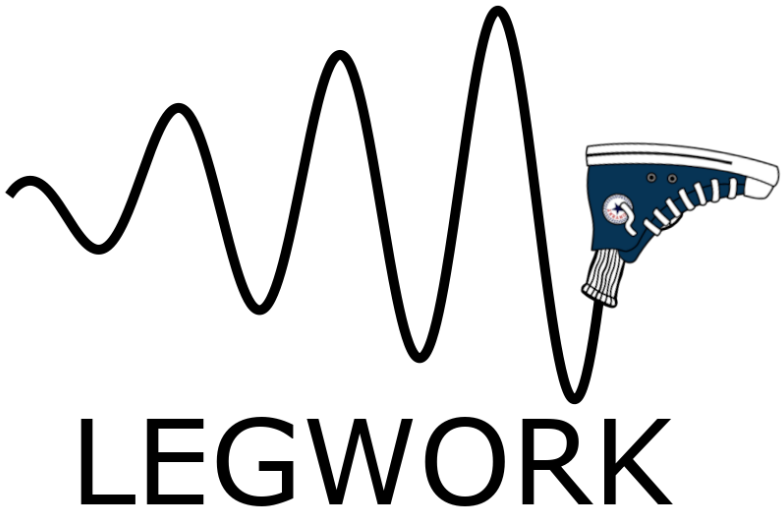
LEGWORK

Tom Wagg, Katie Breivik

Oct 12, 2023

CONTENTS



LEGWORK (**L**ISA **E**volution and **G**ravitational **W**ave **OR**bit **K**it) is a python package designed to calculate signal-to-noise ratios for GWs emitted from inspiraling binary systems that are potentially observable by LISA.

LEGWORK also contains a plotting module so you can show off those detectable GW sources of yours and treats any and all kinds of stellar-origin inspiralling binaries. Circular and stationary? No problem! Eccentric and chirping? We've got you covered!

Want to determine if *your* favorite source is detectable by LISA? Let LEGWORK do the legwork and keep the LISA literature free from spurious factors of 2!

Stable (with conda)

This is our recommend installation method! Follow the steps below to start using `LEGWORK`!

1. `Download the environment.yml file from our repository`
2. Create a new conda environment using this file

   ```
   conda env create -f path/to/environment.yml
   ```

3. Activate the environment by running

   ```
   conda activate legwork
   ```

and you should be all set! Check out our *quickstart tutorial* to learn some LEGWORK basics. Note that if you also want to work with the notebooks in the tutorials and/or demos you'll also need to install jupyter/ipython in this environment!

Stable (without conda)

We don't recommend installing `LEGWORK` without a conda environment but if you prefer to do it this way then all you need to do is run

```
pip install legwork
```

and you should be all set! Check out our *quickstart tutorial* to learn some LEGWORK basics. Note that if you also want to work with the notebooks in the tutorials and/or demos you'll also need to install jupyter/ipython in this environment!

Development (from GitHub)

> **Warning:** We don't guarantee that there won't be mistakes or bugs in the development version, use at your own risk!

The latest development version is available directly from our GitHub Repo. To start, clone the repository onto your machine:

```
git clone https://github.com/TeamLEGWORK/LEGWORK
cd LEGWORK
```

Next, we recommend that you create a Conda environment for working with LEGWORK. You can do this by running

```
conda create --name legwork "python>=3.7" pip "numba>=0.50" "numpy>=1.17" "astropy>=4.0"
↪"scipy>=1.5.0" "matplotlib>=3.3.2" "seaborn>=0.11.1" "schwimmbad>=0.3.2" -c conda-
↪forge -c defaults
```

And then activate the environment by running

```
conda activate legwork
```

At this point, all that's left to do is install LEGWORK!

```
pip install .
```

and you should be all set! Check out our *quickstart tutorial* to learn some LEGWORK basics. Note that if you also want to work with the notebooks in the tutorials and/or demos you'll also need to install jupyter/ipython in this environment!

CHAPTER

ONE

CITING LEGWORK

If you use LEGWORK in a scientific publication we ask that you please cite it. LEGWORK was jointly published in ApJS and JOSS and **we ask that you please cite both papers**.

Below we include two ready-made BibTeX entries, one for each paper. We recommend that you cite the ApJS paper when you mention LEGWORK in the main text and cite the JOSS paper in the Software section at the end of your paper.

```
@ARTICLE{LEGWORK_apjs,
    author = {{Wagg}, T. and {Breivik}, K. and {de Mink}, S.~E.},
    title = "{LEGWORK: A Python Package for Computing the Evolution and Detectability of␣
↪Stellar-origin Gravitational-wave Sources with Space-based Detectors}",
    journal = {\apjs},
    keywords = {Open source software, Gravitational waves, Gravitational wave detectors,␣
↪Compact objects, Orbital evolution, White dwarf stars, Neutron stars, Stellar mass␣
↪black holes, 1866, 678, 676, 288, 1178, 1799, 1108, 1611, Astrophysics - High Energy␣
↪Astrophysical Phenomena, General Relativity and Quantum Cosmology},
    year = 2022,
    month = jun,
    volume = {260},
    number = {2},
    eid = {52},
    pages = {52},
    doi = {10.3847/1538-4365/ac5c52},
    archivePrefix = {arXiv},
    eprint = {2111.08717},
    primaryClass = {astro-ph.HE},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2022ApJS..260...52W},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}

@ARTICLE{LEGWORK_joss,
    author = {{Wagg}, Tom and {Breivik}, Katelyn and {de Mink}, Selma},
    title = "{LEGWORK: A python package for computing the evolution and detectability of␣
↪stellar-origin gravitational-wave sources with space-based detectors}",
    journal = {The Journal of Open Source Software},
    keywords = {neutron stars, white dwarfs, TianQin, Python, stellar mass black holes,␣
↪gravitational wave detectors, orbital evolution, compact objects, gravitational waves,␣
↪LISA, astronomy},
    year = 2022,
    month = feb,
    volume = {7},
```

(continues on next page)

(continued from previous page)

```
    number = {70},
    eid = {3998},
    pages = {3998},
    doi = {10.21105/joss.03998},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2022JOSS....7.3998W},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

More citation methods are available on ADS for the ApJS and JOSS papers if you'd rather use a different format!

CHAPTER

TWO

TUTORIALS

This section contains a series of tutorials on how to use and get the most out of LEGWORK. We are working to add more tutorials over time and are very open to feedback!

The fastest way to get started is our quickstart tutorial:

For more in-depth details and examples, check out our other tutorials:

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.1 Quickstart

In this tutorial, we explain how to quickly use LEGWORK to calculate the detectability of a collection of sources.

Let's start by importing the source and visualisation modules of LEGWORK and some other common packages.

[2]:
```
import legwork.source as source
import legwork.visualisation as vis

import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

Next let's create a random collection of possible LISA sources in order to assess their detectability.

[4]:
```
# create a random collection of sources
n_values = 1500
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.normal(8, 1.5, n_values) * u.kpc
f_orb = 10**(-5 * np.random.power(3, n_values)) * u.Hz
ecc = 1 - np.random.power(5, n_values)
```

We can instantiate a Source class using these random sources in order to analyse the population. There are also a series of optional parameters which we don't cover here but if you are interested in the purpose of these then check out the *Using the Source Class* tutorial.

[5]:
```
sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb)
```

This `Source` class has many methods for calculating strains, visualising populations and more. You can learn more about these in the *Using the Source Class* tutorial. For now, we shall focus only on the calculation of the signal-to-noise ratio.

Therefore, let's calculate the SNR for these sources. We set `verbose=True` to give an impression of what sort of sources we have created. This function will split the sources based on whether they are stationary/evolving and circular/eccentric and use one of 4 SNR functions for each subpopulation.

[6]:
```
snr = sources.get_snr(verbose=True)
```

```
Calculating SNR for 1500 sources
        0 sources have already merged
        1396 sources are stationary
                403 sources are stationary and circular
                993 sources are stationary and eccentric
        104 sources are evolving
                24 sources are evolving and circular
                80 sources are evolving and eccentric
```

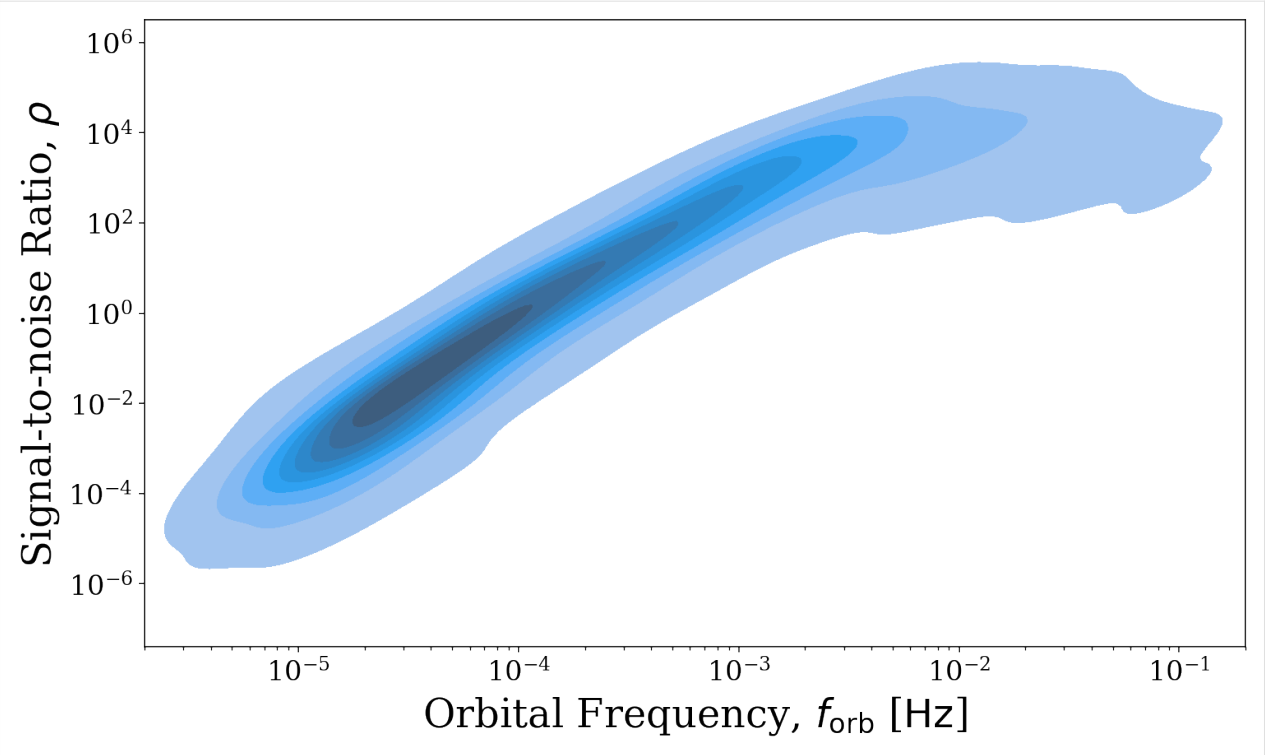
These SNR values are now stored in `sources.snr` and we can mask those that don't meet some detectable threshold.

[7]:
```
detectable_threshold = 7
detectable_sources = sources.snr > 7
print("{} of the {} sources are detectable".format(len(sources.snr[detectable_sources]),␣
↪n_values))
```

```
580 of the 1500 sources are detectable
```

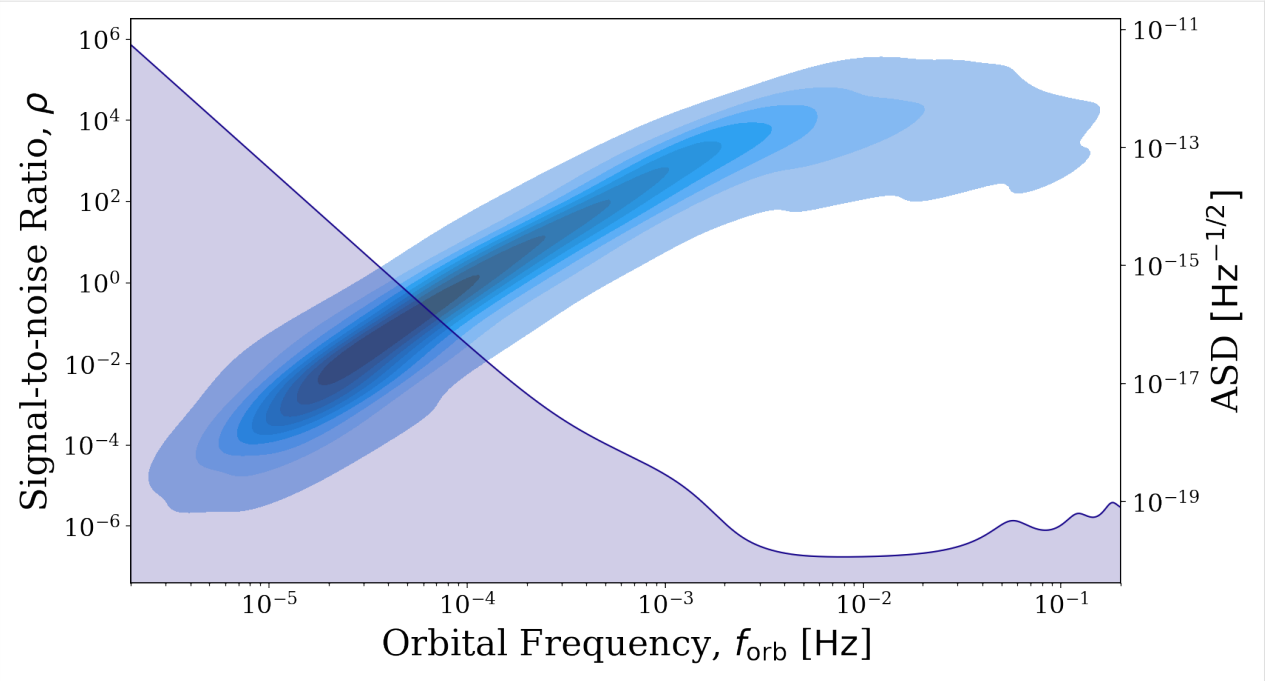
And just like that we know the number of detectable sources! It could be interesting to see how the SNR varies with orbital frequency so let's use the *`legwork.source.Source.plot_source_variables()`* to create a 2D density distribution of these variables.

[8]:
```
fig, ax = sources.plot_source_variables(xstr="f_orb", ystr="snr", disttype="kde", log_
↪scale=(True, True),
                                        fill=True, xlim=(2e-6, 2e-1), which_
↪sources=sources.snr > 0)
```



The reason for this shape may not be immediately obvious. However, if we also use the visualisation module to overlay the LISA sensitivity curve, it becomes clear that the SNRs increase in step with the decrease in the noise and flatten out as the sensitivity curve does as we would expect. To learn more about the visualisation options that LEGWORK offers, check out the *Visualisation* tutorial.

[9]:
```
# create the same plot but set `show=False`
fig, ax = sources.plot_source_variables(xstr="f_orb", ystr="snr", disttype="kde", log_
↪scale=(True, True),
                                        fill=True, show=False, which_sources=sources.snr␣
↪> 0)

# duplicate the x axis and plot the LISA sensitivity curve
right_ax = ax.twinx()
frequency_range = np.logspace(np.log10(2e-6), np.log10(2e-1), 1000) * u.Hz
vis.plot_sensitivity_curve(frequency_range=frequency_range, fig=fig, ax=right_ax)

plt.show()
```



That's it for this quickstart into using LEGWORK. For more details on using LEGWORK to calculate strains, evolve binaries and visualise their distributions check out the *other tutorials* and *demos* in these docs! You can also read more about the scope and limitations of LEGWORK *on this page*.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.2 Using the Source Class

In this tutorial, we'll take an in-depth look at the LEGWORK Source class and its various methods that you can use to your advantage.

Let's start by importing the source module of LEGWORK and some other common packages.

[2]:
```
from legwork import source, utils, visualisation

import numpy as np
import astropy.units as u
from astropy.coordinates import SkyCoord
import matplotlib.pyplot as plt
```

2.2.1 Instantiating a Source Class

The Source class gives you access to the majority of the functionality of LEGWORK from one simple class. Once a collection of sources is instantiated in this class you can calculate their strains and signal-to-noise ratios, evolve them over time and visualise their distributions or plot them on the LISA sensitivity curve.

Basic input

Let's start by creating a random collection of possible LISA sources.

[4]:
```
# create a random collection of sources
n_values = 1500
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.normal(8, 1.5, n_values) * u.kpc
f_orb = 10**(-5 * np.random.power(3, n_values)) * u.Hz
ecc = 1 - np.random.power(3, n_values)
```

We can instantiate a Source class using these random sources as follows:

[5]:
```
sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb, gw_lum_tol=0.
↪05, stat_tol=1e-2,
                        interpolate_g=True, interpolate_sc=True,
                        sc_params={
                            "instrument": "LISA",
                            "custom_psd": None,
                            "t_obs": 4 * u.yr,
                            "L": 2.5e9 * u.m,
                            "approximate_R": False,
                            "confusion_noise": 'robson19'
                        })
```

There's a couple of things to focus on here. The masses, eccentricity and distance are all required for every source. However, you can choose to *either* supply the orbital frequency, f_orb, *or* the semi-major axis, a. Whichever you supply, the other will be immediately calculating using Kepler's third law and can be accessed immediately.

[6]:
```
# the semi-major axes have been calculated
sources.a
```

[6]: $[0.0022907693,\ 0.0001858196,\ 0.022224962,\ \ldots,\ 0.013641543,\ 0.054689977,\ 0.0024257776]$ AU

Gravitational Wave Luminosity Tolerance

Next, we set the tolerance for the accuracy of the gravitational wave luminosity, gw_lum_tol, to 5%. This tolerance is used for two calculations:

1. finding the transition between "circular" and "eccentric" binaries
2. determining the number of harmonics required to capture the entire signal of a binary

Since, a more stringent tolerance will require binaries to be considered eccentric at lower eccentricities (and thus consider more than the $n = 2$ harmonic), whilst also require computing *more* harmonics in order to get the luminosity equal to the true value within the tolerance.

Let's see how this works in practice.

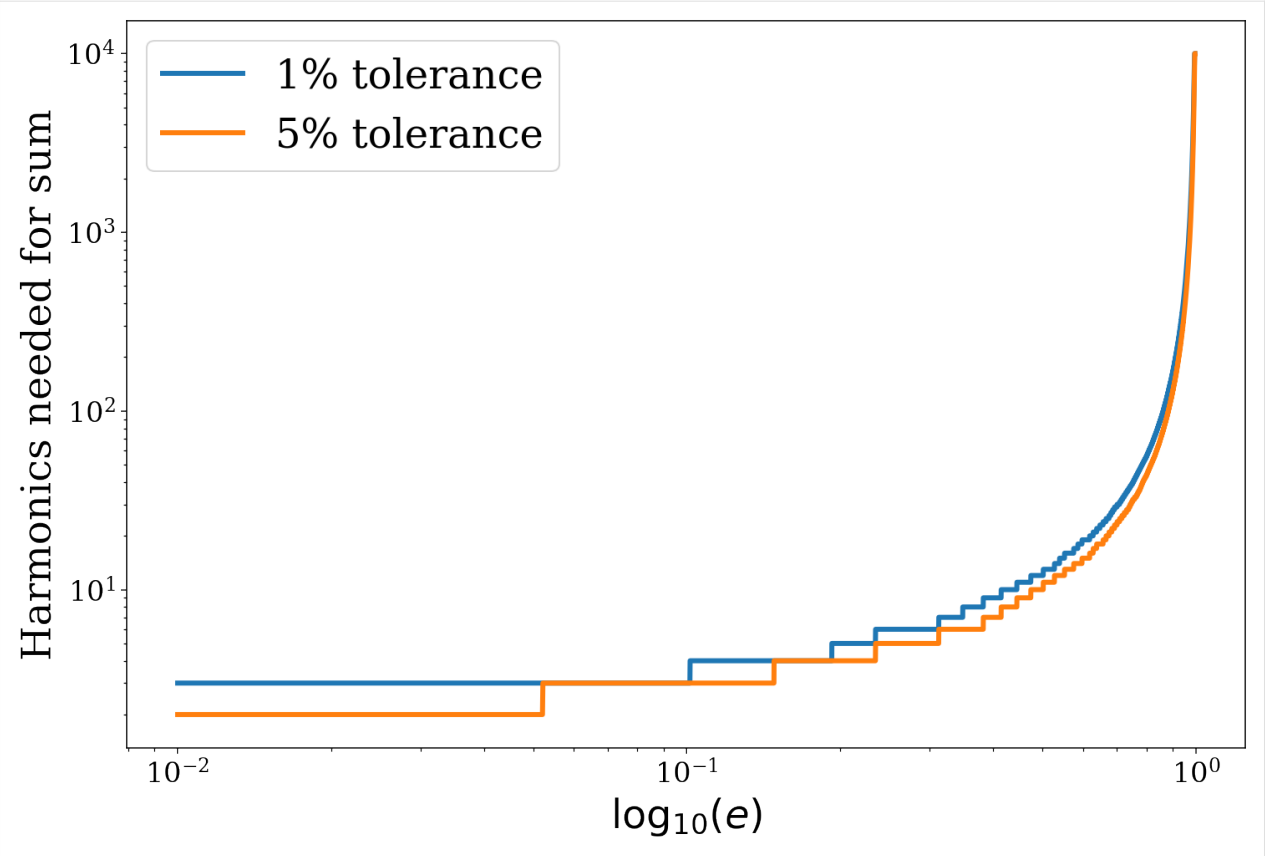
[7]:
```
# start with a tolerance of 5%
sources.update_gw_lum_tol(gw_lum_tol=0.05)
print("Eccentricity at which we consider sources to be eccentric is {:.3}".
↪format(sources.ecc_tol),
      "for a tolerance of {:.2}".format(sources._gw_lum_tol))

# change to a tolerance of 1%
sources.update_gw_lum_tol(gw_lum_tol=0.01)

print("Eccentricity at which we consider sources to be eccentric is {:.3}".
↪format(sources.ecc_tol),
      "for a tolerance of {:.2}".format(sources._gw_lum_tol))
```

```
Eccentricity at which we consider sources to be eccentric is 0.0667 for a tolerance of 0.
↪05
Eccentricity at which we consider sources to be eccentric is 0.0295 for a tolerance of 0.
↪01
```

Warning: if you need to change the `gw_lum_tol` of a Source class then it is critical that you do so using the *legwork.source.Source.update_gw_lum_tol()* (example above) rather than changing `Source._gw_lum_tol` directly. This is because the function also makes sure that `ecc_tol` and `harmonics_required` function are recalculated and kept in sync.

We can also plot the number of required harmonics for each eccentricity for different tolerances and see that you always need a greater or equal number of harmonics for a smaller tolerance.

[8]:
```
fig, ax = plt.subplots()
e_range = np.linspace(0.01, 0.995, 10000)

# change to a tolerance of 1%
sources.update_gw_lum_tol(gw_lum_tol=0.01)
ax.loglog(e_range, sources.harmonics_required(e_range), lw=3, label="1% tolerance")

# reset to a tolerance of 5%
sources.update_gw_lum_tol(gw_lum_tol=0.05)
ax.loglog(e_range, sources.harmonics_required(e_range), lw=3, label="5% tolerance")

ax.legend()

ax.set_xlabel(r"$\log_{10}(e)$")
ax.set_ylabel("Harmonics needed for sum")

plt.show()
```



Stationary Tolerance

We also set the stationary tolerance in instantiating the class. This tolerance is used to determine which binaries are stationary in frequency space, and thus for which ones we can use the stationary approximation of the SNR. We define a binary as stationary in frequency space on the timescale of the LISA mission if the fractional change in orbital frequency, $\Delta f_{\rm orb}/f_{\rm orb}$, is less than or equal to the tolerance.

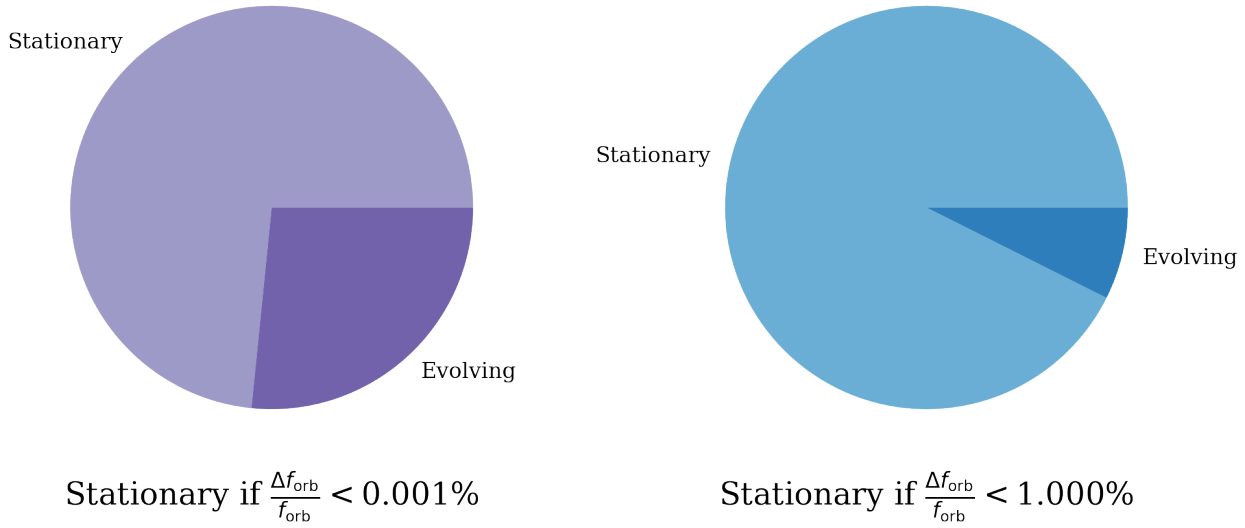
We can see how this changes things in practice.

[9]:
```
# create a plot
fig, axes = plt.subplots(1, 2, figsize=(16, 8))
fig.subplots_adjust(wspace=0.3)

# use the same labels for each
labels = ["Stationary", "Evolving"]

# loop over tolerances and colours
for i, vals in enumerate([(1e-5, "Purples"), (1e-2, "Blues")]):
    tol, col = vals

    # adjust the tolerance
    sources.stat_tol = tol

    # get a mask for the stationary binaries
    stat_mask = sources.get_source_mask(stationary=True)
```

(continues on next page)

(continued from previous page)

```
    # create a pie chart
    axes[i].pie([len(stat_mask[stat_mask]), len(stat_mask[np.logical_not(stat_mask)])],␣
↪labels=labels,
                colors=[plt.get_cmap(col)(0.5), plt.get_cmap(col)(0.7)])

    # write what the tolerance was
    axes[i].set_xlabel("Stationary if "\
                       + r"$\frac{\Delta f_{\rm orb}}{f_{\rm orb}} <$"\
                       + "{:.3f}%".format(tol * 100))

plt.show()
```



$g(n, e)$ Interpolation

Another thing to consider when creating a new source class is whether to interpolate the $g(n, e)$ function from Peters (1963). This function is a complex combination of Bessel functions and thus is slow to compute. We therefore add the option to load in pre-computed $g(n, e)$ values and interpolate them instead of computing the values directly. We perform this interpolation once upon class creation and use it throughout after this. The pre-computed values span 1000 values of eccentricity and 10000 harmonics and thus are accurate for $e < 0.995$.

We can illustrate the difference in speed and results for the interpolated function vs. the real one.

[10]:
```
%%timeit
e_range = np.random.uniform(0.0, 0.995, 1000)
n_range = np.arange(1, 150 + 1).astype(int)

# the argsorts here unsort the output since interp2d automatically sorts it
sources.g(n_range, e_range)[np.argsort(e_range).argsort()]
```

```
2.24 ms ± 324 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[11]:
```
%%timeit
e_range = np.random.uniform(0.0, 0.995, 1000)
n_range = np.arange(1, 150 + 1).astype(int)

E, N = np.meshgrid(e_range, n_range)

utils.peters_g(N, E)
```

```
1.07 s ± 41.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

You can see here that the interpolated function is **roughly 500 times faster**. Now let's ensure that they are giving the same results.

[12]:
```
e_range = np.random.uniform(0.0, 0.9, 10)
n_range = np.arange(1, 150 + 1).astype(int)

g_interpolated = sources.g(n_range, e_range)[np.argsort(e_range).argsort()]

N, E = np.meshgrid(n_range, e_range)

g_true = utils.peters_g(N, E)
```

[13]:
```
difference = np.abs(g_true - g_interpolated)
print("The largest difference between the interpolated and true value is {:.2e}".
↪format(np.max(difference)))
print("Whilst the average difference is {:.2e}".format(np.mean(difference)))
```

```
The largest difference between the interpolated and true value is 1.60e-09
Whilst the average difference is 6.13e-11
```

So the functions also give pretty much the same values. However, if you happen to have a lot of computing power and time is not a factor for you then setting `interpolate_g=False` will give you more accurate results.

You should also consider that if you only have a couple of sources then the time saved by avoiding computing $g(n, e)$ would actually be less than the time it takes to interpolate. So for small populations of sources we recommend testing the runtime with `interpolate_g=False` to check whether it is faster.

Sensitivity Curve Interpolation

Finally, similar to the previous section, we offer the ability to interpolate the sensitivity curve. The only difference is you can also pass `sc_params` to update the arguments that are passed to the sensitivity curve function. This provides increased speed for large samples of sources, timesteps and harmonics but will make little difference for smaller collections of sources.

By default the SNR function (see below) will use the parameters in `sc_params` (e.g. the mission length or confusion noise model) when computing SNRs.

2.2.2 Strain and SNR calculations

Strain functions

The source class provides a convenient wrapper around the functions from the strain module and allows you to compute either the strain or characteristic strain for any number of harmonics and any subset of the sources.

Let's try this out with the same source class instance from earlier.

[14]:
```
# compute h_c_n for every source for the first four harmonics
h_c_n4 = sources.get_h_c_n(harmonics=[1, 2, 3, 4])
print(h_c_n4)
```

```
[[2.46078236e-17 1.42420838e-16 1.07136176e-16 5.88129908e-17]
 [1.06975755e-17 2.06768406e-16 5.27828885e-17 1.00156266e-17]
 [7.47078867e-17 3.89440357e-16 3.17309365e-16 1.87919671e-16]
 ...
 [1.74229531e-17 9.51883434e-17 7.48556894e-17 4.28667975e-17]
 [1.67406319e-17 5.09680121e-18 9.66806812e-18 1.85106547e-17]
 [2.67122069e-17 1.88443050e-16 1.20904130e-16 5.69491486e-17]]
```

Now imagine an (admittedly rather strange) scenario in which you only want to compute the strain for every other source. This is pretty easy to do!

[15]:
```
every_other = np.array([i % 2 == 0 for i in range(sources.n_sources)])

h_0_n = sources.get_h_0_n(harmonics=[1, 2, 3, 4], which_sources=every_other)
print("The shape of this strain array is {}".format(h_0_n.shape))
print("(since we compute the first 4 harmonics and only for every other source)")
```

```
The shape of this strain array is (750, 4)
(since we compute the first 4 harmonics and only for every other source)
```

This may seem a little contrived but we could also use this to isolate the circular sources and compute them separately (since then we needn't bother with any harmonic except $n = 2$).

[16]:
```
circular = sources.get_source_mask(circular=True)

h_c_n = sources.get_h_0_n(harmonics=2, which_sources=circular)
print("The shape of this characteristic strain array is {}".format(h_c_n.shape)),
print("(since we compute only the n=2 harmonic and only for circular sources)")
```

```
The shape of this characteristic strain array is (282, 1)
(since we compute only the n=2 harmonic and only for circular sources)
```

Signal-to-Noise Ratio

The `snr` module of `LEGWORK` contains four functions for calculating the SNR depending on whether a binary is stationary/evolving and circular/eccentric using various approximations. However, the source class takes cares of sending the right binaries to the right functions and all you have to do is set the `gw_lum_tol` and `stat_tol` when instantiating the class (see above) and then call *`legwork.source.Source.get_snr()`*.

This splits the sources into (up to) 4 subsets and calculates their SNRs before recollecting them and storing the result in `source.snr`.

Let's try this out. Note that we run with `verbose=True` here so that you can see the size of subpopulation.

[17]:
```
snr_4 = sources.get_snr(verbose=True)
```

```
Calculating SNR for 1500 sources
        0 sources have already merged
        1389 sources are stationary
                268 sources are stationary and circular
                1121 sources are stationary and eccentric
        111 sources are evolving
                14 sources are evolving and circular
                97 sources are evolving and eccentric
```

We can also adjust the length of the LISA mission to see how this affects the SNR. It is also important in general to update the sensitivity curve parameters for the interpolation so that the interpolated sensitivity curve matches the updated mission length. LEGWORK handles this for `get_snr` as long as `reinterpolate_sc=True`.

[18]:
```
snr_10 = sources.get_snr(t_obs=10 * u.yr, verbose=True)
```

```
Calculating SNR for 1500 sources
        0 sources have already merged
        1360 sources are stationary
                263 sources are stationary and circular
                1097 sources are stationary and eccentric
        140 sources are evolving
                19 sources are evolving and circular
                121 sources are evolving and eccentric
```

Note that you can see that the number of stationary binaries has decreased slightly since some binaries may just be on the cusp of no longer being stationary and extending the time means they change frequency enough to be labelled as evolving. It could be interesting to do this to see how the number of detectable binaries changes.

[19]:
```
n_detect_4 = len(snr_4[snr_4 > 7])
n_detect_10 = len(snr_10[snr_10 > 7])
print("{} binaries are detectable over 4 years".format(n_detect_4))
print("Whilst extending to a 10 year mission gives {}".format(n_detect_10))
```

```
621 binaries are detectable over 4 years
Whilst extending to a 10 year mission gives 621
```

Position-inclination-polarisation specfic sources

For some sources, you may already know the positions and this means that you can use a more specific SNR calculation (see the derivations for more details) rather than an average. Note that since this SNR calculation also considers frequency spreading due to doppler modulation from the detector orbit, the SNR will always be lower than in the fully averaged case.

We can input this when instantiating the source. For positions we use the Skycoord Class from Astropy, which allows you to specific the coordinates in any frame and it will automatically convert it to the necessary coordinates in LEGWORK.

[20]:
```
# redefine f_orb so all binaries are stationary (can't use position-specific snr for␣
↪evolving sources)
specific_f_orb = 10**np.random.uniform(-5, -4, len(m_1)) * u.Hz
sources_specific = source.Source(m_1=m_1, m_2=m_2, f_orb=specific_f_orb, dist=dist,␣
↪ecc=np.repeat(0.0, len(m_1)),
```

(continues on next page)

(continued from previous page)

```
                                  position=SkyCoord(lon=np.random.uniform(0, 2 * np.pi) *␣
↪u.rad,
                                                    lat=np.random.uniform(-np.pi / 2, np.
↪pi / 2) * u.rad,
                                                    distance=dist, frame=
↪"heliocentrictrueecliptic"),
                                  inclination=np.arcsin(np.random.uniform(-1, 1, len(m_
↪1))) * u.rad,
                                  polarisation=np.random.uniform(0, 2 * np.pi, len(m_1))␣
↪* u.rad)
sources_specific.get_snr()
```

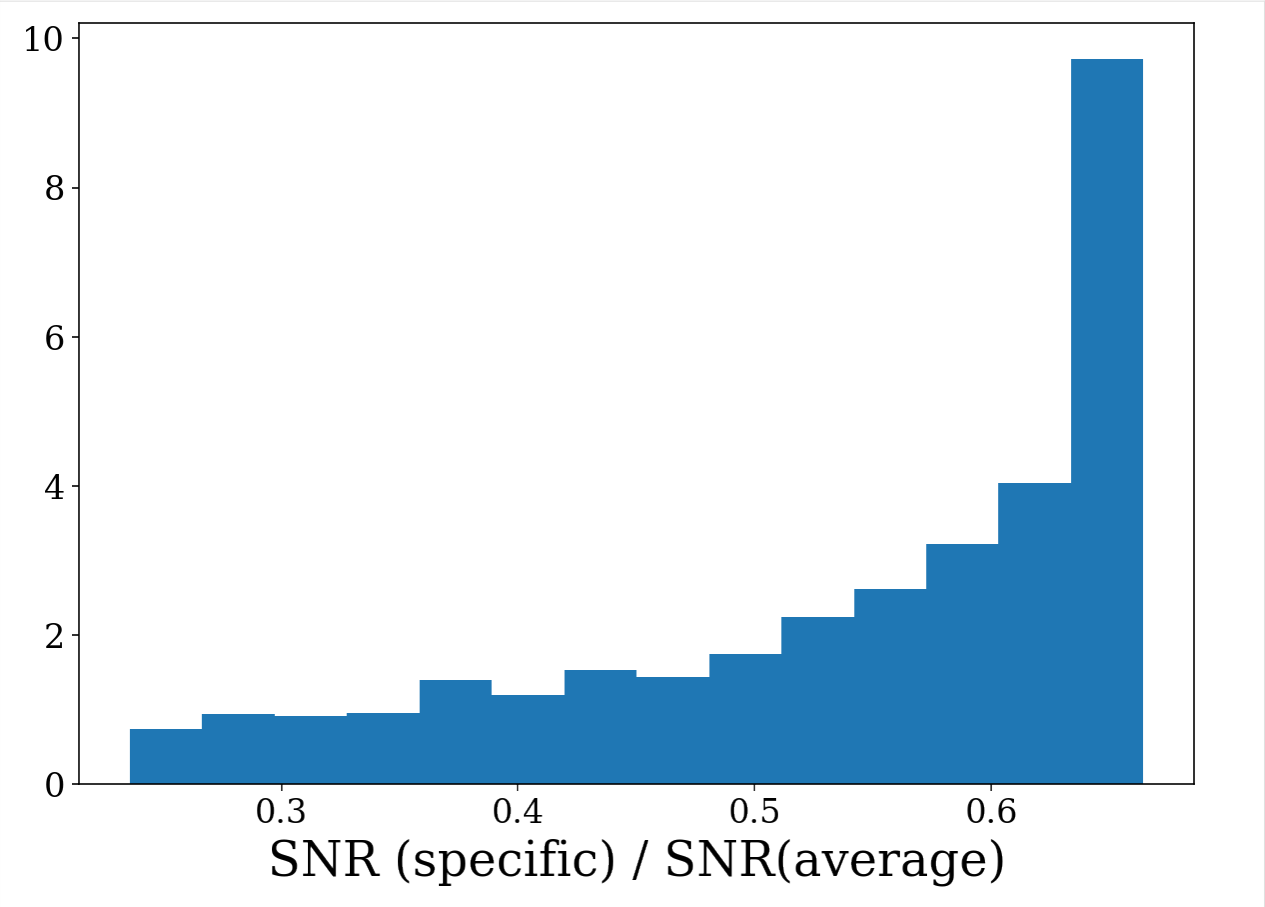
[20]:
```
array([2.36226570e-01, 4.84705414e-03, 4.17744265e-01, ...,
       1.15586765e-04, 1.73422646e-02, 1.40722564e-01])
```

[21]:
```
sources_average = source.Source(m_1=m_1, m_2=m_2, f_orb=specific_f_orb,
                                dist=dist, ecc=np.repeat(0.0, len(m_1)))
sources_average.get_snr()
```

[21]:
```
array([8.82139455e-01, 1.06499265e-02, 8.97087569e-01, ...,
       2.64335535e-04, 2.68516364e-02, 3.72650765e-01])
```

Let's plot the ratio of the SNR for the two cases using the visualisation module.

[22]:
```
snr_ratio = sources_specific.snr / sources_average.snr
fig, ax = visualisation.plot_1D_dist(snr_ratio, bins="fd", xlabel="SNR (specific) /␣
↪SNR(average)")
```



As you can see, the SNR for a specific source is always lower than for the average source. This is because the modulation reduces the strain amplitude as the smearing in frequency reduces the amount of signal build up at the true source frequency.

2.2.3 Visualisation

Although the visualisation model gives more freedom in honing various aspects of your plots, for general analysis the source class has two functions to quickly create plots to investigate distributions.

Parameters Distributions

The first function is *legwork.source.Source.plot_source_variables()* which can create either 1D or 2D distributions of any subpopulation of sources.

Let's take it for a spin with a collection of stationary binaries.

[23]:
```
# create a random collection of sources
n_values = 15000
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.normal(8, 1.5, n_values) * u.kpc
f_orb = 10**(np.random.normal(-5, 0.5, n_values)) * u.Hz
ecc = 1 - np.random.power(3, n_values)
```

(continues on next page)

(continued from previous page)

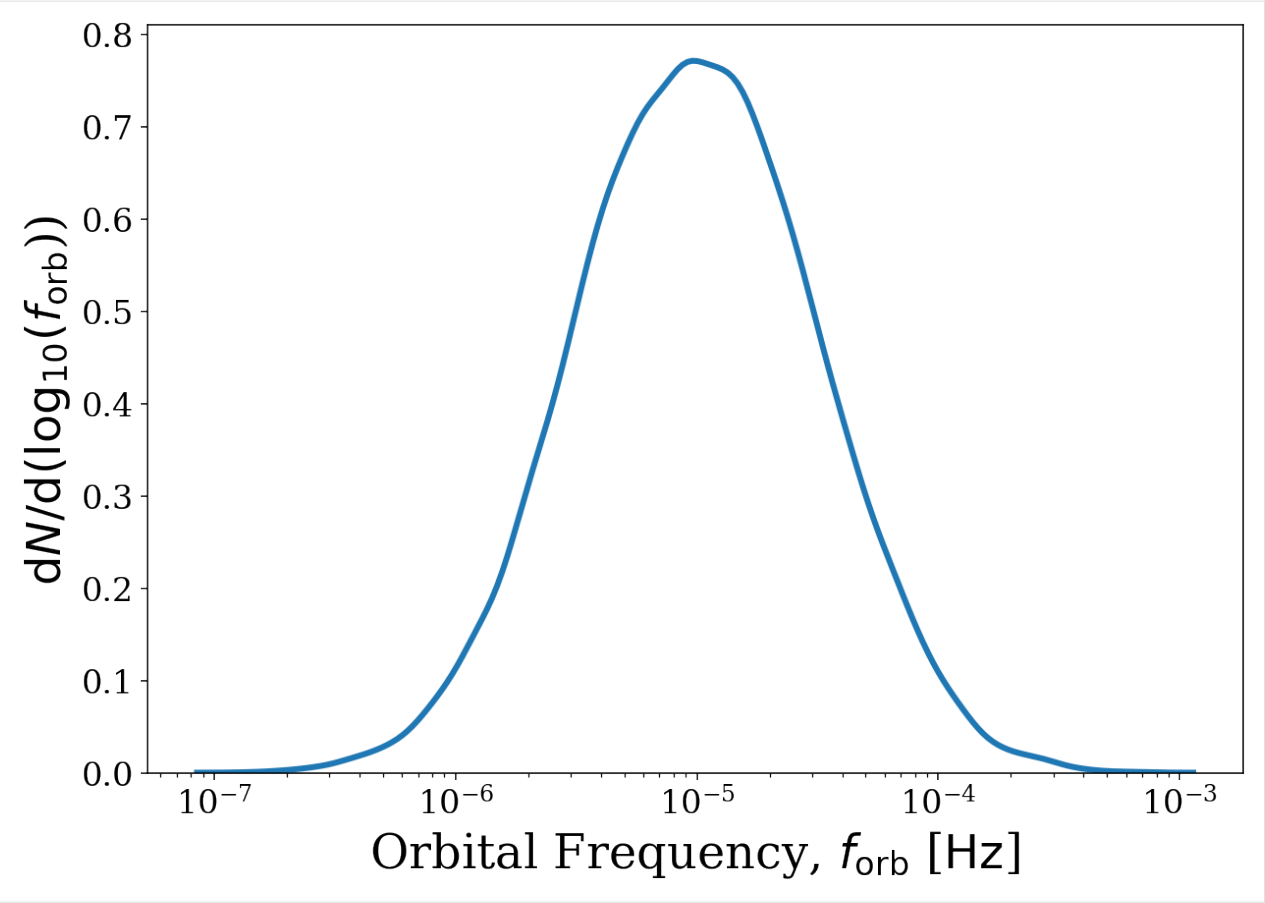
```
sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb)
```

This function will let you plot any of several parameters (listed in the table below) and work out the units for the axes labels automatically based on the values in the source class.

Parameter	Label
Primary Mass	m_1
Secondary Mass	m_2
Chirp Mass	m_c
Eccentricity	ecc
Distance	dist
Orbital Frequency	f_orb
Gravitational Wave Frequency	f_GW
Semi-major Axis	a
Signal-to-noise Ratio	snr

We can start simple and just plot the orbital frequency distribution for all sources. For a 1D distribution we only need to use `xstr` and can leave `ystr` as `None`. The various keyword arguments are passed to *`legwork.visualisation.plot_1D_dist()`* and *`legwork.visualisation.plot_2D_dist()`*, for more info check out the *Visualisation* tutorial!

[24]:
```
fig, ax = sources.plot_source_variables(xstr="f_orb", disttype="kde", log_scale=True,␣
↪linewidth=3)
```



But we could also try to see how the detectable population is different from the entire population. Let's create two frequency KDEs, one for the detectable binaries and another for all of them. For this we will use the `which_sources` parameter and pass a mask on the SNR.

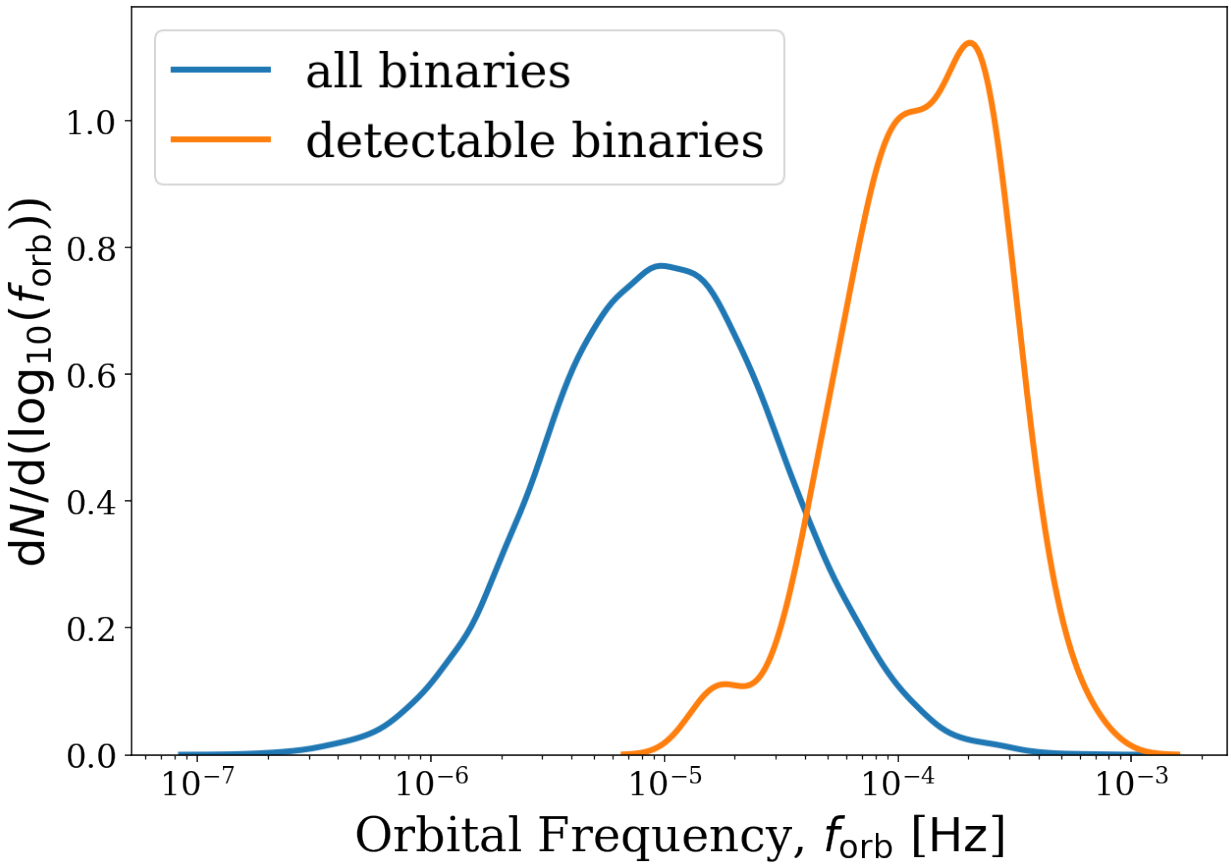
[25]:
```
# calculate the SNR
snr = sources.get_snr(verbose=True)

# mask detectable binaries
detectable = snr > 7

# plot all binaries
fig, ax = sources.plot_source_variables(xstr="f_orb", disttype="kde", log_scale=True,␣
↪linewidth=3,
                                        show=False, label="all binaries")

# plot all binaries
fig, ax = sources.plot_source_variables(xstr="f_orb", disttype="kde", log_scale=True,␣
↪linewidth=3, fig=fig,
                                        ax=ax, which_sources=detectable, label=
↪"detectable binaries",
                                        show=False)

ax.legend()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

```
Calculating SNR for 15000 sources
        0 sources have already merged
        15000 sources are stationary
                2819 sources are stationary and circular
                12181 sources are stationary and eccentric
```



Here's we can see that the distribution is shifted to higher frequencies for detectable binaries which makes sense since these are easier to detect.

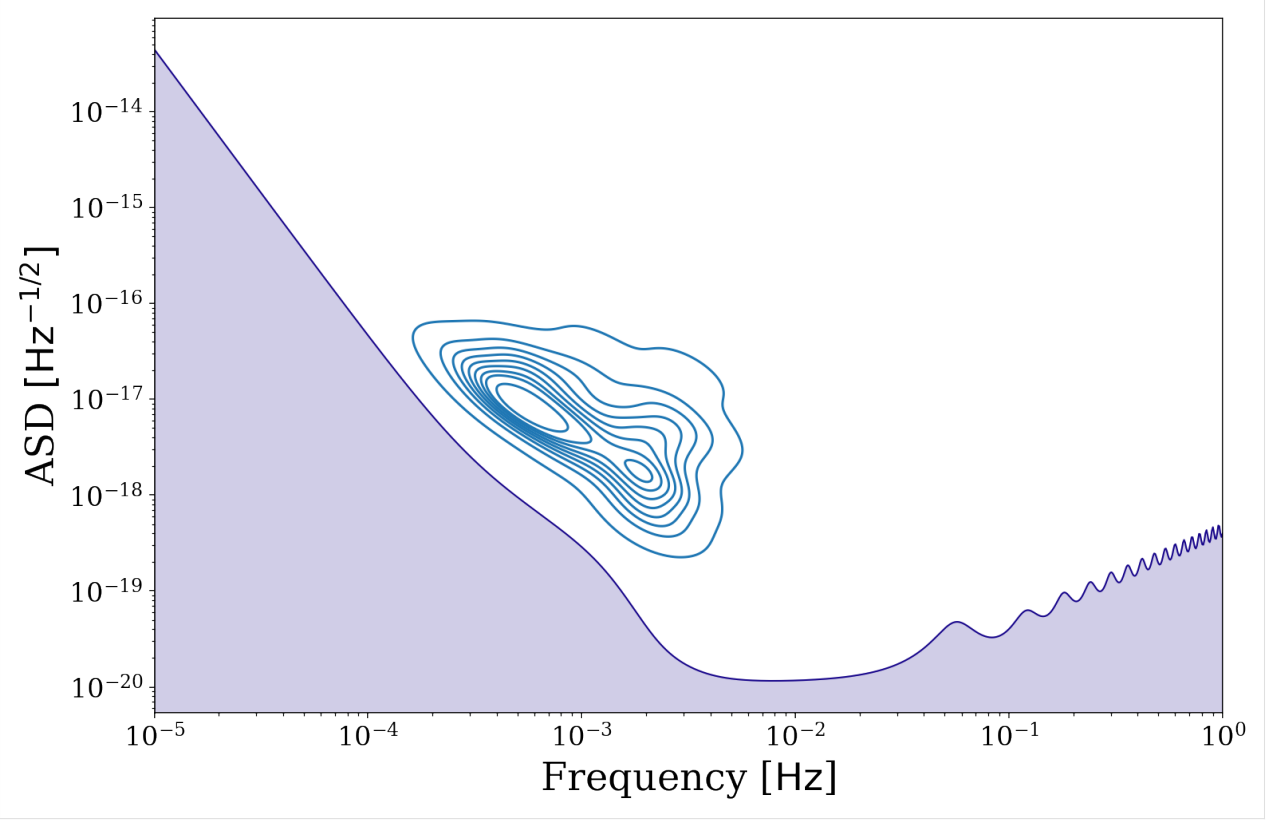
Sources on the Sensitivity Curve

The other function is `legwork.Source.source.plot_sources_on_sc()` which plots all sources on the sensitivity curve.

Note: This is currently only implemented for stationary binaries. We are working on adding the same functionality for evolving binaries.

Here we set `snr_cutoff=7` so that only binaries with SNR > 7 are plotted and switch the `disttype` to a KDE density plot. The circular binaries are plotted in blue and the eccentric binaries in orange.

[26]:
```
fig, ax = sources.plot_sources_on_sc(snr_cutoff=7, disttype="kde")
```



That's all for this tutorial, be sure to check out *the other ones* to find *other* ways to keep your feet up and let us do the LEGWORK!

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.3 Strains

In this tutorial, we explain how to use the functions in the strain module. We also take a look at the relevance of higher harmonics and how to use the output.

Let's import the strain functions and also some other common stuff.

[2]:
```
import legwork.strain as strain

import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

2.3.1 Introduction to function usage

We will focus mainly on the strain amplitude function, *legwork.strain.h_0_n()* but *legwork.strain.h_c_n()* works identically except it computes the *characteristic* strain amplitude rather than the plain old strain amplitude.

It is important to keep in mind that the shape of the output of these functions will always be `(number of sources, number of timesteps, number of harmonics)`. Therefore, even if you only supply a single source, with a single timestep and ask for a single harmonic then the function will still supply a 3D array of shape `(1, 1, 1)`.

[4]:
```
# example with single source
h_0_2 = strain.h_0_n(m_c=10 * u.Msun, f_orb=1e-3 * u.Hz, ecc=0.1, n=2, dist=8 * u.kpc)
print("The strain array that is returned is {} which has a shape of {}".format(h_0_2, h_
↪0_2.shape))
```

```
The strain array that is returned is [[[9.5482392e-21]]] which has a shape of (1, 1, 1)
```

Let's repeat that exercise but for many sources and many harmonics just to emphasise what the shape looks like. First we can simply create 100 random sources using numpy.

[5]:
```
n_sources = 100
m_c = np.random.uniform(0, 10, n_sources) * u.Msun
dist = np.random.uniform(0, 8, n_sources) * u.kpc
f_orb_i = 10**(np.random.uniform(-5, 0, n_sources)) * u.Hz
```

Now let's say that we want to know the strain of each source at linearly spaced timesteps between the initial frequency and twice the initial frequency. In this case let's do 5 timesteps and so we need to define the eccentricity and frequency at these times (let's just do circular sources for simplicity).

[6]:
```
n_times = 5
f_orb = np.array([np.linspace(f.value, f.value * 2, n_times) for f in f_orb_i]) * u.Hz
ecc = np.random.uniform(0, 1, (n_sources, n_times))
```

And perhaps we only care for the first 42 harmonics, so let's define that quickly.

[7]:
```
n_harmonics = 42
harmonics = np.arange(1, n_harmonics + 1).astype(int)
```

Then all we need to do to calculate the strain is this

[8]:
```
# calculate the strain
h_0_n = strain.h_0_n(m_c=m_c, f_orb=f_orb, ecc=ecc, n=harmonics, dist=dist)
print("The shape of this array is {}".format(h_0_n.shape))
print("This makes sense since n_sources is {}, n_times is {}".format(n_sources, n_times),
      "and n_harmonics is {}".format(n_harmonics))
```

```
The shape of this array is (100, 5, 42)
This makes sense since n_sources is 100, n_times is 5 and n_harmonics is 42
```

Then we can access the strain of a particular source and harmonics by using Python slices/indices like so

[9]:
```
ind_source = np.random.choice(np.arange(n_sources).astype(int))
ind_times = np.random.choice(np.arange(n_times).astype(int))
ind_harmonic = np.random.choice(np.arange(n_harmonics).astype(int))

chosen_strain = h_0_n[ind_source, ind_times, ind_harmonic]
```

(continues on next page)

(continued from previous page)

```
print("The strain of source {} at timestep {} and at n = {}".format(ind_source, ind_
↪times, ind_harmonic + 1),
      "is {:1.2e}".format(chosen_strain))
```

```
The strain of source 51 at timestep 4 and at n = 33 is 7.77e-20
```

Additionally, we may only be interested in the *total* strain for each source at each timestep rather than keeping track of the strain in each harmonic. In this case we simply need to sum over the correct axis to get the total.

[10]:
```
h_0 = h_0_n.sum(axis=2)
print("The number of sources is {} and the number of times is {},".format(n_sources, n_
↪times),
      "hence the shape of `h_0` is {}".format(h_0.shape))
```

```
The number of sources is 100 and the number of times is 5, hence the shape of `h_0` is␣
↪(100, 5)
```

2.3.2 Some example uses

Now that we've covered how the functions work, let's explore some examples of how we can use them!

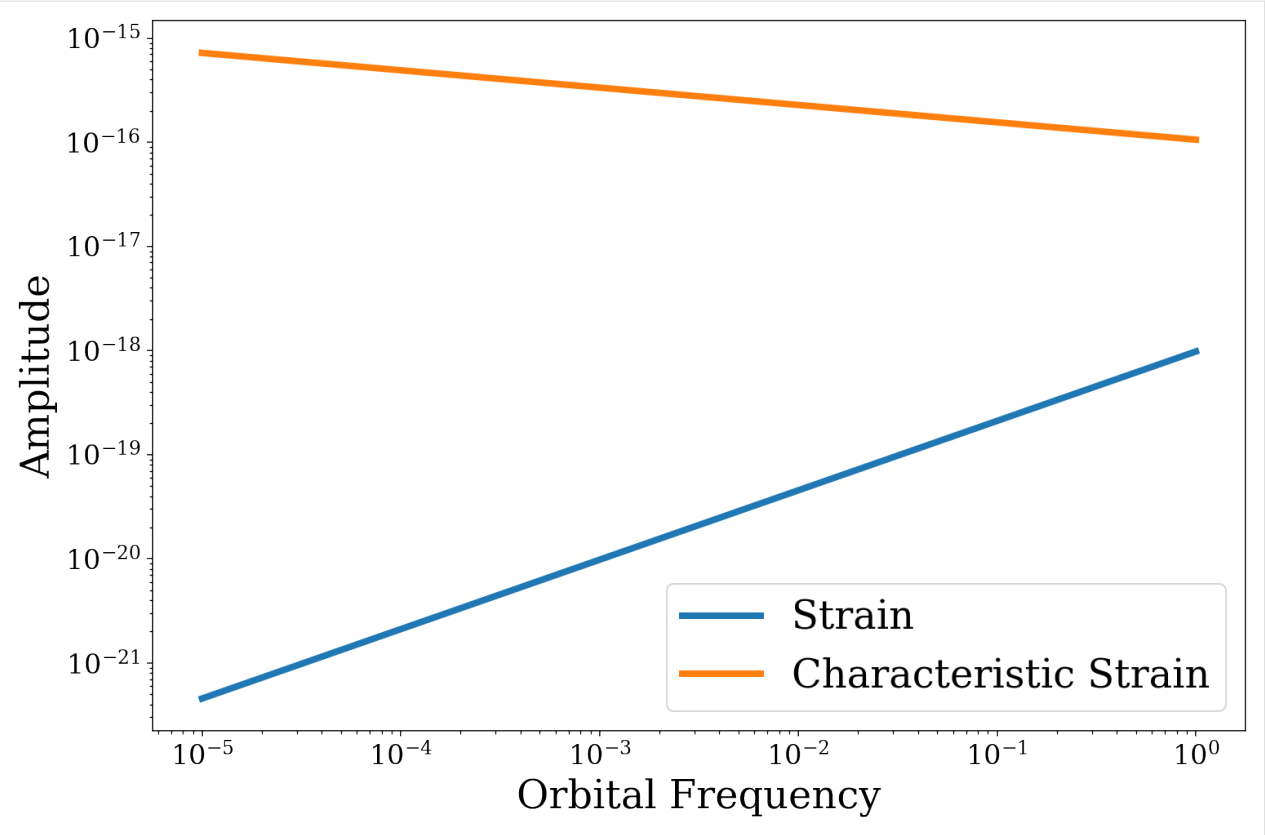
Circular binaries: effect of mass

For circular binaries only the $n = 2$ harmonic has any signal and hence we can plot how the strains change with orbital frequency (assuming a fixed distance and chirp mass)

[11]:
```
# pick some fixed values and a range for frequency
m_c = 10 * u.Msun
f_orb = np.logspace(-5, 0, 100) * u.Hz
dist = 8 * u.kpc
ecc = 0.0
```

[12]:
```
# compute second harmonic for each and flatten to reduce to 1D array
h_0_2 = strain.h_0_n(m_c=m_c, f_orb=f_orb, ecc=ecc, n=2, dist=dist).flatten()
h_c_2 = strain.h_c_n(m_c=m_c, f_orb=f_orb, ecc=ecc, n=2, dist=dist).flatten()
```

[13]:
```
# add the two lines
plt.loglog(f_orb.value, h_0_2, lw=4, label="Strain")
plt.loglog(f_orb.value, h_c_2, lw=4, label="Characteristic Strain")

# label the axes
plt.xlabel("Orbital Frequency")
plt.ylabel("Amplitude")

# show a legend and the plot
plt.legend()
plt.show()
```



So you can see that the strain increases with frequency whilst the characteristic strain actually *decreases*.

Eccentric binaries: harmonic distribution

It's also interesting to look at how the strain is spread over multiple harmonics as we increase eccentricity. Let's take a look at how the strength of the $n = 2, 3, 4$ harmonics change with eccentricity (with fixed frequency, chirp mass and distance).

[14]:
```
# range for eccentricities (limited to 0.9 to reduce number harmonics needed)
ecc = np.linspace(0.01, 0.9, 1000)

# fixed values
m_c = np.ones_like(ecc) * 10 * u.Msun
f_orb = np.ones_like(ecc) * 1e-3 * u.Hz
dist = np.ones_like(ecc) * 8 * u.kpc
```

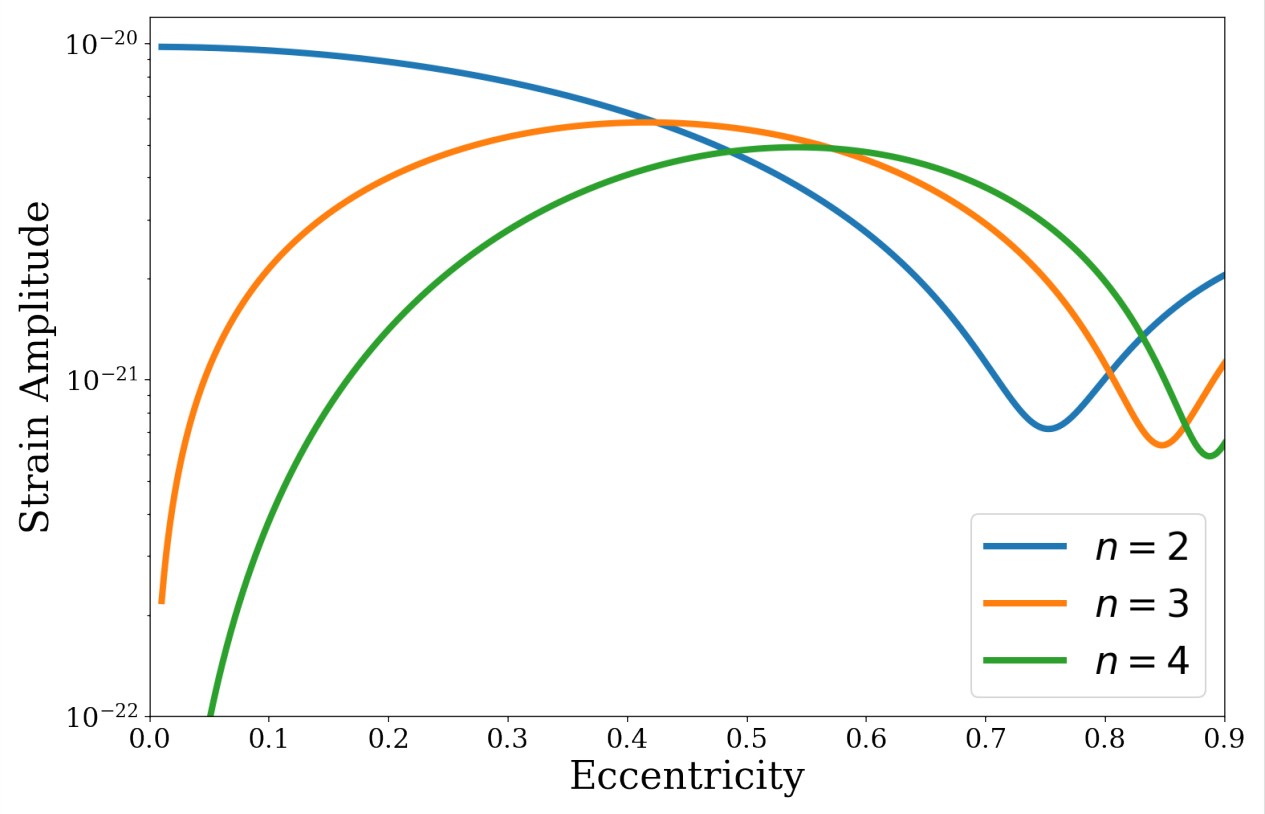
[15]:
```
# calculate the strain for the first 500 harmonics
ecc_plot_h_0_n = strain.h_0_n(m_c=m_c, f_orb=f_orb, ecc=ecc, n=np.arange(1, 500 + 1).
↪astype(int), dist=dist)
```

[16]:
```
# add lines for the n = 2,3,4 harmonics
for n in [2, 3, 4]:
    plt.plot(ecc, ecc_plot_h_0_n[:, 0, n - 1], label=r"$n={{{}}}$".format(n), lw=4)

# label the axes
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Eccentricity")
plt.ylabel("Strain Amplitude")

# use a log scale for strain
plt.yscale("log")

# limit axes
plt.xlim(0, 0.9)
plt.ylim(1e-22, 1.2e-20)

# show a legend and the plot
plt.legend()
plt.show()
```



So here we can see that the strength of the $n = 2$ harmonic declines as binaries become more eccentric, such that the $n = 3$ harmonic actually becomes stronger than the $n = 2$ harmonic around $e \approx 0.33$. This repeats for $n = 4$ and onwards as each harmonic dominates at higher eccentricities.

But what about the total strain? This is just single harmonics, let's look at how the total varies too!

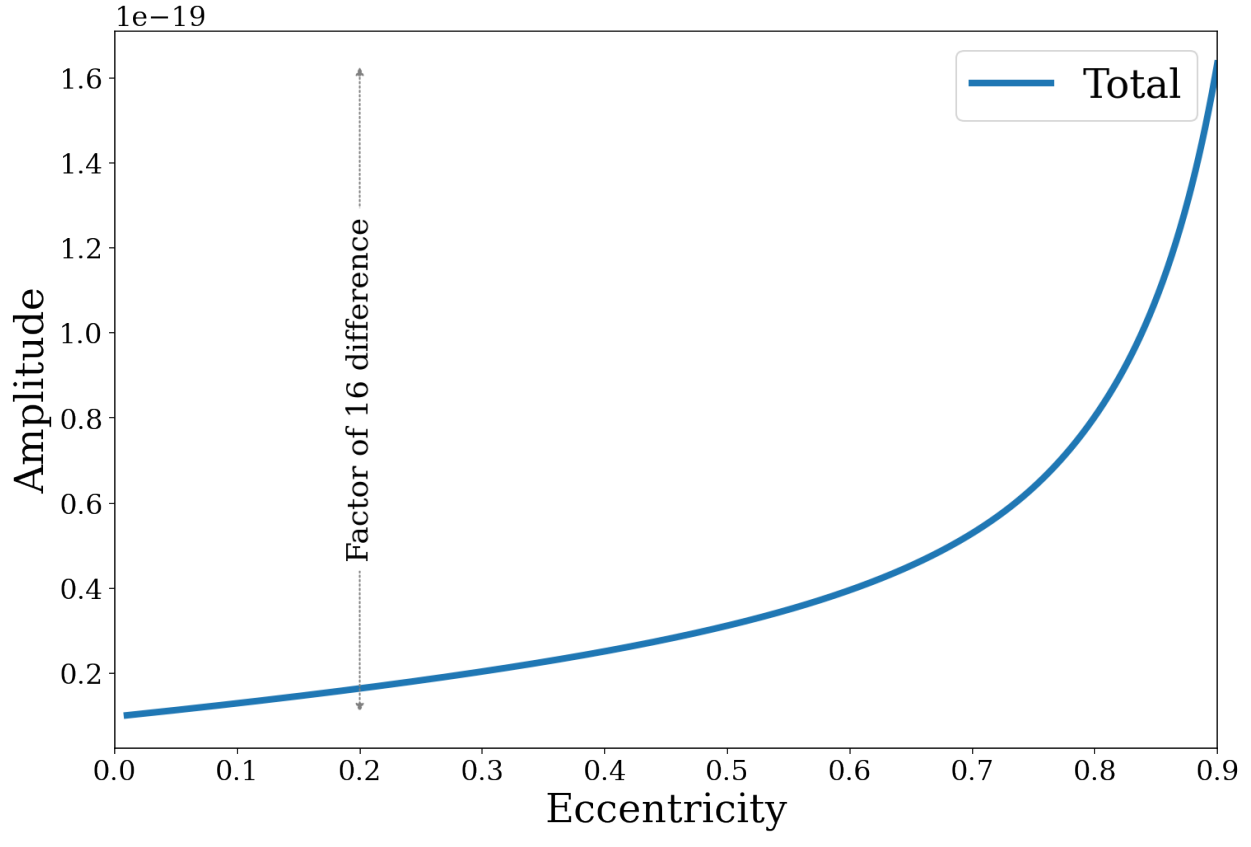
[17]:
```
# sum the strain over all harmonics for each binary
h_0 = ecc_plot_h_0_n.sum(axis=2).flatten()
```

[18]:
```
fig, ax = plt.subplots()

# plot the total
ax.plot(ecc, h_0, label="Total", lw=4)
```

(continues on next page)

(continued from previous page)

```
# add an arrow and label with the change in strain
e_label = 0.2
ax.annotate("", xy=(e_label, h_0.min()), xytext=(e_label, h_0.max()),
            arrowprops=dict(arrowstyle="<|-|>", linestyle="dotted", ec="grey", fc="grey
↪"))
ax.annotate("Factor of {:1.0f} difference".format(h_0.max() / h_0.min()),
            xy=(e_label, h_0.min() + (h_0.max() - h_0.min()) / 2), rotation=90,
            ha="center", va="center", bbox=dict(boxstyle="round", fc="white", ec="none"),
↪ fontsize=18)

# label the axes
ax.set_xlabel("Eccentricity")
ax.set_ylabel("Amplitude")

# show a legend and the plot
ax.legend()
ax.set_xlim(0, 0.9)

plt.show()
```



Here we can see that higher eccentricity can dramatically increase the strain. Note that this may not necessarily increase the *detectability* of the binaries however, since the strain will be shifted to higher frequencies and hence *may* occur over a more noisy part of the LISA sensitivity curve.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.4 Binary evolution

In this tutorial, we take a look at how to evolve binaries with LEGWORK and get the most out of the functions in the `evol` module.

Let's import the evolution functions and also some other common stuff.

[2]:
```
from legwork import evol, utils

import numpy as np
import astropy.units as u
from astropy.visualization import quantity_support
quantity_support()
import matplotlib.pyplot as plt
```

2.4.1 Calculating merger times

When it comes to gravitational wave sources, one of the first questions that comes to mind is: when is this thing going to merge? Well that's exactly the question that *legwork.evol.get_t_merge_circ()* and *legwork.evol.get_t_merge_ecc()* can answer.

We'll focus primarily on the eccentric function since the circular function is just a special case and works exactly the same way.

Flexible input

This function requires the eccentricity as well as information about the masses and separation of the binary. This is done flexibly such that:

- you can *either* enter the individual masses, `m_1, m_2`, *or* the `beta` constant from Peters (1964)
- you can *either* enter the semi-major axis `a_i` *or* the orbital frequency `f_orb_i`

In these cases `beta` and `a_i` take precedence.

[4]:
```
# pick some values
ecc_i = 0.1
m_1 = 7 * u.Msun
m_2 = 42 * u.Msun
f_orb_i = 1e-3 * u.Hz

# calculate other params
beta = utils.beta(m_1=m_1, m_2=m_2)
a_i = utils.get_a_from_f_orb(f_orb=f_orb_i, m_1=m_1, m_2=m_2)

# compute with individual masses and frequency
```

(continues on next page)

(continued from previous page)

```
t_merge = evol.get_t_merge_ecc(ecc_i=ecc_i, m_1=m_1, m_2=m_2, f_orb_i=f_orb_i)

# compute with beta and semi-major axis
t_merge_alt = evol.get_t_merge_ecc(ecc_i=ecc_i, beta=beta, a_i=a_i)

# check you get the same result
print(t_merge == t_merge_alt)
```

```
True
```

Approximations

These functions are based directly on the equations from Peters (1964) and these consist of four cases which each use a different equation from Peters:

Case	Peters (1964) Equation	EccentricityRange
Exactly Circular	Eq. 5.10	e = 0.0
Small Eccentricity Approximation	Eq. after 5.14	e < `small_e_tol`
General Case	Eq. 5.14	`small_e_tol` <= e <= `large_e_tol`
Large Eccentricity Approximation	2nd Eq. after 5.14	`large_e_tol` < e

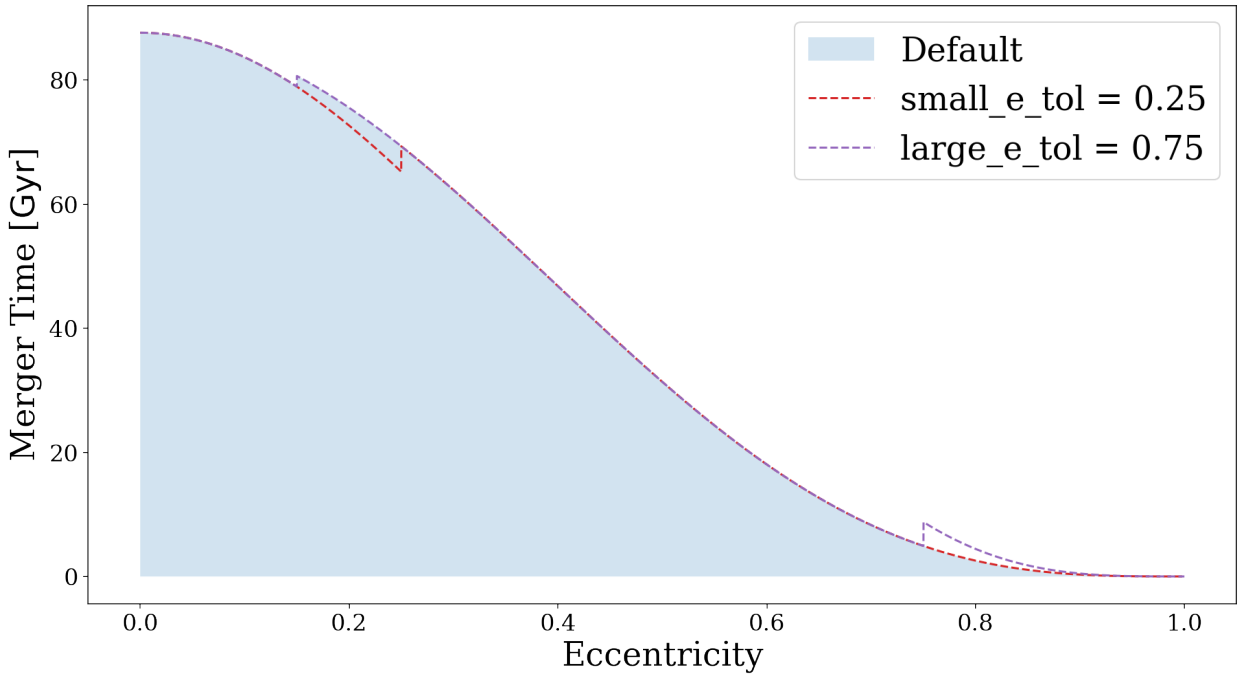
The ranges in which the different functions are used are determined by `small_e_tol` and `large_e_tol` which are `0.01` and `0.99` by default. Let's take a look at how changing these can affect our results.

[6]:
```
# create random binaries
e_range = np.linspace(0.0, 0.9999, 10000)
m_1 = np.repeat(1, len(e_range)) * u.Msun
m_2 = np.repeat(1, len(e_range)) * u.Msun
f_orb_i = np.repeat(1e-5, len(e_range)) * u.Hz

# calculate merger time using defaults
t_merge = evol.get_t_merge_ecc(ecc_i=e_range, m_1=m_1, m_2=m_2, f_orb_i=f_orb_i)

# calculate two examples with different tolerances
t_merge_small_ex = evol.get_t_merge_ecc(ecc_i=e_range, m_1=m_1, m_2=m_2, f_orb_i=f_orb_i,
↪ small_e_tol=0.25)
t_merge_large_ex = evol.get_t_merge_ecc(ecc_i=e_range, m_1=m_1, m_2=m_2, f_orb_i=f_orb_i,
↪ large_e_tol=0.75)

# create a figure
fig, ax = plt.subplots(figsize=(15, 8))

# plot the default as an area
ax.fill_between(e_range, np.zeros_like(t_merge), t_merge, label="Default", alpha=0.2)

# plot the examples with dashed lines
ax.plot(e_range, t_merge_small_ex, label=r"small_e_tol = 0.25", color="tab:red",␣
↪linestyle="--")
ax.plot(e_range, t_merge_large_ex, label=r"large_e_tol = 0.75", color="tab:purple",␣
↪linestyle="--")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Eccentricity")
ax.set_ylabel("Merger Time [{0:latex}]".format(t_merge.unit))

ax.legend()

plt.show()
```



Here we can see how adjusting the tolerances can result in drastic changes to the merger times. We leave it to you to determine your balance of runtime and accuracy!

2.4.2 Evolving binaries

Sometimes you need more information than simply "when will it merge" and may instead want to know how various binary parameters evolve over time. This is where *legwork.evol.evol_circ()* and *legwork.evol.evol_ecc()* can be used to find the evolution of the binary separation, frequency and eccentricity over time.

Flexible output

The same principles as *above* (with the flexible input options of the masses and separation) apply. In addition, we make the *output* flexible here too. When evolving a binary, the final evolution that is returned can contain any of:

- `a_evol` = **"a"**: semi-major axis evolution
- `f_orb_evol` = **"f_orb"**: orbital frequency evolution
- `f_GW_evol` = **"f_GW"**: gravitational wave frequency evolution
- `timesteps` = **"timesteps"**: timesteps corresponding to values in evolutions
- `ecc_evol` = **"ecc"**: eccentricity evolution (`evol_ecc()` only)

You can choose which variables you want to be outputted and their order by including the strings listed above (after the equals signs) in `output_vars`. Let's try this in practice.

[7]:
```
m_1 = 7 * u.Msun
m_2 = 42 * u.Msun
f_orb_i = 1e-3 * u.Hz

# we could just ask for a_evol
a_evol = evol.evol_circ(m_1=m_1, m_2=m_2, f_orb_i=f_orb_i, output_vars="a")

# or timesteps as well
a_evol, timesteps = evol.evol_circ(m_1=m_1, m_2=m_2, f_orb_i=f_orb_i, output_vars=["a",
↪"timesteps"])
```

Timesteps

LEGWORK also allows you to determine the number and spacing of the timesteps to report about the evolution.

There are three relevant parameters for this:

- `t_evol`: the amount of time to evolve each binary for (this will default to the merger time)
- `n_step`: the number of **linearly spaced** steps to take between 0 and `t_evol`
- `timesteps`: an exact custom array of timesteps to use in place of the above parameters

Lets try evolving a binary until its merger and adjusting the number of timesteps

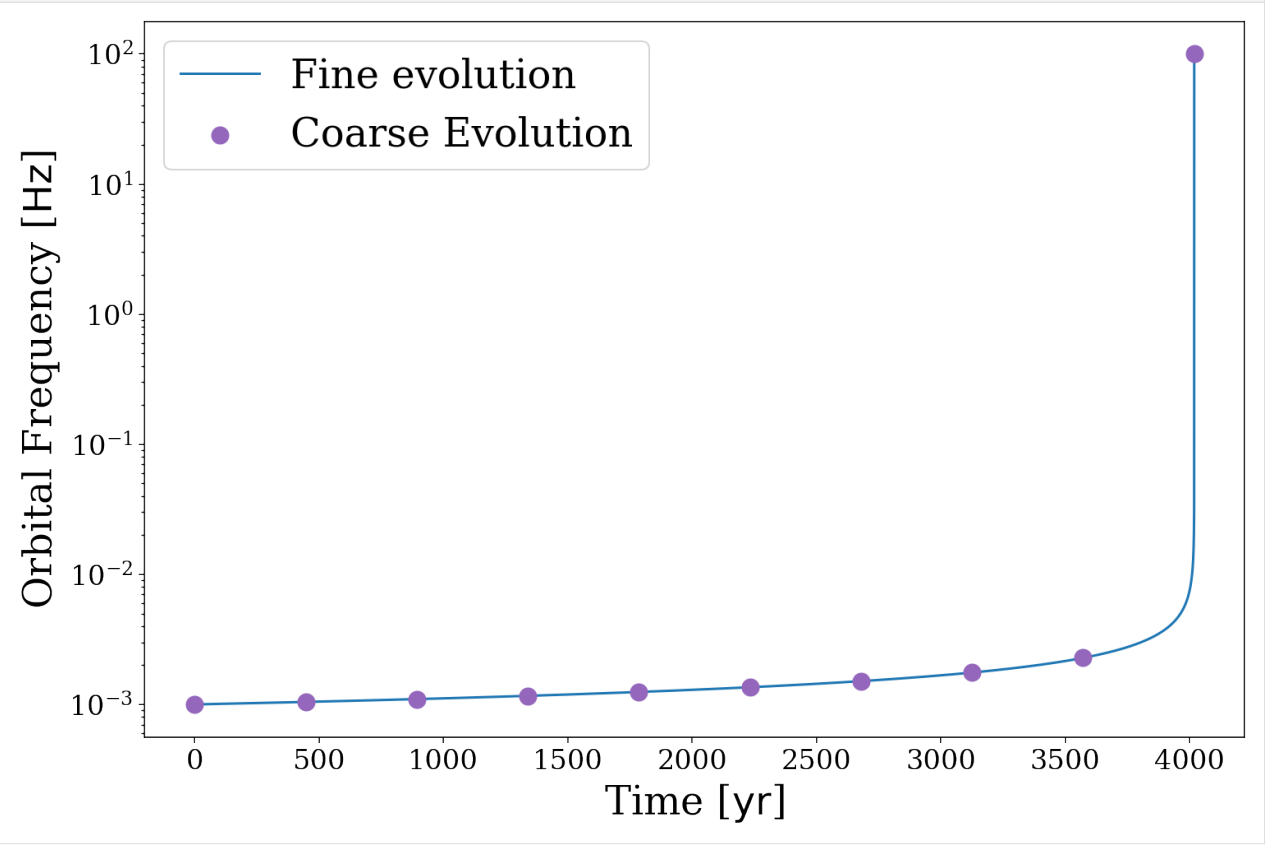
[8]:
```
m_1 = 7 * u.Msun
m_2 = 42 * u.Msun
f_orb_i = 1e-3 * u.Hz

# work out the merger time
t_merge = evol.get_t_merge_circ(m_1=m_1, m_2=m_2, f_orb_i=f_orb_i).to(u.yr)

# evolve with only 10 timesteps
f_orb_evol_coarse, t_coarse = evol.evol_circ(t_evol=t_merge, n_step=10, m_1=m_1, m_2=m_2,
↪ f_orb_i=f_orb_i,
                                             output_vars=["f_orb", "timesteps"])

# same thing but with 10000!
f_orb_evol_fine, t_fine = evol.evol_circ(t_evol=t_merge, n_step=10000, m_1=m_1, m_2=m_2,␣
↪f_orb_i=f_orb_i,
                                         output_vars=["f_orb", "timesteps"])
```

[9]:
```
fig, ax = plt.subplots()

ax.plot(t_fine, f_orb_evol_fine, label="Fine evolution")
ax.scatter(t_coarse, f_orb_evol_coarse, color="tab:purple", zorder=3, s=100, label=
↪"Coarse Evolution")

ax.set_xlabel("Time [{:latex}]".format(t_fine.unit))
ax.set_ylabel("Orbital Frequency [{:latex}]".format(f_orb_evol_fine.unit))

ax.set_yscale("log")
```

(continues on next page)

(continued from previous page)

```
ax.legend()

plt.show()
```



We can see here that we capture most of the evolution well with few timesteps... until the binary approaches its merger. Let's try still taking few timesteps but spacing them such that there are more at the end.

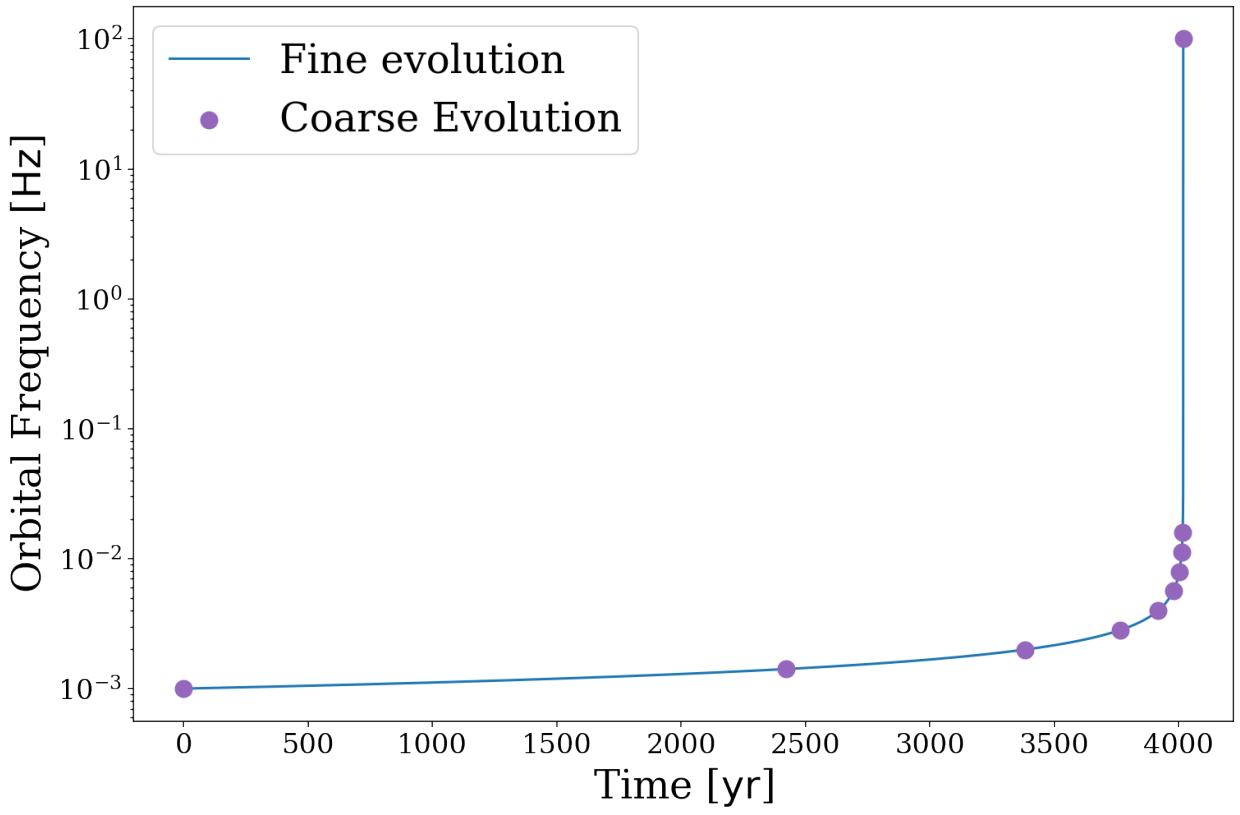
[10]:
```
m_1 = 7 * u.Msun
m_2 = 42 * u.Msun
f_orb_i = 1e-3 * u.Hz

# work out the merger time
t_merge = evol.get_t_merge_circ(m_1=m_1, m_2=m_2, f_orb_i=f_orb_i).to(u.yr).value

# do a sort of lookback time thing
t_coarse = (t_merge - np.logspace(0, np.log10(t_merge), 10))[::-1] * u.yr
t_coarse[-1] = t_merge * u.yr
f_orb_evol_coarse, t_coarse = evol.evol_circ(timesteps=t_coarse, m_1=m_1, m_2=m_2, f_orb_
↪i=f_orb_i,
                                             output_vars=["f_orb", "timesteps"])

# evolve until merger with a bunch of timesteps
f_orb_evol_fine, t_fine = evol.evol_circ(t_evol=t_merge * u.yr, n_step=10000, m_1=m_1, m_
↪2=m_2,
                                         f_orb_i=f_orb_i, output_vars=["f_orb",
```

(continues on next page)

(continued from previous page)

```
↪"timesteps"])
```

[11]:
```
fig, ax = plt.subplots()

ax.plot(t_fine, f_orb_evol_fine, label="Fine evolution")
ax.scatter(t_coarse, f_orb_evol_coarse, color="tab:purple", zorder=3, s=100, label=
↪"Coarse Evolution")

ax.set_xlabel("Time [{:latex}]".format(t_fine.unit))
ax.set_ylabel("Orbital Frequency [{:latex}]".format(f_orb_evol_fine.unit))

ax.set_yscale("log")

ax.legend()

plt.show()
```



And now we can use the same number of timesteps yet resolve the full evolution quite nicely, neat!

Note: It is important to note that the time steps that you provide only determine the times that are *outputted*, internally the functions will determine the time steps necessary to accurately reflect the evolution.

2.4.3 Examples uses

Hopefully by this point you're comfortable with the use of the `evol` module and want to take it for a spin! Let's do just that.

As an example, let's evolve a series of random binaries for a fixed period of time and see how their eccentricities and frequencies change.

[12]:
```
# create some random binaries with the same masses
n_binaries = 2500
m_1 = np.repeat(15, n_binaries) * u.Msun
m_2 = np.repeat(15, n_binaries) * m_1
f_orb_i = 10**(np.random.uniform(-6, -3, n_binaries)) * u.Hz
ecc_i = np.random.uniform(0, 0.9, n_binaries)

# set up timesteps
timesteps = (np.logspace(-2, 7, 1000) * u.yr).to(u.Myr)

# check which binaries will merge during that time
t_merge = evol.get_t_merge_ecc(ecc_i=ecc_i, m_1=m_1, m_2=m_2, f_orb_i=f_orb_i)
merged = t_merge <= timesteps[-1]
inspiral = np.logical_not(merged)

# evolve the binaries that won't merge
ecc_evol, f_orb_evol = evol.evol_ecc(ecc_i=ecc_i[inspiral], m_1=m_1[inspiral], m_2=m_
↪2[inspiral],
                                     f_orb_i=f_orb_i[inspiral], timesteps=timesteps, t_
↪before=10 * u.yr)

print("Completed evolution of {} binaries".format(n_binaries), end="")
print(r" for {:1.0f} {}".format(timesteps[-1].value, timesteps.unit))
```

```
Completed evolution of 2500 binaries for 10 Myr
```

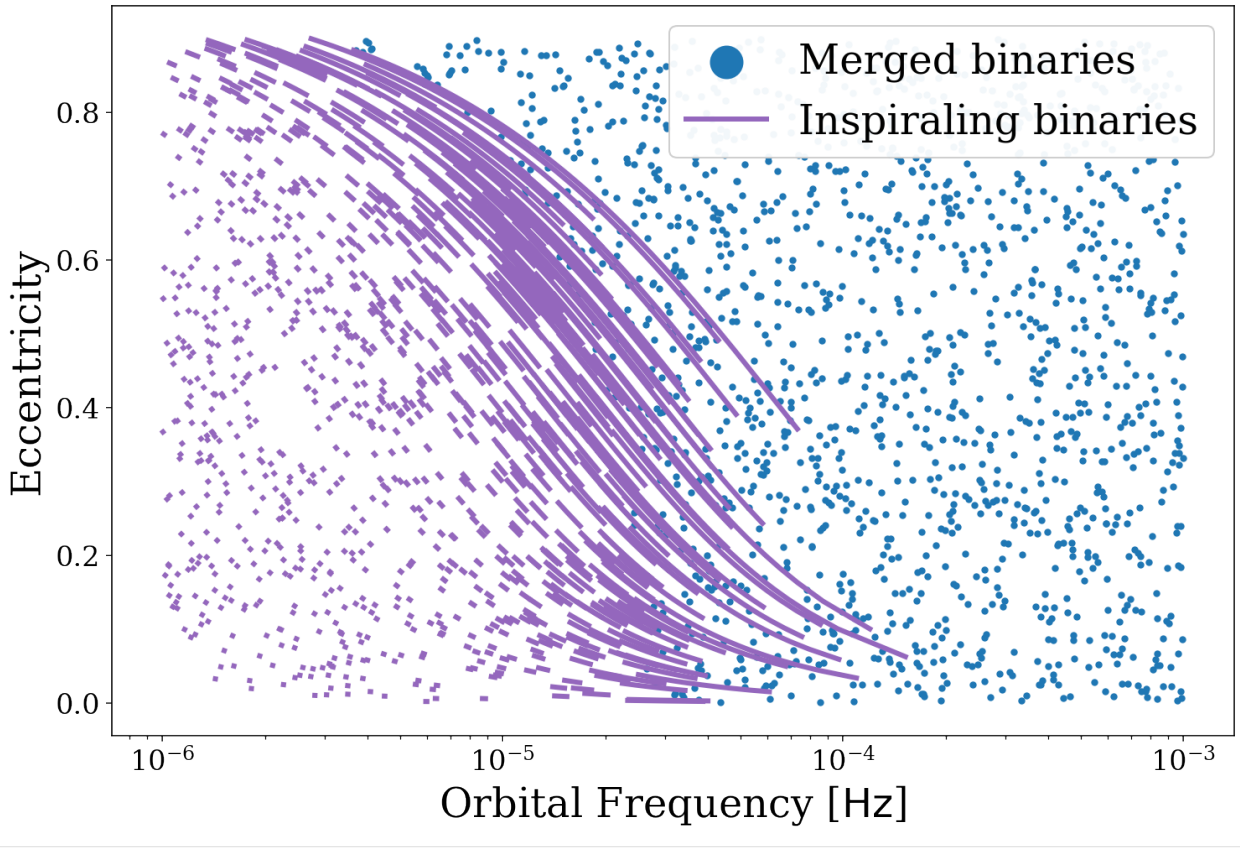
[13]:
```
# create a plot
fig, ax = plt.subplots()

# check which binaries merged
print("{:1.1f}% of binaries merged".format(len(ecc_i[merged]) / n_binaries * 100))

# plot the merged binaries as scatter points
ax.scatter(f_orb_i[merged], ecc_i[merged], label="Merged binaries", s=10)

# plot the inspiraling line by connecting lines from their start to their end
for i in range(len(ecc_evol)):
    label = "Inspiraling binaries" if i == 0 else ""
    mask = f_orb_evol[i] != 1e2 * u.Hz
    ax.plot(f_orb_evol[i][mask], ecc_evol[i][mask], color="tab:purple", lw=3,␣
↪label=label)

ax.legend(loc="upper right", framealpha=0.95, markerscale=6)

ax.set_xscale("log")
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Orbital Frequency [{:latex}]".format(f_orb_evol.unit))
ax.set_ylabel("Eccentricity")

plt.show()
```

```
59.8% of binaries merged
```



We can see here that either higher eccentricity or high frequency can speed up a merger. We also get a sense of how the parameters evolve. Fun!

That concludes this tutorial on binary evolution using the LEGWORK package. Be sure to check out *the other tutorials* to learn more about how LEGWORK can do the legwork for you!

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.5 Visualisation

In this tutorial, we explore how to use the various functions in the visualisation module of LEGWORK. The visualisation module is designed to make it quick and easy to plot various distributions of a collection of gravitational wave sources as well as show them on the LISA sensitivity curve.

2.5.1 Setup

First we need to import `legwork` and some other standard stuff.

[3]:
```
import legwork.visualisation as vis
from legwork import source, psd

import numpy as np
import astropy.units as u
import astropy.constants as const
import matplotlib.pyplot as plt
from matplotlib.colors import TwoSlopeNorm
```

2.5.2 Plotting parameter distributions

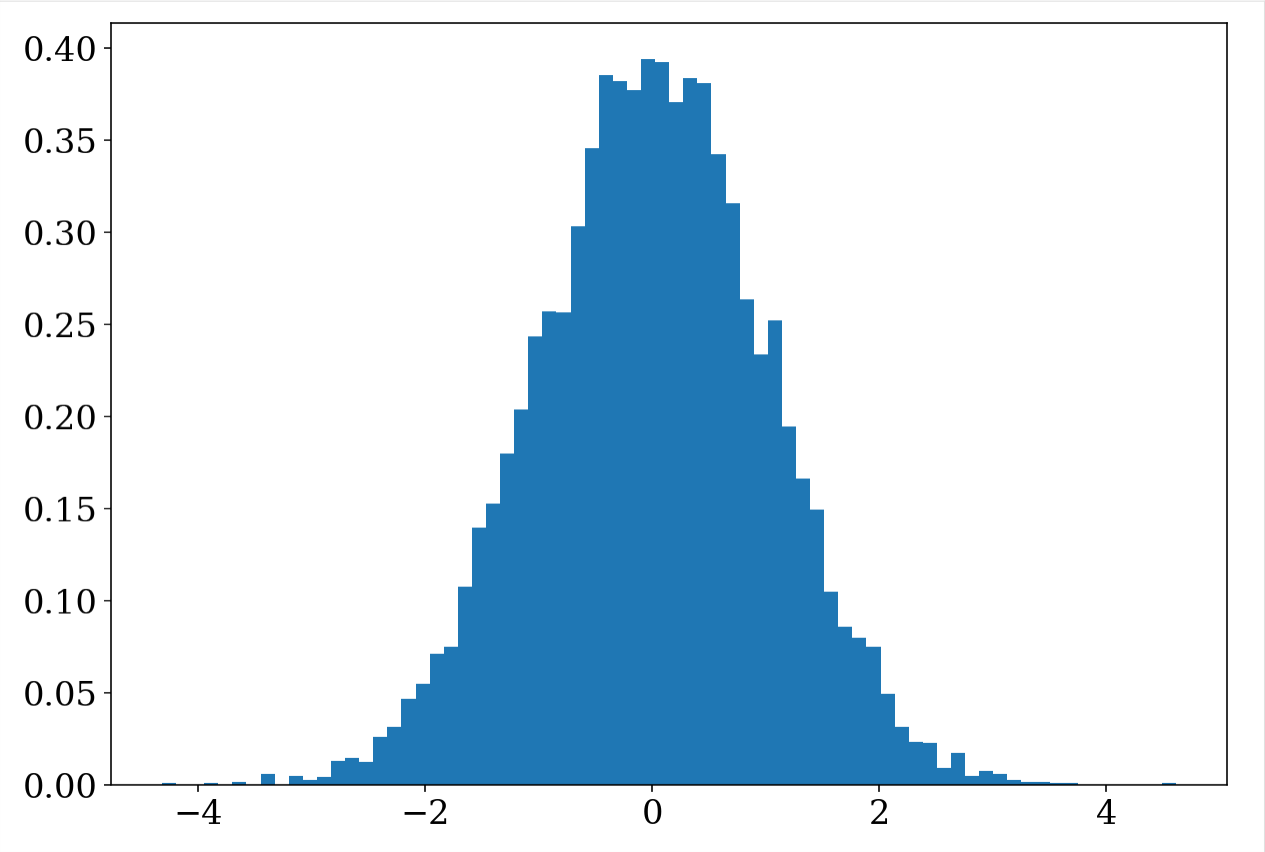
To start, we can explore how we can use the module to investigate distributions of parameters.

1D distributions

The first function we can take a look at is *`legwork.visualisation.plot_1D_dist()`*, which is essentially a wrapper over `matplotlib.pyplot.hist()`, `seaborn.kdeplot()` and `seaborn.ecdfplot()` such that you can dynamically switch between these types of distributions with a single function call. At its most basic level, you can use the function to examine a histogram of some distribution.

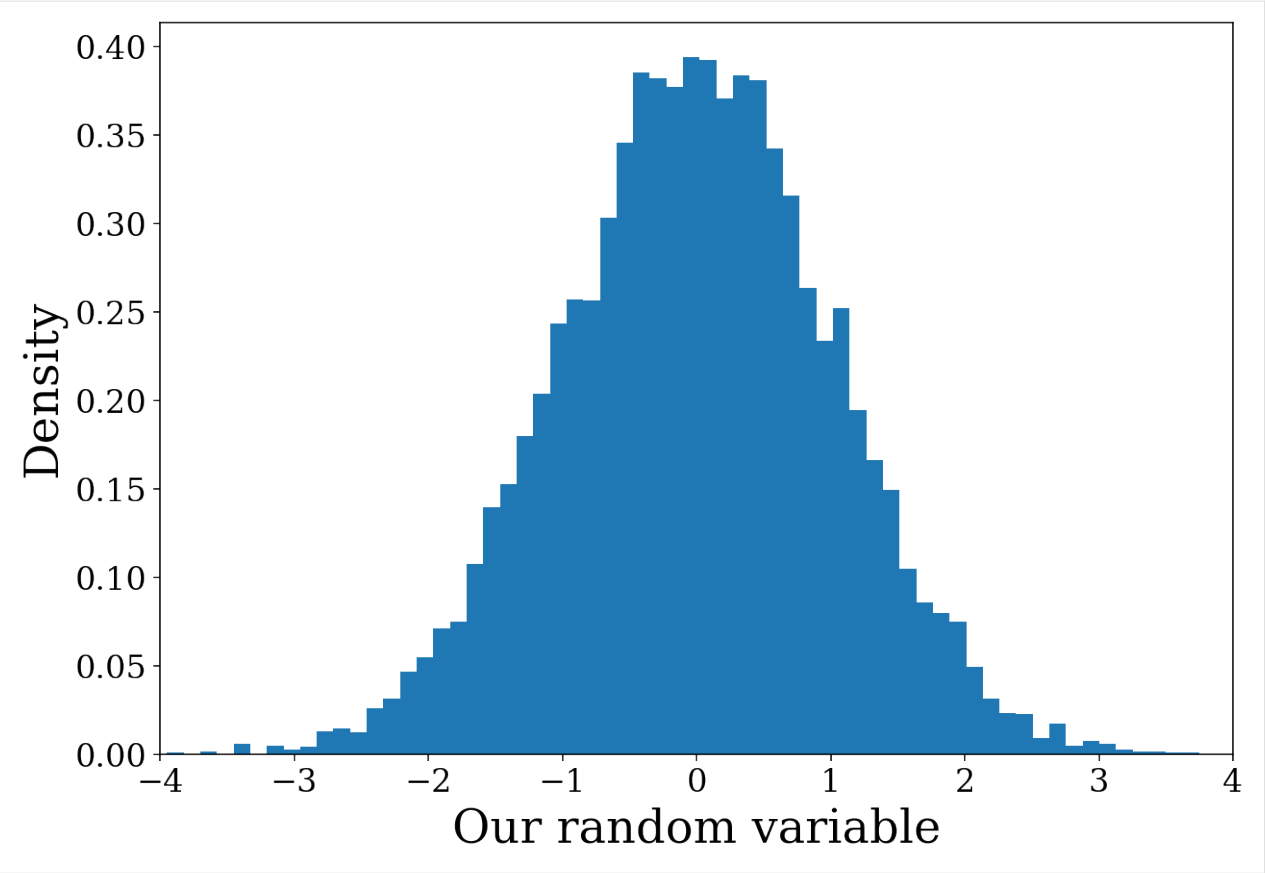
[5]:
```
# create a random normal variable
x = np.random.normal(size=10000)
```

[6]:
```
# plot with all of the default settings
fig, ax = vis.plot_1D_dist(x=x)
```



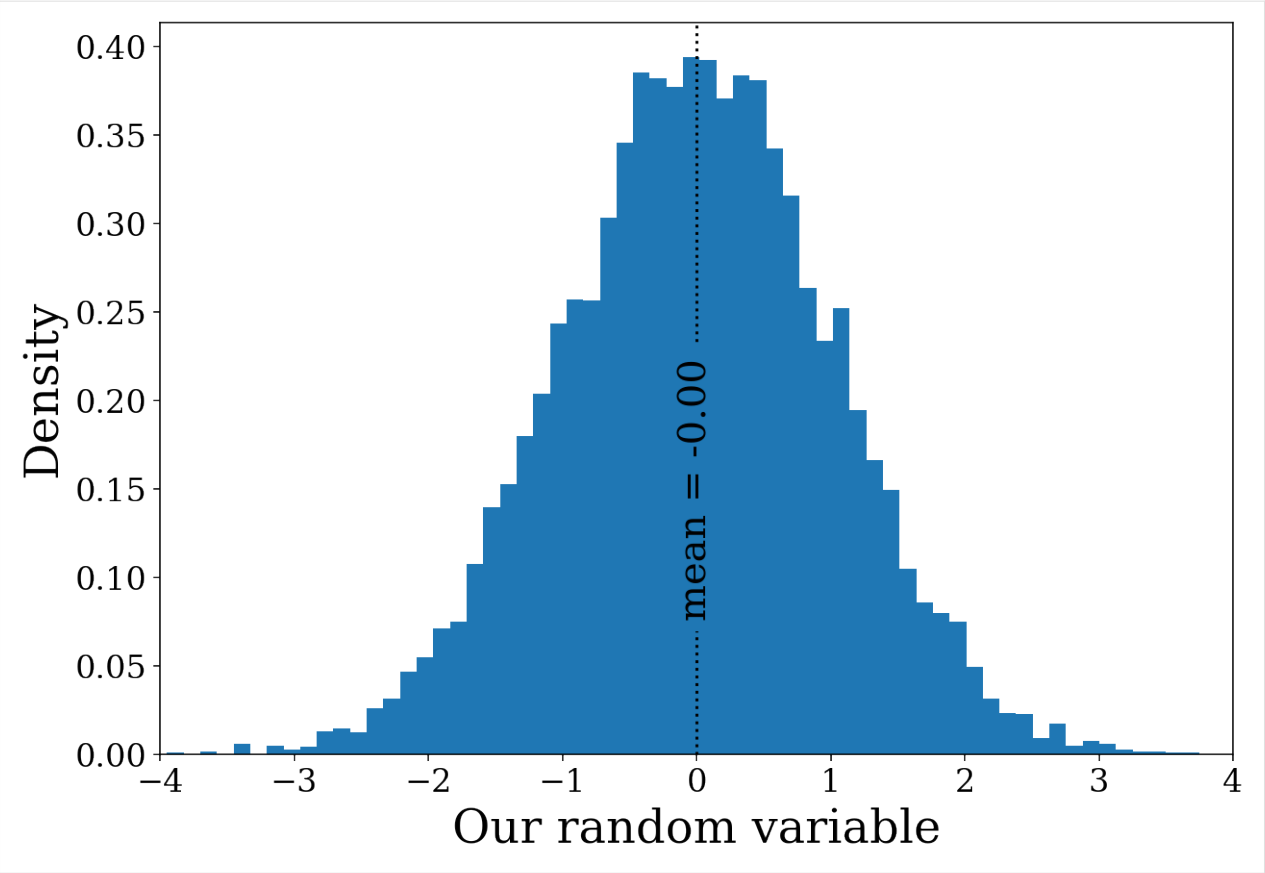
Nice and simple! But this plot isn't particularly informative so we can also specify various plot parameters as part of the function. Let's add some axis labels and change the x limits to ensure symmetry.

[7]:
```
fig, ax = vis.plot_1D_dist(x=x, xlabel="Our random variable", ylabel="Density", xlim=(-4,
↪ 4))
```



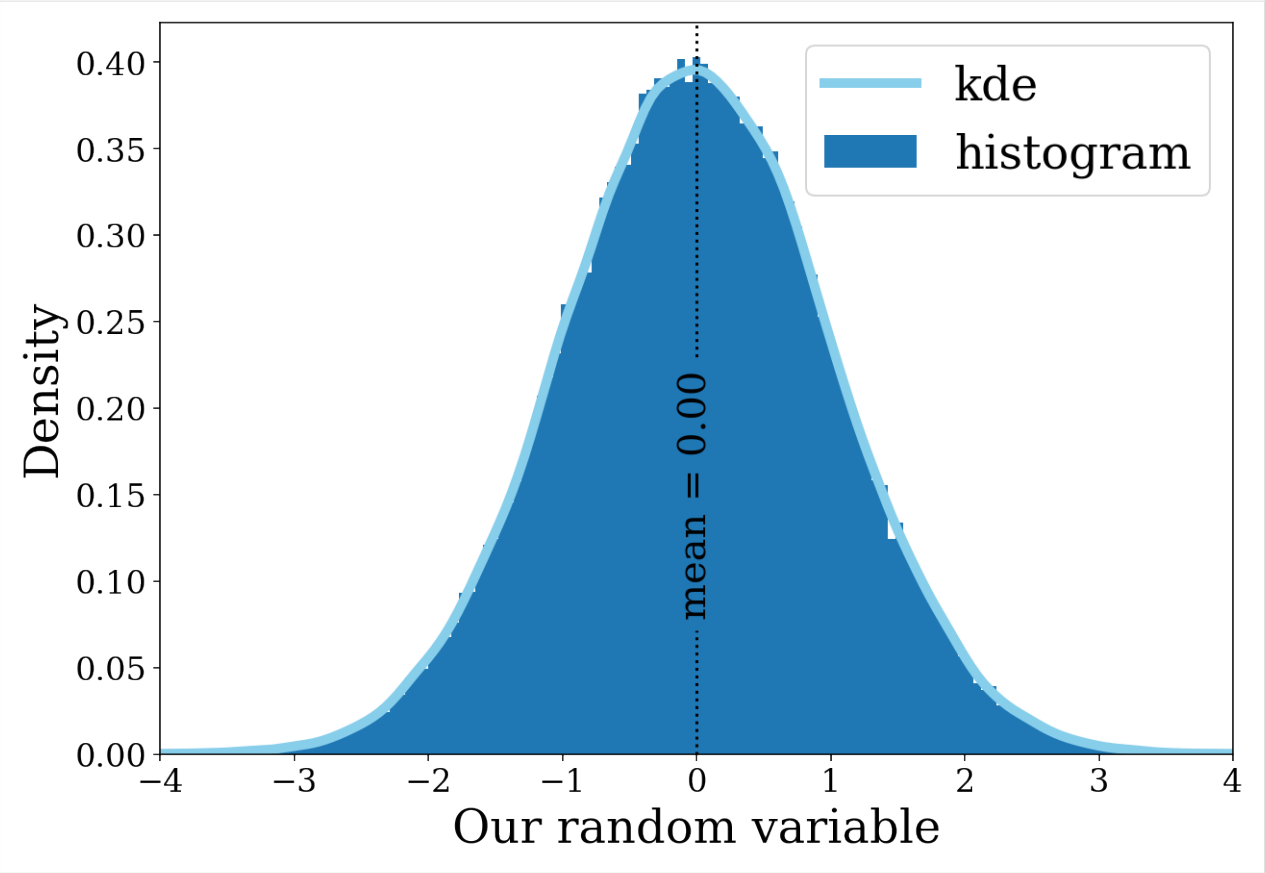
Well that's much better! Perhaps instead of immediately showing the plot, it could be good to add some other features first. We can do this by specifying `show=False` and then using the returned figure and axis. Let's put this into practice by adding a line indicating the mean of the distribution which should be approximately 0.

[8]:
```
# create the histogram but don't show it
fig, ax = vis.plot_1D_dist(x=x, xlabel="Our random variable", ylabel="Density",
                           xlim=(-4, 4), show=False)

# add a line and annotation
ax.axvline(np.mean(x), linestyle="dotted", color="black")
ax.annotate("mean = {0:1.2f}".format(x.mean()), xy=(np.mean(x), 0.15),
            rotation=90, ha="center", va="center", fontsize=20,
            bbox=dict(boxstyle="round", fc="tab:blue", ec="none"))

# show the figure
plt.show()
```



The histogram seems to be working nicely. However, that might not be the only way you want to show the distribution. From here we can explore how we can change the distribution with various arguments. Let's add a KDE (kernel density estimator) in addition to the histogram.

[9]:
```
# create a larger sample for a smoother line
x = np.random.normal(size=100000)

# add the KDE
fig, ax = vis.plot_1D_dist(x=x, disttype="kde", show=False, color="skyblue", label="kde",
↪ linewidth=5)
fig, ax = vis.plot_1D_dist(x=x, xlabel="Our random variable", ylabel="Density", xlim=(-4,
↪ 4),
                           label="histogram", fig=fig, ax=ax, show=False)

# add a line and annotation
ax.axvline(np.mean(x), linestyle="dotted", color="black")
ax.annotate("mean = {0:1.2f}".format(x.mean()), xy=(np.mean(x), 0.15), rotation=90, ha=
↪"center", va="center",
            fontsize=20, bbox=dict(boxstyle="round", fc="tab:blue", ec="none"))

ax.legend()

# show the figure
plt.show()
```



More things you can try out

There's only so much we can show in a tutorial but this function can do much more. Here are some things you may like to try out.

- Explore the different disttypes, you can make an empirical cumulative distribution function with `disttype=ecdf`.
- Try changing the number of bins in the histogram (`bins=10`) or adjusting the bandwidth for the kde (`bw_adjust=0.5`)
- Plot things on a log scale with (`log=True` for histograms or `log_scale=(True, False)` for KDE/ECDFs)

2D distributions

This is very similar to the previous section except now you can make 2D distributions of either scatter plots or density distributions (since *`legwork.visualisation.plot_2D_dist()`* is a wrapper over `matplotlib.pyplot.scatter()` and `seaborn.kdeplot()`). Let's make a simple scatter plot first.

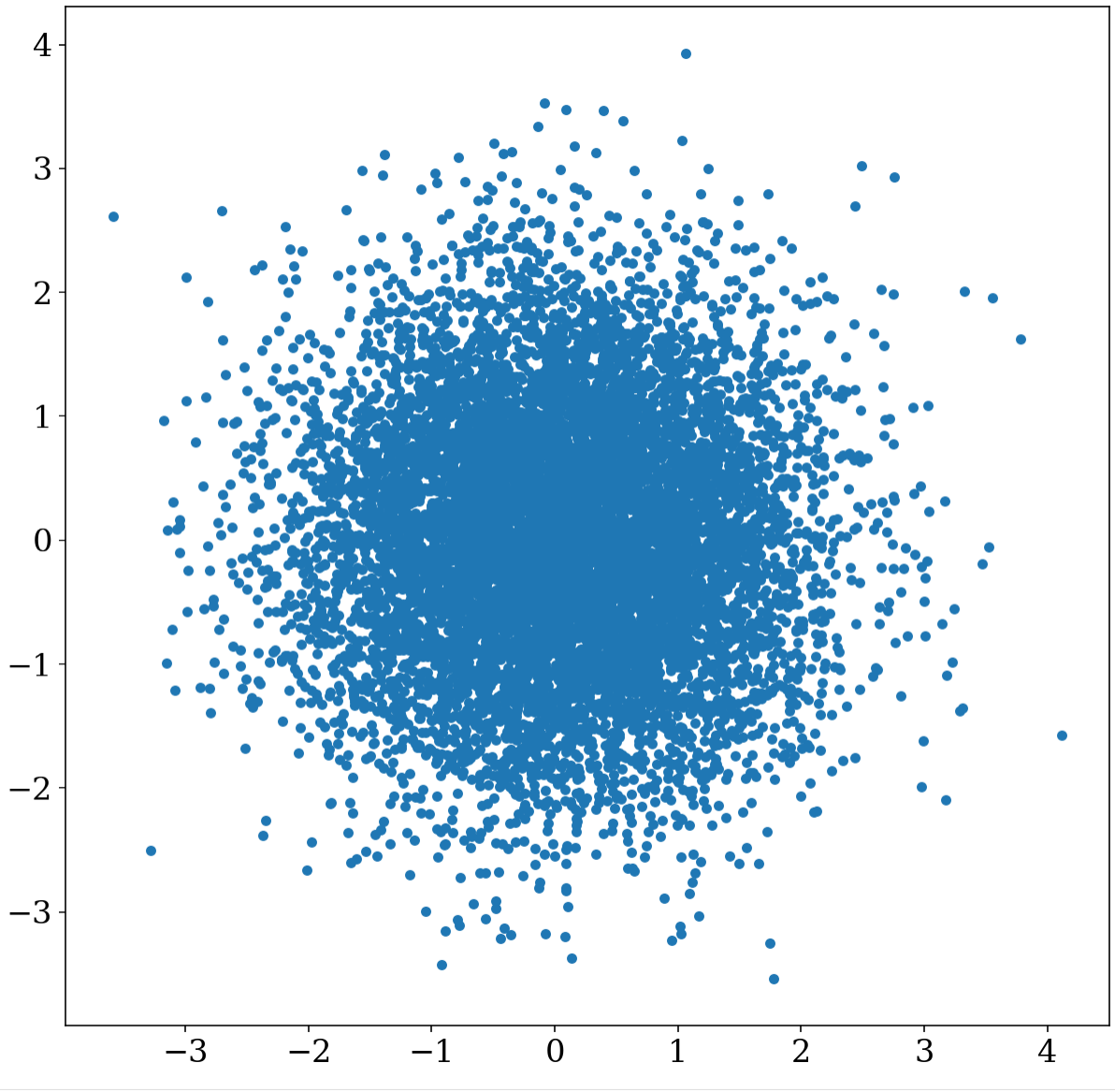
[10]:
```
# create two random normal variables
n_vals = 10000
x = np.random.normal(size=n_vals)
y = np.random.normal(size=n_vals)
```

[11]:
```
# create a square figure
fig, ax = plt.subplots(figsize=(10, 10))
```

(continues on next page)

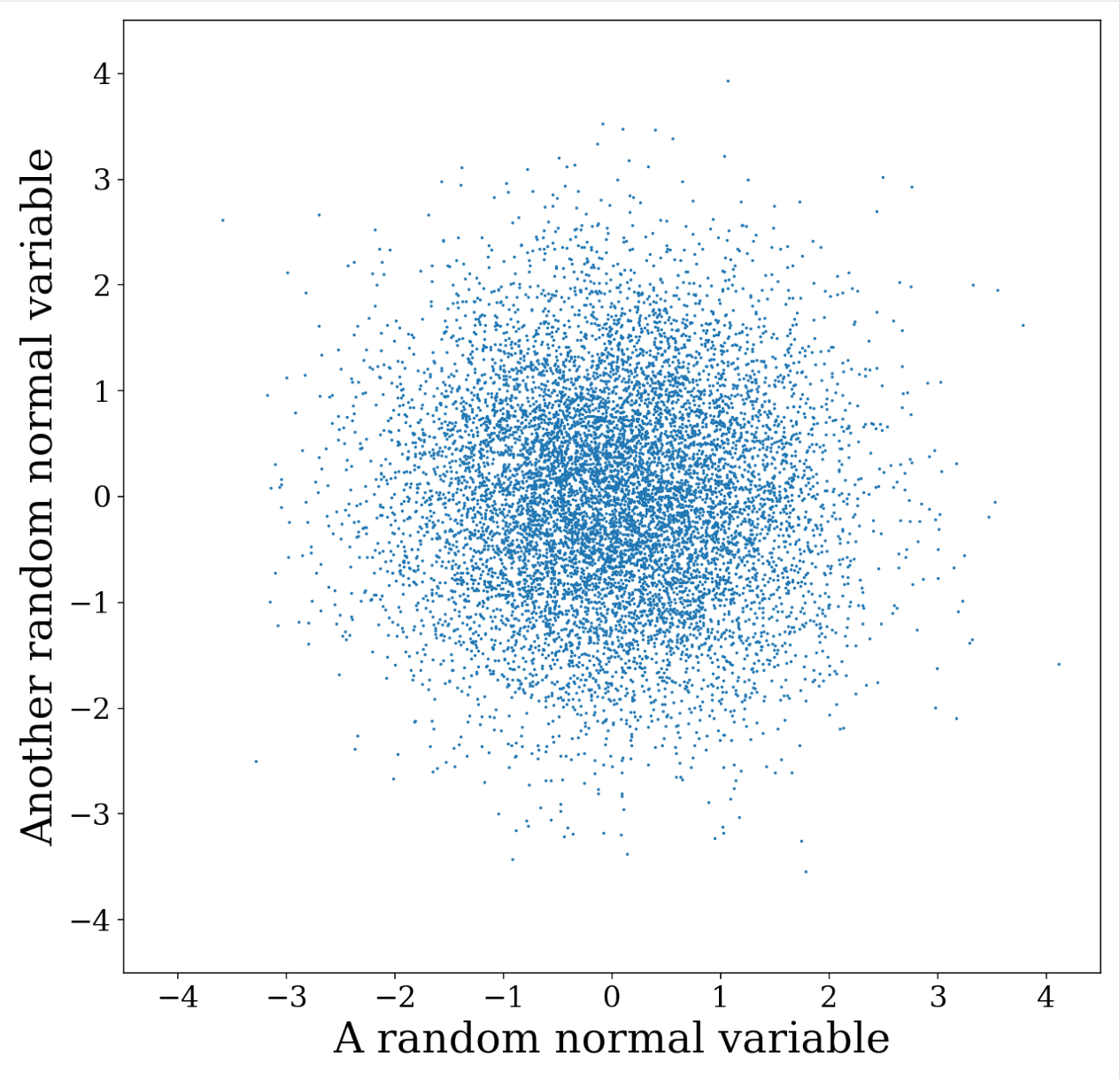
(continued from previous page)

```
# add the scatter plot
fig, ax = vis.plot_2D_dist(x=x, y=y, fig=fig, ax=ax)
```



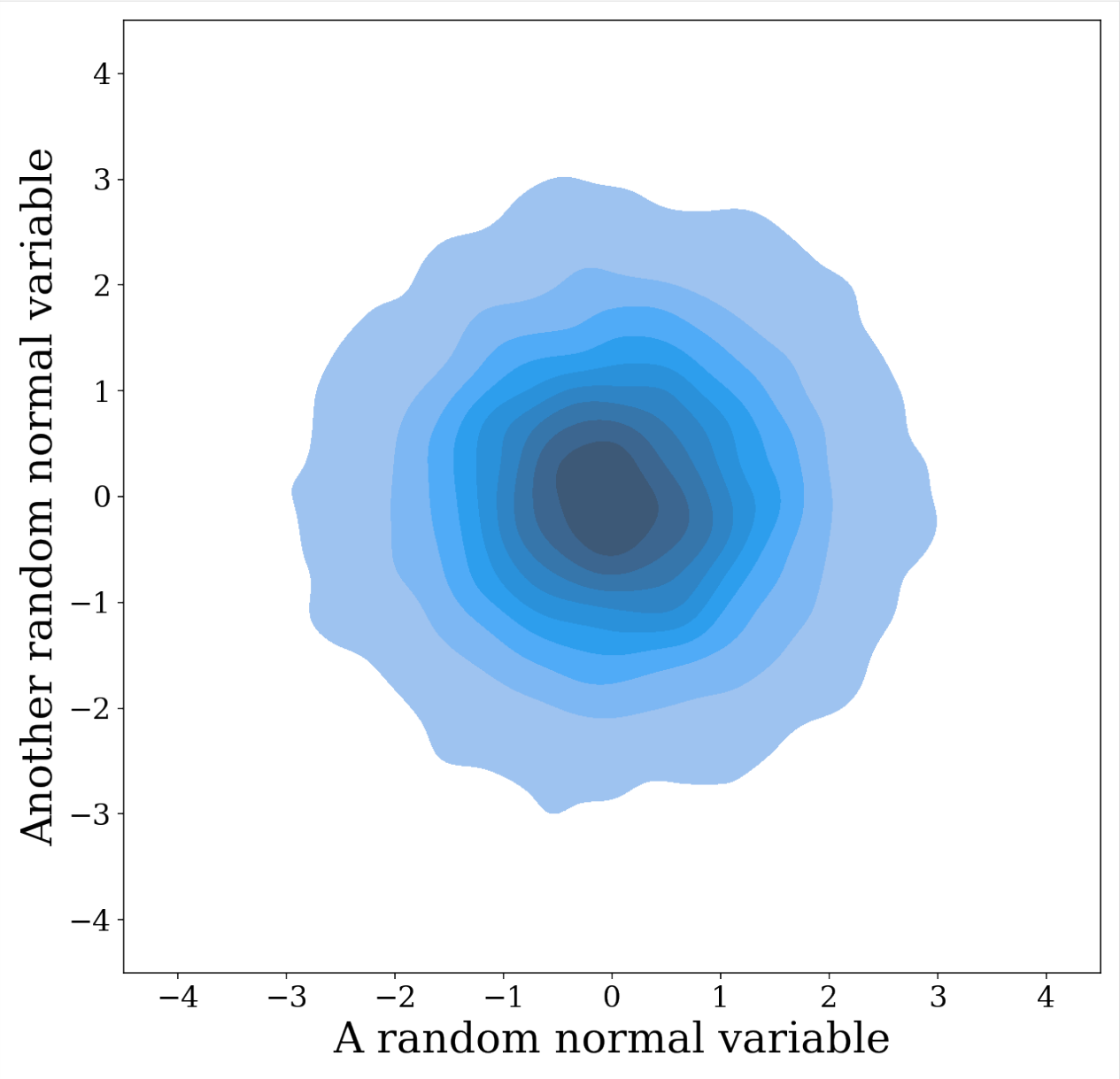
Again, we can make this much better by adding axis labels. But additionally the centre is rather oversaturated with points so it is hard to discern the distribution so let's try decreasing the point size too (to `scatter_s=0.5`).

[12]:
```
# create a square figure
fig, ax = plt.subplots(figsize=(10, 10))

# add the scatter plot
fig, ax = vis.plot_2D_dist(x=x, y=y, fig=fig, ax=ax, xlim=(-4.5, 4.5), ylim=(-4.5, 4.5),
                           xlabel="A random normal variable", ylabel="Another random␣
↪normal variable", scatter_s=0.5)
```



Well perhaps the centre is better...but now it is difficult to see the outliers! This is where the KDE density plot could really come in handy. So we can switch to `disttype=kde` instead.

[13]:
```
# create a square figure
fig, ax = plt.subplots(figsize=(10, 10))

# add the scatter plot
fig, ax = vis.plot_2D_dist(x=x, y=y, fig=fig, ax=ax, disttype="kde", fill=True, xlim=(-4.
↪5, 4.5), ylim=(-4.5, 4.5),
                           xlabel="A random normal variable", ylabel="Another random␣
↪normal variable")
```



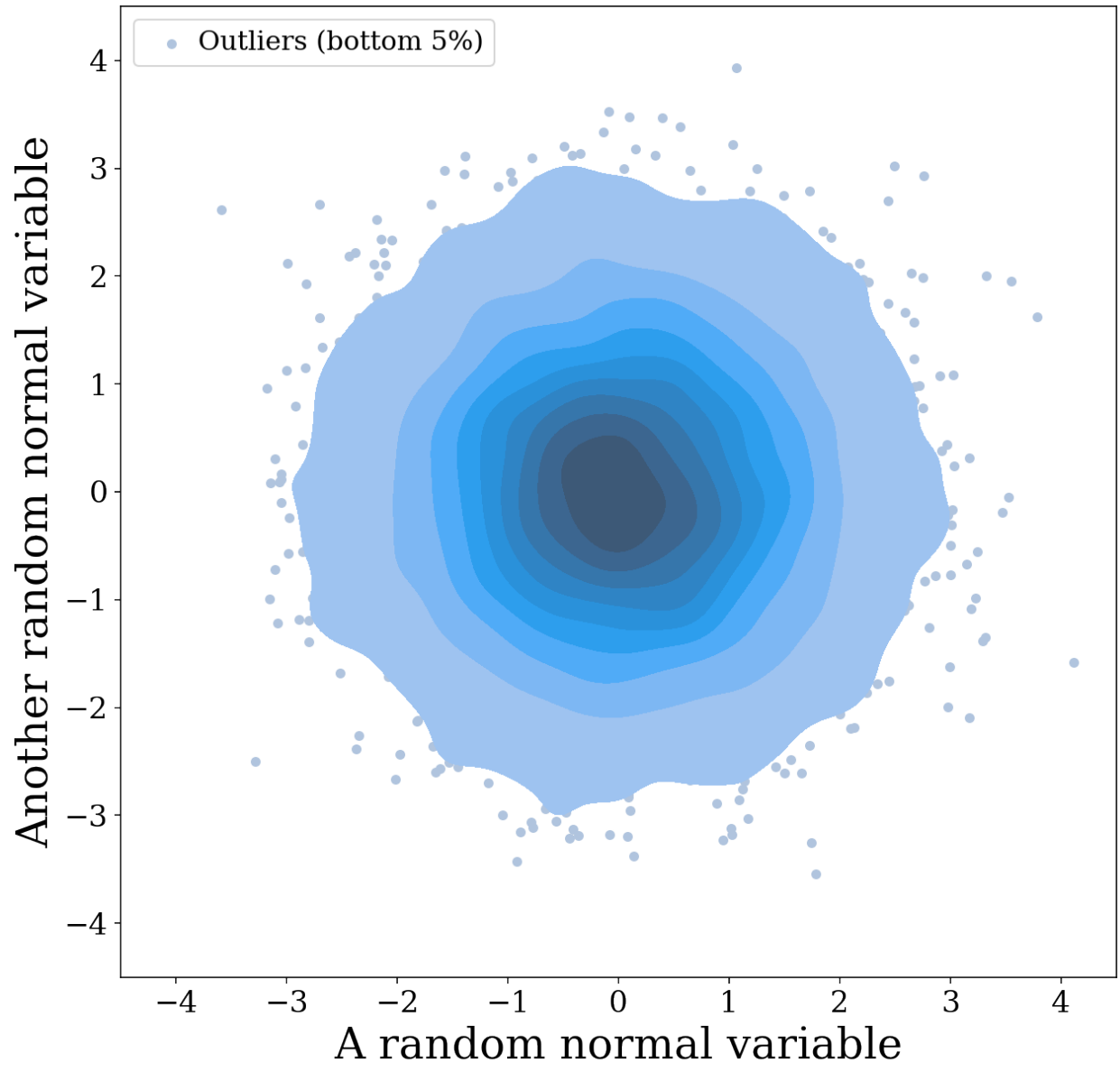
Well that looks rather cool! However, we've added another issue...now we can't see *any* of the outliers as this density plot only goes down to the last 5% (you can adjust this with `thresh=0.01` for example)! So why don't we also add the scatter plot so the outliers are still present.

[14]:
```
# create a square figure
fig, ax = plt.subplots(figsize=(10, 10))

fig, ax = vis.plot_2D_dist(x=x, y=y, fig=fig, ax=ax, show=False, label="Outliers (bottom␣
↪5%)", color="lightsteelblue")
fig, ax = vis.plot_2D_dist(x=x, y=y, fig=fig, ax=ax, disttype="kde", fill=True, xlim=(-4.
↪5, 4.5), ylim=(-4.5, 4.5),
                           show=False, xlabel="A random normal variable", ylabel=
↪"Another random normal variable")
```

(continues on next page)

(continued from previous page)

```
# add legend with information on outliers
ax.legend(loc="upper left", handletextpad=0.0, fontsize=15)
plt.show()
```

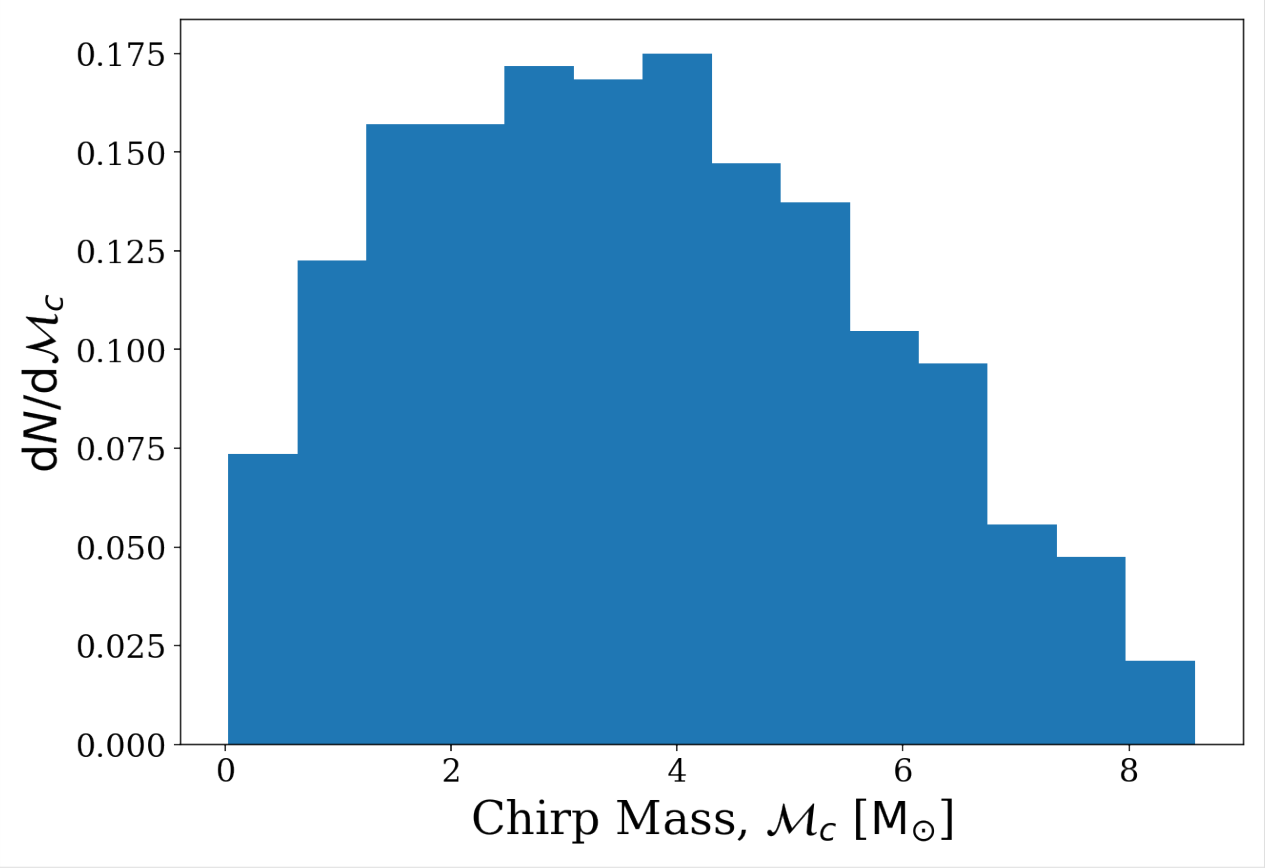


Plotting distributions directly from the Source class

You may be thinking that all of those various options are rather a lot of work and perhaps you'd rather just immediately see the distribution of interesting source parameters. Have no fear, here is also a wrapper to help automatically plot and generate axis labels for plots of source parameters! For example, we can plot the chirp mass distribution like so:

[15]:
```
# create a random collection of sources
n_values = 1000
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.uniform(0, 30, n_values) * u.kpc
f_orb = 10**(np.random.uniform(-5, -2, n_values)) * u.Hz
ecc = np.random.uniform(0.0, 0.2, n_values)

sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb, interpolate_
↪g=False)
```

[16]:
```
# create a histogram of the chirp mass
fig, ax = sources.plot_source_variables(xstr="m_c")
```



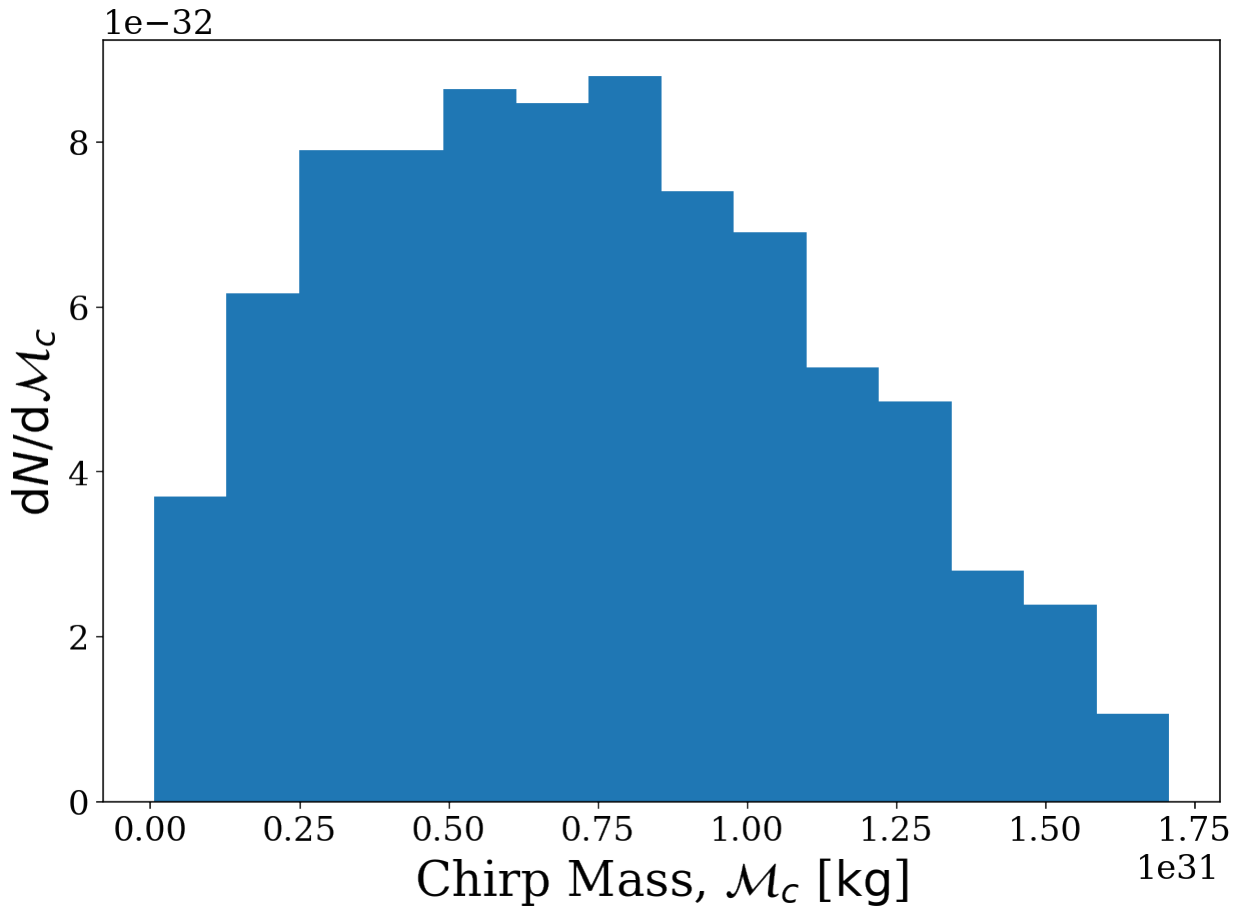
We could have done this for any of the variables, see the documentation *legwork.source.Source.plot_source_variables()* for a full list of parameters you can plot. Also, note that the axis labels are auto-generated based on the units of the variable. For example, let's change the unit of the chirp mass to see how it changes the axis labels.

[17]:
```
# convert chirp mass to kilograms using astropy.units
```

(continues on next page)

(continued from previous page)

```
sources.m_c = sources.m_c.to(u.kg)

# plot the chirp mass using exactly the same code as before
fig, ax = sources.plot_source_variables(xstr="m_c")
```

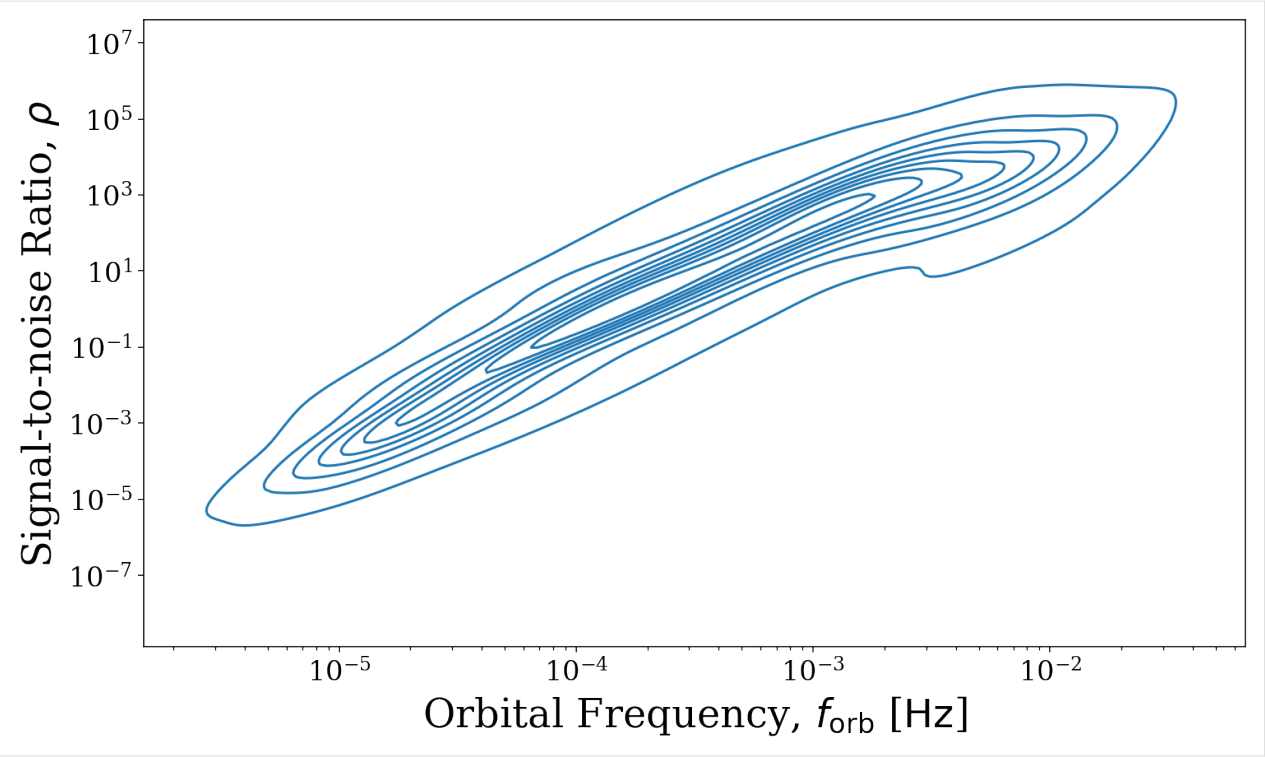


You can also supply two variables and get a 2D distribution instead. Let's compare the orbital frequency and signal-to-noise ratio (which we need to compute first!)

[18]:
```
# compute the SNR (verbosely so you can see what types of sources we have)
snr = sources.get_snr(verbose=True)
```

```
Calculating SNR for 1000 sources
        0 sources have already merged
        957 sources are stationary
                322 sources are stationary and circular
                635 sources are stationary and eccentric
        43 sources are evolving
                11 sources are evolving and circular
                32 sources are evolving and eccentric
```

[19]:
```
fig, ax = sources.plot_source_variables(xstr="f_orb", ystr="snr", disttype="kde", log_
↪scale=(True, True))
```

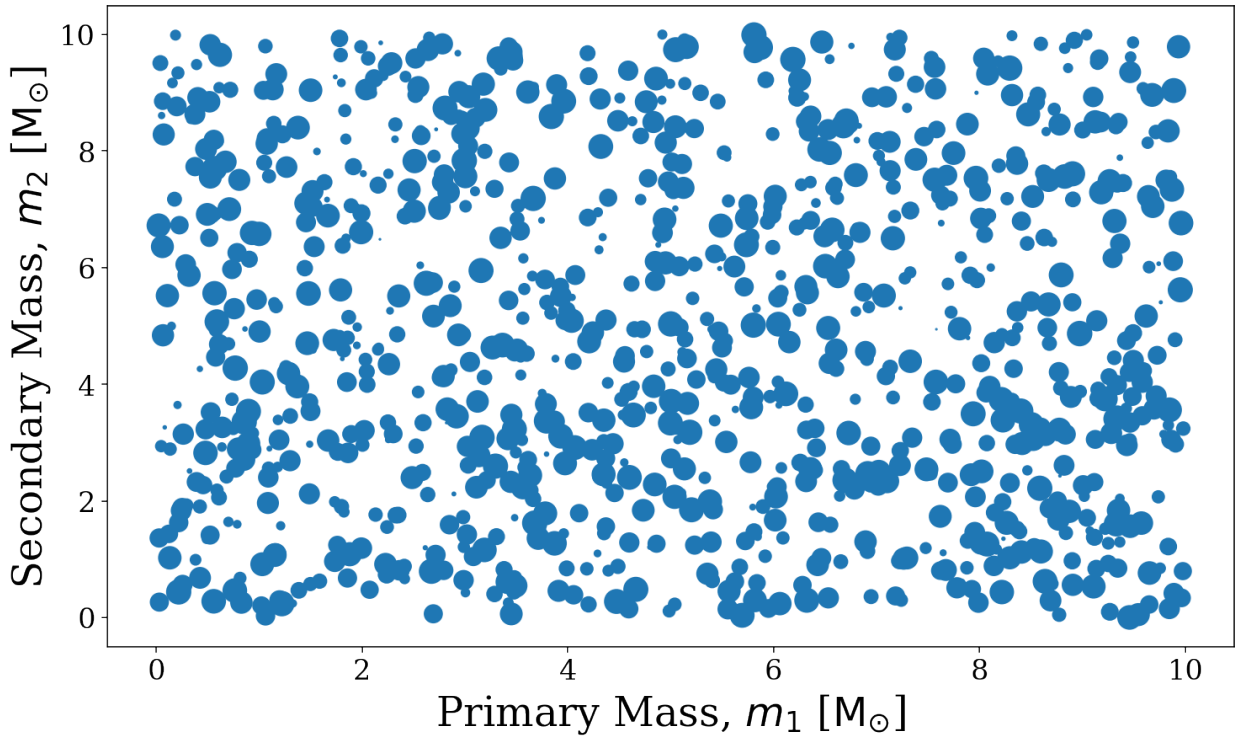


Weighted Samples

What if you prefer one source over the other?? `LEGWORK` handles this too! Just set the `weights` in the source and they'll be used in all histograms and KDEs and scatter plots will have different point sizes

[20]:
```
weights = np.random.uniform(0, 10, n_values)
sources.weights = weights
```

[21]:
```
sources.plot_source_variables(xstr="m_1", ystr="m_2")
```



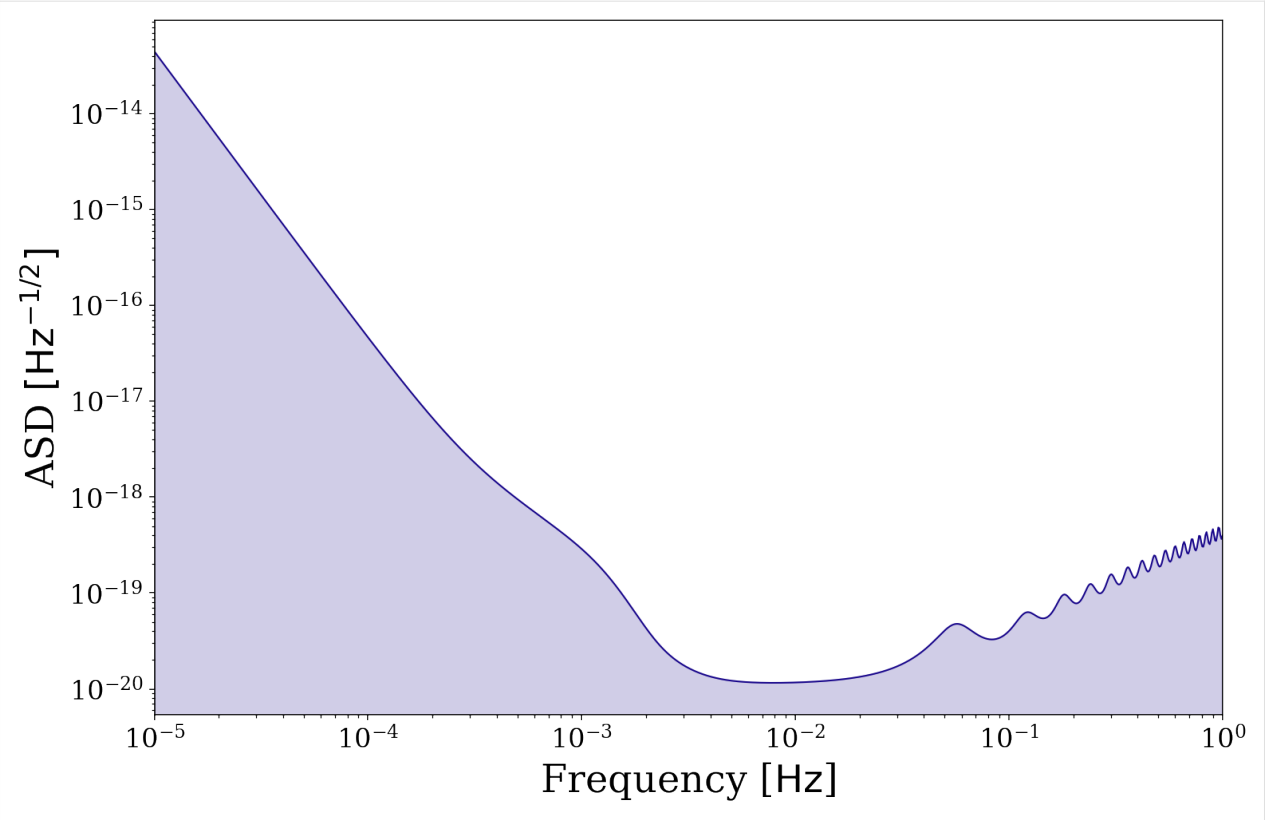
[21]:
```
(<Figure size 864x504 with 1 Axes>,
 <AxesSubplot:xlabel='Primary Mass, $m_1$ [$\\mathrm{M_{\\odot}}$]', ylabel='Secondary␣
↪Mass, $m_2$ [$\\mathrm{M_{\\odot}}$]'>)
```

2.5.3 Plot sensitivity curves

LISA curve

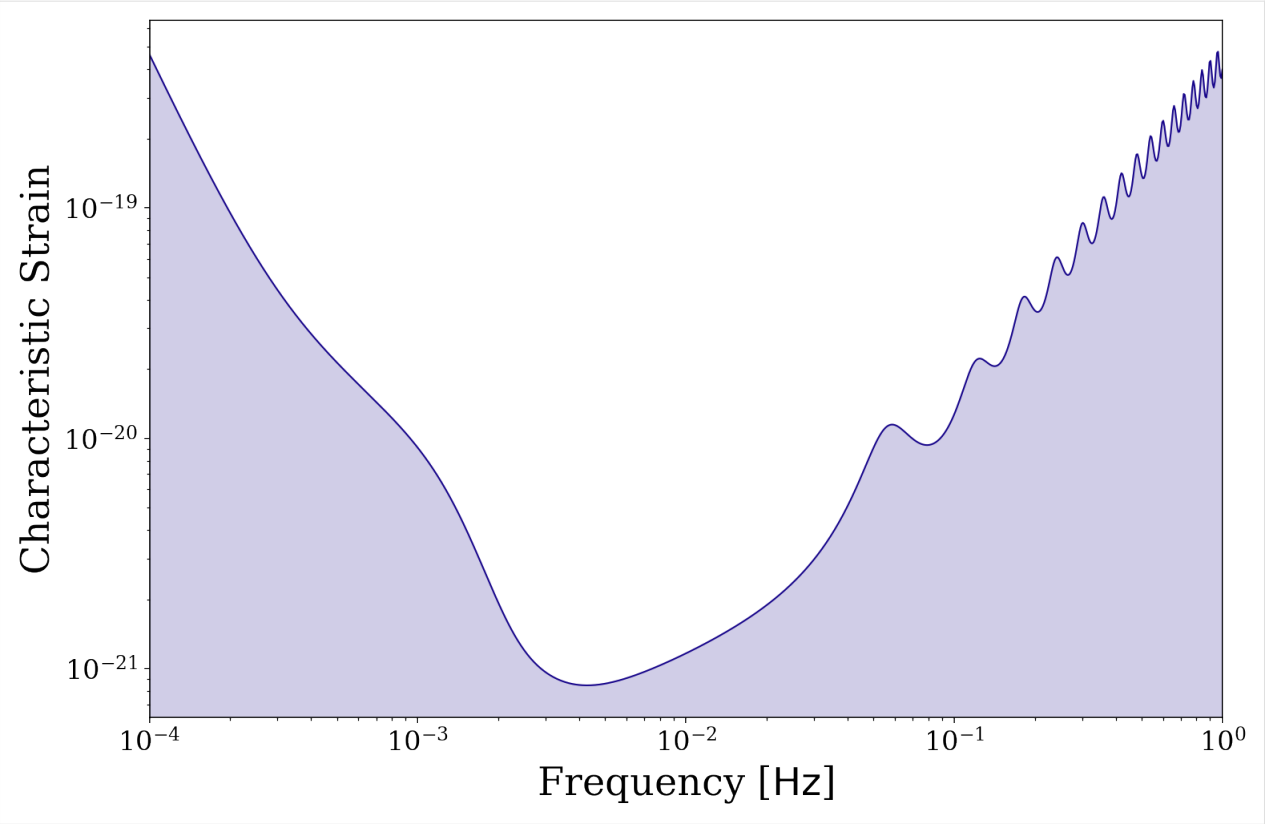
We also add functionality to plot the LISA sensitivity curve. Creating a basic sensitivity curve is as simple as follows. By default it will plot between $10^{-5} < f/\mathrm{Hz} < 1$, with a purple line (the purple matches well with the matplotlib `matplotlib.pyplot.plasma()` colormap) and transparent shading below.

[22]:
```
fig, ax = vis.plot_sensitivity_curve()
```



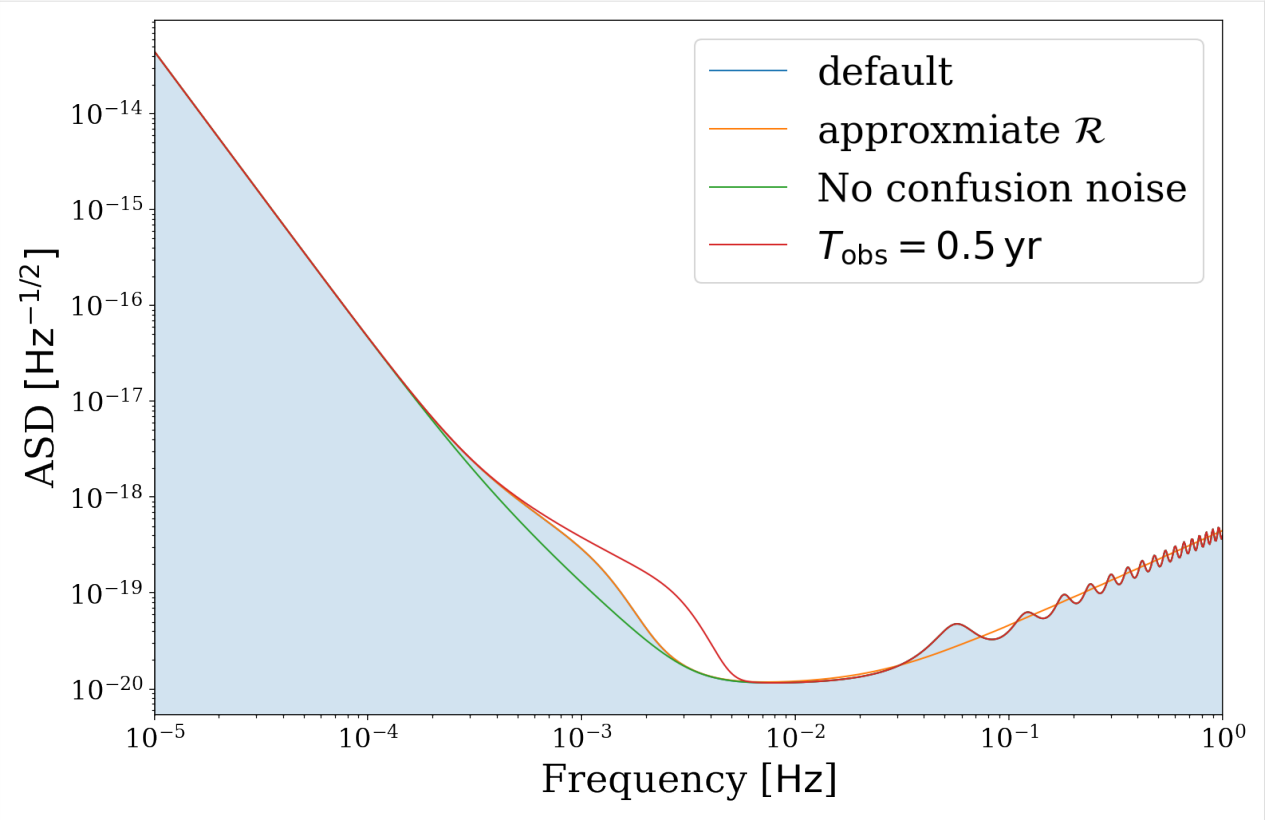
For the default sensitivity curve we plot the ASD, $\sqrt{S_{\rm n}(f)}$, but it is also possible to plot the characteristic strain, $\sqrt{fS_{\rm n}(f)}$, instead. As a demonstration, let's also adjust the frequency range to ignore low frequencies.

[23]:
```
frequency_range = frequency_range=np.logspace(-4, 0, 1000) * u.Hz
fig, ax = vis.plot_sensitivity_curve(frequency_range=frequency_range, y_quantity="h_c")
```



It is also possible to adjust the sensitivity curve by changing the mission length, transfer function approximation and confusion noise. For this plot we set the `color=None` so that it picks default matplotlib colours and remove the fill for clarity.

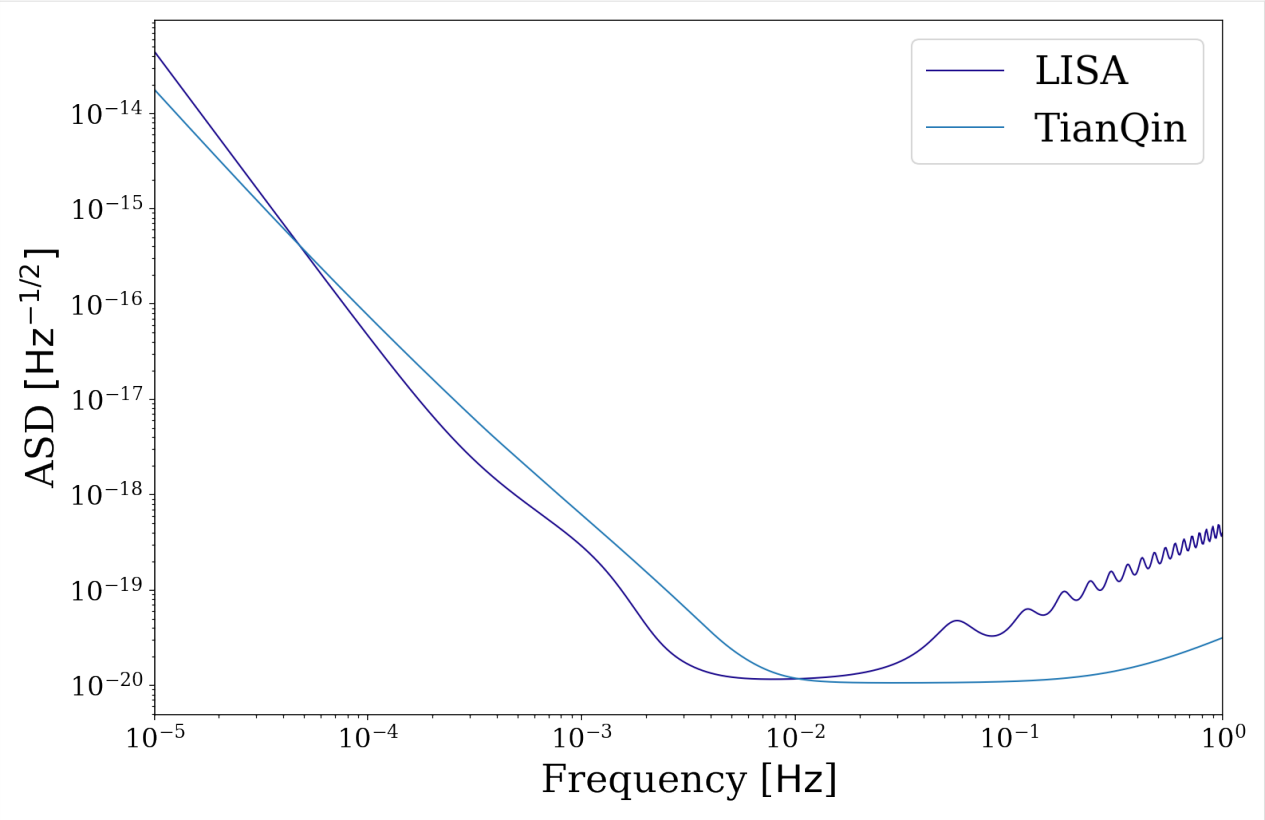
[24]:
```
# default settings (keep fill for this one)
fig, ax = vis.plot_sensitivity_curve(label="default", fill=True, show=False, color=None)

# approximate the LISA transfer function
fig, ax = vis.plot_sensitivity_curve(approximate_R=True, color=None, label=r"approxmiate
↪$\mathcal{R}$",
                                     fill=False, fig=fig, ax=ax, show=False)

# remove all confusion noise
fig, ax = vis.plot_sensitivity_curve(confusion_noise=None, color=None,label=r"No␣
↪confusion noise",
                                     fill=False, fig=fig, ax=ax, show=False)

# shorten the mission length (increases confusion noise)
fig, ax = vis.plot_sensitivity_curve(t_obs=0.5 * u.yr, color=None, label=r"$T_{\rm obs}␣
↪= 0.5 \,\rm{yr}$",
                                     fill=False, fig=fig, ax=ax, show=False)

ax.legend()

plt.show()
```



Alternate instruments

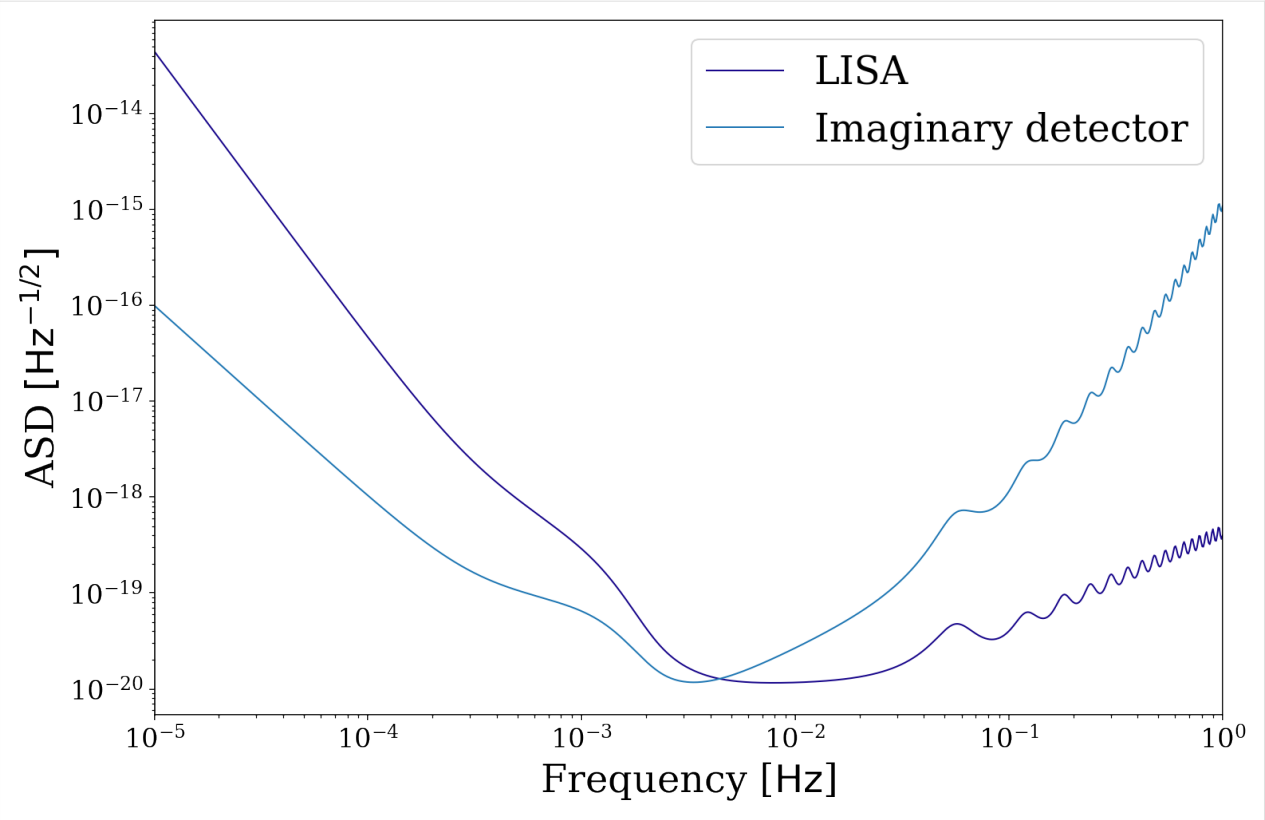
We also support the use of other sensitivity curves. Currently we have implemented the TianQin curve by default but also allow custom curves.

[25]:
```
# compare LISA and TianQin
fig, ax = vis.plot_sensitivity_curve(show=False, fill=False, instrument="LISA", label=
↪"LISA")
fig, ax = vis.plot_sensitivity_curve(show=False, fill=False, instrument="TianQin", label=
↪"TianQin",
                                     color=None, fig=fig, ax=ax)
ax.legend()
plt.show()
```



Let's also compare LISA with some imaginary detector that is a factor different than LISA

[26]:
```
# compare LISA to an imaginary detector
fig, ax = vis.plot_sensitivity_curve(show=False, fill=False, instrument="LISA", label=
↪"LISA")

# note function signature must be the same as lisa_psd, despite some values being ignored
def imaginary_detector(f, t_obs, L, approximate_R, confusion_noise):
    return 5e4 * f.value**2 * np.exp(5 * f.value) * psd.lisa_psd(f=f)

fig, ax = vis.plot_sensitivity_curve(show=False, fill=False, instrument="custom",
                                     custom_psd=imaginary_detector, label="Imaginary␣
↪detector",
                                     color=None, fig=fig, ax=ax, L=5e9)
ax.legend()
plt.show()
```



2.5.4 Plot sources on the sensitivity curve

Finally, we can combine everything we've done so far in this tutorial in order to plot distributions of binaries on the sensitivity curve.

Note: We currently only support this for stationary sources (that are plotted as points rather than lines) but we are working on implementing this for evolving sources too.

Circular and Stationary Binaries

Let's start by creating a collection of circular and stationary binaries and computing the SNR for each.

[27]:
```
n_values = 500
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.uniform(0, 30, n_values) * u.kpc
f_orb = 10**(np.random.uniform(-5, -3, n_values)) * u.Hz
ecc = np.repeat(0.0, n_values)
t_obs = 4 * u.yr
weights = np.random.uniform(0, 1, n_values)

sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb,␣
```

(continues on next page)

(continued from previous page)

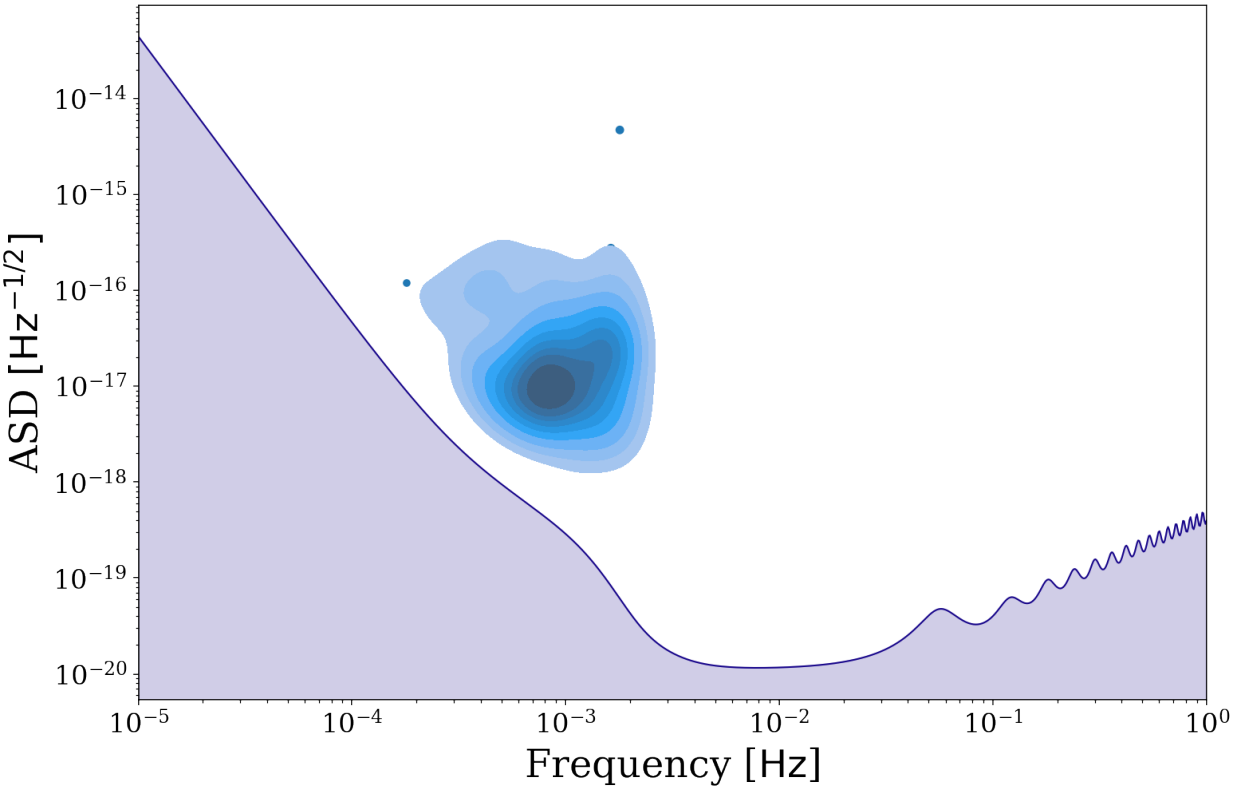
```
↪weights=weights,
                        interpolate_g=False)
```

[28]:
```
circ_stat_snr = sources.get_snr(verbose=True)
```

```
Calculating SNR for 500 sources
        0 sources have already merged
        500 sources are stationary
                500 sources are stationary and circular
```

Now we can plot these binaries on the sensitivity curve using *legwork.source.Source.plot_sources_on_sc()*. Here we only plot those with SNR > 7 and, as earlier, we plot a density distribution for the most common 90% of the distribution and plot the remaining 10% outliers as scatter points. Try changing the value of `thresh` or switching whether to `fill` the KDE.

[29]:
```
fig, ax = sources.plot_sources_on_sc(snr_cutoff=7, show=False)
fig, ax = sources.plot_sources_on_sc(snr_cutoff=7, disttype="kde", thresh=0.1, fill=True,
↪ fig=fig, ax=ax)
```



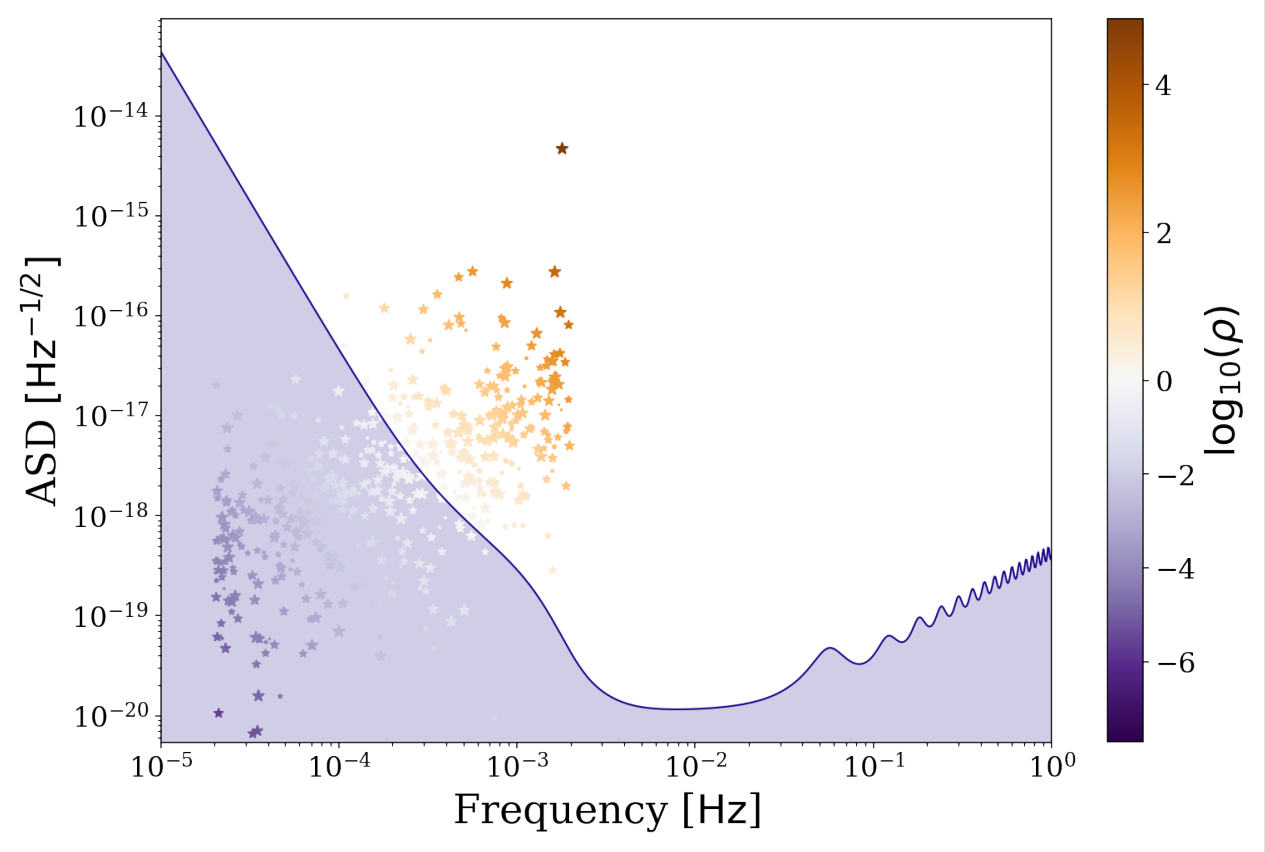
But we may not want to know about only the *detectable* population, why don't we also plot the undetectable binaries. Additionally here we use a colormap to show the SNR of each binary and this map diverges at SNR = 1. Try changing the value of `cutoff` to see how the plot changes.

[30]:
```
# define the detectable parameters
cutoff = 0
detectable_snr = sources.snr[sources.snr > cutoff]
norm = TwoSlopeNorm(vmin=np.log10(np.min(detectable_snr)),
```

(continues on next page)

(continued from previous page)

```
                    vcenter=0, vmax=np.log10(np.max(detectable_snr)))

fig, ax = sources.plot_sources_on_sc(snr_cutoff=cutoff, marker="*", cmap="PuOr_r",
                                     c=np.log10(detectable_snr), show=False, norm=norm,␣
↪scatter_s=50)

# create a colorbar from the scatter points
cbar = fig.colorbar(ax.collections[1])
cbar.ax.set_ylabel(r"$\log_{10} (\rho)$")

plt.show()
```



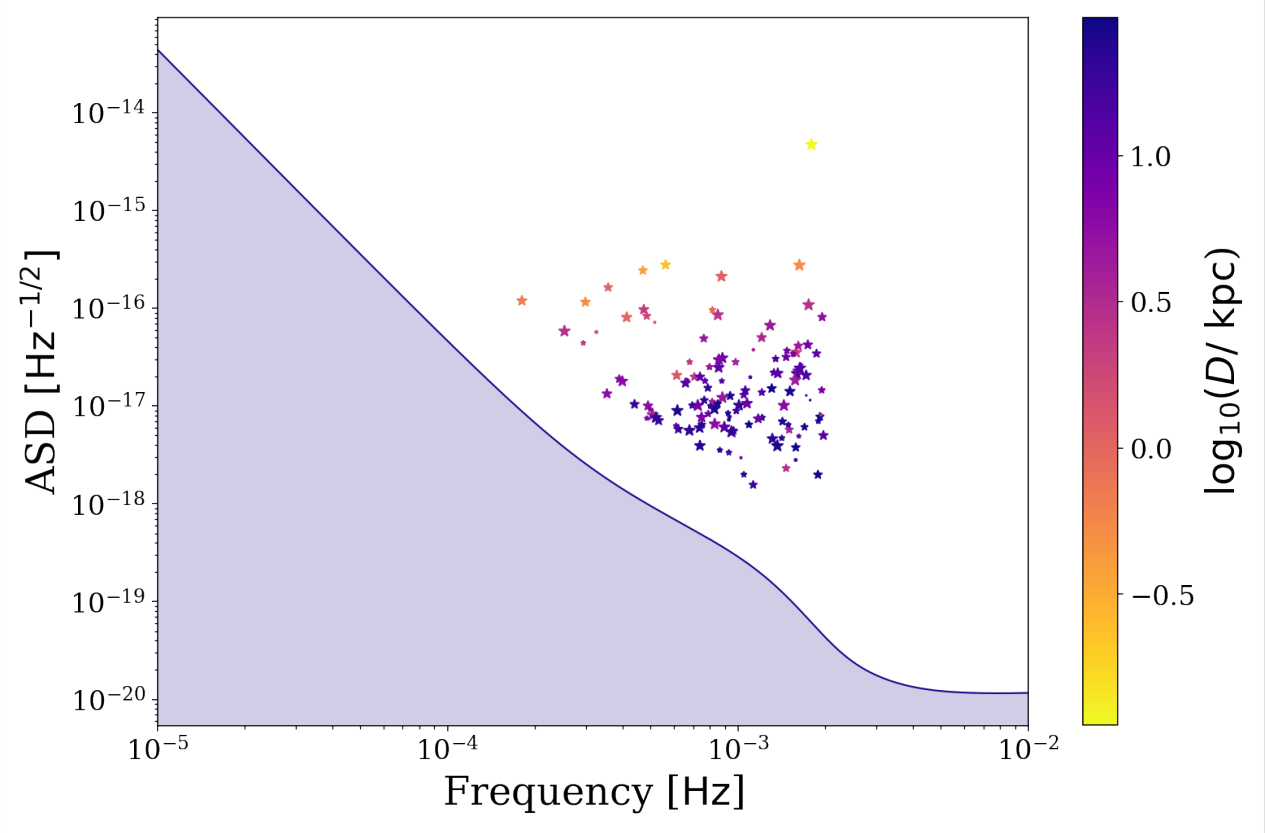
It could also be interesting to see how other parameters are affecting your distribution. Here is the same plot but with only the detectable binaries and coloured by their distance, as well as with a smaller x range. You should hopefully see that the more distant binaries are at higher frequencies and comparatively lower ASDs.

[31]:
```
# define the detectable parameters
cutoff = 7
detectable_dist = sources.dist[sources.snr > cutoff].value

# plot the detectable binaries
fig, ax = sources.plot_sources_on_sc(snr_cutoff=cutoff, marker="*", cmap="plasma_r",
                                     c=np.log10(detectable_dist),
                                     show=False, xlim=(1e-5, 1e-2), scatter_s=50)
# create a colorbar from the scatter points
```

(continues on next page)

(continued from previous page)

```
cbar = fig.colorbar(ax.collections[1])
cbar.ax.set_ylabel(r"{}{:latex})".format("$\log_{{{10}}} (D /$ ", sources.dist.unit))

plt.show()
```



Eccentric and Stationary Binaries

These binaries work in much the same way as circular and stationary binaries. However, now the gravitational wave signal is spread over many harmonics. Therefore the harmonic with the maximum strain or snr may no longer be the $n = 2$ harmonic as with circular binaries. This is best shown in an example.

Let's take a collection of eccentric and stationary sources and show how eccentricity affects the max strain harmonic.

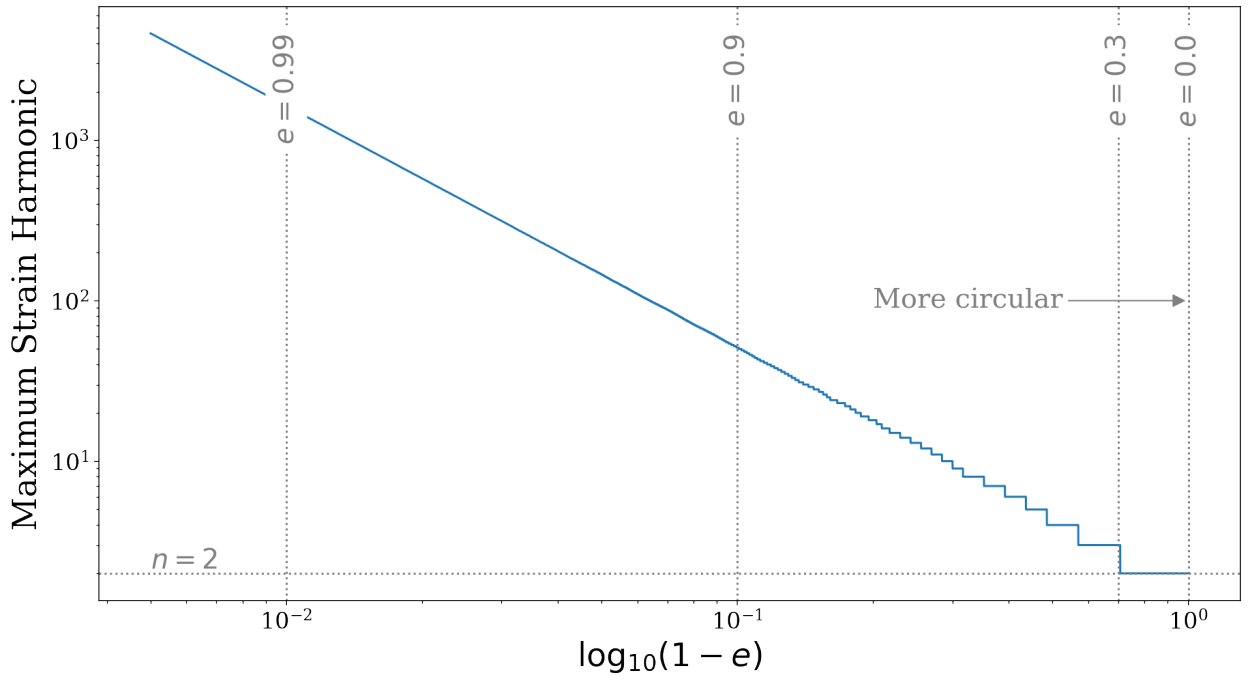
[32]:
```
n_values = 500
m_1 = np.random.uniform(0, 10, n_values) * u.Msun
m_2 = np.random.uniform(0, 10, n_values) * u.Msun
dist = np.random.uniform(0, 30, n_values) * u.kpc
f_orb = 10**(np.random.uniform(-5, -3, n_values)) * u.Hz
ecc = np.random.uniform(0.35, 0.4, n_values)
t_obs = 4 * u.yr
```

[33]:
```
sources = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb, interpolate_
↪g=False)
ecc_stat_snr = sources.get_snr(verbose=True)
```

```
Calculating SNR for 500 sources
        0 sources have already merged
        500 sources are stationary
                500 sources are stationary and eccentric
```

This source Class contains a function that returns the maximum strain harmonic for any given eccentricity.

[34]:
```
e_range = np.linspace(0, 0.995, 10000)
msh = sources.max_strain_harmonic(e_range)

fig = plt.figure(figsize=(15, 8))

plt.loglog(1 - e_range, msh)

plt.axhline(2, linestyle="dotted", color="grey")
plt.annotate(r"$n = 2$", xy=(5e-3, 2), va="bottom", fontsize=20, color="grey")

for e, l in [(0, r"$e=0.0$"), (0.3, r"$e=0.3$"), (0.9, r"$e=0.9$"), (0.99, r"$e=0.99$")]:
    plt.axvline(1 - e, linestyle="dotted", color="grey")
    plt.annotate(l, xy=(1 - e, np.max(msh)), rotation=90, ha="center", va="top",
                 fontsize=20, color="grey", bbox=dict(boxstyle="round", fc="white", ec=
↪"none"))

plt.annotate("More circular", xy=(1, 1e2), xytext=(2e-1, 1e2), fontsize=20, va="center",␣
↪color="grey",
             arrowprops=dict(arrowstyle="-|>", fc="grey", ec="grey"))

plt.xlabel(r"$\log_{10}(1 - e)$")
plt.ylabel("Maximum Strain Harmonic")

plt.show()
```

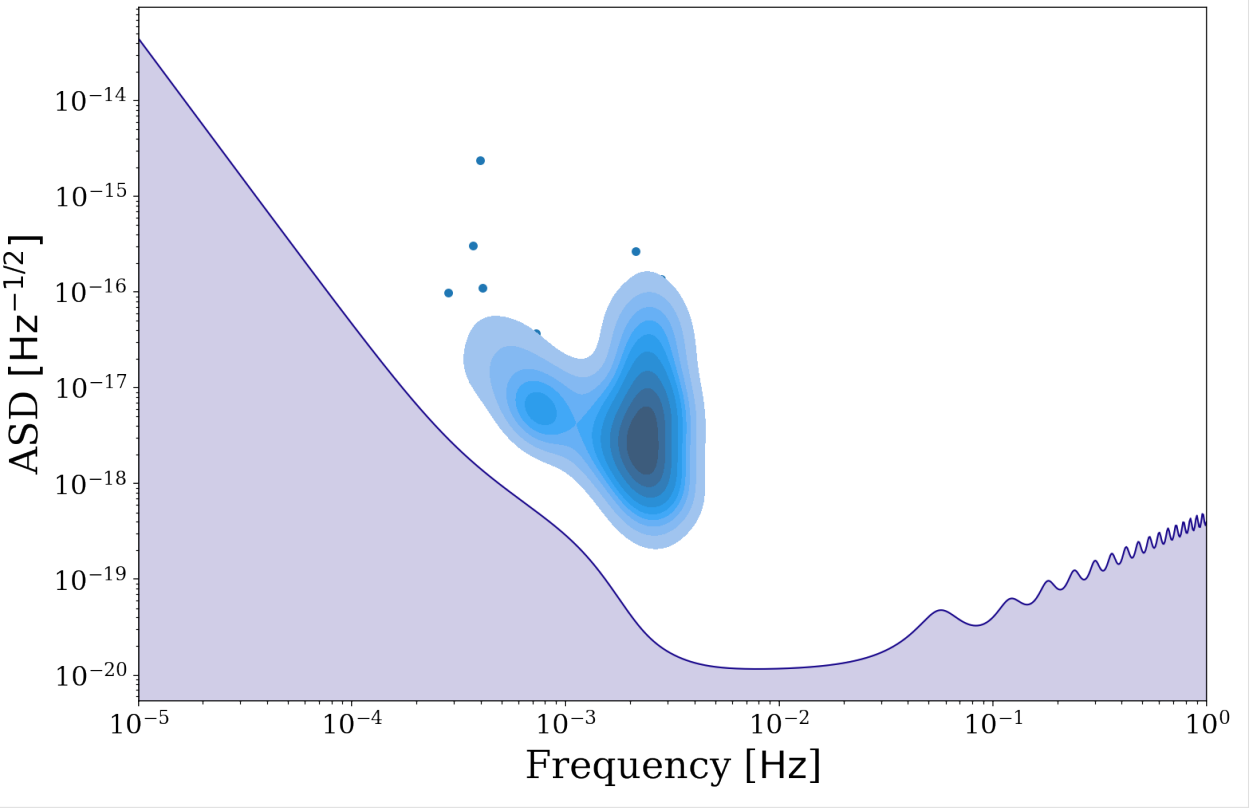


As you can see in the plot above, once the eccentricity is greater than approximately $e = 0.3$, the maximum strain harmonic increases from $n = 2$ to $n = 3$ and this continues as eccentricity increases.

Although it is interesting to see this function, more directly applicable is the maximum *SNR* harmonic. This is useful because we should plot the binary at this harmonic instead of the $n = 2$. This harmonic is calculated for each binary automatically when the SNR is calculated and stored in `self.max_snr_harmonic`.

So now we can plot eccentric binaries on the sensitivity curve at their maximum snr harmonic, such that their height above the sensitivity curve is equal to their total SNR over all harmonics.

[35]:
```
fig, ax = sources.plot_sources_on_sc(snr_cutoff=7, show=False)
fig, ax = sources.plot_sources_on_sc(snr_cutoff=7, disttype="kde", thresh=0.1, fill=True,
                                     fig=fig, ax=ax)
```



We could also repeat this for a different detector like this

[36]:
```
# note that we can the interpolate parameters to TianQin defaults
sources_tq = source.Source(m_1=m_1, m_2=m_2, ecc=ecc, dist=dist, f_orb=f_orb,
                           sc_params={"instrument": "TianQin", "L": np.sqrt(3) * 1e5 * u.
↪km, "t_obs": 5 * u.yr},
                           interpolate_g=False)
ecc_stat_snr_tq = sources_tq.get_snr(instrument="TianQin", verbose=True)
```

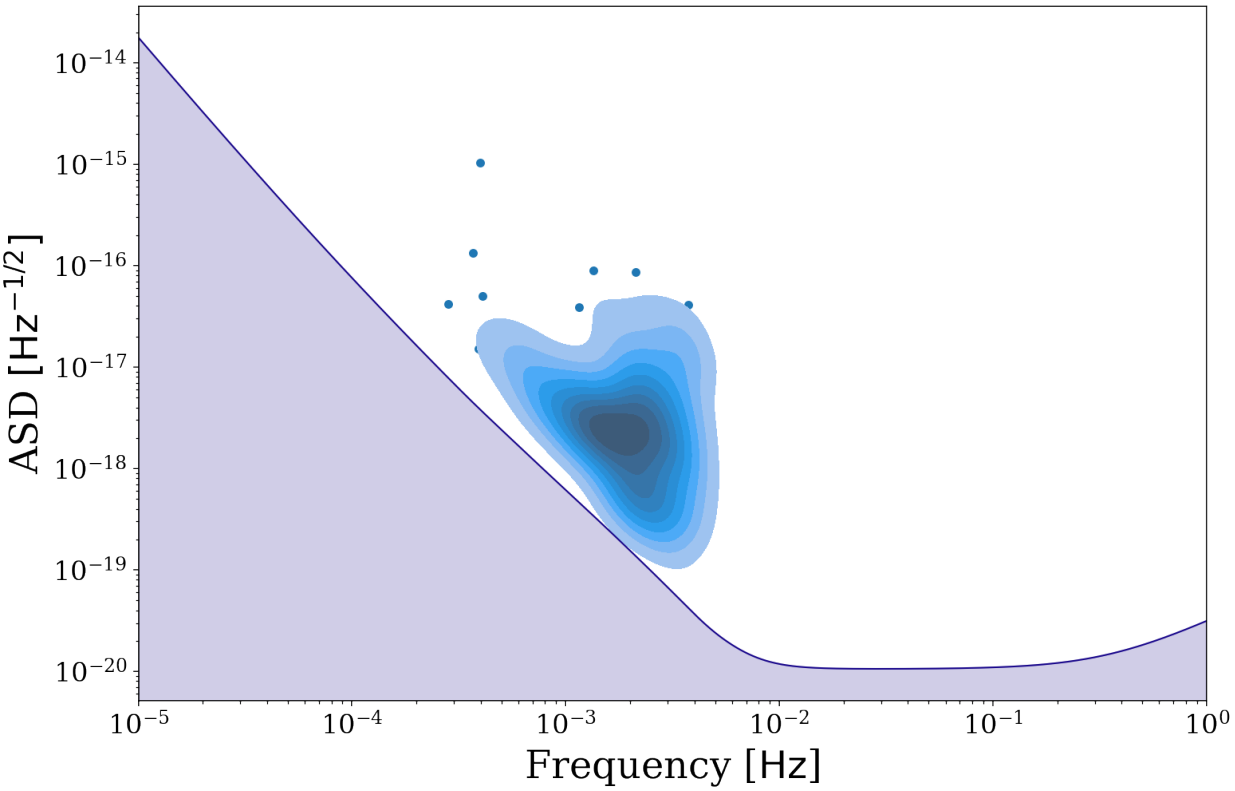
```
Calculating SNR for 500 sources
        0 sources have already merged
        500 sources are stationary
                500 sources are stationary and eccentric
```

[37]:
```
fig, ax = sources_tq.plot_sources_on_sc(snr_cutoff=7, show=False)
```

(continues on next page)

(continued from previous page)

```
fig, ax = sources_tq.plot_sources_on_sc(snr_cutoff=7, disttype="kde", thresh=0.1,␣
↪fill=True,
                                        fig=fig, ax=ax, show=False)
```



This concludes our tutorial for learning about the visualisation module of LEGWORK, be sure to check out *our other tutorials* to learn more about the exciting features that LEGWORK has to offer.

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

2.6 Understanding units in LEGWORK

In this tutorial, we explain how units work in LEGWORK, how to create proper input for functions and how to convert output to units of your choice. Note that LEGWORK uses the `astropy.units` module for units and so this tutorial draws heavily from their documentation!

We've this tutorial up as a sort of FAQ so feel free to skip to the most relevant part!

Let's start by important the LEGWORK source module as well as `astropy.units` and numpy.

[1]:
```
import legwork.source as source
import astropy.units as u

import numpy as np
```

2.6.1 What units can I use?

We follow the Standard Units defined in Astropy. This means that

- **lengths** are defined in terms of **metres** (or equivalent units)
- **masses** are defined in terms of **kilograms** (or equivalent units)
- **times** are defined in terms of **seconds** (or equivalent units)

However, if you're planning to try to measure the gravitational waves from a source for which kilograms is a sensible unit for the mass, I've got some bad news for you...

Therefore, for LEGWORK you are most likely to focus on the following units:

- mass: $\mathrm{M}_\odot$, accessed via `u.Msun`
- frequency: Hz, accessed via `u.Hz`
- distance: $\mathrm{kpc}, \mathrm{Mpc}, \mathrm{Gpc}$, accessed via `u.kpc`, `u.Mpc`, `u.Gpc`
- separation: AU, accessed via `u.AU` or perhaps $\mathrm{R}_\odot$, accessed via `u.Rsun`
- ages: $\mathrm{yr}, \mathrm{Myr}, \mathrm{Gyr}$, accessed via `u.yr`, `u.Myr`, `u.Gyr`

But that doesn't mean you *have* to use these units because of the flexibility of Astropy. LEGWORK will accept any equivalent unit to those listed above.

Astropy provides a very convenient method for getting equivalent units. Say you know you could input the mass of a source in kilograms but you know that this isn't the best unit. You can find some equivalent choices by running

[2]:
```
u.kg.find_equivalent_units()
```

[2]:
```
  Primary name | Unit definition | Aliases
[
  M_e          | 9.10938e-31 kg  |                                ,
  M_p          | 1.67262e-27 kg  |                                ,
  earthMass    | 5.97217e+24 kg  | M_earth, Mearth                ,
  g            | 0.001 kg        | gram                           ,
  jupiterMass  | 1.89812e+27 kg  | M_jup, Mjup, M_jupiter, Mjupiter ,
  kg           | irreducible     | kilogram                       ,
  solMass      | 1.98841e+30 kg  | M_sun, Msun                    ,
  t            | 1000 kg         | tonne                          ,
  u            | 1.66054e-27 kg  | Da, Dalton                     ,
]
```

And thus you can see that you could even use the mass of an electron (`u.M_e`) as your unit if that is your heart's desire.

2.6.2 How do I give a variable units?

Okay great, so you know what unit you want to use, now you just need to apply it to a variable. Say you have a list of masses that looks like this

[3]:
```
# a list of masses
masses = [1.0, 1.4, 10, 30, 50]
print(masses)
```

```
[1.0, 1.4, 10, 30, 50]
```

and you know that each mass is in terms of solar masses. To make sure LEGWORK knows this you multiply your variable by the unit.

[4]:
```
masses_with_units = masses * u.Msun
print(masses_with_units)
```

```
[ 1.   1.4 10.  30.  50. ] solMass
```

And…that's it! Your input has been transformed into an Astropy Quantity rather than a simple Python list and you're good to go!

2.6.3 Could you show me an example of using units with LEGWORK input?

Well, how could I say no when you asked so nicely? Let's create a collection sources and get their SNRs.

[5]:
```
# let's define the primary in solar masses
m_1 = [10, 12, 30] * u.Msun

# and the secondary in electron masses (because why not)
m_2 = [1e60, 5e60, 7.5e60] * u.M_e

# then the frequencies are generally defined in terms of Hertz
f_orb = [1e-3, 1e-4, 1e-2] * u.Hz

# and the distances with kiloparsecs
dist = [1, 8, 50] * u.kpc

# finally, eccentricity has no unit
ecc = [0.7, 0.0, 0.2]

sources = source.Source(m_1=m_1, m_2=m_2, f_orb=f_orb, dist=dist, ecc=ecc, interpolate_
↪g=False)
```

Then if we ask the class for the signal-to-noise ratio it will handle the units cleanly and fully simplify.

[6]:
```
sources.get_snr()
```

[6]:
```
array([3.76050048e+03, 8.61372971e-01, 4.14167952e+03])
```

Be careful though, if you don't use correct units then you'll get a `UnitConversionError` that may be hard to isolate.

[7]:
```
try:
    # give frequency units of length
    f_orb = f_orb.value * u.m
    # try to create a source
    sources = source.Source(m_1=m_1, m_2=m_2, f_orb=f_orb, dist=dist, ecc=ecc,␣
↪interpolate_g=False)
except u.UnitConversionError as error:
    print(error)
```

```
'm(1/3) solMass(1/3) / (kg(1/3) s(2/3))' and 'AU' (length) are not convertible
```

2.6.4 How do you convert between units?

Great work, if you've got this far then you can now provide input to LEGWORK with any unit of your choice.

But what about the output? LEGWORK tries to choose some sensible units for the output but maybe you want something else and can't for the life of you remember the difference between a kiloparsec and a light year. Never fear, Astropy has you covered!

In order to convert between units you can use the `.to()` method of an Astropy quanitity. Let's get the merger times of the sources that we defined in the earlier example and convert the result to different units.

[8]:
```
# get the merger times
t_merge = sources.get_merger_time()

# by default LEGWORK uses Gyr for merger times
t_merge
```

[8]: $[1.4564593 \times 10^{-5},\ 0.013231627,\ 1.8741491 \times 10^{-8}]$ Gyr

[9]:
```
# but we could look at how many years this is
t_merge.to(u.yr)
```

[9]: $[14564.593,\ 13231627,\ 18.741491]$ yr

[10]:
```
# or maybe you just really want to know how many fortnights until your favourite source␣
↪merges
t_merge.to(u.fortnight)
```

[10]: $[379979.83,\ 3.4520369 \times 10^{8},\ 488.95211]$ fortnight

You can also convert to any *combination* of units as long as they simplify to an equivalent unit.

[11]:
```
# let's convert to a combination of units
t_merge.to(u.yr * u.Msun / u.kg)
```

[11]: $[7.3247438 \times 10^{-27},\ 6.6543759 \times 10^{-24},\ 9.425366 \times 10^{-30}]$ $\frac{\mathrm{M}_\odot\ \mathrm{yr}}{\mathrm{kg}}$

Beware though, if you try to convert to a unit isn't equivalent then you'll get an `UnitConversionerror`

[12]:
```
try:
    t_merge.to(u.yr * u.Msun)
except u.UnitConversionError as error:
    print(error)
```

```
'Gyr' (time) and 'solMass yr' are not convertible
```

2.6.5 How do I decompose a variable's value and unit?

So you've got the result and now the pesky unit is perhaps getting in the way of saving the result or doesn't work with another of your functions. If you want to get the value back then just use `.value` like this

[13]:
```
masses_with_units = [1.0, 1.4, 10, 30, 50] * u.Msun
print(masses_with_units)
print(masses_with_units.value)
```

```
[ 1.   1.4 10.  30.  50. ] solMass
[ 1.   1.4 10.  30.  50. ]
```

You can also use `.unit` to get the unit of a variable (this can be very useful when plotting and labelled axes)

[14]:
```
print(masses_with_units)
print(masses_with_units.unit)
```

```
[ 1.   1.4 10.  30.  50. ] solMass
solMass
```

That's all for this tutorial, be sure to check out *the other ones* to find *other* ways to keep your feet up and let us do the LEGWORK! If you still have questions about units we recommend that you take a look at the Astropy documentation directly.

CHAPTER

THREE

DEMOS

This section contains a series of demos that highlight LEGWORK's capabilities with some example use cases. We'll continue to update this page with new demos and if you've got an idea for a great use case for LEGWORK please open a pull request and show off your new demo!

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.1 Demo - Basic SNR Calculation

This demo shows how you can use LEGWORK to compute the SNR of a single binary system, as well as a collection of systems.

[1]:
```
import legwork as lw
import astropy.units as u
```

3.1.1 Single source SNR calculation

The most basic use case of LEGWORK is to calculate the signal-to-noise ratio for a single stellar-mass binary system. Let's create a toy source and calculate its SNR (for a 4-year LISA mission by default).

[3]:
```
source = lw.source.Source(m_1=10 * u.Msun,
                          m_2=10 * u.Msun,
                          ecc=0.2,
                          f_orb=1e-4 * u.Hz,
                          dist=8 * u.kpc,
                          interpolate_g=False)
source.get_snr().round(2)
```

[3]:
```
array([4.49])
```

That's it! Behind the scenes LEGWORK has checked whether the source is eccentric/circular and evolving/stationary and picked the fastest possible way to calculate the SNR accurately.

3.1.2 Population of sources SNR calculation

If we want to know the SNR of three (or any number of) sources then you can instead provide arrays for each of the arguments and execute the code in exactly the same way.

[4]:
```
# supply arrays that are identical to earlier example but different primary masses
sources = lw.source.Source(m_1=[5, 10, 20] * u.Msun,
                           m_2=[10, 10, 10] * u.Msun,
                           ecc=[0.2, 0.2, 0.2],
                           f_orb=[1e-4, 1e-4, 1e-4] * u.Hz,
                           dist=[8, 8, 8] * u.kpc,
                           interpolate_g=False)

sources.get_snr().round(2)
```

[4]:
```
array([2.47, 4.49, 7.85])
```

3.1.3 Change sensitivity curve parameters

If you want to know the SNR of these sources in a different type of detector you can also specify this with `sc_params`. Let's repeat the same calculation but now find the SNR in the TianQin detector in a 5-year mission but excluding any confusion noise.

[9]:
```
# supply arrays that are identical to earlier example but different primary masses
sources = lw.source.Source(m_1=[5, 10, 20] * u.Msun,
                           m_2=[10, 10, 10] * u.Msun,
                           ecc=[0.2, 0.2, 0.2],
                           f_orb=[1e-4, 1e-4, 1e-4] * u.Hz,
                           dist=[8, 8, 8] * u.kpc,
                           interpolate_g=False,
                           sc_params={
                               "instrument": "TianQin",
                               "t_obs": 5 * u.yr,
                               "confusion_noise": None
                           })

sources.get_snr().round(2)
```

[9]:
```
array([1.07, 1.95, 3.41])
```

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.2 Demo - The Role of Eccentricity

This demo uses LEGWORK to illustrate the role of eccentricity in the detectability of a gravitational wave source in LISA.

[2]:
```
import legwork as lw
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rc('font', family='serif')
plt.rcParams['text.usetex'] = False
fs = 24

# update various fontsizes to match
params = {'figure.figsize': (12, 8),
          'legend.fontsize': fs,
          'axes.labelsize': fs,
          'xtick.labelsize': 0.9 * fs,
          'ytick.labelsize': 0.9 * fs,
          'axes.linewidth': 1.1,
          'xtick.major.size': 7,
          'xtick.minor.size': 4,
          'ytick.major.size': 7,
          'ytick.minor.size': 4}
plt.rcParams.update(params)
```

3.2.1 How does eccentricity affect the detectability of a source?

Demonstration of trends with LEGWORK

Eccentricity plays a complex role in the detection of LISA sources. Although eccentric binaries emit stronger gravitational waves, they also shift the emission to higher frequency harmonics and cause a faster inspiral. These effects in tandem produce some interesting trends that we can demonstrate with LEGWORK.

Let's create three toy sources with different eccentricities and see how their eccentricities change.

[4]:
```
# set eccentricities
ecc = np.array([1e-6, 0.6, 0.9])
n_binaries = len(ecc)

# use constant values for mass, f_orb and distance
m_1 = np.repeat(0.6, n_binaries) * u.Msun
m_2 = np.repeat(0.6, n_binaries) * u.Msun
f_orb = np.repeat(1.5e-3, n_binaries) * u.Hz
dist = np.repeat(15, n_binaries) * u.kpc

# get the SNR in each harmonic
snr2_n = lw.snr.snr_ecc_evolving(m_1=m_1, m_2=m_2, f_orb_i=f_orb, ecc=ecc, dist=dist,
                                 harmonics_required=100, t_obs=4 * u.yr, n_step=1000,
                                 ret_snr2_by_harmonic=True)
```

(continues on next page)

(continued from previous page)

```
snr = snr2_n.sum(axis=1)**(0.5)
print(snr)
```

```
[31.73932899 50.18790614 38.77725186]
```

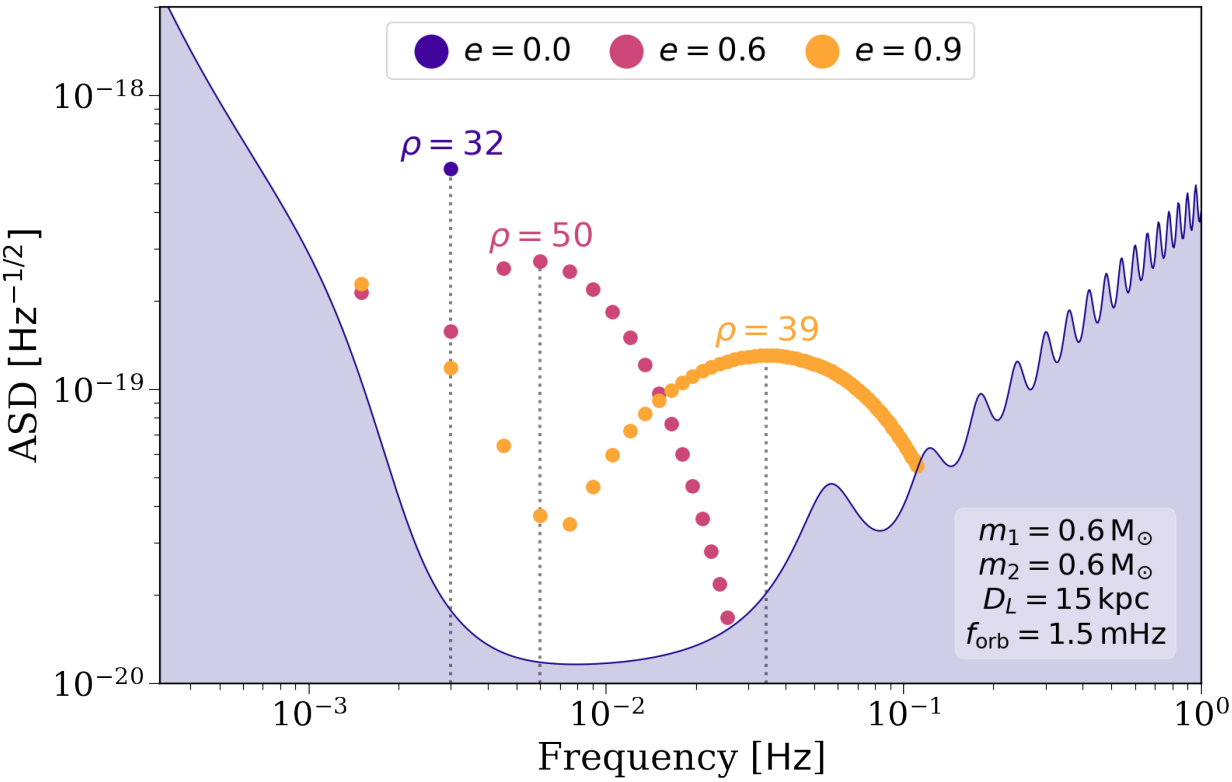
We can see from this that increasing the eccentricity does not always increase the SNR. This can be better understood by plotting the SNR of each harmonic of each source and seeing how the distribution shifts.

[5]:
```
# plot LISA sensitivity curve
fig, ax = lw.visualisation.plot_sensitivity_curve(frequency_range=np.logspace(-3.5, 0,␣
↪1000) * u.Hz,
                                                  show=False)

# plot each sources
colours = [plt.get_cmap("plasma")(i) for i in [0.1, 0.5, 0.8]]
for i in range(len(snr2_n)):
    # work out the harmonic frequencies and ASDs
    f_harm = f_orb[i] * range(1, len(snr2_n[0]) + 1)
    y_vals = lw.psd.lisa_psd(f_harm)**(0.5) * np.sqrt(snr2_n)[i]

    # only plot points above the sensitivity curve
    mask = np.sqrt(snr2_n)[i] > 1.0

    # compute the index of the maximal SNR value
    max_index = np.argmax(y_vals[1:]) + 1

    # plot each harmonic
    ax.scatter(f_harm[mask], y_vals[mask],
               s=70, color=colours[i],
               label=r"$e={{{:1.1f}}}$".format(ecc[i]))

    # annotate each source with its SNR at the max SNR value
    ax.annotate(r"$\rho={{{:1.0f}}}$".format(snr2_n[i].sum()**(0.5)),
                xy=(f_harm[max_index].value, y_vals[max_index].value * 1.05),
                ha="center", va="bottom", fontsize=0.9*fs, color=colours[i])
    # plot a dotted line to highlight where the signal is concentrated
    ax.plot([f_harm[max_index].value] * 2, [1e-20, y_vals[max_index].value],
               color="grey", linestyle="dotted", lw=2, zorder=0)

# add a legend and annotate the other source properties
ax.legend(markerscale=2.5, handletextpad=0.0, ncol=3, loc="upper center",
          columnspacing=0.75, fontsize=0.85 * fs)

annotation_string = r"$m_1 = 0.6 \, {\rm M_{\odot}}$"
annotation_string += "\n"
annotation_string += r"$m_2 = 0.6 \, {\rm M_{\odot}}$"
annotation_string += "\n"
annotation_string += r"$D_L = 15 \, {\rm kpc}$"
annotation_string += "\n"
annotation_string += r"$f_{\rm orb} = 1.5 \, {\rm mHz}$"

ax.annotate(annotation_string, xy=(3.5e-1, 1.3e-20), ha="center", va="bottom",␣
```

(continues on next page)

(continued from previous page)

```
↪fontsize=0.8 * fs,
            bbox=dict(boxstyle="round", fc="white", ec="white", alpha=0.3))

ax.set_ylim(1e-20, 2e-18)

plt.show()
```



Explanation of the trends

We see two effects in the signal-to-noise ratio here. First, increasing the eccentricity from essentially circular to $e = 0.6$ results in a higher signal-to-noise ratio ($\rho = 31.7 \rightarrow \rho = 50.2$). This is because an eccentric binary has enhanced energy emission via gravitational waves. This means that an eccentric binary will not only inspiral faster than an otherwise identical circular binary, but also will always have a stronger gravitational wave strain.

The second effect is more intriguing. We see that increasing the eccentricity from $e = 0.6$ to $e = 0.9$ results in a relative *decrease* in SNR ($\rho = 50.2 \rightarrow \rho = 38.8$). The reason for this is that eccentric binaries emit gravitational waves at many harmonic frequencies (unlike circular binaries, which emit predominantly twice the orbital frequency). This leads to the gravitational wave signal being diluted over many frequencies higher than the orbital frequency, where the higher the eccentricity, the more harmonics are required to capture all of the gravitational luminosity. Therefore, if the eccentricity is too high, the majority of the signal may be emitted at a frequency to which LISA is less sensitive.

From the plot, we can better understand why a source with $e = 0.9$ has a lower SNR than the same source with $e = 0.6$. From the dotted lines, we can note that the signal from the $e = 0.9$ source is concentrated at a frequency of around 4×10^{-2} Hz. The LISA sensitivity at this point is much weaker than the 6×10^{-3} Hz at which the $e = 0.6$ source is concentrated. Therefore, although the strain from a more eccentric binary is stronger, the SNR is lower due to the increased noise in the LISA detector.

Overall, we can therefore conclude that for LISA sources of this nature, higher eccentricity will produce more detectable

binaries only if the orbital frequency is not already at or above the minimum of the LISA sensitivity curve. Another consideration for more massive binaries is whether the increased eccentricity will cause the binary to merge before the mission ends, which would cause a significant decrease in signal-to-noise ratio.

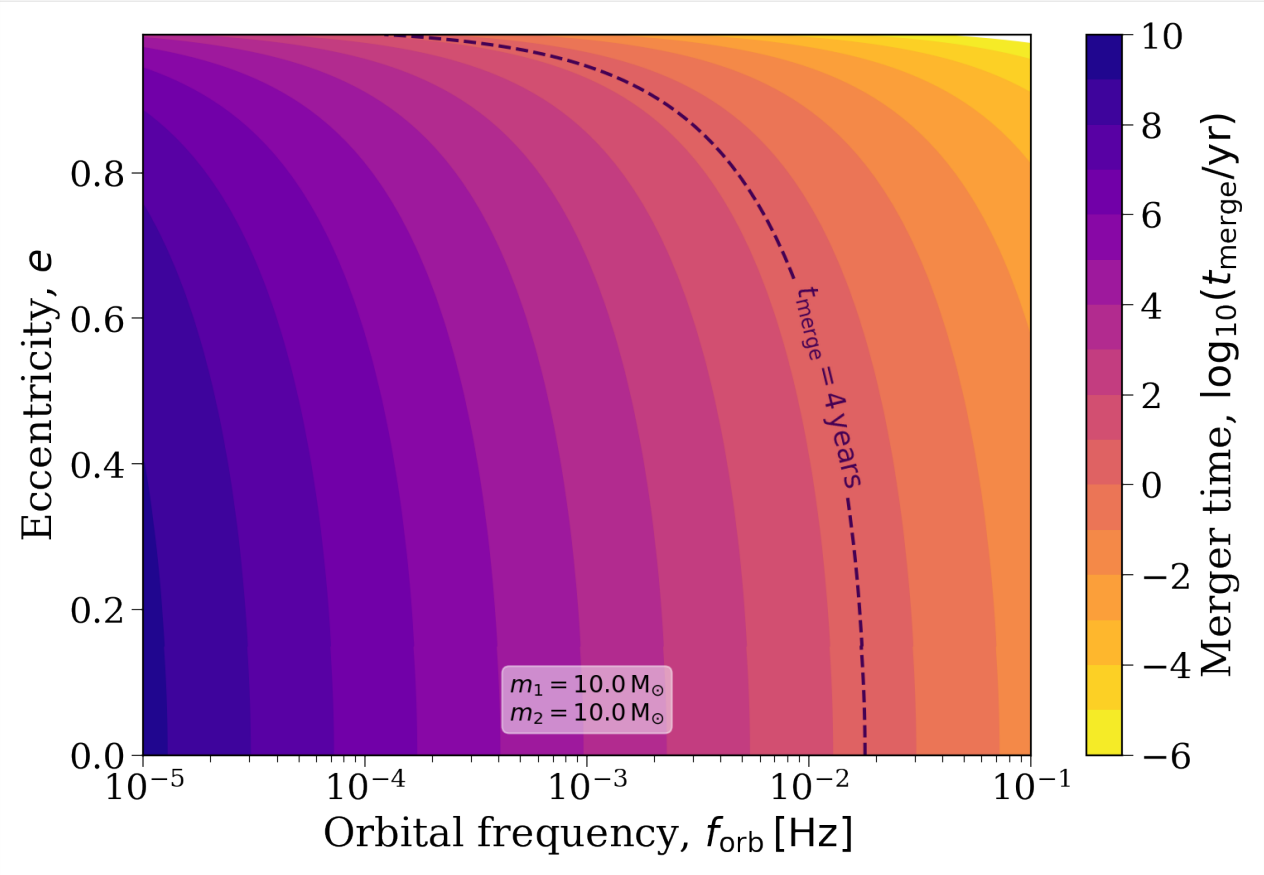
3.2.2 How does eccentricity affect the merger time?

As mentioned above, eccentric binaries will merge more quickly than circular ones and we can show this with LEGWORK.

Below, we make a grid of frequencies and eccentricities, flatten them and calculate their merger times, before reshaping and plotting the result.

[6]:
```
f_range = np.logspace(-5, -1, 100) * u.Hz
e_range = np.linspace(0, 0.99, 500)

m_1 = np.repeat(10, len(f_range) * len(e_range)) * u.Msun
m_2 = np.repeat(10, len(f_range) * len(e_range)) * u.Msun

F, E = np.meshgrid(f_range, e_range)

t_merge = lw.evol.get_t_merge_ecc(ecc_i=E.flatten(), f_orb_i=F.flatten(), m_1=m_1, m_2=m_
↪2,
                                  small_e_tol=0.15, large_e_tol=0.9999).reshape(F.shape)
```

[7]:
```
fig, ax = plt.subplots()

cont = ax.contourf(F, E, np.log10(t_merge.to(u.yr).value), cmap="plasma_r", levels=np.
↪linspace(-6, 10, 17))
cbar = fig.colorbar(cont, label=r"Merger time, $\log_{10} (t_{\rm merge} / {\rm yr})$")
ax.set_xscale("log")

# hide edges that show up in rendered PDFs
for c in cont.collections:
    c.set_edgecolor("face")

mass_string = ""
mass_string += r"$m_1 = {{{}}} \, {{ \rm M_{{\odot}}}}$".format(m_1[0].value)
mass_string += "\n"
mass_string += r"$m_2 = {{{}}} \, {{ \rm M_{{\odot}}}}$".format(m_2[0].value)
ax.annotate(mass_string, xy=(0.5, 0.04), xycoords="axes fraction", fontsize=0.6*fs,
            bbox=dict(boxstyle="round", color="white", ec="white", alpha=0.5), ha="center
↪", va="bottom")

ax.set_xlabel(r"Orbital frequency, $f_{\rm orb} \, [\rm Hz]$")
ax.set_ylabel(r"Eccentricity, $e$")

mission_length = ax.contour(F, E, np.log10(t_merge.to(u.yr).value), levels=np.log10([4]),
                            linestyles="--", linewidths=2)
ax.clabel(mission_length, fmt={np.log10(4): r"$t_{\rm merge} = 4\,{\rm years}$"},␣
↪fontsize=0.7*fs, manual=[(1e-2, 0.5)])

plt.show()
```



Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.3 Demo - Compare sensitivity curves

This demo shows how you can use LEGWORK to compare different detector sensitivitity curves. We illustrate these sensitivity curves as well as how the SNR of a source changes in different detectors.

[2]:
```
import legwork
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
from matplotlib.colors import TwoSlopeNorm

from copy import copy
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rc('font', family='serif')
plt.rcParams['text.usetex'] = False
```

(continues on next page)

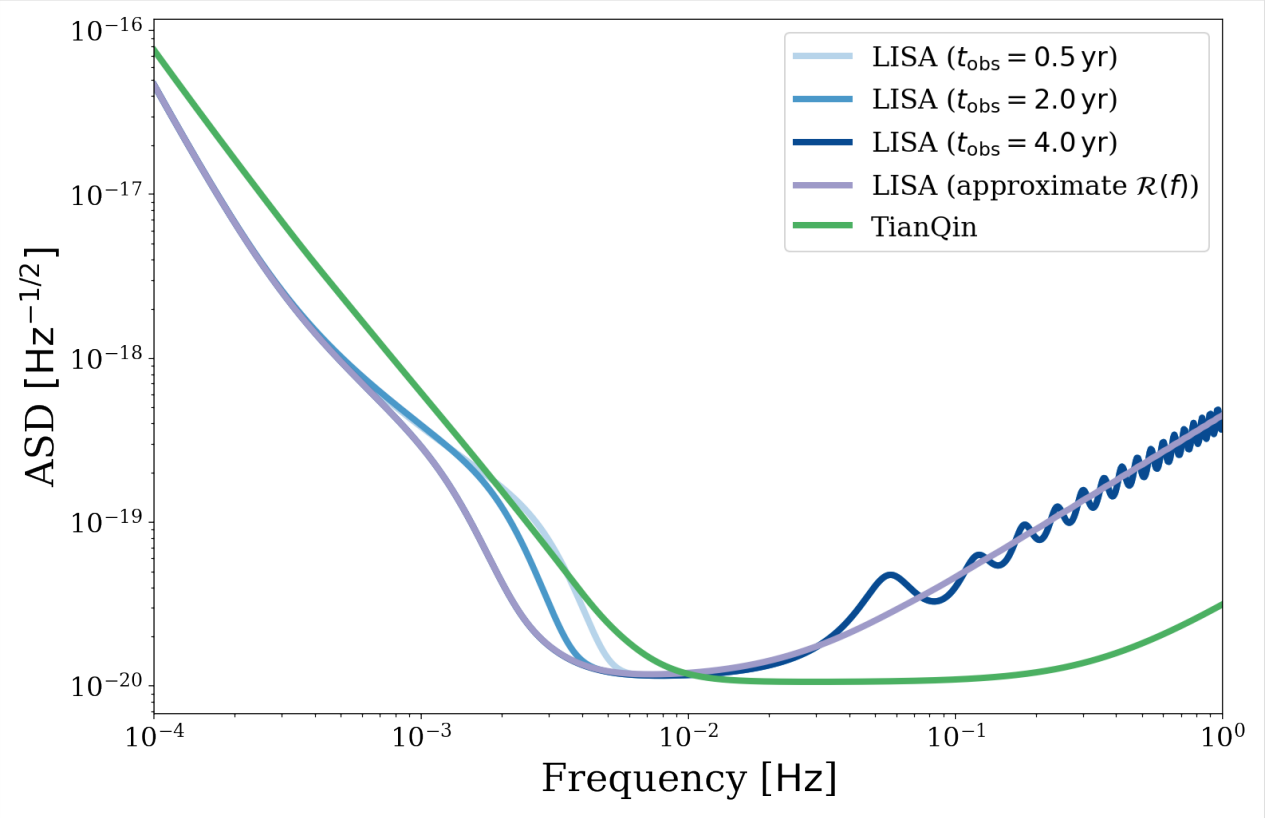
(continued from previous page)

```
fs = 24

# update various fontsizes to match
params = {'figure.figsize': (12, 8),
          'legend.fontsize': fs,
          'axes.labelsize': fs,
          'xtick.labelsize': 0.7 * fs,
          'ytick.labelsize': 0.7 * fs}
plt.rcParams.update(params)
```

3.3.1 Direct sensitivity curve comparisons

First, we can plot the sensitivity curves directly to see which regimes are better for different detectors. Let's look at different LISA specifications as well as the TianQin detector.

[4]:
```
# create new figure
fig, ax = plt.subplots()

# define the frequency range of interest
fr = np.logspace(-4, 0, 1000) * u.Hz

# plot a sensitivity curve for different mission lengths
linewidth = 4
for i, t_obs in enumerate([0.5, 2.0, 4.0]):
    legwork.visualisation.plot_sensitivity_curve(frequency_range=fr, t_obs=t_obs * u.yr,
                                                 fig=fig, ax=ax, show=False, fill=False,␣
↪linewidth=linewidth,
                                                 color=plt.get_cmap("Blues")((i + 1) * 0.
↪3),
                                                 label=r"LISA ($t_{{\rm obs}} = {{{}}} \,
↪ {{\rm yr}}$)".format(t_obs))

# plot the LISA curve with an approximate response function
legwork.visualisation.plot_sensitivity_curve(frequency_range=fr, approximate_R=True,
                                             fig=fig, ax=ax, show=False, fill=False,␣
↪linewidth=linewidth,
                                             color=plt.get_cmap("Purples")(0.5),
                                             label=r"LISA (approximate $\mathcal{R}(f)$)
↪")

# plot the TianQin curve
legwork.visualisation.plot_sensitivity_curve(frequency_range=fr, instrument="TianQin",␣
↪label="TianQin",
                                             fig=fig, ax=ax, show=False,␣
↪linewidth=linewidth,
                                             color=plt.get_cmap("Greens")(0.6),␣
↪fill=False)

ax.legend(fontsize=0.7*fs)

plt.show()
```



3.3.2 SNR in different detectors

But we need not just look at the sensitivity curve, we could also investigate in what regimes of frequency and eccentricity each detector is superior. Let's compare a LISA to TianQin with a grid of sources across eccentricity and frequency space.

[5]:
```
# spread out some frequencies and eccentricities
f_orb_s = np.logspace(-4, -1, 200) * u.Hz
ecc_s = np.linspace(0, 0.9, 150)

# turn them into a grid
F, E = np.meshgrid(f_orb_s, ecc_s)

# flatten the grid
F_flat, E_flat = F.flatten(), E.flatten()

# put all of the sources at the same distance with the same mass
m_1 = np.repeat(10, len(F_flat)) * u.Msun
m_2 = np.repeat(10, len(F_flat)) * u.Msun
dist = np.repeat(8, len(F_flat)) * u.kpc

# define a set of sources
sources = legwork.source.Source(m_1=m_1, m_2=m_2, f_orb=F_flat, ecc=E_flat, dist=dist,␣
↪gw_lum_tol=1e-3)
sources.get_merger_time()
```

[5]:
$[0.0040682461,\ 0.0037085677,\ 0.0033806889,\ \ldots,\ 1.6679556 \times 10^{-13},\ 1.5204897 \times 10^{-13},\ 1.3860614 \times 10^{-13}]$ Gyr

[6]:
```
# compute the LISA SNR
LISA_snr = copy(sources.get_snr(verbose=True, which_sources=sources.t_merge > 0.1 * u.
↪yr))
```

```
Calculating SNR for 26248 sources
        0 sources have already merged
        13486 sources are stationary
                222 sources are stationary and circular
                13264 sources are stationary and eccentric
        12762 sources are evolving
                158 sources are evolving and circular
                12604 sources are evolving and eccentric
```

[7]:
```
# compute the TianQin SNR
sources.update_sc_params({"instrument": "TianQin"})
TQ_snr = sources.get_snr(verbose=True, which_sources=sources.t_merge > 0.1 * u.yr)
```

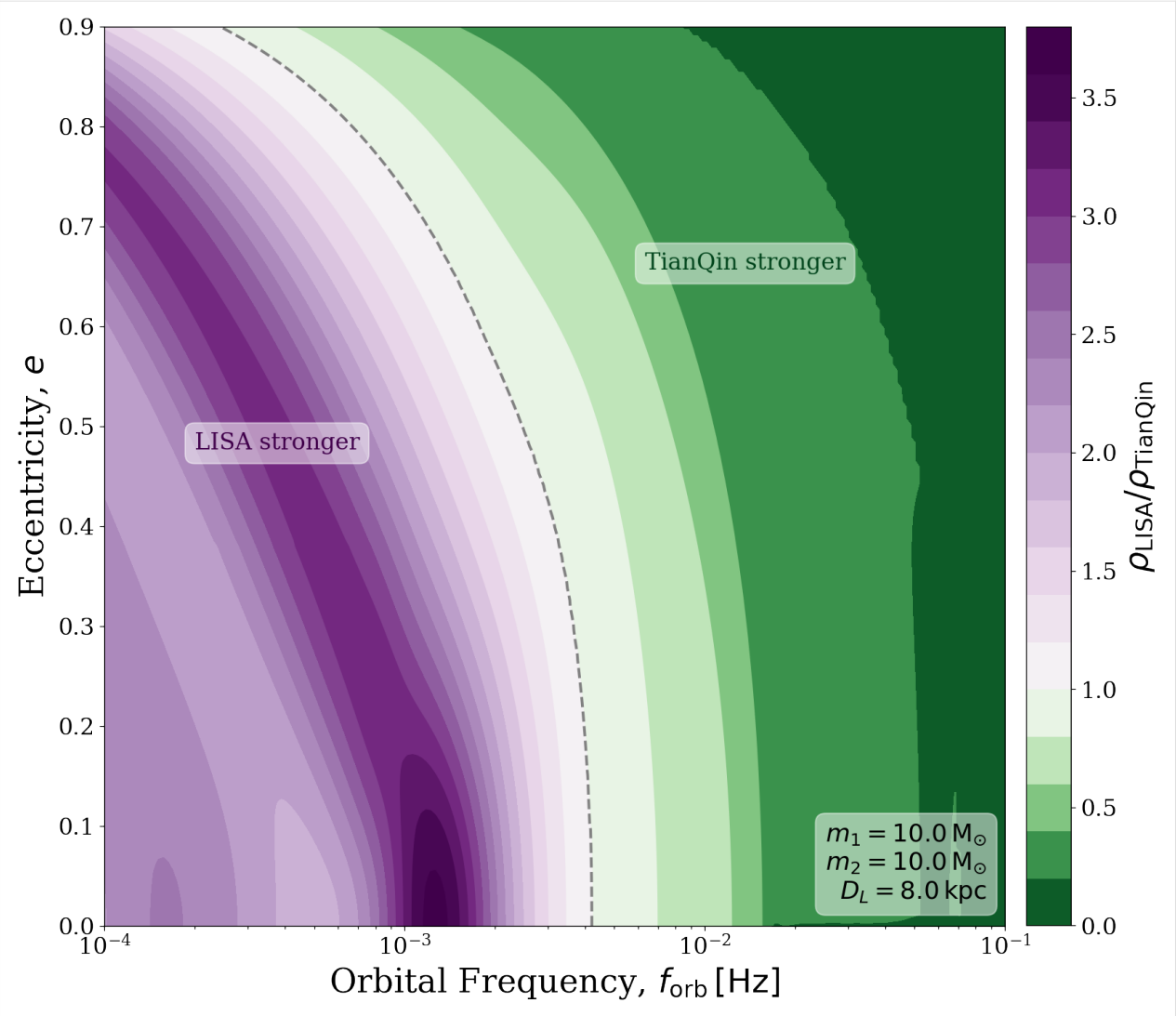
```
Calculating SNR for 26248 sources
        0 sources have already merged
        13126 sources are stationary
                216 sources are stationary and circular
                12910 sources are stationary and eccentric
        13122 sources are evolving
                164 sources are evolving and circular
                12958 sources are evolving and eccentric
```

[10]:
```
# create a figure
fig, ax = plt.subplots(figsize=(14, 12))
ax.set_xscale("log")
ax.set_xlabel(r"Orbital Frequency, $f_{\rm orb} \, [{\rm Hz}]$")
ax.set_ylabel(r"Eccentricity, $e$")

ratio = np.zeros_like(LISA_snr)
ratio[LISA_snr > 0] = (LISA_snr[LISA_snr > 0] / TQ_snr[LISA_snr > 0])
ratio = ratio.reshape(F.shape)

# make contours of the ratio of SNR
ratio_cont = ax.contourf(F, E, ratio, cmap="PRGn_r",
                         norm=TwoSlopeNorm(vcenter=1.0, vmin=0.0, vmax=3.6), levels=20)

for c in ratio_cont.collections:
    c.set_edgecolor("face")

# add a line when the SNRs are equal
ax.contour(F, E, ratio, levels=[1.0], colors="grey", linewidths=2.0, linestyles="--")

# add a colourbar
cbar = fig.colorbar(ratio_cont, fraction=2/14, pad=0.02,
                    label=r"$\rho_{\rm LISA} / \rho_{\rm TianQin}$",
```

(continues on next page)

(continued from previous page)

```
                    ticks=np.arange(0, 3.5 + 0.5, 0.5))

# annotate which regions suit each detector
ax.annotate("LISA stronger", xy=(0.1, 0.53), xycoords="axes fraction", fontsize=0.7 * fs,
            color=plt.get_cmap("PRGn_r")(1.0),
            bbox=dict(boxstyle="round", facecolor="white", edgecolor="white", alpha=0.5,␣
↪pad=0.4))
ax.annotate("TianQin stronger", xy=(0.6, 0.73), xycoords="axes fraction", fontsize=0.7 *␣
↪fs,
            color=plt.get_cmap("PRGn_r")(0.0),
            bbox=dict(boxstyle="round", facecolor="white", edgecolor="white", alpha=0.5,␣
↪pad=0.4))

# annotate with source details
source_string = r"$m_1 = {{{}}} \, {{ \rm M_{{\odot}}}}$".format(m_1[0].value)
source_string += "\n"
source_string += r"$m_2 = {{{}}} \, {{ \rm M_{{\odot}}}}$".format(m_1[0].value)
source_string += "\n"
source_string += r"$D_L = {{{}}} \, {{ \rm kpc}}$".format(dist[0].value)
ax.annotate(source_string, xy=(0.98, 0.03), xycoords="axes fraction", ha="right",␣
↪fontsize=0.75*fs,
            bbox=dict(boxstyle="round", facecolor="white", edgecolor="white", alpha=0.5,␣
↪pad=0.4))

plt.show()
```



From this plot we can see that for circular sources at low frequenies LISA produces a stronger SNR. As eccentricity is increases, the range of frequencies at which LISA is superior decreases until at $e = 0.9$, TianQin is better at every frequency greater than approximately 0.3mHz!

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.4 Demo - LISA Horizon Distance

This demo shows how to use LEGWORK to compute the horizon distance for a collection of sources.

[2]:
```
import legwork as lw
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rc('font', family='serif')
plt.rcParams['text.usetex'] = False
fs = 24

# update various fontsizes to match
params = {'figure.figsize': (12, 8),
          'legend.fontsize': fs,
          'axes.labelsize': fs,
          'xtick.labelsize': 0.9 * fs,
          'ytick.labelsize': 0.9 * fs,
          'axes.linewidth': 1.1,
          'xtick.major.size': 7,
          'xtick.minor.size': 4,
          'ytick.major.size': 7,
          'ytick.minor.size': 4}
plt.rcParams.update(params)
```

3.4.1 Horizon distance of circular binaries

The horizon distance for a source is the maximum distance at which the SNR of a source is still above some detectable threshold. The horizon distance can be computed from the SNR as follows since it is inversely proportional to the distance.

$$D_{\rm hor} = \frac{\rho(D)}{\rho_{\rm detect}} \cdot D, \tag{3.1}$$

Where $\rho(D)$ is the SNR at some distance D and $\rho_{\rm detect}$ is the SNR above which we consider a source detectable.

Let's start doing this by creating a grid of chirp masses and orbital frequencies and creating a Source class from them.

[4]:
```
# create a list of masses and frequencies
m_c_grid = np.logspace(-1, np.log10(50), 500) * u.Msun
f_orb_grid = np.logspace(np.log10(4e-5), np.log10(3e-1), 400) * u.Hz

# turn the two lists into grids
MC, FORB = np.meshgrid(m_c_grid, f_orb_grid)

# flatten grids
m_c, f_orb = MC.flatten(), FORB.flatten()

# convert chirp mass to individual masses for source class
q = 1.0
m_1 = m_c / q**(3/5) * (1 + q)**(1/5)
```

(continues on next page)

(continued from previous page)

```
m_2 = m_1 * q

# use a fixed distance and circular binaries
dist = np.repeat(1, len(m_c)) * u.kpc
ecc = np.zeros(len(m_c))

# create the source class
sources = lw.source.Source(m_1=m_1, m_2=m_2, dist=dist, f_orb=f_orb, ecc=ecc, gw_lum_
↪tol=1e-3)
```

Next, we can use LEGWORK to compute their merger times and SNRs for the contours.

[5]:
```
# calculate merger times and then SNR
sources.get_merger_time()
sources.get_snr(verbose=True)
```

```
Calculating SNR for 200000 sources
        0 sources have already merged
        125169 sources are stationary
                125169 sources are stationary and circular
        74831 sources are evolving
                74831 sources are evolving and circular
```

[5]:
```
array([5.37039831e-04, 5.48303595e-04, 5.59803603e-04, ...,
       1.06420933e+04, 1.07531750e+04, 1.08654183e+04])
```

We flattened the grid to fit into the Source class but now we can reshape the output to match the original grid.

[6]:
```
# reshape the output into grids
t_merge_grid = sources.t_merge.reshape(MC.shape)
snr_grid = sources.snr.reshape(MC.shape)
```

Now we can define a couple of functions for formatting the time, distance and galaxy name contours.

[7]:
```
def fmt_time(x):
    if x == 4:
        return r"$t_{\rm merge} = T_{\rm obs}$"
    elif x >= 1e9:
        return "{0:1.0f} Gyr".format(x / 1e9)
    elif x >= 1e6:
        return "{0:1.0f} Myr".format(x / 1e6)
    elif x >= 1e3:
        return "{0:1.0f} kyr".format(x / 1e3)
    elif x >= 1:
        return "{0:1.0f} yr".format(x)
    elif x >= 1/12:
        return "{0:1.0f} month".format(x * 12)
    else:
        return "{0:1.0f} week".format(x * 52)

def fmt_dist(x):
    if x >= 1e9:
        return "{0:1.0f} Gpc".format(x / 1e9)
    elif x >= 1e6:
```

(continues on next page)

(continued from previous page)

```
        return "{0:1.0f} Mpc".format(x / 1e6)
    elif x >= 1e3:
        return "{0:1.0f} kpc".format(x / 1e3)
    else:
        return "{0:1.0f} pc".format(x)

def fmt_name(x):
    if x == np.log10(8):
        return "MW Centre"
    elif x == np.log10(50):
        return "SMC/LMC"
    elif x == np.log10(800):
        return "Andromeda"
    elif x == np.log10(40000):
        return "GW170817"
```

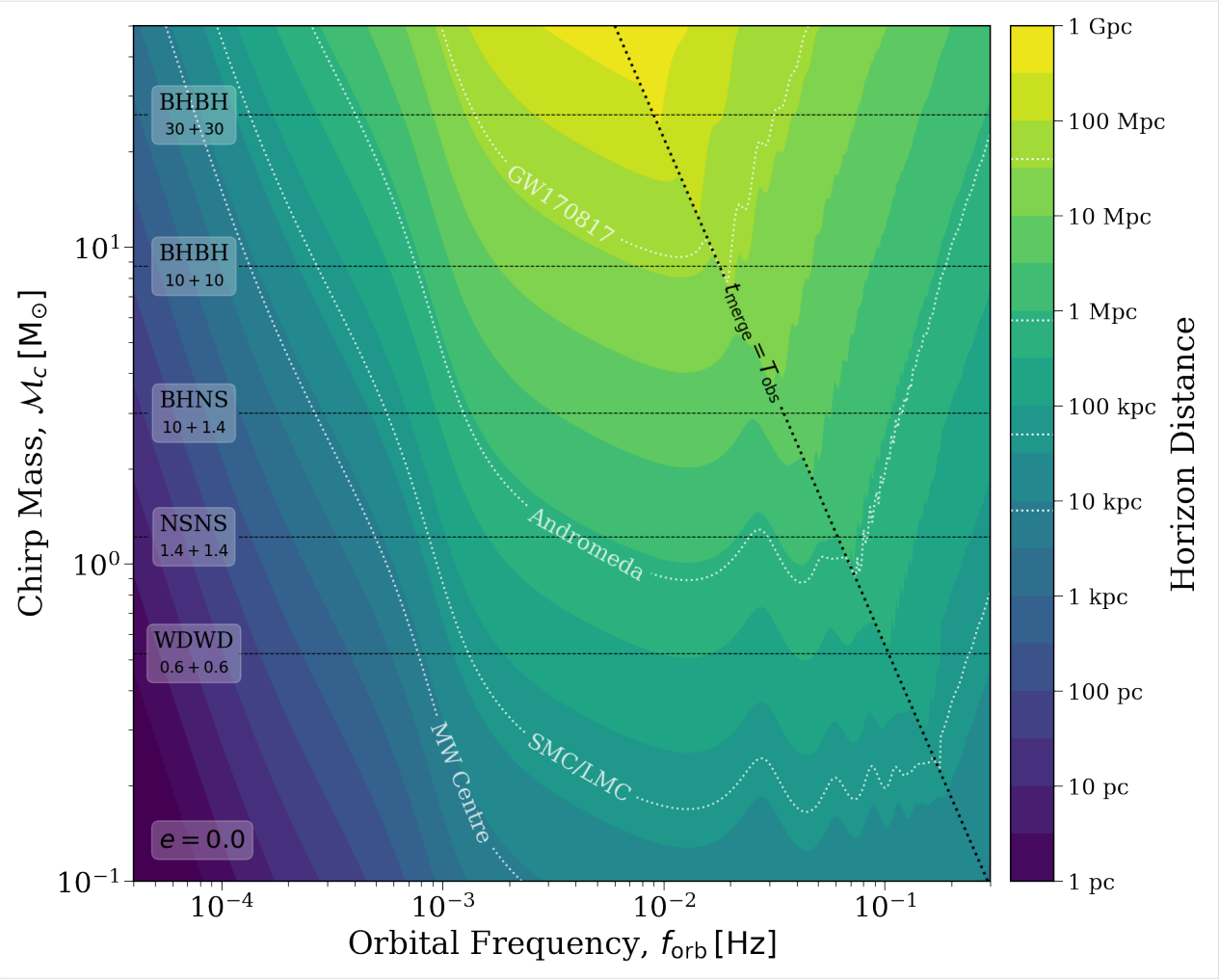
Finally, we put it all together to create a contour plot with all of the information.

[8]:
```
# create a square figure plus some space for a colourbar
size = 12
cbar_space = 2
fig, ax = plt.subplots(figsize=(size + cbar_space, size))

# set up scales early so contour labels show up nicely
ax.set_xscale("log")
ax.set_yscale("log")

# set axes labels and lims
ax.set_xlabel(r"Orbital Frequency, $f_{\rm orb} \, [\rm Hz]$")
ax.set_ylabel(r"Chirp Mass, $\mathcal{M}_c \, [\rm M_{\odot}]$")
ax.set_xlim(4e-5, 3e-1)

# calculate the horizon distance
snr_threshold = 7
horizon_distance = (snr_grid / snr_threshold * 1 * u.kpc).to(u.kpc)

# set up the contour levels
distance_levels = np.arange(-3, 6 + 0.5, 0.5)
distance_tick_levels = distance_levels[::2]

# plot the contours for horizon distance
distance_cont = ax.contourf(FORB, MC, np.log10(horizon_distance.value), levels=distance_
↪levels)

# hide edges that show up in rendered PDFs
for c in distance_cont.collections:
    c.set_edgecolor("face")

# create a colour with custom formatted labels
cbar = fig.colorbar(distance_cont, ax=ax, pad=0.02, ticks=distance_tick_levels,␣
↪fraction=cbar_space / (size + cbar_space))
cbar.ax.set_yticklabels([fmt_dist(np.power(10, distance_tick_levels + 3)[i]) for i in␣
```

(continues on next page)

(continued from previous page)

```
↪range(len(distance_tick_levels))])
cbar.set_label(r"Horizon Distance", fontsize=fs)
cbar.ax.tick_params(axis="both", which="major", labelsize=0.7 * fs)

# annotate the colourbar with some named distances
named_distances = np.log10([8, 50, 800, 40000])
for name in named_distances:
    cbar.ax.axhline(name, color="white", linestyle="dotted")

# plot the same names as contours
named_cont = ax.contour(FORB, MC, np.log10(horizon_distance.value), levels=named_
↪distances,
                        colors="white", alpha=0.8, linestyles="dotted")
ax.clabel(named_cont, named_cont.levels, fmt=fmt_name, use_clabeltext=True, fontsize=0.
↪7*fs,
          manual=[(1.1e-3, 2e-1), (4e-3, 2.2e-1), (4e-3,1e0), (3e-3, 1.2e1)])

# add a line for when the merger time becomes less than the inspiral time
time_cont = ax.contour(FORB, MC, t_merge_grid.to(u.yr).value, levels=[4],
                       colors="black", linewidths=2, linestyles="dotted") #[1/52, 1/12,␣
↪4, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10]
ax.clabel(time_cont, time_cont.levels, fmt=fmt_time, fontsize=0.7*fs, use_
↪clabeltext=True, manual=[(2.5e-2, 5e0)])

# plot a series of lines and annotations for average DCO masses
for m_1, m_2, dco in [(0.6, 0.6, "WDWD"), (1.4, 1.4, "NSNS"),
                      (10, 1.4, "BHNS"), (10, 10, "BHBH"), (30, 30, "BHBH")]:
    # find chirp mass
    m_c_val = lw.utils.chirp_mass(m_1, m_2)

    # plot lines before and after bbox
    ax.plot([4e-5, 4.7e-5], [m_c_val, m_c_val],
            color="black", lw=0.75, zorder=1, linestyle="--")
    ax.plot([1.2e-4, 1e0], [m_c_val, m_c_val],
            color="black", lw=0.75, zorder=1, linestyle="--")

    # plot name and bbox, then masses below in smaller font
    ax.annotate(dco + "\n", xy=(7.5e-5, m_c_val), ha="center", va="center", fontsize=0.
↪7*fs,
                bbox=dict(boxstyle="round", fc="white", ec="white", alpha=0.25))
    ax.annotate(r"${{{}}} + {{{}}}$".format(m_1, m_2),
                xy=(7.5e-5, m_c_val * 0.95), ha="center", va="top", fontsize=0.5*fs)

# ensure that everyone knows this only applies for circular sources
ax.annotate(r"$e = 0.0$", xy=(0.03, 0.04), xycoords="axes fraction", fontsize=0.8*fs,
            bbox=dict(boxstyle="round", fc="white", ec="white", alpha=0.25))

ax.set_facecolor(plt.get_cmap("viridis")(0.0))

plt.show()
```



Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.5 Demo - SNR evolution over time

This demo uses LEGWORK to show how we could track the SNR of a source if we observed it at different times

[2]:
```
import legwork as lw
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rc('font', family='serif')
plt.rcParams['text.usetex'] = False
```

(continues on next page)

(continued from previous page)

```
fs = 24

# update various fontsizes to match
params = {'figure.figsize': (12, 8),
          'legend.fontsize': fs,
          'axes.labelsize': 0.7 * fs,
          'xtick.labelsize': 0.6 * fs,
          'ytick.labelsize': 0.6 * fs,
          'axes.linewidth': 1.1,
          'xtick.major.size': 7,
          'xtick.minor.size': 4,
          'ytick.major.size': 7,
          'ytick.minor.size': 4}
plt.rcParams.update(params)
```

To start we can define the initial parameters of the source and work out how long it will take until it will merge.

Feel free to change these values to see how the plot changes!!

[4]:
```
m_1 = 15 * u.Msun
m_2 = 15 * u.Msun
dist = 20 * u.kpc
ecc = 0.5
f_orb = 3e-5 * u.Hz
```

[5]:
```
t_merge = lw.evol.get_t_merge_ecc(m_1=m_1, m_2=m_2, ecc_i=ecc, f_orb_i=f_orb)
```

Next we can evolve the system up until a year before it merges with 1000 timesteps and record the eccentricity and frequency evolution as well as the timesteps.

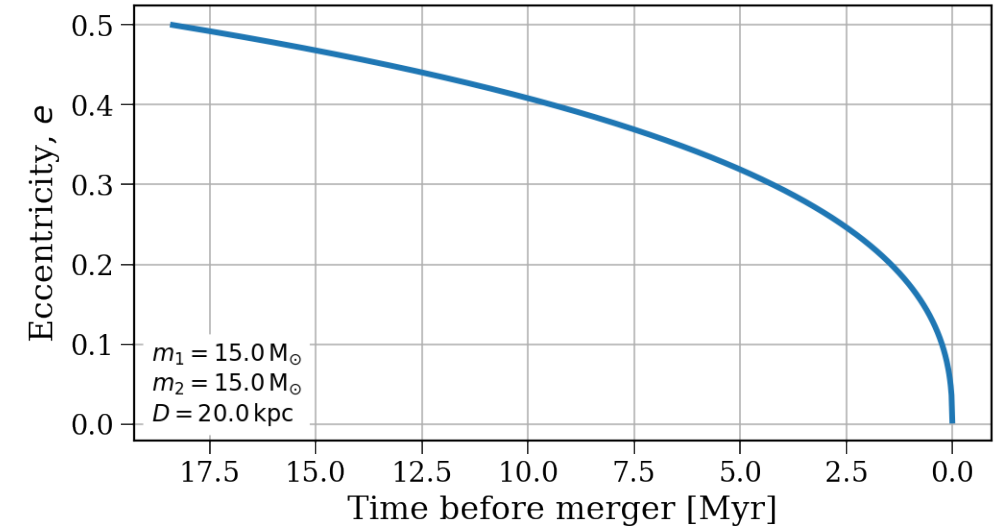
[6]:
```
ecc_evol, f_orb_evol, timesteps = lw.evol.evol_ecc(m_1=m_1,
                                                   m_2=m_2,
                                                   ecc_i=ecc,
                                                   f_orb_i=f_orb,
                                                   t_evol=t_merge - 100 * u.yr,
                                                   avoid_merger=False,
                                                   output_vars=["ecc", "f_orb",
↪"timesteps"],
                                                   n_step=1000)
```

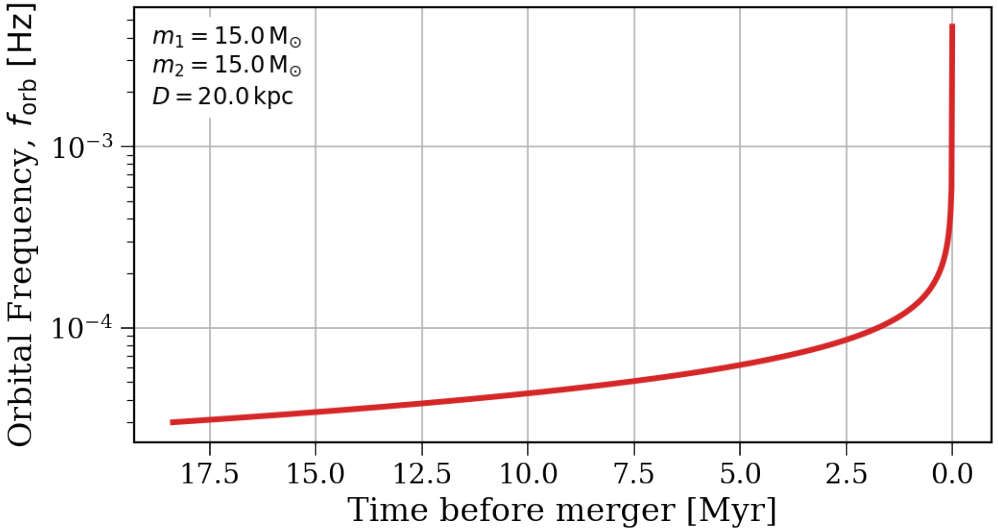
We then take these evolved values and treat them as different sources that occur at different times and calculate the SNR for each one.

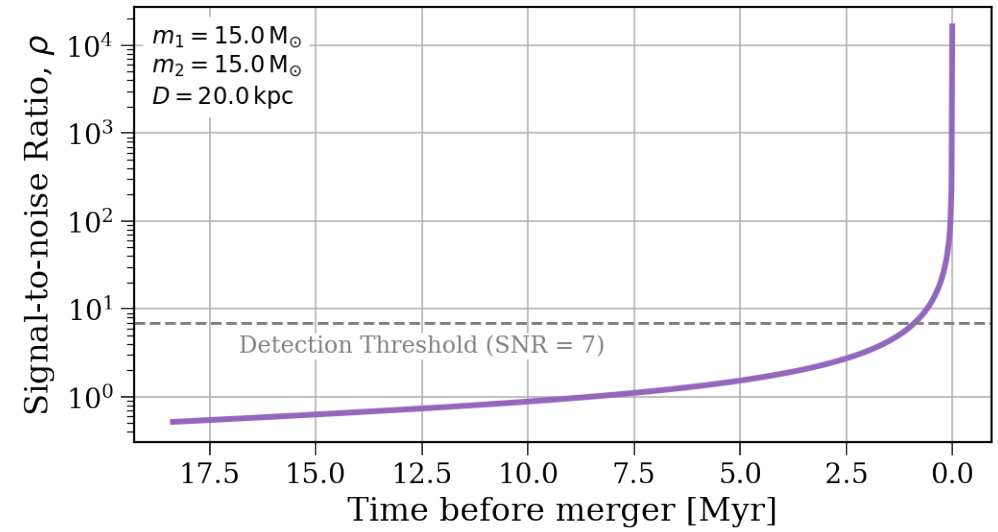
[11]:
```
source = lw.source.Source(m_1=np.repeat(m_1, len(ecc_evol)),
                          m_2=np.repeat(m_2, len(ecc_evol)),
                          dist=np.repeat(dist, len(ecc_evol)),
                          ecc=ecc_evol,
                          f_orb=f_orb_evol, interpolate_g=False)

snr = source.get_snr()
```

Now we can plot the results! Let's compare how the eccentricity, frequency and SNR change as we approach the merger.

[12]:
```
fig, axes = plt.subplots(3, 1, figsize=(8, 15))
fig.subplots_adjust(hspace=0.3)

# calculate the time before the merger rather than timesteps
t_before_merger = (t_merge - timesteps).to(u.Myr)

# plot the eccentricity
axes[0].plot(t_before_merger, source.ecc, lw=3)
axes[0].set_ylabel(r"Eccentricity, $e$")

# plot the frequency on a log scale
axes[1].plot(t_before_merger, source.f_orb, lw=3, color="tab:red")
axes[1].set_yscale("log")
axes[1].set_ylabel(r"Orbital Frequency, $f_{\rm orb}\ [\rm Hz]$")

# plot the SNR on a log scale
axes[2].plot(t_before_merger, source.snr, lw=3, color="tab:purple")
axes[2].set_yscale("log")
axes[2].set_ylabel(r"Signal-to-noise Ratio, $\rho$")

# add a line to indicate the detection threshold
axes[2].axhline(7, color="grey", linestyle="--")
axes[2].annotate("Detection Threshold (SNR = 7)", xy=(12.5, 5), ha="center", va="top",␣
↪fontsize=0.5*fs, color="grey",
                 bbox=dict(boxstyle="round", fc="white", ec="white", pad=0.05))


params_string = r"$m_1 = {{{}}}\,${:latex}".format(m_1.value, m_1.unit)
params_string += "\n" + r"$m_2 = {{{}}}\,${:latex}".format(m_2.value, m_2.unit)
params_string += "\n" + r"$D = {{{}}}\,${:latex}".format(dist.value, dist.unit)

for ax in axes:
    ax.set_xlim(reversed(ax.get_xlim()))
    ax.grid()
    ax.set_xlabel("Time before merger [Myr]")
    ax.annotate(params_string, xy=(0.02, 0.96 if ax != axes[0] else 0.03),
                xycoords="axes fraction", va="top" if ax != axes[0] else "bottom",
                fontsize=0.5*fs, bbox=dict(boxstyle="round", fc="white", ec="white"))

plt.show()
```


Eccentricity, e
0.5
0.4
0.3
0.2
0.1
0.0
$m_1 = 15.0\,M_\odot$
$m_2 = 15.0\,M_\odot$
$D = 20.0\,kpc$
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Time before merger [Myr]


Orbital Frequency, f_{orb} [Hz]
10^{-3}
10^{-4}
$m_1 = 15.0\,M_\odot$
$m_2 = 15.0\,M_\odot$
$D = 20.0\,kpc$
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Time before merger [Myr]


Signal-to-noise Ratio, ρ
10^4
10^3
10^2
10^1
10^0
$m_1 = 15.0\,M_\odot$
$m_2 = 15.0\,M_\odot$
$D = 20.0\,kpc$
Detection Threshold (SNR = 7)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Time before merger [Myr]

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

3.6 Demo - Verification binaries

This demo uses LEGWORK to show how you can interact with the LISA verification binaries

[2]:
```
import legwork as lw
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rc('font', family='serif')
plt.rcParams['text.usetex'] = False
fs = 24

# update various fontsizes to match
params = {'figure.figsize': (12, 8),
          'legend.fontsize': fs,
          'axes.labelsize': fs,
          'xtick.labelsize': 0.6 * fs,
          'ytick.labelsize': 0.6 * fs,
          'axes.linewidth': 1.1,
          'xtick.major.size': 7,
          'xtick.minor.size': 4,
          'ytick.major.size': 7,
          'ytick.minor.size': 4}
plt.rcParams.update(params)
```

In order to gain access to all of the verification binary data from Kupfer+18, you can simply run the following.

[4]:
```
vbs = lw.source.VerificationBinaries()
```
```
Generating random values for source polarisations
```

This returns a Source class that contains all of the verification binary data in the usual variables, but with two extra variables: - labels: contains the designation of each binary - true_snr: the SNR calculated by Kupfer+18

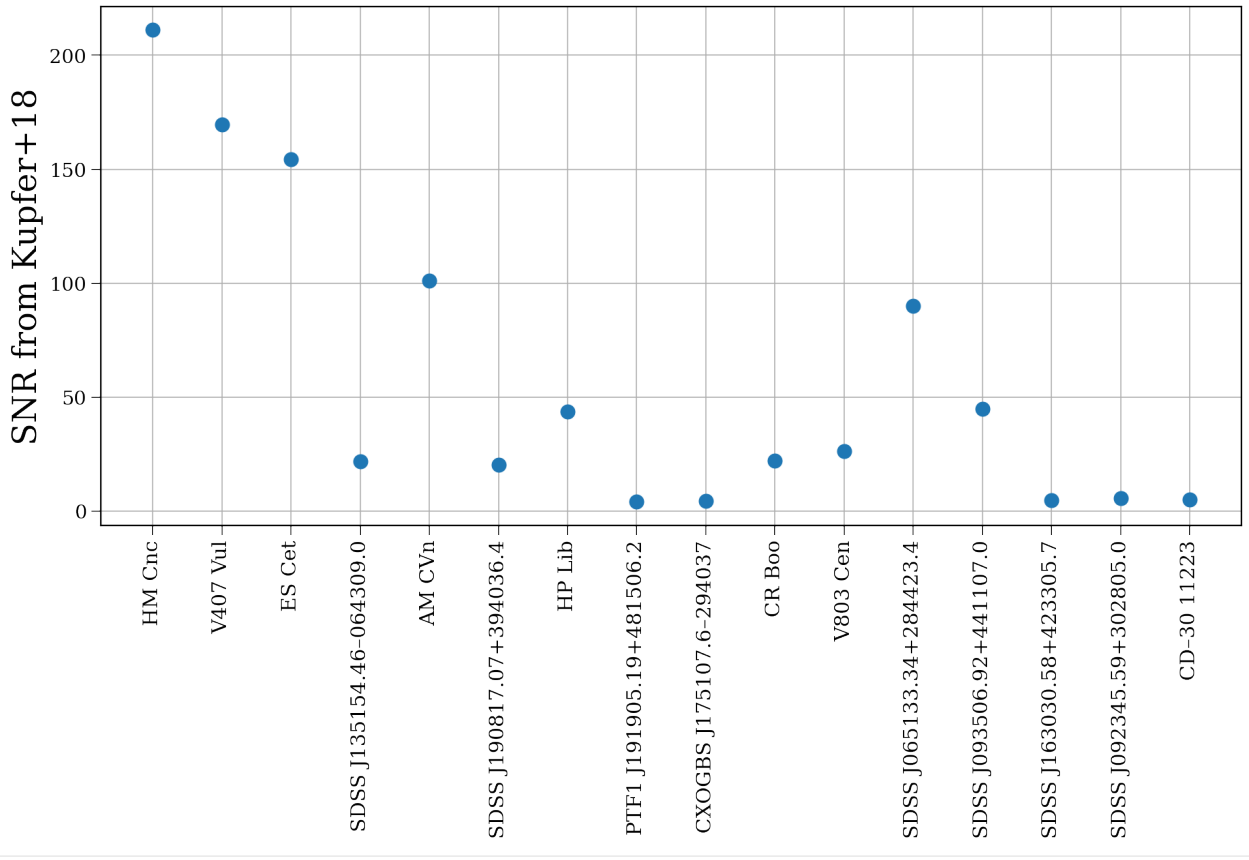
For instance, we can plot the SNR each source and label them:

[5]:
```
fig, ax = plt.subplots(figsize=(15, 7))

ax.scatter(range(vbs.n_sources), vbs.true_snr, zorder=10, s=100)

ax.set_xticks(range(vbs.n_sources))
ax.set_xticklabels(vbs.labels, rotation=90, fontsize=15)

ax.grid()
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("SNR from Kupfer+18")

plt.show()
```



It is important to highlight that SNR returned by LEGWORK will not give the same values as the `true_snr`. This is because Kupfer+18 use a full LISA simulation which would not be feasible for a large number of sources and is beyond the scope of LEGWORK. We instead would use averaged equations as outlined in our derivations and thus find different values for the SNR.

[6]:
```
vbs.get_snr()
print(vbs.snr / vbs.true_snr)
```

```
[0.22679184 0.31142177 0.28776068 0.49147054 0.53278135 0.64859775
 0.35889764 0.4979882  0.4420345  0.22664842 0.23872143 0.17332405
 0.60920814 0.43898847 0.31181575 0.42892542]
```

So for now let's simply set the SNR equal to the true SNR and plot them on the sensitivity curve!

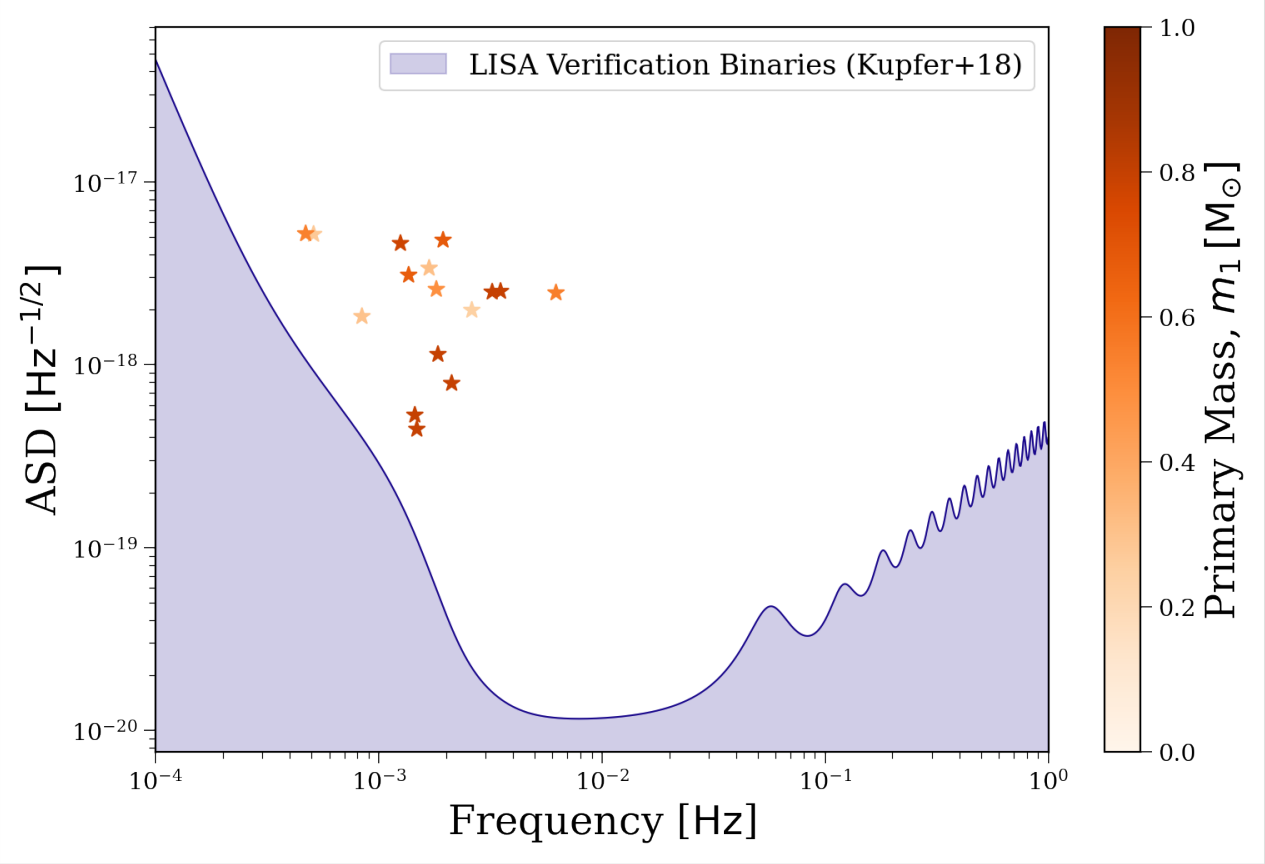
[7]:
```
vbs.snr = np.array(vbs.true_snr)
```

[8]:
```
fig, ax = lw.visualisation.plot_sensitivity_curve(frequency_range=np.logspace(-4, 0,␣
↪1000) * u.Hz,
                                                  show=False)
fig, ax = vbs.plot_sources_on_sc(scatter_s=100, marker="*", c=vbs.m_1.to(u.Msun),
```

(continues on next page)

(continued from previous page)

```
                                    fig=fig, ax=ax, show=False, cmap="Oranges", vmin=0.0,␣
↪vmax=1.0)
cbar = fig.colorbar(ax.get_children()[2])
cbar.set_label(r"Primary Mass, $m_1 \, [{\rm M_{\odot}}]$")

ax.legend(handles=[ax.get_children()[1]], labels=["LISA Verification Binaries (Kupfer+18)
↪"], fontsize=0.7*fs, markerscale=2)

plt.show()
```

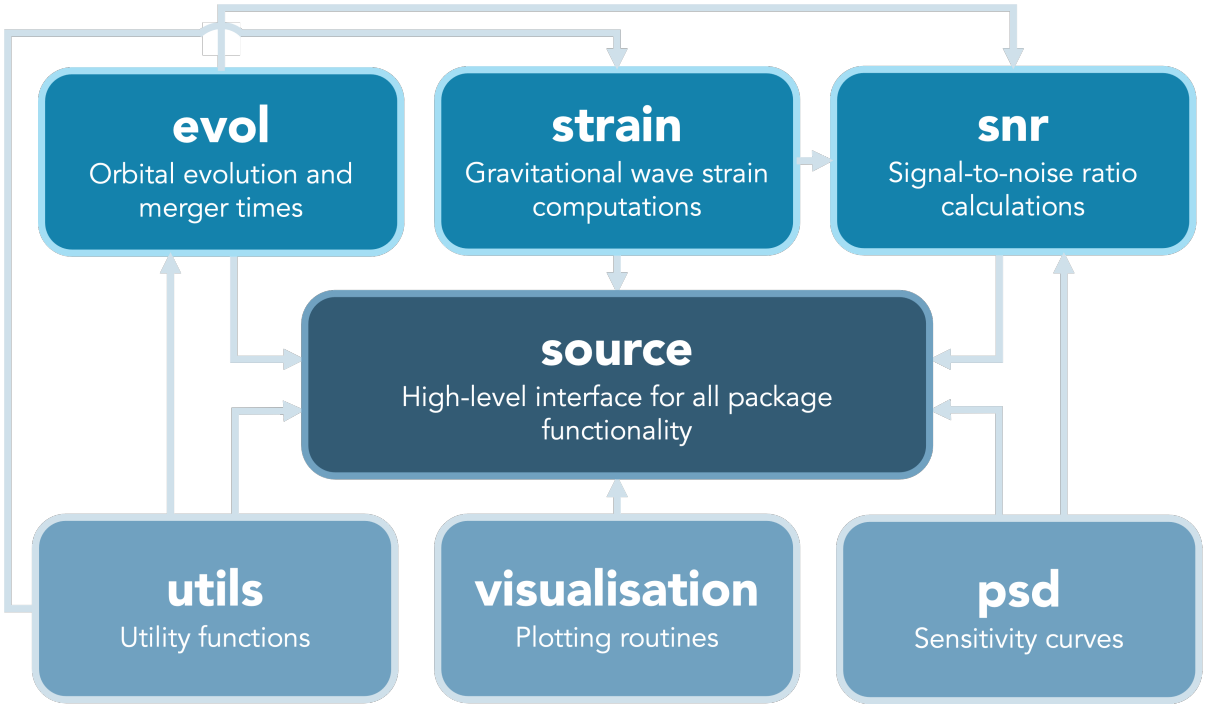


CHAPTER

FOUR

LEGWORK MODULES AND API REFERENCE

`LEGWORK` is composed of 7 different modules, each with a different focus. The diagram below illustrates how each module is connected to the others as well as listing the general purpose of each module. In particular, note that the `source` module provides a simple interface to the functions in all other modules!



The rest of this page contains the API reference for each individual function in every module, feel free to use the table of contents on the left to easily navigate to the function you need.

4.1 legwork.evol Module

Functions using equations from Peters and Mathews (1964) to calculate inspiral times and evolve binary parameters.

4.1.1 Functions

de_dt(e, times, beta, c_0)	Compute eccentricity time derivative
integrate_de_dt(args)	Wrapper that integrates *legwork.evol.de_dt()* with odeint
evol_circ([t_evol, n_step, timesteps, beta, ...])	Evolve an array of circular binaries for `t_evol` time
evol_ecc(ecc_i[, t_evol, n_step, timesteps, ...])	Evolve an array of eccentric binaries for `t_evol` time
get_t_merge_circ([beta, m_1, m_2, a_i, f_orb_i])	Computes the merger time for circular binaries
get_t_merge_ecc(ecc_i[, a_i, f_orb_i, beta, ...])	Computes the merger time for binaries
t_merge_mandel_fit(ecc_i)	A fit to the Peters 1964 merger time equation (5.14) by Ilya Mandel.
evolve_f_orb_circ(f_orb_i, m_c, t_evol[, ...])	Evolve orbital frequency for `t_evol` time.
check_mass_freq_input([beta, m_1, m_2, a_i, ...])	Check that mass and frequency input is valid
create_timesteps_array(a_i, beta[, ecc_i, ...])	Create an array of timesteps
determine_stationarity(f_orb_i, t_evol, ecc_i)	Determine whether a binary is stationary

de_dt

`legwork.evol.`**de_dt**(*e*, *times*, *beta*, *c_0*)

Compute eccentricity time derivative

Computes the evolution of the eccentricity from the emission of gravitational waves following Peters & Mathews (1964) Eq. 5.13

Parameters

e
[*float*] Initial eccentricity

times
[*float/array*] Evolution timestep. Not actually used in function but required for use with scipy's `scipy.integrate.odeint()`

beta
[*float*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()*

c_0
[*float*] Constant defined in Peters and Mathews (1964) Eq. 5.11. See *legwork.utils.c_0()*

Returns

dedt
[*float/array*] Eccentricity time derivative

integrate_de_dt

legwork.evol.**integrate_de_dt**(*args*)

Wrapper that integrates *legwork.evol.de_dt()* with odeint

Parameters

args
[*list*] List of arguments for *legwork.evol.de_dt()* including [e, times, beta, c_0]

Returns

ecc_evol
[*array*] eccentricity evolution

evol_circ

legwork.evol.**evol_circ**(*t_evol=None*, *n_step=100*, *timesteps=None*, *beta=None*, *m_1=None*, *m_2=None*, *a_i=None*, *f_orb_i=None*, *output_vars='f_orb'*)

Evolve an array of circular binaries for `t_evol` time

This function implements Peters & Mathews (1964) Eq. 5.9.

Note that all of {`beta`, `m_1`, `m_2`, `a_i`, `f_orb_i`} must have the same dimensions.

Parameters

t_evol
[*float/array*] Amount of time for which to evolve each binaries. Required if `timesteps` is None. Defaults to merger times.

n_steps
[*int*] Number of timesteps to take between t=0 and t=``t_evol``. Required if `timesteps` is None. Defaults to 100.

timesteps
[*float/array*] Array of exact timesteps to output when evolving each binary. Must be monotonically increasing and start with t=0. Either supply a 1D array to use for every binary or a 2D array that has a different array of timesteps for each binary. `timesteps` is used in place of `t_evol` and `n_steps` and takes precedence over them. Note that these are *not* the timesteps that will be taken whilst the sources are evolved since these are determined adaptively during the integration by scipy's odeint

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()* (if supplied `m_1` and *m_2* are ignored)

m_1
[*float/array*] Primary mass (required if `beta` is None or if `output_vars` contains a frequency)

m_2
[*float/array*] Secondary mass (required if `beta` is None or if `output_vars` contains a frequency)

a_i
[*float/array*] Initial semi-major axis (if supplied `f_orb_i` is ignored)

f_orb_i
[*float/array*] Initial orbital frequency (required if `a_i` is None)

output_vars
[*str/array*] List of **ordered** output vars, or a single var. Choose from any of `timesteps`, `a`, `f_orb` and `f_GW` for which of timesteps, semi-major axis and orbital/GW frequency that you want. Default is `f_orb`.

Returns

evolution
[*array*] Array containing any of semi-major axis, timesteps and frequency evolution. Content determined by `output_vars`.

evol_ecc

`legwork.evol.`**`evol_ecc`**(*ecc_i, t_evol=None, n_step=100, timesteps=None, beta=None, m_1=None, m_2=None, a_i=None, f_orb_i=None, output_vars=['ecc', 'f_orb'], n_proc=1, avoid_merger=True, exact_t_merge=False, t_before=<Quantity 1. Myr>, t_merge=None*)

Evolve an array of eccentric binaries for `t_evol` time

This function use Peters & Mathews (1964) Eq. 5.11 and 5.13.

Note that all of {`beta`, `m_1`, `m_2`, `ecc_i`, `a_i`, `f_orb_i`} must have the same dimensions.

Parameters

ecc_i
[*float/array*] Initial eccentricity

t_evol
[*float/array*] Amount of time for which to evolve each binaries. Required if `timesteps` is None. Defaults to merger times.

n_steps
[*int*] Number of timesteps to take between t=0 and t=``t_evol``. Required if `timesteps` is None. Defaults to 100.

timesteps
[*float/array*] Array of exact timesteps to take when evolving each binary. Must be monotonically increasing and start with t=0. Either supply a 1D array to use for every binary or a 2D array that has a different array of timesteps for each binary. `timesteps` is used in place of `t_evol` and `n_steps` and takes precedence over them.

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()* (if supplied `m_1` and *m_2* are ignored)

m_1
[*float/array*] Primary mass (required if `beta` is None or if `output_vars` contains a frequency)

m_2
[*float/array*] Secondary mass (required if `beta` is None or if `output_vars` contains a frequency)

a_i
[*float/array*] Initial semi-major axis (if supplied `f_orb_i` is ignored)

f_orb_i
[*float/array*] Initial orbital frequency (required if `a_i` is None)

output_vars
[*array*] List of **ordered** output vars, choose from any of `timesteps`, `ecc`, `a`, `f_orb` and `f_GW` for which of timesteps, eccentricity, semi-major axis and orbital/GW frequency that you want. Default is [`ecc`, `f_orb`]

n_proc
[*int*] Number of processors to split eccentricity evolution over, where the default is n_proc=1

avoid_merger
[*boolean*] Whether to avoid integration around the merger of the binary. Warning: setting this to false will result in many LSODA errors to be outputted since the derivatives get so large.

exact_t_merge
[*boolean*] Whether to calculate the merger time exactly or use a fit (only relevant when `avoid_merger` is set to True

t_before
[*float*] How much time before the merger to cutoff the integration (default is 1 Myr - this will prevent all LSODA warnings for e < 0.95, you may need to increase this time if your sample is more eccentric than this)

t_merge
[*float/array*] Merger times for each source to be evolved. Only used when *avoid_merger=True*. If *None* then these will be automatically calculated either approximately or exactly based on the values of *exact_t_merge*.

Returns

evolution
[*array*] Array possibly containing eccentricity, semi-major axis, timesteps and frequency evolution. Content determined by `output_vars`

get_t_merge_circ

`legwork.evol.`**`get_t_merge_circ`**(*beta=None*, *m_1=None*, *m_2=None*, *a_i=None*, *f_orb_i=None*)

Computes the merger time for circular binaries

This function implements Peters & Mathews (1964) Eq. 5.10

Parameters

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()* (if supplied *m_1* and *m_2* are ignored)

m_1
[*float/array*] Primary mass (required if *beta* is None)

m_2
[*float/array*] Secondary mass (required if *beta* is None)

a_i
[*float/array*] Initial semi-major axis (if supplied *f_orb_i* is ignored)

f_orb_i
[*float/array*] Initial orbital frequency (required if *a_i* is None)

Returns

t_merge
[*float/array*] Merger time

get_t_merge_ecc

`legwork.evol.get_t_merge_ecc(ecc_i, a_i=None, f_orb_i=None, beta=None, m_1=None, m_2=None, small_e_tol=0.15, large_e_tol=0.9999, exact=True)`

Computes the merger time for binaries

This function implements Peters (1964) Eq. 5.10, 5.14 and the two unlabelled equations after 5.14 (using a different one depending on the eccentricity of each binary)

Parameters

ecc_i
[*float/array*] Initial eccentricity (if *ecc_i* is known to be 0.0 then use *get_t_merge_circ* instead)

a_i
[*float/array*] Initial semi-major axis (if supplied *f_orb_i* is ignored)

f_orb_i
[*float/array*] Initial orbital frequency (required if *a_i* is None)

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()* (if supplied *m_1* and *m_2* are ignored)

m_1
[*float/array*] Primary mass (required if *beta* is None)

m_2
[*float/array*] Secondary mass (required if *beta* is None)

small_e_tol
[*float*] Eccentricity below which to apply the small e approximation (see first unlabelled equation following Eq. 5.14 of Peters 1964), defaults to 0.15 to keep relative error below approximately 2%

large_e_tol
[*float*] Eccentricity above which to apply the large e approximation (see second unlabelled equation following Eq. 5.14 of Peters 1964), defaults to 0.9999 to keep relative error below approximately 2%

exact
[*boolean*] Whether to calculate the merger time exactly with numerical integration or to instead use the fit from Mandel 2021 (see *legwork.evol.t_merge_mandel_fit()*)

Returns

t_merge
[*float/array*] Merger time

t_merge_mandel_fit

`legwork.evol.t_merge_mandel_fit(ecc_i)`

A fit to the Peters 1964 merger time equation (5.14) by Ilya Mandel. This function gives a factor which, when multiplied by the circular merger time, gives the eccentric merger time with 3% errors. We add a rudimentary polynomial fit to further reduce these errors to within 0.5%. ADS Link: https://ui.adsabs.harvard.edu/abs/2021RNAAS...5..223M/abstract

Parameters

ecc_i
[*float/array*] Initial eccentricity

Returns

factor
[*float/array*] Factor by which to multiply the circular merger timescale by to get the overall merger time.

evolve_f_orb_circ

`legwork.evol.evolve_f_orb_circ(f_orb_i, m_c, t_evol, ecc_i=0.0, merge_f=<Quantity 1.e+09 Hz>)`

Evolve orbital frequency for `t_evol` time.

This gives the exact final frequency for circular binaries. However, it will overestimate the final frequency for an eccentric binary and if an exact value is required then *`legwork.evol.evol_ecc()`* should be used instead.

Parameters

f_orb_i
[*float/array*] Initial orbital frequency

m_c
[*float/array*] Chirp mass

t_evol
[*float*] Time over which the frequency evolves

ecc_i
[*float/array*] Initial eccentricity

merge_f
[*float*] Frequency to assign if the binary has already merged after `t_evol`

Returns

f_orb_f
[*bool/array*] Final orbital frequency

check_mass_freq_input

`legwork.evol.check_mass_freq_input(beta=None, m_1=None, m_2=None, a_i=None, f_orb_i=None)`

Check that mass and frequency input is valid

Helper function to check that either `beta` or (`m_1` and `m_2`) is provided and that `a_i` or `f_orb_i` is provided as well as calculate quantities that are not passed as arguments.

Parameters

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()* (if supplied `m_1` and *m_2* are ignored)

m_1
[*float/array*] Primary mass (required if `beta` is None)

m_2
[*float/array*] Secondary mass (required if `beta` is None)

a_i
[*float/array*] Initial semi-major axis (if supplied `f_orb_i` is ignored)

f_orb_i
[*float/array*] Initial orbital frequency (required if `a_i` is None)

Returns

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()*

a_i
[*float/array*] Initial semi-major axis

create_timesteps_array

`legwork.evol.create_timesteps_array(a_i, beta, ecc_i=None, t_evol=None, n_step=100, timesteps=None)`

Create an array of timesteps

Parameters

a_i
[*float/array*] Initial semi-major axis

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq. 5.9. See *legwork.utils.beta()*

ecc_i
[*float/array*] Initial eccentricity

t_evol
[*float/array*] Amount of time for which to evolve each binaries. Required if `timesteps` is None. If None, then defaults to merger times.

n_steps
[*int*] Number of timesteps to take between t=0 and t=``t_evol``. Required if `timesteps` is None. Defaults to 100.

timesteps
[*float/array*] Array of exact timesteps to output when evolving each binary. Must be monotonically increasing and start with t=0. Either supply a 1D array to use for every binary or a 2D array that has a different array of timesteps for each binary. `timesteps` is used in place of `t_evol` and `n_steps` and takes precedence over them. Note that these are *not* the timesteps that will be taken whilst the sources are evolved since these are determined adaptively during the integration by scipy's odeint

Returns

timesteps
[*float/array*] Array of timesteps for each binary

determine_stationarity

`legwork.evol.`**`determine_stationarity`**(*f_orb_i*, *t_evol*, *ecc_i*, *m_1=None*, *m_2=None*, *m_c=None*, *stat_tol=0.01*)

Determine whether a binary is stationary

Check how much a binary's orbital frequency changes over `t_evol` time. This function provides a conservative estimate in that some binaries that are stationary may be marked as evolving. This is because the eccentricity also evolves but only use the initial value. Solving this in full would require the same amount of time as assuming the binary is evolving.

Parameters

forb_i
[*float/array*] Initial orbital frequency

t_evol
[*float*] Time over which the frequency evolves

ecc
[*float/array*] Initial eccentricity

m_1
[*float/array*] Primary mass (required if `m_c` is None)

m_2
[*float/array*] Secondary mass (required if `m_c` is None)

m_c
[*float/array*] Chirp mass (overrides *m_1* and *m_2*)

stat_tol
[*float*] Fractional change in frequency above which we do not consider a binary to be stationary

Returns

stationary
[*bool/array*] Mask of whether each binary is stationary

Tip: Feeling a bit spun around by all this binary evolution? Check out our tutorial on using functions in the `evol` module *here!*

4.2 legwork.psd Module

Functions to compute various power spectral densities for sensitivity curves

4.2.1 Functions

load_response_function(f[, fstar])	Load in LISA response function from file
approximate_response_function(f, fstar)	Approximate LISA response function
power_spectral_density(f[, instrument, ...])	Calculates the effective power spectral density for all instruments.
lisa_psd(f[, t_obs, L, approximate_R, ...])	Calculates the effective LISA power spectral density sensitivity curve
tianqin_psd(f[, L, t_obs, approximate_R, ...])	Calculates the effective TianQin power spectral density sensitivity curve
get_confusion_noise(f, model[, t_obs])	Calculate the confusion noise for a particular model
get_confusion_noise_robson19(f[, t_obs])	Calculate the confusion noise using the model from Robson+19 Eq.
get_confusion_noise_huang20(f[, t_obs])	Calculate the confusion noise using the model from Huang+20 Table II.
get_confusion_noise_thiele21(f)	Calculate the confusion noise using the model from Thiele+20 Eq.

load_response_function

legwork.psd.**load_response_function**(*f*, *fstar=0.01909*)

Load in LISA response function from file

Load response function and interpolate values for a range of frequencies. Adapted from https://github.com/eXtremeGravityInstitute/LISA_Sensitivity to use binary files instead of text. See Robson+19 for more details.

Parameters

f
[*float/array*] Frequencies at which to evaluate the sensitivity curve

fstar
[*float*] f* from Robson+19 (default = 19.09 mHz)

Returns

R
[*float/array*] LISA response function at each frequency

approximate_response_function

legwork.psd.**approximate_response_function**(*f*, *fstar*)

Approximate LISA response function

Use Eq.9 of Robson+19 to approximate the LISA response function.

Parameters

f
[*float/array*] Frequencies at which to evaluate the sensitivity curve

fstar
[*float*] f* from Robson+19 (default = 19.09 mHz)

Returns

R
[*float/array*] response function at each frequency

power_spectral_density

legwork.psd.**power_spectral_density**(*f*, *instrument='LISA'*, *custom_psd=None*, *t_obs='auto'*, *L='auto'*, *approximate_R=False*, *confusion_noise='auto'*)

Calculates the effective power spectral density for all instruments.

Parameters

f
[*float/array*] Frequencies at which to evaluate the sensitivity curve

instrument: {{ `LISA`, `TianQin`, `custom` }}
Instrument to use. LISA is used by default. Choosing *custom* uses `custom_psd` to compute PSD.

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some.

t_obs
[*float*] Observation time (default 4 years for LISA and 5 years for TianQin)

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

Returns

psd
[*float/array*] Effective power strain spectral density

lisa_psd

legwork.psd.**lisa_psd**(*f*, *t_obs=<Quantity 4. yr>*, *L=<Quantity 2.5e+09 m>*, *approximate_R=False*, *confusion_noise='robson19'*)

Calculates the effective LISA power spectral density sensitivity curve

Using equations from Robson+19, calculate the effective LISA power spectral density for the sensitivity curve

Parameters

f
[*float/array*] Frequencies at which to evaluate the sensitivity curve

t_obs
[*float*] Observation time (default 4 years)

L
[*float*] LISA arm length in metres (default = 2.5Gm)

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are one of the values listed in *legwork.psd.get_confusion_noise()* or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

Returns

Sn
[*float/array*] Effective power strain spectral density

tianqin_psd

legwork.psd.**tianqin_psd**(*f*, *L=<Quantity 173205.08075689 km>*, *t_obs=<Quantity 5. yr>*, *approximate_R=None*, *confusion_noise='huang20'*)

Calculates the effective TianQin power spectral density sensitivity curve

Using Eq. 13 from Huang+20, calculate the effective TianQin PSD for the sensitivity curve

Note that this function includes an extra factor of 10/3 compared Eq. 13 in Huang+20, since Huang+20 absorbs the factor into the waveform but we instead follow the same convention as Robson+19 for consistency and include it in this 'effective' PSD function instead.

Parameters

f
[*float/array*] Frequencies at which to evaluate the sensitivity curve

L
[*float*] Arm length

t_obs
[*float*] Observation time (default 5 years)

approximate_R
[*boolean*] Ignored for this function

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are one of the values listed in `legwork.psd.get_confusion_noise()` or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

Returns

psd
[*float/array*] Effective power strain spectral density

get_confusion_noise

`legwork.psd.get_confusion_noise(f, model, t_obs='auto')`

Calculate the confusion noise for a particular model

Parameters

f
[*float/array*] Frequencies at which to calculate the confusion noise, must have units of frequency

model
[str, optional] Which model to use for the confusion noise. Must be one of 'robson19', 'huang20', 'thiele21' or None.

t_obs
[*float*, optional] Mission length. Default is 4 years for robson19 and thiele21 and 5 years for huang20.

Returns

confusion_noise
[*float/array*] The confusion noise at each frequency.

Raises

ValueError
When a model other than those defined above is used.

get_confusion_noise_robson19

`legwork.psd.get_confusion_noise_robson19(f, t_obs=<Quantity 4. yr>)`

Calculate the confusion noise using the model from Robson+19 Eq. 14 and Table 1

Also note that this fit is designed based on LISA sensitivity and so it is likely not sensible to apply it to TianQin or other missions.

Parameters

f
[*float/array*] Frequencies at which to calculate the confusion noise, must have units of frequency

t_obs
[*float*, optional] Mission length, parameters are defined for 0.5, 1, 2 and 4 years, the closest mission length to the one inputted will be used. By default 4 years.

Returns

confusion_noise
[*float/array*] The confusion noise at each frequency

get_confusion_noise_huang20

`legwork.psd.get_confusion_noise_huang20(f, t_obs=<Quantity 5. yr>)`

Calculate the confusion noise using the model from Huang+20 Table II. Note that we set the confusion noise to be exactly 0 outside of the range [1e-4, 1] Hz as the fits are not designed to be used outside of this range.

Also note that this fit is designed based on TianQin sensitivity and so it is likely not sensible to apply it to LISA or other missions.

Parameters

f
[*float/array*] Frequencies at which to calculate the confusion noise, must have units of frequency

t_obs
[*float*, optional] Mission length, parameters are defined for 0.5, 1, 2, 4 and 5 years, the closest mission length to the one inputted will be used. By default 5 years.

Returns

confusion_noise
[*float/array*] The confusion noise at each frequency

get_confusion_noise_thiele21

`legwork.psd.get_confusion_noise_thiele21(f)`

Calculate the confusion noise using the model from Thiele+20 Eq. 16 and Table 1. This fit uses a metallicity-dependent binary fraction.

Note: This fit only applies to LISA and only when the mission length is 4 years.

Parameters

f
[*float/array*] Frequencies at which to calculate the confusion noise, must have units of frequency

Returns

confusion_noise
[*float/array*] The confusion noise at each frequency

4.3 legwork.snr Module

Functions to calculate signal-to-noise ratio in four different cases

4.3.1 Functions

snr_circ_stationary(m_c, f_orb, dist, t_obs)	Computes SNR for circular and stationary sources
snr_ecc_stationary(m_c, f_orb, ecc, dist, ...)	Computes SNR for eccentric and stationary sources
snr_circ_evolving(m_1, m_2, f_orb_i, dist, ...)	Computes SNR for circular and stationary sources
snr_ecc_evolving(m_1, m_2, f_orb_i, dist, ...)	Computes SNR for eccentric and evolving sources.

snr_circ_stationary

legwork.snr.**snr_circ_stationary**(*m_c*, *f_orb*, *dist*, *t_obs*, *position=None*, *polarisation=None*, *inclination=None*, *interpolated_g=None*, *interpolated_sc=None*, ***kwargs*)

Computes SNR for circular and stationary sources

Parameters

m_c
[*float/array*] Chirp mass

f_orb
[*float/array*] Orbital frequency

dist
[*float/array*] Distance to the source

t_obs
[*float*] Total duration of the observation

position
[*SkyCoord/array*, optional] Sky position of source. Must be specified using Astropy's `astropy.coordinates.SkyCoord` class.

polarisation
[*float/array*, optional] GW polarisation angle of the source. Must have astropy angular units.

inclination
[*float/array*, optional] Inclination of the source. Must have astropy angular units.

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

interpolated_sc
[*function*] A function returned by `scipy.interpolate.interp1d` that computes the LISA sensitivity curve. Default is None and uses exact values. Note: take care to ensure that your interpolated function has the same LISA observation time as `t_obs` and uses the same instrument.

****kwargs**
[*various*] Keyword args are passed to *legwork.psd.power_spectral_density()*, see those docs for details on possible arguments.

Returns

snr
[*float/array*] SNR for each binary

snr_ecc_stationary

legwork.snr.**snr_ecc_stationary**(*m_c*, *f_orb*, *ecc*, *dist*, *t_obs*, *harmonics_required*, *interpolated_g=None*, *interpolated_sc=None*, *ret_max_snr_harmonic=False*, *ret_snr2_by_harmonic=False*, ***kwargs*)

Computes SNR for eccentric and stationary sources

Parameters

m_c
[*float/array*] Chirp mass

f_orb
[*float/array*] Orbital frequency

ecc
[*float/array*] Eccentricity

dist
[*float/array*] Distance to the source

t_obs
[*float*] Total duration of the observation

harmonics_required
[*integer*] Maximum integer harmonic to compute

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

interpolated_sc
[*function*] A function returned by `scipy.interpolate.interp1d` that computes the LISA sensitivity curve. Default is None and uses exact values. Note: take care to ensure that your interpolated function has the same LISA observation time as `t_obs` and uses the same instrument.

ret_max_snr_harmonic
[*boolean*] Whether to return (in addition to the snr), the harmonic with the maximum SNR

ret_snr2_by_harmonic
[*boolean*] Whether to return the SNR^2 in each individual harmonic rather than the total. The total can be retrieving by summing and then taking the square root.

****kwargs**
[*various*] Keyword args are passed to *`legwork.psd.power_spectral_density()`*, see those docs for details on possible arguments.

Returns

snr
[*float/array*] SNR for each binary

max_snr_harmonic
[*int/array*] harmonic with maximum SNR for each binary (only returned if `ret_max_snr_harmonic=True`)

snr_circ_evolving

legwork.snr.**snr_circ_evolving**(*m_1*, *m_2*, *f_orb_i*, *dist*, *t_obs*, *n_step*, *t_merge=None*, *interpolated_g=None*, *interpolated_sc=None*, ***kwargs*)

Computes SNR for circular and stationary sources

Parameters

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

f_orb_i
[*float/array*] Initial orbital frequency

dist
[*float/array*] Distance to the source

t_obs
[*float*] Total duration of the observation

n_step
[*int*] Number of time steps during observation duration

t_merge
[*float/array*] Time until merger

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

interpolated_sc
[*function*] A function returned by `scipy.interpolate.interp1d` that computes the LISA sensitivity curve. Default is None and uses exact values. Note: take care to ensure that your interpolated function has the same LISA observation time as `t_obs` and uses the same instrument.

****kwargs**
[*various*] Keyword args are passed to *`legwork.psd.power_spectral_density()`*, see those docs for details on possible arguments.

Returns

sn
[*float/array*] SNR for each binary

snr_ecc_evolving

legwork.snr.**snr_ecc_evolving**(*m_1*, *m_2*, *f_orb_i*, *dist*, *ecc*, *harmonics_required*, *t_obs*, *n_step*, *t_merge=None*, *interpolated_g=None*, *interpolated_sc=None*, *n_proc=1*, *ret_max_snr_harmonic=False*, *ret_snr2_by_harmonic=False*, ***kwargs*)

Computes SNR for eccentric and evolving sources.

Note that this function will not work for exactly circular (ecc = 0.0) binaries.

Parameters

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

f_orb_i
[*float/array*] Initial orbital frequency

dist
[*float/array*] Distance to the source

ecc
[*float/array*] Eccentricity

harmonics_required
[*int*] Maximum integer harmonic to compute

t_obs
[*float*] Total duration of the observation

n_step
[*int*] Number of time steps during observation duration

t_merge
[*float/array*] Time until merger

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

interpolated_sc
[*function*] A function returned by `scipy.interpolate.interp1d` that computes the LISA sensitivity curve. Default is None and uses exact values. Note: take care to ensure that your interpolated function has the same LISA observation time as `t_obs` and uses the same instrument.

n_proc
[*int*] Number of processors to split eccentricity evolution over, where the default is n_proc=1

ret_max_snr_harmonic
[*boolean*] Whether to return (in addition to the snr), the harmonic with the maximum SNR

ret_snr2_by_harmonic
[*boolean*] Whether to return the SNR^2 in each individual harmonic rather than the total. The total can be retrieving by summing and then taking the square root.

****kwargs**
[*various*] Keyword args are passed to *legwork.psd.power_spectral_density()*, see those docs for details on possible arguments.

Returns

snr
[*float/array*] SNR for each binary

max_snr_harmonic
[*int/array*] harmonic with maximum SNR for each binary (only returned if `ret_max_snr_harmonic=True`)

4.4 legwork.source Module

A collection of classes for analysing gravitational wave sources

4.4.1 Classes

Source(m_1, m_2, ecc, dist[, n_proc, f_orb, ...])	Class for generic GW sources
Stationary(m_1, m_2, ecc, dist[, n_proc, ...])	Subclass for sources that are stationary
Evolving(m_1, m_2, ecc, dist[, n_proc, ...])	Subclass for sources that are evolving
VerificationBinaries()	Generate a Source class with the LISA verification binaries preloaded.

Source

class legwork.source.**Source**(*m_1*, *m_2*, *ecc*, *dist*, *n_proc=1*, *f_orb=None*, *a=None*, *position=None*, *polarisation=None*, *inclination=None*, *weights=None*, *gw_lum_tol=0.05*, *stat_tol=0.01*, *interpolate_g=True*, *interpolate_sc=True*, *sc_params={}*)

Bases: `object`

Class for generic GW sources

This class is for analysing a generic set of sources that may be stationary/evolving and circular/eccentric. If the type of sources are known, then a more specific subclass may be more useful

Parameters

m_1
[*float/array*] Primary mass. Must have astropy units of mass.

m_2
[*float/array*] Secondary mass. Must have astropy units of mass.

ecc
[*float/array*] Initial eccentricity

dist
[*float/array*] Luminosity distance to source. Must have astropy units of distance.

n_proc
[*int*] Number of processors to split eccentric evolution over if needed

f_orb
[*float/array*] Orbital frequency (either *a* or *f_orb* must be supplied). This takes precedence over *a*. Must have astropy units of frequency.

a
[*float/array*] Semi-major axis (either *a* or *f_orb* must be supplied). Must have astropy units of length.

position
[*SkyCoord/array*, optional] Sky position of source. Must be specified using Astropy's `astropy.coordinates.SkyCoord` class.

polarisation
[*float/array*, optional] GW polarisation angle of the source. Must have astropy angular units.

inclination
[*float/array*, optional] Inclination of the source. Must have astropy angular units.

weights
[*float/array*, optional] Statistical weights associated with each sample (used for plotted), default is equal weights

gw_lum_tol
[*float*] Allowed error on the GW luminosity when calculating SNRs. This is used to calculate maximum harmonics needed and transition between 'eccentric' and 'circular'. This variable should be updated using the function *legwork.source.Source.update_gw_lum_tol()* (not `Source._gw_lum_tol =`) to ensure the cached calculations match the current tolerance.

stat_tol
[*float*] Fractional change in frequency over mission length above which a binary should be considered to be stationary

interpolate_g
[*boolean*] Whether to interpolate the g(n,e) function from Peters (1964). This results in a faster runtime for large collections of sources.

interpolate_sc
[*boolean*] Whether to interpolate the LISA sensitivity curve

sc_params
[*dict*] Parameters for interpolated sensitivity curve. Include any of `instrument`, `custom_psd`, `t_obs`, `L`, `approximate_R` and `confusion_noise`. Default values are: "LISA", None, "auto", "auto", False and "auto".

Raises

ValueError
If both `f_orb` and `a` are missing. If only part of the position, inclination, and polarization are supplied. If array-like parameters don't have the same length.

AssertionError
If a parameter is missing units

Attributes

m_c
[*float/array*] Chirp mass. Set using `m_1` and `m_2` in *legwork.utils.chirp_mass()*

ecc_tol
[*float*] Eccentricity above which a binary is considered eccentric. Set by *legwork.source.Source.find_eccentric_transition()*

snr
[*float/array*] Signal-to-noise ratio. Set by *legwork.source.Source.get_snr()*

max_snr_harmonic
[*int/array*] Harmonic with the maximum snr. Set by *legwork.source.Source.get_snr()*

n_sources
[*int*] Number of sources in class

Methods Summary

create_harmonics_functions()	Create two harmonics related functions as methods for the Source class
evolve_sources(t_evol[, create_new_class])	Evolve sources forward in time for `t_evol` amount of time.
find_eccentric_transition()	Find the eccentricity at which we must treat binaries at eccentric.
get_h_0_n(harmonics[, which_sources])	Computes the strain for binaries for the given `harmonics`.
get_h_c_n(harmonics[, which_sources])	Computes the characteristic strain for binaries for the given `harmonics`.
get_merger_time([save_in_class, ...])	Get the merger time for each source.
get_snr([t_obs, instrument, custom_psd, L, ...])	Computes the SNR for a generic binary.
get_snr_evolving([t_obs, instrument, ...])	Computes the SNR assuming an evolving binary
get_snr_stationary([t_obs, instrument, ...])	Computes the SNR assuming a stationary binary
get_source_mask([circular, stationary, t_obs])	Produce a mask of the sources.
plot_source_variables(xstr[, ystr, ...])	Plot distributions of Source variables.
plot_sources_on_sc([snr_cutoff, fig, ax, ...])	Plot all sources in the class on the sensitivity curve
set_g(interpolate_g)	Set Source g function if user wants to interpolate g(n,e).
set_sc()	Set Source sensitivity curve function
update_gw_lum_tol(gw_lum_tol)	Update GW luminosity tolerance.
update_sc_params(sc_params)	Update sensitivity curve parameters

Methods Documentation

create_harmonics_functions()

Create two harmonics related functions as methods for the Source class

The first function is stored at `self.harmonics_required(ecc)`. This calculates the index of the highest harmonic required to calculate the SNR of a system with eccentricity *ecc* assuming the provided tolerance *gw_lum_tol*. This is equivalent to the total number of harmonics required since, when calculating SNR, harmonics in the range [1, harmonics_required(ecc)] are used. Note that the value returned by the function slightly conservative as we apply *ceil* to the interpolation result.

The second function is stored at `self.max_strain_harmonic(ecc)`. This calculates the harmonic with the maximum strain for a system with eccentricity *ecc*.

evolve_sources(*t_evol*, *create_new_class=False*)

Evolve sources forward in time for `t_evol` amount of time. If `create_new_class` is `True` then save the updated sources in a new Source class, otherwise, update the values in this class.

Parameters

t_evol
[*float/array*] Amount of time to evolve sources. Either a single value for all sources or an array of values corresponding to each source.

create_new_class
[bool, optional] Whether to save the evolved binaries in a new class or not. If not simply update the current class, by default False.

Returns

evolved_sources
[*Source*] The new class with evolved sources, only returned if `create_new_class` is `True`.

`find_eccentric_transition()`

Find the eccentricity at which we must treat binaries at eccentric. We define this as the maximum eccentricity at which the n=2 harmonic is the total GW luminosity given the tolerance `self._gw_lum_tol`. Store the result in `self.ecc_tol`

`get_h_0_n`(*harmonics*, *which_sources=None*)

Computes the strain for binaries for the given `harmonics`. Use `which_sources` to select a subset of the sources. Merged sources are set to have 0.0 strain.

Parameters

harmonics
[*int/array*] Harmonic(s) at which to calculate the strain

which_sources
[*boolean/array*] Mask on which sources to compute values for (default is all)

Returns

h_0_n
[*float/array*] Dimensionless strain in the quadrupole approximation (unitless) shape of array is `(number of sources, number of harmonics)`

`get_h_c_n`(*harmonics*, *which_sources=None*)

Computes the characteristic strain for binaries for the given `harmonics`. Use `which_sources` to select a subset of the sources. Merged sources are set to have 0.0 characteristic strain.

Parameters

harmonics
[*int/array*] Harmonic(s) at which to calculate the strain

which_sources `boolean/array`
Mask on which sources to compute values for (default is all)

Returns

h_c_n
[*float/array*] Dimensionless characteristic strain in the quadrupole approximation shape of array is `(number of sources, number of harmonics)`

`get_merger_time`(*save_in_class=True*, *which_sources=None*, *exact=True*)

Get the merger time for each source. Set `save_in_class` to true to save the values as an instance variable in the class. Use `which_sources` to select a subset of the sources in the class. Note that if `save_in_class` is set to `True`, `which_sources` will be ignored.

Parameters

save_in_class
[*bool*, optional] Whether the save the result into the class as an instance variable, by default True

which_sources
[*bool/array*, optional] A mask for the subset of sources for which to calculate the merger time, by default all sources (None)

exact
[*boolean*, optional] Whether to calculate the merger time exactly with numerical integration or to instead use a fit

Returns

t_merge
[*float/array*] Merger times

`get_snr`(*t_obs=None*, *instrument=None*, *custom_psd=None*, *L=None*, *approximate_R=None*, *confusion_noise=None*, *n_step=100*, *verbose=False*, *re_interpolate_sc=True*, *which_sources=None*)

Computes the SNR for a generic binary. Also records the harmonic with maximum SNR for each binary in `self.max_snr_harmonic`.

Parameters

t_obs
[*array*] Observation duration (default: value from sc_params)

instrument
[*{{ 'LISA', 'TianQin', 'custom' }}*] Instrument to observe with. If 'custom' then `custom_psd` must be supplied. (default: value from sc_params)

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some. (default: function from sc_params)

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

n_step
[*int*] Number of time steps during observation duration

verbose
[*boolean*] Whether to print additional information to user

re_interpolate_sc
[*boolean*] Whether to re-interpolate the sensitivity curve if the observation time or instrument changes. If False, warning will instead be given

which_sources
[*boolean/array*] Mask of which sources to calculate the SNR for. If None then calculate SNR for all sources.

Returns

SNR
[*array*] The signal-to-noise ratio

`get_snr_evolving`(*t_obs=None*, *instrument=None*, *custom_psd=None*, *L=None*, *approximate_R=None*, *confusion_noise=None*, *re_interpolate_sc=True*, *n_step=100*, *which_sources=None*, *verbose=False*)

Computes the SNR assuming an evolving binary

Parameters

t_obs
[*array*] Observation duration (default: follow sc_params)

instrument
[*{{ 'LISA', 'TianQin', 'custom' }}*] Instrument to observe with. If 'custom' then `custom_psd` must be supplied.

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some.

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

re_interpolate_sc
[*boolean*] Whether to re-interpolate the sensitivity curve if the observation time or instrument changes. If False, warning will instead be given

n_step
[*int*] Number of time steps during observation duration

which_sources
[*bool/array*] Mask on which sources to consider evolving and calculate (default is all sources in Class)

verbose
[*boolean*] Whether to print additional information to user

Returns

SNR
[*array*] The signal-to-noise ratio

`get_snr_stationary`(*t_obs=None*, *instrument=None*, *custom_psd=None*, *L=None*, *approximate_R=None*, *confusion_noise=None*, *re_interpolate_sc=True*, *which_sources=None*, *verbose=False*)

Computes the SNR assuming a stationary binary

Parameters

t_obs
[*array*] Observation duration (default: follow sc_params)

instrument
[*{{ 'LISA', 'TianQin', 'custom' }}*] Instrument to observe with. If 'custom' then `custom_psd` must be supplied.

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some.

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

re_interpolate_sc
[*boolean*] Whether to re-interpolate the sensitivity curve if the observation time or instrument changes. If False, warning will instead be given

which_sources
[*bool/array*] Mask on which sources to consider stationary and calculate (default is all sources in Class)

verbose
[*boolean*] Whether to print additional information to user

Returns

SNR
[*array*] The signal-to-noise ratio

`get_source_mask`(*circular=None*, *stationary=None*, *t_obs=None*)

Produce a mask of the sources.

Create a mask based on whether binaries are circular or eccentric and stationary or evolving. Tolerance levels are defined in the class.

Parameters

circular
[*bool*] `None` means either, `True` means only circular binaries and `False` means only eccentric

stationary
[*bool*] `None` means either, `True` means only stationary binaries and `False` means only evolving

t_obs
[*float*] Observation time, default is value from *self._sc_params*

Returns

mask
[*bool/array*] Mask for the sources

`plot_source_variables`(*xstr*, *ystr=None*, *which_sources=None*, *exclude_merged_sources=True*, *log_scale=False*, ***kwargs*)

Plot distributions of Source variables. If two variables are specified then produce a 2D distribution, otherwise a 1D distribution.

Parameters

xstr
[*{ 'm_1', 'm_2', 'm_c', 'ecc', 'dist', 'f_orb', 'f_GW', 'a', 'snr' }*] Which variable to plot on the x axis

ystr
[*{ 'm_1', 'm_2', 'm_c', 'ecc', 'dist', 'f_orb', 'f_GW', 'a', snr' }*] Which variable to plot on the y axis (if None then a 1D distribution is made using *xstr*)

which_sources
[*boolean array*] Mask for which sources should be plotted (default is all sources)

exclude_merged_sources
[*boolean*] Whether to exclude merged sources in distributions (default is True)

log_scale
[*bool or tuple of bools*] Whether to use a log scale for the axes. For a 1D plot, only a bool can be supplied and it applies to the x-axis. For a 2D plot, a single *bool* is applied to both axes, a tuple is applied to the x- and y-axis respectively.

****kwargs**
[*various*] When only `xstr` is provided, the kwargs are the same as *legwork.visualisation.plot_1D_dist()*. When both `xstr` and `ystr` are provided, the kwargs are the same as *legwork.visualisation.plot_2D_dist()*. Note that if `xlabel` or `ylabel` is not passed then this function automatically creates one using a default string and (if applicable) the Astropy units of the variable.

Returns

fig
[*matplotlib Figure*] The figure on which the distribution is plotted

ax
[*matplotlib Axis*] The axis on which the distribution is plotted

`plot_sources_on_sc`(*snr_cutoff=0*, *fig=None*, *ax=None*, *show=True*, *label='Stationary sources'*, *sc_vis_settings={}*, ***kwargs*)

Plot all sources in the class on the sensitivity curve

Parameters

snr_cutoff
[*float*] SNR below which sources will not be plotted (default is to plot all sources)

fig: `matplotlib Figure`
A figure on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

ax: `matplotlib Axis`
An axis on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

show
[*boolean*] Whether to immediately show the plot

label
[*str*] Label to use for the plotted points

sc_vis_settings
[*dict*] Dictionary of parameters to pass to *plot_sensitivity_curve()*, e.g. {"y_quantity": "h_c"} will plot characteristic strain instead of ASD

****kwargs**
[*various*] Keyword arguments to be passed to plotting functions

Returns

fig
[*matplotlib Figure*] The figure on which the sources are plotted

ax
[*matplotlib Axis*] The axis on which the sources are plotted

Notes

Warning: Note that this function is not yet implemented for evolving sources. Evolving sources will not be plotted and a warning will be shown instead. We are working on implementing this soon!

`set_g`(*interpolate_g*)

Set Source g function if user wants to interpolate g(n,e). Otherwise just leave the function as None.

Parameters

interpolate_g
[*boolean*] Whether to interpolate the g(n,e) function from Peters (1964)

`set_sc`()

Set Source sensitivity curve function

If user wants to interpolate then perform interpolation of LISA sensitivity curve using `sc_params`. Otherwise just leave the function as None.

`update_gw_lum_tol`(*gw_lum_tol*)

Update GW luminosity tolerance. Use the updated value to recalculate harmonics_required function and transition to eccentric

Parameters

gw_lum_tol
[*float*] Allowed error on the GW luminosity when calculating SNRs

`update_sc_params`(*sc_params*)

Update sensitivity curve parameters

Update the parameters used to interpolate sensitivity curve and perform interpolation again to match new params

Stationary

class legwork.source.**Stationary**(*m_1*, *m_2*, *ecc*, *dist*, *n_proc=1*, *f_orb=None*, *a=None*, *position=None*, *polarisation=None*, *inclination=None*, *weights=None*, *gw_lum_tol=0.05*, *stat_tol=0.01*, *interpolate_g=True*, *interpolate_sc=True*, *sc_params={}*)

Bases: *Source*

Subclass for sources that are stationary

Methods Summary

get_snr([t_obs, instrument, custom_psd, verbose])	Computes the SNR for a generic binary.

Methods Documentation

get_snr(*t_obs=None*, *instrument=None*, *custom_psd=None*, *verbose=False*)

Computes the SNR for a generic binary. Also records the harmonic with maximum SNR for each binary in `self.max_snr_harmonic`.

Parameters

t_obs
[*array*] Observation duration (default: value from sc_params)

instrument
[*{{ 'LISA', 'TianQin', 'custom' }}*] Instrument to observe with. If 'custom' then `custom_psd` must be supplied. (default: value from sc_params)

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some. (default: function from sc_params)

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

n_step
[*int*] Number of time steps during observation duration

verbose
[*boolean*] Whether to print additional information to user

re_interpolate_sc
[*boolean*] Whether to re-interpolate the sensitivity curve if the observation time or instrument changes. If False, warning will instead be given

which_sources
[*boolean/array*] Mask of which sources to calculate the SNR for. If None then calculate SNR for all sources.

Returns

SNR
[*array*] The signal-to-noise ratio

Evolving

class legwork.source.**Evolving**(*m_1*, *m_2*, *ecc*, *dist*, *n_proc=1*, *f_orb=None*, *a=None*, *position=None*, *polarisation=None*, *inclination=None*, *weights=None*, *gw_lum_tol=0.05*, *stat_tol=0.01*, *interpolate_g=True*, *interpolate_sc=True*, *sc_params={}*)

Bases: *Source*

Subclass for sources that are evolving

Methods Summary

get_snr([t_obs, instrument, custom_psd, ...])	Computes the SNR for a generic binary.

Methods Documentation

get_snr(*t_obs=None*, *instrument=None*, *custom_psd=None*, *n_step=100*, *verbose=False*)

Computes the SNR for a generic binary. Also records the harmonic with maximum SNR for each binary in `self.max_snr_harmonic`.

Parameters

t_obs
[*array*] Observation duration (default: value from sc_params)

instrument
[*{{ 'LISA', 'TianQin', 'custom' }}*] Instrument to observe with. If 'custom' then `custom_psd` must be supplied. (default: value from sc_params)

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some. (default: function from sc_params)

L
[*float*] LISA arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

n_step
[*int*] Number of time steps during observation duration

verbose
[*boolean*] Whether to print additional information to user

re_interpolate_sc
[*boolean*] Whether to re-interpolate the sensitivity curve if the observation time or instrument changes. If False, warning will instead be given

which_sources
[*boolean/array*] Mask of which sources to calculate the SNR for. If None then calculate SNR for all sources.

Returns

SNR
[*array*] The signal-to-noise ratio

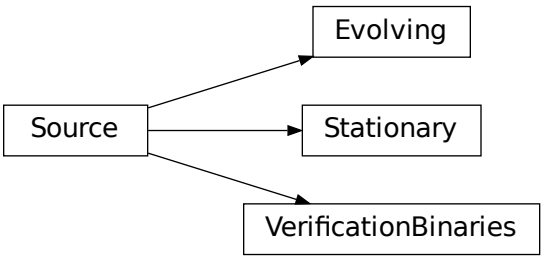
VerificationBinaries

class legwork.source.**VerificationBinaries**

Bases: *Source*

Generate a Source class with the LISA verification binaries preloaded. Data for the binaries is gathered from Kupfer+18 Table 1 and 2.

4.4.2 Class Inheritance Diagram



Tip: Unable to find the source of your issues? Never fear! Check out our tutorial on using the Source class *here!*

4.5 legwork.strain Module

Computes several types of gravitational wave strains

4.5.1 Functions

amplitude_modulation(position, polarisation, ...)	Computes the modulation of the strain due to the orbit averaged response of the detector to the position, polarisation, and inclination of the source using Cornish+03 Eq.42 and Babak+21.
h_0_n(m_c, f_orb, ecc, n, dist[, position, ...])	Computes strain amplitude
h_c_n(m_c, f_orb, ecc, n, dist[, position, ...])	Computes characteristic strain amplitude

amplitude_modulation

legwork.strain.**amplitude_modulation**(*position*, *polarisation*, *inclination*)

Computes the modulation of the strain due to the orbit averaged response of the detector to the position, polarisation, and inclination of the source using Cornish+03 Eq.42 and Babak+21.

Note that since the majority of the calculations in LEGWORK are carried out for the full position, polarisation, and inclination averages, we include a pre-factor of 5/4 on the amplitude modulation to undo the factor of 4/5 which arises from the averaging of *legwork.utils.F_plus_squared()* and *legwork.utils.F_cross_squared()*.

Additionally, note that this function does not include the factor of 1/2 from the Cornish+03 paper in order to remain in the frequency domain. More recent papers (e.g. Babak+21) define the strain as h(f)~A(f)*e^(2*i*psi(f)) and so the inner product is 1 instead of 1/2 as in Cornish+03.

Parameters

position
[*SkyCoord/array*, optional] Sky position of source. Must be specified using Astropy's `astropy.coordinates.SkyCoord` class.

polarisation
[*float/array*, optional] GW polarisation angle of the source. Must have astropy angular units.

inclination
[*float/array*, optional] Inclination of the source. Must have astropy angular units.

Returns

modulation
[*float/array*] modulation to apply to strain from detector response

h_0_n

legwork.strain.**h_0_n**(*m_c*, *f_orb*, *ecc*, *n*, *dist*, *position=None*, *polarisation=None*, *inclination=None*, *interpolated_g=None*)

Computes strain amplitude

Computes the dimensionless power of a general binary radiating gravitational waves in the quadrupole approximation at `n` harmonics of the orbital frequency

In the docs below, x refers to the number of sources, y to the number of timesteps and z to the number of harmonics.

Parameters

m_c
[*float/array*] Chirp mass of each binary. Shape should be (x,).

f_orb
[*float/array*] Orbital frequency of each binary at each timestep. Shape should be (x, y), or (x,) if only one timestep.

ecc
[*float/array*] Eccentricity of each binary at each timestep. Shape should be (x, y), or (x,) if only one timestep.

n
[*int/array*] Harmonic(s) at which to calculate the strain. Either a single int or shape should be (z,)

dist
[*float/array*] Distance to each binary. Shape should be (x,)

position
[*SkyCoord/array*, optional] Sky position of source. Must be specified using Astropy's `astropy.coordinates.SkyCoord` class.

polarisation
[*float/array*, optional] GW polarisation angle of the source. Must have astropy angular units.

inclination
[*float/array*, optional] Inclination of the source. Must have astropy angular units.

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

Returns

h_0
[*float/array*] Strain amplitude. Shape is (x, y, z).

h_c_n

legwork.strain.**h_c_n**(*m_c*, *f_orb*, *ecc*, *n*, *dist*, *position=None*, *polarisation=None*, *inclination=None*, *interpolated_g=None*)

Computes characteristic strain amplitude

Computes the dimensionless characteristic power of a general binary radiating gravitational waves in the quadrupole approximation at `n` harmonics of the orbital frequency

In the docs below, x refers to the number of sources, y to the number of timesteps and z to the number of harmonics.

Parameters

m_c
[*float/array*] Chirp mass of each binary. Shape should be (x,).

f_orb
[*float/array*] Orbital frequency of each binary at each timestep. Shape should be (x, y), or (x,) if only one timestep.

ecc
[*float/array*] Eccentricity of each binary at each timestep. Shape should be (x, y), or (x,) if only one timestep.

n
[*int/array*] Harmonic(s) at which to calculate the strain. Either a single int or shape should be (z,)

dist
[*float/array*] Distance to each binary. Shape should be (x,)

position
[*SkyCoord/array*, optional] Sky position of source. Must be specified using Astropy's `astropy.coordinates.SkyCoord` class.

polarisation
[*float/array*, optional] GW polarisation angle of the source. Must have astropy angular units.

inclination
[*float/array*, optional] Inclination of the source. Must have astropy angular units.

interpolated_g
[*function*] A function returned by `scipy.interpolate.interp2d` that computes g(n,e) from Peters (1964). The code assumes that the function returns the output sorted as with the interp2d returned functions (and thus unsorts). Default is None and uses exact g(n,e) in this case.

Returns

h_c
[*float/array*] Characteristic strain. Shape is (x, y, z).

Tip: Feeling a little strained trying to parse these docs? Check out our tutorial on using functions in the `strain` module *here!*

4.6 legwork.utils Module

A collection of miscellaneous utility functions

4.6.1 Functions

chirp_mass(m_1, m_2)	Computes chirp mass of binaries
peters_g(n, e)	Compute g(n, e) from Peters and Mathews (1963) Eq.20
peters_f(e)	f(e) from Peters and Mathews (1963) Eq.17
get_a_from_f_orb(f_orb, m_1, m_2)	Converts orbital frequency to semi-major axis
get_f_orb_from_a(a, m_1, m_2)	Converts semi-major axis to orbital frequency
get_a_from_ecc(ecc, c_0)	Convert eccentricity to semi-major axis
beta(m_1, m_2)	Compute beta defined in Peters and Mathews (1964) Eq.5.9
c_0(a_i, ecc_i)	Computes the c_0 factor in Peters and Mathews (1964) Eq.5.11
fn_dot(m_c, f_orb, e, n)	Rate of change of nth frequency of a binary
ensure_array(*args)	Convert arguments to numpy arrays
D_plus_squared(theta, phi)	Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>
D_cross_squared(theta, phi)	Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>
D_plus_D_cross(theta, phi)	Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>
F_plus_squared(theta, phi, psi)	Compute the auto-correlated detector response for the plus polarization
F_cross_squared(theta, phi, psi)	Compute the auto-correlated detector response for the cross polarization

chirp_mass

legwork.utils.**chirp_mass**(*m_1*, *m_2*)

Computes chirp mass of binaries

Parameters

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

Returns

m_c
[*float/array*] Chirp mass

peters_g

legwork.utils.**peters_g**(*n*, *e*)

Compute g(n, e) from Peters and Mathews (1963) Eq.20

This function gives the relative power of gravitational radiation at the nth harmonic

Parameters

n
[*int/array*] Harmonic(s) of interest

e
[*float/array*] Eccentricity

Returns

g
[*array*] g(n, e) from Peters and Mathews (1963) Eq. 20

peters_f

legwork.utils.**peters_f**(*e*)

f(e) from Peters and Mathews (1963) Eq.17

This function gives the integrated enhancement factor of gravitational radiation from an eccentric source compared to an equivalent circular source.

Parameters

e
[*float/array*] Eccentricity

Returns

f
[*float/array*] Enhancement factor

Notes

Note that this function represents an infinite sum of g(n, e) - *legwork.utils.peters_g()*

get_a_from_f_orb

legwork.utils.**get_a_from_f_orb**(*f_orb*, *m_1*, *m_2*)

Converts orbital frequency to semi-major axis

Using Kepler's third law, convert orbital frequency to semi-major axis. Inverse of *legwork.utils.get_f_orb_from_a()*.

Parameters

f_orb
[*float/array*] Orbital frequency

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

Returns

a
[*float/array*] Semi-major axis

get_f_orb_from_a

legwork.utils.**get_f_orb_from_a**(*a*, *m_1*, *m_2*)

Converts semi-major axis to orbital frequency

Using Kepler's third law, convert semi-major axis to orbital frequency. Inverse of *legwork.utils.get_a_from_f_orb()*.

Parameters

a
[*float/array*] Semi-major axis

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

Returns

f_orb
[*float/array*] Orbital frequency

get_a_from_ecc

legwork.utils.**get_a_from_ecc**(*ecc*, *c_0*)

Convert eccentricity to semi-major axis

Use initial conditions and Peters (1964) Eq. 5.11 to convert `ecc` to `a`.

Parameters

ecc
[*float/array*] Eccentricity

c_0
[*float*] Constant defined in Peters and Mathews (1964) Eq. 5.11. See *legwork.utils.c_0()*

Returns

a
[*float/array*] Semi-major axis

beta

legwork.utils.**beta**(*m_1*, *m_2*)

Compute beta defined in Peters and Mathews (1964) Eq.5.9

Parameters

m_1
[*float/array*] Primary mass

m_2
[*float/array*] Secondary mass

Returns

beta
[*float/array*] Constant defined in Peters and Mathews (1964) Eq.5.9.

c_0

legwork.utils.**c_0**(*a_i*, *ecc_i*)

Computes the c_0 factor in Peters and Mathews (1964) Eq.5.11

Parameters

a_i
[*float/array*] Initial semi-major axis

ecc_i
[*float/array*] Initial eccentricity

Returns

c_0
[*float*] Constant defined in Peters and Mathews (1964) Eq.5.11

fn_dot

legwork.utils.**fn_dot**(*m_c*, *f_orb*, *e*, *n*)

Rate of change of nth frequency of a binary

Parameters

m_c
[*float/array*] Chirp mass

f_orb
[*float/array*] Orbital frequency

e
[*float/array*] Eccentricity

n
[*int*] Harmonic of interest

Returns

fn_dot
[*float/array*] Rate of change of nth frequency

ensure_array

legwork.utils.**ensure_array**(**args*)

Convert arguments to numpy arrays

Convert arguments based on the following rules

- Ignore any None values
- Convert any lists to numpy arrays
- Wrap any other types in lists and convert to numpy arrays

Parameters

args
[*any*] Supply any number of arguments of any type

Returns

array_args
[*any*] Args converted to numpy arrays

any_not_arrays
[*bool*] Whether any arg is not a list or None or a numpy array

D_plus_squared

legwork.utils.**D_plus_squared**(*theta*, *phi*)

Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>

Parameters

theta
[*float/array*] declination of the source

phi
[*float/array*] right ascension of the source

Returns

D_plus_2
[*float/array*] factor used for response; see eq. 44 of Cornish and Larson (2003)

D_cross_squared

legwork.utils.**D_cross_squared**(*theta*, *phi*)

Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>

Parameters

theta
[*float/array*] declination of the source

phi
[*float/array*] right ascension of the source

Returns

D_cross_2
[*float/array*] factor used for response; see eq. 44 of Cornish and Larson (2003)

D_plus_D_cross

legwork.utils.**D_plus_D_cross**(*theta*, *phi*)

Required for the detector responses <F_+^2>, <F_x^2>, <F_+F_x>

Parameters

theta
[*float/array*] declination of the source

phi
[*float/array*] right ascension of the source

Returns

D_plus_cross
[*float/array*] factor used for response; see eq. 44 of Cornish and Larson (2003)

F_plus_squared

legwork.utils.**F_plus_squared**(*theta*, *phi*, *psi*)

Compute the auto-correlated detector response for the plus polarization

Parameters

theta
[*float/array*] declination of the source

phi
[*float/array*] right ascension of the source

psi
[*float/array*] polarization of the source

Returns

F_plus_2
[*float/array*] Auto-correlated detector response for the plus polarization

F_cross_squared

legwork.utils.**F_cross_squared**(*theta*, *phi*, *psi*)

Compute the auto-correlated detector response for the cross polarization

Parameters

theta
[*float/array*] declination of the source

phi
[*float/array*] right ascension of the source

psi
[*float/array*] polarization of the source

Returns

F_cross_2
[*float/array*] Auto-correlated detector response for the cross polarization

4.7 legwork.visualisation Module

4.7.1 Functions

plot_1D_dist(x[, weights, disttype, ...])	Plot a 1D distribution of `x`.
plot_2D_dist(x, y[, weights, disttype, fig, ...])	Plot a 2D distribution of *x* and *y*
plot_sensitivity_curve([frequency_range, ...])	Plot the LISA sensitivity curve
plot_sources_on_sc(f_dom, snr[, weights, ...])	Overlay *stationary* sources on the LISA sensitivity curve.

plot_1D_dist

legwork.visualisation.**plot_1D_dist**(*x*, *weights=None*, *disttype='hist'*, *log_scale=False*, *fig=None*, *ax=None*, *show=True*, *figsize=(10, 7)*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *color=None*, ***kwargs*)

Plot a 1D distribution of `x`.

This function is a wrapper for `matplotlib.pyplot.hist()`, `seaborn.kdeplot()` and `seaborn.ecdfplot()`.

Parameters

x
[*float/int array*] Variable to plot, should be a 1D array

weights
[*float/int array*] Weights for each variable in `x`, must have the same shape

disttype
[*{{ "hist", "kde", "ecdf" }}*] Which type of distribution plot to use

log_scale
[*bool*] Whether to plot *log10(x)* instead of *x*

fig: `matplotlib Figure`
A figure on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

ax: `matplotlib Axis`
An axis on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

show
[*boolean*] Whether to immediately show the plot or only return the Figure and Axis

figsize
[*tuple*] Tuple with size for the x- and y-axis if creating a new figure (i.e. ignored when fig/ax is not None)

xlabel
[*string*] Label for the x axis, passed to Axes.set_xlabel()

ylabel
[*string*] Label for the y axis, passed to Axes.set_ylabel()

xlim
[*tuple*] Lower and upper limits for the x axis, passed to Axes.set_xlim()

ylim
[*tuple*] Lower and upper limits for the y axis, passed to Axes.set_ylim()

color
[*string or tuple*] Colour to use for the plot, see https://matplotlib.org/tutorials/colors/colors.html for details on how to specify a colour

****kwargs**
[*(if disttype=="hist")*] Include values for any of *bins, range, density, cumulative, bottom, histtype, align, orientation, rwidth, log, label* or more. See `matplotlib.pyplot.hist()` for more details.

****kwargs**
[*(if disttype=="kde")*] Include values for any of *gridsize, cut, clip, legend, cumulative, bw_method, bw_adjust, log_scale, fill, label, linewidth, linestyle* or more. See `seaborn.kdeplot()` for more details.

****kwargs**
[*(if disttype=="ecdf")*] Include values for any of *stat, complementary, log_scale, legend, label, linewidth, linestyle* or more. See `seaborn.edcfplot()` for more details.

Returns

fig
[*matplotlib Figure*] The figure on which the distribution is plotted

ax
[*matplotlib Axis*] The axis on which the distribution is plotted

plot_2D_dist

`legwork.visualisation.`**`plot_2D_dist`**(*x*, *y*, *weights=None*, *disttype='scatter'*, *fig=None*, *ax=None*, *show=True*, *figsize=(12, 7)*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *log_scale=False*, *color=None*, *scatter_s=20*, ***kwargs*)

Plot a 2D distribution of *x* and *y*

This function is a wrapper for `matplotlib.pyplot.scatter()` and `seaborn.kdeplot()`.

Parameters

x
[*float/int array*] Variable to plot on the x axis, should be a 1D array

y
[*float/int array*] Variable to plot on the y axis, should be a 1D array

weights
[*float/int array*] Weights for each variable pair (`x`, `y`), must have the same shape

disttype
[*{{ "scatter", "kde" }}*] Which type of distribution plot to use

fig: `matplotlib Figure`
A figure on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

ax: `matplotlib Axis`
An axis on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

show
[*boolean*] Whether to immediately show the plot or only return the Figure and Axis

figsize
[*tuple*] Tuple with size for the x- and y-axis if creating a new figure (i.e. ignored when fig/ax is not None)

xlabel
[*string*] Label for the x axis, passed to Axes.set_xlabel()

ylabel
[*string*] Label for the y axis, passed to Axes.set_ylabel()

xlim
[*tuple*] Lower and upper limits for the x axis, passed to Axes.set_xlim()

ylim
[*tuple*] Lower and upper limits for the u axis, passed to Axes.set_ylim()

log_scale
[*bool or tuple of bools*] Whether to use a log scale for the axes. A single *bool* is applied to both axes, a tuple is applied to the x- and y-axis respectively.

scatter_s
[*float*, default=20] Scatter point size, passed as `s` to a scatter plot and ignored for a KDE

color
[*string or tuple*] Colour to use for the plot, see https://matplotlib.org/tutorials/colors/colors.html for details on how to specify a colour

****kwargs**
[*(if disttype=="scatter")*] Input any of *s, c, marker, cmap, norm, vmin, vmax, alpha, linewidths, edgecolors* or more. See `matplotlib.pyplot.scatter()` for more details.

****kwargs**
[*(if disttype=="kde")*] Input any of *gridsize, cut, clip, legend, cumulative, cbar, cbar_ax, cbar_kws, bw_method, hue, palette, hue_order, hue_norm, levels, thresh, bw_adjust, log_scale, fill, label*. See `seaborn.kdeplot()` for more details.

Returns

fig
[*matplotlib Figure*] The figure on which the distribution is plotted

ax
[*matplotlib Axis*] The axis on which the distribution is plotted

plot_sensitivity_curve

`legwork.visualisation.`**`plot_sensitivity_curve`**(*frequency_range=None*, *y_quantity='ASD'*, *fig=None*, *ax=None*, *show=True*, *figsize=(10, 7)*, *color='#18068b'*, *fill=True*, *alpha=0.2*, *linewidth=1*, *label=None*, ***kwargs*)

Plot the LISA sensitivity curve

Parameters

frequency_range
[*float array*] Frequency values at which to plot the sensitivity curve

y_quantity
[*{{ "ASD", "h_c" }}*] Which quantity to plot on the y axis (amplitude spectral density or characteristic strain)

fig: `matplotlib Figure`
A figure on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

ax: `matplotlib Axis`
An axis on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

show
[*boolean*] Whether to immediately show the plot or only return the Figure and Axis

figsize
[*tuple*] Tuple with size for the x- and y-axis if creating a new figure (i.e. ignored when fig/ax is not None)

color
[*string or tuple*] Colour to use for the curve, see https://matplotlib.org/tutorials/colors/colors.html for details on how to specify a colour

fill
[*boolean*] Whether to fill the area below the sensitivity curve

alpha
[*float*] Opacity of the filled area below the sensitivity curve (ignored if fill is *False*)

linewidth
[*float*] Width of the sensitivity curve

label
[*string*] Label for the sensitivity curve in legends

****kwargs**
[*various*] Keyword args are passed to `legwork.psd.power_spectral_density()`, see those docs for details on possible arguments.

Returns

fig
[*matplotlib Figure*] The figure on which the distribution is plotted

ax
[*matplotlib Axis*] The axis on which the distribution is plotted

plot_sources_on_sc

`legwork.visualisation.`**`plot_sources_on_sc`**(*f_dom*, *snr*, *weights=None*, *snr_cutoff=0*, *t_obs='auto'*, *instrument='LISA'*, *custom_psd=None*, *L='auto'*, *approximate_R=False*, *confusion_noise='auto'*, *fig=None*, *ax=None*, *show=True*, *sc_vis_settings={}*, ***kwargs*)

Overlay *stationary* sources on the LISA sensitivity curve.

Each source is plotted at its max snr harmonic frequency such that that its height above the curve is equal to it signal-to-noise ratio. For circular sources this is frequency is simply twice the orbital frequency.

Parameters

f_dom
[*float/array*] Dominant harmonic frequency (f_orb * n_dom where n_dom is the harmonic with the maximum snr). You may find the `legwork.source.Source.max_snr_harmonic()` attribute useful (which gets populated) after *legwork.source.Source.get_snr()*.

snr
[*float/array*] Signal-to-noise ratio

weights
[*float/array*, optional, default=None] Statistical weights for each source, default is equal weights

snr_cutoff
[*float*] SNR above which to plot binaries (default is 0 such that all sources are plotted)

instrument: {{ `LISA`, `TianQin`, `custom` }}
Instrument to use. LISA is used by default. Choosing *custom* uses `custom_psd` to compute PSD.

custom_psd
[*function*] Custom function for computing the PSD. Must take the same arguments as *legwork.psd.lisa_psd()* even if it ignores some.

t_obs
[*float*] Observation time (default auto)

L
[*float*] Arm length in metres

approximate_R
[*boolean*] Whether to approximate the response function (default: no)

confusion_noise
[*various*] Galactic confusion noise. Acceptable inputs are either one of the values listed in *legwork.psd.get_confusion_noise()*, "auto" (automatically selects confusion noise based on *instrument* - 'robson19' if LISA and 'huang20' if TianQin), or a custom function that gives the confusion noise at each frequency for a given mission length where it would be called by running *noise(f, t_obs)* and return a value with units of inverse Hertz

fig: `matplotlib Figure`
A figure on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

ax: `matplotlib Axis`
An axis on which to plot the distribution. Both *ax* and *fig* must be supplied for either to be used

show
[*boolean*] Whether to immediately show the plot or only return the Figure and Axis

sc_vis_settings
[*dict*] Dictionary of parameters to pass to *plot_sensitivity_curve()*, e.g. {"y_quantity": "h_c"} will plot characteristic strain instead of ASD

****kwargs**
[*various*] This function is a wrapper on *legwork.visualisation.plot_2D_dist()* and each kwarg is passed directly to this function. For example, you can write *disttype="kde"* for a kde density plot instead of a scatter plot.

Returns

fig
[*matplotlib Figure*] The figure on which the distribution is plotted

ax
[*matplotlib Axis*] The axis on which the distribution is plotted

---

Tip: Not quite sure how things are working vis-à-vis visualisation? Check out our tutorial on using functions in the `visualisation` module *here!*

Note: This tutorial was generated from a Jupyter notebook that can be downloaded here. If you'd like to reproduce the results in the notebook, or make changes to the code, we recommend downloading this notebook and running it with Jupyter as certain cells (mostly those that change plot styles) are excluded from the tutorials.

CHAPTER

FIVE

DERIVATIONS AND EQUATION REFERENCE

This guide explains the origin and derivation of the equations used in `LEGWORK` functions. This follows closely Section 3 of the LEGWORK paper **but with some extra steps shown!**

At the end of this document (*here*) is a table that relates each of the functions in `LEGWORK` to an equation in this document.

5.1 Conversions and Definitions (`utils`)

This section contains a miscellaneous collection of conversions and definitions that are useful in the later derivations. First, the chirp mass of a binary is defined as

$$\mathcal{M}_c = \frac{(m_1 m_2)^{3/5}}{(m_1 + m_2)^{1/5}}, \tag{5.1}$$

where m_1 and m_2 are the primary and secondary mass of the binary. This term often shows up in many equations and hence is easier to measure in gravitational wave data analysis than the individual component masses.

Kepler's third law allows one to convert between orbital frequency, $f_{\rm orb}$, and the semi-major axis, a, of a binary. For convenience we show it here

$$a = \left(\frac{G(m_1 + m_2)}{(2\pi f_{\rm orb})^2}\right)^{1/3}, \qquad f_{\rm orb} = \frac{1}{2\pi}\sqrt{\frac{G(m_1 + m_2)}{a^3}}. \tag{5.2}$$

As we deal with eccentric binaries, the different harmonic frequencies of gravitational wave emission become important. We can write that the relative power radiated into the $n^{\rm th}$ harmonic for a binary with eccentricity e is [Peters-Mathews63] (Eq. 20)

$$\begin{aligned} g(n,e) = \frac{n^4}{32}\Bigg\{&\Big[J_{n-2}(ne) - 2eJ_{n-1}(ne) + \frac{2}{n}J_n(ne) + 2eJ_{n+1}(ne) - J_{n+2}(ne)\Big]^2 \\ &+ \left(1-e^2\right)\left[J_{n-2}(ne) - 2J_n(ne) + J_{n+2}(ne)\right]^2 + \frac{4}{3n^2}\left[J_n(ne)\right]^2\Bigg\}, \end{aligned} \tag{5.3}$$

where $J_n(v)$ is the Bessel function of the first kind. Thus, the sum of $g(n,e)$ over all harmonics gives the factor by which the gravitational wave emission is stronger for a binary of eccentricity e over an otherwise identical circular binary. This enhancement factor is defined by Peters as [PetersMathews63] (Eq. 17)

$$F(e) = \sum_{n=1}^{\infty} g(n,e) = \frac{1 + (73/24)e^2 + (37/96)e^4}{(1-e^2)^{7/2}}. \tag{5.4}$$

Note that $F(0) = 1$ as one would expect. A useful number to remember is that $F(0.5) \approx 5.0$, or in words, a binary with eccentricity 0.5 loses energy to gravitational waves at a rate about 5 times higher than a similar circular binary.

5.2 Binary Evolution (evol)

5.2.1 Circular binaries

For a circular binary, the evolution can be calculated analytically as the rate at which the binary shrinks can be readily integrated [Peters64] (Eq. 5.6)

$$\frac{\mathrm{d}a}{\mathrm{d}t}_{e=0} = -\frac{\beta}{a^3}, \tag{5.5}$$

where β is

$$\beta(m_1, m_2) = \frac{64}{5}\frac{G^3}{c^5} m_1 m_2 (m_1 + m_2). \tag{5.6}$$

Integratig this gives the semi-major axis of a circular binary as a function of time as [Peters64] (Eq. 5.9)

$$a(t, m_1, m_2) = [a_0^4 - 4t\beta(m_1, m_2)]^{1/4}, \tag{5.7}$$

where a_0 is the initial semi-major axis. Moreover, we can set the final semi-major axis in Eq. (5.7) equal to zero and solve for the inspiral time ([Peters64] Eq. 5.10)

$$t_{\mathrm{merge,circ}} = \frac{a_0^4}{4\beta} \tag{5.8}$$

5.2.2 Eccentric binaries

Eccentric binaries are more complicated because the semi-major axis and eccentricity both evolve simultaneously and depend on one another. These equations cannot be solved analytically and require numerical integration. Firstly, we can relate a and e with [Peters64] (Eq. 5.11)

$$a(e) = c_0 \frac{e^{12/19}}{(1 - e^2)} \left(1 + \frac{121}{304} e^2\right)^{870/2299}, \tag{5.9}$$

where c_0 is

$$c_0(a_0, e_0) = a_0 \frac{(1 - e_0^2)}{e_0^{12/19}} \left(1 + \frac{121}{304} e_0^2\right)^{-870/2299} \tag{5.10}$$

where a_0 and e_0 are the initial semi-major axis and eccentricity respectively.

where c_0 is defined in Eq. (5.10) – such that the initial conditions are satisfied. Then we can numerically integrate [Peters64] (Eq. 5.13)

$$\frac{\mathrm{d}e}{\mathrm{d}t} = -\frac{19}{12}\frac{\beta}{c_0^4} \frac{e^{-29/19}\left(1 - e^2\right)^{3/2}}{\left[1 + (121/304)e^2\right]^{1181/2299}}, \tag{5.11}$$

to find $e(t)$ and use this in conjunction with Eq. (5.9) to solve for $a(t)$, which can in turn be converted to $f_{\mathrm{orb}}(t)$.

Furthermore, we can invert this to find the inspiral time by using that $e \to 0$ when the binary merges which gives [Peters64] (Eq. 5.14)

$$t_{\mathrm{merge}} = \frac{12}{19}\frac{c_0^4}{\beta} \int_0^{e_0} \frac{\left[1 + (121/304)e^2\right]^{1181/2299}}{e^{-29/19}\left(1 - e^2\right)^{3/2}} \mathrm{d}e \tag{5.12}$$

For very small or very large eccentricities we can approximate this integral using the following expressions (given in unlabelled equations after [Peters64] Eq. 5.14)

$$t_{\mathrm{merge},\, e^2 \ll 1} = \frac{c_0^4}{4\beta} \cdot e_0^{48/19} \tag{5.13}$$

$$t_{\mathrm{merge},\, (1-e^2) \ll 1} = \frac{768}{425}\frac{a_0^4}{4\beta}(1 - e_0^2)^{7/2} \tag{5.14}$$

The standard threshold employed by LEGWORK for small eccentricities is e = 0.15 and for large eccentricities is e = 0.9999 (as this approximates $t_{\rm merge}$ with an error below roughly 2%), though we note that this can be customised by the user if desired.

In addition, we implement the fit to Eq. 13 from Mandel (2021) that approximates the merger time as

$$t_{\rm merge} \approx t_{\rm merge,circ}(1-e_0^2)^{7/2}(1+0.27e_0^{10}+0.33e_0^{20}+0.2e_0^{1000}) \tag{5.15}$$

which gives $t_{\rm merge}$ with an error below 3% for eccentricities below 0.9999. We additionally add a rudimentary polynomial fit to further reduce this error to below 0.5%. The user may specify whether to use this fit or perform the full integral when calculating merger times in LEGWORK.

5.3 Gravitational Wave Strains (`strain`)

5.3.1 Characteristic Strain

The strength of a gravitational wave in a detector at any one moment is determined by the strain amplitude, h_0. However, for stellar-origin sources at mHz frequencies, the signal can be present in the detector for many years. This means that, the $n^{\rm th}$ harmonic of the binary will spend approximately $f_n/\dot{f}_n$ seconds (or $f_n^2/\dot{f}_n$ cycles) in the vicinity of a frequency f_n [FinnThorne00]. This leads to the signal 'accumulating' at the frequency f_n.

Therefore, to account for the integration of the signal over the mission, we instead use the 'characteristic' strain amplitude of the $n^{\rm th}$ harmonic, $h_{c,n}$, which is the term present in the general signal-to-noise ratio equation. This can be related the to the strain amplitude in the $n^{\rm th}$ harmonic, $h_{0,n}$, as (e.g. [FinnThorne00] [MooreColeBerry15])

$$h_{c,n}^2 = \left(\frac{f_n^2}{\dot{f}_n}\right) h_n^2. \tag{5.16}$$

Note: Note that this is factor of 2 different from [FinnThorne00]. This is because the factor of 2 is already included in the [RobsonCornishLiu19] sensitivity curve and so we remove it here.

The characteristic strain represents the strain measured by the detector over the duration of the mission (approximated as a single broad-band burst), whilst the strain amplitude is the strength of the GW emission at each instantaneous moment. For a stellar mass binary, the characteristic strain in the $n^{\rm th}$ harmonic is given by (e.g. [BarackCutler04] Eq. 56; [FlanaganHughes98] Eq. 5.1)

$$h_{c,n}^2 = \frac{1}{(\pi D_L)^2}\left(\frac{2G}{c^3}\frac{\dot{E}_n}{\dot{f}_n}\right), \tag{5.17}$$

where D_L is the luminosity distance to the binary, $\dot{E}_n$ is the power radiated in the $n^{\rm th}$ harmonic and $\dot{f}_n$ is the rate of change of the $n^{\rm th}$ harmonic frequency. The power radiated in the $n^{\rm th}$ harmonic is given by [PetersMathews63] (Eq. 19)

$$\dot{E}_n = \frac{32}{5}\frac{G^4}{c^5}\frac{m_1^2 m_2^2\,(m_1+m_2)}{a^5}g(n,e), \tag{5.18}$$

where m_1 is the primary mass, m_2 is the secondary mass, a is the semi-major axis of the binary and e is the eccentricity. Using Eq. (5.1) and Eq. (5.2), we can recast Eq. (5.18) in a form more applicable for gravitational wave detections that is a function of only the chirp mass, orbital frequency and eccentricity.

$$\dot{E}_n = \frac{32}{5}\frac{G^4}{c^5}\left(m_1^2 m_2^2\,(m_1+m_2)\right) g(n,e)\cdot\left(\frac{(2\pi f_{\rm orb})^2}{G(m_1+m_2)}\right)^{5/3} \tag{5.19}$$

$$\dot{E}_n = \frac{32}{5}\frac{G^{7/3}}{c^5}\frac{m_1^2 m_2^2}{(m_1+m_2)^{2/3}}(2\pi f_{\rm orb})^{10/3} g(n,e) \tag{5.20}$$

$$\dot{E}_n(\mathcal{M}_c, f_{\rm orb}, e) = \frac{32}{5}\frac{G^{7/3}}{c^5}\left(2\pi f_{\rm orb}\mathcal{M}_c\right)^{10/3} g(n,e) \tag{5.21}$$

The last term needed to define the characteristic strain in Eq. (5.17) is the rate of change of the $n^{\rm th}$ harmonic frequency. We can first apply the chain rule and note that

$$\dot{f}_n = \frac{\mathrm{d}f_n}{\mathrm{d}a}\frac{\mathrm{d}a}{\mathrm{d}t}. \tag{5.22}$$

The frequency of the $n^{\rm th}$ harmonic is simply defined as $f_n = n \cdot f_{\rm orb}$ and therefore we can find an expression for $\mathrm{d}f_n/\mathrm{d}a$ by rearranging and differentiating Eq. (5.2)

$$f_n = \frac{n}{2\pi}\sqrt{\frac{G(m_1+m_2)}{a^3}}, \tag{5.23}$$

$$\frac{\mathrm{d}f_n}{\mathrm{d}a} = -\frac{3n}{4\pi}\frac{\sqrt{G(m_1+m_2)}}{a^{5/2}}. \tag{5.24}$$

The rate at which the semi-major axis decreases is [Peters64] (Eq. 5.6)

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -\frac{64}{5}\frac{G^3 m_1 m_2 (m_1+m_2)}{c^5 a^3}F(e). \tag{5.25}$$

Substituting Eq. (5.24) and Eq. (5.25) into Eq. (5.22) gives an expression for $\dot{f}_n$

$$\dot{f}_n = -\frac{3n}{4\pi}\frac{\sqrt{G(m_1+m_2)}}{a^{5/2}} \cdot -\frac{64}{5}\frac{G^3 m_1 m_2 (m_1+m_2)}{c^5 a^3}F(e), \tag{5.26}$$

$$\dot{f}_n = \frac{48n}{5\pi}\frac{G^{7/2}}{c^5}\left(m_1 m_2 (m_1+m_2)^{3/2}\right)\frac{F(e)}{a^{11/2}}, \tag{5.27}$$

which, as above with $\dot{E}_n$, we can recast using Kepler's third law and the definition of the chirp mass

$$\dot{f}_n = \frac{48n}{5\pi}\frac{G^{7/2}}{c^5}\left(m_1 m_2 (m_1+m_2)^{3/2}\right)F(e) \cdot \left(\frac{(2\pi f_{\rm orb})^2}{G(m_1+m_2)}\right)^{11/6}, \tag{5.28}$$

$$= \frac{48n}{5\pi}\frac{G^{5/3}}{c^5}\frac{m_1 m_2}{(m_1+m_2)^{1/3}} \cdot (2\pi f_{\rm orb})^{11/3} \cdot F(e), \tag{5.29}$$

$$\dot{f}_n(\mathcal{M}_c, f_{\rm orb}, e) = \frac{48n}{5\pi}\frac{(G\mathcal{M}_c)^{5/3}}{c^5}(2\pi f_{\rm orb})^{11/3}F(e) \tag{5.30}$$

With definitions of both $\dot{E}_n$ and $\dot{f}_n$, we are now in a position to find an expression for the characteristic strain by plugging Eq. (5.21) and Eq. (5.30) into Eq. (5.33):

$$h_{c,n}^2 = \frac{1}{(\pi D_L)^2}\left(\frac{2G}{c^3}\frac{\frac{32}{5}\frac{G^{7/3}}{c^5}(2\pi f_{\rm orb}\mathcal{M}_c)^{10/3}\, g(n,e)}{\frac{48n}{5\pi}\frac{(G\mathcal{M}_c)^{5/3}}{c^5}(2\pi f_{\rm orb})^{11/3}F(e)}\right) \tag{5.31}$$

$$= \frac{1}{(\pi D_L)^2}\left(\frac{2^{5/3}\pi^{2/3}}{3}\frac{(G\mathcal{M}_c)^{5/3}}{c^3}\frac{1}{f_{\rm orb}^{1/3}}\frac{g(n,e)}{nF(e)}\right) \tag{5.32}$$

This gives a final simplified expression for the characteristic strain amplitude of a GW source.

$$h_{c,n}^2(\mathcal{M}_c, D_L, f_{\rm orb}, e) = \frac{2^{5/3}}{3\pi^{4/3}}\frac{(G\mathcal{M}_c)^{5/3}}{c^3 D_L^2}\frac{1}{f_{\rm orb}^{1/3}}\frac{g(n,e)}{nF(e)} \tag{5.33}$$

Using the conversion between characteristic strain and regular strain, in addition to Eq. (5.30) and Eq. (5.33), we can write an expression for the strain amplitude of gravitational waves in the $n^{\rm th}$ harmonic

$$h_n^2 = \left(\frac{\dot{f}_n}{f_n^2}\right)h_{c,n}^2, \tag{5.34}$$

$$h_n^2 = \left(\frac{48n}{5\pi}\frac{(G\mathcal{M}_c)^{5/3}}{c^5}F(e) \cdot \frac{(2\pi f_{\rm orb})^{11/3}}{n^2 f_{\rm orb}^2}\right)\left(\frac{2^{5/3}}{3\pi^{4/3}}\frac{(G\mathcal{M}_c)^{5/3}}{c^3 D_L^2}\frac{1}{n f_{\rm orb}^{1/3}}\frac{g(n,e)}{F(e)}\right), \tag{5.35}$$

This gives a final simplified expression for the strain amplitude of a GW source.

$$h_n^2(\mathcal{M}_c, f_{\rm orb}, D_L, e) = \frac{2^{28/3}}{5}\frac{(G\mathcal{M}_c)^{10/3}}{c^8 D_L^2}\frac{g(n,e)}{n^2}(\pi f_{\rm orb})^{4/3} \tag{5.36}$$

5.3.2 Amplitude modulation for orbit averaged sources

Because the LISA detectors are not stationary and instead follow an Earth-trailing orbit, the antenna pattern of LISA is not isotropically distributed or stationary. For sources that have a known position, inclination, and polarisation, we can consider the amplitude modulation of the strain due to the average motion of LISA's orbit. We follow the results of [CornishLarson03] to define the amplitude modulation. However, we follow the conventions of more recent papers (e.g. [BabakHewitsonPetiteau21], Eq.67) to write the amplitude modulation as

$$A^2_{\rm mod} = \frac{1}{4}\left(1+\cos^2\iota\right)^2\left\langle F^2_+\right\rangle + \cos^2\iota\left\langle F^2_\times\right\rangle, \tag{5.37}$$

where $\left\langle F^2_+\right\rangle$ and $\left\langle F^2_\times\right\rangle$, the orbit-averaged detector responses, are defined as

$$\left\langle F^2_+\right\rangle = \frac{1}{4}\big(\cos^2 2\psi\left\langle D^2_+\right\rangle - \sin 4\psi\left\langle D_+D_\times\right\rangle + \sin^2 2\psi\left\langle D^2_\times\right\rangle\big), \tag{5.38}$$

$$\left\langle F^2_\times\right\rangle = \frac{1}{4}\big(\cos^2 2\psi\left\langle D^2_\times\right\rangle + \sin 4\psi\left\langle D_+D_\times\right\rangle + \sin^2 2\psi\left\langle D^2_+\right\rangle\big), \tag{5.39}$$

and

$$\left\langle D_+D_\times\right\rangle = \frac{243}{512}\cos\theta\sin 2\phi\left(2\cos^2\phi-1\right)\left(1+\cos^2\theta\right), \tag{5.40}$$

$$\left\langle D^2_\times\right\rangle = \frac{3}{512}\big(120\sin^2\theta+\cos^2\theta+162\sin^2 2\phi\cos^2\theta\big), \tag{5.41}$$

$$\left\langle D^2_+\right\rangle = \frac{3}{2048}\left[487+158\cos^2\theta+7\cos^4\theta-162\sin^2 2\phi\left(1+\cos^2\theta\right)^2\right]. \tag{5.42}$$

In the equations above, the inclination is given by ι, the right ascension and declination are given by ϕ and θ respsectively, and the polarisation is given by ψ.

The orbital motion of LISA smears the source frequency by roughly $10^{-4}\,\mathrm{mHz}$ due to the antenna pattern changing as the detector orbits, the Doppler shift from the motion, and the phase modulation from the $+$ and $\times$ polarisations in the antenna pattern. Generally, the modulation reduces the strain amplitude because the smearing in frequency reduces the amount of signal build up at the true source frequency.

We note that since the orbit averaging is carried out in Fourier space, this requires the frequency to be monochromatic and thus is only implemented in `LEGWORK` for quasi-circular binaries. We also note that since the majority of the calculations in `LEGWORK` are carried out for the full position, polarisation, and inclination averages, we place a prefactor of $5/4$ on the amplitude modulation in the software implementation to undo the factor of $4/5$ which arises from the averaging of Equations (5.38) and (5.39).

5.4 Sensitivity Curves (psd)

5.4.1 LISA

For the LISA sensitivity curve, we follow the equations from [RobsonCornishLiu19], which we list here for your convenience.

The *effective* LISA noise power spectral density is defined as ([RobsonCornishLiu19] Eq. 2)

$$S_{\rm n}(f) = \frac{P_n(f)}{\mathcal{R}(f)} + S_c(f), \tag{5.43}$$

where $P_{\rm n}(f)$ is the power spectral density of the detector noise and $\mathcal{R}(f)$ is the sky and polarisation averaged signal response function of the instrument. Alternatively if we expand out $P_n(f)$, approximate $\mathcal{R}(f)$ and simplify we find ([RobsonCornishLiu19] Eq. 1)

$$S_{\rm n}(f) = \frac{10}{3L^2}\left(P_{\rm OMS}(f) + 2\left(1+\cos^2\left(\frac{f}{f_*}\right)\right)\frac{P_{\rm acc}(f)}{(2\pi f)^4}\right)\left(1+\frac{6}{10}\left(\frac{f}{f_*}\right)^2\right) + S_c(f) \tag{5.44}$$

where $L = 2.5\,\mathrm{Gm}$ is detector arm length, $f^* = 19.09\,\mathrm{mHz}$ is the response frequency,

$$P_{\mathrm{OMS}}(f) = \left(1.5 \times 10^{-11}\mathrm{m}\right)^2 \left(1 + \left(\frac{2\mathrm{mHz}}{f}\right)^4\right) \mathrm{Hz}^{-1} \tag{5.45}$$

is the single-link optical metrology noise ([RobsonCornishLiu19] Eq. 10),

$$P_{\mathrm{acc}}(f) = \left(3 \times 10^{-15}\mathrm{ms}^{-2}\right)^2 \left(1 + \left(\frac{0.4\mathrm{mHz}}{f}\right)^2\right) \left(1 + \left(\frac{f}{8\mathrm{mHz}}\right)^4\right) \mathrm{Hz}^{-1} \tag{5.46}$$

is the single test mass acceleration noise ([RobsonCornishLiu19] Eq. 11) and

$$S_c(f) = Af^{-7/3}e^{-f^{\alpha}+\beta f \sin(\kappa f)}\left[1 + \tanh\left(\gamma\left(f_k - f\right)\right)\right] \mathrm{Hz}^{-1} \tag{5.47}$$

is the galactic confusion noise ([RobsonCornishLiu19] Eq. 14), where the amplitude A is fixed as 9×10^{-45} and the various parameters change over time:

parameter	6 months	1 year	2 years	4 years
α	0.133	0.171	0.165	0.138
β	243	292	299	-221
κ	482	1020	611	521
γ	917	1680	1340	1680
f_k	0.00258	0.00215	0.00173	0.00113

5.4.2 TianQin

We additionally allow other instruments than LISA. We have the TianQin sensitivity curve built in where we use the power spectral density given in [HuangHuKorol+20] Eq. 13.

$$\begin{aligned} S_N(f) = \frac{10}{3L^2}\left[\frac{4S_a}{(2\pi f)^4}\left(1 + \frac{10^{-4}Hz}{f}\right) + S_x\right] \\ \times \left[1 + 0.6\left(\frac{f}{f_*}\right)^2\right] \end{aligned} \tag{5.48}$$

where $L = \sqrt{3} \times 10^5$ km} is the arm length, $S_a = 1 \times 10^{-30}$ m^2 s^{-4} Hz^{-1} is the acceleration noise, $S_x = 1 \times 10^{-24}$ m^2 Hz^{-1} is the displacement measurement noise and $f_* = c/2\pi L$ is the transfer frequency. Note that Eq. 5.48 includes an extra factor of $10/3$ compared to Eq.13 of [HuangHuKorol+20]. [HuangHuKorol+20] absorb this factor into the waveform rather than include it in the power spectral density. We include it to match the convention used by [RobsonCornishLiu19] for the LISA sensitivity curve (see the factor of $10/3$ in Eq. 5.44) so that the sensitivity curves can be compared fairly.

5.5 Signal-to-Noise Ratios for a 6-link (3-arm) LISA (`snr`)

Please note that this section draws heavily from [FlanaganHughes98] Section II C. We go through the same derivations here in more detail than in a paper and hopefully help clarify all of the different stages.

5.5.1 Defining the general SNR

In order to calculate the signal to noise ratio for a given source of gravitational waves (GWs) in the LISA detector, we need to consider the following parameters:

- position of the source on the sky: (θ, ϕ)
- direction from the source to the detector: (ι, β)
- orientation of the source, which fixes the polarisation of the GW: ψ
- the distance from the source to the detector: D_L

Then, assuming a matched filter analysis of the GW signal $s(t) + n(t)$ (where $s(t)$ is the signal and $n(t)$ is the noise), which relies on knowing the shape of the signal, the signal to noise ratio, ρ, is given in the frequency domain as

$$\rho^2(D_L, \theta, \phi, \psi, \iota, \beta) = \frac{\langle s(t)^\star s(t)\rangle}{\langle n(t)^\star n(t)\rangle} = 2\int_{-\infty}^{+\infty} \frac{|\tilde{s}(f)|^2}{P_{\rm n}(f)} df = 4\int_0^{\infty} \frac{|\tilde{s}(f)|^2}{P_{\rm n}(f)} df, \tag{5.49}$$

where $\tilde{s}(f)$ is the Fourier transform of the signal, $s(t)$, and $P_{\rm n}(f)$ is the one sided power spectral density of the noise defined as as $\langle n(t)^\star n(t)\rangle = \int_0^\infty \frac{1}{2} P_{\rm n}(f) df$ (c.f. [RobsonCornishLiu19] Eq. 2). Here, $\tilde{s}(f)$ is implicitly also dependent on $D_L, \theta, \phi, \psi, \iota$, and β as

$$|\tilde{s}(f)|^2 = |F_+(\theta, \phi, \psi)\tilde{h}_+(t, D_L, \iota, \beta) + F_\times(\theta, \phi, \psi)\tilde{h}_\times(t, D_L, \iota, \beta)|^2, \tag{5.50}$$

where $F_{+,\times}$ are the 'plus' and 'cross' antenna patterns of the LISA detector to the 'plus' and 'cross' strains, $h_{+,\times}$. Note throughout any parameters discussed with the subscript $x_{+,\times}$ refers to both x_+ and $x_\times$.

In LISA's case, when averaged over all angles and polarisations, the antenna patterns are orthogonal thus $\langle F_+ F_\times\rangle = 0$. This means we can rewrite Eq. 5.50 as

$$|\tilde{s}(f)|^2 = |F_+(\theta, \phi, \psi)\tilde{h}_+(t, D_L, \iota, \beta)|^2 + |F_\times(\theta, \phi, \psi)\tilde{h}_\times(t, D_L, \iota, \beta)|^2, \tag{5.51}$$

which can then be applied to Eq. (5.49) as

$$\rho^2(D_L, \theta, \phi, \psi, \iota, \beta) = 4\int_0^{\infty} \frac{|F_+\tilde{h}_+|^2 + |F_\times\tilde{h}_\times|^2}{P_{\rm n}(f)} df. \tag{5.52}$$

5.5.2 Average over position and polarisation

Now, we can consider averaging over different quantities. In particular, we can average over the sky position and polarisation as

$$\langle\rho\rangle^2_{\theta,\phi,\psi} = 4\int_0^\infty df \int \frac{d\Omega_{\theta,\phi}}{4\pi} \int \frac{d\psi}{\pi} \frac{|F_+(\theta, \phi, \psi)\,\tilde{h}_+(\iota, \beta)|^2 + |F_\times(\theta, \phi, \psi)\,\tilde{h}_\times(\iota, \beta)|^2}{P_{\rm n}(f)}. \tag{5.53}$$

From [RobsonCornishLiu19], we can write the position and polarisation average of the signal response function of the instrument, $\mathcal{R}$, as

$$\mathcal{R} = \langle F_+ F_+^\star\rangle = \langle F_\times F_\times^\star\rangle, \text{ where } \langle \mathrm{F}_{+,\times}\mathrm{F}^\star_{+,\times}\rangle = \int \frac{\mathrm{d}\Omega_{\theta,\phi}}{4\pi} \int \frac{\mathrm{d}\psi}{\pi} |\mathrm{F}_{+,\times}|^2. \tag{5.54}$$

Then combining Eq. (5.53) and Eq. (5.54), we then find

$$\langle\rho\rangle^2_{\theta,\phi,\psi} = 4\int_0^\infty df \mathcal{R}(f) \left(\frac{|\tilde{h}_+|^2 + |\tilde{h}_\times|^2}{P_{\rm n}(f)}\right) \tag{5.55}$$

Note that this is written in [FlanaganHughes98] for the LIGO response function which is $\mathcal{R} = \langle F_{+,\times}\rangle^2 = 1/5$.

5.5.3 Average over orientation

Now, we can average over the orientation of the source: (ι, β), noting that the averaging is independent of the distance D_L. Then we can rewrite $|\tilde{h}_+|^2 + |\tilde{h}_\times|^2$ in terms of two functions $|\tilde{H}_+|^2$ and $|\tilde{H}_\times|^2$, where $\tilde{h}_{+,\times} = \tilde{H}_{+,\times}/D_L$. Then, averaging over the source direction gives

$$\langle \rho \rangle^2_{(\theta,\phi,\psi),(\iota,\beta)} = \frac{4}{D_L^2} \int_0^\infty df \mathcal{R}(f) \int \frac{d\Omega_{\iota,\beta}}{4\pi} \frac{|\tilde{H}_+|^2 + |\tilde{H}_\times|^2}{P_{\rm n}(f)}, \tag{5.56}$$

where we would like to express $\tilde{H}_{+,\times}(f)^2$ in terms of the energy spectrum of the GW. To do this, we note that the local energy flux of GWs at the detector is given by (e.g. [PressThorne72] Eq. 6)

$$\frac{dE}{dAdt} = \frac{1}{16\pi} \frac{c^3}{G} \overline{\left[\left(\frac{dh_+}{dt} \right)^2 + \left(\frac{dh_\times}{dt} \right)^2 \right]}, \tag{5.57}$$

where the bar indicates an average over several cycles of the wave which is appropriate for LISA sources. We can transform Eq. (5.57) using Parseval's theorem, where we can write

$$\int_{-\infty}^{+\infty} dt \int dA \frac{dE}{dAdt} = \int_{-\infty}^{+\infty} dt \int dA \frac{1}{16\pi} \frac{c^3}{G} \overline{\left[\left(\frac{dh_+}{dt} \right)^2 + \left(\frac{dh_\times}{dt} \right)^2 \right]} \tag{5.58}$$

$$= \int_{-\infty}^{+\infty} df \int dA \frac{1}{16\pi} \frac{c^3}{G} \left[\left((-2\pi i f) |\tilde{h}_+| \right)^2 + \left((-2\pi i f) |\tilde{h}_\times| \right)^2 \right] \tag{5.59}$$

$$= \int_{-\infty}^{+\infty} df \int dA \frac{1}{16\pi} \frac{c^3}{G} (2\pi f)^2 \left(|\tilde{h}_+|^2 + |\tilde{h}_\times|^2 \right) \tag{5.60}$$

$$= \int_{-\infty}^{+\infty} df \int dA \frac{c^3}{G} \frac{\pi f^2}{4} \left(|\tilde{h}_+|^2 + |\tilde{h}_\times|^2 \right) \tag{5.61}$$

$$= \int_{0}^{\infty} df \int dA \frac{c^3}{G} \frac{\pi f^2}{2} \left(|\tilde{h}_+|^2 + |\tilde{h}_\times|^2 \right). \tag{5.62}$$

Note that we perform a Fourier transform of the square of the time derivatives in the second line. Now, since $A = D_L^2 \Omega$ and $|\tilde{h}_{+,\times}|^2 = |\tilde{H}_{+,\times}|^2/D_L^2$, we know

$$|\tilde{h}_{+,\times}|^2 dA = |\tilde{H}_{+,\times}|^2 d\Omega_{\iota,\beta}, \tag{5.63}$$

then we can write Eq. (5.62) in terms of $|H_{+,\times}|^2$ as

$$\int_{-\infty}^{+\infty} dt \int dA \frac{dE}{dAdt} = \int_{0}^{\infty} df \int dA \frac{c^3}{G} \frac{\pi f^2}{2} \left(|\tilde{h}_+|^2 + |\tilde{h}_\times|^2 \right) \tag{5.64}$$

$$= \int_{0}^{\infty} df \frac{\pi f^2}{2} \frac{c^3}{G} \int d\Omega \left(|\tilde{H}_+|^2 + |\tilde{H}_\times|^2 \right). \tag{5.65}$$

We can note that by using Eq. (5.63) and performing a Fourier transform we also have that

$$\int_{-\infty}^{+\infty} dt \int dA \frac{dE}{dAdt} = \int_{0}^{\infty} df \int d\Omega \frac{dE}{d\Omega df}. \tag{5.66}$$

From inspection of Eq. (5.65) and Eq. (5.66), we can write the spectral energy flux as

$$\int d\Omega \frac{dE}{d\Omega df} = \frac{\pi f^2}{2} \frac{c^3}{G} \int d\Omega \left(|\tilde{H}_+|^2 + |\tilde{H}_\times|^2 \right). \tag{5.67}$$

5.5.4 Fully averaged SNR equation

We are now in a position to write an expression for the fully averaged SNR. Let's take Eq. (5.67) and apply it to Eq. (5.56)

$$\langle\rho\rangle^2_{(\theta,\phi,\psi),(\iota,\beta)} = \frac{4}{D_L^2}\frac{G}{c^3}\int_0^\infty df \frac{1}{P_{\rm n}(f)/\mathcal{R}(f)} \int \frac{d\Omega}{4\pi}\frac{dE}{d\Omega df}\frac{2}{\pi f^2}. \tag{5.68}$$

This simplifies nicely to

$$\langle\rho\rangle^2 = \frac{2G}{(\pi D_L)^2 c^3}\int_0^\infty df \frac{dE}{df}\frac{1}{f^2 P_{\rm n}(f)/\mathcal{R}(f)}. \tag{5.69}$$

Finally, noting that $dE/df = dE/dt \times dt/df = \dot{E}/\dot{f}$, we can use the definition of the characteristic strain from Eq. (5.17) to finish up our position, direction, and orientation/polarisation averaged SNR as

$$\langle\rho\rangle^2_{(\theta,\phi,\psi),(\iota,\beta)} = \int_0^\infty df \frac{h_c^2}{f^2 P_{\rm n}(f)/\mathcal{R}(f)} = \int_0^\infty df \frac{h_c^2}{f^2 S_{\rm n}(f)}, \tag{5.70}$$

where we have used that the effective power spectral density of the noise is defined as $S_{\rm n}(f) = P_{\rm n}(f)/\mathcal{R}(f)$. Note that this defintion is the sensitivity for a 6-link (3-arm) LISA detector in the long wavelength limit, which is appropriate for stellar mass binary LISA sources.

It is also important to note that this is only the SNR for a circular binary for which we need only consider the $n = 2$ harmonic. In the general case, a binary could be eccentric and requires a sum over *all* harmonics. Thus we can generalise Eq. (5.70) to eccentric binaries with

$$\langle\rho\rangle^2_{(\theta,\phi,\psi),(\iota,\beta)} = \sum_{n=1}^\infty \langle\rho_n\rangle^2_{(\theta,\phi,\psi),(\iota,\beta)} = \sum_{n=1}^\infty \int_0^\infty df_n \frac{h_{c,n}^2}{f_n^2 S_{\rm n}(f_n)}, \tag{5.71}$$

where $f_n = n \cdot f_{\rm orb}$ (with n being the harmonic and $f_{\rm orb}$ the orbital frequency), $h_{c,n}$ is defined in Eq. (5.33) and $S_{\rm n}$ in Eq. (5.44).

5.5.5 Different SNR approximations

Although Eq. (5.71) can be used for every binary, it can be useful to consider different cases in which we can avoid unnecessary sums and integrals. There are four possible cases for binaries in which we can use increasingly simple expressions for the signal-to-noise ratio. Binaries can be circular and stationary in frequency space.

- Circular binaries emit only in the $n = 2$ harmonic and so the sum over harmonics can be removed
- Stationary binaries have $f_{n,i} \approx f_{n,f}$ and so the small interval allows one to approximate the integral

We refer to non-stationary binaries as 'evolving' here though many also use 'chirping'.

For an evolving and eccentric binary, no approximation can be made and the SNR is found using Eq. (5.71).

For an evolving and circular binary, the sum can be removed and so the SNR found as

$$\langle\rho\rangle^2_{\rm c,e} = \frac{1}{4}\int_{f_{2,i}}^{f_{2,f}} \frac{h_{c,2}^2}{f_2^2 S_{\rm n}(f_2)}{\rm d}f \tag{5.72}$$

For a stationary and eccentric binary we can approximate the integral.

$$\langle\rho\rangle^2_{\rm e,s} = \frac{1}{4}\sum_{n=1}^{\infty}\lim_{\Delta f\to 0}\int_{f_n}^{f_n+\Delta f_n}\frac{h^2_{c,n}}{f_n^2 S_{\rm n}(f_n)}{\rm d}f_n, \tag{5.73}$$

$$= \frac{1}{4}\sum_{n=1}^{\infty}\frac{\Delta f_n\cdot h^2_{c,n}}{f_n^2 S_{\rm n}(f_n)}, \tag{5.74}$$

$$= \frac{1}{4}\sum_{n=1}^{\infty}\frac{\dot{f}_n\Delta T\cdot h^2_{c,n}}{f_n^2 S_{\rm n}(f_n)}, \tag{5.75}$$

$$= \frac{1}{4}\sum_{n=1}^{\infty}\left(\frac{\dot{f}_n}{f_n^2}h^2_{c,n}\right)\frac{T_{\rm obs}}{S_{\rm n}(f_n)}, \tag{5.76}$$

$$\langle\rho\rangle^2_{\rm e,s} = \frac{1}{4}\sum_{n=1}^{\infty}\frac{h_n^2 T_{\rm obs}}{S_{\rm n}(f_n)}, \tag{5.77}$$

where we have applied Eq. (5.16) to convert between strains and labelled $\Delta t = T_{\rm obs}$. Finally, for a stationary and circular binary the signal-to-noise ratio is simply

$$\langle\rho\rangle^2_{\rm c,s} = \frac{1}{4}\frac{h_2^2 T_{\rm obs}}{S_{\rm n}(f_2)} \tag{5.78}$$

That's all for the derivations of the equations! If you are confused by something or think there is a mistake please feel free to open an issue on GitHub.

Continue reading for the function table and references!

5.6 Equation to Function Table

The following table gives a list of the functions in the modules and which equation numbers in this document that they come from.

Quantity	Equation	Function
$\mathcal{M}_c$	5.1	*legwork.utils.chirp_mass()*
a	5.2	*legwork.utils.get_a_from_forb()*
$f_{\rm orb}$	5.2	*legwork.utils.get_forb_from_a()*
$g(n,e)$	5.3	*legwork.utils.peters_g()*
$F(e)$	5.4	*legwork.utils.peters_f()*
β	5.6	*legwork.utils.beta()*
$a_{\rm circ}(t), f_{\rm orb,circ}(t)$	5.7	*legwork.evol.evol_circ()*
$t_{\rm merge,circ}$	5.8	*legwork.evol.get_t_merge_circ()*
$e(t), a(t), f_{\rm orb}(t)$	5.11	*legwork.evol.evol_ecc()*
$t_{\rm merge}$	5.12	*legwork.evol.get_t_merge_ecc()*
$h_{c,n}$	5.33	*legwork.strain.h_c_n()*
h_n	5.36	*legwork.strain.h_0_n()*
$S_{\rm n}(f)$	5.44	*legwork.psd.power_spectral_density()*
ρ	5.71	*legwork.source.Source.get_snr()*
$\rho_{\rm e,e}$	5.71	*legwork.snr.snr_ecc_evolving()*
$\rho_{\rm c,e}$	5.72	*legwork.snr.snr_circ_evolving()*
$\rho_{\rm e,s}$	5.77	*legwork.snr.snr_ecc_stationary()*
$\rho_{\rm c,s}$	5.78	*legwork.snr.snr_circ_stationary()*

5.7 References

CHAPTER

SIX

SCOPE AND LIMITATIONS

LEGWORK is designed to provide quick estimates of the signal-to-noise ratio of stellar-mass sources of mHz gravitational waves. These calculations can be especially helpful when determining which sources in a large population containing millions of binaries will be potentially detectable by LISA. If you are looking for estimates of non-stellar-origin sources, check out the Gravitational Wave Universe Toolbox or gwplotter. If you are looking for more advanced LISA simulator tools, check out lisacattools or ldasoft.

The calculations done by LEGWORK apply the lowest-order post-Newtonian description of gravitational wave emission and carefully follow the derivation of Flanagan and Hughes 1998a. This means that any higher order effects like spin-orbit coupling or radiation reaction are not accounted for and thus if a source is expected to depend on these higher order effects the SNRs provided by LEGWORK may not be representative of the true SNR. LEGWORK's SNRs are appropriate at mHz frequencies for sources with masses less than a few tens of solar masses.

LEGWORK provides SNRs for two assumption cases: one where the detector orbit, sky position of the source, and inclination of the source are all averaged and one where the detector orbit is averaged while the sky position and inclination of of the source are provided by the user following Cornish and Larson 2003. The calculation which takes the sky position and inclination of the source into account also accounts for frequency spreading due to doppler modulation from the detector's orbit. This means that this calculation is only valid for circular sources and will *always* return SNRs that are lower than the fully averaged SNR calculation because of the effects of the frequency spreading.

BIBLIOGRAPHY

[BabakHewitsonPetiteau21] Stanislav Babak, Martin Hewitson, and Antoine Petiteau. LISA Sensitivity and SNR Calculations. *arXiv e-prints*, pages arXiv:2108.01167, August 2021. arXiv:2108.01167.

[BarackCutler04] Leor Barack and Curt Cutler. LISA capture sources: Approximate waveforms, signal-to-noise ratios, and parameter estimation accuracy. *Physical Review D*, 69(8):082005, April 2004. arXiv:gr-qc/0310125, doi:10.1103/PhysRevD.69.082005.

[CornishLarson03] Neil J. Cornish and Shane L. Larson. LISA data analysis: Source identification and subtraction. *prd* , 67(10):103001, May 2003. arXiv:astro-ph/0301548, doi:10.1103/PhysRevD.67.103001.

[FinnThorne00] Lee Samuel Finn and Kip S. Thorne. Gravitational waves from a compact star in a circular, inspiral orbit, in the equatorial plane of a massive, spinning black hole, as observed by LISA. *Physical Review D*, 62(12):124021, December 2000. arXiv:gr-qc/0007074, doi:10.1103/PhysRevD.62.124021.

[FlanaganHughes98] Éanna É. Flanagan and Scott A. Hughes. Measuring gravitational waves from binary black hole coalescences. I. Signal to noise for inspiral, merger, and ringdown. *Physical Review D*, 57(8):4535–4565, April 1998. arXiv:gr-qc/9701039, doi:10.1103/PhysRevD.57.4535.

[HuangHuKorol+20] Shun-Jia Huang, Yi-Ming Hu, Valeriya Korol, Peng-Cheng Li, Zheng-Cheng Liang, Yang Lu, Hai-Tian Wang, Shenghua Yu, and Jianwei Mei. Science with the TianQin Observatory: Preliminary results on Galactic double white dwarf binaries. *prd* , 102(6):063021, September 2020. arXiv:2005.07889, doi:10.1103/PhysRevD.102.063021.

[MooreColeBerry15] C. J. Moore, R. H. Cole, and C. P. L. Berry. Gravitational-wave sensitivity curves. *Classical and Quantum Gravity*, 32(1):015014, January 2015. arXiv:1408.0740, doi:10.1088/0264-9381/32/1/015014.

[Peters64] P. C. Peters. Gravitational Radiation and the Motion of Two Point Masses. *Physical Review*, 136(4B):1224–1232, November 1964. doi:10.1103/PhysRev.136.B1224.

[PetersMathews63] P. C. Peters and J. Mathews. Gravitational Radiation from Point Masses in a Keplerian Orbit. *Physical Review*, 131(1):435–440, July 1963. doi:10.1103/PhysRev.131.435.

[PressThorne72] William H. Press and Kip S. Thorne. Gravitational-Wave Astronomy. *araa* , 10:335, January 1972. doi:10.1146/annurev.aa.10.090172.002003.

[RobsonCornishLiu19] Travis Robson, Neil J. Cornish, and Chang Liu. The construction and use of LISA sensitivity curves. *Classical and Quantum Gravity*, 36(10):105011, May 2019. arXiv:1803.01944, doi:10.1088/1361-6382/ab1101.

PYTHON MODULE INDEX

INDEX

M

P

S

T

U

V