Monthly Notices
of the
ROYAL ASTRONOMICAL SOCIETY

MNRAS **504,** 3561–3579 (2021)
Advance Access publication 2021 April 21

doi:10.1093/mnras/stab1114

PyTorchDIA: a flexible, GPU-accelerated numerical approach to Difference Image Analysis

James A. Hitchcock [1]★ Markus Hundertmark,[2] Daniel Foreman-Mackey,[3] Etienne Bachelet,[4] Martin Dominik [1] Rachel Street[4] and Yiannis Tsapras[2]

[1]*SUPA, School of Physics and Astronomy, University of St Andrews, North Haugh KY16 9SS, UK*
[2]*Astronomisches Rechen-Institut, Zentrum für Astronomie der Universität Heidelberg (ZAH), D-69120 Heidelberg, Germany*
[3]*Center for Computational Astrophysics, Flatiron Institute, Simons Foundation, 162 5th Ave, New York, NY 10010, USA*
[4]*Las Cumbres Observatory, 6740 Cortona Drive, Suite 102, Goleta, CA 93117, USA*

Accepted 2021 April 14. Received 2021 April 10; in original form 2020 October 26

ABSTRACT

We present a GPU-accelerated numerical approach for fast kernel and differential background solutions. The model image proposed in the Bramich Difference Image Analysis (DIA) algorithm is analogous to a very simple convolutional neural network (CNN), with a single convolutional filter (i.e. the kernel) and an added scalar bias (i.e. the differential background). Here, we do not solve for the discrete pixel array in the classical, analytical linear least-squares sense. Instead, by making use of PyTorch tensors (GPU compatible multidimensional matrices) and associated deep learning tools, we solve for the kernel via an inherently massively parallel optimization. By casting the DIA problem as a GPU-accelerated optimization that utilizes automatic differentiation tools, our algorithm is both flexible to the choice of scalar objective function, and can perform DIA on astronomical data sets at least an order of magnitude faster than its classical analogue. More generally, we demonstrate that tools developed for machine learning can be used to address generic data analysis and modelling problems.

Key words: methods: data analysis – techniques: image processing – software: development.

1 INTRODUCTION

Difference Image Analysis (DIA) describes several astronomical image processing algorithms with the shared goal of delivering precise photometric measurements of variable astronomical sources. Given two images of the same scene acquired at different times, in a DIA framework, one aims to subtract one image from the other and recover a *difference image* from which differential fluxes can be directly measured. This makes DIA a particularly effective approach to measuring the photometric variability of objects in crowded stellar fields – such as microlensing campaigns (e.g. Wozniak 2000; Bond et al. 2001) and studies of globular clusters (e.g. Bramich et al. 2011; Kains et al. 2012; Figuera Jaimes et al. 2013) – where the blending of light from neighbouring sources cannot otherwise be easily disentangled.

Even for two perfectly aligned images from the same instrument, in order to produce a clean subtraction, the DIA algorithm of choice must model the inevitable changes in (1) the point spread function (PSF) due to variations in seeing, telescope focus or tracking errors, (2) the photometric scaling from differences in atmospheric transparency, exposure time or instrument throughput, and (3) the Sky background caused by e.g. changes in the position and phase of the moon.[1]

The differential change in PSF and photometric scaling can be found by inferring the *convolution kernel*, which 'blurs' the sharper of the two images to match the PSF in the other. Alard & Lupton (1998, hereafter AL98) demonstrated that the kernel can be found using linear least-squares by decomposing it as a set of user-specified basis functions. Specifically, AL98 opted for Gaussian basis functions modified by low-order polynomials. A decade later, the linear least-squares solution was advanced by Bramich (2008, hereafter B08) who modelled the kernel as a highly flexible (albeit computationally more expensive) discrete pixel array, which is analogous to the AL98 algorithm, but with a choice of delta basis functions for the kernel model. Later advances included extending the DIA solution to a spatially varying kernel and differential background to model position-dependent variations in seeing, transparency, and airmass across the field of view (FoV) in wide-field imaging data (Alard 2000; Bramich et al. 2013).

In this analytical linear least-squares framework, a normal matrix for the linear system of equations is required. It is the construction of the normal matrix for these approaches that results in a computational

★ E-mail: jah36@st-andrews.ac.uk

[1]We note that there are other differences between images that cannot be captured by current DIA models, such as differential refraction or extinction, and we do not address these in this work.

bottleneck. For example, for a (square) pair of images each of size n, and a (square) kernel of size m the construction of the normal matrix for the B08 approach – which implements the same model for the kernel as our algorithm – scales as $\mathcal{O}(n^2m^4)$ (i.e. the run-time increases with the square of the input image size and with the kernel size to the power of four). These problems are exacerbated by the need to iteratively fit for the model parameters in the linear least-squares sense (as the problem is in fact non-linear, see Section 2.1) and, typically, the normal matrix will need to be constructed about 3–4 times until convergence is reached. Attempts have been made to exploit symmetries in the problem (see section 5.2 of Bramich et al. 2013) or bin pixels around the kernel edges to speed up this construction (Albrow et al. 2009). In the era of wide FoV sky surveys such as the upcoming Legacy Survey of Space and Time (LSST; Ivezić et al. 2019), or the ongoing Zwicky Transient Facility (ZTF; Bellm et al. 2018), etc., which aim to deliver prompt alerts of transient astronomical phenomena detected through image subtraction, many current popular approaches make real-time event discovery impractical. This has driven some recent advances in the DIA literature, most notably the ZOGY algorithm (Zackay, Ofek & Gal-Yam 2016), and even a machine-learning approach (Sedaghat & Mahabal 2018).

Image processing tasks – where some common operation is performed on very many pixels – are inherently massively parallel. Mild to substantial computational speed-up by parallelizing the construction of the normal matrix in classical DIA algorithms with GPUs has been demonstrated in the literature (e.g. Hartung et al. 2012; Li et al. 2013; Zhao et al. 2013). Adoption by the larger astronomical community, however, has been slow.

The main barrier to adoption for most astronomers is likely the lack of a working knowledge of CUDA, NVIDIA's parallel computing platform, which is required to develop GPU-accelerated applications on supported devices. In astronomy, the Python programming language has firmly established itself as a favourite of the majority of the community, and most importantly, will be the primary user-interface language for the next generation of astronomical data management, processing and analysis systems (Perkel 2018). The appropriateness of DIA image models can be highly data set dependent, and tuning by individual researchers to meet their science goals within a Pythonic framework is to be expected. The work presented here is one of a few recent attempts[2] to address the paired issues of performance and accessibility.

In this paper, we present a novel Pythonic implementation of an alternative route to the B08 solution, without the need for constructing the computationally expensive normal matrix. Conceptually, our approach is unique in that we model the kernel as if it were the convolutional filter of a very simple convolutional neural network (CNN), which can be solved for efficiently with GPU-accelerated optimization tools originally developed for deep learning applications. As we will describe, the machine-learning framework which we use to approach this problem also equips us with powerful modelling tools, and frees us from a number of restrictive assumptions inherent to classical approaches. Our implementation is also unique among GPU-accelerated DIA algorithms for being written *entirely* with standard Python packages.

We begin Section 2 with an introduction of the basic image model used in our DIA implementation. We then outline how PyTorch could be well suited to addressing existing astronomical image modelling and data processing challenges in general, before describing the details of our own DIA implementation, PyTorchDIA. We quantify the model fit quality and photometric accuracy of PyTorchDIA with tests on both synthetic and real images in Sections 3 and 4, comparing it directly to the performance of its classical DIA analogue, the B08 algorithm. In Section 5, we compare the speed of our GPU-accelerated numerical solution against a fast Cython implementation of the B08 algorithm[3] used in the ROME/REA project (Tsapras et al. 2019), and explore how our algorithm scales with image and kernel size. We summarize our conclusions in Section 6.

2 PROBLEM FORMULATION

In this section, we outline the DIA problem, and provide a motivating overview for using PyTorch as a tool to address image modelling challenges – of which DIA is just one example – and describe the details of our DIA implementation.

2.1 Difference Image Analysis

Given a reference image with pixels R_{ij}, ideally of excellent spatial resolution and high signal-to-noise, and a target image with pixels I_{ij}, of the same scene taken at some other epoch and aligned on the same pixel grid, the model image which represents I_{ij} is given by

$$M_{ij} = [R \otimes K]_{ij} + B_{ij} \ , \tag{1}$$

and so the challenge is to find the fit for an accurate kernel, K, and differential background B_{ij}.

Following B08, by representing the kernel as a discrete array K_{lm} containing a total of N_K pixels, and including an additive scalar differential background, B_0, we can rewrite equation (1) as

$$M_{ij} = \sum_{lm} K_{lm} R_{(i+l)(j+m)} + B_0 \ . \tag{2}$$

The photometric scale factor – which encodes any differences in atmospheric transparency and/or exposure time between the images – is simply the sum of the kernel pixels,

$$P = \sum_{lm} K_{lm} \ . \tag{3}$$

Assuming that the pixel values of the target image, I_{ij}, are independently drawn from normal distributions $\mathcal{N}(M_{ij}, \sigma_{ij}^2)$ – where M_{ij} is parametrized by the vector $\boldsymbol{\theta} = [K_{lm}, B_0]$ (see Section 2.3) – the negative log-likelihood function for the target image takes the form

$$-\ln \mathrm{p}(I_{ij}|\boldsymbol{\theta}) = \frac{1}{2}\chi^2 + \sum_{ij} \ln \sigma_{ij} + \frac{N_{\mathrm{data}}}{2}\ln(2\pi) \ , \tag{4}$$

where σ_{ij} are the pixel uncertainties, N_{data} is the number of pixels in the target image, and the χ^2 is equal to

$$\chi^2 = \sum_{ij} \left(\frac{I_{ij} - M_{ij}}{\sigma_{ij}} \right)^2 \ . \tag{5}$$

It is important to note that N_{data} is *not* equal to the total number of pixels in the target image, as the convolution operation is undefined for pixels within half a kernel's width from the target image edges i.e. for a kernel of size $(2n + 1) \times (2n + 1)$ and (square) reference and target images of size $N \times N$, $N_{\mathrm{data}} = (N - 2n)^2$.

[2]Including approaches attempting to parallise the construction of the B08 normal matrix (Albrow 2017)

[3]The kernel solution method heavily borrows from the relevant section of the pyDANDIA microlensing reduction pipeline, https://github.com/pyDANDIA.

The σ_{ij} pixel uncertainties are dependent on the image model M_{ij}, and so fitting this model to the target image I_{ij} is therefore a non-linear optimization task. Linear least-squares approaches like AL98 and B08 must then approach this iteratively, by minimizing the χ^2 with fixed estimates for σ_{ij}. After the first estimate of M_{ij} is acquired, the σ_{ij} are computed, and then plugged into the χ^2 for the next iteration. This process continues for at least three iterations, until some convergence condition is met.

In this work, we use both simulated CCD images and real Electron Multiplying CCD (EMCCD) images, acquired on the Danish 1.54-m (DK154) Lucky Imaging camera (Skottfelt et al. 2015) to verify our implementation, each of which requires a different noise model. In what follows, we ignore the noise contributions from the reference image, since these are negligible in the experiments performed in this work.[4]

For CCD images, we adopt a noise model for the σ_{ij} pixel uncertainties of M_{ij} as

$$\sigma_{ij}^2 = \frac{\sigma_0^2}{F_{ij}^2} + \frac{M_{ij}}{G\ F_{ij}}, \tag{6}$$

where σ_0 is the read noise (ADU), G is the detector gain (e^-/ADU), and F_{ij} is the master flat-field.

The EMCCD images are constructed from typically thousands of shift-and-added sub-second exposures. The EM gain reduces the read out noise to negligible levels, but the cascade amplification process effectively doubles the variance of the photon noise. For the real EMCCD images used in this work, we adopt a noise model of the form

$$\sigma_{ij}^2 = E\frac{M_{ij}}{G_{\rm Total}\ F_{ij}}, \tag{7}$$

where $G_{\rm Total}$ represents the combined gain ($e^-_{\rm phot}$/ADU) of the DK154 Lucky Imaging camera, which is calculated as the ratio of the CCD gain of 25.8 ($e^-_{\rm EM}$/ADU) over the electron-multiplying (EM) gain of 300 ($e^-_{\rm EM}/e^-_{\rm phot}$). Following Harpsøe et al. (2012), we differentiate between electrons before and after the cascade amplification with the notation $e^-_{\rm phot}$ and $e^-_{\rm EM}$, respectively. E represents the 'excess noise factor', and accounts for the probabilistic nature of the cascade amplification as the EMCCD is read out, and is set to be $E = 2$. EMCCD images are flat corrected in the same way as conventional CCD images, and as in equation (6), F_{ij} is the master flat-field.

2.2 Astronomical image processing with PyTorch

The model image of equation (1) is a convolution with some added scalar constant. In the computer science literature, this is *exactly* analogous to an extremely simple CNN, with a single input and output related by a single convolutional filter, with some additional scalar bias added to the output. Efficient solutions for the 'weights' (the kernel) and 'bias' (background term) to this convolution operation can be implemented within the Python package, PyTorch (Paszke et al. 2019).

PyTorch is a popular open source machine learning framework. Its constant development is motivated by the impressive advances in deep learning (for an overview, see e.g. LeCun, Bengio & Hinton 2015; Goodfellow et al. 2016), in which CNNs have played an important role in processing images. Specifically, this package supports CUDA-enabled GPU acceleration and automatic differentiation to perform efficient optimizations. One key motivation behind these developments is the *training* of complex CNNs, consisting of thousands, to hundreds of thousands of parameters. The 2D convolution (as in equation 1) is *the* core processing operation in these networks, and is straightforward to implement in PyTorch's modelling architecture. Indeed, many useful image models in astronomy can be written as a convolution, and the tools outlined in this work are therefore broadly applicable to many astronomical image processing problems.

We stress that although PyTorch's powerful tools were designed for machine learning, they can be turned to generic data analysis and modelling problems, as shown in this work, of which image models are just a subset. In particular, efficient computation of gradients via automatic differentiation frees the user from having to manually recompute gradients if the parametrization of the model is changed. This flexibility allows models written in PyTorch to be easily tuned to meet the variety of science goals arising from a diversity of data sets. In astronomical image processing in particular, this flexibility combined with GPU acceleration could make PyTorch a valuable tool for addressing challenges associated with both model complexity and data volume. As PyTorch is Pythonic, these implementations could be easily integrated into existing Python stacks.

2.3 Difference Image Analysis as an optimization

The crucial advance in AL98 was to formulate DIA as a linear least-squares problem, and several efficient algorithms for solving these problems exist (Golub & Van Loan 1996). The computational bottleneck associated with this approach is the construction of the normal matrix (see Section 1). Models in PyTorch, however, are generally fit with a numerical optimization procedure – making use of automatic differentiation – which we outline here.

For a given target image with Gaussian noise contributions, I_{ij}, and current estimates for the kernel K_{lm} and differential background B_0 which transform the reference image, R_{ij}, we define our vector of weights as $\boldsymbol{\theta} = [K_{lm}, B_0]$. Ignoring the irrelevant normalization constant, the maximum-likelihood-estimate (MLE) for $\boldsymbol{\theta}$ can be found by minimizing the overall *loss* (or negative log-likelihood cf. equation 4),

$$\begin{aligned}&\underset{\boldsymbol{\theta}=[K_{lm},B_0]}{\arg\min}\ \mathcal{L}_0(\boldsymbol{\theta})\\ &= \underset{\boldsymbol{\theta}}{\arg\min}\ \left[\frac{1}{2}\sum_{ij}\left(\frac{I_{ij}-M_{ij}(\boldsymbol{\theta})}{\sigma_{ij}(\boldsymbol{\theta})}\right)^2 + \sum_{ij}\ln\ \sigma_{ij}(\boldsymbol{\theta})\right].\end{aligned} \tag{8}$$

We note here that both the image model and noise model are both functions of $\boldsymbol{\theta}$. We drop the notation explicitly showing the dependence of M_{ij} and σ_{ij} on $\boldsymbol{\theta}$ from the rest of the manuscript.

Equation (8) can be solved by a process of (steepest) gradient descent. At each iteration t in the optimization, we use the update rule

$$\boldsymbol{\theta}^{(t+1)} \longleftarrow \boldsymbol{\theta}^{(t)} - \alpha^{(t)}\nabla_{\boldsymbol{\theta}^{(t)}}\mathcal{L}_0(\boldsymbol{\theta}^{(t)})\ , \tag{9}$$

where α^t is the *learning rate* (or step-size) at each iteration. PyTorch includes implementations of several more sophisticated gradient descent algorithms, which can compute *adaptive* learning rates for *each* parameter (see Ruder 2016 for a good overview). In our implementation, we use the Adaptive Moment Estimation (Adam) algorithm, which computes learning rates for the kernel pixels and the background parameters from estimates of the first and second

[4]Our interest in testing the performance of our DIA algorithm on DK154 EMCCD images relates to our ongoing microlensing follow-up campaign, which is the main science project of the MiNDSTEp consortium, http://www.mindstep-science.org/.

moments of their gradients at each step (Kingma & Ba 2014). Adam has been empirically demonstrated to work well on non-convex optimization problems, is computationally efficient, and the hyper-parameters are both intuitive and require little tuning from the astronomer (see Section 2.9 for an overview of the 'engineering' aspects of the optimization).

With an excellent choice of learning rate, solutions via steepest descent (SD) when accelerated on the GPU are lightning fast (see Section 5), but this approach in general can be slow. Solving this problem on the CPU – and therefore forgoing the massive inherent parallelism otherwise exploited in equation (9) – would result in a substantial performance hit. This problem is particularly severe when close to the minimum of the loss function, where the gradients become increasingly shallow and the gradient steps become correspondingly smaller. An effective solution to this problem is to make use of quasi-Newton optimization methods that approximate the curvature of the loss surface.

These approaches converge extremely quickly where the loss surface can be modelled quadratically, and use an approximation of the inverse of the Hessian at each t iteration, $B(\boldsymbol{\theta})^{(t)}$, to condition the search direction, giving the update rule

$$\boldsymbol{\theta}^{(t+1)} \longleftarrow \boldsymbol{\theta}^{(t)} - \alpha^{(t)} B^{-1}(\boldsymbol{\theta})^{(t)} \nabla_{\boldsymbol{\theta}^{(t)}} \mathcal{L}_0(\boldsymbol{\theta}^t) \, . \tag{10}$$

For this convex optimization, this is a very good assumption when close to the minimum. Newton methods in general are sensitive to bad parameter initialization (i.e. when we're initially far from the minimum), and so we advocate for optimizing via SD steps – as in equation (9) – to first get close to the minimum, before converging with the more memory intensive quasi-Newton procedure once the relative change in the loss between any two optimization steps falls below a user-specified threshold. Specifically, we use a L-BFGS method (see chapter 7.2 of Nocedal & Wright 2006) which is available as an in-built algorithm in PyTorch.[5]

2.4 Robust loss function

In general, the assumption that all the pixel values in the target image are drawn from $\mathcal{N}(M_{ij}, \sigma_{ij}^2)$ will be violated for real images. Instrumental defects, cosmic ray hits and variable or transient sources, etc. will all arise as outlying pixel values. Our Gaussian loss function (equation 4) will be badly affected by these outliers, as it minimizes the squared difference between the model and the data. The standard approach to outlier rejection in astronomy is to iteratively remove these 'bad' pixels by *sigma-clipping*, and B08 uses this approach to mitigate the impact of outliers to the least-squares solution. In addition to being very sensitive to the accuracy of the adopted noise model, this *procedure* does not explicitly penalize the rejection of data, nor is it necessarily clear how many iterations should be performed. In the context of classical DIA algorithms, these iterations incur an extra computational expense, as a new normal matrix must be constructed each time.

As we are not restricted to standard least-squares minimization with our optimization procedure, we have the freedom to choose *robust* alternatives to equation (8) which are less sensitive to outlying values. Huber (1992) identified a family of wide-tailed univariate distributions. The 'ϵ-contaminated' Gaussian model in particular corresponds to a unimodal symmetric distribution that resembles a Gaussian for central values and a Laplacian in the tails. The

[5]https://pytorch.org/docs/stable/optim.html

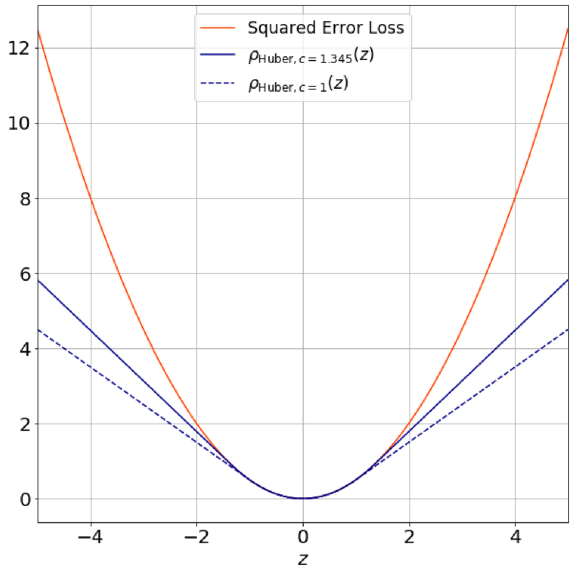

Figure 1. A comparison of squared error loss against Huber loss for different values of c.

probability density function (up to a normalizing constant) with location and scale parameters μ and σ has the form

$$f(x) \propto \frac{1}{\sigma} \exp\left[-\rho_{\mathrm{Huber},c}\left(\frac{x-\mu}{\sigma}\right)\right] \, , \tag{11}$$

Substituting $z = (x-\mu)/\sigma$, $\rho_{\mathrm{Huber},c}(z)$ is the loss function, which is defined as

$$\rho_{\mathrm{Huber},c}(z) = \begin{cases} \frac{1}{2}z^2, & \text{for } |z| \leq c \\ c(|z| - \frac{1}{2}c), & \text{for } |z| > c \, . \end{cases} \tag{12}$$

We see that $\rho_{\mathrm{Huber},c}(z)$ computes the squared error between the model and data for small residuals, and the absolute deviation for outliers above some user-specified threshold, c. This threshold defines the switch between quadratic and linear treatment of the error (see Fig. 1), and Huber & Ronchetti (2009) recommend $c = 1.345$ as a suitable value for enforcing robustness while retaining reasonable efficiency for normally distributed data.

Using the same notation as in Section 2.1, if we assume the pixel values in the target image I_{ij} are drawn from the distribution in equation (11), the negative log-likelihood function takes the general form

$$-\ln p(I_{ij}|\boldsymbol{\theta}) = \sum_{ij} \rho_{\mathrm{Huber},c}\left(\frac{I_{ij} - M_{ij}}{\sigma_{ij}}\right) + \sum_{ij} \ln \sigma_{ij} + Q \, , \tag{13}$$

where Q is the normalizing constant,

$$Q = N \ln\left(\sqrt{2\pi}\,\mathrm{erf}\left(\frac{c}{\sqrt{2}}\right) + \frac{2\exp\left[\frac{-c^2}{2}\right]}{c}\right) \, . \tag{14}$$

Note that when c becomes large, $Q = (N/2) \times \ln 2\pi$, which is equivalent to the normalization constant of a Gaussian log-likelihood (cf. equation 4).

As our implementation makes use of automatic differentiation, it is straightforward for the user to experiment with loss functions, as they are freed from having to manually recompute gradients. We highlight the Huber loss in particular as this enforces robustness without sacrificing performance (see Section 5.1), but any number of wide-tailed distributions (e.g. Student's t distribution) could be used as drop-in replacements.

2.5 Uncertainty estimation – Observed Fisher Information

The central limit theorem states that any well-behaved likelihood function approaches a Gaussian near its maximum. The *Fisher Information Matrix* (FIM) is a measure of the curvature of the likelihood function with respect to the model parameters – intuitively, this can be thought of as a 'sensitivity' – and its inverse provides a lower bound on the asymptotic variance of the MLE.

The observed FIM, $\boldsymbol{F}(\boldsymbol{\theta})$ for our $N_K + 1$ parameters is the $(N_K + 1, N_K + 1)$ matrix containing the entries

$$F_{pq} = \frac{\partial^2}{\partial\theta_p \partial\theta_q} \ln \mathrm{p}(I_{ij}|\boldsymbol{\theta}) \ , \ 1 \leq p, \ q \leq N_K + 1 \ , \tag{15}$$

where $\ln \mathrm{p}(I_{ij}|\boldsymbol{\theta})$ is the log-likelihood (equation 4).

The inverse of $\boldsymbol{F}(\boldsymbol{\theta})$ evaluated at the MLE for $\boldsymbol{\theta}$ (i.e. $\hat{\boldsymbol{\theta}}_{\mathrm{MLE}}$) can then be used as an estimate for the covariance matrix

$$\hat{\boldsymbol{\Sigma}} = \mathrm{Cov}(\hat{\boldsymbol{\theta}}_{\mathrm{MLE}}) = [\boldsymbol{F}(\hat{\boldsymbol{\theta}}_{\mathrm{MLE}})]^{-1} \ , \tag{16}$$

which provides an estimate of the uncertainties on the model parameters by taking the square roots of the diagonal elements of $\mathrm{Cov}(\hat{\boldsymbol{\theta}}_{\mathrm{MLE}})$. Fisher information provides us with the limiting precision with which model parameters can be estimated for any given data set i.e. the subsequent error bars cannot be smaller (Heavens 2009). Formally, the *Cramér–Rao* inequality states that the uncertainty on some parameter θ_p is given by

$$\Delta\theta_p \geq (F^{-1})^{1/2}_{pp} \ . \tag{17}$$

This method is subject to two assumptions (1) the likelihood function correctly describes the data generating process (i.e. the error distribution of the measurements is correctly described by the likelihood), and (2) the likelihood really is approximately Gaussian at the MLE. In practice, both of these assumptions will likely be violated, and in these situations, the uncertainty estimates by this approach can be severely underestimated. Given this, Andrae (2010) strongly recommend to test this assumption by checking the validity of $\hat{\boldsymbol{\Sigma}}$. In general, a valid covariance matrix $\hat{\boldsymbol{\Sigma}}$ must be positive definite (i.e. for any non-zero vector $\boldsymbol{x}, \boldsymbol{x}^T.\hat{\boldsymbol{\Sigma}}.\boldsymbol{x} > 0$). If either the determinant of $\hat{\boldsymbol{\Sigma}}$ is negative, or (after diagonalizing the matrix) any eigenvalue is found to be negative or zero, then $\hat{\boldsymbol{\Sigma}}$ is not valid. We include both these tests in our code release, and a warning flag is raised if any is failed.

2.6 Extension to a spatially varying background

Within the PyTorch architecture, a spatially varying background can easily modelled by replacing B_{ij} in equations (1) and (8) with a linear combination of functions of x and y. We adopt a polynomial model of some user-specified degree d,

$$B(x, y) = \sum_{m=0}^{d} \sum_{n=0}^{d-m} b_{mn} \, \eta(x)^m \xi(y)^n, \tag{18}$$

where b_{mn} are the polynomial coefficients to be inferred, and $\eta(x)$ and $\xi(y)$ are the normalized spatial coordinates,

$$\begin{aligned} \eta(x) &= (x - x_c)/N_x \ , \\ \xi(y) &= (y - y_c)/N_y \ , \end{aligned} \tag{19}$$

which result from a Taylor expansion of coordinates (x, y) about the image centre (x_c, y_c), for an image of $N_x \times N_y$ pixels. This choice of coordinates is recommended by Bramich et al. (2013), and has the effect of (1) improving the orthogonality of the spatial polynomial terms, and (2) preventing the coefficients of the higher order polynomial terms from being pushed towards zero.

2.7 Regularizing the kernel pixels

As noted by Becker et al. (2012), while kernels modelled as a discrete array of pixels are very flexible, the consequent fidelity with the data can result in significant overfitting. To guard against excessively noisy kernels, we provide the option to regularize the loss with the addition of a penalty term. Following the notation in Bramich et al. (2016, hereafter B16) – where the strength of the regularization is controlled by the parameter λ, which must be tuned empirically – the scalar objective function to minimize now takes the form,

$$\arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \left[\mathcal{L}_0(\boldsymbol{\theta}) + \lambda N_{\mathrm{data}} \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{L}^{\mathrm{T}} \boldsymbol{L} \boldsymbol{\theta}\right] \ , \tag{20}$$

where N_{data} is the number of target image pixels, $\boldsymbol{\theta}$ is the vector of parameters to optimize, $\mathcal{L}_0(\boldsymbol{\theta})$ is the loss function, and $\boldsymbol{L}$ is the symmetric and positive-semidefinite $(N_K + 1) \times (N_K + 1)$ *Laplacian matrix*, which represents the connectivity graph of the set of kernel pixels, and has elements

$$L_{uv} = \begin{cases} N_{\mathrm{adj},u}, & \text{for } v = u \leq N_K, \text{ and } N_{\mathrm{adj},u} \text{ is the number of kernel pixels adjacent to the kernel pixel corresponding to } u \\ -1, & \text{for } u \leq N_K, v \leq N_K, v \neq u, \text{ and } u \text{ and } v \text{ are adjacent kernel pixels.} \\ 0, & \text{otherwise.} \end{cases} \tag{21}$$

This penalty term is derived from an approximation of the second derivative of the kernel surface (Becker et al. 2012). Intuitively, it favours compact kernels, where adjacent kernel pixels should not vary too sharply. The optimal value of λ should, ideally, be tuned for each image, and B16 found optimal values could be anywhere in the range $\lambda = 0.1$–100.

By regularizing the kernel weights, we are in effect introducing a Bayesian prior, which would then transform our solution into a maximum a posteriori (MAP) optimization. In our experiments in the following sections, we restrict ourselves to the MLE solution by simply minimizing either equation (8) or (13).

2.8 General purpose computing on GPUs

For this work, we ran our PyTorchDIA implementation on two different NVIDIA GPUs. The computations associated with the results presented in Sections 3 and 4 were performed on a GeForce GTX 1050 on a local machine, and the speed tests in Section 5 were run on a Tesla K80 on Google's Colab[6] service, a free-to-use cloud-based computation environment. There are a couple of important details worth describing on using these devices for scientific computing.

First, while modern CPUs are able to handle computations on 64-bit floating point numbers efficiently, such operations are either not supported on GPUs, or are associated with a significant reduction in performance (Göddeke, Strzodka & Turek 2005). It is for this reason that 32-bit precision – or increasingly commonly, 16-bit precision – is used in deep learning. For this problem, we have found computations typically take at least 2–5 times longer for fiducial image sizes using 64-bit precision with a Tesla K80.[7] For memory

[6] https://colab.research.google.com

[7] Tesla cards are designed to perform fast computations at higher precision than most GPU models, and a slowdown of ∼2–5 at 64-bit would still outperform some state-of-the-art classical DIA methods (see Section 5). Most NVIDIA models, however, (including GeForce cards), excel at 32-bit and lower precisions, but would suffer a more severe slowdown at 64-bit.

intensive optimizers such as L-BFGS, the hit to performance will be even worse. For this reason, for the results in this work we use 32-bit floating point precision for our PyTorchDIA implementation. This speed-up comes at the expense of some accuracy, but we show this difference to be small in Sections 3 and 4.

Convolution computations on NVIDIA GPUs can be accelerated by highly tuned algorithms available in specialized libraries. One such NVIDIA library, cuDNN, specializes in efficient computations with small kernels (i.e. those in the regime of DIA). These algorithms are selected by cuDNN heuristically, and not all are deterministic. As such, for an identical pair of images, the number of computations used to infer the convolution kernel may be different on different runs. Consequently, the optimizer may converge at a slightly different point in the parameter space. We explore the speed-up associated with enabling cuDNN tools in Section 5.

Finally, while not explored in this work, we highlight *mixed precision* 'training' as a technique which our approach could benefit from. PyTorch now supports automatic mixed precision (AMP) training, in which 32-bit data can be automatically cast to half precision for some types of computationally expensive operations on the GPU.[8] By appropriately scaling the loss during the training/optimization, AMP prevents *underflow* that would otherwise cause gradients to drop to 0 at half precision, and can achieve the same accuracy as training only with 32-bit precision with significant speed-up, depending on the GPU architecture and model design (Micikevicius et al. 2017). Some recent NVIDIA GPUs now include *Tensor cores*, which are specifically designed to perform highly optimized 16-bit matrix multiplications. Consequently, cuDNN convolution algorithms such as 'GEMM' are particularly well suited to benefit from this technique (Jordà, Valero-Lara & Peña 2019).

2.9 Optimization as an engineering problem

As we fit for the convolution kernel via an optimization, hardware alone will not determine the solution time. There is an 'engineering' aspect to optimization that the user should be aware of. Here, we summarize three key choices that the user must make: (1) Parameter initialization; (2) the learning rate (or step size), and (3) the convergence condition. We also provide an overview of (and justification for) the specific choices made when generating the results presented in this manuscript.

Throughout this work, we *always* background subtract the reference image, and so we initially set the differential background parameter, B_0, to be equal to the median pixel value of the given target image. For the fit quality and photometric accuracy tests in Sections 3 and 4, we make no assumptions about the shape of the kernel at initialization. We set all kernel pixels to have the same value, with the only condition being that they sum to 1 (i.e. each pixel is initialized as $1/N_K$). In Section 5, we experiment with initializing the kernel as a symmetric Gaussian, with a shape parameter judiciously set with knowledge of either the known or measured differences in the PSFs of the reference and target image pair.

The Adam algorithm used at the start of the optimization – with which we perform steepest gradient descent – allows us the freedom to set individual learning rates for our model parameters, which will be adaptively tuned (see Section 2.3). For all tests in this work, we set the learning rate of the Adam optimizer for the pixels in the kernel at 0.001, as recommended in Kingma & Ba (2014). We have found that it is advantageous to use a fairly high learning rate for B_0, and we set this to either 1, 10 or even 100, dependent on the data set. There is a strong anticorrelation between P and B_0, and quickly finding the approximate photometric scaling between the two images allows us to disentangle this offset from the inference of the spatial differences in the images (associated with the different PSFs), which is encoded in the shape (and not the 'scale') of the convolution kernel. The learning rate of the L-BFGS optimiser is always set to 1.

We use the same convergence condition for all tests in this work. This decision was made on the basis of the model performance and photometric accuracy metrics in Sections 3 and 4, and should balance a satisfactory model accuracy with the time taken to converge. If between any two SD updates the relative change in the loss is less than 10^{-6} (i.e. we are getting close to the minimum) a L-BFGS optimizer takes over. For each subsequent quasi-Newton update, the L-BFGS function evaluation routine terminates when either the change in loss between evaluations reaches the limit of our numerical precision – which corresponds to a relative change in loss in the range 10^{-8}–10^{-7} between the last two optimizations steps – or a first-order optimality condition is met, such that the gradient of the scalar objective function to be minimized with respect to each model parameter is less than 10^{-7}.

2.10 The algorithm

Our DIA algorithm is as follows. Decisions to be made by the user/human are in italics. 'tol' is the user-specified relative change in the loss between successive SD updates, at which point the optimization routine switches from SD to L-BFGS.

(i) *Choose the square kernel dimensions (must be odd), and whether to use a scalar, or polynomial fit of degree d, for the differential background*
(ii) *Set Adam's per-parameter learning rates and 'tol'*
(iii) *Set the convergence condition*
(iv) *Initialise* $\boldsymbol{\theta}$
(v) Begin minimizing the chosen scalar objective function [equations 4 or 13, with or without the optional penalty term (equation 20)] with steepest gradient descent (equation 9)
(vi) At each iteration **if** the relative change in the loss is less than 'tol', switch to a quasi-Newton update (equation 10) and **continue** until convergence condition satisfied.

3 SIMULATED IMAGE TESTS

Artificially generated images can be used to assess the accuracy of our algorithm. In this section, we compare the fit quality and photometric accuracy of our numerical, GPU-accelerated algorithm against an implementation of the analytical B08 algorithm with a data set of 71 569 synthetic images.

3.1 Generating Artifical Images

We base our image generation procedure on Section 5.1 of B16. The results from the simulated image tests in that work were shown to closely agree with similar tests using real CCD data. Specifically, we generate images similar to those in the 'S10' set of that work, wherein the reference image of each image pair has 10 times less variance than the target.

We first generate a noiseless reference, $R_{\mathrm{noiseless},\,ij}$, of size 142×142 pixels as follows.

[8] https://pytorch.org/docs/stable/notes/amp_examples.html

(i) We generate a normalized 2D symmetric Gaussian PSF for this image, parametrized by a standard deviation, ϕ_R, drawn from the continuous uniform distribution $U \sim (0.5, 2.5)$.

(ii) We populate the 142 × 142 pixel array template of $R_{\text{noiseless},\,ij}$ with N_{stars}, whose lg density lg ρ_{stars} per 100 × 100 pixel region, is drawn from the uniform distribution $U \sim (0, 3)$. The fractional pixel coordinates for each star are uniformly drawn between the image axis lengths.

(iii) For each of the N_{stars}, we draw a value of $\mathcal{F}^{-3/2}$ from the uniform distribution $U \sim (10^{-9}, 10^{-9/2})$, where $\mathcal{F}$ is a given star's flux (ADU). This flux distribution is a good approximation when imaging to a fixed depth for certain regions in the Galaxy. For reasons of performing PSF photometry at the position of the brightest star in the difference images, we move the pixel coordinates of the star associated with the largest $\mathcal{F}$ value to the centre of the image.

(iv) For each star, the normalized Gaussian profile generated in (i) is placed at the appropriate pixel coordinates, and scaled by the star flux from (iii).

(v) Finally, a constant background level S_R is drawn from the continuous distribution, $U \sim (10, 1000)$, is added to the image.

(vi) It is common practice to create a high signal-to-noise reference frame by either stacking images or increasing the exposure time. To simulate this, we generate a 'noise map', $\sigma_{R,\,ij}$ to apply to $R_{\text{noiseless},\,ij}$ with 10 times less pixel variance as is applied to the target image. This can be achieved by scaling the uncertainties by a factor of $10^{-1/2} \sim 0.316$. Adopting the usual CCD noise model (equation 6) with $\sigma_0 = 5$ (ADU), $G = 1$ (e^-/ADU) and $F_{ij} = 1$, we compute the reference frame pixel uncertainties as

$$\sigma_{R,ij} = 10^{-1/2}\sqrt{\sigma_0^2 + R_{\text{noiseless},ij}} \ . \tag{22}$$

(vii) A 142 × 142 pixel image, W_{ij}, with values drawn from a standard normal distribution, $\mathcal{N}(0, 1)$, was also generated, and the noisy reference, $R_{\text{noisy},\,ij}$, is formed as

$$R_{\text{noisy},ij} = R_{\text{noiseless},ij} + W_{ij}\sigma_{R,ij} \ . \tag{23}$$

For each $R_{\text{noisy},\,ij}$, we then generate a corresponding target image.

(viii) As with the reference images, we choose a symmetrical Gaussian PSF for the target images, parametrized by ϕ_I. As the convolution of a Gaussian with a Gaussian is another Gaussian, the kernel is itself a Gaussian, and we draw the corresponding kernel width ϕ_K from $U \sim (0.5, 2.5)$. We can then compute the width of the PSF in the target image,

$$\phi_I^2 = \phi_R^2 + \phi_K^2 \ , \tag{24}$$

and repeat steps (iv)–(v), using ϕ_I in place of ϕ_R and S_I in place of S_R to generate $I_{\text{noiseless},\,ij}$. Additionally, we also apply sub-pixel shifts to the stellar fractional pixel positions along both the x- and y-axis, with the Δx and Δy shifts each drawn separately for each simulation from $U \sim (-0.5, 0.5)$.

(ix) Similar to (vi), we then generate the noise map

$$\sigma_{I,ij} = \sqrt{\sigma_0^2 + I_{\text{noiseless},ij}} \ , \tag{25}$$

(x) and then generate a 142 × 142 pixel image, W_{ij}, with values drawn from $\mathcal{N}(0, 1)$, and create $I_{\text{noisy},\,ij}$,

$$I_{\text{noisy},ij} = I_{\text{noiseless},ij} + W_{ij}\sigma_{I,ij} \ . \tag{26}$$

The signal-to-noise ratio (SNR) of the target image is computed as

$$\text{SNR}_I = \frac{\sum_{ij}(I_{\text{noiseless},ij} - S_I)}{\sqrt{\sum_{ij}\sigma_{I,ij}^2}} \ . \tag{27}$$

Table 1. The model accuracy, fit quality, and photometric performance metrics used to assess the DIA implementations in this work.

Metric	Definition	Equation
P	Photometric Scale Factor	(3)
MSE	Mean Squared Error	(28)
MFB	Mean Fit Bias	(29)
MFV	Mean Fit Variance	(30)
MPB	Mean Photometric Bias	(32)
MPV	Mean Photometric Variance	(33)

3.2 Performance metrics

We assess the fit quality and photometric accuracy of the kernel and background solutions with the following performance metrics. The derivations of these metrics are based on Section 5.2 of B16. For easy reference, the definitions and equation numbers for all metrics are listed in Table 1.

(i) Model error. The mean squared error (MSE) assesses how well the inferred model image, M_{ij}, matches the true target image, $I_{ij,\,\text{noiseless}}$,

$$\text{MSE} = \frac{1}{N_{\text{data}}}\sum_{ij}(M_{ij} - I_{\text{noiseless},ij})^2 \ . \tag{28}$$

The smallest values of MSE indicate the best fit in terms of model error.

As noted in Bramich et al. (2015), systematic errors in the photometric scale factor, P, can badly influence photometric accuracy. Given this, we also assess the inferred P and B_0 parameters, whose true values are equal to 1 and 0, respectively, in these simulations.

(ii) Fit quality. The mean fit bias (MFB) and mean fit variance (MFV) are measures of the bias and excess variance in M_{ij}, and are given as

$$\text{MFB} = \frac{1}{N_{\text{data}}}\sum_{ij}\frac{I_{\text{noisy},ij} - M_{ij}}{\sigma_{I,ij}} \ , \tag{29}$$

$$\text{MFV} = \frac{1}{N_{\text{data}} - 1}\sum_{ij}\left(\frac{I_{\text{noisy},ij} - M_{ij}}{\sigma_{I,ij}} - \text{MFB}\right)^2 \ . \tag{30}$$

Kernel and background solutions with an MFB close to 0, and an MFV close to unity are measured to have a good fit quality.

(iii) Photometric accuracy. To assess the photometric accuracy of the solution, we perform PSF fitting photometry at the position of the brightest star in the difference image. We generate a normalized PSF object parametrized by ϕ_R, centred at the position of the brightest star, and convolve this with the kernel solution. This is renormalized, giving us a normalized PSF object to fit to the difference image. The true target image PSF width is known by equation (24), and we set the size of the renormalized PSF object 'stamp' to be $9\phi_I \times 9\phi_I$ pixels large. We then fit this PSF stamp to an equally sized cutout from the difference image (centred at the position of brightest star) with a scaling factor $\mathcal{F}_{\text{diff}}$ and an additive constant, weighted with the known pixel variances in the target image, $\sigma_{I,ij}^2$. The fitted $\mathcal{F}_{\text{diff}}$ is then scaled to the photometric scale of the reference image, giving $\mathcal{F}_{\text{measured}} = \mathcal{F}_{\text{diff}}/P$.

The theoretical minimum variance of $\mathcal{F}_{\text{measured}}$ for a PSF-fitting procedure that scales a PSF model to a stamp with pixel indices rs is

$$\sigma_{\text{min}}^2 = \frac{1}{P_{\text{true}}^2}\left(\sum_{rs}\frac{\mathcal{P}_{I,rs}^2}{\sigma_{I,rs}^2}\right)^{-1} \ , \tag{31}$$

where $P_{\rm true}$ is the true photometric scale factor (equal to 1 in our simulations) and $\mathcal{P}_{I,rs}$ is the true PSF of the brightest star in the target image i.e. a normalized Gaussian with standard deviation ϕ_I.

As there are no variable sources, over $N_{\rm sim}$ simulations of accurate kernel and background solutions we should expect a distribution of $\mathcal{F}_{\rm measured}/\sigma_{\rm min}$ values with mean 0 and a variance of unity. The mean photometric bias (MPB) and mean photometric variance (MPV) over $N_{\rm sim}$ simulations, each indexed by k, would be equal to the statistics

$$\mathrm{MPB} = \frac{1}{N_{\rm sim}} \sum_k \frac{\mathcal{F}_{{\rm measured},k}}{\sigma_{{\rm min},k}} \tag{32}$$

$$\mathrm{MPV} = \frac{1}{N_{\rm sim}-1} \sum_k \left(\frac{\mathcal{F}_{{\rm measured},k}}{\sigma_{{\rm min},k}} - \mathrm{MPB} \right)^2 . \tag{33}$$

As noted in B16, although the MPV should be close to unity, it may have values less than this when the target image is overfitted, and/or when the model PSF fitted to the difference image (i.e. formed from the convolution of the reference image PSF with the inferred kernel) is different than the true PSF of the target image.

Fig. 2 shows an example 142 × 142 [pixels] reference and target image pair generated for these tests. The difference images and kernels corresponding to the B08 and PyTorchDIA solutions, annotated with the fit quality and photometric accuracy metrics, are shown below.

3.3 Simulated image test results

In these tests, both B08 and PyTorchDIA attempt to fit the image model in equation (1), with an unregularized (square) 19 × 19 pixel kernel. The generated (square) reference and target images are each 142 × 142 pixels large, and so the number of target image pixels that enter the fit is $N_{\rm data} = (142 - 2 \times 9)^2 = 15\,376$. B16 used 141 × 141 large reference images and included 10 201 target image pixels in their solution, and so the results presented here can be meaningfully compared against that prior work.

Following B16, we first divide the results of our 71 569 simulation tests into three regimes by the SNR of the target image, I: (1) $8 < \mathrm{SNR}_I < 40$; (2) $40 < \mathrm{SNR}_I < 200$, and (3) $200 < \mathrm{SNR}_I < 1000$. Each of these three categories is then divided into four further categories by the sampling regime of the images: (i) $\phi_R > 1$ and $\phi_K > 1$, (ii) $\phi_R > 1$ and $\phi_K < 1$, (iii) $\phi_R < 1$ and $\phi_K > 1$, (iv) $\phi_R < 1$ and $\phi_K < 1$.

The distributions of the fit metrics are in general skewed, and so we report the median values as a robust estimate of the central value in Fig. 3 for both the B08 and PyTorchDIA implementations. Each sub-plot pair in Fig. 3 pertains to a single metric, with the B08 results in blue in the left column, and PyTorchDIA results in red in the right column. On the x-axis, we plot the SNR regime of the target image, categorized as above. There are four points in each SNR regime in each sub-plot, each of which corresponds to a different sampling regime, and we offset these from each other for clarity. We use circular markers to denote the sampling regime of the reference image, and crosses to indicate the sampling regime of the kernel. A big circle or cross corresponds to an oversampled reference image or kernel, respectively, and a small circle or cross corresponds to an undersampled reference image or kernel. The green dashed lines for each sub-plot pair represent the 'best' value for each metric. We also tabulate these results in Table 2, and include the 16th and 84th percentiles about the median of the distributions. In the bottom section of this table, we note the number of simulations that fall into each SNR and sampling regime category.

Across all the metrics, one of the biggest differences between the B08 and PyTorchDIA solutions are seen in the values for the photometric scale factor. B08 is more accurate in general, but differences between the median values in each SNR and sampling regime are typically small (∼0.001), and become negligible when the SNR is high. B16 showed that the accuracy of P is strongly determined by the SNR of the target image, and in Table 2, we also see that the distribution of P values about the median becomes tightly concentrated about unity as the SNR increases. Interestingly, for results in any given sampling regime, there is no clear similarity in trends between the two approaches with the target image SNR.

There are no large differences between B08 and PyTorchDIA in terms of MSE, with B08 better only at the level of the first decimal place. Looking at Table 2, B08 does, however, begin to slightly outperform PyTorchDIA in the highest SNR regimes when both the kernel and the reference frame are undersampled.

As observed in B16, both approaches show overall gradually decreasing MFB as the SNR increases, although B08 shows a negative bias while PyTorchDIA is typically positively bias. PyTorchDIA again only appears to do noticeably worse than B08 in the highest SNR category when both the reference image and the kernel are undersampled.

The MFV values returned by B08 and PyTorchDIA are also very similar. The B08 MFV values are usually lower than those for PyTorchDIA, although all median values are less than unity. These results too are consistent with B16, who also found the unregularized kernel was prone to overfitting when the SNR of the target image is low.

The differences between B08 and PyTorchDIA in terms of MPB are small, with both showing the same trends with SNR and sampling regimes. As found by B16, with increasing SNR, the MPB greatly improves. That this is typically best when the kernel is oversampled, and worst when both the kernel and reference image are undersampled, is likely in part due to sampling issues associated with the PSF model which is scaled to the flux in the difference image. Encouragingly, with reference to the values in Table 2, we can see that in general, PyTorchDIA performs even better than B08. That the MPB metrics are similar despite the differences in MFB is likely due to our choice to simultaneously fit for a constant scalar offset in our PSF fitting procedure, which would correct for any net over-/underestimation in the background parameter of the image model.

The same trends in the MPV values with both SNR and sampling regimes can again be seen by the two approaches. With an increasing SNR, the MPV increases. This behaviour was also seen by B16, and it can be explained as reduced overfitting of the bright central stars from which this metric is computed. For both approaches, apart from some cases when both the kernel and reference image are undersampled, the MPV is always less than unity, and similar results were obtained by B16 (see fig. 6 therein). We experimented by performing an additional 21 271 tests in which the photometry of the faintest star in the image was used to compute the MPV. In this instance, no MPV values were less than 1.1.

While being overall very similar, the PyTorchDIA MPV values are slightly higher than their B08 counterparts. Because B08 overfits more strongly than PyTorchDIA (see MFV values), we should expect the noise in the B08 difference images to be slightly more suppressed. Consequently, the photometry will typically have a lower variance. The difference in MPV is greatest when both the kernel and reference image are undersampled. This may be associated with the worse MFB values for PyTorchDIA in this category, as while the additive constant in our PSF fitting procedure will be able to correct for inaccuracies in the differential background, it will do so at the cost of slightly increased variance in the photometry.

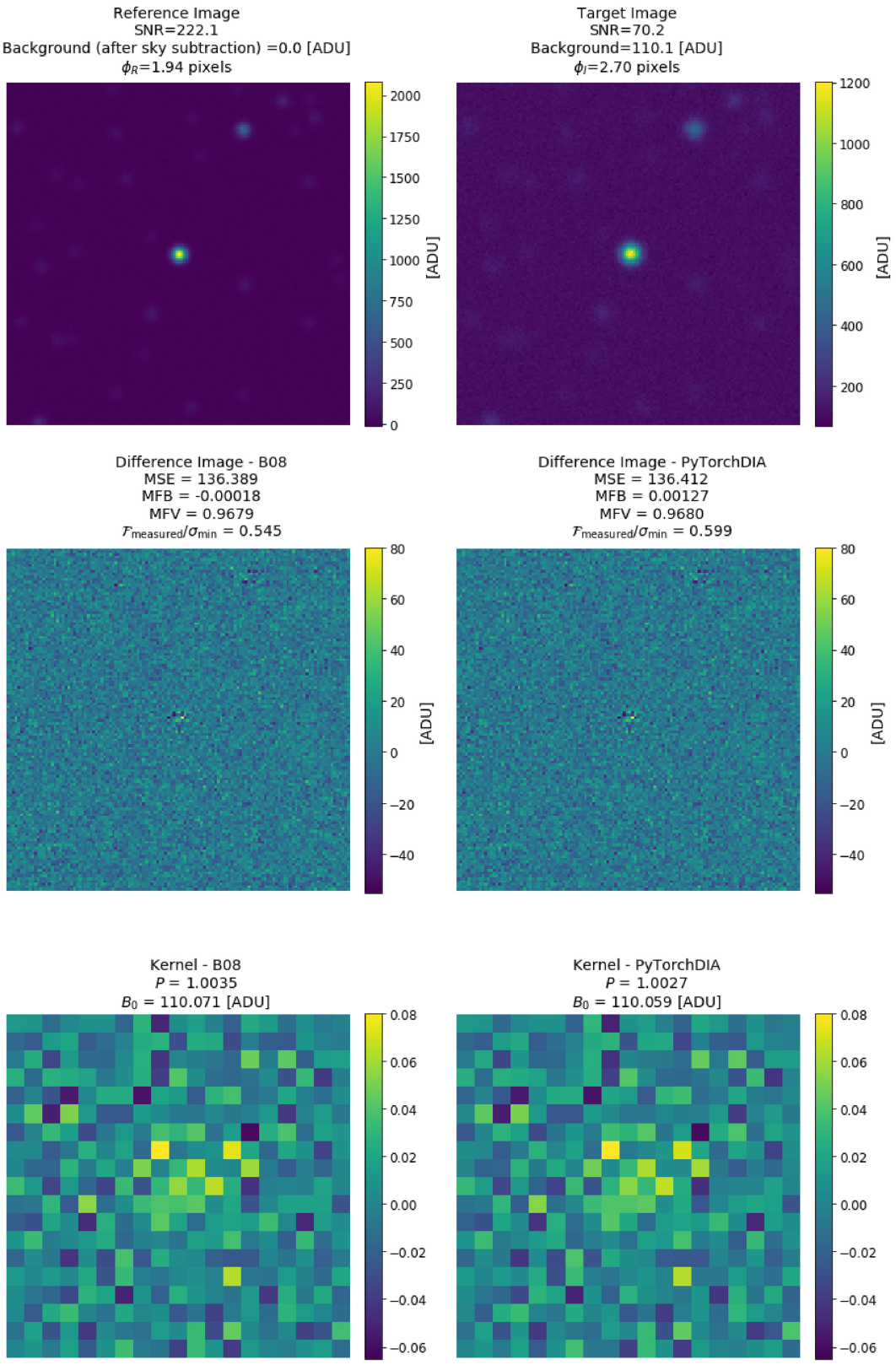

Figure 2. Example images and fit quality metrics from the simulated image experiments. (*Top row*) An example reference (left) and target (right) image pair generated in the simulation tests. (*Middle row*) The subsequent difference images and fit metrics recovered by the B08 approach (left) and our PyTorch implementation (right). (*Bottom row*) The corresponding convolution kernels and fit parameters for the B08 (left) and PyTorch (right) solutions. The median pixel value is subtracted from the reference image before fitting the kernel and differential background term, and so B_0 will not be equal to zero. The B08 and PyTorchDIA solutions are very similar.

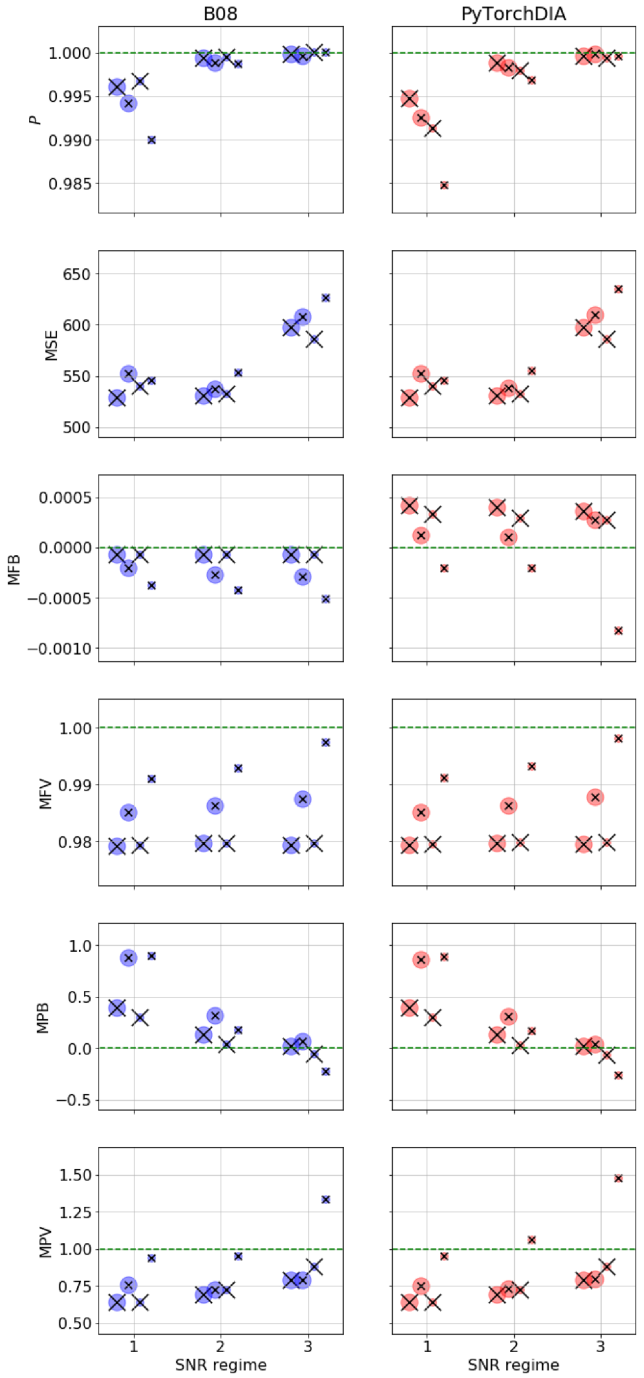

Figure 3. Fit quality and photometric accuracy metrics from the 71 569 simulated image tests. Results for the B08 algorithm are in blue (left column), and the PyTorchDIA results are in red (right column). The signal-to-noise regime of the target image is shown on the x-axis, increasing from left to right. Metrics in each SNR regime are offset from each other for clarity. We use circular markers to denote the sampling regime of the reference image, and crosses to indicate the sampling regime of the kernel. A big circle or cross corresponds to an oversampled reference image or kernel, respectively, and a small circle or cross corresponds to an undersampled reference image or kernel; there are therefore four possible combinations of marker for each SNR regime. The green dashed lines for each sub-plot pair represent the correct, 'ideal' value for each metric.

In conclusion, PyTorchDIA performs very similarly to the B08 algorithm in these tests on synthetic images, and it is encouraging to see that the major conclusions drawn from these tests are consistent with those in B16. These images have Gaussian noise contributions, and have no outliers, and it is therefore not surprising that the B08 algorithm does slightly better overall, since the linear least-squares formulation (within an iterative scheme) correctly describes the observation model, and this approach operates at twice the floating-point precision of PyTorchDIA.

3.4 No 'algorithmic' bias

As PyTorchDIA is a numerical algorithm, it is essential to know to what precision it can recover the correct solution if random noise has been removed from the data. By default, PyTorchDIA operates at F32, giving us 6–8 decimal digits of precision.

Following section 3.1 of B08, we perform the following tests (i)–(iv) where known, noiseless transformations are applied to a reference image to generate a target image. The 142 × 142 pixel reference image used in what follows is generated randomly, as outlined in Section 3.1. We compute the fractional error, ϵ, of P and B_0, as $\epsilon_P = |(P - P_{\rm True})/P_{\rm True}|$ and $\epsilon_{B_0} = |(B_0 - B_{0,\rm True})/B_{0,\rm True}|$.[9]

(i) The simplest possible test we can perform is to difference an image against a copy of itself. The kernel should be exactly 1 at the centre and 0 everywhere else. Encouragingly, PyTorchDIA is able to return the correct answer to within the theoretical convergence tolerance, with fractional errors of $\epsilon_P = 4 \times 10^{-7}$ and $\epsilon_{B_0} = 1 \times 10^{-8}$. The central kernel pixel was correct to a precision of 7 decimal digits.

(ii) Next, we convolve the reference with a Gaussian kernel of $\phi_K = 1.5$ pixels. Once again, PyTorchDIA returns this Gaussian kernel to within its convergence tolerance, with $\epsilon_P = 3 \times 10^{-7}$ and $\epsilon_{B_0} = 1 \times 10^{-8}$.

(iii) The target image is created by shifting the reference image by one pixel in each of the positive x and y spatial directions, without resampling. The corresponding kernel should be the identity kernel (central pixel value of 1 and 0 elsewhere) shifted by one pixel in each of the negative l and m kernel coordinates. PyTorchDIA returns a kernel with a peak pixel value of unity precise to F32 machine precision, and $\epsilon_P = 4 \times 10^{-7}$ and $\epsilon_{B_0} = 1 \times 10^{-8}$.

(iv) The reference image is shifted by half a pixel in each of the positive x and y spatial directions to create the target image, an operation that requires the resampling of the reference image. We use a bicubic spline resampling method.[10] PyTorchDIA performs very well, successfully returning the complicated kernel, with fractional errors on P and B_0 of $\epsilon_P = 4 \times 10^{-7}$ and $\epsilon_{B_0} = 1 \times 10^{-8}$.

As PyTorchDIA is able to recover the true fit parameters correct to the data type's decimal digit precision, we conclude that there is no bias associated with our numerical algorithm.

4 REAL IMAGE TESTS

DIA is a particularly effective tool for measuring the flux and positions of variable sources in crowded fields (see the start of Section 1 for some examples). The MiNDSTEp consortium performs follow-up observations of microlensed targets towards the galactic bulge with the high frame-rate EMCCD cameras on the Danish

[9] As always, we background subtract the reference frame, so $B_0 > 0$.
[10] Specifically, we use the function scipy.ndimage.interpolation.shift, with no pre-filtering.

Table 2. Fit quality and photometric accuracy metrics for the 71 569 simulated image tests for both an implementation of the B08 algorithm and PyTorchDIA. We divide the results of these tests by the SNR regime of the target image and the sampling regimes of the reference image and convolution kernel. The number of simulations used for the computation of the metrics in each of the 12 possible SNR and sampling regime categories is shown in the bottom section.

Metric	SNR regime	$\phi_R > 1, \phi_K > 1$	$\phi_R > 1, \phi_K < 1$	$\phi_R < 1, \phi_K > 1$	$\phi_R < 1, \phi_K < 1$
P (B08)	1	$0.9961^{+0.0335}_{-0.0358}$	$0.9942^{+0.0316}_{-0.0380}$	$0.9968^{+0.0273}_{-0.0323}$	$0.9900^{+0.0279}_{-0.0342}$
P (PyTorchDIA)	1	$0.9948^{+0.0335}_{-0.0360}$	$0.9926^{+0.0320}_{-0.0376}$	$0.9913^{+0.0277}_{-0.0328}$	$0.9848^{+0.0285}_{-0.0368}$
P (B08)	2	$0.9994^{+0.0125}_{-0.0125}$	$0.9989^{+0.0124}_{-0.0134}$	$0.9996^{+0.0101}_{-0.0108}$	$0.9987^{+0.0103}_{-0.0111}$
P (PyTorchDIA)	2	$0.9989^{+0.0124}_{-0.0126}$	$0.9983^{+0.0128}_{-0.0135}$	$0.9980^{+0.0098}_{-0.0112}$	$0.9969^{+0.0114}_{-0.0135}$
P (B08)	3	$0.9998^{+0.0045}_{-0.0046}$	$0.9997^{+0.0045}_{-0.0046}$	$1.0001^{+0.0037}_{-0.0040}$	$1.0001^{+0.0039}_{-0.0040}$
P (PyTorchDIA)	3	$0.9996^{+0.0045}_{-0.0045}$	$0.9999^{+0.0047}_{-0.0050}$	$0.9995^{+0.0038}_{-0.0040}$	$0.9996^{+0.0042}_{-0.0049}$
MSE (B08)	1	$528.53^{+328.79}_{-329.61}$	$552.62^{+310.51}_{-349.03}$	$539.85^{+318.94}_{-342.21}$	$545.34^{+313.22}_{-340.47}$
MSE (PyTorchDIA)	1	$528.54^{+328.80}_{-329.60}$	$552.63^{+310.51}_{-349.03}$	$539.86^{+318.95}_{-342.21}$	$545.42^{+314.59}_{-340.54}$
MSE (B08)	2	$530.88^{+343.43}_{-330.54}$	$537.47^{+337.46}_{-328.53}$	$532.74^{+342.01}_{-337.18}$	$553.47^{+336.57}_{-347.22}$
MSE (PyTorchDIA)	2	$530.90^{+343.43}_{-330.55}$	$537.92^{+337.16}_{-328.88}$	$532.82^{+341.94}_{-337.24}$	$555.51^{+337.07}_{-346.81}$
MSE (B08)	3	$597.17^{+352.67}_{-362.18}$	$608.09^{+361.21}_{-361.96}$	$585.85^{+365.45}_{-356.65}$	$626.40^{+415.64}_{-366.67}$
MSE (PyTorchDIA)	3	$597.33^{+352.64}_{-362.28}$	$610.09^{+361.05}_{-363.08}$	$585.93^{+365.41}_{-356.49}$	$634.81^{+415.59}_{-371.31}$
MFB (B08)	1	$-0.0001^{+0.0000}_{-0.0001}$	$-0.0002^{+0.0001}_{-0.0002}$	$-0.0001^{+0.0000}_{-0.0001}$	$-0.0004^{+0.0002}_{-0.0003}$
MFB (PyTorchDIA)	1	$0.0004^{+0.0004}_{-0.0003}$	$0.0001^{+0.0003}_{-0.0003}$	$0.0003^{+0.0004}_{-0.0004}$	$-0.0002^{+0.0005}_{-0.0006}$
MFB (B08)	2	$-0.0001^{+0.0000}_{-0.0001}$	$-0.0003^{+0.0001}_{-0.0002}$	$-0.0001^{+0.0000}_{-0.0001}$	$-0.0004^{+0.0002}_{-0.0004}$
MFB (PyTorchDIA)	2	$0.0004^{+0.0005}_{-0.0004}$	$0.0001^{+0.0009}_{-0.0008}$	$0.0003^{+0.0004}_{-0.0003}$	$-0.0002^{+0.0044}_{-0.0060}$
MFB (B08)	3	$-0.0001^{+0.0001}_{-0.0001}$	$-0.0003^{+0.0001}_{-0.0003}$	$-0.0001^{+0.0001}_{-0.0002}$	$-0.0005^{+0.0003}_{-0.0008}$
MFB (PyTorchDIA)	3	$0.0004^{+0.0007}_{-0.0007}$	$0.0003^{+0.0018}_{-0.0013}$	$0.0003^{+0.0005}_{-0.0004}$	$-0.0008^{+0.0025}_{-0.0044}$
MFV (B08)	1	$0.9792^{+0.0116}_{-0.0112}$	$0.9851^{+0.0116}_{-0.0115}$	$0.9794^{+0.0114}_{-0.0115}$	$0.9911^{+0.0115}_{-0.0127}$
MFV (PyTorchDIA)	1	$0.9793^{+0.0116}_{-0.0112}$	$0.9851^{+0.0117}_{-0.0115}$	$0.9794^{+0.0114}_{-0.0115}$	$0.9912^{+0.0116}_{-0.0127}$
MFV (B08)	2	$0.9796^{+0.0113}_{-0.0116}$	$0.9862^{+0.0121}_{-0.0123}$	$0.9797^{+0.0113}_{-0.0117}$	$0.9930^{+0.0143}_{-0.0133}$
MFV (PyTorchDIA)	2	$0.9797^{+0.0113}_{-0.0116}$	$0.9863^{+0.0123}_{-0.0123}$	$0.9798^{+0.0113}_{-0.0117}$	$0.9933^{+0.0147}_{-0.0134}$
MFV (B08)	3	$0.9794^{+0.0114}_{-0.0112}$	$0.9875^{+0.0122}_{-0.0117}$	$0.9797^{+0.0118}_{-0.0113}$	$0.9974^{+0.0304}_{-0.0156}$
MFV (PyTorchDIA)	3	$0.9796^{+0.0114}_{-0.0112}$	$0.9878^{+0.0128}_{-0.0118}$	$0.9799^{+0.0118}_{-0.0114}$	$0.9982^{+0.0384}_{-0.0161}$
MPB (B08)	1	0.401	0.880	0.300	0.900
MPB (PyTorchDIA)	1	0.400	0.862	0.300	0.887
MPB (B08)	2	0.138	0.324	0.038	0.184
MPB (PyTorchDIA)	2	0.136	0.313	0.036	0.176
MPB (B08)	3	0.027	0.074	−0.055	−0.219
MPB (PyTorchDIA)	3	0.023	0.046	−0.059	−0.257
MPV (B08)	1	0.640	0.760	0.640	0.937
MPV (PyTorchDIA)	1	0.639	0.754	0.640	0.955
MPV (B08)	2	0.692	0.728	0.725	0.951
MPV (PyTorchDIA)	2	0.693	0.728	0.726	1.066
MPV (B08)	3	0.789	0.788	0.880	1.339
MPV (PyTorchDIA)	3	0.789	0.794	0.883	1.478
$N_{\text{Simulations}}$	**1**	12332	4151	4078	1349
	2	13224	4447	4387	1526
	3	14673	4900	4829	1673

1.54-m telescope (DK154), at ESO La Silla (Skottfelt et al. 2015). Short 0.1 s exposures are shift-and-stacked to eliminate tip tilt distortions, and recover scenes with ∼2–3 times better resolution than conventional long integrations. The combined effects of the mount (the DK154 rests on three points) and the shift-and-add procedure produces irregular, triangular PSFs, with a sharp peak and diffuse halo. With a scale of 0.09 arcsec per pixel, this high resolution EMCCD instrument explores a subset of the sampling regimes covered in the CCD simulations in Section 3.

4.1 Data and reductions

We use a sample of 251 512 × 512 pixel images, each consisting of up to 3000 shift-and-stacked 0.1 s short exposures[11] (i.e. each stack is equivalent to at most a 5 min exposure), obtained with

[11] During the shift-and-stack procedure, if a shift is above a certain threshold the frame is rejected, and so some stack consist of slightly less than 3000 images.

the 'red' camera (approximately equivalent to a broad SDSS 'z' filter; Fukugita et al. 1996) on the Danish 1.54-m Lucky Imaging instrument between 2019 July 17 and 2019 July 30. The observations are of the centre of the OGLE-III-BLG101 microlensing field, (l, b) = (0.1331°, −1.9643°). With a pixel scale of 0.09 arcsec per pixel, the camera covers a FoV of 45 × 45 arcsec^2. All images are bias subtracted and flat corrected, and the master flats associated with each night of observing are used in the noise model for the following DIA performance tests (equation 7).

To create a high signal-to-noise reference, 13 sharp images acquired on 2019 July 20 were registered to the same pixel grid using bicubic spline resampling. The stacked reference frame was constructed by then summing the registered images, and dividing by 13. To measure the PSF of this stacked image, a (symmetric) Gaussian PSF model was independently fit to 5 bright, isolated stars, from which we found a median width of $\phi_R = 2.89$ pixels. The median ϕ_I of the 238 remaining target images was 3.83 pixels, with a maximum of 6.67 pixels.

In preparation for assessing the photometric accuracy of the algorithms, we measured the fluxes and positions of the stars in the reference image. A total of 236 candidate sources were detected in the image. We avoided all stars near the reference image edges and those with a peak pixel flux less than 3×10^4 ADU. This gave us a reduced sample size of the 37 brightest stars. Due to processing time constraints (see below), this sample was further reduced to 30, approximately uniformly spaced stars. The light curves of these 30 stars can be used to compute the MPB and MPV metrics (see Section 4.2).

For both the reference and each of the 238 target images, a 142 × 142 pixel region was cutout around the positions of each of the 30 stars. This approach avoids resampling the target images to align them with the reference, and prevents introducing correlated noise into the target images by interpolating between pixels.[12]

Cursory image model fits to these target image stamps identified kernels with a size of $7\phi_I \times 7\phi_I$ to give good results. For the tests in Section 3, a $9\phi_I \times 9\phi_I$ kernel corresponded to a 19 × 19 pixel array. For these tests on real images, even though the kernel is set to be just $7\phi_I \times 7\phi_I$ large, due to the fine sampling of the LI images, the median size of the kernel in pixel-space for this data set is 27 × 27 pixels, with a maximum size of 47 × 47 pixels. For the B08 algorithm in particular – which scales with the square of the number of N_K kernel pixels – processing times are much more expensive than in Section 3, and this is why we further reduced our sample of bright stars down from 37 to 30 to keep this investigation tractable.

This should give us a total of 7140 reference-target image pairs (i.e. ∼ 10 per cent the size of the simulated image data set), but in some instances, for stars close to the borders of the target image, large shifts between this image and the reference meant that part of the 142 × 142 pixel region fell outside the image edge. Additionally, the measured SNR of the target image in some instances exceeded 1000, and so these were rejected to provide a more consistent comparison with the results from Section 3. After these cuts, we had a final sample of 6989 reference-target image pair stamps.

4.2 Model performance metrics for real data

For this test, we compare the PyTorchDIA code implementing our robust loss function with $c = 1.345$ (equation 13), against the B08 solution making use of sigma-clipping. We run the B08 approach for a total of 3 iterations, and clip $|\epsilon_{ij}| > 5$ pixels on the third iteration only.

Although we are both limited to a smaller sample size and do not know the true noise properties or model parameters of the actual images and kernel solutions, respectively, we can still use almost all of the metrics outlined in Section 3.2 to assess the accuracy of the implementations. As the true model image is unknown, we are unable to use the MSE metric to measure the model error. The photometric scale factor, P, while also unknown, can however be compared on a relative scale, which requires an estimate for $P_{\rm true}$.

It was found that the inferred P was correlated with the distance from the image centre along the x-axis. This is due to a change in the sky level along this axis associated with instrumental effects, which influences P due to the anticorrelation between the photometric scale factor and B_0. Consequently, $P_{\rm true}$ for the entire image is not well represented by a single number (e.g. the median value from all 30 stamps). Given this, we divided the x-axis into four equally spaced regions, and computed four separate $P_{\rm true}$ values using the subset of stamps in each of these regions. The measured P from any stamp was then normalized by the $P_{\rm true}$ corresponding to the region it belonged to along the chip's x-axis. As B08 outperformed PyTorchDIA in terms of P in Section 3, we use the P values inferred by this approach to determine $P_{\rm true}$.

The remaining metrics require an accurate noise model to be meaningful. For these EMCCD images, we use equation (7), and substitute the M_{ij} obtained on the third and final iteration of the B08 solution of each reference-target image pair to estimate the pixel uncertainties. These pixel uncertainties can then be used to calculate the MFB and MFV metrics, in place of $\sigma_{I,ij}$. Similarly, by substituting the target image for $I_{\rm noiseless,\,ij}$ in equation (27) we can use these pixel uncertainties to calculate the SNR of the target image for each reference-target image pair. For the third iteration of the B08 approach, we also sigma-clip $|\epsilon_{ij}| > 5$ pixels to guard against variable sources or spurious pixels affecting the kernel solution. We use this associated bad pixel mask to omit these pixels from the calculation of the corresponding MFB and MFV metrics for both the B08 and PyTorchDIA solutions.

We perform PSF fitting photometry at the centre of the difference images obtained from each reference-target image pair (i.e. at the position of one of the 30 possible stars) to assess the photometric accuracy of the approach. We infer an empirical model PSF for the target image in the following way. For each reference-target image pair, we identify the peak pixel values of bright sources in the reference, and set all other pixel values to 0. We then use our PyTorchDIA implementation to infer the 'PSF' which convolves this scene of (approximate) delta-functions to the target image. That is, we are inferring an estimate of the PSF of the target image with the 'kernel' solution of the PyTorchDIA code. When inferring this empirical PSF, we weight all pixels equally, which gives the pixel values of the very brightest stars more weight relative to the other pixels in the image.

As in Section 3.2, we use a small square stamp for the PSF fitting. We set the radius of this stamp (i.e. the half-width of the stamp) to 1.5 FWHMs of the target image (i.e. $\sim 1.5 \times 2.355 \times \phi_I$ pixels), rounded up to the nearest odd integer pixel. The PSF object is normalized, and scaled to the flux of the difference image. To guard against the influence of (possibly variable) neighbouring sources or spurious pixels in the stamp not captured by the PSF model, we scale the model to a star's differential flux under our robust loss function (equation 12), with fixed pixel uncertainties and $c = 1.345$.

[12]We note that the noise in all shift-and-stacked images will be correlated to some degree due to the shift procedure.

As with the MFB and MFV metrics, we adopt equation (7) as the noise model, substituting the M_{ij} values obtained on the final iteration of the B08 solution. In order to calculate $\sigma^2_{\rm min}$ (equation 31), we substitute the normalized empirical PSF inferred by the PyTorchDIA code for $\mathcal{P}_{I,rs}$, and substitute the region-dependent $P_{\rm true}$ inferred by the B08 implementation. An example reference-target image pair and associated fit quality metrics for both the B08 and PyTorchDIA implementations are shown in Fig. 4.

4.3 Real image test results

We again start our analysis by splitting our results into three SNR regimes by the signal-to-noise of the target image, I: (1) $8 < {\rm SNR}_I < 40$; (2) $40 < {\rm SNR}_I < 200$ and (3) $200 < {\rm SNR}_I < 1000$. We found that no reference-target image pairs correspond to the very lowest SNR regime, and so we are forced to restrict our analysis to regimes (2) and (3). Also, unlike the simulated image tests, all 251 real images used here are well sampled. A majority 195 of the target images correspond to the case where the kernel is oversampled (i.e. $\phi_K > 1$), and for the other images the kernel is undersampled (i.e. $\phi_K < 1$). We therefore further divide each signal-to-noise regime into two sampling regimes, dependent on whether the kernel is approximately over or undersampled. We plot the median metric values and the MPB and MPV for both B08 (blue) and PyTorchDIA (red), each split into the four possible sub-categories in Fig. 5. As all images are oversampled, the circular markers are all large. The small and large crosses correspond to under- and oversampled kernels. The theoretical best value for each metric is plotted as a dashed green line. The median metrics and the 16th and 84th percentiles of their distributions are tabulated in Table 3.

It was noted that some star differential light curves could have a noticeable non-zero offset from their reference frame flux level. In order to correct for this before grouping the normalized photometric residuals from across all light curves into each of the SNR and sampling regime categories, each star light curve was shifted such that its median photometric residual value was 0. Additionally, there was a small subset of outlying photometric residuals that may badly affect the MPB and MPV metrics. To remove the influence of these bad outliers in each category, we first calculate the median absolute deviation of the $\mathcal{F}_{{\rm measured},k}/\sigma_{{\rm min},k}$ values scaled to a standard deviation $\sigma_{\rm MAD}$ as a robust estimate of the spread of the underlying distribution. This is defined as

$$\sigma_{\rm MAD} = \kappa \times {\rm median}\left|\frac{\mathcal{F}_{{\rm measured},k}}{\sigma_{{\rm min},k}} - {\rm median}\left(\frac{\mathcal{F}_{\rm measured}}{\sigma_{\rm min}}\right)\right|, \qquad (34)$$

where $\kappa = 1.4826$ for approximately normally distributed data. Before calculating the MPB and MPV metrics shown in Fig. 5 and Table 3, we have removed all photometric residuals with absolute values more than $4.5\sigma_{\rm MAD}$ from median$(\mathcal{F}_{\rm measured}/\sigma_{\rm min})$ for each SNR and sampling regime category. Note that the number of outlying points may differ for the B08 and PyTorchDIA solutions.

We tabulate the number of reference-target image pairs in each category in Table 3. Due to the sigma-clipping used before computing the MPB and MPV, we write the number of image pairs in each category used to compute these particular metrics as a fraction of the total number of image pairs. All image pairs in each category were used to compute all the other metrics, as these are robust to outliers.

We can again see that the PyTorchDIA and B08 approaches are broadly consistent, with both DIA implementations returning very similar metric values. Across all SNR and sampling regimes, only the median MFB values show consistently large differences. PyTorchDIA exhibited notably larger MFB values in the tests on simulated images, and here too, the B08 approach performs better with respect to fit bias. The P values for B08 appear to be superior, but recall that these are normalized to the median P values obtained from each x-axis sub-region by the B08 algorithm, so we should expect the B08 P values to be closer to unity. For both implementations, as seen in the tests in Section 3 and similar experiments in B16, the accuracy with which P can be determined clearly improves with increasing SNR.

Similar trends with both MFV and MPB are seen from both approaches. There is a relative paucity of results corresponding to an undersampled kernel – particularly for SNR category 2 – but for the richly populated categories corresponding to oversampled kernels, both of these metrics improve with increasing SNR, as also observed in the simulated tests.

The MPV metrics for both approaches are noticeably larger here than in the simulations. However, we again see that the photometric variance goes up with the SNR of the target image, and that both approaches are overall very similar. Unlike the simulations, there is no guarantee that the 30 selected stars are the brightest in their respective stamps, and each stamp is well populated with stars in this crowded field (see top panel of Fig. 4). We should not then be overfitting the target stars, and so MPVs greater than unity are expected, since the fractional contribution of flux of the target to the entire stamp will, in general, be much less than unity (see e.g. fig. 10 of B16). Also, of the 43 images corresponding to an undersampled kernel, 23 of these images have a PSF sharper than the stacked reference. In these instances, the model image will inevitably fail to match the PSFs – as it would in fact have to de-convolve, rather than convolve the reference – which will result in bad artefacts at the position of sources in the difference image. Indeed, in each SNR regime, the MPV corresponding to the undersampled kernel is much worse than its oversampled counterpart. However, these effects are common to both approaches, and here, we stress that we simply want to highlight the similarity of the MPV metrics obtained by the B08 and PyTorchDIA implementations. Indeed, the PyTorchDIA metrics can only be compared in a relative sense to the B08 results, as the B08 solutions for P and σ_{ij} are used to compute these metrics for both approaches. As we do not know the ground-truth for these values, we cannot say whether B08 or PyTorchDIA is closest to the correct answer, but only how similar they are to each other.

In addition to the differences in floating-point precision that would have influenced the small differences in results in Section 3, the two approaches now assume two different noise distributions for the target image pixels. For this real, outlier populated data, PyTorchDIA optimises a robust scalar objective function, while the B08 approach uses an iterative sigma-clipping procedure, so we should expect this to contribute to differences between the metric values also. And of course, the sample sizes in each category are smaller than their simulated data counterparts (particularly so for the instance of $\phi_K < 1$ in SNR category 2) and therefore noisier, which could partly explain some of the small differences between the metric values for the two approaches.

In summary, similar trends in metrics with SNR and sampling regime categories are shown by both the B08 and PyTorchDIA implementations, and these trends resemble those found in our experiments on simulated images. This both validates that our simulated images are reasonable approximations of real data, and that our robust PyTorchDIA solution provides competitive photometric performance with the B08 algorithm when applied to real astronomical data sets.

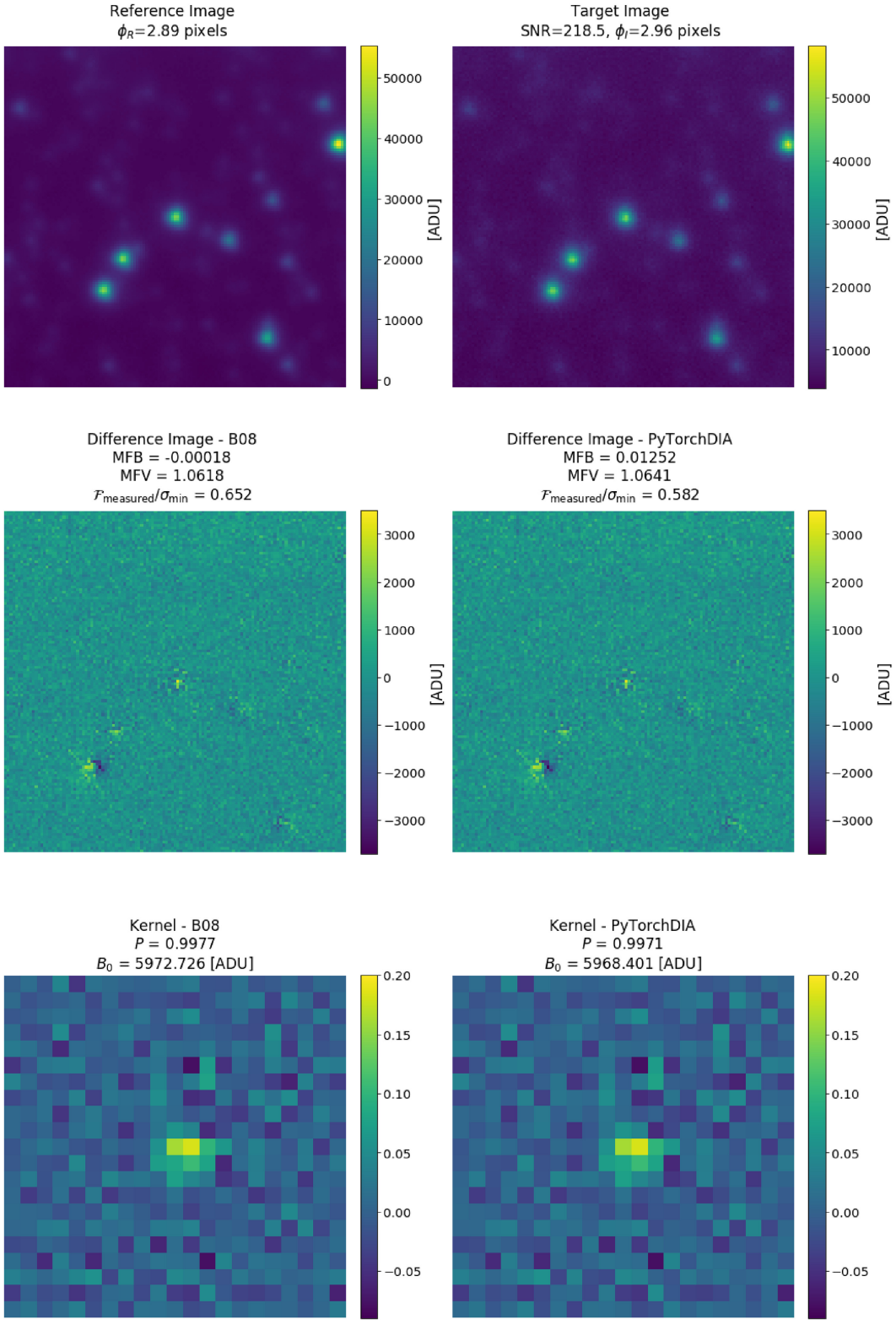

Figure 4. Example reference-target image pair from the real image performance tests, in the same style as Fig. 2. Note the unusual PSFs in the reference and target images, and the correspondingly irregular kernels. The target star with which the photometric accuracy was assessed is at the centre of the reference-target image pair.

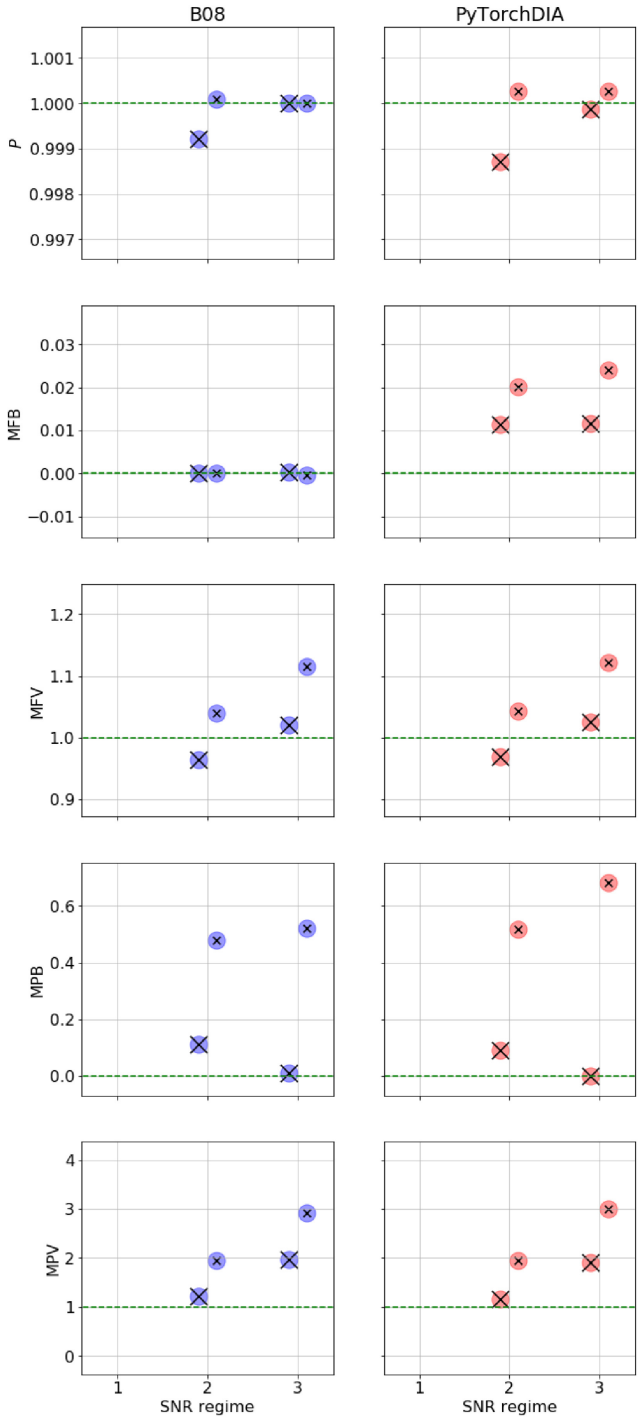

Figure 5. Fit quality and photometric accuracy metrics from the 6989 real image tests. Results for the B08 algorithm are in blue (left column), and the PyTorchDIA results are in red (right column). The signal-to-noise regime of the target image is shown on the *x*-axis, increasing from left to right. We use circular markers to denote the sampling regime of the reference image, and crosses to indicate the sampling regime of the kernel. As the reference image is oversampled, all circular markers are large. A large cross corresponds to an oversampled kernel, and a small cross corresponds to an undersampled kernel; there are therefore two possible combinations of marker for each SNR regime. No reference-target image pairs used to compute these metrics fell into the lowest SNR regime, and so that is left blank. The green dashed lines for each sub-plot pair represent the correct, 'ideal' value for each metric.

Table 3. Fit quality and photometric accuracy metrics for the 6989 real image tests for both an implementation of the B08 algorithm and PyTorchDIA, separated into the SNR and sampling regimes. The number of image stamps used to compute the metrics in each of the SNR and sampling regime categories is shown in the bottom section of the table. As outlying photometric residuals have been removed from some categories prior to computing the MPB and MPV metrics, we write the number of $N_{\rm Stamps}$ values used in their computation as a fraction of the total number of stamp samples.

Metric	**SNR regime**	$\phi_{\rm R} > 1, \phi_{\rm K} > 1$	$\phi_{\rm R} > 1, \phi_{\rm K} < 1$
P (B08)	1	–	–
P (PyTorchDIA)	1	–	–
P (B08)	2	$0.9992^{+0.0136}_{-0.0122}$	$1.0000^{+0.0070}_{-0.0036}$
P (PyTorchDIA)	2	$0.9987^{+0.0134}_{-0.0121}$	$1.0004^{+0.0065}_{-0.0048}$
P (B08)	3	$1.0000^{+0.0086}_{-0.0063}$	$1.0000^{+0.0050}_{-0.0048}$
P (PyTorchDIA)	3	$0.9999^{+0.0083}_{-0.0066}$	$1.0003^{+0.0049}_{-0.0052}$
MFB (B08)	1	–	–
MFB (PyTorchDIA)	1	–	–
MFB (B08)	2	$0.0001^{+0.0002}_{-0.0001}$	$0.0000^{+0.0002}_{-0.0002}$
MFB (PyTorchDIA)	2	$0.0114^{+0.0038}_{-0.0036}$	$0.0198^{+0.0082}_{-0.0054}$
MFB (B08)	3	$0.0003^{+0.0005}_{-0.0004}$	$-0.0003^{+0.0007}_{-0.0009}$
MFB (PyTorchDIA)	3	$0.0116^{+0.0045}_{-0.0044}$	$0.0244^{+0.0098}_{-0.0067}$
MFV (B08)	1	–	–
MFV (PyTorchDIA)	1	–	–
MFV (B08)	2	$0.9649^{+0.0480}_{-0.0431}$	$1.0392^{+0.0328}_{-0.0204}$
MFV (PyTorchDIA)	2	$0.9699^{+0.0477}_{-0.0402}$	$1.0416^{+0.0331}_{-0.0209}$
MFV (B08)	3	$1.0200^{+0.0809}_{-0.0475}$	$1.1168^{+0.0901}_{-0.0561}$
MFV (PyTorchDIA)	3	$1.0253^{+0.0827}_{-0.0473}$	$1.1246^{+0.1212}_{-0.0602}$
MPB (B08)	1	–	–
MPB (PyTorchDIA)	1	–	–
MPB (B08)	2	0.112	0.415
MPB (PyTorchDIA)	2	0.090	0.456
MPB (B08)	3	0.008	0.530
MPB (PyTorchDIA)	3	−0.001	0.703
MPV (B08)	1	–	–
MPV (PyTorchDIA)	1	–	–
MPV (B08)	2	1.220	1.902
MPV (PyTorchDIA)	2	1.165	1.892
MPV (B08)	3	1.979	2.851
MPV (PyTorchDIA)	3	1.929	2.945
$N_{\rm Stamps}$ (B08)	**1**	–	–
$N_{\rm Stamps}$ (PyTorchDIA)	**1**	–	–
$N_{\rm Stamps}$ (B08)	**2**	1981/1992	57/57
$N_{\rm Stamps}$ (PyTorchDIA)	**2**	1983/1992	57/57
$N_{\rm Stamps}$ (B08)	**3**	4060/4097	843/843
$N_{\rm Stamps}$ (PyTorchDIA)	**3**	4056/4097	842/843

5 SPEED TESTS

In the speed tests in this section, we compare the performance of our GPU-accelerated numerical implementation against a fast, analytical least-squares fit solution using compiled Cython code. The PyTorchDIA code is run on Google's Colab[13] service, a free-to-use cloud-based computation environment. The instance used for these experiments was equipped with a NVIDIA Tesla K80 GPU, with 2496 CUDA cores (with 'compute capability' 3.7), with up to 16280 Mb of GPU RAM free. The Cythonized B08 solution is run

[13] https://colab.research.google.com

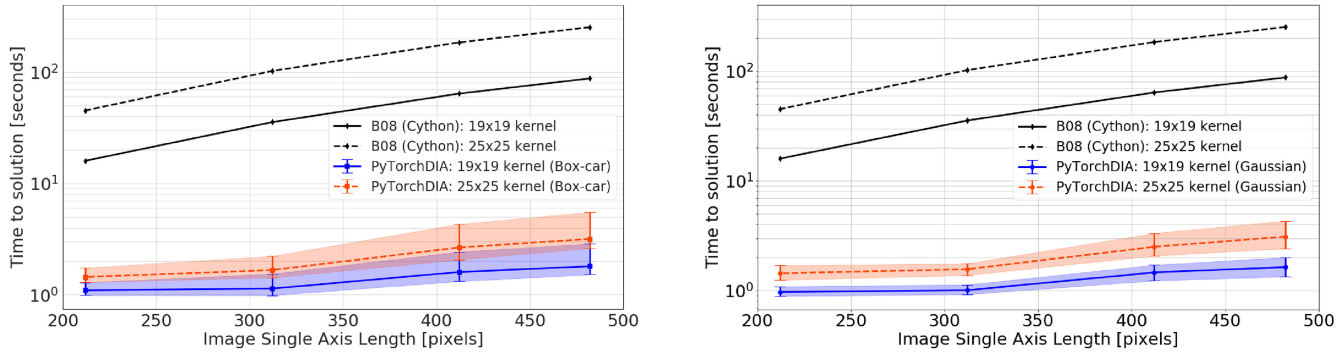

Figure 6. The time taken to solve for square kernels of different sizes against image single axis length (as a proxy for image size) for our PyTorchDIA implementation on the GPU and the B08 approach for images in a typical DK154 microlensing data set. The PyTorchDIA kernel pixels were initialized with a (left plot) 'flat', box-car kernel and (right plot) a symmetric Gaussian with width estimated as $\phi_K = \sqrt{\phi_I^2 - \phi_R^2}$.

on an Intel(R) Xeon(R) CPU @ 2.30GHz, with ∼12.6 Gb RAM available.

Having established the accuracy of our algorithm on both synthetic and real images in Sections 3 and 4, we first provide a motivating test exploring the computational speed-up over the state of the art classical approach on a set of real DK154 microlensing observations in Section 5.1. We then use a pair of large synthetic images to formally explore the scaling of our algorithm with image and kernel size in Sections 5.2 and 5.3.

5.1 Real EMCCD images

For these tests, we use a typical microlensing data set obtained with the DK154's 'red' camera during the 2019 MiNDSTEp observing season. The data set consists of 159 shift-and-added exposures, each consisting of as many as 1200 short 0.1 s exposures (i.e. ∼2 min integrations).

The sharpest image was chosen as the reference, with a $\phi_R = 1.82$ pixels. The median target image width was $\phi_I = 3.18$ pixels. All images were bias subtracted and flat corrected. The target images were then re-sampled using bicubic spline interpolation to align them with the reference. We use the flat-field used for the data reduction in our noise model (equation 7). For the B08 implementation, we perform 3 model iterations and employ sigma-clipping, masking pixels associated with $|\epsilon_{ij}| > 5$, to guard against variable sources (e.g. the gravitationally lensed target). For our PyTorchDIA implementation, we minimize the negative log-likelihood corresponding to our robust loss function (equation 13).

We test how the speed of our implementation scales with both image and kernel size by symmetrically cropping the borders of the images to different extents, and solving for both a 19 × 19 and 25 × 25 kernel. We measure the solutions times for each of the 158-large set of target images for our GPU-accelerated numerical algorithm, and plot the median solution time for a single image against the single axis length of the cropped square images (as a proxy for image size) in Fig. 6, for two different choices of parameter initialization. The 'error' bars on the median image solution times are the 16th and 84th percentiles of the distribution of the 158 target image solution times. For comparison, we plot the time taken for 3 model iterations of the B08 analytical least-squares approach. We do not plot uncertainties for the B08 solution times. The total time of this analytic approach is dominated by the normal matrix construction, which is dependent only on the size of the kernel and images, and therefore effectively consistent across all 158 target images for each image and kernel size combination.

Unsurprisingly, the massive parallelization inherent to the convolution computations in our implementation means it performs very well for even quite large image and kernel size combinations (see Section 5.2 for a formal discussion of how the algorithm scales). The median solution times for both the 19 × 19 and 25 × 25 kernels are at least an order of magnitude faster than for the B08 approach. When scaled to data sets consisting of hundreds or thousands of images, this clearly provides substantial savings in processing times.

The two plots in Fig. 6 only differ in the choice of initialization of the kernel pixels for PyTorchDIA. The plot on the left shows the solution times for a kernel initialized as a 'flat', box-car which sums to 1. The right-hand plot shows the results for kernels initialized as symmetric Gaussians, normalized to sum to 1, with shape parameter estimated as $\phi_K = \sqrt{\phi_I^2 - \phi_R^2}$. For these stacks of short EMCCD exposures, neither ϕ_I nor ϕ_R are particularly well approximated with a Gaussian, and the median solution times are at most only ∼1.1 times faster. However, the spread of the distribution of solution times is clearly tighter when a Gaussian in used for the initialization compared to the flat box-car.

To gain some insight into the cause of the differences between the spread of values for the two different kernel initializations (and therefore inform us of how to get the best performance out of PyTorchDIA), we plotted the solution times against the PSF width, ϕ_I, and SNR of the target image for the set of 482 × 482 images fit with a 19 × 19 kernel as a pairplot in Fig. 7. We quantified the correlation between solution times and these two image properties with the Spearman's rank correlation coefficient, ρ.[14] Both kernel initializations result in solution times which are anticorrelated with ϕ_I ($\rho_{\phi_I,\text{Box-car}} = -0.69$ and $\rho_{\phi_I,\text{Gaussian}} = -0.29$) and correlated with the SNR ($\rho_{\text{SNR, Box-car}} = 0.63$ and $\rho_{\text{SNR, Gaussian}} = 0.26$). We expect, however, that the cause of these trends is due only to ϕ_I and specifically, the *difference* between the width of the PSFs in the target image and reference image. That the solution time is correlated with the signal-to-noise of the target image is due to the fact that the signal-to-noise is strongly anticorrelated with ϕ_I.

The very strong anticorrelation between the solution time for the box-car kernel and ϕ_I is very likely due to this 'flat' kernel surface

[14]This provides a measure of *any* monotonic relation between two variables. The distributions (versus solution time) are clearly not linearly related, and so Pearson's correlation coefficient – which assumes a linear relation – is not suitable here.

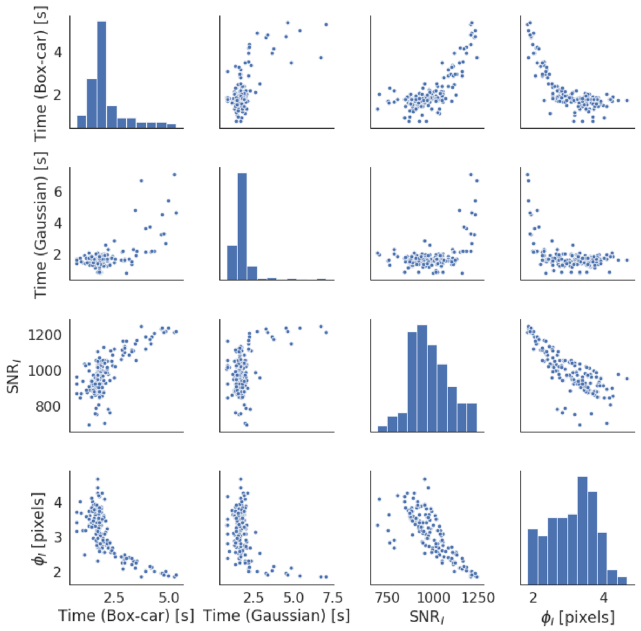

Figure 7. Pair plot showing the optimization solution times for a 19 × 19 kernel – initialized as either a flat box-car or Gaussian – on a 482 × 482 large image, against the ϕ_I and SNR of the target image. The histograms of the distributions are shown on the diagonal.

being a much better initialization for wide kernels. When the PSFs between the reference and the target images are very similar, the kernel is sharply peaked at the centre, and a flat kernel is far from the solution. Initializing with a Gaussian, although imperfect for the irregular PSFs associated with these images, appears to mitigate this problem to some degree, although the solution time still blows-up when the kernel is approximately undersampled.

5.2 Synthetic CCD images

Here, we test the performance of our implementation on synthetic images. We use our synthetic image generation procedure (see Section 3.1) to first generate a large 4000 × 4000 pixel noiseless reference image. We set the logarithm of the stellar density per 100 × 100 pixels to be 1.5, we set $\phi_R = 1$ and the sky level at 100 ADU. Fluxes are assigned randomly as described in Section 3.1. We set $\phi_K = 1.5$, and generate a target image with $\phi_I = \sqrt{\phi_R^2 + \phi_K^2}$ in the same manner as above. We then randomly add noise to the images in way described in Section 3.1, such that the target image has 10 times more pixel variance than the reference.

As there are no outlying pixels in this simulated pair of images, for the PyTorchDIA implementation, we minimize the Gaussian negative log-likelihood (equation 4). The pair of 4000 × 4000 images are symmetrically cropped to assess how the solution times scale with image size. For each kernel and image size combination (and choice of kernel initialization), we plot the solution times in Fig. 8, for kernels initialized with either a flat box-car, or symmetrical Gaussian with width equal to $\phi_K = 1.5$.

As with the speed results in the prior section, for larger images, the PyTorchDIA 19 × 19 and 25 × 25 kernel solution times are at least an order of magnitude faster than their B08 counterparts. In this case, since the correct convolution kernel really is a Gaussian, initializing the kernel as a Gaussian substantially reduces the number of optimization steps required before convergence, with these solution

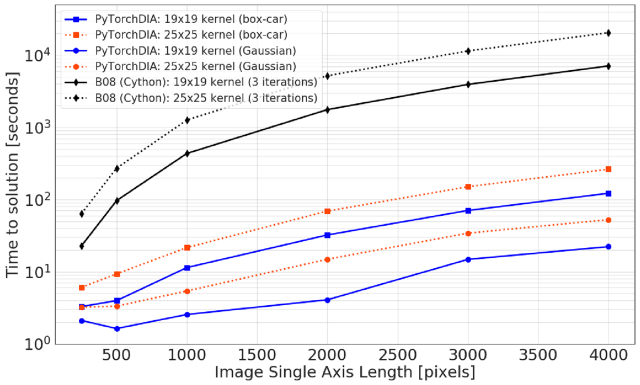

Figure 8. The time taken to solve for (square) kernels of different sizes against image single axis length (as a proxy for image size) for our PyTorchDIA implementation on the GPU and the B08 approach for a pair of square synthetic images, cropped to different sizes.

times typically being ∼4–5 faster than with a box-car initialization for the kernel. Formally, the B08 normal matrix construction time-scales as $\mathcal{O}(n^2m^4)$, where n and m are the sizes of the (square) images and kernel, respectively. The two black curves in both Figs 6 and 8 follow this scaling (i.e. $\sim(25/19)^4$). Our optimization procedure computes a direct convolution, which is predicted to scale as $\mathcal{O}(n^2m^2)$. The data points corresponding to the PyTorchDIA solution times in Fig. 8 approximately obey this rule – in both the separation of the two curves and the separation between data points on each curve – although we should expect some variation due to differences in the number of optimization steps required before converging and, potentially, cuDNN's choice of algorithm for the convolution computations, which can depend on both kernel and image size (see Section 5.3). In general, we found the number of optimization steps required for convergence to slightly decrease with increasing image size. This makes sense, in that the effective information content of the image is greater the larger it is, and so the gradient steps are more accurate. This explains why for the smallest image size in Fig. 8 the time to solve for some of the kernels takes longer than expected. Our solution time is then, not completely determined by the formal $\mathcal{O}(n^2m^2)$ scaling.

5.3 cuDNN: accelerating convolution computations on NVIDIA GPUs

One major use-case of PyTorch is for deep learning, in which CNNs, consisting of many small kernels, are used for feature detection in image recognition tasks (e.g. Lawrence et al. 1997; Krizhevsky, Sutskever & Hinton 2012). The current research interest in deep learning applications has motivated device manufacturers such as NVIDIA to develop highly tuned GPU-accelerated libraries, specifically designed to improve the performance of common processing operations (e.g. forward and backward convolution) with these small kernels. By default, PyTorch makes use of NVIDIA's cuDNN library to accelerate convolutions on NVIDIA GPUs. This library has access to both deterministic and non-deterministic algorithms to compute convolutions, which are selected heuristically to accelerate computations.[15] As the sizes of useful kernels in DIA are themselves

[15]While not explored here, cuDNN also contains tools to benchmark convolution computations for a given kernel and image size combination. When processing many images, it performs tests to assess which cuDNN algorithm

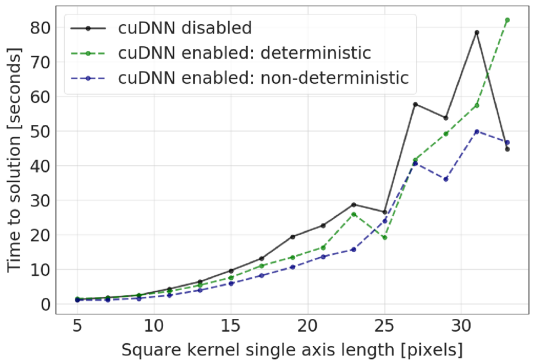

Figure 9. A comparison of the times taken to infer square kernels of different sizes with varying cuDNN settings for a given pair of 1000 × 1000 pixel synthetic images.

fairly small, it is expected that PyTorchDIA will also benefit from these tools. Indeed, we have enabled the use of non-deterministic cuDNN algorithms to accelerate the computation of convolution operations in all the results presented so far in this paper. In this subsection, we explore the benefit of this library in the context of DIA.

For this test, we measure how for an image pair of fixed sizes, the time to infer the kernel scales with kernel size. We use the pair of synthetic 4000 × 4000 pixel images from the prior section, symmetrically cropped to smaller 1000 × 1000 images. We infer the associated convolution kernel (initialized with a box-car) by minimizing the Gaussian negative log-likelihood for this pair of synthetic images, which contain no outlying pixels. The time to infer the kernel on the GPU for different cuDNN settings is shown in Fig. 9.

For all small kernels tested here, allowing cuDNN to use non-deteriministic algorithms for the convolution computations results in slightly faster inference times. There is a clear change in behaviour for kernels larger than 23 × 23 pixels, and while the non-deteriministic computations again seem to win overall, the scaling with kernel size is now far less predictable. This makes sense, as cuDNN is optimized for working with small kernels. Clearly, however, for the larger kernels, small changes in kernel size can lead to sporadic changes in performance. This will most likely be due to changes in the choice of convolution algorithm implemented by the software for any given image-kernel size combination, although a full exploration of the different CUDA convolution algorithms, how they are chosen, and how they scale, is outside the scope of this work.

6 CONCLUSIONS

We have presented a new algorithm for DIA, where we model the target image as the output of a simple CNN. The kernel is treated as a discrete pixel convolutional filter, with weights which can be fit for efficiently within the PyTorch architecture. Specifically, we make use of automatic differentiation and GPU support to perform a lightning fast optimization of the image model. We validate the fit quality and photometric accuracy of our implementation against its closest classical DIA analogue, with both simulated and real images. Our algorithm is simple to understand, and written entirely in standard Python packages, with an emphasis on accessibility to the wider astronomical community.

Its benefits over some traditional approaches can be summarized as follows:

(i) *Speed*: We exploit the massive parallelism inherent to the convolution operation by making use of highly tuned GPU-accelerated deep learning libraries to perform efficient convolution computations. With a good choice of learning rate, the optimization procedure converges rapidly, and our algorithm can perform DIA on astronomical data sets at least an order of magnitude faster than its classical analogue.

(ii) *Scalability*: For a pair of (square) images, each of size n, convolved with a (square) kernel of size m, our implementation approximately scales as $\mathcal{O}(n^2m^2)$, while the classical approach goes as $\mathcal{O}(n^2m^4)$.

(iii) *Flexibility*: In our optimization framework, we can maximize the (correct) Gaussian likelihood suitable for conventional CCD/EMCCD astronomical exposures (where the Gaussian approximation of Poissonian photon noise is valid), without resorting to an iterative procedure of χ^2 minimization. This framework also allows us to relax the Gaussian noise assumption, and optimize robust scalar objective functions for images affected by outlying pixels. This provides a justifiable alternative to procedural sigma-clipping approaches. Further, we make use of automatic differentiation tools that free the user from having to manually recompute gradients if the parametrization of the model is changed, making experimentation by users straightforward.

The current main disadvantage to this approach is the use of 32-bit numerical precision to ensure the GPU-accelerated convolution computations are performed rapidly. Also, some engineering (e.g. choice of learning rates) is required by the researcher to get the best performance. And while the $\mathcal{O}(n^2m^2)$ scaling typically holds for a given image-pair and kernel combination, our approach is a non-linear (convex) optimization, and so the solution times for our approach are not entirely deterministic. We also note that access to mid- to high-end GPUs can be an issue, and their use in both the gaming market and cryptocurrency mining has caused a surge in prices. Given this, Tensor Processing Units – which now support floating point computations – could be a viable alternative for some use-cases.

We highlight AMP training, available since PyTorch 1.6.0, as an area to explore in future work to further accelerate the convolution computations. Given the impressive recent advances in general purpose GPU computing, in large part driven by the deep learning community, we are well positioned to take advantage of improvements to these tools. Lastly, we again stress that the application to DIA is just one possible example of astronomical image processing that can benefit from deep learning tools, as very many useful image models include a convolution operation.

All code used in this work can be found at https://github.com/jah1994/PyTorchDIA.

ACKNOWLEDGEMENTS

We would like to thank the referee, Daniel Bramich, for his excellent review, which greatly improved the quality of this work. We also thank Keith Horne for his advice on specific details of our Difference Image Analysis (DIA) implementation. We also extend a thank you to the MiNDSTEp consortium for allowing us use of imaging data from the 2019 observing season. JAH gratefully acknowledges funding

performs best on the first example, caches this information, and uses this best performing algorithm on all further images in the data set. Turning this feature towards astronomical data set reduction with PyTorchDIA will be explored in future work.

from the Science and Technology Facilities Council of the United Kingdom.

DATA AVAILABILITY

The data underlying this article will be shared on reasonable request to the corresponding author.

REFERENCES

Alard C., 2000, A&AS, 144, 363
Albrow M., 2017, MichaelDAlbrow/pyDIA: Initial Release on Github,Version v1.0.0, Zenodo, doi:10.5281/zenodo.268049
Alard C., Lupton R. H., 1998, ApJS, 503, 325 (AL98)
Albrow M. et al., 2009, MNRAS, 397, 2099
Andrae R., 2010, preprint (arXiv:1009.2755)
Becker A., Homrighausen D., Connolly A., Genovese C., Owen R., Bickerton S., Lupton R., 2012, MNRAS, 425, 1341
Bellm E. C. et al., 2018, PASP, 131, 018002
Bond I. et al., 2001, MNRAS, 327, 868
Bramich D., 2008, MNRAS, 386, L77 (B08)
Bramich D., Figuera Jaimes R., Giridhar S., Arellano Ferro A., 2011, MNRAS, 413, 1275
Bramich D. et al., 2013, MNRAS, 428, 2275
Bramich D., Bachelet E., Alsubai K., Mislis D., Parley N., 2015, A&A, 577, A108
Bramich D., Horne K., Alsubai K., Bachelet E., Mislis D., Parley N., 2016, MNRAS, 457, 542 (B16)
Figuera Jaimes R., Arellano Ferro A., Bramich D., Giridhar S., Kuppuswamy K., 2013, A&A, 556, A20
Fukugita M. et al., 1996, Technical Report, The Sloan Digital Sky Survey Photometric System. SCAN-9601313
Göddeke D., Strzodka R., Turek S., 2005, Accelerating double precision FEM simulations with GPUs. Univ. Dortmund, Fachbereich Mathematik
Golub G., Van Loan C., 1996, Matrix Computations. Johns Hopkins University Press, Baltimore
Goodfellow I., Bengio Y., Courville A., Bengio Y., 2016, Deep Learning. MIT press, Cambridge
Harpsøe K. B., Jørgensen U. G., Andersen M. I., Grundahl F., 2012, A&A, 542, A23
Hartung S., Shukla H., Miller J. P., Pennypacker C., 2012, in 19th IEEE International Conference on Image Processing, IEEE, Piscataway, NJ, p. 1685
Heavens A., 2009, preprint (arXiv:0906.0664)
Huber P., Ronchetti E., 2009, in Robust Statistics, 2nd Edition, Wiley Series in Probability and Statistics. Wiley-IEEE. New York, NY, USA
Huber P. J., 1992, Breakthroughs in Statistics. Springer-Verlag, Berlin, p. 492
Ivezić Ž. et al., 2019, ApJS, 873, 111
Jordà M., Valero-Lara P., Peña A. J., 2019, IEEE Access, 7, 70461
Kains N., Bramich D., Figuera Jaimes R., Arellano Ferro A., Giridhar S., Kuppuswamy K., 2012, A&A, 548, A92
Kingma D. P., Ba J., 2015, In Proceedings of 3rd International Conference on Learning Representations, preprint (arXiv:1412.6980)
Krizhevsky A., Sutskever I., Hinton G. E., 2012, in Pereira F., Burges C. J. C., Bottou L., Weinberger K. Q., eds, Advances in Neural Information Processing Systems, Vol. 25. Curran Associates, Inc., Red Hook, New York, p. 1097
Lawrence S., Giles C. L., Tsoi A. C., Back A. D., 1997, IEEE Trans. Neural Netw., 8, 98
LeCun Y., Bengio Y., Hinton G., 2015, nature, 521, 436
Li J., Yu C., Sun J., Xiao J., 2013, in Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC). IEEE, Piscataway, NJ, p. 1937
Micikevicius P. et al., 2018, ICLR, preprint (arXiv:1710.03740)
Nocedal J., Wright S., Sequential quadratic programming, Numerical Optimization, 2006, Springer, New York, p. 529
Paszke A. et al., 2019, preprint (arXiv:1912.01703)
Perkel J. M., 2018, Nature, 563, 145
Ruder S., 2016, preprint (arXiv:1609.04747)
Sedaghat N., Mahabal A., 2018, MNRAS, 476, 5365
Skottfelt J. et al., 2015, A&A, 574, A54
Tsapras Y. et al., 2019, PASP, 131, 124401
Wozniak P., 2000, Acta Astronomica, 50, 42
Zackay B., Ofek E. O., Gal-Yam A., 2016, ApJS, 830, 27
Zhao Y., Luo Q., Wang S., Wu C., 2013, in 2013 IEEE 9th International Conference on e-Science, IEEE Computer Society, Washington, DC, p. 70

This paper has been typeset from a TEX/LATEX file prepared by the author.