Efficient Verification of CPA Lyapunov Functions

Sigurdur Freyr Hafstein
Science Institute, University of Iceland, Dunhagi 3, 107 Reykjavík, Iceland

Keywords: Lyapunov Function, CPA Verification, Efficient Algorithms.

Abstract: Lyapunov functions can be used to characterize the stability and basins of attraction for dynamical systems, whose dynamics are defined by ordinary differential equations. Since the analytic generation of Lyapunov functions for nonlinear systems is a formidable task, one often resorts to numerical methods. In this paper we study the efficient verification of the conditions for a Lyapunov function using affine interpolation over a triangulation; the values of the Lyapunov function candidate at the vertices of the triangulation can be generated using various different formulas from converse theorems in the Lyapunov stability theory. Further, we give an implementation in C++ and demonstrate its efficiency and applicability.

1 INTRODUCTION

In applications of dynamical systems in science and engineering the stability of equilibria and other invariant sets is often a necessary requirement. In particular, control theory is concerned with the designing of controllers and observers, such that the resulting closed-loop systems are stable, e.g. have exponentially stable equilibria. The Lyapunov stability theory is a much used tool in this regard and has been intensively studied since its introduction by Lyapunov in 1892 (Lyapunov, 1992); see e.g. the textbooks (Zubov, 1964; Yoshizawa, 1966; Hahn, 1967; Sastry, 1999; Vidyasagar, 2002; Khalil, 2002). For a physical system the obvious candidate for a Lyapunov function is the system's (free) energy and a dissipative physical system approaches a local minimum of the energy.

The analytical generation of a Lyapunov function for a system is in general a formidable problem. Therefore numerous numerical methods have been developed, e.g. parameterizing rational (Vannelli and Vidyasagar, 1985; Valmorbida and Anderson, 2017) or polynomial (Parrilo, 2000; Chesi, 2011; Anderson and Papachristodoulou, 2015; Ratschan and She, 2010; Kamyar and Peet, 2015) Lyapunov functions; for an overview of numerical methods see (Giesl and Hafstein, 2015b).

In (Julian, 1999; Julian et al., 1999; Marinósson, 2002a) linear programming was used to parameterize continuous and piecewise affine (CPA) Lyapunov functions. In this approach, a subset of the state space is first triangulated, i.e. subdivided into simplices, and then a number of constraints are derived for a given nonlinear system, such that a feasible solution to the resulting linear programming problem allows for the parametrization of a CPA Lyapunov function for the system. We refer to this method as the CPA algorithm and the constraints of the linear programming problem as the CPA constraints.

In (Hafstein, 2004; Giesl and Hafstein, 2014) it was proved that the CPA algorithm is always able to compute a Lyapunov function for systems with an exponentially stable equilibrium. Another approach uses the CPA constraints, but instead of solving the linear programming problem, a faster method to compute values for the variables of the problem that likely constitute a solution is used. Thus, the CPA constraints of the slow but rigorous CPA method are combined with a faster but less rigorous method to compute Lyapunov functions to deliver a fast and rigorous method. The fast method has been based on an integration- or sum formula for a Lyapunov function from a converse theorem, see e.g. (Hafstein et al., 2014a; Hafstein et al., 2014b; Björnsson et al., 2014; Li et al., 2015; Hafstein et al., 2015; Björnsson et al., 2015; Doban and Lazar, 2016; Doban, 2016; Björnsson and Hafstein, 2017; Hafstein and Valfells, 2017; Hafstein and Valfells, 2019; Gudmundsson and Hafstein, 2015; Hafstein, 2019), or on collocation using radial basis functions, see (Giesl and Hafstein, 2015a).

In this paper we present a fast verification of the conditions of the linear programming problem from

Hafstein, S.
Efficient Verification of CPA Lyapunov Functions.
DOI: 10.5220/0011231700003271
In *Proceedings of the 19th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2022)*, pages 120-129
ISBN: 978-989-758-585-2; ISSN: 2184-2809

Listing 1: Simple thread-parallelization in C++.

```
using bint = long long;  // bint = big integer

void ParallelFor(bint _beg, bint _end, function<void(bint)> func, bint NrThreads){
  for (bint i = _beg; i < _end; i += NrThreads) {
    vector<thread> threads(NrThreads);
    for (bint j = i; j < i + NrThreads && j < _end; j++) {
      threads[j % NrThreads] = thread(func, j);
    }
    for (bint j = i; j < i + NrThreads && j < _end; j++) {
      threads[j % NrThreads].join();
    }
  }
}

void ParallelFor(bint _end, function<void(bint)> func, bint NrThreads) {
    ParallelFor(0, _end, func, NrThreads);
}
```

(Marinósson, 2002a) and its implementation in C++ using the Armadillo linear algebra library (Sanderson, 2010; Sanderson and Curtin, 2016). The implementation does not actually construct the linear programming problem, but verifies the constraints on the fly, and the procedure is very fast and memory efficient.

The paper is organized as follows. After presenting the necessary notation and prerequisites we discuss and develop CPA constraints that are particularly well suited for an efficient verification in Section 2. Then we discuss the implementation of the verification in C++ in Section 3 and conclude the paper in Section 4.

1.1 Notation and Prerequisites

We denote the set $\{1,2,\dots,\}$ by $\mathbb{N}$ and set $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. For a vector $\mathbf{x} \in \mathbb{R}^n$ and $p \geq 1$ we define the norms $\|\mathbf{x}\|_p = (\sum_{i=1}^n |x_i|^p)^{1/p}$ and $\|\mathbf{x}\|_\infty = \max_{i\in\{1,\dots,n\}} |x_i|$.

We utilize a bold-face font for (column) vectors, e.g. $\mathbf{x} \in \mathbb{R}^{n\times 1} = \mathbb{R}^n$, and we denote by $\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n$ the standard orthonormal basis of $\mathbb{R}^n$. For a vector $\mathbf{x}$ we write x_i for its ith component (U_i for the ith component of $\mathbf{U}$) and for a matrix $A \in \mathbb{R}^{n\times n}$ we write a_{ij} for its (i,j)th element. Their transposes are denoted $\mathbf{x}^{\mathrm{T}}$ and A^{T}. The zero vector in $\mathbb{R}^n$ is written $\mathbf{0}$ and $\mathbf{1} := (1,1,\dots,1)^{\mathrm{T}} \in \mathbb{R}^n$. For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ the inequality $\mathbf{x} \leq \mathbf{y}$ is understood component-wise, i.e. $x_i \leq y_i$ for $i = 1,2,\dots,n$. $B = \mathrm{diag}(a_1, a_2, \dots, a_n)$ defines a matrix $B \in \mathbb{R}^{n\times n}$ with $b_{11} = a_1, b_{22} = a_2, \dots, b_{nn} = a_n$ and $b_{ij} = 0$ for $i \neq j$, i.e. B is a diagonal matrix.

Our C++ implementation makes heavy use of the STL library and the Armadillo linear algebra library, that is very well documented on its webpage http://arma.sourceforge.net. Just a few comments: the STL library delivers a routine to iterate through permutations called `next_permutation`; if one starts with the identity permutation `int v[i]=i` one iterates through all permutations. In Armadillo the most important types, at least here, are `vec` for a column vector of `double` and `ivec` for a column vector of integers; `mat` and `imat` are the matrix versions. Since C++11 multithreading has been made very easy in C++, we use the simple code in Listing 1 for multithreading.

2 CPA CONSTRAINTS

We consider systems, whose dynamics are given by an ODE of the form

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \quad \mathbf{f} \in C^2(\mathbb{R}^n; \mathbb{R}^n). \tag{1}$$

Typically one assumes that the system in question has an equilibrium w.l.o.g. at the origin, but that is not necessary for our discussion. We assume there is a Lyapunov function candidate $V\colon \mathcal{D} \to \mathbb{R}$ given, together with a triangulation of its domain $\mathcal{D} \subset \mathbb{R}^n$. Our intention is to verify where the continuous and affine interpolation of the function V over the simplices of the triangulation has a negative orbital derivative. Recall that a negative orbital derivative implies that the function is decreasing along solution trajectories of the system (1). For concretizing our ideas a few definitions are necessary.

Let $C = \{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n\} \subset \mathbb{R}^n$ be a set of affinely independent vectors, i.e. the augmented vectors $(\mathbf{x}_0, 1), (\mathbf{x}_1, 1), \dots, (\mathbf{x}_n, 1) \in \mathbb{R}^{n+1}$ are linearly independent. The convex hull of the vectors in C, i.e. the

set

$$\mathrm{co}C := \left\{ \sum_{k=0}^{n} \lambda_k \mathbf{x}_k \,:\, \mathbf{x}_k \in C,\ \ \lambda_k \in [0,1],\ \ \sum_{k=0}^{n} \lambda_k = 1 \right\},$$

is called a proper n-simplex and the vectors in C are said to be its vertices.

Let $\{S_\nu\}_{\nu\in T} = \mathcal{T}$, T an index set, be a set of proper n-simplices in $\mathbb{R}^n$, such that different simplices $S_\nu, S_\mu \in \mathcal{T}$ intersect in a common face or not at all and such that the interior of the set $\mathcal{D}_\mathcal{T} = \bigcup_{\nu\in T} S_\nu$ is a simply connected set. The set $\mathcal{T}$ is said to be a *shape-regular triangulation* in $\mathbb{R}^n$. We refer to the set

$$\mathcal{V}_\mathcal{T} := \{\mathbf{x}_i \,:\, \mathbf{x}_i \text{ is a vertex of a simplex } S_\nu \in \mathcal{T}\}$$

as the *vertex set* of the triangulation $\mathcal{T}$.

Let a shape-regular triangulation $\mathcal{T} = \{S_\nu\}_{\nu\in T}$ in $\mathbb{R}^n$ be given, together with the system (1) and the Lyapunov functions candidate $V\colon \mathcal{D} \to \mathbb{R}$ with $\mathcal{D} = \mathcal{D}_\mathcal{T}$. The following estimates are of essential importance for the CPA algorithm, because they allow us to check certain inequalities at the vertices $\mathcal{V}_\mathcal{T}$ of the simplices in $\mathcal{T}$ to obtain estimates on the entire domain $\mathcal{D}_\mathcal{T}$.

For a proper n-simplex $S_\nu = \mathrm{co}\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n\}$ in $\mathcal{T}$ and the C^2 vector field $\mathbf{f} = (f_1, f_2, \ldots, f_n)^\mathrm{T}$ define a constant $B^\nu_{r,s}$ such that

$$B^\nu_{r,s} \geq \max_{\substack{\mathbf{x}\in S_\nu \\ m=1,2,\ldots,n}} \left| \frac{\partial^2 f_m}{\partial x_r \partial x_s}(\mathbf{x}) \right|. \tag{2}$$

Further, for each (vertex) $\mathbf{y}$ of S_ν define

$$C^\nu_{\mathbf{y},s} := \max_{j=0,1,\ldots,n} |\mathbf{e}_s \cdot (\mathbf{x}_j - \mathbf{y})|,$$

where $\cdot$ denotes the scalar-product, and set

$$E^\mathbf{y}_{\nu,\mathbf{x}_i} := \frac{1}{2} \sum_{r,s=1}^{n} B^\nu_{r,s} |\mathbf{e}_r \cdot (\mathbf{x}_i - \mathbf{y})| (C^\nu_{\mathbf{y},s} + |\mathbf{e}_s \cdot (\mathbf{x}_i - \mathbf{y})|) \tag{3}$$

for $i = 0, 1, \ldots, n$. The constants $E^\mathbf{y}_{\nu,\mathbf{x}_i}$ are defined such that for a fixed vector $\mathbf{v} \in \mathbb{R}^n$ we have that

$$\mathbf{v}\cdot\mathbf{f}(\mathbf{x}_i) + E^\mathbf{y}_{\nu,\mathbf{x}_i} \|\mathbf{v}\|_1 \leq 0 \tag{4}$$

for $i = 0, 1, \ldots, n$ implies $\mathbf{v}\cdot\mathbf{f}(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in \mathrm{co}C$.

This follows by the estimate (proved in (Marinósson, 2002b, Lemma 4.16))

$$\left\| \mathbf{f}(\mathbf{x}) - \sum_{i=0}^{n} \lambda_i \mathbf{f}(\mathbf{x}_i) \right\|_\infty \leq \sum_{i=0}^{n} \lambda_i E^\mathbf{y}_{\nu,\mathbf{x}_i} \tag{5}$$

for all convex combinations $\mathbf{x} = \sum_{i=0}^{n} \lambda_i \mathbf{x}_i \in S_\nu$ and Hölder's inequality:

$$\begin{aligned}
\mathbf{v}\cdot\mathbf{f}(\mathbf{x}) &= \sum_{i=0}^{n} \lambda_i \mathbf{v}\cdot\mathbf{f}(\mathbf{x}_i) + \mathbf{v}\cdot\left[\mathbf{f}(\mathbf{x}) - \sum_{i=0}^{n} \lambda_i \mathbf{f}(\mathbf{x}_i)\right] \\
&\leq \sum_{i=0}^{n} \lambda_i \mathbf{v}\cdot\mathbf{f}(\mathbf{x}_i) + \|\mathbf{v}\|_1 \left\| \mathbf{f}(\mathbf{x}) - \sum_{i=0}^{n} \lambda_i \mathbf{f}(\mathbf{x}_i) \right\|_\infty \\
&\leq \sum_{i=0}^{n} \lambda_i \mathbf{v}\cdot\mathbf{f}(\mathbf{x}_i) + \|\mathbf{v}\|_1 \cdot \sum_{i=0}^{n} \lambda_i E^\mathbf{y}_{\nu,\mathbf{x}_i} \\
&= \sum_{i=0}^{n} \lambda_i \left(\mathbf{v}\cdot\mathbf{f}(\mathbf{x}_i) + \|\mathbf{v}\|_1 E^\mathbf{y}_{\nu,\mathbf{x}_i} \right) \\
&\leq 0.
\end{aligned}$$

Note that in the condition (4) the $B^\nu_{r,s}$ are just upper bounds and that the vertex $\mathbf{y}$ of S_ν for $E^\mathbf{y}_{\nu,\mathbf{x}_i}$ is arbitrary but fixed for $i = 0, 1, \ldots, n$.

Given a triangulation $\mathcal{T}$, a continuous and piecewise affine function, a so-called CPA function, can be defined by fixing its values at $\mathcal{V}_\mathcal{T}$.

Definition 2.1 (CPA function)**.** *For a shape-regular triangulation $\mathcal{T}$ in $\mathbb{R}^n$ we denote by* $\mathrm{CPA}[\mathcal{T}]$ *the set of all continuous functions*

$$V \,:\, \mathcal{D}_\mathcal{T} \to \mathbb{R}$$

that are affine on each simplex $S_\nu \in \mathcal{T}$. Hence, for each $S_\nu \in \mathcal{T}$ there exists a vector $\mathbf{v}_\nu \in \mathbb{R}^n$ and a number $a_\nu \in \mathbb{R}$ such that

$$V(\mathbf{x}) = \mathbf{v}_\nu \cdot \mathbf{x} + a_\nu$$

for all $\mathbf{x} \in S_\nu$.

It is not difficult to see that V in the definition above is completely determined by its values at the vertices $\mathcal{V}_\mathcal{T}$ and with $S_\nu = \mathrm{co}\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n\}$,

$$\mathbf{w}_\nu := \begin{pmatrix} V(\mathbf{x}_1) - V(\mathbf{x}_0) \\ V(\mathbf{x}_2) - V(\mathbf{x}_0) \\ \vdots \\ V(\mathbf{x}_n) - V(\mathbf{x}_0) \end{pmatrix} \in \mathbb{R}^n,$$

and

$$X_\nu := \begin{pmatrix} (\mathbf{x}_1 - \mathbf{x}_0)^\mathrm{T} \\ (\mathbf{x}_2 - \mathbf{x}_0)^\mathrm{T} \\ \vdots \\ (\mathbf{x}_n - \mathbf{x}_0)^\mathrm{T} \end{pmatrix} \in \mathbb{R}^{n\times n},$$

the formula for $\mathbf{v}_\nu$ is

$$\mathbf{v}_\nu = X_\nu^{-1} \mathbf{w}_\nu. \tag{6}$$

The system (1) together with the $B^\nu_{r,s}$ is implemented as in Listing 2, where we show it for the concrete example of the time reversed van der Pol system

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \quad \text{with} \quad \mathbf{f}(x,y) = \begin{pmatrix} -y \\ x + (x^2 - 1)y \end{pmatrix}. \tag{7}$$

Listing 2: Implementation of system (7).

```
extern const unsigned int n = 2;

struct System {
    virtual vec f(const vec &x) = 0;     // the vector-field f in x'=f(x)
    // B(r,s,xl,xu) >= max_{i=1,..,n} sup_{xl <= x <=xu} |D_{rs}f_i(x)|
    virtual double B(int r, int s, const vec &xl, const vec &xu) = 0;
};

struct IVDP : public System {
    vec f(const vec &x) override {
        vec fx(n);
        fx(0) = -x(1);
        fx(1) = x(0) + (pow(x(0),2) - 1.0) * x(1);
        return fx;
    }
    double B(int r, int s, const vec &xl, const vec &xu) override {
        if(r == 0 && s == 0){
            return 2.0*max(abs(xl(1)),abs(xu(1)));
        }
        else if(r==0 && s==1 || r==1 && s==0){
            return 2.0*max(abs(xl(0)), abs(xu(0)));
        }
        else{
            return 0.0;
        }
    }
};
```

A concrete system inherits the virtual class `System` and delivers implementations for the member functions `System.f` and `System.B`. The latter gives an upper bound as in (2), however not for $\mathbf{x} \in S_\nu$ but for $\mathbf{x}$ in the cube $\{\mathbf{x} \in \mathbb{R}^n \colon \texttt{xl} \le \mathbf{x} \le \texttt{xu}\}$. Note that $r,s = 0$ corresponds to the variable `x(0)` $= x$ and $r,s = 1$ to the variable `x(1)` $= y$ and that the only non-zero second-order derivatives of the components of $\mathbf{f} = (f_1, f_2)^{\mathrm{T}}$ in (7) are

$$\frac{\partial^2 f_2}{\partial x^2}(x,y) = 2y \quad \text{and} \quad \frac{\partial^2 f_2}{\partial x \partial y}(x,y) = 2x.$$

For an efficient implementation of the verification it is advantageous to use particularly simple and regular triangulations, both in terms of speed but more importantly in terms of size. We will use a stair case triangulation, which is the so-called standard triangulation $\mathcal{T}_{\text{std}}$, see e.g. (Albertsson et al., 2020) and for extensions (Giesl and Hafstein, 2021b; Giesl and Hafstein, 2021a), but restricted to $\{\mathbf{x} \in \mathbb{R}^n \colon \mathbf{0} \le \mathbf{x} \le \mathbf{U}\}$ for a vector $\mathbf{U} \in \mathbb{N}^n$. This triangulation is then scaled along the axes and translated to the area of interest.

Definition 2.2 (Stair case Triangulation)**.** *The* stair case triangulation $\mathcal{T}_{\text{sc}}^{\mathbf{U}}$ *for a vector* $\mathbf{U} \in \mathbb{N}^n$ *is a triangulation* $\mathcal{T}_{\text{sc}}^{\mathbf{U}} = \{\widetilde{S}_\nu\}_{\nu \in T}$ *with indices* $\nu = (\mathbf{z}, \sigma)$*, for all* $\mathbf{z} \in \mathbb{N}_0$ *fulfilling* $\mathbf{0} \le \mathbf{z} \le \mathbf{U} - \mathbf{1}$ *and all permutations* σ *of* $\{1,2,\dots,n\}$*. The vertices of each* $\widetilde{S}_\nu = (\widetilde{\mathbf{x}}_0^\nu, \widetilde{\mathbf{x}}_1^\nu, \dots, \widetilde{\mathbf{x}}_n^\nu)$ *are given by*

$$\widetilde{\mathbf{x}}_k^\nu = \mathbf{z} + \sum_{j=1}^{k} \mathbf{e}_{\sigma(j)} = \mathbf{z} + \mathbf{u}_k^\sigma \tag{8}$$

where

$$\mathbf{u}_k^\sigma = \sum_{j=1}^{k} \mathbf{e}_{\sigma(j)}.$$

Given a star case triangulation $\mathcal{T}_{\text{sc}}^{\mathbf{U}}$ *and vectors* $\boldsymbol{\ell}, \mathbf{u} \in \mathbb{R}^n$*,* $\boldsymbol{\ell} < \mathbf{u}$*, we define* $P_i := (u_i - \ell_i)/U_i$ *for* $i = 1,2,\dots,n$ *and the matrix* $P := \operatorname{diag}(P_1, P_2, \dots, P_n) \in \mathbb{R}^{n \times n}$*. The triangulation* $\mathcal{T}_{\text{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}} = \{S_\nu\}_{\nu \in T}$ *is now defined by mapping the simplices* $\widetilde{S}_\nu$ *of* $\mathcal{T}_{\text{sc}}^{\mathbf{U}}$ *with the mapping* $\mathbf{x} \mapsto P\mathbf{x} + \boldsymbol{\ell}$*, i.e.* S_ν *in* $\mathcal{T}_{\text{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$ *is the image of the simplex* $\widetilde{S}_\nu$*. The vertices of the simplex* S_ν *are given by*

$$\mathbf{x}_k^\nu := P\widetilde{\mathbf{x}}_k^\nu + \boldsymbol{\ell}$$

for $k = 0,1,\dots,n$*.*

Note that since $\mathcal{T}_{\text{sc}}^{\mathbf{U}}$ is a triangulation of $\{\mathbf{x} \in \mathbb{R}^n \colon \mathbf{0} \le \mathbf{x} \le \mathbf{U}\}$ we have that $\mathcal{T}_{\text{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$ is a triangulation of $\{\mathbf{x} \in \mathbb{R}^n \colon \boldsymbol{\ell} \le \mathbf{x} \le \mathbf{u}\}$.

Using the triangulation $\mathcal{T}_{\text{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$ the constraints (4) for $\nu = (\mathbf{z}, \sigma)$, $S_\nu = \operatorname{co}\{\mathbf{x}_0^\nu, \mathbf{x}_1^\nu, \dots, \mathbf{x}_n^\nu\}$ and with $\mathbf{v}_\nu :=$

$\mathbf{v}$ and $E_i^{\nu} := E_{\nu,\mathbf{x}_i}^{\mathbf{x}_0^{\nu}}$ can be written in the form

$$0 \geq \sum_{j=1}^{n} \frac{V(\mathbf{x}_j^{\nu}) - V(\mathbf{x}_{j-1}^{\nu})}{\mathbf{e}_{\sigma(j)} \cdot (\mathbf{x}_j^{\nu} - \mathbf{x}_{j-1}^{\nu})} f_{\sigma(j)}(\mathbf{x}_i^{\nu}) + E_i^{\nu} \sum_{j=1}^{n} \left| \frac{V(\mathbf{x}_j^{\nu}) - V(\mathbf{x}_{j-1}^{\nu})}{\mathbf{e}_{\sigma(j)} \cdot (\mathbf{x}_j^{\nu} - \mathbf{x}_{j-1}^{\nu})} \right|, \tag{9}$$

for $i = 0, 1, \ldots, n$, where

$$E_i^{\nu} = \frac{1}{2} \sum_{r,s=1}^{n} B_{r,s}^{\nu} A_{r,i}^{\sigma} (A_{s,i}^{\sigma} + A_{s,n}^{\sigma}) \tag{10}$$

and

$$A_{k,i}^{\sigma} := |\mathbf{e}_k \cdot (\mathbf{x}_i^{\nu} - \mathbf{x}_0^{\nu})|$$

for $k = 1, 2, \ldots, n$ and $i = 0, 1, \ldots, n$.

Note that $A_{k,i}^{\sigma} = P_k$ if $k \in \{\sigma(1), \sigma(2), \ldots, \sigma(i)\}$ and $A_{k,i}^{\sigma} = 0$ otherwise and that $A_{s,n}^{\sigma} = C_{\mathbf{x}_0^{\nu},s}^{\nu}$ for all $\nu = (\mathbf{z}, \sigma)$. Further,

$$\mathbf{e}_{\sigma(j)} \cdot (\mathbf{x}_j^{\nu} - \mathbf{x}_{j-1}^{\nu}) = \mathbf{e}_{\sigma(j)} \cdot P \mathbf{e}_{\sigma(j)} = P_{\sigma(j)},$$

which simplifies the formulas even further, and the $A_{k,i}^{\sigma}$ only depend on σ in $\nu = (\mathbf{z}, \sigma)$. Hence, the $A_{k,i}^{\sigma}$, for all permutations σ of $\{1, 2, \ldots, n\}$, all $k = 1, 2, \ldots, n$ and all $i = 0, 1, \ldots, n$, can be computed once and for all before the verification of (9) for all the simplices $S_{\nu} \in \mathcal{T}_{\mathrm{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$.

3 EFFICIENT CPA CONSTRAINTS VERIFICATION

For the implementation of the simplicial complexes we use two grids. The first one is of type `struct ZGrid` and models the vertices of $\mathcal{T}_{\mathrm{sc}}^{\mathbf{U}}$ and the second is of type `struct xGrid` and models the vertices of $\mathcal{T}_{\mathrm{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$. Their declarations and the definition of `xGrid` are given in Listing 3. The implementation of `ZGrid` was discussed in (Hafstein, 2013) (as `class Grid`). Its main purpose here is to enable linear indexing of the grid points, i.e. `G.V2I(i)` for `i=0,1,...,G.NrPoints()-1` iterates through all the grid points of `G`. The implementation of `xGrid` is essentially trivial, just note that `/` and `%` are componentwise division and multiplication of vectors in Armadillo, respectively; similar to `./` and `.*` in Matlab. The correspondence between the variables and the mathematical symbols is `xL` $= \boldsymbol{\ell}$, `xU` $= \mathbf{u}$, `iU` $= \mathbf{U}$ and `hv` $= (P_1, P_2, \ldots, P_n)$.

The verification is then implemented in the function `VerifyLya`, see Listing 4. The arguments of the function are `System *psys`, a pointer to a concrete system that implements `psys->f` and `psys->B`, see Listing 2. The argument `const vec &V` is a vector containing the values of the function V at the vertices of the triangulation $\mathcal{T}_{\mathrm{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$ modelled by the argument `xGrid xG`. More exactly, `V(i)` is the value of V at the vertex `xG.I2vec(i)` (index to vector). If V fails the condition (9) for some permutation σ in $\nu = (\mathbf{z}, \sigma)$, then the index of the vertex $\mathbf{x} = P\mathbf{z} + \boldsymbol{\ell}$ is written to the vector `vector<bint> &Failed`, that is also an argument to the function. We do not keep track of the permutation σ, and thus the exact simplices, where the condition fails. This is faster and requires less memory and usually the information about the cube $P(\mathbf{z} + [0.1]^n) + \boldsymbol{\ell}$ containing a simplex where the Lyapunov function candidate V fails to have a negative orbital derivative is detailed enough. The vector `Failed` is sorted before the function returns.

A few comments on the implementation:

- The code is more optimized for size than for speed. The reason is that the computation of the values in `V` typically takes much more time than the verification anyways. Storing the values of `psys->f` at all vertices of $\mathcal{T}_{\mathrm{sc},\boldsymbol{\ell},\mathbf{u}}^{\mathbf{U}}$ would speed up the verification, but the storing needs n times the memory needed to store the values in `V`, where n is the dimension of the system, and would introduce a new limiting factor.
- It is important when verifying the condition (9) to not use `orbder + errbound > 0.0` to check if the condition fails on a simplex, but to instead use `!(orbder + errbound <= 0.0)`. If `orbder+errbound` is not `NaN` (not a number) then both are equivalent. However, since the methods used to compute the values in `V`, i.e. the values of V at the vertices, sometimes produce `NaN`, there is an important difference. A comparison with `NaN` always produces `false`, even `NaN == NaN` is `false`, but we want `orbder + errbound > 0.0` to deliver `true` if `orbder+errbound` is `NaN` and this is achieved with `!(orbder + errbound <= 0.0)`.
- We use the infamous `goto` to break out of nested loops when we have identified a cube

  $$P(\mathbf{z} + [0.1]^n) + \boldsymbol{\ell} \supset S_{\nu}$$

  containing a simplex S_{ν} where the condition (9) fails; the alternatives are simply more messy.
- Finally, it might seem at first glance that the indices of the vertices where the condition (9) fails are already sorted, and thus an extra sorting at the end of the function is unnecessary. A closer inspection, however, reveals that they are already sorted with respect to the order of the vertices in `ZGrid ROG(xG.G.L, xG.G.U - ones<ivec>(n))`, over which the algorithm iterates to obtain the $\mathbf{z}$s

Listing 3: Code for the grids.

```
// for L,U in Z^n all z in Z^n fulfilling L <= z <= U component-wise
struct ZGrid {
    ivec L,U;
    bint EndIndex;
    ZGrid(const ivec &_L, const ivec &_U);
    bint V2I(ivec) const;  // ivec to index
    ivec I2V(bint) const;  // index to ivec
    bint NrPoints(void);
};

struct xGrid {
    ZGrid G;
    vec xL, xU, hv;
    vec I2vec(bint Index) const{       // index to vec
        return xL + conv_to<vec>::from(G.I2V(Index)) % hv;
    }
    xGrid(const vec &_xL, const vec &_xU, const ivec &iU)
      : xL(_xL), xU(_xU), G(zeros<ivec>(n), iU){
        hv = (xU-xL) / conv_to<vec>::from(iU);
    }
};
```

in $\nu = (\mathbf{z}, \sigma)$, but not with respect to the order of the vertices in xG modeling $\mathcal{T}^{\hat{\mathbf{U}}}_{\mathrm{sc},\boldsymbol{\ell},\mathbf{u}}$.

To give an idea of the performance of the algorithm we computed on a Threadripper 3990X a Lyapunov function candidate using the RBF method (Giesl, 2007). We used 12,480 collocation points and the computation of the parameters for the function took 3.5 sec, i.e. writing and solving a system of 12,480 linear equations in 12,480 unknowns. The values of V are then computed at 1,002,001 vertices in 4.9 sec. Finally, the negativity of the orbital derivative in 2,000,000 simplices was verified in 0.026 sec; see Figure 1 for the results.

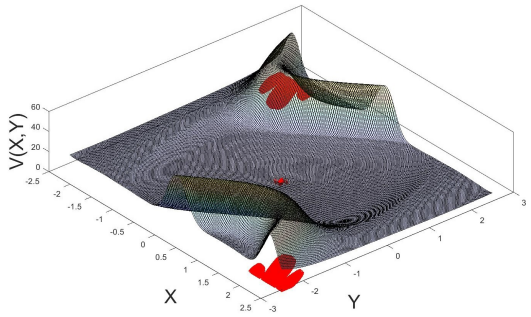

Figure 1: Lyapunov function candidate for system (7) and the area of its domain where the function is not decreasing along solution trajectories (red).

4 CONCLUSIONS

We presented an algorithm to verify the negativity of the orbital derivative of CPA Lyapunov function candidates interpolated over shape-regular triangulations. By carefully choosing a regular triangulation the algorithm is very fast and memory efficient. Sublevel sets of Lyapunov functions are positively invariant sets and this can be efficiently computed for CPA Lyapunov functions (Giesl et al., 2020). We gave an implementation in C++ of the algorithm and demonstrated its applicability and effectiveness for an example.

ACKNOWLEDGEMENT

The research done for this paper was partially supported by the Icelandic Research Fund in the grant 228725-051, *Linear Programming as an Alternative to LMIs for Stability Analysis of Switched Systems*, which is gratefully acknowledged.

Listing 4: Code for the verification.

```
bint factorial(int k) { bint f=1; for (int i = 1; i <= k; i++) f*=i; return f; }

void VerifyLya(System *psys, const vec &V, xGrid xG, vector<bint> &Failed,
                                                              int NrThreads) {
    vector<int> iv(n);
    for (int i = 0; i < n; i++) iv[i] = i;
    vector<vector<int>> Sigma;
    Sigma.reserve(factorial(n));
    do {
        Sigma.push_back(iv);
    } while (next_permutation(&iv[0], &iv[n]));
    vector<mat> A;
    A.reserve(factorial(n));
    for (auto psigma = Sigma.begin(); psigma != Sigma.end(); psigma++) {
        mat rA(n, n + 1);
        vec sx = zeros<vec>(n);
        for (int r = 0; r < n; r++) {
            rA(r, 0) = 0.0;
            double valrA;
            for (int i = 0; i < n; i++) {
                if ((*psigma)[i] == r) valrA = xG.hv(r);
                rA(r, i + 1) = valrA;
            }
        }
        A.push_back(rA);
    }

    wall_clock clock; clock.tic();
    ZGrid ROG(xG.G.L, xG.G.U - ones<ivec>(n));
    cout << "Verifying " << ROG.NrPoints() * factorial(n) << " simplices" << endl;

    vector<vector<int>> failedPart;
    failedPart.resize(NrThreads);

    function<void(bint)> parfor = [&](bint Th) {
        bint lowB = (ROG.NrPoints() * Th) / NrThreads;
        bint upB = (ROG.NrPoints() * (Th + 1)) / NrThreads;
        ivec z, zx;
        vec gradV(n);
        vec fxi(n), fx0(n), x0(n), xi(n);
        mat B(n, n);
        double V0, Vold, Vnew;
        int r, s, i, j, k;
        bint I0, Inew;
        double orbder, grbound, errbound;

        for (bint pnr = lowB; pnr < upB; pnr++) {  // z
            z = ROG.I2V(pnr);
            I0 = xG.G.V2I(z);
            x0 = xG.I2vec(I0);
            fx0 = psys->f(x0);
            V0 = V(I0);
            for (r = 0; r < n; r++) {
                for (s = 0; s < r; s++) {
                    B(s, r) = B(r, s) = psys->B(r, s, x0, x0 + xG.hv);
                }
                B(r, r) = psys->B(r, r, x0, x0 + xG.hv);
            }
```

Listing 4: Code for the verification (cont.).

```
            for (k = 0; k < Sigma.size(); k++) {  // sigma
                vector<int> psigma = Sigma[k];
                Vold = V0;
                zx = z;
                for (int j = 0; j < n; j++) {  // nabla V
                    zx(psigma[j]) += 1;
                    Inew = xG.G.V2I(zx);
                    Vnew = V(Inew);
                    gradV(psigma[j]) = (Vnew - Vold) / xG.hv(psigma[j]);
                    Vold = Vnew;
                }

                // i=0
                orbder = dot(gradV, fx0);
                if (!(orbder <= 0.0)) {  // errbound=0.0
                    goto CUBEFAILED;
                }
                grbound = norm(gradV, 1);
                zx = z;
                for (int i = 0; i < n; i++) {
                    zx(psigma[i]) += 1;
                    Inew = xG.G.V2I(zx);
                    xi = xG.I2vec(Inew);
                    fxi = psys->f(xi);
                    orbder = dot(gradV, fxi);
                    errbound = 0.0;
                    for (r = 0; r < n; r++) {
                        for (s = 0; s < n; s++) {
                            errbound += B(r, s) * A[k](r, i + 1) *
                                        (A[k](s, i + 1) + A[k](s, n));
                        }
                    }
                    errbound *= 0.5 * grbound;
                    if (!(orbder + errbound <= 0.0)) {
                        goto CUBEFAILED;
                    }
                }
            }
            continue;
        CUBEFAILED:
            failedPart[Th].push_back(I0);
        }
    };

    ParallelFor(NrThreads, parfor, NrThreads);
    Failed.clear();
    bint NrFailed = 0;
    for (int i = 0; i < NrThreads; i++) {
        NrFailed += failedPart[i].size();
    }
    Failed.reserve(NrFailed);
    for (int i = 0; i < NrThreads; i++) {
        Failed.insert(Failed.end(), failedPart[i].begin(), failedPart[i].end());
    }
    sort(Failed.begin(), Failed.end());
}
```

REFERENCES

Albertsson, S., Giesl, P., Gudmundsson, S., and Hafstein, S. (2020). Simplicial complex with approximate rotational symmetry: A general class of simplicial complexes. *J. Comput. Appl. Math.*, 363:413–425.

Anderson, J. and Papachristodoulou, A. (2015). Advances in computational Lyapunov analysis using sum-of-squares programming. *Discrete Contin. Dyn. Syst. Ser. B*, 20(8):2361–2381.

Björnsson, J., Giesl, P., Hafstein, S., Kellett, C., and Li, H. (2014). Computation of continuous and piecewise affine Lyapunov functions by numerical approximations of the Massera construction. In *Proceedings of the CDC, 53rd IEEE Conference on Decision and Control*, pages 5506–5511, Los Angeles (CA), USA.

Björnsson, J., Giesl, P., Hafstein, S., Kellett, C., and Li, H. (2015). Computation of Lyapunov functions for systems with multiple attractors. *Discrete Contin. Dyn. Syst. Ser. A*, 35(9):4019–4039.

Björnsson, J. and Hafstein, S. (2017). Efficient Lyapunov function computation for systems with multiple exponentially stable equilibria. *Procedia Computer Science*, 108:655–664. Proceedings of the International Conference on Computational Science (ICCS), Zurich, Switzerland, 2017.

Chesi, G. (2011). *Domain of Attraction: Analysis and Control via SOS Programming*. Lecture Notes in Control and Information Sciences, vol. 415, Springer.

Doban, A. (2016). *Stability domains computation and stabilization of nonlinear systems: implications for biological systems*. PhD thesis: Eindhoven University of Technology.

Doban, A. and Lazar, M. (2016). Computation of Lyapunov functions for nonlinear differential equations via a Yoshizawa-type construction. *IFAC-PapersOnLine*, 49(18):29 – 34.

Giesl, P. (2007). *Construction of Global Lyapunov Functions Using Radial Basis Functions*. Lecture Notes in Math. 1904, Springer.

Giesl, P. and Hafstein, S. (2014). Revised CPA method to compute Lyapunov functions for nonlinear systems. *J. Math. Anal. Appl.*, 410:292–306.

Giesl, P. and Hafstein, S. (2015a). Computation and verification of Lyapunov functions. *SIAM J. Appl. Dyn. Syst.*, 14(4):1663–1698.

Giesl, P. and Hafstein, S. (2015b). Review of computational methods for Lyapunov functions. *Discrete Contin. Dyn. Syst. Ser. B*, 20(8):2291–2331.

Giesl, P. and Hafstein, S. (2021a). System specific triangulations for the construction of CPA Lyapunov functions. *Discrete Contin. Dyn. Syst. Ser. B*, 26(12):6027–6046.

Giesl, P. and Hafstein, S. (2021b). Uniformly regular triangulations for parameterizing Lyapunov functions. In *Proceedings of the 18th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, pages 549–557.

Giesl, P., Osborne, C., and Hafstein, S. (2020). Automatic determination of connected sublevel sets of CPA Lyapunov functions. *SIAM J. Appl. Dyn. Syst.*, 19(2):1029–1056.

Gudmundsson, S. and Hafstein, S. (2015). Lyapunov function verification: MATLAB implementation. In *Proceedings of the 1st Conference on Modelling, Identification and Control of Nonlinear Systems (MICNON)*, number 0235, pages 806–811.

Hafstein, S. (2004). A constructive converse Lyapunov theorem on exponential stability. *Discrete Contin. Dyn. Syst. Ser. A*, 10(3):657–678.

Hafstein, S. (2013). Implementation of simplicial complexes for CPA functions in C++11 using the Armadillo linear algebra library. In *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, pages 49–57, Reykjavik, Iceland.

Hafstein, S. (2019). *Computational Science - ICCS 2019: 19th International Conference, Faro, Portugal, June 12-14, 2019, Proceedings, Part V*, chapter Numerical Analysis Project in ODEs for Undergraduate Students, pages 412–434. Springer.

Hafstein, S., Kellett, C., and Li, H. (2014a). Computation of Lyapunov functions for discrete-time systems using the Yoshizawa construction. In *Proceedings of 53rd IEEE Conference on Decision and Control (CDC)*.

Hafstein, S., Kellett, C., and Li, H. (2014b). Continuous and piecewise affine Lyapunov functions using the Yoshizawa construction. In *Proceedings of the 2014 American Control Conference (ACC)*, pages 548–553 (no. 0170), Portland (OR), USA.

Hafstein, S., Kellett, C., and Li, H. (2015). Computing continuous and piecewise affine Lyapunov functions for nonlinear systems. *J. Comput. Dyn*, 2(2):227 – 246.

Hafstein, S. and Valfells, A. (2017). Study of dynamical systems by fast numerical computation of Lyapunov functions. In *Proceedings of the 14th International Conference on Dynamical Systems: Theory and Applications (DSTA)*, volume Mathematical and Numerical Aspects of Dynamical System Analysis, pages 220–240.

Hafstein, S. and Valfells, A. (2019). Efficient computation of Lyapunov functions for nonlinear systems by integrating numerical solutions. *Nonlinear Dynamics*, 97(3):1895–1910.

Hahn, W. (1967). *Stability of Motion*. Springer, Berlin.

Julian, P. (1999). *A High Level Canonical Piecewise Linear Representation: Theory and Applications*. PhD thesis: Universidad Nacional del Sur, Bahia Blanca, Argentina.

Julian, P., Guivant, J., and Desages, A. (1999). A parametrization of piecewise linear Lyapunov functions via linear programming. *Int. J. Control*, 72(7-8):702–715.

Kamyar, R. and Peet, M. (2015). Polynomial optimization with applications to stability analysis and control – an alternative to sum of squares. *Discrete Contin. Dyn. Syst. Ser. B*, 20(8):2383–2417.

Khalil, H. (2002). *Nonlinear Systems*. Pearson, 3. edition.

Li, H., Hafstein, S., and Kellett, C. (2015). Computation of continuous and piecewise affine Lyapunov functions

for discrete-time systems. *J. Difference Equ. Appl.*, 21(6):486–511.

Lyapunov, A. M. (1992). The general problem of the stability of motion. *Internat. J. Control*, 55(3):521–790. Translated by A. T. Fuller from Édouard Davaux's French translation (1907) of the 1892 Russian original, With an editorial (historical introduction) by Fuller, a biography of Lyapunov by V. I. Smirnov, and the bibliography of Lyapunov's works collected by J. F. Barrett, Lyapunov centenary issue.

Marinósson, S. (2002a). Lyapunov function construction for ordinary differential equations with linear programming. *Dynamical Systems: An International Journal*, 17:137–150.

Marinósson, S. (2002b). *Stability Analysis of Nonlinear Systems with Linear Programming: A Lyapunov Functions Based Approach*. PhD thesis: Gerhard-Mercator-University Duisburg, Duisburg, Germany.

Parrilo, P. (2000). *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. PhD thesis: California Institute of Technology Pasadena, California.

Ratschan, S. and She, Z. (2010). Providing a basin of attraction to a target region of polynomial systems by computation of Lyapunov-like functions. *SIAM J. Control Optim.*, 48(7):4377–4394.

Sanderson, C. (2010). Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA.

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

Sastry, S. (1999). *Nonlinear Systems: Analysis, Stability, and Control*. Springer.

Valmorbida, G. and Anderson, J. (2017). Region of attraction estimation using invariant sets and rational Lyapunov functions. *Automatica*, 75:37–45.

Vannelli, A. and Vidyasagar, M. (1985). Maximal Lyapunov functions and domains of attraction for autonomous nonlinear systems. *Automatica*, 21(1):69–80.

Vidyasagar, M. (2002). *Nonlinear System Analysis*. Classics in Applied Mathematics. SIAM, 2. edition.

Yoshizawa, T. (1966). *Stability theory by Liapunov's second method*. Publications of the Mathematical Society of Japan, No. 9. The Mathematical Society of Japan, Tokyo.

Zubov, V. I. (1964). *Methods of A. M. Lyapunov and their application*. Translation prepared under the auspices of the United States Atomic Energy Commission; edited by Leo F. Boron. P. Noordhoff Ltd, Groningen.