A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning

Filip Šrajer[a] Zuzana Kukelova[b*] Andrew Fitzgibbon[c*]

[a]*Department of Computer Science, ETH Zurich, Switzerland*
[b]*Faculty of Electrical Engineering, CTU in Prague, Czech Republic*
[c]*Microsoft, Cambridge, United Kingdom*

Algorithmic differentiation (AD) allows exact computation of derivatives given only an implementation of an objective function. Although many AD tools are available, a proper and efficient implementation of AD methods is not straightforward. The existing tools are often too different to allow for a general test suite. In this paper, we compare fifteen ways of computing derivatives including eleven automatic differentiation tools implementing various methods and written in various languages (C++, F#, MATLAB, Julia and Python), two symbolic differentiation tools, finite differences, and hand-derived computation.

We look at three objective functions from computer vision and machine learning. These objectives are for the most part simple, in the sense that no iterative loops are involved, and conditional statements are encapsulated in functions such as `abs` or `logsumexp`. However, it is important for the success of algorithmic differentiation that such 'simple' objective functions are handled efficiently, as so many problems in computer vision and machine learning are of this form.

Of course, our results depend on programmer skill, and familiarity with the tools. However, we contend that this paper presents an important datapoint: a skilled programmer devoting roughly a week to each tool produced the timings we present. We have made our implementations available as open source to allow the community to replicate and update these benchmarks.

Keywords: automatic differentiation; benchmark; machine learning; computer vision

AMS Subject Classification: 65D; 68T

1. Introduction

Algorithmic differentiation (AD) is a set of methods for automatic and exact computation of derivatives given a definition, in source code, of a function to be differentiated. It includes automatic differentiation, where derivatives are forward and/or back propagated through the chain rule. This is possible since even the most complicated functions are composed of elementary operations and functions such as addition, multiplication, logarithm, exponential, *etc.*

Alternative approaches to automatic differentiation include symbolic differentiation, finite differences and differentiation by hand. Symbolic differentiation using symbolic algebra systems typically has to represent the whole function as a single expression, which is limited by available memory, meaning it cannot handle larger functions. For efficient code generation, it should also include common subexpression elimination.

Finite differences (FD) is a numerical method and therefore does not compute exact derivatives. Note however that for most computer vision an machine learning problems, this inaccuracy is often unimportant [26]. Of more importance is the computational cost: the asymptotic time complexity is dependent on the number of input variables whereas

Shorter versions of this article appeared at AD2016—7th International Conference on Algorithmic Differentiation, and in Optimization Methods and Software, Taylor and Francis, Feb 2018 (online).

*Corresponding authors. Email: kukelzuz@fel.cvut.cz, awf@fitzgibbon.ie

the complexity of so called reverse mode of AD is independent of it.

Finally, differentiating functions manually by a human is very time consuming and also error prone, but almost always results in the fastest runtime code.

As mentioned above, AD exploits the chain rule for computing derivatives. The chain rule is typically traversed either in the direction from the input variables to the output variables (forward mode) or the other way around (reverse mode). Asymptotic time complexity of forward mode is dependent on the number of input variables and complexity of reverse mode on the number of output variables. Hence, a mode should be chosen based on a function to be differentiated. Note that there are also hybrid ways of computing derivatives using AD which are not precisely forward or reverse mode. For a more detailed explanation of AD methods, see Griewank and Walther [11] and Baydin *et al.* [5].

Usually, AD is implemented by operator overloading (OO) or source transformation (ST). As an example, consider a C++ function working with floating point variables. An operator overloading tool requires that the function is written in terms of a templated type. Then, the tool instantiates the function template with a custom type which stores not only a variable but also a value of its derivative. This custom type overloads all elementary operations to also update the derivative value. Consequently, the output of the function includes the final value of the derivative. This corresponds to the forward mode. Reverse mode is sometimes considered more complicated, but the main idea is similar. On the other hand, source transformation tools analyze the original function, somewhat as a compiler would, and output source code for a function which computes the derivative. Source transformation can potentially output a code computing derivatives more efficiently than operator overloading tools but it is usually much more difficult to implement as it has to know the syntax of the desired programming language.

Many AD tools exist (see [6] and Tab. 1). Nevertheless, it is not trivial to implement one properly, especially so that it could be used for complicated objective functions. The existing tools are in various languages and implement various AD methods. Hence, most of the tools are too different to allow for a straightforward implementation of benchmark suites.

We propose to take three objective functions from machine learning and computer vision, to benchmark eleven selected AD tools covering various languages and AD methods (see Tab. 1), two symbolic differentiation tools, finite differences and also hand-derived derivative computation. The objective functions considered are: log-likelihood of a Gaussian mixture model, bundle adjustment [26], and hand tracking [24]. These functions include features such as sparse Jacobians, matrix expressions, and domain-specific special functions such as logsumexp, defined stably as

$$\text{logsumexp}(\boldsymbol{x} : \mathbb{R}^n) = \log(\text{sum}(\exp(\boldsymbol{x} - \max(\boldsymbol{x})))) + \max(\boldsymbol{x}) \tag{1}$$

Recently, Siskind and Pearlmutter [22] presented a benchmark of several AD tools. They show runtimes relative to the runtime of their own tool whereas we give absolute runtimes as well as runtimes normalized with respect to individual languages (see Sec. 5). Their objective functions are simple with a fixed number of input and output variables. On the other hand, all our problems have varying number of variables. Dürrbaum et al. [9] benchmarked ADOLC versus symbolic differentiation and found significant speed differences, also borne out by our experiments.

We first give an overview of the AD tools selected for benchmarking. Next, we briefly present how AD is used in machine learning and computer vision followed by a description of objective functions used for benchmarking in this work. Then, we present the results

and finally give our conclusions, foremost among which is that even with reasonable care devoted to efficiency in each of the input languages, the runtimes vary through four orders of magnitude. While factors other than speed are important, it should always be kept in mind that for many applications, finite difference computation is sufficiently accurate, and it is certainly the easiest to use, so any tool, to be valuable, must beat FD for speed.

1.1 *Notation*

In this paper, we use the following notation for variables: scalar s or S, vector $\boldsymbol{v}$, matrix $\mathbf{M}$, and tensor $\mathbf{T}$. We symbolize a concatenation of multiple column vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ as a matrix $\mathbf{V}$. Similarly, a concatenation of multiple matrices $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_m$ as a tensor $\mathbf{M}$.

Special functions are matrix determinant or scalar absolute value $|\cdot|$, and Euclidean norm $||\cdot||$. Function logsumexp is always defined stably as presented above.

2. Benchmarked Tools

We have chosen several well-known or promising AD tools (see Tab. 1). The selection covers various languages and AD approaches as well as symbolic differentiation. The newest version of all the tools that was available in the period July-August 2015 was used. In addition, we give results for *finite differences* and *manual*, i.e., a hand-derived optimized implementation.

The tools that have both forward and reverse mode are called with the one that is more suitable for the given objective function. Diffsharp in particular runs significantly slower in its default mode so it is called in its special forward and reverse modes for first-order derivatives.

Tapenade offers differentiation of both Fortran and clean C code but we use it only with C. Unfortunately, it does not support C fully and its source transformation occasionally produced non-compiling output, so we had to fix a few errors.

From the chosen tools, we did not benchmark ADiGator because it generated syntactically incorrect code for GMM, clad as it did not have support for arrays, and ADIC2 as our attempts to compile it were unsuccessful. Consider that this supports the statement that it is more difficult to implement a source transformation than an operator overloading tool.

Adept, ADOL-C and Ceres are all operator overloading C++ tools. They all require a templated objective function as input so that it could be run with their custom types computing derivatives. Ceres has a straightforward implementation of forward mode only. ADOL-C implements both forward and reverse modes using so called *taping* which is basically a process of storing all calculations involving active variables. Importantly, the tape can be reused for successive computations assuming that certain conditions hold. Adept is based on a similar idea but it makes use of expression templates. That makes the taping process efficient enough so that it can be run for every computation without incurring any significant slowdown.

MuPAD (called from MATLAB) optimizes code using common subexpression elimination and compiles it via C++ to MEX. Theano input needs to be written in a modified Python and is then compiled either into optimized Python or C++. Theano is always ran in CPU mode to allow a fair comparison since all the tools use only CPU.

Table 1.: List of tools. OO: operator overloading, ST: source transformation: F: forward, R: reverse.

Language	Tool	Approach	Mode
C++		Manual (by hand)	
C++		Finite differences	
C++	Adept [15]	OO	F, R
C++	ADIC2 [17]	ST	F, R
C++	ADOL-C [28]	OO	F, R
C++	Ceres Solver [1]	OO	F
C++	clad [27]	ST via compiler	F
C/Fortran	Tapenade [14]	ST	F, R
F#	DiffSharp [5]	OO	F, R
MATLAB	ADiGator [18]	ST via OO	F
MATLAB	ADiMat [7]	OO via ST	F, R
MATLAB	MuPAD [25]	Symbolic	
Julia	ForwardDiff.jl [19]	OO	F
Python	Autograd [16]	OO	F
Python	Theano [4]	Symbolic	

3. Automatic Differentiation in Computer Vision and Machine Learning

Problems in computer vision and machine learning are often formulated as non-linear optimization. Some of these problems are neural network training, bundle adjustment, clustering or tracking, to name a few. Optimization algorithms typically require derivatives in the form of gradients, Jacobians, or Hessians. Therefore, AD methods can be applied in these fields. They can prove very useful, especially during prototyping, as the objective function may be changed as often as the programmer wishes without putting any effort into derivative-computation implementation and still get exact derivatives. Nonetheless, AD methods are still not widely known in the machine learning and computer vision community.

In the cases, where the community applies AD or AD-like techniques, specialized tools are typically employed instead of existing general AD implementations. This also motivates our benchmark to see how they compare. These specialized tools are Ceres [1], Autograd [16], and Theano [4], for example. Ceres implements a simple forward mode AD in C++, Autograd is a reverse mode implementation for Python, and Theano is a collection of symbolic and AD-like differentiation methods using its own syntax based on Python.

Another related technique, used for training neural networks, is the backpropagation algorithm, essentially a special case of reverse-mode AD. For a more comprehensive survey of AD in machine learning, see Baydin *et al*. [5].

4. Objective Functions

In this section, we present the three objective functions used for benchmarking AD tools. The functions are: log-likelihood of a Gaussian mixture model, bundle adjustment, and hand tracking.

4.1 *Objective GMM: Gaussian Mixture Model Fitting*

The Gaussian mixture model can be used in a wide range of applications. Consider clustering, deblurring of images [31] and speech recognition [30] for instance. The GMM

has likelihood function

$$p(\mathbf{X};\boldsymbol{w},\mathbf{M},\boldsymbol{\Sigma}) = \prod_{i=1}^{N}\sum_{k=1}^{K} w_k |2\pi\boldsymbol{\Sigma}_k|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}_i-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}_i-\boldsymbol{\mu}_k)\right)$$
$$\text{s.t.} \sum_{k=1}^{K} w_k = 1 \text{ and } \boldsymbol{\Sigma}_k \text{ is positive-definite } \forall k \in \{1,\ldots,K\} \tag{2}$$

where variables $\boldsymbol{x}_i \in \mathbb{R}^D$ are data points, $w_k \in \mathbb{R}$ weights, $\boldsymbol{\mu}_k \in \mathbb{R}^D$ means, and $\boldsymbol{\Sigma}_k \in \mathbb{R}^{D\times D}$ covariance matrices. Function inputs $\mathbf{X}, \boldsymbol{w}, \mathbf{M}$, and $\boldsymbol{\Sigma}$ are their concatenations as explained in Sec. 1.1.

We parametrize the positive-definite covariance matrices by the square roots of their inverses. We introduce variables $\boldsymbol{q}_k \in \mathbb{R}^D$ and $\boldsymbol{l}_k \in \mathbb{R}^{\frac{D(D-1)}{2}}$ and function $Q(\boldsymbol{q},\boldsymbol{l})$ which assembles a $D \times D$ lower triangular matrix in the following way

$$Q(\boldsymbol{q},\boldsymbol{l}) = \begin{bmatrix} \exp(q_1) & 0 & \cdots & 0 \\ l_1 & \exp(q_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{D-1} & l_{D-1+D-2} & \cdots & \exp(q_D) \end{bmatrix}. \tag{3}$$

from which we assemble $\boldsymbol{\Sigma}^{-1} = Q(\boldsymbol{q},\boldsymbol{l})Q(\boldsymbol{q},\boldsymbol{l})^\top$.
Positive weights w_k are parameterized by log-parameters $\alpha_k \in \mathbb{R}$:

$$w_k = \frac{\exp(\alpha_k)}{\sum_{k'=1}^{K}\exp(\alpha_{k'})}. \tag{4}$$

In addition, we include an Identity-Wishart prior over the covariances

$$p(\boldsymbol{\Sigma}) = \prod_{k=1}^{K} C(D,m)|\boldsymbol{\Sigma}_k|^m \exp\left(-\frac{1}{2}\operatorname{trace}(\boldsymbol{\Sigma}_k)\right) \tag{5}$$

where variable m is a Wishart prior hyperparameter and C is a function not dependent on independent variables.

The goal of GMM inference is to maximise the posterior probability of data given parameters, or equivalently to minimize the negative log posterior

$$L(\boldsymbol{w},\mathbf{M},\boldsymbol{\Sigma};\mathbf{X}) = -\log\big(p(\mathbf{X};\boldsymbol{w},\mathbf{M},\boldsymbol{\Sigma})p(\boldsymbol{\Sigma})\big)$$

Discarding function C and simplifying using the described parametrization, the final

function to be optimized looks like

$$
\begin{aligned}
L(\boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}, \mathbf{L}) = & \sum_{i=1}^{N} \operatorname{logsumexp}\left(\left[\alpha_k + \operatorname{sum}(\boldsymbol{q}_k) - \frac{1}{2}||Q(\boldsymbol{q}_k, \boldsymbol{l}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)||^2\right]_{k=1}^{K}\right) \\
& - N \operatorname{logsumexp}\left([\alpha_k]_{k=1}^{K}\right) \\
& + \frac{1}{2}\sum_{k=1}^{K}\left(||\exp(\boldsymbol{q}_k)||^2 + ||\boldsymbol{l}_k||^2\right) - m \operatorname{sum}(\boldsymbol{q}_k)
\end{aligned} \tag{6}
$$

We benchmark the AD tools on gradient computation of Eq. (6). The size of the gradient changes with D and K, while $\boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}$ and $\mathbf{L}$ are independent variables.

Note that it is possible to implement the first line of Eq. (6) using large matrix operations provided that enough memory is available. This can significantly speed up some languages and tools (see Sec. 5). The main idea is to work with all the data at once instead of using an outer loop over the data. For instance, we can compute

$$
Q(\boldsymbol{q}_k, \boldsymbol{l}_k)\left(\mathbf{X} - \begin{bmatrix}\boldsymbol{\mu}_k & \boldsymbol{\mu}_k & \dots & \boldsymbol{\mu}_k\end{bmatrix}\right) \tag{7}
$$

at the cost of $O(ND)$ words of storage.

4.2 *Objective BA: Bundle Adjustment*

In computer vision, 3D reconstruction is a widely studied problem [3, 23]. Given a visual input (*e.g.* images or video) observing the same scene, the goal is to reconstruct a 3D model of this scene. Even though the creation of 3D models can be a goal on its own, 3D reconstruction is necessary for a number of other applications such as localization [20], robot navigation [8], augmented reality or virtual reality [21].

Consider so called sparse 3D reconstruction. In this problem, given only images we want to find 3D coordinates of some points observed in the images together with parameters of cameras for the images, *i.e.*, where the cameras were in the world when images were taken. That can be done by various approaches but most of them run an optimization procedure called bundle adjustment (BA) [1, 26]. This procedure optimizes simultaneously all the parameters, *i.e.*, all 3D point coordinates and parameters of cameras. We benchmark the AD tools by computing the Jacobian used in BA.

Let us first introduce the projection function for one camera and one point. Consider a weight $w \in \mathbb{R}$, a 3D point $\boldsymbol{x} \in \mathbb{R}^3$ and a camera with parameters $\boldsymbol{p} = [\boldsymbol{r}; \boldsymbol{c}; f; \boldsymbol{x}_0; \boldsymbol{\kappa}] \in \mathbb{R}^{11}$, i.e., rotation, camera center, focal length, principal point and radial distortion. The point $\boldsymbol{x}$ can be projected by the camera as

$$
\operatorname{project}(\boldsymbol{p}, \boldsymbol{x}) = \operatorname{distort}(\boldsymbol{\kappa}, \operatorname{p2e}(\operatorname{rodrigues}(\boldsymbol{r}, \boldsymbol{x} - \boldsymbol{c})))f + \boldsymbol{x}_0 \tag{8}
$$

where

$$
\operatorname{distort}(\boldsymbol{\kappa}, \boldsymbol{u}) = \boldsymbol{u}(1 + \kappa_1||\boldsymbol{u}||^2 + \kappa_2||\boldsymbol{u}||^4) \tag{9}
$$

$$
\operatorname{p2e}(\boldsymbol{x}) = \frac{\boldsymbol{x}_{1:2}}{x_3} \tag{10}
$$

$$
\operatorname{rodrigues}(\boldsymbol{r}, \boldsymbol{x}) = \boldsymbol{x}\cos\theta + (\boldsymbol{v} \times \boldsymbol{x})\sin\theta + \boldsymbol{v}(\boldsymbol{v}^\top \boldsymbol{x})(1 - \cos\theta), \quad \theta = ||\boldsymbol{r}||, \boldsymbol{v} = \frac{\boldsymbol{r}}{||\boldsymbol{r}||} \tag{11}
$$

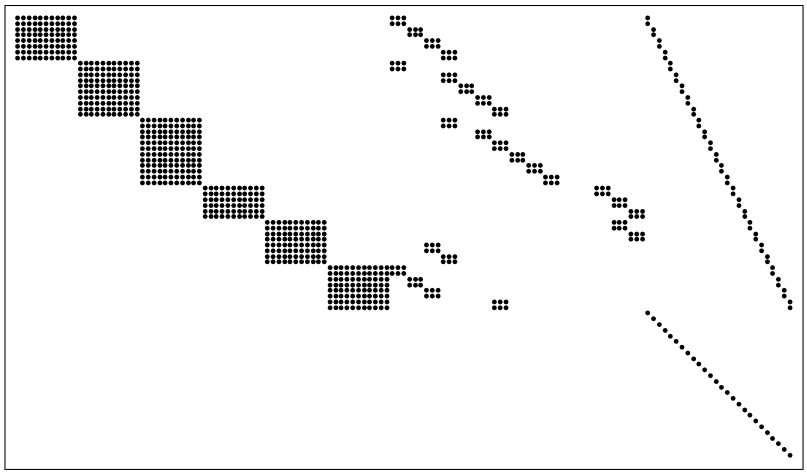

Figure 1.: Sparsity pattern of the Jacobian for an example instance of bundle adjustment. The first set of wider blocks corresponds to camera parameters, the middle set to 3D points, and the last set to weights. Rows have been permuted so the "weights-only" rows appear after all the "reprojection error" rows.

The observed image point is $\boldsymbol{m} \in \mathbb{R}^2$ and the residual $\boldsymbol{e}$ concatenates its reprojection error [13] and w's regularizer

$$\boldsymbol{e} = [w(\boldsymbol{m} - \text{project}(\boldsymbol{r}, \boldsymbol{c}, f, \boldsymbol{x}_0, \boldsymbol{\kappa}, \boldsymbol{x}))^\top; 1 - w^2]^\top \tag{12}$$

The Jacobian of the whole system where multiple cameras observe multiple points has a special form. It has only 15 non-zero entries in every reprojection-error row and one non-zero in every weight-term row. See Fig. 1 for a visualization. Importantly, every residual is independent of others. It is thus possible to compute small (3×15) dense Jacobians corresponding to individual residuals by directly differentiating the residual function (see Eq. (12)). Then, it is straightforward to distribute the entries across the final sparse Jacobian. Hence, AD tools are not required to support sparsity in any way in order to compute the Jacobian of this problem. This strategy is applied also in the popular optimizer Ceres, that is quite often used to solve BA problem in computer vision [1]. Also note that because of the sparsity, the width of the Jacobian is not important and time complexity depends only on the number of observations.

4.3 *Objective HT: Hand Tracking*

In hand tracking [24], we are given a model of a hand and a stream from a depth sensor. The goal is tracking a real hand observed by the depth sensor, *i.e.*, fitting the model to the depth information. An application requiring hand tracking is remote control and interaction [29], for instance.

For benchmark purposes, let us consider only the optimization part of the hand tracking problem. We are given the hand model aligned to the previous frame. The model is a set of points $\mathbf{X} \in \mathbb{R}^{3\times M}$ and their triangulation, i.e., a collection of adjacent triangles, which make up a surface. The motion of the model is parametrized by the variable $\boldsymbol{p} \in \mathbb{R}^{26}$. Then, we are given N correspondences between the triangles and measured 3D points $\mathbf{Y} \in \mathbb{R}^{3\times N}$ obtained from the current depth frame. The variable $\mathbf{U} \in \mathbb{R}^{2\times N}$ are barycentric coordinates defining exact spots of correspondence inside the triangles.

Additionally, we are given weights $\mathbf{W} \in \mathbb{R}^{22\times M}$ defining which points lie on which parts of the hand (see the procedure below).

The variable $\boldsymbol{p}$ contains 3 parameters for global translation, 3 for global rotation parametrized using angle-axis representation and 4 angles for every finger.

The procedure for computing the error for all measurements is based on linear blend skinning:

(1) Use the finger parameters to assemble 22 transformations $\mathbf{T} \in \mathbb{R}^{4\times4\times22}$ corresponding to parts of hand. This operation first assembles individual independent relative transformations corresponding to joints using the Euler angles approach and then hierarchically combines them to the absolute transformations $\mathbf{T}$.

(2) Transform all model vertices by all transformations and weight by those that are relevant, *i.e.*,

$$\mathbf{Z} = \sum_{i=1}^{22} \mathbf{T}_i \begin{bmatrix} \bar{\boldsymbol{x}}_1^i & \bar{\boldsymbol{x}}_2^i & \dots & \bar{\boldsymbol{x}}_M^i \end{bmatrix} \in \mathbb{R}^{4\times M}, \quad \bar{\boldsymbol{x}}_j^i = w_{i,j} \begin{bmatrix} \boldsymbol{x}_j \\ 1 \end{bmatrix} \in \mathbb{R}^4 \tag{13}$$

(3) Apply global rotation and translation

$$\mathbf{V} = \begin{bmatrix} \mathbf{R} & \boldsymbol{t} \end{bmatrix} \mathbf{Z} \in \mathbb{R}^{3\times M} \tag{14}$$

Note that we can take 3×4 matrix because all $\mathbf{T}_i$ are composed of rotation and translation only and weights for every point sum up to one. Therefore all $\boldsymbol{z}_j$ have the fourth coordinate equal to one.

(4) Having transformed the hand model, find the exact correspondence spots inside the triangles. For q-th measurement corresponding to the triangle (i, j, k):

$$\boldsymbol{y}'_q = u_{q,1}\boldsymbol{v}_i + u_{q,2}\boldsymbol{v}_j + (1 - u_{q,1} - u_{q,2})\boldsymbol{v}_k \tag{15}$$

which gives us $\mathbf{Y}' \in \mathbb{R}^{3\times M}$.

(5) Finally, the errors for all points are simply $\mathbf{E} = \mathbf{Y} - \mathbf{Y}'$.

The independent variables are $\boldsymbol{p}$ and $\mathbf{U}$. We benchmark the Jacobian computation which has a special structure. It has a semi-dense mostly unstructured part composed of columns of $\boldsymbol{p}$ and a sparse part corresponding to $\mathbf{U}$, where every row has two non-zero entries. See Fig. 2 for a visualization. In contrast to BA (see Sec. 4.2), it is not possible to compute individual blocks of the Jacobian independently. Therefore, sparsity has to be exploited differently for efficient Jacobian computation.

One has to create a seed matrix which defines the compression, feed it to an AD tool and decompress the resulting matrix. Having the sparsity pattern, it is possible to compute a seed matrix automatically using ColPack [10], for example. Nevertheless, we propose to exploit the properties of the HT problem and design the seed matrix manually. The sparsity pattern of the (left) semi-dense part of the HT Jacobian can change in every iteration. Therefore, we propose to treat the left part as a dense Jacobian in order to avoid seed matrix computation cost. The number of columns of the left part is constant. Hence, the AD tools will always need the same number of function passes. The sparsity pattern of the (right) part is of a diagonal structure and does not change. It is straightforward to create a seed matrix which compresses the pattern of the right side into two columns.

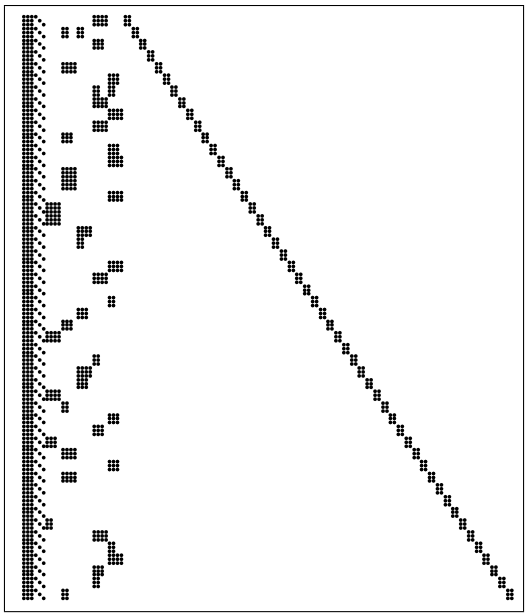

Figure 2.: Sparsity pattern of the Jacobian for an example instance of hand tracking. The left part corresponds to motion parameters and the diagonal part on the right to barycentric coordinates.

5. Experiments

To benchmark the AD tools, we first ran pre-processing routines (e.g. source transformation, symbolic differentiation, taping). All of the routines that need to be run only once for different data are not included in the runtimes that we provide. This is justified since a user of AD tools would typically run it only once on the objective before calling the differentiated function many times to optimize parameters.

The benchmarking is done on random data. The resulting runtimes are averaged over 1000 runs if one run is less than 5 seconds, over 100 runs if 5-30 seconds and over 10 runs if 30-120 seconds. Otherwise, the runtimes are not averaged. The time limit for a single run is 40k seconds. A single machine with a processor Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz, memory 32GB and OS Windows 10 64-bit was used for all the experiments.

We measure not only derivative-computation runtimes for every differentiation approach but also objective-computation runtimes for every language. Hence, we are able to show derivative runtimes for every approach relative to objective runtimes. Note that this measure attempts to minimize the dependence of results on individual languages. Throughout this section, relative runtimes refer to absolute derivative-computation runtimes divided by absolute objective-computation runtimes measured in a corresponding language. Special case are the symbolic differentiation tools Theano and MuPAD. For them, we record runtimes of objective computation which is optimized by their internal engines.

Note that visualizations and tables with results for absolute derivative-computation runtimes are provided in the supplementary material.

Experiment: Gaussian Mixture Model (GMM)

Fig. 3 shows gradient-computation runtimes for GMM with 10k data points. Alternatively, see Tab. 2 for a subset of the results. We have noticed that tools Adept and ADOL-C do not handle bigger instances. DiffSharp and Autograd crash even for smaller instances. The biggest instance size ($D = 64$, $K = 200$) was taken from Zoran and Weiss [31]. We help out these tools by manually exploiting partial separability by splitting the gradient computation into functions f, applied per datapoint, and g, the parts independent of datapoints

$$\nabla L(\boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}, \mathbf{L}) = \sum_{i=1}^{N} \nabla f(\boldsymbol{x}_i; \boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}, \mathbf{L}) + \nabla g(\boldsymbol{\alpha}, \mathbf{Q}, \mathbf{L}) \tag{16}$$

which is symbolized by *(split)* in the figures. This way, the tools are able to handle even the larger problem instances even though they require a lot of memory.

Moreover, GMM allows for an opposite approach to (split), a vectorized implementation (denoted by *(vector)*), where most necessary computations are done in one huge matrix multiplication (see Sec. 4.1). We show this (vector) version with languages that are able to utilize it. Notice how Theano and ADiMat are boosted by (vector).

Note that MuPAD is the only tool having problems with compilation. It could not compile for larger problem sizes and it could take up to several hours to compile the others. Next, we point out that Ceres and Julia-ForwardDiff have forward mode only and as can be seen, not having a reverse mode really puts them in a severe disadvantage, especially as the problem size grows. The same holds for finite differences.

The relative runtimes for most of the tools fall in the range of two orders of magnitude. Interestingly, some tools perform very differently for different problem sizes. Taking ADiMat (vector), for instance, one can see that its relative runtime for the smallest problem size is much higher than for the largest one. We can only reason that it cannot utilize MATLAB's strength of matrix operations so much for the smaller data.

Comparing standard and (split) versions of Adept, ADOL-C and Autograd, we observe a drop in runtime for all these tools when the (split) version is used. We argue that this is caused by multiple invocations of the taping process instead of just one. This claim is supported by the measured runtime difference between standard and (split) versions of ADOL-C and Adept. Both tools are written in C++ and use similar ideas but Adept employs efficient expression templates for taping. Hence, multiple invocations of the taping process do not incur a significant slowdown as opposed to ADOL-C.

We have also tried running the tools with 2.5M data points which is a number reported to be used in [31]. With so many points, no tool could handle the biggest problem sizes without manual exploitation of partial separability. Implementations utilizing large matrix operations (denoted by (vector)) did not work at all as they need too much memory and cannot exploit partial separability by definition.

Table 2.: Absolute runtimes for GMM with 10k data points. The bullet symbolizes that a tool crashed and no entry means that a tool did not finish in the time limit.

# parameters		3.00e+1	3.30e+2	1.20e+3	3.30e+3	1.07e+4	2.15e+4	5.36e+4	4.29e+5
Manual	C++	2.96e−3	1.12e−2	1.04e−1	1.11e−1	3.59e−1	7.90e−1	2.08	2.32e+1
Finite differences	C++	6.07e−2	1.58	7.72e+1	1.42e+2	8.64e+2	3.23e+3	2.08e+4	
Adept	C++	1.70e−2	9.61e−2	5.12e−1	9.76e−1	3.11	6.24	1.80e+1	•
Adept (split)	C++	2.86e−2	1.65e−1	8.54e−1	1.57	4.15	7.03	2.00e+1	1.48e+2
ADOLC	C++	3.08e−2	8.79e−2	8.84e−1	8.49e−1	1.90	•	•	•
ADOLC (split)	C++	4.71e−1	8.22e−1	3.58	4.17	1.01e+1	1.97e+1	4.45e+1	8.66e+2
Ceres	C++	5.80e−2	7.85	1.46e+2	8.65e+2	•	•	•	•
Tapenade	C	7.21e−3	3.35e−2	2.61e−1	3.68e−1	1.08	2.24	6.29	5.25e+1
DiffSharp (split)	F#	1.81e−1	9.36e−1	8.22	1.14e+1	4.64e+1	1.96e+2	6.13e+2	8.53e+3
ADiMat	MATLAB	4.16e+1	4.24e+1	1.36e+3	3.59e+2	4.25e+1	7.75e+1	1.77e+2	1.43e+3
ADiMat (vector)	MATLAB	2.53e−1	2.73e−1	1.49	6.77e−1	4.75e−1	7.39e−1	1.50	1.10e+1
MuPAD (split)	MATLAB	4.64e−3	3.66e−2	2.38e−1	5.06e−1	•	•	•	•
Julia-F	Julia	4.28e−1	1.29e+1	1.53e+2	8.42e+2	1.19e+4			
Julia-F (vector)	Julia	5.83e−1	1.93e+1	•	•	•	•	•	•
Autograd	Python	5.76e+1	•	•	•	•	•	•	•
Autograd (split)	Python	9.07e+1	7.82e+2	3.30e+3	8.22e+3	•	•	•	•
Theano	Python	1.11e+1	1.52e+1	2.99e+2	6.53e+1	1.88e+1	4.26e+1	8.00e+1	6.58e+2
Theano (vector)	Python	1.82e−2	5.38e−2	8.01e−1	5.64e−1	9.22e−1	2.03	5.03	•

Table 3.: Absolute runtimes for GMM with 2.5M data points. The bullet symbolizes that a tool crashed and no entry means that a tool did not finish in the time limit. Only tools that could compute at least one problem instance are shown.

# parameters		3.00e+1	3.30e+2	1.20e+3	3.30e+3	1.07e+4	2.15e+4	5.36e+4	4.29e+5
Manual	C++	8.43e−1	3.29	2.85e+1	3.01e+1	7.65e+1	3.80e+2	3.89e+2	6.16e+3
Finite differences	C++	1.25e+1	3.54e+2	1.76e+4	3.31e+4				
Adept	C++	3.61	•	•	•	•	•	•	•
Adept (split)	C++	5.32	3.50e+1	1.66e+2	3.72e+2	7.86e+2	2.31e+3	4.09e+3	3.99e+4
ADOLC (split)	C++	9.83e+1	1.77e+2	7.91e+2	9.88e+2	2.32e+3	4.83e+3	1.04e+4	
Ceres	C++	1.59e+1	2.27e+3	3.26e+4					
Tapenade	C	1.60	8.58	6.68e+1	8.56e+1	•	•	•	•
Tapenade (split)	C	3.92	1.33e+1	7.97e+1	1.09e+2	2.68e+2	9.59e+2	1.32e+3	1.59e+4
DiffSharp (split)	F#	4.35e+1	2.44e+2	1.94e+3	3.34e+3	3.19e+4			
MuPAD (split)	MATLAB	1.45	1.09e+1	•	•	•	•	•	•
Julia-F	Julia	9.86e+1	2.59e+3						
Julia-F (vector)	Julia	1.03e+3	•	•	•	•	•	•	•
Autograd (split)	Python	2.35e+4							
Theano	Python	3.23e+3	2.79e+3	•	•	•	•	•	•
Theano (vector)	Python	5.48	•	•	•	•	•	•	•

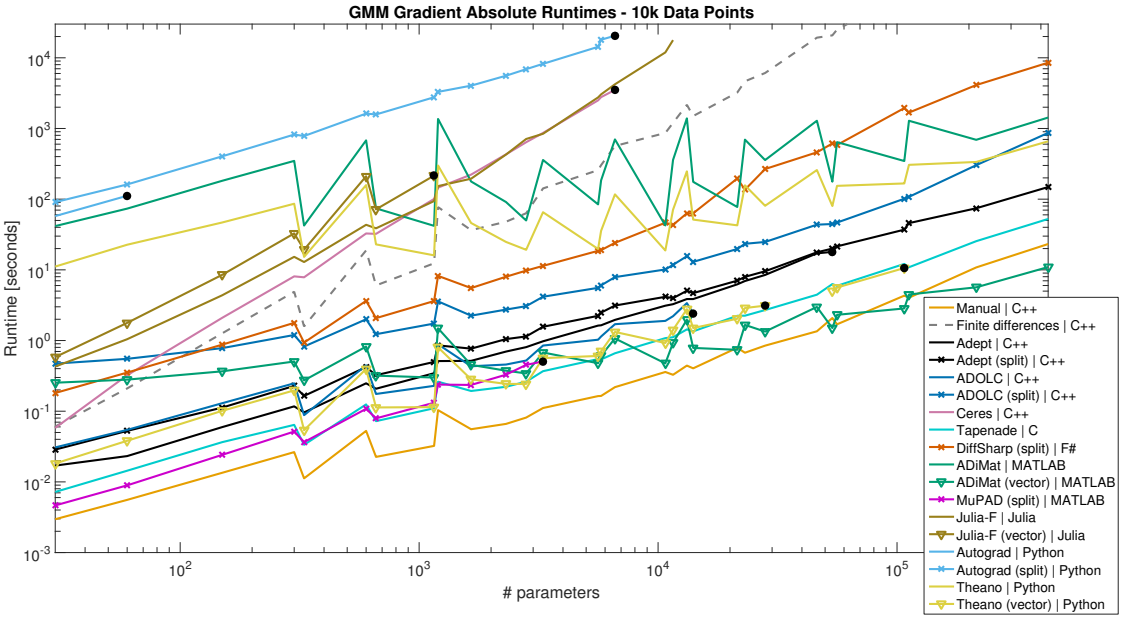

Figure 3.: Absolute runtimes for GMM with 10k data points. Some of the tools were run with (split) or (vector) implementations (see Sec. 5). The curve endings emphasized by the black dots symbolize that the tools crashed on bigger instances and those not emphasized did not finish in our time limit. Note that both axes are log-scaled. Best viewed in color.

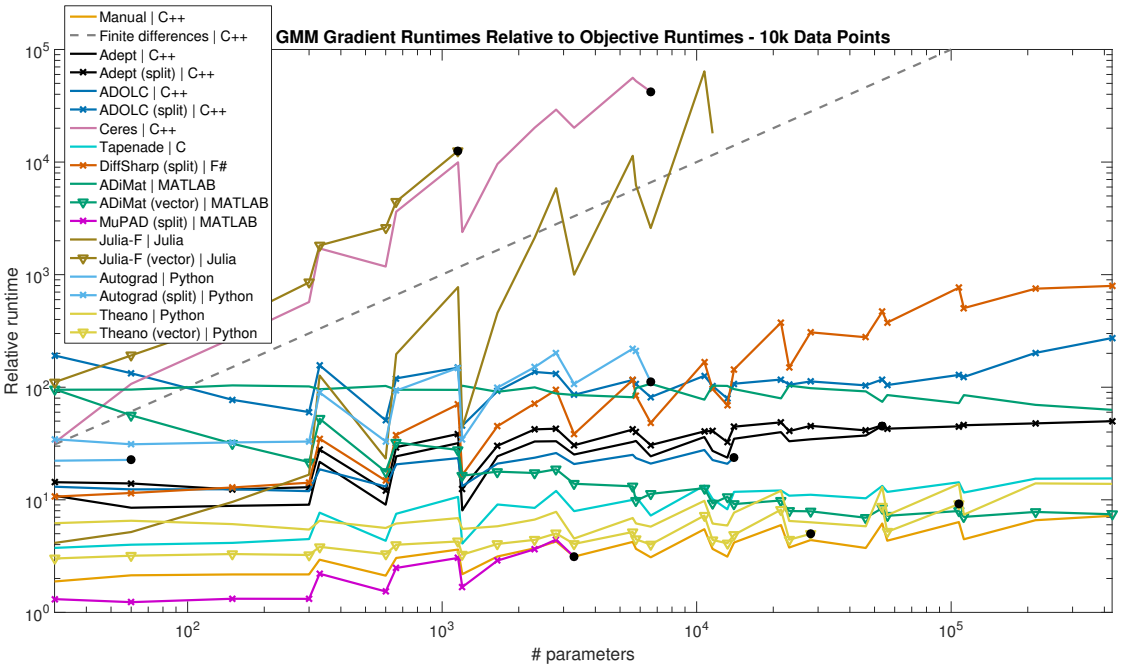

Figure 4.: Relative runtimes for GMM with 10k data points. Some of the tools were run with (split) or (vector) implementations (see Sec. 5). The curve endings emphasized by the black dots symbolize that the tools crashed on bigger instances and those not emphasized did not finish in our time limit. Note that both axes are log-scaled. Best viewed in color.

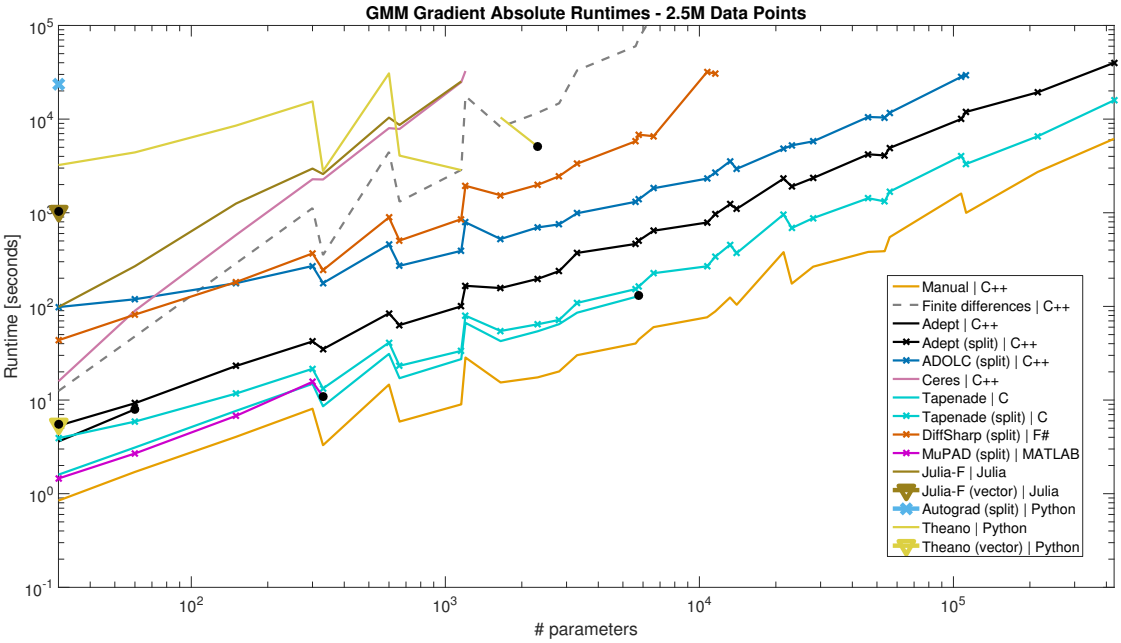

Figure 5.: Absolute runtimes for GMM with 2.5M data points. Some of the tools were run with (split) or (vector) implementations (see Sec. 5). The curve endings emphasized by the black dots symbolize that the tools crashed on bigger instances and those not emphasized did not finish in our time limit. Only tools that could compute at least one problem instance are shown. Note that both axes are log-scaled. Best viewed in color.

Experiment: Bundle Adjustment (BA)

Next, we show Jacobian computation runtimes for BA in Fig. 6 and Tab. 4. We have chosen various problem sizes ranging from 21 cameras, 11k 3D points and 36k observations to 14k cameras, 4M 3D points, 29M observations. The problem sizes are samples of real-world dataset sizes [2].

The more suitable mode for BA is reverse (see Sec. 4.2). Nevertheless, by comparing Ceres and ADOL-C, for instance, we can deduce that choosing either forward or reverse mode does not have so large significance in this case. ADiMat and Theano give inferior absolute runtimes as opposed to GMM with 10k data points, where they could utilize vectorization in the large matrix multiplication. Nevertheless, their relative runtimes are comparable to the other tools. Further notice that MuPAD is as good as manual implementation of the derivative computation. The reason for that is the use of common subexpression elimination and compilation into C++.

Table 4.: Absolute runtimes for BA. Note that Eigen matrix library [12] was utilized for implementing hand-derived derivatives. The bullet symbolizes that a tool crashed and no entry means that a tool did not finish in the time limit.

# measurements		3.18e+4	2.04e+5	2.87e+5	5.64e+5	1.09e+6	4.75e+6	9.13e+6	2.90e+7
Manual	C++	1.96e−2	1.32e−1	1.76e−1	3.26e−1	6.32e−1	2.85	5.58	1.62e+1
Finite differences	C++	4.25e−2	2.77e−1	3.85e−1	7.66e−1	1.48	6.41	1.27e+1	3.96e+1
Adept	C++	6.79e−2	4.38e−1	6.28e−1	1.21	2.38	1.03e+1	2.03e+1	6.63e+1
ADOLC	C++	8.50e−1	5.25	7.68	1.45e+1	2.99e+1	1.25e+2	2.16e+2	7.09e+2
Ceres	C++	2.26e−1	1.62	2.30	4.63	9.11	4.85e+1	1.12e+2	•
Tapenade	C	2.43e−2	1.55e−1	2.18e−1	4.30e−1	8.26e−1	3.67	7.09	2.27e+1
DiffSharp	F#	5.37e−1	3.52	4.79	8.98	1.68e+1	7.32e+1	1.46e+2	4.39e+2
ADiMat	MATLAB	5.54e+2	3.60e+3	6.01e+3	1.10e+4				
MuPAD	MATLAB	2.69e−2	1.20e−1	1.66e−1	3.36e−1	6.25e−1	2.66	5.20	1.65e+1
Julia-F	Julia	1.34	9.51	1.22e+1	2.61e+1	5.10e+1	1.77e+2	3.52e+2	1.19e+3
Autograd	Python	1.73e+2	1.00e+3	1.48e+3	2.67e+3	5.32e+3	•	•	•
Theano	Python	1.81e+1	1.18e+2	1.64e+2	3.00e+2	5.92e+2	•	•	•

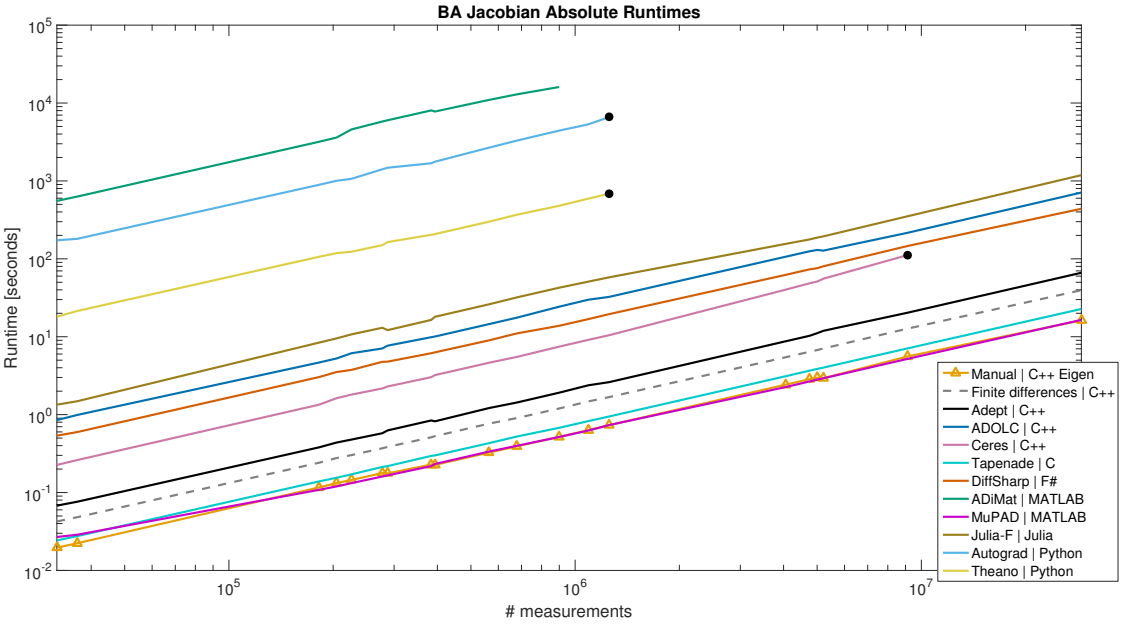

Figure 6.: Absolute runtimes for BA. Note that Eigen matrix library [12] was utilized for implementing hand-derived derivatives. The curve endings emphasized by the black dots symbolize that the tools crashed on bigger instances and those not emphasized did not finish in our time limit. Note that both axes are log-scaled. Best viewed in color.

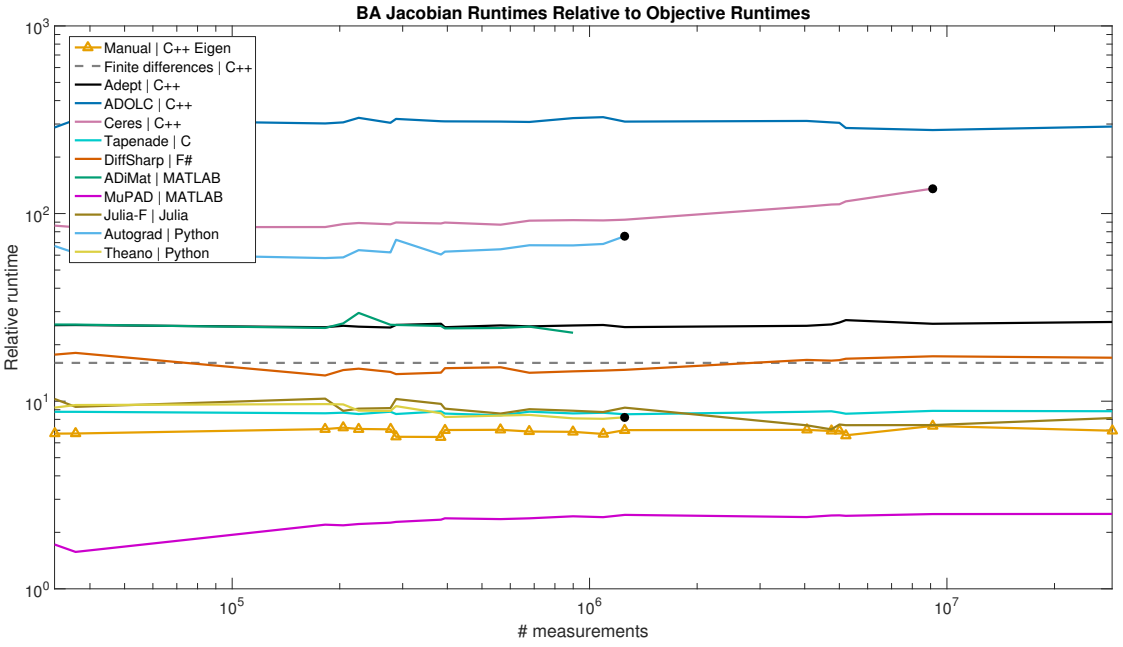

Figure 7.: Relative runtimes for BA. Note that Eigen matrix library [12] was utilized for implementing hand-derived derivatives. The curve endings emphasized by the black dots symbolize that the tools crashed on bigger instances and those not emphasized did not finish in our time limit. Note that both axes are log-scaled. Best viewed in color.

Experiment: Hand Tracking (HT)

For HT, we have chosen a small model size suitable for a real-time application and a larger one which would be typically run offline. The small instance has 544 points on the hand model and 192 correspondences whereas the big one has 10k points and 100k correspondences. We give Jacobian-computation runtimes for varying number of correspondences for the small model in Fig. 9. The results for the large model are visualized in Fig. 11.

Several tools were not benchmarked on HT. MuPAD had compilation issues. Julia and Ceres did not allow for use of a custom seed matrix. Tapenade was not benchmarked because the objective contains a lot of matrix operations which would have to be implemented in clean C. That is surely possible but consider that Tapenade does not support C fully and manually fixing the generated errors would require a significant effort. Finally, Autograd was not benchmarked as it implements only reverse mode.

The objective function in C++ was implemented in two different ways. One is using the Eigen matrix library [12] and the other using a custom lightweight matrix class (denoted in the figures by *light*). We use the custom class only with Adept because it is not compatible with Eigen and ADOL-C because Eigen is not optimized for the adouble class of ADOL-C. As can be seen, ADOL-C gives almost an order of magnitude worse results for the Eigen implementation than for the custom matrix class in terms of relative runtime of Jacobian-computation. Further note that Theano's AD-like mode called R-op is used for HT. Its standard symbolic mode would not handle sparsity.

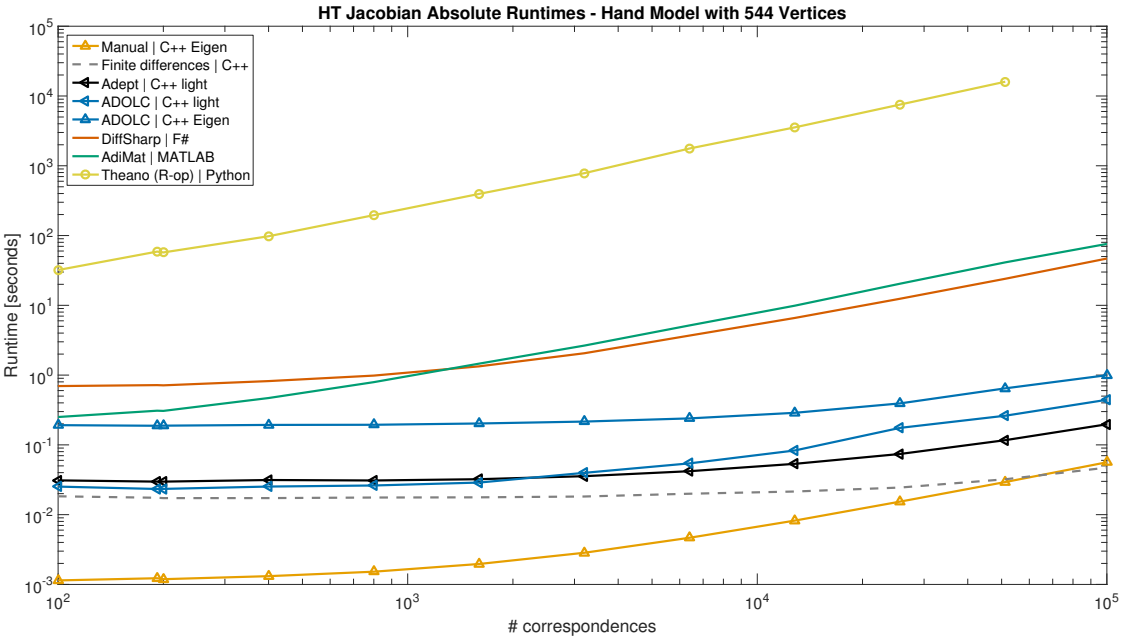

Figure 8.: Absolute runtimes for HT with the smaller hand model. Only some tools were benchmarked (see Sec. 5). Theano did not finish in our time limit for the largest number of correspondences. Note that both axes are log-scaled. Best viewed in color.

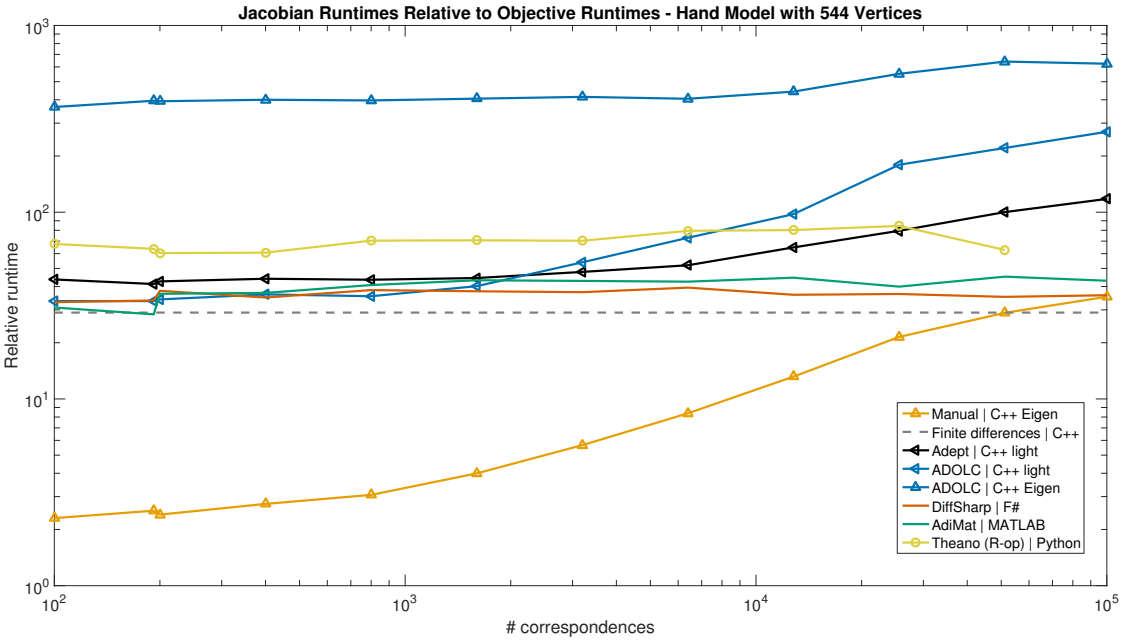

Figure 9.: Relative runtimes for HT with the smaller hand model. Only some tools were benchmarked (see Sec. 5). Theano did not finish in our time limit for the largest number of correspondences. Note that both axes are log-scaled. Best viewed in color.

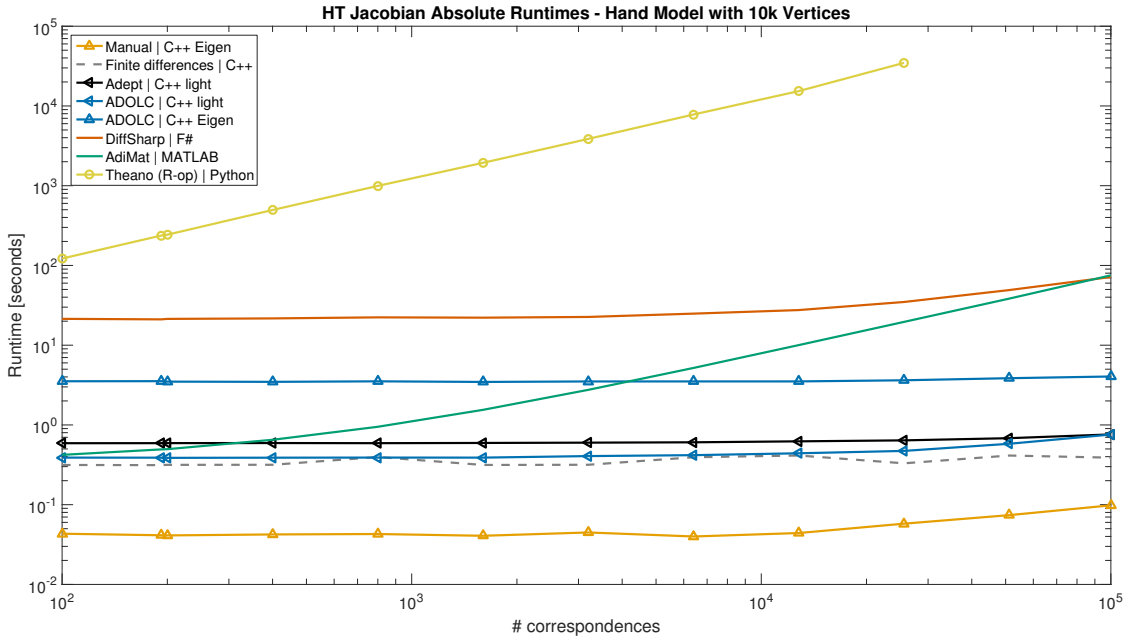

Figure 10.: Absolute runtimes for HT with the larger hand model. Only some tools were benchmarked (see Sec. 5). Theano did not finish in our time limit for the two largest numbers of correspondences. Note that both axes are log-scaled. Best viewed in color.

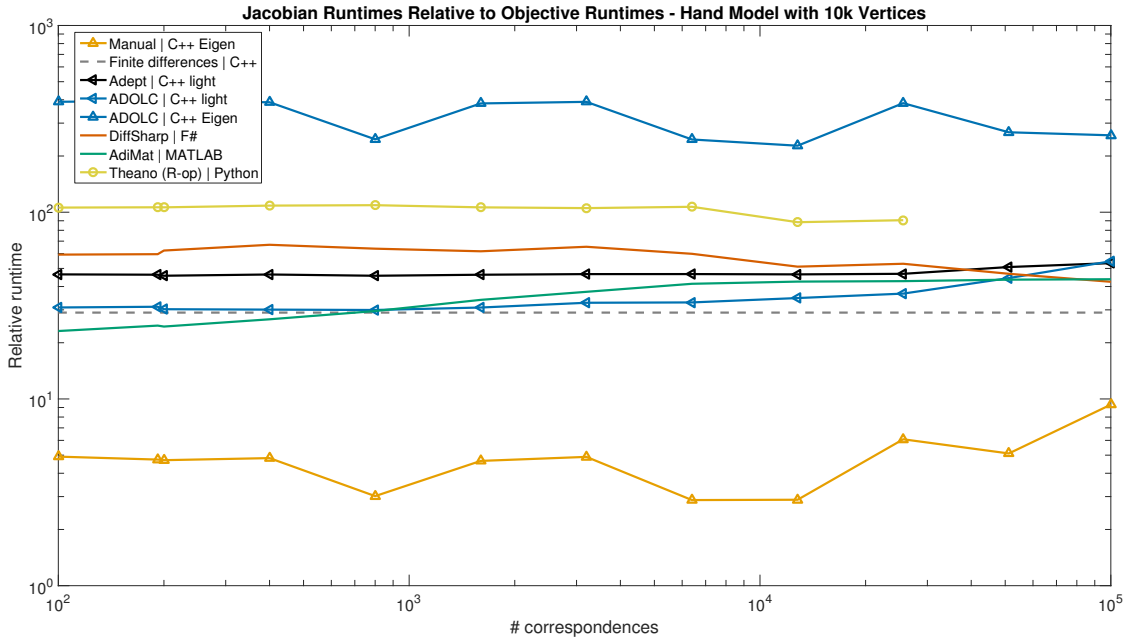

Figure 11.: Relative runtimes for HT with the larger hand model. Only some tools were benchmarked (see Sec. 5). Theano did not finish in our time limit for the two largest numbers of correspondences. Note that both axes are log-scaled. Best viewed in color.

6. Conclusion

First, we have introduced automatic differentiation and chosen several tools for computing derivatives to be benchmarked. Second, we have pointed out the significance of derivatives in machine learning and computer vision and subsequently described three real-world objective functions from these areas. Then, we have provided relative runtimes for computing derivatives.

We have seen that the relative runtimes of derivative computation range through three orders of magnitude. The relative runtime minimizes the effect of a programming language. Nevertheless, the runtime will still depend on programmer skill, and familiarity with the tools, so we have made open source all our materials[1], in order that others may improve on our efforts. However, we contend that this paper presents an important datapoint: a skilled programmer devoting roughly a week to each tool produced the timings above. For many projects, these will represent typical results achieved before a tool is selected.

We conclude that there are useful tools in most languages but there is also still some space for improvement. Availability of various features proves to be crucial for the success and efficiency of algorithmic differentiation. Important features for our objectives include ability to use a custom seed matrix, support of matrix libraries, partial separability detection, and memory optimizations for big problem instances. Moreover, using the more suitable mode (forward or reverse) can really make a difference, especially for large problems. Therefore, availability of both modes in the AD tools is an advantage. Importantly, note that we benchmarked only computation of the first-order derivatives and some tools do not support higher-order derivatives.

Acknowledgements

This work was done while the first author was an intern at Microsoft Research. We thank Jonathan Taylor for an example implementation of a hand tracking function in Python.

Funding

Zuzana Kukelova was supported by The Czech Science Foundation Project GACR P103/12/G084.

References

[1] S. Agarwal, K. Mierle, and Others, *Ceres Solver*, `http://ceres-solver.org`.
[2] S. Agarwal, N. Snavely, S.M. Seitz, and R. Szeliski, *Bundle Adjustment in the Large*, in *ECCV 2010*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 29–42.
[3] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S.M. Seitz, and R. Szeliski, *Building rome in a day*, Commun. ACM 54 (2011), pp. 105–112.
[4] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I.J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, *Theano: new features and speed improvements*, DLUFL NIPS Workshop (2012).
[5] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, and J.M. Siskind, *Automatic differentiation in machine learning: a survey*, arXiv preprint arXiv:1502.05767 (2015).

[1] https://github.com/awf/autodiff

[6] M. Bckler, P. Hovland, C. Wente, and H. Bach, *Community portal for automatic differentiation*, `http://www.autodiff.org/`.

[7] C.H. Bischof, H.M. Bücker, B. Lang, A. Rasch, and A. Vehreschild, *Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs*, in *SCAM*, 2002, pp. 65–72.

[8] A.J. Davison, I.D. Reid, N.D. Molton, and O. Stasse, *Monoslam: Real-time single camera slam*, IEEE transactions on pattern analysis and machine intelligence 29 (2007).

[9] A. Dürrbaum, W. Klier, and H. Hahn, *Comparison of automatic and symbolic differentiation in mathematical modeling and computer simulation of rigid-body systems*, Multibody System Dynamics 7 (2002), pp. 331–355.

[10] A.H. Gebremedhin, D. Nguyen, M.M.A. Patwary, and A. Pothen, *Colpack: Software for graph coloring and related problems in scientific computing*, ACM Trans. Math. Softw. 40 (2013), pp. 1:1–1:31.

[11] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM, 2008.

[12] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org (2010).

[13] R.I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2004.

[14] L. Hascoët and V. Pascual, *The Tapenade automatic differentiation tool: Principles, model, and specification*, ACM TOMS 39 (2013), pp. 20:1–20:43.

[15] R.J. Hogan, *Fast reverse-mode automatic differentiation using expression templates in C++*, ACM TOMS 40 (2014), pp. 26:1–26:24.

[16] D. Maclaurin, D. Duvenaud, and R.P. Adams, *Autograd: Effortless gradients in numpy*, in *ICML 2015 AutoML Workshop*, 2015.

[17] S.H.K. Narayanan, B. Norris, and B. Winnicka, *ADIC2: Development of a component source transformation system for differentiating c and c++*, Procedia Computer Science 1 (2010), pp. 1845 – 1853.

[18] M.A. Patterson, M. Weinstein, and A.V. Rao, *An efficient overloaded method for computing derivatives of mathematical functions in matlab*, ACM Trans. Math. Softw. 39 (2013), pp. 17:1–17:36.

[19] J. Revels, M. Lubin, and T. Papamarkou, *Forward-mode automatic differentiation in julia*, arXiv:1607.07892 [cs.MS] (2016).

[20] T. Sattler, B. Leibe, and L. Kobbelt, *Towards Fast Image-Based Localization on a City-Scale*, in *Outdoor and Large-Scale Real-World Scene Analysis: 15th International Workshop on Theoretical Foundations of Computer Vision*, Springer Berlin Heidelberg, 2012, pp. 191–211.

[21] L. Shapira and D. Freedman, *Reality Skins: Creating Immersive and Tactile Virtual Environments*, in *Mixed and Augmented Reality (ISMAR), IEEE International Symposium on*, 2016, pp. 115–124.

[22] J.M. Siskind and B.A. Pearlmutter, *Efficient Implementation of a Higher-Order Language with Built-In AD*, in *AD2016: Programme and Abstracts*, 2016.

[23] N. Snavely, S.M. Seitz, and R. Szeliski, *Photo tourism: Exploring photo collections in 3D*, in *SIGGRAPH*, 2006, pp. 835–846.

[24] J. Taylor, R. Stebbing, V. Ramakrishna, C. Keskin, J. Shotton, S. Izadi, A. Hertzmann, and A. Fitzgibbon, *User-Specific Hand Modeling from Monocular Depth Sequences*, in *CVPR*, 2014.

[25] The MathWorks, Inc., *Mupad*, MATLAB Symbolic Math Toolbox.

[26] B. Triggs, P.F. McLauchlan, R.I. Hartley, and A.W. Fitzgibbon, *Bundle Adjustment - A Modern Synthesis*, in *Proc. of the Intl. Workshop on Vision Algorithms: Theory and Practice*, ICCV, 2000, pp. 298–372.

[27] V. Vassilev, V. Ilieva, L. Moneta, A. Penev, and M. Vassilev, *clad*, `https://github.com/vgvassilev/clad`.

[28] A. Walther and A. Griewank, *Getting started with ADOL-C*, in *Combinatorial Scientific Computing*, chap. 7, Chapman-Hall CRC Computational Science, 2012, pp. 181–202.

[29] E. Wood, J. Taylor, J. Fogarty, A. Fitzgibbon, and J. Shotton, *ShadowHands: High-Fidelity Remote Hand Gesture Visualization Using a Hand Tracker*, in *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, Niagara Falls, Ontario, Canada, ACM, 2016, pp. 77–84.

[30] D. Yu and L. Deng, *Automatic speech recognition: A deep learning approach*, Springer, 2014.

[31] D. Zoran and Y. Weiss, *From learning models of natural image patches to whole image restoration*, in *ICCV*, Nov, 2011, pp. 479–486.