Université Clermont-Auvergne

Study and optimization of PSF processing for the ZTF experiment

VOISIN Sybille

Supervised by: GADRAT Sébastien and RIGAULT Mickaël

CC-IN2P3 Lyon
M2 Univers et Particules
March 01, 2024 - July 31, 2024

Contents

Acknowledgments

I would like to thank my internship supervisors Sébastien Gadrat and Mickaël Rigault for guiding me throughout my internship, Sébastien for the computing and GPU part and Mickaël for the cosmology and ZTF part. I would like to thank the Centre de Calcul (CC-IN2P3) and IP2I for welcoming me and showing me their confidence.

Then I would like to acknowledge the CC user engagement team who integrated me from the start and with whom I was able to forge links. In particular, Théo Turlin, who helped me develop the Gaussian fitting code and has become a good friend. I would also like to mention Bertrand Rigaud from the computing team for his training and help with GPUs.

I would also like to thank all those who took the time to review this report and provide me with constructive comments to help improve it: Alexandre Boucaud from CNRS, Gino Marchetti from CC-IN2P3 and Fabio Hernandez from CC-IN2P3.

Thank you to my teachers for giving me the knowledge I needed to succeed in this internship.

Finally, I would like to thank my family and friends, who have always supported and encouraged me in all my projects.

Introduction

Based on scientific knowledge, we think that the universe is composed of just 5% observable matter (stars, planets, gas, etc.), 26% dark matter, and 69% dark energy [1]. However, the exact composition of the cosmos is yet unknown. Dark matter and dark energy have been conceptualized to match with current models, but their nature remains unknown. Because supernovae are standard candles (§1), the study of supernovae, and in particular Type Ia (SNeIa), is especially valuable for these tasks. Indeed, SNeIa are essential instruments for cosmology because they allow us to measure cosmological distances, verify theoretical models, study element production, and get to know the dynamic history of the universe.

Located atop Palomar Mountain in California (USA), the ZTF (Zwicky Transient Facility) telescope [2] is an astronomy observatory(§2.1). It was inaugurated in March 2018 and is supported jointly by an international partnership of universities and institutes of Europe and Asia and the US National Science Foundation. Its purpose is to scan the night sky for transient and variable astronomical phenomena such as SNeIa. To study supernovae, ZTF team uses an image processing pipeline that consists in cleaning the data (bias, flat, non-linearity effects, etc. see §2.2) and then analyzing the images, in particular with the PSF (Point Spread Function §2.3).

The pipeline is constrained by the execution time of the PSF (currently written in C++), which takes longer than the other steps. The aim of my internship is to optimize the PSF code in order to reduce the execution time. To achieve this, the first step is to rewrite and optimize the code on CPU (it will be a reference for benchmarks), then to run it on GPU (§3.1) using the JAX framework (from Google, §3.2). To ease code implementation and user adaptation, a framework close to the existing one is required.

1 Notions on cosmology

SNeIa occur in binary systems that comprise a white dwarf and another star. The white dwarf takes on material from its partner star. It is primarily made of carbon and oxygen. The electrons' degenerative pressure can no longer sustain the growing gravitational force when the white dwarf approaches the Chandrasekhar limit, which is considered to be 1.4 times the mass of the Sun. When this threshold is reached, the white dwarf's core temperature and pressure increase to the point where carbon and oxygen can undergo nuclear fusion. The white dwarf explodes into a supernova as a result of this quick fusion, which releases a massive quantity of energy (around 10^{44} joules [3]).

A standard candle in astronomy is a celestial object whose intrinsic (or absolute) luminosity is well known and relatively constant. This means that the amount of light emitted by this object is predictable. Standard candles are used to measure distances in the universe, because by comparing their apparent luminosity (the luminosity we see from Earth) with their intrinsic luminosity, we can determine how far away they are from us. The intrinsic luminosity of SNeIa is remarkably clearly defined, with an absolute magnitude of 19.3 and they enable us to measure distances to distant galaxies, and thus map the universe on a large scale. Their study has contributed to our knowledge of dark energy and the accelerating expansion of the universe [4]. To measure distance, we use the notion of redshift z. The light emitted by distant objects is stretched as it travels through expanding space, resulting in a redshift. This redshift is directly related to the distance of the objects observed relative to the observer: the further away a galaxy is, the greater its redshift. Then, thanks to the Hubble's law see Eq. 1 we can deduce the cosmological distances.

Hubble's law [5] is expressed by:

$$v = H_0 \cdot d \tag{1}$$

where v is the galaxy's recession velocity (link to the redshift), H_0 is Hubble's constant (the rate of expansion of the universe), and d is the galaxy's distance from Earth. This law is shown in Fig. 1.

The data from Type Ia supernovae are instrumental in refining cosmological models, particularly the ΛCDM (Lambda Cold Dark Matter) model [6]. This model includes the cosmological constant (Λ), which represents dark energy, and cold dark matter (CDM). Type Ia supernovae serve as precise distance markers that help calibrate other methods of distance measurement and test the predictions of cosmological theories. By comparing the distances measured using supernovae with the distances inferred from the cosmic microwave background (CMB) [7] and the large-scale structure of the universe, cosmologists can fine-tune the parameters of the ΛCDM model. These parameters include the density of dark matter, the density of dark energy, and the Hubble constant.

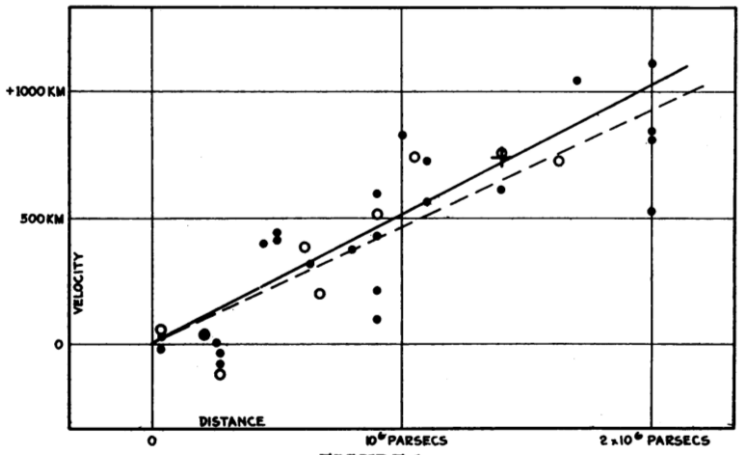


Figure 1: Velocity-Distance Relation among Extra-Galactic Nebulae [5].

In addition, nucleosynthesis [8] occurs in SNeIa, where a large number of heavy elements are created such as nickel and iron. Type Ia supernovae are particularly important for the production of iron-group elements. The explosion results in the synthesis of these elements via the complete fusion of carbon and oxygen at very high temperatures (around 5 billion degrees Kelvin). The elements produced in SNeIa are ejected into interstellar space, enriching the interstellar medium with heavy elements. This process contributes to the chemical evolution of galaxies, modifying the composition of future generations of stars and planets. Stars that form in regions enriched by SNeIa contain higher proportions of heavy elements, influencing their evolution, structure and spectral characteristics. Thus we can learn more about the chemical enrichment of the universe and the evolution of galaxies by examining them and the elements they produce.

In short, SNeIa are indispensable resources for cosmology, enabling us to measure cosmological distances, test theoretical models, study the chemical composition and understand the chronological evolution of the universe.

2 The Zwicky Transient Facility

A number of observatories on Earth have set out in search of supernovae, including ZTF, located atop Palomar Mountain in California (USA).

2.1 The detector

The camera of ZTF is made up of sixteen 6144×6160 pixel CCDs (Charge-Coupled Devices), arranged into four quadrants (Fig. 2). Three filter bands (g, r, and i) are employed, and each picture covers 47 square degrees. These filters correspond to different wavelengths of visible light: the g filter (for green) has a wavelength of around 475 nm, the r filter (for red) has a wavelength of around 625 nm and the i filter (for infrared) has a wavelength of around 775 nm. ZTF aims to scan the Milky Way plane twice a night and map the entire northern sky in three nights.

A CCD is an electronic sensor that transforms electrically neutral photons into electrically charged electrons. Photons from an astronomical object are absorbed by a semiconductor material (in this case, silicon) as they hit the CCD surface. Electrons are liberated from the semiconductor's crystal lattice by this energy absorption, forming electron-hole pairs. As a result, electrons are trapped in potential wells on the surface of the CCD, where they create an electrical charge that is proportional to the amount of photons that are received, this process is named ADU (Analog-to-Digital Unit). To be read and transformed into a digital signal, the charges accumulated in the pixels are transmitted to the ADU. The unprocessed digital images, known as raw images, are then calibrated to create scientific images that can be used for physical analysis.

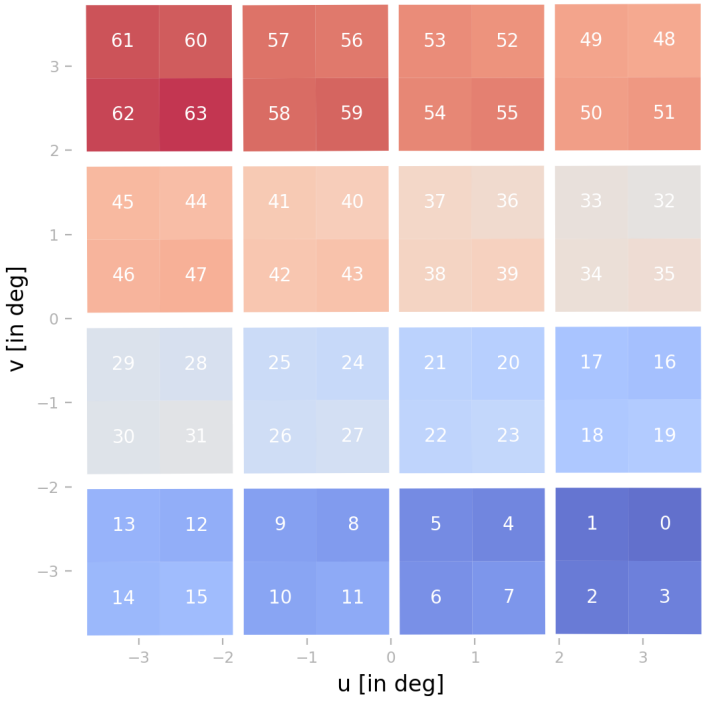


Figure 2: Diagram showing the distribution of the 16 CCDs and 64 quadrants of the ZTF camera (taken from ztfimg documentation [9]). The color scale represents the reading order of the quadrant numbers, from smallest in blue to largest in red.

In order to determine whether the number of stars obtained after my selection is coherent with official data, I estimate the average number of stars in a quadrant, I used a file based on the GAIA DR3 catalog [10], with ZTF's own selection of sufficiently isolated main-sequence stars. These serve as a reference for photometric calibration. This selection is made for a complete sky survey. There is no data for fields before 250 as this corresponds to the southern hemisphere, which is not observed by ZTF. We note that the average value per quadrant is around 280 stars as seen in Fig. 3 but the number of stars can reach 500 (for more figures see Appendix A)).

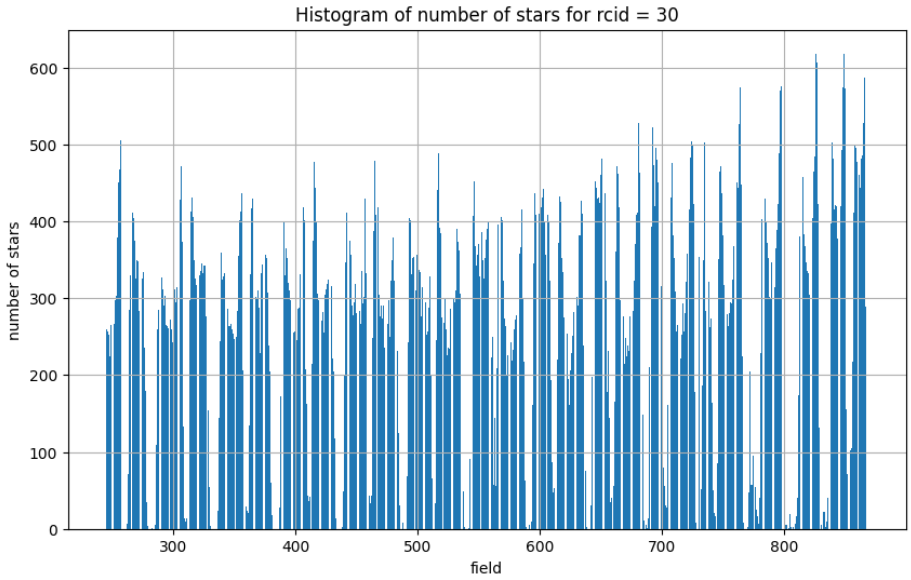


Figure 3: Distribution of the number of stars for the quadrant 30, extracted from the GAIA DR3 catalog [10].

2.2 Image calibration pipeline

Some processing [11] is required before using scientific images for physical studies because the digital images obtained are raw images.

- The bias adjustment is the first treatment, and it occurs every morning when the telescope is turned on. The temperature can lead electrons to propagate through pixels and produce background noise.
- We want to check how many electrons are beyond the CCD. Since the pixel grid is 6144 x 6160 pixels, we add 20 to 30 pixels to our image to detect any electrons that need to be returned to their place on the CCD. This principle is called overscan.
- There are non-linearity effects due to brighter-fatter. It is a distortion of light source images: as brightness increases, the electrons generated by the photons spread non-uniformly across the sensor pixels. This distorts the image of the light source.
- There are non-linearity effects due to electron saturation effect. When there are too many electrons, a saturation threshold is reached, making conversion to ADU difficult.

Every day, twenty sequences are carried out by the ZTF team, and an average is calculated. This is called a master-bias. Several tests were realized, and after twenty trials, the results no longer changed, and the signal-to-noise ratio was sufficient.

Each pixel reacts differently to photon stimulation. The aim is to know the response of each pixel so as to be able to homogenize everything. To do this, we place a flat screen in front of the telescope, sending the same light source to each pixel, and observe their reaction. In this way, we will know which factor (in ADU) to apply to each pixel.

Next, we apply the bias to the flat, repeat the operation twenty times and calculate the average to create a master-flat.

Once these steps have been completed, we have scientific images.

2.3 Point Spread Function

The PSF [11] [12] uses a mathematical function to model the spread of light intensity from a point source over an image, ideally described by a Dirac peak on one pixel. Due to defects in the optical instrument, atmospheric effects and mirror curvature, the PSF is actually expanded over several pixels. Each image's PSF must be measured and modeled in order to produce precise photometric measurements. In this work, two distributions have been used: the Gaussian (§3.4.1) and the Moffat (§3.4.2). Since the Gaussian function [13] is symmetrical, it is a good approximation of the PSF under ideal observation conditions, where dispersion is mainly due to atmospheric and optical effects that produce a symmetrical shape around the source. The Moffat model [14] is particularly effective for modeling the wings of the PSF, which are the outside parts of the light distribution. These wings can be larger than predicted by a Gaussian distribution, which could take into account to the atmospheric turbulence effects.

In addition to bias and flat correction (code written in Python), there is PSF adjustment (code written in C++). The execution time of the PSF code (around 20 seconds for 100 stars), despite being highly optimized, is much longer than the execution time of the bias and flat code (a few milliseconds). This is why it is so important to reduce the runtime of the PSF code.

3 Optimization of the PSF computation

The current code of the PSF is written in C++ in non-parallel way and therefore runs on CPUs. This code consists mainly of image processing, and more specifically pixel processing. So, to optimize the PSF code, it would be interesting to parallelized the processing and therefore use GPU hardware.
Three main frameworks can be used to compare GPU gains with the baseline on CPU:

- TensorFlow (developed by Google Brain in 2015 [15])
- PyTorch (developed by Facebook AI Research in 2017 [16])
- JAX (developed by Google Research in 2018 [17])

We chose the last option. Since JAX is as simple to write as NumPy [18] and SciPy [19], two packages that many physicists of the ZTF team use, integrating it into existing code would be easier.

3.1 Processing unit architectures

A computer's central processing unit, or CPU, is its orchestra conductor. It is an electronic chip that carries out computation, data processing, and control tasks by executing computer program instructions. The CPU includes several components, including the control unit, which coordinates operations, and the

arithmetic and logic unit, which performs calculations.

A GPU, or graphics processing unit, is a type of processor made specifically to manage and speed up graphics and visualization-related tasks. Unlike the CPU, which focuses on a wide range of tasks, the GPU is optimized to perform many simple operations in parallel. It is therefore ideal for generating high-performance images and performing high performance computations like physics simulations.
I was able to work with NVIDIA V100 GPUs [20] containing 5120 cores with a CUDA version 12.2, that were available at the Centre de Calcul (CC-IN2P3).

3.2 JAX framework

As the objective is to reduce the execution time of the PSF code, we need to keep certain constraints in mind, in particular the ability to run on CPU if necessary (problem on GPU or GPU unavailable). That is why JAX is a good choice.

The last updated version of JAX was 0.4.28 [21], released in May 2024, and since January 2024 there have been 5 releases. So there is a lot of movement from one version to the next, which means that the code often has to be updated, and the documentation does not provide any information on older versions. In addition, JAX uses both Autograd [22],such as TensorFlow, for automatic differentiation, which is a technique used to calculate the derivatives of a function in an algorithmic way, and XLA (Accelerated Linear Algebra) [23], such as PyTorch, for optimizing and accelerating computations, making it a powerful tool for scientific research and machine learning applications. JAX seems to be a great alternative.

3.3 Data processing

I have built a Python environment on top of the official ZTF environment [9] [24] [25], in which I have deployed all the required python modules to be able to work with JAX. My environment, which overloads the ZTF environment, has made it possible to use GPUs, which were not previously compatible. Consequently, I was able to work with functions used in the official analysis framework and even submit a few merge requests to bring the documentation up-to-date. I have used one the most recent versions of JAX (0.4.26 released in April 2024) in my environment as close as possible to the official environment as far as the limitations of the NVIDIA V100 GPU (CUDA version 12.2) allowed it.

The first step is to extract images, so I relied on the *ztfimg* documentation. [9]. During the image selection stage, I had the problem of corrupted files. In fact, for one image, there are actually two files per quadrant (the raw image and the science image), i.e. 128 files, and it happens that one file may be downloaded incorrectly. So I have written a Jupyter notebook which, thanks to the tools provided by ZTF, enables you to find the file in question and download it again. Once this first task was completed, I was able to focus on optimizing the PSF.

I started by taking an image, from which I chose a CCD (ID 1) and then a quadrant (ID 0) (Fig. 4). I then imported the corresponding data referenced in the Gaia DR2 catalog [26] (I not used the GAIA DR3

catalog because it is not in the official ZTF archives). Since the aim is to analyze each star independently, a stamp image (17x17 pixels) will be extracted for each star. Knowing that a PSF spreads out on average between 1.8 and 4 arcsec [27], and that we want to be sure to have the wings of the PSF, we multiply by 5, finally isolating the stars over 20 arcsec. This ensures that we don't pick up light from an nearby star, but also that we have enough pixels to have the entire PSF. I have also removed the stars on the edge of the image, i.e. fewer than 15 pixels from the edge, as they would not be centered in an image of this size (17x17). After these treatments, in the quadrant (ID 0), I was left with 2075 stars compared with 15174 at the start.

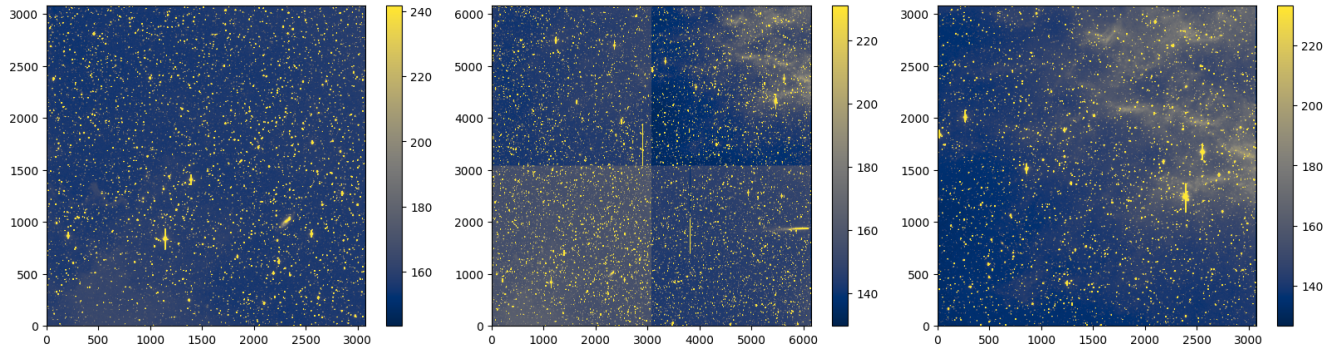


Figure 4: Left: Starting image **ztf_20200924431759_000655_zr_c13_o_q3_sciimg.fits** corresponds to the observation of field 655, taken the 24th of September 2020 with the ztf r filter. Center: Image corresponding to the CCD ID 1, extracted from the starting image at the bottom right. Right: Image corresponding to the quadrant ID 0 extracted from the CCD ID 1 at the top right.

Next, a magnitude selection is made. In fact, as Fig. 5 illustrates, too low a magnitude results in insufficient photons and hence more bias, whereas too high a magnitude results in a high sensitivity to electrons and possible distortion of the PSF. That's why I chose to save the stars that ranged in magnitude from 14 to 18 and I was left with 473 stars.

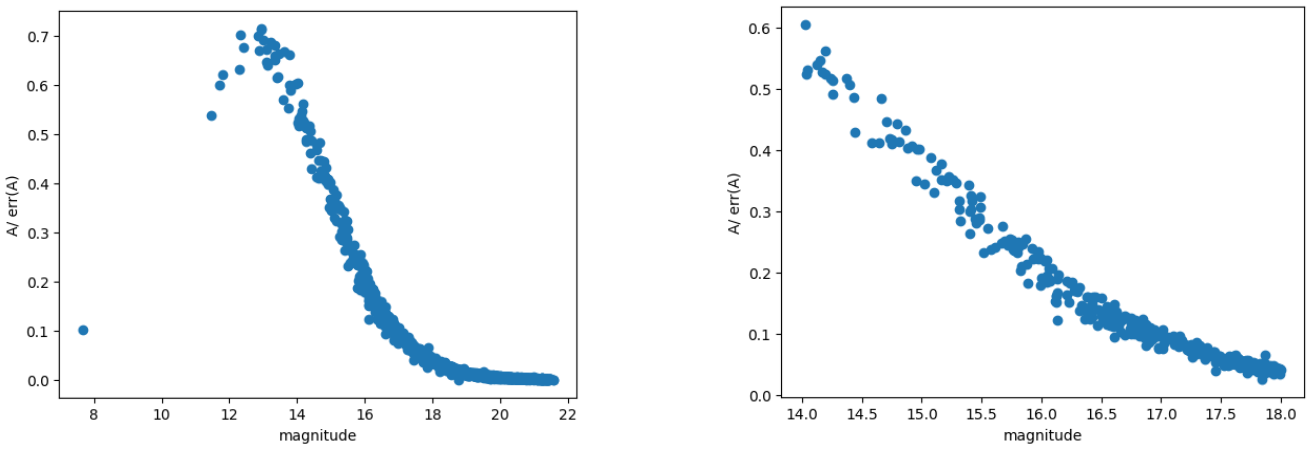


Figure 5: Signal-to-noise ratio as a function of magnitude, on the right a zoom for magnitudes between 14 and 18 with A the amplitude, err(A) the amplitude error and A/err(A) a representation of the signal-to-noise ratio.

3.4 Models

Once I have finished processing the data, I can create an image for each isolated star. Several steps were taken to analyze the PSF of these stars. First, the use of a simple model such as a Gaussian (§3.4.1) to understand how best to adjust the parameters, and learning how to use the optimizer *scipy.optimize.minimize* (§3.5.1) on a CPU. Once the optimized parameters had been found, I was able to calculate the residuals and display the results in 2D and 3D. The results will be discussed in section 4. Afterwards, I repeated the same process with JAX on a GPU with two optimizer: *optax.adam* (§3.5.2) and the *TN-CG* (§3.5.3) method. I then moved on to another model, more representative of reality: the Moffat distribution (§3.4.2), and I reproduced the same process as for the Gaussian distribution.

3.4.1 Gaussian distribution

The Gaussian model [13] is a good first approach for adjusting PSF parameters as I mentioned in §2.3. The normalized distribution is represented by the Eq. 2 where A is the amplitude, x is the value to be valued, Σ is the covariant matrix and μ the mean of the distribution.

$$f(x) = A \exp\left[-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right] \quad \text{with} \quad A = \frac{1}{\sqrt{2\pi \det(\Sigma)}} \tag{2}$$

3.4.2 Moffat distribution

The Moffat model was found by A.F.J Moffat [14] and is better fitted to the PSF as we can show Fig. 11, which actually stretches more than a simple Gaussian. The distribution of this function is represented by the Eq. 3 where A is the amplitude, γ is the width parameter, α is the shape parameter (decay rate), x, y are the parameters to be adjusted and x_0, y_0 are the center coordinates. The greater the γ, the larger the Moffat. The greater the α, the faster the Moffat decreases and the narrower the wings.

$$f(x, y, \alpha, \gamma) = A\left[1 + \left(\frac{(x-x_0)^2 + (y-y_0)^2}{\gamma^2}\right)\right]^{-\alpha} \quad \text{with} \quad A = \frac{\alpha - 1}{\pi \gamma^2} \tag{3}$$

3.5 Optimizers

3.5.1 Minimize optimizer

scipy.optimize.minimize [28] is a function in the SciPy library that provides tools for minimizing functions. This function uses various optimization algorithms to find the minimum value of a given function. There are several algorithms but the method used is "BFGS" (Broyden-Fletcher-Goldfarb-Shanno) because it made it easy to obtain errors on optimized parameters. BFGS uses an iterative method to construct an approximation of the Hessian matrix.
The Hessian matrix is a square matrix of second-order partial derivatives of a scalar-valued function. For a function $f(x)$ where x is a vector of parameters, the Hessian matrix H is defined as Eq. 4.

$$H(f) = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{4}$$

3.5.2 Adam optimizer

Adaptive Moment Estimation, often known as the Adam optimizer [29], is a well-liked optimization technique that is especially useful for neural network training in machine learning. It combines the benefits of moment and learning rate adaptation techniques with stochastic gradient descent methods. Gradient descent is an optimization method that progressively adjusts the parameters of a model by moving in the opposite direction to the gradient of the loss function, to minimize it (it follows the steepest slope). Adam adjusts each model parameter's learning rate dynamically based on the gradient history of that parameter. As a result, it can converge more quickly by adaptively changing the learning rates for every parameter. It is well-known to be reliable and effective for a range of optimization issues and neural network architectures. Because of its overall performance, it is therefore frequently chosen as the default option.

3.5.3 TN-CG optimizer

Truncated Newton Conjugate Gradient, or TN-CG for short, is a second-order optimization algorithm written by Marc Betoule and described in an article by James Martens [30]. It does this by combining truncated Newton methods with conjugate gradient methods. Compared to first-order techniques like gradient descent, TN-CG is more efficient since it makes use of information about the function's curvature. This technique uses an approximation of the Hessian via truncated Newton methods to direct the search towards convergence more quickly. Search directions are constrained to remain within a subspace generated by previous gradients. It dynamically adjusts its parameters, such as truncation factor and step size, according to the progress made in minimizing the function. In this way, it can converge faster to the global minimum of the function.

Calculating the exact Hessian matrix can be computationally expensive and impractical for several reasons in our case:

- Efficiency: The method approximates the product of the Hessian matrix and a vector, rather than computing the entire matrix. This can be done using finite differences of gradients or other approximation techniques, which are much more efficient.
- Iterative Solvers: Like the Conjugate Gradient method, the TN-CG approach iteratively approximates the solution to the Newton step (i.e., solving $H_x = -\Delta f$). These solvers only require the Hessian-vector products, which can be computed more efficiently.
- Scalability: Approximating the Hessian allows the method to scale to large models and datasets, making it practical for deep learning applications where full Hessian computation is not feasible.
- Avoiding Exact Second Derivatives: The method leverages techniques that approximate the second-order information without directly calculating second derivatives, thus reducing the computational

burden while still capturing the necessary curvature information to make informed optimization steps.

3.6 Loss functions

Calculating the quantity that a model should aim to minimize during training is the goal of loss functions [31]. It is therefore a quantitative measure of model performance, indicating the extent to which model predictions differ from actual values. Loss calculation follows the same fundamental principles, but differs slightly in implementation due to the specific characteristics of each optimization algorithm. For example, for a distribution that can be modeled by a linear function, we can use the least squares method to evaluate the loss function.

For TN-CG, the loss function is given by the user and must return a scalar value representing the loss. $jax.value\_and\_grad(func\_)$ is used to obtain both the value of the loss and its gradient with respect to the parameters. At each iteration, the loss and gradients for the current parameters are calculated. The loss is added to the *losses* list at each iteration to track the evolution of the loss over time.

For Adam, as for TN-CG, the loss function must return a scalar value representing the loss. $jax.grad(func\_)$ is used to obtain the gradients of the loss function with respect to the parameters. At each iteration, the gradients are calculated. Parameter updates are calculated using gradients, and the loss is calculated after each parameter update. It is added to the *losses* list at each iteration to track the evolution of the loss over time.

In both functions, the loss is calculated at each iteration by evaluating the loss function with the current parameters. To know when the training should stop, we specify a tolerance threshold, which corresponds to the difference between two iterations. This threshold is chosen so as to have enough iterations to be sure that the minimum is reached, and at the same time not too many iterations to gain time. In our case, we have chosen the tolerance equal to 10^{-5}.

4 Results

For the two distributions used (Gaussian and Moffat), different codes were written on CPU and GPU. To compare execution times, benchmarks were run under the same conditions as similar as possible and all the following figures will represent the same star (Index 3292 in my table).

4.1 Gaussian model

4.1.1 CPU framework: Minimize optimizers

For the first optimization, I used the *stats.multivariate_normal* function from Scipy to model the gaussian function which I multiplied the amplitude parameter and I added the background parameter (to place the scale at zero). Then I used the *optimize.minimize* function from Scipy with BFGS method to fit the data on the model. I first wrote the code for one star and once it was working, I ran a for loop on

10, 50, 100, 300 and all the stars in the table, called DataFrame in computing language (i.e. 473 stars). I have calculated the execution time for each and the results are shown in Fig. 10 (for more results see Appendix B). To optimize this part, I have eliminated the for loop by creating a *custom_function* with pandas [32]. Finally, execution time increase linearly but is not impacted by the loop change (Fig. 10). In order to make a statistical analysis of the results, each code was executed 100 times (which took over 8 hours), I stored the values in a table and calculated $\frac{\sigma}{\sqrt{100}}$ with σ the standard deviation. Finally, the error bars were added to the figure.

For a star (shown in Fig. 6), the χ^2 obtained which actually corresponds to the least square, because I do not have access to the error map, is around 7.45×10^{-5}. The value is therefore not relevant but the figures show that the fit is coherent. The fitted parameters obtained are shown in the table 1. The errors for the parameters A, σ_x, σ_y and b seem coherent with their fitted parameter however errors for x_0 and y_0 are higher than the fitted values which is suspicious. I calculated them recently, I will come back to check these values.

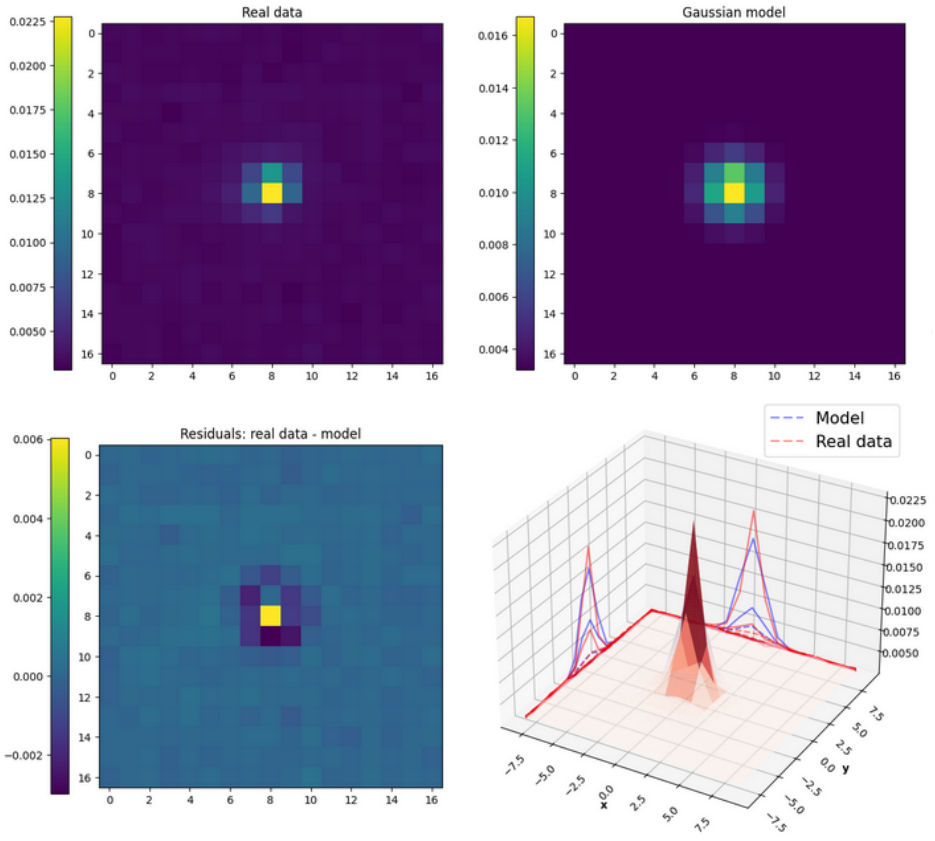


Figure 6: Plot showing 2D distribution of real data, fitted Gaussian model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the CPU framework with the *optimize.minimize* optimizer from Scipy.

Parameters	Optimized Value	Error
x_0	0.47144	0.65313
y_0	0.24933	0.34438
A	0.08768	0.05297
σ_x	0.99996	0.00953
σ_y	0.99996	0.00948
b	0.00319	0.00084
χ^2	7.44529e-05	

Table 1: Fitted parameters obtained for a star (Index 3292) with Gaussian distribution and *optimize.minimize* on CPU.

4.1.2 GPU framework: Adam and TN-CG optimizers

For GPU optimization, two methods were used: a general method (Adam) and a more complex method (TN-CG). In this part, I do not have time to compute the errors on the fitted parameters because I got the results while writing these report.

Adam optimizer

I followed the same procedure as before, the change being the use of the *optax.adam* [33] optimizer and the use of JAX on GPU. I have also removed the for loop and replaced it with broadcasting [34], since the GPU is adapted to parallelization. For a star, the χ^2 obtained, which corresponds to the least square, is around 0.144 (see the fitted parameters on table 2). The value is quite small and therefore shows that the chosen model is consistent with the real data as we can show on the Fig. 7 (for more results see Appendix C). As for the execution time, the fact that for a sample of 473 stars, the time is constant and small (around 6 seconds see Fig. 10) is a very good thing. Keep in mind, however, that this sample does not fill the GPU, so we need to test with more images to see if the execution time increases (we would expect it to increase when changing batch and then become constant again afterwards, like stairs).

Parameters	Optimized Value
x_0	0.018556
y_0	0.254136
A	0.082898
σ_x	0.914114
σ_y	0.916828
b	0.082898
χ^2	0.14382

Table 2: Fitted parameters obtained for a star (Index 3292) with Gaussian distribution and *Adam* optimizer on CPU.

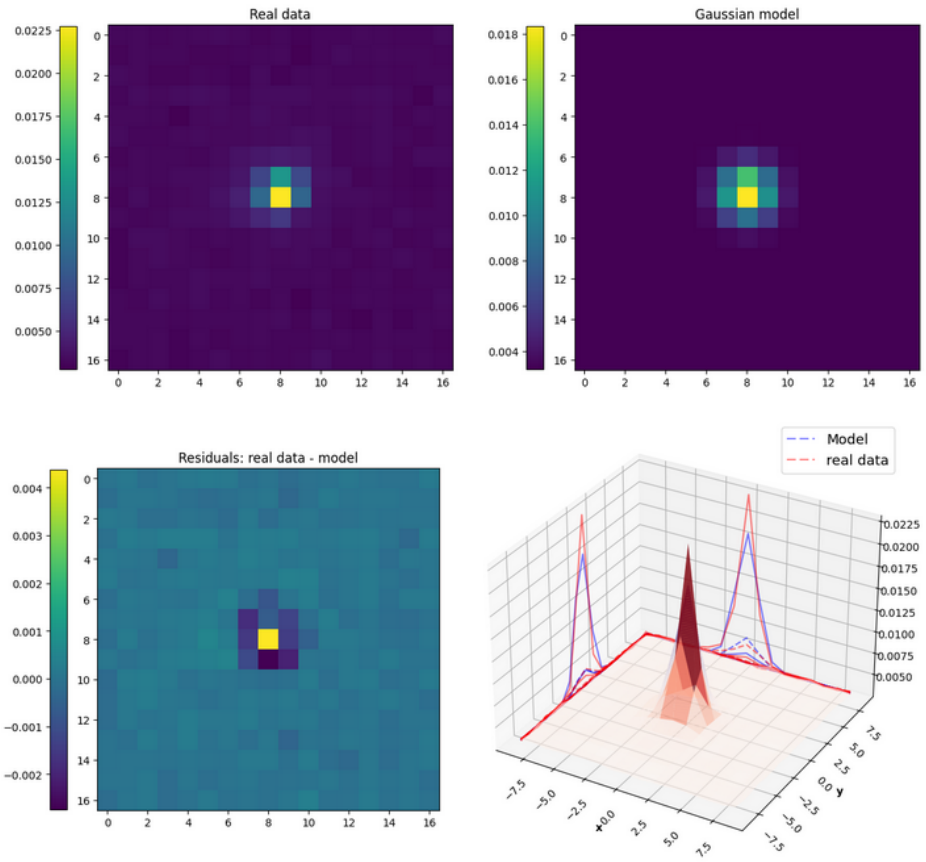


Figure 7: Plot showing 2D distribution of real data, fitted Gaussian model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the GPU framework with the *optax.adam* method.

TN-CG optimizer

For the benchmark, I followed the same procedure as before, the change being the use of the TN-CG optimizer. For a star, the χ^2 obtained, which corresponds to the least square, is around 0.10 (see the fitted parameters on table 3). The value is quite small and therefore shows that the chosen model is consistent with the real data as we can show on the Fig. 8 (for more results see Appendix D). Moreover, this model seems more representative than Adam model because the least square is better. For execution time, we can apply the same analyse as for the fit with Adam, but it is faster (around 3 seconds see Fig. 10).

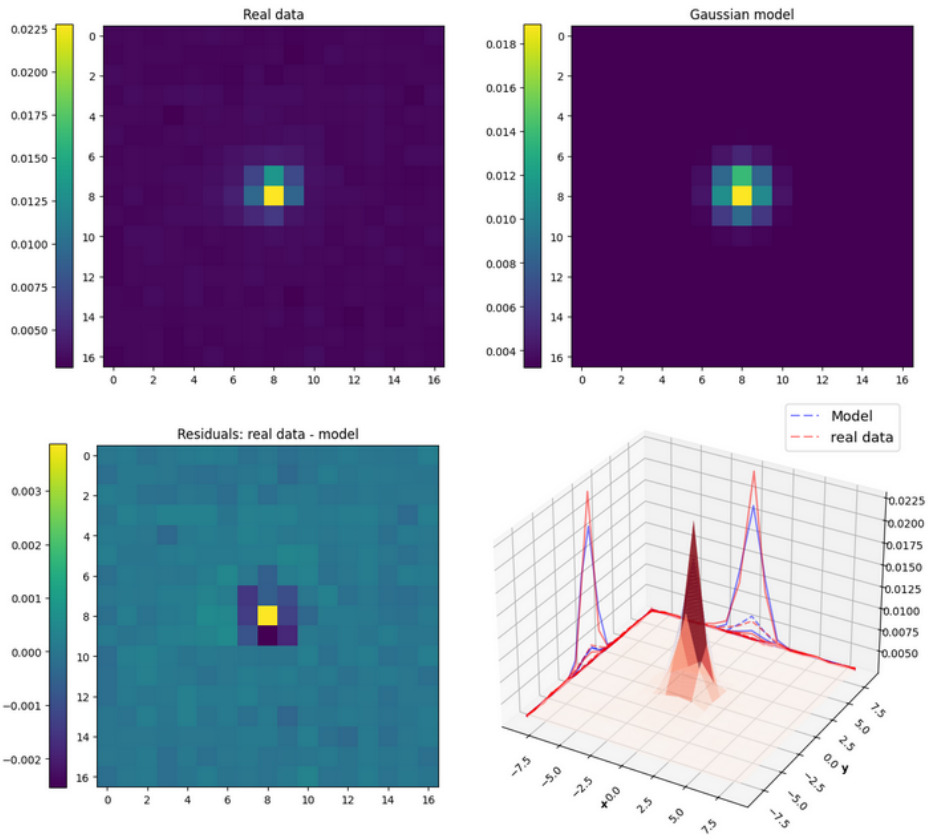


Figure 8: Plot showing 2D distribution of real data, fitted Gaussian model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the GPU framework with the TN-CG method.

Parameters	Optimized Value
x_0	0.016890
y_0	0.247583
A	0.079343
σ_x	0.879041
σ_y	0.882730
b	0.003217
χ^2	0.10835

Table 3: Fitted parameters obtained for a star (Index 3292) with Gaussian distribution and *TN-CG* optimizer on CPU.

Loss functions

The loss function displayed in Fig. 9 represents the sum for all stars (in this case 473 stars), as the model assumes that all stars have the same shape. The loss function is defined to within a constant, so it does not matter where the function starts or how long it takes, the most important thing is to find out when the global minimum is reached. We can compare the execution time for one iteration for each optimizer knowing that the loss function training stops when the tolerance threshold of 10^{-5} is reached.

For 473 stars:

- For Adam: execution time = 21.7s, there were 500 iterations which means 0.00434s by iteration.
- For TN-CG: execution time = 5.39s, there were 50 iterations which means 0.10789s by iteration.

This means that Adam is faster at doing each iteration, but slower at finding the global minimum, so it needs more iterations.

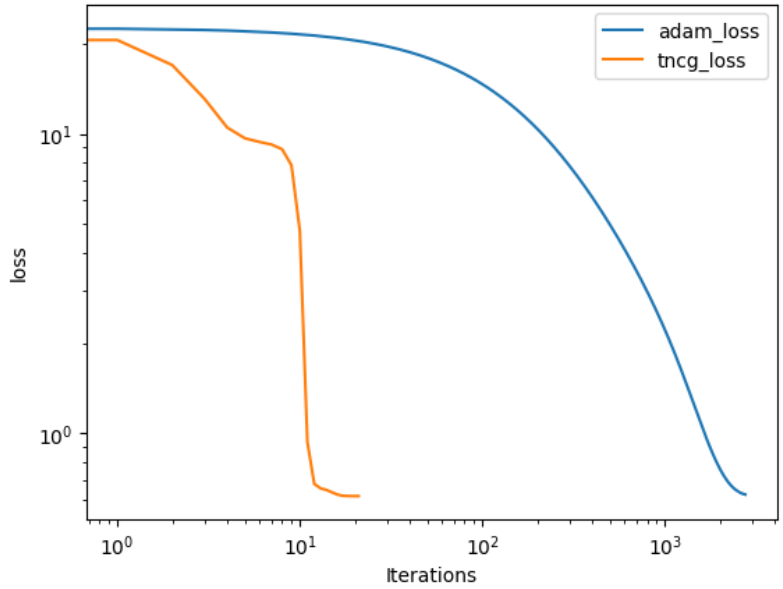


Figure 9: Loss function for Gaussian fitting with Adam and TN-CG optimizers on GPU with JAX.

4.1.3 Summary

By comparing execution times for CPU and GPU frameworks Fig. 10, we have shown that GPU usage is much more efficient when we want to process a large data set. We can also note that the use of the TN-CG optimization method is more efficient than Adam optimizer.

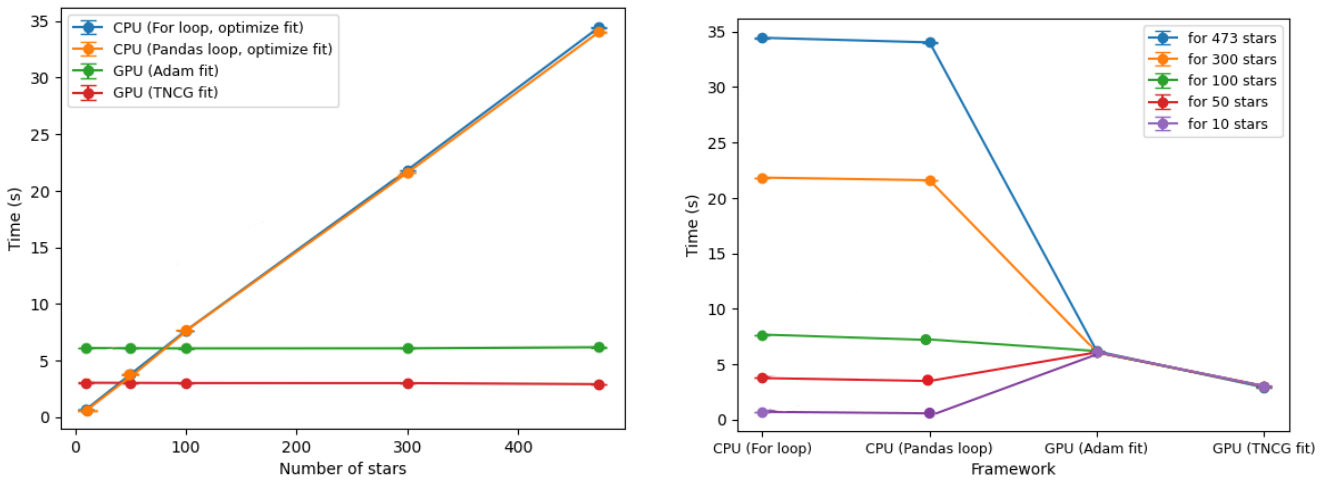


Figure 10: Left: Execution time for PSF optimization with the Gaussian model, as a function of the number of stars, for different frameworks. Right: Execution time for PSF optimization using the Gaussian model, as a function of different frameworks for a given number of stars.

4.2 Moffat model

The Moffat distribution as I explained in §2.3, is a better representation of the PSF than the Gaussian distribution and we can see the 3D representation on Fig. 11.

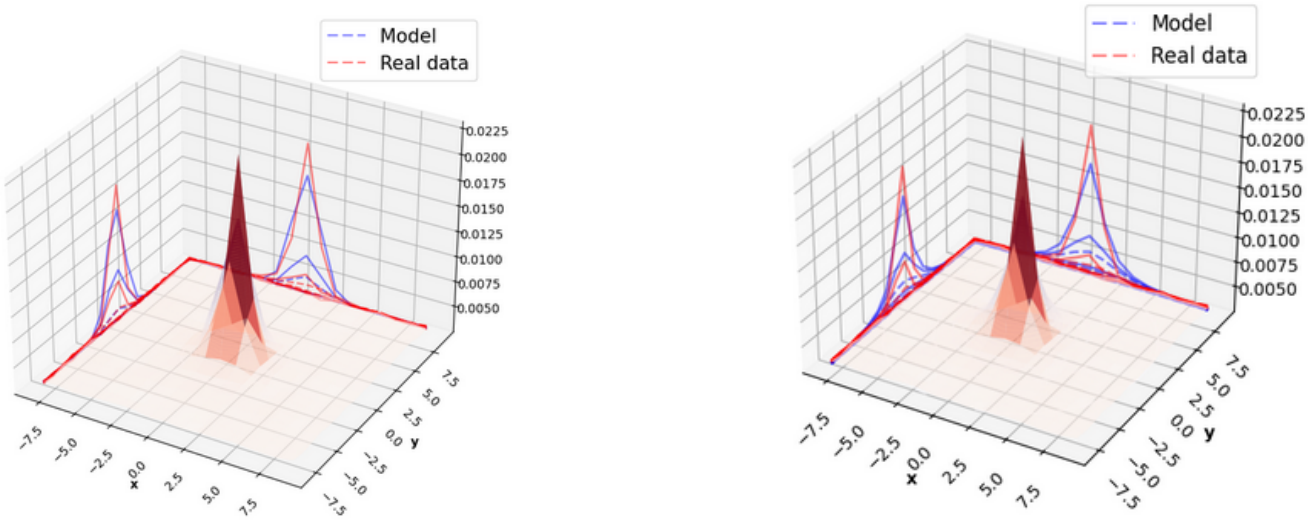


Figure 11: Comparison of the optimization method (in blue) on the real data (in red) for one star: on the left the Gaussian distribution and on the right the Moffat distribution.

4.2.1 CPU framework: Minimize optimizer

The same steps as describe in §3.5.1 were applied for the Moffat function. The χ^2 for one star, which corresponds to the least square, is 0.99×10^{-4}. For the same reason as explained on §4.1.1, the value is not relevant but the Fig. 12 show that the Moffat is a good fitter (for more results see Appendix E). The fitted parameters are shown in table 4. The fact that the errors on the parameters x_0 and y_0 are similar to those obtained with the Gaussian distribution seems suspicious. Indeed, the Moffat distribution should

have lower errors than the Gaussian distribution, since it is more closely adapted to reality.

Concerning the execution time, as for §3.5.1, we find that it increases linearly and does not depend on the method to loop the data (for or pandas loop) see Fig. 16.

Parameters	Optimized Value	Error
x_0	0.48614	0.6460
y_0	0.27890	0.3563
A	0.12203	0.0376
γ	0.99994	0.0100
α	1.00007	0.0100
b	0.00280	0.0010
χ^2	9.85163e-05	

Table 4: Fitted parameters obtained for a star (Index 3292) with Moffat distribution and *optimize.minimize* on CPU.

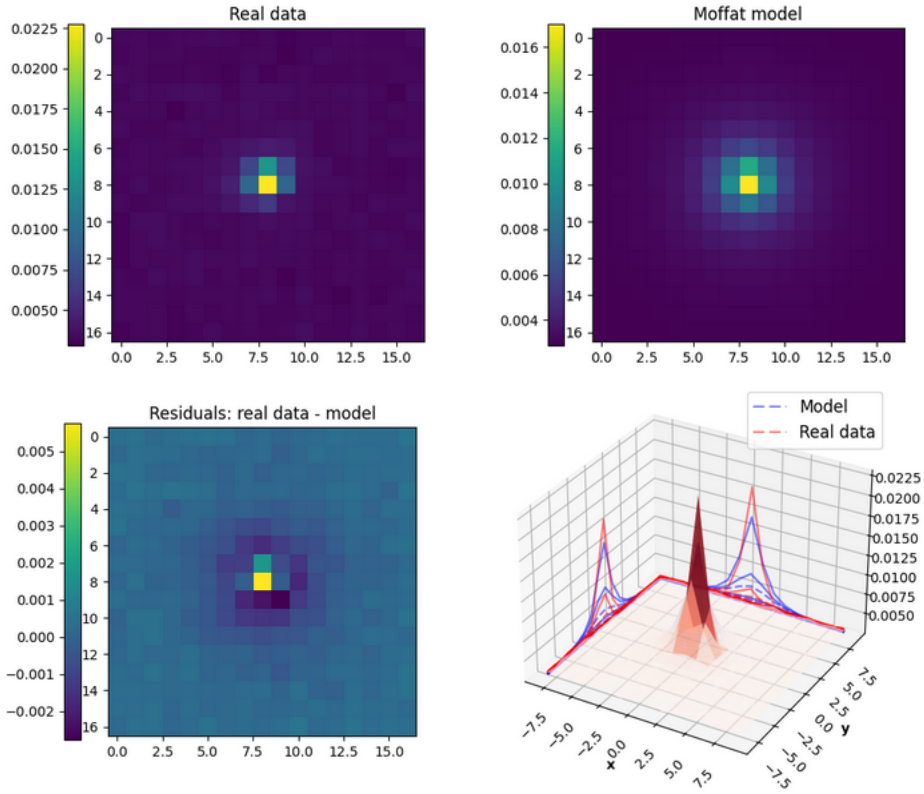


Figure 12: Plot showing 2D distribution of real data, fitted Moffat model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the CPU framework with the *optimize.minimize* optimizer from Scipy.

4.2.2 GPU framework: Adam and TC-CG optimizer

As before, after testing my code on the CPU, I ran it on the GPU with the Adam and TN-CG optimizers. In this part, I don't have time to compute the errors on the fitted parameters because I got the results in the last few days.

Adam optimizer

I followed the same procedure as describe in §4.1.2. For a star, the χ^2 obtained is around 0.139 which corresponds to the least square (see the fitted parameters on table 5) and is slightly smaller than with the Gaussian distribution, which means that the model fits better. The value is quite small and therefore shows that the chosen model is consistent and reliable for real data as we can show on the Fig. 13 (for more results see Appendix F). As for the execution time, the fact that for a sample of 473 stars, the time is constant but not small (around 30 seconds see Fig. 16) is not a very good thing.

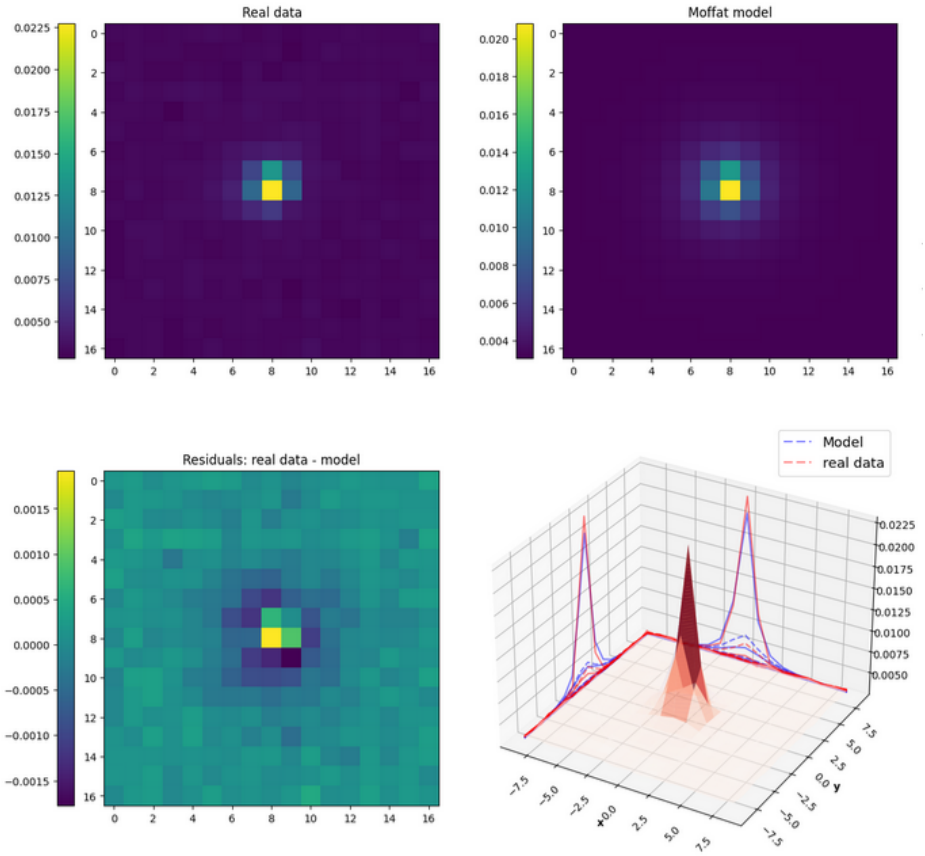


Figure 13: Plot showing 2D distribution of real data, fitted Moffat model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the GPU framework with the *optax.adam* method.

Parameters	Optimized Value
x_0	0.104471
y_0	0.284396
A	0.020781
γ	0.792515
α	1.130729
b	0.003004
χ^2	0.13909

Table 5: Fitted parameters obtained for a star (Index 3292) with Moffat distribution and *Adam* optimizer on CPU.

TN-CG optimizer

The results of this final stage are promising. For a star, the χ^2 obtained is around 0.154 which corresponds to the least square (see the fitted parameters on table 6). The value is higher than that obtained with the Gaussian distribution, but it is to be hoped that the errors will be smaller and therefore the likelihood better. In addition, the chosen model is consistent and reliable for real data as we can show on the Fig. 14 (for more results see Appendix G). For execution time, we can apply the same analyse as for the fit with Adam, but it is considerably faster (around 2 seconds see Fig. 16).

Parameters	Optimized Value
x_0	0.019067
y_0	0.248053
A	0.019664
γ	0.843164
α	1.154012
b	0.002990
χ^2	0.15470

Table 6: Fitted parameters obtained for a star (Index 3292) with Moffat distribution and *TN-CG* optimizer on CPU.

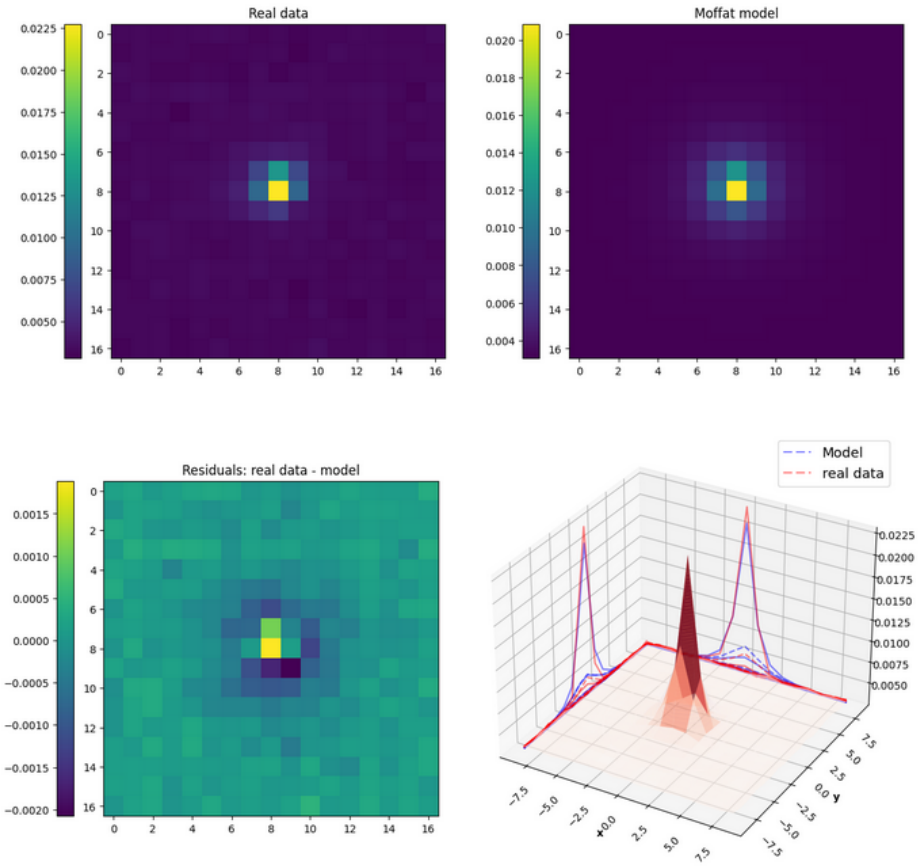


Figure 14: Plot showing 2D distribution of real data, fitted Moffat model and residuals for one star. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the GPU framework with the TN-CG method.

Loss functions

Here (Fig. 15) we see, as before, that the TN-CG function is more efficient and reaches the global minimum faster than the Adam function. What we did not have before, however, was the bump on the Adam loss function. In fact, there are two possibilities, either the loss function as reached a local minimum before reaching the global minimum or there could be an over fitting, which I will check in the future by comparing the test function and the training function. We can compare the execution time for one iteration for each optimizer.
For 473 stars:

- For Adam: execution time = 78s, there were 15000 iterations which means 0.0052s by iteration
- For TN-CG: execution time = 2.52s, there were 50 iterations which means 0.0504s by iteration

we can conclude the same thing as before, Adam is faster at doing each iteration, but slower at finding the global minimum, so he needs more iterations.

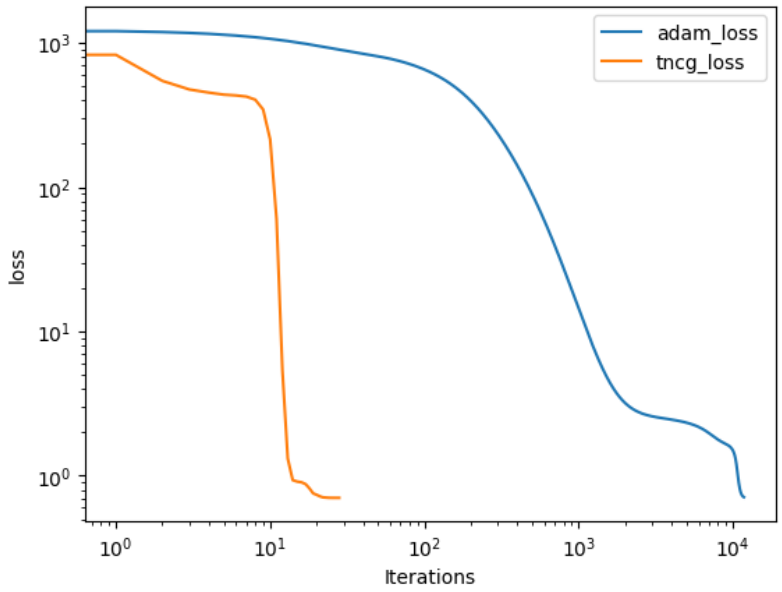


Figure 15: Loss function for Moffat fitting with Adam and TN-CG optimizers on GPU with JAX.

4.2.3 Summary

The same conclusion can be made as above, with the Gaussian distribution. By comparing execution times for CPU and GPU frameworks Fig. 16, we have proof that GPU usage is much more efficient when we want to process a large data set. We can specify that the use of the TN-CG optimization method is very more efficient than Adam optimizer (more than a minute on CPU and 30 seconds for Adam compared with 2 seconds for TN-CG on GPU beyond 300 stars). NVIDIA promises that training on V100 GPUs will be 32 times faster than a CPU [20], which is the case here

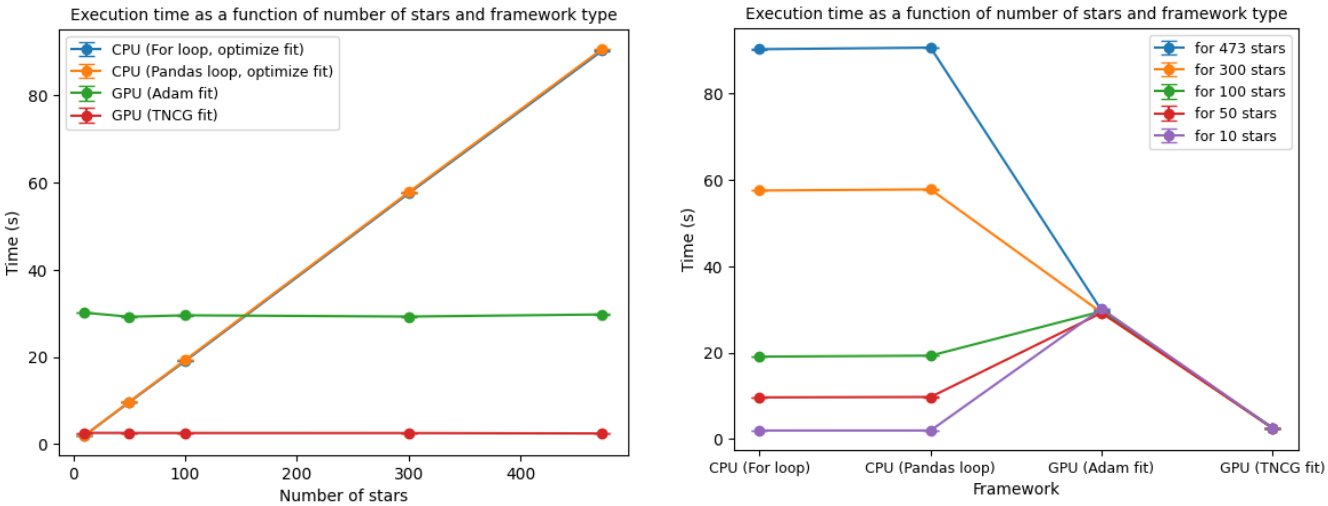


Figure 16: Left: Execution time for PSF optimization with the Moffat model, as a function of the number of stars, for different frameworks. Right: Execution time for PSF optimization using the Moffat model, as a function of different frameworks for a given number of stars.

Conclusions and Perspectives

The aim of my internship was to find a way of optimizing the PSF code for the ZTF environment. To achieve this, I used two distributions to model the PSF. To develop and validate the code on CPU and GPU I used, as a first approach, the Gaussian distribution model, even if it does not correspond well to reality. The rest of the work for the ZTF project will be done on GPU using a Moffat distribution (current model used in cosmology), since it better models the tails of the PSF distribution.

As the official code (written in C++) is to be used on a CPU, I started by testing my code on a CPU, then parallelized it using a GPU. On CPU, we observe that the greater the data quantity, the longer the code execution time, which can exceed one minute for 500 stars. This is a major inconvenience, given that we know this corresponds to a single quadrant (there are 64 quadrants in total). After using a GPU, we realize that we gained a factor of 30 in code execution time as predicted by NVIDIA. Moreover, the execution time remains approximately the same regardless of the data to be processed, as we have not yet filled the GPU cores. The GPU is therefore undeniably more efficient than the CPU. We can also say that JAX is a good framework, well adapted to our needs because it is a NumPy-like code. However, it is worth pointing out that the CPU tests only run on a single core, as we wanted to respect as closely as possible the way in which the official code is executed. GPU tests are necessarily carried out on several cores, since the aim is to parallelize it. One way of parallelizing on CPU would be to use the Dask framework [35] on Python, with structures similar to NumPy's arrays and Pandas' DataFrames. This would allow to choose between CPU and GPU, depending on the machines available, while having the most optimized code possible.

To complete this study, in the next month, I will:

- check the error calculation of the optimized parameters for the *optimize.minimize* optimizer with the Gaussian and Moffat models discussed on §4.1.1 and §4.2.1 and also do the calculation with Adam and TN-CG optimizer with JAX,
- compare the test and training functions with respect to the loss function as explained on §4.2.2,
- add another parameter in the fit, the pixelgrid (6144×6160 pixels) which means that for the Moffat, x_0 and y_0 parameters should no longer depend only on the location of the pixel on the image but should also depend on the location on the pixel itself,
- run my code with the whole image (all 64 quadrants) and for several images afterwards,
- implement Dask option to parallelize on CPU.

On longer term, the aim is to implement my code in the official ZTF software and, if time allows, to add a mechanism in the form of a back-end to use the most appropriate CPU or GPU framework to model the PSF, depending on the computing machine chosen and/or available (JAX or Dask).

References

[1] Un univers sans matière noire? https://lejournal.cnrs.fr/articles/un-univers-sans-matiere-noire-0. Accessed: May 21, 2024.

[2] Zwicky transient facility website. https://www.ztf.caltech.edu/. Accessed: May 14, 2024.

[3] M. R. Magee et al : Determining the 56ni distribution of type ia supernovae from observations within days of explosion. *Astronomy & Astrophysics*, 634(A37), 2020.

[4] Adam G. Riess et al : Observational evidence from supernovae for an accelerating universe and a cosmological constant. *The Astronomical Journal*, 116, 1998.

[5] Edwin Hubble : A relation between distance and radial velocity among extra-galactic nebulae. *PNAS*, 15, 1929.

[6] Ozgur Akarsu et al : λscdm model: A promising scenario for alleviation of cosmological tensions. *arXiv*, 2023.

[7] y D. Scott et G.F. Smo : Cosmic microwave background. *arXiv*, 2019.

[8] James D Keegans et al : Type Ia Supernova Nucleosynthesis: Metallicity-Dependent Yields. *The Astrophysical Journal Supplement Series*, 268, 2023.

[9] Mickaël Rigault : ztfimg repository. https://github.com/MickaelRigault/ztfimg. Accessed: May 13, 2024.

[10] Gaia Data Release 3 (Gaia DR3). https://www.cosmos.esa.int/web/gaia/dr3. Accessed: May 21, 2024.

[11] P.Astier et al : Photometry of supernovae in an image series: methods and application to the supernova legacy survey (SNLS). *Astronomy & Astrophysics*, 8557(A55), 2013.

[12] Yuhan Yao et al : ZTF early observations of type ia supernovae i: Properties of the 2018 sample. *The Astrophysical Journal*, 866(2), 2019.

[13] Gaussian function. https://en.wikipedia.org/wiki/Gaussian_function. Accessed: May 29, 2024.

[14] A.F.J Moffat : A theoretical investigation of focal stellar images in the photographic emulsion and application to photographic photometry. *Astronomy & Astrophysics*, 3, 1969.

[15] official website of TensorFlow. https://www.tensorflow.org/?hl=fr. Accessed: May 21, 2024.

[16] official website of PyTorch. https://pytorch.org/. Accessed: May 21, 2024.

[17] JAX documentation. https://jax.readthedocs.io/en/latest/quickstart.html. Accessed: May 13, 2024.

[18] NumPy documentation. https://numpy.org/. Accessed: May 21, 2024.

[19] SciPy documentation. https://scipy.org/. Accessed: May 21, 2024.

[20] Sign up and get instant access to V100 GPUs. https://nebius.ai/v100. Accessed: May 17, 2024.

[21] JAX github. https://github.com/google/jax. Accessed: May 21, 2024.

[22] Autograd github. https://github.com/hips/autograd. Accessed: May 29, 2024.

[23] XLA : optimiser le compilateur pour le machine learning. https://www.tensorflow.org/xla?hl=fr. Accessed: May 29, 2024.

[24] Mickaël Rigault : ztfquery repository. https://github.com/MickaelRigault/ztfquery. Accessed: May 13, 2024.

[25] Mickaël Rigault : ztfin2p3 repository. https://github.com/MickaelRigault/ztfin2p3. Accessed: May 17, 2024.

[26] Gaia DR2 catalog. https://www.cosmos.esa.int/web/gaia/dr2. Accessed: May 15, 2024.

[27] Rigault, Smith et al : submitted. *Astronomy & Astrophysics*, 2024.

[28] scipy.optimize.minimize documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html. Accessed: May 28, 2024.

[29] Diederik P. Kingma et Jimmy Lei Ba : ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *conference paper at the 3rd International Conference for Learning Representations, San Diego,*, 2015.

[30] James Martens : Deep learning via hessian-free optimization. *University of Toronto*, 2010.

[31] What are loss functions? https://towardsdatascience.com/what-is-loss-function-1e2605aeb904. Accessed: May 29, 2024.

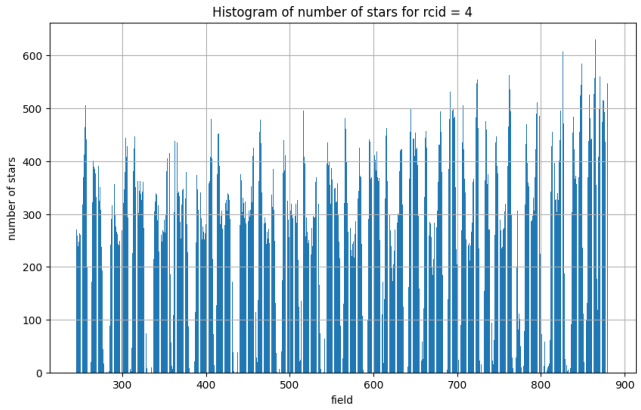
[32] pandas documentation. https://pandas.pydata.org/. Accessed: May 21, 2024.

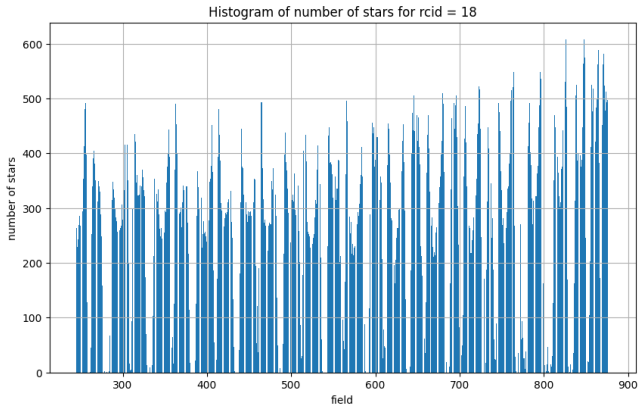
[33] Optax documentation. https://optax.readthedocs.io/en/latest/api/optimizers.html. Accessed: May 21, 2024.

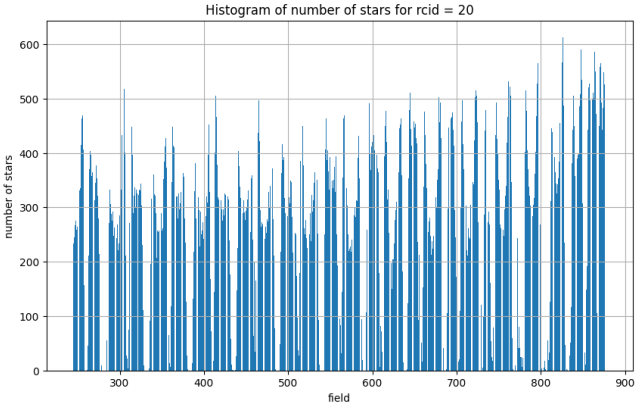
[34] Broadcasting. https://numpy.org/doc/stable/user/basics.broadcasting.html. Accessed: May 28, 2024.

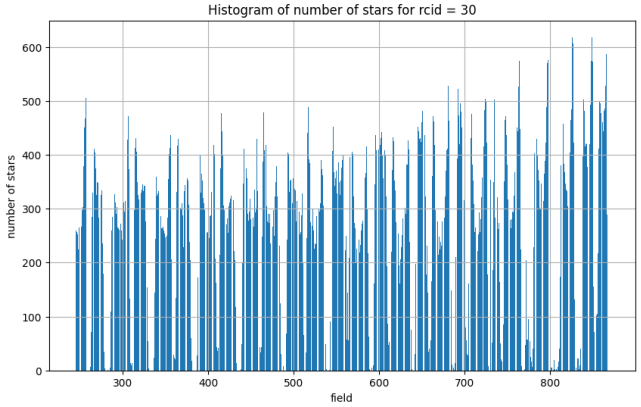
[35] Dask. https://www.dask.org/. Accessed: May 30, 2024.

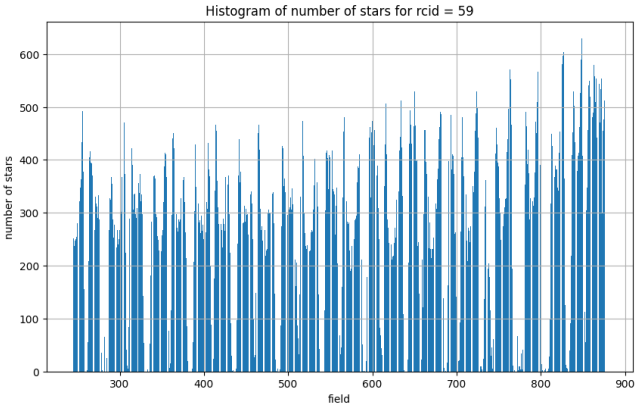
Appendix A: Distribution of number of stars for different quadrants

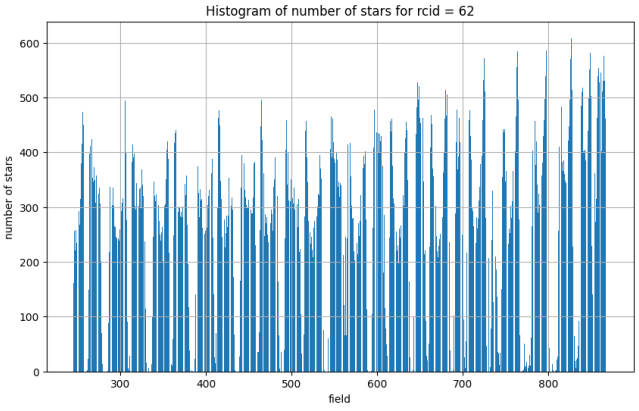






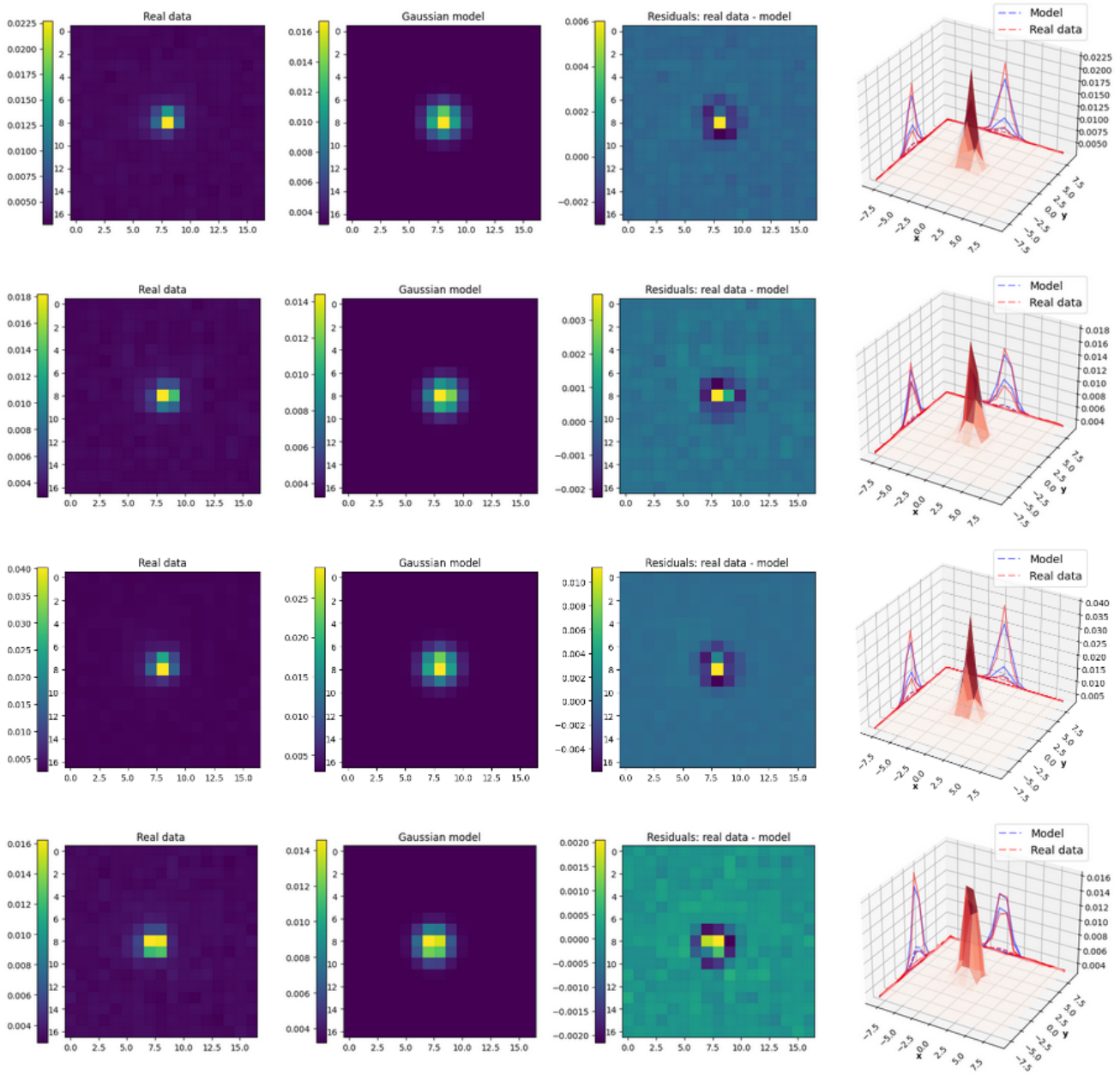


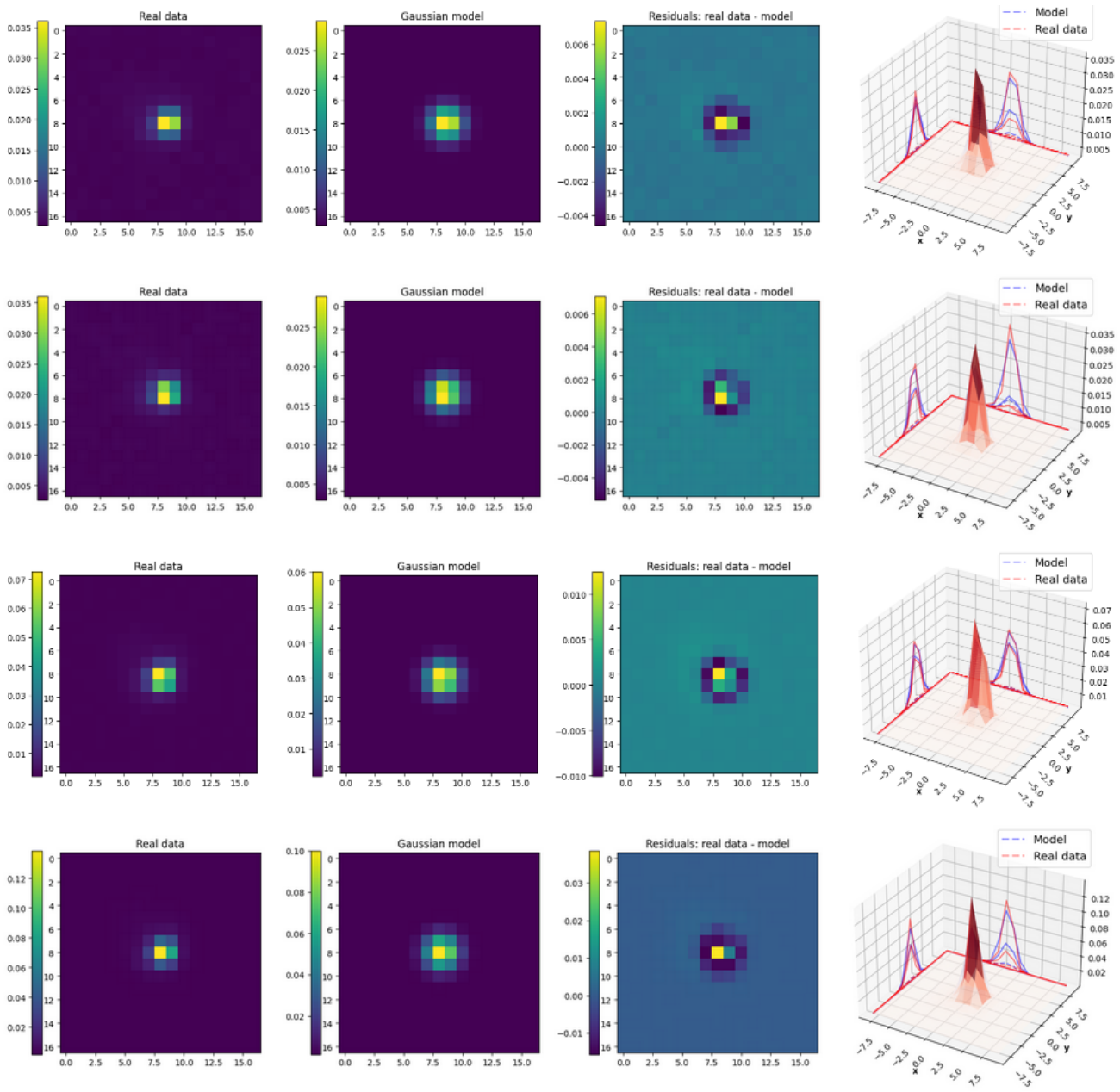




Appendix B: Plots 2D and 3D for Gaussian model with CPU framework

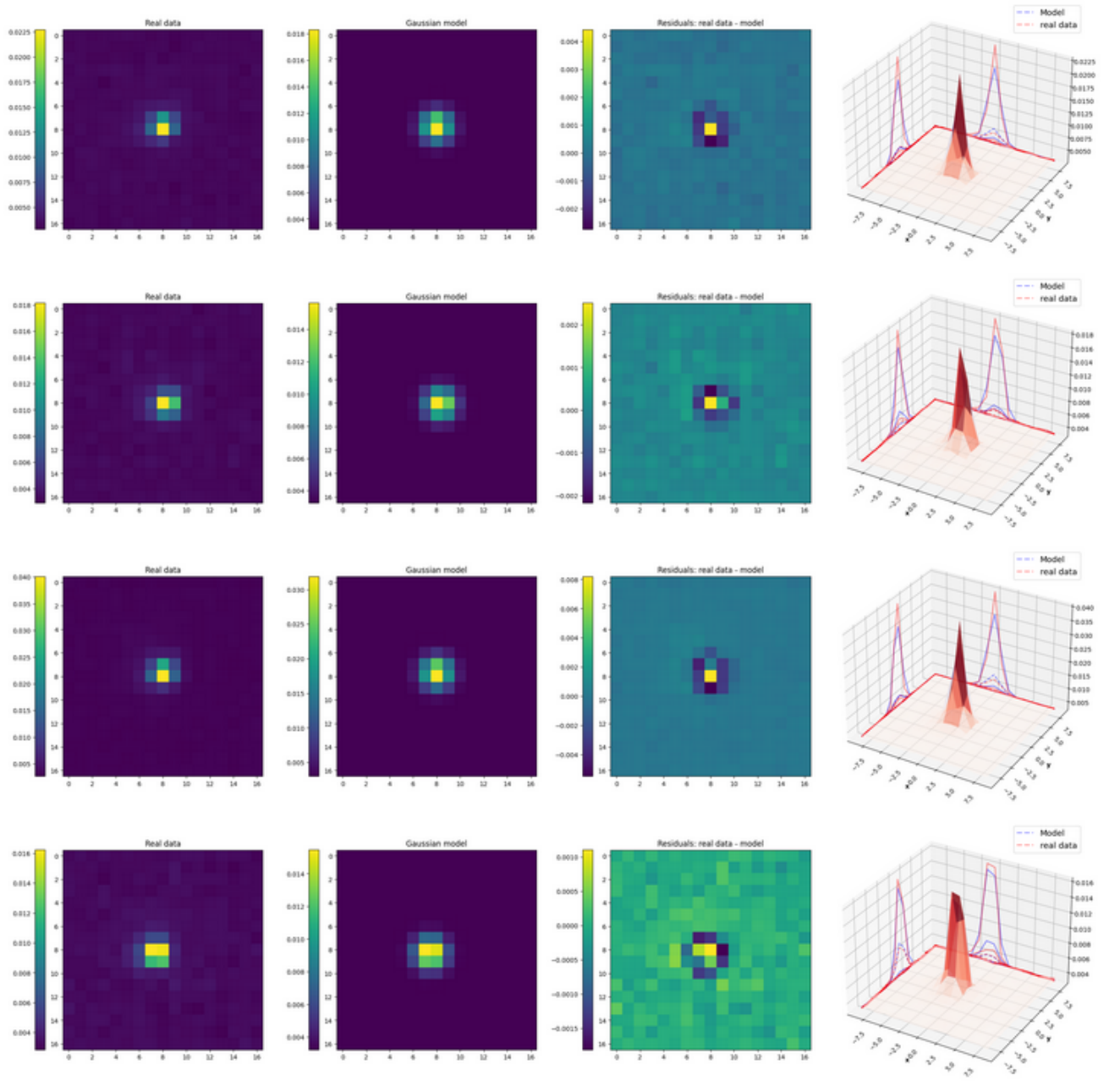
Plots showing 2D distribution of real data, fitted Gaussian model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the CPU framework with the *optimize.minimize* optimizer from Scipy.

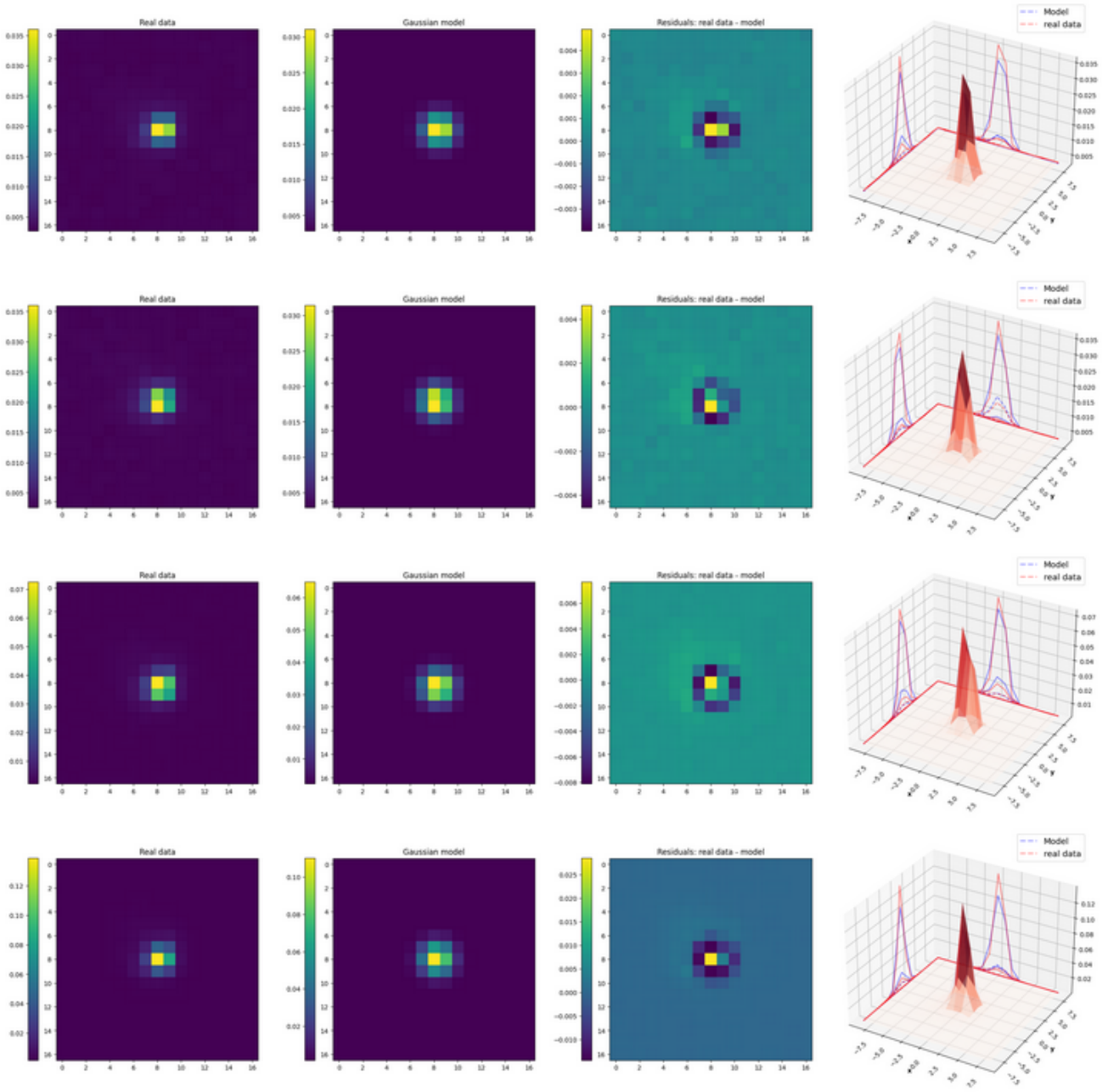



Real data
Gaussian model
Residuals: real data - model
Model
Real data
x
y

Appendix C: Plots 2D and 3D for Gaussian model with GPU framework and Adam optimizer

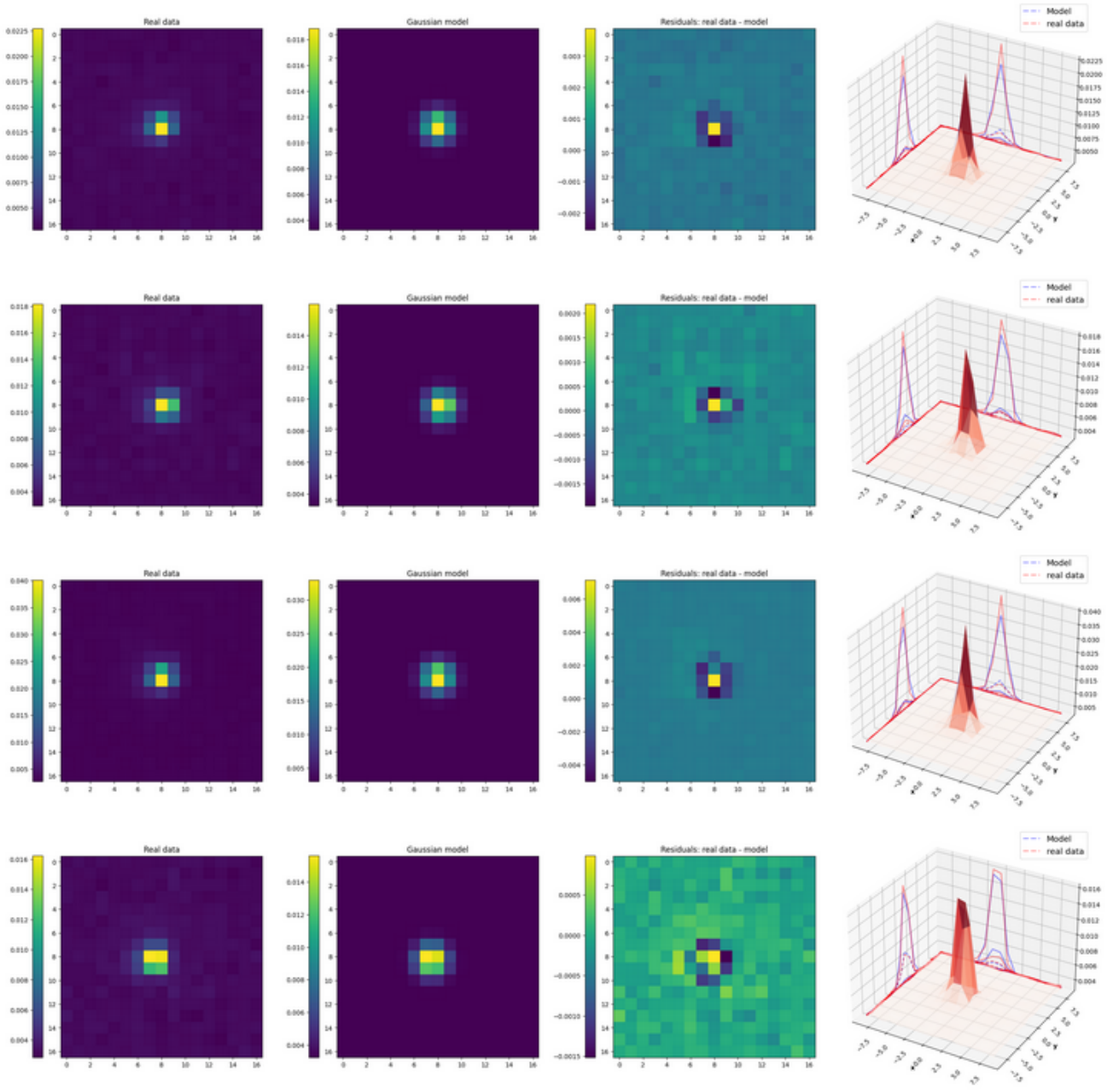
Plots showing 2D distribution of real data, fitted Gaussian model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the GPU framework with the *optax.adam* method.

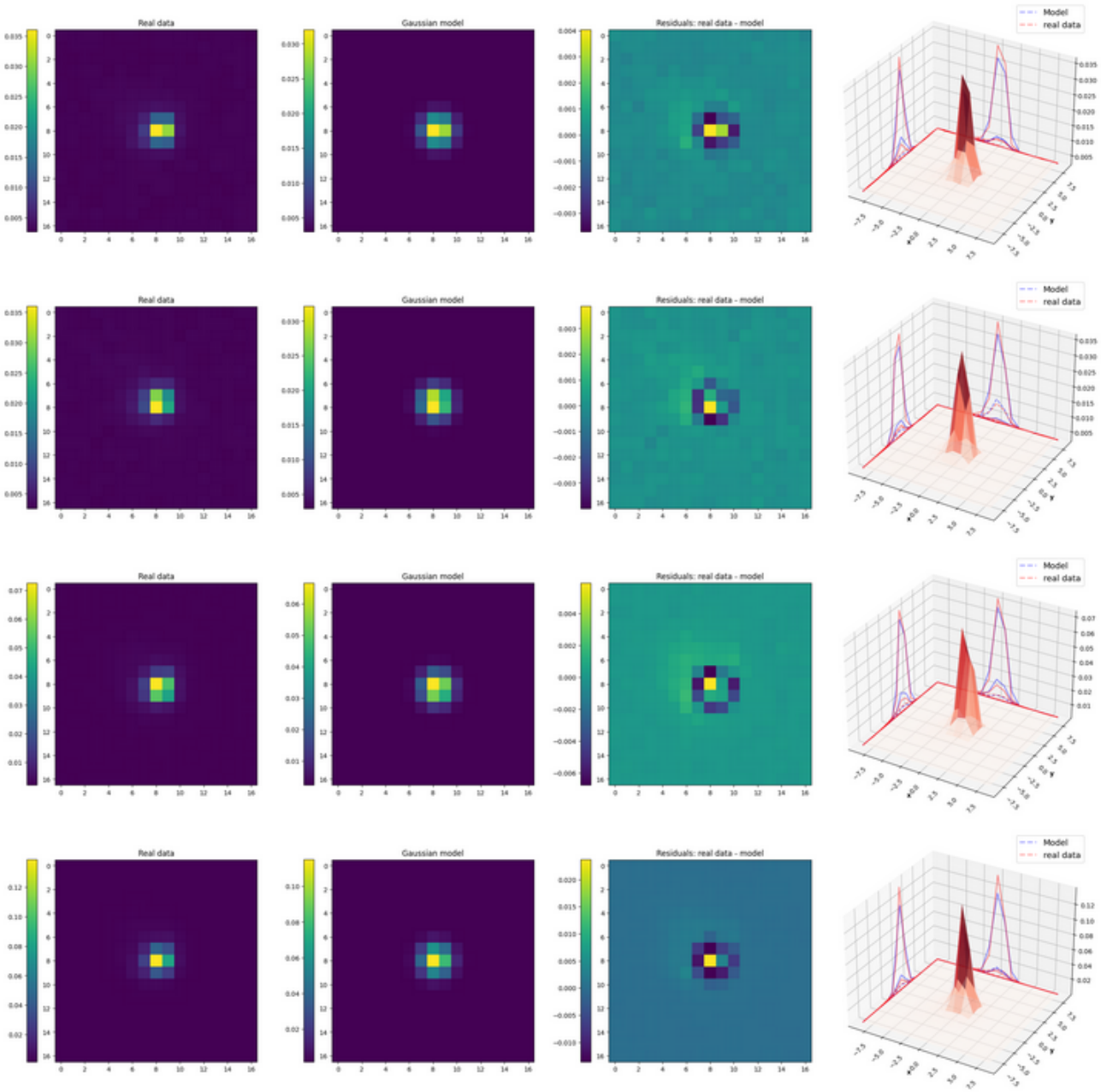



Real data
Gaussian model
Residuals: real data - model
Model
real data
Real data
Gaussian model
Residuals: real data - model
Model
real data
Real data
Gaussian model
Residuals: real data - model
Model
real data
Real data
Gaussian model
Residuals: real data - model
Model
real data

Appendix D: Plots 2D and 3D for Gaussian model with GPU framework and TN-CG optimizer

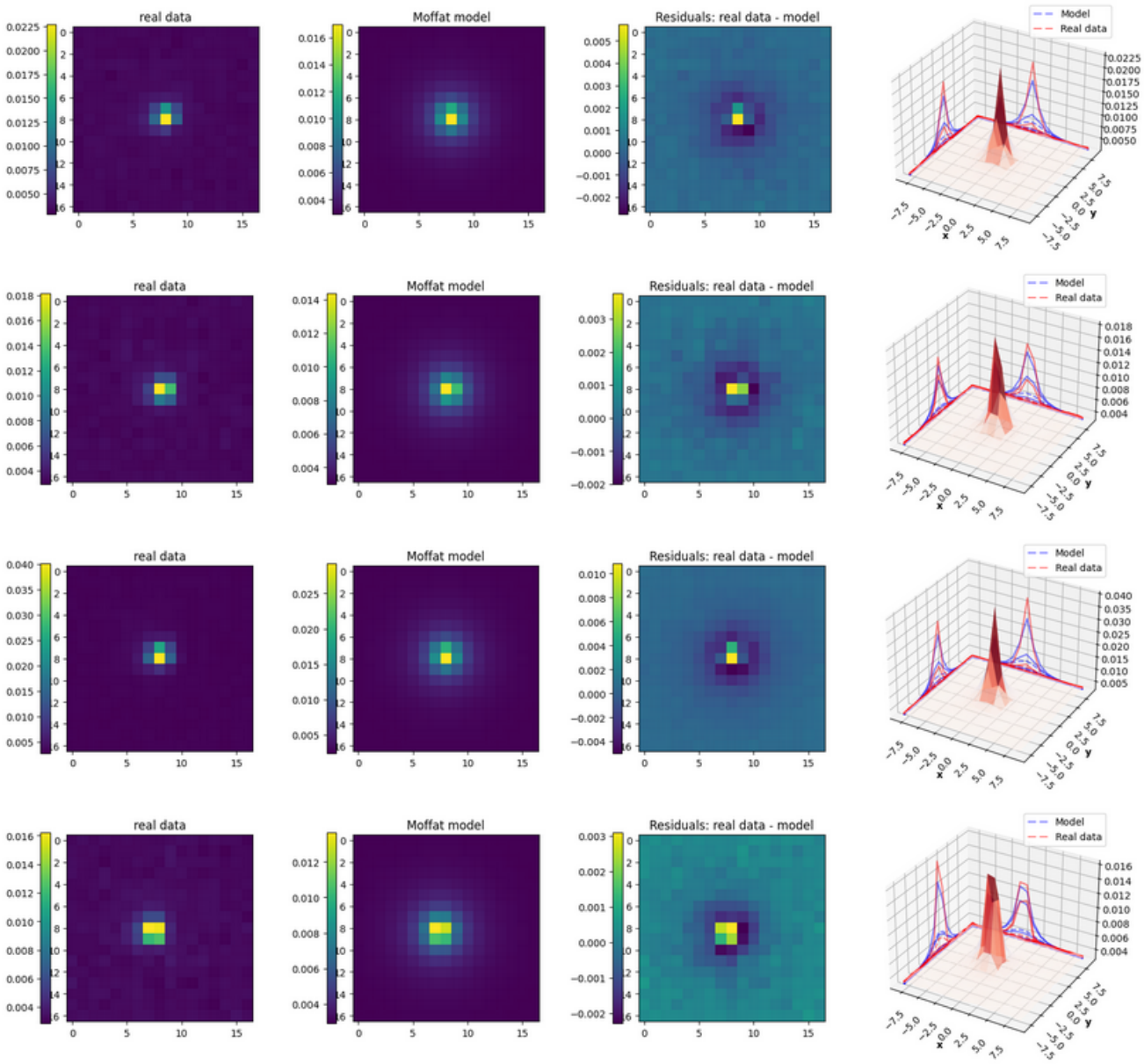
Plots showing 2D distribution of real data, fitted Gaussian model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Gaussian model (in blue). Each plot is made on the GPU framework with the TN-CG method.

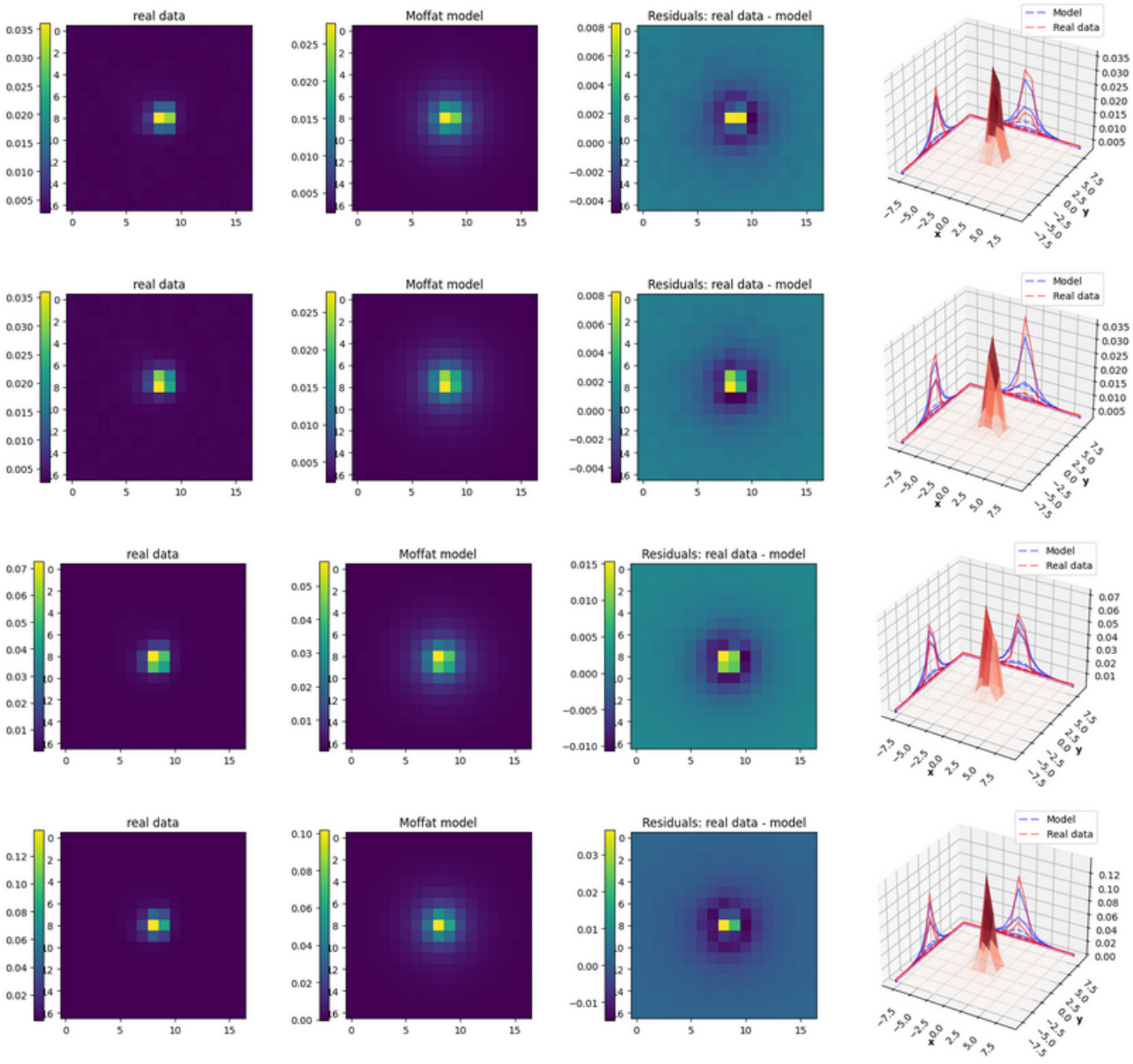



Real data
Gaussian model
Residuals: real data - model
Model
real data

Appendix E: Plots 2D and 3D for Moffat model with CPU framework

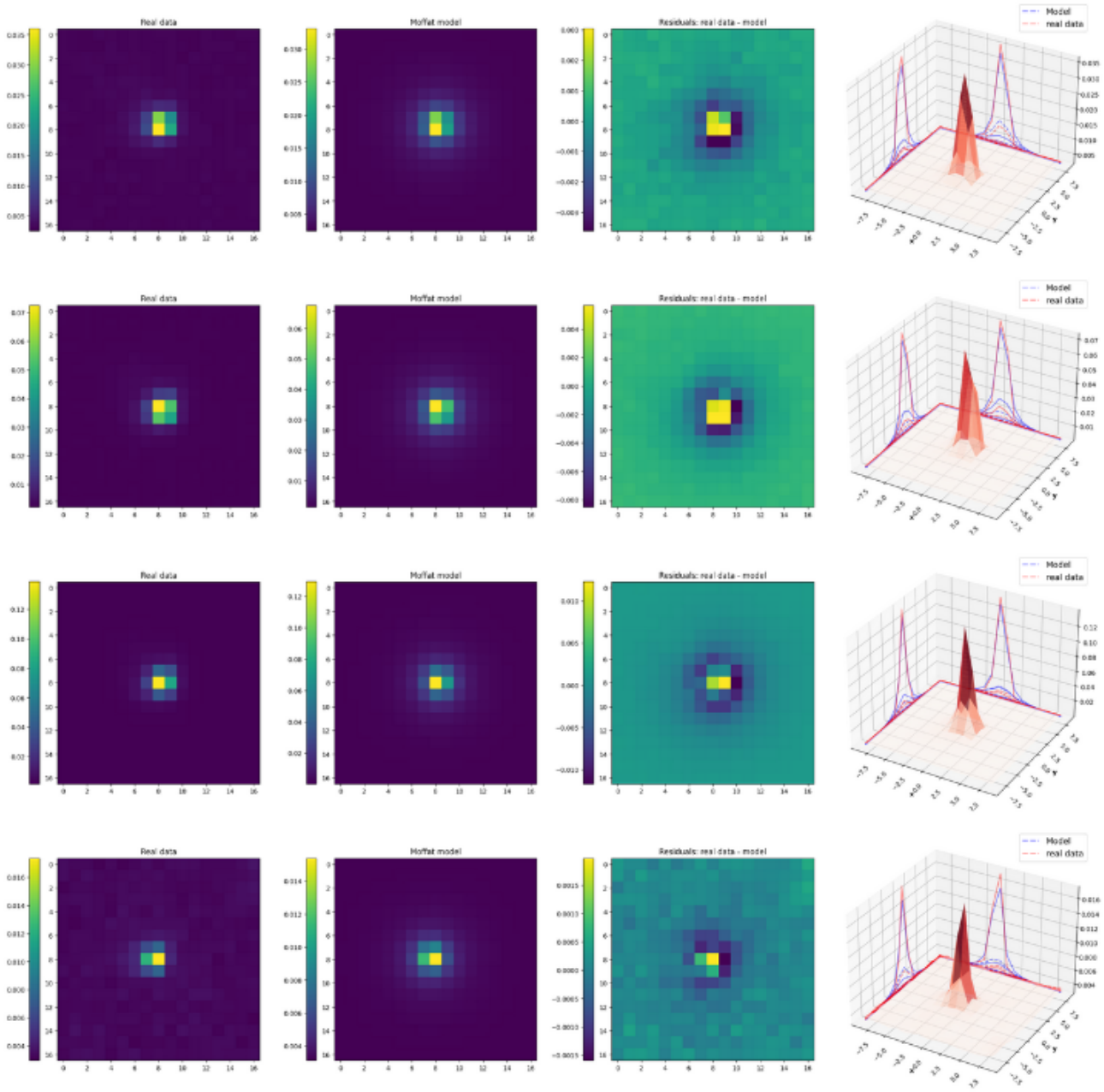
Plots showing 2D distribution of real data, fitted Moffat model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the CPU framework with the *optimize.minimize* optimizer from Scipy.

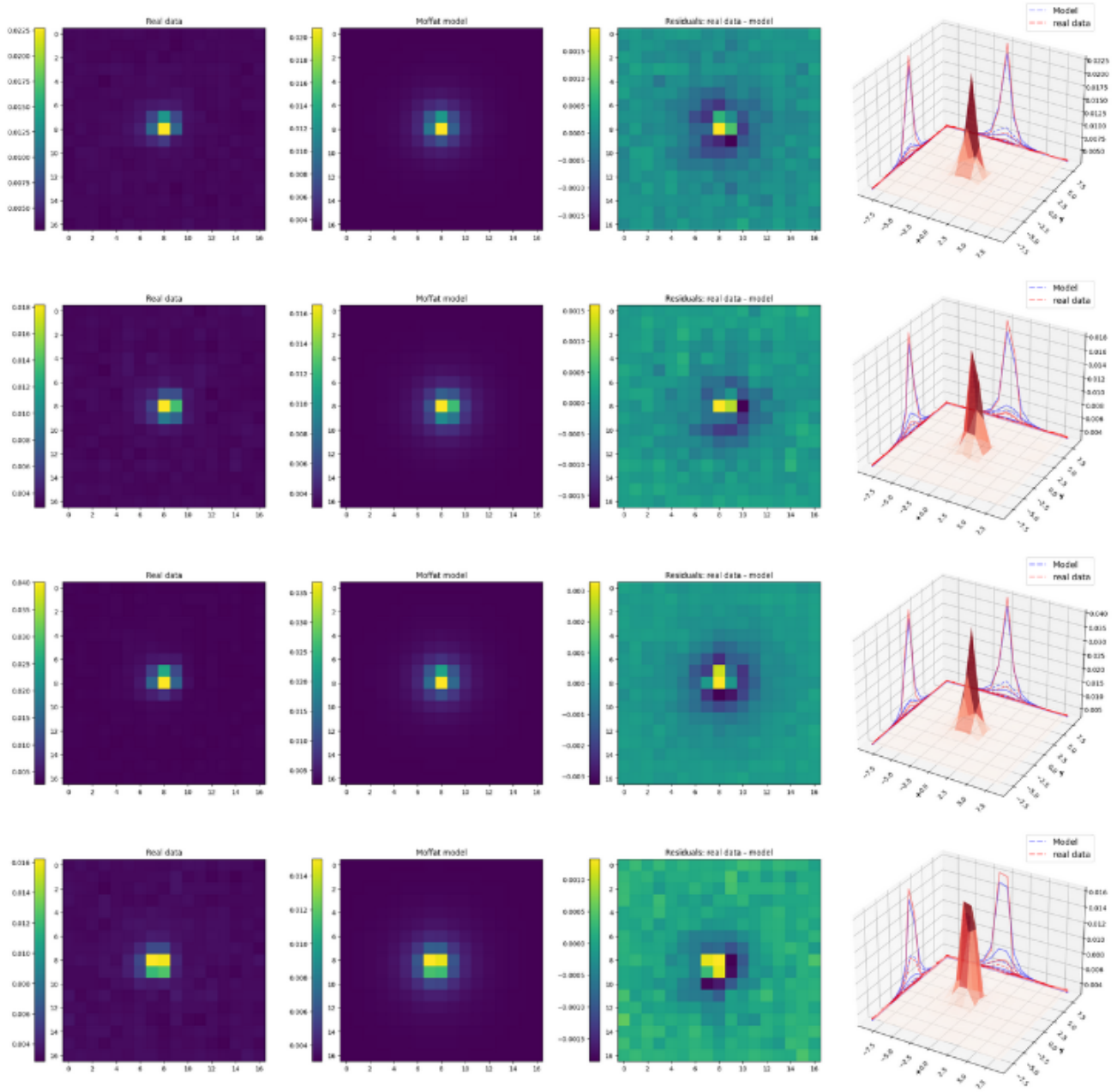



real data
Moffat model
Residuals: real data - model
Model
Real data
x
y

Appendix F: Plots 2D and 3D for Moffat model with GPU framework and Adam optimizer

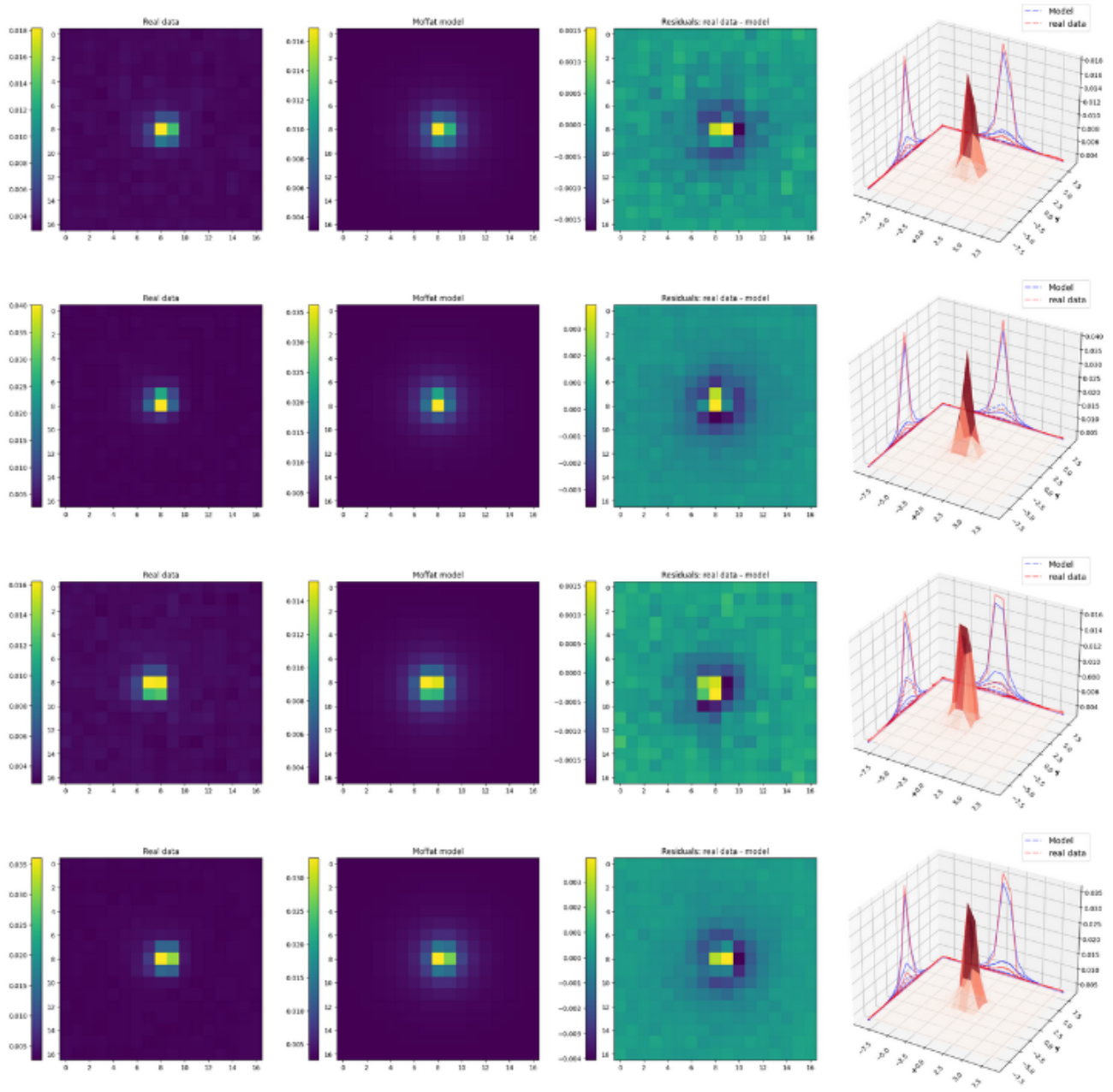
Plots showing 2D distribution of real data, fitted Moffat model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the GPU framework with the *optax.adam* method.

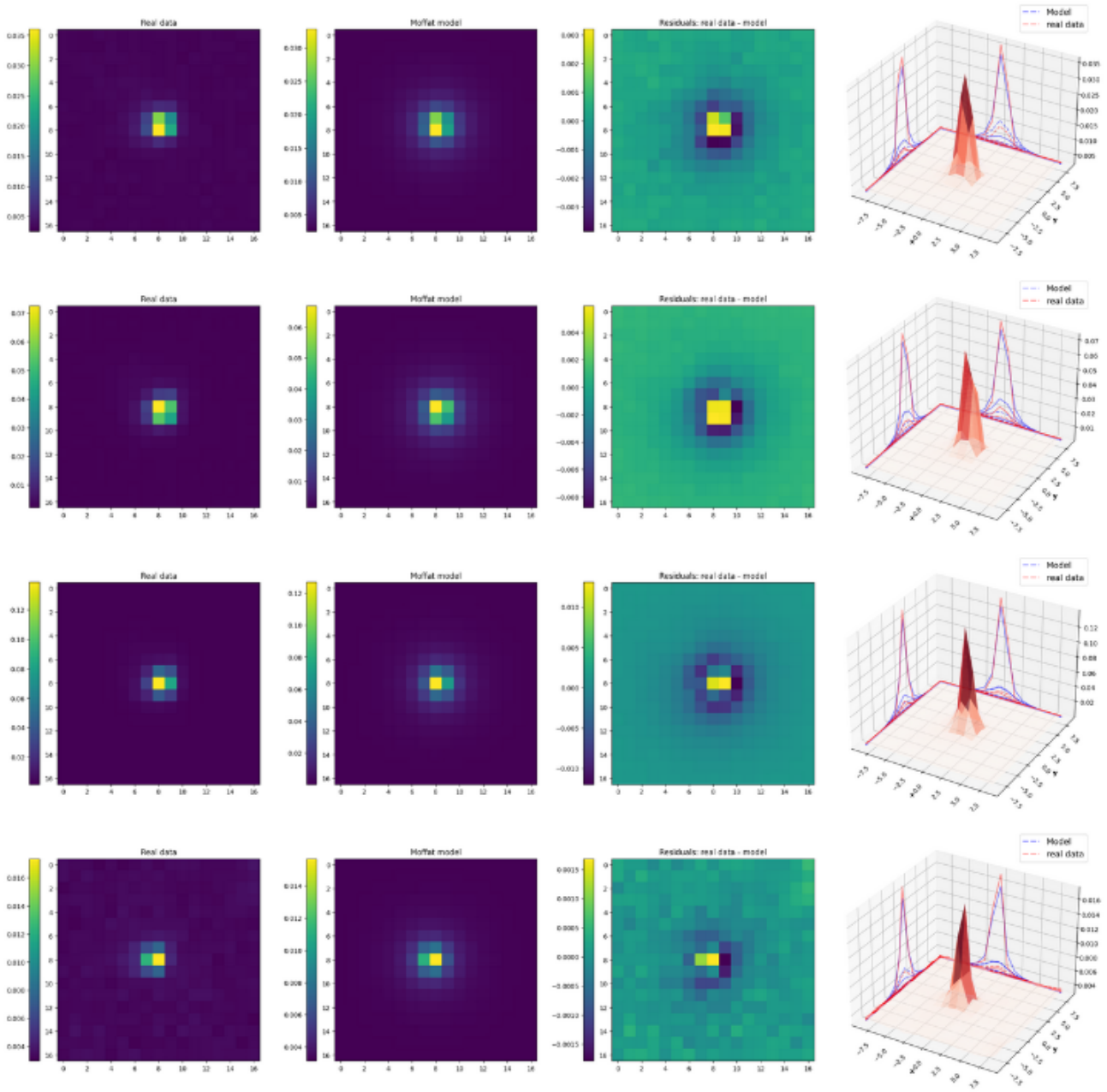



Real data
Moffat model
Residuals: real data - model
Model
real data

Appendix G: Plots 2D and 3D for Moffat model with GPU framework and TN-CG optimizer

Plots showing 2D distribution of real data, fitted Moffat model and residuals for different stars. As well as a figure showing the 3D distribution of the real data (in red) and the fitted Moffat model (in blue). Each plot is made on the GPU framework with the TN-CG method.




Real data
Moffat model
Residuals: real data - model
Model
real data