magwell-test

July 26, 2022

1 Synopsis

Update 3: I think this implementation is correct. Can't find anything that would be wrong. Maybe send me an ATF STELLOPT run though, (I only have access to educational VMEC).

- New grid update somewhat fixes the symmetry bug I mentioned in June. But turning it off gives better matches to STELLOPT.
- Heliotron and dshape are near-exact matches to STELLOPT with symmetry off, close with symmetry on.
- All implementations of the Magnetic Well parameter between DESC and STELLOPT agree on crossing.
- Increasing the number of surfaces (rho resolution) on educational VMEC input from 256 to 1024 didn't change anything significantly. Educational VMEC's heliotron plot's crossing did shift left toward the point where the STELLOPT data and DESC magnetic wells cross.
- I had mentioned at the meeting that the DESC plots don't match educational vmec's plots. I don't think that means much anymore. Previously all the curves (desc, stellopt, edu-vmec) disagreed, causing me confusion. I couldn't single out one of them as correct or incorrect. Now that STELLOPT and DESC agree, I think we can claim educational-vmec is doing something wrong.
- I think what took me longer than I thought with this was:
  - I was learning basics of DESC code as it was my first project in DESC.
  - The STELLOPT code really looks like they implement the magnetic well parameter from Landreman equation 3.2. So when my implementation didn't match that, I was scratching my head wondering why. Based on these results it looks like STELLOPT secretly implements equation 4.19.
  - Didn't know edu-vmec was giving bad results (I wonder if this is also true for rotational transform stuff).
  - I had to account for sym / NFP / single surface bugs in the grid class to fix $d\theta$ * $d\zeta$ stuff.

Update 2: works with arbitrary grids and computes magnetic well parameter over multiple flux surfaces simultaneously Made use of the `surface_integrals` functionality I added while coding the new rotational transform function. The objective will still function in the same way as before if a single rho surface is given as input. However, computing magnetic well over multiple flux surfaces at once offers at least two benefits. It * Simplifies end user code. They don't have to create routines and loop through things. * Increases performance. I used to wait for this jupyter notebook to run for 30 minutes to get values to plot. Now it runs in 1 minute. * Please see

a bug in the grid class. The solution to this bug for grids with a single rho surface is simple. If the functions I added to `compute.utils` are used then the developer/user doesn't have to worry about it. However, for multiple rho surface grids, the fix isn't as simple. Until that fix is implemented, if the user wants magnetic well computed on a grid with multiple values of rho, for best accuracy they need to set NFP=1 in the grid provided to magnetic well.

Update 1: alternative version of magnetic well in Landreman paper added. It seems to have more responsive curves, and it has the same abrupt change as STELLOPT for HELIOTRON. It yields a perfect match for Heliotron.

1.1 the Bad

The curves produced by DESC and STELLOPT for the magnetic well parameter as a function of ρ look different. I think STELLOPT does not implement the well parameter we thought it does. We should just pay attention to the crossing.

1.2 the Good

The magnetic well parameter computed by DESC matches the sign of the magnetic well parameter computed by STELLOPT. This is technically all we need.

The volume computation is now 100% correct. The solution to the bug in the `grid.py` class works, which was necessary to correct volume and $\mathrm{dv}/\mathrm{d}\rho$. An explanation to the solution is given here. Might be useful in case we have run into this bug somewhere else in the code.

All the plots of DESC's intermediate quantities are consistent with each other. For every point below, I have confirmed their validity with code tests and math where applicable. We should be sure to preserve these signs if any modifications are made:

- The volume of the torus is exactly quadratic with ρ. The derivative of the volume enclosed by the flux surface is exactly linear. For the other stellarators, the volume and derivative are approximately quadratic and linear, respectively. The derivative computations also flatten when the volume curves linearize, and the opposite holds as well. All this matches our expectation that volume ~ $\rho 2$.
- The volume enclosed by the ρ=1 surface matches the volume of the stellarator as computed by `data["V"]` on a grid with a sequence of ρ values.
- The derivative of the thermal pressure average is always negative. This is good since our notion that the well parameter being positive implies stability is only valid when thermal pressure decreases away from the magnetic axis (ρ=0). Otherwise negative well parameters would imply stability.
- The derivative of the thermal pressure average looks very similar for stellarators with similar pressure profile inputs.
- Confidence in the new more complicated flux surface average
  - We now properly differentiate under the flux surface average operation. Pushing a volume derivative into a flux surface average requires more care than simply differentiating under the integral. We want to do this in the first place to take advantage of automatic differentiation of `B_r` and `p_r`.
  - Have shown the implementation has the desired additive homomorphism property; meaning average(a + b) = average(a) + average(b). Sanity check for correctness.

  - Confirmed `jnp.mean(f) / jnp.mean(`$\sqrt{g}$` )` always = `jnp.sum(dtdz * f) /` $\mathrm{dv}/\mathrm{d}\rho$. Helps confirm $\mathrm{dv}/\mathrm{d}\rho$ is correct among other things.
  - Proved that this new flux surface average is an identity operation for the average of any quantity constant over the surface.
- The thermal pressure is a flux surface function; it is constant on a surface of constant rho. Therefore, the average of thermal pressure over a flux surface should be an identity operation. This is reflected in the code. Also, we take advantage of the homomorphism discussed above to avoid an unnecessary flux surface average operation.
- The derivative of the magnetic pressure average is positive (negative) precisely when B2 average is increasing (decreasing). We should expect this since magnetic pressure ~ B2, and its average over a surface should likely share that relation.
- No jacobian sign errors. FYI: we always want them to be positive here.
- The volume derivative is not necessary for the magnetic well parameter for Landreman eq. 3.2. I replaced the flux surface label of volume with ρ in that magnetic well formula. The output of the magnetic well plot (this version is in purple color) is the same (as the volume derivative version in blue), ignoring the scale. Plots of both are included.

1.3 the Pretty

This notebook should make it easy to inspect intermediate variables in the `MagneticWell` class as well as compute quantities for all the stellarators in a concise manner. Add an entry to the dictionary and everything is automatic. Note this notebook should run pretty quickly. If the plots for many stellarators at once are requested, increase the memory given to jupyter notebook and/or compute on 3 stellarators at a time instead of 5+, and/or reduce plot dpi.

~~Given this, if the equilibrium solutions DESC and STELLOPT are computing magnetic well parameters on are indeed the same, I think any issue must be in the automatic differentiation of `B_r` and `p_r`.~~

[1]:
```
import sys
import os

sys.path.insert(0, os.path.abspath("."))
sys.path.append(os.path.abspath("../"))
```

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

from netCDF4 import Dataset
import pickle
import scipy.io as sio
import desc.io

from desc.compute import compute_geometry, data_index
from desc.equilibrium import Equilibrium
from desc.grid import LinearGrid
from desc.objectives import MagneticWell
```

```
from desc.plotting import plot_surfaces
from desc.transform import Transform
```

```
DESC version 0.5.1+104.g2946310.dirty, using JAX backend, jax version=0.2.25,
jaxlib version=0.1.76, dtype=float64
Using device: CPU, with 6.52 GB available memory
```

[3]:
```
np.set_printoptions(precision=4, floatmode="fixed")
```

[4]:
```
class MagneticWellVisual:
    """
    To print and plot more (less) quantities, add (remove) them to the dict
    returned from MagneticWell.compute().
    Everything else is automatic.
    """

    def __init__(
        self,
        name,
        eq=None,
        use_pickle=False,
        rho=np.linspace(1 / 64, 1, 64),
    ):
        """
        Make a MagneticWellVisual from either the provided equilibrium
        or the final equilibrium loaded from the output.h5 solution.

        Parameters
        ----------
        name : str
            Name of the equilibrium.
        eq : Equilibrium
            The equilibrium.
        use_pickle : bool
            True to use compute values from a previous run stored in pickle␣
 ↪file; False to recompute.
        rho : ndarray
            The flux surface values to compute the magnetic well on.
        """
        self.name = name
        if eq is None:
            eq = desc.io.load(load_from="../examples/DESC/" + name + "_output.
 ↪h5")[-1]
        self.eq = eq

        if use_pickle:
            with open(name + " DESC magwell.pkl", "rb") as file:
```

```
                self.st = pickle.load(file)
            self.rho = self.st["volume"][0]
            self.has_values = True
        else:
            # values are tuples of (x[i], y[i]) of plotable data
            # y points are cached when compute_plot() is called
            self.st = dict()
            try:
                mat = sio.loadmat(name + "_magwell.mat")
                self.st["0. STELLOPT Magnetic Well"] = mat["rho"],␣
↪mat["magwell"]
                rho = mat["rho"]
            except FileNotFoundError:
                pass

            try:
                f = Dataset("../iota/edu-vmec/input-iota/wout_" + name + ".nc")
                # print(f.variables.keys())
                vmec_rho = np.sqrt(
                    f.variables["phi"] / np.array(f.variables["phi"])[-1]
                )
                vmec_well = np.asarray(f.variables["DWell"])
                self.st["0. VMEC Magnetic Well"] = vmec_rho, vmec_well
            except FileNotFoundError:
                pass

            self.rho = rho
            self.has_values = False

    def print_values(self, grid=None):
        """
        Parameters
        ----------
        grid : LinearGrid
            The grid used for a MagneticWell objective.
            The default grid=None uses the rho=1 flux surface.
        """
        print(self.name)
        print(self.eq)
        mw = MagneticWell(eq=self.eq, grid=grid)
        print("grid.spacing(dr,dt,dz)", mw.grid.spacing[0])
        m = mw.compute(
            self.eq.R_lmn,
            self.eq.Z_lmn,
            self.eq.L_lmn,
            self.eq.p_l,
            self.eq.i_l,
```

```
            self.eq.c_l,
            self.eq.Psi,
        )
        for key, val in sorted(m.items()):
            print(key, val)
        self._print_data_V()
        print()

    def _print_data_V(self):
        """
        Print the volume of the stellarator device as computed by data["V"].
        Should match V surface integral when the default grid with rho = 1
        is used to construct the MagneticWell() object.
        """
        grid = LinearGrid(
            L=self.eq.L_grid,
            M=self.eq.M_grid,
            N=self.eq.N_grid,
            NFP=self.eq.NFP,
            sym=self.eq.sym,
        )
        R_transform = Transform(
            grid, self.eq.R_basis, derivs=data_index["sqrt(g)"]["R_derivs"],␣
↪build=True
        )
        Z_transform = Transform(
            grid, self.eq.Z_basis, derivs=data_index["sqrt(g)"]["R_derivs"],␣
↪build=True
        )
        data = compute_geometry(self.eq.R_lmn, self.eq.Z_lmn, R_transform,␣
↪Z_transform)
        print('data["V"]', data["V"])

    # same as other compute_plot but builds plot through computing on single␣
↪rho surfaces
    # def compute_plot(self):
    #     """Compute and cache MagneticWell.compute() values."""
    #     if self.has_values:
    #         return
    #
    #     for i in range(len(self.rho)):
    #         m = MagneticWell(
    #             eq=self.eq,
    #             grid=LinearGrid(
    #                 M=self.eq.M_grid,
    #                 N=self.eq.N_grid,
    #                 NFP=self.eq.NFP,
```

```
    #                 sym=self.eq.sym,
    #                 rho=self.rho[i],
    #             ),
    #         ).compute(
    #             self.eq.R_lmn,
    #             self.eq.Z_lmn,
    #             self.eq.L_lmn,
    #             self.eq.p_l,
    #             self.eq.i_l,
    #             self.eq.c_l,
    #             self.eq.Psi,
    #         )
    #         for key, val in m.items():
    #             self.st.setdefault(key, (self.rho, np.empty_like(self.
↪rho)))[1][i] = val
    #     self.has_values = True

    def compute_plot(self):
        """Compute and cache MagneticWell.compute() values."""
        if self.has_values:
            return

        m = MagneticWell(
            eq=self.eq,
            grid=LinearGrid(
                M=self.eq.M_grid,
                N=self.eq.N_grid,
                NFP=1,  # required for accuracy when grid.num_rho != 1
                sym=False,  # off for better matches to STELLOPT. Doesn't␣
↪affect crossings.
                rho=self.rho,
            ),
        ).compute(
            self.eq.R_lmn,
            self.eq.Z_lmn,
            self.eq.L_lmn,
            self.eq.p_l,
            self.eq.i_l,
            self.eq.c_l,
            self.eq.Psi,
        )
        for key, val in m.items():
            self.st[key] = self.rho, val
        self.has_values = True

    def save(self):
        """Save computed values to pickle file."""
```

```
        with open(self.name + " DESC magwell.pkl", "wb") as file:
            pickle.dump(self.st, file)

    @staticmethod
    def _color(key):
        """Return the correct plot color."""
        if "0. STELLOPT" in key:
            return "tab:green"
        if "0. VMEC" in key:
            return "tab:brown"
        if "1." in key:
            return "tab:orange"
        if "2." in key:
            return "tab:purple"
        return "tab:blue"

    def plot(self, dpi=150):
        """Plot all quantities from compute_plot() in st."""
        fig, ax = plt.subplots(
            nrows=4,
            ncols=(len(self.st) + 3) // 4,
            figsize=(len(self.st) * 2, 4 * 5),
            dpi=dpi,
        )
        ax = ax.flatten()

        for i, e in enumerate(sorted(self.st.items())):
            key, val = e
            x, y = val

            # edu vmec usually blows up near rho=1. messes up plot scaling if␣
↪included
            if key == "0. VMEC Magnetic Well":
                x = x[:-3]
                y = y[:-3]

            color = MagneticWellVisual._color(key)
            ax[i].scatter(x, y, color=color, s=(5 if len(x) > 128 else 10))
            ax[i].plot(x, y, color=color)
            ax[i].set(xlabel=r"$\rho$", ylabel=key, title=self.name,␣
↪facecolor="white")
            if "Magnetic Well" in key or "Mercier" in key:
                if np.any(~np.isclose(y, 0)):
                    ax[i].set(yscale="symlog")
                ax[i].axhline(color="tab:red")
            ax[i].grid()
```

```
    def plot_magnetic_wells(self, dpi=300):
        """Plots the magnetic wells together on the same scale."""
        fig, ax = plt.subplots(dpi=dpi)

        # computed on other codes
        if (key := "0. STELLOPT Magnetic Well") in self.st:
            color = MagneticWellVisual._color(key)
            rho, well = self.st[key]
            ax.plot(rho, well, color=color, label="STELLOPT")
            ax.scatter(rho, well, color=color, s=1)
        if (key := "0. VMEC Magnetic Well") in self.st:
            color = MagneticWellVisual._color(key)
            rho, well = self.st[key]
            # edu vmec usually blows up near rho=1. messes up plot scaling if␣
↪included
            rho, well = rho[:-3], well[:-3]
            ax.plot(rho, well, color=color, label="edu-vmec")
            ax.scatter(rho, well, color=color, s=1)

        # computed with DESC
        well = self.st["1. DESC Magnetic Well: M. Landreman eq. 4.19"][1]
        color = MagneticWellVisual._color("1.")
        ax.plot(
            self.rho,
            well,
            color=color,
            label="DESC M. Landreman eq. 4.19",
        )
        ax.scatter(self.rho, well, color=color, s=1)

        well = self.st["2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho *␣
↪d/drho"][
            1
        ]
        color = MagneticWellVisual._color("2.")
        ax.plot(
            self.rho,
            well,
            color=color,
            label="DESC M. Landreman eq. 3.2 with rho * d/drho",
        )
        ax.scatter(self.rho, well, color=color, s=1)

        well = self.st["3. DESC Magnetic Well: M. Landreman eq. 3.2"][1]
        color = MagneticWellVisual._color("3.")
        ax.plot(self.rho, well, color=color, label="DESC M. Landreman eq. 3.2")
        ax.scatter(self.rho, well, color=color, s=1)
```

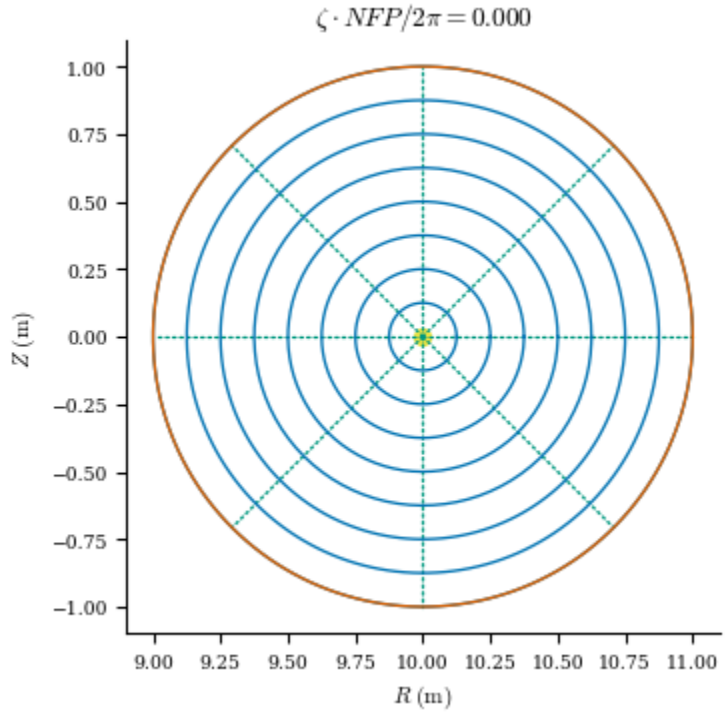
```
        ax.set(
            yscale="symlog",
            xlabel=r"$\rho$",
            ylabel="magnetic well",
            title=self.name,
            facecolor="white",
        )
        ax.axhline(color="tab:red")
        ax.grid()
        fig.legend(fontsize="xx-small")
        fig.savefig(self.name + " magnetic wells sym false.png")
```

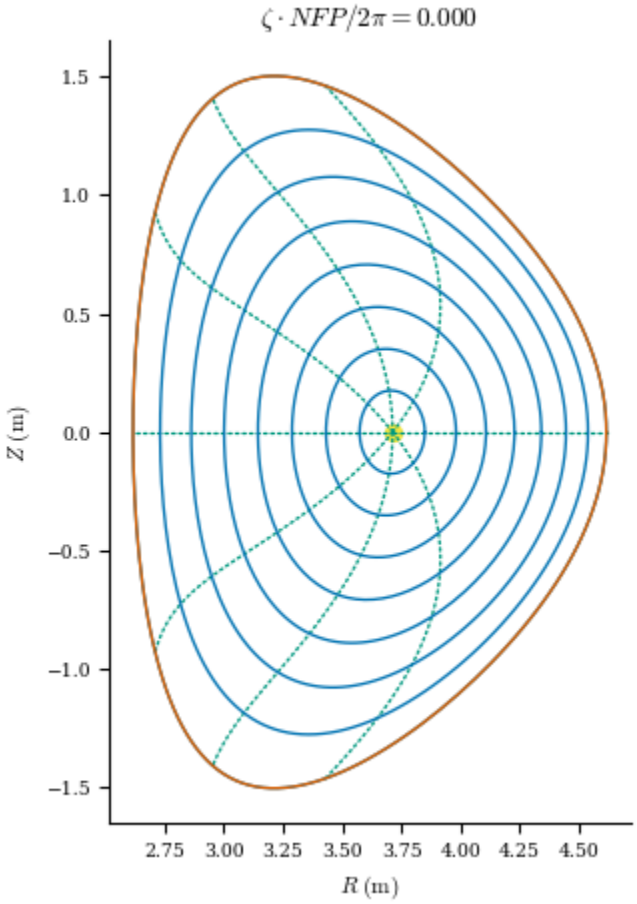
[5]:
```
torus = MagneticWellVisual("torus", Equilibrium())
dshape = MagneticWellVisual("DSHAPE")
heliotron = MagneticWellVisual("HELIOTRON")
atf = MagneticWellVisual("ATF")
axisym = MagneticWellVisual("AXISYM")
stellarators = (torus, dshape, heliotron, atf, axisym)
```

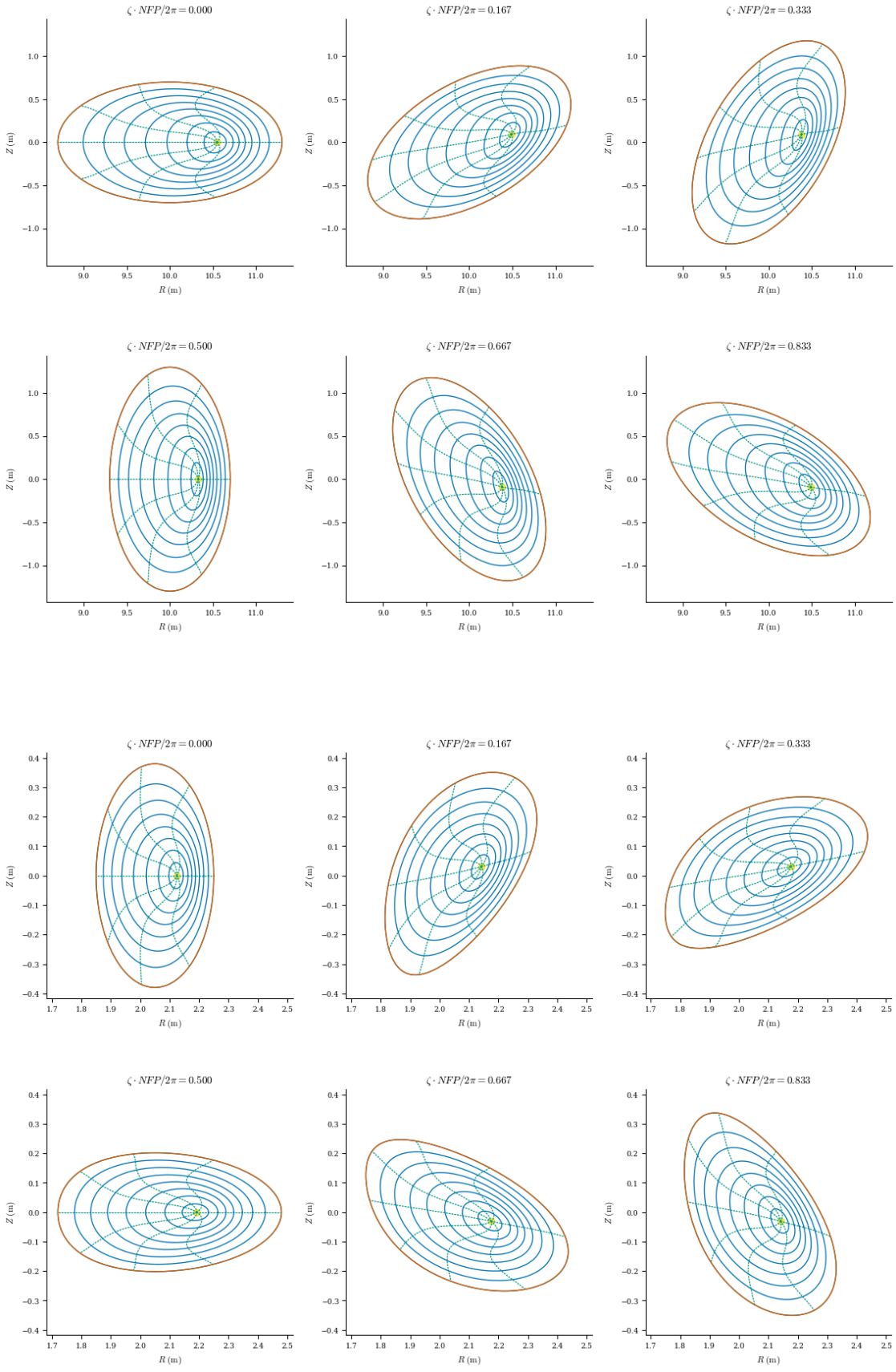
```
/home/kaya/Documents/edu/pton/plasma/DESC/desc/configuration.py:344:
UserWarning: Must specify either iota or current. Using default profile of
iota=0.
  warnings.warn(
/home/kaya/Documents/edu/pton/plasma/DESC/desc/io/hdf5_io.py:108:
RuntimeWarning: Save attribute '_current' was not loaded.
  warnings.warn(
```

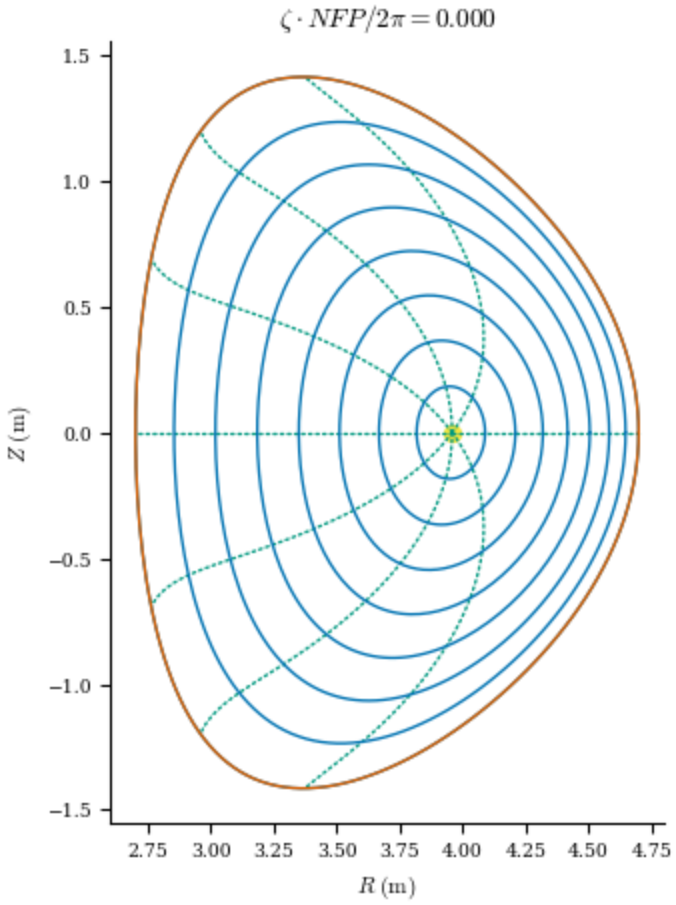
2 Equilibrium plots

[6]:
```
# to make sure equilibrium were solved correctly on my computer
for s in stellarators:
    plot_surfaces(s.eq)
```


$\zeta \cdot NFP/2\pi = 0.000$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Z (m)
9.00
9.25
9.50
9.75
10.00
10.25
10.50
10.75
11.00
R (m)


$\zeta \cdot NFP/2\pi = 0.000$
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Z (m)
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50
R (m)


$\zeta \cdot NFP/2\pi = 0.000$
$\zeta \cdot NFP/2\pi = 0.167$
$\zeta \cdot NFP/2\pi = 0.333$
$\zeta \cdot NFP/2\pi = 0.500$
$\zeta \cdot NFP/2\pi = 0.667$
$\zeta \cdot NFP/2\pi = 0.833$
$\zeta \cdot NFP/2\pi = 0.000$
$\zeta \cdot NFP/2\pi = 0.167$
$\zeta \cdot NFP/2\pi = 0.333$
$\zeta \cdot NFP/2\pi = 0.500$
$\zeta \cdot NFP/2\pi = 0.667$
$\zeta \cdot NFP/2\pi = 0.833$
R (m)
Z (m)



3 $\rho = 1$ flux surface

[7]:
```
for s in stellarators:
    s.print_values()
```

```
torus
Equilibrium at 0x7fb3e852d7f0 (L=1, M=1, N=0, NFP=1, sym=False,
spectral_indexing=ansi)
Precomputing transforms
grid.spacing(dr,dt,dz) [1.0000 1.2566 6.2832]
1. DESC Magnetic Well: M. Landreman eq. 4.19 [0.0000]
2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho * d/drho [-3.2159e-16]
3. DESC Magnetic Well: M. Landreman eq. 3.2 [-1.6080e-16]
B square average [0.1013]
Dcurr [0.0000]
```

```
Dgeod [-0.0000]
Dshear [0.0000]
Mercier [0.0000]
d(magnetic pressure average)/drho [-3.2584e-17]
d(thermal pressure)/drho [0.0000]
d(total pressure average)/drho [-3.2584e-17]
d(total pressure average)/dvolume [-8.2537e-20]
d^2(volume)/d(rho)^2 [394.7842]
dtdz [39.4784]
dvolume/drho [394.7842]
volume [197.3921]
data["V"] 197.39208802178715

DSHAPE
Equilibrium at 0x7fb39aedac70 (L=26, M=13, N=0, NFP=1.0, sym=1,
spectral_indexing=fringe)
Precomputing transforms
grid.spacing(dr,dt,dz) [1.2599 0.1466 7.9163]
1. DESC Magnetic Well: M. Landreman eq. 4.19 [0.0000]
2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho * d/drho [0.0629]
3. DESC Magnetic Well: M. Landreman eq. 3.2 [0.0369]
B square average [0.0575]
Dcurr [-0.0008]
Dgeod [-1.7297e-06]
Dshear [0.1122]
Mercier [0.1115]
d(magnetic pressure average)/drho [0.0036]
d(thermal pressure)/drho [0.0000]
d(total pressure average)/drho [0.0036]
d(total pressure average)/dvolume [2.1337e-05]
d^2(volume)/d(rho)^2 [305.9274]
dtdz [39.4784]
dvolume/drho [169.7811]
volume [99.4570]
data["V"] 99.1767790025823

HELIOTRON
Equilibrium at 0x7fb39aee7580 (L=24, M=12, N=3, NFP=19.0, sym=1,
spectral_indexing=fringe)
Precomputing transforms
grid.spacing(dr,dt,dz) [3.3620 0.5559 0.0855]
1. DESC Magnetic Well: M. Landreman eq. 4.19 [0.0000]
2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho * d/drho [-2.2638]
3. DESC Magnetic Well: M. Landreman eq. 3.2 [-0.9240]
B square average [0.1253]
Dcurr [0.1825]
Dgeod [-0.1342]
Dshear [0.5625]
```

```
Mercier [0.6108]
d(magnetic pressure average)/drho [-0.2837]
d(thermal pressure)/drho [0.0000]
d(total pressure average)/drho [-0.2837]
d(total pressure average)/dvolume [-0.0006]
d^2(volume)/d(rho)^2 [1483.7368]
dtdz [39.4784]
dvolume/drho [440.0806]
volume [179.6268]
data["V"] 180.78725668914205

ATF
Equilibrium at 0x7fb39aac7c70 (L=24, M=12, N=4, NFP=12.0, sym=1,
spectral_indexing=fringe)
Precomputing transforms
grid.spacing(dr,dt,dz) [2.8845 0.5331 0.1162]
1. DESC Magnetic Well: M. Landreman eq. 4.19 [0.0000]
2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho * d/drho [-1.3759]
3. DESC Magnetic Well: M. Landreman eq. 3.2 [-0.5825]
B square average [16.7004]
Dcurr [-0.0176]
Dgeod [-0.0315]
Dshear [0.1056]
Mercier [0.0566]
d(magnetic pressure average)/drho [-22.9789]
d(thermal pressure)/drho [0.0000]
d(total pressure average)/drho [-22.9789]
d(total pressure average)/dvolume [-3.1056]
d^2(volume)/d(rho)^2 [18.0245]
dtdz [39.4784]
dvolume/drho [7.3991]
volume [3.1325]
data["V"] 3.1505967937703065

AXISYM
Equilibrium at 0x7fb39afb3cd0 (L=40, M=20, N=0, NFP=1.0, sym=1,
spectral_indexing=fringe)
Precomputing transforms
grid.spacing(dr,dt,dz) [1.2599 0.0965 7.9163]
1. DESC Magnetic Well: M. Landreman eq. 4.19 [0.0000]
2. DESC Magnetic Well: M. Landreman eq. 3.2 with rho * d/drho [-0.1511]
3. DESC Magnetic Well: M. Landreman eq. 3.2 [-0.0958]
B square average [0.0683]
Dcurr [0.0003]
Dgeod [-5.1028e-06]
Dshear [0.0992]
Mercier [0.0995]
d(magnetic pressure average)/drho [-0.0103]
```

```
d(thermal pressure)/drho [0.0000]
d(total pressure average)/drho [-0.0103]
d(total pressure average)/dvolume [-6.6500e-05]
d^2(volume)/d(rho)^2 [493.9344]
dtdz [39.4784]
dvolume/drho [155.3259]
volume [98.4592]
data["V"] 98.19957214489737
```

4 Magnetic well plots

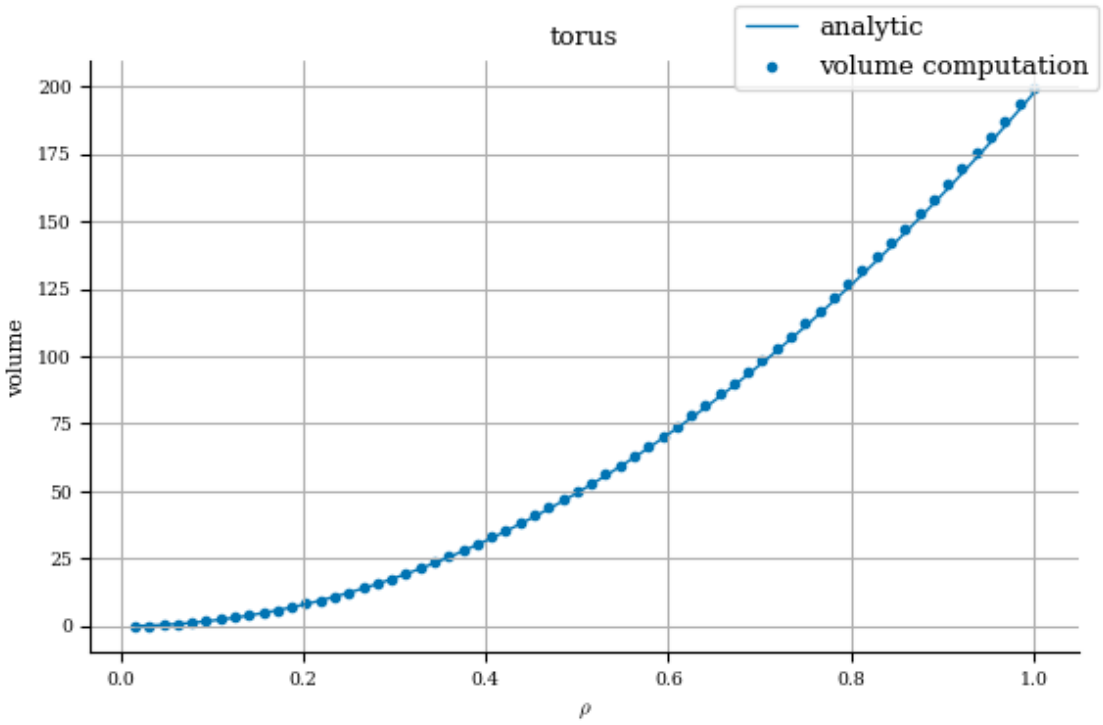
Although the symmetry bug is fixed, forcing it off results in a closer match between the STELLOPT data and DESC 4.19 well. Here symmetry is not forced to false because I think its close enough.

[8]:
```
for s in stellarators:
    s.compute_plot()
```

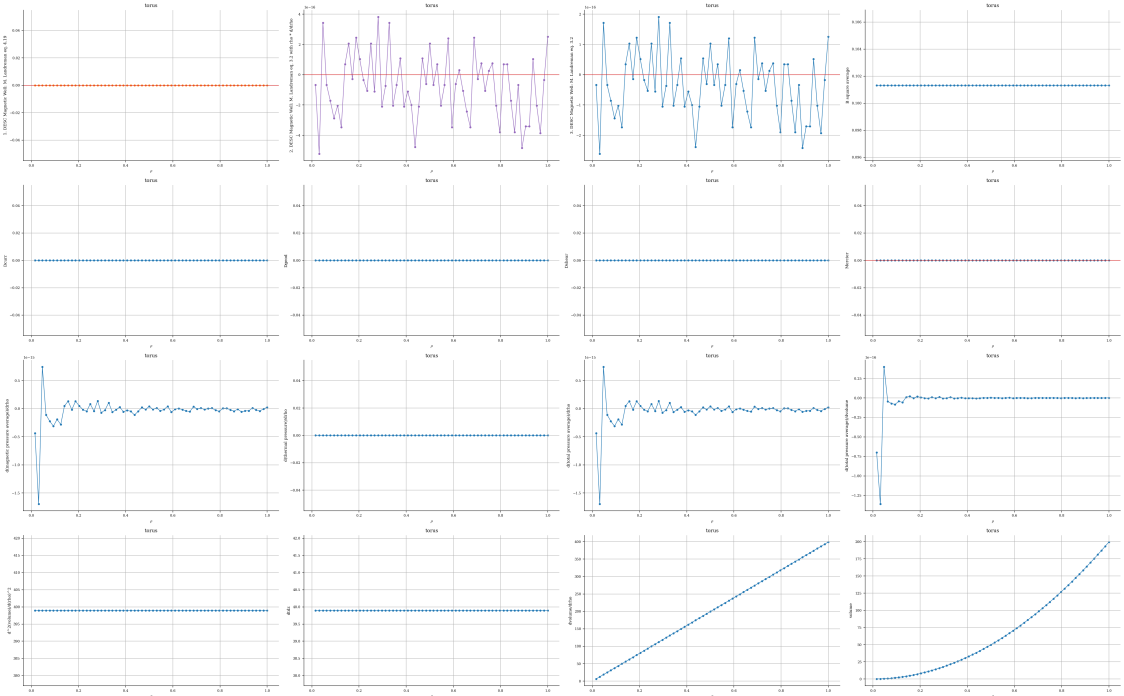
```
Precomputing transforms
Precomputing transforms
Precomputing transforms
Precomputing transforms
Precomputing transforms
```

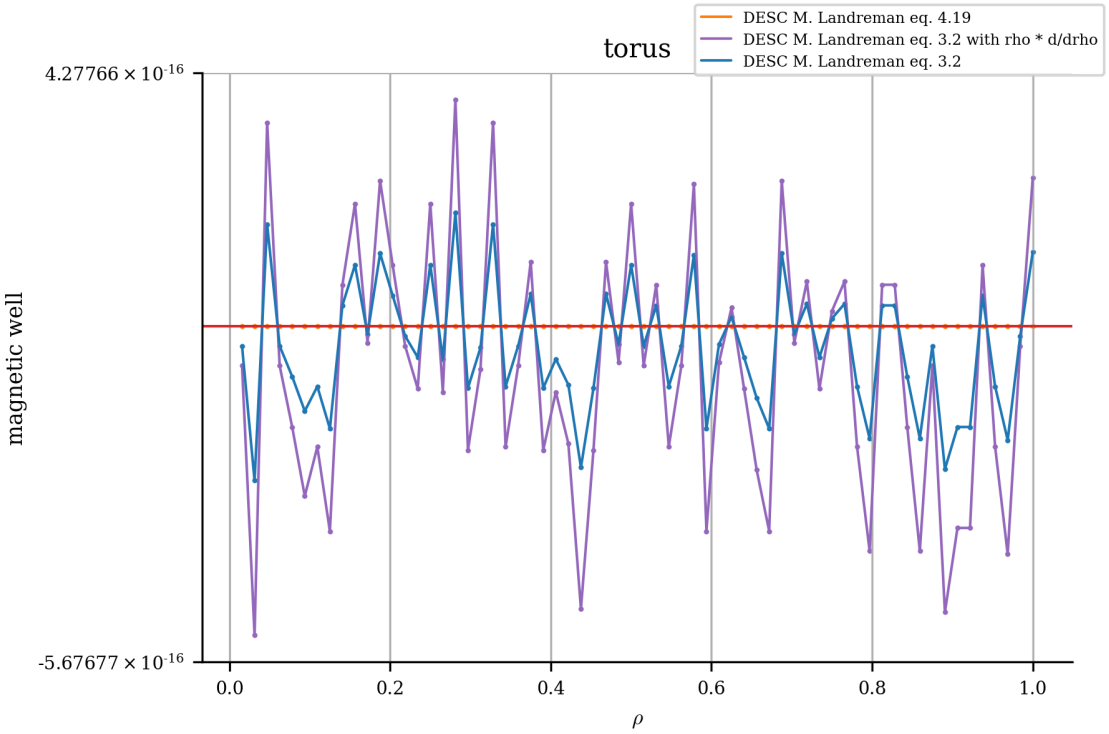
[9]:
```
# compare volume computation to known analytic case for torus
fig, ax = plt.subplots()
y = "volume"
rho, volume = torus.st[y]
ax.scatter(rho, volume, label=y + " computation", s=10)
ax.plot(rho, 20 * (np.pi * rho) ** 2, label="analytic")
ax.set(xlabel=r"$\rho$", ylabel=y, title=torus.name)
ax.grid()
fig.legend()
```

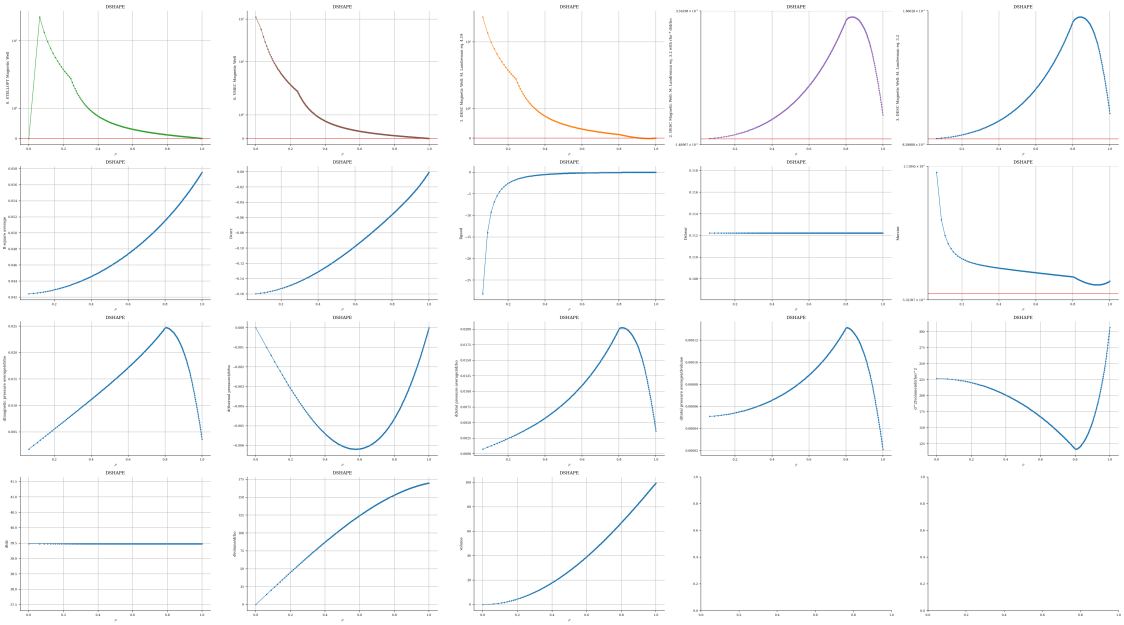
[9]:
```
<matplotlib.legend.Legend at 0x7fb3980b47f0>
```

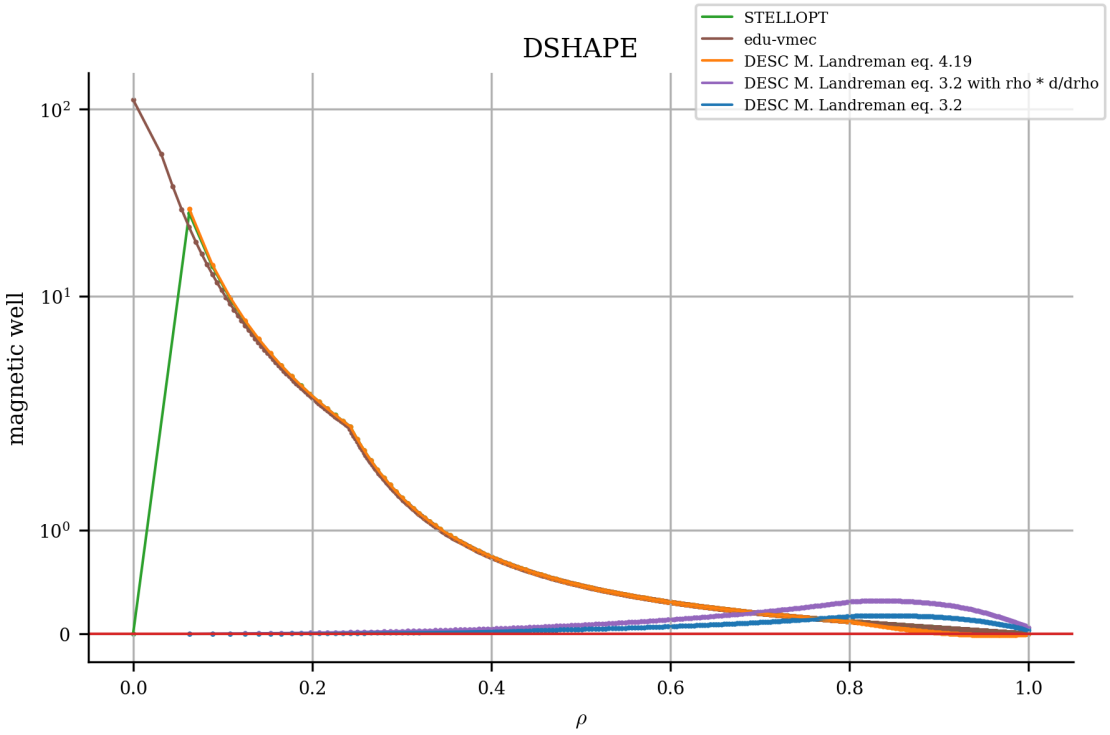


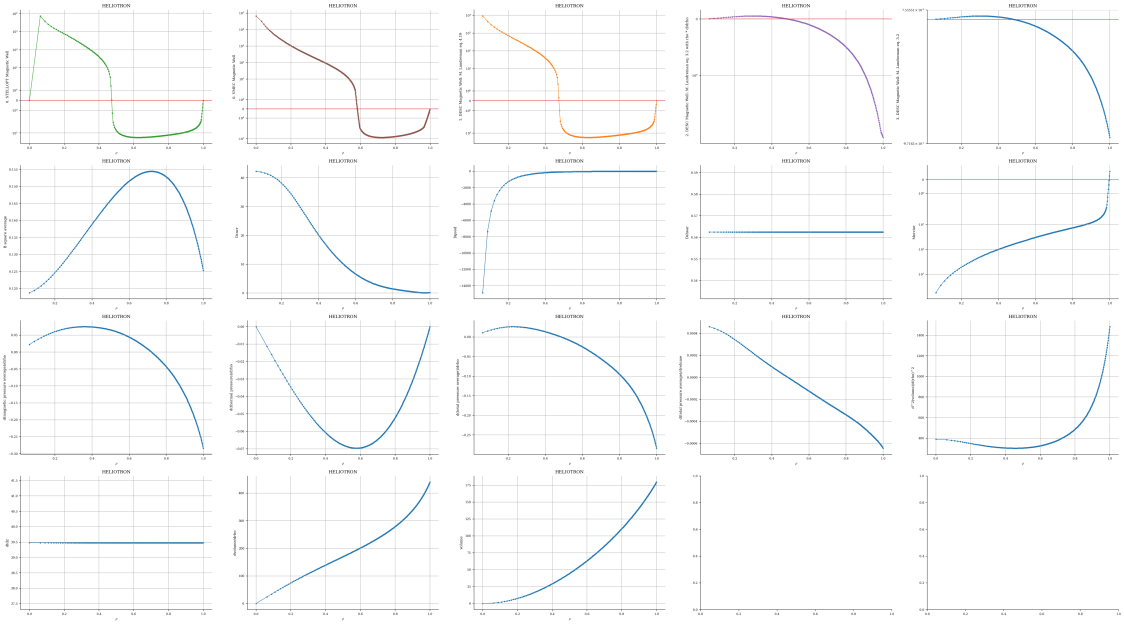
[10]:
```
for s in stellarators:
    s.plot()
    s.plot_magnetic_wells()
```

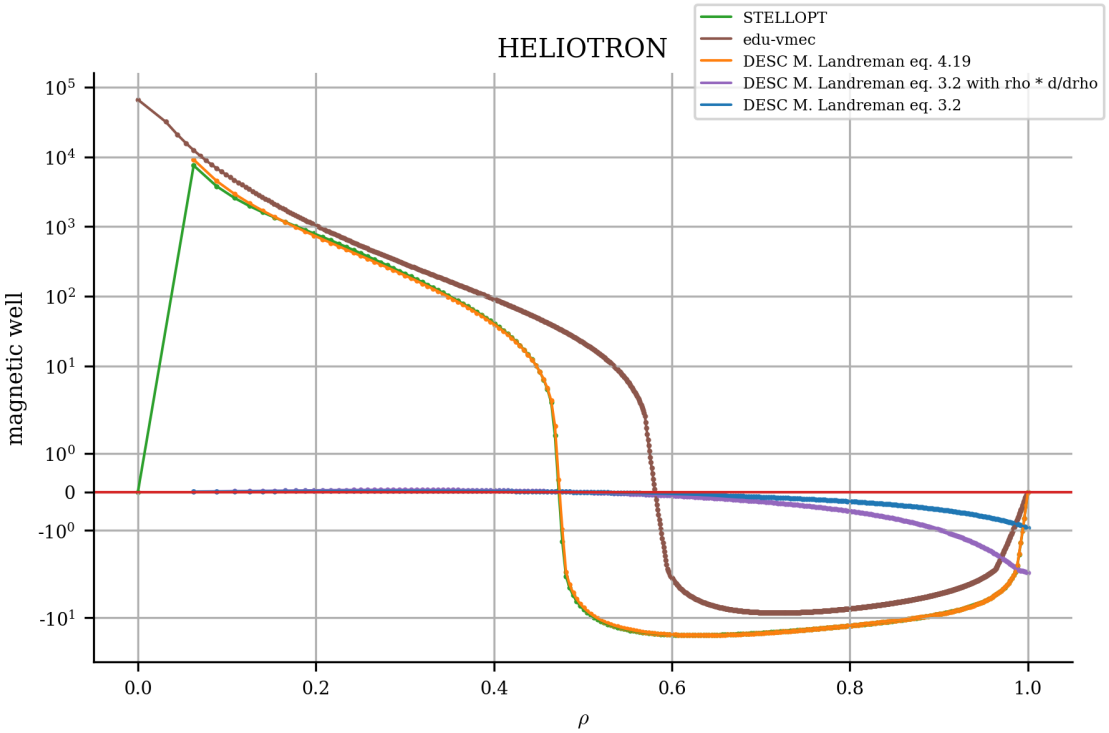


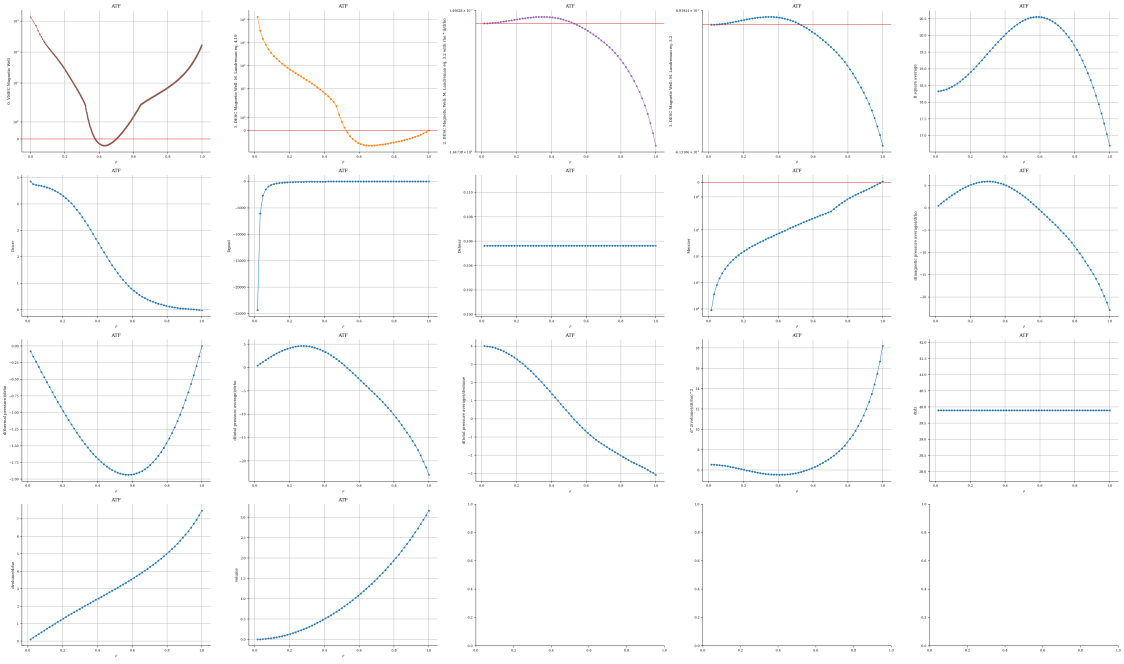

DESC M. Landreman eq. 4.19
DESC M. Landreman eq. 3.2 with rho * d/drho
DESC M. Landreman eq. 3.2
torus
magnetic well
4.27766 × 10^-16
-5.67677 × 10^-16
0.0
0.2
0.4
0.6
0.8
1.0
ρ

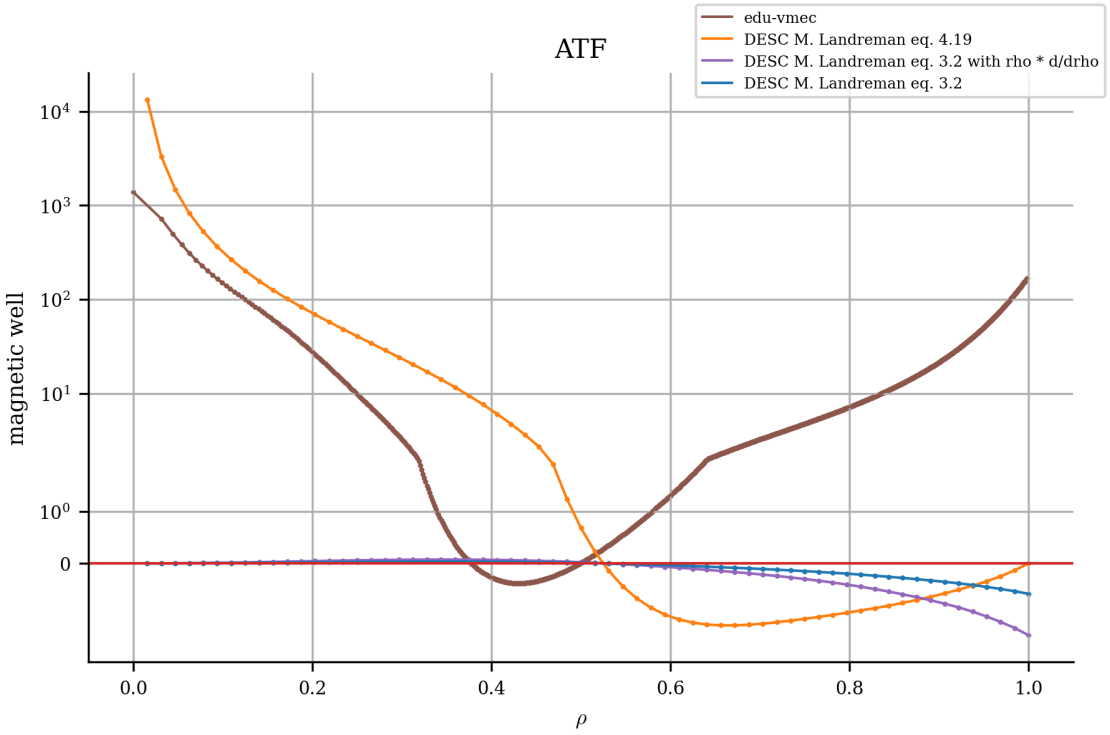


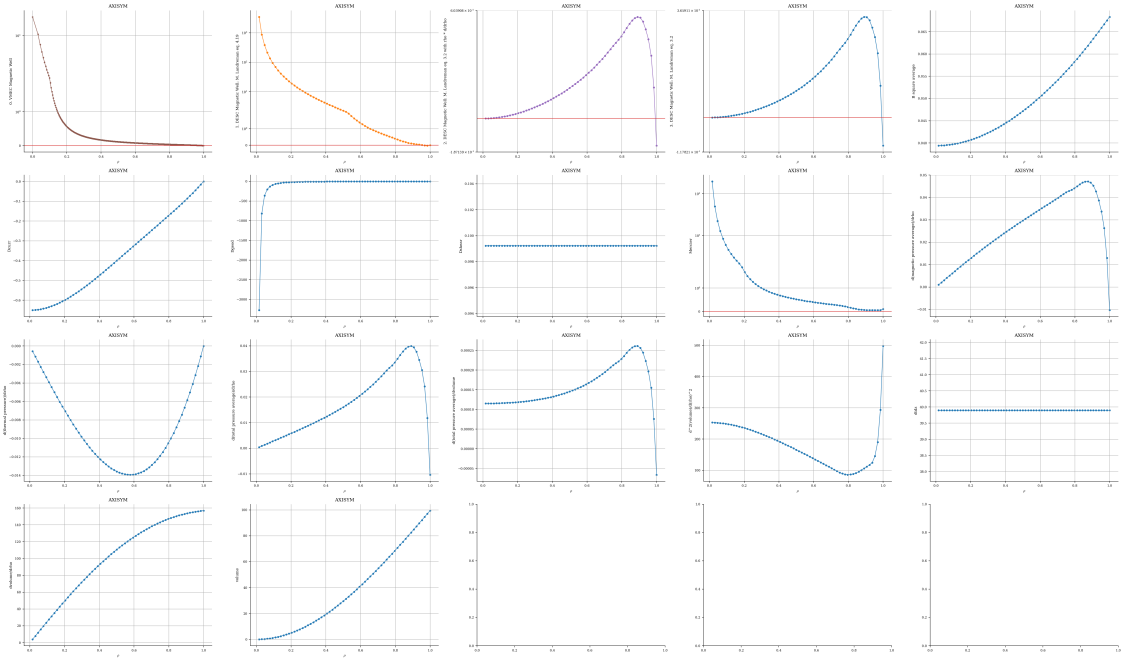

DSHAPE
STELLOPT
edu-vmec
DESC M. Landreman eq. 4.19
DESC M. Landreman eq. 3.2 with rho * d/drho
DESC M. Landreman eq. 3.2
magnetic well
10^2
10^1
10^0
0
0.0
0.2
0.4
0.6
0.8
1.0
ρ

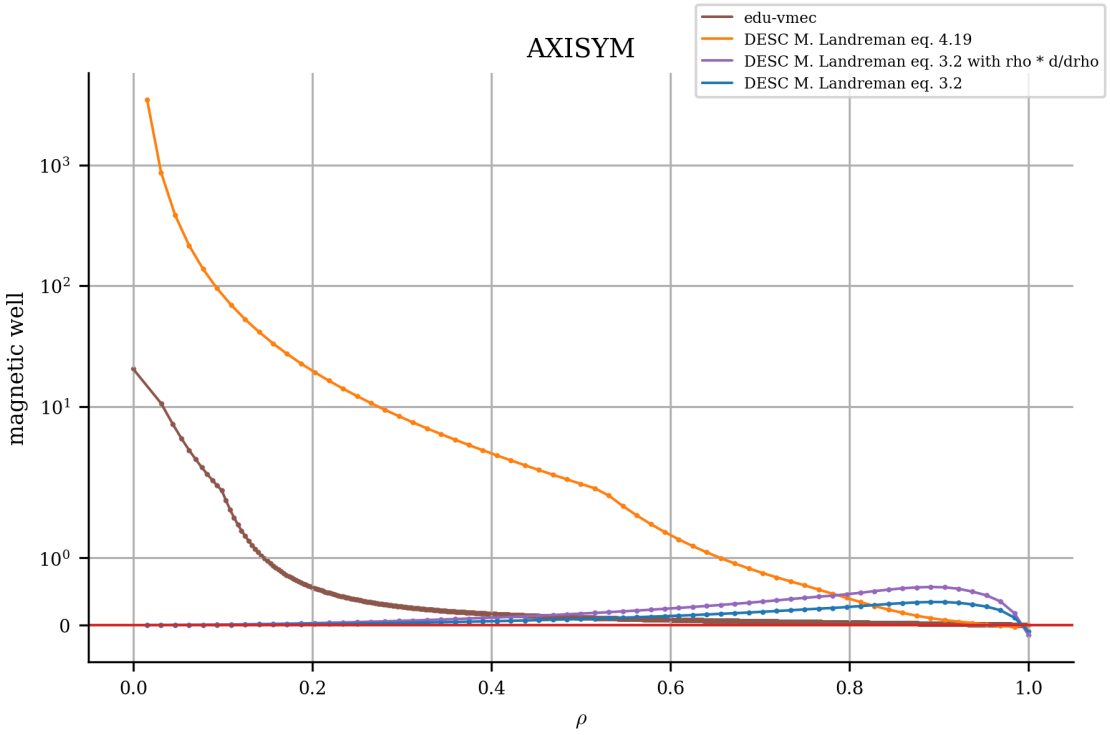



HELIOTRON
STELLOPT
edu-vmec
DESC M. Landreman eq. 4.19
DESC M. Landreman eq. 3.2 with rho * d/drho
DESC M. Landreman eq. 3.2
magnetic well
10^5
10^4
10^3
10^2
10^1
10^0
0
-10^0
-10^1
0.0
0.2
0.4
0.6
0.8
1.0
ρ




ATF
edu-vmec
DESC M. Landreman eq. 4.19
DESC M. Landreman eq. 3.2 with rho * d/drho
DESC M. Landreman eq. 3.2
magnetic well
10^4
10^3
10^2
10^1
10^0
0
0.0
0.2
0.4
0.6
0.8
1.0
ρ





[11]:
```
for s in stellarators:
    s.save()
```

5 Surface element computation problem and solution

In `MagneticWell.compute()`, the volume computation (and others) rely on the quantity $d\theta$ * $d\zeta$. This is obtained from splicing `grid.spacing`. The grid class takes advantage of things like reducing duplicated node weight, stellarator symmetry, and the number of field periods to reduce the amount of computation.

The problem is that, in the grid class's current implementation, the modifications made to `grid.spacing` to take advantage of symmetry and NFP do not preserve the value of $d\theta$ * $d\zeta$. So independently plucking out $d\theta$ * $d\zeta$ from a grid will not give the value its name implies.

Why is this? Because rescaling is done to preserve the full volume, or the weights ($d\rho$ * $d\theta$ * $d\zeta$) so that `grid.spacing.prod(axis=1).sum()` $= 4\pi 2$, at the expense of changing $d\theta$ * $d\zeta$. Here's a visual:

[12]:
```
# For any constant rho surface and NFP=1, grid.spacing is going
# to be an M*N length stack of the row vector (1, 2pi/M, 2pi/N).
g1 = LinearGrid(M=2, N=2, rho=np.array(1))
print(g1.spacing)
print()
# When NFP != 1, the values of drho are increased (decreased) while
```

```python
# dtheta*dzeta decreases (increases).
# That's why the wrong volume was being computed for the heliotron with NFP > 1.
NFP = np.random.random_sample() * 100
print("NFP:", NFP)
g2 = LinearGrid(M=2, N=2, NFP=NFP, rho=np.array(1))
print(g2.spacing)

# the good thing is that the differential volume drho*dtheta*dzeta is preserved
# since that was the (main?) goal of the rescaling
assert np.allclose(g1.spacing.prod(axis=1), g2.spacing.prod(axis=1))
# and that, because rho=constant forces the drho vector to = 1 (with NFP=1),
# grid.weights is actually the value we expect from dtheta*dzeta (for any NFP).
if g1.num_rho == 1:
    assert np.allclose(g1.spacing.prod(axis=1), g1.spacing[:, 1:].prod(axis=1))
assert np.allclose(g1.spacing.prod(axis=1), g1.weights)  # definition of weights
```

```
[[1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]
 [1.0000 1.2566 1.2566]]

NFP: 70.21113199014958
[[4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
```

```
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]
 [4.1254 5.1842 0.0738]]
```

5.1 Solution

On grids which are defined completely by a single rho surface (`grid.num_rho = 1`), the correct volume will be computed $\forall NFP \in \mathbb{R}^+$, if we take d$\theta$ * dζ to be `grid.weights` instead of `grid.spacing[:, 1:].prod(axis=1)`.

On grids which are defined over a range of rho surfaces (`grid.num_rho != 1`) this will not work because dρ != 1 on those grids. A general solution follows for computing the correct value for dθ * dζ to other grids $g \in Grid : g.num\_rho \in \mathbb{N}$. * Let $g1 \in Grid : g1.NFP = 1$ and $g2 \in Grid : g2.NFP \in \mathbb{R}^+$. Then given g2, d$\theta$ * dζ = `g2.weights / g1.spacing[:, 0]`. * I don't know how to retrieve g1.dρ from g2, other than storing it during grid construction. * Until this can be fixed, any computation which requires differential surface elements (i.e. flux surface averages require the columns from `grid.spacing`) will yield incorrect quantities $\forall g \in Grid : g.num\_rho \in \mathbb{N} \;\; 1 \; and \; g.NFP \in \mathbb{R}^+ \;\; 1$.

The last source of error in the volume is the stellarator symmetry boolean. `enforce_symmetry()` again modifies dθ * dζ. Visual given below.

The error in computed volume that results from this is much smaller than the NFP error fixed above. The nodes with $\theta > \pi$ are removed and so too are the spacings in the grid that correspond to those nodes. Now because `grid.spacing` is missing these nodes, to preserve the overall volume in (ρ, θ, ζ) space (`grid.weights.sum()`), the differential volume dρ * dθ * dζ of the remaining spaces is increased so that the sum is still $4\pi 2$. This makes sense since we do want to increase the weight of the nodes 0 to π to double count for the removed nodes. This process has preserving the volume dρ * dθ * dζ in mind. It's not clear to me if a different scale factor (other than $4\pi 2$) would be better to preserve an area element dθ * dζ. Might be dependent on ρ.

[13]:
```
g3 = LinearGrid(M=2, N=2, sym=True, rho=np.array(1))
print(g3.spacing)
```

```
[[1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]
 [1.2599 1.3194 1.5833]]
```