harmonica

Release 0.1.1

David Grant

Apr 30, 2024

CONTENTS

1 Installation **3**

2 Transmission strings **5**

3 Quick start **7**

4 Tutorials **9**
- 4.1 Generate a light curve . . . 9
- 4.2 Maximum likelihood estimation of a transmission string . . . 11
- 4.3 Basic inference of a transmission string . . . 17
- 4.4 Gradient-based inference of a transmission string . . . 25
- 4.5 Assessing model evidence . . . 34
- 4.6 Calculate Fourier parameters from shape data . . . 39
- 4.7 Time-dependent shapes . . . 46

5 API **49**
- 5.1 Primary Interface . . . 49
- 5.2 Subpackages . . . 49

6 Citation **51**

7 Acknowledgements **53**

Python Module Index **55**

Index **57**

Harmonica is a python package for computing transit light curves of irregularly-shaped occultors. The primary utility of this code is for mapping exoplanet atmospheres around their terminators, however the technique can be applied to any transit light curve to derive the shape of the transiting body. Be sure to have a quick read through of how the shapes, referred to as *transmission strings*, are defined before you get going.

CHAPTER

ONE

INSTALLATION

Install harmonica with pip:

```
pip install planet-harmonica
```

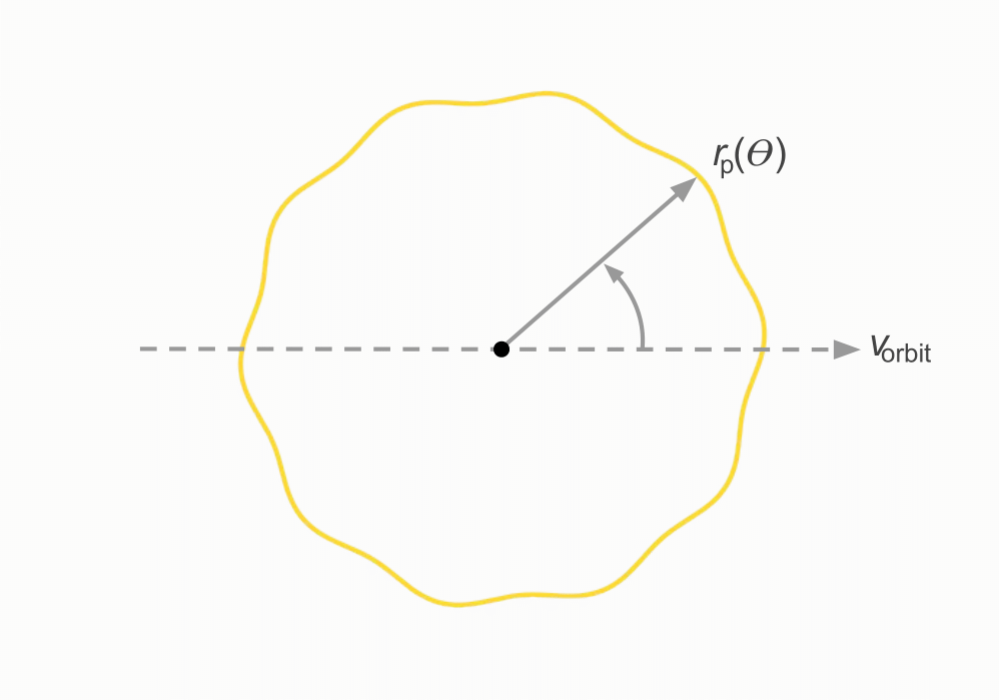
We provide pre-built wheels for python 3.6 – 3.10 on Linux and Mac (intel), as well as wheels for python 3.8 – 3.10 on Mac (apple silicon).

CHAPTER

TWO

TRANSMISSION STRINGS

Harmonica generates light curves for transiting objects, such as exoplanets, where the sky-projected shape of these objects may deviate from circles. A given shape is defined by a single-valued function of angle around the objects terminator, called a transmission string. In the diagram below we illustrate a transmission string, $r_{\rm p}(\theta)$, where $r_{\rm p}$ is the effective radius of the object and θ is the angle around the terminator from the object's orbital velocity vector.

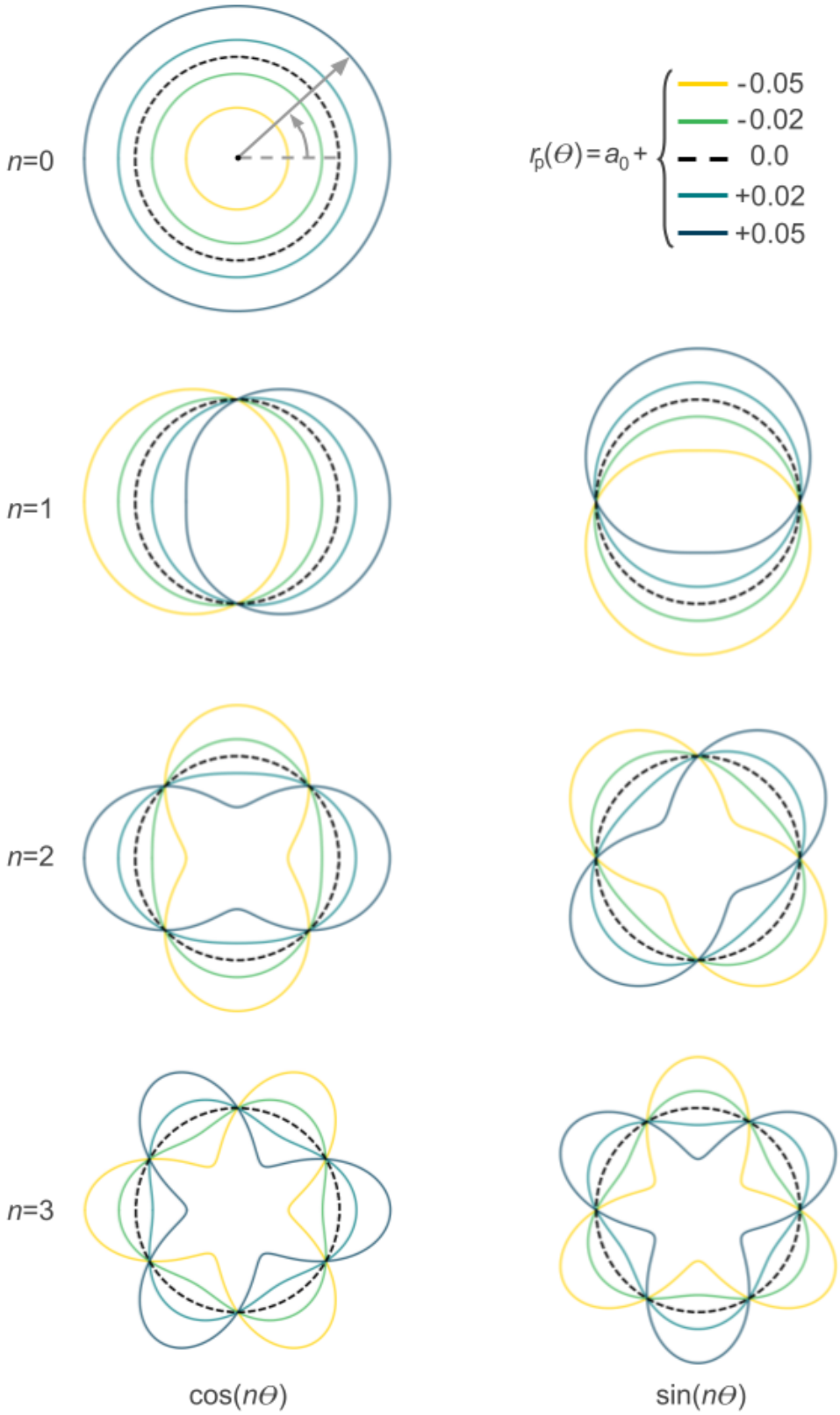


In Harmonica, a transmission string is parametrised in terms of a Fourier series. Mathematically we can write

$$r_{\rm p}(\theta) = \sum_{n=0}^{N_c} a_n \cos{(n\theta)} + \sum_{n=1}^{N_c} b_n \sin{(n\theta)},$$

where a_n and b_n are each nth harmonic's amplitude. The total number of terms is equal to $2N_c + 1$. Below we show the shape contributions from the first 7 terms.

The above shapes demonstrate the basis (shown up to $n = 3$) for generating various shapes. A transmission string may then be constructed from a linear combination of each shape contribution. To construct shapes of increasing complexity, more and more harmonics must be included. For further reference see Grant and Wakeford 2023.


n=0
n=1
n=2
n=3
$r_p(\theta) = a_0 +$
-0.05
-0.02
0.0
+0.02
+0.05
cos(nθ)
sin(nθ)

CHAPTER

THREE

QUICK START

After installing the code (see *installation*) you are ready to generate light curves. Below we demonstrate a minimal example.

First, import the HarmonicaTransit class and specify the times at which you want to evaluate the light curve model.

```
import numpy as np
from harmonica import HarmonicaTransit


ht = HarmonicaTransit(times=np.linspace(-0.2, 0.2, 500))
```

Next, set the orbit, limb-darkening, and transmission string parameters.

```
ht.set_orbit(t0=0., period=4., a=7., inc=88. * np.pi / 180.)
ht.set_stellar_limb_darkening(u=np.array([0.074, 0.193]), limb_dark_law="quadratic")
ht.set_planet_transmission_string(r=np.array([0.1, -0.003, 0.]))
```

Finally, generate the transit light curve.

```
light_curve = ht.get_transit_light_curve()
```

CHAPTER

FOUR

TUTORIALS

4.1 Generate a light curve

In this tutorial we take a look at how to generate transit light curves for a specified transmission string. Let us start by importing the required packages and instantiating the HarmonicaTransit class. Here you must specify the times at which you want to evaluate the light curve model.

[1]:
```
import numpy as np
from harmonica import HarmonicaTransit


times = np.linspace(-0.2, 0.2, 500)  # [days]

ht = HarmonicaTransit(times)
```

Next, you must set the system parameters. First, set the orbital parameters.

[2]:
```
t0=0.                   # [days]
period=4.               # [days]
a=7.                    # [stellar radii]
inc=86. * np.pi / 180.  # [radians]
ecc=0.                  # []
omega=0.                # [radians]

ht.set_orbit(t0, period, a, inc, ecc, omega)
```

Second, set the limb-darkening law and its parameters.

[3]:
```
u = np.array([0.074, 0.193])

ht.set_stellar_limb_darkening(u, limb_dark_law="quadratic")
```

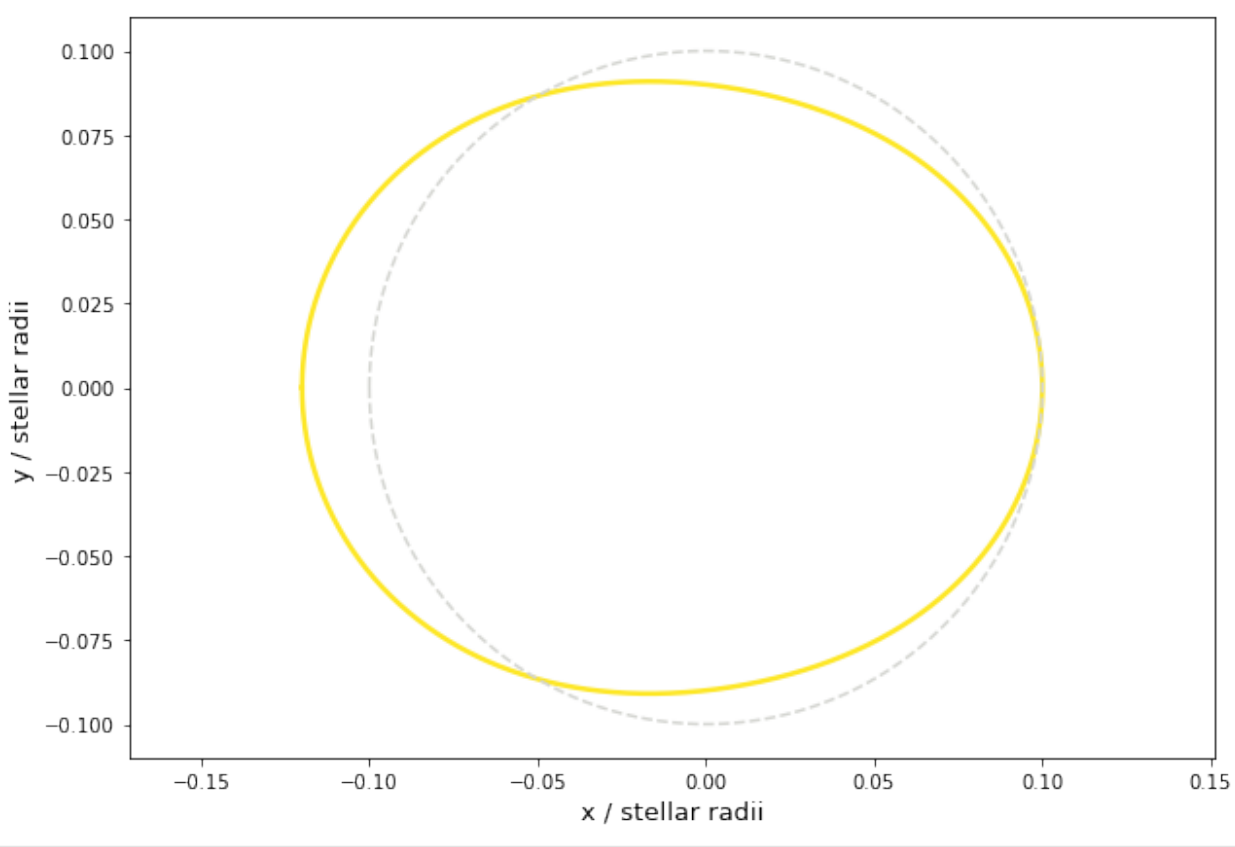
And third, set the planet's transmission string. The transmission string parameters should be provided in an array with order $r = [a_0, a_1, b_1, a_2, b_2, ..]$, where a_n and b_n are the Fourier parameters as described in the intro to transmission strings. In this case we specifiy a 5-parameter transmission string.

[4]:
```
r = np.array([0.1, -0.01, 0., 0.01, 0.])  # [stellar radii]

ht.set_planet_transmission_string(r)
```

You can visualise your transmission string by calling:

[5]:
```
theta = np.linspace(-np.pi, np.pi, 1000)
transmission_string = ht.get_planet_transmission_string(theta)
```

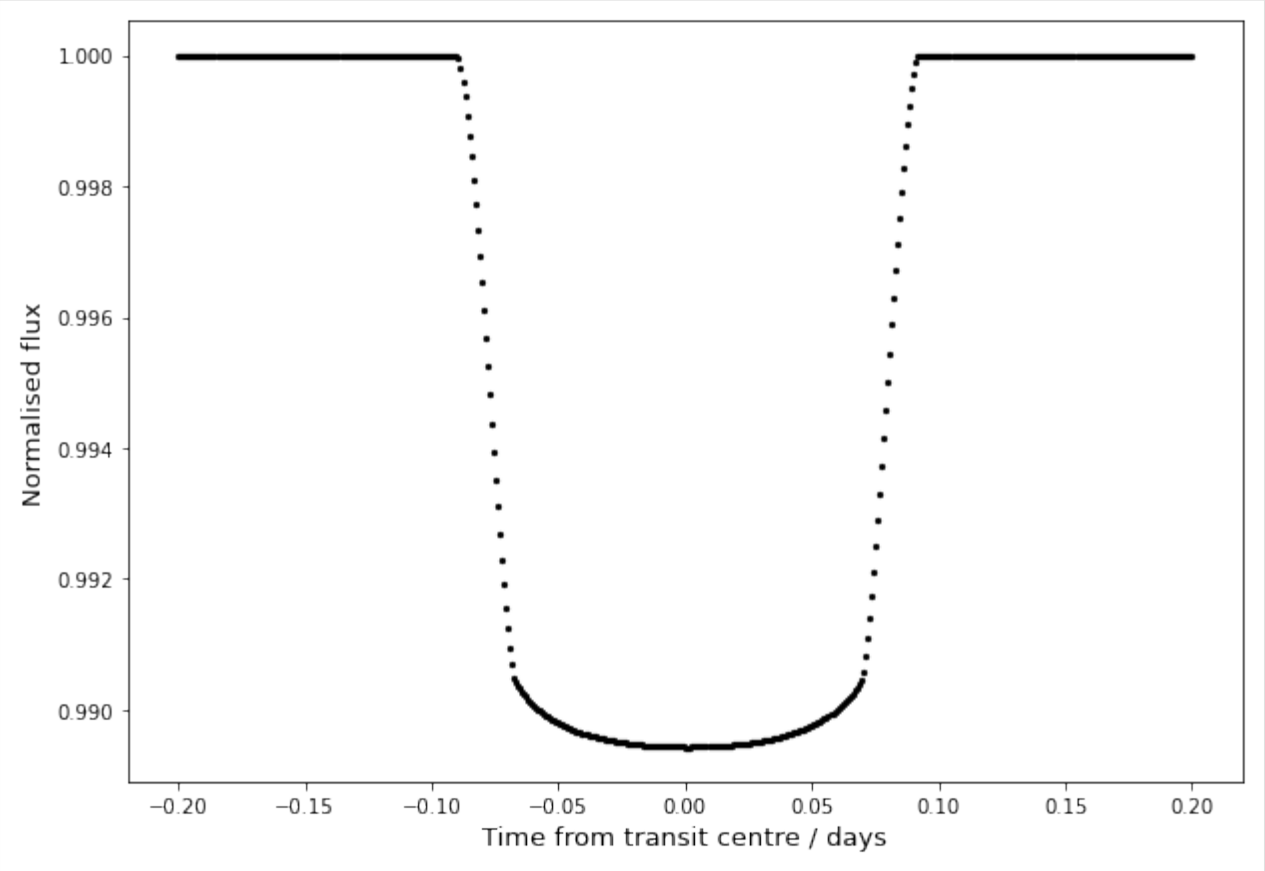
[6]:
```
import matplotlib.cm as cm
import matplotlib.pyplot as plt


plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.plot(transmission_string * np.cos(theta), transmission_string * np.sin(theta),
         c=cm.viridis(1.), lw=2.5, label="Transmission string")
plt.plot(r[0] * np.cos(theta), r[0] * np.sin(theta),
         c="#d5d6d2", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.show()
```



After the system parameters have been set, the transit light curve can be computed:

[7]:
```
light_curve = ht.get_transit_light_curve()
```

[8]:
```
plt.figure(figsize=(10, 7))
plt.scatter(times, light_curve, c="#000000", s=5)
plt.xlabel("Time from transit centre / days", fontsize=13)
plt.ylabel("Normalised flux", fontsize=13)
plt.show()
```



4.2 Maximum likelihood estimation of a transmission string

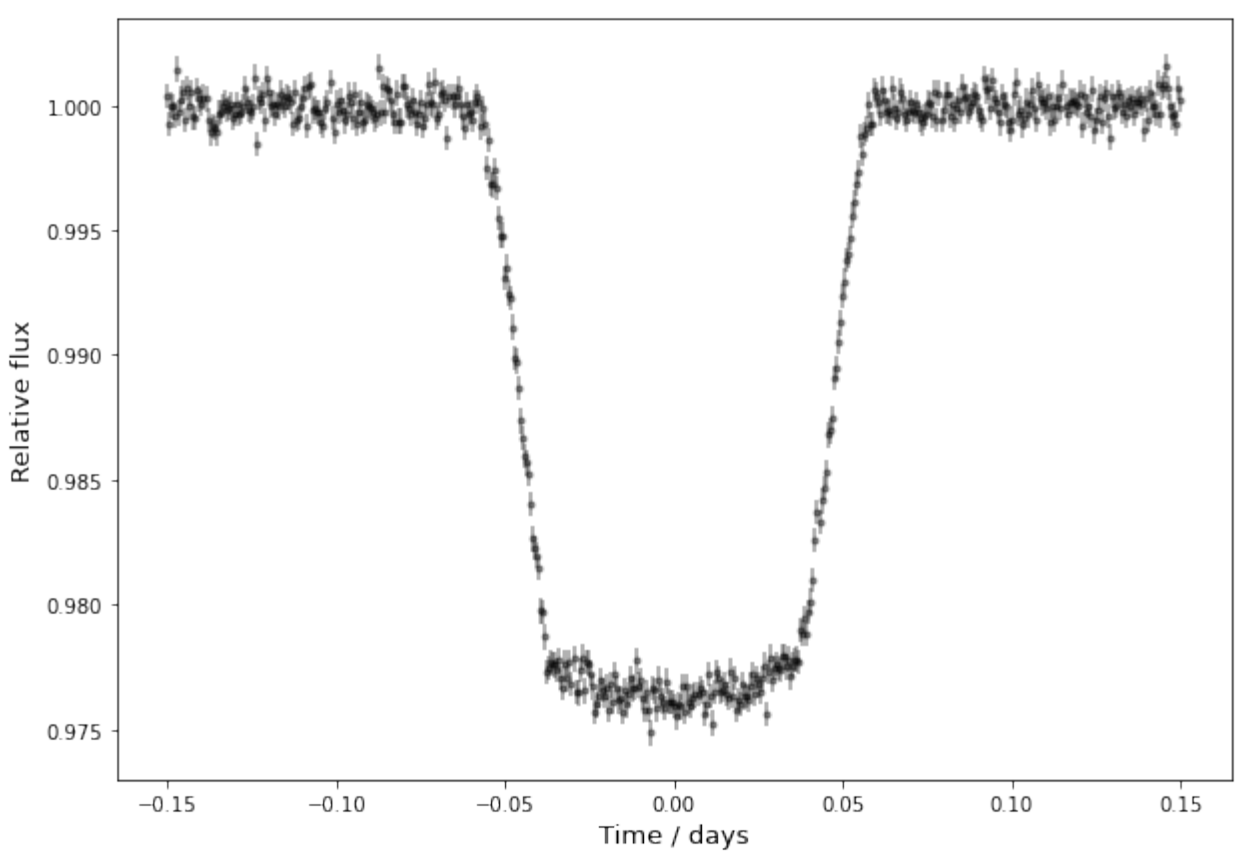
In this tutorial we take a look at how to quickly fit a light curve, and estimate the maximum-likelihood transmission string. Let us start by simulating a transit light curve for a known 7-parameter transmission string.

[1]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from harmonica import HarmonicaTransit


np.random.seed(12)

times = np.linspace(-0.15, 0.15, 500)
r_mean = np.array([0.15])
r_dev = np.random.uniform(-0.1, 0.1, size=6)
injected_r = np.concatenate([r_mean, r_dev * r_mean])

ht = HarmonicaTransit(times)
ht.set_orbit(t0=0., period=4., a=11., inc=87. * np.pi / 180.)
ht.set_stellar_limb_darkening(np.array([0.027, 0.246]), limb_dark_law='quadratic')
ht.set_planet_transmission_string(injected_r)
```

(continues on next page)

(continued from previous page)

```
theta = np.linspace(-np.pi, np.pi, 1000)
injected_transmission_string = ht.get_planet_transmission_string(theta)

flux_sigma = 500.e-6 * np.ones(times.shape[0])
flux_errs = np.random.normal(loc=0., scale=flux_sigma, size=times.shape[0])
observed_fluxes = ht.get_transit_light_curve() + flux_errs

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



This transit light curve corresponds to the following transmission string:

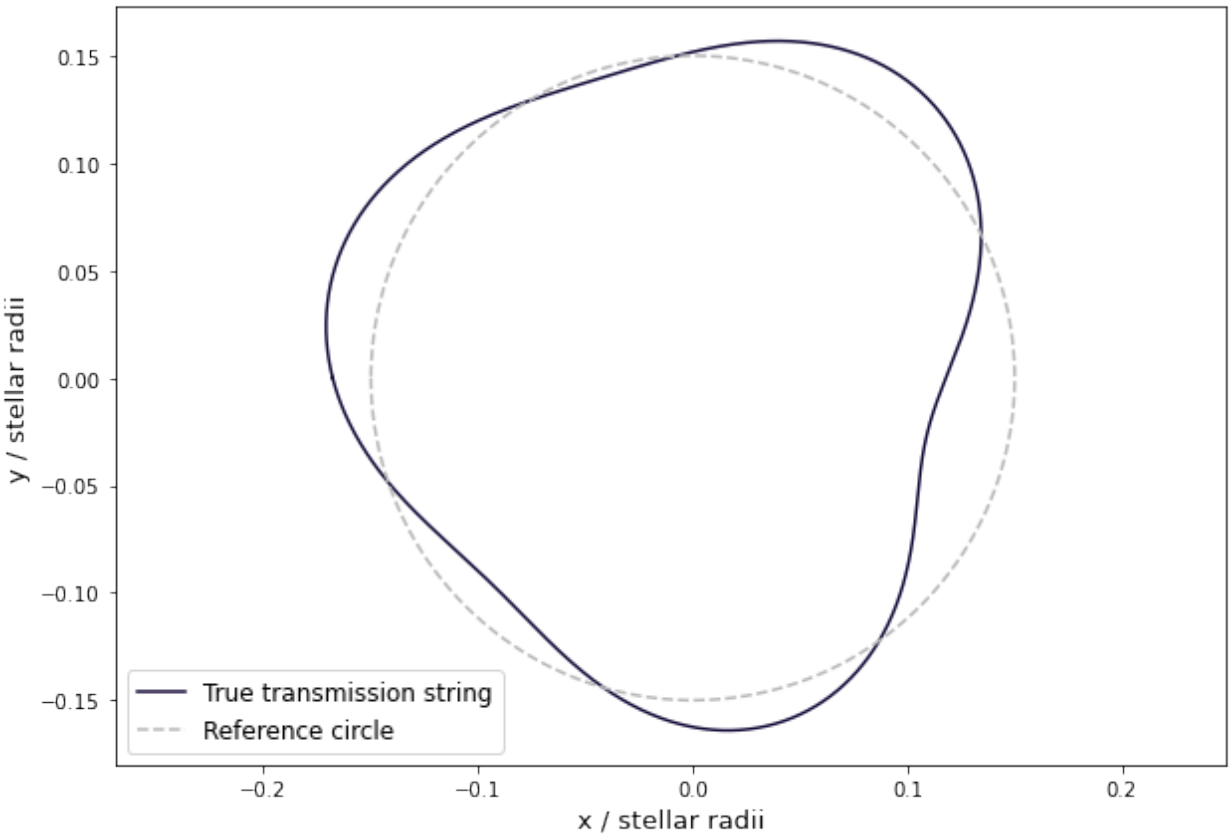
```
[2]: print("r = {0:.3f}{1:+.3f}cos(t){2:+.3f}sin(t)"
           "{3:+.3f}cos(2t){4:+.3f}sin(2t)"
           "{5:+.3f}cos(3t){6:+.3f}sin(3t)".format(*injected_r))

     plt.figure(figsize=(10, 7))
     plt.gca().set_aspect("equal", "datalim")
     plt.plot(injected_transmission_string * np.cos(theta),
              injected_transmission_string * np.sin(theta),
              c=cm.inferno(0.1), label="True transmission string")
```

(continues on next page)

(continued from previous page)

```
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```

```
r = 0.150-0.010cos(t)+0.007sin(t)-0.007cos(2t)+0.001sin(2t)-0.015cos(3t)+0.013sin(3t)
```



Now you can apply Harmonica and estimate the transmission string parameters which maximise the likelihood of the simulated data. To do this we make use of the `scipy.optimize` module.

[3]:
```
from scipy.optimize import curve_fit


def transit_model(_, *params):
    ht.set_planet_transmission_string(np.array(params))
    model = ht.get_transit_light_curve()

    return model


popt, pcov = curve_fit(
    transit_model, times, observed_fluxes, sigma=flux_sigma,
    p0=np.concatenate([r_mean, r_dev * r_mean]),
```

(continues on next page)

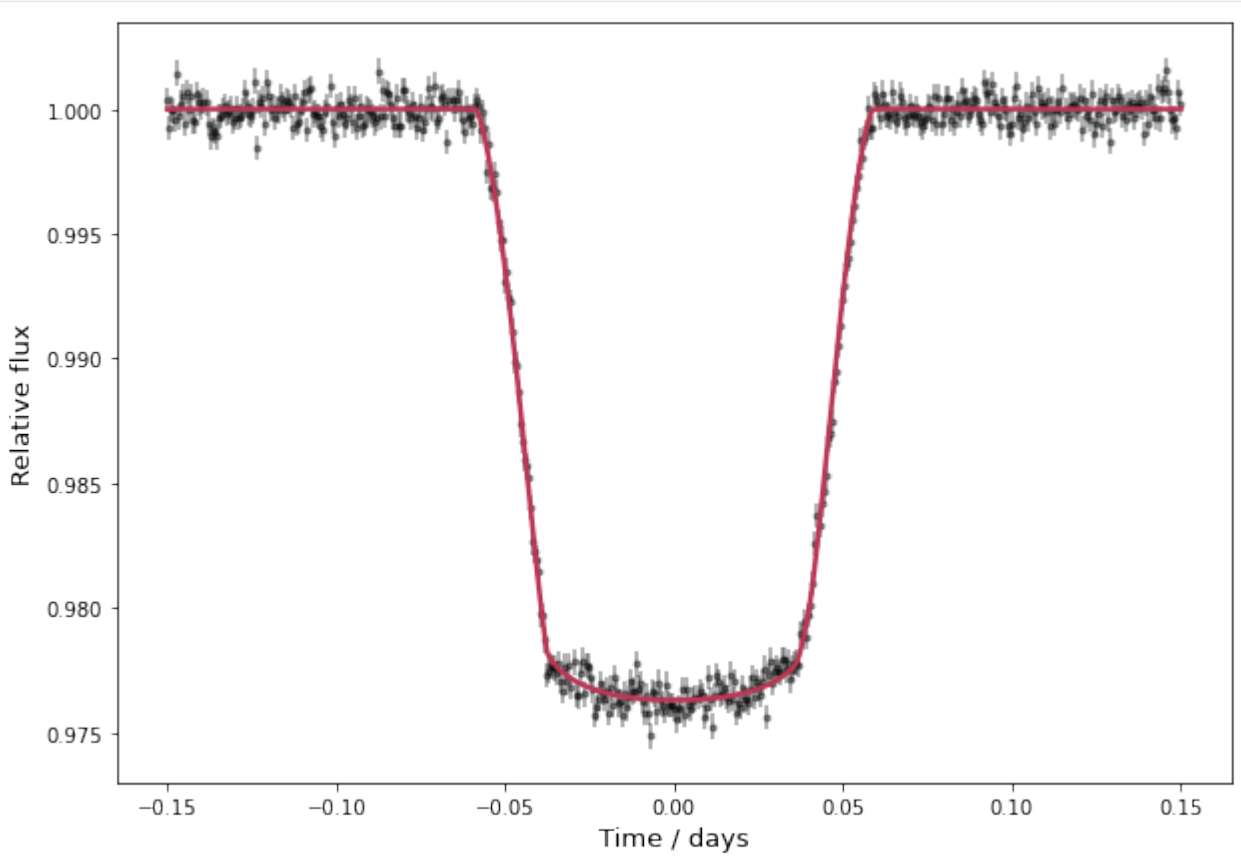
(continued from previous page)

```
    method='lm')

print("r = {0:.3f}{1:+.3f}cos(t){2:+.3f}sin(t)"
      "{3:+.3f}cos(2t){4:+.3f}sin(2t)"
      "{5:+.3f}cos(3t){6:+.3f}sin(3t)".format(*injected_r))
```

```
r = 0.150-0.010cos(t)+0.007sin(t)-0.007cos(2t)+0.001sin(2t)-0.015cos(3t)+0.013sin(3t)
```

This fast method gives you a point estimate of the best-fit parameters. Take a look at the fitted transit light curve:

[4]:
```
ht.set_planet_transmission_string(popt)

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4, zorder=1)
plt.plot(times, ht.get_transit_light_curve(), c=cm.inferno(0.5), lw=2.5, zorder=2)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



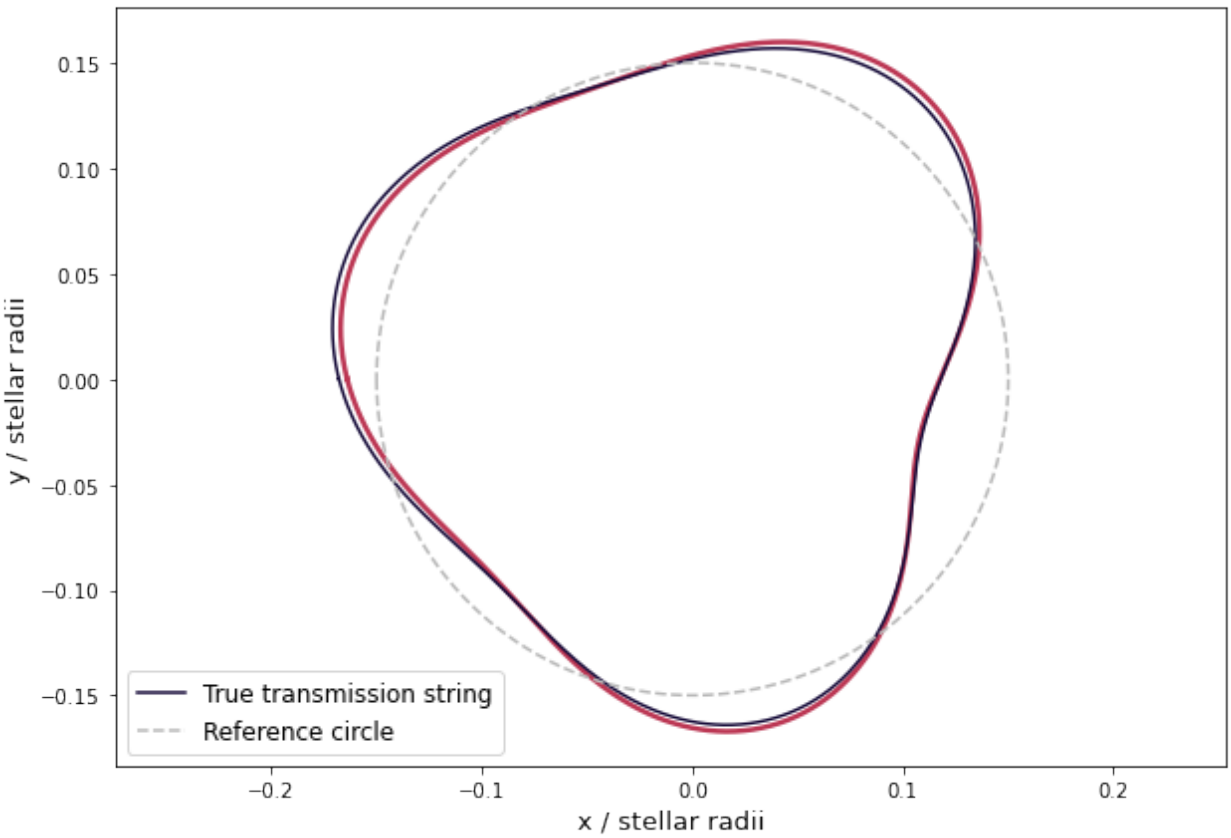
And here is the estimated transmission string:

[5]:
```
plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.plot(ht.get_planet_transmission_string(theta) * np.cos(theta),
         ht.get_planet_transmission_string(theta) * np.sin(theta),
         c=cm.inferno(0.5), lw=2.5)
```

(continues on next page)

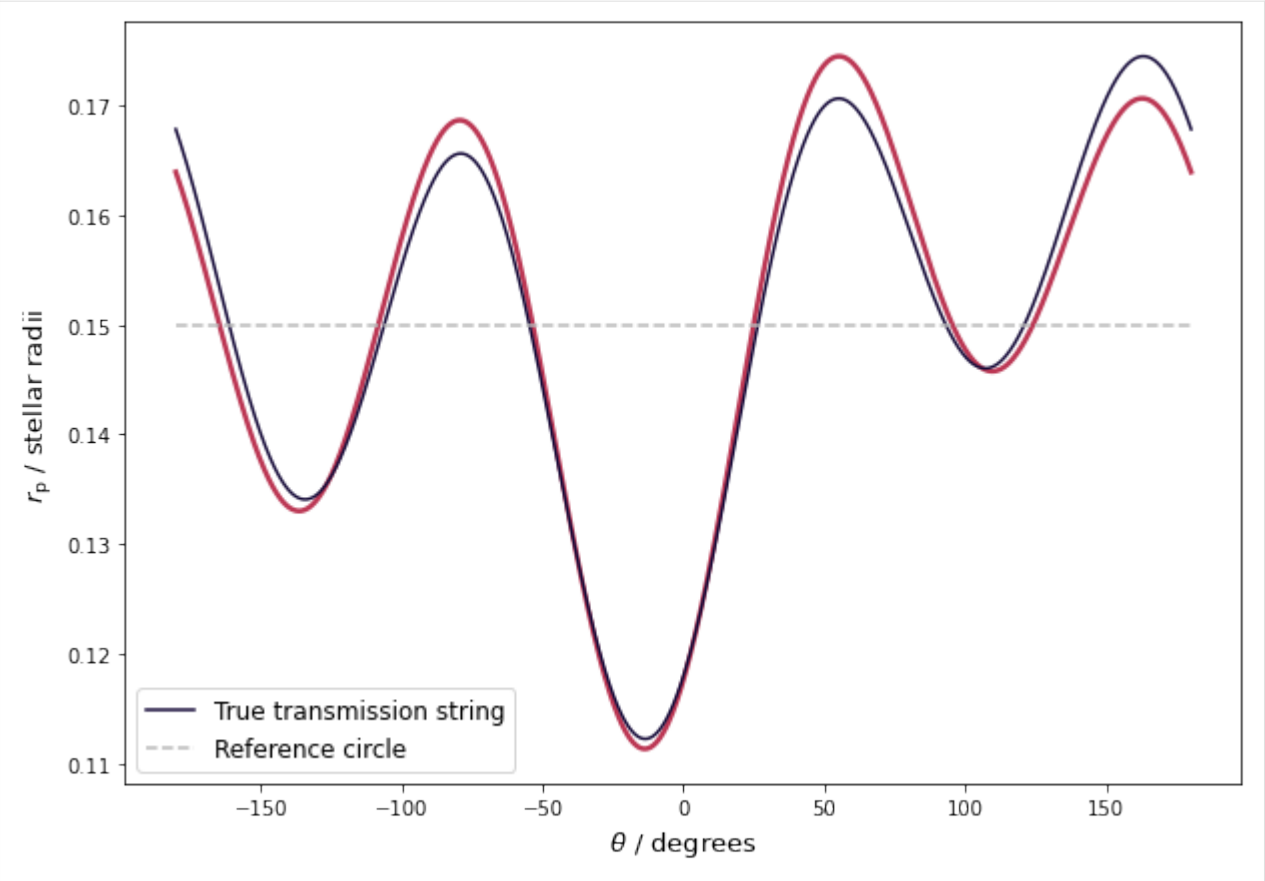
(continued from previous page)

```
plt.plot(injected_transmission_string * np.cos(theta),
         injected_transmission_string * np.sin(theta),
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



You may find it preferable to visualise these transmission strings in polar coordinates.

[6]:
```
plt.figure(figsize=(10, 7))
plt.plot(theta * 180. / np.pi, ht.get_planet_transmission_string(theta),
         c=cm.inferno(0.5), lw=2.5)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



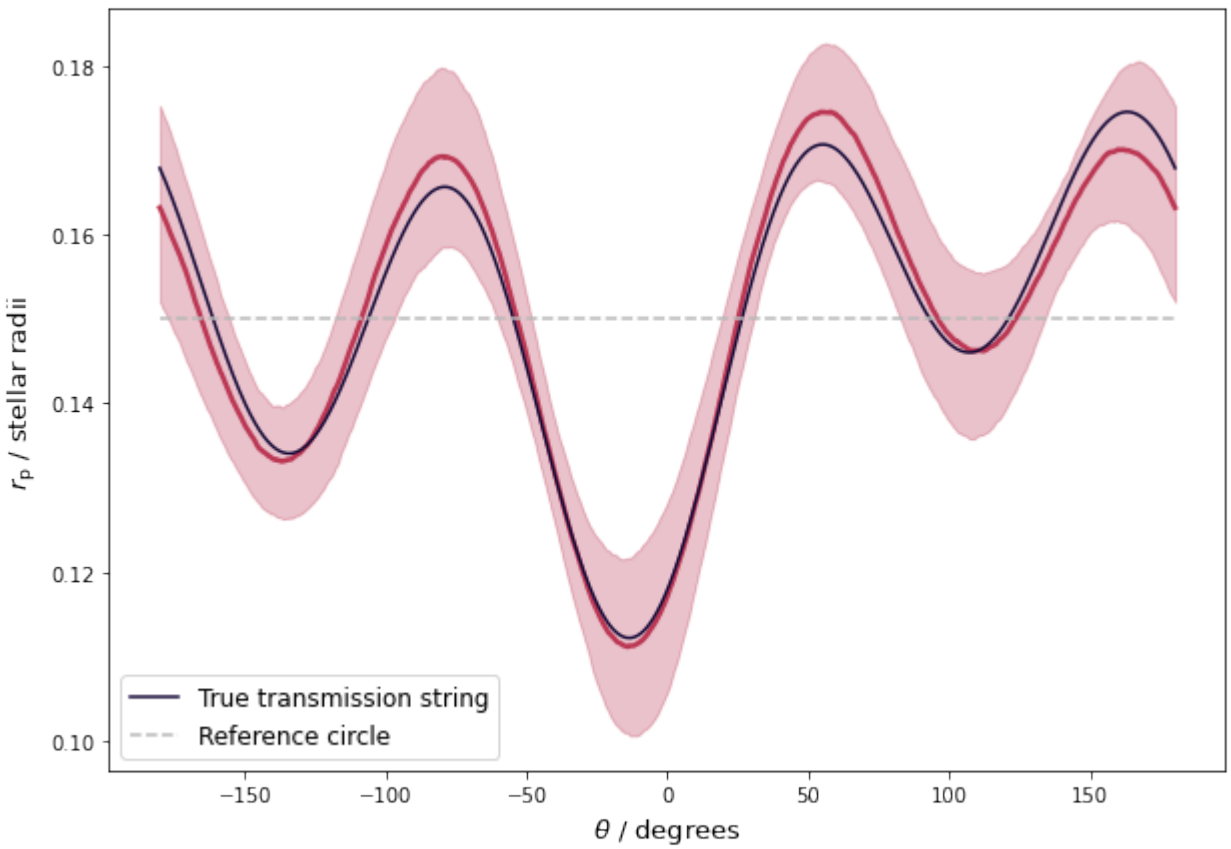
Further to this point estimate, you may also use the curvature of the likelihood function to inform the uncertainty in your estimates. The inverse of the Hessian matrix, used in the minimisation procedure, is equivalent to the asymptotic covariance matrix. The standard errors of the transmission string parameters can then easily be calculated; and in fact, we already have this in hand from our curve fitting.

[7]:
```
# Sample MLE transmission string parameter distributions.
n_mc_samples = 1000
mle_r_sigma = np.sqrt(np.diag(pcov))
mle_r_samples = np.random.normal(loc=popt, scale=mle_r_sigma, size=(n_mc_samples,␣
↪len(popt)))
ht.set_planet_transmission_string(mle_r_samples)
ts_samples = ht.get_planet_transmission_string(theta)

# Get 16th, 50th, 84th percentiles.
ts_16, ts_50, ts_84 = np.percentile(ts_samples, [16., 50., 84.], axis=0)

plt.figure(figsize=(10, 7))
plt.plot(theta * 180. / np.pi, ts_50, c=cm.inferno(0.5), lw=2.5)
plt.fill_between(theta * 180. / np.pi, ts_16, ts_84, color=cm.inferno(0.5), alpha=0.3)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



4.3 Basic inference of a transmission string

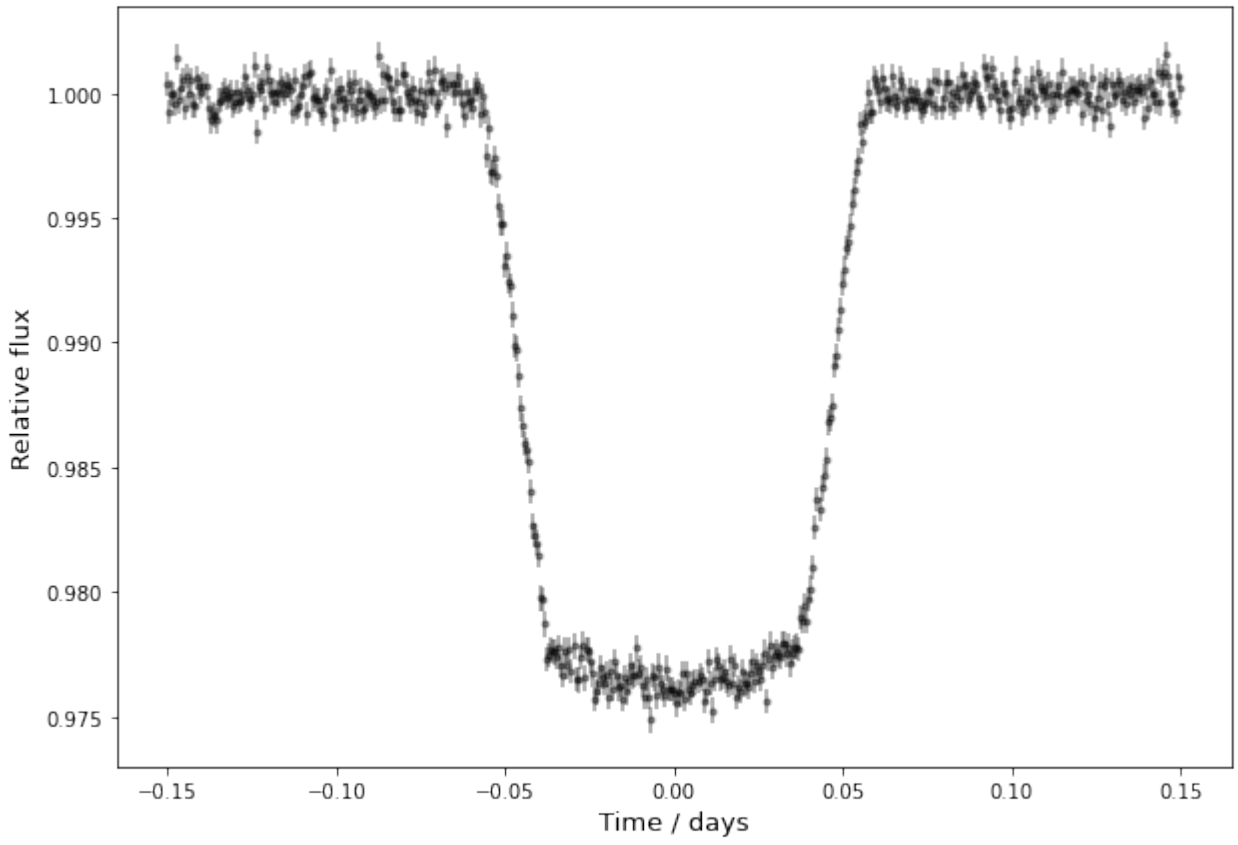
In this tutorial we take a look at how to fit a light curve with a basic Markov chain Monte Carlo (MCMC) approach, and infer a distribution of transmission strings. As in the previous tutorial, let us start by simulating a transit light curve for a known 7-parameter transmission string.

[1]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from harmonica import HarmonicaTransit


np.random.seed(12)

times = np.linspace(-0.15, 0.15, 500)
r_mean = np.array([0.15])
r_dev = np.random.uniform(-0.1, 0.1, size=6)
injected_r = np.concatenate([r_mean, r_dev * r_mean])
```

(continues on next page)

(continued from previous page)

```
ht = HarmonicaTransit(times)
ht.set_orbit(t0=0., period=4., a=11., inc=87. * np.pi / 180.)
ht.set_stellar_limb_darkening(np.array([0.027, 0.246]), limb_dark_law='quadratic')
ht.set_planet_transmission_string(injected_r)

theta = np.linspace(-np.pi, np.pi, 1000)
injected_transmission_string = ht.get_planet_transmission_string(theta)

flux_sigma = 500.e-6 * np.ones(times.shape[0])
flux_errs = np.random.normal(loc=0., scale=flux_sigma, size=times.shape[0])
observed_fluxes = ht.get_transit_light_curve() + flux_errs

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



This transit light curve corresponds to the following transmission string:

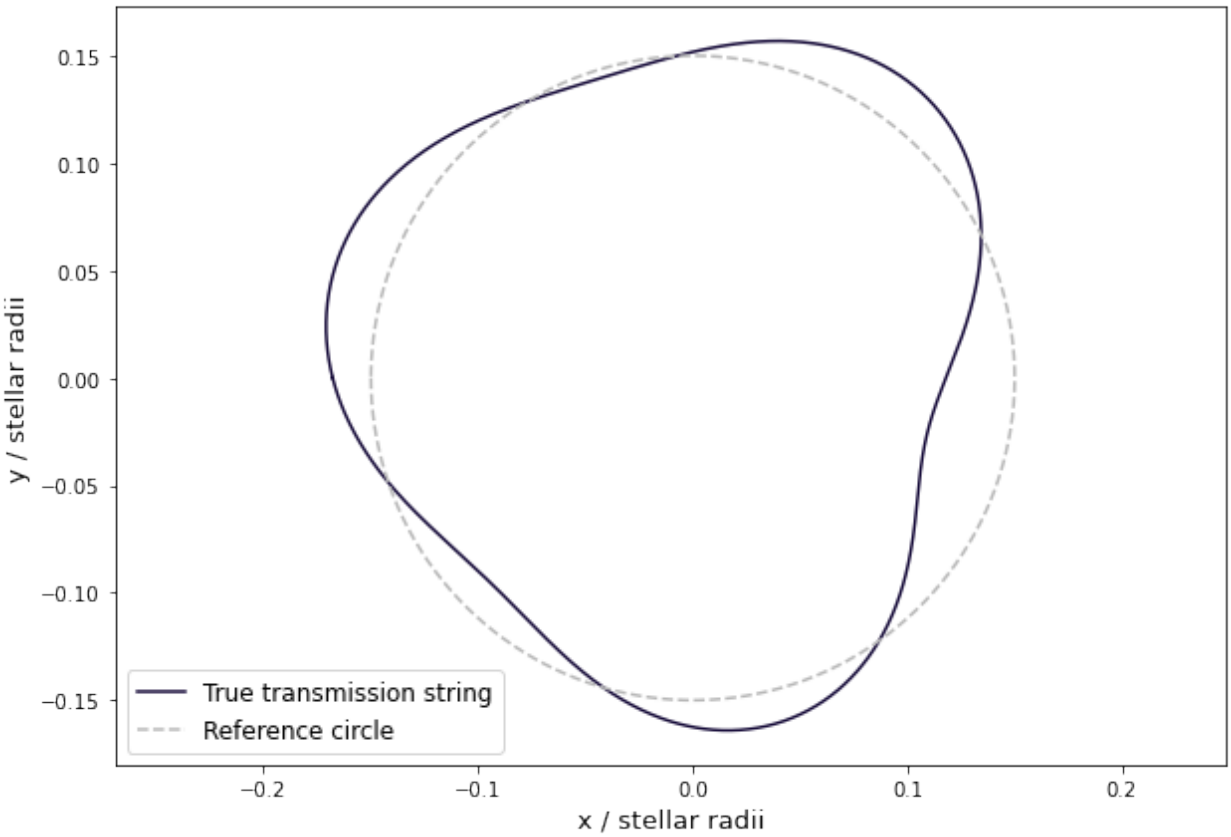
[2]:
```
print("r = {0:.3f}{1:+.3f}cos(t){2:+.3f}sin(t)"
      "{3:+.3f}cos(2t){4:+.3f}sin(2t)"
      "{5:+.3f}cos(3t){6:+.3f}sin(3t)".format(*injected_r))

plt.figure(figsize=(10, 7))
```

(continues on next page)

(continued from previous page)

```
plt.gca().set_aspect("equal", "datalim")
plt.plot(injected_transmission_string * np.cos(theta),
         injected_transmission_string * np.sin(theta),
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```

```
r = 0.150-0.010cos(t)+0.007sin(t)-0.007cos(2t)+0.001sin(2t)-0.015cos(3t)+0.013sin(3t)
```



Now you can apply Harmonica and sample the transmission string parameters. To do this we make use of the MCMC code, `emcee`.

[3]:
```
import emcee


def log_prob(params):
    # Ln(prior).
    ln_prior = -0.5 * np.sum(((params[0] - 0.15) / 0.05)**2)
    ln_prior += -0.5 * np.sum((params[1:] / 0.1)**2)

    # Typical Gaussian ln(likelihood).
```

(continues on next page)

(continued from previous page)

```
    ht.set_planet_transmission_string(params)
    model = ht.get_transit_light_curve()
    ln_like = -0.5 * np.sum((observed_fluxes - model)**2 / flux_sigma**2
                            + np.log(2 * np.pi * flux_sigma**2))

    return ln_like + ln_prior


coords = injected_r + 1.e-6 * np.random.randn(18, len(injected_r))
sampler = emcee.EnsembleSampler(coords.shape[0], coords.shape[1], log_prob)
state = sampler.run_mcmc(coords, 2000, progress=True)
sampler.reset()
state = sampler.run_mcmc(state, 5000, progress=True)
```

```
100%|| 2000/2000 [03:17<00:00, 10.10it/s]
100%|| 5000/5000 [08:17<00:00, 10.04it/s]
```

For this example, the runtime of this method is over 11 minutes. If you are interested in potentially more efficient sampling methods, then checkout the tutorial on gradient-based inference.

Next, you can check the key metrics of this run and take a look at the posterior parameter distributions using `arviz` and `corner`, respectively.

[4]:
```
import arviz as az


emcee_data = az.from_emcee(sampler,
                           var_names=["$a_0$", "$a_1$", "$b_1$", "$a_2$",
                                      "$b_2$", "$a_3$", "$b_3$"])
az.summary(emcee_data, round_to=5)
```

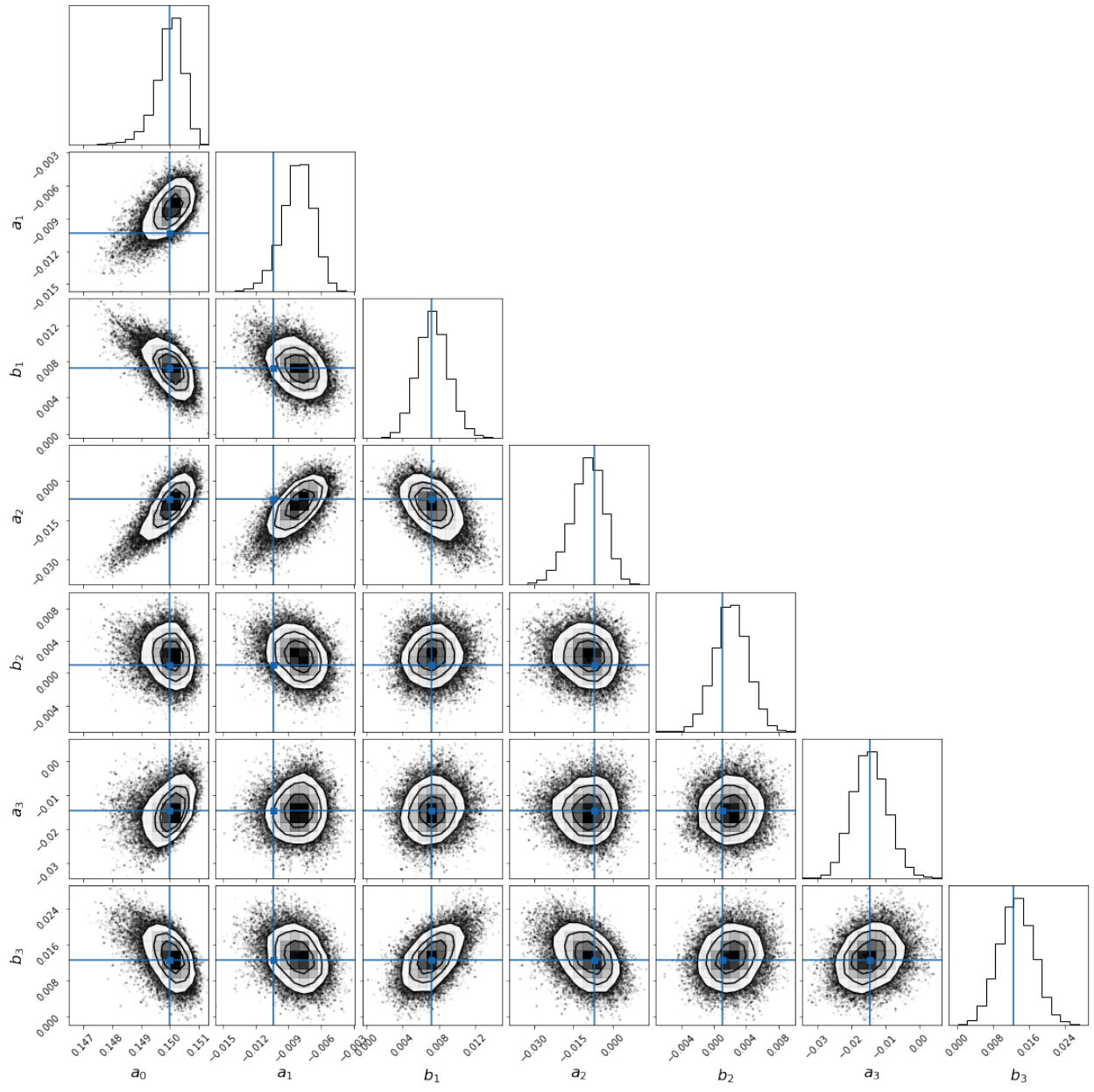
[4]:
```
          mean       sd   hdi_3%  hdi_97%  mcse_mean  mcse_sd    ess_bulk  \
$a_0$  0.14997  0.00049  0.14905  0.15083    0.00002  0.00002   597.50609
$a_1$ -0.00820  0.00145 -0.01095 -0.00552    0.00006  0.00004   632.64520
$b_1$  0.00730  0.00175  0.00397  0.01058    0.00006  0.00004   927.85854
$a_2$ -0.00997  0.00638 -0.02209  0.00178    0.00026  0.00019   687.49741
$b_2$  0.00209  0.00204 -0.00191  0.00582    0.00006  0.00005  1029.61370
$a_3$ -0.01449  0.00515 -0.02415 -0.00501    0.00023  0.00016   508.94111
$b_3$  0.01315  0.00392  0.00576  0.02044    0.00013  0.00009   913.52774

         ess_tail    r_hat
$a_0$  1288.90802  1.03484
$a_1$  1701.75242  1.03699
$b_1$  1899.00101  1.02438
$a_2$  1377.68028  1.03682
$b_2$  2713.65107  1.02179
$a_3$  1840.61672  1.03138
$b_3$  2313.72410  1.02280
```

[5]:
```
import corner


mcmc_r_samples = sampler.get_chain(flat=True)
```

(continues on next page)

(continued from previous page)

```
figure = corner.corner(mcmc_r_samples, truths=injected_r,
                       truth_color=cm.Blues(0.8), bins=15,
                       label_kwargs={"fontsize": 16},
                       labels=["$a_0$", "$a_1$", "$b_1$", "$a_2$",
                               "$b_2$", "$a_3$", "$b_3$"])
plt.show()
```



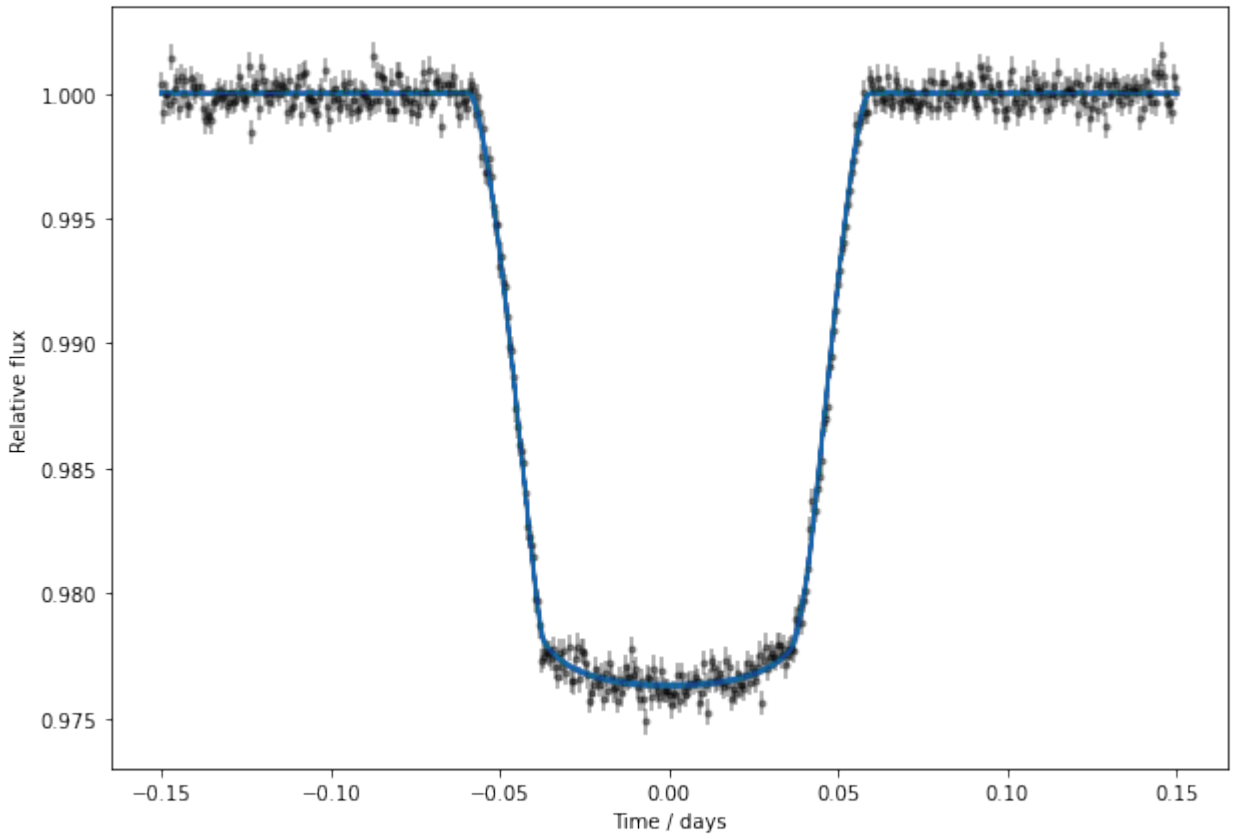
You can sample the posteriors and plot realisations of the fitted transit light curve:

[6]:
```
sample_idxs = np.random.randint(0, len(mcmc_r_samples), 150)

plt.figure(figsize=(10, 7))
```

(continues on next page)

(continued from previous page)

```
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4, zorder=1)
for r_sample in mcmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(times, ht.get_transit_light_curve(), c=cm.Blues(0.8), alpha=0.03, zorder=2)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



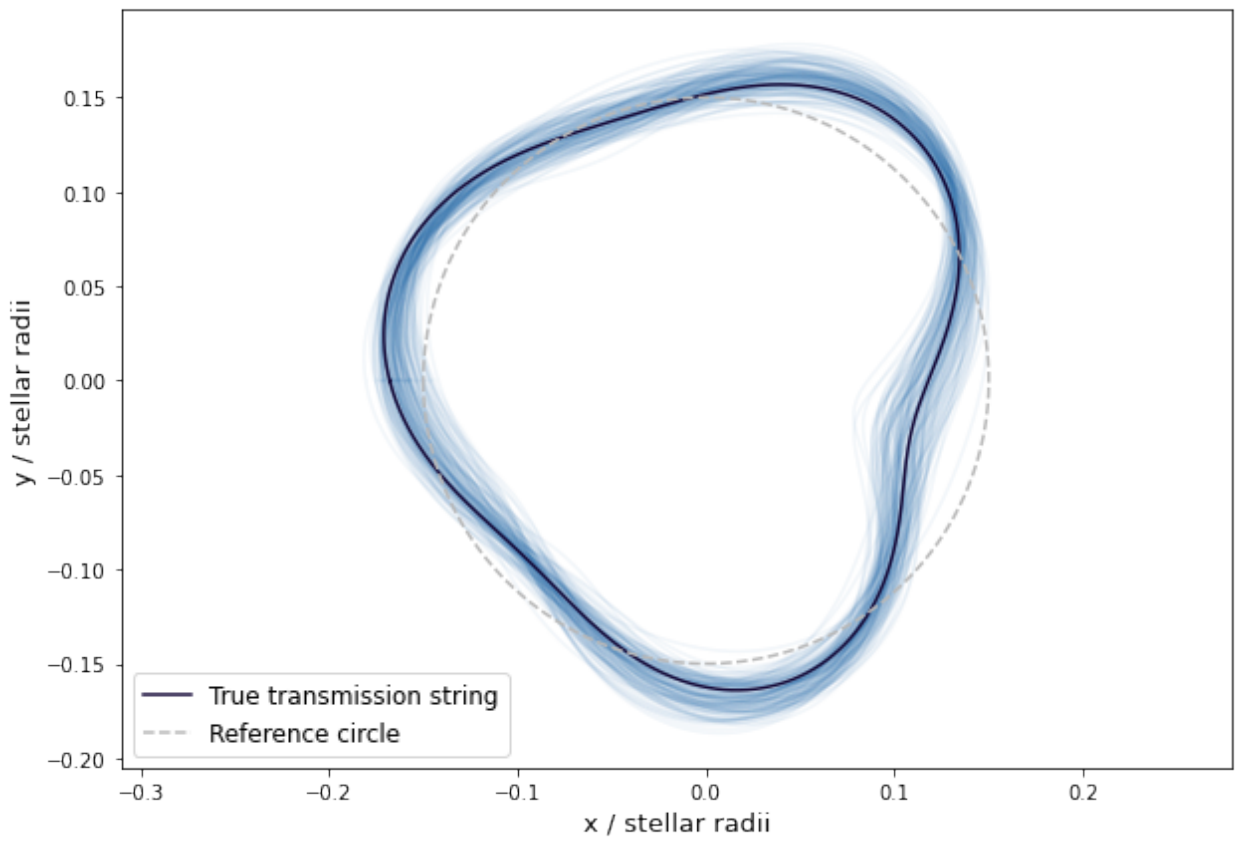
And here are sampled transmission strings:

[7]:
```
plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
for r_sample in mcmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(ht.get_planet_transmission_string(theta) * np.cos(theta),
             ht.get_planet_transmission_string(theta) * np.sin(theta),
             c=cm.Blues(0.8), alpha=0.05)
plt.plot(injected_transmission_string * np.cos(theta),
         injected_transmission_string * np.sin(theta),
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
```

(continues on next page)

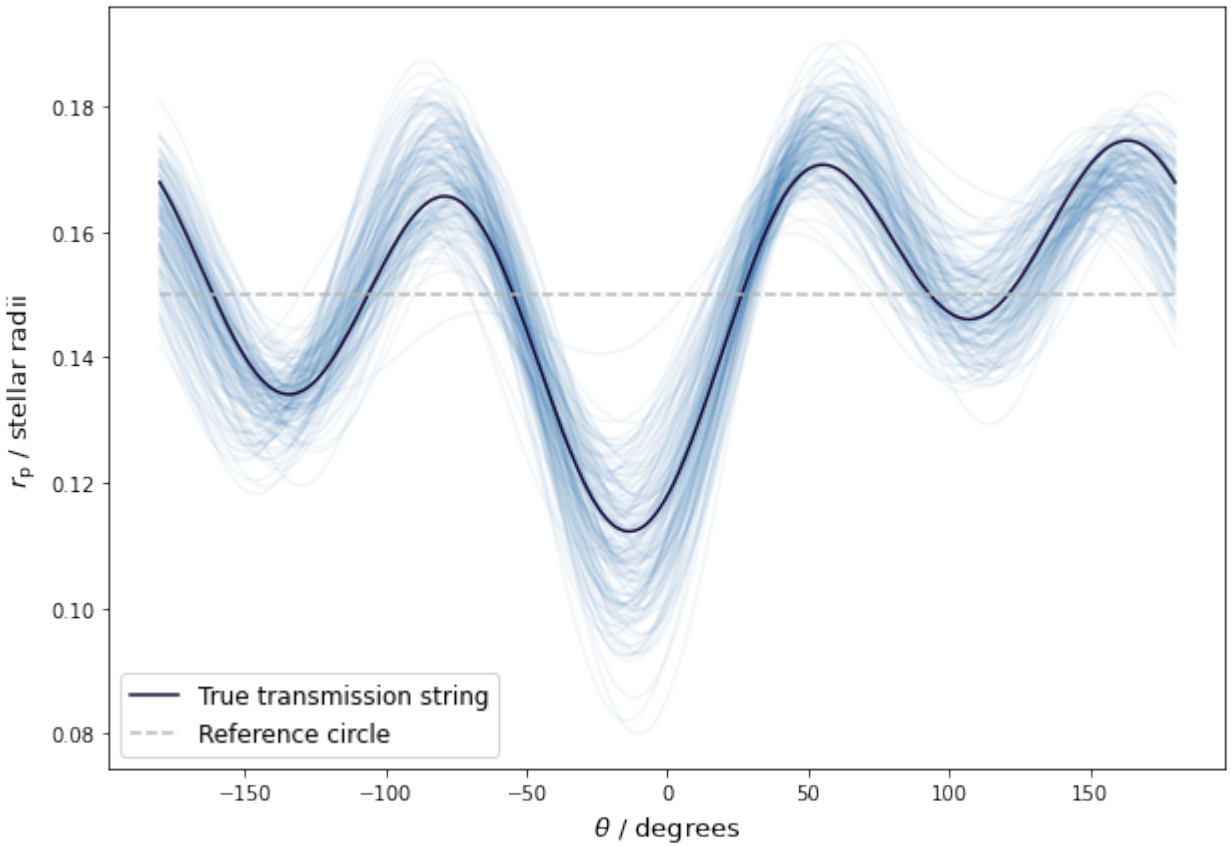
(continued from previous page)

```
plt.show()
```



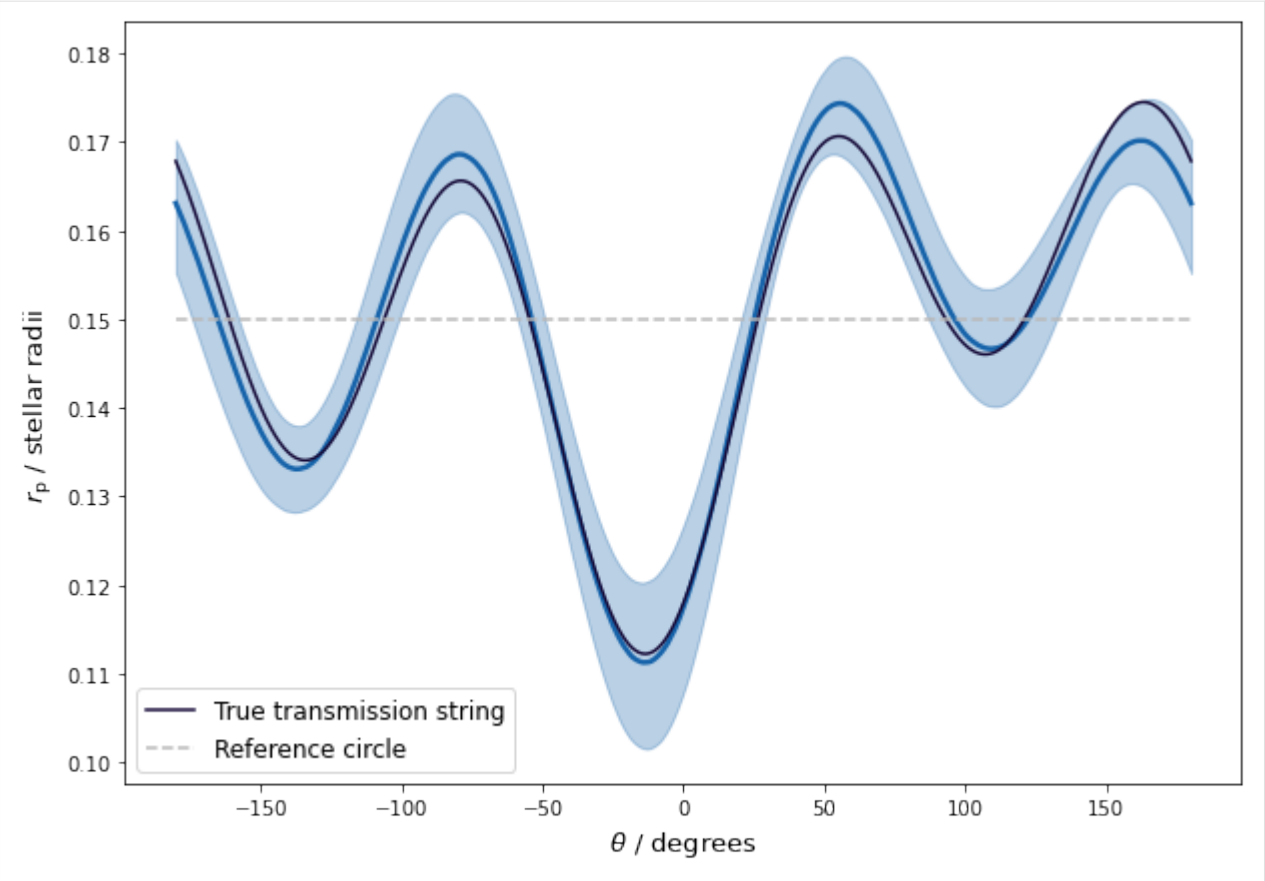
In polar coordinates:

[8]:
```
plt.figure(figsize=(10, 7))
for r_sample in mcmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(theta * 180. / np.pi, ht.get_planet_transmission_string(theta),
             c=cm.Blues(0.8), alpha=0.05)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



From these samples, you can estimate the inferred distribution of transmission strings. Here you can see the injected transmission string is nicely recovered.

[9]:
```
# Sample MCMC transmission string parameter distributions.
ht.set_planet_transmission_string(mcmc_r_samples)
ts_samples = ht.get_planet_transmission_string(theta)

# Get 16th, 50th, 84th percentiles.
ts_16, ts_50, ts_84 = np.percentile(ts_samples, [16., 50., 84.], axis=0)

plt.figure(figsize=(10, 7))
plt.plot(theta * 180. / np.pi, ts_50, c=cm.Blues(0.8), lw=2.5)
plt.fill_between(theta * 180. / np.pi, ts_16, ts_84, color=cm.Blues(0.8), alpha=0.3)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



4.4 Gradient-based inference of a transmission string

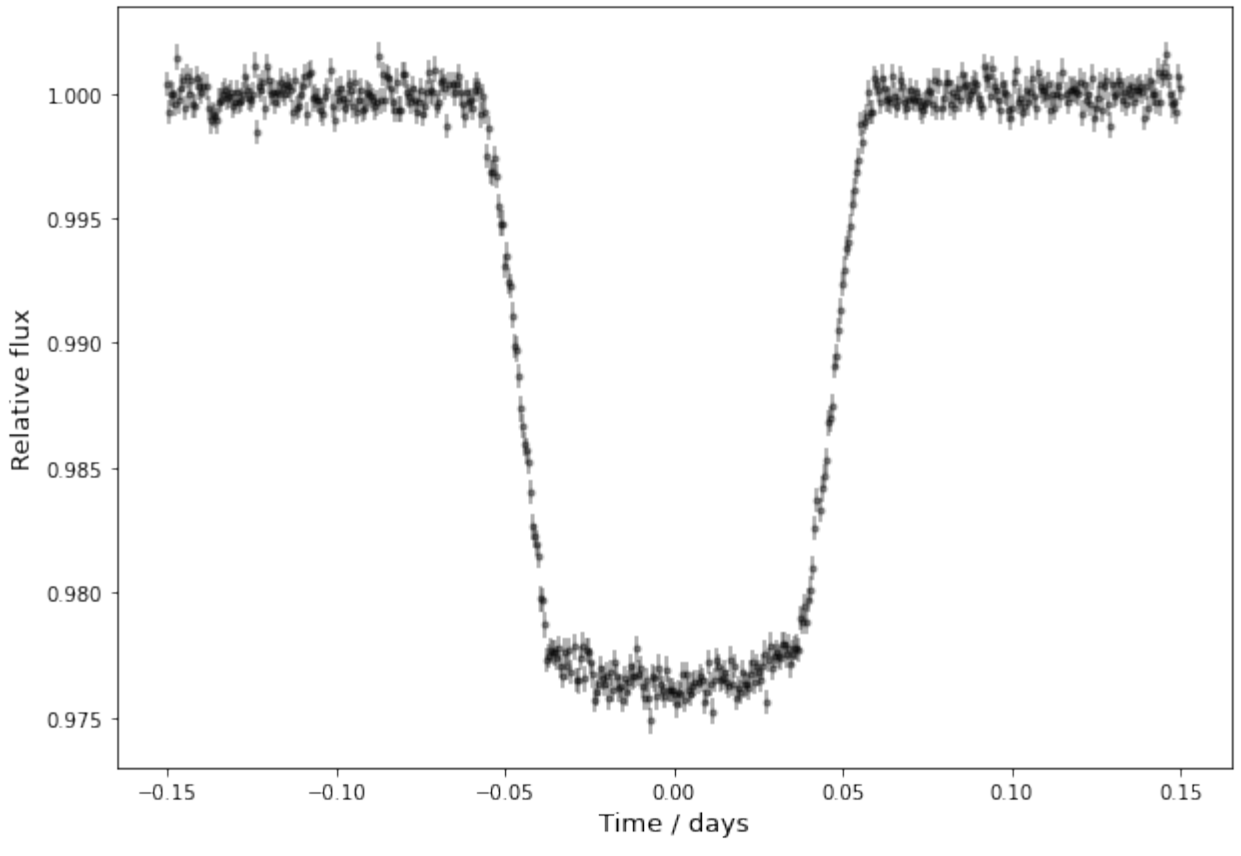
In this tutorial we take a look at how to fit a light curve with a gradient-based Markov chain Monte Carlo (MCMC) approach, known as Hamiltonian Monte Carlo (HMC), and infer a distribution of transmission strings. HMC has the advantage that is reduces the correlation between successive samples, thereby enabling more efficient sampling and lower utilisation of computation energy and time. As in the previous tutorial, let us start by simulating a transit light curve for a known 7-parameter transmission string.

[1]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from harmonica import HarmonicaTransit


np.random.seed(12)

times = np.linspace(-0.15, 0.15, 500)
r_mean = np.array([0.15])
r_dev = np.random.uniform(-0.1, 0.1, size=6)
injected_r = np.concatenate([r_mean, r_dev * r_mean])

ht = HarmonicaTransit(times)
ht.set_orbit(t0=0., period=4., a=11., inc=87. * np.pi / 180.)
```

(continues on next page)

(continued from previous page)

```
ht.set_stellar_limb_darkening(np.array([0.027, 0.246]), limb_dark_law='quadratic')
ht.set_planet_transmission_string(injected_r)

theta = np.linspace(-np.pi, np.pi, 1000)
injected_transmission_string = ht.get_planet_transmission_string(theta)

flux_sigma = 500.e-6 * np.ones(times.shape[0])
flux_errs = np.random.normal(loc=0., scale=flux_sigma, size=times.shape[0])
observed_fluxes = ht.get_transit_light_curve() + flux_errs

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



This transit light curve corresponds to the following transmission string:

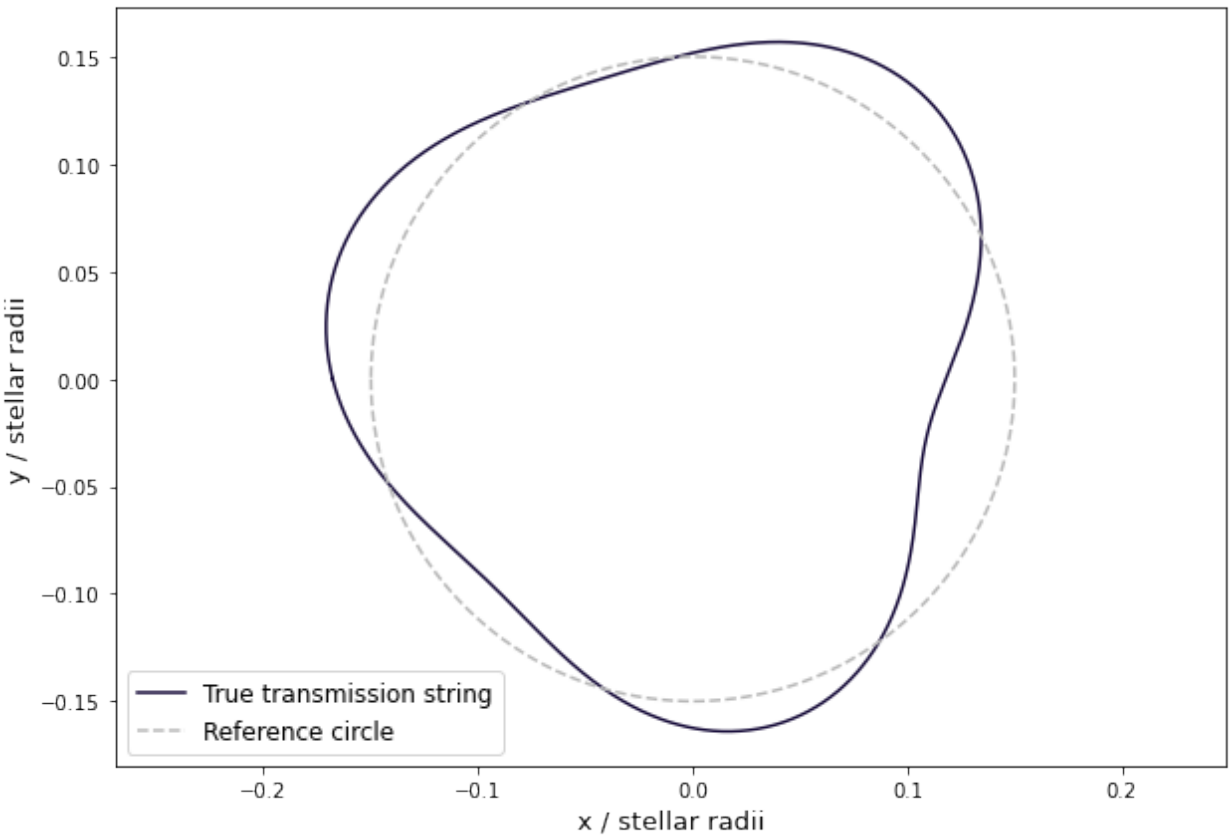
[2]:
```
print("r = {0:.3f}{1:+.3f}cos(t){2:+.3f}sin(t)"
      "{3:+.3f}cos(2t){4:+.3f}sin(2t)"
      "{5:+.3f}cos(3t){6:+.3f}sin(3t)".format(*injected_r))

plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.plot(injected_transmission_string * np.cos(theta),
```

(continues on next page)

(continued from previous page)

```
        injected_transmission_string * np.sin(theta),
        c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
        c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```

```
r = 0.150-0.010cos(t)+0.007sin(t)-0.007cos(2t)+0.001sin(2t)-0.015cos(3t)+0.013sin(3t)
```



Now you can apply Harmonica and sample the transmission string parameters. To do this we make use of the HMC code, `numpyro`. This code requires `jax` in order to compute derivatives. In order to be compatible with these requirements, we provide a subpackage, harmonica.jax, which can be incorporated into a numpyro model as follows:

[3]:
```
import jax
import numpyro
import jax.numpy as jnp
import numpyro.distributions as dist
from harmonica.jax import harmonica_transit_quad_ld
from numpyro.infer import MCMC, NUTS, init_to_median


def numpyro_model(t, f_obs=None):
    # Zeroth-order planet radius: r0.
```

(continues on next page)

(continued from previous page)

```
    r0 = numpyro.sample('r0', dist.Uniform(0.15 - 0.05, 0.15 + 0.05))

    # Higher-order radius harmonics: rn/r0.
    rn_frac = numpyro.sample('rn_frac', dist.Normal(0.0, 0.1), sample_shape=(6,))

    # Transmission string parameter vector.
    r = numpyro.deterministic('r', jnp.concatenate([jnp.array([r0]), rn_frac * r0]))

    # Model evaluation: this is our custom JAX primitive.
    fs = harmonica_transit_quad_ld(
        t, t0=0., period=4., a=11., inc=87. * np.pi / 180., u1=0.027, u2=0.246, r=r)

    # Condition on the observations.
    numpyro.sample('obs', dist.Normal(fs, flux_sigma), obs=f_obs)


nuts_kernel = NUTS(numpyro_model, dense_mass=True, adapt_mass_matrix=True,
                   max_tree_depth=7, target_accept_prob=0.75,
                   init_strategy=init_to_median())

hmc = MCMC(nuts_kernel, num_warmup=200, num_samples=500,
           num_chains=2, chain_method="sequential", progress_bar=True)

hmc.run(jax.random.PRNGKey(2), times, f_obs=observed_fluxes)
```

```
sample: 100%|| 700/700 [02:09<00:00,  5.39it/s, 7 steps of size 3.76e-01. acc. prob=0.91]
sample: 100%|| 700/700 [02:25<00:00,  4.82it/s, 7 steps of size 4.17e-01. acc. prob=0.90]
```

For this example, the runtime of this method is now ~4 minutes. Compared to the basic inference tutorial, this represents a good efficiency improvement for a similar number of effective samples.

Next, you can check the key metrics of this run and take a look at the posterior parameter distributions using `arviz` and `corner`, respectively.

[4]:
```
import arviz as az


numpyro_data = az.from_numpyro(hmc)
az.summary(numpyro_data, var_names=['r'], round_to=5)
```

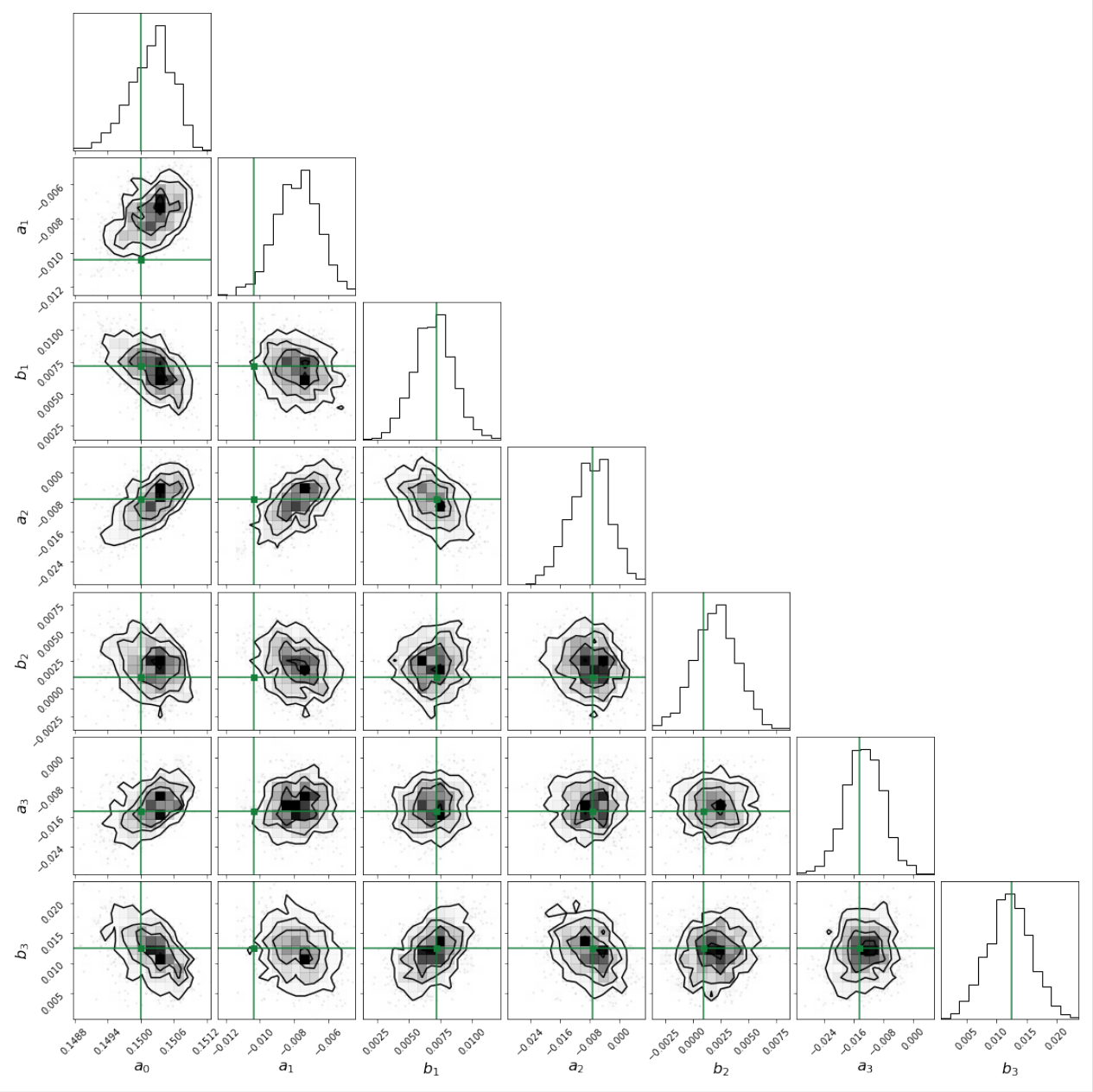
[4]:
```
         mean       sd   hdi_3%  hdi_97%  mcse_mean  mcse_sd    ess_bulk  \
r[0]  0.15018  0.00040  0.14940  0.15084    0.00001  0.00001   953.40429
r[1] -0.00783  0.00126 -0.01038 -0.00563    0.00004  0.00003  1180.15532
r[2]  0.00691  0.00159  0.00386  0.00980    0.00005  0.00004  1170.08260
r[3] -0.00808  0.00554 -0.01870  0.00179    0.00019  0.00015   889.68144
r[4]  0.00203  0.00192 -0.00108  0.00600    0.00007  0.00005   847.36444
r[5] -0.01326  0.00508 -0.02277 -0.00377    0.00020  0.00015   673.15398
r[6]  0.01209  0.00363  0.00488  0.01850    0.00014  0.00010   622.63127

       ess_tail    r_hat
r[0]  694.18954  1.00468
r[1]  948.02794  0.99826
r[2]  699.29110  1.00211
r[3]  618.98218  0.99909
```

(continues on next page)

(continued from previous page)

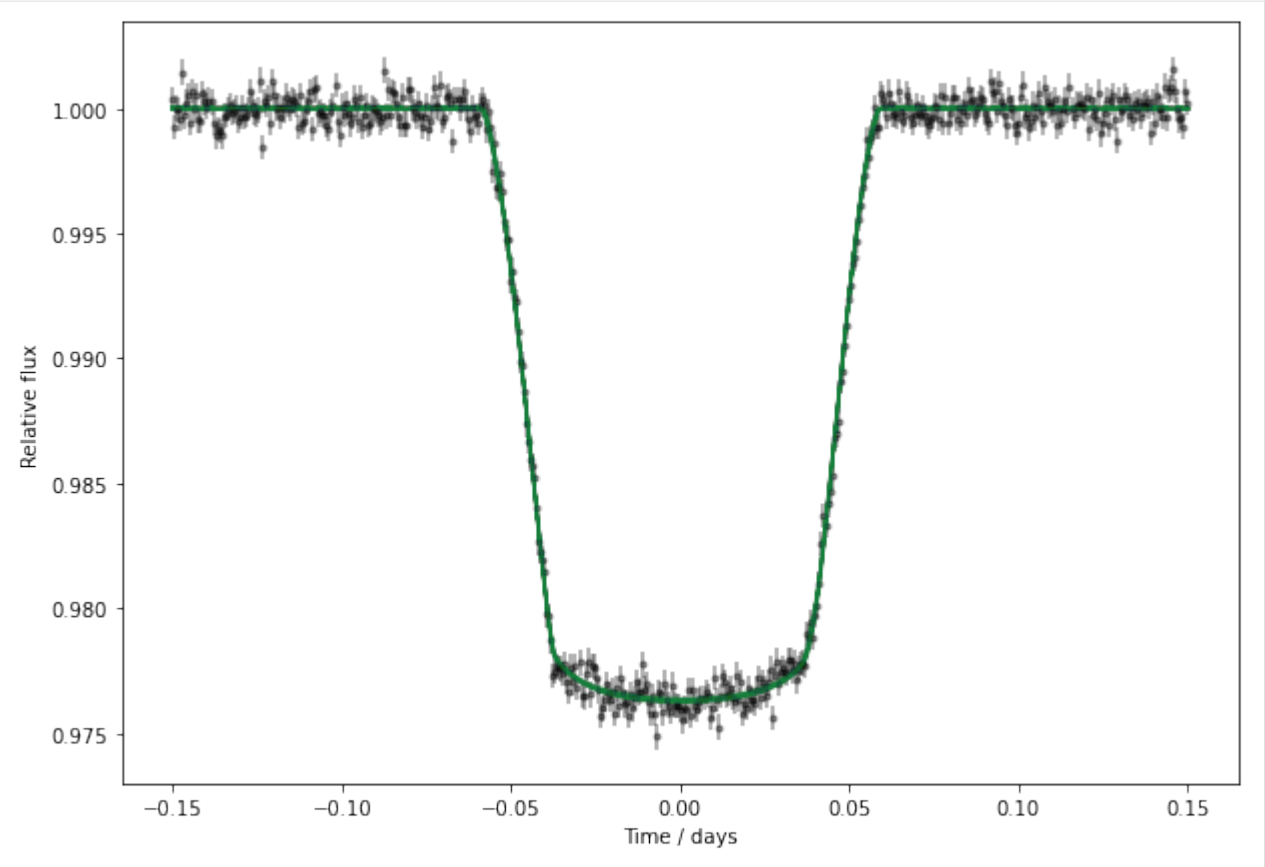
```
r[4]  721.85776  0.99874
r[5]  528.34178  1.00432
r[6]  678.51718  1.00031
```

[5]:
```
import corner


hmc_r_samples = np.array(hmc.get_samples()['r'])
figure = corner.corner(hmc_r_samples, truths=injected_r,
                       truth_color=cm.BuGn(0.8), bins=15,
                       label_kwargs={"fontsize": 16},
                       labels=["$a_0$", "$a_1$", "$b_1$", "$a_2$",
                               "$b_2$", "$a_3$", "$b_3$"])
plt.show()
```



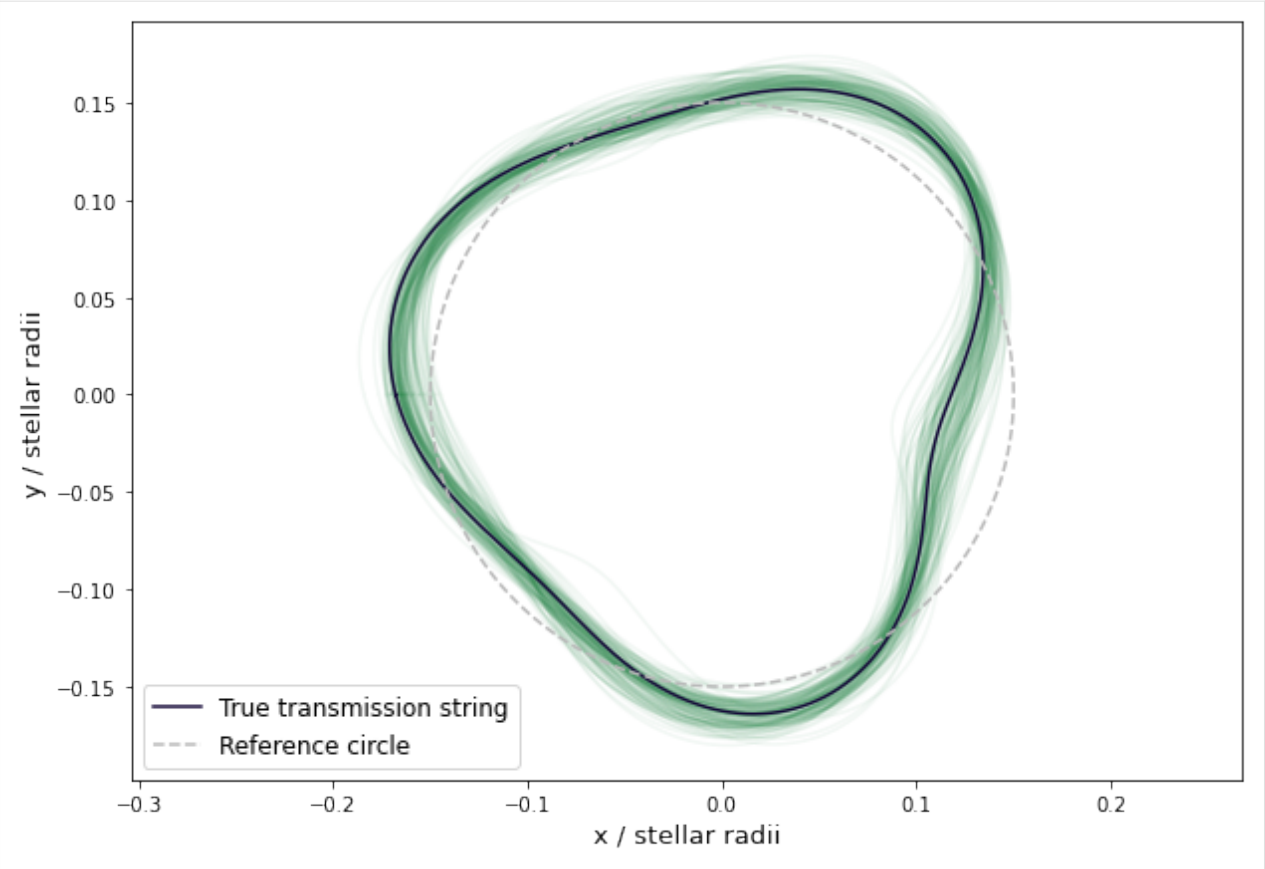
You can sample the posteriors and plot realisations of the fitted transit light curve:

[6]:
```
sample_idxs = np.random.randint(0, len(hmc_r_samples), 150)

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4, zorder=1)
for r_sample in hmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(times, ht.get_transit_light_curve(), c=cm.BuGn(0.8), alpha=0.03, zorder=2)
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



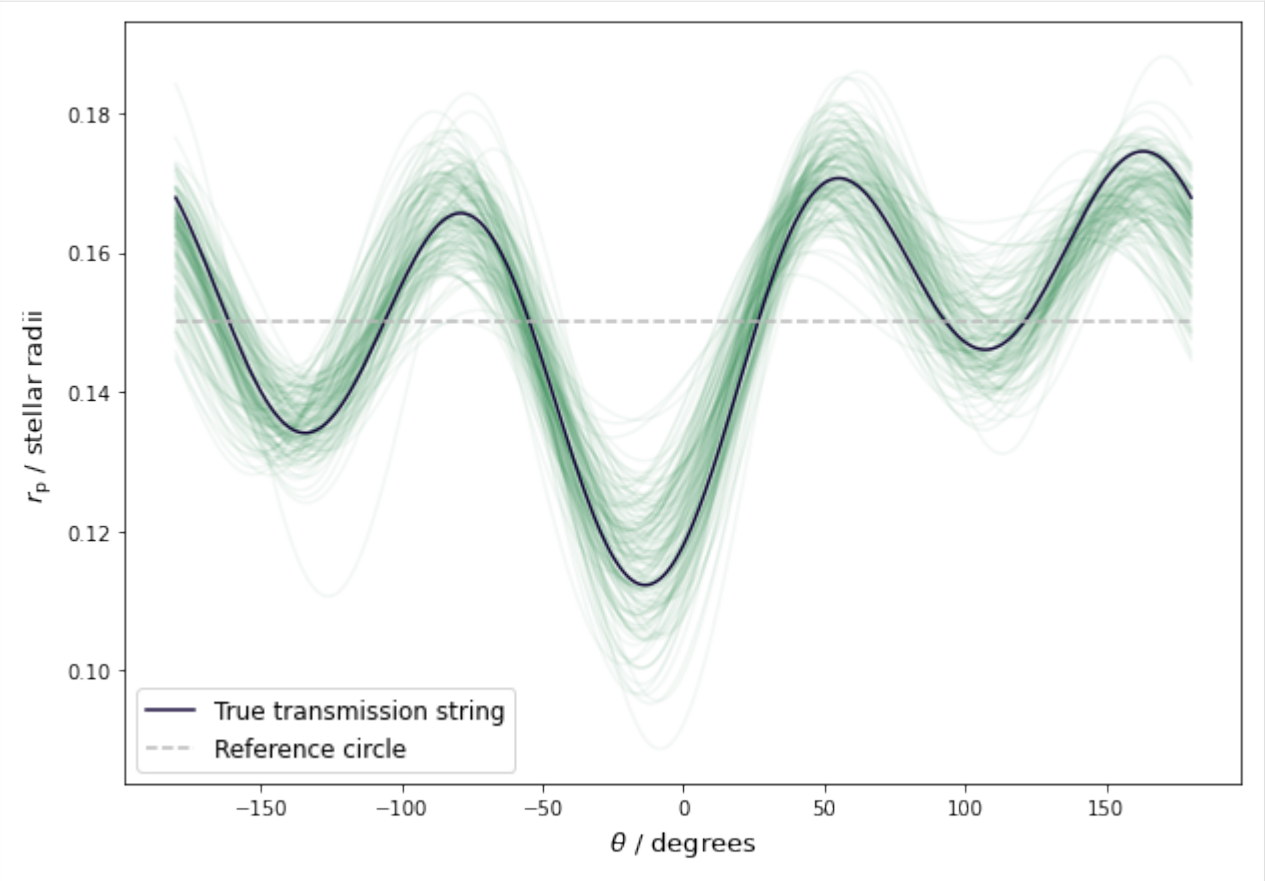
And here are sampled transmission strings:

[7]:
```
plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
for r_sample in hmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(ht.get_planet_transmission_string(theta) * np.cos(theta),
             ht.get_planet_transmission_string(theta) * np.sin(theta),
             c=cm.BuGn(0.8), alpha=0.05)
plt.plot(injected_transmission_string * np.cos(theta),
         injected_transmission_string * np.sin(theta),
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



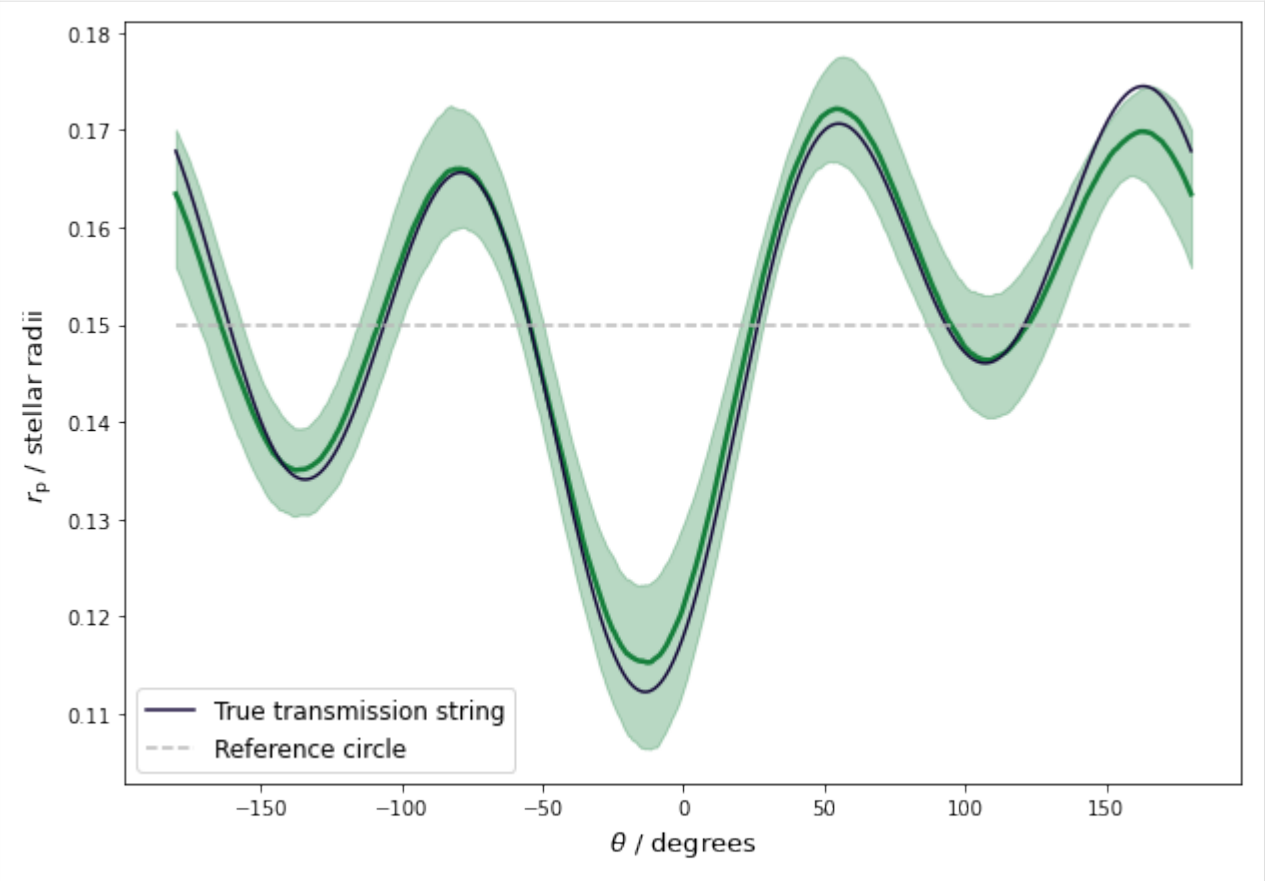
In polar coordinates:

[8]:
```
plt.figure(figsize=(10, 7))
for r_sample in hmc_r_samples[sample_idxs]:
    ht.set_planet_transmission_string(r_sample)
    plt.plot(theta * 180. / np.pi, ht.get_planet_transmission_string(theta),
             c=cm.BuGn(0.8), alpha=0.05)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



From these samples, you can estimate the inferred distribution of transmission strings. Here you can see the injected transmission string is nicely recovered.

[9]:
```
# Sample HMC transmission string parameter distributions.
ht.set_planet_transmission_string(hmc_r_samples)
ts_samples = ht.get_planet_transmission_string(theta)

# Get 16th, 50th, 84th percentiles.
ts_16, ts_50, ts_84 = np.percentile(ts_samples, [16., 50., 84.], axis=0)

plt.figure(figsize=(10, 7))
plt.plot(theta * 180. / np.pi, ts_50, c=cm.BuGn(0.8), lw=2.5)
plt.fill_between(theta * 180. / np.pi, ts_16, ts_84, color=cm.BuGn(0.8), alpha=0.3)
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(theta * 180. / np.pi, injected_r[0] * np.ones(theta.shape[0]),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



4.5 Assessing model evidence

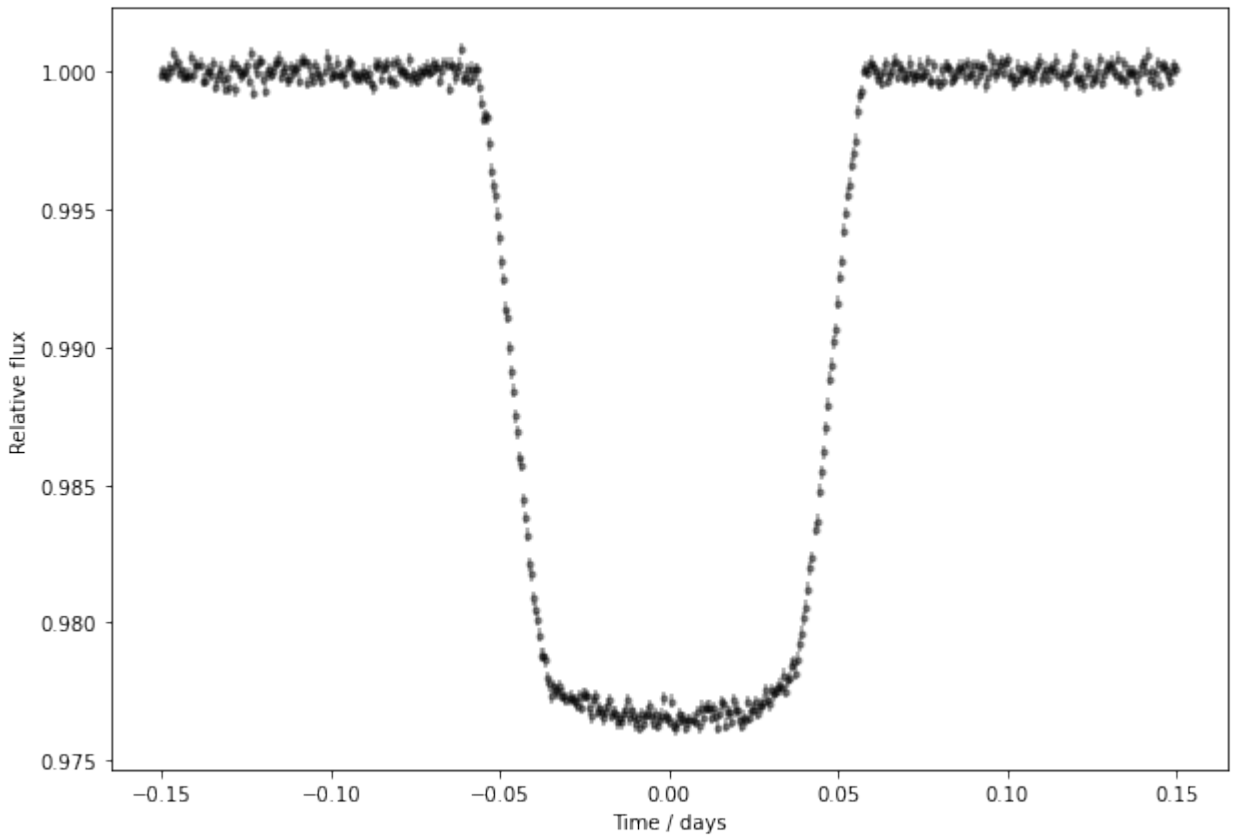
In this tutorial we take a look at how to assess how many parameters may be statistically justified in a given transmission string. Let us start by simulating a high-precision transit light curve with a 4-parameter transmission string.

[1]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from harmonica import HarmonicaTransit


np.random.seed(12)

times = np.linspace(-0.15, 0.15, 500)
r_mean = np.array([0.15])
r_dev = np.random.uniform(-0.1, 0.1, size=3)
injected_r = np.concatenate([r_mean, r_dev * r_mean])

ht = HarmonicaTransit(times)
ht.set_orbit(t0=0., period=4., a=11., inc=87. * np.pi / 180.)
ht.set_stellar_limb_darkening(np.array([0.027, 0.246]), limb_dark_law='quadratic')
ht.set_planet_transmission_string(injected_r)
```

(continues on next page)

(continued from previous page)

```
theta = np.linspace(-np.pi, np.pi, 1000)
injected_transmission_string = ht.get_planet_transmission_string(theta)

flux_sigma = 250.e-6 * np.ones(times.shape[0])
flux_errs = np.random.normal(loc=0., scale=flux_sigma, size=times.shape[0])
observed_fluxes = ht.get_transit_light_curve() + flux_errs

plt.figure(figsize=(10, 7))
plt.errorbar(times, observed_fluxes, yerr=flux_sigma, fmt=".k", alpha=0.4)
plt.xlabel('Time / days')
plt.ylabel('Relative flux')
plt.show()
```



This transit light curve corresponds to the following transmission string:

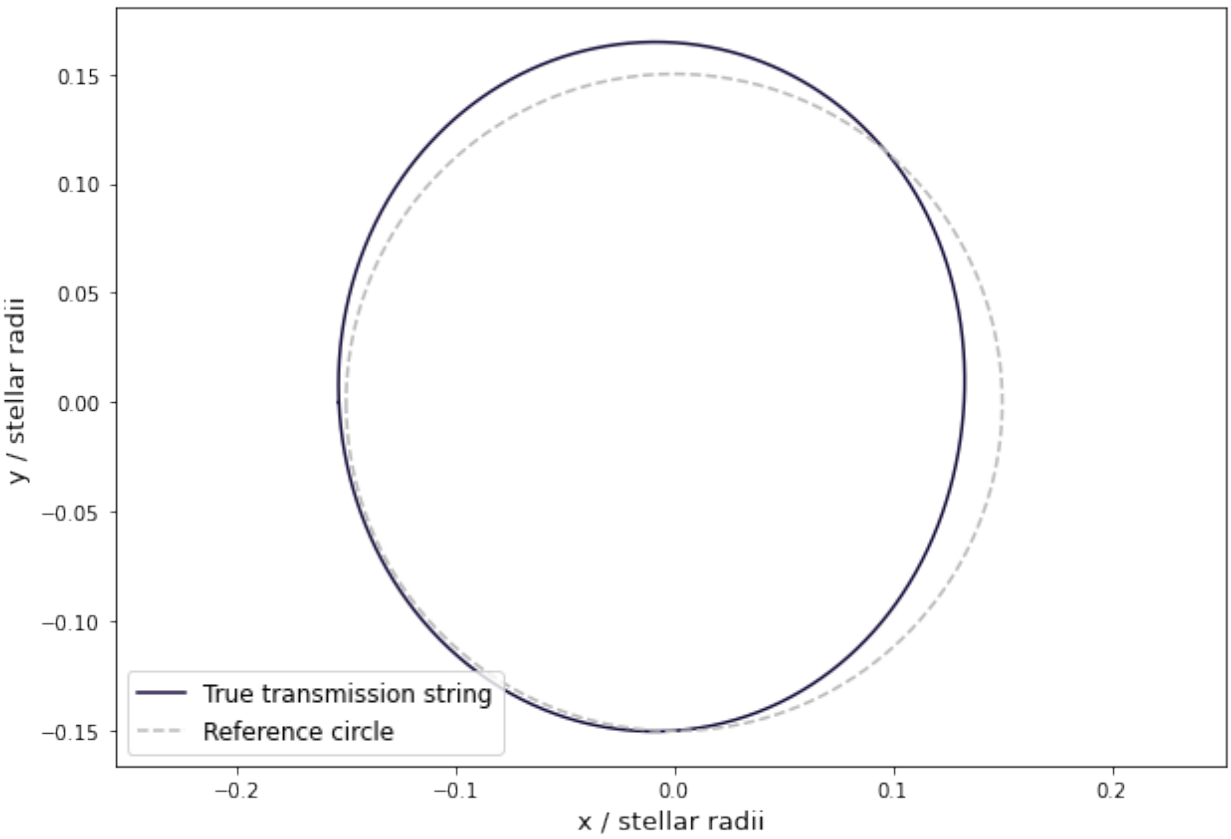
```
[2]: print("r = {0:.3f}{1:+.3f}cos(t){2:+.3f}sin(t){3:+.3f}cos(2t)".format(*injected_r))

plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.plot(injected_transmission_string * np.cos(theta),
         injected_transmission_string * np.sin(theta),
         c=cm.inferno(0.1), label="True transmission string")
plt.plot(injected_r[0] * np.cos(theta), injected_r[0] * np.sin(theta),
         c="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```

```
r = 0.150-0.010cos(t)+0.007sin(t)-0.007cos(2t)
```



Now, following the maximum likelihood tutorial, you can apply Harmonica with successively more and more transmission string parameters. For each fit you can calculate some model selection metric, such as the Bayesian information criterion (BIC), and use this to inform how much model complexity is justified by the data.

[3]:
```
from scipy.optimize import curve_fit


def transit_model(_, *params):
    ht.set_planet_transmission_string(np.array(params))
    model = ht.get_transit_light_curve()

    return model


def bic(model, n):
    ln_max_likelihood = -0.5 * np.sum((observed_fluxes - model)**2 / flux_sigma**2
                                      + np.log(2 * np.pi * flux_sigma**2))
    k = injected_r.shape[0]
    bic = k * np.log(n) - 2. * ln_max_likelihood
```

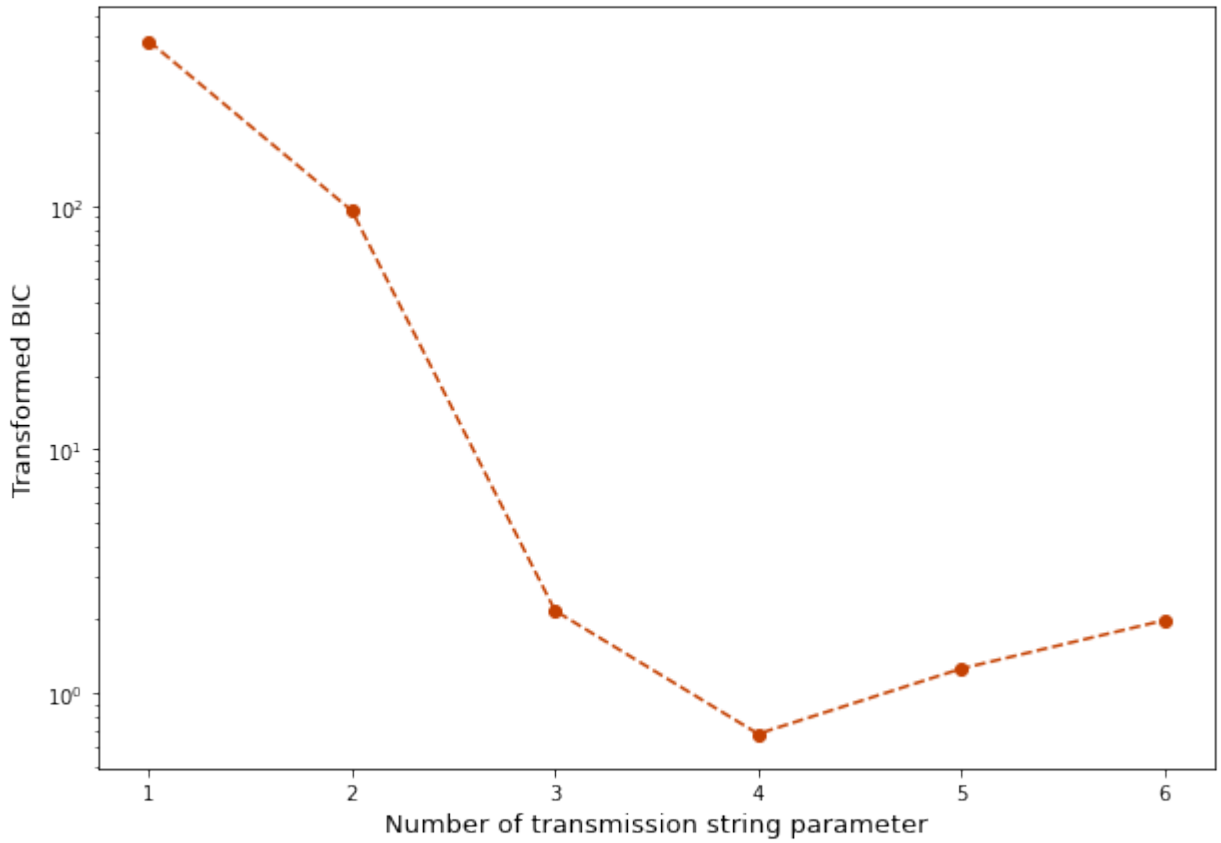
(continues on next page)

(continued from previous page)

```
    return bic


n_params = np.arange(1, 7, 1)
bic_scores = []
mle_transmission_strings = []
for n_p in n_params:

    popt, pcov = curve_fit(
        transit_model, times, observed_fluxes, sigma=flux_sigma,
        p0=np.concatenate([r_mean, np.zeros(n_p - 1)]),
        method='lm')

    bic_score = bic(transit_model(times, *popt), n_p)
    bic_scores.append(bic_score)

    ht.set_planet_transmission_string(np.array(popt))
    transmission_string = ht.get_planet_transmission_string(theta)
    mle_transmission_strings.append(transmission_string)
```

After the fitting is complete you can plot the BIC scores. Note it is only the relative scores which matter, and so we transform the scores for easier visualisation.

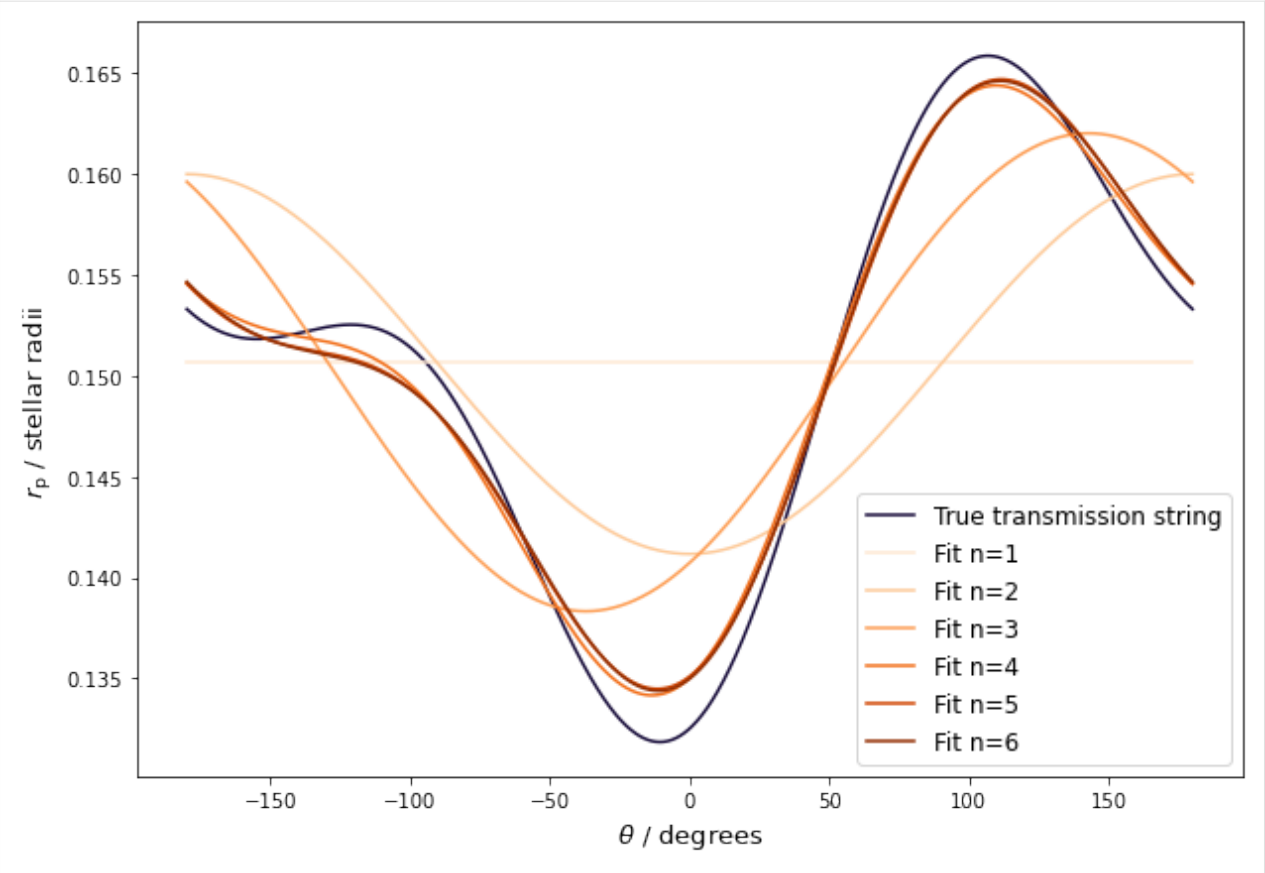
[4]:
```
bic_scores_tf = np.array(bic_scores) - np.floor(np.min(bic_scores))

plt.figure(figsize=(10, 7))
plt.plot(n_params, bic_scores_tf, color=cm.Oranges(0.8), ls="--")
plt.scatter(n_params, bic_scores_tf, color=cm.Oranges(0.8))
plt.yscale("log")
plt.xlabel('Number of transmission string parameter', fontsize=13)
plt.ylabel('Transformed BIC', fontsize=13)
plt.show()
```



The model with the lowest BIC is generally preferred. Here the model with a 4-parameter transmission string is correctly ascertained to be the statistically preferred model.

The meaning behind the above trend can be seen more clearly by plotting the MLE transmission strings for each model. Here it can be seen how the model improves substantially between 1 and 4 parameters, but after this any small improvements are not justifed given the number of free parameters.

[5]:
```
plt.figure(figsize=(10, 7))
plt.plot(theta * 180. / np.pi, injected_transmission_string,
         c=cm.inferno(0.1), label="True transmission string")
for i, transmission_string in enumerate(mle_transmission_strings):
    plt.plot(theta * 180. / np.pi, transmission_string,
             color=cm.Oranges(i/len(mle_transmission_strings) + 0.1), label="Fit n={}".
↪format(n_params[i]))

plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower right", fontsize=12)
plt.show()
```



Aside from model complexity, the data quality also plays an important role. You can try it yourself, but if you double the data's uncertainties in this tutorial, then the statistically preferred number of parameters drops from 4 to 3.

4.6 Calculate Fourier parameters from shape data

In this tutorial we take a look at how to construct Harmonica's Fourier parameterisation of a transmission string from shape data. This is useful for forward modelling transit light curves of various objects with known shapes. You only need to determine some high-order Fourier representation of the shape, and then you may easily simulate high-precision light curves.

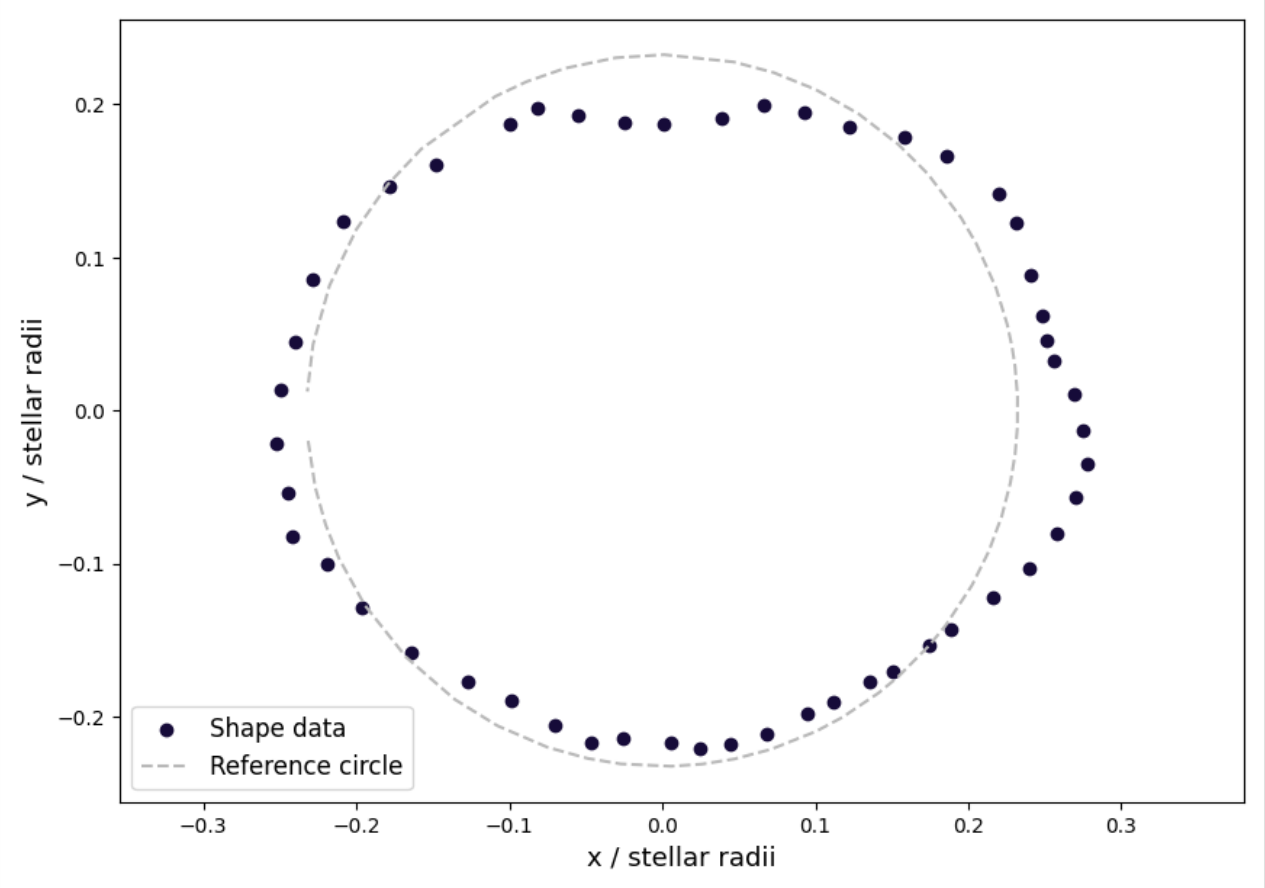
Let us start by loading some shape data. Here is some data made earlier, based on the transit of Phobos as viewed from the Perseverance rover on Mars.

[1]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt


r_data = np.array([ 0.25299594, 0.25049962, 0.2555872 , 0.24131962, 0.23499399, 0.
↪2281387 , 0.21770261, 0.21355986, 0.21756298, 0.22216417, 0.215748  , 0.21737008, 0.
↪22181932, 0.22220379, 0.22215942, 0.2191899 , 0.22060524, 0.22316191, 0.22795516, 0.
↪23203883, 0.23715028, 0.24817015, 0.26129245, 0.2701211 , 0.27631858, 0.279554  , 0.
↪27550068, 0.26970037, 0.25844681, 0.2551263 , 0.2564253 , 0.25649795, 0.2619272 , 0.
↪26137834, 0.24959362, 0.23925815, 0.2218478 , 0.21541933, 0.21006674, 0.1952882 , 0.
```

(continues on next page)

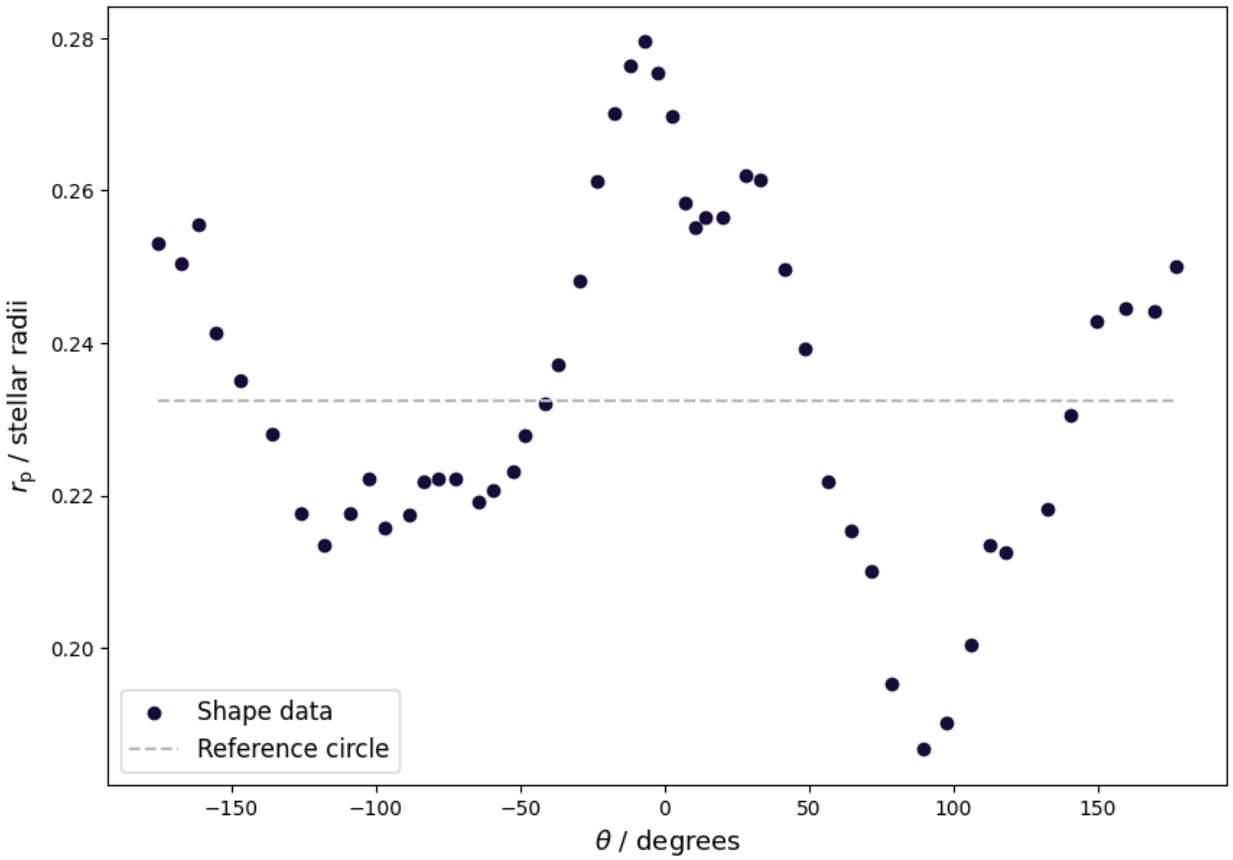
(continued from previous page)

```
↪18672611, 0.19014673, 0.20044651, 0.2134876 , 0.21243847, 0.21826262, 0.23046709, 0.
↪24290151, 0.24455098, 0.24425059, 0.25006916])
theta_data = np.array([-3.05626382, -2.92458806, -2.81512117, -2.71398938, -2.56361121, -
↪2.37500762, -2.19475882, -2.05340439, -1.89915229, -1.78311113, -1.68792003, -1.
↪54576688, -1.45830389, -1.36707739, -1.26004262, -1.12498761, -1.03931859, -0.91754355,
↪ -0.8460734 , -0.72241586, -0.64825102, -0.51378753, -0.40700302, -0.30187693, -0.
↪20512046, -0.1247507 , -0.0469304 , 0.04035871, 0.12437005, 0.17993957, 0.24397848, 0.
↪35182409, 0.48800481, 0.57248726, 0.72834884, 0.84510295, 0.98657143, 1.12488908, 1.
↪2518673 , 1.36829014, 1.56557413, 1.70233616, 1.84736078, 1.96211635, 2.0600249 , 2.
↪31436316, 2.4535093 , 2.60749337, 2.78292412, 2.95552791, 3.08844006])

plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.scatter(r_data * np.cos(theta_data), r_data * np.sin(theta_data),
            color=cm.inferno(0.1), label="Shape data")
plt.plot(0.2324 * np.cos(theta_data), 0.2324 * np.sin(theta_data),
         color="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



In polar coordinates:

[2]:
```
plt.figure(figsize=(10, 7))
plt.scatter(theta_data * 180. / np.pi, r_data,
            color=cm.inferno(0.1), label="Shape data")
plt.plot(theta_data * 180. / np.pi, 0.2324 * np.ones(theta_data.shape[0]),
         color="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



The Fourier parameters of Harmonica's transmission strings can be calculated from these data. This involves solving:

$$a_0 = \frac{1}{2\pi} \int_{-\pi}^{\pi} r(\theta)\, d\theta,$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} r(\theta) \cos(n\theta)\, d\theta,$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} r(\theta) \sin(n\theta)\, d\theta,$$

where each integrand is a generated interpolation function based on the data. Here let us compute up to $n = 15$.

[3]:
```
from scipy import interpolate, integrate
```

(continues on next page)

(continued from previous page)

```
n_harmonics = 15
transmission_string_params = []
for n in range(n_harmonics + 1):

    if n == 0:
        # a_0 term.
        integrand_cos0_func = interpolate.interp1d(
            theta_data, r_data,
            kind="cubic", bounds_error=False, fill_value="extrapolate")
        cn = integrate.quad(
            integrand_cos0_func, -np.pi, np.pi,
            epsabs=1.e-7, epsrel=1.e-7, limit=500)[0] / (2. * np.pi)
        transmission_string_params.append(cn)
    else:
        # a_n terms.
        integrand_cosn_func = interpolate.interp1d(
            theta_data, r_data * np.cos(n * theta_data),
            kind="cubic", bounds_error=False, fill_value="extrapolate")
        cn = integrate.quad(
            integrand_cosn_func, -np.pi, np.pi,
            epsabs=1.e-7, epsrel=1.e-7, limit=500)[0] / (1. * np.pi)
        transmission_string_params.append(cn)

        # b_n terms.
        integrand_sinn_func = interpolate.interp1d(
            theta_data, r_data * np.sin(n * theta_data),
            kind="cubic", bounds_error=False, fill_value="extrapolate")
        sn = integrate.quad(
            integrand_sinn_func, -np.pi, np.pi,
            epsabs=1.e-7, epsrel=1.e-7, limit=500)[0] / (1. * np.pi)
        transmission_string_params.append(sn)

transmission_string_params = np.array(transmission_string_params)
```

In practice the above method tends to produce numerical ringing when a large number of harmonics are computed. As an alternative, you may find using numpy's discrete Fourier transform is a more stable approach.

[4]:
```
theta_data_even_spacing = np.linspace(-np.pi, np.pi, 101, endpoint=True)
r_data_even_spacing = np.interp(
    theta_data_even_spacing, theta_data, r_data)  # Interpolate onto evenly-spaced grid.
r_data_even_spacing = np.concatenate(
    [r_data_even_spacing[theta_data_even_spacing >= 0.],
     r_data_even_spacing[theta_data_even_spacing < 0.]])  # Reorder for the interval 0 -␣
↪2*pi.

n_terms = 2 * n_harmonics + 1
n_data = len(r_data_even_spacing)

c = np.fft.fft(r_data_even_spacing) / n_data
r_coeffs = np.empty(n_data)
r_coeffs[0] = c[0].real
r_coeffs[1::2] = 2*c[1:n_data//2+1].real
```

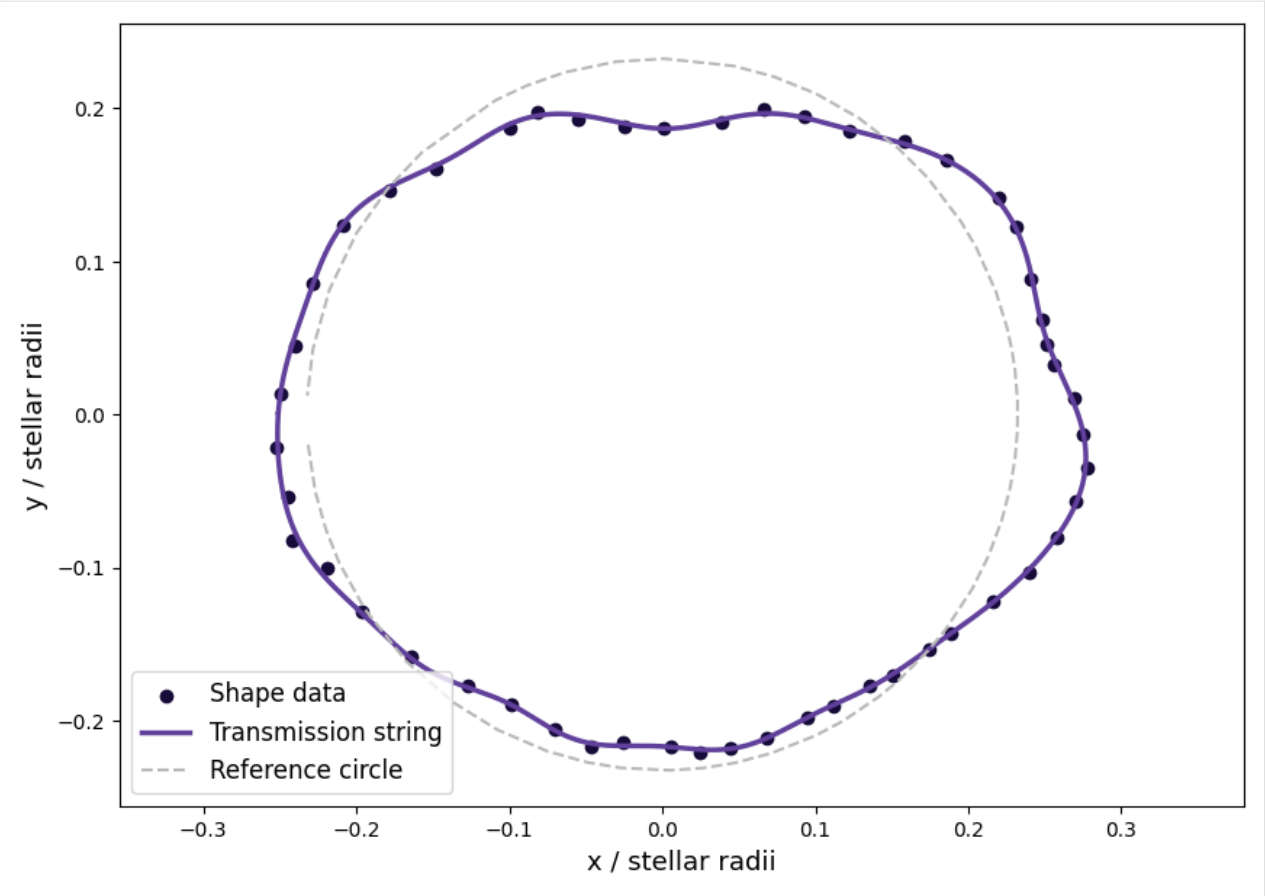
(continues on next page)

(continued from previous page)

```
r_coeffs[2::2] = -2*c[1:n_data//2+1].imag
transmission_string_params = r_coeffs[:n_terms]
```

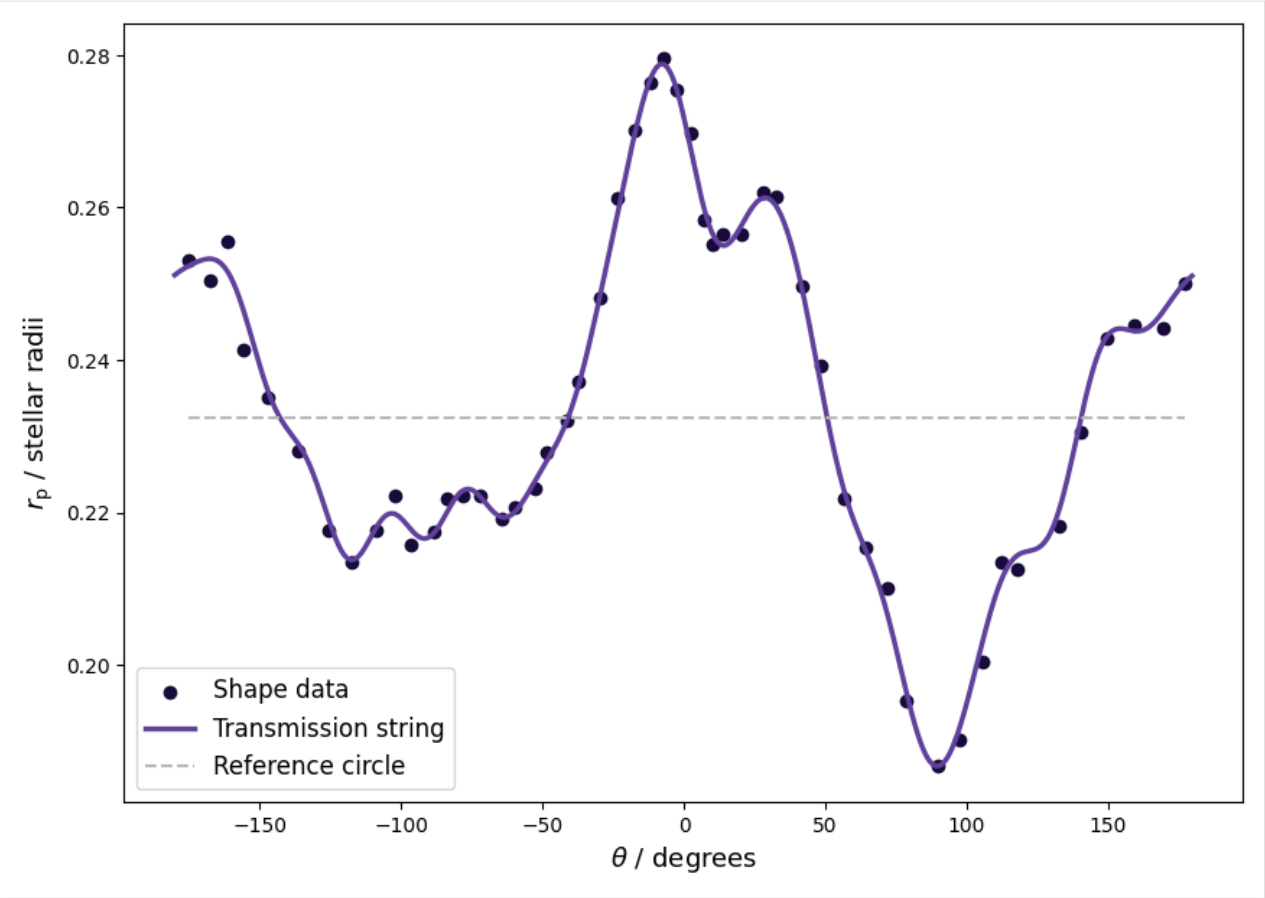
Now plot the resulting transmission string and check you have a good approximation of your input shape:

[5]:
```
from harmonica import HarmonicaTransit


ht = HarmonicaTransit()
ht.set_planet_transmission_string(transmission_string_params)
theta_model = np.linspace(-np.pi, np.pi, 1000, endpoint=False)

plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")
plt.scatter(r_data * np.cos(theta_data), r_data * np.sin(theta_data),
            color=cm.inferno(0.1), label="Shape data")
plt.plot(ht.get_planet_transmission_string(theta_model) * np.cos(theta_model),
         ht.get_planet_transmission_string(theta_model) * np.sin(theta_model),
         c=cm.Purples(0.8), lw=2.5, label="Transmission string")
plt.plot(0.2324 * np.cos(theta_data), 0.2324 * np.sin(theta_data),
         color="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```



In polar coordinates:

[6]:
```
plt.figure(figsize=(10, 7))
plt.scatter(theta_data * 180. / np.pi, r_data,
            color=cm.inferno(0.1), label="Shape data")
plt.plot(theta_model * 180. / np.pi, ht.get_planet_transmission_string(theta_model),
         c=cm.Purples(0.8), lw=2.5, label="Transmission string")
plt.plot(theta_data * 180. / np.pi, 0.2324 * np.ones(theta_data.shape[0]),
         color="#b9b9b9", ls="--", label="Reference circle")
plt.xlabel("$\\theta$ / degrees", fontsize=13)
plt.ylabel("$r_{\\rm{p}}$ / stellar radii", fontsize=13)
plt.legend(loc="lower left", fontsize=12)
plt.show()
```

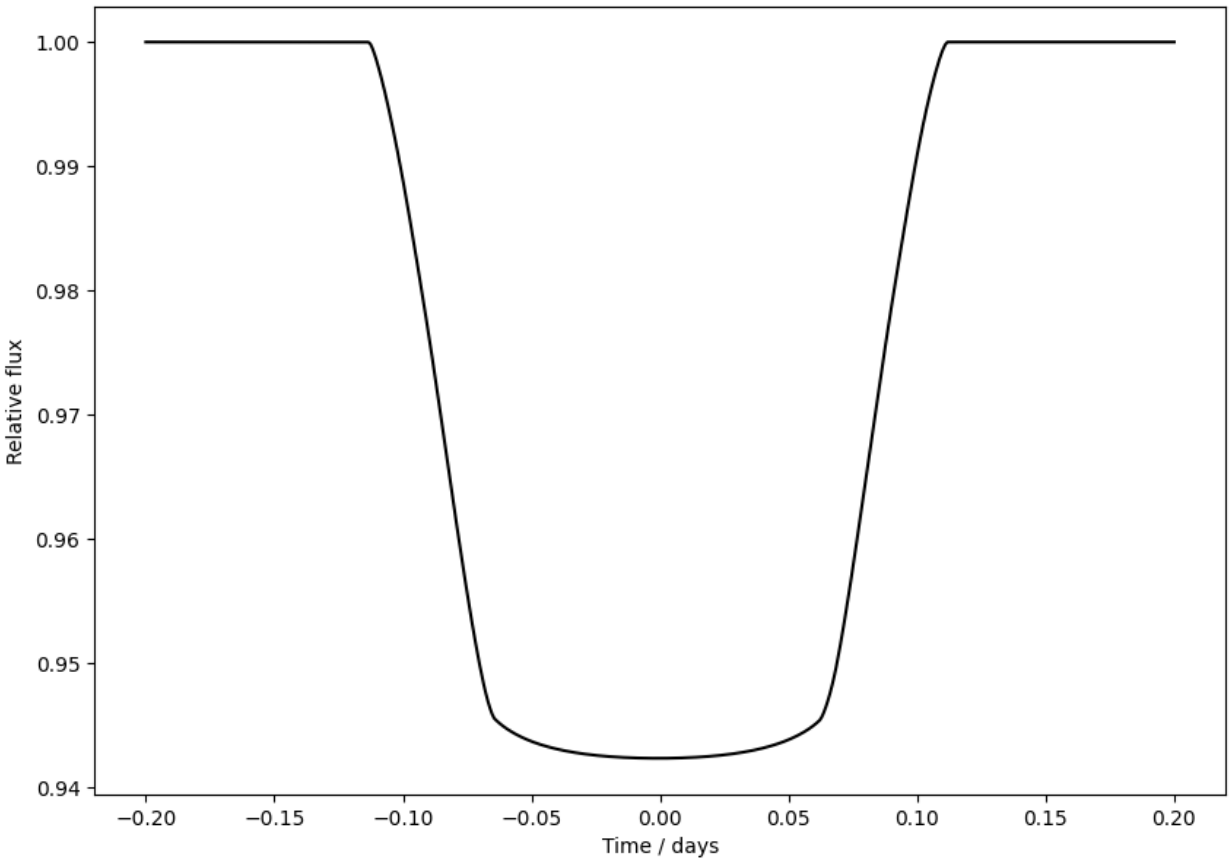


These parameters can now be used to generate transit light curves of your object. The only thing left to do is specify the orbit and stellar limb darkening.

Note

When using many Fourier coefficients you may need to increase the number of terms used to compute the light curve solutions, in order to generate a sufficiently precise result for your use case. This can easily be accomplished by manually setting pnl_c and pnl_e when you instantiate the HarmonicaTransit class. You can also check the estimated precision with the get_precision_estimate() method.

[7]:
```
times = np.linspace(-0.2, 0.2, 500)
ht = HarmonicaTransit(times, pnl_c=100, pnl_e=100)
ht.set_orbit(t0=0., period=4., a=7., inc=88. * np.pi / 180.)
ht.set_stellar_limb_darkening(u=np.array([0.074, 0.193]), limb_dark_law="quadratic")
ht.set_planet_transmission_string(transmission_string_params)
light_curve = ht.get_transit_light_curve()

plt.figure(figsize=(10, 7))
plt.plot(times, light_curve, c="#000000")
plt.xlabel('Time / days')
plt.ylabel('Relative flux')
plt.show()
```



4.7 Time-dependent shapes

In this tutorial we take a look at how to generate transit light curves where the shape of the transiting object is time-dependent. Let us start by creating a 2D array of transmission string parameters, where the parameter a_1 linearly varies between ingress and egress.

[1]:
```
import numpy as np


r_1 = np.array([0.1, -0.02, 0., 0.02, 0.])
r_2 = np.array([0.1, 0.02, 0., 0.02, 0.])
r = np.concatenate([np.linspace(r_1, r_1, 160),
                    np.linspace(r_1, r_2, 180),
                    np.linspace(r_2, r_2, 160)])
```

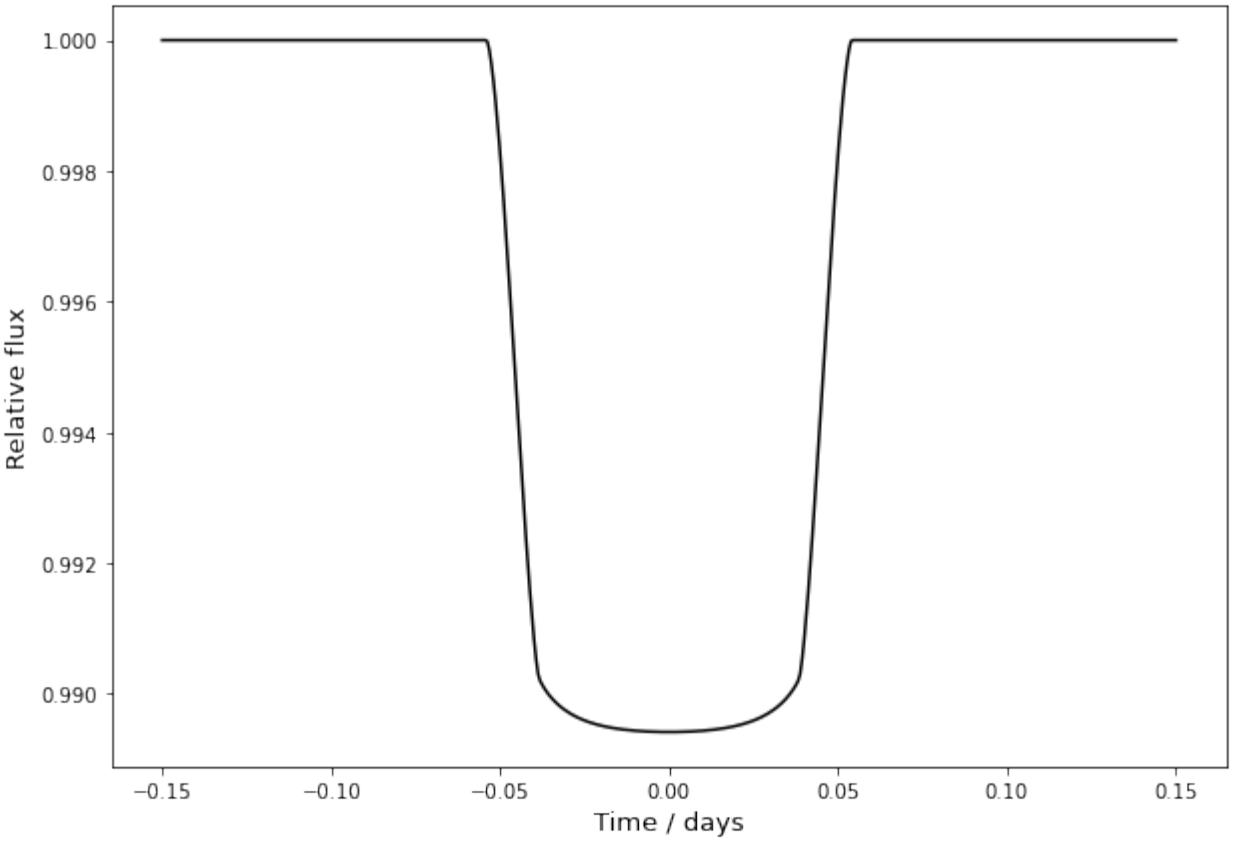
Harmonica can generate the transit light curve for this time-dependent transmission string as follows:

[2]:
```
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from harmonica import HarmonicaTransit
```

(continues on next page)

(continued from previous page)

```
times = np.linspace(-0.15, 0.15, 500)
ht = HarmonicaTransit(times)
ht.set_orbit(t0=0., period=4., a=11., inc=87. * np.pi / 180.)
ht.set_stellar_limb_darkening(np.array([0.027, 0.246]), limb_dark_law='quadratic')
ht.set_planet_transmission_string(r)
observed_fluxes = ht.get_transit_light_curve()

plt.figure(figsize=(10, 7))
plt.plot(times, observed_fluxes, c="#000000")
plt.xlabel('Time / days', fontsize=13)
plt.ylabel('Relative flux', fontsize=13)
plt.show()
```



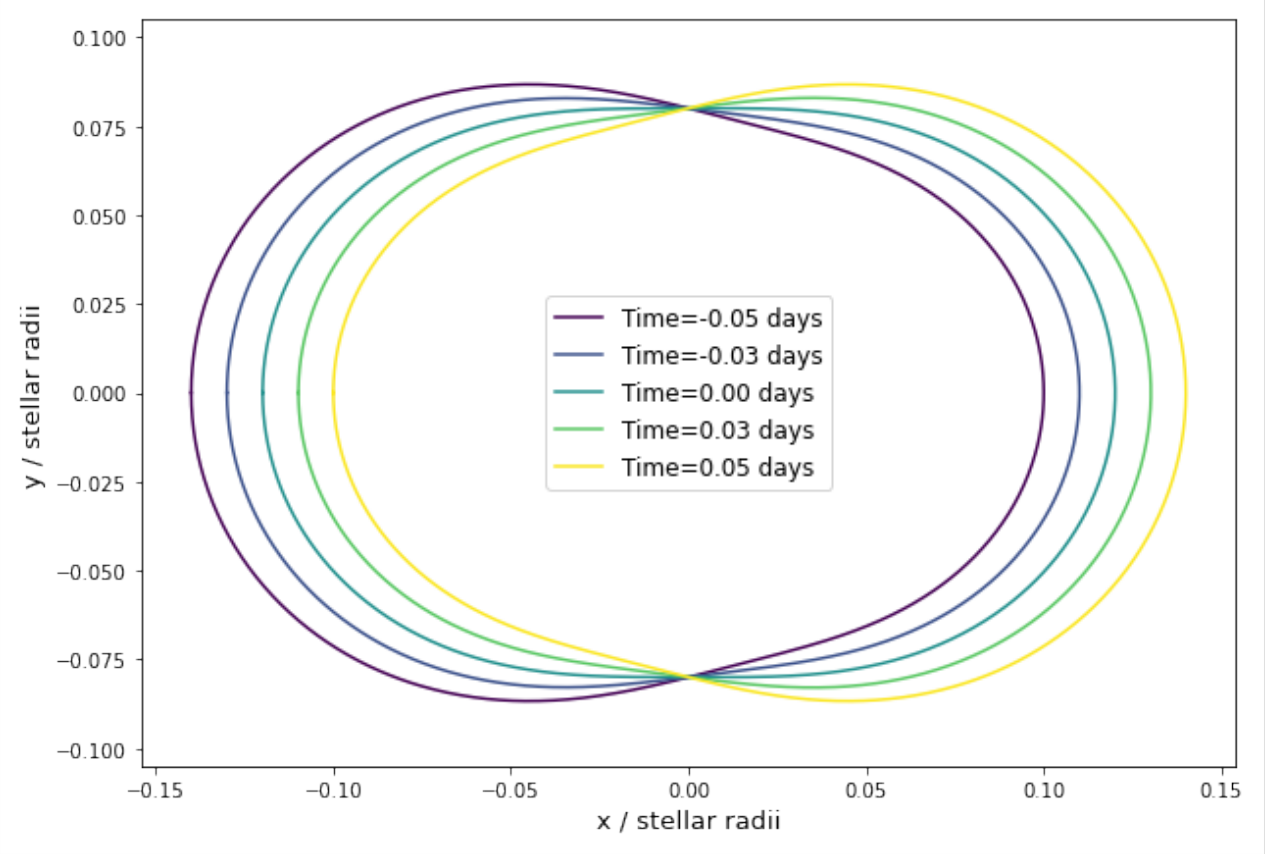
And you can visualise the transmission strings at any number of epochs too.

[3]:
```
theta = np.linspace(-np.pi, np.pi, 1000)
transmission_strings = ht.get_planet_transmission_string(theta)

plt.figure(figsize=(10, 7))
plt.gca().set_aspect("equal", "datalim")


for i, ts_idx in enumerate([160, 205, 250, 295, 340]):
    plt.plot(transmission_strings[ts_idx] * np.cos(theta),
             transmission_strings[ts_idx] * np.sin(theta),
```

(continues on next page)

(continued from previous page)

```
            c=cm.viridis(i/4.03),
            label="Time={:.2f} days".format(times[ts_idx]))

plt.xlabel("x / stellar radii", fontsize=13)
plt.ylabel("y / stellar radii", fontsize=13)
plt.legend(loc="center", fontsize=12)
plt.show()
```



CHAPTER

FIVE

API

5.1 Primary Interface

`HarmonicaTransit`([times, pnl_c, pnl_e])	Harmonica transit class.

5.2 Subpackages

5.2.1 harmonica.jax

harmonica.jax.**harmonica_transit_quad_ld**(*times*, *t0*, *period*, *a*, *inc*, *ecc=0.*, *omega=0.*, *u1=0.*, *u2=0.*, *r=jnp.array([0.1])*)

Harmonica transits with jax – quadratic limb darkening.

Parameters

- **times** (*ndarray*) – 1D array of model evaluation times [days].
- **t0** (*float*) – Time of transit [days].
- **period** (*float*) – Orbital period [days].
- **a** (*float*) – Semi-major axis [stellar radii].
- **inc** (*float*) – Orbital inclination [radians].
- **ecc** (*float, optional*) – Eccentricity [], 0 <= ecc < 1. Default=0.
- **omega** (*float, optional*) – Argument of periastron [radians]. Default=0.
- **u1** (*float.*) – Quadratic limb-darkening coefficient.
- **u2** (*float.*) – Quadratic limb-darkening coefficient.
- **r** (*ndarray*) – Transmission string coefficients. 1D array of N Fourier coefficients that specify the planet radius as a function of angle in the sky-plane. The length of r must be odd, and the final two coefficients must not both be zero.

$$r_{\rm p}(\theta) = \sum_{n=0}^{N} a_n \cos(n\theta) + \sum_{n=1}^{N} b_n \sin(n\theta)$$

The input array is given as r=[a_0, a_1, b_1, a_2, b_2,..].

Returns
Normalised transit light curve fluxes [].

Return type
array

harmonica.jax.**harmonica_transit_nonlinear_ld**(*times*, *t0*, *period*, *a*, *inc*, *ecc=0.*, *omega=0.*, *u1=0.*, *u2=0.*, *u3=0.*, *u4=0.*, *r=jnp.array([0.1])*)

Harmonica transits with jax – non-linear limb darkening.

Parameters

- **times** (*ndarray*) – 1D array of model evaluation times [days].
- **t0** (*float*) – Time of transit [days].
- **period** (*float*) – Orbital period [days].
- **a** (*float*) – Semi-major axis [stellar radii].
- **inc** (*float*) – Orbital inclination [radians].
- **ecc** (*float, optional*) – Eccentricity [], 0 <= ecc < 1. Default=0.
- **omega** (*float, optional*) – Argument of periastron [radians]. Default=0.
- **u1** (*float.*) – Non-linear limb-darkening coefficient.
- **u2** (*float.*) – Non-linear limb-darkening coefficient.
- **u3** (*float.*) – Non-linear limb-darkening coefficient.
- **u4** (*float.*) – Non-linear limb-darkening coefficient.
- **r** (*ndarray*) – Transmission string coefficients. 1D array of N Fourier coefficients that specify the planet radius as a function of angle in the sky-plane. The length of r must be odd, and the final two coefficients must not both be zero.

$$r_{\mathrm{p}}(\theta) = \sum_{n=0}^{N} a_n \cos(n\theta) + \sum_{n=1}^{N} b_n \sin(n\theta)$$

The input array is given as r=[a_0, a_1, b_1, a_2, b_2,..].

Returns
Normalised transit light curve fluxes [].

Return type
array

CHAPTER

SIX

CITATION

If you make use of Harmonica in your research, please cite Grant and Wakeford 2023:

```
@article{grant2022transmission,
  title={Transmission strings: a technique for spatially mapping exoplanet atmospheres␣
↪around their terminators},
  author={Grant, David and Wakeford, Hannah R},
  journal={Monthly Notices of the Royal Astronomical Society},
  volume={519},
  number={4},
  pages={5114--5127},
  year={2023},
  publisher={Oxford University Press}
}
```

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

Built by David Grant, Hannah Wakeford, and contributors. We are very happy to collaborate if you want to reach out.

If you make use of Harmonica in your research, see the *citation page* for info on how to cite this package. You can find other software from the Exoplanet Timeseries Characterisation (ExoTiC) ecosystem over on GitHub.

PYTHON MODULE INDEX

h

harmonica, 49
harmonica.jax, 49

INDEX

H

harmonica
    module, 49
harmonica.jax
    module, 49
harmonica_transit_nonlinear_ld() (*in module harmonica.jax*), 50
harmonica_transit_quad_ld() (*in module harmonica.jax*), 49

M

module
    harmonica, 49
    harmonica.jax, 49