universität wien

# MASTERARBEIT | MASTER'S THESIS

Titel | Title

Non-parametric Galaxy Deprojection using JAX

verfasst von | submitted by

Marco Janach BSc BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

| | |
|---|---|
| Studienkennzahl lt. Studienblatt \| Degree programme code as it appears on the student record sheet: | UA 066 861 |
| Studienrichtung lt. Studienblatt \| Degree programme as it appears on the student record sheet: | Masterstudium Astronomie |
| Betreut von \| Supervisor: | Prashin Jethwa MSc MA PhD |

# Abstract

Galaxy deprojection is a crucial step for the dynamical modeling of galaxies, which can recover many intrinsic parameters of these galaxies. Furthermore it can be used to investigate the shape of a galaxy's dark matter halo and give information about the merging of Super Massive Black Hole binaries. The main goal behind this thesis is to investigate non-parametric methods for galaxy deprojection, which is the process of transforming 2D images of galaxies into 3D models. Using the JAX framework for high performance numerical computing to improve their performance, I compared two methods to each other: Stochastic Variational Inference (SVI), which uses a probabilistic model implemented with `NumPyro` and Simulation Based Inference (SBI), which uses neural networks via the `sbi` library. Stochastic Variational Inference performed significantly better with the downside of not having realistic error estimates. On the other hand Simulation Based Inference failed to produce meaningful results, even after hyper-parameter optimization and increasing the training data significantly. Further investigations into the already promising SVI method could lead to deprojection techniques on par with state-of-the-art approaches, offering researchers more flexibility depending on the galaxies that they aim to deproject.

# Kurzfassung

Die Deprojektion von Galaxien ist ein entscheidender Schritt für die dynamische Modellierung von Galaxien, durch welche viele intrinsische Parameter dieser Galaxien gewonnen werden können. Darüber hinaus kann sie verwendet werden, um die Form des Dunkle-Materie-Halos einer Galaxie zu untersuchen und kann Informationen über die Verschmelzung supermassiver Schwarzer Löcher liefern. Das Hauptziel dieser Arbeit ist die Untersuchung nicht-parametrischer Methoden für die Deprojektion von Galaxien, d.h. die Transformation von 2D-Bildern von Galaxien zu 3D-Modelle dieser Galaxien. Ich habe zwei Methoden miteinander verglichen, deren Leistung mit Hilfe des JAX-Frameworks für numerisches Hochleistungsrechnen verbessert wurde: Stochastic Variational Inference (SVI), die ein mit `NumPyro` implementiertes probabilistisches Modell verwendet, und Simulation Based Inference (SBI), die neuronale Netze verwendet und mit Hilfe der die `sbi` Bibliothek implementiert wurde. Die Stochastic Variational Inference schnitt deutlich besser ab, allerdings mit dem Nachteil, dass keine realistischen Fehlerschätzungen vorliegen. Andererseits lieferte die Simulation Based Inference keine aussagekräftigen Ergebnisse, auch nicht nach einer Optimierung der Hyperparameter und einer erheblichen Erhöhung der Trainingsdaten. Weitere Untersuchungen der bereits vielversprechenden SVI-Methode könnten zu Deprojektionstechniken führen, die den modernsten Ansätzen ebenbürtig sind und den Forschern je nachdem, welche Galaxien sie deprojizieren möchten, mehr Flexibilität bieten.

# Contents

# 1 Introduction

When observing galaxies, we face one big problem: Our observations give us 2D images of galaxies that we can analyze, but these galaxies are intrinsically 3D objects, and we can only observe their projection onto the night sky. We can learn a lot about galaxies just by looking at their projections, but obtaining 3D models of them is often the first step for further more detailed investigations.
Galaxy deprojection is the process of transforming 2D galaxy images as we observe them on the night sky, into 3D models of its intrinsic structure.

One common use for these 3D models is, that they are the starting point for dynamical modeling methods. For example in Santucci et al. (2022) the Schwarzschild orbit-superposition method is used. As a first step a model for the gravitational potential of the galaxy has to be constructed. This potential is a combination of a stellar component, a dark-matter component, and a super-massive black hole. For the stellar component, common deprojection methods are used.
In the end this method is able to recover important intrinsic parameters for galaxies like the inner mass distribution, the intrinsic shape of the galaxy, the velocity anisotropy and the orbit circularity distribution.

Furthermore the intrinsic shape that can be obtained from galaxy deprojection can be used to obtain information about the shape of the galaxies dark matter halo. Kazantzidis et al. (2010) investigates the impact of the intrinsic galaxy shape on dark matter distribution in the halo using N-body simulations.

The triaxial shape of the stellar distribution is also an important factor for understanding the merging of Super Massive Black Hole (SMBH) binaries. In Holley-Bockelmann & Sigurdsson (2006) the authors investigate the loss cone of galaxies. This describes the population of stars and compact objects that are on an orbit that will interact with the SMBH at the center of the galaxy. They describe how the shape of the stellar distribution impacts the replenishing of the loss cone and how this could impact the merging of two SMBHs.

One of the most common methods used for the purpose of galaxy deprojection is Multi Gaussian Expansion also called MGE, which was first introduced in Cappellari (2002). This method uses the useful properties of Gaussians, that convolution and deprojections can be performed analytically.

On the other hand, non-parametric deprojection methods were not commonly used because of the large computational effort needed to run them on a reasonable timescale. With the improvement in computational power in recent years and the development of new numerical techniques, the use of these methods became more feasible. In these methods, no explicit assumptions about the distribution of the underlying data are made, which makes them more flexible and enables them to capture complex structures more easily. In de Nicola et al. (2020) the authors present a non-parametric numerical method which they apply on triaxial galaxies. It provides more flexibility and in certain tested cases, even outperforms the MGE approach.

Inspired by these results, we investigated different numerical methods for non-parametric galaxy deprojection over the course of this thesis.

# 2 Fundamentals

In the following section we will discuss fundamental topics that are relevant for the work done in this thesis. In Section 2.1 we describe how we model our density distribution, the quantity that we want to obtain after the deprojection. Following this we give a short description of how computational hardware improved over the years to motivate the use of new methods that leverage this, in Section 2.2. In Section 2.3 we continue with a brief introduction to the hardware acceleration library JAX, which we use extensively during our probabilistic modeling. Next we introduce our probabilistic modeling framework which utilizes NumPyro in Section 2.4. We describe how one can easily construct a probabilistic model using this framework. Furthermore we discuss in more detail the inference method that we use called Stochastic Variational Inference (SVI). In Section 2.5 we give a introduction to Gaussian processes, which we use in our probabilistic model to fit radially dependent variables. Lastly in Section 2.6 we describe a second method that we investigated called Simulation Based Inference (SBI). We also talk about the `sbi` library, which we use to apply this method to our problem.

## 2.1 Modeling Density and Surface Brightness

Our main goal in this thesis to investigate methods that can be used for galaxy deprojection. This means we want to transform 2D observations of galaxies, which show us their surface brightness, and transform them into 3D models of said galaxy. To accomplish this we first need to choose a modeling function for our density. We take the same approach as de Nicola et al. (2020) by assuming that our density is stratified on shells of the form:

$$r^{2-\xi(x)} = x^{2-\xi(x)} + \left(\frac{y}{p(x)}\right)^{2-\xi(x)} + \left(\frac{z}{q(x)}\right)^{2-\xi(x)} \tag{2.1}$$

where r represents our radius, p is the ratio of the intermediate and semi-major axis and q is the ratio of the semi-minor and the semi-major axis. $\xi$ represents the deformation of the elliptical density distribution.
In Figure 2.1 we can see how the p parameter influences the density distribution. A p parameter = 1 leads to a perfect circle, while decreasing it leads to a lengthening along the x-axis in the x-y plane. The q parameter has a similar effect but we get a lengthening along the x-axis in the x-z plane instead.
Figure 2.2 shows the impact of the $\xi$ parameter on the density distribution. $\xi = 0$ leads to a ellipse with no deformation. Negative $\xi$ values lead to a boxy appearance, while positive ones lead to disky appearances.

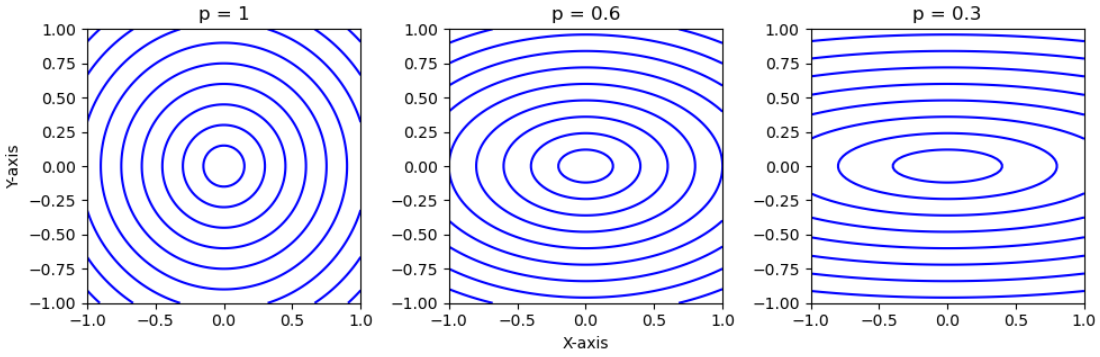


Figure 2.1: Impact of the p parameter on the density distribution. p = 1 leads to a circular distribution, while decreasing p leads to a lengthening along the x-axis in the x-y plane. These images represent slices taken from the density distribution at z =0.

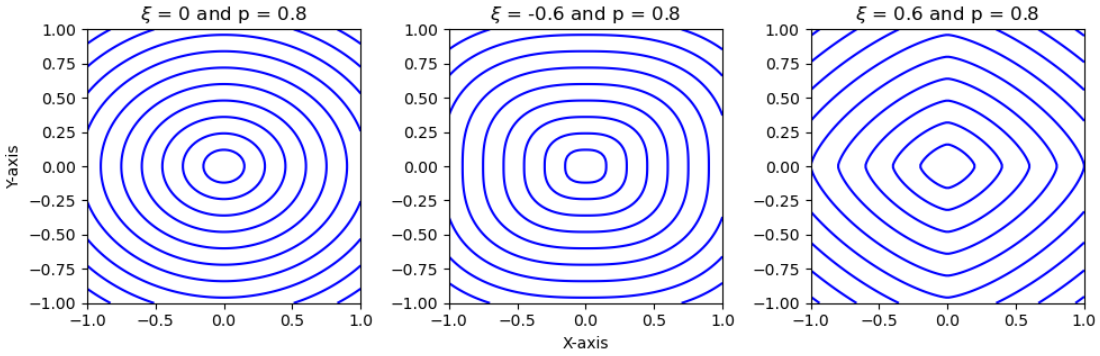


Figure 2.2: Impact of $\xi$ on the density distribution with a fixed value of p = 0.8. $\xi$ = 0 leads to a ellipse with no deformation. A negative $\xi$ leads to a boxy appearance, while a positive one leads to disky appearances.

It is important to note that the values for $p$, $q$ and $\xi$ do not have to be fixed they can also change with the radius.
Next we define our density as in de Nicola et al. (2020) with a double-power-law model with the parameters $\alpha = 2$ and $\beta = 4$ (Binney & Tremaine (2008), equation 2.64):

$$\rho(r) = \frac{\rho_0}{\left(\frac{r}{s}\right)^{\alpha}\left(1+\frac{r}{s}\right)^{\beta-\alpha}} \tag{2.2}$$

where $\rho_0$ is the central density and s is the scale radius. The parameters for $\alpha$ and $\beta$ are fixed for now but for real observations we do not know them beforehand.

Lastly as in de Nicola et al. (2020) we have two coordinate systems which are of interest to us. There is the intrinsic coordinate system of the galaxy we are observing, represented

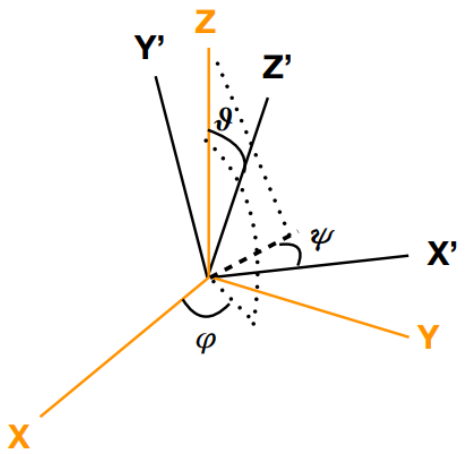


Figure 2.3: Illustration of the intrinsic coordinate system of the galaxy and the coordinate system of the plane of the sky. Three viewing angles explain the connection between the two. $\varphi$ and $\vartheta$ show the direction of the line of sight (z') in respect to the intrinsic coordinate system. $\psi$ describes a rotation around the line of sight (de Nicola et al. 2020).

by x, y and z. These correspond to the semi-major, intermediate and semi-minor axis respectively. We also use the coordinate system of the plane of the sky which represents the observed projection of the galaxy that we are trying to deproject. This system is represented by x', y' and z', where z' represents our line of sight. Furthermore we have three viewing angles connecting these two coordinate systems. $\varphi$ and $\vartheta$ representing polar coordinates which correspond to the direction of the line of sight in respect to the intrinsic coordinate system, and $\psi$ which describes the a rotation around the line of sight.

This connection between the two coordinate systems and the viewing angles is illustrated in Figure 2.3.
In the end we still need a way to confirm if the predicted density matches the original image. For this we have to project the density $\rho(x, y, z)$ along the line of sight to obtain the resulting surface brightness $\Sigma(x', y')$:

$$\Sigma(x', y') = \int \rho(r)\, dz' \tag{2.3}$$

## 2.2 High-Performance Computing

The increase in computational power in the last two decades had a large impact on how research is done today. High-performance computing (HPC) is used across many different fields in the natural sciences. But how can we quantify this improvement? The authors of Dongarra et al. (2003) use their LINPACK Benchmark as a way of determining the performance of different systems. Performance is measured by solving a general dense matrix problem $Ax = b$ in 64-bit floating-point arithmetic. The resulting floating point operations per second (Flop/s) are then used to rank the best-performing systems. Since 1993 these results are published twice a year as the TOP500 list on top500.org, which gives us a good indication how hardware performance increased over the years. In Figure 2.4 we can see how the performance of the most powerful system changed over the years:

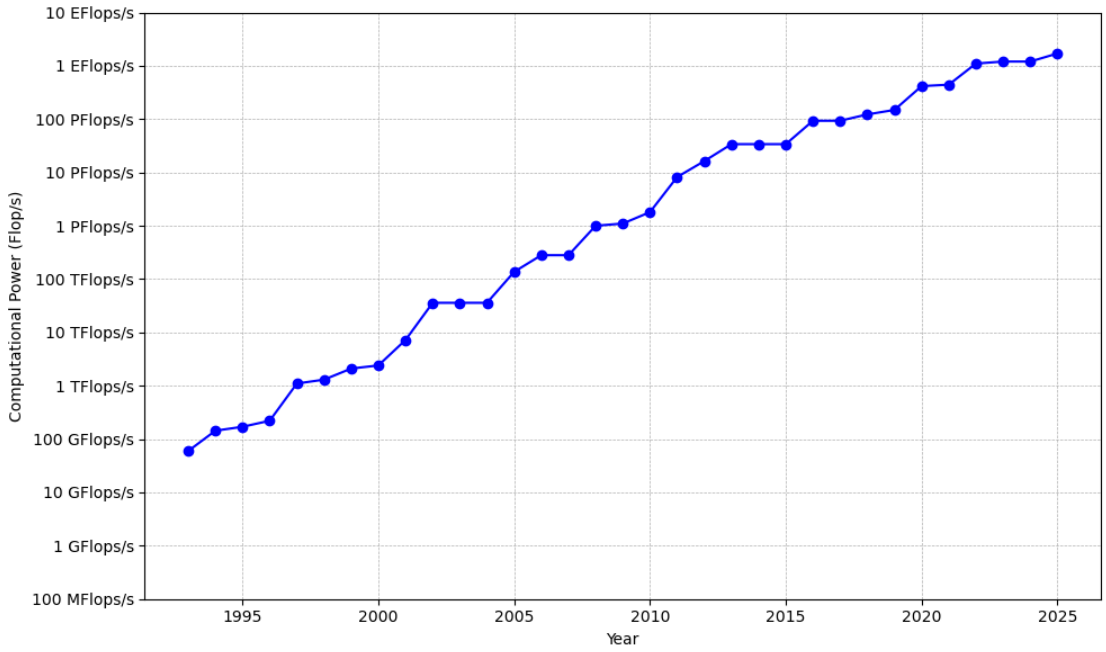


Figure 2.4: Performance of the most powerful system each year according to the TOP500 list (Dongarra et al. 2003).

We can see that HPC performance increased by a factor of $\sim 3 \times 10^7$ since 1993. It is now realistic to use methods that were not feasible in the past. But all the increase in computational power does not mean much if our software can not utilize it to its fullest. One library which was developed for this exact purpose will be discussed in the next chapter.

## 2.3 JAX

JAX (Bradbury et al. 2018) is a python library that provides a straightforward way of using high-performance hardware. Most commonly used Python and NumPy functions are already implemented in the JAX framework. JAX uses the XLA compile structure which enables it to efficiently use GPUs and other hardware accelerators. For custom functions one can use the Just-In-Time (JIT) compilation to achieve the same effect. JAX also provides functions for straightforward vectorization to increase performance. Furthermore JAX supports automatic differentiation (auto-diff), which can automatically calculate gradients for Python and NumPy functions. Higher order derivatives and backpropagation are also supported, which leads to straightforward implementations for gradient based methods (Frostig et al. 2018).
Here we present a short example of using JAX and autodiff in an optimization problem. We start with 2 sets of parameters for the axis ratios p and q. We choose a set of true parameters and create surface brightness images over the whole parameter space we defined. Then we define a function that calculates the $\text{Chi}^2$ error between the true image and our other images. Using the `jax.grad()` function we can easily obtain gradients for the $\text{Chi}^2$ error function. In Figure 2.5 we can see the gradients calculated this way:

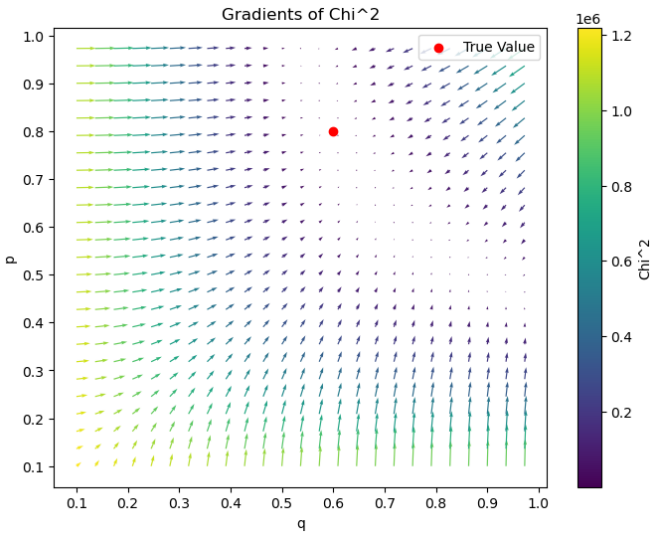


Figure 2.5: Gradients of the $\text{Chi}^2$ error calculated using the JAX `grad()` function. The x-axis shows our q values while the y-axis shows the p values of our parameter space. The red dot represents the true parameters.

These gradients can then be used for other gradient based methods like `jaxopt.ScipyBoundedMinimize` to find the parameter combination with the lowest error. We can even plot the route the optimizer takes:

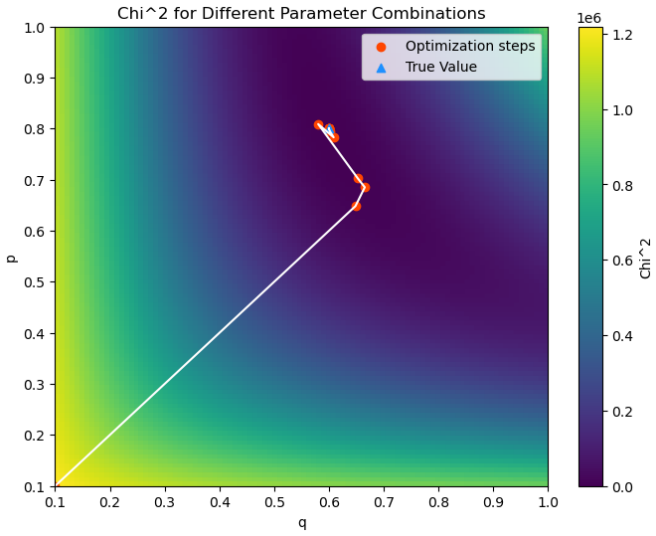


Figure 2.6: Optimization route using `jaxopt.ScipyBoundedMinimize`. The x-axis shows our q values while the y-axis shows the p values of our parameter space. The red dots represents the optimization steps while the blue triangle represents the true parameters.

## 2.4 Probabilistic Modeling

### 2.4.1 NumPyro

NumPyro is a probabilistic modeling library based on JAX (Phan et al. 2019; Bradbury et al. 2018). It utilizes three main advantages of JAX: automatic differentiation, vectorization and JIT compilation of Python and NumPy functions, which enables us to use familiar NumPy functions and arrays for our probabilistic models. Additionally we can use the acceleration provided by JAX for our probabilistic modeling.

In NumPyro we define our model using Python functions. The inputs for these models are coordinates where the model should be evaluated and a observation that we want to fit our model parameters to. We declare random variables using `numpyro.sample` statements and we can record deterministic quantities using `numpyro.deterministic` statements. There is a wide range of distributions implemented in the `numpyro.distributions` module, which we use as priors for our parameters by adding them to the `numpyro.sample` statement. The most commonly used distributions in our case are `Uniform`, `Normal` and `TruncatedNormal`. As a last step for our model we have a sample statement, which tells NumPyro to condition the model on the observed data. We also assume that our data is generated from a specific probability distribution that we provide.

The next step is to optimize our model using an inference algorithm. In our case we are using Stochastic Variational Inference (SVI), which is described in more detail in Section 2.4.2. To use it in NumPyro we need our model function, a guide function, an optimizer and a loss function. We already described how to create a model above. For the guide one can implement a custom guide function, but there is also the option to use the provided auto guides like `numpyro.infer.autoguide.AutoNormal` or `numpyro.infer.autoguide.AutoDelta`. Optimizers are also implemented in the `numpyro.optim` module, a common choice is the `Adam` optimizer. For the loss function there are multiple options implemented in the `numpyro.infer` module.

With this preparation out of the way we can create a SVI object using `numpyro.infer.SVI`. In the end we can then run the inference using `SVI.run()` and access the resulting variational parameters in the `params` attribute of the returned object. Lastly we can sample from the approximated posterior distribution using the `Predictive` function.

### 2.4.2 Stochastic Variational Inference

One of the methods we investigated for the use in galaxy deprojection was Stochastic Variational Inference (SVI).

The main goal for probabilistic inference is to obtain a posterior distribution of latent variables given a set of observation points. To obtain these distributions one needs a likelihood function, which is not tractable in many cases. To circumvent this problem Variational Inference (VI) tries to approximate the posterior distribution using a variational distribution, which contain a set of variational parameters. These parameters are then optimized to provide the best results and the resulting variational distribution can be used instead of the original posterior. To achieve this optimization a common loss function used is the so called Evidence Lower BOund (ELBO), which describes the lower bound of the marginal probability distribution of the data (Zhang et al. 2018).

One of the main issues of these traditional VI approaches is that they scale very poorly with larger datasets. To help with these problems Stochastic Variational Inference (SVI) (Hoffman et al. 2013) uses stochastic gradient descent to improve scalability.

In traditional VI the gradient of the ELBO is computed over the whole data-set, which scales poorly for large data sets. The SVI method samples a random subset (mini-batches) of data points in each iteration and uses these to calculate an estimate of the ELBO. Then the gradient is computed for this estimated ELBO, which results in a noisy estimate for the steepest ascent of the true ELBO over the whole dataset. Another important fact is that natural gradients (Amari 1985) are used instead of standard gradients which simplifies the updates for certain models. On the other hand it is important to consider the size of the used mini batches. Computational saving are only significant for mini-batches that are a lot smaller than then total number of data points. One has to balance the upside of larger batches, which result in lower gradient noise and a larger learning rate with the upside of smaller batches, which result in faster computation (Zhang et al. 2018).

## 2.5 Gaussian Processes

Gaussian Processes are an essential part of our non-parametric deprojection method, because they are used to model our latent radially dependent parameters $p(r)$, $q(r)$ and $\xi(r)$. Using a Gaussian process for each of these parameters allows us to infer their functional shape without imposing a fixed analytical function for them.

To set up these Gaussian Processes we first need to define a mean $\mu$ and a covariance matrix $\Sigma$ for their underlying Gaussian distributions. In general a mean of 0 is assumed even for distributions with a non zero mean, because it can just be added after the prediction. The covariance matrix describes the similarity between each of our training points and in the end determines the shape that predicted functions can take. To construct it we use so called kernel functions (Görtler et al. 2019). One of the most commonly used kernels is the so called radial basis function (RBF) kernel:

$$k(t, t') = \sigma^2 \exp\left(-\frac{\|t - t'\|^2}{2\ell^2}\right)$$

where $\sigma$ is our variance and $l$ is the length parameter, which describes the reach of the influence on neighbors.

There are many different kernel functions that can be used to construct the covariance matrix. They can be separated into two groups stationary kernels and non-stationary kernels. Stationary kernels like the RBF kernel, look at the relative position of two points. Non-stationary kernels on the other hand look at the absolute location of the points. By adding or multiplying these kernel functions with each other they can be combined into more complex kernels, that can be used to model more complex functions depending on the problem at hand. Other operations are possible but it is important that the kernels maintain the properties of a covariance matrix, which are positive semi-definite (Görtler et al. 2019).
If we have knowledge about the function we are trying to model we can use different kernel functions and combinations of them to create a kernel that fits the data and thus increase model performance.

Gaussian processes describe a probability distribution over possible functions. For the case where we did not observe any data yet we call it a prior distribution. Using the set up from above we can now sample possible functions from this distribution. In Figure 2.7 we can see sampled functions for the case of an RBF kernel with different kernel parameters.

This also gives us a nice way to see how the kernel influences the distribution of functions. In the first plot in Figure 2.7 we can see a function, which is rapidly changing due to the low length parameter $\ell$. Going to the right we see how the function gets smoother due to an increase in $\ell$. In the last plot we can see that the function is not as constrained as in the previous two plots due to a higher variance $\sigma$.

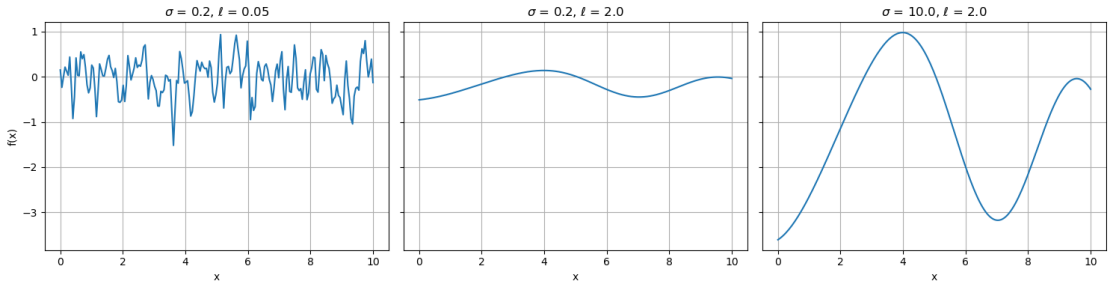


Figure 2.7: Influences of the kernel parameters $\ell$ and $\sigma$ on the functions sampled from a distribution with a RBF kernel.

As a next step we want to add observational data to our Gaussian Process. By conditioning our Gaussian process on this observed data we obtain a so called posterior distribution. Sampling from this distribution will return functions that most likely fit the functional form determined by our kernel, but also fit the data that we have observed.

There are many options to implement these Gaussian processes in Python, in our case we use the `tinygp` library.

Setting up a Gaussian process using `tinygp` is very straightforward. The most commonly used kernel functions are implemented in the `tinygp.kernels` module, which also includes the `ExpSquared` kernel that is equivalent to the RBF kernel we described before.
Then using the kernel, a mean and a position vector where it should be evaluated, one can create a Gaussian process object using `GaussianProcess()`. This object can then be conditioned using `condition()` or sampled from using `sample()`.

## 2.6 Simulation Based Inference (SBI)

The second inference method that we want to investigate further for our use case in galaxy deprojection is Simulation Based Inference (SBI). Simulations are an important part in many different fields of science, because they let us model complex processes and generate synthetic observations, which can then be used for further study. This is also an important part for our problem, where we want to model the surface brightness and density of our galaxies. One problem of these simulations is that they are not well suited for statistical inference because the likelihood function for these simulated observations is intractable in most cases (Diggle & Gratton 1984).

Simulation based inference describes the use of statistical inference on these intractable likelihood functions. There are two main methods which were historically used for this:

The first is Approximate Bayesian Computation (ABC). ABC draws parameters from a prior distribution and then simulates an observation using these inputs. In the end the observed data and the simulated data are compared using a comparison function and if they are within a predefined threshold we have an approximation of our true posterior (Rubin 1984; Beaumont et al. 2002).

The second are so called density estimation methods, that try to approximate the likelihood function by estimating the distribution of simulated data with histograms or kernel density estimation (Diggle & Gratton 1984).

These traditional methods have their own set of limitations. Firstly they need a large amount of simulated data to perform well, low sampling efficiency can be a bottleneck in many cases. Secondly these methods struggle with higher dimensional data common in many different problems. This can be circumvented by summarizing information in summary statistics but leads to worse performance due to a loss of information. Lastly in the case of ABC there is no armortization i.e most of the inference has to be repeated for new observations which is not ideal if one wants to use it on many observations (Cranmer et al. 2020).

Many of these problems can be improved with new developments in the field of simulation based inference. Advancements in the Machine Learning field, especially further development of deep neural networks helps with the problem of higher dimensional data. Using active learning i.e. using acquired knowledge to continuously guide the simulators, can help with sampling efficiency. Lastly allowing inference to directly use the internal details of the simulator leads to deeper connection between the simulation and the inference workflow, moving beyond treating the simulation like a black box (Cranmer et al. 2020).

To implement galaxy deprojection using SBI we use the `sbi` python library, which has modern implementations of different SBI approaches.

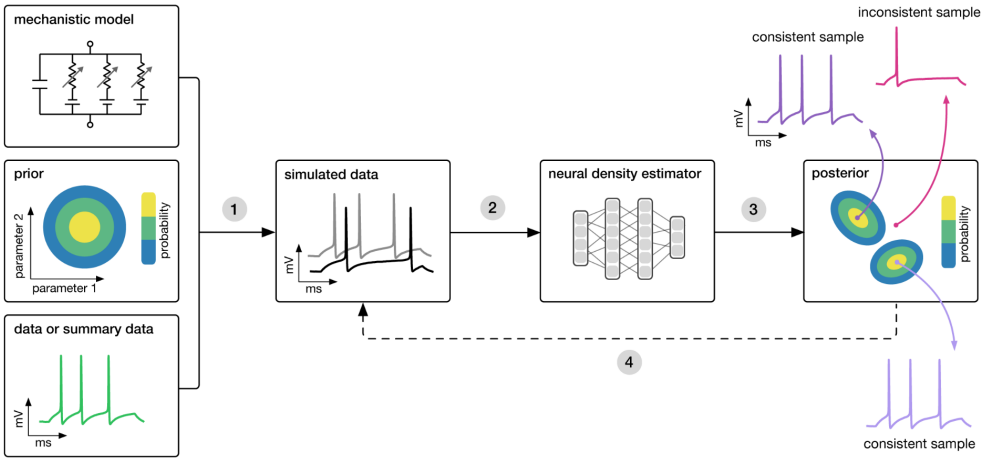


Figure 2.8: General simulation based inference process used in the `sbi` library. Image taken from Álvaro Tejero-Cantero et al. (2020). Steps 1-4 are described in detail in the text.

### 2.6.1 `sbi` a simulation based inference library

The `sbi` package is a PyTorch based library that implements SBI algorithms that use neural networks (Tejero-Cantero et al. 2020; Paszke 2019).

Three types of neural inference algorithms are currently implemented:

- Sequential Neural Posterior Estimation (SNPE), which trains a deep neural density estimator of the posterior distribution of the parameters (Greenberg et al. 2019).

- Sequential Neural Likelihood Estimation (SNLE), which trains a deep neural density estimator of the likelihood (Papamakarios et al. 2019).

- Sequential Neural Ratio Estimation (SNRE), which trains a classifier to the likelihood ratios (Durkan et al. 2020; Hermans et al. 2020).

Each of these methods needs three input parameters: a model for the simulator, some prior knowledge on the model parameters and some observed data (Álvaro Tejero-Cantero et al. 2020).

In Figure 2.8 we can then see the general process how SBI is performed in the `sbi` package.
First data is simulated by sampling input parameters from a prior (1).
Then a neural density estimator is trained using the simulated data and the corresponding input parameters (2).

Then this trained neural network is applied to observed data to find the posterior distribution, this posterior assigns a high probability to parameters, which fit the prior and the observation. In the case of the NPE method the posterior distribution is trained directly, while for the NLE and NRE methods an additional Markov-chain Monte Carlo (MCMC) sampling steps is needed to construct a posterior distribution (3).
Optionally using the posterior one can simulate new data (4) (Álvaro Tejero-Cantero et al. 2020).

# 3 Methods

In this section, we will describe the modeling setup for the two methods we investigated. In Section 3.1 we will go over in detail how we setup the probabilistic model in NumPyro. We will also take a look at how we can use Stochastic Variational Inference (SVI) to optimize our models.

In Section 3.2 we will talk about setting up the model using the `sbi` library. We will describe how we generated our training data and go into detail how we can use Simulation Based Inference (SBI) using this package.

In Section 3.3 we will describe how the test suite for both methods was created and how we evaluate the performance of our models.

## 3.1 SVI

In the following we will give an overview how to set up a model for SVI. We begin with defining our model in Section 3.1.1. Following this we describe the priors for our parameters in Section 3.1.2. We continue by choosing a guide function for our inference process in Section 3.1.3. Then we describe our inference settings in Section 3.1.4. Lastly we describe all models that were optimized in Section 3.1.5.

### 3.1.1 Model Setup

Our model has five inputs, three vectors describing the grid in which we want to evaluate the density, a radius computed for all the points on the grid and the image that we want to deproject.

There are three ways we define the parameters in our model. The simplest case is when we know the values of the parameter and do not want the model to approximate it. In this case we define the parameters in the model as the values that we expect them to be. The second way is when we know that the parameter is a constant, but we want the model to approximate it. In this case we use a `numpyro.sample` statement. The input for this is the name of the parameter and a prior distribution from which it will be sampled. We wrap this sample statement in a `numpyro.deterministic` statement, so that we can access the variables later.

The last and most complicated way to define the parameters in our model is when we want to model radially dependent parameters. For our problem this only applies to p, q and $\xi$. We use Gaussian processes to accomplish this, but instead of fitting the GP directly to our data, we use it as an intermediate layer to give us information about the functional shape of our latent parameters p(r), q(r) and $\xi$(r).

We define a Gaussian process for each of these parameters but the general setup is the same. We start by defining our kernel function. In all three cases we use an RBF kernel, where we sample our length parameter $\ell$ from a uniform distribution between 1 and 5 and the variance $\sigma$ from a normal distribution with $\mu = 0$ and $\sigma = 0.1$ truncated between $10^{-10}$ and 3.

Next we need to define the training points for the Gaussian process. For this we use a sub-sample of 1000 points in between the maximum and minimum of the radius passed to the model. Using the original radius vector leads to problems because of its large size.

We continue by defining our Gaussian process using `tinygp` and its `GaussianProcess` function, with the kernel and sub-sampled radius as input. Additionally we set the mean $\mu = 0$ and $diag = 10^{-5}$, which adds a small value to the diagonal of the covariance matrix to prevent numerical issues.

This returns a distribution from which we can sample values for p, q and $\xi$ by using `numpyro.sample`. These returned values still need to be transformed to fit their definitions, for this we use the `jax.numpy.clip` function with parameters min = $10^{-10}$ and max = 1, for p and q and min = -1 and max = 1, for $\xi$. Additionally to model the global trends of these functions we add a linear mean to the sampled values. First we sample two values $y_0$ and $y_5$ using `numpyro.sample`, with a uniform distribution between 0.001 and 1. These represent function values at x = 0 and x = 5 respectively. We use $y_0$ as the intersect and $\frac{y_5 - y_0}{5}$ as the slope for the linear mean. In the end we wrap everything in a `numpyro.deterministic` statement so that we can access it later. Since this gives us back function values for our sub-sampled radius we now use

`jax.scipy.RegularGridInterpolator` to interpolate values for the complete radius vector r.

Using all of these parameters we can finally calculate our density $\rho$. Next using `jax.numpy.sum` we project our $\rho$ along the z-axis to obtain the surface brightness $\Sigma$, which we again wrap in a `numpyro.deterministic` statement for later access.

As a last step in the model we have a last `numpyro.sample` statement, from a truncated Gaussian distribution with $\mu$ set to our surface brightness $\Sigma$ and a $\sigma$ of $0.2 \cdot \Sigma$ with a lower bound of $10^{-10}$. In this case we also set the `obs` keyword to our target image. This sample statement tells NumPyro to condition the model using the observed image, with the added assumption that it comes from a truncated normal distribution.

### 3.1.2 Priors

A comprehensive list of our model parameters, the priors we chose for them and their constraints can be found in Table 3.1.

| Parameter | Distribution | Lower Bound | Upper Bound |
|---|---|---|---|
| $q$ | Uniform | $1 \times 10^{-10}$ | 1 |
| $e$ | Uniform | $-1$ | 1 |
| $\rho_0$ | Uniform | $1 \times 10^{-10}$ | 1 |
| $s$ | Uniform | 0.2 | 5 |
| $a$ | Uniform | 1 | 5 |
| $b$ | Uniform | 1 | 5 |
| $i$ | Uniform | 0 | 360 |
| $\phi$ | Uniform | 0 | 360 |
| $\theta$ | Uniform | 0 | 360 |

Table 3.1: Model parameters with associated distributions and lower and upper bounds.

### 3.1.3 Guide Function

For the guide we initially chose `numpyro.infer.autoguide.AutoNormal`, which also allows for error estimation of our predictions. Since we had problems with the quality of the errors we decided to shift to a `numpyro.infer.autoguide.AutoDelta` guide, to speed up the fitting process.

### 3.1.4 Inference

The next step is using a inference method to optimize our model. We us Stochastic Variational Inference (SVI), which is already implemented in NumPyro. To initialize this `numpyro.infer.SVI` object we need four components. The model, a guide function, an optimizer and a loss function. The model and guide function were already described in detail in the sections above.

For the loss function we chose `numpyro.infer.Trace_ELBO`, which is a basic implementation of the Evidence Lower Bound described in Section 2.4. We use it with a sample parameter of 10, thus it will take 10 samples per gradient step to approximate the ELBO more accurately. For the optimizer we use `numpyro.optim.Adam` with a learning rate of 0.02.

After this setup we can then we run the inference using `SVI.run` using 300, 600 and 900 steps. Next we sample 1000 data points from our posterior using our guide and `numpyro.infer.Predictive`, where we include the optimized parameters we obtained from SVI.

In the end we use these posterior samples and our model together with `numpyro.infer.Predictive` to obtain our predicted observables, like the surface brightness $\Sigma$.

### 3.1.5 Model Summary

In the following we will describe all the models that we optimized i.e which parameters where radially dependent, which were left as fixed constants and which were constants chosen to be optimized.

For the most simple models we use a radially dependent p and one other free parameter, while all others are set to constant values. We do this for each combination of p with another parameter.
For the next more complex case we again start with a radially dependent p and for each of the models we add one more free parameter until all parameters are free.
The next two sets of models are constructed similarly, but this time our radially dependent parameters are p and q, and p and $\xi$ respectively.
For the last most complex set of models we again do something similar, but this time p, q and $\xi$ are all radially dependent.

## 3.2 SBI

In the following we will give an overview how to set up a model for SBI. We begin by simulating data that is used for training and testing our models in Section 3.2.1. Then we continue by defining our model in Section 3.2.2. Next we construct a convolutional neural network used for dimension reduction in Section 3.2.3. Following this we describe the priors for our parameters in Section 3.2.4. Then we talk about our inference settings in Section 3.2.5. Lastly we describe how we tried to improve the model by optimizing its hyperparameters in Section 3.2.6.

### 3.2.1 Simulating Data

We start by creating different data sets to train our models. For the initial investigation we create four different data sets with different constraints. We start with the least complex data set, where we have no radial dependence for p, q and $\xi$ and constrained viewing angles. We create a grid of parameters from which we construct a total of 10.125 images. Additionally we add Gaussian noise to emulate the uncertainty of real observations. The parameters used can be seen in Table 3.2.

| **Parameter** | **Values** |
|---|---|
| $e$ | $[-1.0,\ -0.5,\ 0.0,\ 0.5,\ 1.0]$ |
| $p$ | $[0.1,\ 0.325,\ 0.55,\ 0.775,\ 1.0]$ |
| $q$ | $[0.1,\ 0.325,\ 0.55,\ 0.775,\ 1.0]$ |
| $\rho_0$ | $[0.1,\ 1.0,\ 10.0]$ |
| $s$ | $[0.5,\ 1.0,\ 3.0]$ |
| $a$ | $[1.0,\ 2.0,\ 3.0]$ |
| $b$ | $[2.0,\ 3.0,\ 4.0]$ |

Table 3.2: Parameter values used for training data generation.

For the second data set we now use radially dependent p, q and $\xi$, while still keeping the viewing angles constrained. We create 20.736 images with the parameters shown in Table 3.3.

For the third set we create our data with radially dependent p, q and $\xi$ and free viewing angles for a total of 20.736 images. The parameters used can be seen in Table 3.4.

For the fourth data set we create a total of 20.736 images with no radial dependence in p, q and $\xi$ and free viewing angles. The parameters used can be seen in Table 3.5.

For our last investigation where we want to see the impact of a larger training set. So we create 41.472 images with no radial dependence in p, q and $\xi$ while keeping the viewing angles constrained. The parameters used can be seen in Table 3.6.

| Parameter | Values |
|---|---|
| $e_k$ | [0.0, 0.0, 0.0, 0.22, −0.22] |
| $e_d$ | [0.0, 1.0, −1.0, −1.0, 1.0] |
| $p_a$ | [0.0091, 0.0016, 0.0, 0.0, 0.0, 0.0] |
| $p_b$ | [−0.1209, −0.0212, 0.0, 0.0, 0.0, 0.0] |
| $p_c$ | [0.4476, 0.0771, 0.0, 0.0, −0.088, 0.088] |
| $p_d$ | [0.0776, 0.5425, 0.2, 0.8, 0.9, 0.1] |
| $q_a$ | [0.0091, 0.0016, 0.0, 0.0, 0.0, 0.0] |
| $q_b$ | [−0.1209, −0.0212, 0.0, 0.0, 0.0, 0.0] |
| $q_c$ | [0.4476, 0.0771, 0.0, 0.0, −0.088, 0.088] |
| $q_d$ | [0.0776, 0.5425, 0.2, 0.8, 0.9, 0.1] |
| $\rho_0$ | [0.1, 1.0, 10.0] |
| $s$ | [0.5, 1.0, 3.0] |
| $a$ | [1.0, 2.0, 3.0] |
| $b$ | [2.0, 3.0, 4.0] |

Table 3.3: Parameter values used for training data generation. $e_k$ represents the slope of a linear function and $e_d$ represents the intersection. $p_a$, $p_b$, $p_c$ and $p_d$ represent the coefficients of a polynomial of third degree. The same applies for $q$.

| Parameter | Values |
|---|---|
| $e_k$ | [0.0, 0.22, −0.22] |
| $e_d$ | [0.0, −1.0, 1.0] |
| $p_a$ | [0.0091, 0.0016, 0.0, 0.0] |
| $p_b$ | [−0.1209, −0.0212, 0.0, 0.0] |
| $p_c$ | [0.4476, 0.0771, −0.088, 0.088] |
| $p_d$ | [0.0776, 0.5425, 0.9, 0.1] |
| $q_a$ | [0.0091, 0.0016, 0.0, 0.0] |
| $q_b$ | [−0.1209, −0.0212, 0.0, 0.0] |
| $q_c$ | [0.4476, 0.0771, −0.088, 0.088] |
| $q_d$ | [0.0776, 0.5425, 0.9, 0.1] |
| $\rho_0$ | [1.0, 10.0] |
| $s$ | [1.0, 3.0] |
| $a$ | [2.0, 3.0] |
| $b$ | [3.0, 4.0] |
| $i$ | [0.0, 45.0, 90.0] |
| $\phi$ | [0.0, 45.0, 90.0] |
| $\theta$ | [0.0, 45.0, 90.0] |

Table 3.4: Parameter values used for training data generation. $e_k$ represents the slope of a linear function and $e_d$ represents the intersection. $p_a$, $p_b$, $p_c$ and $p_d$ represent the coefficients of a polynomial of third degree. The same applies for $q$.

| Parameter | Values |
|---|---|
| $e$ | $[-1.0,\ 0.0,\ 1.0]$ |
| $p$ | $[0.1,\ 0.4,\ 0.7,\ 1.0]$ |
| $q$ | $[0.1,\ 0.4,\ 0.7,\ 1.0]$ |
| $\rho_0$ | $[1.0,\ 10.0]$ |
| $s$ | $[1.0,\ 3.0]$ |
| $a$ | $[2.0,\ 3.0]$ |
| $b$ | $[3.0,\ 4.0]$ |
| $i$ | $[0.0,\ 45.0,\ 90.0]$ |
| $\phi$ | $[0.0,\ 45.0,\ 90.0]$ |
| $\theta$ | $[0.0,\ 45.0,\ 90.0]$ |

Table 3.5: Parameter values used for training data generation.

| Parameter | Values |
|---|---|
| $e$ | $[-1.0,\ -0.71,\ -0.43,\ -0.14,\ 0.14,\ 0.43,\ 0.71,\ 1.0]$ |
| $p$ | $[0.1,\ 0.24,\ 0.39,\ 0.53,\ 0.68,\ 0.82,\ 0.96,\ 1.0]$ |
| $q$ | $[0.1,\ 0.24,\ 0.39,\ 0.53,\ 0.68,\ 0.82,\ 0.96,\ 1.0]$ |
| $\rho_0$ | $[0.1,\ 1.0,\ 10.0]$ |
| $s$ | $[0.5,\ 1.0,\ 3.0]$ |
| $a$ | $[1.0,\ 2.0,\ 3.0]$ |
| $b$ | $[2.0,\ 3.0,\ 4.0]$ |

Table 3.6: Parameter values used for training data generation.

### 3.2.2 Model Setup

We define a simple model which takes the parameters that we want to optimize as an input. Using these inputs we calculate the density over our coordinate grid and then project the density along the z-axis using `jax.numpy.sum`. As a last step we transform our image to a `torch.tensor` so that we can use it in our inference process.

### 3.2.3 Convolutional Neural Network

We also define a convolutional neural network that transforms our image into a vector for easier processing during inference. For this we use `CNNEmbedding` implemented in `sbi`. We set the parameters of the embedding net as described in Table 3.7.

Here the `input_shape` describes the shape of our image, `in_channels` describes the color channels, `out_channels_per_layer` describes the number of convolutional out channels for each layer, `num_conv_layers` describes the number of convolutional layers,

| Parameter | Value |
|---|---|
| `input_shape` | (100, 100) |
| `in_channels` | 1 |
| `out_channels_per_layer` | 6 |
| `num_conv_layers` | 1 |
| `num_linear_layers` | 1 |
| `output_dim` | 8 |
| `kernel_size` | 5 |
| `pool_kernel_size` | 8 |

Table 3.7: CNN configuration parameters.

`num_linear_layers` describes the number of fully connected layers, `output_dim` describes the output dimension of the final layer, `kernel_size` describes the kernel size for the convolutional layers and `pool_size` is the size of the pool operation after the final layer.

### 3.2.4 Priors

For our priors we use `sbi.utils.BoxUniform`, which creates uniform distributions for our parameters. They can be seen in Table 3.8

| Parameter | Lower Bound | Upper Bound |
|---|---|---|
| $e_k$ | $-0.4$ | 0.4 |
| $e_d$ | $-1$ | 1 |
| $p_a$ | 0 | 0.01 |
| $p_b$ | $-0.15$ | 0 |
| $p_c$ | $-0.1$ | 0.5 |
| $p_d$ | 0.1 | 1 |
| $q_a$ | 0 | 0.01 |
| $q_b$ | $-0.15$ | 0 |
| $q_c$ | $-0.1$ | 0.5 |
| $q_d$ | 0.1 | 1 |
| $\rho_0$ | $1 \times 10^{-3}$ | 10 |
| $s$ | 0.5 | 3 |
| $a$ | 1 | 3 |
| $b$ | 2 | 4 |
| $i$ | 0 | 360 |
| $\phi$ | 0 | 360 |
| $\theta$ | 0 | 360 |

Table 3.8: Model parameter list with lower and upper bounds. $e_k$ represents the slope of a linear function and $e_d$ represents the intersection. $p_a$, $p_b$, $p_c$ and $p_d$ represent the coefficients of a polynomial of third degree. The same applies for $q$.

### 3.2.5 Inference

To create our neural posterior estimator we then use `sbi.neural_nets.posterior_nn`. As inputs we choose the Masked Autoregressive Flow (MAF) model and the embedding network we created before. We can then setup the inference process by creating a `sbi.inference.NPE` Neural Posterior Estimation object, using the priors and the posterior neural network we define earlier.

To train our model we use the `append_simulations` method to add our training data and train it using the `train` method with a batch size of 256. At last we can create our posterior using `build_posterior`.

To evaluate our model at different test points we can then use the `set_default_x` method to sample the posterior using our test inputs.

For each of our initial four data sets, we pick 10.000, 5.000 and 1.000 images at random and then split them into training and testing sets with an 80/20 split. We do this to compare the impact of different sized data sets.

### 3.2.6 Hyperparameter Optimization

In the end we try to optimize the parameters of the neural posterior estimator. We change the `hidden_features` option between 50, 150 and 300 features. We also change the `num_transforms` option, which describes the number of transformations for the MAF model between 10, 50 and 100.

We train these models with 5000 trainings samples from our fifth data set. We then choose the hyperparameter combination, which leads to the lowest image score on our testing data. Finally we train a model with these optimized parameters on the largest dataset, consisting of 40.000 training samples and 1.000 test samples, to see how a larger dataset impacts performance.

## 3.3 Test Suite and Performance Metrics

### 3.3.1 SVI Test Suite

We begin by creating the test suite for the SVI method. Starting with simpler models where all of our model parameters are constants except for p, which is dependent on r. In the case of p we are using six different functions to model its radial dependence. We use two polynomials of degree 3, one with a steeper incline and one, which is flatter. Additionally, we use two linear functions, one increasing from 0 to 1 and one decreasing from 1 to 0. Our last two functions are constant over our radial range, one at a value of p = 0.8 and one at p = 0.2. All of these functions are illustrated in Figure 3.1.

We also generate more complex images where p and q are radially dependent. In this case, we use the same functions for q that were used for p. Lastly, our most complex images are created by also introducing radial dependence to the $\xi$ parameter, which we model with a linear increase between -1 and 1.

Additionally we add Gaussian noise to our images to emulate the uncertainty of real observations.

In Figure 3.2 we can see images created for the most complex case of radially dependent p, q and $\xi$. The layout is the same as in Figure 3.1 and the position corresponds to which model function was used for p and q.

### 3.3.2 SBI Test Suite

For the SBI method we choose a subset of the simulated data to test the performance of our models. For the initial four data sets consisting of 10.000, 5.000 and 1.000 images we chose 20% of the data at random. For the largest dataset of 41.000 images we chose 1.000 images at random to test the model.

### 3.3.3 Performance Metrics

To check the performance of our optimized models we do the following: For constant parameters we use the absolute difference between the true and the fitted parameters. We sum all of these parameter scores up to obtain our input recovery score. For the radially dependent parameters we calculate the difference between the true function and the modeled function along the radius. For the images we use the sum of squared differences to obtain a measure of similarity between them, which we call image score. Adding these scores we obtain a total score.

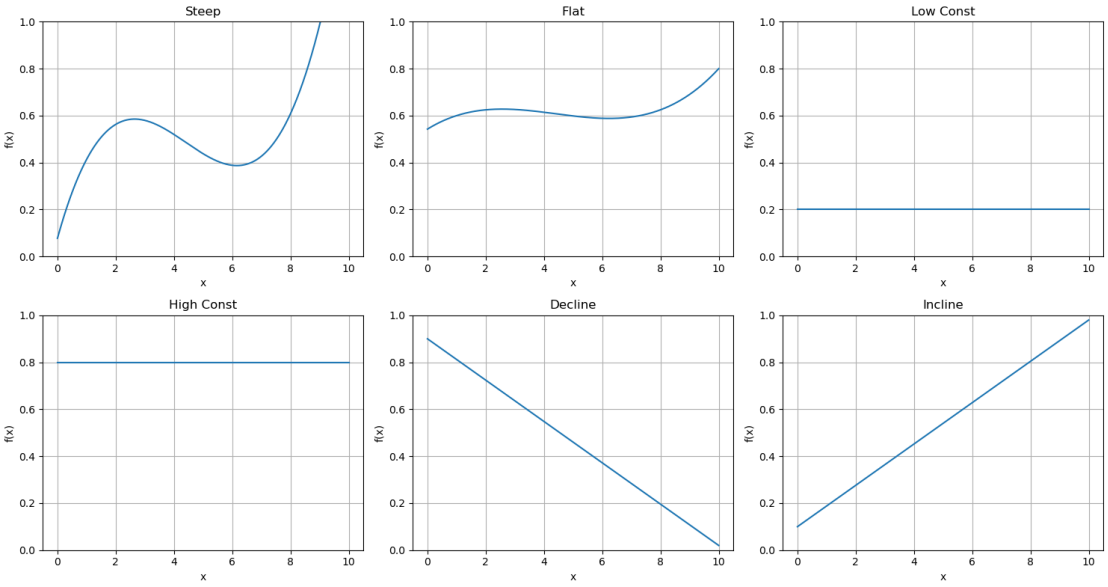


Figure 3.1: Functions used for $p$ and $q$.

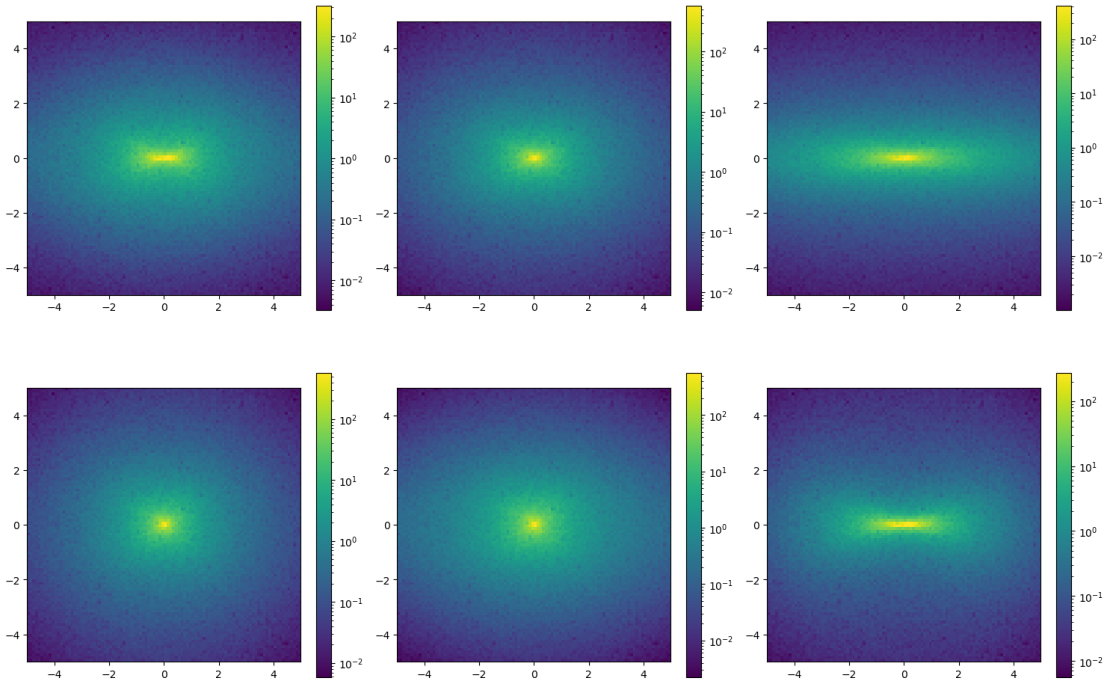

Figure 3.2: Images created using the functions shown in Figure 3.1.

# 4 Results

The results section is split into two parts. In Section 4.1 we will introduce some of the results obtained using the SVI method. Additionally we will present some statistics over all the models created using that method. In Section 4.2 we will introduce some results obtained by using the SBI method and also show some statistics over all model results. We will also explore the optimization of hyper-parameters for this method and see the impact of larger data sets .

## 4.1 SVI

Since many models where created using this method we use a subset to illustrate interesting findings. We split the model results into different categories for this purpose.

For our first set of result, we split our models into three groups depending on the number of steps used during the inference process: 300 steps, 600 steps and 900 steps. We compare three models in Figures 4.1, 4.2 and 4.3, beginning with 300 steps and ending with 900 steps. It is clearly visible that an increase in inference steps leads to a better image score. The same trend can be observed when looking at the function scores.

For the second set of results we split our models into two groups, depending if we are optimizing for the viewing angles $\varphi$, $\vartheta$ or $\psi$ or not. We compare two models beginning with Figure 4.4, where the viewing angles are know. While Figure 4.5 shows a model, where the viewing angles are unknown. The model performance clearly decreases for the more complex case of unknown viewing angles. The same trend can be observed for the function scores. The input recovery scores for the viewing angles also show that they are not predicted correctly.

For the last set of results we split the model into two groups. We use one model where $p$, $q$ and $\xi$ are all radially dependent in Figure 4.7 and another model where only $p$ is radially dependent in Figure 4.6. The simpler model with only a radially dependent $p$ performs better, which is reflected in lower image, function and input recovery scores.

One last result we want to illustrate is that there are cases where we reproduce the images very well, but we do not reproduce the true parameters correctly. This we can see in Figure 4.8. We observe a good image score, while not predicting the viewing angles correctly, which can be seen in the large input recovery scores.

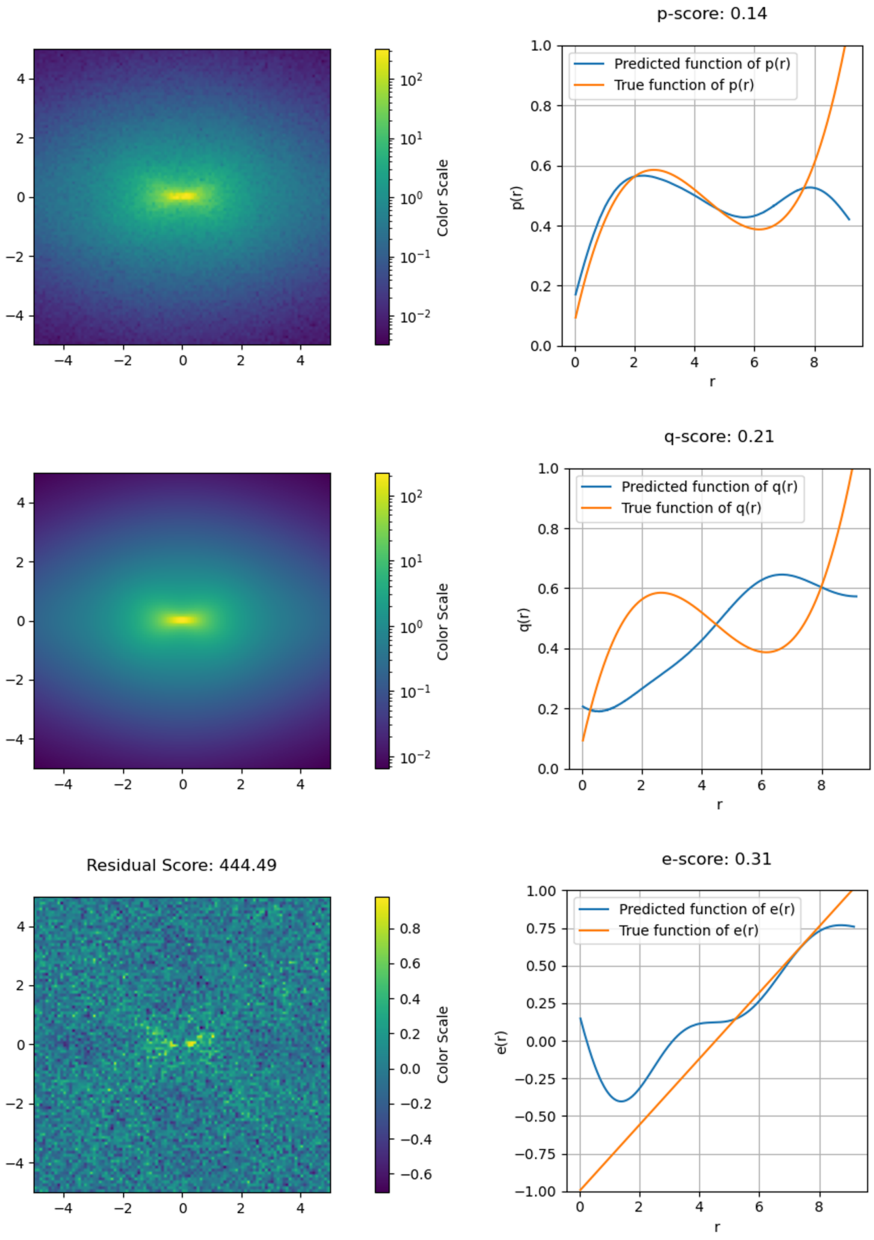


Figure 4.1: Model fitting $p(r)$, $q(r)$ and $\xi(r)$, using 300 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

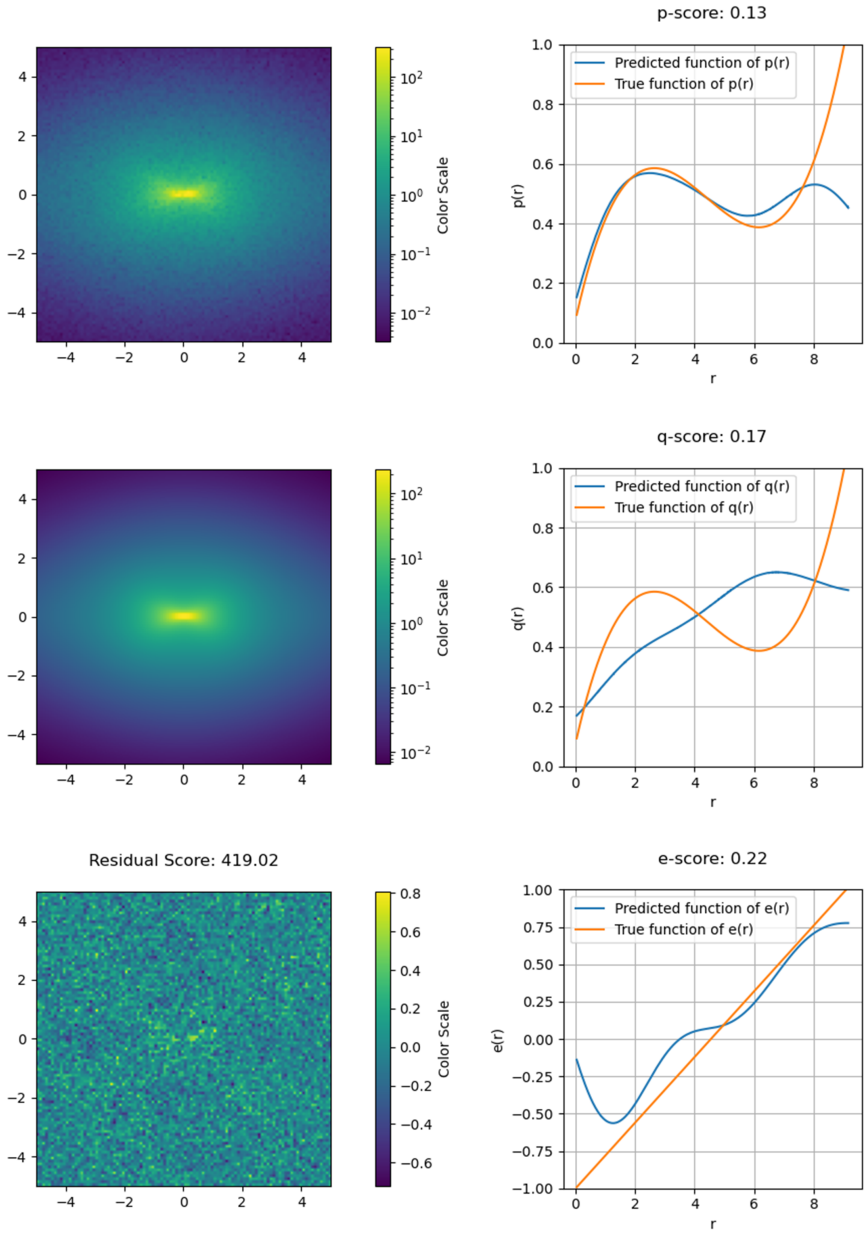


Figure 4.2: Model fitting $p(r)$, $q(r)$ and $\xi(r)$, using 600 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

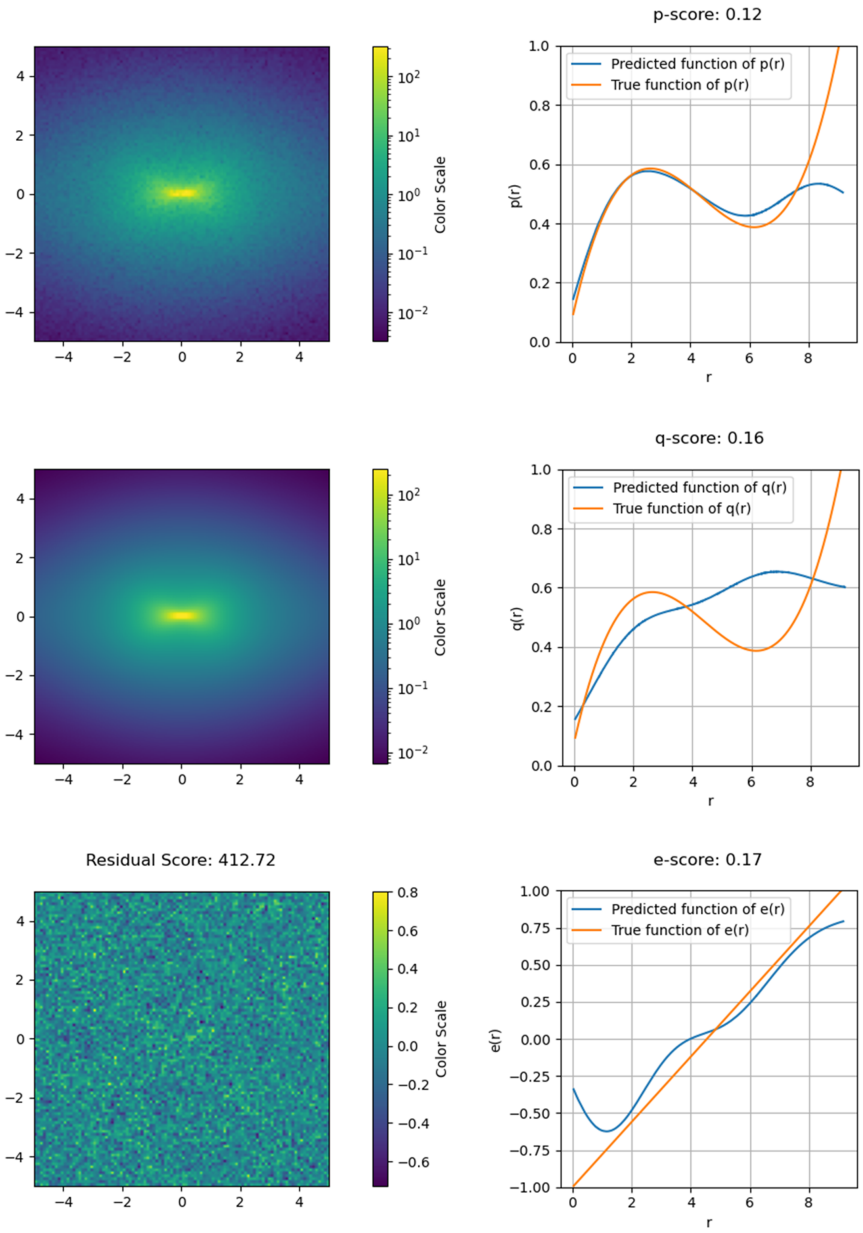


Figure 4.3: Model fitting $p(r)$, $q(r)$ and $\xi(r)$, using 900 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

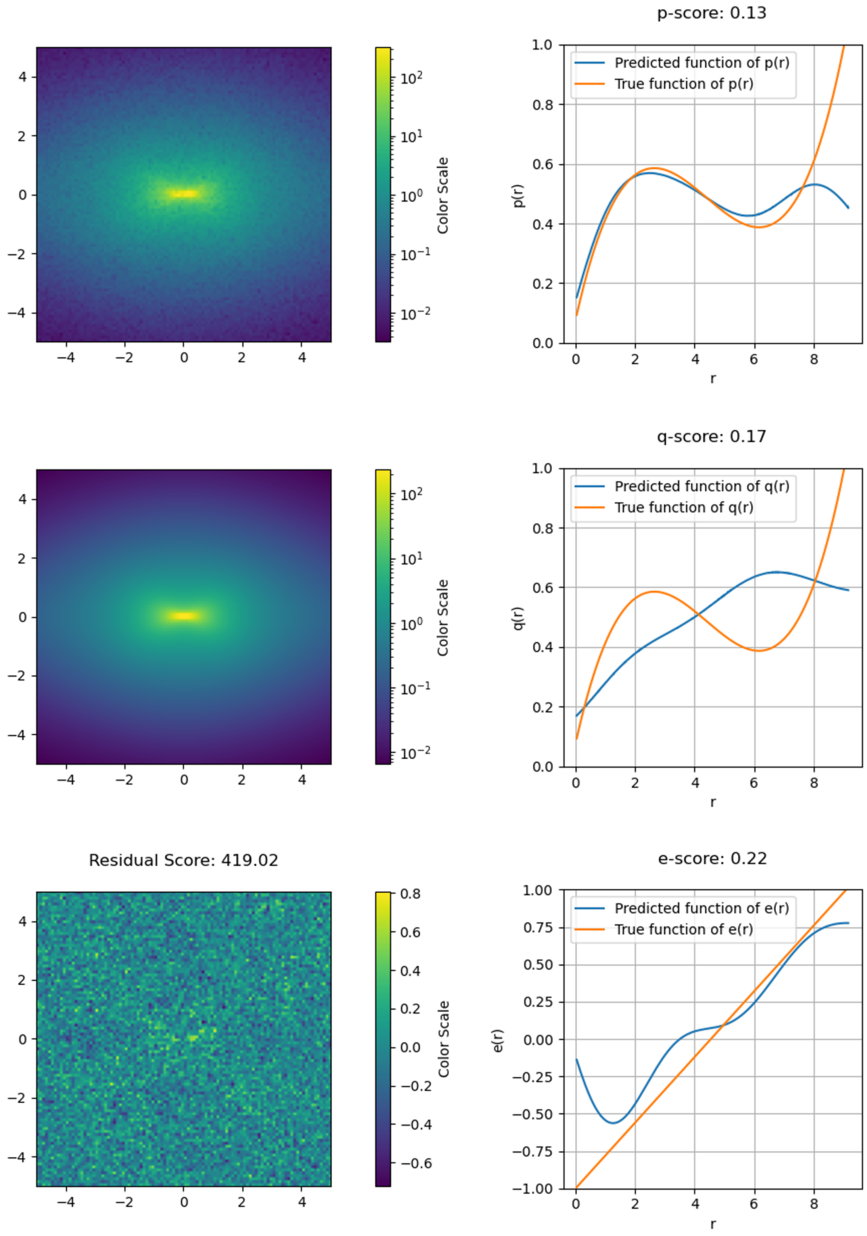


Figure 4.4: Model fitting $p(r)$, $q(r)$ and $\xi(r)$, using 600 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

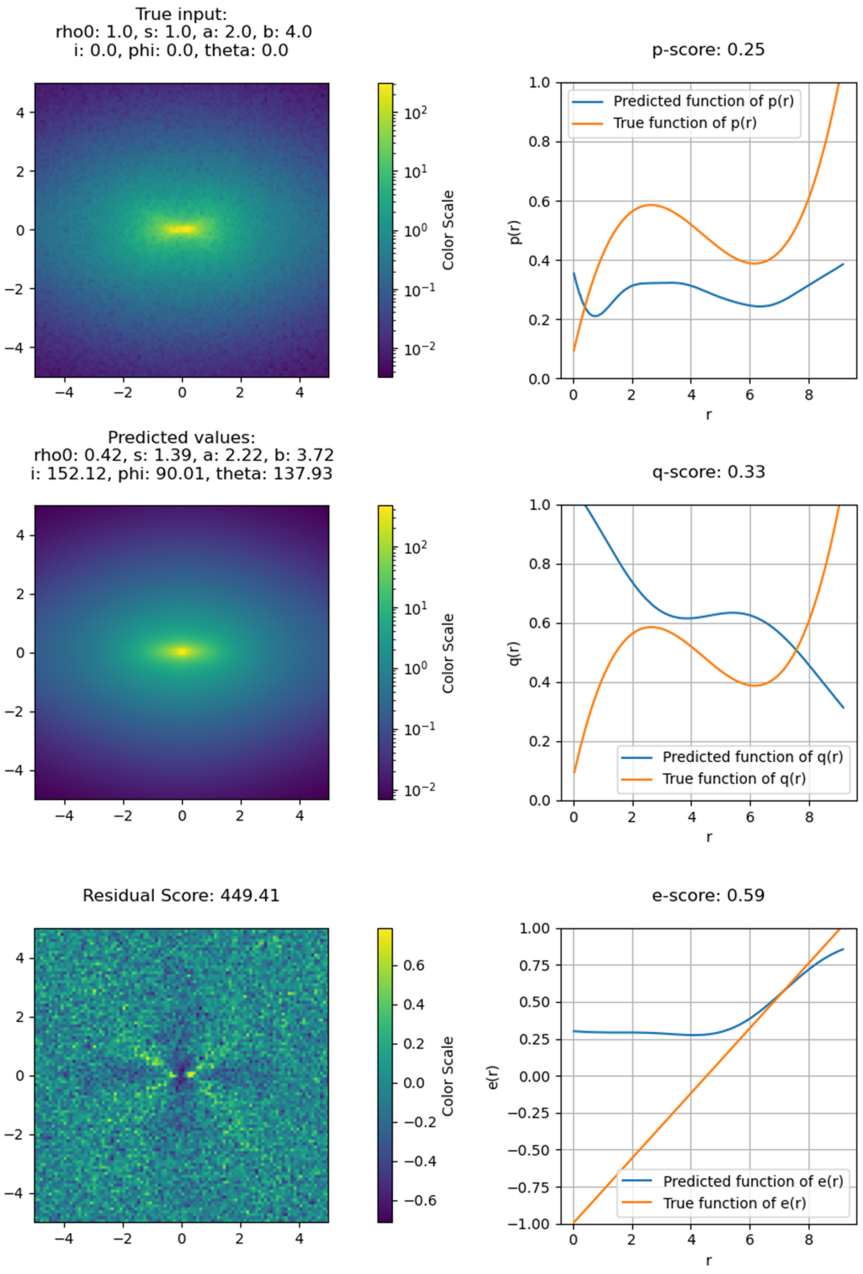


Figure 4.5: Model fitting all parameters including the viewing angles, using 600 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. Input and predicted values are also shown. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

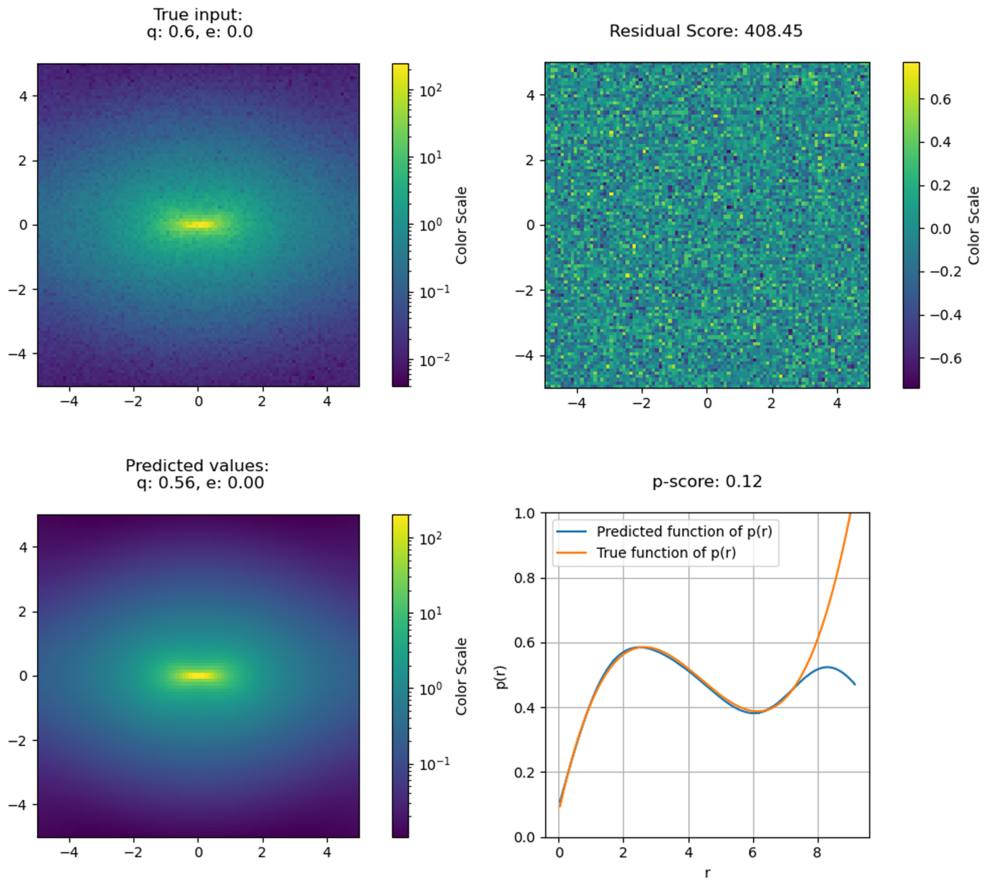


Figure 4.6: Model fitting $p(r)$, $q$ and $\xi$, using 600 inference steps. The left column shows the input image, the modeled image. Input and predicted values are also shown. The right column shows their residual including its residual score, which we also call image score and the radial profile for $p$. The orange lines represent the true function while the blue lines show the modeled functions. Additionally a score for the $p$ function is shown.

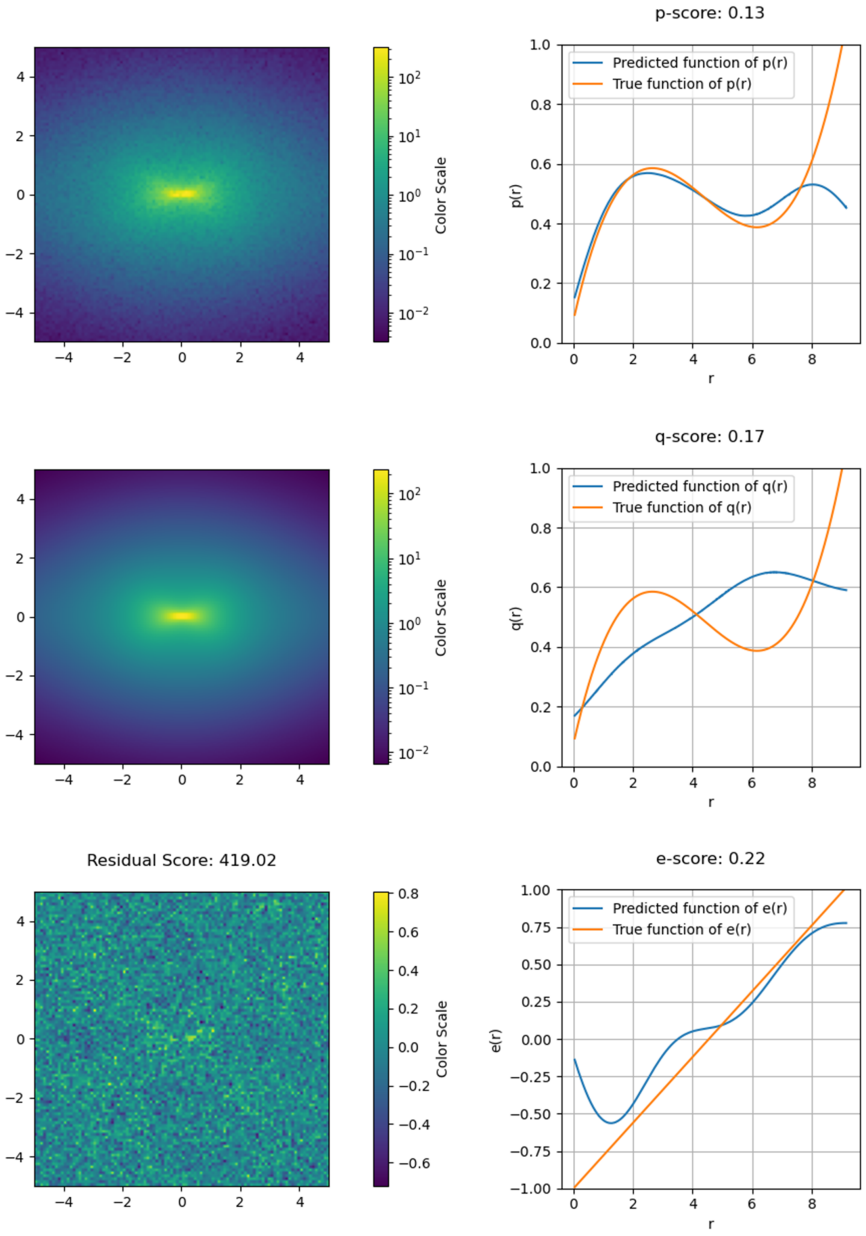


Figure 4.7: Model fitting $p(r)$, $q(r)$ and $\xi(r)$, using 600 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

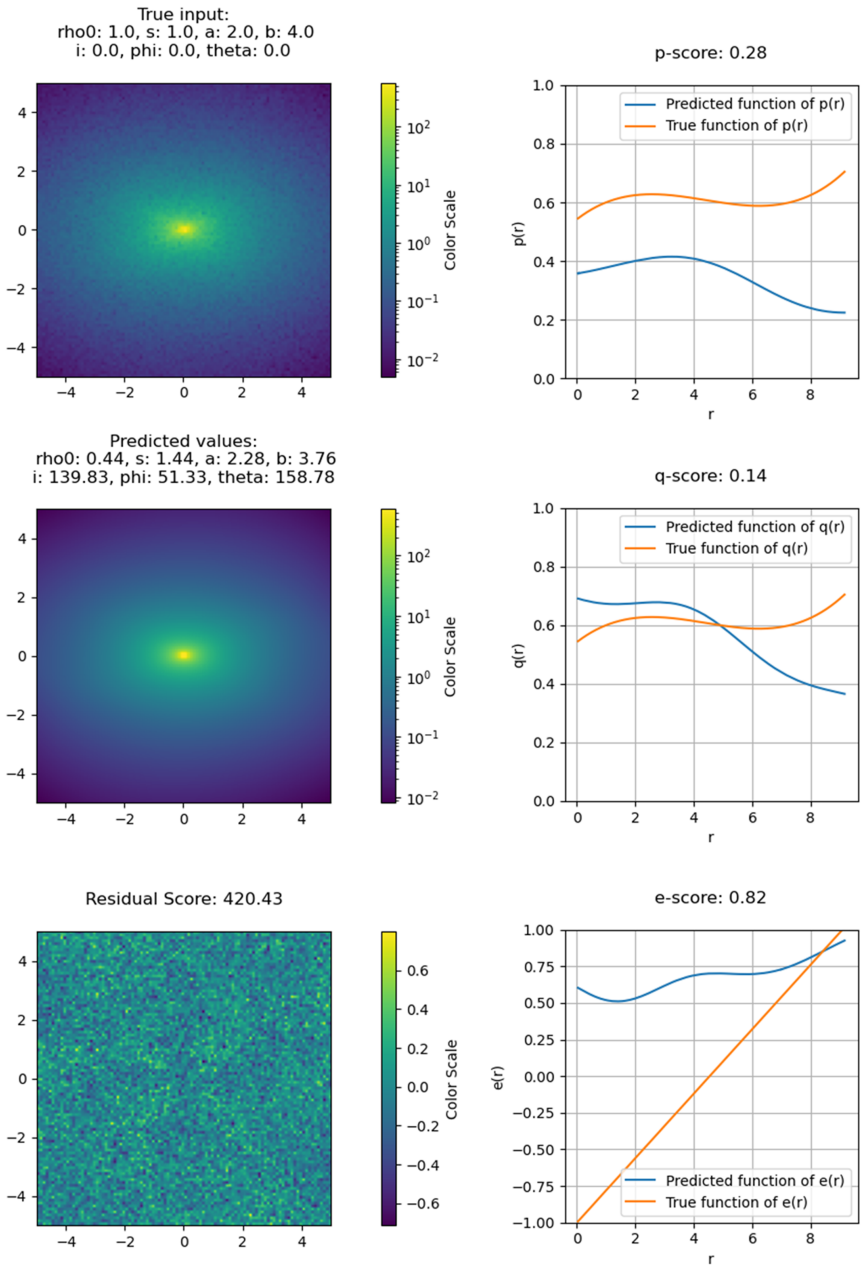


Figure 4.8: Model fitting all parameters, using 900 inference steps. The left column shows the input image, the modeled image and their residual including its residual score, which we also call image score. Input and predicted values are also shown. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

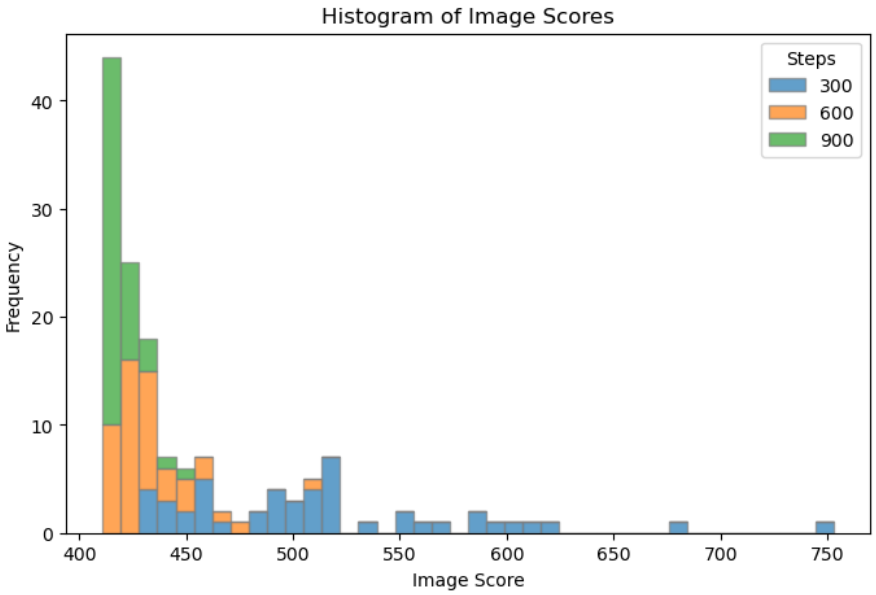


Figure 4.9: Histogram containing the image scores for all models. The data is separated into three groups, depending if 300, 600 or 900 steps where used during inference.

### 4.1.1 Statistics

In a similar manner we present statistics for these three categories.
First we present the case for different number of inference steps in Figure 4.9. We can clearly see that a more inference steps generally lead to a lower image score.
Next we present the case of different numbers of radially dependent parameters in Figure 4.10. We can observe lower image scores in models where $p$, $q$ and $\xi$ are not all treated as radially dependent.
Then for the case of known or unknown viewing angles in Figure 4.11, we do not see a clear distinction when looking at the image score. To make this case clearer we present a histogram over the total scores in Figure 4.12. Here we can clearly see a separation between known and unknown viewing angles, showing that models with known viewing angles have lower total scores.

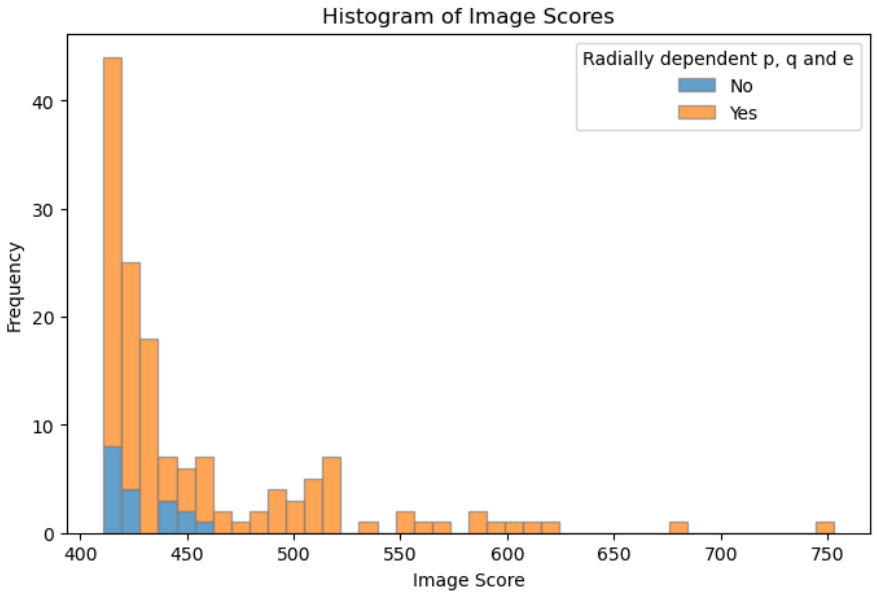


Figure 4.10: Histogram containing the image scores for all models. The data is separated into two groups, depending if $p$, $q$ and $\xi$ are all radially dependent or not.

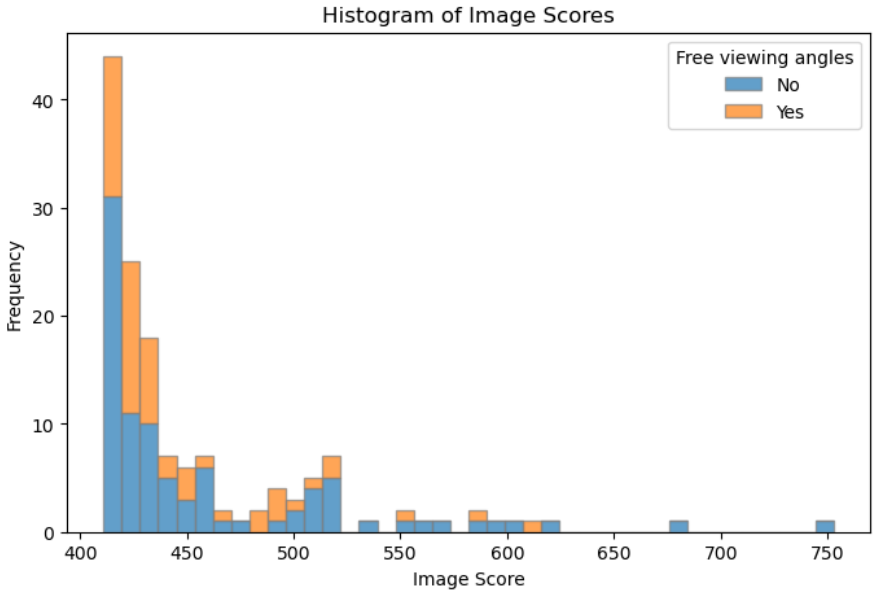


Figure 4.11: Histogram containing the image scores for all models. The data is separated into two groups, depending if the viewing angles were constrained or left as free parameters in the model.

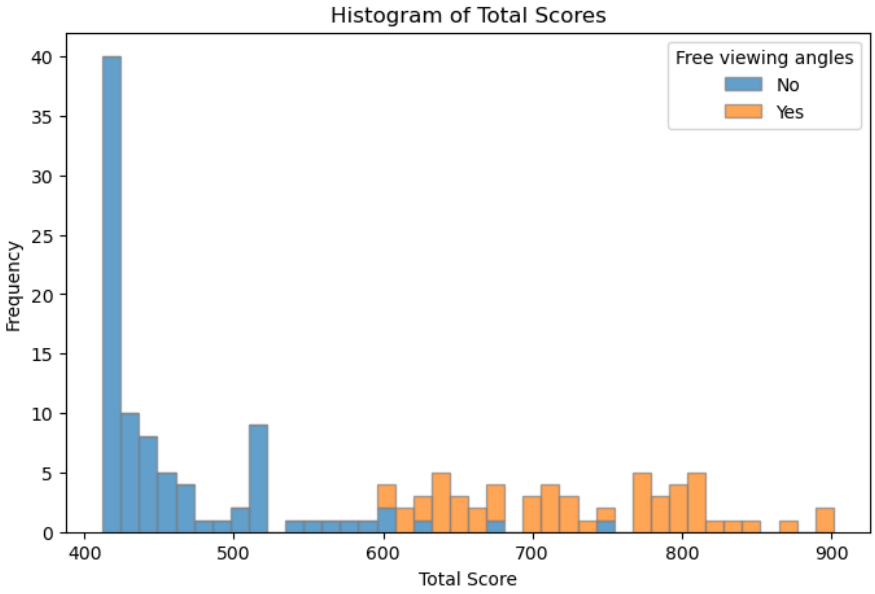


Figure 4.12: Histogram containing the total scores for all models. The data is separated into two groups, depending if the viewing angles were constrained or left as free parameters in the model.

## 4.2 SBI

We present the results for the different models described in Section 3.2. We will only show results for three of these models. The simplest model with constrained viewing angles and no radial dependence, the most complex model with free viewing angles and radial dependence and the model with the most training data. The results of the remaining models will not be presented, as they are similar and lead to the same conclusions as the ones shown.

For the first set of results we look at the models where we have constrained viewing angles and no radial dependence. We present the images for which the model performs the best and the worst. The average image score over all test images is 5.066. The best performing image can be seen in Figure 4.13 and the worst performing one in Figure 4.14. In both cases the modeled images look very similar even though the input images are very different. The good score of the best performing image is most likely due to randomly picking an image that fits the model.

For the second set of results we look at the models where we have free viewing angles and radial dependence. We present the images for which the model performs the best and the worst. The average image score over all test images is 12.321.119. The the best performing image can be seen in Figure 4.15 and the worst performing one in Figure 4.16. Again in both cases the modeled images look very similar even though the input images are very different. If we randomly pick an image that fits the model we get a good image and function score.

At last, for the hyper-parameter optimization the following values lead to the best results: `hidden_features` = 150 and `num_transforms` = 10. In Figure 4.17 and 4.18, we present the best and worst performing images on a model trained with these hyper-parameters and 40.000 training observations. The average image score over all test images is 31.810. Also in this case we observe similar output images. Again it seems to be the case that we only get a good performance if we randomly picked an image that fit the model.

### 4.2.1 Statistics

In Figure 4.19 we present one histogram for all models we have trained. Overall the scores are by far larger than the ones observed when using SVI. Only $\sim 1.35\%$ of results have an image score in the same range as seen in the SVI method.

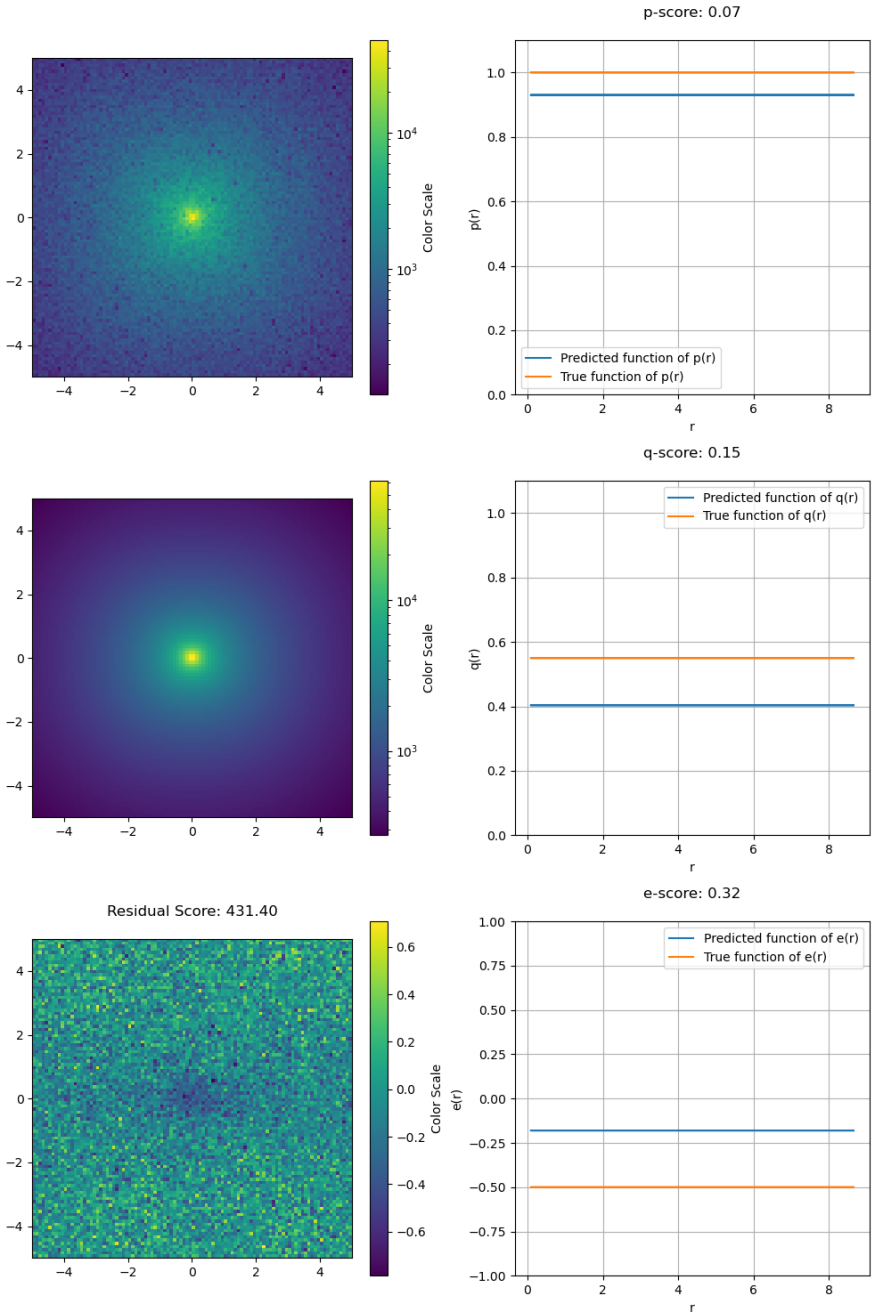


Figure 4.13: Best performing image for the model with no radial dependence and constrained viewing angles. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

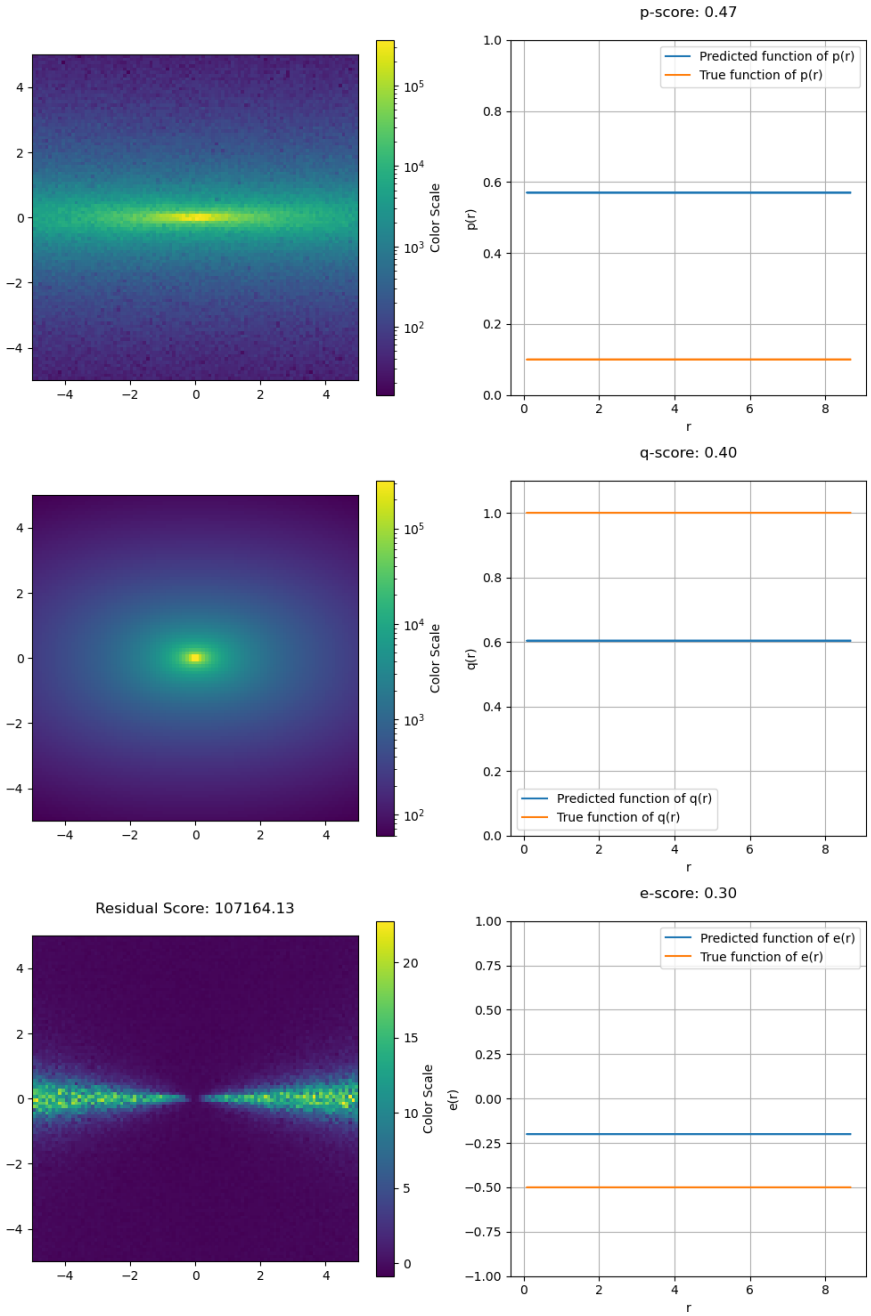


Figure 4.14: Worst performing image for the model with no radial dependence and constrained viewing angles. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

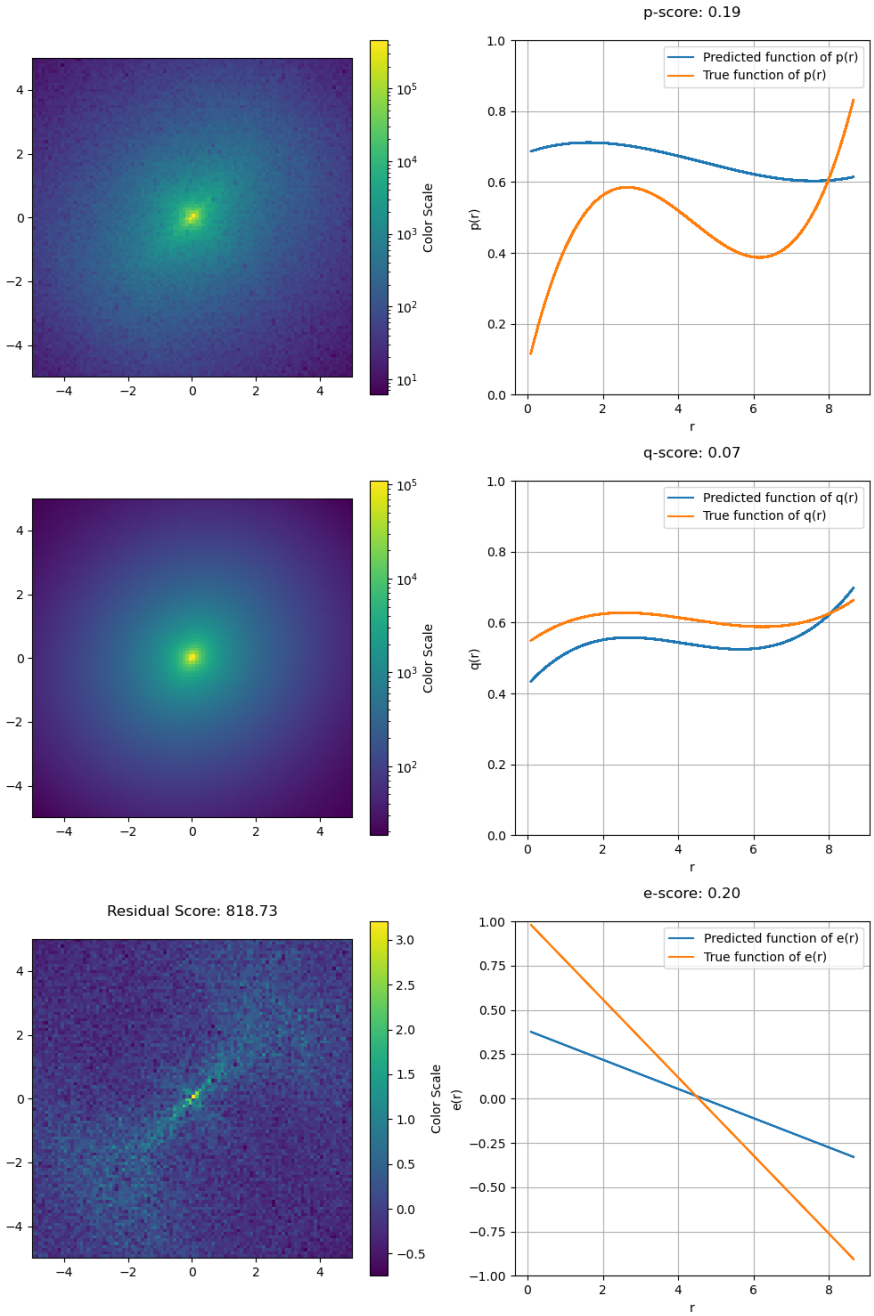


Figure 4.15: Best performing image for the model with radial dependence and free viewing angles. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

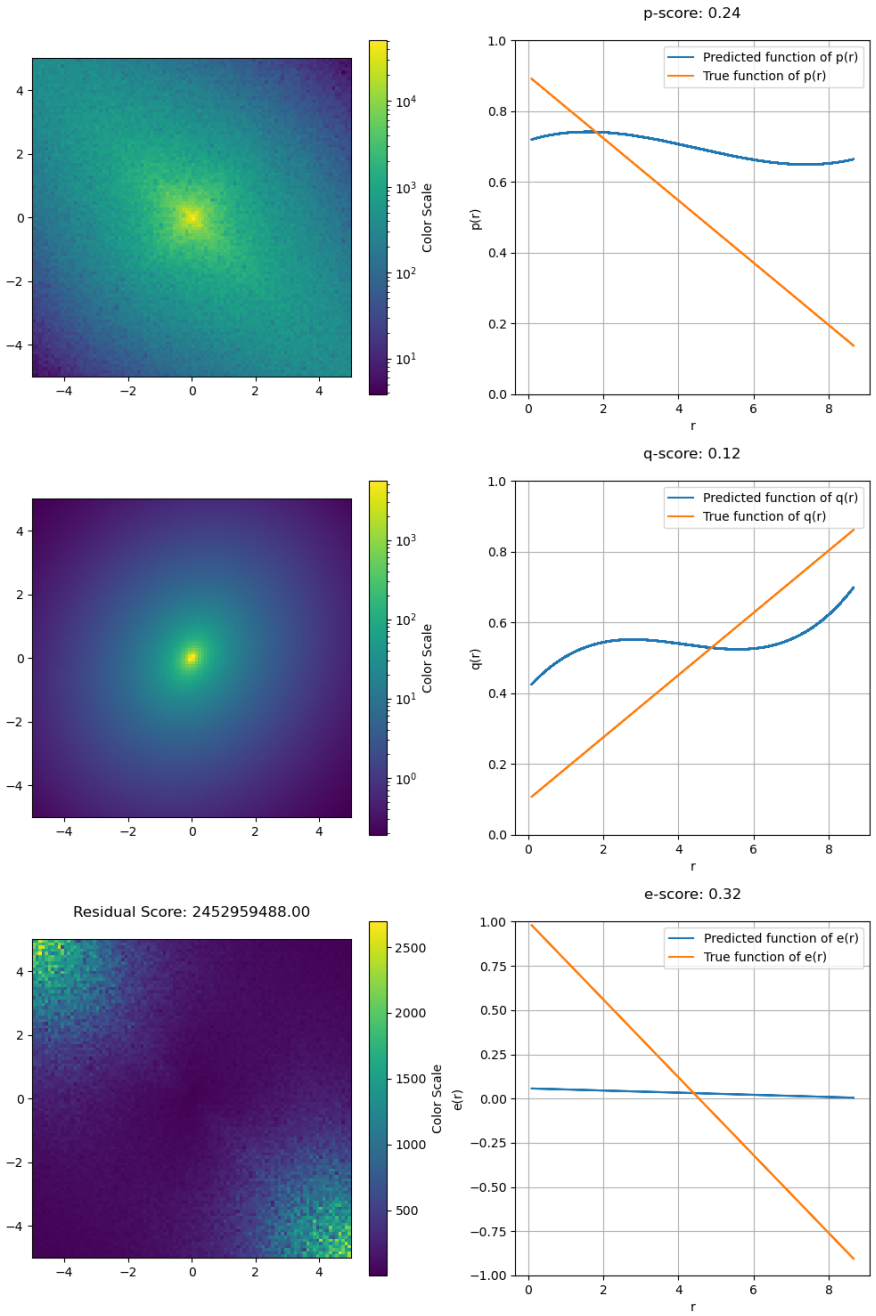


Figure 4.16: Worst performing image for the model with radial dependence and free viewing angles. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

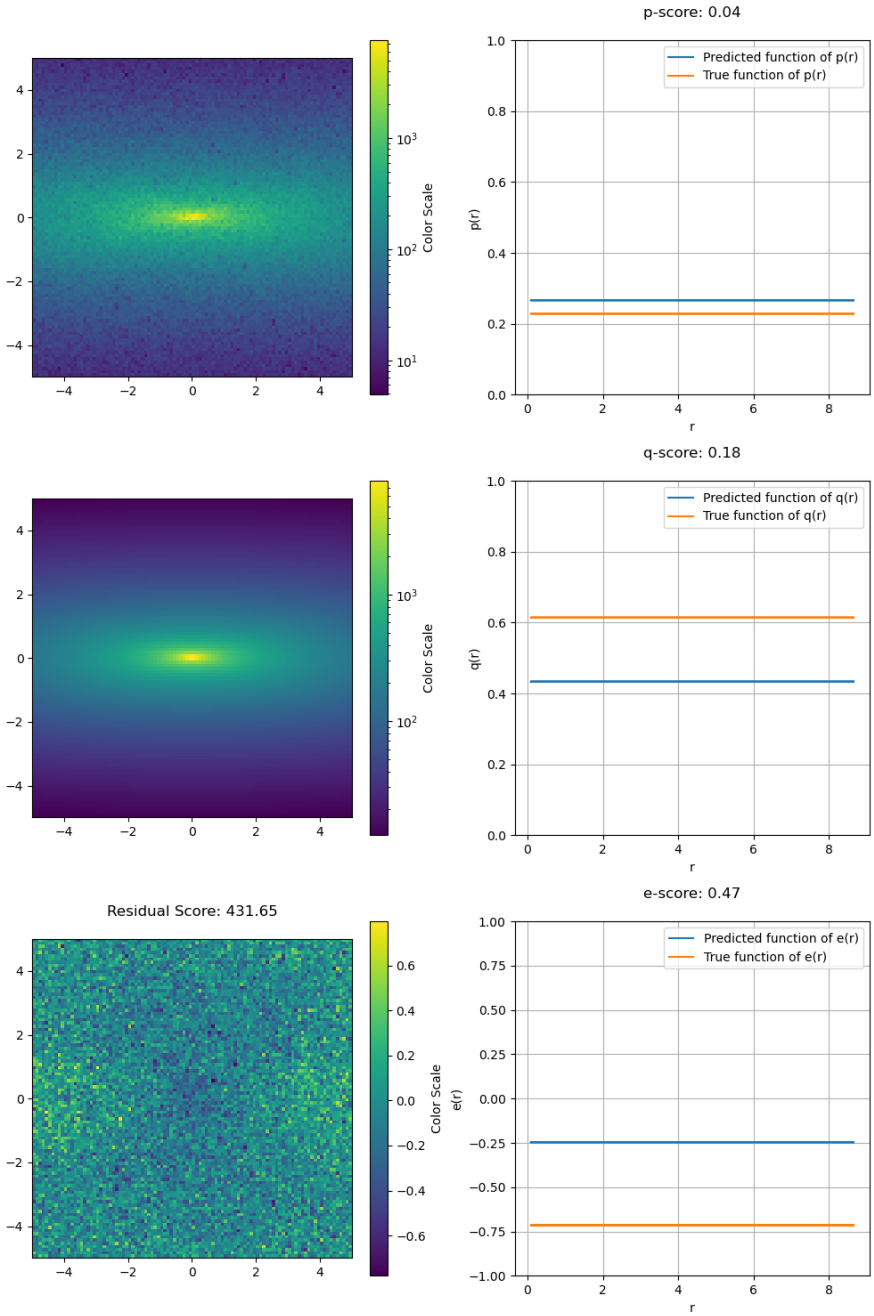


Figure 4.17: Best performing image for the model with optimized hyper-parameters and 40.000 training images. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.

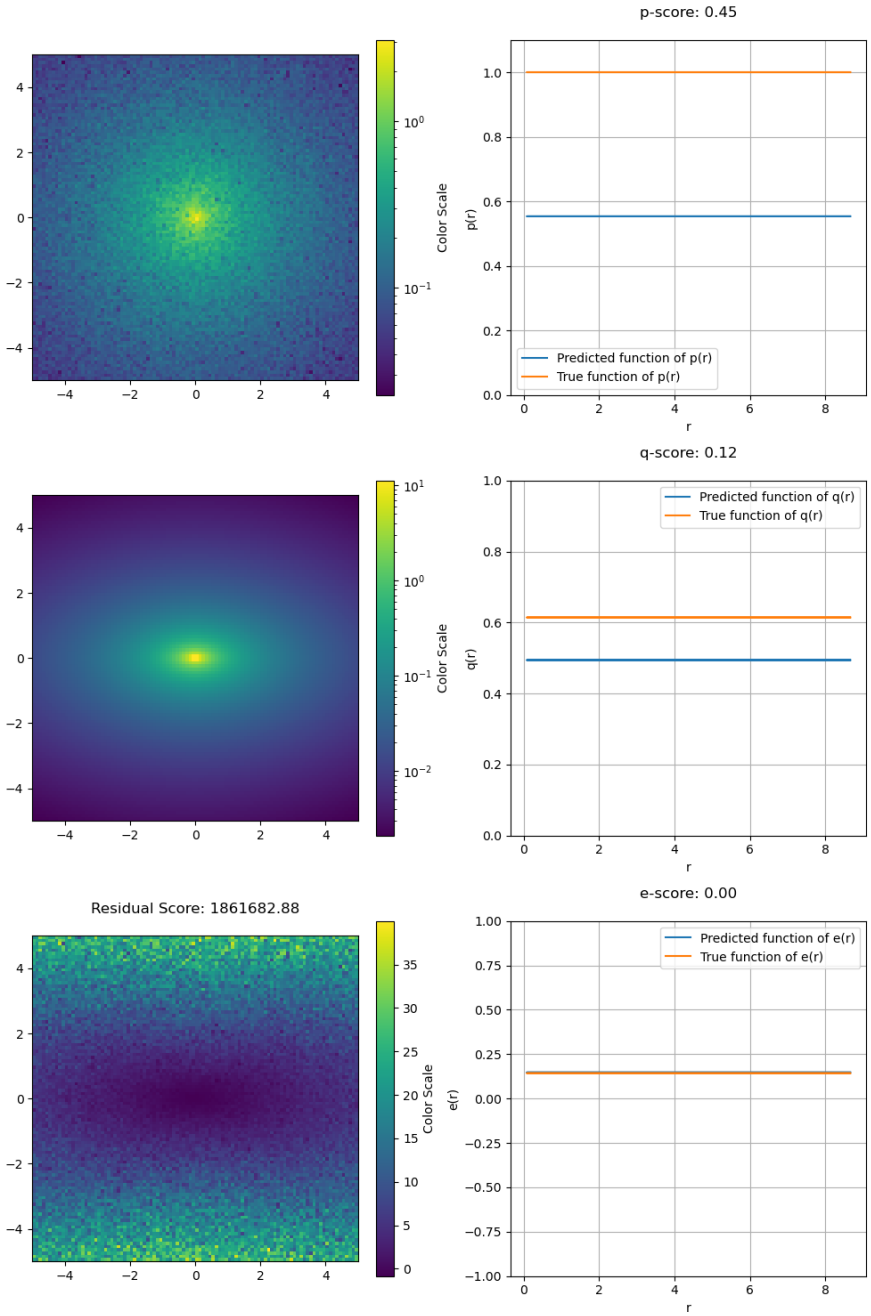


Figure 4.18: Worst performing image for the model with optimized hyper-parameters and 40.000 training images. The left column shows the input image, the modeled image and the residual including its residual score, which we also call image score. The right column shows the radial profiles for $p$, $q$ and $\xi$, which is called $e$ in the image. The orange lines represent the true function while the blue lines show the modeled functions. Additionally scores for these functions are shown.


Figure 4.19: Histogram containing the image scores for all 5 models. Only the bottom 85% of scores are used to get a better visualization of our data.

# 5 Discussion

In the following we discuss the results of Section 4 and their implications. We also give an outlook how the used methods could potentially be improved upon.

## 5.1 SVI

We begin by looking at our first set of results for the SVI method, where we want to see the impact of more steps during the inference process. When looking at Figures 4.1, 4.2 and 4.3, we can see that the image score improves with an increase in inference steps. A similar trend can be observed when looking at the function scores, they also improve with an increase in inference steps. The model performs well in all three cases, the main difference between the input image and the modeled image seems to be the Gaussian noise added to the input, which is not correctly replicated by our model. For the three modeled functions $p(r)$, $q(r)$ and $\xi(r)$, p is already modeled well with a low number of inference steps, we see only small improvements when increasing the number of steps. q on the other hand is not modeled that well initially but shows small improvements when increasing the number of steps. $\xi$ is also not modeled well initially but improves significantly with an increase in inference steps. The reason for the discrepancy in q is most likely due to the projection of the image. In our projected image most of the information to replicate p seems to be present, while some information on the behavior of q seems to be lost. This is a general trend that can be observed over all of our results. Looking at the statistics for this first set of results in Figure 4.9, we can confirm the same trend as above: Increasing the number of inference steps, generally leads to better image scores and model performance.

For the second set of result we investigate the impact of modeling the viewing angles on our model performance by looking at Figure 4.4 and 4.5. When looking at the image score we can see that it gets worse when we are modeling a more complex case with unknown viewing angles. A similar decline in quality can be seen when looking at the function scores. When looking at the true and predicted values of the viewing angles we can see that they are not predicted correctly, but even with this discrepancy we obtain a relatively good image score. A similar case is illustrated in Figure 4.8. This is the case because fitting these viewing angles is a highly degenerate problem i.e. different combinations of viewing angles can lead to very similar images when projected.

When looking at the statistics for this case in Figure 4.11 it is not immediately clear if modeling the viewing angles leads to worse model performance since the distribution of image scores look similar. To investigate this further we can look at Figure 4.12, which shows the total scores instead of the image scores. Here we can clearly see a separation, models with constrained viewing angles lead to a better performance, while modeling the viewing angles leads to a worse one.

For the third and last set of result we compare a simple model where only p is radially dependent with a more complex model where $p$, $q$ and $\xi$ are radially dependent. In the case of only having a radially dependent p and constant q and $\xi$ (Figure 4.6) our model performs really well. $\xi$ is approximated correctly while q is close too the true value. The predicted p function is also very close to the true one, which is represented by a low score. The residual between the modeled and input image seems to only consist of random noise, which can also be seen in the low image score. When looking at the more complex model (Figure 4.7) the function score for p stays generally the same. Since we are introducing more complexity in q and $\xi$ this leads to a worse performing model, as visible in the image score.

A similar trend can be observed when looking at the statistics for this case in Figure 4.10. Simpler models with less radial dependence lead to a better image score, while the more complex models on average have a higher image score.

One of the downsides is that the error estimation does not work correctly. We initially tried using a `AutoNormal` guide in the inference process, which enabled error estimates. These estimates where not realistic because they were to small. Even when increasing the error in our observations no significant improvement could be seen. One way to potentially solve this problem is to add a learnable noise term in the model. More sophisticate guide functions could also alleviate the problem.

Overall we can say that an increase in inference steps and less complex models perform better. The Stochastic Variational Inference method works reasonably well in the context of galaxy deprojection.

There is still potential to improve the model performance by trying out more complex combinations of different kernel functions, instead of the standard RBF kernel. On the other hand using more sophisticated loss functions instead of `Trace_ELBO` could also lead to improved model performance.

## 5.2 SBI

When analyzing the results of the SBI method, the first thing that sticks out are the very large average image scores for the different models, which range from $\sim 5.000$ to $\sim 12 \cdot 10^6$, where the average for SVI was $\sim 450$. This already indicates that this method is not working particularly well for our problem. We dive a bit deeper by looking at images which work the best and the worst for each of the data sets created. Looking at the modeled images, we see that they do look very similar even when the input images are vastly different, as can be see in Figures 4.13 - 4.18. We see that the images produced by one model all look very similar with minor changes between them, even if the image that we are trying to model looks different. For the cases that do work well, rather then the model correctly recreating the input we seem to have randomly chosen an image that matches the output of the model. This can also be seen in the summary statistics in Figure 4.19. Only around $\sim 1.35\%$ of images achieve a score $< 1000$ i.e. in the range of SVI. Most modeled images are very different to the input provided, which is reflected in the large image scores. The same applies to our more optimized model that was trained using a large data set of 40.000 images. The modeled images in Figure 4.17 and 4.18 look similar to each other. In Figure 4.17 the input image and the model image have a high similarity, which results in a low image score. When looking at Figure 4.18, we can see that the modeled image does not match the input, which is also shown in the large image score.

Overall we can say that using Simulation Based Inference on our particular problem does not lead to correct results and is not suited for our use case. Common causes for poor performance are a low variety of training data and overfitting. Low variety of data should not be an issue because we ensure to have a high diversity of training images. Overfitting also seems unlikely because we see the same problems for small and large training data sets. Further investigations into using Gaussian Processes for radially dependent parameters in the SBI model could lead to improvements of the model performance.

# 6 Conclusion

In this thesis, we investigated non-parametric approaches for galaxy deprojection using the JAX framework, focusing on two inference methods: Stochastic Variational Inference (SVI) and Simulation Based Inference (SBI). The goal of these deprojections was to transform 2D observations of galaxies into 3D models, which can tell us more about their intrinsic structures.

By creating many different models and testing them, we found that SVI outperformed SBI. SVI implemented via NumPyro, demonstrated good performance, even with increased model complexity, such as radially dependent profiles or unconstrained viewing angles.

In contrast, SBI, implemented via the `sbi` library, struggled to produce meaningful results. Even after optimizing hyper-parameters and significantly expanding the training data, the performance of the method did not improve.

While our SVI approach lacks realistic uncertainties, it shows great potential for future application. Enhancing its uncertainty modeling and extending it to real observational data could yield a viable alternative to existing parametric methods, such as Multi-Gaussian Expansion (MGE). Overall, our results show the feasibility of non-parametric galaxy deprojection, especially when supported by modern computational tools like JAX.

# Bibliography


Amari, S. 1985, Lecture Notes in Statistics

Beaumont, M. A., Zhang, W., & Balding, D. J. 2002, Genetics, 162, 2025

Binney, J. & Tremaine, S. 2008, Galactic Dynamics: Second Edition (Princeton University Press)

Bradbury, J., Frostig, R., Hawkins, P., et al. 2018, JAX: composable transformations of Python+NumPy programs

Cappellari, M. 2002, Monthly Notices of the Royal Astronomical Society, 333, 400

Cranmer, K., Brehmer, J., & Louppe, G. 2020, Proceedings of the National Academy of Sciences, 117, 30055

de Nicola, S., Saglia, R. P., Thomas, J., Dehnen, W., & Bender, R. 2020, Monthly Notices of the Royal Astronomical Society, 496, 3076–3100

Diggle, P. J. & Gratton, R. J. 1984, Journal of the Royal Statistical Society Series B: Statistical Methodology, 46, 193

Dongarra, J. J., Luszczek, P., & Petitet, A. 2003, Concurrency and Computation: practice and experience, 15, 803

Durkan, C., Murray, I., & Papamakarios, G. 2020, in International conference on machine learning, PMLR, 2771–2781

Frostig, R., Johnson, M., & Leary, C. 2018, in Compiling machine learning programs via high-level tracing

Greenberg, D., Nonnenmacher, M., & Macke, J. 2019, in International conference on machine learning, PMLR, 2404–2414

Görtler, J., Kehlbeck, R., & Deussen, O. 2019, Distill, https://distill.pub/2019/visual-exploration-gaussian-processes

Hermans, J., Begy, V., & Louppe, G. 2020, in Proceedings of Machine Learning Research, Vol. 119, Proceedings of the 37th International Conference on Machine Learning, ed. H. D. III & A. Singh (PMLR), 4239–4248

Hoffman, M. D., Blei, D. M., Wang, C., & Paisley, J. 2013, the Journal of machine Learning research, 14, 1303

Holley-Bockelmann, K. & Sigurdsson, S. 2006, A Full Loss Cone For Triaxial Galaxies

Kazantzidis, S., Abadi, M. G., & Navarro, J. F. 2010, The Astrophysical Journal, 720, L62–L66

Papamakarios, G., Sterratt, D., & Murray, I. 2019, in The 22nd international conference on artificial intelligence and statistics, PMLR, 837–848

Paszke, A. 2019, arXiv preprint arXiv:1912.01703

Phan, D., Pradhan, N., & Jankowiak, M. 2019, arXiv preprint arXiv:1912.11554

Rubin, D. B. 1984, The Annals of Statistics, 1151

Santucci, G., Brough, S., van de Sande, J., et al. 2022, The Astrophysical Journal, 930, 153

Tejero-Cantero, A., Boelts, J., Deistler, M., et al. 2020, Journal of Open Source Software, 5, 2505

Zhang, C., Bütepage, J., Kjellström, H., & Mandt, S. 2018, IEEE transactions on pattern analysis and machine intelligence, 41, 2008

Álvaro Tejero-Cantero, Macke, J. H., Lückmann, J.-M., Deistler, M., & Bölts, J. F. 2020, sbi: simulation-based inference toolkit, https://sbi-dev.github.io/sbi/v0.23.1 [Accessed, 24.06.2025]