An efficiency-driven approach for real-time optical flow processing on parallel hardware

Mickael Seznec, Nicolas Gac, F. Orieux, A. Sashala Naik

AN EFFICIENCY-DRIVEN APPROACH FOR REAL-TIME OPTICAL FLOW PROCESSING ON PARALLEL HARDWARE

Mickaël Seznec[*,†], *Nicolas Gac*[*], *François Orieux*[*], *Alvin Sashala Naik*[†]

[*] Univ. Paris-Saclay, CNRS, CentraleSupelec, L2S, Gif-sur-Yvette, 91192 France
[†] Thales Research and Technology, Palaiseau, 91120 France

ABSTRACT

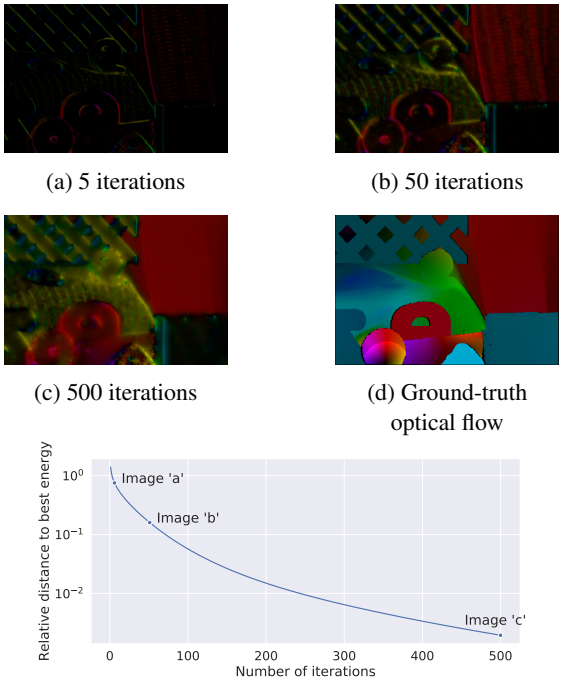
This article tackles the entire lifecycle of an algorithm: from its design to its implementation. It exhibits a method for making efficient choices at algorithm design time knowing the characteristics of the underlying hardware target. As of today, computing the optical flow of a stream of images is still a demanding task. In the meantime, the use of Graphics Processing Units (GPU) has become mainstream and allows substantial gains in processing frame rate. In this paper, we focus on a specific variational method (CLG [1]) where linear systems have to be solved. They depend on two parameters α and ρ. To efficiently solve the problem, we look at convergence speed with respect to the model's parameters. We benchmark usual linear solvers with preconditioners to identify the fastest in terms of convergence per iteration. We then show that once implemented on GPUs, the most efficient solver changes depending on the model parameters. For 640 $\times$ 480 images, with the right choice of solver and parameters, our implementation can solve the system with relative $10e^{-7}$ accuracy in 0.25 ms on a Titan V GPU. All the results are aggregated on a 30-image set to increase confidence in their extendability.

***Index Terms*—** Optical Flow, GPU, Linear Solvers, Matrix Conditioning

1. INTRODUCTION

Optical flow is the apparent movement of objects in a sequence of images. As a computer vision task, that means finding the displacement of every pixel from one image to another. Numerical methods are based on a constrained model for the pixels' movement. Early works on optical flow have been done by Lucas & Kanade[2] and Horn & Schunck[3].

The latter uses a variational regularisation: an energy penalisation is defined for a candidate flow. The goal is to find the flow that would minimise this penalisation. This method is challenging as it is not possible to have a direct solution pixel per pixel. It involves iterations over the whole image. Fig. 1 shows that many iterations are required to get a satisfying flow. Speeding-up this convergence process is thus required for real-time.



(a) 5 iterations (b) 50 iterations (c) 500 iterations (d) Ground-truth optical flow (e) Energy levels of the first three images

Fig. 1: Colour representation of optical flow evolution after an increasing number of Jacobi iterations

To tackle this issue, many recent works on optical flow leverage the massive computing power of GPUs[4, 5, 6]. Originally reserved for computer graphics and 3D-scenes rendering, they now play a prominent role in image processing.

The issue with GPU is that the performance gains dramatically depends on the algorithm. Usually, authors either directly port existing methods on GPU or design an original algorithm from scratch and with parallel platforms in mind [7]. The former approach may be sub-efficient or helpless if the algorithm is not well suited for GPUs. The latter requires demanding work to develop a whole new method.

In this article, we try to find a compromise between the two. The overall algorithm structure stays the same, but the parametrisation and the solver change, to fit GPUs better

while keeping similar results.

This article is organised as follows. In Sec. 2, a model of optical flow is presented, and we set the notations. In Sec. 3, we present some linear solvers with preconditioning techniques. In Sec. 4, we analyse the condition number of our problem depending on its parameters. Then, we use several solvers on our dataset to find the one that converges the fastest in terms of iterations. Lastly, we implement those solvers on GPU to compare their convergence against the time they take. Sec. 5 concludes about how to choose a solver and parameters with a GPU target in mind. It also aims at widening the results' scope to other problems that rely on linear solvers too.

2. OPTICAL FLOW

In the optical flow problem, we wish to find the displacement field $\bar{\boldsymbol{w}}_{x,y,t} = (u_{x,y,t}, v_{x,y,t}, 1)^T$ for every frame of size (w, h) in an image sequence. Let $f_{x,y,t}$ be the intensity of a pixel at coordinates (x, y) at time t.

In a variational method, the solution flow is the one that minimises an energy of the form

$$E(\boldsymbol{w}) = \int_\Omega D_{f,\bar{\boldsymbol{w}}}(\boldsymbol{w}_{x,y,t}) + R_{\boldsymbol{w}}(\boldsymbol{w}_{x,y,t})\,\mathrm{d}x\,\mathrm{d}y$$

where D and R are two functionals that define the optical flow model. D depends on the image sequence f and the unknown field $\bar{\boldsymbol{w}}$. R only depends on $\bar{\boldsymbol{w}}$ and plays the role of a regularisation. For clarity, we set $\boldsymbol{w} = \bar{\boldsymbol{w}}_{x,y,t}$. Horn & Schunck's choice for D and R becomes

$$D(\boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{J}_0 \boldsymbol{w},\ \boldsymbol{J}_0 = (\nabla f)(\nabla f)^T$$
$$R(\boldsymbol{w}) = \alpha(\|\nabla u\|^2 + \|\nabla v\|^2),\ \alpha \in \mathbb{R}^+$$

Bruhn et al.[1] use a version of D that takes into account the pixel's neighbourhood

$$D(\boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{J}_\rho \boldsymbol{w},\ \boldsymbol{J}_\rho = \boldsymbol{K}_\rho * \boldsymbol{J}_0$$
$$\boldsymbol{J}_\rho = \begin{bmatrix} j_\rho^{00} & j_\rho^{01} & j_\rho^{02} \\ j_\rho^{10} & j_\rho^{11} & j_\rho^{12} \\ j_\rho^{20} & j_\rho^{21} & j_\rho^{22} \end{bmatrix}$$

where K_ρ denotes a Gaussian kernel and $*$ is the convolution operator.

To minimise E, the Euler-Lagrange equations can be used to take the derivative of the integral then set it to zero. Another option is to discretise the integral form. We use $\bar{\bar{\boldsymbol{x}}} = \mathrm{flat}(\bar{\boldsymbol{x}})$, the operator that takes a 2D-field $\bar{\boldsymbol{x}}$ and reshape it into a vector $\boldsymbol{x}$ in the row-major order, and $\mathrm{diag}(\boldsymbol{x})$ that builds a diagonal matrix $\boldsymbol{X}$ that holds $\boldsymbol{x}$ in its main diagonal. Here is an example of discretising the CLG energy

$$\begin{aligned} E_{CLG}(\boldsymbol{w}) &= \int_\Omega \boldsymbol{w}^T \boldsymbol{J}_\rho \boldsymbol{w} + \alpha(\|\nabla u\|^2 + \|\nabla v\|^2)\,\mathrm{d}x\,\mathrm{d}y \\ &= \|\boldsymbol{H}\bar{\bar{\boldsymbol{w}}} - \boldsymbol{g}\|^2 + \alpha\Big(\|\boldsymbol{D}_x \boldsymbol{S}_u \bar{\bar{\boldsymbol{w}}}\|^2 + \|\boldsymbol{D}_y \boldsymbol{S}_u \bar{\bar{\boldsymbol{w}}}\|^2 + \\ &\qquad \|\boldsymbol{D}_x \boldsymbol{S}_v \bar{\bar{\boldsymbol{w}}}\|^2 + \|\boldsymbol{D}_y \boldsymbol{S}_v \bar{\bar{\boldsymbol{w}}}\|^2\Big) \end{aligned}$$

$\boldsymbol{S}_u$ and $\boldsymbol{S}_v$ are diagonal matrices that respectively select the u and v parts of $\bar{\bar{\boldsymbol{w}}}$. $\boldsymbol{D}_x$ and $\boldsymbol{D}_y$ are the derivatives along the x and y axes. To minimise this quantity, let's take the derivative with respect to $\bar{\bar{\boldsymbol{w}}}$ and set it to zero

$$\begin{aligned} \boldsymbol{H}^T\boldsymbol{H}\bar{\bar{\boldsymbol{w}}} + \alpha\Big[&\boldsymbol{S}_u^T(\boldsymbol{D}_x^T\boldsymbol{D}_x + \boldsymbol{D}_y^T\boldsymbol{D}_y)\boldsymbol{S}_u\bar{\bar{\boldsymbol{w}}} + \\ &\boldsymbol{S}_v^T(\boldsymbol{D}_x^T\boldsymbol{D}_x + \boldsymbol{D}_y^T\boldsymbol{D}_y)\boldsymbol{S}_v\bar{\bar{\boldsymbol{w}}}\Big] = -\boldsymbol{g}^T\boldsymbol{H} \end{aligned}$$

We can express it as $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, with

$$\boldsymbol{A} = \begin{bmatrix} \mathrm{diag\,flat}\,\bar{j}_\rho^{00} & \mathrm{diag\,flat}\,\bar{j}_\rho^{10} \\ \mathrm{diag\,flat}\,\bar{j}_\rho^{10} & \mathrm{diag\,flat}\,\bar{j}_\rho^{11} \end{bmatrix} - \alpha \begin{bmatrix} \boldsymbol{L} & 0 \\ 0 & \boldsymbol{L} \end{bmatrix}$$
$$\boldsymbol{b} = -\boldsymbol{g}^T\boldsymbol{H} = -\begin{bmatrix} \mathrm{flat}\,\bar{j}_\rho^{02} \\ \mathrm{flat}\,\bar{j}_\rho^{12} \end{bmatrix}$$

with $\boldsymbol{L}$ representing the Laplacian operator. It is now clear that finding the optical flow is a matter of solving a system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where $\boldsymbol{A} \in \mathbb{R}^{(2\cdot\mathrm{w}\cdot\mathrm{h})\times(2\cdot\mathrm{w}\cdot\mathrm{h})}$ is a very large and sparse matrix, $\boldsymbol{x} \in \mathbb{R}^{2\cdot\mathrm{w}\cdot\mathrm{h}}$ in the wanted flow and $\boldsymbol{b} \in \mathbb{R}^{2\cdot\mathrm{w}\cdot\mathrm{h}}$ completes the equations.

3. SOLVERS REVIEW

To solve linear systems such as $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, many methods can be used. We must, however, restrict ourselves to the ones relevant for very large and sparse $\boldsymbol{A}$. In previous works, matrix splitting methods have often been used. Horn & Schunck rely on a Jacobi-like iteration, while Jara-Wilde *et al.* use a so-called Pointwise-Coupled Gauss-Seidel in [8]. Krylov methods such as Conjugate Gradient (CG) can be found in [4]. This section will recall the foundations of these methods.

3.1. Matrix Splitting

The matrix splitting methods partition the matrix into two

$$\boldsymbol{A} = \boldsymbol{B} - \boldsymbol{C}$$

By replacing $\boldsymbol{A}$ in $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, we have

$$\boldsymbol{B}\boldsymbol{x} = \boldsymbol{b} + \boldsymbol{C}\boldsymbol{x}$$

Assuming that $\boldsymbol{B}$ is invertible, that leads to the following fixed-point iteration

$$\boldsymbol{x}^{n+1} = \boldsymbol{B}^{-1}(\boldsymbol{b} + \boldsymbol{C}\boldsymbol{x}^n)$$

The key idea is to choose $\boldsymbol{B}$ to be easily invertible. Setting $\boldsymbol{B}$ to hold the diagonal elements of $\boldsymbol{A}$ forms the Jacobi method. Using the lower or upper triangular part $\boldsymbol{A}$ is known as the Gauss-Seidel method.

These methods are generic and can be adapted to a particular problem. For optical flow, we use the upper and lower diagonal blocks for the inversion. We modify the Jacobi method to be

$$\boldsymbol{B} = \begin{bmatrix} \mathrm{diag\,flat}\,\bar{j}_\rho^{00} + 4\alpha & \mathrm{diag\,flat}\,\bar{j}_\rho^{10} \\ \mathrm{diag\,flat}\,\bar{j}_\rho^{10} & \mathrm{diag\,flat}\,\bar{j}_\rho^{11} + 4\alpha \end{bmatrix}$$

This example is what we call "preconditioned" Jacobi. This process applies to other splitting methods.

3.2. Krylov methods

Krylov solvers all emerge from the same premise: at each iteration, increasing the dimension of the space where to look for a solution. The Krylov space of order n is defined as

$$K_n(\boldsymbol{A}, \boldsymbol{b}) = \text{span}\{\boldsymbol{b}, \boldsymbol{A}\boldsymbol{b}, \boldsymbol{A}^2\boldsymbol{b}, \dots, \boldsymbol{A}^{n-1}\boldsymbol{b}\}.$$

The choice of the solution in this space leads to different methods: Conjugate Gradient (CG), minimal residual (MINRES), generalised minimal residual (GMRES), *etc*. Preconditioned methods rely on having a matrix $\boldsymbol{M}$ similar to $\boldsymbol{A}$ but easily invertible. This way, the problem becomes

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}.$$

As with splitting methods, we choose $\boldsymbol{M}$ to hold the main diagonal and the upper and lower diagonal blocks of $\boldsymbol{A}$. A preconditioner is useful when $\boldsymbol{M}^{-1}\boldsymbol{A}$ has a better conditioning than $\boldsymbol{A}$ alone.

4. RESULTS

Throughout this section, we display results aggregated on 30 images from several public datasets: Middlebury [9], MPI Sintel [10] and KITTI [11]. The error bands give the 95% confidence interval computed using the bootstrap method [12]. In Subsec. 4.3, we use a Titan V GPU from Nvidia, that we implement using CUDA/C++.

4.1. Matrix conditioning

The matrix condition number κ quantifies by how much the result of our model would change with a small perturbation in the input data. A low condition number reflects the robustness of the problem to noise and also hints that the solvers are to perform well [13].

Our optical flow model has two parameters: ρ and α. ρ is the radius parameter, it enforces a local regularisation of the flow. α ponderates the regularisation based on the gradient of the flow.

Fig. 2 shows the matrix condition number with varying parameters in the model. The results are normalised with respect to the condition number when $\alpha = 0$ and $\rho = 0$. For the condition number to be computationally tractable, the images have been downsized.

Without preconditioning, κ follows a V-shape with respect to α: increasing α is first beneficial as it adds regularisation to the problem. After a certain amount, however, it makes the problem unstable as it would be almost only determined by the regularisation and hardly by the data.

Rho can also make κ decrease, as it averages constraints over a neighbourhood, making it less sensitive to outliers.

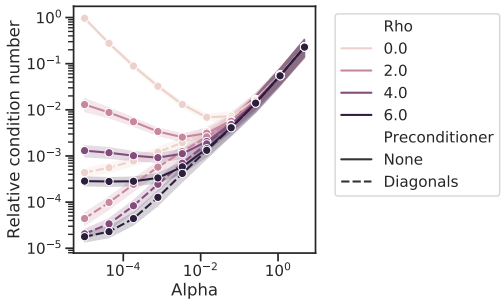


Fig. 2: Normalised condition number of the problem versus parameters' value.

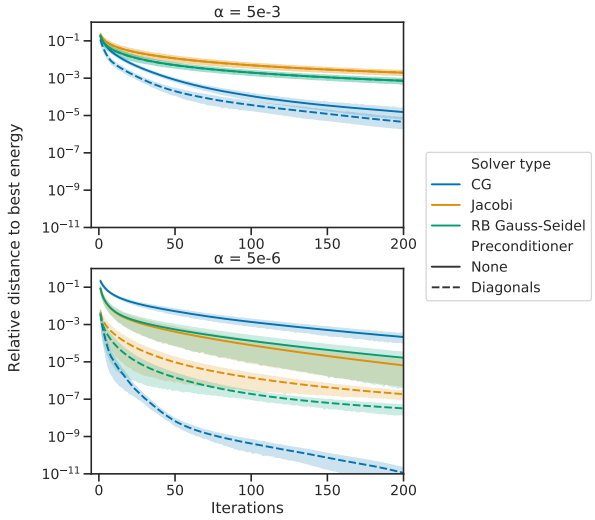


Fig. 3: Convergence vs iterations with $\rho = 2.5$. On the top $\alpha = 5e^{-3}$, on the bottom $\alpha = 5e^{-6}$

However, when alpha is already too large, ρ hardly helps counteract the ill-posedness.

The preconditioner is helpful only with a low α. This seems logical as the preconditioner we defined depends on the data term and not much on the regularisation.

With that in mind, one could tune the parameters to decrease the condition number of the matrix and thus converge as fast as possible.

4.2. Solvers' convergence speed

For Fig. 3, we chose two sets of parameters to compare the convergence of the aforementioned solvers. We tried several Krylov solvers from Python *scipy.sparse* [14] library but only reported CG as they all had similar results. Splitting methods were developed by ourselves, with Red-Black Gauss-Seidel being a more parallel version of Gauss-Seidel [15].

The results are strikingly different depending on α. When α is low, preconditioned method converges quickly (up to 10^{-11} in 200 iterations). CG is faster but splitting methods are not far behind. On the contrary, when α is higher, all

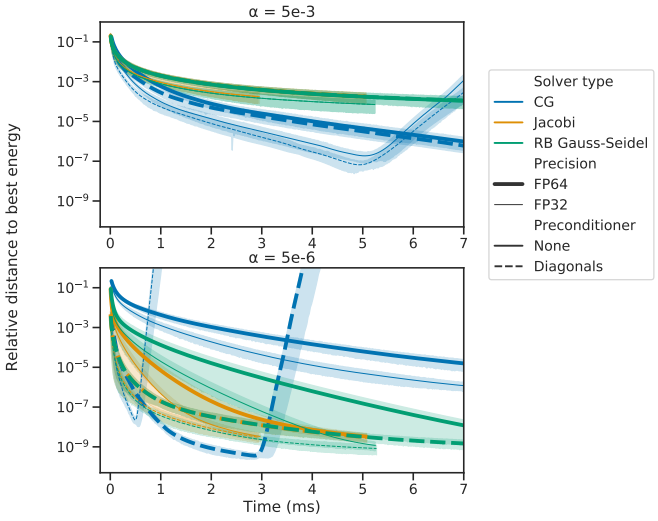


Fig. 4: Convergence vs time on GPU. Same parameters as in Fig 3.

solvers converge slowly (10^{-7} in 500 iterations) and splitting methods are even worse.

Consistently with the results found in 4.1, the effects of the preconditioner are less visible with higher α. On the top figure, preconditioned and non-preconditioned versions of splitting solvers are confounded.

4.3. Implementation Performance

The previous section has established a ranking amongst solvers but only in terms of iterations. When processing optical flow in real-time, we are rather interested in the time of convergence.

The GPU implementations for the splitting methods are straightforward: they only rely on pixel-wise operations and are hence "embarrassingly parallel". Regarding Krylov methods, they must use reductions to compute vector norms. This operation is not well adapted to GPU. We took extra care to let the reduction result stay on GPU to avoid expensive latency in CPU-GPU communication. This was done using the *CUB* library (CUDA UnBound) that provides state-of-the-art performance for GPU reductions.

Fig. 4 shows the convergence timings for different solvers on GPU with a maximum number of iterations of 1000. Globally, the curves follow the same trend as Fig. 3. However, the relative speed of solvers changes once implemented on GPU. For a low α (bottom figures), while CG goes faster theoretically, Jacobi and RB Gauss-Seidel start faster. One iteration of Jacobi is indeed faster than one iteration of CG on GPU. With a larger alpha, the results are coherent with Fig. 3. CG is the fastest. The velocity of splitting methods do not catch up with their lower theoretical convergence speed.

One caveat of CG is that it is very sensitive to its search direction, conditioned by the reduction operation. With the 32-bit floating-point (FP32) precision, it becomes unstable after a low number of iterations. FP64 makes it better, the solver diverges later, but it could still be an issue.

Globally, FP32 methods are faster than FP64, as more FP32 compute units are available on GPU. It could be interesting to use FP32 first to benefit from their speed and finish using FP64 for stability reasons.

5. CONCLUSION

In this paper, we presented the variational optical flow problem from the algorithm to its implementation. We made an in-depth exploration of the effects of the parameters on the conditioning of the problem. This way, we were able to tell which problems would be easier to solve and where the preconditioner could help.

We then reviewed the theoretical speed of solvers regarding the number of iterations and compared them to their actual performance once implemented on GPU. We showed that the splitting methods being better suited for GPUs, they overcome their theoretical weaknesses for some problem parameters. However, the performance of CG remains better on harder problems, where α is higher. This work highlights that knowing both the algorithm and the target is needed to choose the most efficient solver.

While being specific to variational optical flow, these conclusions may be useful to other problems involving linear solvers. The timings are ultimately problem-dependent. The speed of solvers, relatively to each other, and their characteristics should, however, remain constant in any setting.

In the future, our analysis could be further extended to non-quadratic models, multigrid solvers, or upon the use of embedded GPUs, such as the Nvidia Jetson devices.

6. REFERENCES

[1] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr, "Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods," *International Journal of Computer Vision*, vol. 61, no. 3, pp. 1–21, Feb. 2005.

[2] Bruce D. Lucas and Takeo Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, Vancouver, BC, Canada, 1981, IJCAI'81, pp. 674–679, Morgan Kaufmann Publishers Inc.

[3] Berthold K. P. Horn and Brian G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1, pp. 185–203, Aug. 1981.

[4] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer, "Dense Point Trajectories by GPU-Accelerated Large Displacement Optical Flow," in *Computer Vision – ECCV 2010*, vol. 6311, pp. 438–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] Clément Moussu, "GPU based real-time optical ow computation," Tech. Rep.

[6] Andrea Petreto, Arthur Hennequin, Thomas Koehler, Thomas Romera, Yohan Fargeix, Boris Gaillard, Manuel Bouyer, Quentin L Meunier, and Lionel Lacassagne, "Energy and execution time comparison of optical flow algorithms on simd and gpu architectures," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2018, pp. 25–30.

[7] Juan David Adarve Bermudez, *Real-Time Visual Flow Algorithms for Robotic Applications*, Ph.D. thesis.

[8] Jorge Jara-Wilde, Mauricio Cerda, José Delpiano, and Steffen Härtel, "An Implementation of Combined Local-Global Optical Flow," *Image Processing On Line*, vol. 5, pp. 139–158, June 2015.

[9] Simon Baker, Stefan Roth, TU Darmstadt, and Daniel Scharstein, "A Database and Evaluation Methodology for Optical Flow," p. 8.

[10] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, "A naturalistic open source movie for optical flow evaluation," in *European Conf. on Computer Vision (ECCV)*, A. Fitzgibbon et al. (Eds.), Ed. Oct. 2012, Part IV, LNCS 7577, pp. 611–625, Springer-Verlag.

[11] A Geiger, P Lenz, C Stiller, and R Urtasun, "Vision meets robotics: The KITTI dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, Sept. 2013.

[12] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesbeck, Antony Lee, and Adel Qalieh, "Mwaskom/seaborn: V0.8.1 (September 2017)," Zenodo, Sept. 2017.

[13] Jonathan Richard Shewchuk, *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*, Carnegie-Mellon University. Department of Computer Science, 1994.

[14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors, "SciPy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, 2020.

[15] Yousef Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*, SIAM, Jan. 2003.