isochrones Documentation

Release 2.1

Timothy Morton

Feb 06, 2020
## Table of 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 DFInterpolator 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>StarPopulation object</td>
<td>57</td>
</tr>
<tr>
<td>9.2</td>
<td>ModelGridInterpolator.generate_binary</td>
<td>60</td>
</tr>
</tbody>
</table>
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 `nose tests isochrones` again, for good measure, to confirm that **MultiNest** works.)
## 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>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>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>
<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>
<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>
<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>
<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)
(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></td>
<td>36</td>
<td>298.570836</td>
<td>-1.0</td>
<td>0.003573</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37</td>
<td>297.180748</td>
<td>-1.0</td>
<td>0.004247</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>295.526946</td>
<td>-1.0</td>
<td>0.004217</td>
</tr>
<tr>
<td></td>
<td></td>
<td>39</td>
<td>293.589960</td>
<td>-1.0</td>
<td>0.004189</td>
</tr>
</tbody>
</table>

[2]: from isochrones.mist import MISTEvolutionTrackGrid

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

<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></td>
<td>143.524548</td>
<td>3.033277</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>145.419039</td>
<td>3.038935</td>
<td>2.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>147.409881</td>
<td>3.044805</td>
<td>3.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>149.499346</td>
<td>3.050886</td>
<td>4.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>1.593804</td>
<td>3.620834</td>
<td>0.1</td>
<td>0.034823</td>
<td>5.132871</td>
</tr>
<tr>
<td>1.583455</td>
<td>3.620769</td>
<td>0.1</td>
<td>0.035510</td>
<td>5.147664</td>
</tr>
<tr>
<td>1.572790</td>
<td>3.620702</td>
<td>0.1</td>
<td>0.036237</td>
<td>5.163015</td>
</tr>
<tr>
<td>1.561817</td>
<td>3.620631</td>
<td>0.1</td>
<td>0.037006</td>
<td>5.178922</td>
</tr>
<tr>
<td>1.550499</td>
<td>3.620558</td>
<td>0.1</td>
<td>0.037823</td>
<td>5.195452</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phase</th>
<th>feh</th>
<th>Teff</th>
<th>logL</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td>-3.978406</td>
<td>4176.707371</td>
<td>-0.157148</td>
</tr>
<tr>
<td>-1.0</td>
<td>-3.978406</td>
<td>4176.085183</td>
<td>-0.163066</td>
</tr>
<tr>
<td>-1.0</td>
<td>-3.978406</td>
<td>4175.435381</td>
<td>-0.169206</td>
</tr>
<tr>
<td>-1.0</td>
<td>-3.978406</td>
<td>4174.757681</td>
<td>-0.175569</td>
</tr>
<tr>
<td>-1.0</td>
<td>-3.978406</td>
<td>4174.049081</td>
<td>-0.182181</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dt_deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.026168</td>
</tr>
<tr>
<td>0.026121</td>
</tr>
<tr>
<td>0.026016</td>
</tr>
<tr>
<td>0.025996</td>
</tr>
<tr>
<td>0.025996</td>
</tr>
</tbody>
</table>
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'])
```

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

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

```python
(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
```

```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...

intermediate 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()

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

TESS Kepler
0 22.229761 22.950946
1 24.588818 26.383334
7 18.902509 19.105022
8 16.530374 16.884757
9 18.172883 18.341059

[5 rows x 29 columns]
2.4 Fit physical parameters of a star to observed data

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:

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

[11]:
<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>9.400976</td>
<td>-0.070581</td>
<td>99.008511</td>
<td>0.085589</td>
</tr>
<tr>
<td>50%</td>
<td>345.473042</td>
<td>9.630834</td>
<td>0.033488</td>
<td>104.310959</td>
<td>0.183091</td>
</tr>
<tr>
<td>75%</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:

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

<table>
<thead>
<tr>
<th></th>
<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>
<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>
<td>4.491137</td>
</tr>
<tr>
<td>std</td>
<td>0.823044</td>
<td>1.209516</td>
<td>3.740466</td>
<td>5501.680997</td>
<td>4.396153</td>
</tr>
<tr>
<td>min</td>
<td>0.899485</td>
<td>1.643165</td>
<td>3.750771</td>
<td>5634.112958</td>
<td>4.476522</td>
</tr>
<tr>
<td>25%</td>
<td>0.919331</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
<td>4.492272</td>
</tr>
<tr>
<td>50%</td>
<td>0.940155</td>
<td>1.832595</td>
<td>3.758868</td>
<td>5740.000379</td>
<td>4.506173</td>
</tr>
<tr>
<td>max</td>
<td>1.058840</td>
<td>2.217147</td>
<td>3.782650</td>
<td>6062.846454</td>
<td>4.553723</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<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>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.982881</td>
<td>1.738516</td>
<td>3.755075</td>
<td>5690.649562</td>
<td>4.491137</td>
</tr>
<tr>
<td>std</td>
<td>0.010738</td>
<td>0.107383</td>
<td>0.005747</td>
<td>75.536605</td>
<td>0.020924</td>
</tr>
<tr>
<td>min</td>
<td>0.941378</td>
<td>1.740170</td>
<td>3.754596</td>
<td>5683.909988</td>
<td>4.492272</td>
</tr>
<tr>
<td>25%</td>
<td>0.968818</td>
<td>1.832595</td>
<td>3.758868</td>
<td>5740.000379</td>
<td>4.506173</td>
</tr>
<tr>
<td>50%</td>
<td>0.990155</td>
<td>2.217147</td>
<td>3.782650</td>
<td>6062.846454</td>
<td>4.553723</td>
</tr>
<tr>
<td>max</td>
<td>1.058840</td>
<td>2.217147</td>
<td>3.782650</td>
<td>6062.846454</td>
<td>4.553723</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<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>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.828552</td>
<td>4.982881</td>
<td>157.087407</td>
<td>3538.322417</td>
</tr>
<tr>
<td>std</td>
<td>0.010738</td>
<td>0.010738</td>
<td>0.005747</td>
<td>75.536605</td>
</tr>
<tr>
<td>min</td>
<td>0.899485</td>
<td>4.913786</td>
<td>157.283081</td>
<td>3545.198779</td>
</tr>
<tr>
<td>25%</td>
<td>0.919331</td>
<td>4.985717</td>
<td>157.283081</td>
<td>3545.198779</td>
</tr>
<tr>
<td>50%</td>
<td>0.940155</td>
<td>5.061043</td>
<td>161.294116</td>
<td>3664.317241</td>
</tr>
<tr>
<td>max</td>
<td>0.968818</td>
<td>5.303972</td>
<td>177.056425</td>
<td>4130.359858</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<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>
<td>4643.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.008306</td>
<td>10.42067</td>
<td>9.539072</td>
<td>9.837970</td>
<td>101.801691</td>
</tr>
<tr>
<td>std</td>
<td>0.000654</td>
<td>0.009287</td>
<td>0.009087</td>
<td>0.0382167</td>
<td>3.989609</td>
</tr>
<tr>
<td>min</td>
<td>0.002593</td>
<td>10.386946</td>
<td>9.505070</td>
<td>8.421924</td>
<td>88.634174</td>
</tr>
<tr>
<td>25%</td>
<td>0.007966</td>
<td>10.413741</td>
<td>9.533035</td>
<td>9.586720</td>
<td>99.008511</td>
</tr>
<tr>
<td>50%</td>
<td>0.008247</td>
<td>10.420135</td>
<td>9.539162</td>
<td>9.844220</td>
<td>101.582447</td>
</tr>
<tr>
<td>75%</td>
<td>0.008619</td>
<td>10.426205</td>
<td>9.545132</td>
<td>10.100142</td>
<td>104.310959</td>
</tr>
<tr>
<td>max</td>
<td>0.011076</td>
<td>10.453965</td>
<td>9.574724</td>
<td>11.282330</td>
<td>118.737714</td>
</tr>
</tbody>
</table>

AV

| count  | 4643.000000 |
| mean   | 0.136494    |
isochrones Documentation, Release 2.1

(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()
``

<table>
<thead>
<tr>
<th>eep_0</th>
<th>eep_1</th>
<th>age</th>
<th>feh</th>
<th>distance</th>
<th>AV</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>359.086356</td>
<td>249.801089</td>
<td>9.821476</td>
<td>-0.073397</td>
<td>98.727349</td>
</tr>
</tbody>
</table>

(continues on next page)
```r
mod2.corner_observed();
```

```r
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
```

```r
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 11597.77825 0.0 0.002549 15.919824 13.882693
3 486.986772 13422.770521 0.0 0.006254 17.143644 14.860496
4 562.84178 15244.528561 0.0 0.007484 17.169701 14.890180
```

```r
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]
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>3</td>
<td>5</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>14</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>11</td>
<td>55</td>
<td>1024</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>17</td>
<td>102</td>
<td>3125</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>12</td>
<td>60</td>
<td>3125</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>18</td>
<td>128</td>
<td>4096</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>22</td>
<td>110</td>
<td>16805</td>
</tr>
</tbody>
</table>
```
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
[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:

```python
[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:

```python
[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:

```python
[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
```
```
<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>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>15</td>
<td>243</td>
<td>243</td>
</tr>
</tbody>
</table>
```

```
[7]: interp_missing = DFInterpolator(df_missing)
interp_missing([[1.3, 2.2]])
```
```
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]])
```
```
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>143.52</td>
<td>3.03</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.0</td>
<td>145.41</td>
<td>3.04</td>
<td>2.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.0</td>
<td>147.41</td>
<td>3.05</td>
<td>3.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.0</td>
<td>149.49</td>
<td>3.06</td>
<td>4.0</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.0</td>
<td>151.70</td>
<td>3.07</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>1.59</td>
<td>3.62</td>
<td>0.1</td>
<td>0.03</td>
<td>5.13</td>
</tr>
<tr>
<td>1.58</td>
<td>3.62</td>
<td>0.1</td>
<td>0.03</td>
<td>5.14</td>
</tr>
<tr>
<td>1.57</td>
<td>3.62</td>
<td>0.1</td>
<td>0.03</td>
<td>5.16</td>
</tr>
<tr>
<td>1.56</td>
<td>3.62</td>
<td>0.1</td>
<td>0.03</td>
<td>5.18</td>
</tr>
<tr>
<td>1.55</td>
<td>3.62</td>
<td>0.1</td>
<td>0.03</td>
<td>5.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phase</th>
<th>feh</th>
<th>Teff</th>
<th>logL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.0</td>
<td>-3.98</td>
<td>4176</td>
</tr>
<tr>
<td>2</td>
<td>-1.0</td>
<td>-3.98</td>
<td>4176</td>
</tr>
<tr>
<td>3</td>
<td>-1.0</td>
<td>-3.98</td>
<td>4176</td>
</tr>
<tr>
<td>4</td>
<td>-1.0</td>
<td>-3.98</td>
<td>4176</td>
</tr>
<tr>
<td>5</td>
<td>-1.0</td>
<td>-3.98</td>
<td>4176</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>21.78</td>
<td>False</td>
<td>13343</td>
<td>4.12</td>
</tr>
<tr>
<td>21.99</td>
<td>False</td>
<td>14171</td>
<td>4.15</td>
</tr>
<tr>
<td>22.22</td>
<td>False</td>
<td>15048</td>
<td>4.17</td>
</tr>
<tr>
<td>22.46</td>
<td>False</td>
<td>16962</td>
<td>4.22</td>
</tr>
<tr>
<td>22.71</td>
<td>False</td>
<td>18975</td>
<td>4.24</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>35</td>
<td>-4.0</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>5.0</td>
<td>36</td>
<td>-4.0</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>5.0</td>
<td>37</td>
<td>-4.0</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>5.0</td>
<td>38</td>
<td>-4.0</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>5.0</td>
<td>39</td>
<td>-4.0</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>radius</th>
<th>density</th>
<th>logTeff</th>
<th>Teff</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>
```
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
```

### 4.2. Model Grid Objects and Interpolation 23
<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>acoustic_cutoff</th>
<th>max_conv_vel_div_csound</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>2233.536029</td>
<td>0.127243</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>2228.014832</td>
<td>0.128938</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>2218.440338</td>
<td>0.130528</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>2207.403678</td>
<td>0.132657</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>2194.776391</td>
<td>0.134294</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>max_gradT_div_grada</th>
<th>gradT_excess_alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>1.095544</td>
<td>0.0</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>1.101114</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>1.109114</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>1.116760</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>1.124050</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>min_Pgas_div_P</th>
<th>max_L_rad_div_Ledd</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>0.999989</td>
<td>0.000016</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>0.999989</td>
<td>0.000017</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>0.999988</td>
<td>0.000019</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>0.999987</td>
<td>0.000021</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>0.999986</td>
<td>0.000022</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>log10_isochrone_age_yr</th>
<th>feh</th>
<th>EEP</th>
<th>e_thermal</th>
<th>phase</th>
<th>feh</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>-4.0</td>
<td>35</td>
<td>3.002314E+46</td>
<td>-1.0</td>
<td>-4.0</td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>3.106838E+46</td>
<td>-1.0</td>
<td>-4.0</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>3.264230E+46</td>
<td>-1.0</td>
<td>-4.0</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>3.424460E+46</td>
<td>-1.0</td>
<td>-4.0</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>3.587613E+46</td>
<td>-1.0</td>
<td>-4.0</td>
<td></td>
</tr>
</tbody>
</table>

[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_psi', '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 \texttt{.interp} method:

```python
[5]: track_grid.interp([-0.12, 1.01, 353.1], ['mass', 'radius', 'logg', 'Teff'])
```

```python
[5]: array([1.00983180e+00, 1.04691913e+00, 4.40266419e+03, 6.0338320e+03])
```

Similarly, the \texttt{.interp_orig} method interpolates any of the original columns by name:

```python
[6]: track_grid.interp_orig([-0.12, 1.01, 353.1], ['v_wind_Km_per_s'])
```

```python
[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:

```python
[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]) == interp([-0.12, 1.01, 353.1])
```

```python
[8]: %timeit interp([-0.12, 1.01, 353.1])
```

```python
%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:

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

```python
EeP 193 9.0 0.042799 0.100000 0.100000 0.126216 70.115891
194 9.0 0.042805 0.103449 0.103449 0.129201 67.622476
195 9.0 0.042814 0.108103 0.108103 0.133343 64.282466
196 9.0 0.042824 0.112932 0.112932 0.137791 60.858291
197 9.0 0.042834 0.117543 0.117543 0.142177 57.660112
```

(continues on next page)
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)
```

---

<table>
<thead>
<tr>
<th></th>
<th>Log(age)</th>
<th>Log(Teff)</th>
<th>Log(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>27533.135930</td>
<td>-1.0</td>
<td>0.003449</td>
</tr>
<tr>
<td>194</td>
<td>27025.521973</td>
<td>-1.0</td>
<td>0.004051</td>
</tr>
<tr>
<td>195</td>
<td>26339.315728</td>
<td>-1.0</td>
<td>0.004742</td>
</tr>
<tr>
<td>196</td>
<td>25622.963627</td>
<td>-1.0</td>
<td>0.004720</td>
</tr>
<tr>
<td>197</td>
<td>24938.691941</td>
<td>-1.0</td>
<td>0.004735</td>
</tr>
</tbody>
</table>
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()

<table>
<thead>
<tr>
<th>Teff</th>
<th>logg</th>
<th>[Fe/H]</th>
<th>Av</th>
<th>g</th>
<th>r</th>
<th>i</th>
<th>J</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.00</td>
<td>-6.534742</td>
<td>-3.332877</td>
<td>-1.617626</td>
<td>1.845781</td>
<td>2.927064</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.05</td>
<td>-6.590469</td>
<td>-3.375570</td>
<td>-1.650338</td>
<td>1.831466</td>
<td>2.917990</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.10</td>
<td>-6.646182</td>
<td>-3.418258</td>
<td>-1.683043</td>
<td>1.817153</td>
<td>2.908916</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.15</td>
<td>-6.701881</td>
<td>-3.460939</td>
<td>-1.715740</td>
<td>1.802841</td>
<td>2.899842</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.20</td>
<td>-6.757566</td>
<td>-3.503615</td>
<td>-1.748429</td>
<td>1.788530</td>
<td>2.890769</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Teff</th>
<th>logg</th>
<th>[Fe/H]</th>
<th>Av</th>
<th>K</th>
<th>G</th>
<th>BP</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.00</td>
<td>3.436304</td>
<td>-2.181986</td>
<td>-4.652544</td>
<td>-0.881255</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.05</td>
<td>3.430463</td>
<td>-2.211637</td>
<td>-4.697700</td>
<td>-0.909057</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.10</td>
<td>3.424623</td>
<td>-2.241240</td>
<td>-4.742838</td>
<td>-0.936829</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.15</td>
<td>3.418782</td>
<td>-2.270797</td>
<td>-4.787959</td>
<td>-0.964571</td>
</tr>
<tr>
<td>2500.0</td>
<td>-4.0</td>
<td>-4.0</td>
<td>0.20</td>
<td>3.412942</td>
<td>-2.300306</td>
<td>-4.833062</td>
<td>-0.992285</td>
</tr>
</tbody>
</table>
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
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'])
```

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

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

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

```
(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:

```
[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]:
<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>223</td>
<td>223.0</td>
<td>9.53</td>
<td>0.150280</td>
<td>0.143050</td>
<td>0.143050</td>
<td>0.174516</td>
</tr>
<tr>
<td>224</td>
<td>224.0</td>
<td>9.53</td>
<td>0.150322</td>
<td>0.147584</td>
<td>0.147584</td>
<td>0.178799</td>
</tr>
<tr>
<td>225</td>
<td>225.0</td>
<td>9.53</td>
<td>0.150371</td>
<td>0.152520</td>
<td>0.152521</td>
<td>0.183594</td>
</tr>
<tr>
<td>226</td>
<td>226.0</td>
<td>9.53</td>
<td>0.150419</td>
<td>0.157318</td>
<td>0.157319</td>
<td>0.184463</td>
</tr>
<tr>
<td>227</td>
<td>227.0</td>
<td>9.53</td>
<td>0.150468</td>
<td>0.161795</td>
<td>0.161796</td>
<td>0.189168</td>
</tr>
</tbody>
</table>

logTeff  | Teff   | logg   | ... | H_mag   | K_mag   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>223</td>
<td>3.477544</td>
<td>3003.536405</td>
<td>5.121475</td>
<td>...</td>
<td>8.785652</td>
</tr>
<tr>
<td>224</td>
<td>3.479902</td>
<td>3019.769652</td>
<td>5.112821</td>
<td>...</td>
<td>8.713187</td>
</tr>
<tr>
<td>225</td>
<td>3.482375</td>
<td>3036.910262</td>
<td>5.103613</td>
<td>...</td>
<td>8.635963</td>
</tr>
<tr>
<td>226</td>
<td>3.480519</td>
<td>3024.116433</td>
<td>5.101786</td>
<td>...</td>
<td>8.629300</td>
</tr>
<tr>
<td>227</td>
<td>3.482801</td>
<td>3040.176145</td>
<td>5.093340</td>
<td>...</td>
<td>8.558717</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G_mag</th>
<th>BP_mag</th>
<th>RP_mag</th>
<th>W1_mag</th>
<th>W2_mag</th>
<th>W3_mag</th>
<th>TESS_mag</th>
</tr>
</thead>
</table>

Kepler_mag
```

223 12.864034
224 12.755405
225 12.640135

(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.

```python
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)
```

```
   nu_max    logg    eep    initial_mass    radius    logTeff    mass
0  4254.6296  4.548780  350.0    0.80  0.787407  3.707984  0.799894
1  3622.3209  4.495440  350.0    0.90  0.888064  3.741043  0.899876
2  3041.1079  4.432089  350.0    1.00  1.006928  3.766249  0.999860
```

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.

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

```
   nu_max    logg    eep    initial_mass    radius    logTeff    mass
0  4787.5983  4.595858  320.808  0.81  0.750611  3.699978  0.809963
1  3986.6718  4.535170  332.280  0.91  0.853120  3.737424  0.909935
2  3154.6779  4.447853  343.800  1.01  0.993830  3.766201  1.009887
```

6.3. Generating synthetic properties
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)
```

(continues on next page)
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()
```

```
nu_max -0.002517
logg  -0.000240
eep   0.001600
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_xaxis=True, legend_position='bottom_left', width=600)
```

```
[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)
```
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.

```python
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
```

```plaintext
{'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,
```

(continues on next page)
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]
```

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

```python
(-23.05503287088296, -20.716150242083508, -43.77118311296647)
```

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

```python
pars2 = [eep + 3, age - 0.05, feh + 0.02, distance, AV]
```

```python
mod.lnprior(pars2), mod.lnlike(pars2), mod.lnpost(pars2)
```

```python
(-23.251706955307853, -85.08590699022739, -108.3376139453524)
```

How long does a posterior evaluation take?

```python
%timeit mod.lnpost(pars)
```
from isochrones import BinaryStarModel
mod2 = BinaryStarModel(mist, **props)

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.

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

7.2 Priors

As you may have noticed, we have not explictly 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.

mod._priors

{'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:

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>0.775384</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>0.954659</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>10.057273</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)
Remember, these are the fit parameters:

\[\text{mod.param_names} = ('\text{eep}', '\text{age}', '\text{feh}', '\text{distance}', '\text{AV}')\]

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

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

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

```python
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()
```

<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>0 306.562211</td>
<td>8.867352</td>
<td>-0.084787</td>
<td>99.754595</td>
<td>0.128567</td>
<td>-51.124447</td>
</tr>
<tr>
<td>1 385.106002</td>
<td>9.744207</td>
<td>0.179155</td>
<td>99.818131</td>
<td>0.492972</td>
<td>-49.729296</td>
</tr>
<tr>
<td>2 301.018106</td>
<td>8.745846</td>
<td>-0.030370</td>
<td>100.473273</td>
<td>0.579248</td>
<td>-49.425361</td>
</tr>
<tr>
<td>3 259.680008</td>
<td>8.214644</td>
<td>0.010053</td>
<td>98.376332</td>
<td>0.363801</td>
<td>-48.479515</td>
</tr>
<tr>
<td>4 380.210824</td>
<td>9.700131</td>
<td>-0.178482</td>
<td>99.398149</td>
<td>0.633511</td>
<td>-48.453800</td>
</tr>
</tbody>
</table>

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

```text
<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>count</td>
<td>5344.000000</td>
<td>5344.000000</td>
<td>5344.000000</td>
<td>5344.000000</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>373.193478</td>
<td>9.686096</td>
<td>-0.045856</td>
<td>100.021338</td>
<td>0.146828</td>
</tr>
<tr>
<td>std</td>
<td>19.800711</td>
<td>0.181541</td>
<td>0.076720</td>
<td>0.993574</td>
<td>0.111217</td>
</tr>
<tr>
<td>min</td>
<td>217.053516</td>
<td>7.653568</td>
<td>-0.301781</td>
<td>96.356886</td>
<td>0.000078</td>
</tr>
<tr>
<td>25%</td>
<td>359.164345</td>
<td>9.599845</td>
<td>-0.098619</td>
<td>99.355457</td>
<td>0.060500</td>
</tr>
<tr>
<td>50%</td>
<td>375.054046</td>
<td>9.712589</td>
<td>-0.045498</td>
<td>100.013786</td>
<td>0.124330</td>
</tr>
<tr>
<td>75%</td>
<td>387.980876</td>
<td>9.806347</td>
<td>0.007986</td>
<td>100.685282</td>
<td>0.208648</td>
</tr>
<tr>
<td>max</td>
<td>420.506604</td>
<td>10.089448</td>
<td>0.193737</td>
<td>103.660815</td>
<td>0.755101</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()
```

| eep | age     | feh | mass  | initial_mass | radius | density | logTeff | Teff   | logg | ... | BP_mag | RP_mag | W1_mag | logTeff | Teff   | logg | ... | BP_mag | RP_mag | W1_mag | logTeff | Teff   | logg | ... | BP_mag | RP_mag | W1_mag | logTeff | Teff   | logg | ... | BP_mag | RP_mag | W1_mag | logTeff | Teff   | logg | ... | BP_mag | RP_mag | W1_mag |
|-----|---------|-----|-------|--------------|--------|---------|--------|--------|------|-----|-------|--------|--------|--------|--------|--------|------|-----|-------|--------|--------|--------|--------|------|-----|-------|--------|--------|--------|--------|------|-----|-------|--------|--------|--------|--------|------|-----|-------|--------|--------|--------|--------|------|-----|-------|--------|--------|

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
```

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

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

```
[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.
```python
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.

```
8.1. Unresolved multiple systems

[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(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
```
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:

```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()
```

```plaintext
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=(5834.78299979719397, 100), logg=(4.435999146983706, 0.15),
    →feh=(-0.01251905060143500, 0.1), parallax=(2, 0.05)
    AO delta-K=(2.43, 0.02) @(0.20, 100 [0.10])
    0_1, parallax=(2, 0.05)
```

```python
[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
[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()
```

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
[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:

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

(continues on next page)
Here is a situation where the faint visual binary is an unrelated binary star:

```ipython
[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
      0_1
    AO delta-K=(2.43, 0.02) @ (0.20, 100 [0.10])
      0_2
```

Here, both AO stars are unresolved binaries:

```ipython
[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...
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 \text{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:

```python
from scipy.stats import uniform, norm
from isochrones import get_ichrone
from isochrones.priors import GaussianPrior, SalpeterPrior, DistancePrior, FlatPrior
```
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.

```
df = pop.generate(1000)
df.head()
```

```
    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       WI_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  14.208519  0.014534
2  1.740697  2.306514  3.794831 -0.366979  14.208519  0.014534
3  1.199744  3.803016  3.815517 -0.668881  14.208519  0.014534
4  2.463683  2.543589  3.785193 -0.307018  14.208519  0.014534

    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  13.316868  0.001710  15.193251  0.114144  15.567852
3  14.327035  0.017125  14.301929  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
```
Note that this operation is not nearly as fast as directly interpolating an isochrone or evolution track grid (since generating properties given mass, age, and metallicity necessarily involves solving for EEP first):

```
%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):

```
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:

```
', '.join(df.columns)

'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, 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 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_[x] columns, which give the specific extinction per band for each system (and for the individual components of the binary).

```
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)
```

9.1. StarPopulation object
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:

```python
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)
```

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:

```python
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']]
```

<table>
<thead>
<tr>
<th>G_mag</th>
<th>BP_mag</th>
<th>RP_mag</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.459204</td>
<td>9.822055</td>
<td>8.928917</td>
</tr>
<tr>
<td>9.669059</td>
<td>10.018673</td>
<td>9.150093</td>
</tr>
<tr>
<td>9.709775</td>
<td>10.046349</td>
<td>9.204878</td>
</tr>
<tr>
<td>9.718787</td>
<td>10.050889</td>
<td>9.219106</td>
</tr>
</tbody>
</table>