FittingMethods

May 1, 2023

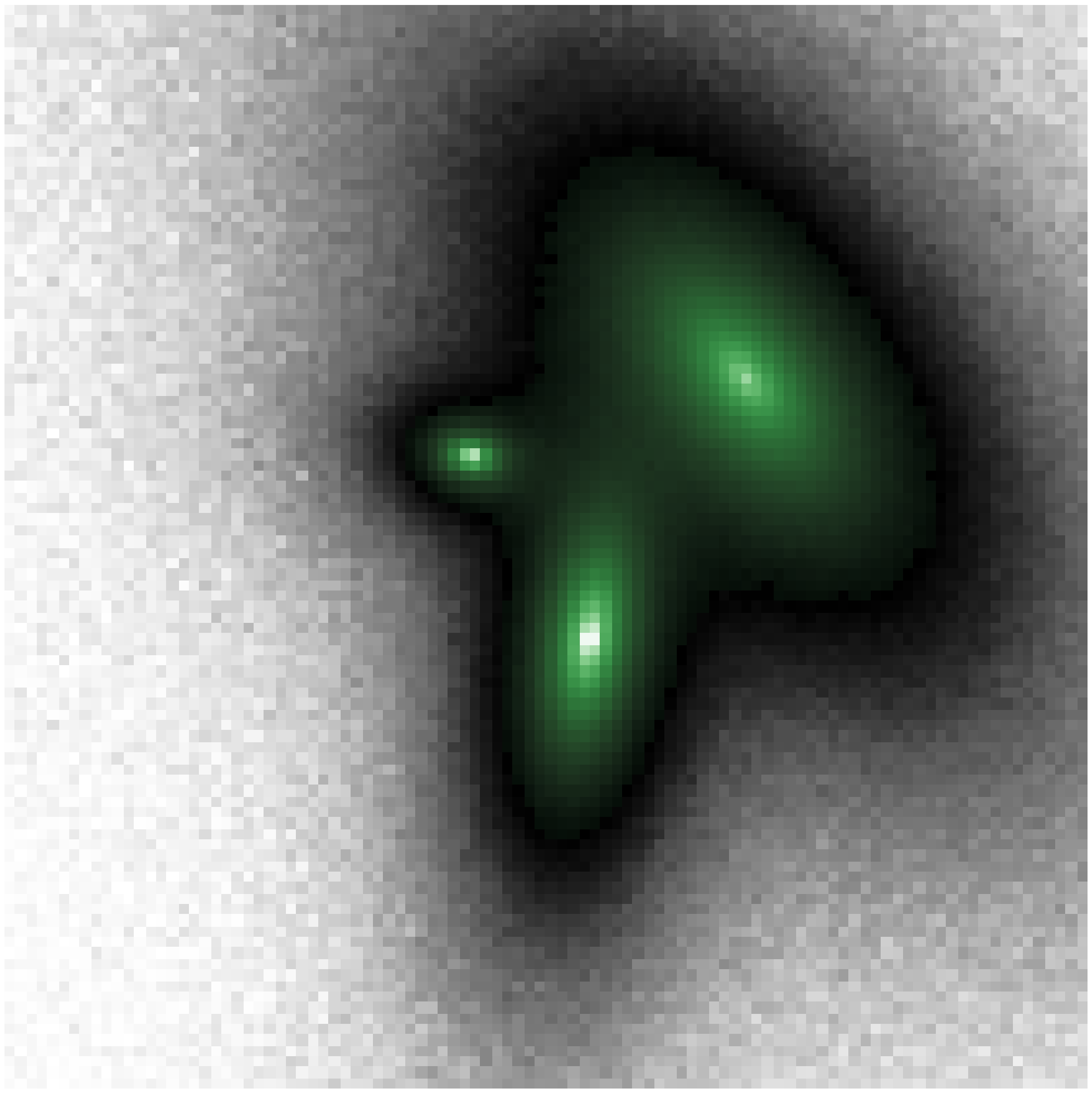
1 Fitting Methods

Here we will explore the various fitting methods in AutoProf. You have already encountered some of the methods, but here we will take a more systematic approach and discuss their strengths/weaknesses. Each method will be applied to the same problem with the same initial conditions so you can see how they operate.

[1]:
```python
import torch
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from scipy.stats import gaussian_kde as kde
from scipy.stats import norm

%matplotlib inline
import autoprof as ap
```

[2]:
```python
# Setup a fitting problem. You can ignore this cell to start, it just makes␣
↪some test data to fit

def true_params():

    # just some random parameters to use for fitting. Feel free to play around␣
↪with these to see what happens!
    sky_param = np.array([1.5])
    sersic_params = np.array([
        [ 68.44035491,  65.58516735,   0.54945988, 127.19794926*np.pi/180,   2.
↪14513004,   22.05219055,   2.45583024],
        [ 54.00353786,  41.54430634,   0.40203928,  82.03862521*np.pi/180,   2.
↪88613347,   12.095631,     2.76711163],
        [ 43.13601431,  58.3422508,    0.71894728, 167.07973506*np.pi/180,   3.
↪964371,     5.3767236,     2.41520244],
    ])

    return sersic_params, sky_param

def init_params():
    sky_param = np.array([1.4])
    sersic_params = np.array([
        [ 67.,  66.,   0.6, 130.*np.pi/180,   1.5,   25.,   2.],
        [ 55.,  40.,   0.5,  80.*np.pi/180,   2.,   10.,     3.],
        [ 41.,  60.,    0.8, 170.*np.pi/180,   3.,     4.,     2.],
    ])

    return sersic_params, sky_param

def initialize_model(target, use_true_params = True):

    # Pick parameters to start the model with
    if use_true_params:
        sersic_params, sky_param = true_params()
    else:
        sersic_params, sky_param = init_params()

    # List of models, starting with the sky
    model_list = [ap.models.AutoProf_Model(
        name = "sky",
        model_type = "flat sky model",
        target = target,
        parameters = {"sky": sky_param[0]},
    )]
    # Add models to the list
    for i, params in enumerate(sersic_params):
        model_list.append([
            ap.models.AutoProf_Model(
                name = f"sersic {i}",
                model_type = "sersic galaxy model",
                target = target,
                parameters = {
                    "center": [params[0],params[1]],
                    "q": params[2],
                    "PA": params[3],
                    "n": params[4],
                    "Re": params[5],
                    "Ie": params[6],
                },
                #psf_mode = "full", # uncomment to try everything with PSF␣
↪blurring (takes longer)
            )
        ])

    MODEL = ap.models.Group_Model(
        name = "group",
        model_list = model_list,
        target = target,
    )
    # Make sure every model is ready to go
    MODEL.initialize()

    return MODEL

def generate_target():

    N = 100
    pixelscale = 1.
    rng = np.random.default_rng(42)

    # PSF has sigma of 2x pixelscale
    PSF = ap.utils.initialize.gaussian_psf(2, 21, pixelscale)
    PSF /= np.sum(PSF)

    target = ap.image.Target_Image(
        data = np.zeros((N,N)),
        pixelscale = pixelscale,
        psf = PSF,
    )

    MODEL = initialize_model(target, True)

    # Sample the model with the true values to make a mock image
    img = MODEL().data.detach().cpu().numpy()
    # Add poisson noise
    target.data = torch.Tensor(img + rng.normal(scale = np.sqrt(img)/2))
    target.variance = torch.Tensor(img/4)

    fig, ax = plt.subplots(figsize = (8,8))
    ap.plots.target_image(fig, ax, target)
    ax.axis("off")
    plt.show()

    return target


def corner_plot(chain, labels=None, bins=None, true_values=None,␣
↪plot_density=True, plot_contours=True, figsize=(10, 10)):
    ndim = chain.shape[1]

    fig, axes = plt.subplots(ndim, ndim, figsize=figsize)
    plt.subplots_adjust(wspace=0., hspace=0.)
    if bins is None:
        bins = int(np.sqrt(chain.shape[0]))

    for i in range(ndim):
        for j in range(ndim):
            ax = axes[i, j]

            i_range = (np.min(chain[:, i]), np.max(chain[:, i]))
            j_range = (np.min(chain[:, j]), np.max(chain[:, j]))
            if i == j:
                # Plot the histogram of parameter i
                #ax.hist(chain[:, i], bins=bins, histtype="step", range =␣
↪i_range, density=True, color="k", lw=1)

                if plot_density:
                    # Plot the kernel density estimate
                    kde_x = np.linspace(i_range[0], i_range[1], 100)
                    kde_y = kde(chain[:, i])(kde_x)
                    ax.plot(kde_x, kde_y, color="green", lw=1)

                if true_values is not None:
                    ax.axvline(true_values[i], color='red', linestyle='-', lw=1)
                ax.set_xlim(i_range)

            elif i > j:
                # Plot the 2D histogram of parameters i and j
                #ax.hist2d(chain[:, j], chain[:, i], bins=bins, cmap="Greys")

                if plot_contours:
                    # Plot the kernel density estimate contours
                    kde_ij = kde([chain[:, j], chain[:, i]])
                    x, y = np.mgrid[j_range[0]:j_range[1]:100j, i_range[0]:
↪i_range[1]:100j]
                    positions = np.vstack([x.ravel(), y.ravel()])
                    kde_pos = np.reshape(kde_ij(positions).T, x.shape)
                    ax.contour(x, y, kde_pos, colors="green", linewidths=1,␣
↪levels=3)


                if true_values is not None:
                    ax.axvline(true_values[j], color='red', linestyle='-', lw=1)
                    ax.axhline(true_values[i], color='red', linestyle='-', lw=1)
                ax.set_xlim(j_range)
                ax.set_ylim(i_range)

            else:
                ax.axis("off")

            if j == 0 and labels is not None:
                ax.set_ylabel(labels[i])
            ax.yaxis.set_major_locator(plt.NullLocator())

            if i == ndim - 1 and labels is not None:
                ax.set_xlabel(labels[j])
            ax.xaxis.set_major_locator(plt.NullLocator())

    plt.show()

def corner_plot_covariance(cov_matrix, mean, labels=None, figsize=(10, 10),␣
 ↪true_values = None, ellipse_colors='g'):
    num_params = cov_matrix.shape[0]
    fig, axes = plt.subplots(num_params, num_params, figsize=figsize)
    plt.subplots_adjust(wspace=0., hspace=0.)

    for i in range(num_params):
        for j in range(num_params):
            ax = axes[i, j]

            if i == j:
                x = np.linspace(mean[i] - 3 * np.sqrt(cov_matrix[i, i]),␣
 ↪mean[i] + 3 * np.sqrt(cov_matrix[i, i]), 100)
                y = norm.pdf(x, mean[i], np.sqrt(cov_matrix[i, i]))
                ax.plot(x, y, color='g')
                ax.set_xlim(mean[i] - 3 * np.sqrt(cov_matrix[i, i]), mean[i] +␣
 ↪3 * np.sqrt(cov_matrix[i, i]))
                if true_values is not None:
                    ax.axvline(true_values[i], color='red', linestyle='-', lw=1)
            elif j < i:
                cov = cov_matrix[np.ix_([j, i], [j, i])]
                lambda_, v = np.linalg.eig(cov)
                lambda_ = np.sqrt(lambda_)
                angle = np.rad2deg(np.arctan2(v[1, 0], v[0, 0]))
                for k in [1, 2]:
                    ellipse = Ellipse(xy=(mean[j], mean[i]),
                                      width=lambda_[0] * k * 2,
                                      height=lambda_[1] * k * 2,
                                      angle=angle,
                                      edgecolor=ellipse_colors,
                                      facecolor='none')
                    ax.add_artist(ellipse)

                # Set axis limits
                margin = 3
                ax.set_xlim(mean[j] - margin * np.sqrt(cov_matrix[j, j]),␣
 ↪mean[j] + margin * np.sqrt(cov_matrix[j, j]))
```

```
                ax.set_ylim(mean[i] - margin * np.sqrt(cov_matrix[i, i]),␣
↪mean[i] + margin * np.sqrt(cov_matrix[i, i]))

                if true_values is not None:
                    ax.axvline(true_values[j], color='red', linestyle='-', lw=1)
                    ax.axhline(true_values[i], color='red', linestyle='-', lw=1)

            if j > i:
                ax.axis('off')

            if i < num_params - 1:
                ax.set_xticklabels([])
            else:
                if labels is not None:
                    ax.set_xlabel(labels[j])
            ax.yaxis.set_major_locator(plt.NullLocator())

            if j > 0:
                ax.set_yticklabels([])
            else:
                if labels is not None:
                    ax.set_ylabel(labels[i])
            ax.xaxis.set_major_locator(plt.NullLocator())

    plt.show()


target = generate_target()
```



1.1 Levenberg-Marquardt

This fitter is identitied as `ap.fit.LM` and it employs a variant of the second order Newton's method to converge very quickly to the local minimum. This is the generally accepted best algorithm for most use cases in χ^2 minimization. If you don't know what to pick, start with this minimizer. The LM optimizer bridges the gap between first-order gradient descent and second order Newton's method. When far from the minimum, Newton's method is unstable and can give wildly wrong results, so LM takes gradient descent steps. However, near the minimum it switches to the Newton's method which has "quadratic convergence" this means that it takes only a few iterations to converge to several decimal places. This can be represented as:

$(H + LI)h = g$

Where H is the Hessian matrix of second derivatives, L is the damping parameter, I is the identity matrix, h is the step we will take in parameter space, and g is the gradient. We solve this linear system for h to get the next update step. The "L" scale parameter goes from L >> 1 which represents gradient descent to L << 1 which is Newton's Method. When L >> 1 the hessian is effectively zero and we get $h = g/L$ which is just gradient descent with $1/L$ as the learning rate. In AutoProf the damping parameter is treated somewhat differently, but the concept is the same.

LM can handle a lot of scenarios and converge to the minimum. Keep in mind, however, that it is seeking a local minimum, so it is best to start off the algorithm as close as possible to the best fit parameters. AutoProf can automatically initialize, as discussed in other notebooks, but even that needs help sometimes (often in the form of a segmentation map).

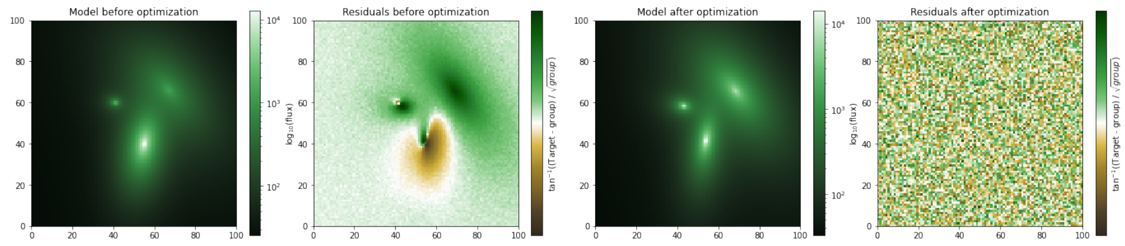
The main drawback of LM is its memory consumption which goes as $\mathcal{O}(PN)$ where P is the number of pixels and N is the number of parameters.

[3]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,4, figsize = (24,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")

res_lm = ap.fit.LM(MODEL, verbose = 1).fit()

ap.plots.model_image(fig, axarr[2], MODEL)
axarr[2].set_title("Model after optimization")
ap.plots.residual_image(fig, axarr[3], MODEL, normalize_residuals = True)
axarr[3].set_title("Residuals after optimization")
plt.show()
```

```
L: 1.0
---------init---------
LM loss: 283.81094319158024
L: 1.0
---------iter---------
LM loss: 849.0109828400438
reject
L: 11.0
---------iter---------
LM loss: 192.57226752058315
accept
L: 1.2222222222222223
---------iter---------
LM loss: 69.76483619240399
accept
L: 0.1358024691358025
---------iter---------
LM loss: 24.256005308222395
accept
```

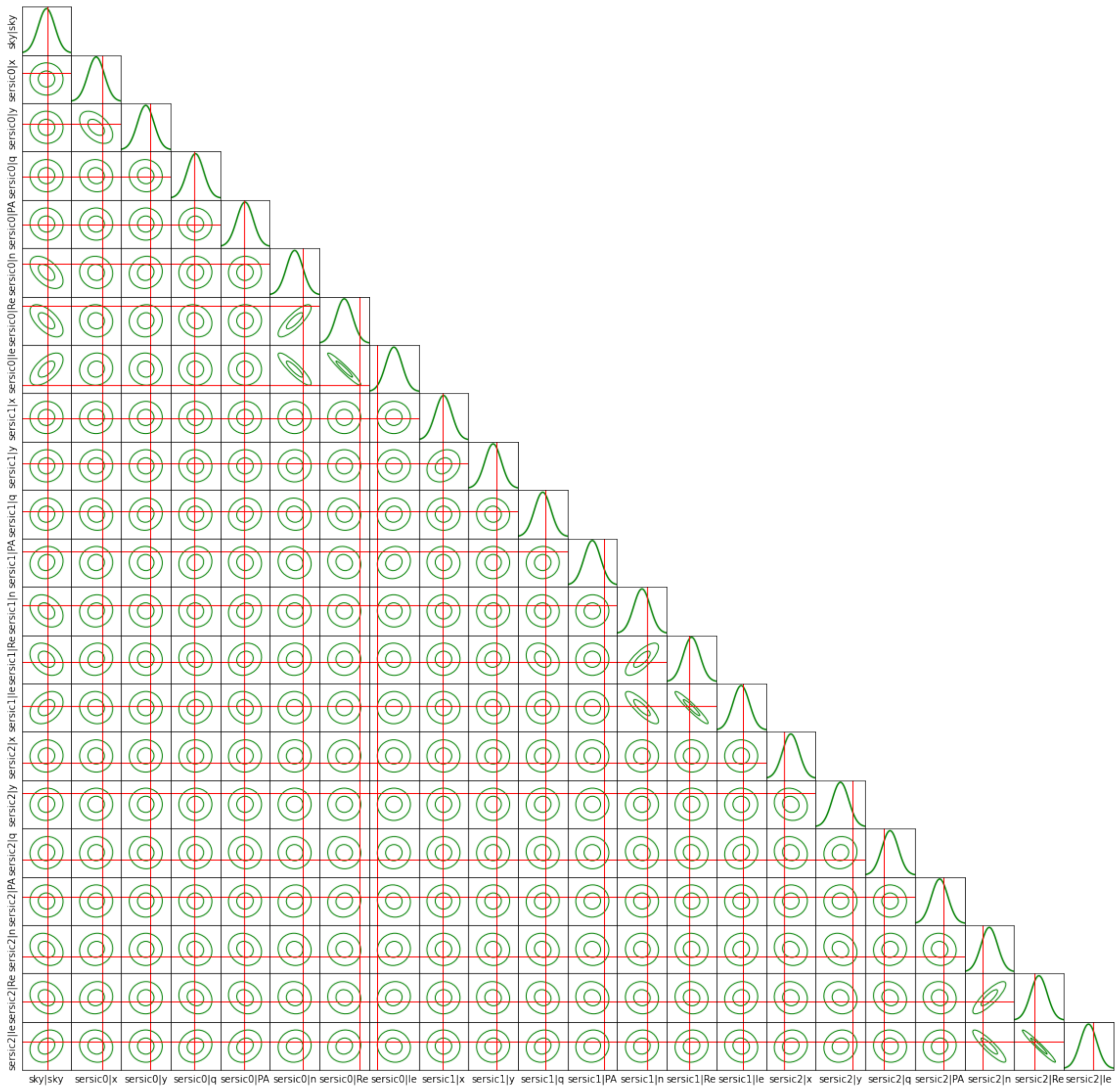
```
L: 0.015089163237311388
---------iter---------
LM loss: 30.071938222875325
reject
L: 0.16598079561042528
---------iter---------
LM loss: 27.261758221743143
reject
L: 1.825788751714678
---------iter---------
LM loss: 18.69506679958759
accept
L: 0.20286541685718645
---------iter---------
LM loss: 28.386073296858303
reject
L: 2.231519585429051
---------iter---------
LM loss: 15.242646605322426
accept
L: 0.2479466206032279
---------iter---------
LM loss: 22.387939636837555
reject
L: 2.727412826635507
---------iter---------
LM loss: 12.461517579583184
accept
L: 0.3030458696261674
---------iter---------
LM loss: 12.126933297954073
reject
L: 3.3335045658878415
---------iter---------
LM loss: 10.564772629729806
accept
L: 0.3703893962097602
---------iter---------
LM loss: 6.123883804201112
accept
L: 0.04115437735664002
---------iter---------
LM loss: 3.3033211897345005
accept
L: 0.004572708595182225
---------iter---------
LM loss: 1.9106270717708704
accept
```

```
L: 0.000508078732798025
---------iter---------
LM loss: 1.4800231508207649
accept
L: 5.6453192533113885e-05
---------iter---------
LM loss: 1.012878922723731
accept
L: 6.272576948123765e-06
---------iter---------
LM loss: 1.0127841836614357
accept
L: 6.969529942359739e-07
---------iter---------
LM loss: 1.0127841824837123
accept
```



Now that LM has found the χ^2 minimum, we can do a really neat trick. Since LM needs the hessian matrix, we have access to the hessian matrix at the minimum. This is in fact equal to the negative Fisher information matrix. If we take the matrix inverse of this matrix then we get the covariance matrix for a multivariate gaussian approximation of the χ^2 surface near the minimum. With the covariance matrix we can create a corner plot just like we would with an MCMC. We will see later that the MCMC methods (at least the ones which converge) produce very similar results!

```
[4]: param_names = list(MODEL.parameter_order())
     i = 0
     while i < len(param_names):
         param_names[i] = param_names[i].replace(" ", "")
         if "center" in param_names[i]:
             center_name = param_names.pop(i)
             param_names.insert(i, center_name.replace("center", "y"))
             param_names.insert(i, center_name.replace("center", "x"))
         i += 1
     ser, sky = true_params()
     corner_plot_covariance(
         res_lm.covariance_matrix.detach().cpu().numpy(),
         MODEL.get_parameter_vector().detach().cpu().numpy(),
         labels = param_names,
         figsize = (20,20),
         true_values = np.concatenate((sky,ser.ravel()))
     )
```



1.2 Iterative Fit (models)

An iterative fitter is identified as `ap.fit.Iter`, this method is generally employed for large models where it is not feasible to hold all the relevant data in memory at once. The iterative fitter will cycle through the models in a `Group_Model` object and fit them one at a time to the image, using the residuals from the previous cycle. This can be a very robust way to deal with some fits, especially if the overlap between models is not too strong. It is however more dependent on good initialization than other methods like the Levenberg-Marquardt. Also, it is possible for the Iter method to get stuck in a local minimum under certain circumstances.

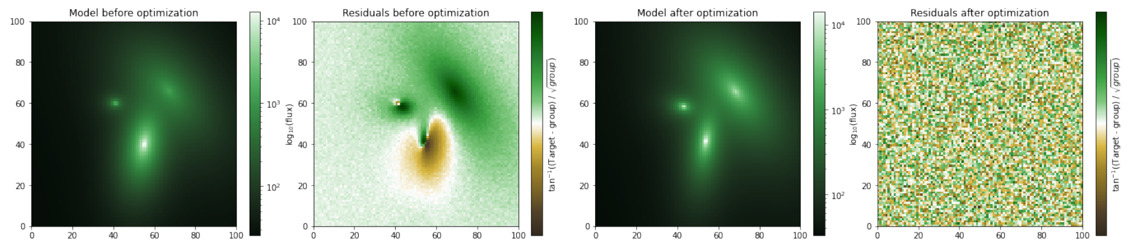
Note that while the Iterative fitter needs a `Group_Model` object to iterate over, it is not necessarily true that the sub models are `Component_Model` objects, they could be `Group_Model` objects as well. In this way it is possible to cycle through and fit "clusters" of objects that are nearby, so long as it doesn't consume too much memory.

By only fitting one model at a time it is possible to get caught in a local minimum. For this reason it can be good to mix-and-match the iterative optimizers so they can help each other get unstuck.

[5]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,4, figsize = (24,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")

res_iter = ap.fit.Iter(MODEL, verbose = 1).fit()

ap.plots.model_image(fig, axarr[2], MODEL)
axarr[2].set_title("Model after optimization")
ap.plots.residual_image(fig, axarr[3], MODEL, normalize_residuals = True)
axarr[3].set_title("Residuals after optimization")
plt.show()
```

```
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 7.205865375699771
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 2.4705932063631284
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.5805326550442294
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.2512768478124128
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.1127864669960037
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0530447882827145
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0282650623186584
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0184151513545818
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0147107760651968
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0134398452263913
--------iter-------
sky
```

```
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0129958243247057
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.012848036312808
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.0128020490769345
--------iter-------
sky
sersic 0
sersic 1
sersic 2
Update Chi^2 with new parameters
Loss: 1.012788813135204
```



1.3 Iterative Fit (parameters)

This is an iterative fitter identified as `ap.fit.Iter_LM` and is generally employed for large models where it is not feasible to hold all the relevant data in memory at once. This iterative fitter will cycle through chunks of parameters and fit them one at a time to the image. This can be a very robust way to deal with some fits, especially if the overlap between models is not too strong. This is very similar to the other iterative fitter, however it is necessary for certain fitting circumstances when the problem can't be broken down into individual component models. This occurs, for example, when the models have many shared (constrained) parameters and there is no obvious way to break down sub-groups of models (an example of this is discussed in the AutoProf paper).

Note that this is iterating over the parameters, not the models. This allows it to handle parameter covariances even for very large models (if they happen to land in the same chunk). However, for this to work it must evaluate the whole model at each iteration making it somewhat slower than the regular `Iter` fitter, though it can make up for it by fitting larger chunks at a time which makes the whole optimization faster.

By only fitting a subset of parameters at a time it is possible to get caught in a local minimum. For this reason it can be good to mix-and-match the iterative optimizers so they can help each other get unstuck.

```
[6]: MODEL = initialize_model(target, False)
     fig, axarr = plt.subplots(1,4, figsize = (24,5))
     plt.subplots_adjust(wspace= 0.1)
     ap.plots.model_image(fig, axarr[0], MODEL)
     axarr[0].set_title("Model before optimization")
     ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
     axarr[1].set_title("Residuals before optimization")

     res_iterlm = ap.fit.Iter_LM(MODEL, chunks = 11, verbose = 1).fit()

     ap.plots.model_image(fig, axarr[2], MODEL)
     axarr[2].set_title("Model after optimization")
     ap.plots.residual_image(fig, axarr[3], MODEL, normalize_residuals = True)
     axarr[3].set_title("Residuals after optimization")
     plt.show()
```

```
--------iter-------
['140099848431264:0' '140100290926240:0' '140099852362176:0'
 '140103645984896:0' '140099848431360:0' '140099848432320:0'
 '140099848430880:0' '140099848430112:0' '140099848430976:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 35.836873026504335
['140099848432224:0' '140099848431264:1' '140100290925376:0'
 '140103645984704:0' '140099848431360:1' '140099848433040:0'
 '140099848430352:0' '140099848430352:1' '140099848431552:0'
 '140099848399648:0' '140099848400800:0']
chunk loss: 14.562455505279104
Loss: 14.562455505279104
--------iter-------
['140099848431264:0' '140100290925376:0' '140100290926240:0'
 '140103645984704:0' '140103645984896:0' '140099848431360:0'
 '140099848432320:0' '140099848433040:0' '140099848430352:1'
 '140099848399648:0' '140099848400800:0']
chunk loss: 13.152171158257392
['140099848432224:0' '140099848431264:1' '140099852362176:0'
 '140099848431360:1' '140099848430880:0' '140099848430112:0'
 '140099848430976:0' '140099848430352:0' '140099848431552:0'
 '140099848399696:0' '140099848398544:0']
```

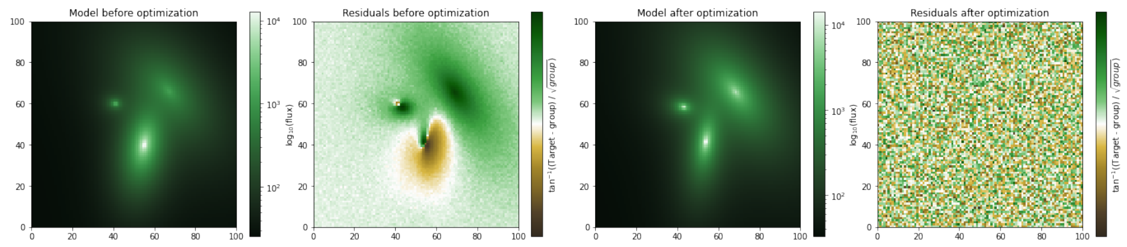
```
chunk loss: 3.510773458706077
Loss: 3.510773458706077
--------iter-------
['140099848432224:0' '140099848431264:1' '140100290925376:0'
 '140099848431360:0' '140099848430880:0' '140099848433040:0'
 '140099848430352:0' '140099848430352:1' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 3.353452269676198
['140099848431264:0' '140100290926240:0' '140099852362176:0'
 '140103645984704:0' '140103645984896:0' '140099848431360:1'
 '140099848432320:0' '140099848430112:0' '140099848430976:0'
 '140099848431552:0' '140099848400800:0']
chunk loss: 3.0961597807131325
Loss: 3.0961597807131325
--------iter-------
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290925376:0' '140103645984896:0' '140099848431360:1'
 '140099848430112:0' '140099848433040:0' '140099848430352:0'
 '140099848399648:0' '140099848398544:0']
chunk loss: 2.8631077057761045
['140100290926240:0' '140099852362176:0' '140103645984704:0'
 '140099848431360:0' '140099848432320:0' '140099848430880:0'
 '140099848430976:0' '140099848430352:1' '140099848431552:0'
 '140099848399696:0' '140099848400800:0']
chunk loss: 2.7339689038865473
Loss: 2.7339689038865473
--------iter-------
['140099848432224:0' '140100290926240:0' '140099852362176:0'
 '140099848432320:0' '140099848430976:0' '140099848433040:0'
 '140099848430352:0' '140099848430352:1' '140099848399648:0'
 '140099848399696:0' '140099848400800:0']
chunk loss: 1.821970456388208
['140099848431264:0' '140099848431264:1' '140100290925376:0'
 '140103645984704:0' '140103645984896:0' '140099848431360:0'
 '140099848431360:1' '140099848430880:0' '140099848430112:0'
 '140099848431552:0' '140099848398544:0']
chunk loss: 1.5444639707866126
Loss: 1.5444639707866126
--------iter-------
['140099848431264:0' '140100290925376:0' '140103645984704:0'
 '140099848431360:1' '140099848432320:0' '140099848430880:0'
 '140099848433040:0' '140099848430352:0' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.4684887724034323
['140099848432224:0' '140099848431264:1' '140100290926240:0'
 '140099852362176:0' '140103645984896:0' '140099848431360:0'
 '140099848430112:0' '140099848430976:0' '140099848430352:1'
 '140099848431552:0' '140099848400800:0']
```

```
chunk loss: 1.432121259627228
Loss: 1.432121259627228
--------iter-------
['140100290925376:0' '140103645984704:0' '140103645984896:0'
 '140099848431360:0' '140099848431360:1' '140099848430112:0'
 '140099848433040:0' '140099848431552:0' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.404506527696386
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290926240:0' '140099852362176:0' '140099848432320:0'
 '140099848430880:0' '140099848430976:0' '140099848430352:0'
 '140099848430352:1' '140099848400800:0']
chunk loss: 1.3726253391697867
Loss: 1.3726253391697867
--------iter-------
['140099848432224:0' '140099848431264:0' '140100290926240:0'
 '140103645984896:0' '140099848431360:0' '140099848431360:1'
 '140099848430880:0' '140099848430976:0' '140099848430352:0'
 '140099848431552:0' '140099848399648:0']
chunk loss: 1.372466005269034
['140099848431264:1' '140100290925376:0' '140099852362176:0'
 '140103645984704:0' '140099848432320:0' '140099848430112:0'
 '140099848433040:0' '140099848430352:1' '140099848399696:0'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.3197302033549454
Loss: 1.3197302033549454
--------iter-------
['140099848431264:1' '140100290926240:0' '140099852362176:0'
 '140099848430880:0' '140099848430976:0' '140099848430352:0'
 '140099848430352:1' '140099848431552:0' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.3037811179452266
['140099848432224:0' '140099848431264:0' '140100290925376:0'
 '140103645984704:0' '140103645984896:0' '140099848431360:0'
 '140099848431360:1' '140099848432320:0' '140099848430112:0'
 '140099848433040:0' '140099848400800:0']
chunk loss: 1.2850000299916242
Loss: 1.2850000299916242
--------iter-------
['140099848431264:1' '140099848431360:0' '140099848432320:0'
 '140099848430880:0' '140099848430112:0' '140099848430976:0'
 '140099848433040:0' '140099848430352:0' '140099848430352:1'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.1041517041052007
['140099848432224:0' '140099848431264:0' '140100290925376:0'
 '140100290926240:0' '140099852362176:0' '140103645984704:0'
 '140103645984896:0' '140099848431360:1' '140099848431552:0'
 '140099848399648:0' '140099848399696:0']
```

```
chunk loss: 1.0489710124807763
Loss: 1.0489710124807763
--------iter-------
['140099848431264:0' '140099848431264:1' '140099848432320:0'
 '140099848430880:0' '140099848430112:0' '140099848430976:0'
 '140099848433040:0' '140099848430352:0' '140099848431552:0'
 '140099848399648:0' '140099848398544:0']
chunk loss: 1.0285384354855114
['140099848432224:0' '140100290925376:0' '140100290926240:0'
 '140099852362176:0' '140103645984704:0' '140103645984896:0'
 '140099848431360:0' '140099848431360:1' '140099848430352:1'
 '140099848399696:0' '140099848400800:0']
chunk loss: 1.0196543663874342
Loss: 1.0196543663874342
--------iter-------
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290925376:0' '140100290926240:0' '140103645984704:0'
 '140099848431360:1' '140099848430880:0' '140099848433040:0'
 '140099848399696:0' '140099848400800:0']
chunk loss: 1.0190655127690322
['140099852362176:0' '140103645984896:0' '140099848431360:0'
 '140099848432320:0' '140099848430112:0' '140099848430976:0'
 '140099848430352:0' '140099848430352:1' '140099848431552:0'
 '140099848399648:0' '140099848398544:0']
chunk loss: 1.018786867713684
Loss: 1.018786867713684
--------iter-------
['140099848432224:0' '140099848431264:1' '140100290925376:0'
 '140099852362176:0' '140103645984704:0' '140103645984896:0'
 '140099848431360:1' '140099848430976:0' '140099848399648:0'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.0187091690365844
['140099848431264:0' '140100290926240:0' '140099848431360:0'
 '140099848432320:0' '140099848430880:0' '140099848430112:0'
 '140099848433040:0' '140099848430352:0' '140099848430352:1'
 '140099848431552:0' '140099848399696:0']
chunk loss: 1.0180926607221428
Loss: 1.0180926607221428
--------iter-------
['140099848432224:0' '140100290925376:0' '140100290926240:0'
 '140099852362176:0' '140099848431360:0' '140099848432320:0'
 '140099848430880:0' '140099848430112:0' '140099848430352:0'
 '140099848430352:1' '140099848400800:0']
chunk loss: 1.0180737867688092
['140099848431264:0' '140099848431264:1' '140103645984704:0'
 '140103645984896:0' '140099848431360:1' '140099848430976:0'
 '140099848433040:0' '140099848431552:0' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
```

```
chunk loss: 1.0163310631654936
Loss: 1.0163310631654936
--------iter-------
['140099848432224:0' '140099848431264:1' '140099848432320:0'
 '140099848430880:0' '140099848430112:0' '140099848430976:0'
 '140099848430352:0' '140099848430352:1' '140099848399648:0'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.0152477444592884
['140099848431264:0' '140100290925376:0' '140100290926240:0'
 '140099852362176:0' '140103645984704:0' '140103645984896:0'
 '140099848431360:0' '140099848431360:1' '140099848433040:0'
 '140099848431552:0' '140099848399696:0']
chunk loss: 1.0151040664642486
Loss: 1.0151040664642486
--------iter-------
['140099852362176:0' '140103645984704:0' '140103645984896:0'
 '140099848431360:0' '140099848432320:0' '140099848433040:0'
 '140099848430352:1' '140099848431552:0' '140099848399648:0'
 '140099848399696:0' '140099848400800:0']
chunk loss: 1.0150082921675936
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290925376:0' '140100290926240:0' '140099848431360:1'
 '140099848430880:0' '140099848430112:0' '140099848430976:0'
 '140099848430352:0' '140099848398544:0']
chunk loss: 1.0149317275999186
Loss: 1.0149317275999186
--------iter-------
['140099852362176:0' '140103645984896:0' '140099848431360:0'
 '140099848431360:1' '140099848432320:0' '140099848430112:0'
 '140099848430976:0' '140099848433040:0' '140099848430352:1'
 '140099848399648:0' '140099848400800:0']
chunk loss: 1.0135314229680514
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290925376:0' '140100290926240:0' '140103645984704:0'
 '140099848430880:0' '140099848430352:0' '140099848431552:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.0131967739454029
Loss: 1.0131967739454029
--------iter-------
['140099848432224:0' '140099848431264:0' '140099848431360:0'
 '140099848431360:1' '140099848432320:0' '140099848430880:0'
 '140099848430112:0' '140099848430976:0' '140099848431552:0'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.0131223233464857
['140099848431264:1' '140100290925376:0' '140100290926240:0'
 '140099852362176:0' '140103645984704:0' '140103645984896:0'
 '140099848433040:0' '140099848430352:0' '140099848430352:1'
 '140099848399648:0' '140099848399696:0']
```

```
chunk loss: 1.0130944490316525
Loss: 1.0130944490316525
--------iter-------
['140099848431264:0' '140099848431264:1' '140099852362176:0'
 '140103645984704:0' '140099848431360:1' '140099848432320:0'
 '140099848430112:0' '140099848430976:0' '140099848433040:0'
 '140099848430352:0' '140099848399696:0']
chunk loss: 1.012978072153977
['140099848432224:0' '140100290925376:0' '140100290926240:0'
 '140103645984896:0' '140099848431360:0' '140099848430880:0'
 '140099848430352:1' '140099848431552:0' '140099848399648:0'
 '140099848398544:0' '140099848400800:0']
chunk loss: 1.0129278984762435
Loss: 1.0129278984762435
--------iter-------
['140099848432224:0' '140099848431264:0' '140099848431264:1'
 '140100290925376:0' '140100290926240:0' '140103645984704:0'
 '140099848431360:1' '140099848432320:0' '140099848430880:0'
 '140099848430112:0' '140099848400800:0']
chunk loss: 1.0129235676912725
['140099852362176:0' '140103645984896:0' '140099848431360:0'
 '140099848430976:0' '140099848433040:0' '140099848430352:0'
 '140099848430352:1' '140099848431552:0' '140099848399648:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.0128991377677936
Loss: 1.0128991377677936
--------iter-------
['140099848431264:1' '140100290926240:0' '140099852362176:0'
 '140103645984704:0' '140099848432320:0' '140099848430112:0'
 '140099848430352:0' '140099848430352:1' '140099848431552:0'
 '140099848399648:0' '140099848400800:0']
chunk loss: 1.0128928911590456
['140099848432224:0' '140099848431264:0' '140100290925376:0'
 '140103645984896:0' '140099848431360:0' '140099848431360:1'
 '140099848430880:0' '140099848430976:0' '140099848433040:0'
 '140099848399696:0' '140099848398544:0']
chunk loss: 1.012888742507812
Loss: 1.012888742507812
```



1.4 Gradient Descent

A gradient descent fitter is identified as `ap.fit.Grad` and uses standard first order derivative methods as provided by PyTorch. These gradient descent methods include Adam, SGD, and LBFGS to name a few. The first order gradient is faster to evaluate and uses less memory, however it is considerably slower to converge than Levenberg-Marquardt. The gradient descent method with a small learning rate will reliably converge towards a local minimum, it will just do so slowly.

In the example below we let it run for 1000 steps and even still it has not converged. In general you should not use gradient descent to optimize a model. However, in a challenging fitting scenario the small step size of gradient descent can actually be an advantage as it will not take any unedpectedly large steps which could mix up some models, or hop over the χ^2 minimum into impossible parameter space. Just make sure to finish with LM after using Grad so that it fully converges to a reliable minimum.

[7]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,4, figsize = (24,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")

res_grad = ap.fit.Grad(MODEL, verbose = 1, max_iter = 1000, optim_kwargs =␣
 ↪{"lr": 5e-3}).fit()

ap.plots.model_image(fig, axarr[2], MODEL)
axarr[2].set_title("Model after optimization")
ap.plots.residual_image(fig, axarr[3], MODEL, normalize_residuals = True)
axarr[3].set_title("Residuals after optimization")
plt.show()
```

```
loss: 283.81094319158024
loss: 275.0393610548054
loss: 268.85931406428676
loss: 263.2779117938314
loss: 257.877334915675
loss: 252.5360252575934
loss: 247.22217833403124
loss: 241.9701048873146
loss: 236.72684346247385
loss: 231.53330760658764
loss: 226.40130852071508
loss: 221.33173382541167
loss: 216.3513607420002
loss: 211.46963084714196
loss: 206.67612250453496
```

```
loss: 201.96134579688248
loss: 197.32261129787932
loss: 192.78048552272938
loss: 188.33554270003418
loss: 183.9870655839911
loss: 179.73410690859117
loss: 175.57551803358464
loss: 171.51038920453948
loss: 167.53759711396364
loss: 163.65617554818527
loss: 159.8652348940428
loss: 156.163961996483
loss: 152.61719579475093
loss: 149.11110570683581
loss: 145.69366428214747
loss: 142.36503784991376
loss: 139.12461441907936
loss: 135.96429158199467
loss: 132.89874932029176
loss: 129.91871581018518
loss: 127.02642003435994
loss: 124.21990899280156
loss: 121.49877852316133
loss: 118.86847043127084
loss: 116.31611907280967
loss: 113.80266466016582
loss: 111.41655078144984
loss: 109.15484193103487
loss: 106.9293482488707
loss: 104.7819205352155
loss: 102.71124240337855
loss: 100.71290152999458
loss: 98.78799440108169
loss: 96.93523238749158
loss: 95.15292247895407
loss: 93.43757297433952
loss: 91.78849248086725
loss: 90.20322758570488
loss: 88.67957914006332
loss: 87.03614800064638
loss: 85.81043348397667
loss: 84.46036613864449
loss: 83.16223606790172
loss: 81.91533305248507
loss: 80.71778309850568
loss: 79.56593553098415
loss: 78.46037031794732
loss: 77.39901928454577
```

```
loss: 76.37742165934047
loss: 75.39555461692545
loss: 74.45128094558623
loss: 73.54272029656175
loss: 72.6682689512887
loss: 71.81121247776811
loss: 71.02388000140472
loss: 70.24056824710891
loss: 69.4839204427075
loss: 68.75459843132698
loss: 68.05645002551107
loss: 67.37906975291364
loss: 66.71981403719367
loss: 66.0942237172398
loss: 65.47331700431742
loss: 64.87915583932175
loss: 64.3038006813147
loss: 63.7459644252015
loss: 63.20449157233736
loss: 62.53774380579923
loss: 62.169108825074545
loss: 61.672264256625084
loss: 61.18858973442914
loss: 60.71738663820906
loss: 60.25805923181737
loss: 59.809655957768854
loss: 59.37223591832943
loss: 58.944248330795034
loss: 58.525495300818555
loss: 58.11487993634246
loss: 57.71301049142119
loss: 57.31885724775598
loss: 56.93180595750731
loss: 56.551543197369455
loss: 56.177264935117805
loss: 55.80847977794872
loss: 55.44460186401591
loss: 55.084907720845045
loss: 54.72925132391115
loss: 54.37588112105742
loss: 54.02539449414436
loss: 53.67682707601258
loss: 53.329785064119534
loss: 52.98378258450447
loss: 52.639044128128894
loss: 52.295253129444646
loss: 51.952976370315184
loss: 51.61205414241519
```

```
loss: 51.27252856110209
loss: 50.934747865042254
loss: 50.5985548983575
loss: 50.264039362003096
loss: 49.931297280319605
loss: 49.600269875927665
loss: 49.2674138611058
loss: 48.939602307062366
loss: 48.613473003544506
loss: 48.28898704889232
loss: 47.96586138920102
loss: 47.64416182217012
loss: 47.322474831830334
loss: 47.00352778989465
loss: 46.68597373413877
loss: 46.369315791492
loss: 46.053805459858154
loss: 45.73948105898749
loss: 45.42636992681638
loss: 45.11452835217678
loss: 44.80389092169222
loss: 44.494640306921994
loss: 44.18665689335939
loss: 43.88009320802736
loss: 43.57300103609254
loss: 43.269319897660814
loss: 42.967352933933704
loss: 42.66687638203881
loss: 42.36803948768871
loss: 42.070700831677904
loss: 41.77749176343157
loss: 41.4838492051131
loss: 41.19215037929946
loss: 40.90212985135737
loss: 40.61384125058809
loss: 40.32701664578645
loss: 40.04188080592544
loss: 39.75742063032564
loss: 39.475049527702176
loss: 39.19316102686258
loss: 38.91173068350094
loss: 38.630702972639845
loss: 38.348117697231395
loss: 38.066677305914574
loss: 37.784982799531356
loss: 37.504383317654785
loss: 37.22090068636923
loss: 36.922914854807736
```

```
loss: 36.6442725930131
loss: 36.387938590407764
loss: 36.112693315210436
loss: 35.83279221413085
loss: 35.5404101757898
loss: 35.24256727571578
loss: 34.96345927640203
loss: 34.47763320039172
loss: 34.35928090153894
loss: 34.069276938216056
loss: 33.78209347404544
loss: 33.49745712575716
loss: 33.2265587982232
loss: 32.94606064304948
loss: 32.66807479769871
loss: 32.391804391030526
loss: 32.11838394994759
loss: 31.835146837977668
loss: 31.563544791458085
loss: 31.293544861131974
loss: 31.02560369537414
loss: 30.759498009430544
loss: 30.49521834110536
loss: 30.23211303627716
loss: 29.971263386632568
loss: 29.712149632832134
loss: 29.455069626947868
loss: 29.1992114676351
loss: 28.945010041845382
loss: 28.691153555277147
loss: 28.440239908115455
loss: 28.190899397199754
loss: 27.943150591194307
loss: 27.697141632364985
loss: 27.4528360089844
loss: 27.210236379731242
loss: 26.969188560006266
loss: 26.730477506929798
loss: 26.493837233423616
loss: 26.25751462214387
loss: 26.036874155110823
loss: 25.806536866406432
loss: 25.57997171500205
loss: 25.356232980614895
loss: 25.1353751722483
loss: 24.90626186736333
loss: 24.690955635542924
loss: 24.476945960554506
```

```
loss: 24.266361882969836
loss: 24.05780442323526
loss: 23.85084545429805
loss: 23.646167301629585
loss: 23.44327581573657
loss: 23.24190031906016
loss: 23.04212392118846
loss: 22.843895677205335
loss: 22.647221439864854
loss: 22.45205275337509
loss: 22.25842200015532
loss: 22.066226172399187
loss: 21.87551674332084
loss: 21.686230804575615
loss: 21.498497881479693
loss: 21.312221320165037
loss: 21.127696336371265
loss: 20.944449776015766
loss: 20.770464991834793
loss: 20.590305310533353
loss: 20.411578881239336
loss: 20.234320887873324
loss: 20.058628164411964
loss: 19.879212740014566
loss: 19.706271839378346
loss: 19.531904932157577
loss: 19.361987012542407
loss: 19.193332927348074
loss: 19.025705944799476
loss: 18.860693971366228
loss: 18.696255539839274
loss: 18.53596658073086
loss: 18.374169972885387
loss: 18.213680928799977
loss: 18.05453265330074
loss: 17.896663040704713
loss: 17.740007649036798
loss: 17.58371879909154
loss: 17.430496011000677
loss: 17.27491295348596
loss: 17.123364568140865
loss: 16.972101481150467
loss: 16.824802069726733
loss: 16.67683386345954
loss: 16.52726574716731
loss: 16.38456510381751
loss: 16.240083385926873
loss: 16.09712765882679
```

```
loss: 15.955215906705252
loss: 15.814495790919342
loss: 15.675091275721316
loss: 15.536829006059069
loss: 15.399766012374299
loss: 15.239802313589175
loss: 15.101188327118653
loss: 14.963640141787184
loss: 14.82716633814122
loss: 14.69172165490637
loss: 14.557320130163893
loss: 14.4239152717357
loss: 14.29161519619461
loss: 14.16031571354628
loss: 14.030041176042072
loss: 13.900790881760438
loss: 13.77255493889022
loss: 13.645355061553001
loss: 13.51939405220547
loss: 13.39443442900879
loss: 13.270722136028748
loss: 13.148231199482389
loss: 13.025606172044263
loss: 12.904217924263856
loss: 12.782136237818841
loss: 12.660035727904695
loss: 12.538157834198797
loss: 12.417972369702559
loss: 12.298011977973445
loss: 12.179604878874276
loss: 12.061709495542363
loss: 11.945075516109627
loss: 11.828674243580766
loss: 11.713730112021816
loss: 11.598965305131074
loss: 11.485612498331683
loss: 11.372489714113556
loss: 11.260795393642873
loss: 11.149382273875604
loss: 11.039455980055397
loss: 10.930098197576728
loss: 10.82154604479049
loss: 10.713279196084123
loss: 10.60485977446622
loss: 10.496368329493494
loss: 10.38845063912835
loss: 10.28148823993134
loss: 10.175588196429896
```

```
loss: 10.070798095618787
loss: 9.98647970675148
loss: 9.883569896297136
loss: 9.78162481347711
loss: 9.680635542251979
loss: 9.580585473643822
loss: 9.48145675936899
loss: 9.38322690627232
loss: 9.285890824470751
loss: 9.189454882227304
loss: 9.093899822096043
loss: 8.999230430530554
loss: 8.90548532162485
loss: 8.812549667093432
loss: 8.720624301456981
loss: 8.629424198811945
loss: 8.539066513832823
loss: 8.449576630562976
loss: 8.360935357531575
loss: 8.27315964645698
loss: 8.214547546778421
loss: 8.098956934323098
loss: 8.059067071970999
loss: 7.945142629317991
loss: 7.845147680285465
loss: 7.7653290080828405
loss: 7.682456222656608
loss: 7.602159011989532
loss: 7.563838884001277
loss: 7.48387058589519
loss: 7.404132490379967
loss: 7.325099847398162
loss: 7.2483588576804365
loss: 7.17053968077787
loss: 7.094016204001684
loss: 7.020877108118594
loss: 6.942051886287375
loss: 6.86888817811633
loss: 6.7894276396528594
loss: 6.719089888583661
loss: 6.642483520917179
loss: 6.5687063726967185
loss: 6.497670952803442
loss: 6.42546783462923
loss: 6.355497334761103
loss: 6.284327402947529
loss: 6.215406374028144
loss: 6.145305150744525
```

```
loss: 6.077372086034397
loss: 6.008195921600021
loss: 5.94503056964956
loss: 5.872256177417711
loss: 5.807891294829307
loss: 5.732946496051477
loss: 5.662197919295544
loss: 5.59733142457868
loss: 5.5532645446750575
loss: 5.50405154403209
loss: 5.455333855667066
loss: 5.407106895515275
loss: 5.359367822921792
loss: 5.3122079559713224
loss: 5.265690646147662
loss: 5.219747994709774
loss: 5.174476725173939
loss: 5.12965003298771
loss: 5.085484945731889
loss: 5.041707827121716
loss: 4.998538748462445
loss: 4.955836481940574
loss: 4.934619143769808
loss: 4.890275903375149
loss: 4.846416265078942
loss: 4.803058592000096
loss: 4.760263635325948
loss: 4.71801862923297
loss: 4.676370912670523
loss: 4.635367856950673
loss: 4.595014689230378
loss: 4.555343886873921
loss: 4.516354661379579
loss: 4.478016847313252
loss: 4.440348560092482
loss: 4.403292206036532
loss: 4.366840536935825
loss: 4.33097799400552
loss: 4.295700945421462
loss: 4.26098239315739
loss: 4.226857927222629
loss: 4.19330462018127
loss: 4.16029447173593
loss: 4.127816974261961
loss: 4.0958769589513855
loss: 4.0644034246779395
loss: 4.033409539217486
loss: 4.002882279847432
```

```
loss: 3.972864077050237
loss: 3.943660875203663
loss: 3.9150362152557485
loss: 3.8869205562903946
loss: 3.8592465134331495
loss: 3.831991689307706
loss: 3.805167340213097
loss: 3.778769437224301
loss: 3.752773487085383
loss: 3.7272403831032954
loss: 3.701697871183084
loss: 3.6769633399025206
loss: 3.6526371026579114
loss: 3.6287211994296733
loss: 3.6051946998540987
loss: 3.5815818412422598
loss: 3.558822265945834
loss: 3.536439115861003
loss: 3.514418994782947
loss: 3.4927563855245434
loss: 3.4717065437501295
loss: 3.4507328753565005
loss: 3.430114145573814
loss: 3.4097984490257987
loss: 3.389827468074674
loss: 3.3701091209521286
loss: 3.3507520447283152
loss: 3.3316351326983447
loss: 3.312864244151869
loss: 3.294336592203487
loss: 3.2761389669637864
loss: 3.258178910695792
loss: 3.2405522050197106
loss: 3.2231426806253003
loss: 3.2067943412886613
loss: 3.1893348868272544
loss: 3.1736302960176914
loss: 3.156654175530025
loss: 3.14156347582154
loss: 3.1258380239706276
loss: 3.11037530940688
loss: 3.0950531628622406
loss: 3.0798177779458764
loss: 3.0649197000333475
loss: 3.04993209028727
loss: 3.035272438287063
loss: 3.020536358859174
loss: 3.0061292659049146
```

```
loss: 2.9916714366956523
loss: 2.977585886856908
loss: 2.9634869583303174
loss: 2.9498938473091054
loss: 2.9360629205143205
loss: 2.9228061624792008
loss: 2.9095213560369237
loss: 2.8966440841979098
loss: 2.8839466123879594
loss: 2.8713089922106305
loss: 2.858849699246146
loss: 2.846563420517306
loss: 2.8344450173394256
loss: 2.8224903013609572
loss: 2.8106955863862524
loss: 2.7990582458639923
loss: 2.78758935885855
loss: 2.776261655592545
loss: 2.7650876008330623
loss: 2.7540669370041004
loss: 2.743199041718603
loss: 2.732482486104699
loss: 2.7219147405060338
loss: 2.7114919149950243
loss: 2.7012094517315326
loss: 2.69106985584824
loss: 2.6810529458119396
loss: 2.671161698486393
loss: 2.661392154209994
loss: 2.6517406744801884
loss: 2.6422059462727496
loss: 2.632779781166247
loss: 2.6226841319261927
loss: 2.613420692127659
loss: 2.604258440269077
loss: 2.595204769074917
loss: 2.586261459722816
loss: 2.5774216103968968
loss: 2.5686875550887707
loss: 2.5600588392508166
loss: 2.551536122206053
loss: 2.543120873194891
loss: 2.5337630611327078
loss: 2.5266375226231506
loss: 2.5175553460864006
loss: 2.509601895317351
loss: 2.501793551696587
loss: 2.494084976156532
```

```
loss: 2.4865037759831647
loss: 2.4790141353400448
loss: 2.47162079269828
loss: 2.4643186874974474
loss: 2.4571063181449975
loss: 2.4499766937990497
loss: 2.4429297867215345
loss: 2.4359637656818904
loss: 2.429099629989008
loss: 2.4222910792360177
loss: 2.415559445302636
loss: 2.408903566263706
loss: 2.4023311531939693
loss: 2.395839846778023
loss: 2.3894052474661107
loss: 2.3830427019579674
loss: 2.376751457807765
loss: 2.3705308433785115
loss: 2.364380312140846
loss: 2.3582994599920624
loss: 2.352288084021293
loss: 2.346346228518163
loss: 2.340474248184974
loss: 2.3346728542291717
loss: 2.3289431021093794
loss: 2.3232862986352694
loss: 2.317703729627853
loss: 2.312196273664787
loss: 2.3067639744418633
loss: 2.301413169105231
loss: 2.2959070451738746
loss: 2.2909084530974617
loss: 2.285538522698172
loss: 2.280668217734376
loss: 2.275450490540954
loss: 2.270698838413069
loss: 2.2655729018946884
loss: 2.2609330597166313
loss: 2.25592284556476
loss: 2.2511802542845563
loss: 2.2464908974977567
loss: 2.2418538262231826
loss: 2.237285900257911
loss: 2.2327514950293774
loss: 2.228103650010411
loss: 2.2236502258775896
loss: 2.2192657650970897
loss: 2.214920872526156
```

```
loss: 2.2106429104442933
loss: 2.20639473795068
loss: 2.202212375547929
loss: 2.1980594864710086
loss: 2.1948069025468198
loss: 2.1899360237352767
loss: 2.1867398769086934
loss: 2.181974741191468
loss: 2.17886458958324
loss: 2.17419510464482
loss: 2.1711762786239097
loss: 2.166796643457763
loss: 2.1636678818377995
loss: 2.1593871980256267
loss: 2.156356537973522
loss: 2.152161644253236
loss: 2.1492212723548563
loss: 2.145227324167646
loss: 2.142282536381573
loss: 2.1384405097080474
loss: 2.135591891239162
loss: 2.1318714098038845
loss: 2.1280776524241536
loss: 2.1255454763020167
loss: 2.1218879923004788
loss: 2.1194840384188534
loss: 2.1159626458759
loss: 2.1136694212659806
loss: 2.107289345161222
loss: 2.1084556972835014
loss: 2.102221124674192
loss: 2.104120919503242
loss: 2.0971172190782585
loss: 2.098701421515709
loss: 2.0914884697412486
loss: 2.092798519199246
loss: 2.0853569355526567
loss: 2.0857609990114203
loss: 2.0789905757552822
loss: 2.079448251902591
loss: 2.0727489799903136
loss: 2.0732651517995437
loss: 2.0667835357899222
loss: 2.0673357546970186
loss: 2.0638174871392803
loss: 2.0614959643545383
loss: 2.058167056586004
loss: 2.056009979097942
```

```
loss: 2.0528227178004324
loss: 2.050765099306131
loss: 2.048696489136842
loss: 2.0455556349439115
loss: 2.0437607803929536
loss: 2.0406825380249134
loss: 2.038948517206224
loss: 2.0366014925869678
loss: 2.0342697293899983
loss: 2.031963806218327
loss: 2.0296825197328943
loss: 2.0273125152113076
loss: 2.0250798282853135
loss: 2.022869831070167
loss: 2.0206821648690125
loss: 2.01851632343009
loss: 2.016367147395759
loss: 2.0142486196630323
loss: 2.0121411895866586
loss: 2.0100668087738693
loss: 2.0080047502135634
loss: 2.005962480691096
loss: 2.003939720409795
loss: 2.0019361622868534
loss: 1.9999051150779978
loss: 1.9979860292097769
loss: 1.9959891515861212
loss: 1.9940572120883908
loss: 1.992143013388988
loss: 1.9902464756412719
loss: 1.9883671900380986
loss: 1.9865049999254722
loss: 1.9846595692864433
loss: 1.9828307020254416
loss: 1.9810181046967827
loss: 1.9792215624069955
loss: 1.9774408079121013
loss: 1.975675618745095
loss: 1.9739257449229264
loss: 1.9721909625446727
loss: 1.9704710339642928
loss: 1.968765737214768
loss: 1.9670835142621357
loss: 1.9654068137299423
loss: 1.9637440764756757
loss: 1.9620950933536876
loss: 1.9604855976975082
loss: 1.9588634615185618
```

```
loss: 1.9572544497685678
loss: 1.9556583592556191
loss: 1.9540870528877632
loss: 1.9525162291282376
loss: 1.9509577374376175
loss: 1.9494113833006537
loss: 1.9478769807269312
loss: 1.9463543419992329
loss: 1.9448455008070262
loss: 1.9433436011066934
loss: 1.9418573741972667
loss: 1.9403799784298537
loss: 1.9389134598237603
loss: 1.9374576473477132
loss: 1.9360123750398135
loss: 1.9345774774431705
loss: 1.9331683512017945
loss: 1.9317537029007525
loss: 1.9303489506134752
loss: 1.9289905109291368
loss: 1.9276042198286283
loss: 1.9262316407216287
loss: 1.9248641333836705
loss: 1.9234673010583845
loss: 1.9221199026880817
loss: 1.9207813561947955
loss: 1.9194515355294772
loss: 1.9181302880734101
loss: 1.916817486959542
loss: 1.9155129889183193
loss: 1.9142497512814571
loss: 1.9129614598790858
loss: 1.9116810862915734
loss: 1.9104085000365503
loss: 1.9091435810430277
loss: 1.9071146038642182
loss: 1.9058673795280896
loss: 1.9046275095949265
loss: 1.9033948698765304
loss: 1.9021693363651162
loss: 1.9009507902464857
loss: 1.8997391132586203
loss: 1.8985341913517304
loss: 1.8973359114934392
loss: 1.8961441641901637
loss: 1.8949588412419516
loss: 1.8937798374887875
loss: 1.8925715114203818
```

```
loss: 1.8914045987199457
loss: 1.8902436984528623
loss: 1.8890887139390882
loss: 1.8879395503084029
loss: 1.8867961144186478
loss: 1.885658314830163
loss: 1.8845260617542179
loss: 1.8833992670140554
loss: 1.882277844009449
loss: 1.8811617076764533
loss: 1.8800507744578736
loss: 1.8789449622620291
loss: 1.877844190442866
loss: 1.8767483797567488
loss: 1.875657452350947
loss: 1.8745713317184534
loss: 1.873489942693858
loss: 1.8724132114057614
loss: 1.8713410652794804
loss: 1.8702734329864166
loss: 1.869210244453376
loss: 1.8681514308081597
loss: 1.8670969243954996
loss: 1.8660466587183628
loss: 1.8650005684612938
loss: 1.8639585894232023
loss: 1.8629206585515208
loss: 1.8618867138693773
loss: 1.8610060327634237
loss: 1.8599808483600697
loss: 1.8589594631468551
loss: 1.8579418629636284
loss: 1.856927945290403
loss: 1.855917689949036
loss: 1.8549110067246017
loss: 1.8539078706489205
loss: 1.85290820181381
loss: 1.8519119725901239
loss: 1.8509191120345712
loss: 1.849929591707705
loss: 1.8489433489654972
loss: 1.847960356461992
loss: 1.8469805600027334
loss: 1.8460039356905729
loss: 1.8450304390599452
loss: 1.842336140227497
loss: 1.8431142194235917
loss: 1.8404557349105375
```

```
loss: 1.841764652848314
loss: 1.8386008584602656
loss: 1.8399384515557997
loss: 1.8368163309262056
loss: 1.8378348331406875
loss: 1.834164717315391
loss: 1.8365017871123677
loss: 1.8345081013854856
loss: 1.8352194215154403
loss: 1.8332352105495002
loss: 1.8341759916250755
loss: 1.8331780227191692
loss: 1.8330119150390827
loss: 1.83199743874719
loss: 1.8317879506307202
loss: 1.8313566974335325
loss: 1.830344587423631
loss: 1.8297162886365388
loss: 1.8284871620649426
loss: 1.8276371381481376
loss: 1.8262153633879175
loss: 1.8245314460335122
loss: 1.8236628028649036
loss: 1.8219499593565904
loss: 1.8211007588026111
loss: 1.8185349111492266
loss: 1.818906631050457
loss: 1.816486931849436
loss: 1.8168237712483606
loss: 1.8143991169453189
loss: 1.8147825614425614
loss: 1.8123785889053745
loss: 1.8127915282825213
loss: 1.8104084744024835
loss: 1.810850227264154
loss: 1.80699659233488
loss: 1.8091094205958274
loss: 1.805257875840772
loss: 1.8073480044130084
loss: 1.8035192271301426
loss: 1.8056076347227576
loss: 1.8017815205497223
loss: 1.8038683579539792
loss: 1.8015053195214874
loss: 1.8018415367226035
loss: 1.7981753426096496
loss: 1.800140570355523
loss: 1.7964771047094052
```

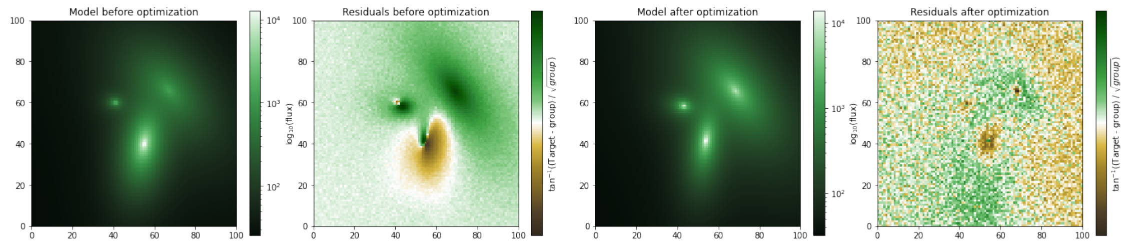
```
loss: 1.7984401762506135
loss: 1.7947797049564622
loss: 1.7967406365447935
loss: 1.7930834191543341
loss: 1.7950510328138305
loss: 1.7913866283144966
loss: 1.7933524426301053
loss: 1.7911140810544834
loss: 1.7916950641926064
loss: 1.7878934973522838
loss: 1.7898653248080698
loss: 1.7862263212443323
loss: 1.7881957427634416
loss: 1.7845608148247154
loss: 1.7867272315472664
loss: 1.7843197115454206
loss: 1.7849397368384485
loss: 1.7811519531021398
loss: 1.7833208793719895
loss: 1.7795345963724367
loss: 1.7816993354316804
loss: 1.7792913016009162
loss: 1.7802863008603613
loss: 1.7761756746057336
loss: 1.778341009426077
loss: 1.7759378194864752
loss: 1.776946579275208
loss: 1.7728537346947693
loss: 1.7750186361491425
loss: 1.7726187235300994
loss: 1.773639145916413
loss: 1.7695630300841025
loss: 1.7720757608144462
loss: 1.769341803730547
loss: 1.7703704368484277
loss: 1.7663097123143763
loss: 1.7688174085480393
loss: 1.766085789667531
loss: 1.7671219941251217
loss: 1.7630762526370567
loss: 1.7655788534681696
loss: 1.7628494580916432
loss: 1.7638926729695337
loss: 1.7611742623286344
loss: 1.7622290805540448
loss: 1.758209741954773
loss: 1.7607068783998767
loss: 1.757988006920101
```

```
loss: 1.7590466730547625
loss: 1.7563370766240367
loss: 1.7574054716354661
loss: 1.7534104998129407
loss: 1.755900944308063
loss: 1.7531895070002133
loss: 1.7542599246987445
loss: 1.7515568417109713
loss: 1.7526358710174013
loss: 1.7499395248183351
loss: 1.7510255941390762
loss: 1.7470633864542393
loss: 1.749544691894521
loss: 1.7468447746784626
loss: 1.7479304560879168
loss: 1.7452374836327005
loss: 1.7463301341763504
loss: 1.7436429088173435
loss: 1.7447413437734993
loss: 1.7420590499442392
loss: 1.7431623938980105
loss: 1.74048438480713
loss: 1.7415919869130043
loss: 1.7389177579923647
loss: 1.740029125238227
loss: 1.7373582881796104
loss: 1.7384730381921822
loss: 1.7358052977196055
loss: 1.7369231258616409
loss: 1.7342582601148766
loss: 1.7353789167650537
loss: 1.7327167612332752
loss: 1.733840036183025
loss: 1.731180470772833
loss: 1.73230618252155
loss: 1.7296491212581493
loss: 1.7307771096393028
loss: 1.7281224925246033
loss: 1.7292526135757527
loss: 1.7266004001895074
loss: 1.727732522524705
loss: 1.7250826870233873
loss: 1.7262166892099464
loss: 1.7235692164412524
loss: 1.724704985051259
loss: 1.7220598675563816
loss: 1.7231972956783366
loss: 1.7205545313992965
```

```
loss: 1.7216935174724703
loss: 1.7190622732431213
loss: 1.7201935562851502
loss: 1.7175646624194174
loss: 1.7187044568905698
loss: 1.7172537691872514
loss: 1.7158414595482758
loss: 1.7132151958541104
loss: 1.7143779956781056
loss: 1.7129215134761113
loss: 1.712828139310406
loss: 1.7102084276471685
loss: 1.7113750583792615
loss: 1.7087570695147463
loss: 1.709926152964093
loss: 1.7084721594406176
loss: 1.708391358222552
loss: 1.7057795883714904
loss: 1.706952711053431
loss: 1.7055014501032688
loss: 1.704141242789741
loss: 1.702792105854768
loss: 1.7026421723465028
loss: 1.7013094903996464
loss: 1.7011550839412715
loss: 1.699826383464564
loss: 1.6991091841826054
loss: 1.698359058152422
loss: 1.6976444709685174
loss: 1.6968970276451594
loss: 1.6961743285039201
loss: 1.6954404360267157
loss: 1.69472076348199
loss: 1.693988324380067
loss: 1.6932705905893128
loss: 1.6925400980075174
loss: 1.6918242199225295
loss: 1.6910955886988797
loss: 1.690381502617122
loss: 1.6896546691494152
loss: 1.688942325062543
loss: 1.6882172404046103
loss: 1.6875065981897708
loss: 1.6867832236551565
loss: 1.6860742505639392
loss: 1.6853525549467616
loss: 1.6846559680368351
loss: 1.6839250703250686
```

```
loss: 1.6832300941509544
loss: 1.6825116739757269
loss: 1.6818075593318353
loss: 1.6810907545626455
loss: 1.6803882230276483
loss: 1.679673018828669
loss: 1.6789720583862997
loss: 1.6782584431347618
loss: 1.6775590440671047
loss: 1.676847008615624
loss: 1.6761491633033039
loss: 1.6754387007420704
loss: 1.6747424035773575
loss: 1.6740335091171112
loss: 1.673338756503739
loss: 1.672631427476623
loss: 1.6719382179290352
loss: 1.6712324539127033
loss: 1.6705407882722154
loss: 1.6698365913519302
loss: 1.6691464731446175
loss: 1.668443848339766
loss: 1.6677552843125494
loss: 1.6670542402092052
loss: 1.6663672410970014
loss: 1.66566779074851
loss: 1.6649823723484412
loss: 1.6642845345336477
loss: 1.663600719194365
loss: 1.662904520161548
loss: 1.6622223388409463
loss: 1.6615278147189367
loss: 1.6614001342352485
loss: 1.6601504645557705
loss: 1.660023579200089
loss: 1.6587761403412662
loss: 1.6586500627143093
loss: 1.6574048245379838
loss: 1.656728399072158
loss: 1.6560398482961343
loss: 1.6559155714966647
loss: 1.6546746447686145
loss: 1.654551171793466
loss: 1.6533123973145778
loss: 1.6531897351546685
loss: 1.6519530934192956
loss: 1.651831245744091
loss: 1.6505967203055683
```

```
loss: 1.6499274494511 89
loss: 1.6492461891395451
loss: 1.649126173837288
loss: 1.6478958643406396
loss: 1.647776651548971
loss: 1.646548436830631
loss: 1.6464300305014075
loss: 1.6452038969227052
loss: 1.6450862984647026
loss: 1.6438622338493942
loss: 1.643745443933312
loss: 1.6425234375135938
loss: 1.6424074568626885
loss: 1.6411874996041054
loss: 1.6410723298647851
loss: 1.639854415252756
loss: 1.639196259936101
loss: 1.6385262372702691
loss: 1.6384128603093893
loss: 1.637199012656819
loss: 1.6370864275714128
loss: 1.6358746082643436
loss: 1.6357628208676782
loss: 1.6345530248955886
loss: 1.63444204311657
```



1.5 No U-Turn Sampler (NUTS)

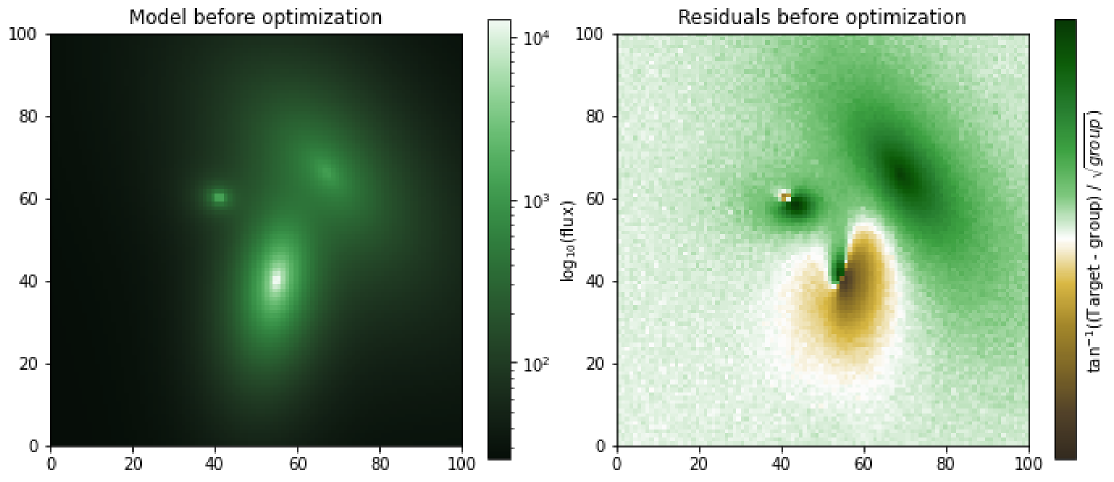
Unlike the above methods, `ap.fit.NUTS` does not stricktly seek a minimum χ^2, instead it is an MCMC method which seeks to explore the likelihood space and provide a full posterior in the form of random samples. The NUTS method in AutoProf is actually just a wrapper for the Pyro implementation (**link here**). Most of the functionality can be accessed this way, though for very advanced applications it may be necessary to manually interface with Pyro (this is not very challenging as AutoProf is fully differentiable).

The first iteration of NUTS is always very slow since it compiles the forward method on the fly, after that each sample is drawn much faster. The warmup iterations take longer as the method is exploring the space and determining the ideal step size and mass matrix for fast integration with minimal numerical error (we only do 20 warmup steps here, if something goes wrong just try rerunning). Once the algorithm begins sampling it is able to move quickly (for an MCMC) throught the parameter space. For many models, the NUTS sampler is able to collect nearly completely uncorrelated samples, meaning that even 100 is enough to get a good estimate of the posterior.

NUTS is far faster than other MCMC implementations such as a standard Metropolis Hastings MCMC. However, it is still a lot slower than the other optimizers (LM) since it is doing more than seeking a single high likelihood point, it is fully exploring the likelihood space. In simple cases, the automatic covariance matrix from LM is likely good enough, but if one really needs access to the full posterior of a complex model then NUTS is the best way to get it.

For an excellent introduction to the Hamiltonian Monte-Carlo and a high level explanation of NUTS see this review: **Betancourt 2018**

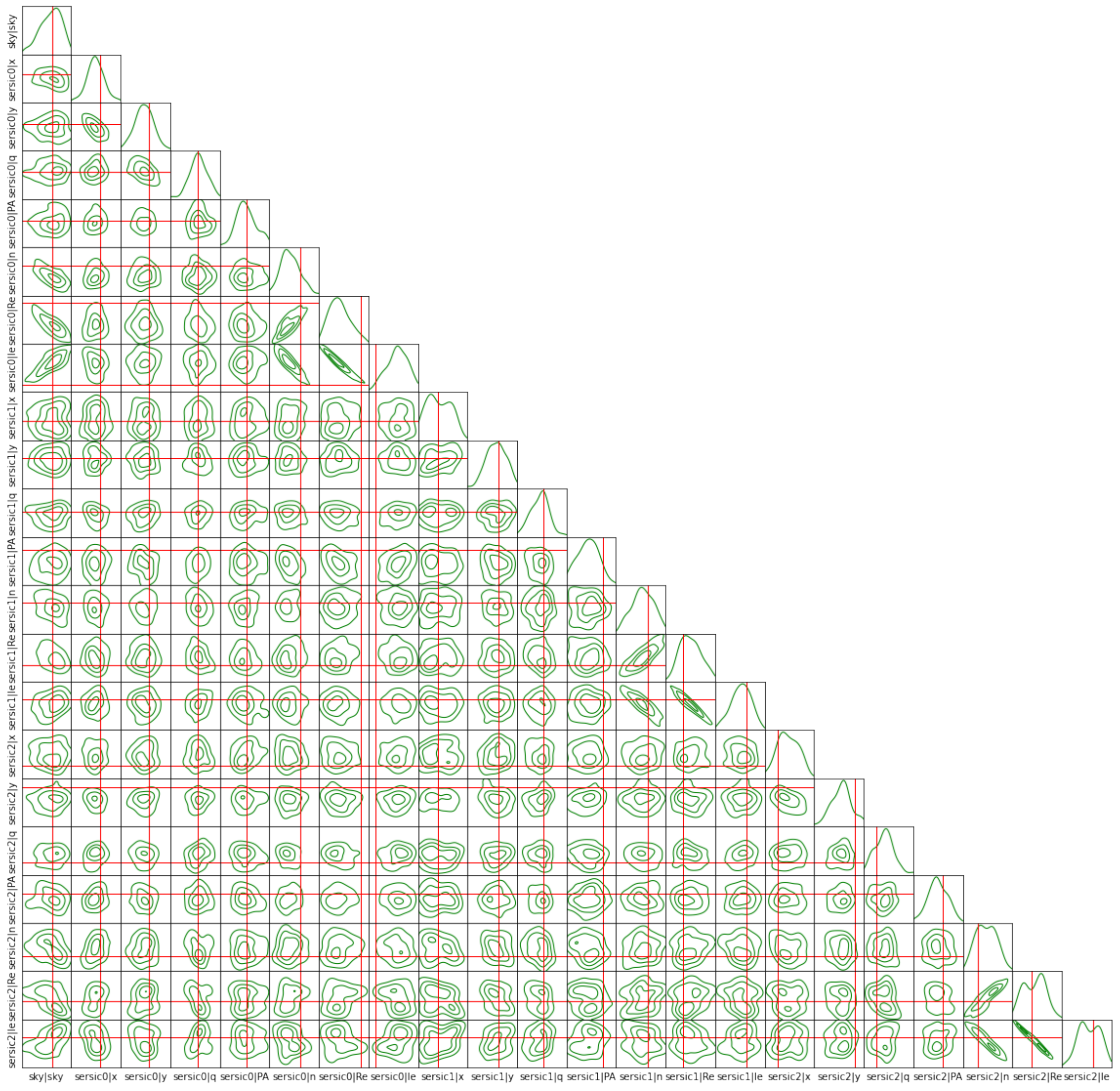
[8]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,2, figsize = (12,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")
plt.show()

# Use LM to start the sampler at a high likelihood location, no burn-in needed!
# In general, NUTS is quite fast to do burn-in so this is often not needed
res1 = ap.fit.LM(MODEL).fit()

# Run the NUTS sampler
res_nuts = ap.fit.NUTS(MODEL, warmup = 20, max_iter = 100).fit()
```



```
Sample: 100%|      | 120/120 [06:03,  3.03s/it, step size=2.45e-02, acc.
prob=0.653]
```

Note that there is no “after optimization” image above, because optimization was not done, it was full likelihood exploration. We can now create a corner plot with 2D projections of the 22 dimensional space that NUTS was exploring. The resulting corner plot is about what you would expect to get with 100 samples drawn from the multivariate gaussian found by LM above. If you run it again with more samples then the results will get even smoother.

[9]:
```
# corner plot of the posterior
# observe that it is very similar to the corner plot from the LM optimization␣
↪since this case can be roughly
# approximated as a multivariate gaussian centered on the maximum likelihood␣
↪point
param_names = list(MODEL.parameter_order())
i = 0
while i < len(param_names):
    param_names[i] = param_names[i].replace(" ", "")
    if "center" in param_names[i]:
        center_name = param_names.pop(i)
        param_names.insert(i, center_name.replace("center", "y"))
        param_names.insert(i, center_name.replace("center", "x"))
    i += 1

ser, sky = true_params()
corner_plot(
    res_nuts.chain.detach().cpu().numpy(),
    labels = param_names,
    figsize = (20,20),
    true_values = np.concatenate((sky,ser.ravel()))
)
```

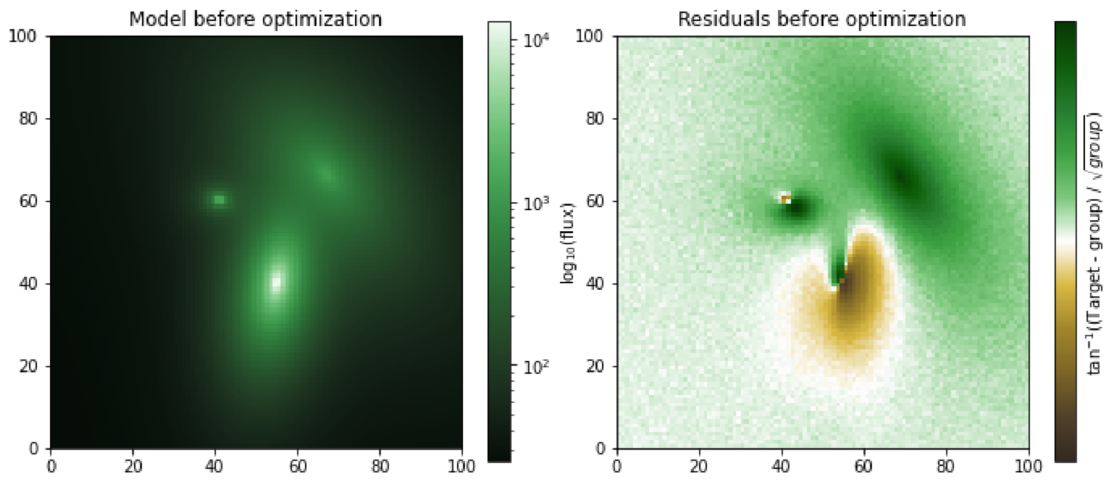


1.6 Hamiltonian Monte-Carlo (HMC)

The `ap.fit.HMC` is a simpler variant of the NUTS sampler. HMC takes a fixed number of steps at a fixed step size following Hamiltonian dynamics. This is in contrast to NUTS which attempts to optimally choose these parameters. HMC may be suitable in some cases where NUTS is unable to find ideal parameters. Also in some cases where you already know the pretty good step parameters HMC may run faster. If you don't want to fiddle around with parameters then stick with NUTS, HMC results will still have autocorrelation which will depend on the problem and choice of step parameters.

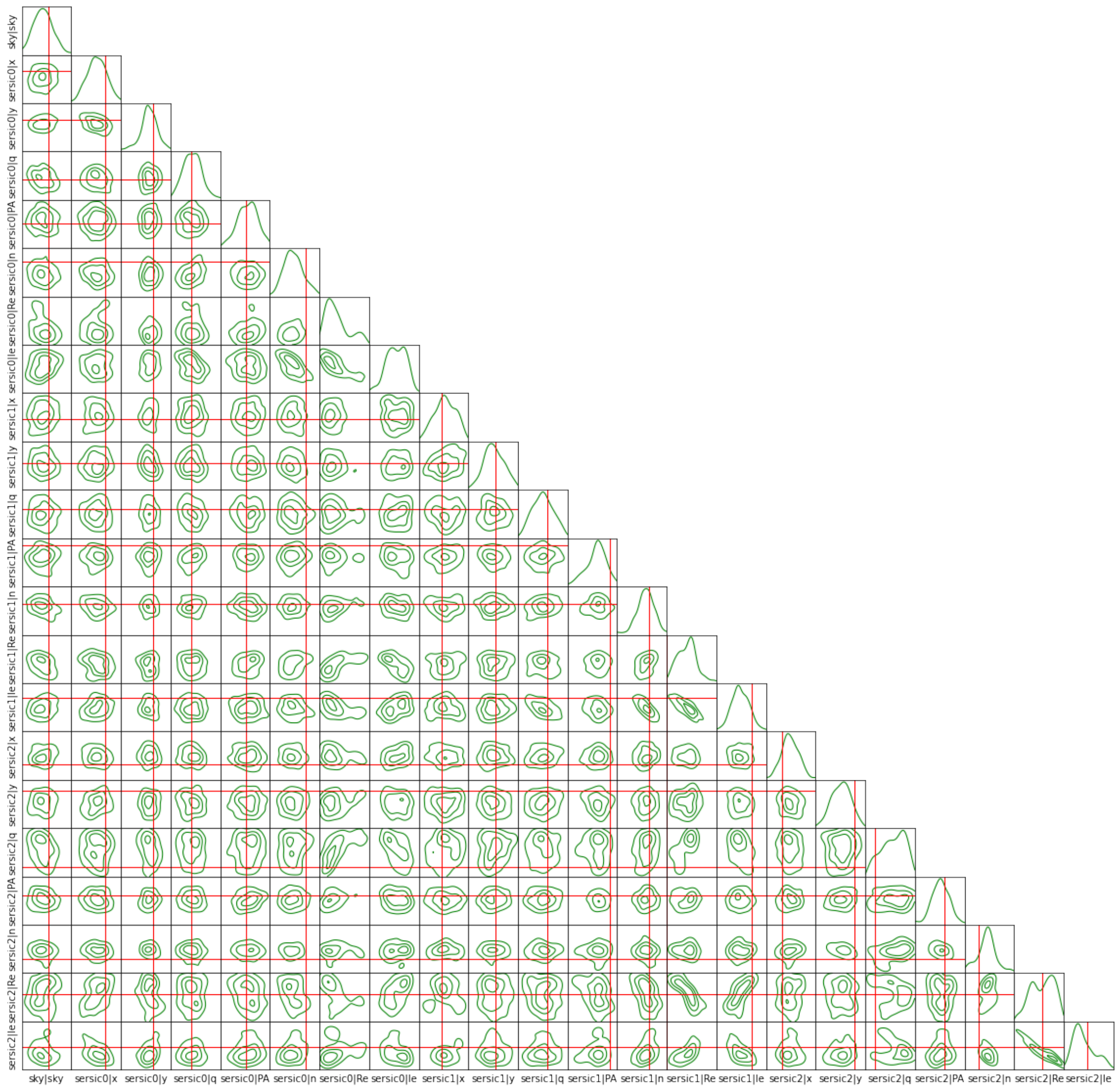
[10]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,2, figsize = (12,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
```

```
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")
plt.show()

# Use LM to start the sampler at a high likelihood location, no burn-in needed!
res1 = ap.fit.LM(MODEL).fit()

# Run the HMC sampler
res_hmc = ap.fit.HMC(MODEL, warmup = 20, max_iter = 200, epsilon = 1e-2,␣
 ↪leapfrog_steps = 20).fit()
```



```
Sample: 100%|      | 220/220 [02:26,  1.50it/s, step size=1.00e-02, acc.
prob=0.939]
```

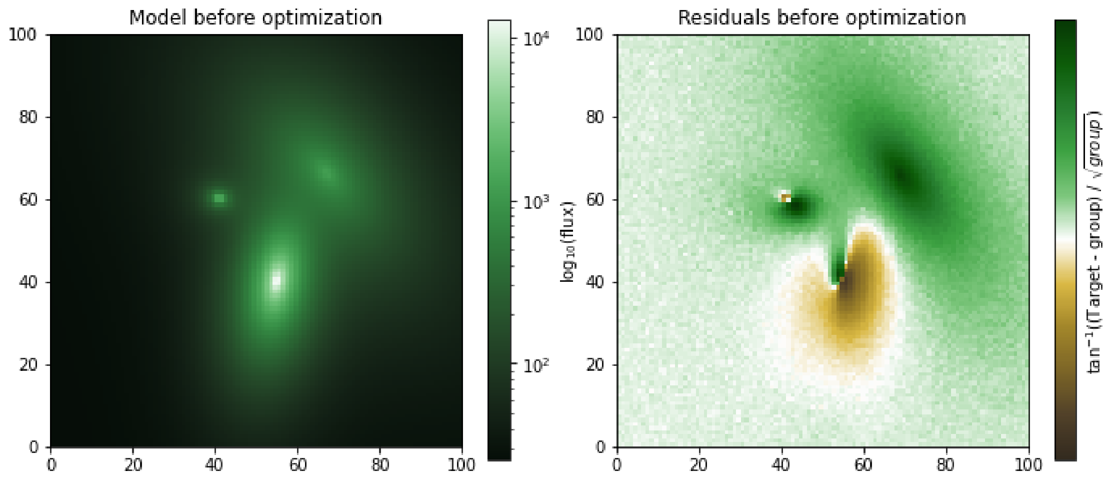
[11]:
```
# corner plot of the posterior
# note that for the HMC, 200 samples is not enough to overcome the␣
 ↪autocorrelation so the posterior has not converged
param_names = list(MODEL.parameter_order())
i = 0
while i < len(param_names):
    param_names[i] = param_names[i].replace(" ", "")
    if "center" in param_names[i]:
        center_name = param_names.pop(i)
        param_names.insert(i, center_name.replace("center", "y"))
        param_names.insert(i, center_name.replace("center", "x"))
    i += 1

ser, sky = true_params()
corner_plot(
    res_hmc.chain.detach().cpu().numpy(),
    labels = param_names,
    figsize = (20,20),
    true_values = np.concatenate((sky,ser.ravel()))
)
```



1.7 Metropolis Hastings

This is the classic MCMC algorithm using the Metropolis Hastngs accept step identified with `ap.fit.MHMCMC`. One can set the gaussian random step scale and then explore the posterior. While this technically always works, in practice it can take exceedingly long to actually converge to the posterior. This is because the step size must be set very small to have a reasonable likelihood of accepting each step, so it never moves very far in parameter space. With each subsequent sample being very close to the previous sample it can take a long time for it to wander away from its starting point. In the example below it would take an extremely long time for the chain to converge. Instead of waiting that long, we demonstrate the functionality with 5000 steps, but suggest using NUTS for any real world problem. Still, if there is something NUTS can't handle (a function that isn't differentiable) then MHMCMC can save the day (even if it takes all day to do it).

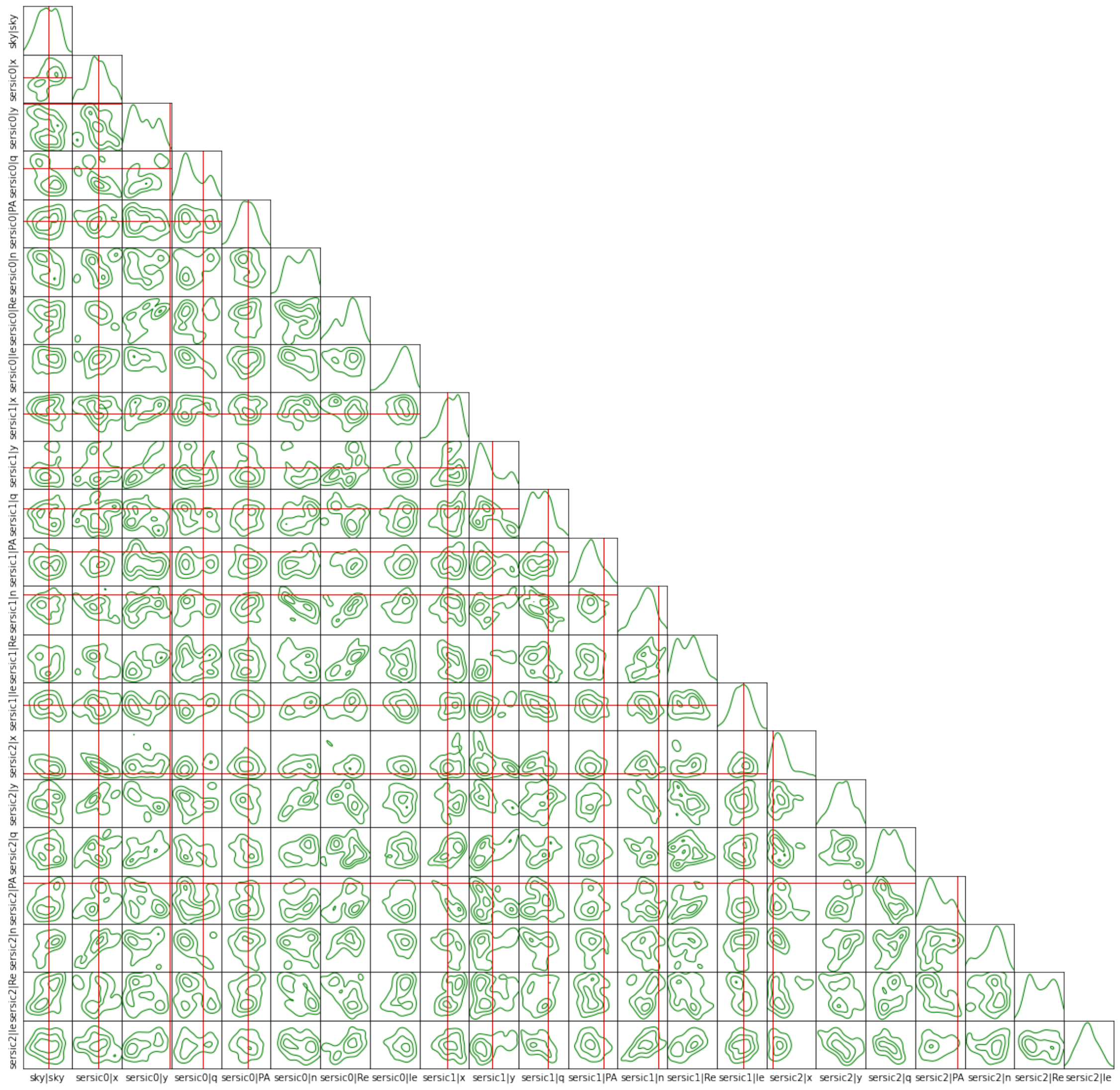
[12]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,2, figsize = (12,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals before optimization")
plt.show()

# Use LM to start the sampler at a high likelihood location, no burn-in needed!
res1 = ap.fit.LM(MODEL).fit()

# Run the HMC sampler
res_mh = ap.fit.MHMCMC(MODEL, verbose = 1, max_iter = 5000, epsilon = 1e-4,␣
 ↪report_after = np.inf).fit()
```



```
100%|          | 5000/5000 [02:41<00:00, 31.04it/s]

Acceptance: 0.765999972820282
```

[13]:
```
# corner plot of the posterior
# note that, even 5000 samples is not enough to overcome the autocorrelation so␣
 ↪the posterior has not converged.
# In fact it is not even close to convergence as can be seen by the multi-modal␣
 ↪blobs in the posterior since this
```

```
# problem is unimodal (except the modes where models are swapped). It is almost␣
↪never worthwhile to use this
# sampler except as a sanity check on very simple models.
param_names = list(MODEL.parameter_order())
i = 0
while i < len(param_names):
    param_names[i] = param_names[i].replace(" ", "")
    if "center" in param_names[i]:
        center_name = param_names.pop(i)
        param_names.insert(i, center_name.replace("center", "y"))
        param_names.insert(i, center_name.replace("center", "x"))
    i += 1

ser, sky = true_params()
corner_plot(
    res_mh.chain[::10], # thin by a factor 10 so the plot works in reasonable␣
↪time
    labels = param_names,
    figsize = (20,20),
    true_values = np.concatenate((sky,ser.ravel()))
)
```


sky|sky
sersic0|x
sersic0|y
sersic0|q
sersic0|PA
sersic0|n
sersic0|Re
sersic0|le
sersic1|x
sersic1|y
sersic1|q
sersic1|PA
sersic1|n
sersic1|Re
sersic1|le
sersic2|x
sersic2|y
sersic2|q
sersic2|PA
sersic2|n
sersic2|Re
sersic2|le

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```