# Multi-Core Markov-Chain Monte Carlo ($MC^3$) User's Manual

Patricio E. Cubillos

November 11, 2014

## Contents

# 1 Team Members


- Patricio Cubillos[1] (author), University of Central Florida (pcubillos@fulbrightmail.org).
- Joseph Harrington, University of Central Florida.
- Madison Stemm, University of Central Florida.


# 2 The Markov-Chain Monte Carlo

## 2.1 Introduction:

**TL;DR:** You provide the data and the modeling function, MC$^3$ gives you the parameter posteriors.

The MC$^3$ package provides a set of routines to sample the parameter posterior probability distributions for a user-provided fitting function to a data set. To do so it uses Bayesian Inference through a Markov-chain Monte Carlo algorithm. The MCMC follows, either, Differential-Evolution (recomended) or Metropolis Random Walk. It handles Bayesian priors (uniform, Jeffrey's, or informative), Gelman-Rubin convergence test, shared-parameter fitting. Functionality through the Python Interpreter or from the shell. Single-CPU or Multi-core (supported by Messaging Passing Interface, MPI) computation.

## 2.2 The Math Behind:

### 2.2.1 Metropolis Acceptance Rule:

The Markov-chain Monte Carlo (MCMC) algorithm is a powerful Bayesian numerical tool to estimate confidence intervals of model-fitting parameters. Given a data set $\mathbf{D}$ and a model $M$, parameterized by the set of fitting variables $\mathbf{X}$, MCMC algorithm draws random samples from the parameter phase space with a probability density that approximates the posterior distribution, $p(\mathbf{x}|\mathbf{D}, M)$. The parameter's $1\sigma$ confidence intervals are defined as the boundaries that enclose 68.3% of the marginalized distribution.

The random sampling follows a Markov chain, where at each iteration, $t$, a new state, $\mathbf{Y}$, is drawn via a proposal distribution, $q(\mathbf{Y}|\mathbf{X}_t)$, which depends only on the current state, $\mathbf{X}_t$. The method evaluates then the *Metropolis ratio*:

$$r = \frac{p(\mathbf{Y}|\mathbf{D}, M)q(\mathbf{X}_t|\mathbf{Y})}{p(\mathbf{X}_t|\mathbf{D}, M)q(\mathbf{Y}|\mathbf{X}_t)}, \tag{1}$$

and accepts the proposed state with an acceptance probability $\alpha(\mathbf{Y}|\mathbf{X}_t) = \min(1, r)$. If the proposal is accepted, it adopts $\mathbf{Y}$ as the next state, $\mathbf{X}_{t+1} = \mathbf{Y}$; otherwise it stays at the current state, $\mathbf{X}_{t+1} = \mathbf{X}_t$. Efficient random walks should have acceptance rates in the range $\sim 25\% - 50\%$.

By using a symmetrical proposal distribution, the ratio $q(\mathbf{X}_t|\mathbf{Y})/q(\mathbf{Y}|\mathbf{X}_t)$ equals 1. Furthermore, under Bayesian statistics, the posterior probability distribution can be rewritten as:

$$p(\mathbf{X}|\mathbf{D}, M) \propto p(\mathbf{X}, M)p(\mathbf{D}|\mathbf{X}, M), \tag{2}$$

[1] https://github.com/pcubillos/

where $p(\mathbf{X}, M)$ is the prior probability distribution and $p(\mathbf{D}|\mathbf{X}, M)$ is the likelihood function, $\mathcal{L}(\mathbf{X})$. Then, the Metropolis ratio reduces to:

$$r = \frac{p(\mathbf{Y}|M)\mathcal{L}(\mathbf{Y})}{p(\mathbf{X}_t|M)\mathcal{L}(\mathbf{X}_t)}. \tag{3}$$

### 2.2.2 Metropolis Random Walk:

The Metropolis Random walk algorithm typically uses a multivariate Gaussian distribution as the proposal distribution:

$$q(\mathbf{Y}|\mathbf{X}_t, \boldsymbol{\sigma}) = \mathcal{N}(\mathbf{X}_t, \boldsymbol{\sigma}) = \frac{1}{\sqrt{2\pi\boldsymbol{\sigma}^2}} \exp\left(-\frac{(\mathbf{Y} - \mathbf{X}_t)^2}{2\boldsymbol{\sigma}^2}\right), \tag{4}$$

where the proposal jump steps $\boldsymbol{\sigma}$ must be defined and heuristically adjusted by the user.

### 2.2.3 Differential Evolution:

Highly correlated parameter spaces render the Metropolis random walk with Gaussian proposals inefficient (as many of the proposed states will be rejected). Furthermore, the heuristic tuning of the proposal jump scales requires a significant interaction with the user before achieving an efficient configuration.

The differential-evolution Markov-chain algorithm (DEMC) automatically adjusts the jumps' scales and orientations eliminating the need for manual tuning. DEMC runs several chains in parallel, where for a given chain $i$, it draws the next-iteration state from the difference between the current state of two other randomly-selected chains ($j$ and $k$):

$$\mathbf{X}^i_{t+1} = \mathbf{X}^i_t + \gamma\left(\mathbf{X}^j_t - \mathbf{X}^k_t\right) + \gamma_2\mathbf{e}^i_t, \tag{5}$$

where $\gamma = 2.38/\sqrt{2d}$ is a scaling factor, with $d$ being the number of free parameters. The last term, $\gamma_2\mathbf{e}$, is a random distribution (of smaller scale than the posterior distribution) that ensures a complete exploration of posterior parameter space; MC$^3$ implements a multivariate Gaussian distribution for $\mathbf{e}$.

### 2.2.4 Parameter Priors:

MC$^3$ supports three types of priors. A uniform non-informative prior (default)

$$p(x) = 1/(x_{\max} - x_{\min}), \tag{6}$$

is appropriate when the parameter is bound between $x_{\min}$ and $x_{\max}$, but there is no prior knowledge of the value of $x$. 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.

A Jeffrey's non-informative prior

$$p(x) = \frac{1}{x\ln(x_{\max}/x_{\min})}, \tag{7}$$

is valid when the parameter can take only positive values between $x_{\rm min}$ and $x_{\rm max}$. This is a more appropriate prior than a uniform prior when $x$ can take values over several orders of magnitude, since it represents an equal probability per order of magnitude.

Note that even when the prior range may be large, if the data constrains the MCMC sampled distribution to a small region around the best-fitting value, both the uniform and Jeffrey's prior will agree on an the parameter posterior distribution.

Lastly, a Gaussian informative prior

$$p(x) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(\frac{-(x-x_p)^2}{2\sigma_p^2}\right), \tag{8}$$

incorporates prior knowledge on the parameter. Say, $x$ was previously estimated to be $x_p \pm \sigma_p$.

## 2.3 Modules Overview

The MC$^3$ Package's directory tree is organized as follow:

```
MCcubed
├── src
│   └── cfuncs
│       ├── src
│       └── lib
├── examples
│       ├── example01
│       └── example02
└── doc
```

The `/MCcubed/src/` and `/MCcubed/src/cfuncs/src/` directories contain the Pyhton and C source code, repectively. The `/MCcubed/src/cfuncs/lib/` directory contains the compiled C-extension binaries.

Python modules (`/MCcubed/src/`):

- `mcmc.py`
  Core module that implements the MCMC algorithm.

- `mccubed.py`
  Wrapper of mcmc.py that provides support for MPI multiprocessing and use of configuration files.

- `func.py`
  Subroutine that handles the model fitting function when run in parallel under MPI.

- `modelfit.py`
  Module that implements the Levenberg-Marquardt least-squares minimization and calculates chi-squared.

- `mcutils.py`
  Module with utility functions used in the project's code.

- `mcplots.py`
  Module with functions to plot the parameter trace curves, pairwise posterior distributions, marginalized posterior histograms, model fit and data, and rms-vs-bin size plots.

- `prayer.py`
  Module that implements the residual-permutation algorithm (also known as prayer-bead).

C-extension modules (`/MCcubed/src/cfuncs/src/`):

- `chisq.c`
  Calculate the chi-squared weighted value of the model fit to a data set.

- `dwt.c`
  Calculate the wavelet-based likelihood function of a dataset.

- `timeavg.c`
  Calculate the binned root mean square (rms) of the residuals and Gaussian-noise rms extrapolation.

- `binarray.c`
  Calculate the mean-weighted binned values and standard deviation of an data set.

- `stats.h`
  Assorted statistical routines.

- `wavelet.h`
  Calculate the Daubechies 4-coefficient discrete wavelet transform (also known as D4 or db2).

# 3 Installation and System Requirements

MC$^3$ is a (mostly) Python code with C extensions. To download the latest MC$^3$ stable version, go to the releases Github page.

Before using MC$^3$, compile the C extensions. cd into the `/MCcubed/src/cfuncs/` directory and run:

```
make
```

To remove the program binaries, run (from the same directory):

```
make clean
```

## 3.1 System Requirements

- A Python 2.3+ distribution.

- NumPy.

- matplotlib.
- mpi4py.
- A working MPI distribution (MPICH preferred).

Not that mpi4py and the MPI distribution are only required for a multi-core run (A single-thread run should still work without MPI).

# 4 Examples:

## 4.1 Quick Demo

Here is a quick demo with the minimal inputs required for an MC$^3$ run. In this example we'll fit a quadratic polynomial to a noisy data set. To start, cd into the `/MCcubed/` directory and open the Python interpreter. Then follow the steps from this script:

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

# Get a function to model (and sample):
sys.path.append("./examples/example01/")
from quadratic import quad

# Create a synthetic dataset:
x = np.linspace(0, 10, 100)          # Independent model variable
p0 = 30, -2.4, 0.6                   # 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

# To see the MCMC docstring run:
help(mc3.mcmc)

# Fit the quad polynomial coefficients:
params = np.array([ 10.0, 25.0, -1.8]) # Initial guess

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

That's it!. You will see that every 10% of completed MCMC iterations, MC$^3$ prints a progress report with the percentage of completed iterations, date and time, The number of times a parameter has gone out of bounds, the current best-fitting parameters and the corresponding chi-squared value.

At the end you will see also a summary report with basic information about the MCMC run results:

```
Fin, MCMC Summary:
------------------
 Burned in iterations per chain:   100
 Number of iterations per chain:  3000
 MCMC sample size:               29000
 Acceptance rate:   39.64%

 Best-fit params    Uncertainties   Signal/Noise       Sample Mean
   3.2055627e+01    1.6325217e+00          19.64     3.1999302e+01
  -3.4654585e+00    8.0761582e-01           4.29    -3.4510235e+00
   7.1208855e-01    8.2552189e-02           8.63     7.1097733e-01

 Best-parameter's chi-squared:      96.8259
 Bayesian Information Criterion:   110.6415
 Reduced chi-squared:                0.9982
 Standard deviation of residuals:  5.84005
```

The module `mcplots.py` enables functions to plot the MCMC results: Data *vs.* the best fitting model, trace plots of the MCMC fitting parameter iterations, pairwise posterior distributions, marginal posterior histograms, and and rms-*vs.*-bin size plots.
These plots can be generated automatically during the MCMC call (set `plots=True`) or produced manually, which can be then tuned up at the user's preference. E.g. continuing from the previous script:

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

# Plot the data and best-fitting model:
mp.modelfit(data, uncert, x, model, nbins=100, title="Quadratic fit")
plt.plot(x, y,     "-r", lw=2, label='True')
plt.plot(x, guess, "-g", lw=2, label='Initial guess')
plt.legend(loc="best")

# The module mcplots provides helpful plotting functions:
# Plot trace plot:
parname = ["constant", "linear", "quadratic"]
mp.trace(allp, title="Fitting-parameter Trace Plots", parname=parname)

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

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

Figure (1) shows the output plots from this script.

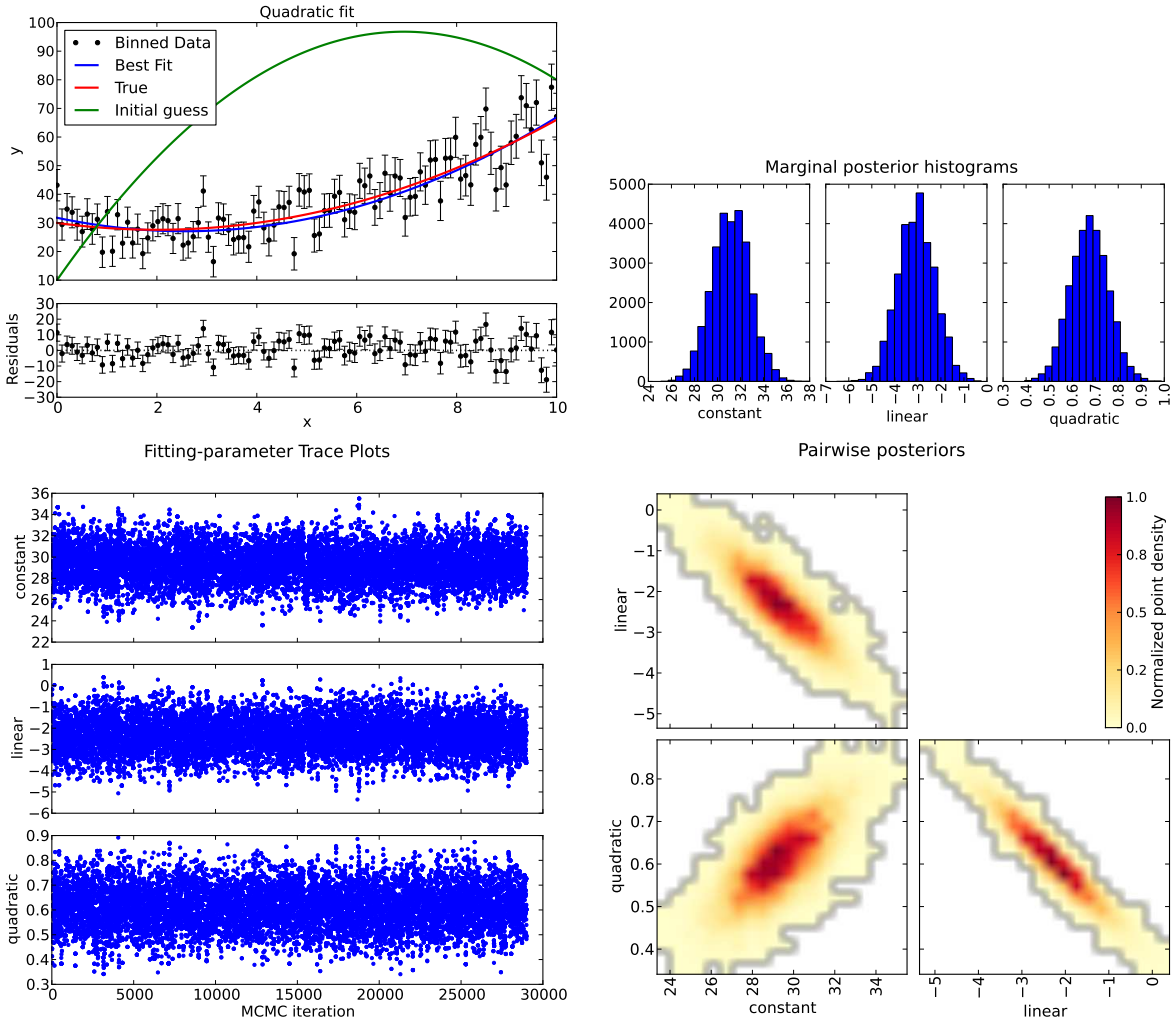


Figure 1: Basic plots produced by MC$^3$. Best-fitting model and data (Top left), parameter chain traces (Bottom left), parameter histogram posteriors (Top right), and parameter pairwise posteriors (Bottom right).

## 4.2 Example 01: Basics

This script shows how to run MCMC from the Python interpreter with more details. To start, cd into the `/MCcubed/examples/example01/` directory and start an interactive session. Generate a noisy data set that will be later fit by a quadratic polynomial:

```
import sys
import numpy as np
import matplotlib.pyplot as plt
sys.path.append("../../src")
import mccubed as mc3
import mcplots as mp
import mcutils as mu
```

```
# Get function to model/sample:
from quadratic import quad

# Create a synthetic dataset:
x  = np.linspace(0, 10, 100)  # Independent model variable
p0 = 30, -2.4, 0.6            # 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
```

Now lets set the MCMC arguments, remember to check the help to display the full list of arguments along with a brief description:

```
# Display the MCMC function docstring:
help(mc3.mcmc)

# The modeling function can be defined as callable:
func = quad
# Or by a tuple with the function and module names:
#     func = ("quad", "quadratic")
# If the module is out of the python-path scope, set the path as the
# 3rd element of the tuple:
#     func = ("quad", "quadratic", "/home/path/to/file/")
# The only requirement for the modeling function is that its first
# argument must be the array of fitting parameters:
#   model = func(params, *indparams)

# indparams is a list that contains every additional parameter of func,
# one func argument per item in the list.
# For this example we have:
indparams = [x]
# If func does not require additional arguments define indparams as:
# indparams=[], or simply leave it undefined.

# We will fit the quadratic polynomial coefficients:
# params contains the initial guess of fitting parameters:
params   = np.array([ 10.0,  16.0,  -1.8])

# The pmin and pmax arguments set the parameter phase-space boundaries:
pmin     = np.array([-10.0, -20.0, -10.0])
pmax     = np.array([ 60.0,  20.0,  10.0])

# stepsize has multiple functionalities.
# At the beginning, it works as the standard deviation of a Gaussian
# function to draw the initial state of the chains, about the provided
```

```
# initial guess (params).
# For Metropolis Random Walk, stepsize works as the standard deviation
# of the Gaussian function that draws the parameter proposals.
stepsize = np.array([  1.0,   0.5,   0.1])
# Additionally, the value of a fitting parameter can be kept fixed if its
# stepsize is 0.  To force a parameter to share its value with a
# second one, set its stepsize to the negative index of the second
# parameter.  E.g., to set the third parameter equal to the first one,
# set: stepsize[2] = -1.

# MCMC setup:
numit    = 3e4      # Total number of samples
nchains = 10        # Number of simultaneous chains
# Each chain will run (numit/nchains) iterations
burnin   = 100      # Number of iteration to discard (per chain)

walk     = 'demc'  # Choose between 'mrw' or 'demc' (recommended)
grtest   = True    # Evaluate the Gelman-Rubin convergence test
plots    = True    # Generate and save plots
savefile = 'output_ex1.npy'    # Save results to file
savemodel = 'output_model.npy' # Save the model for each MCMC sample
# Perform a least-square minimization before the MCMC to find the best
# fitting parameters:
leastsq    = True
# Scale the uncertainties to enforce reduced chi-square = 1 (only if
# you have a strong argument not to believe your data uncertainties):
chisqscale = True

# In a multi-processor run, MCMC will spawn one process for each chain,
# calculating each model in an independent CPU:
mpi=True

# Run the MCMC:
allp, bp = mc3.mcmc(data, uncert, func, indparams,
                    params, pmin, pmax, stepsize,
                    numit=numit, nchains=nchains, walk=walk, grtest=grtest,
                    leastsq=leastsq, chisqscale=chisqscale,
                    burnin=burnin, plots=plots, savefile=savefile,
                    savemodel=savemodel,  mpi=mpi)
```

If `grtest=True`, the module will display if the criterion has been fulfilled. Likewise, when `chisqscale=True`, the final report will display the scaling factor:

```
...

[::::::::: ]  90.0% completed  (Mon Jun  2 18:18:45 2014)
```

```
Out-of-bound Trials:
[0 0 0]
Best Parameters:   (chisq=97.0000)
[ 29.95607203  -2.39091023   0.58851943]
Gelman-Rubin statistic for free parameters:
[ 1.00111026  1.00084598  1.00118197]
All parameters have converged to within 1% of unity.

...

Fin, MCMC Summary:
------------------
 Burned in iterations per chain:   100
 Number of iterations per chain:  3000
 MCMC sample size:               29000
 Acceptance rate:   39.81%

 Best-fit params    Uncertainties   Signal/Noise       Sample Mean
   2.9956072e+01    1.5927766e+00          18.81     3.0025987e+01
  -2.3909102e+00    7.8486027e-01           3.05    -2.4199849e+00
   5.8851943e-01    8.0404908e-02           7.32     5.9076564e-01

 sqrt(reduced chi-squared) factor:   0.9851
 Best-parameter's chi-squared:      97.0000
 Bayesian Information Criterion:   110.8155
 Reduced chi-squared:                1.0000
 Standard deviation of residuals:  6.03983

Plotting figures ...
```

If `savefile` is not `None`, the resulting sample distribution is stored as a binary `.npy` file. This file can be later read with the NumPy's `nupy.load()` function. Likewise for the `savemodel` argument.

### 4.2.1 Inputs From Files

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.py` 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
```

### 4.2.2 Use of Configuration Files

MC$^3$ supports the use of a configuration file to set the MCMC arguments. The configuration file follows the `ConfigParser` module syntax. Check out the provided configuration file example `/MCcubed/examples/example01/config_demc.cfg`:

```
# Don't mess with the [MCMC] line, it must always be there.
# Arguments are set by providing the keyword, followed by an equal sign,
# and then the value(s).
# Strings don't need quotation marks
# Arrays are set by white-space separated values (no commas, no brackets)
# Comment lines are allowed (and ignored)
# Comments after an argument are allowed only with the ';' character.
[MCMC]
# DEMC general options:
numit      = 1e5
nchains    = 10
walk       = demc
grtest     = True
burnin     = 100
plots      = True
leastsq    = True
chisqscale = True
savefile   = output_ex01.npy
```

```
savemodel  = output_model.npy
mpi        = True
# Fitting function options:
func       = quad quadratic ../../examples/example01
params     = pars_ex01.dat
indparams  = indp_ex01.dat
# The data and uncertainties:
data       = data_ex01.dat
```

The configuration file is specified through the `cfile` argument. In case of conflict, when an argument is specified twice, the inline-command argument overrides the configuration-file argument. This simplifies the function call a lot:

```
allp, bp = mc3.mcmc(cfile='config_demc.cfg')
```

### 4.2.3 Run From Shell

To see the full list of available arguments, run from the shell:

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

MC$^3$ can be run from the shell by using the `mpiexec` command, the path to the `mccubed.py` file and the command line arguments:

```
mpiexec ../../src/mccubed.py -c config_demc.cfg
```

If you don't have MPI installed, you can still run the code in single-thread mode (make sure to leave the `mpi` argument unspecified, or set to `False`):

```
../../src/mccubed.py -c config_demc.cfg
```

Arguments can be passed from the command line or from a configuration file (recommended), or both. Command-line arguments override configuration-file arguments.

## 4.3 Example 02: Advanced

This script shows some more advanced features of the MC$^3$ package. To run the example commands, first cd into the `/MCcubed/examples/example02/` directory.

### 4.3.1 Parameter Priors

The default priors are uniform non-informative. To set an informative Gaussian prior for a parameter, representing a previous observation $y^{+u}_{-l}$, set the `prior`, `priorlow`, and `priorup` values to $y$, $u$, and $l$, respectively (noth tha both `priolow` and `priorup` must be $> 0$). To set a uniform non-informative prior, set the `priorlow` value to $0$. To set a Jeffrey's non-informative prior, set the `priorlow` value to $-1$.
For example, the file `/MCcubed/examples/example02/pars_ex02.dat` sets a prior on the linear term of the quadratic function, and leaves the other two with the default uniform non-informative:

```
# param         pmin      pmax  stepsize     prior  priorlow   priorup
     10          -10        60         1       0.0       0.0       0.0
     16          -20        20       0.5      -2.4       0.3       0.3
   -1.8          -10        10       0.1       0.0       0.0       0.0
```

Since this prior imposes a tighter constrain on the linear parameter than the data set alone, the signal-to-noise found are larger:

```
mpiexec ../../src/mccubed.py -c config_demc.cfg

...

 Best-fit params    Uncertainties   Signal/Noise       Sample Mean
   2.8858654e+01    9.4698175e-01          30.47     2.8852173e+01
  -2.3320419e+00    2.8287321e-01           8.24    -2.3271135e+00
   6.2138497e-01    3.4698982e-02          17.91     6.2082910e-01
...
```

### 4.3.2 Wavelet-based Likelihood

TBD (ask me: pcubillos@fulbrightmail.org).

# 5 Be Kind

Please cite this work if you found this package useful for your research:

- Cubillos et al. (2014): On the Correlated Noise Analyses Applied to Exoplanet Light Curves (in preparation).

Thanks!

# 6 Further Reading

- Bayesian statistics:
  Gregory (2005): Bayesian Logical Data Analysis for the Physical Sciences.
- Differential-evolution Markov-chain algorithm:
  ter Braak (2006): An MCMC version of the genetic algorithm Differential Evolution.
- Gelman-Rubin convergence test:
  Gelman & Rubin (1992): Inference from Iterative Simulation Using Multiple Sequences.
- Wavelet-based likelihood:
  Carter & Winn (2009): Parameter Estimation from Time-series Data with Correlated Errors.

# 7 License

Multi-Core Markov-chain Monte Carlo (MC3), a code to estimate model-parameter best-fitting values and Bayesian posterior distributions.

This project was completed with the support of the NASA Planetary Atmospheres Program, grant NNX12AI69G, held by Principal Investigator Joseph Harrington. Principal developers included graduate student Patricio E. Cubillos and programmer Madison Stemm. Statistical advice came from Thomas J. Loredo and Nate B. Lust.



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

Patricio Cubillos pcubillos@fulbrightmail.org
Joseph Harrington jh@physics.ucf.edu

or alternatively,

Joseph Harrington and Patricio Cubillos
UCF PSB 441
4111 Libra Drive
Orlando, FL 32816-2385
USA

Thank you for using MC3!