emcee: The MCMC Hammer

Daniel Foreman-Mackey[1], David W. Hogg, *et al.*

Center for Cosmology and Particle Physics, Department of Physics, New York University, 4 Washington Place, New York, NY, 10003, USA

ABSTRACT

We introduce a stable, well tested Python implementation of the affine-invariant ensemble sampler for Markov chain Monte Carlo (MCMC) proposed by Goodman & Weare (2010). The code is open source and has already been used in several published projects in the Astrophysics literature. The algorithm behind emcee has several advantages over traditional MCMC sampling methods and it has excellent performance measured by the sampling autocorrelation time on several test problems. One major advantage of the algorithm is that it requires the hand-tuning of only 2 parameters compared to the $\sim N^2$ for a traditional algorithm in an N dimensional parameter space. In this document, we describe the Goodman & Weare algorithm and the details of our implementation. emcee takes advantage of the naturally parallel nature of the algorithm allowing *any* user to take advantage of multiple CPUs without significant extra effort. We discuss the subtleties associated with this parallelization.

The code is available online at http://danfm.ca/emcee under the GNU General Public License (v2).

Subject headings: methods: data analysis — methods: numerical — methods: statistical

1. Introduction

Probabilistic data analysis — including Bayesian methods — has transformed scientific research in the past decade. The most significant gains have been due to numerical methods of approximate inference; especially Markov chain Monte Carlo (MCMC) methods. In particular, many problems in cosmology and astrophysics benefit from these advances because they lie in the low signal-to-noise regime with incomplete observations. In these fields, it is

[1]danfm@nyu.edu

often possible to exploit a physically justified generative model of the data, making these research questions tractable. In principle, there is no limit to the complexity of this generative process but many interesting research questions can only be solved with a combination of more computing power and more efficient algorithms.

The general probabilistic data analysis procedure involves examining either the posterior probability density function (PDF) of the parameters of the model or the likelihood function of the data. In some cases, it is sufficient to just find the maximum of this function but it is often of interest to (at least approximately) measure this distribution in more detail. In particular, if we wish to propagate the effects of measurement uncertainties through the analysis to the conclusions, we must sample the PDF on sufficiently small scales. MCMC methods are designed to efficiently sample and approximate the PDF even in high dimensional parameter spaces. This has proved useful in too many research applications to list here (e.g. WMAP, etc. DFM/Hogg: add citations).

Arguably the most important advantage of Bayesian data analysis is that it is possible to *marginalize* over nuisance parameters. A nuisance parameter is a component of the generative model that is of little physical interest who's value must be known in order to produce data. Marginalization is the process of integrating over all possible values of these parameters and hence propagating the effects of your uncertainty about the "true" value to the final result. The exact result of marginalization is the marginalized likelihood function $p(\mathbf{D}|\mathbf{\Theta})$ of the data $\mathbf{D}$ given the physical model parameters $\mathbf{\Theta}$

$$p(\boldsymbol{D}|\mathbf{\Theta}) = \int p(\boldsymbol{D}|\mathbf{\Theta}, \boldsymbol{\alpha})\, p(\boldsymbol{\alpha})\, \mathrm{d}\boldsymbol{\alpha} \tag{1}$$

where $\boldsymbol{\alpha}$ is vector of nuisance parameters. In some specific cases, the integral in Equation (1) is analytically tractable but in general, the likelihood function $p(\boldsymbol{D}|\mathbf{\Theta}, \boldsymbol{\alpha})$ is not a simple integrable function. In fact, in many problems, the likelihood function is actually the result of an extremely expensive numerical simulation. In this regime, the integration must be calculated numerically and it is often relatively efficient to approximate Equation (1) using MCMC sampling. If the likelihood function is expensive to calculate, it is advantageous to use a sampling algorithm that reduces the necessary number of likelihood evaluations. This also precludes the use of second order methods (such as hybrid/Hamiltonian Monte Carlo) that require the calculation of (numerical) gradients of the likelihood function.

Most uses of MCMC in the astrophysics literature are based on slight modifications to the Metropolis-Hastings (M-H) method (e.g. MacKay 2003). Each step in a M-H chain is proposed using a multivariate Gaussian centered on the current position of the chain. Since each term in the covariance matrix of this proposal distribution is an unspecified parameter, this method has $N\,[N+1]/2$ tuning parameters (where N is the dimension of the parameter

space). To make matters worse, the performance of this sampler is very sensitive to the optimality of these tuning parameters and there is no fool-proof method for choosing the values correctly. As a result, many heuristic methods have been developed to attempt to determine the optimal parameters in a data-driven way (e.g. Gregory 2005; Dunkley *et al.* 2005; Widrow *et al.* 2008). Unfortunately, these methods all require "burn-in" phases where shorter Markov chains are sampled and the results are used to tune the hyperparameters. This extra cost is unacceptable when the likelihood calls are computationally heavy.

The problem with traditional sampling methods can be visualized by looking at the simple but highly anisotropic density

$$p(\mathbf{x}) \propto \exp\left(-\frac{(x_1 - x_2)^2}{2\,\epsilon} - \frac{(x_1 + x_2)^2}{2}\right) \tag{2}$$

which would be considered "difficult" (in the low-ϵ regime) by standard MCMC algorithms. In principle, it is possible to tune the hyperparameters of a M-H sampler but if calculating the density in Equation (2) is computationally expensive and the covariance is not known *a priori* then the tuning procedure gets quickly intractable. Also, since the number of parameters scales as $\sim N^2$, this problem gets mush worse in higher dimensions. Equation (2) can, however, be transformed into the much easier problem of sampling an isotropic Gaussian by an *affine transformation* of the form

$$y_1 = \frac{x_1 - x_2}{\sqrt{\epsilon}}\,, \qquad y_2 = x_1 + x_2. \tag{3}$$

Therefore, an algorithm that is *affine invariant* will be insensitive to covariances between parameters. An affine invariant algorithm is unaffected by any transformation of the density of the form

$$\mathbf{Y} = \mathbf{A}\,\mathbf{X} + \mathbf{b}. \tag{4}$$

Extending earlier work by Christen (2007), Goodman & Weare (2010, hereafter GW10) proposed an affine invariant sampling algorithm (§2) with only two hyperparameters that can be tuned for performance. Hou *et al.* (2011) were the first group to implement this algorithm to solve a physics problem. The implementation presented here is an independent effort that has already proved effective in several projects (Lang & Hogg 2011; Bovy *et al.* 2011; Dorman *et al.* 2012, Foreman-Mackey & Widrow 2012, in prep.). In what follows, we summarize the GW algorithm and the implementation decisions made in emcee. We also describe the small changes that must be made to the algorithm to parallelize it. Finally, in §A, we outline the installation, usage and troubleshooting of the package.

2. The Algorithm

A complete discussion of MCMC methods is beyond the scope of this document. Instead, the interested reader is directed to a classic reference like MacKay (2003) and we will summarize some key concepts below.

The general goal of MCMC algorithms is to draw samples $\{\boldsymbol{\Theta}_i \, \forall i = 1, \ldots, M\}$ from the joint probability distribution

$$p(\boldsymbol{\Theta}, \boldsymbol{\alpha}, \boldsymbol{D}) = p(\boldsymbol{\Theta}, \boldsymbol{\alpha})\, p(\boldsymbol{D}|\boldsymbol{\Theta}, \boldsymbol{\alpha}) \tag{5}$$

where the prior distribution $p(\boldsymbol{\Theta}, \boldsymbol{\alpha})$ and the likelihood function $p(\boldsymbol{D}|\boldsymbol{\Theta}, \boldsymbol{\alpha})$ can be relatively easily (but not necessarily quickly) computed for a particular value of $(\boldsymbol{\Theta}_i, \boldsymbol{\alpha}_i)$. Since the normalization $p(\boldsymbol{D})$ is independent of $\boldsymbol{\Theta}$ and $\boldsymbol{\alpha}$, the joint distribution above is proportional to the posterior probability $p(\boldsymbol{\Theta}, \boldsymbol{\alpha}|\boldsymbol{D})$ given any one choice of generative model. Therefore, once the samples produced by MCMC are available, the marginalized constraints on $\boldsymbol{\Theta}$ (Equation (1)) can be approximated by the histogram of the samples projected into the subspace spanned by $\boldsymbol{\Theta}$. In particular, the expectation value of a particular parameter $\phi \in \boldsymbol{\Theta}$ given the samples $\{\phi_i\}$ is

$$E[\phi] = \int \phi\, p(\boldsymbol{\Theta}, \boldsymbol{\alpha}|\boldsymbol{D})\, \mathrm{d}\boldsymbol{\Theta}\, \mathrm{d}\boldsymbol{\alpha} \approx \frac{1}{M} \sum_{i=1}^{M} \phi_i. \tag{6}$$

Generating the samples $\boldsymbol{\Theta}_i$ is a non-trivial process unless $p(\boldsymbol{\Theta}, \boldsymbol{\alpha}, \boldsymbol{D})$ is a very specific analytic distribution (e.g. Gaussian). MCMC is a procedure for generating a random walk in the parameter space the relatively efficiently draws a representative set of samples from the distribution. Each point in a Markov chain $X(t) = [\boldsymbol{\Theta}(t), \boldsymbol{\alpha}(t)]$ depends only on the position of the previous link $X(t-1)$.

The Metropolis-Hastings (M-H) Algorithm The simplest and most commonly used MCMC algorithm is the M-H method (Algorithm (1)). The iterative procedure is as follows: (1) given a position $X(t)$ sample a proposal position Y from the transition distribution $Q(Y; X(t))$, (2) accept this proposal with probability

$$\min\left\{1, \frac{p(Y|\boldsymbol{D})}{p(X(t)|\boldsymbol{D})} \frac{Q(X(t); Y)}{Q(Y; X(t))}\right\}. \tag{7}$$

It is worth emphasizing that if this step is accepted $X(t+1) = Y$; Otherwise, the new position is set to the previous one $X(t+1) \leftarrow X(t)$ (i.e. the position $X(t)$ is *double counted*).

The M-H algorithm converges (as $t \to \infty$) to a stationary set of samples from the distribution but there are many algorithms with faster convergence and varying levels of

implementation difficulty (CITECITECITE). Faster convergence is preferred because of the reduction of computational cost due to the smaller number of likelihood computations necessary to obtain the equivalent level of accuracy. The efficiency of an algorithm can be measured by the autocorrelation function and more specifically, the integrated autocorrelation time (see §3). This quantity is an estimate of the number of steps needed in the chain in order to draw independent samples from the target density. Therefore, a more efficient chain will have a shorter autocorrelation time.

Algorithm 1 The procedure for a single Metropolis-Hastings MCMC step.

1: Draw a sample $Y \sim Q(Y; X(t))$
2: $q \leftarrow [p(Y)\, Q(X(t); Y)]/[p(X(t))\, Q(Y; X(t))]$
3: $r \leftarrow R \sim [0, 1]$
4: **if** $r \geq q$ **then**
5: $\quad X(t+1) \leftarrow Y$
6: **else**
7: $\quad X(t+1) \leftarrow X(t)$
8: **end if**

The stretch move GW10 proposed an affine invariant ensemble sampling algorithm informally called the "stretch move". For completeness and for clarity of notation, we summarize the algorithm here and refer the interested reader to the original paper for more details. This method involves simultaneously evolving an ensemble of K *walkers* $S = \{X_j, \forall j = 1, \ldots, K\}$ where the proposal distribution for one walker k is based on the current positions of the $K - 1$ walkers in the *complementary ensemble* $S_{[k]} = \{X_j, \forall j \neq k\}$. In general, each X_j is also a vector in N dimensions (the dimension of the parameter space).

To update the position of a walker at position X_k, another walker X_j with $j \neq k$ is randomly chosen and then a new position is proposed:

$$X_k(t) \to Y = X_j + Z\,[X_k(t) - X_j] \tag{8}$$

where Z is a random variable drawn from a distribution $g(Z = z)$. It is clear that if g satisfies

$$g(z^{-1}) = z\, g(z), \tag{9}$$

the proposal of Equation (8) is symmetric. In this case, the chain will satisfy detailed balance if the proposal is accepted with probability

$$q = \min\left\{1, Z^{n-1}\, \frac{p(Y)}{p(X_k(t))}\right\} \tag{10}$$

where n is the dimension of the parameter space. This procedure is then repeated for each walker in the ensemble *in series* following the procedure shown in Algorithm (2).

GW10 advocate for a particular form of $g(z)$, namely

$$g(z) \propto \begin{cases} \frac{1}{\sqrt{z}} & \text{if } z \in \left[\frac{1}{a}, a\right] \\ 0 & \text{otherwise} \end{cases}, \tag{11}$$

where a is an adjustable scale parameter that GW10 set to 2.

Algorithm 2 A single stretch move update step from GW10 where line 5 is generally the most computationally expensive step.

1: **for** $k = 1, \ldots, K$ **do**
2: Draw a walker X_j at random from the complementary ensemble $S_{[k]}(t)$
3: $z \leftarrow Z \sim g(z)$, Equation (11)
4: $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
5: $q \leftarrow z^{n-1}\,p(Y)/p(X_k(t))$
6: $r \leftarrow R \sim [0, 1]$
7: **if** $R \geq q$, Equation (10) **then**
8: $X_k(t+1) \leftarrow Y$
9: **else**
10: $X_k(t+1) \leftarrow X_k(t)$
11: **end if**
12: **end for**

The parallel stretch move It is tempting to naïvely parallelize the stretch move algorithm by simultaneously advancing each walker based on the state of the ensemble instead of evolving the walkers in series. Unfortunately, this would no longer satisfy detailed balance. Instead, we must split the full ensemble into two subsets ($S^{(0)} = \{X_k\,\forall k = 1, \ldots, K/2\}$ and $S^{(1)} = \{X_k\,\forall k = K/2+1, \ldots, K\}$) and simultaneously update all the walkers in $S^{(0)}$ — using the stretch move procedure from Algorithm (2) — based *only* on the positions of the walkers in the other set ($S^{(1)}$). Then, using the new positions $S^{(0)}$, we can update $S^{(1)}$. In this case, the outcome is a valid step for all of the walkers. The pseudocode for this procedure is shown in Algorithm (3). This code appears similar to Algorithm (2) but now the computationally expensive inner loop (starting at line 2 in Algorithm (3)) can be run in parallel.

The performance of this method — quantified by the autocorrelation time — is comparable to the traditional stretch move algorithm but the fact that one can now take advantage of generic parallelization makes this generalization extremely powerful.

Algorithm 3 The parallel stretch move update step

for $i \in \{0, 1\}$ **do**
  for $k = 1, \ldots, K/2$ **do**
    Draw a walker X_j at random from the complementary ensemble $S^{(\sim i)}(t)$
    $X_k \leftarrow S_k^{(i)}$
    $z \leftarrow Z \sim g(z)$, Equation (11)
    $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
    $q \leftarrow z^{n-1}\,p(Y)/p(X_k(t))$
    $r \leftarrow R \sim [0, 1]$
    if $r \geq q$, Equation (10) **then**
      $X_k(t + \frac{1}{2}) \leftarrow Y$
    else
      $X_k(t + \frac{1}{2}) \leftarrow X_k(t)$
    end if
  end for
  $t \leftarrow t + \frac{1}{2}$
end for

3. Benchmarks & Tests

Measuring the performance A standard method of quantifying the performance of an MCMC sampler is to estimate the autocorrelation time of the sampler on several densities.

The main goal of running a Markov chain is to measure the expectation value (and variance) of a particular value (e.g. f)

$$\langle f(\mathbf{x}) \rangle = \int f(\mathbf{x})\, p(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{12}$$

which can be approximated as

$$\langle f(\mathbf{x}) \rangle \approx \frac{1}{T_s} \sum_{t=1}^{T_s} f(\mathbf{X}(t)) \tag{13}$$

where T is the length of the chain. The generalization of Equation (13) to the case of the ensemble sampler is

$$\langle f(\mathbf{x}) \rangle \approx \frac{1}{T_e} \sum_{t=1}^{T_s} \left[\frac{1}{K} \sum_{k=1}^{K} f(\mathbf{X}_k(t)) \right] \tag{14}$$

where K is the number of walkers. The autocorrelation function of the chain is then given

by

$$C(t) = \frac{1}{K^2} \lim_{t' \to \infty} \mathrm{cov} \left[\sum_{k=1}^{K} f(\mathbf{X}_k(t+t')), \sum_{k=1}^{K} f(\mathbf{X}_k(t')) \right] \tag{15}$$

and the integrated autocorrelation time is given by

$$\tau = \sum_{t=-\infty}^{\infty} \frac{C(t)}{C(0)}. \tag{16}$$

emcee can optionally calculate the autocorrelation time using the Python module acor[1] to estimate the autocorrelation time. This module is a direct port of the original algorithm (described by GW10) and implemented by those authors in C++.[2]

Multivariate Gaussian distribution The simplest test of an MCMC sampler is its sampling performance on a highly covariant multivariate Gaussian density. For the tests in the paper, we randomly generated a 50 dimensional positive definite covariance tensor and initial conditions for each walker. Then, each sampler was tested with the same initial conditions for its performance on the density

$$\pi(\mathbf{x}) \propto \exp\left(-\frac{1}{2} \mathbf{x}^T \, \Sigma^{-1} \, \mathbf{x} \right) \tag{17}$$

for the same covariance tensor Σ in each trial.

DFM: actually do some tests here...

Rosenbrock density

4. Discussion & Tips

REFERENCES

Bovy, J., Rix, H.-W., Liu, C., Hogg, D. W., Beers, T. C., & Lee, Y. S., 2011, ApJ, submitted, arXiv:1111.1724 [astro-ph.GA]

[1] http://github.com/dfm/acor

[2] http://www.math.nyu.edu/faculty/goodman/software/acor

Christen, J., *A general purpose scale-independent MCMC algorithm*, technical report I-07-16, CIMAT, Guanajuato, 2007.

Dorman, C., Guhathakurta, P., Fardal, M. A., Geha, M. C., Howley, K. M., Kalirai, J. S., Lang, D., Cuillandre, J., Dalcanton, J., Gilbert, K. M., Seth, A. C., Williams, B. F., & Yniguez, B., 2012, ApJ, submitted

Dunkley, J., Bucher, M., Ferreira, P. G., Moodley, K., & Skordis, C., 2005, MNRAS, 356, 925-936

Goodman, J., & Weare, J., 2010, Comm. App. Math. Comp. Sci., 5, 65

Gregory, P. C., *Bayesian Logical Data Analysis for the Physical Sciences*, Cambridge University Press, 2005

Hou, F., Goodman, J., Hogg, D. W., Weare, J., & Schwab, C., 2011, arXiv:1104.2612

Lang, D. and Hogg, D. W., 2011, arXiv:1103.6038

MacKay, D., *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003

Widrow, L. M. and Pym, B. and Dubinski, J., 2008, ApJ, 679, 1239

A. Usage

Installation The easiest way to install emcee is using pip[3]. Running the command

```
% pip install emcee
```

at the command line of a UNIX-based system will install the package and its Python dependencies. If you would like to install for all users, you might need to run the above command with superuser permissions. emcee depends on Python (> 2.7) and numpy[4] (> 1.6) and the associated dev headers. On some systems, you might need to install these packages separately. On Ubuntu, you can install these dependencies using the command:

This preprint was prepared with the AAS LaTeX macros v5.2.

[3] http://pypi.python.org/pypi/pip/

[4] http://numpy.scipy.org

```
% apt-get python python-dev numpy numpy-dev
```

An alternative installation method is to download the source code from http://danfm.ca/emcee and run

```
% python setup.py install
```

in that directory. Make sure that you have numpy installed as well.

Issues & Contributions The development of emcee is being coordinated on GitHub at http://github.com/dfm/emcee and contributions are welcome. If you encounter any problems with the code, please report them at http://github.com/dfm/emcee/issues (DFM: check this url) and consider contributing a patch.

```
import numpy as np
import emcee

def lnprobfn(x, mu, icov):
    diff = x-mu
    return -np.dot(diff,np.dot(icov,diff))/2.0

ndim  = 10
means = np.random.rand(ndim)
cov   = 0.5-np.random.rand(ndim**2).reshape((ndim, ndim))
cov   = np.triu(cov)
cov  += cov.T - np.diag(cov.diagonal())
cov   = np.dot(cov,cov)
icov  = np.linalg.inv(cov)

nwalkers = 100
p0 = [np.random.rand(ndim) for i in xrange(nwalkers)]

sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprobfn,
                        args=[means, icov])

pos,prob,state = sampler.run_mcmc(p0, None, 500)
sampler.clear_chain()

sampler.run_mcmc(pos, state, 2000)

print "Mean acceptance fraction:", np.mean(sampler.acceptance_fraction)
```

B. API

`emcee.ensemble` emcee.`EnsembleSampler(*args, **kwargs)`

Ensemble sampling following Goodman & Weare (2010) with optional parallelization

Arguments

`k` (`int`): The number of Goodman & Weare "walkers".

`dim` (`int`): Number of dimensions in the parameter space.

`lnpostfn` (`callable`): A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability for that position.

Keyword Arguments

`a` (`float`): The proposal scale parameter from Equation (11). (default: `2.0`)

`args` (`list`): Optional list of extra arguments for `lnpostfn`. `lnpostfn` will be called with the sequence `lnpostfn(p, *args)`.

`postargs` (`list`): Alias of `args` for backwards compatibility.

`threads` (`int`): The number of threads to use for parallelization. If `threads == 1`, then the `multiprocessing` is not used but if `threads > 1`, then a `Pool` object is created and calls to `lnpostfn` are run in parallel following Algorithm (3). (default: `1`)

`pool` (`multiprocessing.Pool`): An alternative method of using the parallelized algorithm. If `pool is not None`, the value of `threads` is ignored and the provided `Pool` is used for all parallelization. (default: `None`)

Exceptions

`AssertionError`: If `k < 2*dim` or if `k` is not even.

Properties

`chain` (`numpy.ndarray`): A pointer to the Markov chain itself. The shape of this array is `(k, dim, iterations/resample)`

`flatchain` (`numpy.ndarray`): A shortcut for accessing `chain` flattened along the zeroth (walker) axis.

`lnprobability` (`numpy.ndarray`): A pointer to the matrix of the value of `lnprobfn` produced at each step for each walker. The shape is `(k, iterations/resample)`.

`iterations` (`int`): The number of steps that have been run in the chain.

`acceptance_fraction` (`numpy.ndarray`): An array (length: `k`) of the fraction of steps accepted for each walker.

`acor` (`numpy.ndarray`): The autocorrelation time of each parameter in the chain (length: `dim`) as estimated by the `acor` module.

`random_state` (`tuple`): The state of the internal random number generator. In practice, it's the result of calling `get_state()` on a `numpy.random.mtrand.RandomState` object. You can try to set this property but be warned that if you do this and it fails, it will do so silently.

Methods

`emcee.EnsembleSampler.sample(pos0, lnprob0=None, rstate0=None, iterations=1)`

Advances the chain `iterations` steps as an iterator

Arguments

`pos0` (`numpy.ndarray`): A list of the initial positions of the walkers in the parameter space. The shape is `(k, dim)`.

Keyword Arguments

`lnprob0` (`numpy.ndarray`): The list of log posterior probabilities for the walkers at positions given by the `p0`. If `lnprob is None`, the initial values are calculated. The shape is `(k, dim)`.

`rstate0` (`tuple`): The state of the random number generator.
See `EnsembleSampler.random_state` for details.

`iterations` (`int`): The number of steps to run. (default: `1`)

Yields

`pos` (`numpy.ndarray`): A list of the current positions of the walkers in the parameter space. The shape is `(k, dim)`.

`lnprob` (`numpy.ndarray`): The list of log posterior probabilities for the walkers at positions given by the `pos`. The shape is `(k, dim)`.

`rstate` (`tuple`): The state of the random number generator.

`emcee.EnsembleSampler.run_mcmc(pos0, N, rstate0=None, lnprob0=None, **kwargs)`

Iterate `sample` for `N` iterations and return the result. The parameters are passed directly to `sample` so see above for details.

Returns

`pos` (`numpy.ndarray`): A list of the final positions of the walkers in the parameter space. The shape is `(k, dim)`.

`lnprob` (`numpy.ndarray`): The list of log posterior probabilities for the walkers at positions given by the `pos`. The shape is `(k, dim)`.

`rstate` (`tuple`): The final state of the random number generator.

`emcee.EnsembleSampler.reset()`

Clear `chain`, `lnprobability` and the bookkeeping parameters.

`emcee.EnsembleSampler.clear_chain()`

An alias of `reset` kept for backwards compatibility.