parameters along the MCMC exploration, 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 current 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.

Additionally, when `walk='mrw'` (see *Random Walk* section), `stepsize` sets the standard deviation, σ, of the Gaussian proposal jump for the given parameter (see Eq. (5)).

Lastly, `stepsize` sets the standard deviation of the initial sampling for the chains (see *MCMC Config* section).

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.

Parameter Names

The `pnames` argument (optional) define the names of the model parametes to be shown in the scren output and figure labels. In figures, the names can use LaTeX syntax. The screen output will display up to 11 characters. Thus, the user can define the `texnames` argument (optional), display the appropriate syntax for screen output and figures, for example:

```
pnames   = ["log(alpha)", "beta", "Teff"]
texnames = [r"$\log(\alpha)$", r"$\beta$", r"$T_{\rm eff}$"]
```

If `texnames` is `None`, it defaults to `pnames`. If `pnames` is `None`, it defaults to `texnames`. If both arguments are `None`, they default to a generic `[Param 1, Param 2, ...]` list.

Random Walk

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

```
# Choose between: 'snooker', 'demc', or 'mrw':
walk = 'snooker'
```

The standard Differential-Evolution MCMC algorithm (`walk = 'demc'`, [terBraak2006]) proposes for each chain i in state $\mathbf{x}_i$:

$$\mathbf{x}^* = \mathbf{x}_i + \gamma(\mathbf{x}_{R1} - \mathbf{x}_{R2}) + \mathbf{e}, \tag{4}$$

where $\mathbf{x}_{R1}$ and $\mathbf{x}_{R2}$ are randomly selected without replacement from the population of current states without $\mathbf{x}_i$. This implementation adopts $\gamma = f_\gamma 2.38/\sqrt{2N_{\rm free}}$, and $\mathbf{e} \sim N(0, f_e\,\text{stepsize})$, with $N_{\rm free}$ the number of free parameters. The scaling factors are defaulted to $f_\gamma = 1.0$ and $f_e = 0.0$ (see *Fine-tuning*).

If `walk = 'snooker'` (default, recommended), `MC3` will use the DEMC-z algorithm with snooker propsals (see [BraakVrugt2008]).

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:

$$q(\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\theta-\theta_0)^2}{2\sigma^2}\right) \tag{5}$$

Note: For `walk=snooker`, an MCMC works well from 3 chains. For `walk=demc`, [terBraak2006] suggest using $2 * d$ chains, with d the number of free parameters.

I recommend any of the `snooker` or `demc` algorithms, as they are more efficient than most others MCMC random walks. From experience, when deciding between these two, consider that when the initial guess lays far from the lowest chi-square region, `snooker` seems to produce lower acceptance rates than ideal (which is solvable setting `leastsq=True`). On the other hand, `demc` is limited to a high number of chains when there is a high number of free parameters.

MCMC Config

The following arguments set the MCMC chains configuration:

```
nsamples =  1e5     # Number of MCMC samples to compute
nchains  =    7     # Number of parallel chains
nproc    =    7     # Number of CPUs to use for chains (default: nchains)
burnin   = 1000     # Number of burned-in samples per chain
thinning =    1     # Thinning factor for outputs

# Distribution for the initial samples:
kickoff = 'normal'  # Choose between: 'normal' or  'uniform'
hsize = 10          # Number of initial samples per chain
```

The `nsamples` argument (optional, float, default=1e5) sets the total number of samples to compute. The approximate number of iterations run for each chain will be `nsamples/nchains`.

The `nchains` argument (optional, integer, default=7) sets the number of parallel chains to use. The number of iterations run for each chain will be approximately `nsamples/nchains`.

`MC3` runs in multiple processors through the `mutiprocessing` package. The `nproc` argument (optional, integer, default= `nchains`) sets the number CPUs to use for the chains. Additionaly, the central MCMC hub will use one extra CPU. Thus, the total number of CPUs used is `nchains + 1`.

Note: If `nproc+1` is greater than the number of available CPUs in the machine (`nCPU`), `MC3` will set `nproc = nCPU-1`. To keep a good balance, I recommend setting `nchains` equal to a multiple of `nproc`.

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). To reduce the memory usage, when requested, only the thinned samples are stored (and returned).

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

To set the starting point of the MCMC chains, `MC3` draws samples either from a normal (default) or uniform distribution (determined by the `kickoff` argument). The mean and standard deviation of the normal distribution are set by the `params` and `stepsize` arguments, respectively. The uniform distribution is constrained between the `pmin` and `pmax` boundaries. The `hsize` argument determines the size of the starting sample. All draws from the initial sample are discarded from the returned posterior distribution.

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 (`lm=True`) via `scipy.optimize.leastsq` or Trust Region Reflective (`lm=False`) via `scipy.optimize.least_squares`.

Note: The parameter boundaries (for TRF only, see *Optimization Tutorial*), fixed and shared-values, and priors 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 1.0. The scaling applies by multiplying all uncertainties by a common scale factor.

```
leastsq    = True   # Least-squares minimization prior to the MCMC
lm         = True   # Choose Levenberg-Marquardt (True) or TRF algorithm (False)
chisqscale = False  # Scale the data uncertainties such that red. chisq = 1
```

Convergence

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 larger than 1.01 are indicative of non-convergence. See [GelmanRubin1992] for further information.

Additionally, the `grbreak` argument (optional, boolean, default=0.0) sets a convergence threshold to stop an MCMC when GR drops below `grbreak`. Reasonable values seem to be `grbreak` ~1.001–1.005. The default behavior is not to break (`grbreak=0.0`).

Lastly, the `grnmin` argument (optional, integer or float, default=0.5) sets a minimum number of valid samples (after burning and thinning) required for `grbreak`. If `grnmin` is an integer, require at least `grnmin` samples to break out of the MCMC. If `grnmin` is a float (in the range 0.0–1.0), require at least `grnmin` times the maximum possible number of valid samples to break out of the MCMC.

```
grtest  = True   # Calculate the GR convergence test
grbreak = 0.0    # GR threshold to stop the MCMC run
grnmin  = 0.5    # Minimum fraction or number of samples before grbreak
```

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.
```

Fine-tuning

The f_γ and f_e factors scale the DEMC proposal distributions.

```
fgamma   = 1.0  # Scale factor for DEMC's gamma jump.
fepsilon = 0.0  # Jump scale factor for DEMC's "e" distribution
```

The default $f_\gamma = 1.0$ value is set such that the MCMC acceptance rate approaches 25-40%. Therefore, most of the time, the user won't need to modify this. Only if the acceptance rate is very low, we recommend to set $f_\gamma < 1.0$. The f_e factor sets the jump scale for the $\mathbf{e}$ distribution, which has to have a small variance compared to the posterior. For further information see [terBraak2006].

File Outputs

The following arguments set the output files produced by MC3:

```
log       = 'MCMC.log'          # Save the MCMC screen outputs to file
savefile  = 'MCMC_sample.npz'   # Save the MCMC parameters sample to file
plots     = True                # Generate best-fit, trace, and posterior plots
rms       = False               # Compute and plot the time-averaging test
full_output = False             # Return the full posterior sample
chireturn = False               # Return chi-square statistics
```

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

The `savefile` arguments (optional, string, default = `None`) set the file names where to store the MCMC outputs into a `.npz` file, with keywords `bestp`, `CRlo`, `CRhi`, `stdp`, `meanp`, `Z`, `Zchain`, and `Zchisq`, `bestchisq`, `redchisq`, `chifactor`, `BIC`, and standard deviation of the residuals `sdr`. The files can be read with the `numpy.load()` function. See *Returned Values* and the description of `chireturn` below for details on the output values.

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 (see [Winn2008]).

The `full_output` argument (optional, bool, default = `False`) flags the code to return the full posterior sampling array (`Z`), including the initial and burnin samples. The posterior will still be thinned though.

If the `chireturn` argument (optional, bool, default = `False`) is `True`, `MC3` will return an additional tuple containing the chi-square stats: lowest χ^2 (`bestchisq`), $\chi^2_{\rm red}$ (`redchisq`), scaling factor to enforce $\chi^2_{\rm red} = 1$ (`chifactor`), and the Bayesian Information Criterion BIC (`BIC`).

Returned Values

When run from a pyhton interactive session, `MC3` will return a tuple with six elements (seven if `chireturn=True`, see above):

- `bestp`: a 1D array with the best-fitting parameters (including fixed and shared parameters).
- `CRlo`: a 1D array with the lower boundary of the marginal 68%-highest posterior density (the credible region) for each parameter, with respect to `bestp`.
- `CRhi`:a 1D array with the upper boundary of the marginal 68%-highest posterior density for each parameter, with respect to `bestp`.
- `stdp`: a 1D array with the standard deviation of the marginal posterior for each parameter (including that of fixed and shared parameters).
- `posterior`: a 2D array containing the burned-in, thinned MCMC sample of the parameters posterior distribution (with dimensions [nsamples, nfree], excluding fixed and shared parameters).
- `Zchain`: a 1D array with the indices of the chains for each sample in `posterior`.

```
# Run the MCMC:
bestp, CRlo, CRhi, stdp, posterior, Zchain = mc3.mcmc(data=data,
    uncert=uncert, func=func, indparams=indparams,
    params=params, pmin=pmin, pmax=pmax, stepsize=stepsize,
    prior=prior, priorlow=priorlow, priorup=priorup,
    walk=walk, nsamples=nsamples,  nchains=nchains,
```

(continues on next page)

(continued from previous page)

```
    nproc=nproc, burnin=burnin, thinning=thinning,
    leastsq=leastsq, lm=lm, chisqscale=chisqscale,
    hsize=hsize, kickoff=kickoff,
    grtest=grtest,  grbreak=grbreak, grnmin=grnmin,
    wlike=wlike, log=log,
    plots=plots, savefile=savefile, rms=rms, full_output=full_output)
```

Note: Note that `bestp`, `CRlo`, `CRhi`, and `stdp` include the values for all model parameters, including fixed and shared parameters, whereas `posterior` includes only the free parameters. Be careful with the dimesions.

3.2.4 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, 1000)       # 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              40             1.0
         -2.0             -20              20             0.5
          0.1             -10              10             0.1
```

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

```
params   = [ 10, -2.0,  0.1]
pmin     = [-10,  -20, -10]
pmax     = [ 40,   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:
bestp, CRlo, CRhi, stdp, posterior, Zchain = mc3.mcmc(data=data,
    func=func, indparams=indparams, params=params,
    walk=walk, nsamples=nsamples,  nchains=nchains,
    nproc=nproc, burnin=burnin, thinning=thinning,
    leastsq=leastsq, lm=lm, chisqscale=chisqscale,
    hsize=hsize, kickoff=kickoff,
    grtest=grtest, grbreak=grbreak, grnmin=grnmin,
    wlike=wlike, log=log,
    plots=plots, savefile=savefile, rms=rms, full_output=full_output)
```

3.2.5 References

3.3 Optimization Tutorial

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

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

3.3.1 Optimization Algorithm

The `lm` argument (default: `False`) determines the optimization algorithm. If `lm=True`, use the Levenberg-Marquardt algorithm (through `scipy.optimize.leastsq`). If `lm=False`, use the Trust Region Reflective algorithm (through `scipy.optimize.least_squares`).

Note that although LM is more efficient than TRF, LM does not support parameter boundaries. A LM run will find the un-bounded best-fitting solution, regardless of `pmin` and `pmax`.

For the same reason, if the model parameters are not bounded (i.e., `np.all(pmin==-np.inf)` and `np.all(pmax==np.inf)`), `modelfit` will use the LM algorithm.

3.3.2 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.

3.3.3 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)
```

3.3.4 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.

3.3.5 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.

3.3.6 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.

3.3.7 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.

3.3.8 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:

$$p(\theta) = \frac{1}{\theta_{\rm max} - \theta_{\rm min}}, \tag{6}$$

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:

$$p(\theta) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(\frac{-(\theta-\theta_p)^2}{2\sigma_p^2}\right), \tag{7}$$

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).

3.3.9 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` is the output from the `scipy` optimization routine.

3.3.10 Example

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

# Get a modeling function (quadractic polynomial):
sys.path.append("./examples/models/")  # Set the appropriate path
from quadratic import quad
```

(continues on next page)

(continued from previous page)

```
# Create a synthetic dataset using a quadratic polynomial curve:
x   = np.linspace(0, 10, 1000)          # 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])

func = 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, quad,
    data, uncert, indparams=indparams,
    stepsize=stepsize, pmin=pmin, pmax=pmax,
    prior=prior, priorlow=priorlow, priorup=priorup, lm=True)
```

3.4 Time Averaging

The `MCcubed.rednoise.binrms` routine computes the binned RMS array (as function of bin size) used in the the time-averaging procedure. Given a (model-data) residuals array. The routine returns the RMS of the binend data (rms_N), the lower and upper RMS uncertainties, the extrapolated RMS for Gaussian (white) noise (σ_N), and the bin-size array (N).

This function uses an asymptotic approximation to compute the RMS uncertainties ($\sigma_{\mathrm{rms}} = \sqrt{\mathrm{rms}_N/2M}$) for number of bins $M > 35$. For smaller values of M (equivalently, large bin size) this routine computes the errors from the posterior PDF of the RMS (an inverse-gamma distribution). See [Cubillos2017].

3.4.1 Example

```
import numpy as np
import matplotlib.pyplot as plt
import MCcubed as mc3   # Add path to mc3 if necessary
plt.ion()

# Generate residuals signal:
N = 1000
# White-noise signal:
white = np.random.normal(0, 5, N)
```

(continues on next page)

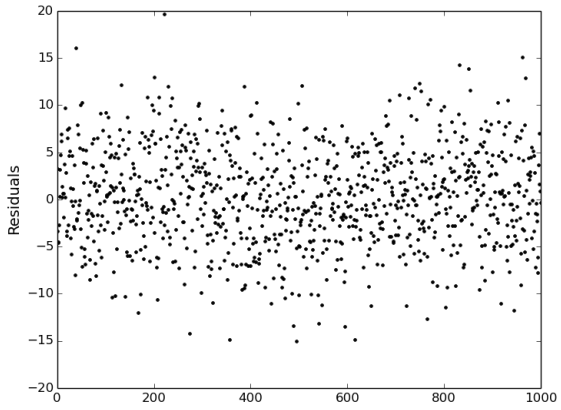
(continued from previous page)

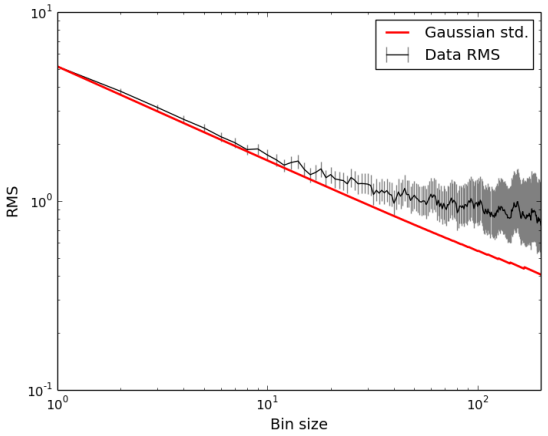
```
# (Sinusoidal) time-correlated signal:
red = np.sin(np.arange(N)/(0.1*N))*np.random.normal(1.0, 1.0, N)

# Plot the time-correlated residuals signal:
plt.figure(0)
plt.clf()
plt.plot(white+red, ".k")
plt.ylabel("Residuals", fontsize=14)

# Compute the residuals rms-vs-binsize:
maxbins = N/5
rms, rmslo, rmshi, stderr, binsz = mc3.rednoise.binrms(white+red, maxbins)

# Plot the rms with error bars along with the Gaussian standard deviation curve:
plt.figure(-6)
plt.clf()
plt.errorbar(binsz, rms, yerr=[rmslo, rmshi], fmt="k-", ecolor='0.5', capsize=0,␣
↪label="Data RMS")
plt.loglog(binsz, stderr, color='red', ls='-', lw=2, label="Gaussian std.")
plt.xlim(1,200)
plt.legend(loc="upper right")
plt.xlabel("Bin size", fontsize=14)
plt.ylabel("RMS", fontsize=14)
```





3.4.2 References

3.5 Contributing

Feel free to contribute to this repository by submitting code pull requests, raising issues, or emailing the administrator directly.

3.5.1 Raising Issues

Whenever you want to raise a new issue, make sure that it has not already been mentioned in the issues list. If an issue exists, consider adding a comment if you have extra information that further describes the issue or may help to solve it.

If you are reporting a bug, make sure to be fully descriptive of the bug, including steps to reproduce the bug, error output logs, etc.

Make sure to designate appropriate tags to your issue.

An issue asking for a new functionality must include the `wish list` tag. These issues must include an explanation as to why is such feature necessary. Note that if you also provide ideas, literature references, etc. that contribute to the implementation of the requested functionality, there will be more chances of the issue being solved.

3.5.2 Programming Style

Everyone has his/her own programming style, I respect that. However, some people have terrible style (see http://www.abstrusegoose.com/432). Following good coding practices make everyone happy, it will increase the chances of your code being added to the main repository, and it will make me work less. I strongly recommend the following programming guidelines:

- Always keep it simple.
- **Lines are strictly 80 character long, no more.**
- **Never ever! use tabs (for any reason, just don't).**
- Avoid hard-coding values at all cost.
- One–two character variable names are too short to be meaningful.
- Indent with 2 spaces.
- Put whitespace around operators and after commas.
- Separate lines (within a common block of code) by at most 0 whitespace lines (yes, zero).
- Separate blocks of code by at most 1 whitespace lines.
- Separate methods/functions/clasess by at most 2 whitespace lines.
- Use a header comment (1+ whole line) to describe a code block.
- Use in-line comments to describe code withing a block.
- Necessary contraptions require meaningful comments.
- Always, always make a docstring.
- Use `is` to compare with `None`, `True`, and `False`.
- Limit try clauses to the bare minimum.

Good pieces of code that do not follow these principles will still be gratefully accepted, but with a frowny face.

3.5.3 Pull Requests

To submit a pull request you will need to first (only once) fork the repository into your account. Edit the changes in your repository. When making a commit, always include a descriptive message of what changed. Then, click on the pull request button.

More on this later, which branch to pull, git Work flow, etc.

3.6 License

4 Be Kind

Please cite this paper if you found `MC3` useful for your research: Cubillos et al. (2017): On the Correlated-noise Analyses Applied to Exoplanet Light Curves, AJ, 153, 3.

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

If you have an older version, you can compile these docs, according to your version into a pdf with the following commands:

```
# cd into MCcubed/docs
make latexpdf
```

The output pdf docs will be located at `.../MCcubed/docs/latex/MC3.pdf`.

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

[BraakVrugt2008] ter Braak & Vrugt (2008): Differential Evolution Markov Chain with snooker updater and fewer chains

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

[Cubillos2017] Cubillos et al. (2017): On the Correlated-noise Analyses Applied to Exoplanet Light Curves