
isochrones Documentation

Release 2.1

Timothy Morton

Feb 06, 2020

Contents

1	Install	3
1.1	Conda environment and testing	3
1.2	Installing MultiNest	3
2	Quick Start	5
2.1	Access stellar model grid data	5
2.2	Interpolate stellar properities	7
2.3	Generate synthetic properties of stars	7
2.4	Fit physical parameters of a star to observed data	9
2.5	Fit a binary star model	13
3	Interpolation: the <code>DFInterpolator</code>	17
4	Stellar model grids	21
4.1	Background and EEPs	21
4.2	Model Grid Objects and Interpolation	21
4.3	Example visualization	26
5	Bolometric correction grids	27
6	ModelGridInterpolator	29
6.1	Isochrones	29
6.2	Evolution tracks	30
6.3	Generating synthetic properties	31
6.4	Demo: Visualize	33
7	Fitting stellar parameters	35
7.1	Defining a star model	35
7.2	Priors	37
7.3	Sampling the posterior	40
8	Multiple star systems	47
8.1	Unresolved multiple systems	47
8.2	Resolved multiple system	51
8.3	Unassociated companions	54
8.4	More complex models	54
9	Simulating stellar populations	57

9.1	StarPopulation object	57
9.2	ModelGridInterpolator.generate_binary	60

Isochrones is a python package that provides a simple interface to grids of stellar evolution models, enabling the following common use cases:

- Interpolating stellar model values at desired locations.
- Generating properties of synthetic stellar populations.
- Determining stellar properties of either single- or multiple-star systems, based on arbitrary observables.

The central goal of **isochrones** is to standardize model-grid-based stellar parameter inference, and to enable such inference under different sets of stellar models. For now, only MIST models are included, but we hope to incorporate YAPSI and PARSEC models as well.

1.1 Conda environment and testing

Isochrones requires python 3. I also recommend using **isochrones** in its own conda environment, to help manage dependencies. For example:

```
conda create -n isochrones numpy numba nose pytables pandas
```

Then

```
conda activate isochrones
pip install isochrones
```

To make sure everything is working, run

```
nosetests isochrones
```

And if anything breaks, please [raise an issue](#).

1.2 Installing MultiNest

It is highly recommended to install **MultiNest/PyMultiNest** for model fitting. First, install/build multinest with

```
git clone https://github.com/johannesBuchner/MultiNest
cd MultiNest/build
cmake -DCMAKE_INSTALL_PREFIX=~ .. # or just "cmake .." if you have root permissions
make
make install
```

(Note that if you don't have cmake available on your system, that you can install it in your environment with `conda install -c conda-forge cmake`.)

If you do not have root permissions and thus installed the **MultiNest** libraries to your home directory, you will also need to make sure that `~/lib` is in your `LD_LIBRARY_PATH` environment variable; e.g., you can include the following line in your `~/ .bash_profile` file:

```
export LD_LIBRARY_PATH=$HOME/lib
```

Then you can install `pymultinest` with

```
pip install pymultinest
```

(And run `nosetests isochrones` again, for good measure, to confirm that **MultiNest** works.)

2.1 Access stellar model grid data

```
[1]: from isochrones.mist import MISTIsochroneGrid
```

```
grid = MISTIsochroneGrid()
print(len(grid.df))
grid.df.head() # Just the first few rows
```

```
1494453
```

```
[1]:
```

log10_isochrone_age_yr	feh	EEP	eep	age	feh	mass	initial_mass	\
5.0	-4.0	35	35	5.0	-3.978406	0.100000	0.100000	
		36	36	5.0	-3.978406	0.102885	0.102885	
		37	37	5.0	-3.978406	0.107147	0.107147	
		38	38	5.0	-3.978406	0.111379	0.111379	
		39	39	5.0	-3.978406	0.115581	0.115581	

log10_isochrone_age_yr	feh	EEP	radius	density	logTeff	Teff	\
5.0	-4.0	35	1.106082	0.104184	3.617011	4140.105252	
		36	1.122675	0.102507	3.618039	4149.909661	
		37	1.147702	0.099921	3.619556	4164.436984	
		38	1.173015	0.097287	3.621062	4178.903372	
		39	1.198615	0.094627	3.622555	4193.289262	

log10_isochrone_age_yr	feh	EEP	logg	logL	Mbol	delta_nu	\
5.0	-4.0	35	3.350571	-0.489734	5.964335	37.987066	
		36	3.347798	-0.472691	5.921728	37.739176	
		37	3.343658	-0.447471	5.858678	37.345115	
		38	3.339612	-0.422498	5.796244	36.923615	
		39	3.335660	-0.397776	5.734440	36.473151	

(continues on next page)

(continued from previous page)

log10_isochrone_age_yr	feh	EEP	nu_max	phase	dm_deep
5.0	-4.0	35	299.346079	-1.0	0.002885
		36	298.570836	-1.0	0.003573
		37	297.180748	-1.0	0.004247
		38	295.526946	-1.0	0.004217
		39	293.589960	-1.0	0.004189

```
[2]: from isochrones.mist import MISTEvolutionTrackGrid
```

```
grid_tracks = MISTEvolutionTrackGrid()
print(len(grid_tracks.df))
grid_tracks.df.head()
```

```
3619652
```

```
[2]:
```

initial_feh	initial_mass	EEP	nu_max	logg	eep	initial_mass	\
-4.0	0.1	1	143.524548	3.033277	1.0	0.1	
		2	145.419039	3.038935	2.0	0.1	
		3	147.409881	3.044805	3.0	0.1	
		4	149.499346	3.050886	4.0	0.1	
		5	151.703570	3.057203	5.0	0.1	

initial_feh	initial_mass	EEP	radius	logTeff	mass	density	Mbol	\
-4.0	0.1	1	1.593804	3.620834	0.1	0.034823	5.132871	
		2	1.583455	3.620769	0.1	0.035510	5.147664	
		3	1.572790	3.620702	0.1	0.036237	5.163015	
		4	1.561817	3.620631	0.1	0.037006	5.178922	
		5	1.550499	3.620558	0.1	0.037823	5.195452	

initial_feh	initial_mass	EEP	phase	feh	Teff	logL	\
-4.0	0.1	1	-1.0	-3.978406	4176.707371	-0.157148	
		2	-1.0	-3.978406	4176.085183	-0.163066	
		3	-1.0	-3.978406	4175.435381	-0.169206	
		4	-1.0	-3.978406	4174.757681	-0.175569	
		5	-1.0	-3.978406	4174.049081	-0.182181	

initial_feh	initial_mass	EEP	delta_nu	interpolated	star_age	age	\
-4.0	0.1	1	21.776686	False	13343.289397	4.125263	
		2	21.993078	False	14171.978264	4.151430	
		3	22.219791	False	15048.910447	4.177505	
		4	22.457004	False	15975.827275	4.203463	
		5	22.706349	False	16962.744747	4.229496	

initial_feh	initial_mass	EEP	dt_deep
-4.0	0.1	1	0.026168
		2	0.026121
		3	0.026016
		4	0.025996
		5	0.025996

2.2 Interpolate stellar properites

```
[3]: from isochrones import get_ichrone
mist = get_ichrone('mist')
eep = mist.get_eep(1.01, 9.76, 0.03, accurate=True)
mist.interp_value([eep, 9.76, 0.03], ['Teff', 'logg', 'radius', 'density'])

[3]: array([5.86016011e+03, 4.36634798e+00, 1.09151255e+00, 1.09589730e+00])

[4]: mist.interp_mag([eep, 9.76, 0.03, 200, 0.1], bands=['G', 'BP', 'RP'])

[4]: (5860.16011294621,
4.366347981387894,
-0.005536922088842331,
array([10.99261956, 11.3150264 , 10.50313434]))
```

2.3 Generate synthetic properties of stars

```
[5]: from isochrones import get_ichrone
tracks = get_ichrone('mist', tracks=True)

mass, age, feh = (1.03, 9.72, -0.11)

tracks.generate(mass, age, feh, return_dict=True) # "accurate=True" makes more_
↪accurate, but slower

[5]: {'nu_max': 2275.6902092679834,
'logg': 4.315208279229787,
'eep': 394.24,
'initial_mass': 1.03,
'radius': 1.1692076259176427,
'logTeff': 3.785191265391399,
'mass': 1.0297274169057322,
'density': 0.9097687776092286,
'Mbol': 4.162373757546131,
'phase': 0.0,
'feh': -0.19095007384845408,
'Teff': 6100.263434973235,
'logL': 0.23105049698154745,
'delta_nu': 114.32933695055772,
'interpolated': 0.0,
'star_age': 5302578707.515498,
'age': 9.722480201790624,
'dt_deep': 0.0036558739980003118,
'J': 3.2044197352759696,
'H': 2.91756110497181,
'K': 2.890399473719951,
'G': 4.085847599912897,
'BP': 4.349405878788243,
'RP': 3.6587316339856084,
'W1': 2.8807983122840044,
'W2': 2.885550073210391,
'W3': 2.8685709557487264,
'TESS': 3.653543903981804,
'Kepler': 4.004222279916473}
```

```
[6]: from isochrones.priors import ChabrierPrior
import numpy as np
```

```
# Simulate a 1000-star cluster at 8kpc
```

```
N = 1000
masses = ChabrierPrior().sample(N)
feh = -1.8
age = np.log10(6e9) # 6 Gyr
distance = 8000. # 8 kpc
AV = 0.15

# By default this will return a dataframe
%timeit tracks.generate(masses, age, feh, distance=distance, AV=AV)
df = tracks.generate(masses, age, feh, distance=distance, AV=AV)
```

The slowest run took 158.58 times longer than the fastest. This could mean that an intermediate result is being cached.
1 loop, best of 3: 9.04 ms per loop

```
[7]: df = df.dropna()
print(len(df)) # about half of the original simulated stars are nans
df.head()
```

```
503
```

```
[7]:
```

	nu_max	logg	eep	initial_mass	radius	logTeff	\
0	10804.874097	4.914275	303.258462	0.418821	0.374324	3.631195	
1	21841.644652	5.197122	252.271094	0.150592	0.161974	3.583987	
7	2838.154305	4.435801	384.922283	0.849837	0.924219	3.833683	
8	180.963558	3.194705	490.813513	0.968456	4.116643	3.742612	
9	1931.725171	4.282014	416.535309	0.911882	1.142684	3.860309	

	mass	density	Mbol	phase	...	H	K	\
0	0.418811	11.354937	8.178493	0.0	...	20.662206	20.501401	
1	0.150591	50.030150	10.467041	0.0	...	22.738821	22.531319	
7	0.849572	1.517288	4.187702	0.0	...	17.866850	17.848031	
8	0.967435	0.019564	1.854663	2.0	...	14.901613	14.851377	
9	0.911438	0.861278	3.460707	0.0	...	17.332797	17.316050	

	G	BP	RP	W1	W2	W3	\
0	23.037457	23.718192	22.251047	20.363681	20.324516	20.219805	
1	25.488818	26.383334	24.589471	22.380110	22.316618	22.177299	
7	18.902509	19.108502	18.534769	17.834259	17.825916	17.803109	
8	16.530374	16.884757	15.996079	14.816213	14.800818	14.770045	
9	18.172883	18.341059	17.864912	17.304300	17.297109	17.275839	

	TESS	Kepler
0	22.229761	22.950946
1	24.559047	25.416412
7	18.528245	18.837217
8	15.985842	16.451727
9	17.857626	18.111831

[5 rows x 29 columns]

```
[8]: import holoviews as hv
hv.extension('bokeh')
```

(continues on next page)

(continued from previous page)

```
import hvplot.pandas
```

```
df['BP-RP'] = df.BP - df.RP
df.hvplot.scatter('BP-RP', 'G', hover_cols=['mass', 'radius', 'Teff', 'logg', 'eep']).
    →options(invert_yaxis=True, width=600)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[8]: :Scatter    [BP-RP]    (G,mass,radius,Teff,logg,eep)
```

2.4 Fit physical parameters of a star to observed data

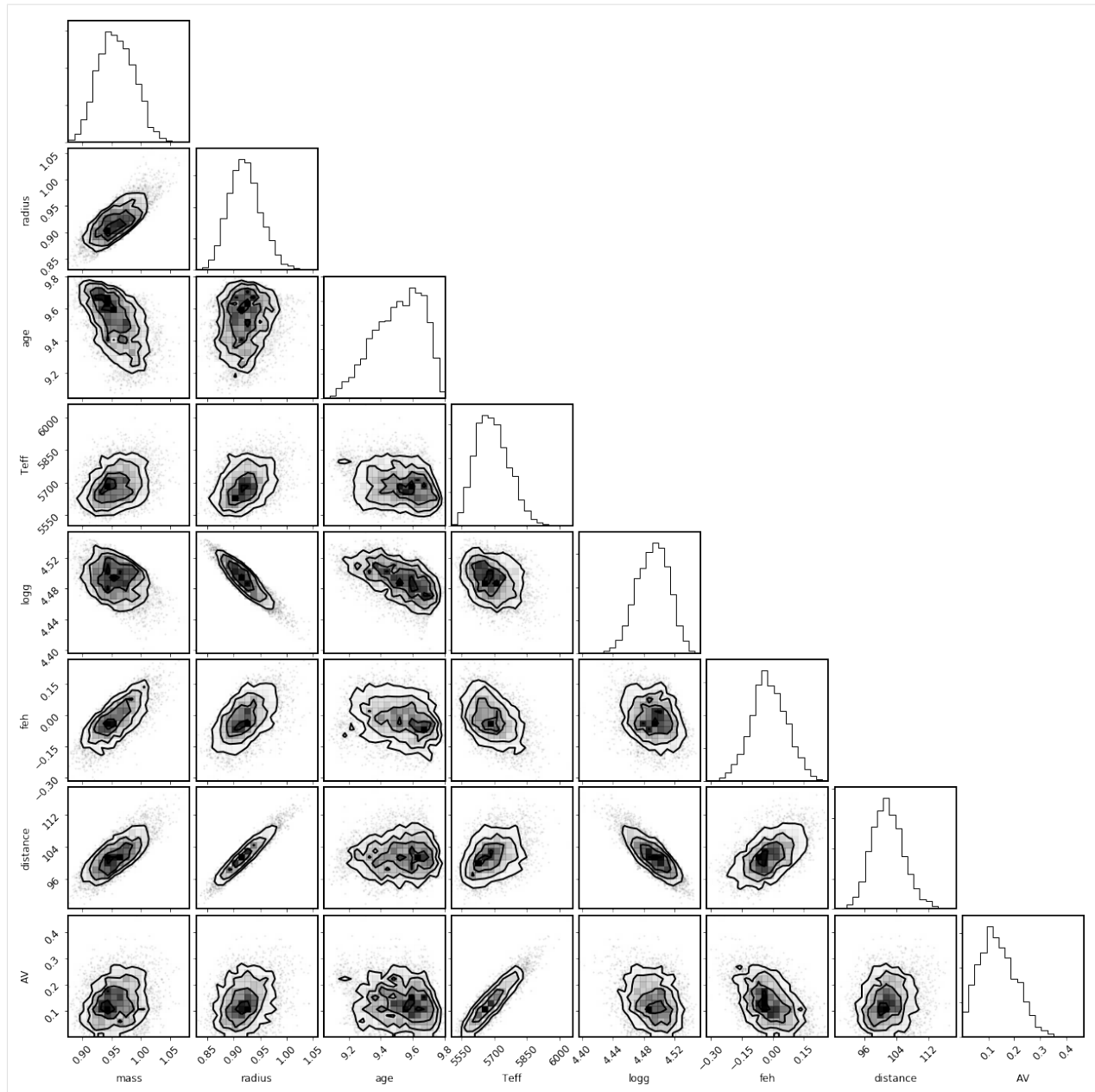
```
[9]: from isochrones import get_ichrone, SingleStarModel
```

```
mist = get_ichrone('mist', bands=['BP', 'RP'])
params = {'Teff': (5700, 100), 'logg': (4.5, 0.1), 'feh': (0.0, 0.15),
          'BP': (10.42, 0.01), 'RP': (9.54, 0.01),
          'parallax': (10, 0.5)} # mas
mod = SingleStarModel(mist, **params)
mod.fit()
```

```
INFO:root:MultiNest basename: ./chains/mist-single-
```

```
[10]: %matplotlib inline
```

```
mod.corner_physical();
```



Check out the numerical sampling results:

```
[11]: mod.samples.describe()
```

```
[11]:
```

	eep	age	feh	distance	AV	\
count	4643.000000	4643.000000	4643.000000	4643.000000	4643.000000	
mean	337.710149	9.509309	-0.020312	101.801691	0.136494	
std	9.624071	0.149170	0.078899	3.989609	0.069615	
min	304.868138	9.043279	-0.300996	88.634174	0.000291	
25%	330.856377	9.400976	-0.070581	99.008511	0.085589	
50%	339.214747	9.526285	-0.022509	101.582447	0.129668	
75%	345.473042	9.630834	0.033488	104.310959	0.183091	
max	364.435350	9.802944	0.243018	118.737714	0.466537	

(continues on next page)

(continued from previous page)

```

lnprob
count 4643.000000
mean -39.022183
std 1.311924
min -48.789323
25% -39.612822
50% -38.744828
75% -38.076162
max -37.088602

```

And the derived parameters at those samples:

```
[12]: mod.derived_samples.describe()
```

```

[12]:
count    eep      age      feh      mass  initial_mass  \
count 4643.000000 4643.000000 4643.000000 4643.000000 4643.000000
mean   337.710149  9.509309  -0.018942  0.958078  0.958169
std     9.624071  0.149170  0.086400  0.030418  0.030410
min    304.868138  9.043279  -0.317669  0.876002  0.876104
25%    330.856377  9.400976  -0.076201  0.936277  0.936370
50%    339.214747  9.526285  -0.022622  0.956896  0.956970
75%    345.473042  9.630834  0.039230  0.979404  0.979464
max    364.435350  9.802944  0.267804  1.083840  1.083927

count    radius  density  logTeff  Teff      logg  \
count 4643.000000 4643.000000 4643.000000 4643.000000 4643.000000
mean   0.921082  1.738516  3.755075 5690.649562  4.491137
std     0.031022  0.137653  0.005747  75.536605  0.020924
min    0.828304  1.209516  3.740466 5501.680997  4.396153
25%    0.899485  1.643165  3.750771 5634.112958  4.476522
50%    0.919331  1.740170  3.754596 5683.909988  4.492272
75%    0.940155  1.832595  3.758868 5740.000379  4.506173
max    1.058840  2.217147  3.782650 6062.846454  4.553723

count    ...      Mbol  delta_nu  nu_max  phase  \
count    ...      4643.000000 4643.000000 4643.000000 4643.0
mean     ...      4.982881 157.087407 3538.322417  0.0
std      ...      0.107383  6.095095  177.717410  0.0
min      ...      4.568552 131.705546 2801.309165  0.0
25%      ...      4.913786 152.933682 3412.744754  0.0
50%      ...      4.985717 157.283081 3545.198779  0.0
75%      ...      5.061043 161.294116 3664.317241  0.0
max      ...      5.303972 177.056425 4130.359858  0.0

count    dm_deep  BP_mag  RP_mag  parallax  distance  \
count 4643.000000 4643.000000 4643.000000 4643.000000 4643.000000
mean   0.008306  10.420067  9.539072  9.837970 101.801691
std     0.000654  0.009287  0.009087  0.382167  3.989609
min    0.002593  10.386946  9.505070  8.421924  88.634174
25%    0.007966  10.413741  9.533035  9.586720  99.008511
50%    0.008247  10.420135  9.539162  9.844220 101.582447
75%    0.008619  10.426205  9.545132 10.100142 104.310959
max    0.011076  10.453965  9.574724 11.282330 118.737714

count    AV
count 4643.000000
mean   0.136494

```

(continues on next page)

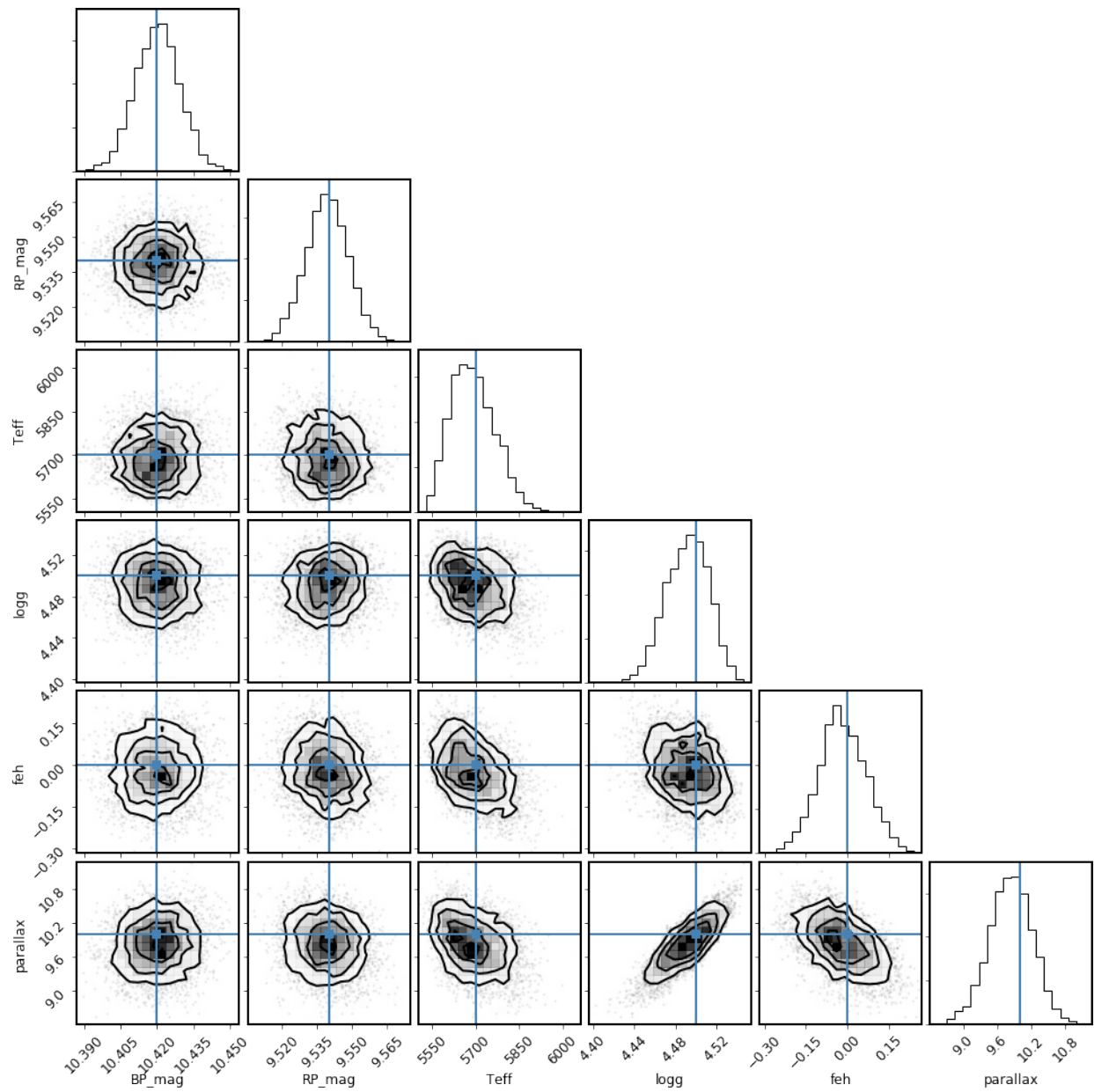
(continued from previous page)

```
std      0.069615
min      0.000291
25%      0.085589
50%      0.129668
75%      0.183091
max      0.466537
```

```
[8 rows x 21 columns]
```

Eyeball your posterior predictive with:

```
[13]: mod.corner_observed();
```

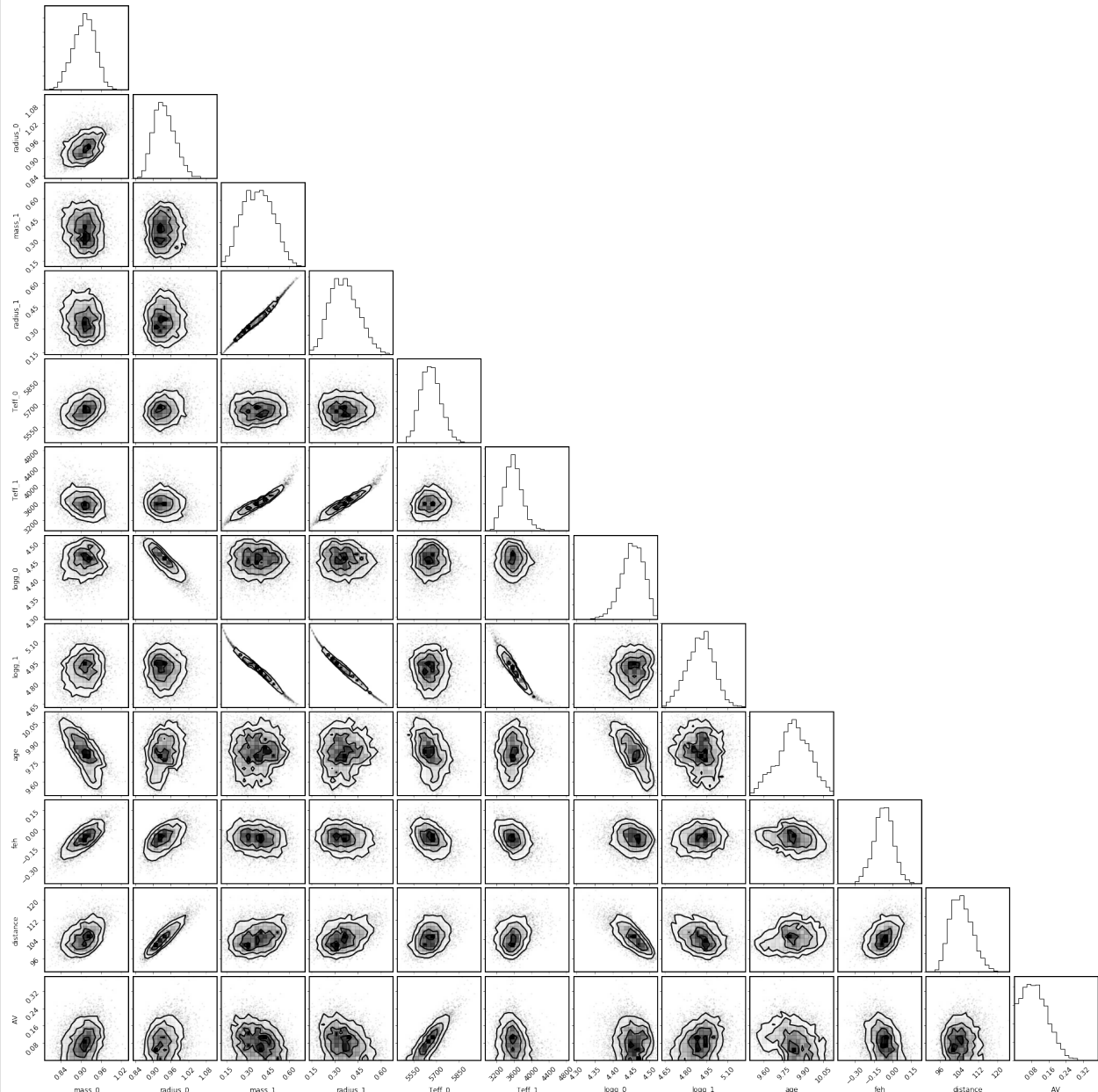


2.5 Fit a binary star model

```
[14]: from isochrones import BinaryStarModel
mod2 = BinaryStarModel(mist, **params)
```

```
[15]: mod2.fit()
mod2.corner_physical();
```

INFO:root:MultiNest basename: ./chains/mist-binary-



```
[16]: mod2.derived_samples.head()
```

```
[16]:      eep_0      eep_1      age      feh      distance      AV \
0  359.086356  249.801089  9.821476 -0.073397  98.727349  0.178347
```

(continues on next page)

(continued from previous page)

```

1  398.209362  283.317080  10.067148 -0.280222  107.778948  0.088950
2  389.035743  301.852188   9.971050 -0.207856  106.526665  0.166244
3  397.668498  266.911796   9.915806 -0.109190  120.644797  0.249099
4  370.982797  256.436383   9.816281 -0.205884  103.041704  0.385523

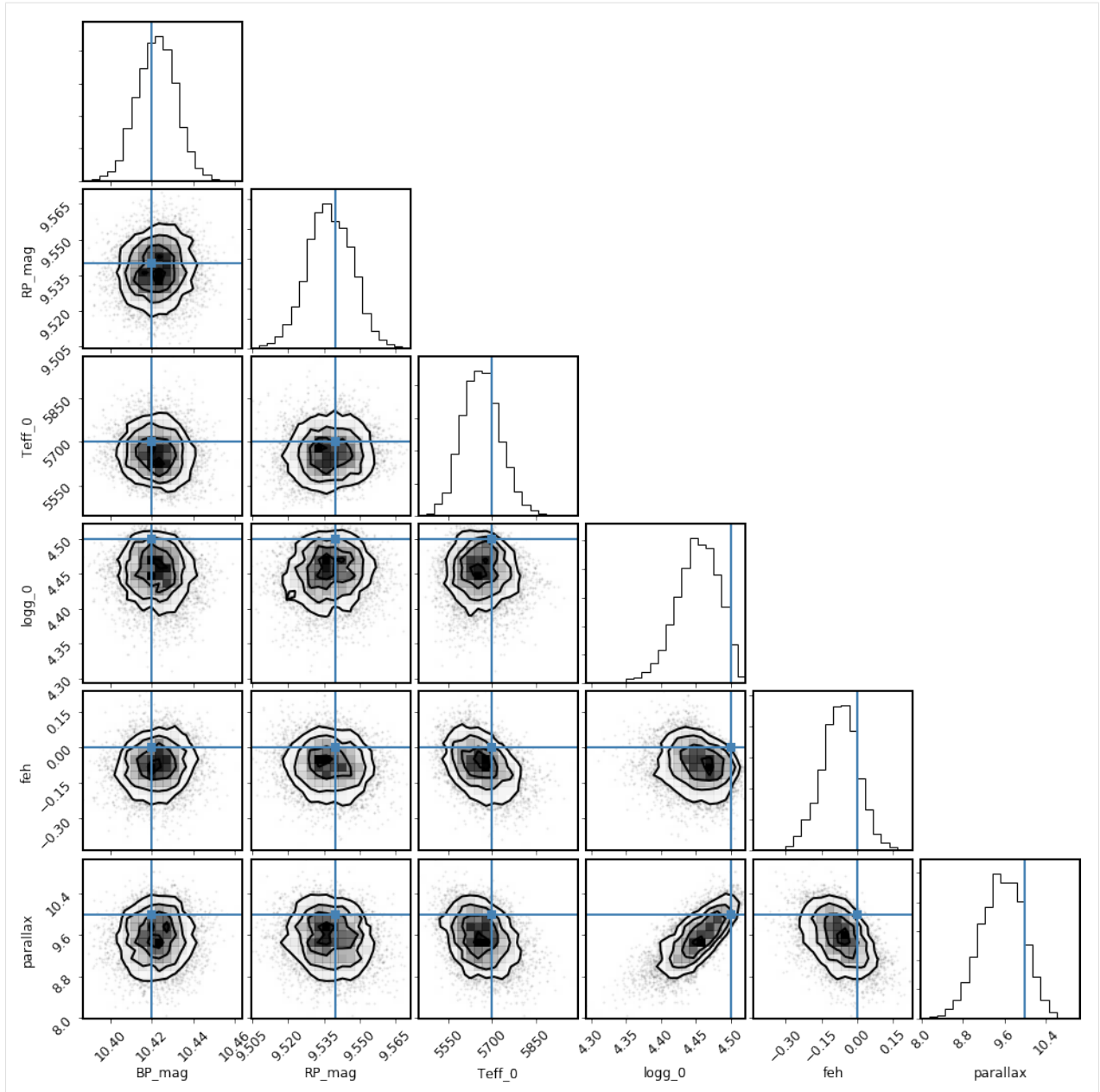
      lnprob      eep_0      feh_0      mass_0      ...      Mbol_1  \
0 -54.808864  359.086356 -0.104045  0.902009  ...      10.367668
1 -54.414634  398.209362 -0.372728  0.826538  ...      9.308760
2 -54.189376  389.035743 -0.278190  0.869004  ...      9.014070
3 -54.100849  397.668498 -0.176003  0.936127  ...      9.592630
4 -53.934142  370.982797 -0.260589  0.914513  ...      9.905946

      delta_nu_1      nu_max_1  phase_1  dm_deep_1  BP_mag_1  RP_mag_1  \
0  615.092182  16669.820345      0.0  0.006549  17.795078  15.260853
1  463.937102  12708.865085      0.0  0.001280  16.173813  14.156854
2  414.876808  11597.777825      0.0  0.002549  15.919824  13.882693
3  486.986772  13422.770521      0.0  0.006254  17.143644  14.860496
4  562.841780  15244.528561      0.0  0.007484  17.169701  14.890180

      BP_mag  RP_mag  parallax
0  10.463678  9.547117  10.128906
1  10.414967  9.566663   9.278250
2  10.391647  9.518239   9.387321
3  10.432474  9.553840   8.288795
4  10.439209  9.511726   9.704808

[5 rows x 44 columns]
```

```
[17]: mod2.corner_observed();
```



Interpolation: the `DFInterpolator`

Linear interpolation between gridded datapoints lies at the heart of much of what **isochrones** does. The custom `DFInterpolator` object manages this interpolation, implemented to optimize speed and convenience for large grids. A `DFInterpolator` is built on top of a pandas multi-indexed dataframe, and while designed with stellar model grids in mind, it can be used with any similarly structured data.

Let's demonstrate with a small example of data on a 2-dimensional grid.

```
[1]: import itertools
import numpy as np
import pandas as pd

x = np.arange(1, 4)
y = np.arange(1, 6)

index = pd.MultiIndex.from_product((x, y), names=['x', 'y'])
df = pd.DataFrame(index=index)

df['sum'] = [x + y for x, y in itertools.product(x, y)]
df['product'] = [x * y for x, y in itertools.product(x, y)]
df['power'] = [x**y for x, y in itertools.product(x, y)]
```

```
df
```

```
[1]:
```

		sum	product	power
x	y			
1	1	2	1	1
	2	3	2	1
	3	4	3	1
	4	5	4	1
	5	6	5	1
2	1	3	2	2
	2	4	4	4
	3	5	6	8
	4	6	8	16
	5	7	10	32

(continues on next page)

(continued from previous page)

3	1	4	3	3
	2	5	6	9
	3	6	9	27
	4	7	12	81
	5	8	15	243

The `DFInterpolator` is initialized with this dataframe and then can interpolate the values of the columns at any location within the grid defined by the multiindex.

```
[2]: from isochrones.interp import DFInterpolator
```

```
interp = DFInterpolator(df)
interp([1.4, 2.1])
```

```
[2]: array([3.5 , 2.94, 2.36])
```

Individual columns may also be accessed by name:

```
[3]: interp([2.2, 4.6], ['product'])
```

```
[3]: array([10.12])
```

This object is very similar to the linear interpolation objects available in `scipy`, but it is significantly faster for single interpolation evaluations:

```
[4]: from scipy.interpolate import RegularGridInterpolator
```

```
nx, ny = len(x), len(y)
grid = np.reshape(df['sum'].values, (nx, ny))
scipy_interp = RegularGridInterpolator([x, y], grid)

# Values are the same
assert(scipy_interp([1.3, 2.2])==interp([1.3, 2.2], ['sum']))

# Timings are different
%timeit scipy_interp([1.3, 2.2])
%timeit interp([1.3, 2.2])
```

10000 loops, best of 3: 176 µs per loop

The slowest run took 7.10 times longer than the fastest. This could mean that an intermediate result is being cached.

100000 loops, best of 3: 7.71 µs per loop

The `DFInterpolator` is about 30x faster than the `scipy` regular grid interpolation, for a single point. However, for vectorized calculations, `scipy` is indeed faster:

```
[5]: N = 10000
pts = [1.3 * np.ones(N), 2.2 * np.ones(N)]
%timeit scipy_interp(np.array(pts).T)
%timeit interp(pts, ['sum'])
```

The slowest run took 7.51 times longer than the fastest. This could mean that an intermediate result is being cached.

100 loops, best of 3: 1.52 ms per loop

The slowest run took 30.75 times longer than the fastest. This could mean that an intermediate result is being cached.

1 loop, best of 3: 15.1 ms per loop

However, the `DFInterpolator` has an additional advantage of being able to manage missing data—that is, the grid doesn't have to be completely filled to construct the interpolator, as it does with `scipy`:

```
[6]: df_missing = df.drop([(3, 3), (3, 4)])
     df_missing
```

```
[6]:
```

		sum	product	power
x	y			
1	1	2	1	1
	2	3	2	1
	3	4	3	1
	4	5	4	1
	5	6	5	1
2	1	3	2	2
	2	4	4	4
	3	5	6	8
	4	6	8	16
	5	7	10	32
3	1	4	3	3
	2	5	6	9
	5	8	15	243

```
[7]: interp_missing = DFInterpolator(df_missing)
     interp_missing([1.3, 2.2])
```

```
[7]: array([3.5 , 2.86, 2.14])
```

However, if the grid cell that the requested point is in is adjacent to one of these missing points, the interpolation will return nans:

```
[8]: interp_missing([2.3, 3])
```

```
[8]: array([nan, nan, nan])
```

In other words, the interpolator can be constructed with an incomplete grid, but it does not fill values for the missing points.

4.1 Background and EEPs

Stellar model grids are typically constructed as a set of evolutionary tracks, where models of stellar evolution are run on grids of initial mass and metallicity, often with some other physical parameter varied as well (e.g., rotation, helium fraction, α -abundance, etc.). Each of these evolutionary tracks predicts various physical properties (temperature, luminosity, etc.) of a star with given initial mass and metallicity, as a function of age.

It is also often of interest to re-organize these evolution track grids into “isochrones”—sets of stars at a range of masses, all with the same age. As described in [this reference](#), in order to construct these isochrones, the time axis of each evolution track gets mapped into a new coordinate, called “equivalent evolutionary phase,” or EEP. The principle of the EEPs is to first identify physically significant stages in stellar evolution, and then subdivide each of these stages into a number of equal steps. This adaptive sampling enables accurate interpolation between evolution tracks even at ages when stars are evolving quickly, in the post-main sequence phases.

Previous versions of **isochrones** relied directly on these precomputed isochrone grids and interpolated between grid points in `(mass, age, feh)` space. This returned [inaccurate results](#) for post-MS stages of stellar evolution, and thus was not reliable for modeling evolved stars. However, beginning with v2.0, **isochrones** now implements all interpolation using EEPs. In addition, it provides direct access to the evolution track grids, in addition to precomputed isochrone grids. Note that version 2.0 includes only the [MIST](#) models; future updates will include more (e.g. PARSEC, YAPSI).

4.2 Model Grid Objects and Interpolation

isochrones provides a simple and direct interface to full grids of stellar models. Upon first access, the grids are downloaded in original form, reorganized, and written to disk in binary format in order to load quickly with subsequent access. The grids are loaded as pandas dataframes with multi-level indexing that reflects the structure of the grids: evolution track grids are indexed by metallicity, initial mass, and EEP; and isochrone grids by metallicity, age, and EEP.

```
[1]: from isochrones.mist import MISTEvolutionTrackGrid, MISTIsochroneGrid
```

```
track_grid = MISTEvolutionTrackGrid()
track_grid.df.head() # just show first few rows
```

```
[1]:
```

initial_feh	initial_mass	EEP	nu_max	logg	eep	initial_mass	\
-4.0	0.1	1	143.524548	3.033277	1.0	0.1	
		2	145.419039	3.038935	2.0	0.1	
		3	147.409881	3.044805	3.0	0.1	
		4	149.499346	3.050886	4.0	0.1	
		5	151.703570	3.057203	5.0	0.1	

initial_feh	initial_mass	EEP	radius	logTeff	mass	density	Mbol	\
-4.0	0.1	1	1.593804	3.620834	0.1	0.034823	5.132871	
		2	1.583455	3.620769	0.1	0.035510	5.147664	
		3	1.572790	3.620702	0.1	0.036237	5.163015	
		4	1.561817	3.620631	0.1	0.037006	5.178922	
		5	1.550499	3.620558	0.1	0.037823	5.195452	

initial_feh	initial_mass	EEP	phase	feh	Teff	logL	\
-4.0	0.1	1	-1.0	-3.978406	4176.707371	-0.157148	
		2	-1.0	-3.978406	4176.085183	-0.163066	
		3	-1.0	-3.978406	4175.435381	-0.169206	
		4	-1.0	-3.978406	4174.757681	-0.175569	
		5	-1.0	-3.978406	4174.049081	-0.182181	

initial_feh	initial_mass	EEP	delta_nu	interpolated	star_age	age	\
-4.0	0.1	1	21.776686	False	13343.289397	4.125263	
		2	21.993078	False	14171.978264	4.151430	
		3	22.219791	False	15048.910447	4.177505	
		4	22.457004	False	15975.827275	4.203463	
		5	22.706349	False	16962.744747	4.229496	

initial_feh	initial_mass	EEP	dt_deep
-4.0	0.1	1	0.026168
		2	0.026121
		3	0.026016
		4	0.025996
		5	0.025996

```
[2]: iso_grid = MISTIsochroneGrid()
iso_grid.df.head() # just show first few rows
```

```
[2]:
```

log10_isochrone_age_yr	feh	EEP	eep	age	feh	mass	initial_mass	\
5.0	-4.0	35	35	5.0	-3.978406	0.100000	0.100000	
		36	36	5.0	-3.978406	0.102885	0.102885	
		37	37	5.0	-3.978406	0.107147	0.107147	
		38	38	5.0	-3.978406	0.111379	0.111379	
		39	39	5.0	-3.978406	0.115581	0.115581	

log10_isochrone_age_yr	feh	EEP	radius	density	logTeff	Teff	\
------------------------	-----	-----	--------	---------	---------	------	---

(continues on next page)

(continued from previous page)

5.0	-4.0	35	1.106082	0.104184	3.617011	4140.105252
		36	1.122675	0.102507	3.618039	4149.909661
		37	1.147702	0.099921	3.619556	4164.436984
		38	1.173015	0.097287	3.621062	4178.903372
		39	1.198615	0.094627	3.622555	4193.289262
			logg	logL	Mbol	delta_nu \
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35	3.350571	-0.489734	5.964335	37.987066
		36	3.347798	-0.472691	5.921728	37.739176
		37	3.343658	-0.447471	5.858678	37.345115
		38	3.339612	-0.422498	5.796244	36.923615
		39	3.335660	-0.397776	5.734440	36.473151
			nu_max	phase	dm_deep	
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35	299.346079	-1.0	0.002885	
		36	298.570836	-1.0	0.003573	
		37	297.180748	-1.0	0.004247	
		38	295.526946	-1.0	0.004217	
		39	293.589960	-1.0	0.004189	

This generally contains only a subset of the original columns provided by the underlying grid, with standardized names. There are also additional computed columns, such as stellar radius and density. The full, original grids, can be found with the `.df_orig` attribute if desired:

```
[3]: iso_grid.df_orig.head() # just show first few rows
```

```
[3]:
```

log10_isochrone_age_yr	feh	EEP	EEP	log10_isochrone_age_yr	initial_mass	\
5.0	-4.0	35	35	5.0	0.100000	
		36	36	5.0	0.102885	
		37	37	5.0	0.107147	
		38	38	5.0	0.111379	
		39	39	5.0	0.115581	
			star_mass	star_mdot	he_core_mass	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35	0.100000	-1.455094e-13	0.0	
		36	0.102885	-1.562027e-13	0.0	
		37	0.107147	-1.707298e-13	0.0	
		38	0.111379	-1.836256e-13	0.0	
		39	0.115581	-1.949639e-13	0.0	
			c_core_mass	o_core_mass	log_L	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35	0.0	0.0	-0.489734	
		36	0.0	0.0	-0.472691	
		37	0.0	0.0	-0.447471	
		38	0.0	0.0	-0.422498	
		39	0.0	0.0	-0.397776	
			log_L_div_Ledd	...	nu_max	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35	-4.167035	...	299.346079	
		36	-4.129623	...	298.570836	
		37	-4.073591	...	297.180748	

(continues on next page)

(continued from previous page)

			38	-4.017245 ...	295.526946	
			39	-3.960633 ...	293.589960	
				acoustic_cutoff	max_conv_vel_div_csound	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35		2233.536029	0.127243	
		36		2228.014832	0.128938	
		37		2218.440338	0.130528	
		38		2207.403678	0.132657	
		39		2194.776391	0.134294	
				max_gradT_div_grada	gradT_excess_alpha	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35		1.095544	0.0	
		36		1.101114	0.0	
		37		1.109114	0.0	
		38		1.116760	0.0	
		39		1.124050	0.0	
				min_Pgas_div_P	max_L_rad_div_Ledd	\
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35		0.999989	0.000016	
		36		0.999989	0.000017	
		37		0.999988	0.000019	
		38		0.999987	0.000021	
		39		0.999986	0.000022	
				e_thermal	phase	feh
log10_isochrone_age_yr	feh	EEP				
5.0	-4.0	35		3.002314e+46	-1.0	-4.0
		36		3.106838e+46	-1.0	-4.0
		37		3.264230e+46	-1.0	-4.0
		38		3.424460e+46	-1.0	-4.0
		39		3.587613e+46	-1.0	-4.0

[5 rows x 80 columns]

```
[4]: iso_grid.df_orig.columns
```

```
[4]: Index(['EEP', 'log10_isochrone_age_yr', 'initial_mass', 'star_mass',
        'star_mdot', 'he_core_mass', 'c_core_mass', 'o_core_mass', 'log_L',
        'log_L_div_Ledd', 'log_LH', 'log_LHe', 'log_LZ', 'log_Teff',
        'log_abs_Lgrav', 'log_R', 'log_g', 'log_surf_z', 'surf_avg_omega',
        'surf_avg_v_rot', 'surf_num_c12_div_num_o16', 'v_wind_Km_per_s',
        'surf_avg_omega_crit', 'surf_avg_omega_div_omega_crit',
        'surf_avg_v_crit', 'surf_avg_v_div_v_crit', 'surf_avg_Lrad_div_Ledd',
        'v_div_csound_surf', 'surface_h1', 'surface_he3', 'surface_he4',
        'surface_li7', 'surface_be9', 'surface_b11', 'surface_c12',
        'surface_c13', 'surface_n14', 'surface_o16', 'surface_f19',
        'surface_ne20', 'surface_na23', 'surface_mg24', 'surface_si28',
        'surface_s32', 'surface_ca40', 'surface_ti48', 'surface_fe56',
        'log_center_T', 'log_center_Rho', 'center_degeneracy', 'center_omega',
        'center_gamma', 'mass_conv_core', 'center_h1', 'center_he4',
        'center_c12', 'center_n14', 'center_o16', 'center_ne20', 'center_mg24',
        'center_si28', 'pp', 'cno', 'tri_alfa', 'burn_c', 'burn_n', 'burn_o',
        'c12_c12', 'delta_nu', 'delta_Pg', 'nu_max', 'acoustic_cutoff',
        'max_conv_vel_div_csound', 'max_gradT_div_grada', 'gradT_excess_alpha',
```

(continues on next page)

(continued from previous page)

```
'min_Pgas_div_P', 'max_L_rad_div_Ledd', 'e_thermal', 'phase', 'feh'],
dtype='object')
```

Any property (or properties) of these grids can be interpolated to any value of the index parameters via the `.interp` method:

```
[5]: track_grid.interp([-0.12, 1.01, 353.1], ['mass', 'radius', 'logg', 'Teff'])
[5]: array([1.00983180e+00, 1.04691913e+00, 4.40266419e+00, 6.03383320e+03])
```

Similarly, the `.interp_orig` method interpolates any of the original columns by name:

```
[6]: track_grid.interp_orig([-0.12, 1.01, 353.1], ['v_wind_Km_per_s'])
[6]: array([2.87408918e-05])
```

Note that these interpolations are fast—30-40x faster than the equivalent interpolation in `scipy`, for evaluating at a single point:

```
[7]: from scipy.interpolate import RegularGridInterpolator

grid = track_grid.interp.grid[:, :, :, 4] # subgrid corresponding to radius
interp = RegularGridInterpolator(track_grid.interp.index_columns, grid)
assert track_grid.interp([-0.12, 1.01, 353.1], ['radius']) == interp([-0.12, 1.01,
↪ 353.1])
```

```
[8]: %timeit interp([-0.12, 1.01, 353.1])
%timeit track_grid.interp([-0.12, 1.01, 353.1], ['radius'])
```

The slowest run took 14.68 times longer than the fastest. This could mean that an_↪
intermediate result is being cached.
100 loops, best of 3: 1.13 ms per loop
The slowest run took 5.04 times longer than the fastest. This could mean that an_↪
intermediate result is being cached.
10000 loops, best of 3: 12.5 µs per loop

In order to select a subset of these grids, you can use `pandas` multi-index magic:

```
[9]: iso_grid.df.xs((9.0, 0.0), level=(0, 1)).head() # just show first few rows
```

```
[9]:
```

	eeP	age	feh	mass	initial_mass	radius	density	\
EEP								
193	193	9.0	0.042799	0.100000	0.100000	0.126216	70.115891	
194	194	9.0	0.042805	0.103449	0.103449	0.129201	67.622476	
195	195	9.0	0.042814	0.108103	0.108103	0.133343	64.282466	
196	196	9.0	0.042824	0.112932	0.112932	0.137791	60.858291	
197	197	9.0	0.042834	0.117543	0.117544	0.142177	57.660112	
	logTeff		Teff	logg	logL	Mbol	delta_nu	\
EEP								
193	3.460248	2885.680821	5.235913	-3.002129	12.245322	1045.120425		
194	3.462649	2901.679870	5.227759	-2.972222	12.170556	1025.261668		
195	3.465890	2923.411541	5.216743	-2.931855	12.069637	998.429682		
196	3.469256	2946.160133	5.205249	-2.889887	11.964717	970.463953		
197	3.472471	2968.048014	5.194272	-2.849811	11.864529	943.753111		
	nu_max	phase	dm_deep					
EEP								

(continues on next page)

(continued from previous page)

193	27533.135930	-1.0	0.003449
194	27025.521973	-1.0	0.004051
195	26339.315728	-1.0	0.004742
196	25622.963627	-1.0	0.004720
197	24938.691941	-1.0	0.004735

4.3 Example visualization

Just for fun, let's plot a few isochrones:

```
[10]: import hvplot.pandas

# Select two isochrones from the grid
iso_df1 = iso_grid.df.xs((9.0, 0.0), level=(0, 1))
iso_df2 = iso_grid.df.xs((9.5, 0.0), level=(0, 1))

options = dict(invert_xaxis=True, legend_position='bottom_left')

# Isn't hvplot/holoviews great?
plot1 = iso_df1.hvplot.line('logTeff', 'logL', label='Log(age) = 9.0')
plot2 = iso_df2.hvplot.line('logTeff', 'logL', label='Log(age) = 9.5')
(plot1 * plot2).options(**options)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[10]: :Overlay
      .Curve.Log_left_parenthesis_age_right_parenthesis_equals_9_full_stop_0 :Curve
      ↪[logTeff]      (logL)
      .Curve.Log_left_parenthesis_age_right_parenthesis_equals_9_full_stop_5 :Curve
      ↪[logTeff]      (logL)
```

Bolometric correction grids

Bolometric correction is defined as the difference between the apparent bolometric magnitude of a star and its apparent magnitude in a particular bandpass:

$$BC_x = m_{bol} - m_x$$

The MIST project provide [grids of bolometric corrections](#) in many photometric systems as a function of stellar temperature, surface gravity, metallicity, and A_V extinction. This allows for accurate conversion of bolometric magnitude of a star (available from the theoretical grids) to magnitude in any band, at any extinction (and distance), without the need for any “effective wavelength” approximation (used in **isochrones** prior to v2.0), which breaks down for broad bandpasses and large extinctions. These grids are downloaded, organized, stored, and interpolated in much the same manner as the model grids.

```
[1]: from isochrones.mist.bc import MISTBolometricCorrectionGrid

bc_grid = MISTBolometricCorrectionGrid(['J', 'H', 'K', 'G', 'BP', 'RP', 'g', 'r', 'i',
↪ ''])
```

```
[2]: bc_grid.df.head()
```

```
[2]:
```

				g	r	i	J	H	\
Teff	logg	[Fe/H]	Av						
2500.0	-4.0	-4.0	0.00	-6.534742	-3.332877	-1.617626	1.845781	2.927064	
			0.05	-6.590469	-3.375570	-1.650338	1.831466	2.917990	
			0.10	-6.646182	-3.418258	-1.683043	1.817153	2.908916	
			0.15	-6.701881	-3.460939	-1.715740	1.802841	2.899842	
			0.20	-6.757566	-3.503615	-1.748429	1.788530	2.890769	
				K	G	BP	RP		
Teff	logg	[Fe/H]	Av						
2500.0	-4.0	-4.0	0.00	3.436304	-2.181986	-4.652544	-0.881255		
			0.05	3.430463	-2.211637	-4.697700	-0.909057		
			0.10	3.424623	-2.241240	-4.742838	-0.936829		
			0.15	3.418782	-2.270797	-4.787959	-0.964571		
			0.20	3.412942	-2.300306	-4.833062	-0.992285		

```
[3]: bc_grid.interp.index_names
[3]: FrozenList(['Teff', 'logg', '[Fe/H]', 'Av'])

[4]: bc_grid.interp([5770, 4.44, 0.0, 0.], ['G', 'K'])
[4]: array([0.0819599 , 1.45398088])
```

The bandpasses provided to initialize the grid object are parsed according to the `.get_band` method, which returns the photometric system and the name of the band in the system:

```
[5]: bc_grid.get_band('G'), bc_grid.get_band('g')
[5]: (('UBVRiplus', 'Gaia_G_DR2Rev'), ('SDSSugriz', 'SDSS_g'))
```

Not all bands have cute nicknames to them, so you can also be explicit, e.g.:

```
[6]: bc_grid.get_band('DECam_g')
[6]: ('DECam', 'DECam_g')
```

See the implementation of `.get_band` for details.

ModelGridInterpolator

In practice, interaction with the model grid and bolometric correction objects is easiest through a `ModelGridInterpolator` object, which brings the two together. This object is the replacement of the `Isochrone` object from previous generations of this package, though it has a slightly different API. It is mostly backward compatible, except for the removal of the `.mag` function dictionary for interpolating apparent magnitudes, this being replaced by the `.interp_mag` method.

6.1 Isochrones

An `IsochroneInterpolator` object takes `[EEP, log(age), feh]` as parameters.

```
[1]: from isochrones.mist import MIST_Isochrone

mist = MIST_Isochrone()

pars = [353, 9.78, -1.24] # eep, log(age), feh
mist.interp_value(pars, ['mass', 'radius', 'Teff'])

[1]: array([7.93829519e-01, 7.91444054e-01, 6.30305932e+03])
```

To interpolate apparent magnitudes, add distance [pc] and A_V extinction as parameters.

```
[2]: mist.interp_mag(pars + [200, 0.11], ['K', 'BP', 'RP']) # Returns Teff, logg, feh,
↪mags

[2]: (6303.059322477636,
4.540738764316164,
-1.377262817643937,
array([10.25117074, 11.73997159, 11.06529993]))
```

6.2 Evolution tracks

Note that you can do the same using an `EvolutionTrackInterpolator` rather than an isochrone grid, using `[mass, EEP, feh]` as parameters:

```
[3]: from isochrones.mist import MIST_EvolutionTrack

mist_track = MIST_EvolutionTrack()

pars = [0.794, 353, -1.24] # mass, eep, feh [matching above]
mist_track.interp_value(pars, ['mass', 'radius', 'Teff', 'age'])

[3]: array([7.93843749e-01, 7.91818696e-01, 6.31006708e+03, 9.77929505e+00])

[4]: mist_track.interp_mag(pars + [200, 0.11], ['K', 'BP', 'RP'])

[4]: (6310.067080800683,
      4.54076772643659,
      -1.372925841944066,
      array([10.24893319, 11.73358578, 11.06056746]))
```

There are also convenience methods (for both isochrones and tracks) if you prefer (and for backward compatibility—note that the parameters must be unpacked, unlike the calls to `.interp_value` and `.interp_mag`), though it is slower to call multiple of these than to call `.interp_value` once with several desired outputs:

```
[5]: mist_track.mass(*pars)

[5]: array(0.79384375)
```

You can also get the dataframe of a single isochrone (interpolated to any age or metallicity) as follows:

```
[6]: mist.isochrone(9.53, 0.1).head() # just show first few rows

[6]:
```

	eep	age	feh	mass	initial_mass	radius	density	\
223	223.0	9.53	0.150280	0.143050	0.143050	0.174516	42.182044	
224	224.0	9.53	0.150322	0.147584	0.147584	0.178799	40.088758	
225	225.0	9.53	0.150371	0.152520	0.152521	0.183594	37.948464	
226	226.0	9.53	0.150419	0.157318	0.157319	0.184463	37.208965	
227	227.0	9.53	0.150468	0.161795	0.161796	0.189168	35.381629	

	logTeff	Teff	logg	...	H_mag	K_mag	\
223	3.477544	3003.536405	5.121475	...	8.785652	8.559155	
224	3.479902	3019.769652	5.112821	...	8.713187	8.487450	
225	3.482375	3036.910262	5.103613	...	8.635963	8.411037	
226	3.480519	3024.116433	5.101786	...	8.629300	8.403586	
227	3.482801	3040.176145	5.093340	...	8.558717	8.333774	

	G_mag	BP_mag	RP_mag	W1_mag	W2_mag	W3_mag	TESS_mag	\
223	12.766111	14.751368	11.522764	8.398324	8.200245	8.032482	11.381237	
224	12.662468	14.612205	11.426131	8.327414	8.129809	7.964879	11.287794	
225	12.552453	14.464800	11.323512	8.251886	8.054820	7.892865	11.188540	
226	12.569050	14.507862	11.334820	8.243224	8.045057	7.881000	11.197325	
227	12.467864	14.371759	11.240553	8.174286	7.976668	7.815386	11.106209	

	Kepler_mag
223	12.864034
224	12.755405
225	12.640135

(continues on next page)

(continued from previous page)

```
226 12.660600
227 12.554499
```

```
[5 rows x 27 columns]
```

6.3 Generating synthetic properties

Often one wants to use stellar model grids to generate synthetic properties of stars. This can be done in a couple different ways, depending on what information you are able to provide. If you happen to have EEP values, you can use the fact that a `ModelGridInterpolator` is callable. Note that it takes the same parameters as all the other interpolation calls, with `distance` and `AV` as optional keyword parameters.

```
[7]: from isochrones.mist import MIST_EvolutionTrack
```

```
mist_track = MIST_EvolutionTrack()
mist_track([0.8, 0.9, 1.0], 350, 0.0, distance=100, AV=0.1)
```

```
[7]:
```

	nu_max	logg	eep	initial_mass	radius	logTeff	mass	\
0	4254.629601	4.548780	350.0	0.8	0.787407	3.707984	0.799894	
1	3622.320906	4.495440	350.0	0.9	0.888064	3.741043	0.899876	
2	3041.107996	4.432089	350.0	1.0	1.006928	3.766249	0.999860	

	density	Mbol	phase	...	H_mag	K_mag	G_mag	\
0	2.309938	5.792554	0.0	...	9.040105	8.972502	10.872154	
1	1.811405	5.200732	0.0	...	8.667003	8.614974	10.224076	
2	1.380733	4.675907	0.0	...	8.312159	8.270380	9.679997	

	BP_mag	RP_mag	W1_mag	W2_mag	W3_mag	TESS_mag	Kepler_mag
0	11.328425	10.258543	8.945414	8.989254	8.921756	10.247984	10.773706
1	10.602874	9.678976	8.593946	8.622577	8.575349	9.671007	10.129692
2	10.005662	9.186910	8.253638	8.269467	8.238306	9.180275	9.590731

```
[3 rows x 29 columns]
```

Often, however, you will not know the EEP values at which you wish to simulate your synthetic population. In this case, you can use the `.generate()` method.

```
[8]: mist_track.generate([0.81, 0.91, 1.01], 9.51, 0.01)
```

```
[8]:
```

	nu_max	logg	eep	initial_mass	radius	logTeff	mass	\
0	4787.598310	4.595858	320.808	0.81	0.750611	3.699978	0.809963	
1	3986.671794	4.535170	332.280	0.91	0.853120	3.737424	0.909935	
2	3154.677953	4.447853	343.800	1.01	0.993830	3.766201	1.009887	

	density	Mbol	phase	...	H	K	G	\
0	2.703461	5.977047	0.0	...	4.154396	4.088644	5.988091	
1	2.066995	5.324246	0.0	...	3.747329	3.699594	5.264620	
2	1.451510	4.705019	0.0	...	3.322241	3.286761	4.620132	

	BP	RP	W1	W2	W3	TESS	Kepler
0	6.444688	5.375415	4.066499	4.117992	4.047535	5.365712	5.887722
1	5.632088	4.731978	3.684034	3.718112	3.670736	4.725020	5.169229
2	4.925805	4.148936	3.276062	3.295002	3.266166	4.143362	4.531319

```
[3 rows x 29 columns]
```

Under the hood, `.generate()` uses an interpolation step to approximate the eep value(s) corresponding to the requested value(s) of mass, age, and metallicity:

```
[9]: mist_track.get_eep(1.01, 9.51, 0.01)
[9]: 343.8
```

Because this is fast, it is pretty inexpensive to generate a population of stars with given properties:

```
[10]: import numpy as np

N = 10000
mass = np.ones(N) * 1.01
age = np.ones(N) * 9.82
feh = np.ones(N) * 0.02
%timeit mist_track.generate(mass, age, feh)

10 loops, best of 3: 112 ms per loop
```

Note though, that this interpolation doesn't do great for evolved stars (this is the fundamental reason why **isochrones** always fits with EEP as one of the parameters). However, if you do want to compute more precise EEP values for given physical properties, you can set the `accurate` keyword parameter, which performs a function minimization:

```
[11]: mist_track.get_eep(1.01, 9.51, 0.01, accurate=True)
[11]: 343.1963539123535
```

This is more accurate, but slow because it is actually performing a function minimization:

```
[12]: %timeit mist_track.get_eep(1.01, 9.51, 0.01, accurate=True)
%timeit mist_track.get_eep(1.01, 9.51, 0.01)

100 loops, best of 3: 4.56 ms per loop
The slowest run took 4.98 times longer than the fastest. This could mean that an
↳ intermediate result is being cached.
100000 loops, best of 3: 4.26 µs per loop
```

Here we can see the effect of accuracy by plugging back in the estimated EEP into the interpolation:

```
[13]: [mist_track.interp_value([1.01, e, 0.01], ['age']) for e in [343.8, 343.
↳ 1963539123535]]
[13]: [array([9.51806019]), array([9.50999994])]
```

So if accuracy is required, definitely use `accurate=True`, but for most purposes, the default should be fine. You can request that `.generate()` run in “accurate” mode, which uses this more expensive EEP computation (it will be correspondingly slower).

```
[14]: mist_track.generate([0.81, 0.91, 1.01], 9.51, 0.01, accurate=True)

[14]:      nu_max      logg      eep  initial_mass      radius  logTeff  \
0  4794.035436  4.596385  320.219650         0.81  0.750156  3.699863
1  3995.692509  4.536089  331.721363         0.91  0.852218  3.737300
2  3168.148566  4.449647  343.196354         1.01  0.991781  3.766083

      mass  density      Mbol  phase  ...      H      K  \
0  0.809963  2.708365  5.979507    0.0  ...  4.156117  4.090301
1  0.909936  2.073560  5.327785    0.0  ...  3.750018  3.702214
2  1.009890  1.460523  4.710671    0.0  ...  3.327067  3.291533
```

(continues on next page)

(continued from previous page)

	G	BP	RP	W1	W2	W3	TESS	\
0	5.990784	6.447681	5.377849	4.068141	4.119700	4.049167	5.368138	
1	5.268320	5.636100	4.735394	3.686635	3.720795	3.673334	4.728428	
2	4.625783	4.931724	4.154311	3.280826	3.299859	3.270940	4.148735	

	Kepler
0	5.890400
1	5.172899
2	4.536929

[3 rows x 29 columns]

Just for curiosity, let's look at the difference in the predictions:

```
[15]: df0 = mist_track.generate([0.81, 0.91, 1.01], 9.51, 0.01, accurate=True)
      df1 = mist_track.generate([0.81, 0.91, 1.01], 9.51, 0.01)
      ((df1 - df0) / df0).mean()
```

```
[15]: nu_max      -0.002617
      logg        -0.000240
      eep         0.001760
      initial_mass 0.000000
      radius      0.001243
      logTeff     0.000032
      mass        -0.000002
      density     -0.003716
      Mbol        -0.000759
      phase       NaN
      feh         -0.057173
      Teff        0.000273
      logL        0.061576
      delta_nu    -0.001803
      interpolated NaN
      star_age    0.018487
      age         0.000837
      dt_deep     -0.007171
      J           -0.000848
      H           -0.000861
      K           -0.000854
      G           -0.000791
      BP          -0.000792
      RP          -0.000823
      W1          -0.000854
      W2          -0.000869
      W3          -0.000857
      TESS        -0.000823
      Kepler      -0.000800
      dtype: float64
```

Not too bad, for this example!

6.4 Demo: Visualize

Now let's make sure that interpolated isochrones fall nicely between ones that are actually part of the grid. In order to execute this code, you will need to

```
conda install -c pyviz pyviz
```

and to execute in JupyterLab, you will need to

```
jupyter labextension install @pyviz/jupyterlab_pyviz
```

```
[16]: import hvplot.pandas

iso1 = mist.model_grid.df.xs((9.5, 0.0), level=(0, 1)) # extract subgrid at log_
↳age=9.5, feh=0.0
iso2 = mist.model_grid.df.xs((9.5, 0.25), level=(0, 1)) # extract subgrid at log_
↳age=9.5, feh=0.25
iso3 = mist.isochrone(9.5, 0.12) # should be between the other two

plot1 = iso1.hvplot.line('logTeff', 'logL', label='[Fe/H] = 0.0')
plot2 = iso2.hvplot.line('logTeff', 'logL', label='[Fe/H] = 0.25')
plot3 = iso3.hvplot.line('logTeff', 'logL', label='[Fe/H] = 0.12')

(plot1 * plot2 * plot3).options(invert_axis=True, legend_position='bottom_left',
↳width=600)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[16]: :Overlay
      .Curve.Left_square_bracket_Fe_over_H_right_square_bracket_equals_0_full_stop_0 :
↳Curve [logTeff] (logL)
      .Curve.Left_square_bracket_Fe_over_H_right_square_bracket_equals_0_full_stop_25 :
↳Curve [logTeff] (logL)
      .Curve.Left_square_bracket_Fe_over_H_right_square_bracket_equals_0_full_stop_12 :
↳Curve [logTeff] (logL)
```

Fitting stellar parameters

The central purpose of **isochrones** is to infer the physical properties of stars given arbitrary observations. This is accomplished via the `StarModel` object. For simplest usage, a `StarModel` is initialized with a `ModelGridInterpolator` and observed properties, provided as (value, uncertainty) pairs. Also, while the vanilla `StarModel` object (which is mostly the same as the **isochrones** v1 `StarModel` object) can still be used to fit a single star, **isochrones** v2 has a new `SingleStarModel` available, that has a more optimized likelihood implementation, for significantly faster inference.

7.1 Defining a star model

First, let's generate some “observed” properties according to the model grids themselves. Remember that `.generate()` only works with the evolution track interpolator.

```
[1]: from isochrones.mist import MIST_EvolutionTrack, MIST_Isochrone

track = MIST_EvolutionTrack()

mass, age, feh, distance, AV = 1.0, 9.74, -0.05, 100, 0.02

# Using return_dict here rather than return_df, because we just want scalar values
true_props = track.generate(mass, age, feh, distance=distance, AV=AV, return_
    ↳dict=True)
true_props

[1]: {'nu_max': 2617.5691700617886,
      'logg': 4.370219109480715,
      'teff': 380.0,
      'initial_mass': 1.0,
      'radius': 1.0813017873811603,
      'logTeff': 3.773295968705084,
      'mass': 0.9997797219140423,
      'density': 1.115827651504971,
      'Mbol': 4.4508474939623826,
```

(continues on next page)

(continued from previous page)

```
'phase': 0.0,
'feh': -0.09685557997282962,
'Teff': 5934.703385987951,
'logL': 0.11566100241504726,
'delta_nu': 126.60871562200438,
'interpolated': 0.0,
'star_age': 5522019067.711771,
'age': 9.74119762492735,
'dt_deep': 0.0036991465241712263,
'J': 8.435233804866742,
'H': 8.124109062114325,
'K': 8.09085566863133,
'G': 9.387465543790636,
'BP': 9.680097761608252,
'RP': 8.928888526297722,
'W1': 8.079124865544092,
'W2': 8.090757185192754,
'W3': 8.06683507215844,
'TESS': 8.923262483762786,
'Kepler': 9.301490687837552}
```

Now, we can define a `starmodel` with these “observations”, this time using the isochrone grid interpolator. We use the optimized `SingleStarModel` object.

```
[2]: from isochrones import SingleStarModel, get_ichrone

mist = get_ichrone('mist')

uncs = dict(Teff=80, logg=0.1, feh=0.1, phot=0.02)
props = {p: (true_props[p], uncs[p]) for p in ['Teff', 'logg', 'feh']}
props.update({b: (true_props[b], uncs['phot']) for b in 'JHK'})

# Let's also give an appropriate parallax, in mas
props.update({'parallax': (1000./distance, 0.1)})

mod = SingleStarModel(mist, name='demo', **props)
```

And we can see the prior, likelihood, and posterior at the true parameters:

```
[3]: eep = mist.get_eep(mass, age, feh, accurate=True)
pars = [eep, age, feh, distance, AV]

mod.lnprior(pars), mod.lnlike(pars), mod.lnpost(pars)

[3]: (-23.05503287088296, -20.716150242083508, -43.77118311296647)
```

If we stray from these parameters, we can see the likelihood decrease:

```
[4]: pars2 = [eep + 3, age - 0.05, feh + 0.02, distance, AV]
mod.lnprior(pars2), mod.lnlike(pars2), mod.lnpost(pars2)

[4]: (-23.251706955307853, -85.08590699022739, -108.33761394553524)
```

How long does a posterior evaluation take?

```
[5]: %timeit mod.lnpost(pars)
```



```
1000 loops, best of 3: 369 µs per loop
```

```
[6]: from isochrones import BinaryStarModel
```

```
mod2 = BinaryStarModel(mist, **props)
```

```
[7]: pars2 = [eep, eep - 20, age, feh, distance, AV]
      %timeit mod2.lnpost(pars2)
```

The slowest run took 373.39 times longer than the fastest. This could mean that an `↪` intermediate result is being cached.
1000 loops, best of 3: 429 µs per loop

```
[8]: from isochrones import TripleStarModel
      mod3 = TripleStarModel(mist, **props)
      pars3 = [eep, eep-20, eep-40, age, feh, distance, AV]
      %timeit mod3.lnpost(pars3)
```

```
1000 loops, best of 3: 541 µs per loop
```

7.2 Priors

As you may have noticed, we have not explicitly defined any priors on our parameters. They were defined for you, but you may wish to know what they are, and/or to change them.

```
[9]: mod._priors
```

```
[9]: {'mass': <isochrones.priors.ChabrierPrior at 0x1c47e270f0>,
      'feh': <isochrones.priors.FehPrior at 0x1c47e27358>,
      'age': <isochrones.priors.AgePrior at 0x1c47e27748>,
      'distance': <isochrones.priors.DistancePrior at 0x1c47e27390>,
      'AV': <isochrones.priors.AVPrior at 0x1c47e27400>,
      'eep': <isochrones.priors.EEP_prior at 0x1c47e274e0>}
```

You can sample from these priors:

```
[10]: samples = mod.sample_from_prior(1000)
      samples
```

```
[10]:
```

	age	feh	distance	AV	eep
0	9.775384	0.004928	9585.312354	0.058294	415
1	9.690678	0.318313	5460.742158	0.212007	295
2	9.317426	-0.008935	5381.226921	0.144259	265
3	9.721345	-0.131058	7867.502875	0.228851	295
4	9.374286	-0.325079	9590.728624	0.954659	350
5	9.293293	0.229220	9574.055273	0.713866	293
6	9.941975	0.178338	7788.336554	0.100102	272
7	9.436477	0.231631	9148.585364	0.204715	314
8	9.743647	-0.396267	6767.456426	0.383272	460
9	9.607588	-0.236938	4317.243131	0.795216	327
10	10.057273	-0.236801	9031.515061	0.488995	314
11	9.645887	-0.338786	9055.303060	0.408045	1702
12	9.997611	-0.214726	8452.833760	0.398581	327
13	9.926111	-0.063765	9726.761618	0.544188	321
14	9.845896	-0.106017	9148.167681	0.455272	292

(continues on next page)

(continued from previous page)

15	9.761015	-0.051480	8247.211954	0.379722	253
16	9.286601	-0.186755	7113.433648	0.504276	451
17	8.124025	-0.138695	9335.360257	0.445926	380
18	9.357514	0.067101	7737.470105	0.752912	300
19	9.595645	0.107345	7115.589991	0.231624	420
20	9.933118	-0.048234	8424.490753	0.139511	1692
21	10.105196	0.292670	9485.836532	0.366055	331
22	10.040531	0.014999	9296.367644	0.185203	314
23	9.967137	0.000672	8552.101985	0.791577	254
24	10.128790	-0.176654	6727.495813	0.494750	345
25	9.854273	-0.183929	5380.753272	0.945978	407
26	9.808855	0.074433	8664.219066	0.692098	455
27	9.917164	0.202846	7496.956324	0.363808	389
28	9.630302	-0.373880	8555.632299	0.287120	322
29	9.179908	-0.199734	9856.026771	0.303459	300
..	...	...	...	...	...
970	10.086391	0.106938	7592.165249	0.180709	314
971	10.085083	-0.036016	7027.634747	0.314156	299
972	9.287464	0.265288	6042.848103	0.144724	402
973	9.955503	-0.080510	6262.492050	0.611200	327
974	9.616142	-0.342509	5069.839242	0.567513	349
975	10.075706	-0.130103	5829.432844	0.892020	299
976	10.028535	0.193184	4257.798238	0.293645	329
977	9.574676	0.085395	9752.502653	0.703944	357
978	9.556853	-0.140753	8530.526505	0.871235	334
979	9.821810	-0.118098	8972.965633	0.026728	397
980	10.105103	0.104010	9992.769437	0.343932	292
981	9.853596	-0.118408	6035.299187	0.686813	254
982	9.312975	-0.038113	8689.991781	0.047170	324
983	8.971418	0.238024	5797.572151	0.773175	268
984	9.664546	-0.028603	9719.254429	0.707218	347
985	9.924745	-0.216946	9422.814918	0.292175	292
986	9.624291	-0.034933	4359.825182	0.661057	317
987	9.947794	-0.264508	8355.572420	0.301372	292
988	9.766796	0.070148	9155.900363	0.597846	292
989	9.960194	0.026427	7655.336536	0.002166	265
990	9.488527	-0.094431	9896.426901	0.662185	271
991	10.105075	0.145627	3359.853867	0.416843	489
992	9.994699	-0.246844	6033.657596	0.198885	271
993	9.369871	0.052412	2669.191340	0.294969	317
994	9.903893	0.176871	8832.480953	0.128152	295
995	9.864971	0.142656	9366.389176	0.782361	347
996	9.968736	0.027372	8748.808096	0.300267	351
997	9.525780	0.005426	6651.084393	0.508013	247
998	9.290521	-0.064370	8489.972371	0.615413	296
999	10.131466	0.077783	8694.197822	0.833533	271

[1000 rows x 5 columns]

Remember, these are the fit parameters:

```
[11]: mod.param_names
```

```
[11]: ('eep', 'age', 'feh', 'distance', 'AV')
```

Let's turn this into a dataframe, and visualize it.

```
[12]: import pandas as pd
import holoviews as hv
import hvplot.pandas
hv.extension('bokeh')

def plot_samples(samples):
    df = pd.DataFrame(samples, columns=['eep', 'age', 'feh', 'distance', 'AV'])
    df['mass'] = mod.ic.interp_value([df.eep, df.age, df.feh], ['mass'])
    return hv.Layout([df.hvplot.hist(c).options(width=300) for c in df.columns]).
    ↪cols(3)

plot_samples(samples)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[12]: :Layout
      .Histogram.I      :Histogram    [eep]    (eep_frequency)
      .Histogram.II     :Histogram    [age]    (age_frequency)
      .Histogram.III    :Histogram    [feh]    (feh_frequency)
      .Histogram.IV     :Histogram    [distance] (distance_frequency)
      .Histogram.V      :Histogram    [AV]     (AV_frequency)
      .Histogram.VI     :Histogram    [mass]   (mass_frequency)
```

Note that there are some built-in defaults here to be aware of. The metallicity distribution is based on a local metallicity prior from SDSS, the distance prior has a maximum distance of 10kpc, and AV is flat from 0 to 1. Now, let's change our distance prior to only go out to 1000pc, and our metallicity distribution to be flat between -2 and 0.5.

```
[13]: from isochrones.priors import FlatPrior, DistancePrior
mod.set_prior(feh=FlatPrior((-2, 0.5)), distance=DistancePrior(1000))
```

```
[14]: plot_samples(mod.sample_from_prior(1000))
```

```
[14]: :Layout
      .Histogram.I      :Histogram    [eep]    (eep_frequency)
      .Histogram.II     :Histogram    [age]    (age_frequency)
      .Histogram.III    :Histogram    [feh]    (feh_frequency)
      .Histogram.IV     :Histogram    [distance] (distance_frequency)
      .Histogram.V      :Histogram    [AV]     (AV_frequency)
      .Histogram.VI     :Histogram    [mass]   (mass_frequency)
```

Also note that the default mass prior is the Chabrier broken powerlaw, which is nifty:

```
[15]: pd.Series(mod._priors['mass'].sample(10000), name='mass').hvplot.hist(bins=100, bin_
    ↪range=(0, 5))
```

```
[15]: :Histogram      [mass]      (mass_frequency)
```

You can also define a metallicity prior to have a different mix of halo and (local) disk:

```
[16]: from isochrones.priors import FehPrior

pd.Series(FehPrior(halo_fraction=0.5).sample(10000), name='feh').hvplot.hist()

[16]: :Histogram      [feh]      (feh_frequency)
```

7.3 Sampling the posterior

Once you have defined your stellar model and are happy with your priors, you may either execute your optimization/sampling method of choice using the `.lnpost()` method as your posterior, or you may use the built-in **MultiNest** fitting routine with `.fit()`. One thing to note especially is that the **MultiNest** chains get automatically created in a `chains` subdirectory from wherever you execute `.fit()`, with a basename for the files that you can access with:

```
[17]: mod.mnest_basename
[17]: './chains/demo-mist-single-'
```

This can be changed or overwritten in two ways, which is often a good idea to avoid clashes between different fits with the same default basename. You can either pass an explicit `basename` keyword to `.fit()`, or you can set a `name` attribute, as we did when initializing this model. OK, now we will run the fit. This will typically take a few minutes (unless the chains for the fit have already completed, in which case it will be read in and finish quickly).

```
[18]: mod.fit()

INFO:root:MultiNest basename: ./chains/demo-mist-single-
```

The posterior samples of the sampling parameters are available in the `.samples` attribute. Note that this is different from the original vanilla `StarModel` object (the one fully backward-compatible with **isochrones v1**), which contained both sampling parameters and derived parameters at the values of those samples.

```
[19]: mod.samples.head()

[19]:
```

	eep	age	feh	distance	AV	lnprob
0	306.566211	8.867352	-0.084787	99.754595	0.128567	-51.124447
1	385.106002	9.744207	0.179155	99.818131	0.492972	-49.729296
2	301.018106	8.745846	-0.030370	100.473273	0.579248	-49.425361
3	259.680008	8.214644	0.010053	98.376332	0.363801	-48.479515
4	380.210824	9.700131	-0.178482	99.398149	0.633511	-48.453800

```
[20]: mod.samples.describe()

[20]:
```

	eep	age	feh	distance	AV	\
count	5344.000000	5344.000000	5344.000000	5344.000000	5344.000000	
mean	373.193478	9.686096	-0.045856	100.021338	0.146828	
std	19.800711	0.181541	0.076720	0.993574	0.111217	
min	217.053516	7.653568	-0.301781	96.356886	0.000078	
25%	359.164345	9.599845	-0.098619	99.355457	0.060500	
50%	375.054046	9.712589	-0.045498	100.013786	0.124330	
75%	387.980876	9.806347	0.007986	100.685282	0.208648	
max	420.506604	10.089448	0.193737	103.660815	0.755101	

(continues on next page)

(continued from previous page)

```

lnprob
count  5344.000000
mean   -40.820994
std     1.525383
min    -51.124447
25%    -41.532015
50%    -40.468810
75%    -39.723250
max     -38.472525

```

The derived parameters are available in `.derived_samples` (StarModel on its own does not have this attribute):

```
[21]: mod.derived_samples.head()
```

```

[21]:
      eep      age      feh      mass  initial_mass  radius  density \
0  306.566211  8.867352 -0.068345  1.098369    1.098407  1.037567  1.391965
1  385.106002  9.744207  0.159727  1.057168    1.057394  1.141433  1.002494
2  301.018106  8.745846 -0.011900  1.152686    1.152721  1.096043  1.237448
3  259.680008  8.214644  0.048588  1.147012    1.147023  1.065911  1.338449
4  380.210824  9.700131 -0.257214  1.010634    1.010881  1.123263  1.005285

      logTeff      Teff      logg      ...  BP_mag  RP_mag  W1_mag  \
0  3.790660  6176.198702  4.447143  ...    9.673531  8.942160  8.124458
1  3.763116  5796.649384  4.347338  ...   10.181673  9.176220  8.004902
2  3.796728  6262.723700  4.420403  ...    9.982068  9.077651  8.030137
3  3.790769  6177.197873  4.442458  ...    9.830745  8.993534  8.044905
4  3.787708  6134.335252  4.341725  ...   10.085126  9.130129  7.986850

      W2_mag  W3_mag  TESS_mag  Kepler_mag  parallax  distance  AV
0  8.127637  8.109427  8.936103    9.307189  10.024601   99.754595  0.128567
1  8.018791  7.972552  9.164784    9.669073  10.018220   99.818131  0.492972
2  8.021413  7.994322  9.066848    9.529013   9.952896  100.473273  0.579248
3  8.045347  8.020660  8.985118    9.409833  10.165047   98.376332  0.363801
4  7.974103  7.941984  9.117792    9.608276  10.060550   99.398149  0.633511

[5 rows x 30 columns]

```

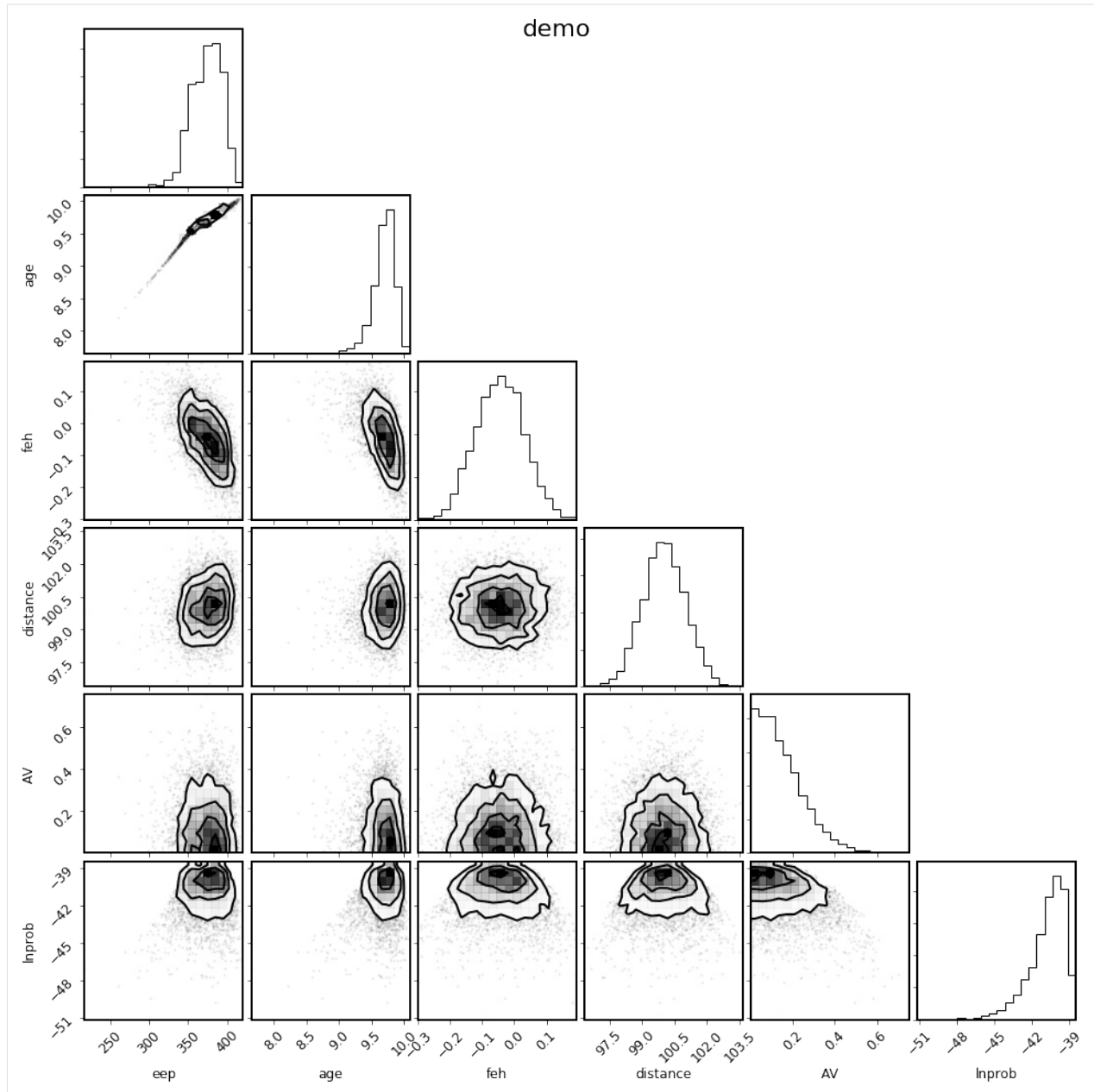
You can make a corner plot of the fit parameters as follows:

```

[22]: %matplotlib inline

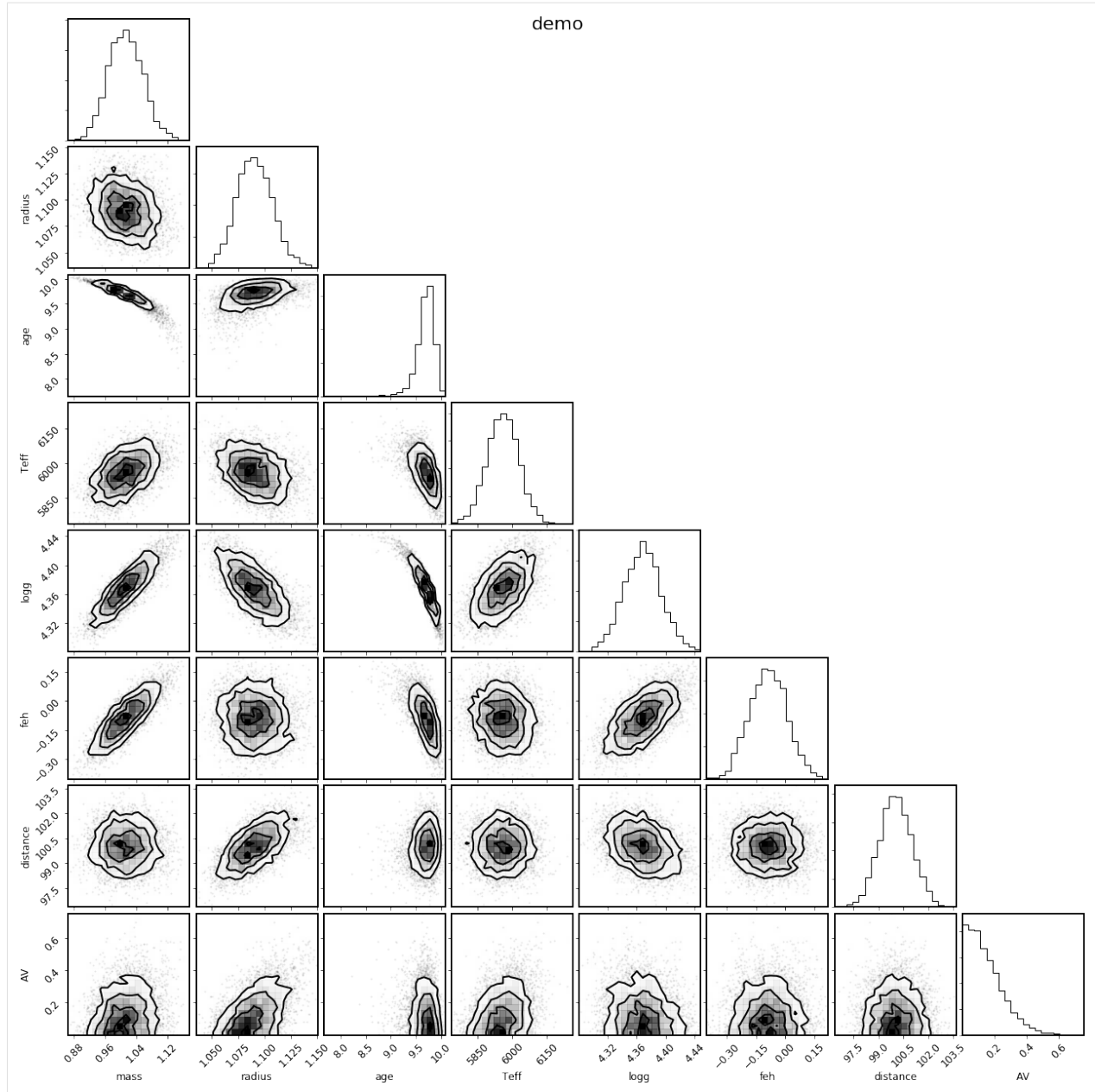
mod.corner_params(); # Note, this is also new in v2.0, for the SingleStarModel object

```



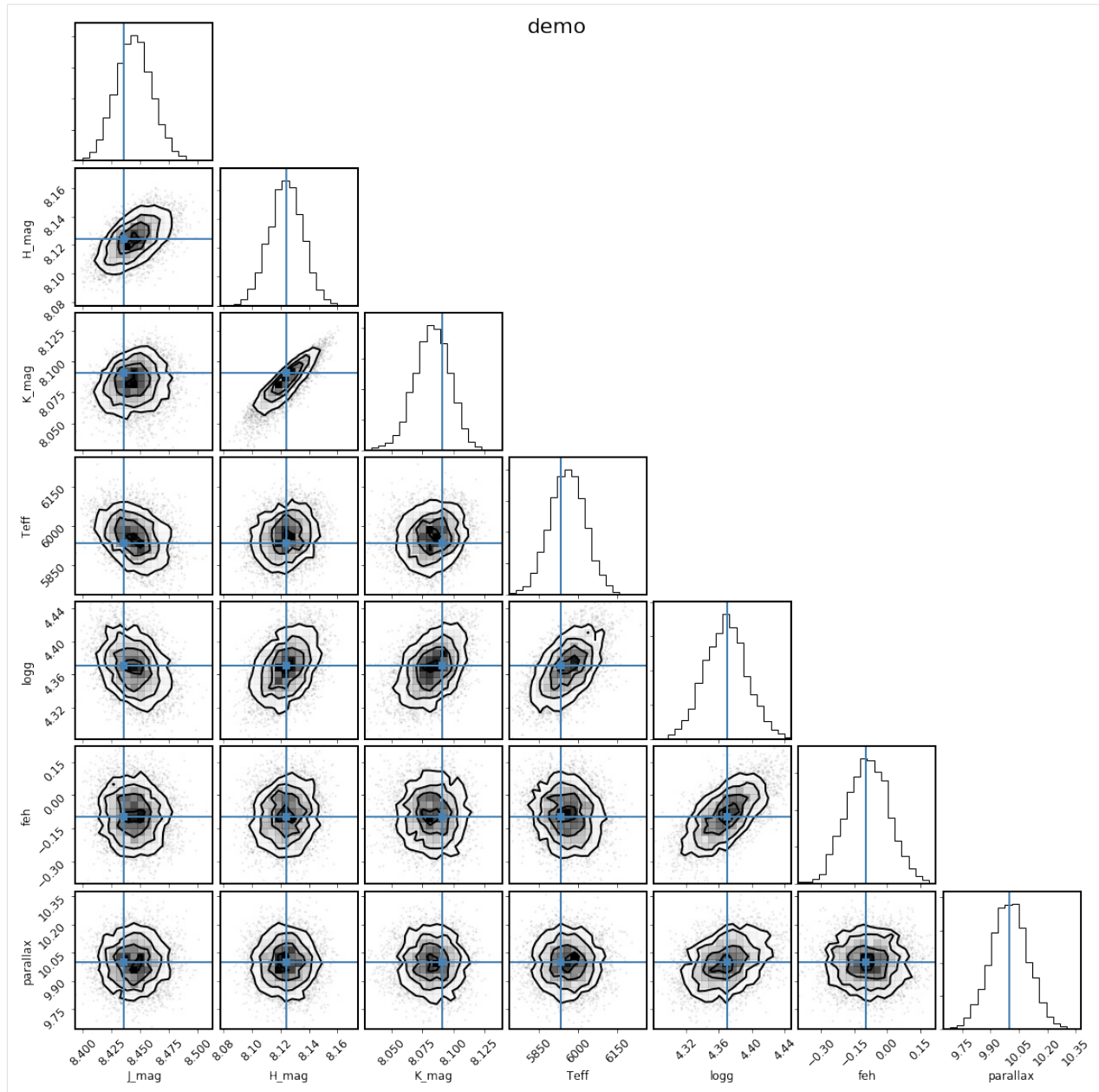
There is also a convenience method to select the parameters of physical interest.

```
[23]: mod.corner_physical();
```



It can also be instructive to see how the derived samples of the observed parameters compare to the observations themselves; the shortcut to this is with the `.corner_observed()` convenience method:

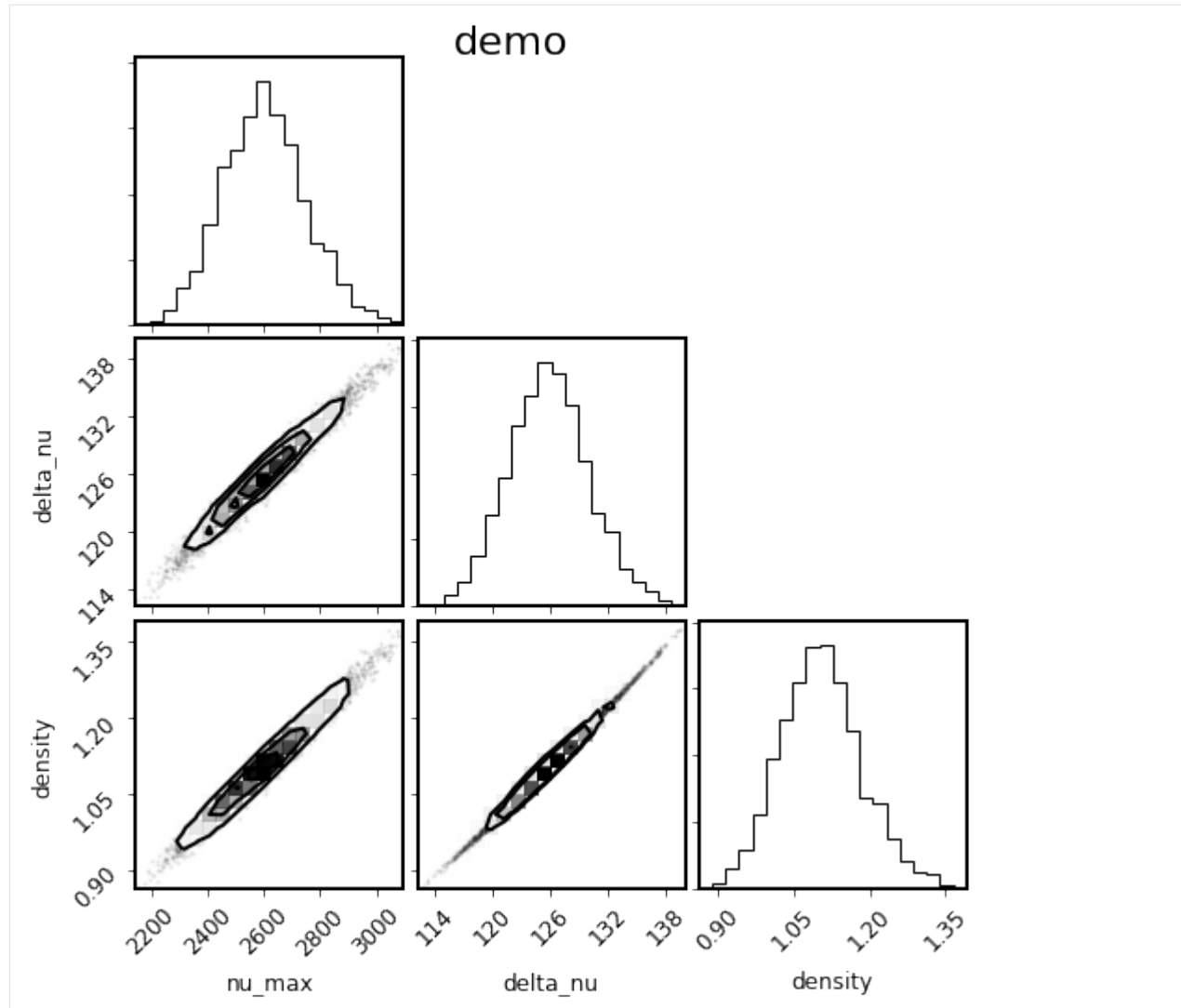
```
[24]: mod.corner_observed();
```



This looks good, because we generated the synthetic observations directly from the same stellar model grids that we used to fit. For real data, this is an important figure to look at to see if any of the observations appear to be inconsistent with the others, and to see if the model is a good fit to the observations.

Generically, you can also make a corner plot of arbitrary derived parameters as follows:

```
[25]: mod.corner_derived(['nu_max', 'delta_nu', 'density']);
```

Multiple star systems

One of the signature capabilities of **isochrones** is the ability to fit multiple star systems to observational data. This works by providing a `StarModel` with more detailed information about the observational data, and about how many stars you wish to fit. There are several layers of potential intricacy here, which we will walk through in stages.

8.1 Unresolved multiple systems

Often it is of interest to know what potential binary star configurations are consistent with observations of a star. For most stars the best available observational data is a combination of broadband magnitudes from various all-sky catalogs and parallax measurements from *Gaia*. Let's first generate synthetic observations of such a star, and then see what we can recover with a binary or triple star model, and also what inference of this system under a single star model would tell us.

Note here that for this simplest of multiple star scenarios—unresolved, physically associated, binary or triple-star systems—there are special `StarModel` objects available that have more highly optimized likelihood calculations, analogous to the `SingleStarModel` that is available for a simple single-star fit. `BinaryStarModel` and `TripleStarModel` are these special objects. In order to accommodate more complex scenarios, such as fitting resolved stellar companions, it is necessary to use the vanilla `StarModel` object.

First, we will initialize the isochrone interpolator. Note that we actually *require* the isochrone interpolator here, rather than the evolution track interpolator, because the model requires the primary and secondary components to have the same age, so that age must be a sampling parameter.

```
[1]: from isochrones import get_ichrone

mist = get_ichrone('mist')
```

Now, define the “true” system parameters and initialize the `StarModel` accordingly, with two model stars. Remember that even though we need to use an isochrone interpolator to fit the model, we have to use the evolution tracks to generate synthetic data; this here shows that you can actually do this by using the `.track` complementary attribute. Note also the use of the utility function `addmags` to combine the magnitudes of the two stars.

```
[2]: from isochrones import BinaryStarModel
    from isochrones.utils import addmags

    distance = 500 # pc
    AV = 0.2
    mass_A = 1.0
    mass_B = 0.5
    age = 9.6
    feh = 0.0

    # Synthetic 2MASS and Gaia magnitudes
    bands = ['J', 'H', 'K', 'BP', 'RP', 'G']
    props_A = mist.track.generate(mass_A, age, feh, distance=distance, AV=AV,
                                  bands=bands, return_dict=True, accurate=True)
    props_B = mist.track.generate(mass_B, age, feh, distance=distance, AV=AV,
                                  bands=bands, return_dict=True, accurate=True)

    unc = dict(J=0.02, H=0.02, K=0.02, BP=0.002, RP=0.002, G=0.001)
    mags_tot = {b: (addmags(props_A[b], props_B[b]), unc[b]) for b in bands}

    # Gaia parallax in mas for a system at 500 pc
    parallax = (2, 0.05)

    mod_binary = BinaryStarModel(mist, **mags_tot, parallax=parallax, name='demo_binary')
```

This model has the following parameters; `eep_0` and `eep_1` correspond to the primary and secondary components, respectively. All the other parameters are assumed to be the same between the two components; that is, they are assumed to be co-eval and co-located.

```
[3]: mod_binary.param_names
[3]: ('eep_0', 'eep_1', 'age', 'feh', 'distance', 'AV')
```

Let's also restrict the prior ranges for the parameters, to help with convergence.

```
[4]: mod_binary.set_bounds(eep=(1, 600), age=(8, 10))
```

Let's test out the posterior computation, and then run a fit to see if we can recover the true parameters.

```
[5]: pars = [350., 300., 9.7, 0.0, 300., 0.1]
    print(mod_binary.lnpost(pars))
    %timeit mod_binary.lnpost(pars)

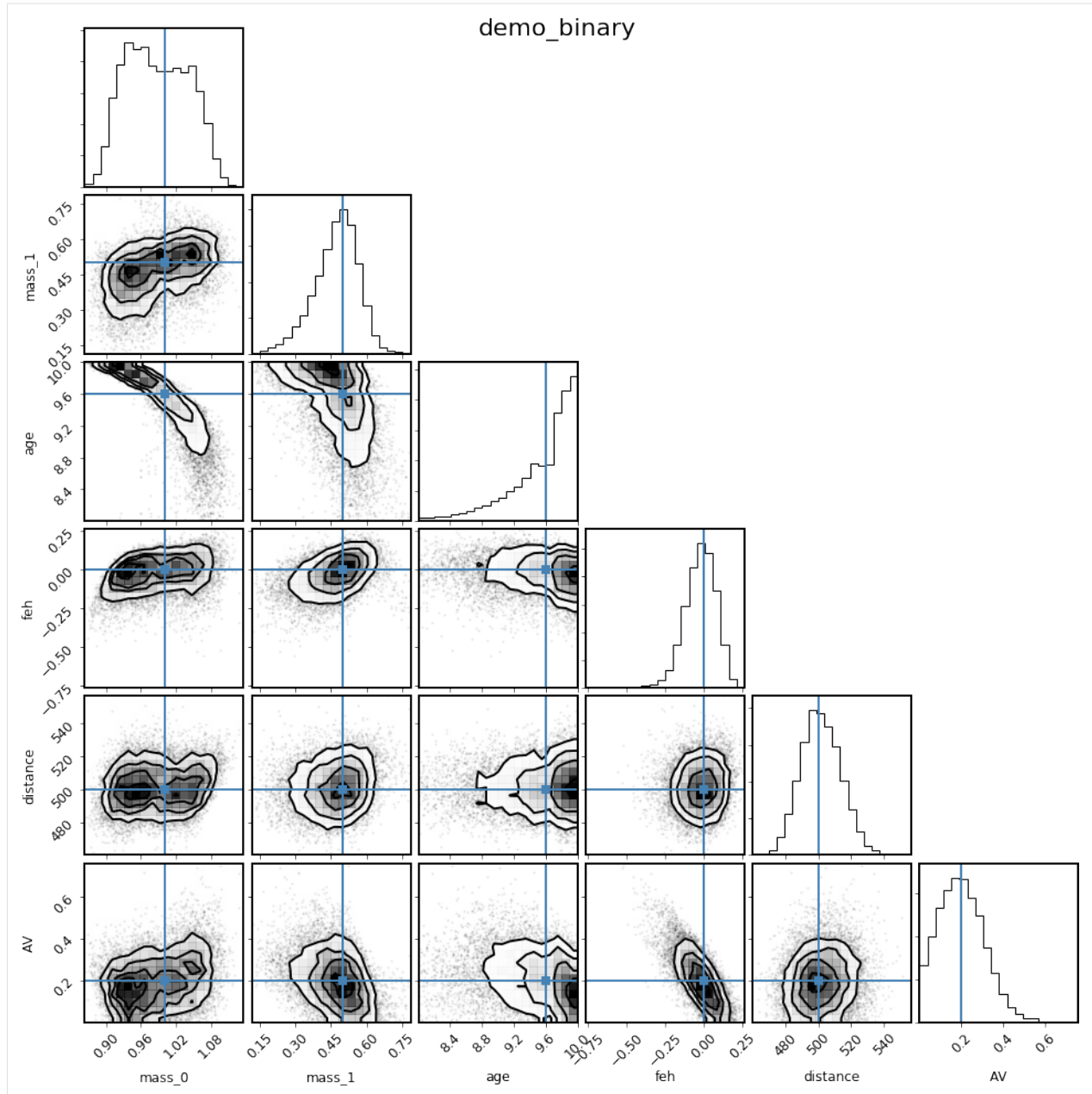
-645802.2025506602
1000 loops, best of 3: 719 µs per loop
```

For a binary fit, it is often desirable to run with more than the default number of live points; here we double from 1000 to 2000.

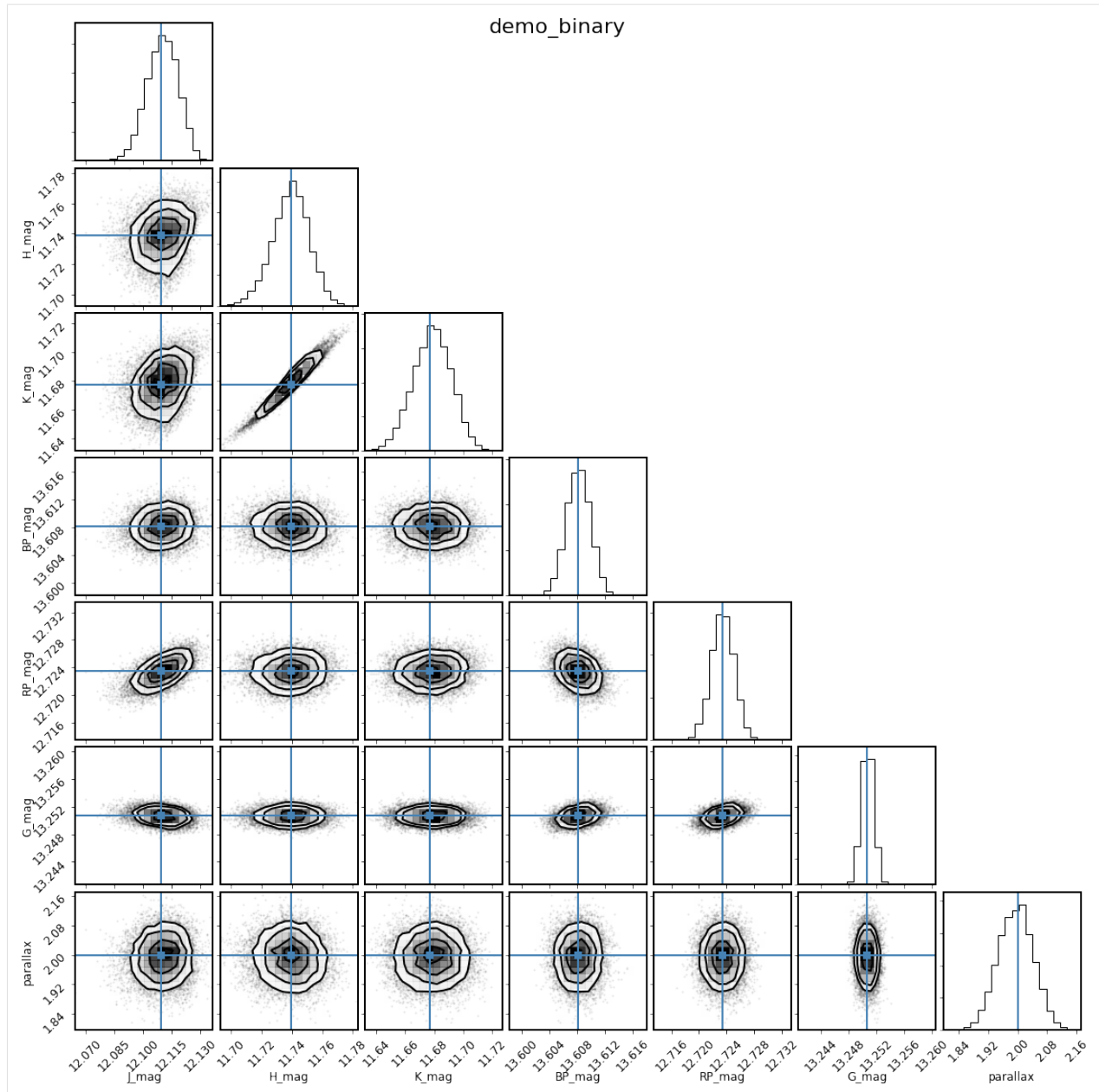
```
[6]: mod_binary.fit(n_live_points=2000) # takes about 14 minutes on my laptop

[7]: %matplotlib inline

columns = ['mass_0', 'mass_1', 'age', 'feh', 'distance', 'AV']
truths = [mass_A, mass_B, age, feh, distance, AV]
mod_binary.corner_derived(columns, truths=truths);
```



```
[8]: mod_binary.corner_observed();
```



Looks like this recovers the injected parameters pretty well, though not exactly. It looks like the flat-linear age prior (which weights the fit significantly to older ages) is biasing the masses somewhat low. Let's explore what happens if we change the prior and try again, imagining we have some other indication the $\log(\text{age})$ should be around 9.6.

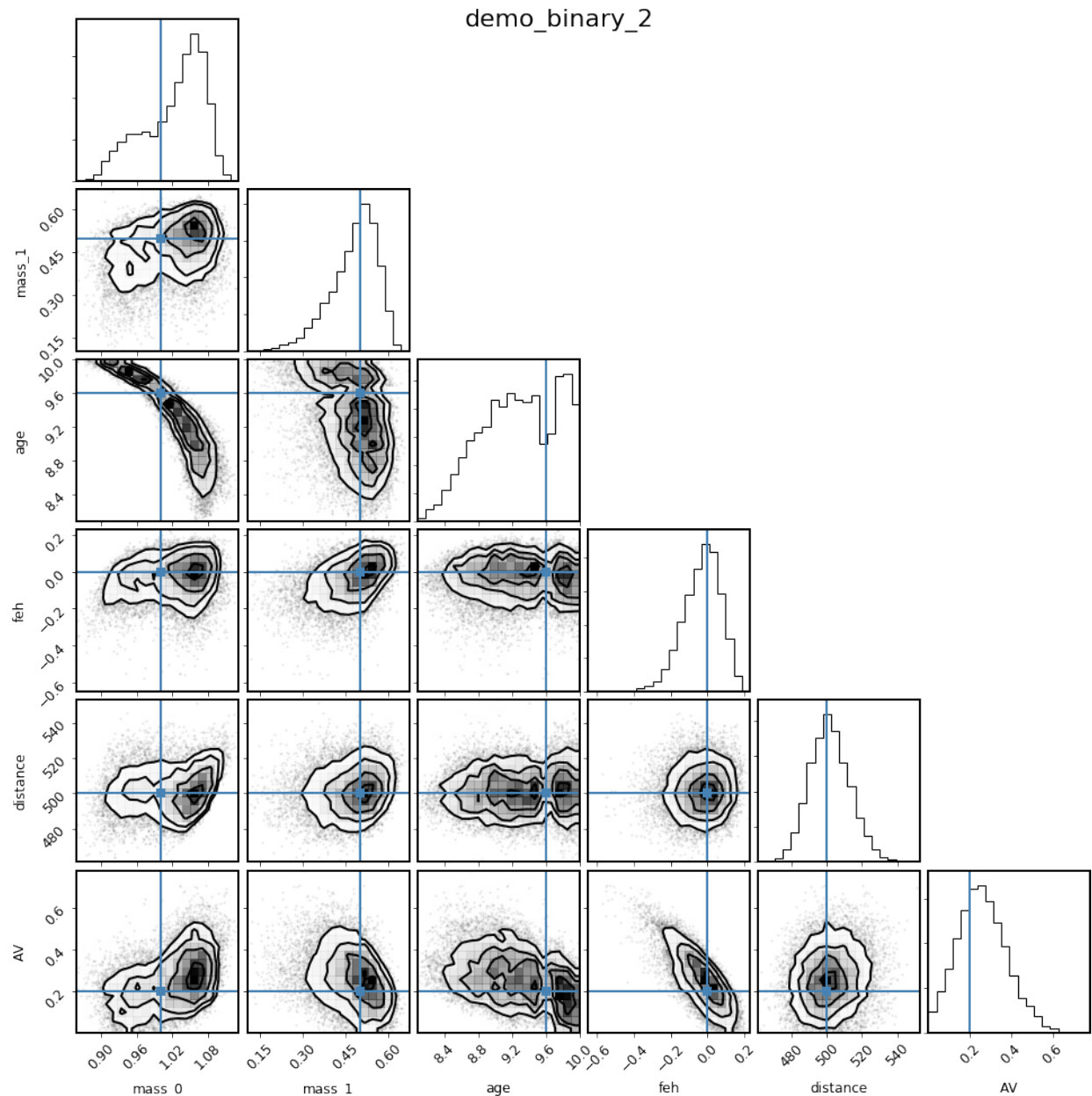
```
[9]: from isochrones.priors import GaussianPrior

mod_binary_2 = BinaryStarModel(mist, **mags_tot, parallax=parallax, name='demo_binary_
↳ 2')
mod_binary_2.set_bounds(eep=(1, 600))
mod_binary_2.set_prior(age=GaussianPrior(9.6, 1, bounds=(8,10)))
mod_binary_2.lnpost(pars)
```

```
[9]: -645802.7700077017
```

```
[10]: mod_binary_2.fit(n_live_points=2000)
```

```
[11]: mod_binary_2.corner_derived(columns, truths=truths);
```



Hmm, doesn't seem to be much different. Looks like this needs more exploration!

8.2 Resolved multiple system

Another useful capability of **isochrones** is the ability to fit binary (or higher-order multiple) systems that are resolved in high-resolution imaging but blended in catalog photometry. This is done by using the `StarModel` object directly (instead of the optimized models) and explicitly passing the observations.

As before, let's begin by using simulating data. Let's pretend that the same binary system from above is resolved in AO *K*-band imaging, but blended in 2MASS catalog data. Let's say this time that we also have spectroscopic constraints of the primary properties.

Inspecting this tree to make sure it accurately represents the desired model becomes more important if the model is more complicated, but this simple case is a good example to review. Each node named with a bandpass represents an observation, with some magnitude and uncertainty (at some separation and position angle—irrelevant for the unresolved case). The model nodes here are named 0_0 and 0_1, with the first index representing the system, and the second index the star number within that system. All stars in the same system share the same age, metallicity, distance, and extinction. In the computation of the likelihood, the apparent magnitude in each observed node is compared with a model-based magnitude that is computed from the *sum of the fluxes of all model nodes underneath that observed node in the tree*. In the unresolved case, this is trivial, but this structure becomes important when a binary is resolved. This model, because the two model stars share all attributes except mass, has the following parameters:

```
[12]: from isochrones import StarModel
      from isochrones.observation import ObservationTree, Observation, Source

      def build_obstree(name):
          obs = ObservationTree(name=name)
          for band in 'JHK':
              o = Observation('2MASS', band, 4) # Name, band, resolution (in arcsec)
              s = Source(addmags(props_A[band], props_B[band]), 0.02)
              o.add_source(s)
              obs.add_observation(o)

              o = Observation('AO', 'K', 0.1)
              s_A = Source(0., 0.02, separation=0, pa=0,
                           relative=True, is_reference=True)
              s_B = Source(props_B['K'] - props_A['K'], 0.02, separation=0.2, pa=100,
                           relative=True, is_reference=False)
              o.add_source(s_A)
              o.add_source(s_B)

              obs.add_observation(o)
          return obs

      obs = build_obstree('demo_resolved')
      mod_resolved = StarModel(mist, obs=obs,
                              parallax=parallax, Teff=(props_A['Teff'], 100),
                              logg=(props_A['logg'], 0.15), feh=(props_A['feh'], 0.1))
      mod_resolved.print_ascii()

      demo_resolved
      2MASS J=(12.11, 0.02) @(0.00, 0 [4.00])
      2MASS H=(11.74, 0.02) @(0.00, 0 [4.00])
      2MASS K=(11.68, 0.02) @(0.00, 0 [4.00])
      AO delta-K=(0.00, 0.02) @(0.00, 0 [0.10])
      0_0, Teff=(5834.782979719397, 100), logg=(4.435999146983706, 0.15),
      ➔feh=(-0.012519050601435218, 0.1), parallax=(2, 0.05)
      AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
      0_1, parallax=(2, 0.05)

[13]: pars = [300, 280, 9.6, 0.0, 400, 0.1]
      mod_resolved.lnpost(pars)

[13]: -8443.175970078633
```

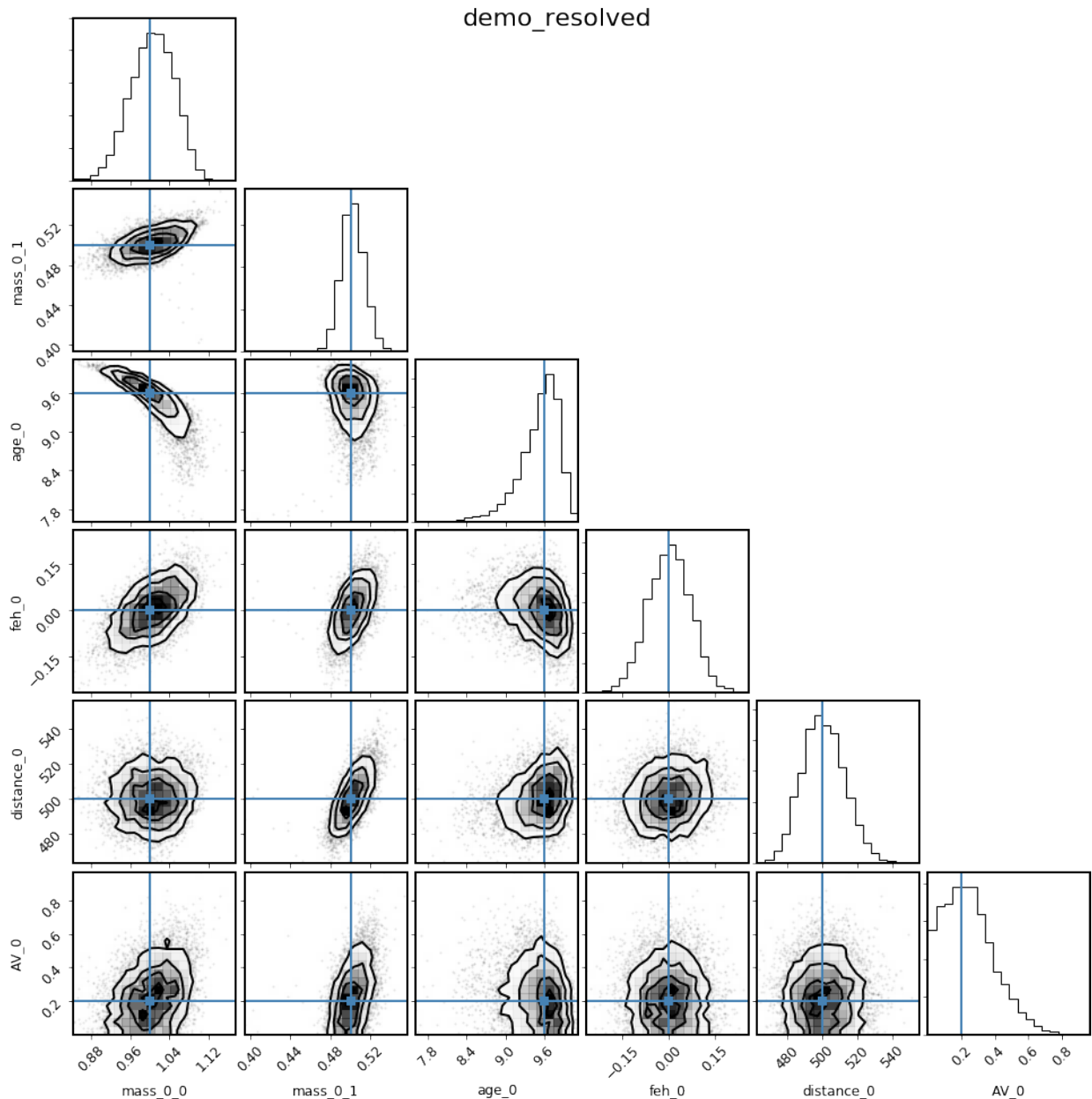


```
[14]: %timeit mod_resolved.lnpost(pars)
100 loops, best of 3: 1.23 ms per loop
```

```
[15]: mod_resolved.fit()
```

```
[16]: %matplotlib inline

columns = ['mass_0_0', 'mass_0_1', 'age_0', 'feh_0', 'distance_0', 'AV_0']
truths = [mass_A, mass_B, age, feh, distance, AV]
mod_resolved.corner(columns, truths=truths);
```



Nailed it! Looks like the spectroscopy was very helpful in getting the fit correct (age in particular).

8.3 Unassociated companions

The previous two examples model a binary star system in which the two components are co-located and co-eval; that is, they have the same age, metallicity, distance, and extinction.

One can imagine, however, wanting to model a scenario in which the two components are *not* physically associated, but rather just chance-aligned in the plane of the sky. In this case, you can set up the `StarModel` with just a small difference:

```
[17]: obs = build_obstree('demo_resolved_unassoc') # N.B., running this again, because the_
      ↪old "obs" was changed by the previous model
mod_resolved_unassoc = StarModel(mist, obs=obs,
                                parallax=parallax, Teff=(props_A['Teff'], 100),
                                logg=(props_A['logg'], 0.15), feh=(props_A['feh'], 0.1),
                                index=[0, 1])
mod_resolved_unassoc.print_ascii()

demo_resolved_unassoc
  2MASS J=(12.11, 0.02) @(0.00, 0 [4.00])
  2MASS H=(11.74, 0.02) @(0.00, 0 [4.00])
  2MASS K=(11.68, 0.02) @(0.00, 0 [4.00])
  AO delta-K=(0.00, 0.02) @(0.00, 0 [0.10])
    0_0, Teff=(5834.782979719397, 100), logg=(4.435999146983706, 0.15),_
    ↪feh=(-0.012519050601435218, 0.1), parallax=(2, 0.05)
  AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
    1_0
```

Note that this model now has ten parameters, since the two systems are now decoupled, so we will not run the fit for this example, but it is in principle possible. (Note that you would probably want to run this with MPI for this number of parameters.)

```
[18]: mod_resolved_unassoc.param_names
```

```
[18]: ['eep_0_0',
      'age_0',
      'feh_0',
      'distance_0',
      'AV_0',
      'eep_1_0',
      'age_1',
      'feh_1',
      'distance_1',
      'AV_1']
```

8.4 More complex models

You can define arbitrarily complex models, by explicitly defining the model nodes by hand, using the `N` and `index` keywords. Below are some examples.

This is a physically associated hierarchical triple, where the bright star from AO is an unresolved binary:

```
[19]: obs = build_obstree('triple1')
      StarModel(mist, obs=obs, N=[2, 1], index=[0, 0]).print_ascii()

triple1
  2MASS J=(12.11, 0.02) @(0.00, 0 [4.00])
```

(continues on next page)

(continued from previous page)

```

2MASS H=(11.74, 0.02) @(0.00, 0 [4.00])
2MASS K=(11.68, 0.02) @(0.00, 0 [4.00])
AO delta-K=(0.00, 0.02) @(0.00, 0 [0.10])
  0_0
  0_1
AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
  0_2

```

Here is a situation where the faint visual binary is an unrelated binary star:

```

[20]: obs = build_obstree('triple2')
      StarModel(mist, obs=obs, N=[1, 2], index=[0, 1]).print_ascii()

triple2
2MASS J=(12.11, 0.02) @(0.00, 0 [4.00])
2MASS H=(11.74, 0.02) @(0.00, 0 [4.00])
2MASS K=(11.68, 0.02) @(0.00, 0 [4.00])
AO delta-K=(0.00, 0.02) @(0.00, 0 [0.10])
  0_0
AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
  1_0
  1_1

```

Here, both AO stars are unresolved binaries:

```

[21]: obs = build_obstree('double_binary')
      StarModel(mist, obs=obs, N=2, index=[0, 1]).print_ascii()

double_binary
2MASS J=(12.11, 0.02) @(0.00, 0 [4.00])
2MASS H=(11.74, 0.02) @(0.00, 0 [4.00])
2MASS K=(11.68, 0.02) @(0.00, 0 [4.00])
AO delta-K=(0.00, 0.02) @(0.00, 0 [0.10])
  0_0
  0_1
AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
  1_0
  1_1

```

You can in principle create even more crazy models, but I don't recommend it...

Simulating stellar populations

Many astronomical investigations require simulating populations of stars, and **isochrones** contains some utilities to help enable this. Given population distributions of the quantities required to simulate individual stars, a `StarPopulation` object can be defined and used to generate sample populations following this distribution. Binary stars, [ubiquitous as they are](#), are necessarily built into this framework, so the parameters needed to simulate an individual stellar observation are the following:

$$M_A, M_B, T, [Fe/H], d, A_V$$

where M_A, M_B are the primary and (if present) secondary masses, T is age, $[Fe/H]$ is the metallicity, d is distance, and A_V is the V -band extinction, quantifying the effect of dust along the line of sight. Generating a population of such stars then requires sampling from distributions of each of the above quantities. A `StarPopulation` takes metallicity, distance, and extinction distributions as arguments, and samples from each of those distributions when generating a sample population.

Sampling primary/secondary masses and ages is a bit less straightforward. For M_A, M_B , **isochrones** parametrizes the distribution with a primary initial mass function (IMF), binary fraction f_B , and mass-ratio ($q = M_B/M_A$) distribution $p(q) \propto q^\gamma$. The age distribution of stars in a population is often described as a “star-formation history” (SFH)—sampling a population with a given SFH is the same as treating the SFH as the probability distribution function of stellar age, sampling ages from this distribution, and then truncating any stars that have reached the end of their evolution. Practically, this truncation happens by rejection sampling: evaluating the `ModelGridInterpolator` at each sampled set of parameters, and rejecting samples for which the interpolator returns `np.nan` values for the observed stellar properties (which will happen when trying to interpolate out-of-bounds, which happens when a star is requested beyond the end of its lifetime).

9.1 StarPopulation object

Here is an example of `StarPopulation` usage:

```
[1]: from scipy.stats import uniform, norm
    from isochrones import get_ichrone
    from isochrones.priors import GaussianPrior, SalpeterPrior, DistancePrior, FlatPrior
```

(continues on next page)

(continued from previous page)

```

from isochrones.populations import StarFormationHistory, StarPopulation

# Initialize interpolator
mist = get_ichrone('mist')

# Initialize distributions

# Ingredients required to generate primary & secondary masses
imf = SalpeterPrior(bounds=(1, 10)) # minimum 1 Msun
fB = 0.4
gamma = 0.3

# SFH distribution takes a scipy stats distribution, of age in Gyr
sfh = StarFormationHistory(dist=uniform(0, 10))

# The following are all isochrones.priors.Prior objects,
# or anything with a .sample(N) method
feh = GaussianPrior(-0.2, 0.2)
distance = DistancePrior(max_distance=3000)
AV = FlatPrior(bounds=[0, 1])

pop = StarPopulation(mist, imf=imf, fB=fB, gamma=gamma, sfh=sfh, feh=feh,
    ↪ distance=distance, AV=AV)

```

Once the object is created, it can be used to generate a population of stars.

```

[2]: df = pop.generate(1000)
df.head()

```

```

[2]:   mass_0    logg_0  delta_nu_0  initial_mass_0  phase_0    eep_0  \
0  1.553332  4.380359  102.670924      1.553406      0.0  299.894473
1  1.549665  4.211669   78.515862      1.549955      0.0  340.291660
2  1.127802  4.009138   66.783488      1.128399      0.0  447.067891
3  1.046129  4.299633  109.308356      1.046413      0.0  384.695516
4  1.267605  3.757970   42.600593      1.268362      2.0  460.286164

   radius_0    Mbol_0  logTeff_0    feh_0  ...    W1_mag    A_W1  \
0  1.332984  2.451403   3.927937 -0.327345  ...  14.208519  0.014534
1  1.617098  2.454235   3.885739 -0.111990  ...  13.706165  0.048358
2  1.740697  3.206514   3.794381 -0.366979  ...  14.346362  0.010542
3  1.199744  3.803016   3.815517 -0.668881  ...  14.342954  0.028781
4  2.463683  2.543589   3.785193 -0.307018  ...  13.221399  0.041585

   W2_mag    A_W2    W3_mag    A_W3  TESS_mag    A_TESS  Kepler_mag  \
0  14.203228  0.008647  14.194746  0.002359  14.477501  0.161784  14.609700
1  13.686343  0.028772  13.662330  0.007847  14.462563  0.529039  14.812933
2  14.338646  0.006274  14.318684  0.001710  15.193251  0.114144  15.567852
3  14.327035  0.017125  14.301296  0.004667  15.269802  0.311150  15.679683
4  13.205129  0.024749  13.170042  0.006745  14.403408  0.445389  14.910002

   A_Kepler
0  0.225833
1  0.730147
2  0.155572
3  0.424941
4  0.604260

```

(continues on next page)

(continued from previous page)

```
[5 rows x 110 columns]
```

Not that is operation is not nearly as fast as directly interpolating an isochrone or evolution track grid (since generating properites given mass, age, and metallicity necessarily involves solving for EEP first):

```
[3]: %timeit pop.generate(1000)

1.24 s ± 152 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Also, this can be made **much** faster if you loosen the requirement on getting *exactly* a particularly desired number of stars (as part of the generating algorithm involves replacing stars that come out as nan until no nans are left):

```
[4]: print(len(pop.generate(1000, exact_N=False)))

%timeit pop.generate(1000, exact_N=False)

255
64.9 ms ± 381 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The full column list of this table of simulated stars is the following:

```
[5]: ', '.join(df.columns)

[5]: 'mass_0, logg_0, delta_nu_0, initial_mass_0, phase_0, eep_0, radius_0, Mbol_0,
↳ logTeff_0, feh_0, density_0, nu_max_0, logL_0, Teff_0, interpolated_0, star_age_0,
↳ age_0, dt_deep_0, J_mag_0, H_mag_0, K_mag_0, G_mag_0, BP_mag_0, RP_mag_0, W1_mag_0,
↳ W2_mag_0, W3_mag_0, TESS_mag_0, Kepler_mag_0, distance_0, AV_0, initial_feh_0,
↳ requested_age_0, A_J_0, A_H_0, A_K_0, A_G_0, A_BP_0, A_RP_0, A_W1_0, A_W2_0, A_W3_0,
↳ A_TESS_0, A_Kepler_0, mass_1, logg_1, delta_nu_1, initial_mass_1, phase_1, eep_1,
↳ radius_1, Mbol_1, logTeff_1, feh_1, density_1, nu_max_1, logL_1, Teff_1,
↳ interpolated_1, star_age_1, age_1, dt_deep_1, J_mag_1, H_mag_1, K_mag_1, G_mag_1,
↳ BP_mag_1, RP_mag_1, W1_mag_1, W2_mag_1, W3_mag_1, TESS_mag_1, Kepler_mag_1,
↳ distance_1, AV_1, initial_feh_1, requested_age_1, A_J_1, A_H_1, A_K_1, A_G_1, A_BP_
↳ 1, A_RP_1, A_W1_1, A_W2_1, A_W3_1, A_TESS_1, A_Kepler_1, J_mag, A_J, H_mag, A_H, K_
↳ mag, A_K, G_mag, A_G, BP_mag, A_BP, RP_mag, A_RP, W1_mag, A_W1, W2_mag, A_W2, W3_
↳ mag, A_W3, TESS_mag, A_TESS, Kepler_mag, A_Kepler'
```

All quantities with a tag `_0` refer to the primary star; all quantities with `_1` refer to the secondary. Columns ending in just `_mag` represent the *combined* magnitude of both primary and secondary component. Let's look the *Gaia* color-magnitude diagram for this simulated population. Note also the `A_[]` columns, which give the specific extinction per band for each system (and for the individual components of the binary).

```
[6]: import holoviews as hv
hv.extension('bokeh')
import hvplot.pandas

def hr_plot(df):
    df['BpRp'] = df.BP_mag - df.RP_mag
    hr = df.hvplot.scatter('BpRp', 'G_mag',
                           hover_cols=['mass_0', 'mass_1', 'age_0', 'AV_0'],
                           color='feh_0')
    return hr.options(height=400, width=500, invert_yaxis=True)

hr_plot(df)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[6]: :Scatter [BpRp] (G_mag, feh_0, mass_0, mass_1, age_0, AV_0)
```

There is also a simple utility function that can “deredden” a generated population dataframe (e.g., recover the true intrinsic magnitudes of each star in the absence of dust), by subtracting off the A_x extinction values from the magnitudes, and setting all extinctions to zero. Let’s use this to deredden the above hr diagram:

```
[7]: from isochrones.populations import deredden

dereddened = deredden(df)

hr_plot(df).options(size=3, alpha=0.2, color='red') * hr_plot(dereddened).
↳ options(alpha=0.2, color='black', size=3)
```

```
[7]: :Overlay
      .Scatter.I :Scatter [BpRp] (G_mag, feh_0, mass_0, mass_1, age_0, AV_0)
      .Scatter.II :Scatter [BpRp] (G_mag, feh_0, mass_0, mass_1, age_0, AV_0)
```

See how the dust (reddened points) moves each star down (fainter) and to the right (redder).

9.2 ModelGridInterpolator.generate_binary

The above-used `StarPopulation.generate` method is a wrapper around the `.generate_binary` method of a `ModelGridInterpolator`, which can also be used directly, if you wish to simulate observations of binary stars with specific properties:

```
[8]: mass_A = 1.0
      mass_B = [0.8, 0.6, 0.4, 0.2]
      age, feh, distance, AV = (9.6, 0.02, 100, 0.1)

      mist.generate_binary(mass_A, mass_B, age, feh, distance=distance, AV=AV)[['G_mag',
      ↳ 'BP_mag', 'RP_mag']]

[8]:      G_mag      BP_mag      RP_mag
0  9.459204  9.822055  8.928917
1  9.669059  10.018673  9.150093
2  9.709775  10.046349  9.204878
3  9.718787  10.050889  9.219106
```