FittingMethods

May 10, 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]:
```
%load_ext autoreload
%autoreload 2

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]:
```
# 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],
    ])
```

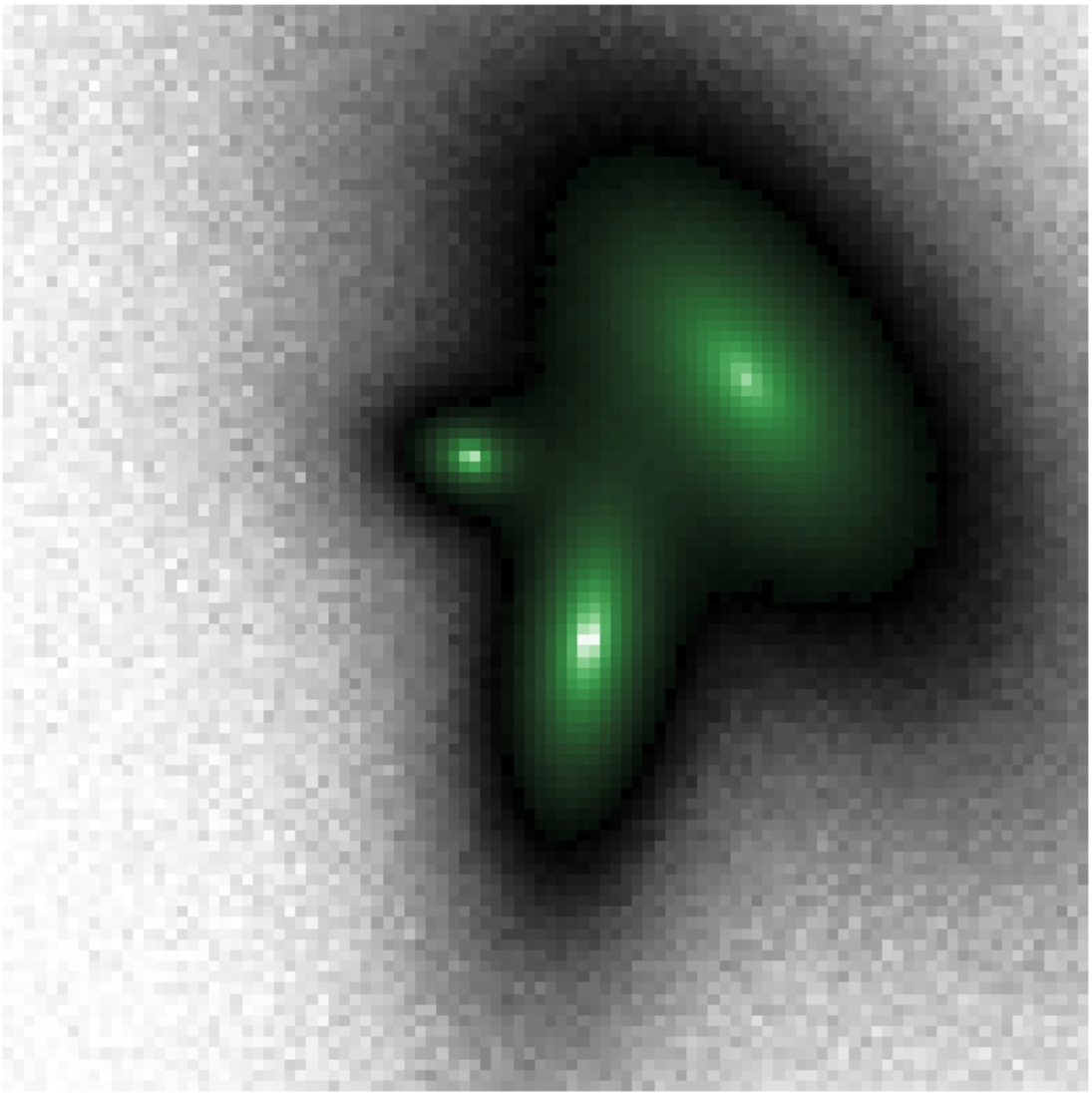
```
    return sersic_params, sky_param

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).

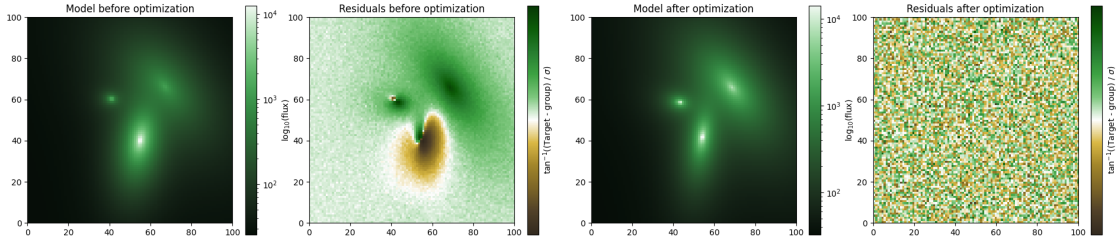
The main drawback of LM is its memory consumption which goes as $\mathcal{O}(PN)$ where P is the number of pixels and N is the number of parameters.

[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
```

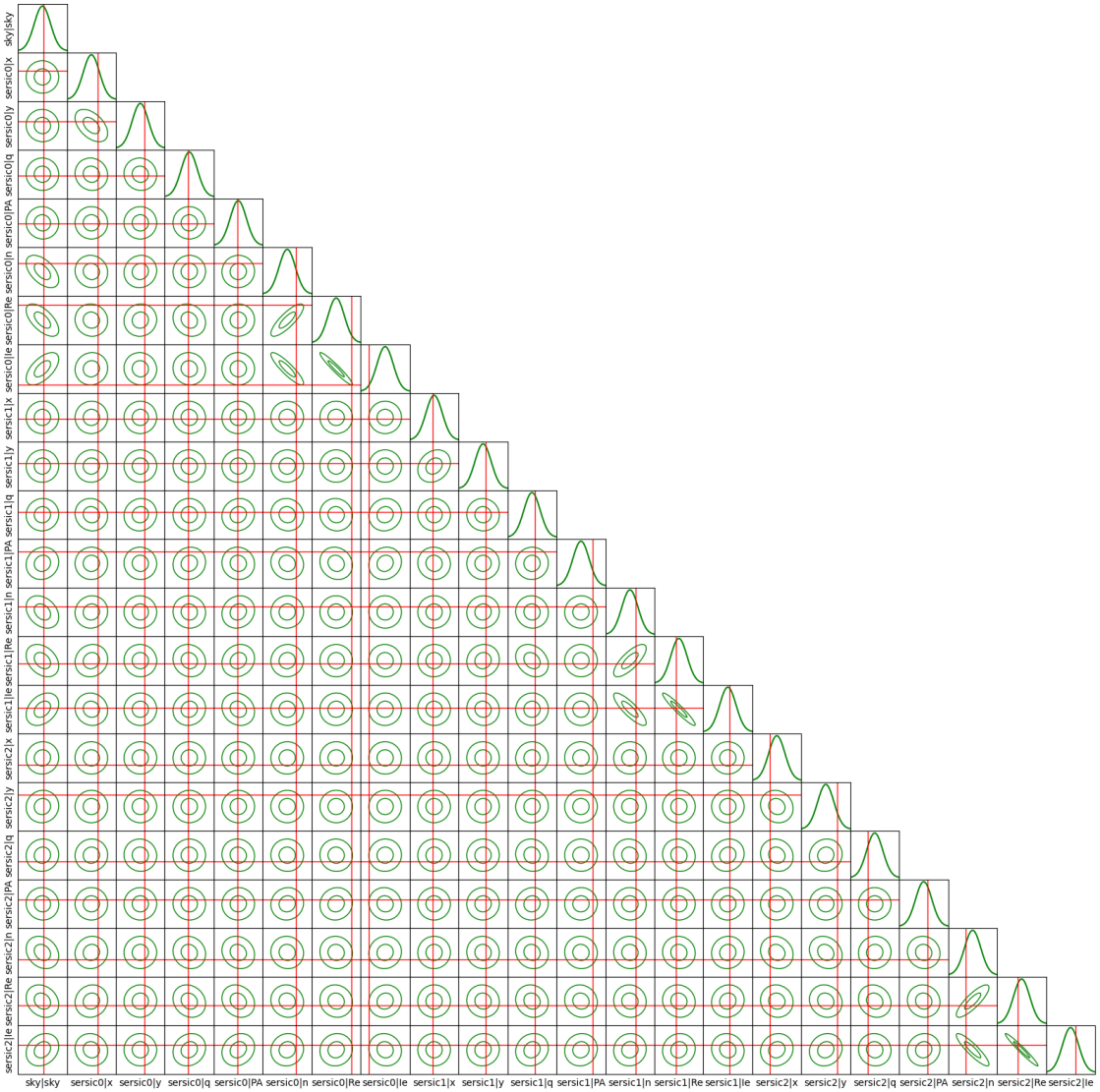
```
L: 0.000508078732798025
---------iter---------
LM loss: 1.4800231508207649
accept
L: 5.6453192533113885e-05
---------iter---------
LM loss: 1.012878922723731
accept
L: 6.272576948123765e-06
---------iter---------
LM loss: 1.0127841836614357
accept
L: 6.969529942359739e-07
---------iter---------
LM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

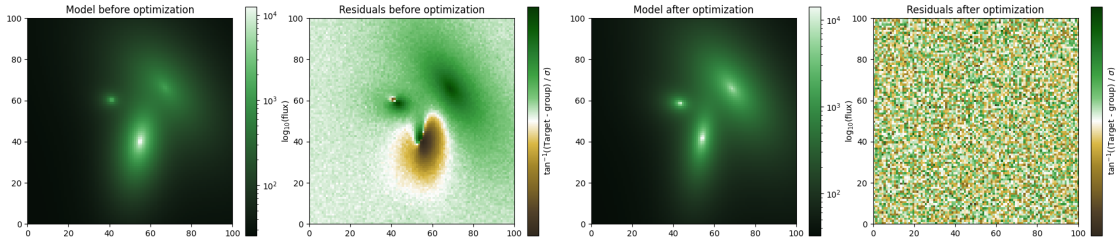
By only fitting one model at a time it is possible to get caught in a local minimum. For this reason it can be good to mix-and-match the iterative optimizers so they can help each other get unstuck.

[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.

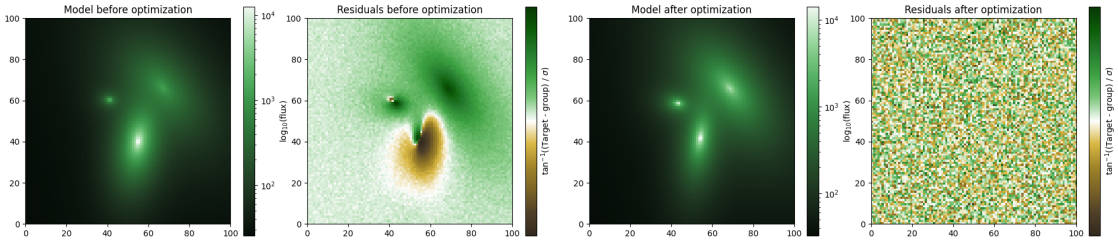
[6]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,4, figsize = (24,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals 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-------
['140494018054176:0' '140494018053312:0' '140494017844752:0'
 '140494017843888:0' '140494017844560:1' '140494017844080:0'
 '140494021100880:1' '140494021100976:0' '140494021100400:0'
 '140494021100496:0' '140494021100304:0']
chunk loss: 67.54358401797523
['140494018053312:1' '140494017843552:0' '140494017843936:0'
 '140494017844992:0' '140494017844560:0' '140494017844128:0'
 '140494017846432:0' '140494017846576:0' '140494021099920:0'
 '140494021100880:0' '140494021100688:0']
chunk loss: 13.294215936761598
Loss: 13.294215936761598
--------iter-------
['140494018053312:1' '140494017844752:0' '140494017843552:0'
 '140494017843936:0' '140494017844992:0' '140494017844128:0'
 '140494021099920:0' '140494021100880:0' '140494021100880:1'
 '140494021100496:0' '140494021100688:0']
chunk loss: 7.1462980411085555
['140494018054176:0' '140494018053312:0' '140494017843888:0'
 '140494017844560:0' '140494017844560:1' '140494017844080:0'
 '140494017846432:0' '140494017846576:0' '140494021100976:0'
 '140494021100400:0' '140494021100304:0']
```

```
chunk loss: 2.0545763781894406
Loss: 2.0545763781894406
--------iter-------
['140494018054176:0' '140494018053312:1' '140494017843888:0'
 '140494017844560:0' '140494017844560:1' '140494017844080:0'
 '140494017844128:0' '140494021100976:0' '140494021100496:0'
 '140494021100688:0' '140494021100304:0']
chunk loss: 1.1467447313672774
['140494018053312:0' '140494017844752:0' '140494017843552:0'
 '140494017843936:0' '140494017844992:0' '140494017846432:0'
 '140494017846576:0' '140494021099920:0' '140494021100880:0'
 '140494021100880:1' '140494021100400:0']
chunk loss: 1.0295993879380652
Loss: 1.0295993879380652
--------iter-------
['140494018053312:1' '140494017843552:0' '140494017843888:0'
 '140494017844560:1' '140494017846432:0' '140494017846576:0'
 '140494021100880:1' '140494021100976:0' '140494021100400:0'
 '140494021100496:0' '140494021100688:0']
chunk loss: 1.0197555225371666
['140494018054176:0' '140494018053312:0' '140494017844752:0'
 '140494017843936:0' '140494017844992:0' '140494017844560:0'
 '140494017844080:0' '140494017844128:0' '140494021099920:0'
 '140494021100880:0' '140494021100304:0']
chunk loss: 1.0155122885253918
Loss: 1.0155122885253918
--------iter-------
['140494018054176:0' '140494018053312:0' '140494017843552:0'
 '140494017844080:0' '140494017844128:0' '140494017846576:0'
 '140494021099920:0' '140494021100880:1' '140494021100976:0'
 '140494021100400:0' '140494021100688:0']
chunk loss: 1.0154651301795083
['140494018053312:1' '140494017844752:0' '140494017843936:0'
 '140494017844992:0' '140494017843888:0' '140494017844560:0'
 '140494017844560:1' '140494017846432:0' '140494021100880:0'
 '140494021100496:0' '140494021100304:0']
chunk loss: 1.0140261191340136
Loss: 1.0140261191340136
--------iter-------
['140494018053312:0' '140494017844752:0' '140494017843552:0'
 '140494017844992:0' '140494017843888:0' '140494017844560:0'
 '140494017844560:1' '140494017844128:0' '140494021100880:0'
 '140494021100880:1' '140494021100976:0']
chunk loss: 1.0139280970014959
['140494018054176:0' '140494018053312:1' '140494017843936:0'
 '140494017844080:0' '140494017846432:0' '140494017846576:0'
 '140494021099920:0' '140494021100400:0' '140494021100496:0'
 '140494021100688:0' '140494021100304:0']
```

```
chunk loss: 1.0133177241208697
Loss: 1.0133177241208697
--------iter-------
['140494018053312:0' '140494017844752:0' '140494017843552:0'
 '140494017844992:0' '140494017844080:0' '140494017844128:0'
 '140494017846432:0' '140494021099920:0' '140494021100880:1'
 '140494021100400:0' '140494021100496:0']
chunk loss: 1.0132302861168139
['140494018054176:0' '140494018053312:1' '140494017843936:0'
 '140494017843888:0' '140494017844560:0' '140494017844560:1'
 '140494017846576:0' '140494021100880:0' '140494021100976:0'
 '140494021100688:0' '140494021100304:0']
chunk loss: 1.013204723506911
Loss: 1.013204723506911
--------iter-------
['140494018054176:0' '140494018053312:0' '140494017843552:0'
 '140494017844560:0' '140494017844560:1' '140494017844128:0'
 '140494017846432:0' '140494021100976:0' '140494021100400:0'
 '140494021100496:0' '140494021100688:0']
chunk loss: 1.0132033196352714
['140494018053312:1' '140494017844752:0' '140494017843936:0'
 '140494017844992:0' '140494017843888:0' '140494017844080:0'
 '140494017846576:0' '140494021099920:0' '140494021100880:0'
 '140494021100880:1' '140494021100304:0']
chunk loss: 1.0129966665428807
Loss: 1.0129966665428807
--------iter-------
['140494017844752:0' '140494017843936:0' '140494017844992:0'
 '140494017844560:0' '140494017844560:1' '140494017846576:0'
 '140494021099920:0' '140494021100880:1' '140494021100976:0'
 '140494021100400:0' '140494021100304:0']
chunk loss: 1.0129959337448946
['140494018054176:0' '140494018053312:0' '140494018053312:1'
 '140494017843552:0' '140494017843888:0' '140494017844080:0'
 '140494017844128:0' '140494017846432:0' '140494021100880:0'
 '140494021100496:0' '140494021100688:0']
chunk loss: 1.012895822409717
Loss: 1.012895822409717
--------iter-------
['140494017844752:0' '140494017844992:0' '140494017844560:1'
 '140494017844080:0' '140494017844128:0' '140494017846576:0'
 '140494021099920:0' '140494021100976:0' '140494021100400:0'
 '140494021100496:0' '140494021100688:0']
chunk loss: 1.0128902051768192
['140494018054176:0' '140494018053312:0' '140494018053312:1'
 '140494017843552:0' '140494017843936:0' '140494017843888:0'
 '140494017844560:0' '140494017846432:0' '140494021100880:0'
 '140494021100880:1' '140494021100304:0']
```

```
chunk loss: 1.0128868415733763
Loss: 1.0128868415733763
```



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: 274.9826791730693
loss: 268.7658186828636
loss: 263.16515170646977
```

```
loss: 257.7561735800684
loss: 252.41252588817505
loss: 247.09945433567142
loss: 241.84976057641862
loss: 236.60943234985226
loss: 231.41908748894244
loss: 226.29032266822134
loss: 221.2239525548909
loss: 216.246688423731
loss: 211.36802742187953
loss: 206.57763954002928
loss: 201.8648079524657
loss: 197.22900311089796
loss: 192.68957667411513
loss: 188.24715682077075
loss: 183.90107795974612
loss: 179.6504174648846
loss: 175.49404102438166
loss: 171.43104880367895
loss: 167.4603260063568
loss: 163.5809138798234
loss: 159.79192972536882
loss: 156.09256657361968
loss: 152.54756549834295
loss: 149.04328323984114
loss: 145.62761776839744
loss: 142.30073948545865
loss: 139.06203955981474
loss: 135.90341907128123
loss: 132.83955561801773
loss: 129.8611778038245
loss: 126.9705142336376
loss: 124.1656087386438
loss: 121.44605466658703
loss: 118.81731101573708
loss: 116.26646663226623
loss: 113.75448508978747
loss: 111.36983432652858
loss: 109.10952004259512
loss: 106.88541967351618
loss: 104.73936811830976
loss: 102.66999206855091
loss: 100.67292646401364
loss: 98.74932231874648
loss: 96.89842007372604
loss: 95.11676487620794
loss: 93.40269098278428
loss: 91.75481043002017
```

```
loss: 90.17072154610224
loss: 88.64821068025043
loss: 87.00605527650818
loss: 85.78128022703504
loss: 84.43227329918555
loss: 83.13516766544939
loss: 81.88925377905423
loss: 80.69265602680676
loss: 79.54172458451515
loss: 78.43700786934149
loss: 77.37651910150704
loss: 76.35571855421338
loss: 75.37461017652632
loss: 74.43105769892225
loss: 73.52318210758962
loss: 72.6493810281627
loss: 71.79296569499803
loss: 71.0061606638469
loss: 70.22340654887347
loss: 69.46728857803556
loss: 68.73846819352829
loss: 68.04079436573379
loss: 67.36386429014235
loss: 66.70503634952377
loss: 66.07985310102497
loss: 65.45932649418697
loss: 64.86552756843759
loss: 64.29051499748269
loss: 63.733002606964135
loss: 63.19183575684664
loss: 62.666584691634
loss: 62.156029473634
loss: 61.65931849050767
loss: 61.17586498055403
loss: 60.7048769845157
loss: 60.24575821835526
loss: 59.79755635302382
loss: 59.36032993951064
loss: 58.9325274249284
loss: 58.51395047503969
loss: 58.10349027087297
loss: 57.70178820179139
loss: 57.30778020760356
loss: 56.92086237017434
loss: 56.54072022726654
loss: 56.16649921246664
loss: 55.79785773016144
loss: 55.43405697712038
```

```
loss: 55.074424395057804
loss: 54.71881592880388
loss: 54.36547926202495
loss: 54.0150156550857
loss: 53.666465030044535
loss: 53.31943798162176
loss: 52.97345256222373
loss: 52.62873601996399
loss: 52.28497332509948
loss: 51.94273121152192
loss: 51.60184937869282
loss: 51.26236875941924
loss: 50.924636239289235
loss: 50.58854832265049
loss: 50.254028599384924
loss: 49.921337251683475
loss: 49.59035965620207
loss: 49.257553746222555
loss: 48.929789113883835
loss: 48.603704763839524
loss: 48.279261768385915
loss: 47.95617705502694
loss: 47.634516488753384
loss: 47.31292757670279
loss: 46.99395548259934
loss: 46.676435427868675
loss: 46.359810221079975
loss: 46.04433160705052
loss: 45.73003811513957
loss: 45.41695728706876
loss: 45.105145579838336
loss: 44.794537749611834
loss: 44.48531653536567
loss: 44.17736248531001
loss: 43.87082815674179
loss: 43.56376609493391
loss: 43.260114399996056
loss: 42.95817687974504
loss: 42.65772985707648
loss: 42.3589224989674
loss: 42.06161334336228
loss: 41.76843608732674
loss: 41.47482260464484
loss: 41.18315265378407
loss: 40.89316077906059
loss: 40.60490035281988
loss: 40.31810361458717
loss: 40.032941201275605
```

```
loss: 39.74862043821234
loss: 39.46621955717126
loss: 39.18436104477242
loss: 38.902963528066664
loss: 38.62197337706313
loss: 38.33943347329723
loss: 38.05804914973965
loss: 37.77642357216347
loss: 37.495908318658
loss: 37.21252689198116
loss: 36.914664275695976
loss: 36.65416898816013
loss: 36.3800020515158
loss: 36.104743993934434
loss: 35.824207973409656
loss: 35.53106360005715
loss: 35.23308520122693
loss: 34.954023127390045
loss: 34.64227326018239
loss: 34.35073319690323
loss: 34.06115731432978
loss: 33.77413271096113
loss: 33.489610753595215
loss: 33.218813893628166
loss: 32.93840711989259
loss: 32.66050559786907
loss: 32.38440010130887
loss: 32.11095822918484
loss: 31.827803146918185
loss: 31.55634628756711
loss: 31.286325717928385
loss: 31.018438733824397
loss: 30.752384870234604
loss: 30.488154583997694
loss: 30.225096749103066
loss: 29.96429208080024
loss: 29.705167469509824
loss: 29.448182473476933
loss: 29.19236115147716
loss: 28.938200950058118
loss: 28.684381673723358
loss: 28.433504027371693
loss: 28.184198529407546
loss: 27.93648377293351
loss: 27.690507948069317
loss: 27.44623452809159
loss: 27.203665905380475
loss: 26.962647717316443
```

```
loss: 26.72396461924977
loss: 26.487349907512648
loss: 26.251050016253508
loss: 26.030427350856606
loss: 25.80010494628867
loss: 25.573550196997886
loss: 25.349818219272667
loss: 25.128964730422254
loss: 24.89985585239572
loss: 24.684553516724304
loss: 24.470549293156296
loss: 24.25997267947144
loss: 24.05142488240385
loss: 23.844477549079823
loss: 23.639812912127766
loss: 23.436936496627727
loss: 23.23557736484448
loss: 23.035818326358285
loss: 22.8376082813879
loss: 22.640952870605897
loss: 22.445803492375095
loss: 22.252192401226317
loss: 22.060016522721096
loss: 21.86932724400969
loss: 21.680061589442403
loss: 21.492349058463827
loss: 21.306092920112334
loss: 21.121587280532413
loss: 20.938361193928824
loss: 20.764395437248787
loss: 20.58421323030186
loss: 20.405550361714955
loss: 20.228342562771918
loss: 20.052640455271085
loss: 19.873246416084253
loss: 19.700326111292803
loss: 19.525980609421378
loss: 19.356083469058287
loss: 19.18745027288751
loss: 19.01984416419076
loss: 18.85485166371165
loss: 18.690435597787193
loss: 18.530167304998894
loss: 18.36839205718152
loss: 18.20792445150235
loss: 18.048797667796233
loss: 17.890949587405824
loss: 17.73431579372602
```

```
loss: 17.57805633898602
loss: 17.424847321536976
loss: 17.26928644808798
loss: 17.117759712673106
loss: 16.966518225377396
loss: 16.819239529116608
loss: 16.67130028297942
loss: 16.521746270363995
loss: 16.37906663744859
loss: 16.234606313727603
loss: 16.091671483728252
loss: 15.94978075008697
loss: 15.809081289763233
loss: 15.669697951494415
loss: 15.53145700666336
loss: 15.394415689393384
loss: 15.234281739538924
loss: 15.095685411540863
loss: 14.958156038740013
loss: 14.821703097195405
loss: 14.686282274182073
loss: 14.551953114331269
loss: 14.41855018232853
loss: 14.286305620921867
loss: 14.15508112408484
loss: 14.024914747542992
loss: 13.895799428790884
loss: 13.767805997666047
loss: 13.640920212314631
loss: 13.515310224697451
loss: 13.390727916390748
loss: 13.2670502729837
loss: 13.144386637141432
loss: 13.021079739821495
loss: 12.898751605115532
loss: 12.775971860618812
loss: 12.6533701265509
loss: 12.5314284677993
loss: 12.411414759065888
loss: 12.291786627540565
loss: 12.173662411638045
loss: 12.055990277521014
loss: 11.939557098133452
loss: 11.8232974320818
loss: 11.70845801366627
loss: 11.593773201417966
loss: 11.480482388881907
loss: 11.367411117058618
```

```
loss: 11.255760528623442
loss: 11.144387130883608
loss: 11.03451801865143
loss: 10.925163461132152
loss: 10.816621063664826
loss: 10.708297517170115
loss: 10.599805497740322
loss: 10.491286193935096
loss: 10.38339228921434
loss: 10.276457696132553
loss: 10.170587692456253
loss: 10.065827849520868
loss: 9.981550617439263
loss: 9.878675483948783
loss: 9.77676510982068
loss: 9.675810812041425
loss: 9.575796061623238
loss: 9.47670315847178
loss: 9.378509571960457
loss: 9.281210360842218
loss: 9.184817087977892
loss: 9.089294554939007
loss: 8.994663442517329
loss: 8.900956988278471
loss: 8.808060330108532
loss: 8.716174063218245
loss: 8.62501339048114
loss: 8.534695261172088
loss: 8.445245012037942
loss: 8.356643360654667
loss: 8.268907201779495
loss: 8.210337622017226
loss: 8.094779302307906
loss: 8.054934676051044
loss: 7.941045909057082
loss: 7.841096036576838
loss: 7.761322250561438
loss: 7.678228121593341
loss: 7.598291518036617
loss: 7.55995936556231
loss: 7.4800428494724756
loss: 7.399461175888678
loss: 7.321293026917392
loss: 7.243706086437298
loss: 7.170990948031578
loss: 7.0899017966119535
loss: 7.016766059882673
loss: 6.936653281958307
```

```
loss: 6.864814057024108
loss: 6.7854060956462545
loss: 6.715139233865455
loss: 6.638622354267306
loss: 6.564944142401653
loss: 6.49401602177297
loss: 6.421921416114807
loss: 6.352047580510524
loss: 6.2809577428181225
loss: 6.212095477771384
loss: 6.142035512447195
loss: 6.074128966606994
loss: 6.004973126527633
loss: 5.941831569919864
loss: 5.86908923200444
loss: 5.804766435723155
loss: 5.729907961781966
loss: 5.659414146514232
loss: 5.5932200536693975
loss: 5.549415566790583
loss: 5.5000908654648395
loss: 5.451373224925978
loss: 5.403193315233512
loss: 5.355543387850087
loss: 5.3085125992734925
loss: 5.262172295990706
loss: 5.216436187716386
loss: 5.171402049707015
loss: 5.126850598543528
loss: 5.082983405744692
loss: 5.039541859513104
loss: 4.996749156117782
loss: 4.954430217939502
loss: 4.912681004226435
loss: 4.871467459965498
loss: 4.854564569001552
loss: 4.811489587601075
loss: 4.768855186249674
loss: 4.72667608692351
loss: 4.684970351371756
loss: 4.643771186465577
loss: 4.603124024609717
loss: 4.563037369109402
loss: 4.5235587673953574
loss: 4.4847180045317465
loss: 4.44653786623721
loss: 4.4090067707812155
loss: 4.372140018465294
```

```
loss: 4.335899613493655
loss: 4.300296773011867
loss: 4.265285313290834
loss: 4.230882462572589
loss: 4.197060083280932
loss: 4.163785540664354
loss: 4.1310442818157505
loss: 4.098832557618403
loss: 4.067115403830294
loss: 4.035865678729581
loss: 4.005094228089605
loss: 3.974850046809177
loss: 3.9454485416381564
loss: 3.9166089537493494
loss: 3.8882814118378706
loss: 3.8603989429563987
loss: 3.8329542036580775
loss: 3.8059338240762677
loss: 3.779350067030999
loss: 3.753197915076942
loss: 3.7274491793957116
loss: 3.70177285565282
loss: 3.676885186856695
loss: 3.652408584176861
loss: 3.6283360353766096
loss: 3.6046707343666116
loss: 3.5809081044060207
loss: 3.5580178567180085
loss: 3.5355012959228325
loss: 3.5133536605588658
loss: 3.491570970783724
loss: 3.470409396436897
loss: 3.44933408378202
loss: 3.428623978377971
loss: 3.4082270479337966
loss: 3.3881846461204486
loss: 3.36840343321497
loss: 3.348990515269834
loss: 3.329823430277529
loss: 3.3110060775893193
loss: 3.292434208902729
loss: 3.2741928224309733
loss: 3.25618817920624
loss: 3.2385144361029687
loss: 3.2210548871042533
loss: 3.2046495039820804
loss: 3.1871153237466308
loss: 3.171344645548707
```

```
loss: 3.1542919466836334
loss: 3.13914579401313
loss: 3.1233903381624417
loss: 3.107929967859025
loss: 3.092652967429948
loss: 3.077512602613134
loss: 3.0627563803363382
loss: 3.047946972751889
loss: 3.033484649000244
loss: 3.0189568922752708
loss: 3.0046870786691007
loss: 2.9903757263020267
loss: 2.9763461893129386
loss: 2.9623060447189657
loss: 2.9486840721496286
loss: 2.9349721928335355
loss: 2.921652995788868
loss: 2.9081872207316666
loss: 2.895485565377089
loss: 2.8825081935902293
loss: 2.8699274032874746
loss: 2.857323467527648
loss: 2.8450112852749916
loss: 2.832872396488621
loss: 2.820901599149377
loss: 2.809095397331208
loss: 2.797450298581632
loss: 2.785964164579945
loss: 2.774648235695409
loss: 2.7634757943716286
loss: 2.7524589747291675
loss: 2.7415968541832765
loss: 2.730887480972063
loss: 2.720327987673016
loss: 2.7099217487639797
loss: 2.699641182864643
loss: 2.6895111016603255
loss: 2.6795038412407797
loss: 2.6696223279417186
loss: 2.659862607537
loss: 2.6502210158119404
loss: 2.6406962326952756
loss: 2.6305410143269
loss: 2.621167087453757
loss: 2.6119126337609275
loss: 2.60275979036469
loss: 2.5937158372089035
loss: 2.584772468820213
```

```
loss: 2.575952890981698
loss: 2.567229390265472
loss: 2.558611558359178
loss: 2.550100162118128
loss: 2.5416968433279092
loss: 2.5323739420803886
loss: 2.5252389370927832
loss: 2.5161719200733113
loss: 2.5082338835712488
loss: 2.50044586649146
loss: 2.492746828340044
loss: 2.485179270565836
loss: 2.477702399121712
loss: 2.4703210741396275
loss: 2.463030408889597
loss: 2.4558290660987456
loss: 2.4487101537854494
loss: 2.441673723673331
loss: 2.4347179862137054
loss: 2.4278639727407083
loss: 2.4210654746305265
loss: 2.414343687191834
loss: 2.4076975570175576
loss: 2.401126092387985
loss: 2.3946529648554398
loss: 2.388227819197178
loss: 2.381874646131376
loss: 2.3755927013074047
loss: 2.369381328154895
loss: 2.3632399967267554
loss: 2.3571683286996095
loss: 2.3511661530667567
loss: 2.345233558458356
loss: 2.339370952088314
loss: 2.333579106299298
loss: 2.327859131034992
loss: 2.3222123640736663
loss: 2.3166400689614535
loss: 2.3111430310222336
loss: 2.3057211395033828
loss: 2.300380557737587
loss: 2.2948843971852293
loss: 2.289895149861382
loss: 2.284534388410537
loss: 2.2796726538313843
loss: 2.274463408815594
loss: 2.2697170727433544
loss: 2.264601931130564
```

```
loss: 2.2599696812540477
loss: 2.2549671472090975
loss: 2.250232020101726
loss: 2.245550014460844
loss: 2.2409201894498
loss: 2.2363594087844016
loss: 2.2318320655995305
loss: 2.2271912524340585
loss: 2.2227447194221392
loss: 2.218367064035959
loss: 2.214028903755643
loss: 2.2097575519384147
loss: 2.205515920594872
loss: 2.2013399592246725
loss: 2.1971933762539915
loss: 2.1939463828925505
loss: 2.1890820283322943
loss: 2.185891078551242
loss: 2.181132000238561
loss: 2.178026465782425
loss: 2.1733623226584893
loss: 2.1703472211447044
loss: 2.165971874746204
loss: 2.16284589919432
loss: 2.158568003353741
loss: 2.1555384233782404
loss: 2.151344283634689
loss: 2.1484023179761964
loss: 2.1444045538463365
loss: 2.14148250712265
loss: 2.137610271813044
loss: 2.1347499515288195
loss: 2.1310143810113247
loss: 2.127201753223553
loss: 2.124645608503749
loss: 2.1209608352132436
loss: 2.118525666708385
loss: 2.11497251460825
loss: 2.1126487383637853
loss: 2.109216537696647
loss: 2.1069871877483917
loss: 2.1006955375476144
loss: 2.101869306316436
loss: 2.0956682257844537
loss: 2.097513285275557
loss: 2.090503960297567
loss: 2.0920300816675335
loss: 2.084820646962066
```

```
loss: 2.0860797560671753
loss: 2.0787197094441905
loss: 2.079143777001521
loss: 2.072486438737378
loss: 2.0730162816185302
loss: 2.0664529942954966
loss: 2.0670509204022536
loss: 2.060702274029751
loss: 2.0613281119130336
loss: 2.0579100150822636
loss: 2.0556758031184157
loss: 2.0524290386641484
loss: 2.0503438891234156
loss: 2.047250665523988
loss: 2.045050808781079
loss: 2.0432037687575253
loss: 2.0401335142099666
loss: 2.038389801211271
loss: 2.035381346432577
loss: 2.033701490208034
loss: 2.0313939940908754
loss: 2.0291124548321458
loss: 2.0267430423266024
loss: 2.024513981602067
loss: 2.022302814845577
loss: 2.020112753529762
loss: 2.0179540337830013
loss: 2.0158075234476773
loss: 2.0136917684626185
loss: 2.0115871566860437
loss: 2.0095156453057883
loss: 2.007456469993649
loss: 2.0054170815311885
loss: 2.0033971950005323
loss: 2.001396491454445
loss: 1.9993682231399281
loss: 1.9974522392974972
loss: 1.9954580226110912
loss: 1.9935287807737745
loss: 1.991617239864293
loss: 1.9897233414595143
loss: 1.9878466564526445
loss: 1.9859870399502253
loss: 1.984144144795876
loss: 1.9823177814515374
loss: 1.9805076504104395
loss: 1.9787135405252205
loss: 1.9769351813283367
```

```
loss: 1.9751723525748546
loss: 1.9734248026552703
loss: 1.9716923090624097
loss: 1.969974633428207
loss: 1.9682715547283742
loss: 1.9665915153728486
loss: 1.9649169632372545
loss: 1.963256339786438
loss: 1.9616094369673958
loss: 1.9600019813885559
loss: 1.9583818589477433
loss: 1.9567748282995385
loss: 1.9551806866087353
loss: 1.9536112965096546
loss: 1.9520423581913349
loss: 1.9504857211368405
loss: 1.9489411907892438
loss: 1.9474085819184739
loss: 1.9458877069252856
loss: 1.9443805943116248
loss: 1.9428804013014285
loss: 1.9413958456967764
loss: 1.9399200985341547
loss: 1.9384552009300087
loss: 1.93700098163458
loss: 1.9355572755419994
loss: 1.9341239171855502
loss: 1.9327163016253615
loss: 1.9313031400811482
loss: 1.9298998488589294
loss: 1.9285429042069373
loss: 1.9271580256525527
loss: 1.9257868302252867
loss: 1.9244228258851668
loss: 1.923025207786758
loss: 1.921679122703747
loss: 1.9203418628908717
loss: 1.9190133087188286
loss: 1.917693303013044
loss: 1.916381723135319
loss: 1.9150784229712463
loss: 1.913816356562116
loss: 1.9125292201081938
loss: 1.911249981417986
loss: 1.9092036031597184
loss: 1.9079425454258514
loss: 1.9066890618774037
loss: 1.9054430197502268
```

```
loss: 1.9042042955976608
loss: 1.902972763778811
loss: 1.901748305434908
loss: 1.9005308008685249
loss: 1.899320135696464
loss: 1.8981161955894614
loss: 1.8969188705145952
loss: 1.8957280510489392
loss: 1.8945436313369346
loss: 1.8933655064659762
loss: 1.8921580198639463
loss: 1.8909919429095807
loss: 1.8898318570944115
loss: 1.8886776669477714
loss: 1.8875292780989972
loss: 1.8863865983585522
loss: 1.8852495367787054
loss: 1.8841180043255898
loss: 1.8829919132997468
loss: 1.8818711777060655
loss: 1.8807557129443906
loss: 1.8796454359492483
loss: 1.8785402650849723
loss: 1.8774401201123208
loss: 1.876344922243757
loss: 1.8752545939709448
loss: 1.8741690592528801
loss: 1.8730882432249156
loss: 1.8720120725040115
loss: 1.8709404747902108
loss: 1.8698733792821944
loss: 1.8688107161745804
loss: 1.867752417184479
loss: 1.8666984149417656
loss: 1.86564864363458
loss: 1.8646030382838692
loss: 1.8635615355196309
loss: 1.8625240727296786
loss: 1.861639138871405
loss: 1.86061049935726
loss: 1.85958573671492
loss: 1.858564776074039
loss: 1.8575475656798677
loss: 1.856534037231177
loss: 1.8555241396490942
loss: 1.8545178091280106
loss: 1.8535149956853554
loss: 1.8525156394832298
```

```
loss: 1.8515196918785608
loss: 1.8505270965405678
loss: 1.849537806309721
loss: 1.8485517679916148
loss: 1.8475689360176177
loss: 1.8465892600601237
loss: 1.8456126962587038
loss: 1.844639196998069
loss: 1.8436687202992703
loss: 1.8427012212483926
loss: 1.8417366600391416
loss: 1.8407749946529983
loss: 1.838111035941641
loss: 1.8388755849538119
loss: 1.8362448581095783
loss: 1.8376995275485326
loss: 1.8344138104723022
loss: 1.8357323934696077
loss: 1.8326463353750904
loss: 1.8340106399340086
loss: 1.8300104044473093
loss: 1.8323573530223938
loss: 1.830377432637763
loss: 1.8311356143379725
loss: 1.8291846768375928
loss: 1.830198254593458
loss: 1.8292689449175636
loss: 1.8291750207183504
loss: 1.8289245040380693
loss: 1.828135423548155
loss: 1.8277587072887524
loss: 1.8267806190793558
loss: 1.8261506112375339
loss: 1.8248946514525504
loss: 1.8239880116868081
loss: 1.8224999427353776
loss: 1.8214016129924768
loss: 1.8198376920995738
loss: 1.818041401872455
loss: 1.8171625538714997
loss: 1.8155904571259223
loss: 1.8147130452232014
loss: 1.8122759122605498
loss: 1.8126576385986684
loss: 1.8102452878571818
loss: 1.8106581508341517
loss: 1.8082699774107895
loss: 1.8087133905204387
```

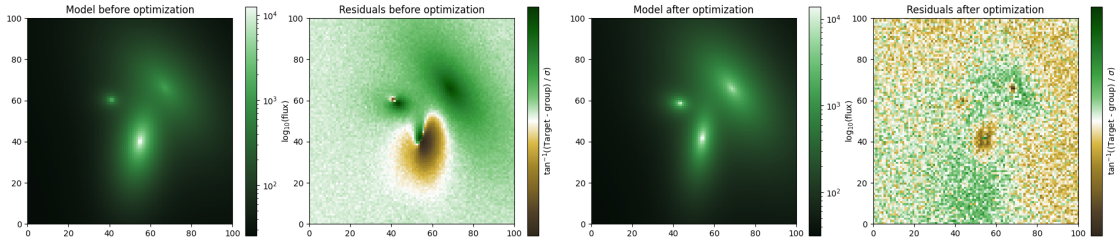
```
loss: 1.8048669519648506
loss: 1.806952010960601
loss: 1.8031283127190096
loss: 1.8052122255003915
loss: 1.8013910151393058
loss: 1.8034739793931571
loss: 1.799655647100523
loss: 1.801737773287655
loss: 1.7979228741027689
loss: 1.8000042343236058
loss: 1.7976391910302365
loss: 1.797988080727814
loss: 1.7943345526696652
loss: 1.7962961557135317
loss: 1.792645752194153
loss: 1.794605411581177
loss: 1.7909584225853337
loss: 1.7929249489219508
loss: 1.7892708887529263
loss: 1.7912357257255924
loss: 1.7875858127869013
loss: 1.789549027262439
loss: 1.7859035531124174
loss: 1.7878652232067167
loss: 1.7856270422180598
loss: 1.7862373238496514
loss: 1.7824434830056282
loss: 1.784610305524521
loss: 1.780818151584291
loss: 1.782980872953607
loss: 1.780573607738488
loss: 1.781561869073399
loss: 1.7774448164382806
loss: 1.7796085316021237
loss: 1.7758349946579097
loss: 1.7779946041799841
loss: 1.7755860237160792
loss: 1.7765881610656253
loss: 1.7724934545460262
loss: 1.7746540288542003
loss: 1.77225103599328
loss: 1.7732670009971958
loss: 1.7691900363525765
loss: 1.7716991838210612
loss: 1.7689638596978223
loss: 1.7699893723674536
loss: 1.7659284813119338
loss: 1.7684329920192012
```

```
loss: 1.7657010805722249
loss: 1.7667350915853075
loss: 1.7626896712945241
loss: 1.7651894060503757
loss: 1.7624604424634698
loss: 1.7635021186454412
loss: 1.7607845022223156
loss: 1.7618380746761646
loss: 1.7578195515777146
loss: 1.7603142303544232
loss: 1.7575963940843857
loss: 1.7586541059906058
loss: 1.7559456762363723
loss: 1.7570132311751427
loss: 1.7530192547247654
loss: 1.7555074657012333
loss: 1.75279725801152
loss: 1.7538669850174773
loss: 1.7511651705788565
loss: 1.7522435466773865
loss: 1.7482728372626457
loss: 1.7507538799360585
loss: 1.748049593445641
loss: 1.7491288228893482
loss: 1.7464324513835037
loss: 1.747519570585669
loss: 1.744829648657304
loss: 1.7459232184053268
loss: 1.7432387269730512
loss: 1.7443376959672263
loss: 1.7416578376373701
loss: 1.7427614343371507
loss: 1.7388402914643657
loss: 1.741306984231616
loss: 1.7386222932594417
loss: 1.7397235135174796
loss: 1.7370443915035543
loss: 1.738150926905906
loss: 1.7354765664072196
loss: 1.7365875856630284
loss: 1.7339174769572578
loss: 1.735032359255638
loss: 1.7323660851815696
loss: 1.7334843624582124
loss: 1.7308215883519993
loss: 1.731942899496299
loss: 1.7292833614615537
loss: 1.7304074202667887
```

```
loss: 1.7277509133703683
loss: 1.728877486307707
loss: 1.726223853758817
loss: 1.7273527446715935
loss: 1.7247018683752842
loss: 1.7258329079828305
loss: 1.7231847005303575
loss: 1.7243177392332483
loss: 1.7216721372421848
loss: 1.7228070401691284
loss: 1.7201731697335045
loss: 1.7200244357754433
loss: 1.7185739764181247
loss: 1.718453842746064
loss: 1.7158194741697506
loss: 1.7169797453758888
loss: 1.7143466091931627
loss: 1.715507248606358
loss: 1.7140545429736218
loss: 1.7139506046872974
loss: 1.7113263342991036
loss: 1.7124926257829813
loss: 1.7098702358798505
loss: 1.7110381116990292
loss: 1.7095863466881835
loss: 1.7094968336139735
loss: 1.7068816342202056
loss: 1.7080529202005414
loss: 1.7066044284412387
loss: 1.7052373440114241
loss: 1.7038855836385167
loss: 1.7037348617347987
loss: 1.7023997534915067
loss: 1.7022446861786726
loss: 1.7009136893578525
loss: 1.7001949518320099
loss: 1.6994433460883265
loss: 1.6987272592505787
loss: 1.697978351429356
loss: 1.6972551240044527
loss: 1.6965190908531016
loss: 1.6957979274927661
loss: 1.6950639680103512
loss: 1.6943447525412305
loss: 1.693612745242133
loss: 1.692895390268251
loss: 1.6921652469448938
loss: 1.6914496857195849
```

```
loss: 1.6907213407471542
loss: 1.6900075213138073
loss: 1.6892809245324314
loss: 1.6885688056185488
loss: 1.6878439176838949
loss: 1.687133465904484
loss: 1.6864102552738889
loss: 1.685701443240528
loss: 1.6849798842055679
loss: 1.684283439876287
loss: 1.6835526639149359
loss: 1.682857808388035
loss: 1.6821394823129314
loss: 1.6814354626664552
loss: 1.6807187349574333
loss: 1.6800162821849416
loss: 1.679301140040987
loss: 1.6786002436565037
loss: 1.677886676864842
loss: 1.6771873281605623
loss: 1.6764753284135416
loss: 1.6757775202728762
loss: 1.6750670809571502
loss: 1.6743708077654302
loss: 1.673661923845422
loss: 1.6729671814654725
loss: 1.6722598494555099
loss: 1.671566635275118
loss: 1.6708608533011555
loss: 1.6701691663628264
loss: 1.6694649343219505
loss: 1.668774775550957
loss: 1.6680720953852624
loss: 1.6673834679404809
loss: 1.6666823440531806
loss: 1.6659952538364307
loss: 1.6652956936921797
loss: 1.6646101500682955
loss: 1.6639121650393918
loss: 1.6632281818417525
loss: 1.6625317883888027
loss: 1.6618493853170997
loss: 1.661154606631123
loss: 1.6604738111934698
loss: 1.659780679480817
loss: 1.659653878396246
loss: 1.6584064881412686
loss: 1.6577286778923164
```

```
loss: 1.6570386499073293
loss: 1.6569135793274457
loss: 1.6556705603488184
loss: 1.6555462734472215
loss: 1.654305438211853
loss: 1.6541819467470138
loss: 1.6529432715052967
loss: 1.652271208383978
loss: 1.651587090234523
loss: 1.6514653932383765
loss: 1.6502309858712945
loss: 1.6501100826249626
loss: 1.648877798361174
loss: 1.6487576957570254
loss: 1.647527517932439
loss: 1.6474082197643356
loss: 1.646180134121588
loss: 1.6460616431255546
loss: 1.6448356377035498
loss: 1.644717956557155
loss: 1.6434940220024732
loss: 1.642831747877296
loss: 1.6421575544029579
loss: 1.6420416694932471
loss: 1.6408218544469666
loss: 1.6407067620946876
loss: 1.639489000355733
loss: 1.639374705596987
loss: 1.6381589892309798
loss: 1.638045497446173
loss: 1.6368318225170866
loss: 1.6361764798560885
loss: 1.6355092795237334
loss: 1.6353975462111838
loss: 1.6341879117233546
loss: 1.6340769679318554
```



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.

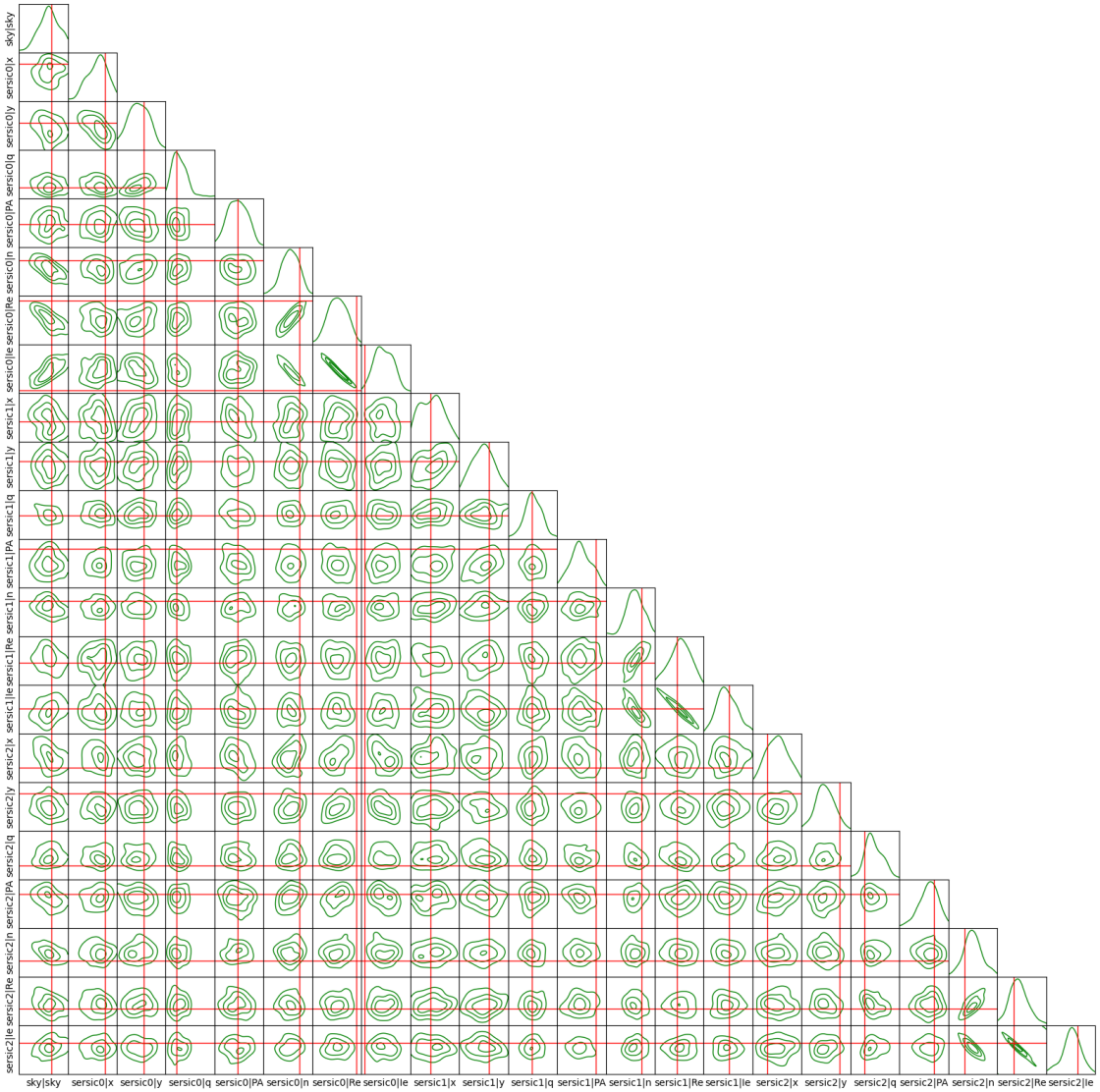
For an excellent introduction to the Hamiltonian Monte-Carlo and a high level explanation of NUTS see this review: **Betancourt 2018**

[8]:
```
MODEL = initialize_model(target, False)

# 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,
    inv_mass = res1.covariance_matrix,
).fit()
```

```
Sample: 100%| | 120/120 [02:03,  1.03s/it, step size=2.01e-01, acc. prob=0.930]
```

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)

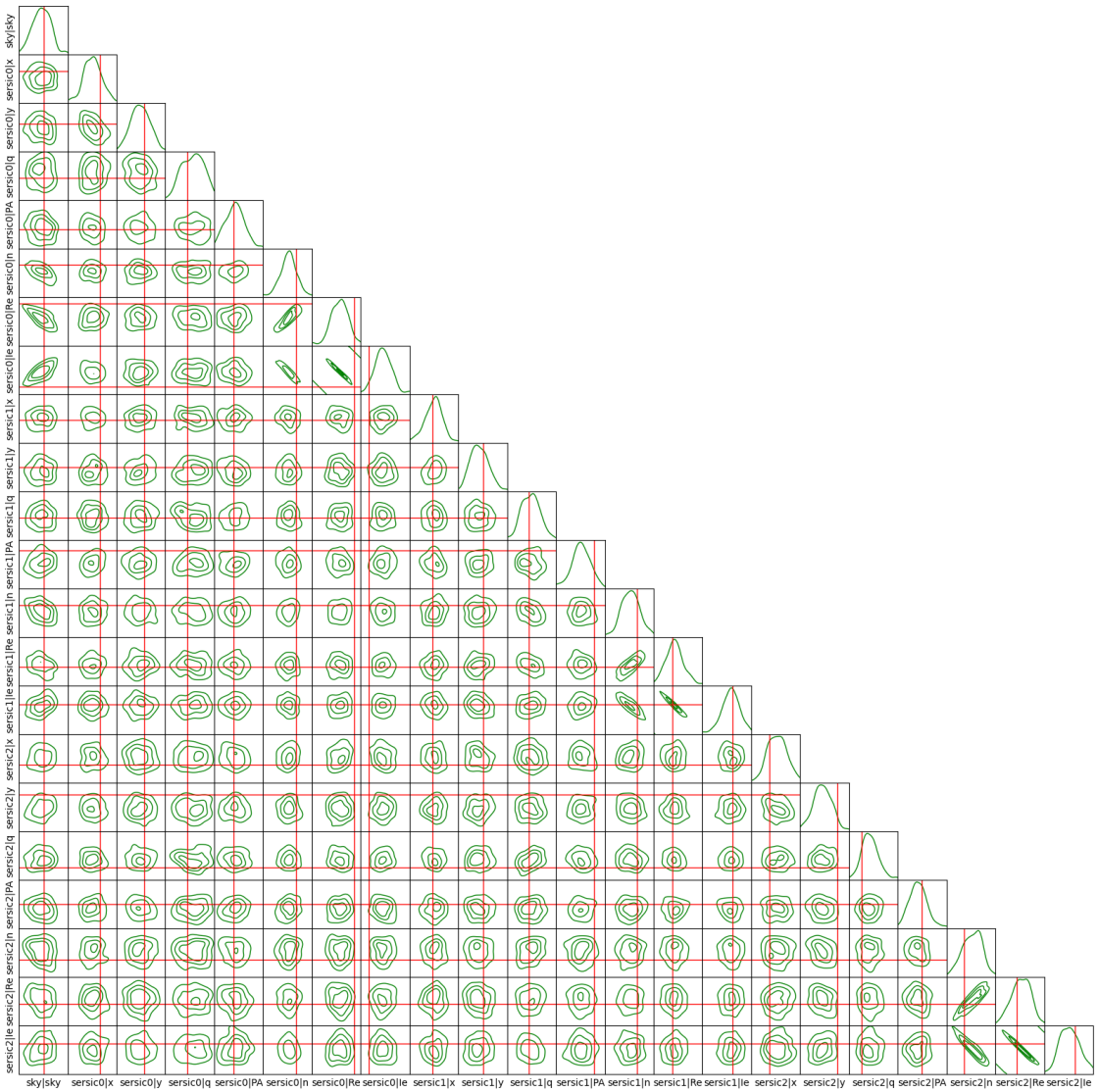
The `ap.fit.HMC` is a simpler variant of the NUTS sampler. HMC takes a fixed number of steps at a fixed step size following Hamiltonian dynamics. This is in contrast to NUTS which attempts to optimally choose these parameters. HMC may be suitable in some cases where NUTS is unable to find ideal parameters. 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)

# 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-1,
    leapfrog_steps = 20,
    inv_mass = res1.covariance_matrix,
).fit()
```

```
Sample: 100%| | 220/220 [01:32,  2.38it/s, step size=1.00e-01, acc. prob=0.958]
```

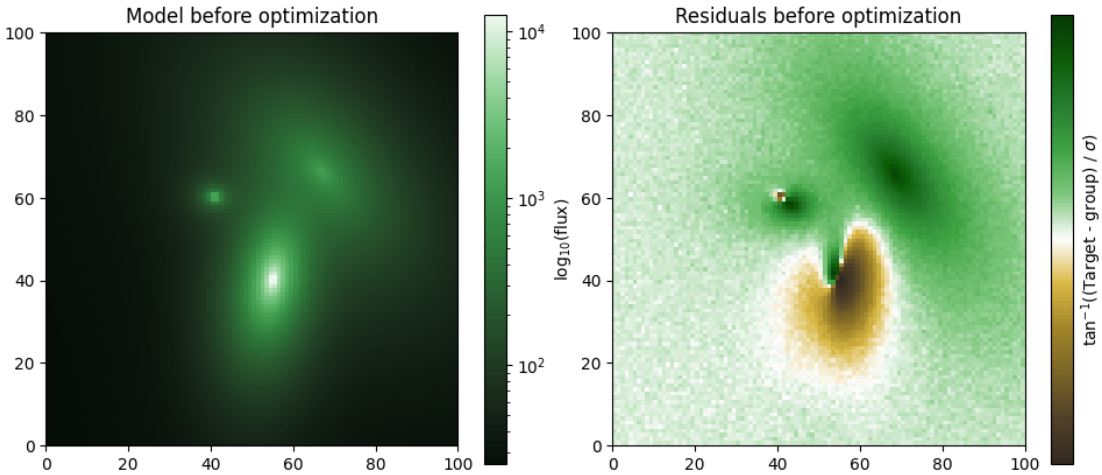
[11]:
```
# corner plot of the posterior
# note that for the HMC, 200 samples is not enough to overcome the␣
 ↪autocorrelation so the posterior has not converged
param_names = list(MODEL.parameter_order())
i = 0
while i < len(param_names):
    param_names[i] = param_names[i].replace(" ", "")
    if "center"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

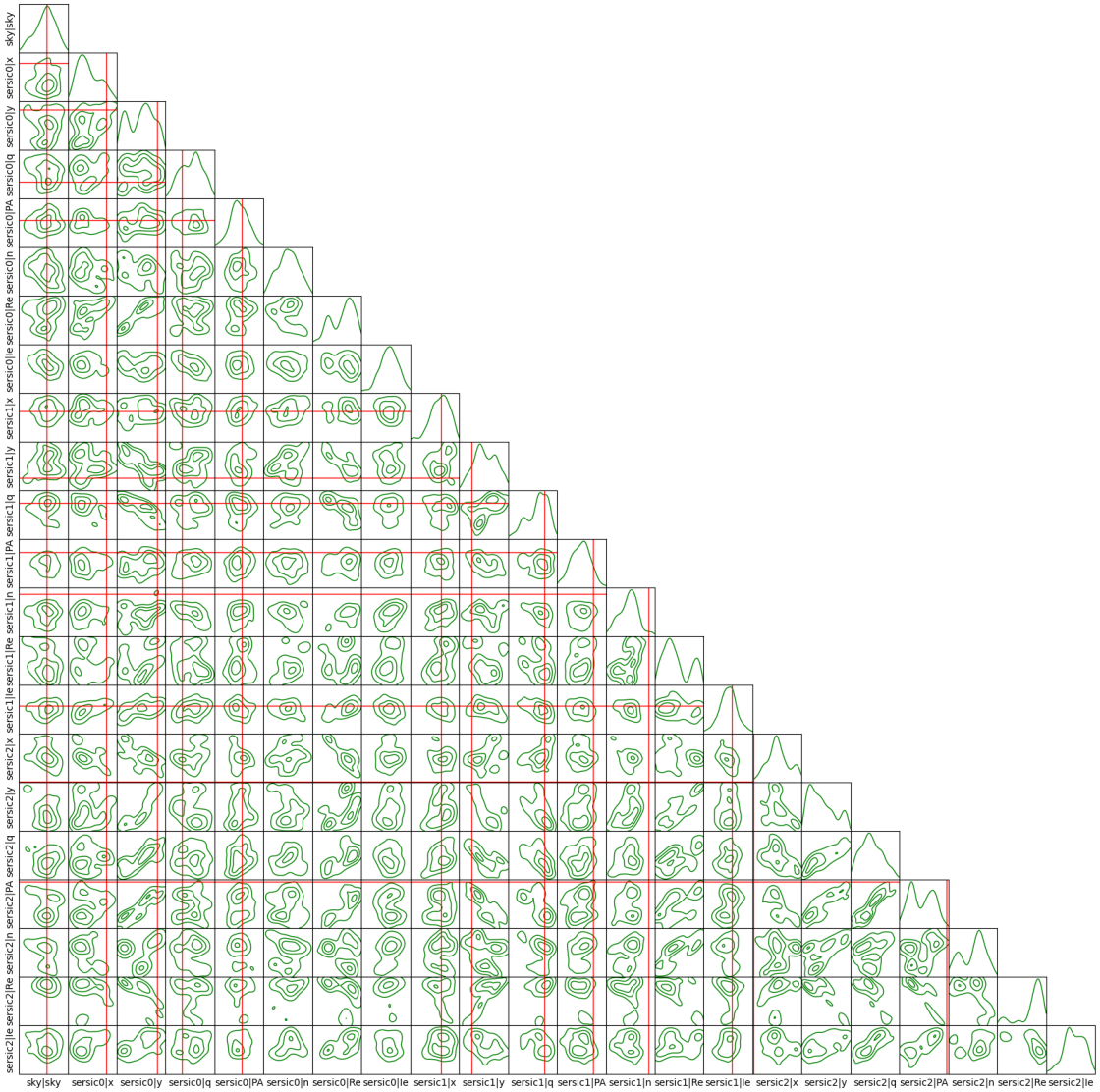
[12]:
```
MODEL = initialize_model(target, False)
fig, axarr = plt.subplots(1,2, figsize = (12,5))
plt.subplots_adjust(wspace= 0.1)
ap.plots.model_image(fig, axarr[0], MODEL)
axarr[0].set_title("Model before optimization")
ap.plots.residual_image(fig, axarr[1], MODEL, normalize_residuals = True)
axarr[1].set_title("Residuals 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 [01:51<00:00, 44.70it/s]
Acceptance: 0.7667999863624573
```

[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|Ie
sersic1|x
sersic1|y
sersic1|q
sersic1|PA
sersic1|n
sersic1|Re
sersic1|Ie
sersic2|x
sersic2|y
sersic2|q
sersic2|PA
sersic2|n
sersic2|Re
sersic2|Ie

[ ]:

[ ]:

[ ]: