Efficient Algorithms for Simplicial Complexes Used in the Computation of Lyapunov Functions for Nonlinear Systems

Sigurdur Freyr Hafstein
Faculty of Physical Sciences, University of Iceland, Dunhagi 5, 107 Reykjavik, Iceland

Keywords: Simplicial Complex, Triangulation, Algorithm, Lyapunov Function, Nonlinear System, Discrete-time System.

Abstract: Several algorithms have been suggested to parameterize continuous and piecewise affine Lyapunov functions for nonlinear systems in various settings. In these algorithms linear constraints are formulated for the values of a Lyapunov function at all vertices of a simplicial complex. They are then either solved using convex optimization or computed by other means and then verified. Originally these algorithms were 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. In this paper we discuss several of these and give efficient implementations in C++.

1 INTRODUCTION

A Lyapunov function V for a dynamical system is a continuous function from the state-space to the real numbers that is decreasing along the system's trajectories. For a continuous-time system given by a differential equation $\mathbf{x}' = \mathbf{f}(\mathbf{x})$ this can be ensured by the condition $\nabla V(\mathbf{x})\bullet\mathbf{f}(\mathbf{x}) < 0$. For a discrete-time system $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ a corresponding condition is

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

In (Giesl and Hafstein, 2014a; Hafstein et al., 2014; Li et al., 2015) 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. (Julian, 1999; Julian et al., 1999; Marinósson, 2002a; Marinósson, 2002b; Hafstein, 2007; Giesl and Hafstein, 2014c; Björnsson et al., 2014), 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 and continuous-time systems can largely be done in similar ways. One first constructs a simplicial complex that triangulates the relevant part of the state-space. Then an appropriate linear programming problem for the system at hand is constructed, of which every feasible solution parameterizes a Lyapunov 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 a different method to compute values that can be expected to fulfill the constraints and then verifies if these 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 algorithms in an efficient way. Namely, whereas $\nabla V(\mathbf{x})\bullet\mathbf{f}(\mathbf{x}) < 0$ for a continuous-time system is a local condition that can be formulated as linear constraints for each simplex, the condition $V(\mathbf{g}(\mathbf{x})) - V(\mathbf{x}) < 0$ 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 a linear constraint. 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 solves this problem for fairly general simplicial complexes appropriate for our problem of computing Lyapunov functions.

The CPA algorithm has additionally been adapted

Hafstein, S.
Efficient Algorithms for Simplicial Complexes Used in the Computation of Lyapunov Functions for Nonlinear Systems.
DOI: 10.5220/0006480003980409
In *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2017)*, pages 398-409
ISBN: 978-989-758-265-3

to compute Lyapunov functions for differential inclusions (Baier et al., 2012) and control Lyapunov functions (Baier and Hafstein, 2014) in the sense of the Clarke subdifferential (Clarke, 1990). The next logical step is to compute control Lyapunov functions in the sense of the Dini subdifferential, a work in progress with very promising first results. For these computations one needs information on the common faces of neighbouring simplices in the simplicial complex and, in case of the Dini subdifferential, 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 are 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, a work in progress.

Before we describe our algorithms in Section 2.1 and Section 3, we first discuss suitable triangulations for the computation of CPA Lyapunov functions in Section 2. In Section 3.2 we additionally analyse the efficiency of our algorithm to search fast for simplices and in Section 4 we 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. 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. An inequality for vectors is understood to be componentwise, 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 Euclidian 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 $\|\mathbf{x}\|_\infty := \max_{i=1,2,\ldots,n} |x_i|$. The transpose of $\mathbf{x}$ 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. If $A \in \mathbb{R}^{n\times n}$ is an invertible matrix we denote its inverse by A^{-1} and the inverse of its transpose by A^{-T}. In the rest of the paper n and in the code the global variable `const int n` is the dimension of the Euclidian 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

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

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 $\operatorname{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$ is called an *m-simplex*. For a subset $\{\mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k}\}$, $0 \le k < m$, of affinely independent vectors $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m\}$, the k-simplex $\operatorname{co}(\mathbf{v}_{i_0}, \mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k})$ is called a *k-face* of the simplex $\operatorname{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$. Note that simplices are usually defined as convex combinations of vectors in a set and not of ordered tuples, i.e. $\operatorname{co}\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m\}$ rather than $\operatorname{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_m)$. For the implementation of the simplicial complexes below it is however very useful to stick to ordered tuples.

In the algorithms we make heavy use of the Standard C++ Library and the Armadillo linear algebra library (Sanderson, 2010). 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 Section 2 are also given in (Hafstein, 2013). 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 SIMPLICIAL COMPLEX $\mathcal{T}_{N,K}^{\text{std}}$

The basic simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ and its efficient implementation is described in (Hafstein, 2013). For completeness we recall its definition but refer to (Hafstein, 2013) for the details.

To define the simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ we first need a few 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, $0 \le k < n$. Such a structure is often referred to as a *simplicial n-complex*.

For the definition of $\mathcal{T}_{N,K}^{\text{std}}$ we use the set S_n of all permutations of the numbers $1, 2, \ldots, n$, 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 $\mathcal{J} \subset \{1, 2, \ldots, n\}$ 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 $\mathcal{T}_{N,K}^{\text{std}}$, we first define the triangulations $\mathcal{T}_{N}^{\text{std}}$ and $\mathcal{T}_{K,\text{fan}}^{\text{std}}$ as intermediate steps.

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

1. For every $\mathbf{z} \in \mathbb{N}_0^n$, every $\mathcal{J} \subset \{1,2,\ldots,n\}$, and every $\sigma \in S_n$ define the simplex
$$\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} := \text{co}(\mathbf{x}_0^{\mathbf{z}\mathcal{J}\sigma}, \mathbf{x}_1^{\mathbf{z}\mathcal{J}\sigma}, \ldots, \mathbf{x}_n^{\mathbf{z}\mathcal{J}\sigma}) \quad (1)$$
where
$$\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} := \mathbf{R}^{\mathcal{J}}\left(\mathbf{z} + \sum_{j=1}^{i} \mathbf{e}_{\sigma(j)}\right) \quad (2)$$
for $i = 0,1,2,\ldots,n$,.
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 $\mathcal{T}_N^{\text{std}}$ is defined by
$$\mathcal{T}_N^{\text{std}} := \{\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} : \mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \subset \mathcal{N}\}. \quad (3)$$
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 $\mathcal{T}_N^{\text{std}}$ 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. $\text{co}(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ with exactly n-vertices. For every such intersection add the origin as a vertex to it, i.e. consider $\text{co}(\mathbf{0}, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$. The set of such constructed n-simplices is denoted $\mathcal{T}_{K,\text{fan}}^{\text{std}}$. It is a triangulation of the hypercube $\mathcal{K}$.
4. Finally, we define our main simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ by letting it contain all simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ in $\mathcal{T}_N^{\text{std}}$, that have an empty intersection with the interior $\mathcal{K}^\circ$ of $\mathcal{K}$, and all simplices in the simplicial fan $\mathcal{T}_{K,\text{fan}}^{\text{std}}$. It is thus a triangulation of $\mathcal{N}$ having a simplicial fan in $\mathcal{K}$.

The term *simplicial fan* of the triangulation of the hypercube $\mathcal{K} := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{K}^m \le \mathbf{x} \le \mathbf{K}^p\}$ seems natural, for mathematically it is a straightforward extension of the 3D graphics primitive *triangular fan* to arbitrary dimensions. For a graphical presentation of the complex $\mathcal{T}_{N,K}^{\text{std}}$ with $n = 2$ see Figure 1. For figures of the complex with $n = 3$ see Figures 2 and 3 in (Hafstein, 2013).

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

```
struct T_std_NK {
  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);   // -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<list<int>> SimN;
  vector<list<int>> SCV;
  vector<list<vector<int>>> Faces;
  vector<int> BSim;
  int SVerNr(int s,int i);
  ivec SVer(int s,int i);
};
```

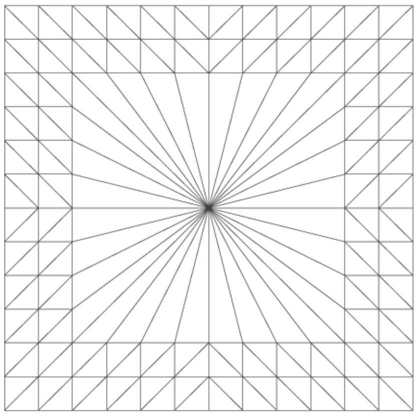
Figure 1: The simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ in two dimensions with $\mathbf{K}^m = (-4,-4)^T, \mathbf{K}^p = (4,4)^T, \mathbf{N}^m = (-6,-6)^T$, and $\mathbf{N}^p = (6,6)^T$.

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, 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. A simplex is basically an ordered tuple of $(n+1)$ vertices. Each simplex is stored as a vector of $(n+1)$-integers, the integers refereing to the positions of the corresponding vertices in vector<ivec> Ver. G is a grid defined by ivec Nm and ivec Np that is used to enumerate all relevant vertices for T_std_NK and vector<zJs> NrInSim and vector<int> Fan are auxiliary structures that allow for the fast computation of a simplex $\mathfrak{S}_\nu$ =Sim[s] such that $\mathbf{x} \in \mathfrak{S}_\nu$ for an arbitrary vector $\mathbf{x} \in \mathbb{R}^n$. Their properties and implementation is described in detail in (Hafstein, 2013). vector<list<int>> SimN, vector<list<int>> SCV, vector<list<vector<int>>> Faces, and vector<int> BSimp are new variables in the class T_std_NK. 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 (Hafstein, 2013). From now on we denote the basic simplicial complex $\mathcal{T}^{\text{std}}_{N,K}$ by $\mathcal{T}$. Further we denote the set of all its vertices by $\mathcal{V}_{\mathcal{T}}$ and its domain by $\mathcal{D}_{\mathcal{T}} := \bigcup_{\mathfrak{S}_\nu \in \mathcal{T}} \mathfrak{S}_\nu$.

Originally we fell back on linear search if $\mathbf{x} \in \mathbb{R}^n$ was in the simplicial fan of $\mathcal{T}^{\text{std}}_{N,K}$. 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. (Polanski, 1997; Branicky, 1998; Johansson and Rantzer, 1998; Johansson, 1999; Polanski, 2000; Ohta, 2001; Ohta and Tsuji, 2003; Yfoulis and Shorten, 2004; Lazar, 2010; Lazar and Jokić, 2010; Lazar and Doban, 2011; Ambrosino and Garone, 2012; Lazar et al., 2013).

We achieve this by first adding such simplices to the `vector<zJs> NrInSim` vector with appropriate values for $\mathbf{z}$, $\mathcal{J}$, and σ. This is very simple to make: In `CODE BLOCK 1` in (Hafstein, 2013) 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 $\mathcal{T}^{\text{std}}_{N,K}$. 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){
  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));
  return found.first->Pos;
  // add this if safety is wanted
  // assert(found.first!=found.second);
  // assert(InSimp(origx,found.first->Pos));
  // where origx is the original x delivered
}
```

The simplices are stored as a `vector` of `vector<int>` of the indices of it's 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<list<int>> SimN` contains the neighbouring simplices for each simplex and `vector<list<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 list of the indices in `Sim` of the simplices neighbouring simplex `Sim[s]` (not including `s` itself) and `SCV[i]` is a sorted list 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<list<int>>::iterator pSCV;
for(pSCV=SCV.begin();pSCV!=SCV.end();pSCV++){
  (*pSCV).sort();
}

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

For every neighbouring simplex `Sim[k]` of the simplex `Sim[s]` we keep track of the common face. For this purpose `T_std_NK` has the member `vector<list<vector<int>>> Faces`. A face is stored as a `vector<int>` of the indices of its vertices in `vector<ivec> Ver`. Each `Faces[s]` is a list of the faces of `Sim[s]` and in the same order as in `SimN[s]`, i.e.

```
list<int>::iterator pSN=SimN[s].begin();
list<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++){
  list<int>::iterator p;
  for(p=SimN[s].begin();p!=SimN[s].end();p++){
    vector<int> F(n);   // Face
    auto F_end=set_intersection(
      Sim[*p].begin(),Sim[*p].end(),
        Sim[s].begin(),Sim[s].end(),
          F.begin());
    F.resize(F_end-F.begin());
    Faces[s].push_back(F);
  }
}
```

The common faces are useful when one uses the CPA method to compute control Lyapunov functions as in (Baier and Hafstein, 2014). Another use is to use the common faces to follow which simplices of $\mathcal{T}$ contribute to the boundary $\partial\mathcal{D}_\mathcal{T}$ of $\mathcal{D}_\mathcal{T}$. We define a simplex $\mathfrak{S}_\nu$ to be an interior simplex in $\mathcal{T}$ if all of its maximal faces, i.e. faces spanned by exactly n of its vertices, are common with other simplices in $\mathcal{T}$. Note that an n-simplex has $\binom{n+1}{n} = n+1$ number of $(n-1)$-faces. If all these $(n-1)$-faces are also faces of other simplices in $\mathcal{T}\setminus\{\mathfrak{S}_\nu\}$, then we define $\mathfrak{S}_\nu$ to be an *interior simplex*. Otherwise, we define $\mathfrak{S}_\nu$ to be a *boundary simplex* in $\mathcal{T}$. 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;
  list<vector<int>>::iterator pF;
  pF=Faces[s].begin()
  for(;pF!=Faces[s].end();pF++){
    if((*pF).size()==n){
      NrMax++;
    }
  }
  if(NrMax<n+1){
    BSim.push_back(s);
  }
}
sort(BSim.begin(),BSim.end());
```

As shown e.g. in (Giesl and Hafstein, 2014c) 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. For discrete time systems $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ analogous propositions hold true (Giesl and Hafstein, 2014a). When actually constructing such a linear programming problem it is most 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 $\mathfrak{S}_\nu := \text{co}(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_n)$ in $\mathcal{T}$ is mapped to the simplex $\mathfrak{S}_\nu^\mathbf{F} = \text{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \dots, \mathbf{F}(\mathbf{v}_n))$ in $\mathcal{T}^\mathbf{F}$. This is implemented by `class FT`, which is the subject of the next section.

3 COMPLEX $\mathcal{T}^\mathbf{F}$ AND `class FT`

As already discussed the simplicial complex $\mathcal{T} = \mathcal{T}_{N,K}^{\text{std}}$ is not adequate for the construction 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 `FT SC` of class `FT` holds a pointer `T_std_NK *pBC` to

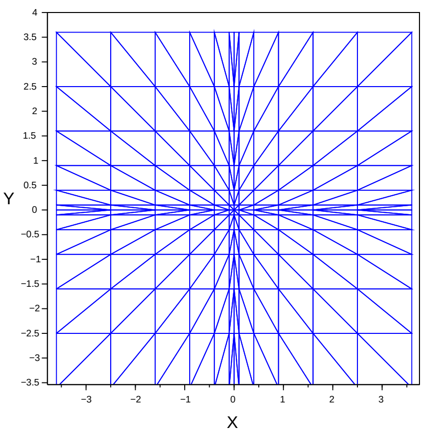

Figure 2: The complex $\mathcal{T}^{\mathbf{F}}$ with $\mathbf{F}(x,y) = (F_1(x), F_2(y))^T$, where $F_1 = F_2$ are strictly monotonically increasing continuous functions, linear on each interval $[k, k+1]$, $k \in \mathbb{Z}$. The smaller simplices at the origin lead to long and thin simplices further from the origin.

an underlying basic simplicial complex $\mathcal{T}$ and a mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$. The relationship between $\mathcal{T}$ and $\mathcal{T}^{\mathbf{F}}$ is that $\mathfrak{S}_\nu^{\mathbf{F}} = \mathrm{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n)) \in \mathcal{T}^{\mathbf{F}}$, if and only if $\mathfrak{S}_\nu := \mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathcal{T}$. Of course the mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ must be chosen such that $\mathcal{T}^{\mathbf{F}}$ is an admissible simplicial complex.

If $\mathbf{F}$ is linear, i.e. there exists a matrix $F \in \mathbb{R}^{n\times n}$ such that $\mathbf{F}(\mathbf{x}) = F\mathbf{x}$, then since

$$\sum_{i=0}^{n} \lambda_i \mathbf{F}(\mathbf{v}_i) = \sum_{i=0}^{n} \lambda_i F\mathbf{v}_i = F \sum_{i=0}^{n} \lambda_i \mathbf{v}_i = \mathbf{F}\left(\sum_{i=0}^{n} \lambda_i \mathbf{v}_i\right)$$

we have

$$\begin{aligned}\mathfrak{S}_\nu^{\mathbf{F}} &:= \mathrm{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n)) \\ &= \mathrm{co}(F\mathbf{v}_0, F\mathbf{v}_1, \ldots, F\mathbf{v}_n) \\ &= F\,\mathrm{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n) \\ &= \mathbf{F}(\mathfrak{S}_\nu)\end{aligned}$$

For many systems, however, one would like to use smaller simplices close to the origin than further away. This can be addressed as in e.g. (Marinósson, 2002a; Marinósson, 2002b; Hafstein, 2007) by setting

$$\mathbf{F}(\mathbf{x}) = (F_1(x_1), F_2(x_2), \ldots, F_n(x_n))^T, \qquad (4)$$

where the $F_i : \mathbb{R} \to \mathbb{R}$ are strictly monotonically increasing continuous functions, linear on each interval $[k, k+1]$, $k \in \mathbb{Z}$. Then $\mathbf{F}$ restricted to any $\mathbf{z} + [0,1]^n$, $\mathbf{z} \in \mathbb{Z}^n$, is linear and again $\mathfrak{S}_\nu^{\mathbf{F}} = \mathbf{F}(\mathfrak{S}_\nu)$ for every simplex $\mathfrak{S}_\nu \in \mathcal{T}$. The problem with this approach can be seen in Figure 2. Smaller simplices at the origin lead to long and thin simplices further a away.

Further, in many concrete examples is seems natural that the simplicial complex should be at least approximately isotropic as viewed from the equilibrium at the origin. Neither of these properties can be achieved by using a linear $\mathbf{F}$ or an $\mathbf{F}$ as in (4). One mapping that fulfills these properties is

$$\mathbf{F}(\mathbf{x}) = \rho(\|\mathbf{x}\|_\infty) \cdot \frac{\|\mathbf{x}\|_\infty}{\|\mathbf{x}\|_2}\mathbf{x}, \qquad (5)$$

where $\rho : \mathbb{R}_+ \to \mathbb{R}_+$ is a nondecreasing continuous function. Note that then $\|\mathbf{F}(\mathbf{x})\|_2 = \rho(\|\mathbf{x}\|_\infty)\|\mathbf{x}\|_\infty$, i.e. $\mathbf{F}$ maps the hypercube $[-a,a]^n$ bijectively to the closed hypersphere centered at the origin and with radius $\rho(a)a$. See Figure 3 for a picture of $\mathcal{T}^{\mathbf{F}}$ with $\mathbf{F}$ as in (5) and with $n = 2$.

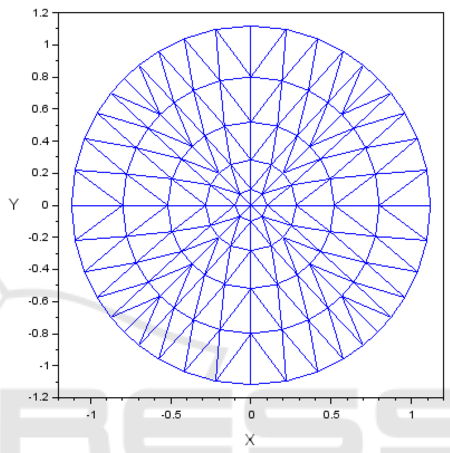

Figure 3: The complex $\mathcal{T}^{\mathbf{F}}$ with $\mathbf{F}$ as in (5) and with $\rho(x) = 0.1\sqrt{x}$.

Such $\mathbf{F}$ have been used for example in (Baier et al., 2012; Björnsson et al., 2014; Baier and Hafstein, 2014; Björnsson et al., 2015). As shown later in this section some algorithms can be implemented much more efficiently if a formula for the inverse $\mathbf{F}^{-1}$ of $\mathbf{F}$ is available. If $\rho(x) = sx^{q-1}$ for some $s, q > 0$, i.e.

$$\mathbf{F}(\mathbf{x}) = \frac{s\|\mathbf{x}\|_\infty^q}{\|\mathbf{x}\|_2}\mathbf{x}, \qquad (6)$$

then the the inverse of $\mathbf{F}$ is given by the formula

$$\mathbf{F}^{-1}(\mathbf{x}) = \left(\frac{\|\mathbf{x}\|_2}{s\|\mathbf{x}\|_\infty^q}\right)^{\frac{1}{q}} \mathbf{x} = \frac{1}{\|\mathbf{x}\|_\infty}\left(\frac{\|\mathbf{x}\|_2}{s}\right)^{\frac{1}{q}} \mathbf{x}. \qquad (7)$$

Note that for $\mathbf{F}$ as in (5) the simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ is not equal to $\mathbf{F}(\mathfrak{S}_\nu)$. Indeed, $\mathbf{F}(\mathfrak{S}_\nu)$ is not even a simplex, cf. Figure 4. A contribution of this paper is a fast algorithm to search for a given $\mathbf{x} \in \mathbb{R}^n$ a simplex $\mathfrak{S}_\nu^{\mathbf{F}} \in \mathcal{T}^{\mathbf{F}}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$, cf. Section 3.2, when the inverse of $\mathbf{F}$ is available but $\mathfrak{S}_\nu^{\mathbf{F}} \neq \mathbf{F}(\mathfrak{S}_\nu)$.

3.1 Implementation of `class FT`

Let us now discuss the implementation of `class FT` modelling $\mathcal{T}^{\mathbf{F}}$. Its definition is:

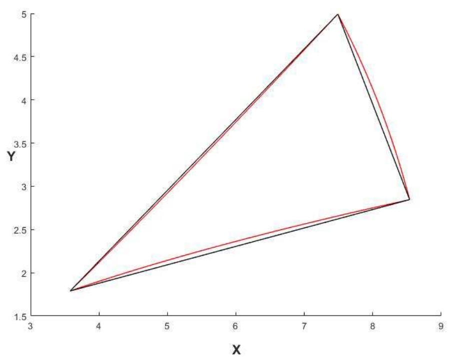

Figure 4: For the simplex $\mathfrak{S}_\nu = \text{co}\{(2,1)^T, (3,1)^T, (3,2)^T\}$ and the mapping $\mathbf{F}(\mathbf{x}) = \|\mathbf{x}\|_\infty^2/\|\mathbf{x}\|_2\,\mathbf{x}$ the simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ (black) and the set $\mathbf{F}(\mathfrak{S}_\nu)$ (red/grey).

```
class FT {
public:
  T_std_K *pBC;
  function<vec(vec)> pF, ipF;
  vec F(vec x);
  vec F(ivec x);
  vec iF(vec x);
  vector<vec> xVer;
  vector<mat> XmT;
  vector<vec> SCC;
  vector<double> SCR;
  vector<list<mat>> Fnor;
  FT(T_std_K *_pBC, function<vec(vec)> _pF,
    function<vec(vec)> _ipF=nullptr);
  ~FT();
  int InSimpNrSlow(vec x,vec &L);
  int InSimpNrFast(vec x,vec &L);
  int InSimpNrAppr(vec x);
  bool InSimp(vec x,int s,vec &L);
  int SVerNr(int s,int i);
  vec SVer(int s,int i);
  bool CFSS;  // default value "true"
};
```

Let us first discuss the constructor of `FT`. The class `FT` holds a pointer `T_std_NK *pBC` to the underlying basic simplicial complex, which is initialized by `T_std_NK *_pBC` in the constructor. The mapping $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ that is used to map the vertices of the basic simplicial complex is stored as `function<vec(vec)> pF`, that is initialized by `function<vec(vec)> _pF` in the constructor, and can be called with `vec F(vec x)` and `vec 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));
}
```

If the inverse mapping $\mathbf{F}^{-1}$ of $\mathbf{F}$ is available, it can be used to decrease the computational complexity of several algorithms considerably. It is stored as `function<vec(vec)> ipF` and is initialized by `function<vec(vec)> _ipF` in the constructor. If it is not available we initialize `ipF=nullptr`. The implementation of the member functions `vec iF(vec x)` is analogous to the implementation of `vec F(vec x)`, just with `function<vec(vec)> ipF` instead of `function<vec(vec)> pF`. To avoid that one delivers a wrong $\mathbf{F}^{-1}$, an error that is bound to happen, it is actually tested that $\mathbf{F}^{-1}(\mathbf{F}(\mathbf{x})) = \mathbf{x}$ for a random sample in the domain under consideration, i.e. $\mathcal{D}_\mathcal{T}$. Additionally, we verify $\mathbf{F}^{-1}(\mathbf{0}) = \mathbf{0}$. 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.

For computational purposes it is advantageous to 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. This is implemented in the constructor of `FT` in the obvious way:

```
vector<ivec>::iterator p;
for(p=pBC->Ver.begin();p!=pBC->Ver.end();p++){
  xVer.push_back(F(*p));
}
```

The implementation of `int FT::SVerNr(int s,int i)` and `vec FT::SVer(int s,int i)` is analogous to the implementation of these member functions in `T_std_NK`:

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

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

Further, `FT` stores for each simplex $\mathfrak{S}_\nu^\mathbf{F}$ the matrix X_ν^{-T}, i.e. the inverse of the transpose of the matrix X_ν, where X_ν is the so-called shape-matrix of $\mathfrak{S}_\nu^\mathbf{F}$, cf. (Hafstein et al., 2015). 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]` corresponds to the simplex `pBC->Sim[s]`. 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 ready 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]`. A formula for the center $\mathbf{c} \in \mathbb{R}^n$ of the circumscribing hypersphere of $\mathfrak{S}_\nu^{\mathbf{F}} = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n)$ can be derived from the conditions that

$$\|\mathbf{c} - \mathbf{v}_i\|_2^2 = r^2 \ \text{ for } i = 0, 1, \ldots, n,$$

where r is the radius of the circumscribed hypersphere. Thus

$$\begin{aligned} 0 &= \|\mathbf{c} - \mathbf{v}_i\|_2^2 - \|\mathbf{c} - \mathbf{v}_0\|_2^2 \\ &= -2\mathbf{c} \bullet (\mathbf{v}_i - \mathbf{v}_0) + \|\mathbf{v}_i\|_2^2 - \|\mathbf{v}_0\|_2^2, \end{aligned}$$

which can be written as

$$\mathbf{c} \bullet (\mathbf{v}_i - \mathbf{v}_0) = \frac{1}{2}(\mathbf{v}_i - \mathbf{v}_0) \bullet (\mathbf{v}_i + \mathbf{v}_0).$$

Subtracting $\mathbf{v}_0 \bullet (\mathbf{v}_i - \mathbf{v}_0)$ from both sides delivers

$$(\mathbf{c} - \mathbf{v}_0) \bullet (\mathbf{v}_i - \mathbf{v}_0) = \frac{1}{2}\|\mathbf{v}_i - \mathbf{v}_0\|_2^2,$$

which gives us a linear equation for $\mathbf{c}$. Set $\mathbf{b} = (b_1, b_2, \ldots, b_n)^T$ with $b_i = \|\mathbf{v}_i - \mathbf{v}_0\|_2^2$ for $i = 1, 2, \ldots, n$. Then

$$X_\nu(\mathbf{c} - \mathbf{v}_0) = \frac{1}{2}\mathbf{b}$$

so

$$\mathbf{c} = \mathbf{v}_0 + \frac{1}{2}X_\nu^{-1}\mathbf{b}.$$

The radius of the circumscribing hypersphere can subsequently be computed as

$$r = \|\mathbf{c} - \mathbf{v}_0\|_2.$$

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 check efficiently if a vector $\mathbf{x} \in \mathbb{R}^n$ is in a particular simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ =`Sim[s]` or not. Recall that for a simplex $\mathfrak{S}_\nu^{\mathbf{F}} = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ and a vector $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$ if and only if $\mathbf{x}$ can be written as a convex combination of the vertices of the simplex. This means that there are numbers $\lambda_0, \lambda_1, \ldots, \lambda_n \geq 0$, $\sum_{i=0}^n \lambda_i = 1$, such that $\mathbf{x} = \sum_{i=0}^n \lambda_i \mathbf{v}_i$ which in turn is equivalent to

$$\mathbf{x} - \mathbf{v}_0 = \sum_{i=1}^{n} \lambda_i(\mathbf{v}_i - \mathbf{v}_0). \qquad (8)$$

Thus, with $\mathbf{L}^* = (\lambda_1, \lambda_2, \ldots, \lambda_n)^T$ equation (8) is equivalent to

$$X_\nu^T \mathbf{L}^* = \mathbf{x} - \mathbf{v}_0 \ \text{ or } \ \mathbf{L}^* = X_\nu^{-T}(\mathbf{x} - \mathbf{v}_0). \qquad (9)$$

Now $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$ if and only if the components of $\mathbf{L}^*$ are all nonnegative and sum up to a number ≤ 1. We then set $\lambda_0 = 1 - \sum_{i=0}^n \lambda_i$. This is implemented as follows with `L`$= (\lambda_0, \lambda_1, \ldots, \lambda_n)^T$:

```
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;
  }
}
```

Note that `vec L` contains the barycentric coordinates of $\mathbf{x}$ if it is in `Sim[s]`.

We now come to the discussion of the normals. Consider a simplex $\mathfrak{S}_\nu = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathcal{T}^{\mathbf{F}}$. Another way to characterize the simplex is to define it as the intersection of $(n+1)$ half-spaces. These half-spaces can be constructed by performing the following for each $i = 0, 1, \ldots, n$:

Set $\mathbf{y}_n := \mathbf{v}_i$ and pick an arbitrary vector $\mathbf{y}_0 \in \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n\} \setminus \{\mathbf{v}_i\}$. Set $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{n-1}\} = \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n\} \setminus \{\mathbf{y}_0, \mathbf{y}_n\}$ and define the $n \times n$ matrix

$$Y_{i,\nu} := (\mathbf{y}_1 - \mathbf{y}_0, \mathbf{y}_2 - \mathbf{y}_0, \ldots, \mathbf{y}_n - \mathbf{y}_0)^T.$$

Because the vectors $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n$ are affinely independent the matrix $Y_{i,\nu}$ is non-singular and the equation

$$Y_{i,\nu}^T \mathbf{x} = \mathbf{e}_n$$

has a unique solution $\mathbf{n}_i^\nu = Y_{i,\nu}^{-T}\mathbf{e}_n$ for $\mathbf{x}$. Note that since

$$\mathbf{n}_i^\nu \bullet (\mathbf{y}_j - \mathbf{y}_0) = \mathbf{e}_n^T Y_{i,\nu}^{-1}(\mathbf{y}_j - \mathbf{y}_0) = \mathbf{e}_n^T \mathbf{e}_j = \delta_{j,n},$$

the vector $\mathbf{n}_i^\nu$ is normal to the unique hyperplane containing the vertices $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n\} \setminus \{\mathbf{v}_i\}$ of $\mathfrak{S}_\nu^{\mathbf{F}}$. Further we have for every vector $\mathbf{x} \in \mathfrak{S}_\nu^{\mathbf{F}}$ that $\mathbf{x}$ is the convex sum of the vertices $\mathbf{v}_j$ of $\mathfrak{S}_\nu^{\mathbf{F}}$ and since the $\mathbf{y}_j$ are the same vectors rearranged we have

$$\mathbf{x} = \sum_{j=0}^n \lambda_j \mathbf{y}_j,\ 0 \le \lambda_j \le 1,\ \sum_{j=0}^n \lambda_j = 1.$$

Hence

$$\begin{aligned}\mathbf{n}_i^\nu \bullet (\mathbf{x} - \mathbf{y}_0) &= \mathbf{n}_i^\nu \bullet \left(\sum_{j=0}^n \lambda_j \mathbf{y}_j - \mathbf{y}_0\right) \\ &= \mathbf{n}_i^\nu \bullet \sum_{j=1}^n \lambda_j (\mathbf{y}_j - \mathbf{y}_0) \\ &= \sum_{j=1}^n \lambda_j \delta_{j,n} = \lambda_n \ge 0.\end{aligned}$$

Thus for any $i = 0, 1, \ldots, n$ the simplex $\mathfrak{S}_\nu^{\mathbf{F}}$ is a subset of the half-space

$$\mathcal{H}_i^\nu := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{n}_i^\nu \bullet (\mathbf{x} - \mathbf{y}_0) \ge 0\}.$$

Moreover, for each $i = 0, 1, \ldots, n$ the vertices $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n\} \setminus \{\mathbf{v}_i\}$ are all in the hyperplane $\partial\mathcal{H}_i^\nu$. It is easily verified that

$$\mathfrak{S}_\nu^{\mathbf{F}} = \bigcap_{i=0}^n \mathcal{H}_i^\nu. \tag{10}$$

For our application in constructing linear programming problems for the computation of control Lyapunov functions using the Dini-subdifferential it is advantageous to store a little more information. For each $\mathfrak{S}_\nu^{\mathbf{F}}$ we consider all its neighbouring simplices $\mathfrak{S}_\mu^{\mathbf{F}}$. For each neighbouring simplex $\mathfrak{S}_\mu^{\mathbf{F}}$ consider their common face $\mathrm{co}(\mathbf{f}_0, \mathbf{f}_1, \ldots, \mathbf{f}_r)$. For each half-space $\mathcal{H}_k^\mu$ such that the vertices $\mathbf{f}_0, \mathbf{f}_1, \ldots, \mathbf{f}_r$ are all in the hyperplane $\partial\mathcal{H}_k^\mu$ we store the corresponding normal $\mathbf{n}_k^\mu$. This can be done by using the `vector<list<int>> T_std_NK::SimN` and `vector<list<vector<int>>> T_std_NK::Faces` of the basic simplicial complex pointed to by `pBC`. First we construct the matrix $Y_{i,\mu}$ by setting $\mathbf{y}_0 = \mathbf{f}_0$. The subscript i is arbitrary, except that the i-th vertex $\mathbf{u}_i$ of $\mathfrak{S}_\mu^{\mathbf{F}}$ should of course not be the vector $\mathbf{f}_0$. Then for every vertex $\mathbf{u}_k$ of $\mathfrak{S}_\mu^{\mathbf{F}} = \mathrm{co}(\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_n)$ such that $\mathbf{u}_k \notin \{\mathbf{f}_0, \mathbf{f}_1, \ldots, \mathbf{f}_r\}$ we compute $\mathbf{n}_k^\mu := Y_{i,\mu}^{-T}\mathbf{e}_m$, where m is such that $\mathbf{u}_k - \mathbf{y}_0$ is the m-th line in $Y_{i,\mu}$. Since the vertices in $\mathfrak{S}_\mu^{\mathbf{F}}$=`Sim[mu]` are in some particular order we just iterate through them and write the $\mathbf{n}_k^\mu$ in the same order in the columns of the matrix `Fnor[nu][mu]`, where `Sim[nu]` corresponds to the simplex $\mathfrak{S}_\nu$. This is implemented as follows:

```
Fnor.resize(pBC->Sim.size());
for(int nu=0;nu<pBC->Sim.size();nu++){
  vector<int>::iterator pSN;
  vector<vector<int>>::iterator pNP;
  pSN=pBC->SimN[nu].begin();
  pNP=pBC->Faces[nu].begin();
  for(;pSN!=pBC->SimpN[nu].end();pSN++,pNP++){
    int mu=*pSN;
    vec y0=Vertices[(*pNP)[0]];
    int numf=(*pNP).size();
    mat Y(n,n),Z(n,n-numf+1,fill::zeros);
    int j=0,k=0;
    for(int i=0;i<=n;i++){
      if(SVerNr(mu,i)!=(*pNP)[0]){
        Y.col(j)=SVer(mu,i)-y0;
        if(binary_search((*pNP).begin(),
        (*pNP).end(),SVerNr(mu,i))==false){
          // vertex i of Sim[mu] is not in
          // the face
          Z(j,k++)=1.0;
        }
        j++;
      }
    }
    Fnor[nu].push_back(solve(Y.t(),Z));
  }
}
```

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

Given an $\mathbf{x} \in \mathbb{R}^n$ one is often interested in finding 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 return value is set to the impossible value -1.

If a formula for the inverse mapping $\mathbf{F}^{-1}$ of $\mathbf{F}$ is available we can do much better. For some application, e.g. when plotting a computed Lyapunov function, it is sometimes enough to get a simplex $\mathfrak{S}_\nu^\mathbf{F}$ such that $\mathbf{x}$ is close to $\mathfrak{S}_\nu^\mathbf{F}$. This can be done by finding a simplex $\mathfrak{S}_\nu$ in $\mathcal{T}$, such that $\mathbf{y} := \mathbf{F}^{-1}(\mathbf{x}) \in \mathfrak{S}_\nu$. Recall the such an $\mathfrak{S}_\nu$ can be computed very efficiently in the member function `T_std_NK::InSimpNr(vec x)` as described in (Hafstein, 2013). This is implemented as

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

For many applications this is, however, not sufficient. An example is when a linear programming problem for discrete-time dynamical systems is constructed, cf. (Giesl and Hafstein, 2014a; Hafstein et al., 2014; Li et al., 2015). Then a simplex $\mathfrak{S}_\nu^\mathbf{F}$ such that $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$ and the barycentric coordinates of $\mathbf{x}$ are necessary.

If one can be sure that $\mathbf{x} \notin \mathcal{D}_\mathcal{T}^\mathbf{F} := \bigcup_{\mathfrak{S}_\nu \in \mathcal{T}} \mathfrak{S}_\nu^\mathbf{F}$ follows from $\mathbf{F}^{-1}(\mathbf{x}) \notin \mathcal{D}_\mathcal{T}$, one can set the member variable `bool CFSS` (careful simplex search), whose default value is `true`, equal to `false`. Note that this holds true in the important case that $\mathbf{F}$ and $\mathbf{F}^{-1}$ are as in (6) and (7) and $\mathbf{N}^p = -\mathbf{N}^m = (N, N, \ldots, N)^T$ for a positive integer N in $\mathcal{T} = \mathcal{T}_{N,K}^{\text{std}}$ (`Np=ivec(n).fill(N)` and `Nm=-Np` in the basic complex pointed to by `pBC`). The reason is that then $\mathbf{F}$ maps the hypercube $\mathcal{D}_\mathcal{T} = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_\infty \leq N\}$ onto the closed hypersphere $\mathcal{S}_R \subset \mathbb{R}^n$ centered at the origin and with radius $R = sN^q$. Since $\mathcal{S}_R$ is convex it clearly contains $\mathcal{D}_\mathcal{T}^\mathbf{F}$ and consequently $\mathbf{F}^{-1}(\mathcal{D}_\mathcal{T}^\mathbf{F}) \subset \mathbf{F}^{-1}(\mathcal{S}_R) = \mathcal{D}_\mathcal{T}$.

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 $\mathbf{x} \in \mathbb{R}^n$ compute a simplex $\mathfrak{S}_\nu \in \mathcal{T}$ such that $\mathbf{F}^{-1}(\mathbf{x}) \in \mathfrak{S}_\nu$. If $\mathbf{x} \in \mathfrak{S}_\nu^\mathbf{F}$ we are finished. If not check if $\mathbf{x} \in \mathfrak{S}_\mu^\mathbf{F}$ for all 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 $\mathbf{x} \in \mathfrak{S}_\xi^\mathbf{F}$ for all neighbouring simplices $\mathfrak{S}_\xi^\mathbf{F}$ of the simplices $\mathfrak{S}_\mu^\mathbf{F}$ that have not already been checked. Repeat this as necessary. The implementation is as follows:

```
int FT::InSimpNrFast(vec x,vec &L){
  int s=InSimpNrAppr(vec 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;
  }
  list<int> TC, CH, CN;
  // TC = To Check
  // CH = already CHecked
  // CN = Check Next
  TC=pBC->SimN[s];
  list<int>::iterator p;
  for(p=TC.begin();p!=TC.end();p++){
    if(InSimp(x,*p,L)==true){
      return *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++){
    list<int> A2CN=pBC->SimN[*p];
      CN.insert(CN.end(),
          A2CN.begin(),A2CN.end());
    }
    CN.sort();
    CN.unique();
    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());
    CH.sort();
  }
  return InSimpNrSlow(x,L);
}
```

We have two comments on this algorithm. First, if the main loop is used to search through the whole complex, then it is considerably slower than a linear search 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 through the whole complex. 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. Second, instead of using all neighbours one can instead use only neighbours with common faces that are maximal, i.e. $(n-1)$-simplices (n common vertices). This actually turned out to be faster in many cases, but since the essential idea is the same we do not discuss it further here.

To give an idea of the improvement in running times in comparison to linear search see Table 1. In all runs we generated 10,000 uniformly distributed random points $\mathbf{x}$ in the hypercube $[-N, N]^n$ and searched for a simplex $\mathfrak{S}_\nu^\mathbf{F}$ such that $\mathbf{F}(\mathbf{x}) \in \mathfrak{S}_\nu^\mathbf{F}$.

As an example, if one wants to generate a linear programming problem to compute a Lyapunov func-

Table 1: The running times for the search of a simplex $\mathfrak{S}_\nu^{\mathbf{F}} \in \mathcal{T}^{\mathbf{F}}$ such that $\mathbf{x} \in \mathcal{T}^{\mathbf{F}}$ for a random sample of 10,000 points $\mathbf{x}$. The mapping $\mathbf{F}$ was as in (6) with $s = 0.1$ and $q = 1.5$. n is the dimension and N and K are the parameters of the complex $\mathcal{T} = \mathcal{T}_{N,K}^{\text{std}}$. [neigh.] is the proportion of simplices such that we do not get a direct hit, i.e. $\mathbf{x} \notin \mathfrak{S}_\nu^{\mathbf{F}}$ although $\mathbf{F}^{-1}(\mathbf{x}) \in \mathfrak{S}_\nu$. [lin.] is the time needed for linear search and [opt.] is the time needed for the new algorithm. The algorithms were run on a PC machine: i4790k@4600MHz with 32GB RAM.

n	N	K	neig.[%]	lin. [sec]	opt.[sec]
2	20	0	1.1	0.699	0.033
2	20	5	0.87	0.670	0.066
2	50	0	0.44	4.445	0.039
2	50	5	0.39	4.417	0.046
2	50	10	0.35	4.313	0.079
2	100	0	0.24	21.61	0.041
2	100	5	0.24	21.98	0.042
2	200	0	0.14	90.50	0.046
2	200	10	0.14	90.29	0.051
3	15	0	2.28	49.33	0.169
3	15	5	2.6	48.93	0.862
3	25	0	1.67	245.8	0.218
3	25	5	1.60	229.6	0.376
4	10	0	6.40	1154	10.12

tion for a time-discrete system $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ using the simplicial complex in the table with $n = 3$, $N = 25$, and $K = 0$, which has $51^3 = 132,651$ vertices and $3! \cdot 50^3 = 750,000$ simplices, the running time for the search for simplices $\mathfrak{S}_\nu^{\mathbf{F}}$ such that $\mathbf{g}(\mathbf{x}) \in \mathfrak{S}_\nu^{\mathbf{F}}$ for all the vertices $\mathbf{x}$ of the complex is reduced from 54 minutes to less than 3 seconds using the new algorithm!

4 SUMMARY

We described the efficient implementation of simplicial complexes for the computation of Lyapunov functions for nonlinear systems. This paper is a logical sequel to the papers (Hafstein, 2013; Giesl and Hafstein, 2014b; Björnsson et al., 2015) and explains how to adapt algorithms for the simplicial complex to compute Lyapunov functions for time-discrete systems and control systems. An additional bonus is that some of the algorithms can be used for different computational methods that depend on conic partitions of the state-space as discussed in Section 2.1.

The probably best know method for computing Lyapunov functions for rather general nonlinear systems is the so-called SOS (sum-of-squares) method, cf. e.g. (Parrilo, 2000; Peet and Papachristodoulou, 2012; Peet, 2009; Anderson and Papachristodoulou, 2015). A reason for this is definitely that its implementation is not too difficult and that a well documented Matlab library, SOSTOOLS, is available (Papachristodoulou et al., 2013). It is the hope of the author that this paper brings the techniques necessary for the efficient implementation of the CPA method to compute Lyapunov functions for nonlinear systems closer to interested researchers. It definitely is an important milestone in the development of an easy-to-use C++ program for such computations for very general nonlinear systems.

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

Ambrosino, R. and Garone, E. (2012). 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*, pages 2872–2877, Maui (HI), USA.

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.

Baier, R., Grüne, L., and Hafstein, S. (2012). Linear programming based Lyapunov function computation for differential inclusions. *Discrete Contin. Dyn. Syst. Ser. B*, 17(1):33–56.

Baier, R. and Hafstein, S. (2014). 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)*, pages 1173–1180 (no. 0232), Groningen, The Netherlands.

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*, Los Angeles (CA), USA.

Björnsson, J., Gudmundsson, S., and Hafstein, S. (2015). 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*, number 0155, pages 788–793.

Branicky, M. (1998). Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Trans*, 43(4):475–482.

Clarke, F. (1990). *Optimization and Nonsmooth Analysis*. Classics in Applied Mathematics. SIAM.

Giesl, P. and Hafstein, S. (2014a). Computation of Lyapunov functions for nonlinear discrete systems by linear programming. *J. Difference Equ. Appl.*, 20:610–640.

Giesl, P. and Hafstein, S. (2014b). Implementation of a fan-like triangulation for the CPA method to compute Lyapunov functions. In *Proceedings of the 2014 American Control Conference*, pages 2989–2994 (no. 0202), Portland (OR), USA.

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

Hafstein, S. (2007). *An algorithm for constructing Lyapunov functions*, volume 8 of *Monograph*. Electron. J. Diff. Eqns.

Hafstein, S. (2013). Implementation of simplicial complexes for CPA functions in C++11 using the armadillo linear algebra library. In *Proceedings of SIMULTECH*, pages 49–57, Reykjavik, Iceland.

Hafstein, S., Kellett, C., and Li, H. (2014). Computation of Lyapunov functions for discrete-time systems using the Yoshizawa construction. In *Proceedings of CDC*.

Hafstein, S., Kellett, C., and Li, H. (2015). Computing continuous and piecewise affine lyapunov functions for nonlinear systems . *Journal of Computational Dynamics*, 2(2):227 – 246.

Johansson, M. (1999). *Piecewise Linear Control Systems*. PhD thesis: Lund University, Sweden.

Johansson, M. and Rantzer, A. (1998). Computation of piecewise quadratic Lyapunov functions for hybrid systems. *IEEE Trans. Automat. Control*, 43(4):555–559.

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.

Lazar, M. (2010). On infinity norms as Lyapunov functions: Alternative necessary and sufficient conditions. In *Proceedings of the 49th IEEE Conference on Decision and Control*, pages 5936–5942, Atlanta, USA.

Lazar, M. and Doban, A. (2011). On infinity norms as Lyapunov functions for continuous-time dynamical systems. In *Proceedings of the 50th IEEE Conference on Decision and Control*, pages 7567–7572, Orlando (Florida), USA.

Lazar, M., Doban, A., and Athanasopoulos, N. (2013). On stability analysis of discrete-time homogeneous dynamics. In *Proceedings of the 17th International Conference on systems theory, control and computing*, pages 297–305, Sinaia, Romania.

Lazar, M. and Jokić, A. (2010). On infinity norms as Lyapunov functions for piecewise affine systems. In *Proceedings of the Hybrid Systems: Computation and Control conference*, pages 131–141, Stockholm, Sweden.

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.

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.

Ohta, Y. (2001). On the construction of piecewise linear Lyapunov functions. In *Proceedings of the 40th IEEE Conference on Decision and Control.*, volume 3, pages 2173–2178.

Ohta, Y. and Tsuji, M. (2003). A generalization of piecewise linear Lyapunov functions. In *Proceedings of the 42nd IEEE Conference on Decision and Control.*, volume 5, pages 5091–5096.

Papachristodoulou, A., Anderson, J., Valmorbida, G., Pranja, S., Seiler, P., and Parrilo, P. (2013). *SOSTOOLS: Sum of Squares Optimization Toolbox for MATLAB*. User's guide. Version 3.00 edition.

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

Peet, M. (2009). Exponentially stable nonlinear systems have polynomial Lyapunov functions on bounded regions. *IEEE Trans. Automat. Control*, 54(5):979 – 987.

Peet, M. and Papachristodoulou, A. (2012). A converse sum of squares Lyapunov result with a degree bound. *IEEE Transactions on Automatic Control*, 57(9):2281–2293.

Polanski, A. (1997). Lyapunov functions construction by linear programming. *IEEE Trans. Automat. Control*, 42:1113–1116.

Polanski, A. (2000). On absolute stability analysis by polyhedral Lyapunov functions. *Automatica*, 36:573–578.

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

Yfoulis, C. and Shorten, R. (2004). A numerical technique for the stability analysis of linear switched systems. *Int. J. Control*, 77(11):1019–1039.