# 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 properties 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 DFieldInterpolator 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 34  

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
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:

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

Then you can install pymultinest with

```bash
pip install pymultinest
```

(And run `nose tests isochrones` again, for good measure, to confirm that **MultiNest** works.)
CHAPTER 2

Quick Start

2.1 Access stellar model grid data

```python
from isochrones.mist import MISTIsochroneGrid
grid = MISTIsochroneGrid()
print(len(grid.df))
grid.df.head()  # Just the first few rows
```

1494453

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>eep</th>
<th>age</th>
<th>feh</th>
<th>mass</th>
<th>initial_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>5.0</td>
<td>-3.978406</td>
<td>0.100000</td>
<td>0.100000</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>5.0</td>
<td>-3.978406</td>
<td>0.102885</td>
<td>0.102885</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>5.0</td>
<td>-3.978406</td>
<td>0.107147</td>
<td>0.107147</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>5.0</td>
<td>-3.978406</td>
<td>0.111379</td>
<td>0.111379</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>5.0</td>
<td>-3.978406</td>
<td>0.115581</td>
<td>0.115581</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>radius</th>
<th>density</th>
<th>logTeff</th>
<th>Teff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.106082</td>
<td>0.104184</td>
<td>3.617011</td>
<td>4140.105252</td>
</tr>
<tr>
<td>1.122675</td>
<td>0.102507</td>
<td>3.618039</td>
<td>4149.909661</td>
</tr>
<tr>
<td>1.147702</td>
<td>0.099921</td>
<td>3.619556</td>
<td>4164.436984</td>
</tr>
<tr>
<td>1.173015</td>
<td>0.097287</td>
<td>3.621062</td>
<td>4178.903372</td>
</tr>
<tr>
<td>1.198615</td>
<td>0.094627</td>
<td>3.622555</td>
<td>4193.289262</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>logg</th>
<th>logL</th>
<th>Mbol</th>
<th>delta_nu</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.350571</td>
<td>-0.489734</td>
<td>5.964335</td>
<td>37.987066</td>
</tr>
<tr>
<td>3.347798</td>
<td>-0.472691</td>
<td>5.921728</td>
<td>37.739176</td>
</tr>
<tr>
<td>3.343658</td>
<td>-0.447471</td>
<td>5.858678</td>
<td>37.345115</td>
</tr>
<tr>
<td>3.339612</td>
<td>-0.422498</td>
<td>5.796244</td>
<td>36.923615</td>
</tr>
<tr>
<td>3.335660</td>
<td>-0.397776</td>
<td>5.734440</td>
<td>36.473151</td>
</tr>
</tbody>
</table>
```

(continues on next page)
isochrones Documentation, Release 2.0.1

(continued from previous page)

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>nu_max</th>
<th>phase</th>
<th>dm_deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>299.346079</td>
<td>-1.0</td>
<td>0.002885</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>298.570836</td>
<td>-1.0</td>
<td>0.003573</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>297.180748</td>
<td>-1.0</td>
<td>0.004247</td>
<td></td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>295.526946</td>
<td>-1.0</td>
<td>0.004217</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>293.589960</td>
<td>-1.0</td>
<td>0.004189</td>
<td></td>
</tr>
</tbody>
</table>

[2]:

```python
from isochrones.mist import MISTEvolutionTrackGrid

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

3619652

[2]:

```python
initial_feh initial_mass EEP
-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
```

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

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

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

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

Chapter 2. Quick Start
2.2 Interpolate stellar properties

```python
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'])
```

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

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

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

2.3 Generate synthetic properties of stars

```python
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
```

```
{'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': -0.1909500738484541,
 'Teff': 6100.263434973235,
 'logL': 0.2310549698154745,
 'delta_nu': 114.32933695055772,
 'interpolated': 0.0,
 'star_age': 5302578707.515498,
 'age': 9.722480201790624,
 'dt_deep': 0.003656739980003118,
 'J': 3.202578707.515498,
 'K': 2.91756110497181,
 'W1': 2.890399473719951,
 'G': 4.08584759912897,
 'BP': 4.349405878788243,
 'RP': 3.658731633956084,
 'W2': 2.8807983122840044,
 'W3': 2.88550073210391,
 'TESS': 3.653543903981804,
 'Kepler': 4.004222279916473)
```
```python
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
 intermediary result is being cached.
1 loop, best of 3: 9.04 ms per loop

df = df.dropna()
print(len(df))  # about half of the original simulated stars are nans
df.head()

503

7]

7]:

mass density Mbol phase ... H K \\n0 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 \\n0 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
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 18.528345 18.837217 
8 15.985842 16.451727 
9 17.857626 18.111831

[5 rows x 29 columns]
```

8: ```python
import holoviews as hv
hv.extension('bokeh')
```
(continues on next 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)

2.4. Fit physical parameters of a star to observed data

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-

%matplotlib inline

mod.corner_physical();
Check out the numerical sampling results:

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

```
<table>
<thead>
<tr>
<th></th>
<th>eep</th>
<th>age</th>
<th>feh</th>
<th>distance</th>
<th>AV</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>337.710149</td>
<td>9.509309</td>
<td>-0.020312</td>
<td>101.801691</td>
<td>0.136494</td>
</tr>
<tr>
<td>std</td>
<td>9.624071</td>
<td>0.149170</td>
<td>0.078899</td>
<td>3.989609</td>
<td>0.069615</td>
</tr>
<tr>
<td>min</td>
<td>304.868138</td>
<td>9.043279</td>
<td>-0.300996</td>
<td>88.634174</td>
<td>0.000291</td>
</tr>
<tr>
<td>25%</td>
<td>330.856377</td>
<td>9.403279</td>
<td>-0.070581</td>
<td>99.008511</td>
<td>0.085589</td>
</tr>
<tr>
<td>50%</td>
<td>339.214747</td>
<td>9.526285</td>
<td>-0.022509</td>
<td>101.582447</td>
<td>0.129668</td>
</tr>
<tr>
<td>75%</td>
<td>345.473042</td>
<td>9.630834</td>
<td>0.033488</td>
<td>104.310959</td>
<td>0.183091</td>
</tr>
<tr>
<td>max</td>
<td>364.435350</td>
<td>9.802944</td>
<td>0.243018</td>
<td>118.737714</td>
<td>0.466537</td>
</tr>
</tbody>
</table>
```
And the derived parameters at those samples:

```
mod.derived_samples.describe()
```

<table>
<thead>
<tr>
<th>eep</th>
<th>age</th>
<th>feh</th>
<th>mass</th>
<th>initial_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>337.710149</td>
<td>9.509309</td>
<td>-0.018942</td>
<td>0.958078</td>
</tr>
<tr>
<td>std</td>
<td>0.031022</td>
<td>0.137653</td>
<td>0.005747</td>
<td>0.020924</td>
</tr>
<tr>
<td>min</td>
<td>0.099485</td>
<td>1.643165</td>
<td>3.750771</td>
<td>5634.112958</td>
</tr>
<tr>
<td>25%</td>
<td>4.913786</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
</tr>
<tr>
<td>50%</td>
<td>4.985717</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
</tr>
<tr>
<td>75%</td>
<td>5.061043</td>
<td>1.832595</td>
<td>3.758868</td>
<td>5740.003739</td>
</tr>
<tr>
<td>max</td>
<td>5.307000</td>
<td>2.217147</td>
<td>3.782610</td>
<td>6062.846454</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>radius</th>
<th>density</th>
<th>logTeff</th>
<th>Teff</th>
<th>logg</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.921082</td>
<td>1.738516</td>
<td>3.755075</td>
<td>5690.649562</td>
</tr>
<tr>
<td>std</td>
<td>0.082304</td>
<td>1.209516</td>
<td>3.740466</td>
<td>5501.680997</td>
</tr>
<tr>
<td>min</td>
<td>0.899485</td>
<td>1.643165</td>
<td>3.750771</td>
<td>5634.112958</td>
</tr>
<tr>
<td>25%</td>
<td>4.913786</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
</tr>
<tr>
<td>50%</td>
<td>4.985717</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
</tr>
<tr>
<td>75%</td>
<td>5.061043</td>
<td>1.832595</td>
<td>3.758868</td>
<td>5740.003739</td>
</tr>
<tr>
<td>max</td>
<td>5.307000</td>
<td>2.217147</td>
<td>3.782610</td>
<td>6062.846454</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mbol</th>
<th>delta_nu</th>
<th>nu_max</th>
<th>phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>...</td>
<td>4.982881</td>
<td>3538.322417</td>
</tr>
<tr>
<td>std</td>
<td>...</td>
<td>0.005747</td>
<td>0.020924</td>
</tr>
<tr>
<td>min</td>
<td>...</td>
<td>4.568552</td>
<td>177.717410</td>
</tr>
<tr>
<td>25%</td>
<td>...</td>
<td>4.913786</td>
<td>3412.744754</td>
</tr>
<tr>
<td>50%</td>
<td>...</td>
<td>4.985717</td>
<td>3664.317241</td>
</tr>
<tr>
<td>75%</td>
<td>...</td>
<td>5.061043</td>
<td>3664.317241</td>
</tr>
<tr>
<td>max</td>
<td>...</td>
<td>5.303972</td>
<td>4130.359858</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dm_deep</th>
<th>BP_mag</th>
<th>RP_mag</th>
<th>parallax</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.008306</td>
<td>9.539072</td>
<td>9.837970</td>
<td>101.801691</td>
</tr>
<tr>
<td>std</td>
<td>0.009287</td>
<td>0.009087</td>
<td>0.382167</td>
<td>3.989609</td>
</tr>
<tr>
<td>min</td>
<td>0.002593</td>
<td>10.42067</td>
<td>9.505070</td>
<td>8.421924</td>
</tr>
<tr>
<td>25%</td>
<td>0.007966</td>
<td>10.43741</td>
<td>9.533035</td>
<td>9.586720</td>
</tr>
<tr>
<td>50%</td>
<td>0.008247</td>
<td>10.42013</td>
<td>9.539162</td>
<td>9.844220</td>
</tr>
<tr>
<td>75%</td>
<td>0.008619</td>
<td>10.426205</td>
<td>9.545132</td>
<td>10.100142</td>
</tr>
<tr>
<td>max</td>
<td>0.011076</td>
<td>10.453965</td>
<td>9.574724</td>
<td>11.282330</td>
</tr>
</tbody>
</table>

AV

<table>
<thead>
<tr>
<th>count</th>
<th>4643.000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0.136494</td>
</tr>
</tbody>
</table>
Eyeball your posterior predictive with:

```python
[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()
```

```
   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)
```
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

<table>
<thead>
<tr>
<th>lnprob</th>
<th>eep_0</th>
<th>feh_0</th>
<th>mass_0</th>
<th>...</th>
<th>Mbol_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-54.808864</td>
<td>359.086356</td>
<td>-0.104045</td>
<td>0.902009</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>-54.414634</td>
<td>398.209362</td>
<td>-0.372728</td>
<td>0.826538</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>-54.189376</td>
<td>389.035743</td>
<td>-0.278190</td>
<td>0.869004</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>-54.100849</td>
<td>397.668498</td>
<td>-0.176003</td>
<td>0.936127</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>-53.934142</td>
<td>370.982797</td>
<td>-0.260589</td>
<td>0.914513</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delta_nu_1</th>
<th>nu_max_1</th>
<th>phase_1</th>
<th>dm_deep_1</th>
<th>BP_mag_1</th>
<th>RP_mag_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>615.092182</td>
<td>16669.820345</td>
<td>0.0</td>
<td>0.006549</td>
<td>17.795078</td>
</tr>
<tr>
<td>1</td>
<td>463.937102</td>
<td>12708.865085</td>
<td>0.0</td>
<td>0.001280</td>
<td>16.173813</td>
</tr>
<tr>
<td>2</td>
<td>414.876808</td>
<td>11597.77825</td>
<td>0.0</td>
<td>0.002549</td>
<td>15.919824</td>
</tr>
<tr>
<td>3</td>
<td>486.986772</td>
<td>13422.770521</td>
<td>0.0</td>
<td>0.006254</td>
<td>17.143644</td>
</tr>
<tr>
<td>4</td>
<td>562.841780</td>
<td>15244.528561</td>
<td>0.0</td>
<td>0.007484</td>
<td>17.169701</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BP_mag</th>
<th>RP_mag</th>
<th>parallax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.463678</td>
<td>9.547117</td>
</tr>
<tr>
<td>1</td>
<td>10.414967</td>
<td>9.566663</td>
</tr>
<tr>
<td>2</td>
<td>10.391647</td>
<td>9.518239</td>
</tr>
<tr>
<td>3</td>
<td>10.432474</td>
<td>9.553840</td>
</tr>
<tr>
<td>4</td>
<td>10.439209</td>
<td>9.511726</td>
</tr>
</tbody>
</table>

[5 rows x 44 columns]
2.5. Fit a binary star model
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.

```python
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
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>sum</th>
<th>product</th>
<th>power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>32</td>
</tr>
</tbody>
</table>

(continues on next page)
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.

```python
from isochrones.interp import DFInterpolator

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

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

Individual columns may also be accessed by name:

```python
interp([2.2, 4.6], ['product'])
```

```
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:

```python
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:

```python
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:

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>sum</th>
<th>product</th>
<th>power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```python
[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:

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

```python
[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, $\alpha$-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.
```python
from isochrones.mist import MISTEvolutionTrackGrid, MISTIsochroneGrid

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

<table>
<thead>
<tr>
<th>initial_feh</th>
<th>initial_mass</th>
<th>EEP</th>
<th>nu_max</th>
<th>logg</th>
<th>eep</th>
<th>initial_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
<td>0.1</td>
<td>1.0</td>
<td>143.524548</td>
<td>3.032277</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>145.419039</td>
<td>3.038935</td>
<td>2.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>147.409881</td>
<td>3.044805</td>
<td>3.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>149.499346</td>
<td>3.050886</td>
<td>4.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>151.703570</td>
<td>3.057203</td>
<td>5.0</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>radius</th>
<th>logTeff</th>
<th>mass</th>
<th>density</th>
<th>Mbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
<td>0.1</td>
<td>1.593804</td>
<td>3.620834</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>1.583455</td>
<td>3.620769</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>1.572790</td>
<td>3.620702</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>1.561817</td>
<td>3.620631</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>1.550499</td>
<td>3.620558</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phase</th>
<th>feh</th>
<th>Teff</th>
<th>logL</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
<td>0.1</td>
<td>-1.0</td>
<td>-3.978406</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>-1.0</td>
<td>-3.978406</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>-1.0</td>
<td>-3.978406</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>-1.0</td>
<td>-3.978406</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>-1.0</td>
<td>-3.978406</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delta_nu</th>
<th>interpolated</th>
<th>star_age</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
<td>0.1</td>
<td>False</td>
<td>13343.289397</td>
</tr>
<tr>
<td>2</td>
<td>0.1</td>
<td>False</td>
<td>14171.978264</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
<td>False</td>
<td>15048.910447</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>False</td>
<td>15975.827275</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>False</td>
<td>16962.744747</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dt_deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

```python
iso_grid = MISTIsochroneGrid()
iso_grid.df.head()  # just show first few rows
```

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>mass</th>
<th>initial_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>35</td>
<td>-3.978406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36</td>
<td>36</td>
<td>-3.978406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37</td>
<td>37</td>
<td>-3.978406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>38</td>
<td>-3.978406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>39</td>
<td>39</td>
<td>-3.978406</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>radius</th>
<th>density</th>
<th>logTeff</th>
<th>Teff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
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:

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

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>initial_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>-4.0</td>
<td>35</td>
<td>5.0</td>
</tr>
<tr>
<td>36</td>
<td>-4.0</td>
<td>36</td>
<td>5.0</td>
</tr>
<tr>
<td>37</td>
<td>-4.0</td>
<td>37</td>
<td>5.0</td>
</tr>
<tr>
<td>38</td>
<td>-4.0</td>
<td>38</td>
<td>5.0</td>
</tr>
<tr>
<td>39</td>
<td>-4.0</td>
<td>39</td>
<td>5.0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>star_mass</th>
<th>star_mdot</th>
<th>he_core_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.100000</td>
<td>-1.455094e-13</td>
<td>0.0</td>
</tr>
<tr>
<td>0.102885</td>
<td>-1.562027e-13</td>
<td>0.0</td>
</tr>
<tr>
<td>0.107147</td>
<td>-1.707298e-13</td>
<td>0.0</td>
</tr>
<tr>
<td>0.111379</td>
<td>-1.836256e-13</td>
<td>0.0</td>
</tr>
<tr>
<td>0.115581</td>
<td>-1.94639e-13</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c_core_mass</th>
<th>o_core_mass</th>
<th>log_L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>-0.489734</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>-0.472691</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>-0.447471</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>-0.422498</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>-0.397776</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>log_L_div_Ledd</th>
<th>nu_max</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>299.346079</td>
</tr>
<tr>
<td>...</td>
<td>298.570836</td>
</tr>
<tr>
<td>...</td>
<td>297.180748</td>
</tr>
</tbody>
</table>

(continues on next 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_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', 'surf_z', 'surf_h1', 'surf_he3', 'surf_he4',
           'surf_li7', 'surf_be9', 'surf_b11', 'surf_c12',
           'surf_c13', 'surf_n14', 'surf_o16', 'surf_f19',
           'surf_ne20', 'surf_na23', 'surf_mg24', 'surf_si28',
           'surf_s32', 'surf_ca40', 'surf_ti48', 'surf_fe56',
           'log_center_T', 'log_center_Rho', 'center_degeneracy', 'center_omega',
           'center_omega', 'mass_conv_core', 'center_h1', 'center_h4',
           '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)
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+03, 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])
```

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
```

```
EEP  
eeep age  feh  mass  initial_mass  radius  density  
193  3.460248  2885.680821  5.235913  12.245322  70.115891
194  3.462649  2901.679870  5.227759  12.170556  67.622476
195  3.465890  2923.411541  5.216743  12.069637  64.282466
196  3.469256  2946.160133  5.205249  11.964717  60.858291
197  3.472471  2968.048014  5.194272  11.864529  57.660112

EEP  
eep age  feh  mass  initial_mass  radius  density  
193  3.460248  2885.680821  5.235913  -3.002129  1045.120425
194  3.462649  2901.679870  5.227759  -2.972222  1025.261668
195  3.465890  2923.411541  5.216743  -2.931855  998.429682
196  3.469256  2946.160133  5.205249  -2.889887  970.463953
197  3.472471  2968.048014  5.194272  -2.849811  943.753111
```

(continues on next page)
4.3 Example visualization

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

```python
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

```
[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)
```

Chapter 4. Stellar model grids
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 provides 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.

```python
from isochrones.mist.bc import MISTBolometricCorrectionGrid
bc_grid = MISTBolometricCorrectionGrid(['J', 'H', 'K', 'G', 'BP', 'RP', 'g', 'r', 'i', 'z'])
bc_grid.df.head()
```

| Teff | logg | [Fe/H] | Av  | g   | r   | i   | J   | H   | K   | G   | BP  | RP  |
|------|------|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 2500.0 | -4.0 | -4.0   | 0.05 | -6.6534742 | -3.332877 | -1.617626 | 1.845781 | 2.927064 | 3.436304 | -2.181986 | -4.652544 | -0.881255 |
| 2500.0 | -4.0 | -4.0   | 0.05 | -6.590469 | -3.375570 | -1.650338 | 1.831466 | 2.917990 | 3.430463 | -2.211637 | -4.697700 | -0.909057 |
| 2500.0 | -4.0 | -4.0   | 0.05 | -6.646182 | -3.418258 | -1.683043 | 1.817153 | 2.908916 | 3.424623 | -2.241240 | -4.742838 | -0.936829 |
| 2500.0 | -4.0 | -4.0   | 0.05 | -6.701881 | -3.460939 | -1.715740 | 1.802841 | 2.899842 | 3.418782 | -2.270797 | -4.787959 | -0.964571 |
| 2500.0 | -4.0 | -4.0   | 0.05 | -6.757566 | -3.503615 | -1.748429 | 1.788530 | 2.890769 | 3.412942 | -2.300306 | -4.833062 | -0.992285 |
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.
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.

```python
[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.

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

[2]: (6303.059322477636,
    4.50738764316164,
    -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:

```python
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'])
```

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:

```python
mist_track.mass(*pars)
```

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

```python
mist.isochrone(9.53, 0.1).head()  # just show first few rows
```

(continues on next page)
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  \
```

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. Note that this only works for EvolutionTrackInterpolator, not IsochroneInterpolator.

```
[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  4.598580  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  5.970747  0.0 ...  4.154396  4.088644  5.988091  \
1  2.066995  5.374246  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  \
```
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
```

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)
```

(continues on next page)
Just for curiosity, let’s look at the difference in the predictions:

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

((df1 - df0) / df0).mean()
```

```
          nu_max       logg      eep     initial_mass      radius   logTeff      mass      density     Mbol     phase       feh     Teff     logL      delta_nu     interpolated  star_age       age     dt_deep        J       H       K       G      BP      RP      W1     W2     W3     TESS     Kepler
 0   -0.002617  -0.000240  0.001760  0.00000000  0.001243  0.000032 -0.000002 -0.003716 -0.000759  NaN   -0.057173  0.000273  0.061576 -0.001803  NaN       0.018487  0.000837 -0.007171 -0.000848 -0.000861 -0.000854 -0.000791 -0.000792 -0.000823 -0.000854 -0.000869 -0.000857 -0.000823 -0.000823
 1   -0.002617  -0.000240  0.001760  0.00000000  0.001243  0.000032 -0.000002 -0.003716 -0.000759  NaN   -0.057173  0.000273  0.061576 -0.001803  NaN       0.018487  0.000837 -0.007171 -0.000848 -0.000861 -0.000854 -0.000791 -0.000792 -0.000823 -0.000854 -0.000869 -0.000857 -0.000823 -0.000823
 2   -0.002617  -0.000240  0.001760  0.00000000  0.001243  0.000032 -0.000002 -0.003716 -0.000759  NaN   -0.057173  0.000273  0.061576 -0.001803  NaN       0.018487  0.000837 -0.007171 -0.000848 -0.000861 -0.000854 -0.000791 -0.000792 -0.000823 -0.000854 -0.000869 -0.000857 -0.000823 -0.000823
```

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

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

and to execute in JupyterLab, you will need to

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

```python
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_xaxis=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

```python
[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)
```

Chapter 6. ModelGridInterpolator
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, 'eep': 380.0, 'initial_mass': 1.0, 'radius': 1.0813017873811603, 'logTeff': 3.773295968705084, 'mass': 0.9997797219140423, 'density': 1.115827651504971, 'Mbol': 4.4508474939623826, ...
```
'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': 9.8288888526297722,
'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.

```python
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:

```python
eep = mist.get_eep(mass, age, feh, accurate=True)
pars = [eep, age, feh, distance, AV]
mod.lnprior(pars), mod.lnlike(pars), mod.lnpost(pars)
```

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

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

How long does a posterior evaluation take?

```python
%timeit mod.lnpost(pars)
```
1000 loops, best of 3: 369 µs per loop

```python
[6]: from isochrones import BinaryStarModel
mod2 = BinaryStarModel(mist, **props)
```

```python
[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
```

```python
[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.

```python
[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:

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

<table>
<thead>
<tr>
<th>age</th>
<th>feh</th>
<th>distance</th>
<th>AV</th>
<th>eep</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.775384</td>
<td>0.004928</td>
<td>9585.312354</td>
<td>0.058294</td>
<td>415</td>
</tr>
<tr>
<td>9.690678</td>
<td>0.318313</td>
<td>5460.742158</td>
<td>0.212007</td>
<td>295</td>
</tr>
<tr>
<td>9.317426</td>
<td>-0.008935</td>
<td>5381.226921</td>
<td>0.144259</td>
<td>265</td>
</tr>
<tr>
<td>9.721345</td>
<td>-0.131058</td>
<td>7867.502875</td>
<td>0.228851</td>
<td>295</td>
</tr>
<tr>
<td>9.374286</td>
<td>-0.325079</td>
<td>9590.728624</td>
<td>0.954695</td>
<td>350</td>
</tr>
<tr>
<td>9.293293</td>
<td>0.229220</td>
<td>9574.055273</td>
<td>0.713866</td>
<td>293</td>
</tr>
<tr>
<td>9.941975</td>
<td>0.178338</td>
<td>7788.336554</td>
<td>0.100102</td>
<td>272</td>
</tr>
<tr>
<td>9.436477</td>
<td>0.231631</td>
<td>9148.585364</td>
<td>0.204715</td>
<td>314</td>
</tr>
<tr>
<td>9.743647</td>
<td>-0.396267</td>
<td>6767.456426</td>
<td>0.383272</td>
<td>460</td>
</tr>
<tr>
<td>9.607588</td>
<td>-0.236938</td>
<td>4317.243131</td>
<td>0.795216</td>
<td>327</td>
</tr>
<tr>
<td>9.605723</td>
<td>-0.236801</td>
<td>9031.515061</td>
<td>0.488995</td>
<td>314</td>
</tr>
<tr>
<td>9.645887</td>
<td>-0.338786</td>
<td>9055.303060</td>
<td>0.408045</td>
<td>1702</td>
</tr>
<tr>
<td>9.997611</td>
<td>-0.214726</td>
<td>8452.833760</td>
<td>0.398581</td>
<td>327</td>
</tr>
<tr>
<td>9.926111</td>
<td>-0.063765</td>
<td>9726.761618</td>
<td>0.544188</td>
<td>321</td>
</tr>
<tr>
<td>9.845896</td>
<td>-0.106017</td>
<td>9148.167681</td>
<td>0.455272</td>
<td>292</td>
</tr>
</tbody>
</table>
```

(continues on next page)
### Chapter 7. Fitting stellar parameters

Remember, these are the fit parameters:

1. `mod.param_names`
2. `('eep', 'age', 'feh', 'distance', 'AV')`

Let’s turn this into a dataframe, and visualize it.
```python
[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)
```

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.

```python
[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))
```

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

```python
[15]:
pd.Series(mod._priors['mass'].sample(10000), name='mass').hvplot.hist(bins=100, bin_.range=(0, 5))
```
You can also define a metallicity prior to have a different mix of halo and (local) disk:

```python
from isochrones.priors import FehPrior
pd.Series(FehPrior(halo_fraction=0.5).sample(10000), name='feh').hvplot.hist()
```

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:

```python
mod.mnest_basename
```

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 by 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).

```python
mod.fit()
```

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.

```python
mod.samples.head()
```

```plaintext
<table>
<thead>
<tr>
<th>eep</th>
<th>age</th>
<th>feh</th>
<th>distance</th>
<th>AV</th>
<th>lnprob</th>
</tr>
</thead>
<tbody>
<tr>
<td>306.56</td>
<td>8.867352</td>
<td>-0.084787</td>
<td>99.754595</td>
<td>0.128567</td>
<td>-51.124</td>
</tr>
<tr>
<td>385.10</td>
<td>9.744207</td>
<td>0.179155</td>
<td>99.818131</td>
<td>0.492972</td>
<td>-49.729</td>
</tr>
<tr>
<td>301.01</td>
<td>8.745846</td>
<td>-0.030370</td>
<td>100.473273</td>
<td>0.579248</td>
<td>-49.425</td>
</tr>
<tr>
<td>259.68</td>
<td>8.214644</td>
<td>0.010053</td>
<td>98.376332</td>
<td>0.363801</td>
<td>-48.479</td>
</tr>
<tr>
<td>380.21</td>
<td>9.700131</td>
<td>-0.178482</td>
<td>99.398149</td>
<td>0.633511</td>
<td>-48.453</td>
</tr>
</tbody>
</table>
```

```python
mod.samples.describe()
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>eep</td>
<td>5344.0</td>
<td>373.1</td>
<td>19.8</td>
<td>217.0</td>
<td>359.1</td>
<td>375.1</td>
<td>387.9</td>
<td>420.5</td>
</tr>
<tr>
<td>age</td>
<td>5344.0</td>
<td>9.686</td>
<td>0.181</td>
<td>0.181</td>
<td>0.181</td>
<td>0.181</td>
<td>0.181</td>
<td>0.181</td>
</tr>
<tr>
<td>feh</td>
<td>5344.0</td>
<td>-0.045</td>
<td>0.076</td>
<td>-0.301</td>
<td>-0.099</td>
<td>-0.045</td>
<td>0.007</td>
<td>0.901</td>
</tr>
<tr>
<td>distance</td>
<td>5344.0</td>
<td>100.0</td>
<td>0.018</td>
<td>96.356</td>
<td>99.355</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>AV</td>
<td>5344.0</td>
<td>0.147</td>
<td>0.112</td>
<td>0.000</td>
<td>0.060</td>
<td>0.124</td>
<td>0.208</td>
<td>0.755</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th>eep</th>
<th>age</th>
<th>feh</th>
<th>mass</th>
<th>initial_mass</th>
<th>radius</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>106.0</td>
<td>8.673</td>
<td>-0.068</td>
<td>1.098369</td>
<td>1.098407</td>
<td>1.03756</td>
</tr>
<tr>
<td>1.01</td>
<td>105.0</td>
<td>8.742</td>
<td>0.159</td>
<td>1.057168</td>
<td>1.057394</td>
<td>1.14133</td>
</tr>
<tr>
<td>2.01</td>
<td>101.0</td>
<td>8.745</td>
<td>-0.011</td>
<td>1.152686</td>
<td>1.152721</td>
<td>1.096043</td>
</tr>
<tr>
<td>3.01</td>
<td>680.0</td>
<td>8.216</td>
<td>0.048</td>
<td>1.147012</td>
<td>1.147023</td>
<td>1.065911</td>
</tr>
<tr>
<td>4.01</td>
<td>210.0</td>
<td>9.700</td>
<td>-0.257</td>
<td>1.010634</td>
<td>1.010881</td>
<td>1.123263</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>logTeff</th>
<th>Teff</th>
<th>logg</th>
<th>...</th>
<th>BP_mag</th>
<th>RP_mag</th>
<th>W1_mag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>106.0</td>
<td>8.673</td>
<td>...</td>
<td>9.673531</td>
<td>8.942160</td>
<td>8.124458</td>
</tr>
<tr>
<td>1.01</td>
<td>105.0</td>
<td>8.742</td>
<td>...</td>
<td>10.181673</td>
<td>9.176220</td>
<td>8.004902</td>
</tr>
<tr>
<td>2.01</td>
<td>101.0</td>
<td>8.745</td>
<td>...</td>
<td>9.982068</td>
<td>9.077651</td>
<td>8.030137</td>
</tr>
<tr>
<td>3.01</td>
<td>680.0</td>
<td>8.216</td>
<td>...</td>
<td>9.830745</td>
<td>8.993534</td>
<td>8.044905</td>
</tr>
<tr>
<td>4.01</td>
<td>210.0</td>
<td>9.700</td>
<td>...</td>
<td>10.085126</td>
<td>9.130129</td>
<td>7.986850</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W2_mag</th>
<th>W3_mag</th>
<th>TESS_mag</th>
<th>Kepler_mag</th>
<th>parallax</th>
<th>distance</th>
<th>AV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>106.0</td>
<td>8.936</td>
<td>9.307189</td>
<td>10.024601</td>
<td>99.754595</td>
<td>0.12857</td>
</tr>
<tr>
<td>1.01</td>
<td>105.0</td>
<td>9.164</td>
<td>9.669073</td>
<td>10.018220</td>
<td>99.818131</td>
<td>0.492972</td>
</tr>
<tr>
<td>2.01</td>
<td>101.0</td>
<td>9.066</td>
<td>9.529013</td>
<td>9.952896</td>
<td>100.473273</td>
<td>0.579248</td>
</tr>
<tr>
<td>3.01</td>
<td>680.0</td>
<td>8.985</td>
<td>9.409833</td>
<td>10.165047</td>
<td>98.376332</td>
<td>0.363801</td>
</tr>
<tr>
<td>4.01</td>
<td>210.0</td>
<td>9.118</td>
<td>9.608276</td>
<td>10.060550</td>
<td>99.398149</td>
<td>0.633511</td>
</tr>
</tbody>
</table>
```

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

```python
[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:

```python
[25]: mod.corner_derived(['nu_max', 'delta_nu', 'density']);
```
7.3. Sampling the posterior
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 steller 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.

```python
[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.
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.

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

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

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.

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.
8.1. Unresolved multiple systems
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(age) should be around 9.6.

[9]: \texttt{from isochrones.priors import GaussianPrior}

\begin{verbatim}
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)
\end{verbatim}

[9]: -645802.7700077017
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:

```python
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=(2.43, 0.02) @(0.20, 100 [0.10])
  parallax=(2, 0.05)
fleh=(-0.12519050601435218, 0.1), parallax=(2, 0.05)

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

[13]: -8443.175970078633
```
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:

```python
from isochrones import StarModel
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()
```

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.)

```python
mod_resolved_unassoc.param_names
```

```
['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:

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

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

(continues on next page)
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])
      0_1
      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
    AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
      0_1
      1_0
      1_1
```

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