Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 54-9 (2021) 310–316

Computing Contraction Metrics: Comparison of Different Implementations

P. Giesl *, **S. Hafstein** **,[1], **I. Mehrabinezhad** ***

* *Department of Mathematics, University of Sussex, Falmer BN1 9QH, United Kingdom, (e-mail: p.a.giesl@sussex.ac.uk).*
** *Faculty of Physical Sciences, University of Iceland, Dunhagi 5, IS-107 Reykjavik, Iceland, (e-mail: shafstein@hi.is)*
*** *Faculty of Physical Sciences, University of Iceland, Dunhagi 5, IS-107 Reykjavik, Iceland, (e-mail: imehrabinzhad@hi.is)*

Abstract: We discuss the implementation of a novel demanding algorithm to compute contraction metrics for nonlinear systems. We compare MATLAB- and C++-implementations and study the effect of parallelizing the code. Further, we explore the use of different low-level math-libraries for the C++-implementation and the use of an undocumented code-path that considerably speeds up MATLAB code on AMD's Ryzen processors.

Copyright © 2021 The Authors. This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)

Keywords: Contraction metric, Riemannian metric, Lyapunov function, Numerical algorithm, Mesh-free collocation

1. INTRODUCTION

In this paper we compare different implementations of a numerical algorithm to compute contraction metrics for dynamical systems introduced in Giesl et al. (2019). In the method a system specific Riemannian metric is computed, which can be used to determine the existence of an exponentially stable equilibrium and its basin of attraction. The numerical computation consists of two parts: first, mesh-free collocation based on radial basis functions is used to approximate a particular contraction metric as the solution to a matrix-valued partial differential equation (PDE). Then, the values of the approximation at the vertices of a given triangulation are used to compute a continuous piecewise affine interpolation and a number of conditions are checked to verify that the interpolated function is indeed a contraction metric. In the paper, we first give a short description of the numerical method and then we will discuss various programming aspects of achieving the computations. In particular, we will discuss and compare MATLAB-, including the use of MEX files, and C++-based implementations and investigate the effect of using parallelization. Further, in the C++ implementation, we explore the running times on our AMD Ryzen processor (2700X, 8 cores, 3.7 GHz) using different low-level math-libraries. More exactly, we study the difference between using OpenBLAS, as e.g. used by the Julia programming language[2], and the Intel Math Kernel Library (MKL), as e.g. used by MATLAB and the open source software Scilab[3]. In the latter case, we additionally apply a fast codepath recently published online that speeds up computations considerably on AMD processors.

Let us give an overview of the paper: the mathematical problem and the numerical method will be presented in Section 2. In Section 3, we will describe the programming approaches and some technical information about the software and software packages we used. In Section 4, we compare the results of the different implementation approaches, in particular with regards to the running time. Finally, in Section 5, we compare and summarize different aspects of the approaches and give some conclusions.

2. NUMERICAL METHOD TO COMPUTE A CONTRACTION METRIC

In this section, we will briefly introduce the concept of a contraction metric and a numerical algorithm to compute it. Since the main focus of this report is on the computation methods, we have skipped many details, but they can be found in Giesl et al. (2019).

We consider a general *ordinary differential equation* (ODE) of the form

$$\dot{x} = f(x), \tag{2.1}$$

which defines a dynamical system. Here, $f\colon \mathbb{R}^n \to \mathbb{R}^n$ is a smooth function and $x \in \mathbb{R}^n$. We are interested in the determination of the basin of attraction of an equilibrium. An equilibrium is a point $x_0 \in \mathbb{R}^n$ with $f(x_0) = 0$, i.e. solutions starting at this point will remain at the point for all future times. Its basin of attraction consists of all initial conditions such that the corresponding solutions converge to x_0 as time tends to infinity. One way of determining the basin of attraction, which is valid even if one considers a perturbed system, is to compute a contraction metric for the system. A contraction metric is a particular kind of a Riemannian metric, i.e. a matrix-valued function $M\colon \mathbb{R}^n \to \mathbb{R}^{n\times n}$, such that $M(x)$ is a positive definite, symmetric matrix for all $x \in \mathbb{R}^n$ and thus defines a point-dependent scalar product through $\langle v, w\rangle_M := v^T M(x)w$ for $v, w \in \mathbb{R}^n$. It is called a

[1] Hafstein's research is partially supported by the Icelandic Research Fund (Rannís), grant number 163074-052, Complete Lyapunov functions: Efficient numerical computation.
[2] https://julialang.org/
[3] https://www.scilab.org/

2405-8963 Copyright © 2021 The Authors. This is an open access article under the CC BY-NC-ND license.
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2021.06.088

contraction metric for the system (2.1), if the distance with respect to the metric between adjacent solutions contracts over time. The contraction property can be expressed by the negative definiteness of

$$F(M)(x) := M(x)Df(x) + Df(x)^T M(x) + M'(x), \quad (2.2)$$

where $M'(x)$ denotes the orbital derivative of $M(x)$, i.e. the derivative along solutions of the ODE, and is defined by $M'_{ij}(x) := \nabla M_{ij}(x) \cdot f(x)$, for $i, j \in \{1, \ldots, n\}$, see Giesl (2015) for more details.

Our numerical algorithm to construct such a contraction metric for system (2.1) works in two steps. In the first step, we characterize the contraction metric for the system as the solution to a matrix-valued PDE, $F(M)(x) = -I$. Numerically, we approximate the solution by fixing the value of (2.2) at a finite number of collocation points and then compute the norm-minimal interpolation (the optimal recovery problem); in practice this is done by fixing a *radial basis function* (RBF) as well as the collocation points, and then solving a system of linear equations, see Giesl and Wendland (2019). We refer to this approach as the *RBF method* and to the numerical solution as the *optimal recovery*; more details are presented in Subsection 2.2. We have used a hexagonal grid (see Iske (1998)) for the collocation points and a Wendland function as radial basis function (see Wendland (2005)).

In the second step, we create a triangulation using a set of simplices $\mathcal{T} = \bigcup_\nu \mathfrak{S}_\nu$ and interpolate the values of the optimal recovery at its vertices by a *continuous piecewise affine* (CPA) function $P\colon \mathbb{R}^n \to \mathbb{R}^{n\times n}$. This is done because we can verify that the CPA function is a contraction metric by checking the following four constraints for $P(x_k)$, where x_k is a vertex, and the variables $C_\nu, D_\nu \in \mathbb{R}^+$ for each simplex $\mathfrak{S}_\nu$:

(VP1) For each vertex x_k of the triangulation, $P(x_k)$ is a positive definite matrix.

(VP2) For each simplex $\mathfrak{S}_\nu$ and each vertex x_k of $\mathfrak{S}_\nu$, C_ν is an upper bound on $\|P(x_k)\|_2$.

(VP3) For each simplex $\mathfrak{S}_\nu$ and all $1 \le i \le j \le n$, D_ν is an upper bound on the gradient of the continuous affine functions $P_{ij}\big|_{\mathfrak{S}_\nu^\circ}$ on the simplex, i.e.

$$\left\| \nabla P_{ij}\big|_{\mathfrak{S}_\nu^\circ} \right\|_1 \le D_\nu.$$

(VP4) For each simplex $\mathfrak{S}_\nu \in \mathcal{T}$ and each vertex x_k of $\mathfrak{S}_\nu$, we have that

$$A_\nu(x_k) + h_\nu^2 E_\nu I$$

is negative definite. Here

$$A_\nu(x_k) := P(x_k)Df(x_k) + Df(x_k)^T P(x_k) + (\nabla P_{ij}\big|_{\mathfrak{S}_\nu^\circ} \cdot f(x_k))_{i,j=1,2,\ldots,n}, \quad (2.3)$$

where $Df(x_k)$ is the Jacobian matrix of f at x_k, and $(\nabla P_{ij}\big|_{\mathfrak{S}_\nu^\circ} \cdot f(x_k))_{i,j=1,2,\ldots,n}$ denotes the symmetric $(n \times n)$-matrix with entries $\nabla P_{ij}\big|_{\mathfrak{S}_\nu^\circ} \cdot f(x_k)$ Further, h_ν denotes the *diameter* of the simplex $\mathfrak{S}_\nu \in \mathcal{T}$,

$$h_\nu := \operatorname{diam}(\mathfrak{S}_\nu) = \max_{x,y \in \mathfrak{S}_\nu} \|x - y\|_2$$

and

$$E_\nu := n^2(1 + 4\sqrt{n}) B_\nu D_\nu + 2\, n^3 B_{3,\nu} C_\nu,$$

where B_ν, and $B_{3,\nu}$ are, respectively, upper bounds on the second-order and third-order derivatives of the components of f on simplex $\mathfrak{S}_\nu$; for details see Giesl et al. (2019).

If the constraints are fulfilled, then the CPA function $P(x)$ is a contraction metric for the system (2.1). A similar approach has been used for computing Lyapunov functions Giesl and Hafstein (2015) for nonlinear systems.

Fixing a compact subset of the basin of attraction, it is shown in Giesl et al. (2019) that the algorithm will succeed in constructing a contraction metric if both the collocation points are sufficiently dense and the triangulation is sufficiently fine. Therefore, the idea is to increase the number of collocation points and simplices gradually until we obtain a contraction metric.

2.1 The Example Considered

In this paper we consider the following example of an ODE for illustration,

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -K_d\, x_2 - x_1 - g x_1^2 \left(\dfrac{x_2}{K_d} + x_1 + 1 \right) \end{cases} \quad (2.4)$$

with $K_d = 1$ and $g = 6$. It models a speed control system and has two asymptotically stable equilibria at $(0, 0)$ and $(-0.7887, 0)$, and a saddle at $(-0.2113, 0)$. The system fails to reach the demanded speed, which corresponds to the equilibrium at $(0, 0)$, for some inputs since the basin of attraction of $x_0 = (0, 0)$ is not the whole phase space, see (Giesl, 2007, Section 6.1) for more details.

For the speed control system, we used $N = 546$ collocation points around $(0, 0)$ as a hexagonal grid with parameter $\alpha = 0.030$ (see Iske (1998)) inside the following area

$$\Big\{(x, y) \in \mathbb{R}^2 \setminus (0, 0) : -0.18 \le y \le 0.85, \quad -2.11x - 0.3 \le y \le -1.79\, x + 0.54, \Big\}.$$

Letting $c = 0.9$, we have used Wendland's function $\psi_{6,4}(cr)$ as the radial basis function given by

$$\psi_{6,4}(cr) = (1 - cr)_+^{10} \left(2145(cr)^4 + 2250(cr)^3 + 1050(cr)^2 + 250cr + 25 \right),$$

in which $x_+ = x$ for $x \ge 0$, $x_+ = 0$ for $x < 0$, and $x_+^l := (x_+)^l$. The triangulation was created over the area $[-0.6,\ 0.5] \times [-0.4,\ 1]$ with different numbers of vertices (varied between 1000^2 and 1600^2). All the triangulations consisted of congruent rectangular triangles.

In Figures 1, and 2, we have illustrated the results of computations for this example with 1400^2 vertices. The blue area in Figure 1, indicate the vertices at which **(VP1)** is not satisfied, and in Figure 2, the vertices at which **(VP4)** is not satisfied are colored as red. Putting these plots together, the intersected white area represents the area where all constrains are fulfilled.

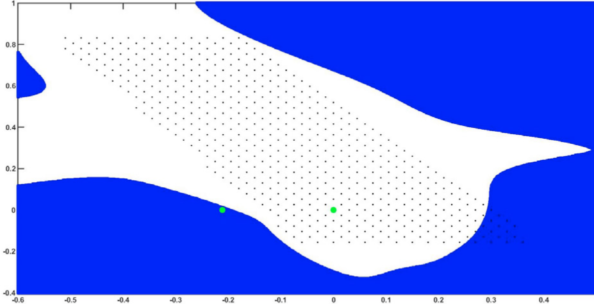

Fig. 1. The black points are the collocation points. The Blue stars indicate the vertices at which **(VP1)** is not satisfied. The Green circles show the equilibrium points.

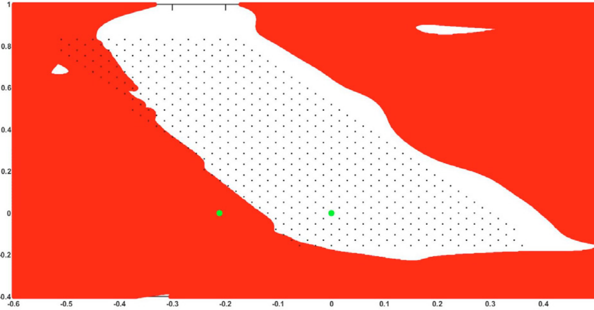

Fig. 2. The black points are the collocation points. The red dots indicate the vertices at which **(VP1)** is not satisfied. The Green circles show the equilibrium points.

2.2 Computational Complexity

In the first step of the method, we numerically solve the matrix valued PDE

$$F(M)(x) = -I, \tag{2.5}$$

where I is the $(n \times n)$ identity matrix, using the RBF method. We set $\psi_0(r) := \psi_{6,4}(cr)$ and denote $\psi_{q+1}(r) = \frac{1}{r}\frac{d\psi_q}{dr}(r)$ for $q = 0, 1$. Denote by N the number of the collocation points used and by n the dimension of the system; in our example $N = 546$ and $n = 2$. We first compute the coefficients $b_{k,\ell,i,j,\mu,\nu}$ with

$$\begin{aligned}
b_{k,\ell,i,j,\mu,\nu} = {} & \psi_0(\|x_k - x_\ell\|_2) \\
& \Big[\sum_{p=1}^{n} Df_{pi}(x_\ell)Df_{p\mu}(x_k)\delta_{\nu j} + Df_{\mu i}(x_\ell)Df_{j\nu}(x_k) \\
& + Df_{i\mu}(x_k)Df_{\nu j}(x_\ell) + \delta_{i\mu}\sum_{p=1}^{n} Df_{p\nu}(x_k)Df_{pj}(x_\ell)\Big] \\
& +\psi_1(\|x_k - x_\ell\|_2)\langle x_k - x_\ell, f(x_k)\rangle \\
& \quad [Df_{\mu i}(x_\ell)\delta_{\nu j} + \delta_{i\mu}Df_{\nu j}(x_\ell)] \\
& +\psi_1(\|x_k - x_\ell\|_2)\langle x_\ell - x_k, f(x_\ell)\rangle \\
& \quad [Df_{i\mu}(x_k)\delta_{\nu j} + \delta_{i\mu}Df_{j\nu}(x_k)] \\
& -\psi_1(\|x_k - x_\ell\|_2)\langle f(x_\ell), f(x_k)\rangle\delta_{i\mu}\delta_{j\nu} \\
& +\psi_2(\|x_k - x_\ell\|_2)\langle x_k - x_\ell, f(x_k)\rangle \\
& \quad \langle x_\ell - x_k, f(x_\ell)\rangle\delta_{i\mu}\delta_{j\nu}
\end{aligned} \tag{2.6}$$

for $1 \le k, \ell \le N$, and $1 \le i, j, \mu, \nu \le n$ (see (Giesl and Wendland, 2019, Subsection 3.2) for more details).

Then we calculate the coefficients $c_{k,\ell,i,i,\mu,\mu}$ with

$$c_{k,\ell,i,i,\mu,\mu} = \frac{1}{4}\left(b_{k,\ell,i,j,\mu,\nu} + b_{k,\ell,j,i,\nu,\mu} + b_{k,\ell,i,j,\nu,\mu} + b_{k,\ell,j,i,\mu,\nu}\right).$$

Finally, we determine the numbers $\gamma_k^{(\mu,\nu)}$, by solving the linear system

$$\sum_{k=1}^{N}\sum_{1\le\mu\le\nu\le n} c_{k,\ell,i,j,\mu,\nu}\gamma_k^{(\mu,\nu)} = (F(S)(x_\ell))_{i,j} = -I_{ij} \tag{2.7}$$

for $1 \le \ell \le N$, and $1 \le i \le j \le n$. Note that (2.7) is a system of $Nn(n+1)/2$ equations in $Nn(n+1)/2$ unknowns. Finally, we compute the $(n \times n)$ matrices $\beta_k \in \mathbb{R}^{n\times n}$ from the numbers $\gamma_k^{(\mu,\nu)}$ with the formulas

$$\beta_k^{(j,i)} = \beta_k^{(i,j)} = \frac{1}{2}\gamma_k^{(i,j)} \quad \text{if } i \ne j,$$
$$\beta_k^{(i,i)} = \gamma_k^{(i,i)}.$$

The optimal recovery now has the formula

$$\begin{aligned}
S(x) = \sum_{k=1}^{N}\Big[&\psi_0(\|x_k - x\|_2)\left[Df(x_k)\beta_k + \beta_k Df(x_k)^T\right] \\
&+ \psi_1(\|x_k - x\|_2)\langle x_k - x, f(x_k)\rangle\beta_k\Big].
\end{aligned} \tag{2.8}$$

Analyzing these equations, we see that the number of elementary operations $(+, \times)$ needed to compute the coefficients for the linear equations (2.7) is of the order $\mathcal{O}(N^2)$ for a fixed n. The order in n for a fixed N is at least $\mathcal{O}(n^5)$ and might be higher depending on f and Df. To solve the linear system we need elementary operations of the order $\mathcal{O}((Nn^2)^3) = \mathcal{O}(N^3n^6)$. Typically N is much larger than n and therefore $\mathcal{O}(N^3n^6)$ is a reasonable estimate on the complexity of the first step of the algorithm, or just $\mathcal{O}(N^3)$ if we consider the dimension n of the system to be fixed. In the following we will disregard the dependance of n, but keep in mind that the computational effort of the method increases very fast with the dimension n of the system considered.

In the second step of the method, we first evaluate formula (2.8) at every vertex of the triangulation, and then we verify the constraints (VP1)-(VP4). For the evaluation of (2.8) at a point we need $\mathcal{O}(N)$ elementary operations, again disregarding dependance of n. Since this must be done for every vertex we need no less than $\mathcal{O}(N_{\text{CPA}}N)$ operations, where N_{CPA} is the number of vertices of the triangulation. It is easy to see that the number of simplices in the triangulation is bounded above by $N_{\text{CPA}}n!$. Thus, it is not difficult to see that the complexity of the verification of the constraints (VD1)-(VD4) is linear in N_{CPA} and does not depend on N. Therefore the complexity of the second step of the method is $\mathcal{O}(N_{\text{CPA}}N)$ for a fixed dimension n, but again the computational effort grows very fast with the dimension n of the system.

3. IMPLEMENTATION DETAILS

In this section we will describe the different softwares and packages we have used with some details. In particular, we will present a MATLAB-based approach and a C++-based one. Furthermore, we describe how we optimized the code to improve the speed in several steps.

BLAS (Basic Linear Algebra Subprograms) [4] is a specification that prescribes a set of common low-level routines in linear algebra, such as matrix-vector multiplication or dot-products. Several highly optimized implementations, taking advantage of specific processor architectures and SIMD (Single Instruction, Multiple Data) instructions are available. LAPACK (Linear Algebra Package) [5] is a standard software library that provides routines for numerical linear algebra operations.

For implementations of BLAS and LAPACK we will be concerned with the Intel Math Kernel Library (MKL) [6] and OpenBLAS. [7] MKL is a library of optimized mathematics routines for science, engineering, and financial applications and provides both the BLAS and LAPACK routines. OpenBLAS provides the BLAS routines and some of the more common LAPACK routines, adds optimized implementations of linear algebra kernels for several processor architectures, and claims to achieve performance comparable to MKL. Both Matlab and Scilab use the MKL library for internal computation, whereas the Julia programming language uses OpenBLAS. The Armadillo [8] C++ library we use in our C++ code can be set to use BLAS and LAPACK routines from MKL or from OpenBLAS, see Sanderson and Curtin (2016, 2018).

3.1 MATLAB-based code

The first draft of the code for the algorithm was created with the MATLAB software. The program in MATLAB is composed of several m-files. Each m-file is a piece of code or a function that could be called in another code.

The optimal recovery problem using RBF, reduces to solving the system of linear equations (2.7), i.e. a linear matrix equation $Ax = b$, where b is a column vector generated in accordance with the identity matrix; and the coefficient matrix A is a big sparse matrix generated using formula (2.6) using several m-file sub-functions representing the radial basis function and its derivatives, the right-hand-side function of the dynamical system equation $\dot{x} = f(x)$ and its derivative.

A matrix *points*, the rows of which store the coordinates of collocation points, is generated by a sub-algorithm implementing the creation of a hexagonal grid and the coefficients_new.m function reads *points*, and an integer N, which is the total number of collocation points, and returns *points*, N, and a vector β which are the coefficients in the formula for S in (2.8). The total time for this procedure is reported as the *RBF time* in tables in the next section.
It is clear from the discussion in Subsection 2.2 that this step is much faster than evaluating S at all the vertices of the triangulation in use and the subsequent CPA verification step, and thus we focus on speed improvement for the latter.

For each vertex x_k in the CPA verification process, $P(x_k) = S(x_k)$ is calculated by using data from all the collocation points, thus, too few collocation points result in a bad approximation, no matter how small the simplices are. On the other hand, too many collocation points will make the CPA verification step unnecessarily time consuming. Therefore, we first optimize the number of collocation points N, and then go to the next step, when we believe that the optimal recovery S is a good approximate solution to the PDE (2.5). This is the reason why the data reported in the tables in next section is for different N_{CPA} values with a fixed N.

After having completed the first step of the algorithm, the CPA.m sub-algorithm is called to create the triangulation $\mathcal{T}$, evaluate $P(x_k) = S(x_k)$ at each vertex x_k, and verify the constraints (VP1)-(VP4). Positive definiteness of the matrices is obtained from the built-in MATLAB function chol, which gives the Cholesky factorization of a presumed symmetric matrix. [∼ ,flag] = chol(A) returns the output flag, indicating whether A is symmetric positive definite. If flag= 0 then the input matrix is symmetric positive definite and the factorization was successful. If flag is not zero, then the input matrix is not symmetric positive definite and flag is an integer indicating the index of the pivot position where the factorization failed. It turns out that using chol is faster than computing the eigenvalues and checking them for positivity.

The program returns a matrix T, where each row represents a simplex, having the identifier number of vertices in that simplex, a matrix *Vertex* containing the coordinates of all vertices, matrices *Peval*, and *Aeval* containing the coordinates of the vertices that fail (VP1) and (VP4), respectively. The code saves these data in a .mat file, and then the total time is measured. Plotting the results is done after this and is not part of the comparison.

MEX file functions
Stated on the Mathworks website is that

> a MEX file is a function, created in MATLAB, that calls a C/C++ program or a Fortran subroutine. A MEX function behaves just like a MATLAB script or function.

MEX-file functions can increase the speed of a program very effectively. In order to create a MEX-file one needs to have the "MATLAB-coder" app included in the MATLAB license. It is important to decide which parts of the code, or which functions we want to change to a MEX file. In our case, it seemed adequate to create two MEX files for the two steps of the algorithm.

There are a few challenges when using MEX-files, for example, not all predefined MATLAB functions can be used inside a MEX-file function. This means that a function might have to be modified to be able to convert it to a MEX-file, and sometimes it may not be possible or not worth the effort. Another disadvantage of MEX-files is that unlike the MATLAB code itself, one cannot just modify it and run it again. Even for small adjustments, one needs to modify the code, recreate the MEX-file, and then use it in the program.

Another important point to mention is that MEX-files are dependent on the operating system or platform of the computer on which it was created. So, for example, if one has created a MEX file on a Windows based PC,

[4] see the official web-page http://netlib.org/blas/.
[5] see http://www.netlib.org/lapack/
[6] see https://software.intel.com/en-us/mkl
[7] see https://www.openblas.net/ for more information.
[8] see http://arma.sourceforge.net/

it cannot necessarily be run on a Linux based server. In the MATLAB documentation three binary MEX file extensions for Linux, Apple Mac, and Windows (all 64-bit versions) are discussed.

Parallel loops
In order to use modern multi-core CPUs efficiently, one can attempt to run some parts of the code in parallel. To do so in MATLAB, one needs to have the Parallel Computing Toolbox. A very simple and effective way to start with, is to change for-loops into parfor-loops.

A disadvantage of using parfor for multithreading is that MATLAB must be able to recognize variables inside the loop as one of the five types it can deal with. The indexing variables of parfor should be consecutive increasing integers. Also using the loop variable for working with different components of a matrix or vector (called sliced variables) can be hard and time-consuming. One cannot just change for to parfor and then expect everything to work. In our case, although we were sure the code could be multithreaded the way it was without any problem (as we did it in C++), MATLAB gives red flag warnings complaining it cannot run parfor loops due to the ambiguity in the way a loop variable is used or a sliced variable is referred to. In the evaluation of constraints (VP2)-(VP4) we were not able to fix the errors and use parfor. However, as the most demanding computations are in evaluating P at the vertices, this is not too important for the running times.

Fast codepath for MKL on AMD processors
During our programming we got to an interesting case of MKL not taking full advantage of SIMD instructions on AMD processors, described in detail in the post "how to force MATLAB to use a fast codepath on AMD" in the MATLAB community on Reddit [9] . Because MATLAB uses the Intel MKL for some operations this makes many operations unnecessarily slow.
However, this can be overridden (or fixed) by adding and setting a new system environment variable

```
MKL_DEBUG_CPU_TYPE=5
```

After doing this all programs using the MKL on the system are affected by the change. In our case, both MATLAB and C++ using MKL show significant speed improvements. It is worth mentioning that MEX files should be created after having the new system variable set.

3.2 C++-based code

We transferred our MATLAB code to C++ in the hope for a better performance. One can object that this may not be the most efficient way to implement in C++ as the language provides wider capabilities, but since we had the MATLAB code this seemed sensible. In order to simplify the transferral we used the Armadillo library, which is described on its website [10] as:

> Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use. It provides high-level syntax and functionality deliberately similar to MATLAB, and various matrix decompositions are provided through integration with LAPACK, or one of its high performance drop-in replacements (eg. multi-threaded Intel MKL, or OpenBLAS).

As mentioned before, the fast code path for MKL on AMD processors also affects the speed of C++ code when it uses MKL. One of the very few changes in the code we needed to do was for verifying the positive definiteness of the matrices (in VP1 and VP4). The chol function, which does the Cholesky decomposition in Armadillo, had some problems with providing only a flag and not a *runtime_error* exception when the decomposition fails. Therefore, we used another built-in function .is_sympd(), which returns *true* if the matrix is symmetric/hermitian positive definite, and returns *false* otherwise.

Parallel loops
As we were using Microsoft Visual Studio on a Windows based system, we decided to use the Parallel Patterns Library (PPL) for multithreading the code. PPL provides algorithms that concurrently perform work on collections of data. These algorithms resemble those provided by the C++ Standard Library.
We used concurrency::parallel_for in the RBF sub-algorithm when we write the matrix and we experimented with having it in different parts of the CPA verification process.

It turns out that placing some functions from Armadillo that use LAPACK in a concurrency::parallel_for loop is either very slow or even leads to hang-ups. However, we only had this problem with the solve function and replacing it with our own routine fixed the problem. Note that we also had this problem when using a computer with an Intel CPU. A possible reason is that the multithreading in our program interferes with the multithreading in LAPACK.

4. RESULTS AND COMPARISON

In this section we provide the test results in several tables and give some comments. All tests were run on an MS Windows 10 system with an AMD Ryzen processor (2700X, 8 cores, 3.7 GHz) and 64GB RAM. In all cases we used the setting described in Section 2.1. In particular we always used $N = 546$ collocation points and only changed the triangulation using different values for the number of vertices N_{CPA}.

In Table 1 and Table 2, we have provided the running times using the MATLAB code. In these tables *code-path* indicates that the fast code-path was used, *parallel* refers to the code having been parallelized, and *MEX* refers to the use of MEX files. All this is discussed in Section 3.1.

The data in Table 1 suggests that the fast code-path can make the program run more than 2.5 times faster. The last column shows a significant decrease of run times due to the multithreading of the evaluation of the optimal recovery at the vertices.

The data from Table 2 shows that the MEX file version of the MATLAB code is much faster in all cases and that

[9] posted by blogger "nedflanders1976" accessible at this address: https://www.reddit.com/r/matlab/comments/dxn38s/howto_force_matlab_to_use_a_fast_codepath_on_amd/
[10] see Armadillo website http://arma.sourceforge.net/

the effect of using the fast code-path is clear, but by far not as impressive as when not using MEX files.

In Table 3 and Table 4 the running time for the code in C++ is shown. In each table we run the program using the MKL library without the fast code-path, then using MKL with the fast code-path, and then using OpenBLAS instead of MKL. In Table 3 the code is not multithreaded and in Table 4 the code is multithreaded.

The data from Table 3 suggests that the fast code-path can make the program 3 times faster and should definitely be employed, just as in the MATLAB case. Comparing Tables 2-4, we come to the conclusion that for our application using MATLAB with MEX-files and the fast code-path results in programs that are as fast as the programs written in C++, and that it does not matter whether we use the MKL library or OpenBLAS, provided that we used the fast code-path for MKL.

5. CONCLUSION

We have used two different ways with several settings for each to compute a numerical algorithm, and have provided the details in previous sections. Here we are going to list a few merits of these methods.

Using MATLAB is simpler than programming with C++ for non professional programmers in need of scientific computing. Although Armadillo has decreased the level of difficulty of programming in C++, having the command window and editor at the same time in MATLAB makes it much easier to translate a mathematical formula/algorithm to a program, checking commands line by line and seeing the changes in part without having a full code written.

In case of time and memory consuming calculations, C++ becomes faster and more efficient. Although one can improve the performance of MATLAB programs considerably with several techniques like using MEX-files, it is important to note that not all MATLAB functions adapt easily to the speedy solutions.

Running the code in parallel is easier in C++ than in MATLAB, even when not considering extra toolboxes and apps that are needed for MATLAB and depend on the operating system. For plotting and evaluating the results after the computation, MATLAB's graphical tools and interface are much more user-friendly and developed than what is available in C++. In fact, we transferred the calculation results from C++ to MATLAB for plotting. Thus, one should also consider the difference between time saved using C++, and the time spent in loading these data into MATLAB; which in turn, leads to the next point.
The format in which one can store the results in MATLAB, as a .mat file, is very efficient and concise, though they are not easily read into other programs not written in MATLAB. In C++, it is not that simple to store arrays with different sizes and dimensions in one single file. Thus, being able to read a single file in a meaningful way or storing several files and loading them properly becomes laborious. Using the Hierarchical Data Format (HDF5) might be a solution.

In summary for our application: provided that we use the fast code-path for the MKL library, MATLAB with MEX-files and parallelization delivers programs that are just as fast as the programs we get using C++. In the latter case MKL and OpenBLAS deliver programs with practically identical running times, again assuming that we use the fast-code path for MKL.

Table 1. Different settings and run times for MATLAB

N_{CPA}	MATLAB	MATLAB code-path	MATLAB code-path parallel
1000^2	2 h 37 min	58 min	11 min
1200^2	3 h 48 min	1 h 25 min	16 min
1400^2	5 h 4 min	1 h 54 min	22 min
1600^2	6 h 30 min	2 h 27 min	28 min

Table 2. Different settings and run times for MATLAB using MEX files

N_{CPA}	MATLAB MEX	MATLAB code-path MEX	MATLAB code-path parallel MEX
1000^2	6 min	5 min	40 s
1200^2	9.5 min	8.5 min	54 s
1400^2	13 min	12 min	71 s
1600^2	17 min	15 min	91 s

Table 3. Different settings and run times for C++, not multithreaded

N_{CPA}	C++ MKL	C++ MKL code-path	C++ OpenBLAS
1000^2	7 min	4 min	4 min
1200^2	12.5 min	5.5 min	5.5 min
1400^2	21 min	7.5 min	7.5 min
1600^2	32 min	10 min	10 min

Table 4. Different settings and run times for C++, multithreaded

N_{CPA}	C++ MKL parallel	C++ MKL code-path parallel	C++ OpenBLAS parallel
1000^2	43 s	39 s	38 s
1200^2	64 s	56 s	54 s
1400^2	87 s	75 s	72 s
1600^2	112 s	97 s	94 s

REFERENCES

Giesl, P. (2007). *Construction of Global Lyapunov Functions Using Radial Basis Functions*, volume 1904 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin.

Giesl, P. (2015). Converse theorems on contraction metrics for an equilibrium. *J. Math. Anal. Appl.*, (424), 1380–1403.

Giesl, P. and Hafstein, S. (2015). Computation and verification of Lyapunov functions. *SIAM Journal on Applied Dynamical Systems*, 14(4), 1663–1698.

Giesl, P. and Wendland, H. (2019). Construction of a contraction metric by meshless collocation. *Discrete Contin. Dyn. Syst. Ser. B*, 24(8), 3843–3863.

Giesl, P., Hafstein, S., and Mehrabinezhad, I. (2019). Computation and verification of contraction metrics for exponentially stable equilibria. *arXiv preprint arXiv:1909.10334*.

Iske, A. (1998). Perfect centre placement for radial basis function methods. Technical Report TUM-M9809, TU Munich, Germany.

Sanderson, C. and Curtin, R. (2016). Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(2), 26.

Sanderson, C. and Curtin, R. (2018). A user-friendly hybrid sparse matrix class in C++. In *International Congress on Mathematical Software*, 422–430. Springer.

Wendland, H. (2005). *Scattered data approximation*, volume 17. Cambridge university press.