EzTao

Release 0.4.3

Weixiang Yu

Dec 15, 2023

GETTING STARTED

EzTao is a Python toolkit for conducting time-series analysis using continuous-time autoregressive moving average (CARMA) processes. It uses celerite (a fast gaussian processes regression library) to compute the likelihood of a set of proposed CARMA parameters given the input time series. Comparing to existing tools for performing CARMA analysis in Python which use *Kalman filter* to evaluate the likelihood function (e.g., Kali), **EzTao** offers a more scalable solution (see the celerite paper for a comparison).

EzTao consist of tools to both simulate CARMA processes and fit (maximum likelihood estimation or MLE) time series to CARMA models. The current version of **EzTao** is built on top of celerite, future versions will take advantage of celerite2 (still under active development) for a better integration with other probabilistic programing libraries such as PyMC3.

CHAPTER

ONE

INSTALLATION

EzTao can be installed with pip using:

```
pip install eztao
```

1.1 Introduction

Here, we provide some background on CARMA models and the connection between CARMA and Gaussian Process (GP).

1.1.1 CARMA

CARMA stands for continuous-time autoregressive moving average, it is the continuous-time version of the better known autoregressive moving average (ARMA) model. In recent years, CARMA have been utilized to study time-domain data across many disciplines. In short, CARMA processes are defined as the solutions to the following stochastic differential equation:

$$\mathrm{d}^p x + \alpha_1 \mathrm{d}^{p-1} x + \ldots + \alpha_{p-1} \mathrm{d}x + \alpha_p x = \beta_0 (\mathrm{d}W) + \ldots + \beta_q \mathrm{d}^q (\mathrm{d}W),$$

where α_* and β_* are the parameters of a CARMA model. The order of the underlying differential equation can be specified using two numbers p and q, where p is the autoregressive (AR) order and q is the moving average (MA) order. Thus, a particular CARMA model can also be referred to as a CARMA(p,q) model. For example, a CARMA(2,1) model will have a corresponding stochastic differential equation defined by:

$$\mathrm{d}^2 x + \alpha_1 \mathrm{d}x + \alpha_2 x = \beta_0 (\mathrm{d}W) + \beta_1 \mathrm{d}^1 (\mathrm{d}W)$$

In fact, the popular (in astronomy) Damped Random Walk (DRW) model is simply the lowest order CARMA model, namely CARMA(1,0) with a stochastic differential equation defined by:

$$\mathrm{d}x + \alpha_1 x = \beta_0 (\mathrm{d}W)$$

For more on CARMA models, please see:

- Jones & Ackerson 1990
- Brockwell 2001
- Kelly et al. 2014
- Kasliwal et al. 2017

Note

The notation used in this section (and through out **EzTao**) follows the standard using in Brockwell 2001.

1.1.2 CARMA as GPs -> EzTao

CARMA processes (driven by Gaussian noises) are in fact Gaussian Processes (GPs) by definition, thus, in principle the likelihood function of CARMA can be evaluated using GPs. However, such as a process requires inverting the autocovariance matrix, which scales as $O(N^3)$, where N is the number data points in a time series. The high computational cost, especially for time series having large number of data points, makes it intractable to use GPs to evaluate the likelihood function of CARMA models. Recent development in GPs, in particular, the generalization of Rybicki & Press 1995's method to a much larger set of kernels (demonstrated in Foreman-Mackey et al. 2017 has made it possible to compute the likelihood of CARMA models in $O(NJ^2)$ operations, where J is the p order of a CARMA(p, q) model. Although the computational complexity for the GP solution is the same as that of the *Kalman filter* solution, the GP solution is about 10 times faster on average.

celerite provides built-in support for evaluation of DRW's likelihood function using the real kernel, but not any other CARMA model. **EzTao** generalizes this approach to **ALL** CARMA models, and provides tools to facilitate the simulation and analysis of time series data using CARMA. **EzTao** uses a set of CARMA kernels, which you can define using regular CARMA notation, to handle the conversion between CARMA and *celerite*'s GPs. Those CARMA kernels are listed in the *eztao.carma.CARMATerm* module.

1.2 Quick Start

In this section, we will walk through a few simple examples to help you get started on **EzTao**. For more in-depth tutorials, please check out the sections listed under **Tutorials** on the *main page* (or other notebooks if you are running this in mybinder).

[1]:
```
# general packages
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline

# packages/modules from eztao and celerite
import eztao
from eztao.carma import DRW_term
from eztao.ts import gpSimRand
from eztao.ts import drw_fit
from celerite import GP

# use eztao matplotlib style
mpl.rc_file(os.path.join(eztao.__path__[0], "viz/eztao.rc"))
```

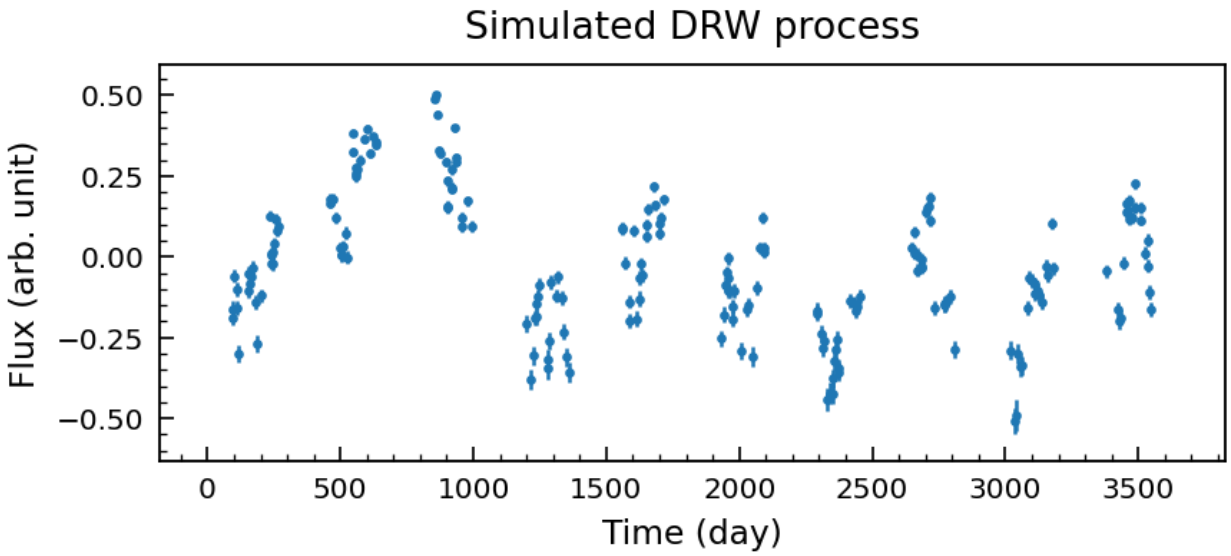
1.2.1 Simulate a DRW process

Here, we are simulating a DRW process with a RMS amplitude of 0.2 and a decorrelation/characteristic timescale of 100 days by first declaring a DRW GP kernel and generating the time series of a signal-to-noise ratio of 10, duration of 10 years and totally 200 data points using *gpSimRand*.

[2]:
```
# initialize DRW kernel, amp is RMS amplitude and tau is the decorrelation timescale
amp = 0.2
tau = 100
DRW_kernel = DRW_term(np.log(amp), np.log(tau))

# generate the time series, y-axis is linear
SNR = 10
duration = 365*10.0
npts = 200
t, y, yerr = gpSimRand(DRW_kernel, SNR, duration, npts, log_flux=False)
```

[3]:
```
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))
ax.errorbar(t, y, yerr, fmt='.')
ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated DRW process')
```

[3]:
```
Text(0.5, 1.0, 'Simulated DRW process')
```



1.2.2 Fit the simulated DRW process

[4]:
```
best_fit = drw_fit(t, y, yerr)
print(f'Best-fit DRW parameter: {best_fit}')
```

```
Best-fit DRW parameter: [ 0.21092082 96.47362586]
```

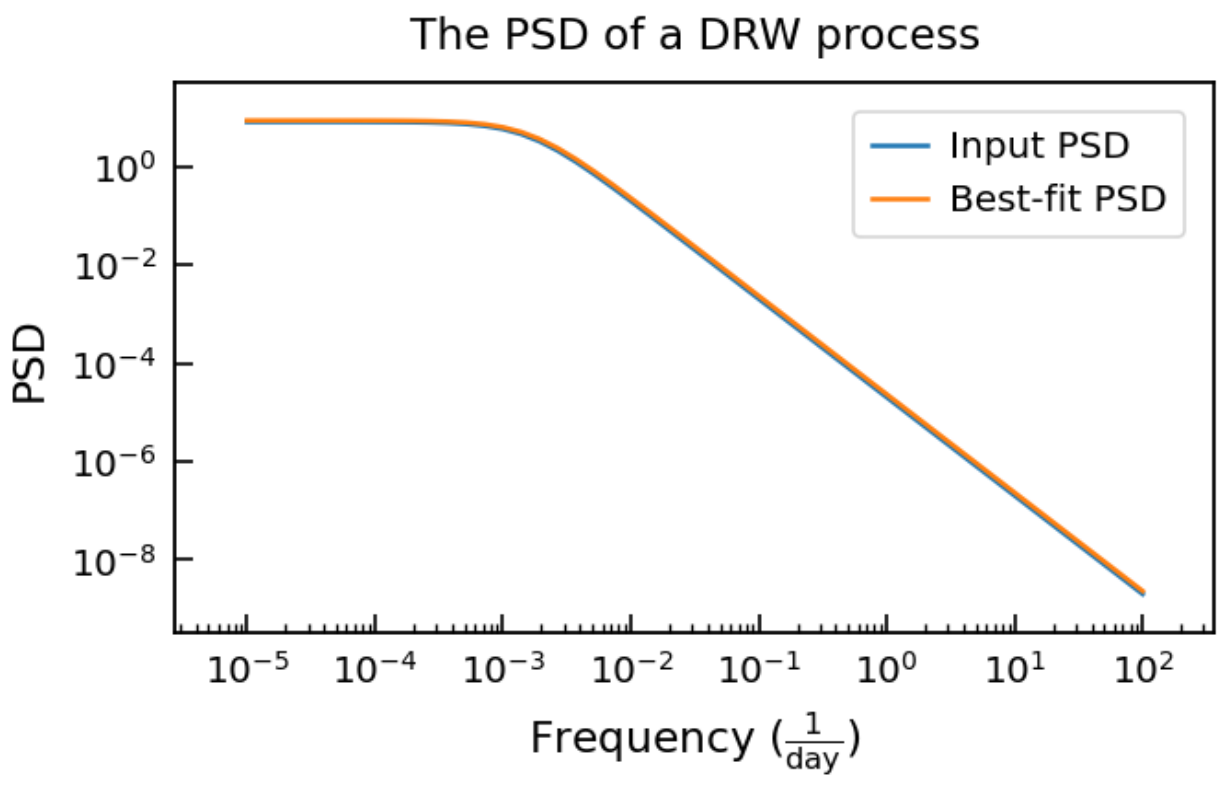
1.2.3 Compare the True PSD with the Best-fit PSD

PSD stands for power spectral density. We will use the PSD function provided the *celerite*.

[5]:
```
# import the GP PSD function from eztao
from eztao.carma import gp_psd
```

[6]:
```
# define the true/best-fit PSD functions using the input kernel
# and a new kernel initalized with the best-fit parameters
best_fit_kernel = DRW_term(*np.log(best_fit))

true_psd = gp_psd(DRW_kernel)
best_psd = gp_psd(best_fit_kernel)
```

[7]:
```
# plot and compare their PSDs
fig, ax = plt.subplots(1,1, dpi=120, figsize=(6,4))
freq = np.logspace(-5, 2)
ax.plot(freq, true_psd(freq), label='Input PSD')
ax.plot(freq, best_psd(freq), label='Best-fit PSD')
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel(r'Frequency ($\frac{1}{\mathrm{day}}$)')
ax.set_ylabel('PSD')
ax.set_title('The PSD of a DRW process', fontsize=14)
ax.legend()
fig.tight_layout()
```



Note

Here, we are using a DRW model for demonstration purpose only, the same analysis can be applied using any CARMA models. We will show that in the next few notebooks.

1.3 Simulations

EzTao provides the tools to easily simulate CARMA processes given a valid (stationary) CARMA kernel. There are three functions in the *eztao.ts.carma_sim* module that can be used to simulate CARMA processes.

- `gpSimFull`: Simulate CARMA processes with a uniform time sampling.
- `gpSimRand`: Simulate CARMA processes with random time sampling (time stamps are drawn from a uniform distribution).
- `gpSimByTime`: Simulate CARMA processes at fixed input time stamps.
- `addNoise`: Add noise to the a simulated CARMA process given input measurement uncertainties.

Each function takes a CARMA kernel as the first argument along with other arguments (please see the *API* for more detail).

Note

1. The above functions require a `SNR` argument, which is defined as the ratio between the variability (RMS) amplitude of the input CARMA model and the median of the measurement errors.
2. The returned array of y values **DO NOT** include simulated errors.
3. The measurement errors are simulated using a log normal distribution and are assigned in a heteroskedastic manner (e.g., small error for small y value if `log_flux` = True; the opposite is true if `log_flux` = False).

Next, we will simulate CARMA processes using a DHO/CARMA(2,1) and a CARMA(5,2) model (no particular reason for choosing these two models).

[1]:
```
# general packages
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline

# eztao imports
import eztao
from eztao.carma import DHO_term, CARMA_term
from eztao.ts import gpSimFull, gpSimByTime, addNoise
from eztao.ts.carma_fit import sample_carma

mpl.rc_file(os.path.join(eztao.__path__[0], "viz/eztao.rc"))
```

1.3.1 Simulate a CARMA(2,1) process

- **At uniformly-spaced time stamps (use `gpSimFull`)**

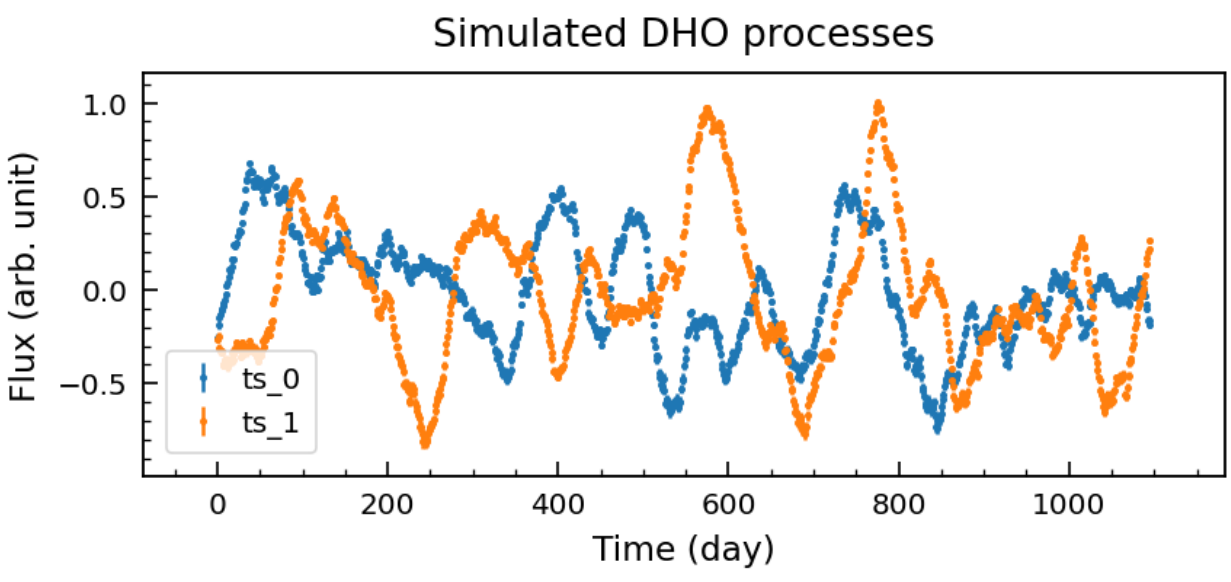
[2]:
```
# define a DHO/CARMA(2,1) kernel
dho_kernel = DHO_term(np.log(0.04), np.log(0.0027941), np.log(0.004672),
                      np.log(0.0257))

# simulate two time series
nLC = 2
SNR = 20
duration = 365*3.0
npts = 1000
t, y, yerr = gpSimFull(dho_kernel, SNR, duration, npts, nLC=nLC, log_flux=False)
```

[3]:
```
print(f'The number returned time series: {y.shape[0]}')
```

```
The number returned time series: 2
```

[4]:
```
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))

for i in range(nLC):
    ax.errorbar(t[i], y[i], yerr[i], fmt='.', label=f'ts_{i}', markersize=4)

ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated DHO processes')
ax.legend(markerscale=1, loc=3)
```

[4]:
```
<matplotlib.legend.Legend at 0x7f8794bf82e0>
```

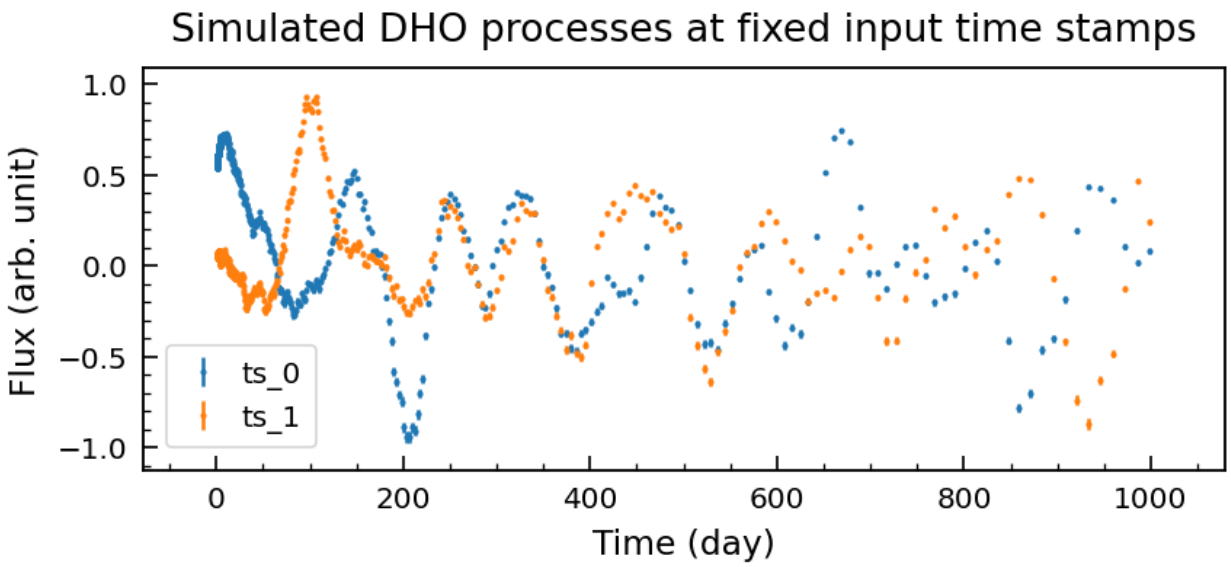


- **At fixed input time stamps (use `gpSimByTime`)**

[5]:
```
# randomly draw time stamps from a log distribution
tIn = np.logspace(0, np.log10(1000), 500)

# simulate
SNR = 20
tOut, yOut, yerrOut = gpSimByTime(dho_kernel, SNR, tIn, nLC=nLC, log_flux=False)
```

[6]:
```
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))

for i in range(nLC):
    ax.errorbar(tOut[i], yOut[i], yerrOut[i], fmt='.', label=f'ts_{i}', markersize=3)

ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated DHO processes at fixed time stamps')
ax.legend(markerscale=1, loc=3)
```

[6]:
```
<matplotlib.legend.Legend at 0x7f87a05fc280>
```



1.3.2 Simulate a CARMA(5,2) process

- **At uniformly-spaced time stamps (use `gpSimFull`)**

[7]:
```
# define a CARMA(5,2) kernel
ARpars = [6.39255585e-01, 8.19334579e-01, 4.74749350e-01,
          4.08631157e-02, 7.22707479e-04]
MApars = [7.04646183, 0.10365114, 0.79552856]
carma_kernel = CARMA_term(np.log(ARpars), np.log(MApars))

# simulate two time series
nLC = 2
SNR = 20
```

(continues on next page)

(continued from previous page)

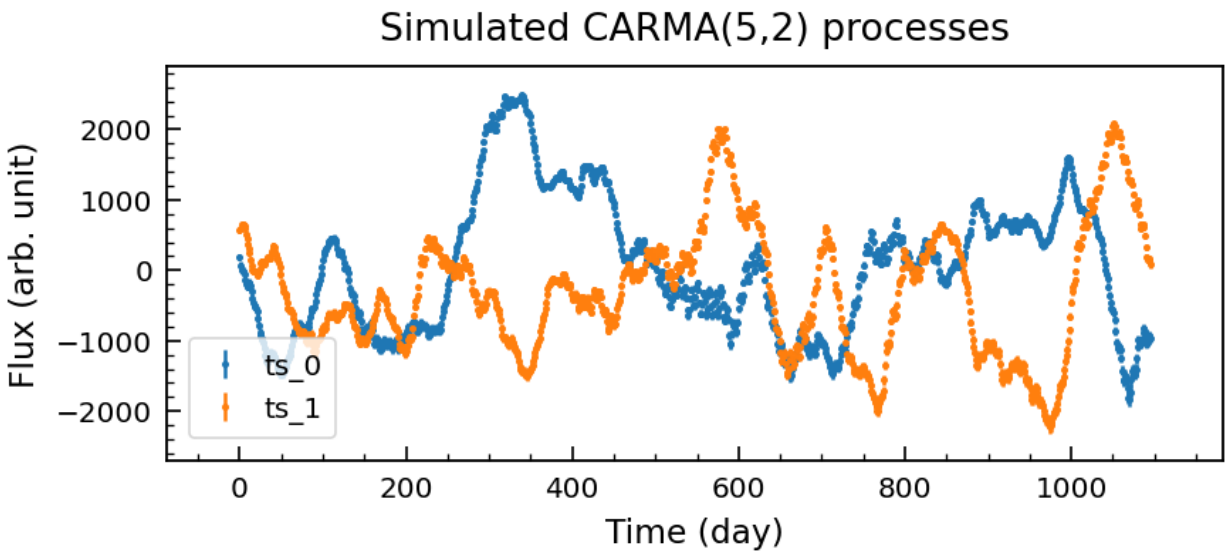
```
duration = 365*3.0
npts = 1000
t, y, yerr = gpSimFull(carma_kernel, SNR, duration, npts, nLC=nLC, log_flux=False)
```

[8]:
```
print(f'The number returned time series: {y.shape[0]}')
```

```
The number returned time series: 2
```

[9]:
```
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))

for i in range(nLC):
    ax.errorbar(t[i], y[i], yerr[i], fmt='.', label=f'ts_{i}', markersize=4)

ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated CARMA(5,2) processes')
ax.legend(markerscale=1, loc=3)
```

[9]:
```
<matplotlib.legend.Legend at 0x7f87a04f18b0>
```



- **At fixed input time stamps (use `gpSimByTime`)**

[10]:
```
# randomly draw time stamps from a log distribution
tIn = np.logspace(0, np.log10(2000), 500)

# simulate
SNR = 20
tOut, yOut, yerrOut = gpSimByTime(carma_kernel, SNR, tIn, nLC=nLC, log_flux=False)
```

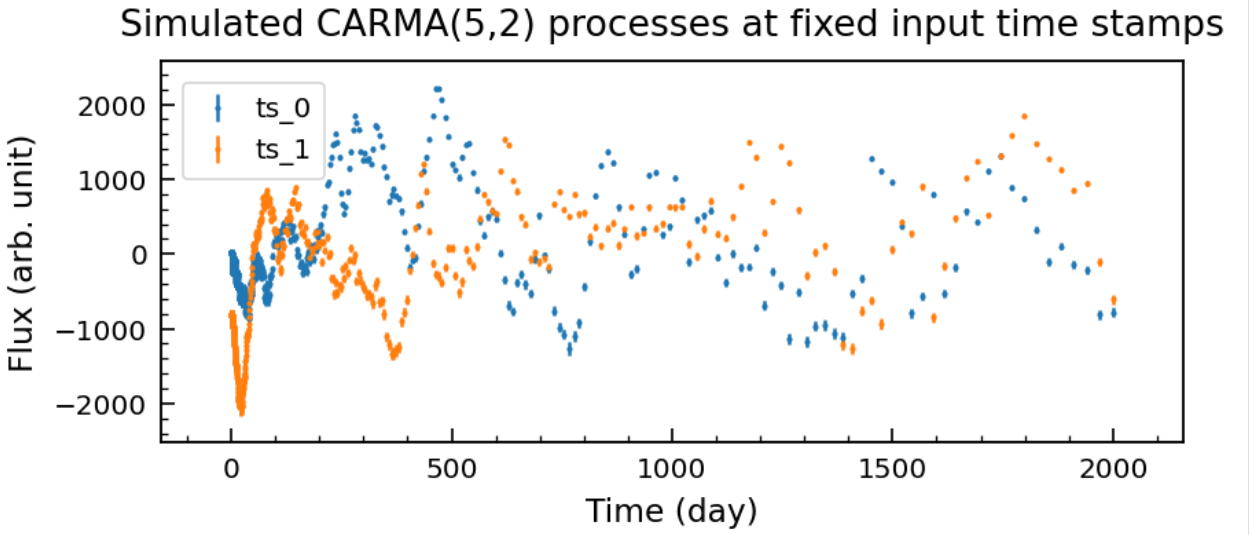
[11]:
```
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))

for i in range(nLC):
```

(continues on next page)

(continued from previous page)

```
    ax.errorbar(tOut[i], yOut[i], yerrOut[i], fmt='.', label=f'ts_{i}', markersize=3)

ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated CARMA(5,2) processes at fixed time stamps')
ax.legend(markerscale=1)
```

[11]:
```
<matplotlib.legend.Legend at 0x7f879ff66c70>
```



Note

For very high-order models (large p and q), extremely high cadence (> 1000 data points/unit time) may introduce numerical instabilities at solving the autocovariance matrix within *celerite*, which means that an error will be thrown. One suggested walk around is changing the time unit in your CARMA parameters (e.g., from day to hour).

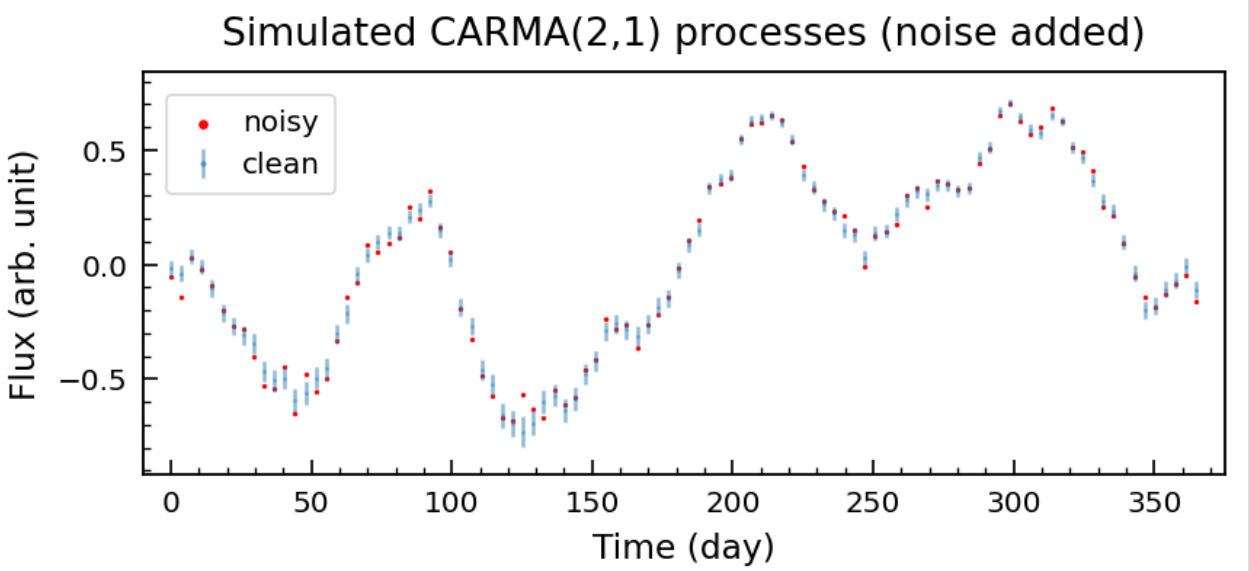
1.3.3 Add noise to a simulated CARMA(2,1) process

- use `addNoise`

[12]:
```
# define a DHO/CARMA(2,1) kernel
dho_kernel = DHO_term(np.log(0.04), np.log(0.0027941), np.log(0.004672),
                      np.log(0.0257))

# simulate a CARMA(2,1) time series
nLC = 1
SNR = 10
duration = 365
npts = 100
t, y, yerr = gpSimFull(dho_kernel, SNR, duration, npts, nLC=nLC, log_flux=False)

# add noise
noisy_y = addNoise(y, yerr)
```

[15]:
```
# overplot the noisy data on top of the 'clean' data
# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))

ax.errorbar(t, y, yerr, fmt='.', markersize=1, alpha=0.5, label='clean')
ax.scatter(t, noisy_y, s=1, label='noisy', color='red')
ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated CARMA(2,1) processes (noise added)')
ax.set_xlim(0-10, duration+10)
ax.legend()
```

[15]:
```
<matplotlib.legend.Legend at 0x7f8790797cd0>
```



1.4 Fitting

Fitting CARMA models to time series data using **EzTao** can be done in just one line. We demonstrate how to quickly get reliable best-fit CARMA parameters using the built-in fitting functions. However, you are not bound to the built-in functions; you can use the CARMA kernels to compute the likelihood and combine that with other optimizers to get the best-fit parameters, which we will also demonstrate.

1.4.1 The built-in fitting functions

There are three built-in fitting functions included in the *eztao.ts.carma_fit* module:

- `drw_fit`: Fit a DRW model to the input time series
- `dho_fit`: Fit a DHO/CARMA(2,1) model to the input time series
- `carma_fit`: Fit an arbitrary CARMA model (must specify p and q order) to the input time series.

The first two should be used if you want to fit either DRW or DHO to the data, and the last one can be used to fit any CARMA models. Next, let's take a closer look at the `carma_fit` function. We will start by simulating a CARMA process and fit using `carma_fit`, then compare the best-fit parameters to the ones used in simulation.

[1]:
```
import numpy as np
from eztao.carma import DHO_term
from eztao.ts import gpSimRand
from eztao.ts import carma_fit

# define a DHO kernel
dho_kernel = DHO_term(np.log(0.04), np.log(0.0027941), np.log(0.004672),
                      np.log(0.0257))

# simulate a DHO process
SNR = 10
duration = 365*10.0
npts = 200
t, y, yerr = gpSimRand(dho_kernel, SNR, duration, npts)
```

[2]:
```
%%time
# fit
p = 2
q = 1
best_fit = carma_fit(t, y, yerr, p, q, n_opt=10)

print(f'True input DHO parameters:{np.exp(dho_kernel.get_parameter_vector())}')
print(f'Best-fit DHO parameters: {best_fit}')
print('-----------------------------------------------------')
```

```
True input DHO parameters:[0.04      0.0027941 0.004672  0.0257   ]
Best-fit DHO parameters: [0.03026792 0.00316738 0.0043432  0.02235601]
-----------------------------------------------------
CPU times: user 1e+03 ms, sys: 0 ns, total: 1e+03 ms
Wall time: 998 ms
```

From the cell above, we can see that it took ~ 1 second (if not shorter) to obtain a best-fit that is quite close to the input. The robust performance is a result of a combination of many carefully chosen fitting components, most importantly, the *celerite* backend for fast evaluation of the likelihood function and a powerful global optimizer for efficient exploration of the parameter space. However, we want to note that to better fit time series to models that are more complex than CARMA(2,1), more robust global optimizer might be needed. In the last section, we will show how to use your own optimizer to obtain best-fit parameters.

Note

The built-in fitting functions use a modified version of the general purpose `scipy.optimize.minimize` optimizer with the 'L-BFGS-B' method. The modification is simply running the optimizing algorithm many more times and each time with a different initialization, as a result, the possibility of getting stuck in a local minimum is significantly reduce. That said, it is still possible to get stuck in a local minimum. Two apparent solutions are: increasing the number of times to run the optimizer through setting the `n_iter` argument or using a more robust optimizer.

Fit CARMA models that are more complex than CARMA(2,1)

Warning

The regular CARMA parameter space (specified by the defining stochastic differential equation, see *Introduction* for a reference) is no longer continuous (meaning giving a stationary process) when p > 2! Read the text below for more details.

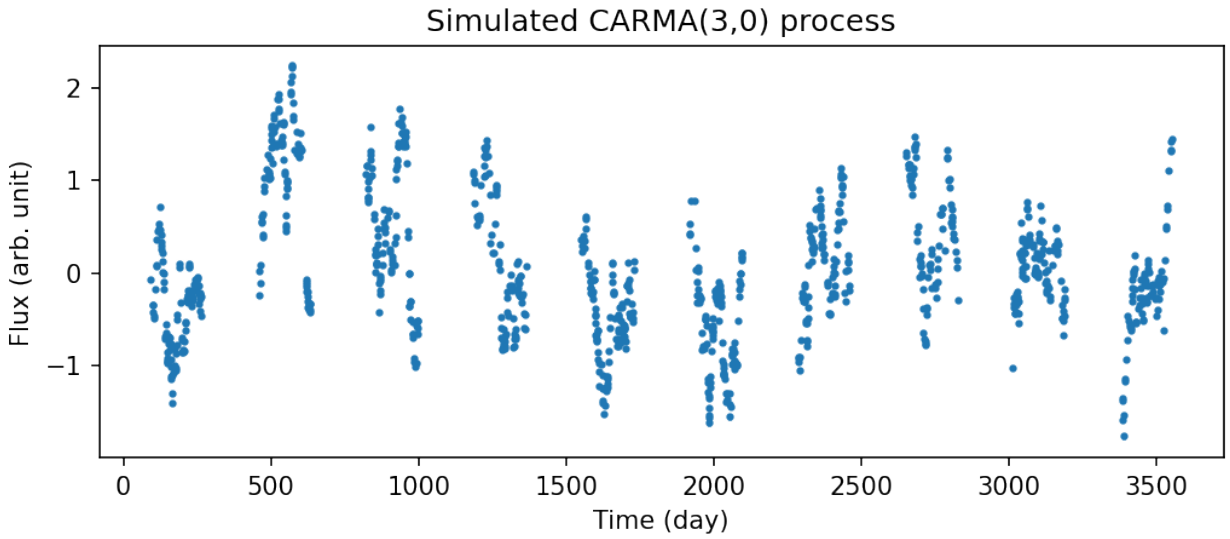
More care must be taken when fitting CARMA models that are more complex than CARMA(2,1). The reason is that the vanilla CARMA parameter space is no longer continuous when p > 2; in other words, a random set of CARMA parameters may not produce a stable CARMA process, thus cannot return a valid likelihood during the fitting process. The solution to this problem is to sample from a space spanned by the coefficients of the AR/MA characteristic polynomials in the factored form. To learn more about this property of CARMA, please check out the papers linked in Introduction.

With **EzTao**, you don't need to worry about the transformation between these two spaces. `carma_fit` will automatically fit in the polynomial space when the p order is greater than 2 and return the best-fit parameters in the regular CARMA space.

[3]:
```
from eztao.carma import CARMA_term

# simulate a CARMA(3,0) process
carma_kernel = CARMA_term(np.log([3, 3.189, 0.05]), np.log([0.5]))

SNR = 50
duration = 365*10.0
npts = 1000
t2, y2, yerr2 = gpSimRand(carma_kernel, SNR, duration, npts, log_flux=False)
```

[4]:
```
import matplotlib.pyplot as plt

# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=150, figsize=(8,3))
ax.errorbar(t2, y2, yerr2, fmt='.', markersize=4)
ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated CARMA(3,0) process')
```

[4]:
```
Text(0.5, 1.0, 'Simulated CARMA(3,0) process')
```



Now, let's fit a CARMA(3,0) model to the time series we just simulated.

[5]:
```
best_fit2 = carma_fit(t2, y2, yerr2, 3, 0, n_opt=20)
print(f'True input CARMA(3,0) parameters:{np.exp(carma_kernel.get_parameter_vector())}
↪')
print(f'Best-fit CARMA(3,0) parameters: {best_fit2}')
print('-----------------------------------------------------')
```

```
True input CARMA(3,0) parameters:[3.    3.189 0.05  0.5  ]
Best-fit CARMA(3,0) parameters: [3.2138727  3.5245589  0.08045436 0.56983312]
-----------------------------------------------------
```

Note

We stress that getting accurate/reliable best-fit parameters for CARMA processes that are more complex than CARMA(2,1) might require more robust optimizers than what's currently implemented. Our experience also tells us that the higher the order of the CARMA model that we are fitting the higher the SNR of the input time series is needed in order to obtain accurate/reliable best-fits.

1.4.2 Use your own optimizer

Using your own optimizer is very straight forward. All you need to do is use your optimizer to generate parameter proposals and use the CARMA kernels from **EzTao** to compose a likelihood function. Caution should be taken when you fit higher-order models (p > 2) for the reasons that I explained above, and we will reemphasize this below. We will use the steps shown in *celerite*'s online tutorials to illustrate this process.

Fit models that are equally or less complex than CARMA(2,1)

The process to fit those three models (DRW, CARMA(2,0) and DHO) is no different than what's shown in *celerite*'s tutorials. You can compose your own likelihood function, `neg_log_like`, like what's being demonstrated; **EzTao** also provides a likelihood function—`neg_param_ll`, where `ll` stands for log likelihood.

[6]:
```
from celerite import GP
from eztao.ts import neg_param_ll
from eztao.ts import sample_carma
from scipy.optimize import minimize
```

Initialize kernel and GP model

[7]:
```
# randomly generate some DHO parameters and use it to define a DHO kernel/GP
dho_ar, dho_ma = sample_carma(2, 1)
dho_kernel_fit = CARMA_term(np.log(dho_ar), np.log(dho_ma))
gp = GP(dho_kernel_fit, mean=np.median(y))
gp.compute(t, yerr)
print("Initial log likelihood: {0}".format(gp.log_likelihood(y)))
```

```
Initial log likelihood: -12242.923471469716
```

Search for the best-fit parameters by minimizing the negative log likelihood

[8]:
```
# optimize to find DHO parameters giving the highest log likelihood (lowest negative␣
↪log likelihood)
initial_params = gp.get_parameter_vector()
bounds = gp.get_parameter_bounds()

r = minimize(neg_param_ll, initial_params, bounds=bounds, method="L-BFGS-B", args=(y,␣
↪gp))
gp.set_parameter_vector(r.x)
```

[9]:
```
print(f'True input DHO parameters:{np.exp(dho_kernel.get_parameter_vector())}')
print(f'Best-fit DHO parameters: {np.exp(r.x)}')
print(f'Final log likelihood: {gp.log_likelihood(y)}')
```

```
True input DHO parameters:[0.04      0.0027941 0.004672  0.0257    ]
Best-fit DHO parameters: [0.03026764 0.0031674  0.00434319 0.02235577]
Final log likelihood: 155.98060793657382
```

Fit models that are more complex than CARMA(2,1)

Fitting higher-order models to a time series needs to use `neg_foceff_ll` instead of `neg_param_ll`. If you would like to compose your own likelihood function, you should use `gp.kernel.set_log_fcoeffs(params)` instead of `gp.set_parameter_vector(params)` for updating the parameters of the CARMA kernel. The current implementation doesn't allow combining CARMA kernels with other kernels (e.g., jitter or a trend), but future versions, especially after switching the backend to *celerite2*, will support this feature.

Initialize kernel and GP model

[10]:
```
from eztao.ts import carma_log_fcoeff_init, neg_fcoeff_ll, sample_carma

# drawing in the polynomial space
init_log_fcoeff = carma_log_fcoeff_init(3, 0)
```

(continues on next page)

(continued from previous page)

```
# convert to CARMA space
init_log_ar, init_log_ma = CARMA_term.fcoeffs2carma_log(init_log_fcoeff, 3)

# initialize kernel, GP
carma_kernel_fit = CARMA_term(init_log_ar, init_log_ma)
gp2 = GP(carma_kernel_fit, mean=np.median(y2))
gp2.compute(t2, yerr2)
print("Initial log likelihood: {0}".format(gp2.log_likelihood(y2)))
```

```
Initial log likelihood: -1022276.573386126
```

Search for the best-fit parameters by minimizing the negative log likelihood

[11]:
```
# give it a 'valid' boundary
ARbounds = [(-8, 8)] * 3
MAbounds = [(-5, 5)] * (0 + 1)
bounds = ARbounds + MAbounds

initial_params = carma_log_fcoeff_init(3, 0)
r2 = minimize(neg_fcoeff_ll, initial_params, bounds=bounds, method="L-BFGS-B",
              args=(y2, gp2))
gp2.kernel.set_log_fcoeffs(r2.x)
```

[12]:
```
print(f'True input DHO parameters:{np.exp(carma_kernel.get_parameter_vector())}')
print(f'Best-fit DHO parameters: {np.exp(gp2.get_parameter_vector())}')
print(f'Final log likelihood: {gp2.log_likelihood(y2)}')
```

```
True input DHO parameters:[3.    3.189 0.05  0.5  ]
Best-fit DHO parameters: [4.37649810e-02 1.18011723e+03 1.31578235e+01 1.11481870e+02]
Final log likelihood: 585.1020341593951
```

A Note on Fitting

- We can only do as good as the likelihood landscape allows (given you are not providing a useful prior landscape).
- The quality of the likelihood landscape (regarding smoothness and the ability to recover true parameters) depends on the properties of the input time series, such as temporal sampling and the level of measurement noise relative the RMS amplitude of the underlying CARMA process.

1.5 MCMC

The maximum likelihood estimation (MLE) only gives you a point estimate, what if you want to get a distribution over the parameter space? That's when Markov Chain Monte Carlo (MCMC) comes into play. **EzTao** provides a simple function, *eztao.ts.carma_mcmc.mcmc*, to quickly run MCMC using *emcee* (the MCMC hammer). However, since it is just a wrapper, it has very limited options. A more advanced/flexible solution is to use **EzTao** for likelihood computation and hook it with a MCMC sampler of your choice.

In this notebook, we will show how to run MCMC using **EzTao**'s built-in MCMC function as well as your own MCMC sampler.

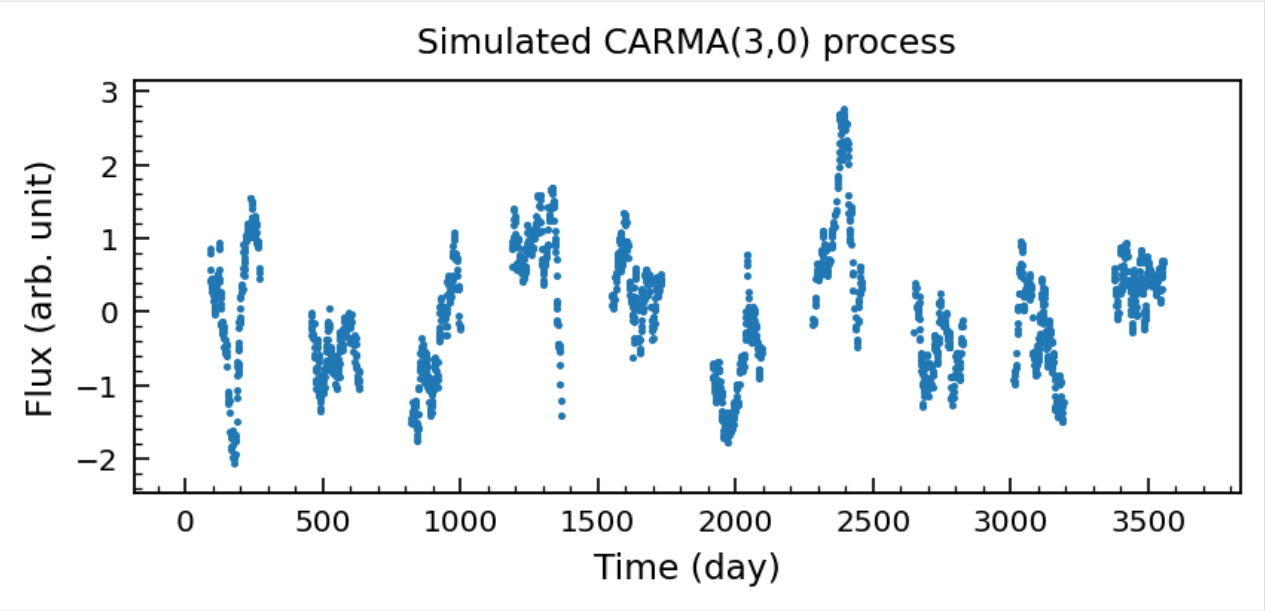
1.5.1 Built-in MCMC

[1]:
```
# general packages
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline

# eztao, emcee, corner
import eztao
from eztao.carma import CARMA_term
from eztao.ts import gpSimRand
from eztao.ts import mcmc
import emcee
import corner

mpl.rc_file(os.path.join(eztao.__path__[0], "viz/eztao.rc"))
```

Simulate a CARMA(3,0) process and plot the time series

[2]:
```
# simulate a CARMA(3,0) process
p = 3
q = 0
SNR = 100
duration = 365*10.0
npts = 2000
carma30_kernel = CARMA_term(np.log([3, 3.189, 0.05]), np.log([0.5]))
t, y, yerr = gpSimRand(carma30_kernel, SNR, duration, npts, log_flux=False)

# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))
ax.errorbar(t, y, yerr, fmt='.', markersize=4)
ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated CARMA(3,0) process', fontsize=14)
```

[2]:
```
Text(0.5, 1.0, 'Simulated CARMA(3,0) process')
```

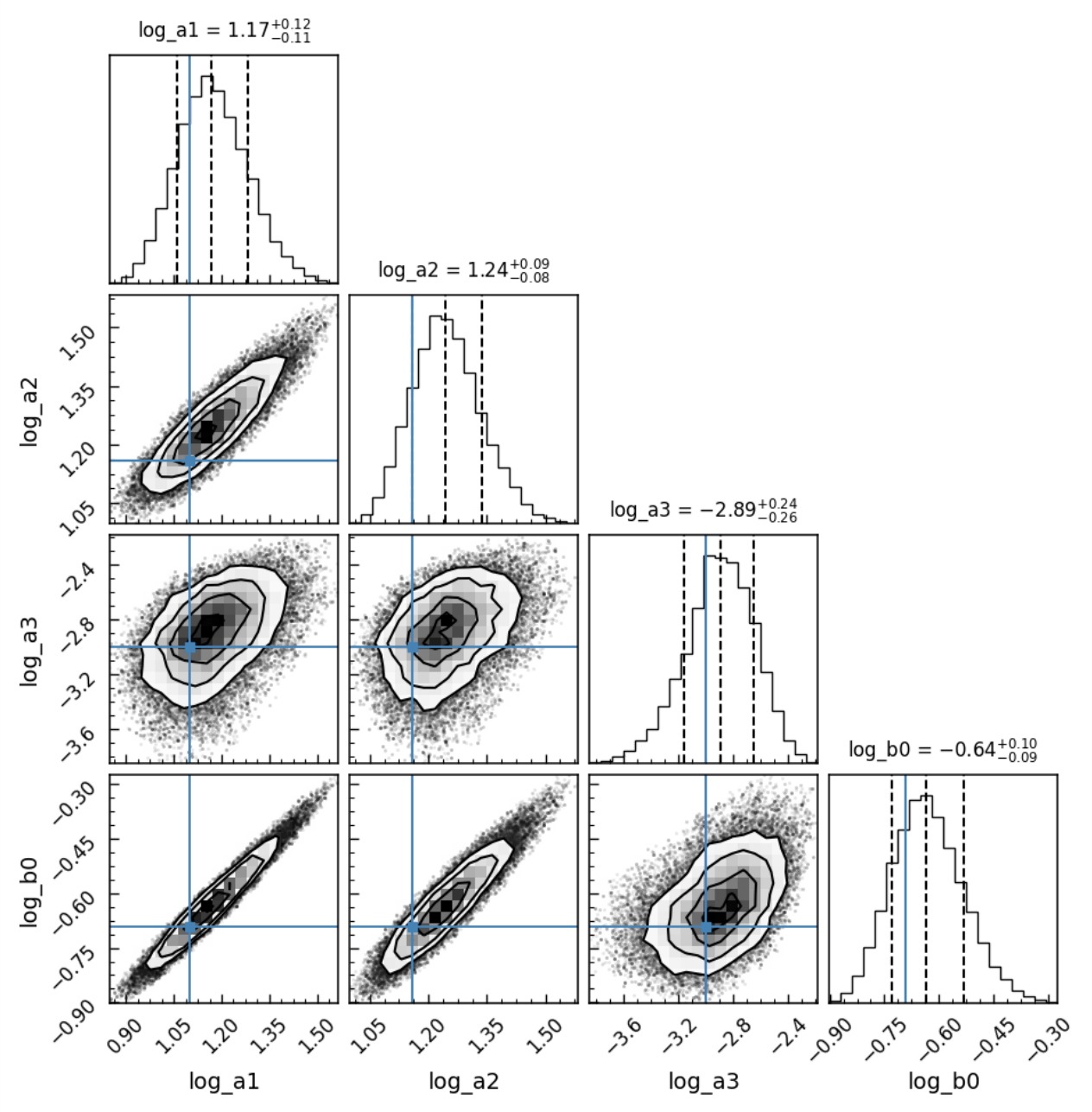


Run MCMC using the built-in function & Generate a corner plot

[3]:
```
# use built-in function to run MCMC
sampler, carma_flatchain, carma_chain = mcmc(t, y, yerr, p, q)
```

```
Searching for best-fit CARMA parameters...
Running burn-in...
Running production...
```

[4]:
```
# remove points with low prob for the sake of making good corner plot
prob_threshold = np.percentile(sampler.flatlnprobability, 5)
clean_chain = carma_flatchain[sampler.flatlnprobability > prob_threshold, :]

# make corner plot
labels = [name for name in carma30_kernel.get_parameter_names()]
corner.corner(clean_chain, truths=carma30_kernel.get_parameter_vector(),
              quantiles=[0.16, 0.5, 0.84], labels=labels, show_titles=True,
              title_kwargs={"fontsize": 12});
```



Note

The `mcmc` function returns three variables, the `emcee` sampler object, the flatchain and the chain in regular CARMA space. For p > 2, given the reasons mentioned in the *Fiitting* notebook (CARMA space vs. polynomial space), the MCMC sampler runs in the polynomial space (which regular user does not need to worry about), `mcmc` returns the chain and flatchain separately from the `emcee` sampler. If the p order is 2 or smaller, the last two variables will just be empty arrays.

1.5.2 Use your own MCMC sampler

Here, I will use `emcee` for demonstration. To run MCMC with `emcee`, you need to define your own probability function. You can either write a simple wrapper around the **EzTao** likelihood function, `neg_param_ll` or `neg_fcoeff_ll` (depending on the p order of the CARMA model), to return the `POSITIVE` log likelihood, or write your own probability function.

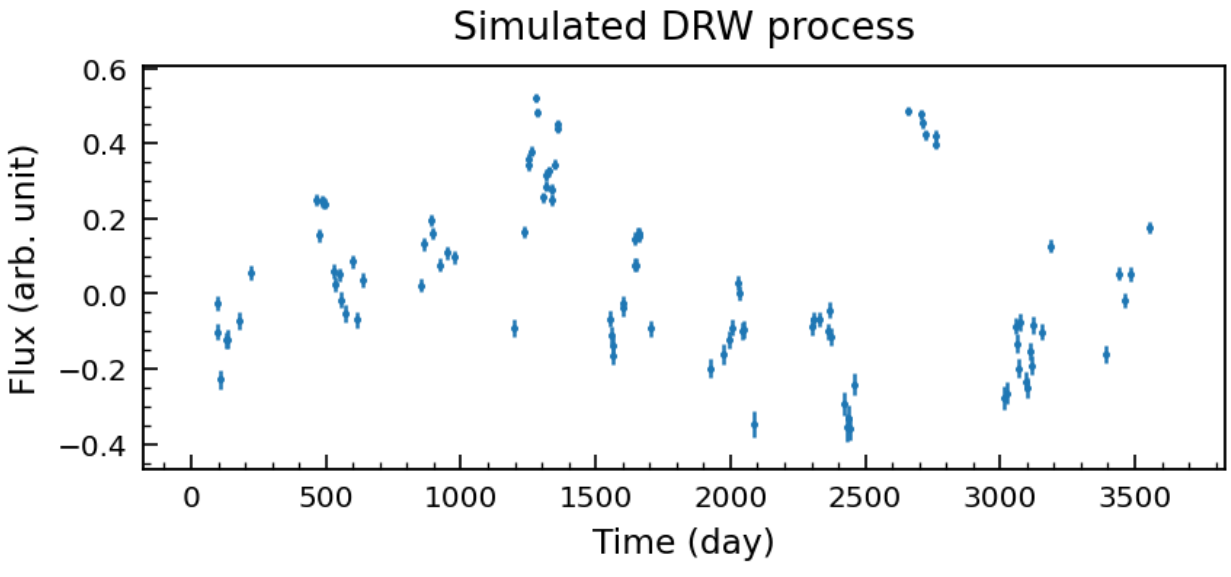
p <= 2

[5]:
```
from eztao.ts import neg_param_ll, drw_fit
from eztao.carma import DRW_term
from celerite import GP
```

Simulate a DRW process and plot the time series

[6]:
```
# simulate a DRW process
amp = 0.2
tau = 120
drw_kernel = DRW_term(np.log(amp), np.log(tau))

SNR = 10
duration = 365*10.0
npts = 100
t2, y2, yerr2 = gpSimRand(drw_kernel, SNR, duration, npts, log_flux=False)

# plot the simulated process
fig, ax = plt.subplots(1,1, dpi=120, figsize=(8,3))
ax.errorbar(t2, y2, yerr2, fmt='.', markersize=4)
ax.set_xlabel('Time (day)')
ax.set_ylabel('Flux (arb. unit)')
ax.set_title('Simulated DRW process')
```

[6]:
```
Text(0.5, 1.0, 'Simulated DRW process')
```



Obtain best-fit parameters and use that to initialize a GP model

[7]:
```
# obtain best-fit
best_drw = drw_fit(t2, y2, yerr2)
print(f'Best-fit DRW: {best_drw}')

# define celerite GP model
drw_gp = GP(DRW_term(*np.log(best_drw)), mean=np.median(y2))
drw_gp.compute(t2, yerr2)
```

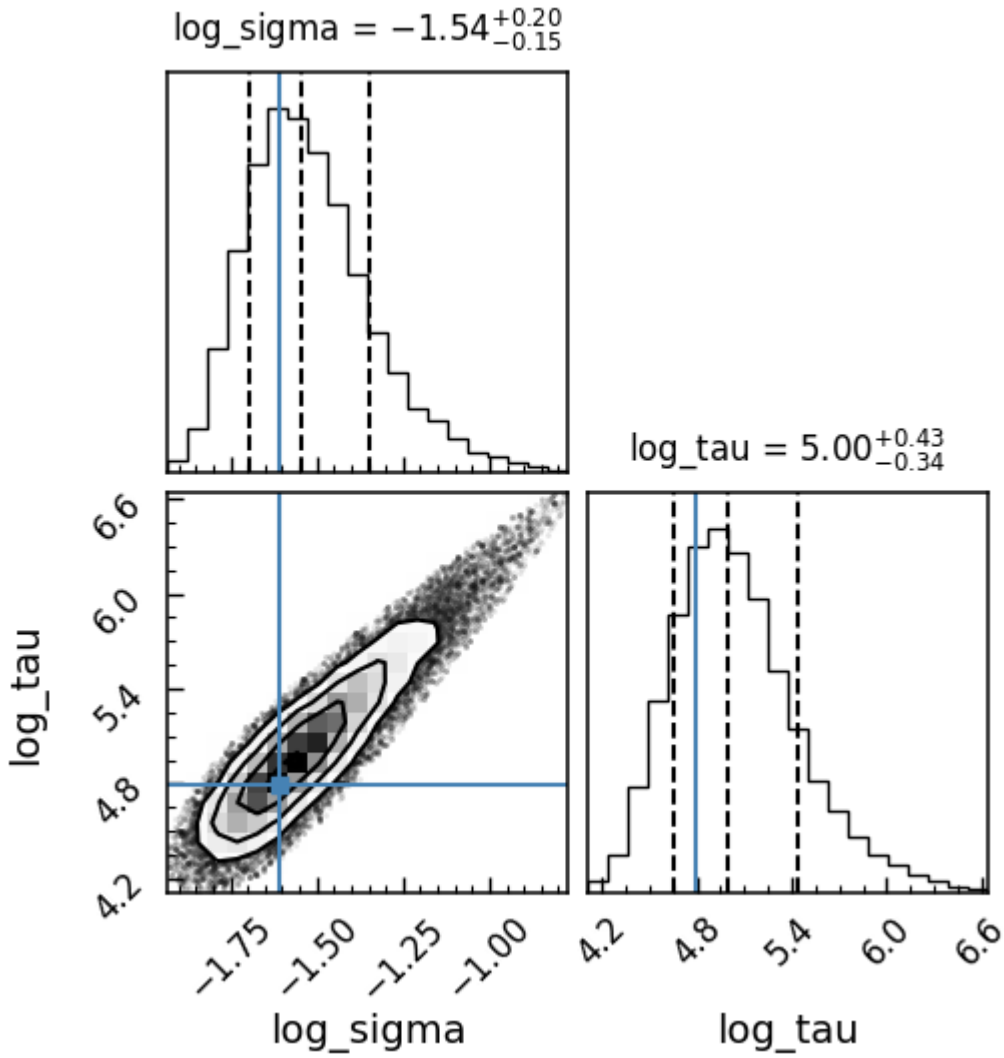
```
Best-fit DRW: [  0.20333737 134.83935452]
```

Run MCMC & Generate a corner plot

[8]:
```
# define log prob function
def param_ll(*args):
    return -neg_param_ll(*args)

# initialize the walker, specify number of walkers, prob function, args and etc.
initial = np.array(np.log(best_drw))
ndim, nwalkers = len(initial), 32
sampler_drw = emcee.EnsembleSampler(nwalkers, ndim, param_ll, args=[y2, drw_gp])

# run a burn-in surrounding the best-fit parameters obtained above
print("Running burn-in...")
p0 = initial + 1e-8 * np.random.randn(nwalkers, ndim)
p0, lp, _ = sampler_drw.run_mcmc(p0, 500)

# clear up the stored chain from burn-in, rerun the MCMC
print("Running production...")
sampler_drw.reset()
sampler_drw.run_mcmc(p0, 2000);
```

```
Running burn-in...
Running production...
```

[9]:
```
# remove points with low prob for the sake of making good corner plot
prob_threshold_drw = np.percentile(sampler_drw.flatlnprobability, 3)
clean_chain_drw = sampler_drw.flatchain[sampler_drw.flatlnprobability > prob_
↪threshold_drw, :]

# make corner plot
labels = [name for name in drw_gp.kernel.get_parameter_names()]
corner.corner(clean_chain_drw, truths=drw_kernel.get_parameter_vector(),
              quantiles=[0.16, 0.5, 0.84], labels=labels, show_titles=True,
              title_kwargs={"fontsize": 12});
```



Note

It is always a good practice to start your MCMC walker at a position that is close to the `truth` (assuming the best-fit is a good estimation) as MCMC is a sampler not an optimizer.

p > 2

In this section, we will reuse the simulated CARMA(3,0) process in *Built-in MCMC*. Since p > 2, we need to start and run MCMC in the factored polynomial space of CARMA, and we will also need to transform the chain from the polynomial space to regular CARMA space once it is done. **EzTao** provides the functions (see *here*) to facilitate this process as we will see below.

Obtain best-fit CARMA parameters & Convert them into the polynomial space

[10]:
```
from eztao.ts import carma_fit
```

(continues on next page)

(continued from previous page)

```
# obtain best-fit as intial position for MCMC
best_cm30 = carma_fit(t, y, yerr, p, q, n_opt=50)
print(f'Best-fit CARMA(3,0) parameters:{best_cm30}')

# get the representation of the best-fit in the polynomial space
best_fcoeffs_log = CARMA_term.carma2fcoeffs_log(np.log(best_cm30[:p]),
                                                np.log(best_cm30[p:]))
best_fcoeffs = np.exp(best_fcoeffs_log)
print(f'Best-fit in polynomial space: {best_fcoeffs}')
```

```
Best-fit CARMA(3,0) parameters:[3.18331428 3.44275143 0.05733812 0.52477284]
Best-fit in polynomial space: [3.1663963  3.3891824  0.01691798 0.52477284]
```

Initialize a celerite GP model

[11]:
```
from eztao.ts import neg_fcoeff_ll

# init GP
cm30_kernel_mcmc = CARMA_term(np.log(best_cm30)[:p], np.log(best_cm30)[p:])
cm30_gp = GP(cm30_kernel_mcmc, mean=np.median(y))
cm30_gp.compute(t, yerr)
```

Run MCMC

[12]:
```
# define log prob function
def fcoeff_ll(*args):
    return -neg_fcoeff_ll(*args)

# run MCMC
initial_cm30 = np.log(best_fcoeffs)
ndim, nwalkers = len(initial_cm30), 32
sampler_cm30 = emcee.EnsembleSampler(nwalkers, ndim, fcoeff_ll, args=[y, cm30_gp])

print("Running burn-in...")
p0 = initial_cm30 + 1e-8 * np.random.randn(nwalkers, ndim)
p0, lp, _ = sampler_cm30.run_mcmc(p0, 500)

print("Running production...")
sampler_cm30.reset()
sampler_cm30.run_mcmc(p0, 2000);
```

```
Running burn-in...
Running production...
```

Convert the chain from the polynomial space to the CARMA space

[13]:
```
# from the polynomial space to CARMA space
vec_fcoeff2carma_log = np.vectorize(CARMA_term.fcoeffs2carma_log, excluded=[1,],
                                    signature="(n)->(m),(k)")
logAR, logMA = vec_fcoeff2carma_log(sampler_cm30.flatchain, p)
cm30_flatchain = np.hstack((logAR, logMA))
```

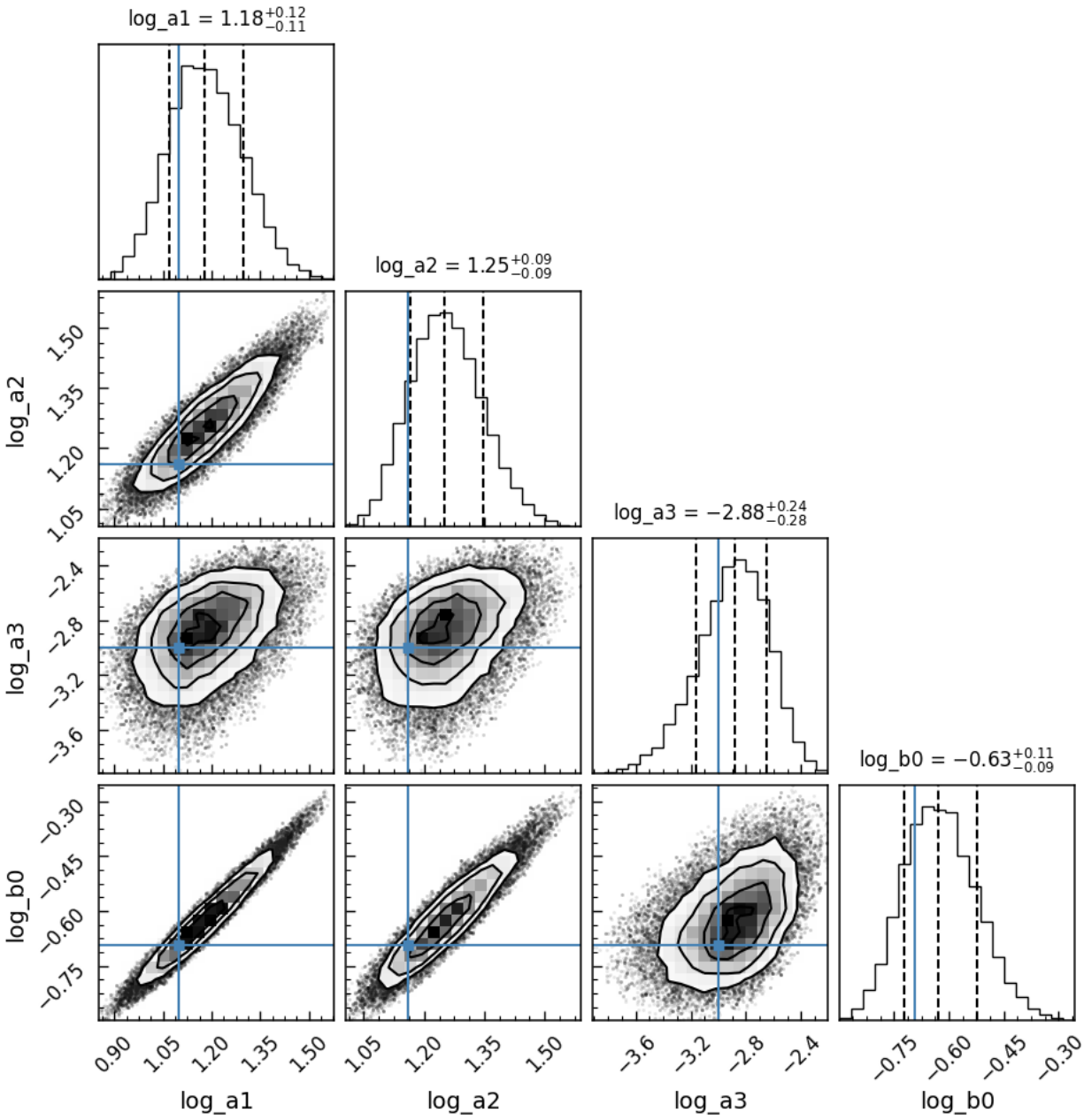
Make a corner plot

[14]:
```
# remove points with low prob for the sake of making good corner plot
prob_threshold_cm30 = np.percentile(sampler_cm30.flatlnprobability, 5)
clean_chain_cm30 = cm30_flatchain[sampler_cm30.flatlnprobability > prob_threshold_
↪cm30, :]
```

(continues on next page)

(continued from previous page)

```
# make corner plot
labels = [name for name in carma30_kernel.get_parameter_names()]
corner.corner(clean_chain_cm30, truths=carma30_kernel.get_parameter_vector(),
              quantiles=[0.16, 0.5, 0.84], labels=labels, show_titles=True,
              title_kwargs={"fontsize": 12});
```



Note:

Extra cautions should be taken when running MCMC with CARMA models of p > 2:

1. Use `neg_fcoeff_ll` as the base log likelihood function.

2. Remember to transform best-fit parameters to the polynomial space when initializing the walker and the chain back to the regular CARMA space once the MCMC is finished.

1.6 Utility Functions

EzTao also provides a series of tools to help you model and understand time series data using CARMA. They are in two categories: `visualization` tools and functions to compute `2nd order` statistics.

1.6.1 Visualization tools (eztao.viz.mpl_viz)

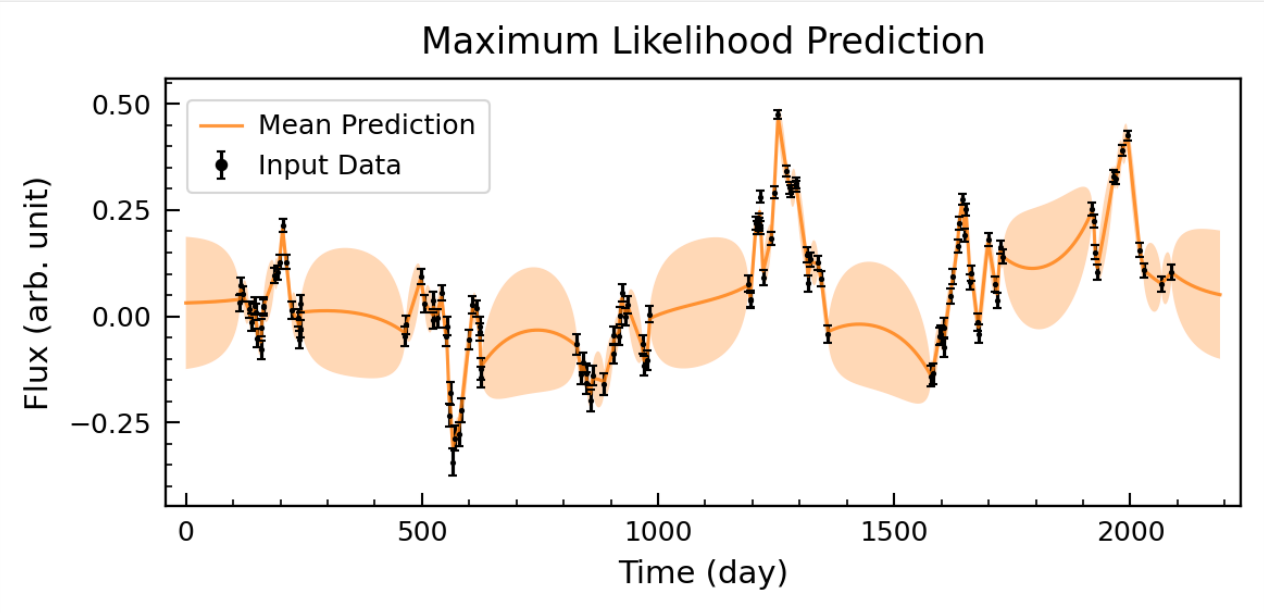
- `plot_pred_lc`: Plotting the predicted time series given best-fit parameters conditioned on the input time series.
- `plot_drw_ll`: Plotting the log likelihood landscape of a DRW model.
- `plot_dho_ll`: Plotting the log likelihood landscape of a DHO/CARMA(2,0) model

[1]:
```
import numpy as np
from eztao.carma import DRW_term
from eztao.ts import gpSimRand
from eztao.ts import drw_fit
from eztao.viz import plot_pred_lc, plot_drw_ll
```

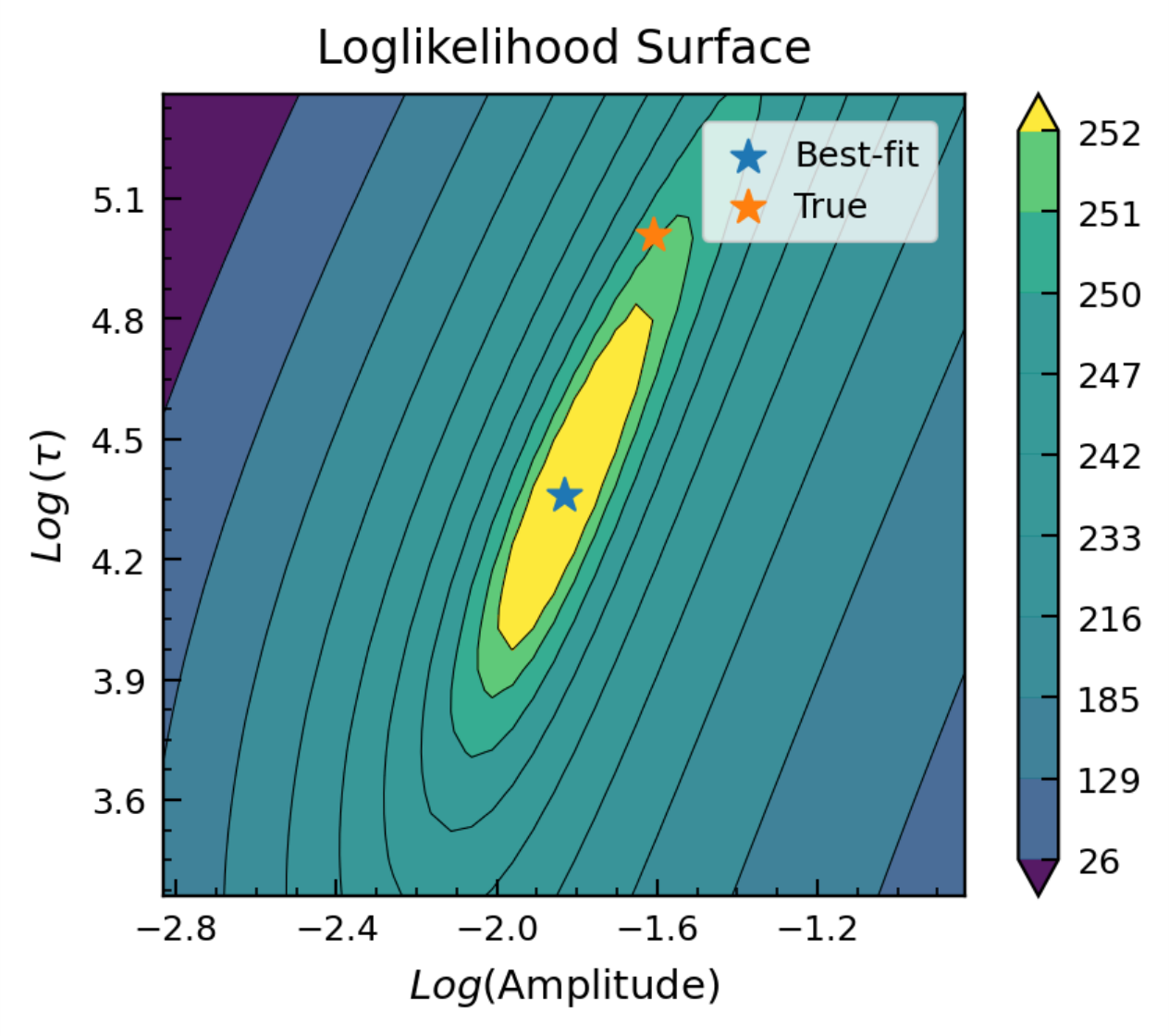
[2]:
```
# initialize a DRW kernel
amp = 0.2
tau = 150
DRW_kernel = DRW_term(np.log(amp), np.log(tau))

# simulate a process using the above model
SNR = 10
duration = 365*10.0
npts = 200
t, y, yerr = gpSimRand(DRW_kernel, SNR, duration, npts, log_flux=False)

# fit
best_fit = drw_fit(t, y, yerr)
print(f'Best-fit DRW parameter: {best_fit}')
```

```
Best-fit DRW parameter: [ 0.16008592 78.28325798]
```

[3]:
```
## plot predicted time series
t_pred = np.linspace(0, 365*6, 2000)

# get best-fit in CARMA space
best_fit_kernel = DRW_term(*np.log(best_fit))
best_fit_arma = best_fit_kernel.get_carma_parameter()

plot_pred_lc(t, y, yerr, best_fit_arma, 1, t_pred)
```



[4]:
```
## plot log likelihood surface
from eztao.ts import neg_param_ll
from celerite import GP

gp = GP(DRW_kernel, mean=np.median(y))
plot_drw_ll(t, y, yerr, best_fit, gp, neg_param_ll, true_params=[amp, tau])
```



2nd Order Statistics (eztao.carma.model_utils)

Given a valid CARMA kernel, you can generate 2nd order statistics at a range of timescales/frequencies. In this section, we will use a DHO/CARMA(2,1) model for demonstration. For a reference of how those statistics can be useful for analyzing time series data, feel free to check out Moreno et al. 2019.

- **PSD**: Power spectrum density
- **ACF**: Auto-correlation function
- **SF**: Structure function

[5]:
```
from eztao.carma import CARMA_term
from eztao.carma import carma_acf, carma_psd, carma_sf
```

Create the PSD, ACF and SF functions given some DHO parameters

[6]:
```
ar = np.array([0.04, 0.0027941])
ma = np.array([0.004672, 0.0257])

psd = carma_psd(ar, ma)
acf = carma_acf(ar, ma)
sf = carma_sf(ar, ma)
```

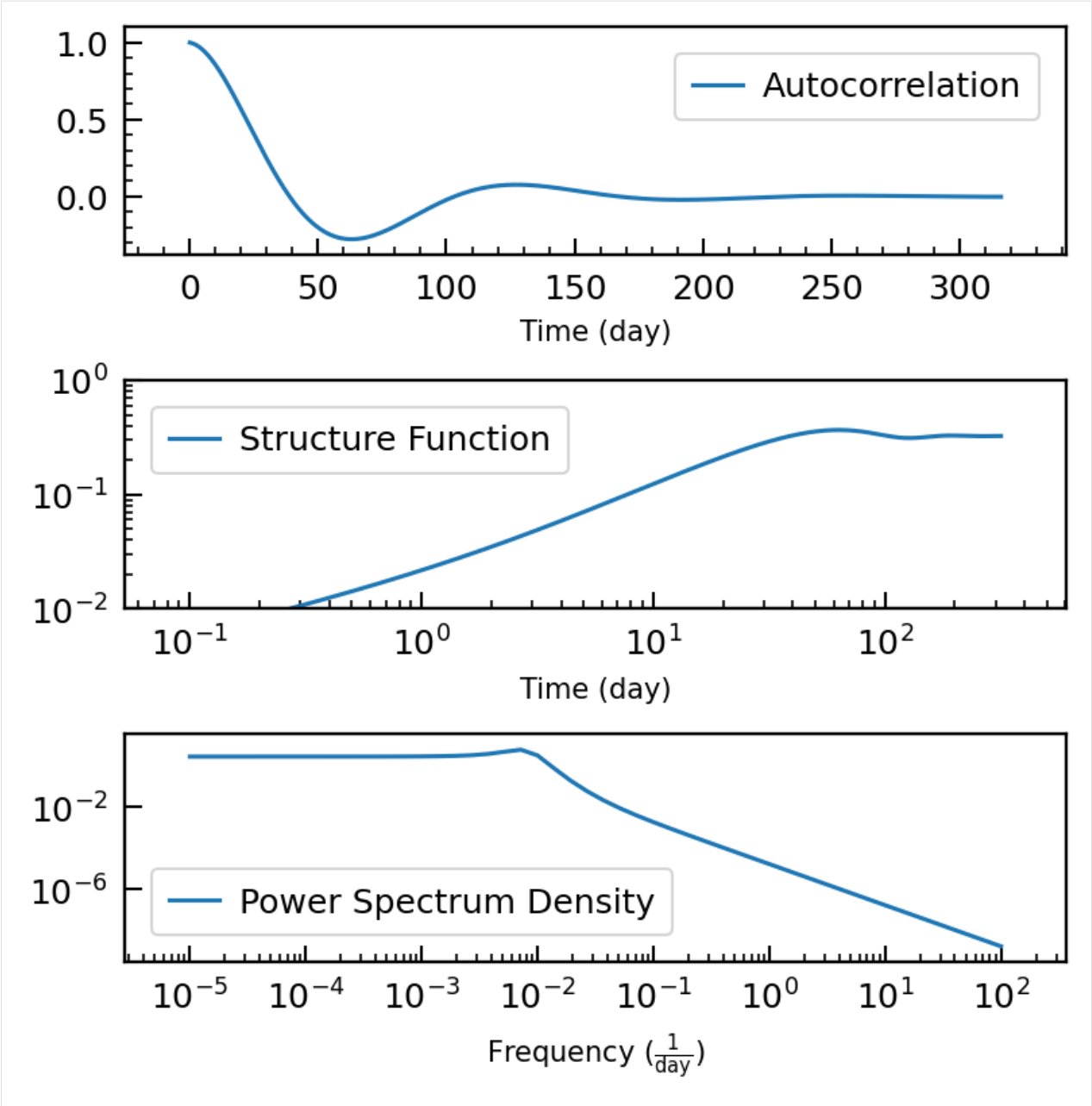
Define a range of time scales and frequencies

[7]:
```
t = np.logspace(-1, 2.5, 1000)
t = np.insert(t, 0, 0)
freq = np.logspace(-5, 2)
```

Next, let's try to plot them

[8]:
```
import matplotlib.pyplot as plt
```

[9]:
```
fig, ax = plt.subplots(3, 1, figsize=(6, 6), dpi=150)

ax[0].plot(t, acf(t), label='Autocorrelation')
ax[0].set_xlabel('Time (day)', size=10, labelpad=5)
ax[0].legend()

ax[1].plot(t, sf(t), label='Structure Function')
ax[1].set_xlabel('Time (day)', size=10, labelpad=5)
ax[1].set_xscale('log')
ax[1].set_yscale('log')
ax[1].set_ylim((10e-3, 1))
ax[1].legend()

ax[2].plot(freq, psd(freq), label='Power Spectrum Density')
ax[2].set_xlabel(r'Frequency ($\mathrm{\frac{1}{day}}$)', size=10, labelpad=5)
ax[2].set_xscale('log')
ax[2].set_yscale('log')
ax[2].legend()

fig.subplots_adjust(hspace=0.55)
```



1.7 CARMA

1.7.1 CARMA GP kernels

Note: *term* is a *celerite* nomenclature, a *term* is just a GP kernel. For example, a `DRW_term` object defines a DRW kernel.

A collection of GP kernels that express the autovariance structure of CARMA models using celerite.

class eztao.carma.CARMATerm.**CARMA_term**(*log_arpars*, *log_mapars*, **args*, ***kwargs*)
A general-purpose CARMA term (kernel)

Parameters

- **log_arpars** (*array (float)*) – Natural log of the AR coefficients.
- **log_mapars** (*array (float)*) – Natural log of the MA coefficients.

static carma2fcoeffs(*log_arpars*, *log_mapars*)
Get the representation of a CARMA kernel in the factored polynomial space

A wrapper of *CARMA_term.carma2fcoeffs_log* for backward compatibility. This function will be deprecated in future releases.

static carma2fcoeffs_log(*log_arpars*, *log_mapars*)
Get the representation of a CARMA kernel in the factored polynomial space

Parameters

- **log_arpars** (*array (float)*) – Natural log of the AR coefficients.
- **log_mapars** (*array (float)*) – Natural log of the MA coefficients.

Returns

The coefficients (in natural log) of the factored polymoical for the CARMA kernel specified by the input parameters. The last coeff is a multiplying factor of the returned polynomial.

Return type array(float)

static fcoeffs2carma(*log_fcoeffs*, *p*)
Get the representation of a CARMA kernel in the nominal CARMA parameter space

A wrapper of *CARMA_term.fcoeffs2carma_log* for backward compatibility. This function will be deprecated in future releases.

static fcoeffs2carma_log(*log_fcoeffs*, *p*)
Get the representation of a CARMA kernel in the nominal CARMA parameter space

Parameters

- **log_coeffs** (*array (float)*) – The array of coefficients for the factored polynomial with the last coeff being a multiplying factor of the polynomial.
- **p** (*int*) – The p order of the CARMA kernel.

Returns Natural log of the AR and MA parameters in two separate arrays.

get_complex_coefficients(*params*)
Get arrays of `alpha_complex_real`, `alpha_complex_imag`, `beta_complex_real` and `beta_complex_imag` (coefficients of celerite's complex kernel)

Parameters **params** (*array (float)*) – Parameters of this kernel.

Returns Arrays of `alpha_complex_real`, `alpha_complex_imag`, `beta_complex_real` and `beta_complex_imag`, one for each.

get_real_coefficients(*params*)
Get arrays of `alpha_real` and `beta_real` (coefficients of celerite's real kernel)

Parameters **params** (*array (float)*) – Parameters of this kernel.

Returns Arrays of `alpha_real` and `beta_real`, one for each.

get_rms_amp()
Get the RMS amplitude of this CARMA kernel

Returns The RMS amplitude of this CARMA kernel.

static rms_amp(*log_arpars*, *log_mapars*)
Compute the RMS amplitude of a CARMA kernel

Parameters

- **log_arpars** (*array (float)*) – Natural log of the AR coefficients.
- **log_mapars** (*array (float)*) – Natural log of the MA coefficients.

Returns The RMS amplitude of the CARMA kernel specified by the input parameters.

set_log_fcoeffs(*log_fcoeffs*)
Set kernel parameters

Use coeffs of the factored polynomial to set CARMA paramters, note that the last input coeff is always the coeff for the highest-order MA differential. While performing the conversion, 1 is added to the AR coeffs to maintain the same formatting (AR polynomials always have the highest order coeff to be 1).

Parameters **log_fcoeffs** (*array (float)*) – Natural log of the coefficients for the factored characteristic polynomial, with the last coeff being an additional multiplying factor on this polynomial.

class eztao.carma.CARMATerm.**DHO_term**(*log_a1*, *log_a2*, *log_b0*, *log_b1*, **args*, ***kwargs*)
Damped Harmonic Oscillator (DHO) term (kernel)

Parameters

- **log_a1** (*float*) – Natural log of the DHO parameter a1.
- **log_a2** (*float*) – Natural log of the DHO parameter a2.
- **log_b0** (*float*) – Natural log of the DHO parameter b0.
- **log_b1** (*float*) – Natural log of the DHO parameter b1.

class eztao.carma.CARMATerm.**DRW_term**(**args*, ***kwargs*)
Damped Random Walk (DRW) term (kernel)

$$k_{DRW}(\Delta t) = \sigma^2 \, e^{-\Delta t/\tau}$$

with the parameters `log_amp` and `log_tau`.

Parameters

- **log_amp** (*float*) – The natural log of the RMS amplitude of the DRW process.
- **log_tau** (*float*) – The natural log of the characteristic timescale of the DRW process.

Note: Conversions between EzTao DRW parameters and some other DRW representations seen in the literature:

$$\mathrm{SF}_\infty = 2 * \sigma^2$$

$$\sigma^2 = \tau * \sigma^2_{KBS}/2$$

$$\tau = 1/\alpha_1;\ \sigma_{KBS} = \beta_0$$

see MacLeod et al. (2010) for SF_{infty}} and Kelly et al. (2009) for sigma_{KBS}. alpha_1 and beta_0 are the AR and MA parameters for a CARMA(1,0) model, respectively.

---

get_carma_parameter()
Get DRW parameters in CARMA notation (alpha_*/beta_*).

Returns [alpha_1, beta_0].

get_perturb_amp()
Get the amplitude of the perturbing noise (beta_0) in DRW

Returns The amplitude of the perturbing noise (beta_0) in the current DRW.

get_real_coefficients(*params*)
Get `alpha_real` and `beta_real` (coeffs of celerite's real kernel)

Parameters **params** (*array (float)*) – Parameters of this kernel.

Returns (`alpha_real`, `beta_real`).

get_rms_amp()
Get the RMS amplitude of this DRW process.

Returns The RMS amplitude of this DRW process.

static perturb_amp(*log_amp*, *log_tau*)
Compute the amplitude of the perturbing noise (beta_0) in DRW.

Parameters

- **log_amp** (*float*) – The natural log of the RMS amplitude of the DRW process.
- **log_tau** (*float*) – The natural log of the characteristic timescale of the DRW process.

Returns The amplitude of the perturbing noise (beta_0) in the DRW specified by the input parameters.

eztao.carma.CARMATerm.**acf**(*arroots*, *arparam*, *maparam*)
Get ACVF coefficients given CARMA parameters

The CARMA noation (index) folows that in Brockwell et al. (2001).

Parameters

- **arroots** (*array (complex)*) – AR roots in a numpy array
- **arparam** (*array (float)*) – AR parameters in a numpy array
- **maparam** (*array (float)*) – MA parameters in a numpy array

Returns ACVF coefficients, each element correspond to a root.

Return type array(complex)

1.7.2 CARMA utility functions

A set of functions to computer 2nd order statistics of CARMA models.

eztao.carma.model_utils.**carma_acf**(*arparams*, *maparams*)
Return a function that computes the model autocorrelation function (ACF) of CARMA.

Parameters

- **arparams** (*array (float)*) – AR coefficients.
- **maparams** (*array (float)*) – MA coefficients.

Returns A function that takes in time lags and returns ACF at the given lags.

eztao.carma.model_utils.**carma_psd**(*arparams*, *maparams*)
Return a function that computes CARMA Power Spectral Density (PSD).

Parameters

- **arparams** (*array (float)*) – AR coefficients.
- **maparams** (*array (float)*) – MA coefficients

Returns A function that takes in frequencies and returns PSD at the given frequencies.

eztao.carma.model_utils.**carma_sf**(*arparams*, *maparams*)
Return a function that computes the CARMA structure function (SF).

Parameters

- **arparams** (*array (float)*) – AR coefficients.
- **maparams** (*array (float)*) – MA coefficients.

Returns A function that takes in time lags and returns CARMA SF at the given lags.

eztao.carma.model_utils.**drw_acf**(*tau*)
Return a function that computes the DRW autocorrelation function (ACF).

Parameters **tau** (*float*) – DRW decorrelation/characteristic timescale.

Returns A function that takes in time lags and returns ACF at the given lags.

eztao.carma.model_utils.**drw_psd**(*amp*, *tau*)
Return a function that computes DRW Power Spectral Density (PSD).

Parameters

- **amp** (*float*) – DRW RMS amplitude
- **tau** (*float*) – DRW decorrelation/characteristic timescale

Returns A function that takes in frequencies and returns PSD at the given frequencies.

eztao.carma.model_utils.**drw_sf**(*amp*, *tau*)
Return a function that computes the structure function (SF) of DRW.

Parameters

- **amp** (*float*) – DRW RMS amplitude
- **tau** (*float*) – DRW decorrelation/characteristic timescale.

Returns A function that takes in time lags and returns DRW SF at the given lags.

eztao.carma.model_utils.**gp_psd**(*carmaTerm*)
Return a function that computes native GP Power Spectral Density (PSD).

Parameters **carmaTerm** (*object*) – A celerite CARMA term.

Returns A function that takes in frequencies and returns PSD at the given frequencies.

1.8 Time Series

1.8.1 CARMA process simulations

A collection of functions for simulating CARMA processes.

eztao.ts.carma_sim.**addNoise**(*y*, *yerr*, *seed=None*)
Add (gaussian) noise to the input simulated time series given the measurement uncertainties.

Parameters

- **y** (*array (float)*) – The 'clean' time series.
- **yerr** (*array (float)*) – The measurement uncertainties for the input time series.

seed (int): Random seed for simulating noise. Defaults to None.

Returns A new time series with simulated (gaussian) noise added on top.

Return type array(float)

eztao.ts.carma_sim.**gpSimByTime**(*carmaTerm*, *SNR*, *t*, *factor=10*, *nLC=1*, *log_flux=True*, *lc_seed=None*)
Simulate CARMA time series at desired time stamps.

This function uses a 'factor' parameter to determine the sampling rate of a full time series to simulate and downsample from. For example, if 'factor' = 10, then the full time series will be 10 times denser than the median sampling rate of the provided time stamps.

Parameters

- **carmaTerm** (*object*) – An EzTao CARMA kernel.
- **SNR** (*float*) – Signal-to-noise defined as ratio between CARMA RMS amplitude and the median of the measurement errors (simulated using log normal).
- **t** (*array (float)*) – The desired time stamps (starting from zero).
- **factor** (*int, optional*) – Parameter to control the ratio in the sampling rate between the simulated full time series and the desired output one. Defaults to 10.
- **nLC** (*int, optional*) – Number of time series to simulate. Defaults to 1.
- **log_flux** (*bool*) – Whether the flux/y values are in astronomical magnitude. This argument affects how errors are assigned. Defaults to True.
- **lc_seed** (*int*) – Random seed for time series simulation. Defaults to None.

Returns Time stamps (default in day), y values and measurement errors of the simulated time series.

Return type (array(float), array(float), array(float))

eztao.ts.carma_sim.**gpSimFull**(*carmaTerm*, *SNR*, *duration*, *N*, *nLC=1*, *log_flux=True*, *lc_seed=None*)
Simulate CARMA time series using uniform sampling.

Parameters

- **carmaTerm** (*object*) – An EzTao CARMA kernel.
- **SNR** (*float*) – Signal-to-noise defined as ratio between CARMA RMS amplitude and the median of the measurement errors (simulated using log normal).
- **duration** (*float*) – The duration of the simulated time series (default in days).
- **N** (*int*) – The number of data points in the simulated time series.

- **nLC** (*int, optional*) – Number of time series to simulate. Defaults to 1.
- **log_flux** (*bool*) – Whether the flux/y values are in astronomical magnitude. This argument affects how errors are assigned. Defaults to True.
- **lc_seed** (*int*) – Random seed for time series simulation. Defaults to None.

Raises **RuntimeError** – If the input CARMA term/model is not stable, thus cannot be solved by celerite.

Returns Time stamps (default in day), y values and measurement errors of the simulated time series.

Return type (array(float), array(float), array(float))

eztao.ts.carma_sim.**gpSimRand**(*carmaTerm*, *SNR*, *duration*, *N*, *nLC=1*, *log_flux=True*, *season=True*, *full_N=10000*, *lc_seed=None*, *downsample_seed=None*)

Simulate CARMA time series randomly downsampled from a much denser full time series.

Parameters

- **carmaTerm** (*object*) – An EzTao CARMA kernel.
- **SNR** (*float*) – Signal-to-noise defined as ratio between CARMA RMS amplitude and the median of the measurement errors (simulated using log normal).
- **duration** (*float*) – The duration of the simulated time series (default in days).
- **N** (*int*) – The number of data points in the simulated time series.
- **nLC** (*int, optional*) – Number of time series to simulate. Defaults to 1.
- **log_flux** (*bool*) – Whether the flux/y values are in astronomical magnitude. This argument affects how errors are assigned. Defaults to True.
- **season** (*bool, optional*) – Whether to simulate 6-months seasonal gaps. Defaults to True.
- **full_N** (*int, optional*) – The number of data points in the full time series (before downsampling). Defaults to 10_000.
- **lc_seed** (*int*) – Random seed for full time series simulation. Defaults to None.
- **downsample_seed** (*int*) – Random seed for downsampling the simulated full time series. Defaults to None.

Returns Time stamps (default in day), y values and measurement errors of the simulated time series.

Return type (array(float), array(float), array(float))

eztao.ts.carma_sim.**pred_lc**(*t*, *y*, *yerr*, *params*, *p*, *t_pred*, *return_var=True*)

Generate predicted values at particular time stamps given the initial time series and a best-fit model.

Parameters

- **t** (*array (float)*) – Time stamps of the initial time series.
- **y** (*array (float)*) – y values (i.e., flux) of the initial time series.
- **yerr** (*array (float)*) – Measurement errors of the initial time series.
- **params** (*array (float)*) – Best-fit CARMA parameters
- **p** (*int*) – The AR order (p) of the given best-fit model.
- **t_pred** (*array (float)*) – Time stamps to generate predicted time series.

- **return_var** (*bool, optional*) – Whether to return uncertainties in the mean prediction. Defaults to True.

Returns t_pred, mean prediction at t_pred and uncertainties (variance) of the mean prediction.

Return type (array(float), array(float), array(float))

1.8.2 Fitting time series to CARMA

A collection of functions to fit/analyze time series using CARMA models.

eztao.ts.carma_fit.**carma_fit**(*t*, *y*, *yerr*, *p*, *q*, *init_func=None*, *neg_lp_func=None*, *optimizer_func=None*, *n_opt=20*, *user_bounds=None*, *scipy_opt_kwargs={}*, *scipy_opt_options={}*, *debug=False*)

Fit an arbitrary CARMA model

The default settings are optimized for normalized LCs.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **p** (*int*) – The p order of a CARMA(p, q) model.
- **q** (*int*) – The q order of a CARMA(p, q) model.
- **init_func** (*object, optional*) – A user-provided function to generate initial guesses for the optimizer. Defaults to None.
- **neg_lp_func** (*object, optional*) – A user-provided function to compute negative probability given an array of parameters, an array of time series values and a celerite GP instance. Defaults to None.
- **optimizer_func** (*object, optional*) – A user-provided optimizer function. Defaults to None.
- **n_opt** (*int, optional*) – Number of optimizers to run. Defaults to 20.
- **user_bounds** (*array (float), optional*) – Parameter boundaries for the default optimizer. If p > 2, these are boundaries for the coefficients of the factored polynomial. Defaults to None.
- **scipy_opt_kwargs** (*dict, optional*) – Keyword arguments for scipy.optimize.minimize. Defaults to {}.
- **scipy_opt_options** (*dict, optional*) – "options" argument for scipy.optimize.minimize. Defaults to {}.
- **debug** (*bool, optional*) – Turn on/off debug mode. Defaults to False.

Raises **celerite.solver.LinAlgError** – For non-positive definite autocovariance matrices.

Returns Best-fit parameters

Return type array(float)

eztao.ts.carma_fit.**carma_log_fcoeff_init**(*p*, *q*, *ar_range=[- 8, 8]*, *ma_range=[- 10, 6]*, *ma_mult_range=[- 10, 0]*, *size=1*)

Randomly generate CARMA coefficients in the space of the factored polynomials

The default ranges are optimized for normalized light curves (with a standard deviation of unity).

Parameters

- **p** (*int*) – The p order of a CARMA(p, q) model.
- **q** (*int*) – The q order of a CARMA(p, q) model.
- **ar_range** (*object, optional*) – The range (in natural log) for AR polynomial coefficients. Defaults to [-8, 8].
- **ma_range** (*object, optional*) – The range (in natural log) for MA polynomial coefficients. Defaults to [-10, 6].
- **ma_mult_range** (*object, optional*) – The range for the MA multiplier (the coefficient of the highest-order term in the MA characteristic polynomial). Defaults to [-10, 0].
- **size** (*int, optional*) – The number of the set of coefficients to generate. Defaults to 1.

Returns A ndarray of coeffs for the factored polynomials in natural log.

Return type array(float)

Note: The notation (index) in the returned coefficients follows that in Jones et al. (1981). The last coefficient in the returned array is not part of the coefficients, rather a simple multiplying factor of the entire polynomial, which is needed to obtain the nominal CARMA representation.

---

eztao.ts.carma_fit.**dho_fit**(*t*, *y*, *yerr*, *init_func=None*, *neg_lp_func=None*, *optimizer_func=None*, *n_opt=20*, *user_bounds=None*, *scipy_opt_kwargs={}*, *scipy_opt_options={}*, *debug=False*)

Fit DHO to time series

The default settings are optimized for normalized LCs.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **init_func** (*object, optional*) – A user-provided function to generate initial guesses for the optimizer. Defaults to None.
- **neg_lp_func** (*object, optional*) – A user-provided function to compute negative probability given an array of parameters, an array of time series values and a celerite GP instance. Defaults to None.
- **optimizer_func** (*object, optional*) – A user-provided optimizer function. Defaults to None.
- **n_opt** (*int, optional*) – Number of optimizers to run.. Defaults to 20.
- **user_bounds** (*list, optional*) – Parameter boundaries for the default optimizer and the default flat prior. Defaults to None.
- **scipy_opt_kwargs** (*dict, optional*) – Keyword arguments for scipy.optimize.minimize. Defaults to {}.
- **scipy_opt_options** (*dict, optional*) – "options" argument for scipy.optimize.minimize. Defaults to {}.
- **debug** (*bool, optional*) – Turn on/off debug mode. Defaults to False.

Raises **celerite.solver.LinAlgError** – For non-positive definite autocovariance matrices.

Returns Best-fit DHO parameters

Return type array(float)

eztao.ts.carma_fit.**dho_log_param_init**(*ar_range=[- 6, 10]*, *ma_range=[- 10, 2]*, *size=1*)

Randomly generate DHO coefficients in the space of the factored polynomials

The default ranges are optimized for normalized light curves (with a standard deviation of unity).

Parameters

- **ar_range** (*object, optional*) – The range (in natural log) for DHO AR parameters. Defaults to [-6, 10].
- **ma_range** (*object, optional*) – The range (in natural log) for DHO MA parameters. Defaults to [-10, 2].
- **size** (*int, optional*) – The number of the set of DHO parameters to generate. Defaults to 1.

Returns A ndarray of DHO parameters in natural log.

Return type array(float)

eztao.ts.carma_fit.**drw_fit**(*t*, *y*, *yerr*, *init_func=None*, *neg_lp_func=None*, *optimizer_func=None*, *n_opt=10*, *user_bounds=None*, *scipy_opt_kwargs={}*, *scipy_opt_options={}*, *debug=False*)

Fit DRW.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **init_func** (*object, optional*) – A user-provided function to generate initial guesses for the optimizer. Defaults to None.
- **neg_lp_func** (*object, optional*) – A user-provided function to compute negative probability given an array of parameters, an array of time series values and a celerite GP instance. Defaults to None.
- **optimizer_func** (*object, optional*) – A user-provided optimizer function. Defaults to None.
- **n_opt** (*int, optional*) – Number of optimizers to run. Defaults to 10.
- **user_bounds** (*list, optional*) – Parameter boundaries for the default optimizer. Defaults to None.
- **scipy_opt_kwargs** (*dict, optional*) – Keyword arguments for scipy.optimize.minimize. Defaults to {}.
- **scipy_opt_options** (*dict, optional*) – "options" argument for scipy.optimize.minimize. Defaults to {}.
- **debug** (*bool, optional*) – Turn on/off debug mode. Defaults to False.

Returns Best-fit DRW parameters

Return type array(float)

eztao.ts.carma_fit.**drw_log_param_init**(*amp_range*, *log_tau_range*, *size=1*)
Randomly generate DRW parameters.

Parameters

- **amp_range** (*object*) – An array containing the range of DRW amplitude to simulate.
- **log_tau_range** (*object*) – An array containing the range of DRW timescale (in natural log) to simulate.
- **size** (*int, optional*) – The number of the set of DRW parameters to generate. Defaults to 1.

Returns A ndarray of DRW parameters in natural log.

Return type array(float)

eztao.ts.carma_fit.**flat_prior**(*log_params*, *bounds*)
A flat prior function. Returns 0 if "log_params" are within the given "bounds", negative infinity otherwise.

Parameters

- **log_params** (*array(float)*) – CARMA parameters in natural log.
- **bounds** (*array((float, float)*) – An array of boundaries.

Returns 0 or negative infinity.

Return type float

eztao.ts.carma_fit.**neg_fcoeff_ll**(*log_fcoeffs*, *y*, *gp*)
Negative log likelihood function for CARMA specified in the factored poly space.

This method will catch 'overflow/underflow' runtimeWarning and return inf as probability.

Parameters

- **log_fcoeffs** (*array(float)*) – Coefficients (in natural log) of a CARMA model in the factored polynomial space.
- **y** (*array(float)*) – y values of the input time series.
- **gp** (*object*) – celerite GP object with a proper CARMA kernel.

Returns Negative log likelihood.

Return type float

eztao.ts.carma_fit.**neg_lp_flat**(*log_params*, *y*, *gp*, *bounds=None*, *mode='fcoeff'*)
Negative log probability function using a flat prior.

Parameters

- **log_params** (*array(float)*) – CARMA parameters (or coefficients of the factored characteristic polynomial) in natural log.
- **y** (*array(float)*) – y values of the input time series.
- **gp** (*object*) – celerite GP object with a proper CARMA kernel.
- **bounds** (*array((float, float)*) – An array of boundaries. Defaults to None.
- **mode** (*str, optional*) – The parameter space in which proposals are made. The mode determines which loglikehood function to use. Defaults to "fcoeff".

Returns Log probability of the proposed parameters.

Return type float

eztao.ts.carma_fit.**neg_param_ll**(*log_params*, *y*, *gp*)
Negative log likelihood function for CARMA specified in the nominal space.

This method will catch 'overflow/underflow' runtimeWarning and return inf as probability.

Parameters

- **log_params** (*array (float)*) – Natural log of CARMA parameters.
- **y** (*array (float)*) – y values of the input time series.
- **gp** (*object*) – celerite GP object with a proper CARMA kernel.

Returns Negative log likelihood.

Return type float

eztao.ts.carma_fit.**sample_carma**(*p*, *q*)
Randomly generate a stationary CARMA model given the orders (p and q).

Parameters

- **p** (*int*) – The p order of a CARMA(p, q) model.
- **q** (*int*) – The q order of a CARMA(p, q) model.

Returns AR and MA coefficients in two separate arrays.

eztao.ts.carma_fit.**scipy_opt**(*y*, *gp*, *init_func*, *neg_lp_func*, *n_opt*, *mode='fcoeff'*, *debug=False*, *opt_kwargs={}*, *opt_options={}*)
A wrapper for scipy.optimize.minimize method.

Parameters

- **y** (*array (float)*) – y values of the input time series.
- **gp** (*object*) – celerite GP object with a proper CARMA kernel.
- **init_func** (*object*) – A user-provided function to generate initial guesses for the optimizer. Defaults to None.
- **neg_lp_func** (*object*) – A user-provided function to compute negative probability given an array of parameters, an array of time series values and a celerite GP instance. Defaults to None.
- **n_opt** (*int*) – Number of iterations to run the optimizer.
- **mode** (*str, optional*) – The parameter space in which to make proposals, this should be determined in the "_fit" functions based on the value of the p order. Defaults to "fcoeff".
- **debug** (*bool, optional*) – Turn on/off debug mode. Defaults to False.
- **opt_kwargs** (*dict, optional*) – Keyword arguments for scipy.optimize.minimize. Defaults to {}.
- **opt_options** (*dict, optional*) – "options" argument for scipy.optimize.minimize. Defaults to {}.

Returns Best-fit parameters if "debug" is False, an array of scipy.optimize.OptimizeResult objects otherwise.

1.8.3 MCMC

A module containing functions to run MCMC using emcee.

eztao.ts.carma_mcmc.**mcmc**(*t*, *y*, *yerr*, *p*, *q*, *n_walkers=32*, *burn_in=500*, *n_samples=2000*, *init_param=None*)

A simple wrapper to run quick MCMC using emcee.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **p** (*int*) – The p order of a CARMA(p, q) model.
- **q** (*int*) – The q order of a CARMA(p, q) model.
- **n_walkers** (*int, optional*) – Number of MCMC walkers. Defaults to 32.
- **burn_in** (*int, optional*) – Number of burn in steps. Defaults to 500.
- **n_samples** (*int, optional*) – Number of MCMC steps to run. Defaults to 2000.
- **init_param** (*array (float), optional*) – The initial position for the MCMC walker. Defaults to None.

Returns The emcee sampler object. The MCMC flatchain (n_walkers*n_samplers, dim) and chain (n_walkers, n_samplers, dim) in CARMA space if p > 2, otherwise empty.

Return type (object, array(float), array(float))

1.8.4 Time series utility functions

A collection of utility functions to assist analysis/simulation of time series data.

eztao.ts.utils.**add_season**(*t*, *lc_start=0*, *season_start=90*, *season_end=270*)

Insert seasonal gaps into time series

Parameters

- **t** (*array (float)*) – Time stamps of the original time series.
- **lc_start** (*float*) – Starting day for the output time series. (0 -> 365.25). Default to 0.
- **season_start** (*float*) – Observing season start day within a year. Default to 90.
- **season_end** (*float*) – Observing season end day within a year. Default to 270.

Returns A 1d array booleans indicating which data points to keep.

eztao.ts.utils.**downsample_byN**(*t*, *nObs*, *seed*)

Randomly choose N data points from a given time series.

Parameters

- **t** (*array (float)*) – Time stamps of the original time series.
- **N** (*int*) – The number of entries in the final time series.
- **seed** (*int*) – Random seed for downsampling. Defaults to None.

Returns A 1d array booleans indicating which data points to keep.

eztao.ts.utils.**downsample_byTime**(*tIn*, *tOut*)
Downsample a time series at desired output time stamps.

Parameters

- **tIn** (*array (float)*) – Time stamps of the original time series.
- **tOut** (*array (float)*) – Time stamps of the output time series.

Returns Indices for which the data points should be kept from the original time series. Note that there could be duplicates.

Return type array(int)

eztao.ts.utils.**median_clip**(*y*, *num_sigma=3*)
Clip time series using a three point median filter.

The sigma (standard deviation) for the time series is computed from the median absolute deviation (MAD) as to reduce the effects from extreme outliers, where sigma sim 1.4826*MAD. If more than 10% of the data points are removed, the upper bound will be lifted gradually until that fraction drops bellow 10%.

Parameters

- **y** (*array (float)*) – y values of the original time series.
- **num_sigma** (*int, optional*) – Data points that are more than this number of sigma away from the three point median will be removed. Defaults to 3.

Returns A 1d array booleans indicating which data points to keep.

1.9 Visualization Tools

A few random plotting functions.

eztao.viz.mpl_viz.**plot_dho_ll**(*t*, *y*, *yerr*, *best_params*, *gp*, *prob_func*, *inner_dim=10*, *outer_dim=4*, *ranges=[(None, None), (None, None), (None, None), (None, None)]*, *nLevels=10*, ***kwargs*)
Plot DHO/CARMA(2,1) log likelihood lanscape.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **best_params** (*array (float)*) – Best-fit DHO parameters in [a1, a2, b0, b1].
- **gp** (*object*) – celerite GP object with a proper DHO kernel.
- **prob_func** (*func*) – Posterior/Likelihood function with args=(params, y, gp).
- **inner_dim** (*int, optional*) – The number of points to eval likelihood along a1 and a2. Defaults to 10.
- **outer_dim** (*int, optional*) – The number of points to eval likelihood along b0 and b1. Defaults to 4.
- **ranges** (*list, optional*) – Parameters ranges (in natural log) within which to plot the surface. Defaults to [(None, None), (None, None), (None, None), (None, None)].
- **nLevels** (*int, optional*) – The number of levels in the final contour plot. Defaults to 10.

Keyword Arguments **true_params** (*array (float)*) – The true DHO parameters of the input time series.

eztao.viz.mpl_viz.**plot_drw_ll**(*t*, *y*, *yerr*, *best_params*, *gp*, *prob_func*, *amp_range=None*, *tau_range=None*, *nLevels=10*, ***kwargs*)

Plot DRW log likelihood surface.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series (the default unit is day).
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **best_params** (*array (float)*) – Best-fit DRW parameters, [amp, tau].
- **gp** (*object*) – celerite GP object with a proper DRW kernel.
- **prob_func** (*func*) – Posterior/Likelihood function with args=(params, y, gp)
- **amp_range** (*tuple, optional*) – The range of parameters to evaluate likelihood. Defaults to None.
- **tau_range** (*tuple, optional*) – The range of parameters to evaluate likelihood. Defaults to None.
- **nLevels** (*int, optional*) – The number of levels in the final contour plot. Defaults to 10.

Keyword Arguments

- **grid_size** (*int*) – The number of points to evaluate likelihood along a given axis.
- **true_params** (*array (float)*) – The true DRW parameters of the input time series.

eztao.viz.mpl_viz.**plot_pred_lc**(*t*, *y*, *yerr*, *params*, *p*, *t_pred*, *plot_input=True*)

Plot GP predicted time series given best-fit parameters.

Parameters

- **t** (*array (float)*) – Time stamps of the input time series.
- **y** (*array (float)*) – y values of the input time series.
- **yerr** (*array (float)*) – Measurement errors for y values.
- **params** (*array (float)*) – Best-fit CARMA parameters
- **p** (*int*) – The AR order (p) of the best-fit model.
- **t_pred** (*array (float)*) – Time stamps at which to generate predictions.
- **plot_input** (*bool*) – Whether to plot the input time series. Defaults to True.

CHAPTER

TWO

CHANGELOG

2.1 0.4.3 (2023-12-15)

- Drop support for Python 3.7
- Bump *numba* requirement to *>=0.57.0*.
- **New Features:** Added seed options to *gpSimRand*, *gpSimFull*, and *addNoise*
- **Bug fixes:** #74, #75

2.2 0.4.1 (2023-06-12)

- Update reference to numpy bool/complex (#71)
- **Bug fixes:** #50, #54, #59

2.3 0.4.0 (2021-07-19)

- Fitting functions (i.e., *drw_fit*) are now fully modular (allow user provided optimization function, prior function and etc.)
- A new *addNoise* function to simulated random noise given measurement errors.
- **Bug fixes:** #44
- **API changes:** *n_iter* -> *n_opt* in fitting functions.

2.4 0.3.0 (2021-01-07)

- update parameter initialization in fit functions; removed *de* option #26, #27
- add few utils functions #30, #25
- add mcmc module #29
- ts simulation now support linear error
- added online documentation

2.5 0.2.3 (2020-12-08)

- add methods to CARMA_term conversion between CARMA and poly space
- fixed bugs and add tests for model 2nd order stat functions
- close #2, close #10

2.6 0.2.1 (2020-12-05)

- A bunch bug fixes in the ts.carma module
- Improved _min_opt optimizer, now added to all fitting functions
- Now using minimizer instead of differential evolution may result in more robust parameter estimates.

2.7 0.2.0 (2020-12-03)

Fixed some bugs and added new features.

- Fixed the instability issue when fitting time series to models higher than DHO/CARMA(2,1)
- Cleaned up the plotting module
- Added PSD, ACVF, and SF functions

2.8 0.1.0 (2020-11-09)

First release!

- Fully working CARMA kernels: *DRW_term*, *DHO_term* and *CARMA_term*
- Functions to simulate CARMA time series given a kernel
- Functions to fit arbitrary time series to CARMA models (still having instability issues with CARMA models higher than DHO/CARMA(2,1))

PYTHON MODULE INDEX

e

INDEX

M

N

P

R

S