Fast Algorithms for Computing Continuous Piecewise Affine Lyapunov Functions

Sigurdur Hafstein(✉)

University of Iceland, Dunhagi 5, 107 Reykjavik, Iceland
`shafstein@hi.is`
`https://notendur.hi.is/shafstein/`

Abstract. Algorithms that parameterize continuous and piecewise affine Lyapunov functions for nonlinear systems, both in continuous and discrete time, have been proposed in numerous publications. These algorithms generate constraints that are linear in the values of a function at all vertices of a simplicial complex. If these constraints are fulfilled for certain values at the vertices, then they can be interpolated on the simplices to deliver a function that is a Lyapunov function for the system used for their generation. There are two different approaches to find values that fulfill the constraints. First, one can use optimization to compute appropriate values that fulfill the constraints. These algorithms were originally designed for continuous-time systems and their adaptation to discrete-time systems and control systems poses some challenges in designing and implementing efficient algorithms and data structures for simplicial complexes. Second, one can use results from converse theorems in the Lyapunov stability theory to generate good candidates for suitable values and then verify the constraints for these values. In this paper we study several efficient data structures and algorithms for these computations and discuss their implementations in C++.

Keywords: Simplicial complex · Algorithm · Lyapunov function
Nonlinear system

1 Introduction

A Lyapunov function V for a dynamical system is a continuous function from the state-space to the real numbers that has a minimum at an equilibrium and is decreasing along the system's trajectories. For a continuous-time system given by a differential equation $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ the decrease along solution trajectories can be ensured by the condition

$$D_{\mathbf{f}}V(\mathbf{x}) := \nabla V(\mathbf{x}) \bullet \mathbf{f}(\mathbf{x}) < 0. \tag{1}$$

For a discrete-time system $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ the corresponding condition is

$$\Delta_{\mathbf{g}}V(\mathbf{x}) := V(\mathbf{g}(\mathbf{x})) - V(\mathbf{x}) < 0. \tag{2}$$

M. S. Obaidat et al. (Eds.): SIMULTECH 2017, AISC 873, pp. 274–299, 2019.
https://doi.org/10.1007/978-3-030-01470-4_15

In [13,19,32] novel algorithms for the computation of Lyapunov functions for nonlinear discrete-time systems were presented. In these algorithms the relevant part of the state-space is first triangulated, i.e. subdivided into simplices, and then a continuous and piecewise affine (CPA) Lyapunov function is parameterized by fixing its values at the vertices of the simplices. These algorithms resemble earlier algorithms for the computation of Lyapunov functions for nonlinear continuous-time systems, cf. e.g. [4,15,16,24,25,33,34], referred to as the CPA algorithm. The essential idea is to formulate the conditions for a Lyapunov function as linear constraints in the values of the Lyapunov function to be computed at the vertices of the simplices of the simplicial complex.

The implementation of these algorithms for discrete-time systems can be done similarly to the continuous-time case. First a simplicial complex is constructed that triangulates the relevant part of the state-space. Then an appropriate linear programming problem for the system at hand is generated, of which any feasible solution parameterizes a Lyapunov function for the system. Then one either uses a linear programming solver, e.g. GLPK or Gurobi, to search for a feasible solution, or one uses results from converse theorems in the Lyapunov stability theory to compute values that can be expected to fulfill the constraints and then verifies if these computed values constitute a feasible solution to the linear programming problem.

The non-locality of the dynamics in the discrete-time case, however, poses an additional challenge in implementing the construction of a suitable simplicial complex and the generation of the linear constraints in an efficient way. Namely, whereas the condition (1) for a continuous-time system is a local condition that can be formulated as linear constraints for each simplex, the condition (2) for a discrete-time system is not local. For a vertex $\mathbf{x}$ of a simplex $\mathfrak{S}_\nu$ in the triangulation $\mathcal{T}$ we must be able find a simplex $\mathfrak{S}_\mu \in \mathcal{T}$ such that $\mathbf{g}(\mathbf{x}) \in \mathfrak{S}_\mu$ to formulate this condition as linear constraints and we must know the barycentric coordinates of $\mathbf{g}(\mathbf{x})$ in $\mathfrak{S}_\mu$. For triangulations consisting of many simplices a linear search is very inefficient and therefore more advanced methods are called for.

The first contribution of this paper is an algorithm that efficiently searches for a simplex $\mathfrak{S}_\mu \in \mathcal{T}$ such that $\mathbf{x} \in \mathfrak{S}_\mu$ and computes its barycentric coordinates for fairly general simplicial complexes, that were specially designed for our problem of computing Lyapunov functions.

The CPA algorithm has additionally been adapted to compute Lyapunov functions for differential inclusions [2] and control Lyapunov functions [3] in the sense of Clarke's subdifferential [9]. The next logical step is to compute control Lyapunov functions in the sense of the Dini subdifferential, a work in progress with promising first results. For these computations one needs information on the common faces of neighbouring simplices in the simplicial complex and detailed information on normals of the hyperplanes separating neighbouring simplices. Efficient algorithms and data structures for these computations are presented. This is the second contribution of this paper.

The third contribution is an algorithms to compute circumscribing hyperspheres of the simplices of the simplicial complex. These can be used to implement more advanced algorithms for the computation of Lyapunov functions for discrete-time systems, also a work in progress.

The fourth contribution is an efficient algorithm combining the four-step Adam-Bashforth method for initial-value problems and Simpson's Rule for numerical integration to approximate values of a Lyapunov function from a converse theorem in the Lyapunov stability theory [35] at the vertices of the simplicial complex. We also undertake a detailed error analysis of our approach.

The first three contributions are the same as in [18] but improved and advanced in numerous ways. We discuss more general functions $\mathbf{F}$ in Sect. 3 than in [18] and in all three the algorithms and data structures have been tweaked for performance. For example, Stroustrup's statements, e.g. [42], motivated us to replace linked lists with vectors in several places, and because the old code relied on member functions of `stl::list` the code had to be adapted to use `stl::vector` and/or suitable functions from `stl::algorithm` where appropriate. Further, care must be taken to avoid methods that are inheritably inefficient for vectors. We try to keep the discussion largely self-contained, but to keep it at a reasonable length we avoid repetitions of material presented in [18] that is not necessary to understand the approach here. We thus refer to [18] for numerous issues and more detailed results and keep the same notation. The fourth contribution has not been published in any form before.

Before we describe our algorithms in Sects. 2.1 and 3, we first discuss suitable triangulations for the computation of CPA Lyapunov functions in Sect. 2. In Sect. 4 we describe, analyze and give the implementation of our method to approximate a Lyapunov function from a converse theorem at the vertices of a simplicial complex and in Sect. 5 we sum up the contributions give a few concluding remarks.

1.1 Notation

We denote by $\mathbb{Z}$, $\mathbb{N}_0$, $\mathbb{R}$, and $\mathbb{R}_+$ the sets of the integers, the nonnegative integers, the real numbers, and the nonnegative real numbers respectively. For integers $r, s \in \mathbb{Z}$, $r < s$, we write $r : s$ for $r, r+1, \ldots, s$. We write vectors in boldface, e.g. $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{Z}^n$, and their components as $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$. All vectors are assumed to be column vectors unless specified otherwise. An inequality for vectors is understood to be component-wise, e.g. $\mathbf{x} < \mathbf{y}$ means that all the inequalities $x_1 < y_2$, $x_2 < y_2, \ldots, x_n < y_n$ are fulfilled. The null vector in $\mathbb{R}^n$ is written as $\mathbf{0}$ and the standard orthonormal basis as $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$, i.e. the i-th component of $\mathbf{e}_j$ is equal to $\delta_{i,j}$, where $\delta_{i,j}$ is the Kronecher delta, equal to 1 if $i = j$ and 0 otherwise. The scalar product of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is denoted by $\mathbf{x} \bullet \mathbf{y}$, the Euclidean norm of $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_2 := \sqrt{\mathbf{x} \bullet \mathbf{x}}$, and the maximum norm of $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_\infty := \max_{i=1:n} |x_i|$. The transpose of a vector $\mathbf{x}$ is denoted by $\mathbf{x}^T$ and similarly the transpose of a matrix $A \in \mathbb{R}^{n \times m}$ is denoted by A^T. For a nonsingular matrix $A \in \mathbb{R}^{n \times n}$ we denote its inverse by A^{-1} and the inverse of its transpose by A^{-T}. This should not lead to misunderstandings since $(A^{-1})^T = (A^T)^{-1}$. In the rest of the paper n and in the code the global variable `const int n` is the dimension of the Euclidean space we are working in.

We write sets $\mathcal{K} \subset \mathbb{R}^n$ in calligraphic and we denote the closure, interior, and the boundary of $\mathcal{K}$ by $\overline{\mathcal{K}}$, $\mathcal{K}^\circ$, and $\partial\mathcal{K}$ respectively.

The *convex hull* of an $(m+1)$-tuple $(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$ of vectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^n$ is defined by

$$\mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m) := \left\{ \sum_{i=0}^{m} \lambda_i \mathbf{v}_i : 0 \leq \lambda_i, \sum_{i=0}^{m} \lambda_i = 1 \right\}.$$

If $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^n$ are affinely independent, i.e. the vectors $\mathbf{v}_1 - \mathbf{v}_0, \mathbf{v}_2 - \mathbf{v}_0, \ldots, \mathbf{v}_m - \mathbf{v}_0$ are linearly independent, the set $\mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$ is called an *m-simplex*. For a subset $\{\mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k}\}$, $0 \leq k < m$, of affinely independent vectors $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m\}$, the k-simplex $\mathrm{co}(\mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k})$ is called a *k-face* of the simplex $\mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$. Note that simplices are usually defined as convex combinations of vectors in a set and not of ordered tuples, i.e. $\mathrm{co}\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m\}$ rather than $\mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$. For the implementation of the simplicial complexes below it is however very useful to stick to ordered tuples. A function $\rho : \mathbb{R}_+ \to \mathbb{R}_+$ is said to be of class $\mathcal{K}_\infty$ if it is continuous, strictly increasing, and fulfills $\rho(0) = 0$ and $\lim_{x\to\infty} \rho(x) = \infty$.

In the implementations of the algorithms we make heavy use of the Standard C++ Library and the Armadillo linear algebra library [40]. We thus always assume in the code that `using namespace std` and `using namespace arma` have been declared. Further, we assume that all the necessary libraries are accessible. Very good documentation on Armadillo is available at http://arma.sourceforge.net and some comments on its use for the implementation of the basic simplicial complex in Sect. 2 are also given in [17]. The vector and matrix types of Armadillo we use in this paper are `ivec`, `vec`, and `mat`, which represent a column vector of `int`, a column vector of `double`, and a matrix of `double` respectively.

2 The Simplicial Complex $\mathcal{T}_{N,K}^{\mathrm{std}}$

In [17] the simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ and its efficient implementation is elaborately described. For completeness we recall its definition but refer to [17] for the details. To define the simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ we first need several preliminary definitions.

An *admissible triangulation* of a set $\mathcal{C} \subset \mathbb{R}^n$ is the subdivision of $\mathcal{C}$ into n-simplices, such that the intersection of any two different simplices in the subdivision is either empty or a common k-face, $0 \leq k < n$. Such a structure is often referred to as a *simplicial n-complex*.

For the definition of $\mathcal{T}_{N,K}^{\mathrm{std}}$ we use the set S_n of all permutations of the set $\{1 : n\}$, the characteristic functions $\chi_{\mathcal{J}}(i)$ equal to one if $i \in \mathcal{J}$ and equal to zero if $i \notin \mathcal{J}$. Further, we use the functions $\mathbf{R}^{\mathcal{J}} : \mathbb{R}^n \to \mathbb{R}^n$, defined for every $\mathcal{J} \subset \{1 : n\}$ by

$$\mathbf{R}^{\mathcal{J}}(\mathbf{x}) := \sum_{i=1}^{n} (-1)^{\chi_{\mathcal{J}}(i)} x_i \mathbf{e}_i.$$

Thus $\mathbf{R}^{\mathcal{J}}(\mathbf{x})$ puts a minus in front of the i-th coordinate of $\mathbf{x}$ whenever $i \in \mathcal{J}$.

To construct the triangulation $\mathcal{T}_{N,K}^{\mathrm{std}}$, we first define the triangulations $\mathcal{T}_{N}^{\mathrm{std}}$ and $\mathcal{T}_{K,\mathrm{fan}}^{\mathrm{std}}$ as intermediate steps.

Definition of $\mathcal{T}_{N,K}^{\mathrm{std}}$

1. For every $\mathbf{z} \in \mathbb{N}_0^n$, every $\mathcal{J} \subset \{1 : n\}$, and every $\sigma \in S_n$ define the simplex

$$\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} := \mathrm{co}(\mathbf{x}_0^{\mathbf{z}\mathcal{J}\sigma}, \mathbf{x}_1^{\mathbf{z}\mathcal{J}\sigma}, \ldots, \mathbf{x}_n^{\mathbf{z}\mathcal{J}\sigma}) \tag{3}$$

   where

$$\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} := \mathbf{R}^{\mathcal{J}}\left(\mathbf{z} + \sum_{j=1}^{i} \mathbf{e}_{\sigma(j)}\right) \quad \text{for } i = 0 : n. \tag{4}$$

2. Let $\mathbf{N}^m, \mathbf{N}^p \in \mathbb{Z}^n$, $\mathbf{N}^m < \mathbf{0} < \mathbf{N}^p$, and define the hypercube $\mathcal{N} := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{N}^m \le \mathbf{x} \le \mathbf{N}^p\}$. The simplicial complex $\mathcal{T}_{N}^{\mathrm{std}}$ is defined by

$$\mathcal{T}_{N}^{\mathrm{std}} := \{\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} : \mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \subset \mathcal{N}\}. \tag{5}$$

3. Let $\mathbf{K}^m, \mathbf{K}^p \in \mathbb{Z}^n$, $\mathbf{N}^m \le \mathbf{K}^m < \mathbf{0} < \mathbf{K}^p \le \mathbf{N}^p$, and consider the intersections of the n-simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ in $\mathcal{T}_{N}^{\mathrm{std}}$ and the boundary of the hypercube $\mathcal{K} := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{K}^m \le \mathbf{x} \le \mathbf{K}^p\}$. We are only interested in those intersections that are $(n-1)$-simplices, i.e. intersections that can be written as $\mathrm{co}(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ with exactly n-vertices. For every such intersection add the origin as a vertex to it, i.e. consider $\mathrm{co}(\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$. The set of such constructed n-simplices is denoted $\mathcal{T}_{K,\mathrm{fan}}^{\mathrm{std}}$. It is a triangulation of the hypercube $\mathcal{K}$.
4. Finally, we define our main simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ by letting it contain all simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ in $\mathcal{T}_{N}^{\mathrm{std}}$, that have an empty intersection with the interior $\mathcal{K}^\circ$ of $\mathcal{K}$, and all simplices in the simplicial fan $\mathcal{T}_{K,\mathrm{fan}}^{\mathrm{std}}$. It is thus a triangulation of $\mathcal{N}$ having a simplicial fan in $\mathcal{K}$.

The triangulation $\mathcal{T}_{K,\mathrm{fan}}^{\mathrm{std}}$ of the hypercube $\mathcal{K} := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{K}^m \le \mathbf{x} \le \mathbf{K}^p\}$ is a straightforward extension of the 3D graphics primitive *triangular fan* to arbitrary dimensions. Therefore the term *simplicial fan*. For a graphical presentation of the complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ with $n = 2$ see Fig. 1 taken form [18]. For figures of the complex with $n = 3$ see Figs. 2 and 3 in [17].

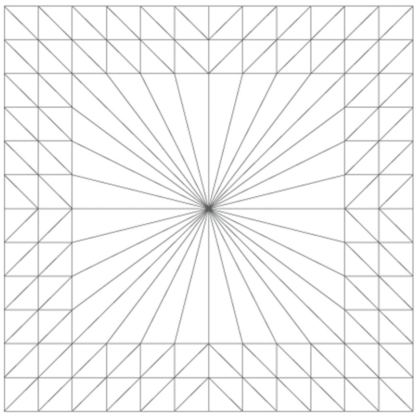

Fig. 1. The simplicial complex $\mathcal{T}^{\text{std}}_{N,K}$ in two dimensions with $\mathbf{K}^m = (-4,-4)^T, \mathbf{K}^p = (4,4)^T, \mathbf{N}^m = (-6,-6)^T$, and $\mathbf{N}^p = (6,6)^T$.

The class `T_std_NK` implements the basic simplicial complex $\mathcal{T}^{\text{std}}_{N,K}$ is. It is defined as follows:

```
class T_std_NK {
public:
  ivec Nm,Np,Km,Kp;
  Grid G;
  int Nr0;
  vector<ivec> Ver;
  vector<vector<int>> Sim;
  vector<zJs> NrInSim;
  vector<int> Fan;
  int InSimpNr(vec x);    // returns -1 if not found
  bool InSimp(vec x,int s);
  T_std_NK(ivec Nm,ivec Np,ivec Km,ivec Kp);
  // added since (Hafstein, 2013) below
  vector<vector<int>> SimN;
  vector<vector<int>> SCV;
  vector<vector<vector<int>>> Faces;
  vector<int> BSim;
  int SVerNr(int s,int i);
  ivec SVer(int s,int i);
};
```

`Nm` = $\mathbf{N}^m$ and `Np` = $\mathbf{N}^p$ define the hypercube $\mathcal{N}$ and `Km` = $\mathbf{K}^m$ and `Kp` = $\mathbf{K}^p$ define the hypercube $\mathcal{K}$. `vector<ivec> Ver` is a vector containing all the vertices of all the simplices in the complex, the value of `int Nr0` is such that `Ver[Nr0]` is the zero vector, and `vector<vector<int>> Sim` is a vector containing all the simplices of the complex. Each simplex is stored as a vector of $(n+1)$-integers, the integers referring to the positions of the vertices of the simplex in `vector<ivec> Ver`. `G` is a `Grid` initialized by `ivec Nm,Np` that is used to enumerate the vertices of `T_std_NK` and `vector<int> Fan` and `vector<zJs> NrInSim` are auxiliary structures that allow for a given $\mathbf{x} \in \mathbb{R}^n$ to efficiently compute an `int s` such that for $\mathfrak{S}_\nu =$`Sim[s]` we have $\mathbf{x} \in \mathfrak{S}_\nu$. Their properties and implementation is described in detail in [17]. `vector<vector<int>> SimN`, `vector<vector<int>> SCV`, `vector<vecotr<vector<int>>> Faces`, and `vector<int> BSimp` are variables in the class `T_std_NK` that have been added since [17]. Their purpose and initialization is described in the next section.

2.1 Added Functionality in `T_std_NK`

In this section we describe the functionality that has been added to the class `T_std_NK` since [17]. The added functionality is as described in [18] but the implementation has been made more efficient. To simplify the notation we write $\mathcal{T}$ for the basic simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ from now on. Further we denote the set of all its vertices by $\mathcal{V}_\mathcal{T}$ and its domain by $\mathcal{D}_\mathcal{T} := \bigcup_{\mathfrak{S}_\nu \in \mathcal{T}} \mathfrak{S}_\nu$.

In [17] a fast algorithm is given to compute a simplex $\mathfrak{S}_\nu \in \mathcal{T}$ such that $\mathbf{x} \in \mathfrak{S}_\nu$, but fell back on linear search if $\mathbf{x}$ was in the simplicial fan $\mathcal{T}_{K,\text{fan}}^{\text{std}}$ of $\mathcal{T}$. Under the premise that the simplicial fan is much smaller than the rest of the simplicial complex this is a reasonable strategy. Therefore we did not add a fast search structure `zJs` to `vector<zJs> NrInSim` for simplices in the fan. However, for a simplicial complex for which this premise is not fulfilled, an improved strategy shortens the search time considerably. This might also be of importance to different computational methods for Lyapunov functions that use conic partitions of the state-space, a topic that has obtained considerable attention cf. e.g. [1,8,22,23,28–31,36–39,44].

We achieve this by first adding the simplices in $\mathcal{T}_{K,\text{fan}}^{\text{std}}$ with appropriate values for $\mathbf{z}$, $\mathcal{J}$, and σ to `vector<zJs> NrInSim`. This is very simple to make: In `CODE BLOCK 1` in [17] directly after `Fan.push_back(SLE);` simply add

```
NrInSim.push_back(zJs(z,J,sigma,SLE));.
```

To actually find such simplices fast through their $\mathbf{z}$, $\mathcal{J}$, σ values a little more effort is needed. If $\mathbf{x} \in \mathbb{R}^n$ is in the simplicial fan of $\mathcal{T}$, i.e. if $\mathbf{K}^m < \mathbf{x} < \mathbf{K}^p$, we project $\mathbf{x}$ radially just below the boundary of the hypercube $\mathcal{K} := \{\mathbf{y} \in \mathbb{R}^n : \mathbf{K}^m \leq \mathbf{y} \leq \mathbf{K}^p\}$. Thus if originally $\mathbf{x} \in \text{co}(\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ its radial projection $\mathbf{x}_r$, $\mathbf{x}_r := r\mathbf{x}$ with an appropriate $r > 0$, will be in $\text{co}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$, where $\mathbf{v}_0$ is the vertex that was replaced by $\mathbf{0}$ as in step 3 in the definition of $\mathcal{T} = \mathcal{T}_{N,K}^{\text{std}}$. When we compute the appropriate $\mathbf{z}$, $\mathcal{J}$, and σ for $\mathbf{x}_r$ we will actually get the position of the simplex $\text{co}(\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$, because of the changes described above in `CODE BLOCK 1`.

```
int T_std_NK::InSimpNr(vec x){
  vec origx=x;
  if(!(min(Np-x)>=0.0 && min(x-Nm)>=0.0)){
    // not in the simplicial complex
    return -1;
  }
  if(min(Kp-x)>0.0 && min(x-Km)>0.0){
    // in the fan
    double eps = 1e-15;
    if(norm(x,"inf")>eps) {
      double r=numeric_limits<double>::max();
      for(int i=0;i<n;i++){
        if(abs(x(i))>eps){
          r=min(r,(x(i)>0 ? Kp(i):Km(i))/x(i));
        }
      }
      x *= r*(1-eps);
    }
    else{
      // be careful, use linear search
      for(int i=0;i<Fan.size();i++){
        if(InSimp(x,Fan[i])){
          return Fan[i];
        }
      }
    }
  }
  // compute the zJs of the simplex,
  // for details cf. (Hafstein, 2013)
  int J=0;
  ivec z(n),sigma;
  for(int i=0;i<n;i++){
    if(x(i)<0){
      x(i)=-x(i);
      J|=1<<i;
    }
    z(i)=static_cast<int>(x(i));
  }
  sigma=conv_to<ivec>::from(sort_index(x-z,1));
  // find and return the appropriate simplex
  auto found=
  equal_range(NrInSim.begin(),NrInSim.end(),zJs(z,J,sigma));
  // If one wants to be sure everything is OK
  assert(found.first!=found.second);
  assert(InSimp(origx,found.first->Pos));
  return found.first->Pos;
}
```

The simplices are stored as a `vector` of `vector<int>`, the integers being indices of vertices in `vector<ivec> Ver`. Thus `vector<vector<int>> Sim` contains the simplices in $\mathcal{T}$ and `Sim[s][i]` is the index of the i-th vertex of simplex number s in `Ver`. To make this access more transparent the member functions `int SVerNr(int s,int i)` and `ivec SVer(int s,int i)` were added:

```
int T_std_NK::SVerNr(int s,int i){
// returns a j such that Ver[j] is the
// i-th vertex of simplex Sim[s]
  return Sim[s][i];
}

ivec T_std_NK::SVer(int s,int i){
// returns the i-th vertex of simplex Sim[s]
  return Ver[SVerNr(s,i)];
}
```

To be able to use the Standard C++ Library functions `set_intersection` and `set_difference` we sort each `vector<int> Sim[s]`. This is implemented in the constructor of `T_std_NK` in the trivial way:

```
for(int s=0;s<Sim.size();s++){
  sort(Sim[s].begin(),Sim[s].end());
}
```

The `vector<vector<int>> SimN` contains the neighbouring simplices for each simplex and `vector<vector<int>> SCV` contains all simplices, of which a particular vertex in $\mathcal{V}_\mathcal{T}$ is a vertex of. More exactly `SimN[s]` is a sorted vector of the indices in `Sim` of the simplices neighbouring simplex `Sim[s]` (not including `s` itself) and `SCV[i]` is a sorted vector of the indices in `Sim` of the simplices, of which `Ver[i]` is a vertex. They are constructed as follows in the constructor of `T_std_NK`:

```
SCV.resize(Vertices.size());
for(int s=0;s<Sim.size();s++){
  for(int i=0;i<=n;i++){
    SCV[SVerNr(s,i)].push_back(s);
  }
}

vector<vector<int>>::iterator pSCV;
for(pSCV=SCV.begin();pSCV!=SCV.end();pSCV++){
  sort((*pSCV).begin(),(*pSCV).end());
}

SimN.resize(Sim.size())
for(int s=0;s<SimN.size();s++){
  list<int> lSimN;
  for(int i=0;i<=n;i++){
    lSimN.insert(lSimN.end(),
      SCV[SVerNr(s,i)].begin(),
        SCV[SVerNr(s,i)].end());
  }
  lSimN.sort();
  lSimN.unique();
  lSimN.remove(s);
  SimN[s].assign(lSimN.begin(),lSimN.end());
}
```

For every neighbouring simplex `Sim[k]` of the simplex `Sim[s]` we keep track of the common face. The member `vector<vector<vector<int>>> Faces` of `T_std_NK` was added for this purpose. A face is stored as a `vector<int>` of the indices of its vertices in `vector<ivec> Ver`. Each `Faces[s]` is a vector containing the faces the simplex `Sim[s]` shares with other simplices in $\mathcal{T}$ and they are listed in the same order as in `SimN[s]`, i.e.

```
vector<int>::iterator pSN=SimN[s].begin();
vector<vector<int>>::iterator p=Faces[s].begin();
for(;pSN!=SimN[s].end();pSN++,p++){
  // here (*p) is a vector<int> containing the
  // indices in Ver of the vertices of the
  // common face of Sim[s] and (*pSN).
}
```

The vector `Faces` is built as follows in the constructor of `T_std_NK`:

```
Faces.resize(Simp.size());
for(int s=0;s<Faces.size();s++){
  for(auto p=SimN[s].begin();p!=SimN[s].end();p++){
    vector<int> F(n);   // Face
    auto Fend=set_intersection(Sim[*p].begin(),Sim[*p].end(),
        Sim[s].begin(),Sim[s].end(),
          F.begin());
    F.resize(Fend-F.begin());
    Faces[s].push_back(F);
  }
}
```

One application of storing the common faces is when one uses the CPA method to compute control Lyapunov functions as in [3]. The faces can also be used to identify the simplices of $\mathcal{T}$ that build the boundary $\partial\mathcal{D}_\mathcal{T}$ of $\mathcal{D}_\mathcal{T}$. A face is said to be maximal if it is an $(n-1)$-simplex and thus spanned by exactly n of its vertices. We define a simplex $\mathfrak{S}_\nu$ to be an *interior simplex* in $\mathcal{T}$ if all of its maximal faces are common with other simplices in $\mathcal{T}$. Otherwise, we define $\mathfrak{S}_\nu$ to be a *boundary simplex* in $\mathcal{T}$. Note that an n-simplex has $\binom{n+1}{n} = n+1$ number of maximal faces. We can thus identify a boundary simplex `Sim[s]` by simply counting the number of its maximal faces in `Faces[s]`. The boundary simplices of $\mathcal{T}$ are stored sorted in `vector<int> BSim`, which is build as follows in the constructor of `T_std_NK`:

```
for(int s=0;s<Sim.size();s++){
  int NrMax=0;
  for(auto pF=Faces[s].begin();pF!=Faces[s].end();pF++){
    if((*pF).size()==n){
      NrMax++;
    }
  }
  if(NrMax<n+1){
    BSim.push_back(s);
  }
}
sort(BSim.begin(),BSim.end());
```

The linear program from the CPA method always posses a feasible solution if the system $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ has an exponentially stable equilibrium at the origin and if the simplices used have a small enough diameter and are not too degenerated, cf. e.g. [15]. For discrete time systems $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ an analogous proposition holds true [13]. When generating such linear programming problems it is convenient to map the basic simplicial complex $\mathcal{T}$ to a simplicial complex $\mathcal{T}^{\mathbf{F}}$ with smaller simplices using a map $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$. A simplex $\mathfrak{S}_\nu := \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ in $\mathcal{T}$ is mapped to the simplex $\mathfrak{S}_\nu^{\mathbf{F}} = \text{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n))$ in $\mathcal{T}^{\mathbf{F}}$. This is implemented by the class `FT`, which is the subject of the next section.

3 The Simplicial Complex $\mathcal{T}^{\mathbf{F}}$

The simplicial complex $\mathcal{T} = \mathcal{T}_{N,K}^{\text{std}}$ is not adequate for the generation of linear programming problems for the computation of Lyapunov functions because its simplices are too large. Our solution to this issue is the simplicial complex $\mathcal{T}^{\mathbf{F}}$, which is implemented in `class FT`. An instance `SC` (Simplicial Complex) of `class FT` holds a pointer `T_std_NK *pBC` to an underlying basic simplicial complex $\mathcal{T}$ and a mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ that maps the vertices of $\mathcal{T}$ to the vertices of $\mathcal{T}^{\mathbf{F}}$.

The relationship between $\mathcal{T}$ and $\mathcal{T}^{\mathbf{F}}$ is that

$$\text{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n)) \in \mathcal{T}^{\mathbf{F}}, \quad \text{if and only if} \quad \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathcal{T}.$$

For $\mathfrak{S}_\nu = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ in $\mathcal{T}$ we denote the corresponding simplex in $\mathcal{T}^{\mathbf{F}}$ by $\mathfrak{S}_\nu^{\mathbf{F}} := \text{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n))$. Clearly the collection of the vertices of the simplices in $\mathcal{T}^{\mathbf{F}}$ are the set $\mathcal{V}_{\mathcal{T}^{\mathbf{F}}} := \mathbf{F}(\mathcal{V}_{\mathcal{T}})$. Note that the mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ must be chosen such that $\mathcal{T}^{\mathbf{F}}$ is an admissible triangulation and in general this is not true, even for a homeomorphism $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$, and in general neither $\mathbf{F}(\mathfrak{S}_\nu) \neq \mathfrak{S}_\nu^{\mathbf{F}}$ nor $\mathbf{F}^{-1}(\mathfrak{S}_\nu^{\mathbf{F}}) = \mathfrak{S}_\nu$.

Suitable candidates for the mapping $\mathbf{F}$ are discussed in [18]. A mapping that generalizes the mapping suggested there is given by the generic form

$$\mathbf{F}(\mathbf{x}) = \frac{\rho(\|\mathbf{x}\|_\infty)}{\|\mathbf{x}\|_*} A\mathbf{x}, \tag{6}$$

where $\|\cdot\|_*$ is any norm on $\mathbb{R}^n$, $\rho : \mathbb{R}_+ \to \mathbb{R}_+$ is a function of class $\mathcal{K}_\infty$, and $A \in \mathbb{R}^{n\times n}$ is a nonsingular matrix.

Note that for $\mathbf{F}$ as in (6) with $A = I$ we have $\|\mathbf{F}(\mathbf{x})\|_* = \rho(\|\mathbf{x}\|_\infty)$, i.e. $\mathbf{F}$ maps the hypercube $[-a, a]^n$ injectively onto the set $\{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_* \le \rho(a)\}$. By fixing the norm $\|\cdot\|_*$ as the energetic norm $\|\mathbf{x}\|_* = \|\mathbf{x}\|_Q := \sqrt{\mathbf{x}^T Q\mathbf{x}}$ for a symmetric, positive definite matrix $Q \in \mathbb{R}^{n\times n}$, we obtain by setting $A := Q^{-\frac{1}{2}}$, i.e. $A = O^T \operatorname{diag}(\mu_1^{-\frac{1}{2}}, \mu_2^{-\frac{1}{2}}, \dots, \mu_n^{-\frac{1}{2}})O$ where $Q = O^T \operatorname{diag}(\mu_1, \mu_2, \dots, \mu_n)O$ is the eigendecomposition of Q with an orthogonal $O \in \mathbb{R}^{n\times n}$, that $\|\mathbf{F}(\mathbf{x})\|_Q = \rho(\|\mathbf{x}\|_\infty)$. That is, $\mathbf{F}$ maps the hypercube $[-a, a]^n$ injectively onto the closed hyperellipsoid centered at the origin and of which the lengths of the principal axes are $\rho(a)/\mu_i^{\frac{1}{2}}$, $\mu_i > 0$ an eigenvalue of Q. See Fig. 2 for a picture of $\mathcal{T}^{\mathbf{F}}$ with $\mathbf{F}$ as in (6) with

$$Q = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} \quad \text{and} \quad A = Q^{-\frac{1}{2}} = \frac{1}{4}\begin{pmatrix} 1+\sqrt{2} & 1-\sqrt{2} \\ 1-\sqrt{2} & 1+\sqrt{2} \end{pmatrix}. \tag{7}$$

Note especially that the axes of the simplicial complex are rotated to better fit the level-sets of the energetic norm $\|\cdot\|_Q$. Such $\mathbf{F}$ have been used, for example, in [2–4,6]. As shown later in this section some algorithms can be made much

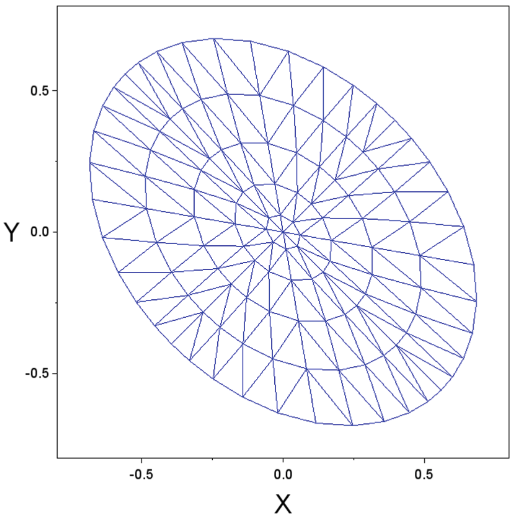

Fig. 2. The complex $\mathcal{T}^{\mathbf{F}}$ with $\mathbf{F}$ as in (6) with $\rho(x) = 0.1x^{\frac{3}{2}}$, Q and A as in (7) and $\|\cdot\|_* = \|\cdot\|_Q$.

more efficient if a formula for the inverse $\mathbf{F}^{-1}$ of $\mathbf{F}$ is available. If $\mathbf{F}$ is as in (6), then its inverse is easily verified to be given by

$$\mathbf{F}^{-1}(\mathbf{x}) = \frac{\rho^{-1}(\|A^{-1}\mathbf{x}\|_*)}{\|A^{-1}\mathbf{x}\|_\infty} A^{-1}\mathbf{x}. \tag{8}$$

Note that $\rho \in \mathcal{K}_\infty$ implies $\rho^{-1} \in \mathcal{K}_\infty$. In Sect. 3.2 we describe a fast algorithm that given an $\mathbf{x} \in \mathbb{R}^n$ and the inverse $\mathbf{F}^{-1}$ of $\mathbf{F}$ searches for a simplex $\mathfrak{S}_\nu^\mathbf{F} \in \mathcal{T}^\mathbf{F}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$.

3.1 Implementation of `class FT`

The data structure `class FT` implements the simplicial complex $\mathcal{T}^\mathbf{F}$. Its definition is:

```
 1 class FT {
 2 public:
 3   T_std_K *pBC;
 4   function<vec(vec)> pF, ipF;
 5   vec F(vec x);
 6   vec F(ivec x);
 7   vec iF(vec x);
 8   vector<vec> xVer;
 9   vector<mat> XmT;
10   vector<vec> SCC;
11   vector<double> SCR;
12   vector<vector<mat>> Fnor;
13   FT(T_std_K *_pBC, function<vec(vec)> _pF,
14     function<vec(vec)> _ipF=nullptr);
15   ~FT();
16   int InSimpNrSlow(vec x,vec &L);
17   int InSimpNrFast(vec x,vec &L,int guess);
18   int InSimpNrAppr(vec x);
19   bool InSimp(vec x,int s,vec &L);
20   int SVerNr(int s,int i);
21   vec SVer(int s,int i);
22   bool CFSS;  // default value "true"
23 };
```

We first discuss the constructor of `FT` declared in lines 13 and 14 in the code above. Its first argument is a pointer `T_std_NK *_pBC` to the underlying basic simplicial complex, which is assigned to the member variable `T_std_NK *pBC`. The second argument of the constructor is the mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ that is used to map the vertices of the basic simplicial complex. It is assigned to the member variable `function<vec(vec)> pF` and can be called with the member functions `vec FT::F(vec x)` and `vec FT::F(ivec x)`. Their implementation is trivial:

```
vec F(vec x){
  return pF(x);
}

vec F(ivec x){
  return pF(conv_to<vec>::from(x));
}
```

The third argument of the constructor is the inverse $\mathbf{F}^{-1}$ of $\mathbf{F}$. If it is available it can be used to decrease the computational complexity of several algorithms considerably. It is stored in the member variable `function<vec(vec)> ipF`. If the inverse is not available we initialize `ipF=nullptr`. The implementation of the member function `vec iF(vec x)` is analogous to the implementation of `vec F(vec x)`, just with `pF` replaced by `ipF`. To avoid that one delivers to `function<vec(vec)> ipF` a function that is not the inverse of $\mathbf{F}$ we verify $\mathbf{F}^{-1}(\mathbf{F}(\mathbf{x})) = \mathbf{x}$ for a random sample in the domain $\mathcal{D}_{\mathcal{T}}$ and we verify $\mathbf{F}^{-1}(\mathbf{0}) = \mathbf{0}$. If these tests are not passed we set `ipF=nullptr` and warn the user. This is implemented as follows in the constructor of `FT`:

```
if(ipF!=nullptr){
  arma_rng::set_seed_random();
  int NrRandVec=1000;
  double tol=1e-10;
  if(norm(iF(F(zeros<vec>(n))),"inf")>tol){
    cerr<<"iF(F(0)) != 0"<<endl;
    ipF=nullptr;
  }
  vec m=F(pBC->Nm);
  vec M=F(pBC->Np);
  for(int i=0;i<NrRandVec;i++){
    vec r=randu<vec>(n);
    vec x=r%(M-m)+m;
    if(norm(iF(F(x))-x,2)>tol*norm(x,2)){
      cerr<<"iF(F(x)) != x for x="<<x.t();
      ipF=nullptr;
      break;
    }
  }
}
```

Note that for `vec x` and `vec y` in Armadillo `x%y` denotes element-by-element multiplication, similar to `x.*y` in Matlab, Scilab, and Octave.

We store the vertices `vector<vec> xVer` of $\mathcal{T}^{\mathbf{F}}$ in the same order as the corresponding integer coordinate vertices `vector<ivec> Ver` of $\mathcal{T}$ in the basic simplicial complex pointed to by `T_std_NK *pBC`. This is implemented in the constructor of `FT` in the obvious way:

```
for(auto p=pBC->Ver.begin();p!=pBC->Ver.end();p++){
  xVer.push_back(F(*p));
}
```

We can thus refer to the simplex `pBC->Sim[s]` or just the simplex `Sim[s]` in $\mathcal{T}^{\mathbf{F}}$. It is the simplex with vertices `xVer[pBC->Sim[s][i]]` for `i`$=0:n$. The implementation of the member function `int FT::SVerNr(int s,int i)` simply calls the homonymous member function in the underlying basic simplicial complex and the implementation of `vec FT::SVer(int s,int i)` is trivial.

```
int FT::SVerNr(int s,int i){
  return pBC->SVerNr(s,i);
}

vec FT::SVer(int s,int i){
  return xVer[SVerNr(s,i)];
}
```

The class `FT` stores for each simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ the transpose of the inverse X_ν^{-T} of the so-called shape-matrix X_ν of $\mathfrak{S}_\nu^{\mathbf{F}}$, cf. [21]. The shape-matrix X_ν of $\mathfrak{S}_\nu^{\mathbf{F}} = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ is defined by writing the vectors $\mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{v}_2 - \mathbf{v}_0$, …, $\mathbf{v}_n - \mathbf{v}_0$ consecutively in its rows. Note that because the vectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$ are affinely independent the matrix X_ν is invertible. The matrices X_ν^{-T} are stored as `vector<mat> XmT` in the same order as the simplices `vector<int> pBC->Sim` in the basic simplicial complex. Thus `XmT[s]` is the transpose the inverse of the shape-matrix of the simplex `pBC->Sim[s]` in $\mathcal{T}^{\mathbf{F}}$. The reason why we store X_ν^{-T} rather than X_ν^{-1} or simply X_ν is that X_ν^{-T} is the most useful form for `bool FT::InSimp(vec x,int s,vec &L)`.

Further information we want to have available for the simplices $\mathfrak{S}_\nu^{\mathbf{F}}$ in $\mathcal{T}^{\mathbf{F}}$ are the centers of their circumscribing hyperspheres and their radii, stored in `vector<vec> SCC` and `vector<double> SCR` respectively. Again, they are stored in the same order as the simplices in the basic simplicial complex, thus `SCC[s]` is the center and `SCR[s]` is the radius of the circumscribing hypersphere of the simplex `pBC->Sim[s]` in $\mathcal{T}^{\mathbf{F}}$. The formulas

$$\mathbf{c} = \mathbf{v}_0 + \frac{1}{2} X_\nu^{-1}\mathbf{b}, \text{ with } b_i = \|\mathbf{v}_i - \mathbf{v}_0\|_2^2 \text{ for } i = 1:n \text{ and } r = \|\mathbf{c} - \mathbf{v}_0\|_2$$

for the center $\mathbf{c} \in \mathbb{R}^n$ of the circumscribing hypersphere of $\mathfrak{S}_\nu^{\mathbf{F}} = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ and its radius r were derived in [18]. The construction of `vector<mat> XmT`, `vector<vec> SCC`, and `vector<double> SCR` is implemented in the constructor of `FT` as follows.

```
for(int s=0;s<pBC->Sim.size();s++){
  mat XT(n,n);
  vec v0=SVer(s,0);
  vec b(n);
  for(int i=1;i<=n;i++){
    XT.col(i-1)=SVer(s,i)-v0;
    b(i-1)=pow(norm(XT.col(i-1),2),2);
  }
  XmT.push_back(XT.i());
  vec c=x0+0.5*XmT[s].t()*b;
  SCC.push_back(c);
  SCR.push_back(norm(c-v0,2));
}
```

Using the matrices `vector<mat> XmT` one can easily check whether a vector $\mathbf{x} \in \mathbb{R}^n$ is in a particular simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ =`Sim[s]` or not. In [18] it was shown that $\mathbf{x} \in \mathbb{R}^n$ is in $\mathfrak{S}_\nu^{\mathbf{F}} = \mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n)$, if and only if

$$(\lambda_1, \lambda_2, \dots, \lambda_n)^T = X_\nu^{-T}(\mathbf{x} - \mathbf{v}_0) \in \mathbb{R}_+^n \quad \text{and} \quad \lambda_0 := 1 - \sum_{i=1}^n \lambda_i \geq 0.$$

Then $\lambda_0, \lambda_1, \dots, \lambda_n$ are the barycentric coordinates of $\mathbf{x}$ in $\mathfrak{S}_\nu^{\mathbf{F}}$. The implementation is:

```
bool FT::InSimp(vec x,int s,vec &L){
  vec mu=XmT[s]*(x-SVer(s,0));
  if(min(mu)>= 0 && sum(mu)<=1){
    L(0)=1.0-sum(mu);
    L(span(1,n))=mu;
    return true;
  }
  else{
    return false;
  }
}
```

If $\mathbf{x}$ is in the simplex `Sim[s]` in $\mathcal{T}^{\mathbf{F}}$ the function returns `true` and assigns the barycentric coordinates of $\mathbf{x}$ to `vec &L`, i.e. `L` $= (\lambda_0, \lambda_1, \dots, \lambda_n)^T$.

Finally, normals to the hyperplanes defining and separating neighbouring simplices are stored. Although we have changed some data structures used from `list` to `vector` the implementation given in [18] works and has not been changed. Since the discussion is quite involved and the normals are not needed in the rest of this paper, we refer the interested reader to [18].

3.2 Fast Search for $\mathfrak{S}_\nu^{\mathbf{F}}$ Such that $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$

The following problem is often of interest: Given an $\mathbf{x} \in \mathbb{R}^n$ find an $\mathfrak{S}_\nu^{\mathbf{F}} \in \mathcal{T}^{\mathbf{F}}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$. Additionally, one often then needs the barycentric coordinates of $\mathbf{x}$ in $\mathfrak{S}_\nu$, i.e. the λ_i such that $\mathbf{x} = \sum_{i=0}^n \lambda_i \mathbf{v}_i$ is the convex combination of the vertices $\mathbf{v}_i$ of $\mathfrak{S}_\nu^{\mathbf{F}}$. Without any additional information one must rely on linear search, implemented as:

```
int FT::InSimpNrSlow(vec x,vec &L){
  for(int s=0;s<pBC->Sim.size();s++){
    if(InSimp(x,s,L)==true){
      return s;
    }
  }
  return -1;
}
```

Note that the `vec &L` corresponds to the vector $(\lambda_0, \lambda_1, \ldots, \lambda_n)^T$ and if a simplex containing $\mathbf{x}$ is not found the impossible value -1 is returned from the function.

If a formula for the inverse mapping $\mathbf{F}^{-1}$ of $\mathbf{F}$ is available a much faster search is possible. For some applications, e.g. when plotting a computed Lyapunov function, it might be sufficient to know a simplex $\mathfrak{S}_\nu^\mathbf{F}$ such that $\mathbf{x}$ is close to $\mathfrak{S}_\nu^\mathbf{F}$. If the mappings $\mathbf{F}$ and $\mathbf{F}^{-1}$ are not too exotic a simplex $\mathfrak{S}_\nu^\mathbf{F} \in \mathcal{T}^\mathbf{F}$, such that $\mathbf{y} := \mathbf{F}^{-1}(\mathbf{x}) \in \mathfrak{S}_\nu$, is often a good candidate. Such an $\mathfrak{S}_\nu$ can be computed very efficiently in the member function `T_std_NK::InSimpNr(vec x)` as described in [17]. This is implemented as

```
int FT::InSimpNrAppr(vec x){
  assert(ipF!=nullptr);
  return pBC->InSimpNr(iF(x));
}
```

For many applications, however, one needs a simplex $\mathfrak{S}_\nu^\mathbf{F}$ such that truly $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$ and one needs the barycentric coordinates of $\mathbf{x}$ in $\mathfrak{S}_\nu^\mathbf{F}$. An important example is when a linear programming problem for discrete-time dynamical systems is constructed, cf. [13,19,32].

The idea behind the fast search for a simplex $\mathfrak{S}_\nu^\mathbf{F} \in \mathcal{T}^\mathbf{F}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$ is as follows: Given an $\mathbf{x} \in \mathbb{R}^n$, for which we need a simplex $\mathfrak{S}_\nu^\mathbf{F} \in \mathcal{T}^\mathbf{F}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$, start with a simplex $\mathfrak{S}_\xi^\mathbf{F} \in \mathcal{T}^\mathbf{F}$ that is a good guess. This guess can either be delivered by the caller of the search function or one can compute an $\mathfrak{S}_\xi \in \mathcal{T}$ such that $\mathbf{F}^{-1}(\mathbf{x}) \in \mathfrak{S}_\xi$. If $\mathbf{x} \in \mathfrak{S}_\xi^\mathbf{F}$ we are finished. If not check if $\mathbf{x} \in \mathfrak{S}_\mu^\mathbf{F}$ for neighbouring simplices $\mathfrak{S}_\mu^\mathbf{F}$ of $\mathfrak{S}_\nu^\mathbf{F}$. If an $\mathfrak{S}_\mu^\mathbf{F}$ is found such that $\mathbf{x} \in \mathfrak{S}_\mu^\mathbf{F}$ we are finished. If not check if $\mathbf{x} \in \mathfrak{S}_\eta^\mathbf{F}$ for neighbouring simplices $\mathfrak{S}_\eta^\mathbf{F}$ of the simplices $\mathfrak{S}_\mu^\mathbf{F}$ that have not already been checked. Repeat this procedure until a simplex is found.

One comment about another speedup of the search before we give the implementation. If $\mathbf{F}^{-1}(\mathbf{x}) \notin \mathcal{D}_\mathcal{T} := \bigcup_{\mathfrak{S}_\nu \in \mathcal{T}} \mathfrak{S}_\nu$ implies $\mathbf{x} \notin \mathcal{D}_\mathcal{T}^\mathbf{F} := \bigcup_{\mathfrak{S}_\nu \in \mathcal{T}} \mathfrak{S}_\nu^\mathbf{F}$ the member variable `bool CFSS` (careful simplex search), whose default value is `true`, should be assigned the value `false`. It was shown in [18] that this implication holds true for many important triangulations and suitable $\mathbf{F}$ and $\mathbf{F}^{-1}$ as in (6) and (8). This simple trick, and variants of it, can save a lot of computations because it is very laborious to search exhaustively for a simplex that is not in the complex $\mathcal{T}^\mathbf{F}$.

```
int FT::InSimpNrFast(vec x,vec &L,int guess=-1){
  int s;
  if(guess!=-1){
    s=guess;
  }
  else{
    s=InSimpNrAppr(x);
  }
  if(s==-1){
    if(CFSS==true){ // might be in complex
      s=InSimpNrSlow(x,L);
    }
    return s;
  }
  if(InSimp(x,s,L)==true){
    return s;
  }
  vector<int> TC, CH, CN, CNtemp;
  // TC = To Check
  // CH = already CHecked
  // CN = Check Next
  // CNtemp = temporary values for CN
  for(int is=0;is<pBC->SimN[s].size();is++){
    if(pBC->Faces[s][is].size()==n){
      TC.push_back(pBC->SimN[s][is]);
    }
  }
  for(auto p=TC.begin();p!=TC.end();p++){
    if(InSimp(x,*p,L)==true){
      return s=*p;
    }
  }
  // initialize the main loop
  CH.push_back(s);
  TC.push_back(s);
  int MaxSweeps=3;
  while(MaxSweeps--){
    CN.clear();
    for(p=TC.begin();p!=TC.end();p++){
      vector<int> A2CN;   // add to CN
      for(int is=0;is<pBC->SimN[*p].size();is++){
        if(pBC->Faces[*p][is].size() == n){
           A2CN.push_back(pBC->SimN[*p][is]);
        }
      CN.insert(CN.end(),A2CN.begin(),A2CN.end());
    }
    sort(CN.begin(),CN.end());
    CNtemp.reserve(CN.size());
    auto it=unique_copy(CN.begin(),CN.end(),CNtemp.begin());
    CN.assign(CNtemp.begin(),it);
    TC.clear();
    set_difference(CN.begin(),CN.end(),CH.begin(),CH.end(),
                                      back_inserter(TC));
    if(TC.empty()==true){
      return -1;
    }
    for(p=TC.begin();p!=TC.end();p++){
      if(InSimp(x,*p,L)==true){
        return *p;
      }
    }
    CH.insert(CH.end(),TC.begin(),TC.end());
    sort(CH.begin(),CH.end());
  }
  // give up on being clever
  return InSimpNrSlow(x,L);
}
```

We have four remarks on this algorithm, which is similar to the one given in [18], but has been tweaked in several ways. First, using `std::vector` instead of `std::list` for `TC,CH,CN,CNtemp` as in [18] turned out to be about 5% faster in most cases and we therefore adapted the implementation accordingly. Second, if one wants α_1 in Eq. (9) in Sect. 4 to be a so-called CPA function defined on the simplicial complex $\mathcal{T}^{\mathbf{F}}$, cf. e.g. [15], then one sequentially searches for simplices for points $\mathbf{x}_i$, $i = 0 : N$, where $\mathbf{x}_i$ is close to $\mathbf{x}_{i+1}$. Thus it might significantly speed up computations storing the last simplex found by this algorithm in a static variable and check directly if the new point delivered to the algorithm is in the same simplex as the point before. This however has the major drawback that it interferes with searching for simplices in parallel using multithreading. We achieve the same speedup of the search by adding the variable `int guess`, which has the (impossible) default value -1. The caller of the function can deliver the number of the simplex in `guess` to the algorithm where the search should start. If nothing is delivered the search starts in the simplex whose number is delivered by `FT::InSimpNrAppr(x)` as in [18]. Third, if the main loop is used to search through the whole simplicial complex, then it is considerably slower than a linear search through the whole simplicial complex by `FT::InSimpNrSlow(x,L)`. Thus, the implementation uses the parameter `int MaxSweeps` to decide when to give up on searching for neighbours and just do a linear search. Its appropriate value will depend on the kind of the problem at hand to be solved. We used `MaxSweeps=3` which worked quite well for our examples. Fourth, instead of using all neighbours as in [18] it turned out to be about 10% faster in our examples to use only neighbours with common faces that are maximal, i.e. $(n-1)$-simplices (n common vertices). Therefore the implementation was changed accordingly.

4 Computing Values of $V(\boldsymbol{\xi})$ for $\boldsymbol{\xi} \in \mathcal{V}_{\mathcal{T}^{\mathrm{F}}}$ Directly

Instead of solving the linear optimization problem generated to obtain a Lyapunov function, one can use a different method to make *educated guesses* of their values and then verify if the linear constraints are fulfilled for these values. This is the approach followed in, e.g. [4,5,7,10–12,19,20,32], and it generates the values much faster than solving the linear programming problem. An additional advantage is that one can localize the area where the constraints are not fulfilled, whereas a solver will simply state that the linear programming problem does not possess a feasible solution when that is the case.

For a continuous-time system $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ with an exponentially stable equilibrium at the origin a Lyapunov function is given by

$$V(\boldsymbol{\xi}) = \int_0^T \alpha_1(\boldsymbol{\phi}(\tau, \boldsymbol{\xi}))d\tau, \tag{9}$$

where $\tau \mapsto \boldsymbol{\phi}(\tau, \boldsymbol{\xi})$ is the solution to $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ with initial-value $\boldsymbol{\xi}$ at time $\tau = 0$, $T > 0$ is a large enough constant, and $\alpha_1 : \mathbb{R}^n \to \mathbb{R}_+$ is continuous and positive definite function, i.e. $\alpha_1(\mathbf{0}) = \mathbf{0}$ and $\alpha_1(\mathbf{x}) > 0$ for $\mathbf{x} \neq \mathbf{0}$. The idea for constructing a Lyapunov function like this goes back to Massera [35], see also [26], and is discussed in many textbooks on nonlinear systems, e.g. [27,43]. Typically for our applications $\alpha_1(\mathbf{x}) = \alpha(\|\mathbf{x}\|_2)$, where α is of class $\mathcal{K}_\infty$, but there are other suitable choices, for example $\alpha_1(\mathbf{x}) = \|\mathbf{f}(\mathbf{x})\|_2$ is used in [7].

Note that although (9) gives an explicit formula for a Lyapunov function, this formula includes the solution $\boldsymbol{\phi}(\tau, \boldsymbol{\xi})$ to the differential equation and the solution is usually not known. It can, however, be approximated at the vertices of the simplicial complex with a subsequent verification of the constraints of the linear programming problem.

To approximate (9) numerically we used the Adam-Bashforth four-step method for obtaining numerically a solution $t \mapsto \boldsymbol{\phi}(t, \boldsymbol{\xi})$ to the initial-value problem

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}), \quad \mathbf{x}(0) = \boldsymbol{\xi},$$

on $[0, T]$ and the composite Simpson's Rule to integrate $\alpha_1(\boldsymbol{\phi}(t, \boldsymbol{\xi}))$ over the same interval. Both are standard methods that are described in most textbooks on numerical analysis, cf. e.g. [41].

For completeness we give a short-description: First choose the number of steps N to be used, an even number ≥ 4, and define $h := T/N$, $t_i := ih$, and $\phi_i := \boldsymbol{\phi}(t_i, \boldsymbol{\xi})$ for $i = 0 : N$. The composite Simpson's Rule approximates the integral in (9) with the sum

$$\frac{h}{3}\left[\alpha_1(\phi_0) + \alpha_1(\phi_N) + 4\sum_{i=1}^{N/2} \alpha_1(\phi_{2i-1}) + 2\sum_{i=1}^{N/2-1} \alpha_1(\phi_{2i})\right] \tag{10}$$

and the error is bounded by C_1h^4 for some constant $C_1 > 0$. The Adam-Bashforth four-step method approximates the ϕ_i with numbers w_i using the formula

$$w_{i+1} = w_i + \frac{h}{24}\left[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}\right], \quad \text{where } f_i := \mathbf{f}(w_i), \tag{11}$$

and an error bound is given by $|w_i - \phi_i| \leq C_2(e^{Lhi} - 1)h^4$ for some constant $C_2 > 0$ and where $L > 0$ is a Lipschitz constant for $\mathbf{f}$ on the relevant subset of $\mathbb{R}^n$. We compute w_1, w_2, w_3 with the Runge-Kutta Method of fourth order (RK4), which has compatible error bounds. Thus, we compute

$$\frac{h}{3}\left[\alpha_1(w_0) + \alpha_1(w_N) + 4\sum_{i=1}^{N/2}\alpha_1(w_{2i-1}) + 2\sum_{i=1}^{N/2-1}\alpha_1(w_{2i})\right] \tag{12}$$

as an approximation to the integral in (9). With $K > 0$ as a Lipschitz constant for α_1 on the relevant subset of $\mathbb{R}^n$ the difference between (10) and (12) is bounded by

$$\begin{aligned}\frac{h}{3}\cdot 2K &\left[\sum_{i=1}^{N}|w_i - \phi_i| + \sum_{i=1}^{N/2-1}|w_{2i-1} - \phi_{2i-1}|\right] \\ &\leq \frac{2Kh}{3}C_2h^4\left[\sum_{i=1}^{N}(e^{Lhi} - 1) + \sum_{i=1}^{N/2-1}(e^{Lh(2i-1)} - 1)\right].\end{aligned} \tag{13}$$

Now

$$\sum_{i=1}^{N} e^{Lhi} = e^{Lh}\frac{e^{LNh} - 1}{e^{Lh} - 1} \leq e^{Lh}\frac{e^{LT} - 1}{Lh}$$

and similarly

$$\sum_{i=1}^{N/2-1} e^{Lh(2i-1)} \leq e^{Lh}\frac{e^{LT} - 1}{2Lh}$$

and since $Nh = T$ we can bound (13) by

$$\frac{2K}{3}C_2h^4\left[\frac{3e^{Lh}}{2L}(e^{LT} - 1) - \frac{3}{2}T + h\right]. \tag{14}$$

Thus we have shown that the error we make when we compute (12) as an approximation to the integral in (9) the error is bound by $Ce^{LT}h^4$ for some constant $C > 0$, or in big-O notation, the error is $O(e^{LT}h^4)$.

The computation of $V(\boldsymbol{\xi})$ was implemented through the class `ODEsolInt` with `ODEsolInt::operator()(vec x)` doing the actual evaluation. Note that one can evaluate $V(\boldsymbol{\xi})$ for many different $\boldsymbol{\xi}$ simultaneously using multithreading. The class `ODEsolInt` has the members `function<vec(vec)> f`, which is the right-hand-side of the differential equation $\mathbf{x}' = \mathbf{f}(\mathbf{x})$, `double T,h` and `int N`, where

`T,h` and `N` correspond to T, h, and N above. A further member is the function `function<double(vec)> alpha1`, which is the function α_1 in the integral (9). By default it is fixed to $\alpha_1(\mathbf{x}) = \|\mathbf{x}\|_2$. The definition of the class `ODEsolInt` and the implementation of its constructor are given below. Note that we need N to be an even integer ≥ 4 for our implementation for the numerical approximation of the integral to be correct, so an inadequate value for `N` is modified accordingly. Recall that for `int i,j` the command `i%j` gives the reminder when `i` is divided by `j`. Thus `i%2` equals zero if `i` is an even number and `i%2` equals one if `i` is odd.

```
double def_alpha1(vec x){ return norm(x, 2); }

class ODEsolInt{
public:
  function<vec(vec)> f;
  function<double(vec)> alpha1;
  double T,h;
  int N;
  ODEsolInt(function<vec(vec)> _f,double _T,int _N,
                  function<double(vec)> _alpha1=def_alpha1);
  ODEsolInt(){};
  double operator()(vec x);
};

ODEsolInt::ODEsolInt(function<vec(vec)> _f,double _T,
      int _N,function<double(vec)> _alpha1) :
         f(_f),T(_T),N(_N),alpha1(_alpha1) {
  // N should be an even number >= 4
  if(N<4){
    N=4;
  }
  if(N%2==1){
    N++;
  }
  h=T/N;
}
```

For an efficient implementation of formulas (11) and (12) we store as little information as possible and plug the values of w_i from (11) immediately into (12). We use the Runga-Kutta Method of fourth-order to compute the first three initial w_1, w_2, w_3 and then we use four-step Adam-Bashforth, which is multistep method that only needs the evaluation of $\mathbf{f}(\mathbf{x})$ at one new point $\mathbf{x}$ for each new value w_i, instead of at four points as in the Runge-Kutta Method. The four-step Adam-Bashforth Method achieves this by using already computed values as can be seen in (11). We thus store $f_i, f_{i-1}, f_{i-2}, f_{i-3}$ in the columns of the matrix `mat AB(n,4)`, and after having used these values to compute w_{i+1} we compute f_{i+1} using w_{i+1}. Then we overwrite f_{i-3}, which is not needed anymore, with

f_{i+1}. This can be implemented by accessing the columns of `mat AB(n,4)` with i modulo 4, that is, f_i =`mat AB.col(i%4)`. Simultaneously to computing the w_i we sum up

$$\alpha_1(w_0)+4\alpha_1(w_1)+2\alpha_1(w_2)+4\alpha_1(w_3)+2\alpha_1(w_4)+\ldots+4\alpha_1(w_{N-1})+2\alpha_1(w_N)$$

and then subtract $\alpha_1(w_N)$ in the end and multiply the sum by $h/3$ to obtain the value from formula (12).

```
vec RK4(function<vec(vec)> f,vec w_i,double h){
  vec s1(n),s2(n),s3(n),s4(n);
  s1=f(w_i);
  s2=f(w_i+h/2*s1);
  s3=f(w_i+h/2*s2);
  s4=f(w_i+h*s3);
  return w_i+h/6*(s1+2*s2+2*s3+s4);
}

double ODEsolInt::operator()(vec x){
  double ret=0.0;
  mat AB(n,4);
  vec x1,x0;
  x0=x;
  AB.col(0)=f(x0);
  ret+=alpha1(x0);
  for(int i=1;i<=3;i++){
    x1=RK4(f,x0,h);
    ret+=2.0*(1+i%2)*alpha1(x1);
    AB.col(i)=f(x1);
    x0=x1;
  }
  for(int i=3;i<N;i++) {
    x1=x0+h/24.0*(55.0*AB.col(i%4)-59.0*AB.col((i-1)%4)
        +37.0*AB.col((i-2)%4)-9.0*AB.col((i-3)%4));
    AB.col((i+1)%4)=f(x1);
    ret+=2.0*(2-i%2)*alpha1(x1);
    x0=x1;
  }
  ret-=alpha1(x1);
  ret*=h/3.0;
  return ret;
}
```

5 Summary

We discussed the implementation in C++ of simplicial complexes using efficient algorithms and data structures for the computation of Lyapunov functions for nonlinear systems. This paper builds upon [6,14,17] and advances and improves the methods presented in [18] in various ways. The algorithms are designed for both continuous-time and discrete-time systems and the implementation for control Lyapunov functions using the Dini subdifferential is accounted for. Additionally, we gave a detailed description and error-analysis for the fast computation of an approximation to a Lyapunov function at the vertices of a simplicial complex using a Lyapunov function candidate from a converse theorem by Massera. This approach has been used in several publications, cf. e.g. [4,5,7,10–12,19,20,32], but its efficient implementation has not been discussed before.

Acknowledgement. The author's research is supported by the Icelandic Research Fund (Rannís) 'Complete Lyapunov functions: Efficient numerical computation' (163074-052) and 'Lyapunov Methods and Stochastic Stability' (152429-051), which is gratefully acknowledged.

References

1. Ambrosino, R., Garone, E.: Robust stability of linear uncertain systems through piecewise quadratic Lyapunov functions defined over conical partitions. In: Proceedings of the 51st IEEE Conference on Decision and Control, Maui (HI), USA, pp. 2872–2877, December 2012
2. Baier, R., Grüne, L., Hafstein, S.: Linear programming based Lyapunov function computation for differential inclusions. Discrete Contin. Dyn. Syst. Ser. B **17**(1), 33–56 (2012)
3. Baier, R., Hafstein, S.: Numerical computation of Control Lyapunov Functions in the sense of generalized gradients. In: Proceedings of the 21st International Symposium on Mathematical Theory of Networks and Systems (MTNS), Groningen, The Netherlands, pp. 1173–1180, no. 0232 (2014)
4. Björnsson, J., Giesl, P., Hafstein, S., Kellett, C., Li, H.: Computation of continuous and piecewise affine Lyapunov functions by numerical approximations of the Massera construction. In: Proceedings of the 53rd IEEE Conference on Decision and Control, Los Angeles (CA), USA, pp. 55056–5511 (2014)
5. Björnsson, J., Giesl, P., Hafstein, S., Kellett, C., Li, H.: Computation of Lyapunov functions for systems with multiple attractors. Discrete Contin. Dyn. Syst. Ser. A **35**(9), 4019–4039 (2015)
6. Björnsson, J., Gudmundsson, S., Hafstein, S.: Class library in C++ to compute Lyapunov functions for nonlinear systems. In: Proceedings of MICNON, 1st Conference on Modelling, Identification and Control of Nonlinear Systems, no. 0155, pp. 788–793 (2015)
7. Björnsson, J., Hafstein, S.: Efficient Lyapunov function computation for systems with multiple exponentially stable equilibria. Procedia Comput. Sci. **108**, 655–664 (2017)
8. Branicky, M.: Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. IEEE Trans. **43**(4), 475–482 (1998)
9. Clarke, F.: Optimization and Nonsmooth Analysis. Classics in Applied Mathematics. SIAM, Philadelphia (1990)
10. Doban, A.: Stability domains computation and stabilization of nonlinear systems: implications for biological systems. Ph.D. thesis, Eindhoven University of Technology (2016)

11. Doban, A., Lazar, M.: Computation of Lyapunov functions for nonlinear differential equations via a Yoshizawa-type construction. IFAC-PapersOnLine **49**(18), 29–34 (2016). 10th IFAC Symposium on Nonlinear Control Systems NOLCOS 2016
12. Giesl, P., Hafstein, S.: Computation and verification of Lyapunov functions. SIAM J. Appl. Dyn. Syst. **14**(4), 1663–1698 (2015)
13. Giesl, P., Hafstein, S.: Computation of Lyapunov functions for nonlinear discrete systems by linear programming. J. Differ. Equ. Appl. **20**, 610–640 (2014)
14. Giesl, P., Hafstein, S.: Implementation of a fan-like triangulation for the CPA method to compute Lyapunov functions. In: Proceedings of the 2014 American Control Conference, Portland (OR), USA, pp. 2989–2994, no. 0202 (2014)
15. Giesl, P., Hafstein, S.: Revised CPA method to compute Lyapunov functions for nonlinear systems. J. Math. Anal. Appl. **410**, 292–306 (2014)
16. Hafstein, S.: An algorithm for constructing Lyapunov functions, volume 8 of Monograph. Electron. J. Diff. Eqns. (2007)
17. Hafstein, S.: Implementation of simplicial complexes for CPA functions in C++11 using the armadillo linear algebra library. In: Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), Reykjavik, Iceland, pp. 49–57 (2013)
18. Hafstein, S.: Efficient algorithms for simplicial complexes used in the computation of Lyapunov functions for nonlinear systems. In: Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), Madrid, Spain, pp. 398–409 (2017)
19. Hafstein, S., Kellett, C., Li, H.: Computation of Lyapunov functions for discrete-time systems using the Yoshizawa construction. In: Proceedings of the 53rd IEEE Conference on Decision and Control (2014)
20. Hafstein, S., Kellett, C., Li, H.: Continuous and piecewise affine Lyapunov functions using the Yoshizawa construction. In: Proceedings of the 2014 American Control Conference, Portland (OR), USA, pp. 548–553, no. 0170 (2014)
21. Hafstein, S., Kellett, C., Li, H.: Computing continuous and piecewise affine Lyapunov functions for nonlinear systems. J. Comput. Dyn. **2**(2), 227–246 (2015)
22. Johansson, M.: Piecewise linear control systems. Ph.D. thesis, Lund University, Sweden (1999)
23. Johansson, M., Rantzer, A.: Computation of piecewise quadratic Lyapunov functions for hybrid systems. IEEE Trans. Automat. Control **43**(4), 555–559 (1998)
24. Julian, P.: A high level canonical piecewise linear representation: theory and applications. Ph.D. thesis, Universidad Nacional del Sur, Bahia Blanca, Argentina (1999)
25. Julian, P., Guivant, J., Desages, A.: A parametrization of piecewise linear Lyapunov functions via linear programming. Int. J. Control **72**(7–8), 702–715 (1999)
26. Kellett, C.: Converse theorems in Lyapunov's second method. Discrete Contin. Dyn. Syst. Ser. B **20**(8), 2333–2360 (2015)
27. Khalil, H.: Nonlinear Systems, 3rd edn. Pearson, London (2002)
28. Lazar, M.: On infinity norms as Lyapunov functions: alternative necessary and sufficient conditions. In: Proceedings of the 49th IEEE Conference on Decision and Control, Atlanta, USA, pp. 5936–5942, December 2010
29. Lazar, M., Doban, A.: On infinity norms as Lyapunov functions for continuous-time dynamical systems. In: Proceedings of the 50th IEEE Conference on Decision and Control, Orlando (Florida), USA, pp. 7567–7572 (2011)

30. Lazar, M., Doban, A., Athanasopoulos, N.: On stability analysis of discrete-time homogeneous dynamics. In: Proceedings of the 17th International Conference on Systems Theory, Control and Computing, Sinaia, Romania, pp. 297–305, October 2013
31. Lazar, M., Jokić, A.: On infinity norms as Lyapunov functions for piecewise affine systems. In: Proceedings of the Hybrid Systems: Computation and Control Conference, Stockholm, Sweden, pp. 131–141, April 2010
32. Li, H., Hafstein, S., Kellett, C.: Computation of continuous and piecewise affine Lyapunov functions for discrete-time systems. J. Differ. Equ. Appl. **21**(6), 486–511 (2015)
33. Marinósson, S.: Lyapunov function construction for ordinary differential equations with linear programming. Dyn. Syst. Int. J. **17**, 137–150 (2002)
34. Marinósson, S.: Stability analysis of nonlinear systems with linear programming: a Lyapunov Functions based approach. Ph.D. thesis, Gerhard-Mercator-University Duisburg, Duisburg, Germany (2002)
35. Massera, J.: Contributions to stability theory. Ann. Math. **64**, 182–206 (1956). Erratum. Ann. Math. **68**, 202 (1958)
36. Ohta, Y.: On the construction of piecewise linear Lyapunov functions. In: Proceedings of the 40th IEEE Conference on Decision and Control, vol. 3, pp. 2173–2178, December 2001
37. Ohta, Y., Tsuji, M.: A generalization of piecewise linear Lyapunov functions. In: Proceedings of the 42nd IEEE Conference on Decision and Control, vol. 5, pp. 5091–5096, December 2003
38. Polanski, A.: Lyapunov functions construction by linear programming. IEEE Trans. Automat. Control **42**, 1113–1116 (1997)
39. Polanski, A.: On absolute stability analysis by polyhedral Lyapunov functions. Automatica **36**, 573–578 (2000)
40. Sanderson, C.: Armadillo: an open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA (2010)
41. Sauer, T.: Numerical Analysis, 2nd edn. Pearson, London (2012)
42. Stroustrup, B.: Software development for infrastructure. Computer **45**(1), 47–58 (2012)
43. Vidyasagar, M.: Nonlinear System Analysis. Classics in Applied Mathematics, 2nd edn. SIAM, Philadelphia (2002)
44. Yfoulis, C., Shorten, R.: A numerical technique for the stability analysis of linear switched systems. Int. J. Control **77**(11), 1019–1039 (2004)