# Implementation of Simplicial Complexes for CPA functions in C++11 using the Armadillo Linear Algebra Library

Sigurdur Freyr Hafstein[1]
[1]*School of Science and Engineering, Reykjavik University, Menntavegur 1, 101 Reykjavik, Iceland*
*sigurdurh@ru.is*

Keywords: CPA function, Lyapunov function, piecewise linear, nonlinear system, triangulation, simplicial complex, C++11, Armadillo linear algebra library.

Abstract: Continuous, piecewise affine (CPA) functions can be algorithmically parameterized to deliver Lyapunov functions for compact invariant sets. We discuss flexible structures and algorithms to manipulate CPA functions for these purposes and discuss their implementation in C++11 using the Armadillo linear algebra library. Especially, we discuss some of the new language features in C++11 that lead to simpler and more readable code. The implementation was developed in the freeware Visual Studio Express 2012 for Windows Desktop (VS2012). Apart from a detailed description and code examples for the construction and manipulation of the simplicial complex that serves as a basis for CPA functions, this contribution includes some discussion on practical implementation details when using VS2012, C++11, and the linking to and use of the excellent Armadillo linear algebra library. Thus, some parts of this paper, especially Section 3, might be useful not only for those interested in the implementation of the simplicial complex for computing CPA Lyapunov functions, but also for those generally interested in using the free Armadillo library for computations in VS2012.

## 1 INTRODUCTION

Lyapunov functions are a fundamental concept in the study of dynamical systems. Their central role in studies of the stability behavior of dynamical systems is well known. Their construction is, however, difficult in the general case, i.e. for nonlinear systems.

Several methods to numerically compute Lyapunov functions for nonlinear systems have been suggested. To name a few, in (Johansson and Rantzer, 1997) a construction method for piecewise quadratic Lyapunov functions for piecewise affine autonomous systems is suggested. In (Eghbal, Pariz, and Karimpour, 2012) the computation of piecewise quadratic Lyapunov functions for planar piecewise affine systems is formulated as linear matrix inequalities. In (Johansen, 2000) linear programming is used to parameterize Lyapunov functions for autonomous nonlinear systems. In (Rezaiee-Pajand and Moghaddasie, 2012) a different collocation method using two classes of basis functions is suggested. In (Giesl, 2007) radial basis functions are used to solve numerically a generalized Zubov equation. In (Peet and Papachristodoulou, 2010) the numerical construction of Lyapunov functions that are presentable as sum of squares of polynomials is considered. The Lyapunov functions are computed by means of convex optimization.

One method that has been studied in some detail recently, uses linear programming to parameterize CPA Lyapunov functions in compact neighbourhoods of exponentially stable equilibria. This approach was first followed in (Julian, Guivant, and Desages, 1999) and was enhanced in (Marinosson, 2002a and 2002b) to compute true Lyapunov functions, rather than approximations requiring a posteriori analysis to determine their quality. In (Hafstein, 2004 and 2005) it was proved that when an arbitrary small hypercube around the equilibrium is excluded from the domain of the to be computed CPA Lyapunov function, the computation would always succeed. The domain of the computed CPA Lyapunov function is otherwise only limited to any compact subset of the equilibrium's region of attraction.

In (Giesl and Hafstein, 2012 and 2013) the necessity of excluding an arbitrary small hypercube around the equilibrium was removed, at the expense of needing a more refined simplicial complex than in pervious works. In this paper we will discuss the implementation of this novel simplicial complex that possesses a simplicial fan at the equilibrium.

The term *simplicial fan* seems natural, for math-

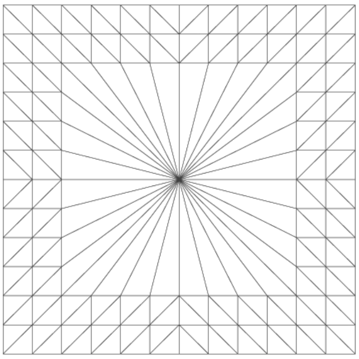

Figure 1: The simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ in two dimensions with $\mathbf{K}^m = (-4,-4)^T, \mathbf{K}^p = (4,4)^T, \mathbf{N}^m = (-6,-6)^T$, and $\mathbf{N}^p = (6,6)^T$.

ematically it is a straightforward extension of the 3D graphics primitive *triangular fan* to arbitrary dimensions. For graphical examples of the simplicial complexes discussed in this paper see Figure 1 and 2.

In Section 2 we define the simplicial complex mathematically. In Section 3 we give a short description of how to include Armadillo in a VS2012 project and discuss the basics of the Armadillo library and then we define in Section 4 the data-structures `Grid`, `zJs`, and `T_std_NK` used to describe the simplicial complex. In Section 5 we implement the construction of the complex. We then discuss the efficient implementation of some non-trivial algorithms for the simplicial complex in Section 6 before making some conclusions at the end.

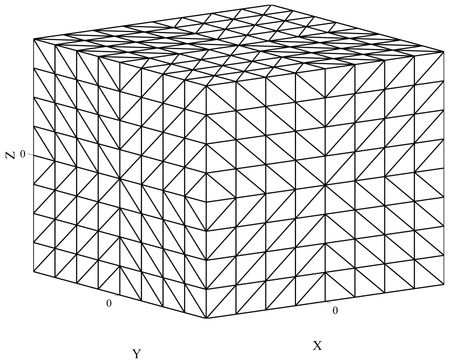

Figure 2: A schematic picture of the simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ in three dimensions. By adding the origin as a vertex to all the simplices in the simplicial 2-complex subdividing the boundary of the hypercube we get a fan-like simplicial 3-complex (tetrahedra) locally at the origin.

## 2 SIMPLICIAL COMPLEX $\mathcal{T}_{N,K}^{\mathrm{std}}$

To define the simplicial complex $\mathcal{T}_{N,K}^{\mathrm{std}}$ we first give a few definitions. We denote by $\mathbb{Z}$, $\mathbb{N}_0$, and $\mathbb{R}$ the sets of the integers, the nonnegative integers, and the 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 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 *convex combination* of an $(m+1)-$tuple $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m)$ of vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^n$ is defined by $\mathrm{co}(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m) := \{\sum_{i=0}^m \lambda_i \mathbf{x}_i : 0 \le \lambda_i \le 1, \sum_{i=0}^m \lambda_i = 1\}$. The set of vectors $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^n$ is called *affinely independent* if $\sum_{i=1}^m \lambda_i (\mathbf{x}_i - \mathbf{x}_0) = \mathbf{0}$ implies $\lambda_i = 0$ for all $i = 1, \ldots, m$. This definition is independent of the order of the vectors. If $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^n$ are affinely independent the set $\mathrm{co}(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m)$ is called an *m-simplex*.

A 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 $k$-simplex, $0 \le k < n$, and then its vertices are the common vertices of the two different $n$-simplices. 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 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 $i \notin \mathcal{J}$, the null vector $\mathbf{0} \in \mathbb{R}^n$ and the standard orthonormal basis $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ of $\mathbb{R}^n$. 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 $\mathbf{R}^{\mathcal{J}}(\mathbf{x}) := \sum_{i=1}^n (-1)^{\chi_{\mathcal{J}}(i)} x_i \mathbf{e}_i$.

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.

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

$$\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} := \mathrm{co}(\mathbf{x}_0^{\mathbf{z}\mathcal{J}\sigma}, \mathbf{x}_1^{\mathbf{z}\mathcal{J}\sigma}, \ldots, \mathbf{x}_n^{\mathbf{z}\mathcal{J}\sigma}) \tag{1}$$

where

$$\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} := \mathbf{R}^{\mathcal{J}} \left( \mathbf{z} + \sum_{j=1}^{i} \mathbf{e}_{\sigma(j)} \right) \tag{2}$$

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 $N := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{N}^m \le \mathbf{x} \le \mathbf{N}^p\}$. The simplicial complex $\mathcal{T}_N^{\mathrm{std}}$ is defined by

$$\mathcal{T}_N^{\mathrm{std}} := \{\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} : \mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \subset N\}. \tag{3}$$

3. Let $\mathbf{K}^m, \mathbf{K}^p \in \mathbb{Z}^n$, $\mathbf{N}^m \leq \mathbf{K}^m < \mathbf{0} < \mathbf{K}^p \leq \mathbf{N}^p$, 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 $K := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{K}^m \leq \mathbf{x} \leq \mathbf{K}^p\}$. 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 $K$.
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 $K^\circ$ of $K$, and all simplices in the simplicial fan $\mathcal{T}_{K,\text{fan}}^{\text{std}}$. It is thus a triangulation of $N$ having a simplicial fan in $K$.

We have several remarks on this construction. First, $\mathcal{T}_{N,K}^{\text{std}}$ is indeed a simplicial complex, as can easily be deducted from the proof of Lemma 3.6 in (Giesl and Hafstein, 2013). Second, if $\mathbf{K}^m = (-1,-1,\ldots,-1)$ and $\mathbf{K}^p = (1,1,\ldots,1)$ the complexes $\mathcal{T}_{N,K}^{\text{std}}$ and $\mathcal{T}_N^{\text{std}}$ are identical. Third, when using the complex $\mathcal{T}_{N,K}^{\text{std}}$ to compute CPA Lyapunov functions one most commonly uses a transformation $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ to deform and scale down the simplices, i.e. every simplex $\text{co}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathcal{T}_{N,K}^{\text{std}}$ is mapped to a simplex $\text{co}(\mathbf{F}(\mathbf{v}_0), \mathbf{F}(\mathbf{v}_1), \ldots, \mathbf{F}(\mathbf{v}_n))$. The transformation $\mathbf{F}$ must be chosen such that the resulting set of simplices is a simplicial complex.

## 3 VS2012 AND ARMADILLO

Before we come to our implementation of the simplicial complex $\mathcal{T}_{N,K}^{\text{std}}$ we explain how to get a project using the Armadillo linear algebra library running in VS2012 on a Windows computer. This is by no means the only nor the most elegant way, but it is very simple and it works.

First download and install Visual Studio Express 2012 for Windows Desktop. Then go to http://arma.sourceforge.net and download and extract Armadillo. Start VS2012 and choose "FILE→New Project". In the window that pops up choose "Visual C++" and "Console Application" and in the following check "Empty project". We assume for simplicity that the name given to the project is "SIMP" and that the location is "c:\". The folder where our program will be running is then "c:\SIMP\SIMP". Where armadillo was extracted, in the "include" folder, there is a file named "armadillo" and a folder named "armadillo_bits". Copy both to "c:\SIMP\SIMP". In the "examples" folder there is a folder named "lib_win32". Also copy its contents to "c:\SIMP\SIMP". Many functions in Armadillo use the LAPACK and BLAS libraries and therefore we have to uncomment (remove "//" in front of) `#define ARMA_USE_LAPACK` and `#define ARMA_USE_BLAS` in "config.hpp" in the folder "armadillo_bits" if we want to use the full functionality of Armadillo.

To actually use the functionality from LAPACK and BLAS we have to link to these libraries dynamically. To enable that choose "DEBUG→SIMPL Properties". In the window that pops up choose "Configuration Properties→Linker→Input" and add "lapack_win32_MT.lib;blas_win32_MT.lib;" (without the quotation marks) to "Additional Dependencies". Do this both with "Configuration:" on "Release" and "Debug".

VS2012 has the unexpected feature (error?) that it does not search for .dll files in the directory where the program generated is running, in our case "c:\SIMP\SIMP". To change this go to "Configuration Properties→Debugging" and add "PATH=%PATH%;$(ProjectDir)" (without the quotation marks) to "Environment". As before do this both with "Configuration:" on "Release" and "Debug".

Now everything should be ready to use Armadillo. Right-click on "Source files" in the "Solution Explorer" and choose "Add New Item". For simplicity we use the default, which is a file named "Source.cpp" in "c:\SIMP\SIMP".

To test if everything is in place we can e.g. try to compile and run the following program:

```
#include "armadillo"
#include<list>
// any other headers we might want to include
using namespace arma;
using namespace std;
int main(int argc, char **argv){
  mat A=randu<mat>(5,5);
  det(A);
}
```

For our implementation of the simplicial complex below we need to include `list`. We also use `vector` and `algorithm` from the Standard Template Library (STL), but they are already included in `armadillo`. To compile and run a console application it is advantageous to choose "DEBUG→Start Without Debugging" (or press Ctrl+F5), for otherwise the console closes immediately when the program has finished running. This procedure above has been tested to work with Armadillo 3.8.0.

Now a few comments on Armadillo: Very good documentation on the library is available at http://arma.sourceforge.net and in (Sanderson, 2010). The vector and matrix types we will use in this paper are `ivec`, `vec`, and `mat`, which are column vector of `int`, column vector of `double`, and matrix of `double` respectively. Armadillo starts indexing of vectors and matrices at zero and not at one, just as in C and C++. Note that Armadillo does not support implicit or explicit conversions between vector and matrix types only because they might make sense mathematically. If e.g. `f` is a function expecting a `vec` as an argument we cannot call it with an `ivec vi`. We have to use `conv_to<vec>::from(vi)` to explicitly convert `vi` to a `vec`.

The compiler expects the result of a matrix multiplication to be a matrix. If we know that it is a scalar ($1 \times 1$ matrix) the function `as_scalar` can be used, e.g. `double y=as_scalar(x.t()*x);` for a vector `x`. In debug modus `as_scalar` will report an error if the argument is not a $1 \times 1$ matrix, in release modus it will simply give incorrect results. Using the "<<" operator is a short and readable way to assign values to vectors and matrices (`endr` stands for end row). It, however, does not work like `push_back` in the STL. Thus `x<<1<<2;` makes `x`$= (1,2)^T$. But if this is followed by `x<<3<<4;` then `x`$= (3,4)^T$ and **not** `x`$= (1,2,3,4)^T$

Lambda functions in C++11, functions that can be written within other functions and have access to their data, are a very nice addition to C++, but there are some pitfalls when using Armadillo. It is safer to specify the return value of a lambda function, for e.g. `[](vec v) {return 1*v; }` otherwise returns an object of type `const eOp<vec,eop_scalar_times>`, but `[](vec v)->vec {return 1*v; }` returns a `vec`. A further nice addition in C++11 are auto types. Thus, if the compiler can determine the type of an entity at compile time, it will assign that type to the entity if it is declared `auto`.

For vectors `x,y` of type `vec` or `ivec` comparisons like `x > y` return a vector with 1 in the entities where the inequality holds true and 0 otherwise. Thus $(1,2,3)^T < (2,2,4)^T$ results in the vector $(1,0,1)^T$. If we want a simple true or false answer to whether the inequality is true for all components we can e.g. use `min(x-y) > 0` .

With the compiler set to "debug" operations in Armadillo are orders of magnitude slower than with the compiler set to "release".

## 4 THE DATA STRUCTURES

We use the data structures `Grid`, `zJs`, and `T_std_NK` to implement the simplicial complex $\mathcal{T}^{\text{std}}_{N,K}$. `Grid` is used to enumerate the vectors in $\{\mathbf{z} \in \mathbb{Z}^n : \mathbf{z} \in N\}$ and other similar grids, `zJs` is a simple container for $\mathbf{z} \in \mathbb{N}_0^n$, $\mathcal{J} \subset \{1,2,\dots,n\}$, and $\sigma \in S_n$ and is used to locate the simplex $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ in the data structure `T_std_NK`, which contains all necessary information on the complex $\mathcal{T}^{\text{std}}_{N,K}$. For simplicity and to shorten the code we use very short variable names and do not care about data encapsulation at all. Further, we omit declaring functions const and using const references and unsigned types, even where it would be more natural and efficient. We make heavy use of the STL in C++ and assume that the dimension, i.e. $n$ in $\mathbb{R}^n$, is already defined by e.g. `int n=4;` .

The data structure `Grid`, initialized with `ivec Nm` and `ivec pN` contains all the vertices in $\mathbf{G}(\mathtt{Nm},\mathtt{Np}) := \{\mathbf{z} \in \mathbb{Z}^n : \mathtt{Nm} \leq \mathbf{z} \leq \mathtt{Np}\}$. It assigns a unique integer to each of these vertices and can calculate the corresponding vertex from this number and vise versa. It is defined as follows:

```
struct Grid {
  ivec mN, pN;
  int EndI;
  int V2I(ivec);
  ivec I2V(int);
  vector<int> V2I(vector<ivec> v);
  bool InGrid(ivec v);
  Grid(ivec _mN,ivec _pN);
  ~Grid() {};
};
```

The numbers assigned to the vectors are $0,1,\dots,N$, where $N = \mathbf{EndI} - 1$. Thus, the constructor can be coded

```
Grid::Grid(ivec _Nm,ivec _Np):Nm(_Nm),Np(_Np){
  EndI=1;
  for(int i=0;i<n;i++){EndI *= Np(i)-Nm(i)+1;}
}
```

A simple method to assign unique numbers to the vertices is to translate them with the vector `-Nm` and then enumerate them starting at the origin. The reverse process is done by using repeated division with reminder. The following implementation of the pair `V2I` (vertex to index) and `I2V` (index to vertex) should illuminate the strategy:

```
int Grid::V2I(ivec v){
  int i, Index, Mul;
  v -= Nm;
  for(i=0,Mul=1,Index=0;i<n;i++){
    Index += v(i)*Mul;
    Mul *= Np(i)-Nm(i)+1;
  }
  return Index;
}

ivec Grid::I2V(int Index){
  ivec v(n);
  for(int i=0;i<n;i++){
    v(i) = Index%(Np(i)-Nm(i)+1);
    Index /= Np(i)-Nm(i)+1;
  }
  return v += Nm;
}
```

Further, it is advantageous to be able to pass a vector of vertices to `V2I`. This is implemented by

`vector<int> Grid::V2I(vector<ivec> v)`,
where it can be seen how lambda functions can lead to efficient and readable code:

```
vector<int> Grid::V2I(vector<ivec> v){
  vector<int> iv;
  for_each(v.begin(),v.end(),
    [&](ivec &val){iv.push_back(V2I(val));}
  );
  return iv;
}
```

The `[&]` allows the lambda function access to all variables of the enclosing function by reference, in this case `iv` and `this`. Note that the call to `V2I(val)` is an abbreviation for `this->V2I(val)` and thus the `this` pointer is implicitly used. If we want the lambda function to use copies of the variables by default we should replace `[&]` by `[=]`. We could also have listed their access mode individually by `[iv&,this]`, because we need to modify `iv` in the lambda function but not `this`.

Only one more member function is needed for Grid, `bool InGrid(ivec v)`, which returns `true` if $\mathbf{v} \in \mathbf{G}(\texttt{Nm},\texttt{Np})$ and `false` otherwise. Here the Armadillo functions `min` and `max`, which deliver the minimum and maximum values of a vector respectively, are useful:

```
  bool Grid::InGrid(ivec v){
    return min(v-Nm) >= 0 && max(v-Np) <= 0;
  }
```

We now come to the structure `zJs`. It is a simple container, on which we define an ordering relation "<". The ordering is used by `T_std_NK` to sort and then find simplices referred to by $\mathbf{z} \in \mathbb{N}_0$, $\mathcal{J} \subset \{1,2,\dots,n\}$, and $\sigma \in S_n$ quickly. The variable `int Pos` in `zJs` is the positioning used by `T_std_NK`.

```
struct zJs {
  int J,Pos;
  ivec z,sig;
  zJs(ivec _z,int _J,ivec _sig,int _Pos=-1):
     z(_z), J(_J), sig(_sig), Pos(_Pos) {};
};
```

The set $\mathcal{J} \subset \{1,2,\dots,n\}$ is stored as an integer `J`. The idea is to use the representation of `J` as a binary number to mark which elements of $\{1,2,\dots,n\}$ are in $\mathcal{J}$ and which are not. This is best shown by examples. The number $0 = (00\dots0000)_2$ is the empty set, $1 = (0\dots0001)_2$ is the set $\{1\}$, $2 = (0\dots0010)_2$ is the set $\{2\}$, $3 = (0\dots0011)_2$ is the set $\{2,1\}$, and e.g. $12 = (0\dots01010)_2$ is the set $\{4,2\}$. In general, $j \in \mathcal{J}$ if and only if the $j$-th bit in the binary representation of `J` is 1. To check whether $j \in \mathcal{J}$ one can use bit-shifts and the bitwise and-operation "&", i.e. `(J>>(j-1))&1` is one if $j \in \mathcal{J}$ and zero otherwise. For `int J` this works for $n \leq 31$, for `unsigned long long J` this works for $n \leq 64$. For $n > 64$ this strategy has to be refined.

The permutation $\sigma \in S_n$ is stored as an `ivec sigma` in its one-line notation, i.e. it is defined through `sigma[i]` $= \sigma(i)$. Here the fact that Armadillo starts indexing of vectors at zero is a little confusing, because `sigma` is a reordering of the indices. Thus `sigma[0], sigma[1], ..., sigma[n-1]` is actually a permutation of the numbers $0,1,\dots,n-1$ rather than the numbers $1,2,\dots,n$. We discuss the interplay between `J` and `sigma` in more detail in the next section, when we give the implementation of `x_zJs_i` that computes the vertices $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma}$ according to the formula (2).

The ordering relation on `zJs` is rather arbitrary, it should just order objects of type `zJs` according to `z,J,` and `sig` adequate to the STL functions `sort` and `equal_range` somehow. `Pos` should not be considered in the ordering. The following definition does the job just fine:

```
bool operator<(zJs lhs,zJs rhs){
  if(lhs.J != rhs.J) return lhs.J < rhs.J;
  int i;
  for(i=0;i<n && lhs.z(i)==rhs.z(i);i++);
  if(i!=n){return lhs.z(i)<rhs.z(i);}
  for(i=0;i<n && lhs.sig(i)==rhs.sig(i);i++);
  if(i!=n){return lhs.sig(i)<rhs.sig(i);}
  return false;  // they are equal
}
```

We come to the main structure `T_std_NK` that describes the simplicial complex $\mathcal{T}^{\text{std}}_{N,K}$. 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 ind);
  T_std_NK(ivec Nm,ivec Np,ivec Km,ivec Kp);
};
```

`Nm`$= \mathbf{N}^m$ and `Np`$= \mathbf{N}^p$ define the hypercube $N$ and `Km`$= \mathbf{K}^m$ and `Kp`$= \mathbf{K}^p$ define the hypercube $K$ from Section 2. `G` is a grid defined by `Nm` and `Np` and is used to have a coherent enumeration of all vertices possibly used by `T_std_NK`. `Ver` is a vector containing all the vertices of all the simplices in the complex and `Nr0` is the position of the zero vector/vertex in this vector, i.e. `Ver[Nr0]` is the zero-vector. `Sim` is a vector containing all the simplices of the complex. A simplex is basically $(n+1)$ vertices. Each simplex is stored as a vector of $(n+1)$-integers, the integers refereing to the positions of the corresponding vertices in `Ver`.

The remaining members are not used for the construction of the simplicial complex. They are, however, advantageous if one wants to use the simplicial complex as a basis to define CPA functions, because given a vector $\mathbf{x}$ in the triangulated hypercube $N$, they enable the fast search of a simplex $\mathfrak{S}$ such that $\mathbf{x} \in \mathfrak{S}$. `NrInSim` contains all simplices of the kind $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ in the complex sorted according to the ordering on `zJs`. `Fan` contains the rest of the simplices, i.e. the simplices in the simplicial fan at the origin. We discuss this in more detail after the next section, in which we discuss the construction of the simplicial complex `T_std_NK`.

# 5 CONSTRUCTION OF T_std_NK

To construct the simplicial complex `T_std_NK` we need a function to compute the vertices $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma}$ as in formula (2), $i = 0, 1, \ldots, n$, for the simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$. As mentioned in the last section the interplay between `J` and `sigma` here play a little confusing role. Because the set `J` is supposed to contain the indices of those coordinates of a vector $\mathbf{v}$, whose coordinates should be multiplied with minus one, and Armadillo starts indexing of vectors at zero, we should multiply the coordinate `v[j]`, which corresponds to the coordinate $v_{j+1}$ of $\mathbf{v}$, by minus one, if and only if `(J>>((j+1)-1))&1`, i.e. `(J>>j)&1`, is equal to one. Further, `sigma` is actually a permutation of the numbers $0, 1, \ldots, n-1$ as discussed above. The formula (2) for $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma}$, $i = 0, 1, \ldots, n$, can thus be implemented as follows:

```
ivec x_zJs_i(ivec z,int J,ivec sigma,int i){
  ivec x_s_i=zeros<ivec>(n), v(n);
  for(int j=0;j<i;j++){
    x_s_i(sigma(j))=1;
  }
  for(int j=0;j<n;j++){
    v(j)=((J>>j)&1 ? -1:1)*(z(j)+x_s_i(j));
  }
  return v;
}
```

We now have everything we need to actually construct `T_std_NK`. The code for the construction can be partitioned into three parts. In the first part some variables and class members are initialized. This is done in an initializer list and at the beginning of the function. In the second part we actually construct the simplicial complex. This involves a triple loop, for we have to iterate over all relevant $\mathbf{z} \in \mathbb{N}_0^n$, all $\mathcal{J} \subset \{1, 2, \ldots, n\}$, and all $\sigma \in S_n$, cf. formulas (1) and (2). In the third part we tidy up, which includes sorting some vectors to make them eligible for binary search, removing duplicates, etc. The body of the implementation for the constructor is as follows:

```
T_std_NK::T_std_NK(ivec _Nm,ivec _Np,
  ivec _Km,ivec _Kp) : Nm(_Nm), Np(_Np),
  Km(_Km),Kp(_Kp),G(_Nm,_Np) {
  // FURTHER INITIALIZATION
  int EndSet=1<<n;
  ivec ZV=zeros<ivec>(n),pQ1N(n),
    IdPerm(n),sigma(n),*pivec,z;
  int N=max(max(Np),max(-Nm));
  pQ1N.fill(N-1);
  for(int i=0;i<n;i++) IdPerm(i)=i;
  vector<ivec> sver(n+1);
  Grid Q1(ZV,pQ1N);
  Grid Ki(Km+1,Kp-1);

  // ACTUAL CONSTRUCTION OF THE COMPLEX
  for(int J=0;J<EndSet;J++){
    for(int zNr=0;zNr<Q1.EndI;zNr++){
      z=Q1.I2V(zNr);
      sigma = IdPerm;
      auto sb=sigma.begin(),se=sigma.end();
      do{
        // CODE BLOCK 1
        // ...
      }while(next_permutation(sb,se))
      ;
    }
  }

  // TIDY UP
  // CODE BLOCK 2
  // ...
}
```

We first concentrate on the initialization, the implementation of `CODE BLOCK 1` and `CODE BLOCK 2` is given below. In the initializer list we assign values to the pairs `Nm`, `Np` and `Km`, `Kp`. They correspond to the vectors $\mathbf{N}^m, \mathbf{N}^p$ and $\mathbf{K}^m, \mathbf{K}^p$ respectively, that define the hypercubes $N$ and $K$ as in Section 2. $N \setminus K^\circ$ is triangulated using the simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ and $K$ is triangulated using a simplicial fan. The grid `G(Nm,Np)` includes all vectors $\mathbf{z} \in \mathbb{Z}^n$ that might be vertices in the triangulation. `EndSet` $:= 2^n$ is chosen such that every subset $\mathcal{J}$ of $\{1, 2, \ldots, n\}$ has a unique representation as a number `J`$= 0, 1, \ldots, \texttt{EndSet} - 1$ as described above. The grid `Q1` is defined with just enough vectors $\mathbf{z} \in \mathbb{N}_0^n$ to suffice for the construction of all $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ relevant for $\mathcal{T}_N^{\text{std}}$, cf. (3). The grid `Ki` is defined such that the relevant intersections of simplices $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \subset N$ with the boundary of $K := \{\mathbf{x} \in \mathbb{R}^n : \mathbf{K}^m \leq \mathbf{x} \leq \mathbf{K}^p\}$ are characterized by having exactly one vertex in `Ki`. That we get any relevant intersection by this characterization is quite clear. The fact that we get every relevant intersection no more than once can be deduced by considering the intersection of two different such simplices, which would clearly not be an allowed intersection of two different simplices in a simplicial complex.

`IdPerm` is defined to be the permutation `IdPerm[i]=i` for $i = 0, 1, \ldots, n-1$. The function `next_permutation` from the STL considers this to be the first permutation. Successive calls to `next_permutation` then iterates through all possible permutations.

For the actual construction of the simplicial complex we iterate over all $\mathbf{z} \in$`Q1`$\subset \mathbb{N}_0^n$, all permutations `sigma` of the numbers $0,1,\ldots,n-1$, and all subsets $\mathcal{J}$ of $\{1,2,\ldots,n\}$. The $\mathbf{z}$ are represented through their unique numbers in `Q1`, the permutations are represented through their one-line form, and the subsets through numbers $0,1,\ldots,2^n-1$. The code for the actual construction is as follows:

```
// CODE BLOC 1 - IMPLEMENTATION
for(int i=0;i<=n;i++){
sver[i] = x_zJs_i(z,J,sigma,i);
}
int NrInN=0,NrInKi=0;
for_each(sver.begin(),sver.end(),
  [&] (ivec &v) {
    if(G.InGrid(v)) NrInN++;
    if(Ki.InGrid(v)){
      pivec=&v; NrInKi++;
    }
  }
);
if(NrInN == n+1){
  if(NrInKi == 0){
    Sim.push_back(G.V2I(sver));
    int SLE=Sim.end()-Sim.begin()-1;
    NrInSim.push_back(zJs(z,J,sigma,SLE));
  } else if(NrInKi == 1){
    *pivec=sver[0];
    sver[0]=ZV;
    Sim.push_back(G.V2I(sver));
    int SLE=Sim.end()-Sim.begin()-1;
    Fan.push_back(SLE);
  }
}
```

First we construct the simplex $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} = \mathrm{co}(\mathbf{x}_0^{\mathbf{z}\mathcal{J}\sigma},\mathbf{x}_1^{\mathbf{z}\mathcal{J}\sigma},\ldots,\mathbf{x}_n^{\mathbf{z}\mathcal{J}\sigma})$ by writing its vertices to `sver`, i.e. `sver[i]` $:=\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma}$ for $i = 0,1,2\ldots,n$. Then we count how many of the vertices are in $N$ and $K^\circ$. Note that $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} \in N$ if and only if `G.InGrid(sver[i]) == true` and $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} \in K^\circ$ if and only if `Ki.InGrid(sver[i]) == true`. If $\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma} \in K^\circ$ we tactically store a pointer to its corresponding `sver[i]`. Then we verify if $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \in \mathcal{T}_N^{\mathrm{std}}$, i.e. if $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma} \subset N$, which holds true if and ony if `NrInN == n+1`. Now there are two relevant cases. One is if no vertex is in $K^\circ$, i.e. `NrInKi == 0`. Then the simplex is added as is to `Sim`, however, using the unique numbers given to its vertices by `G`. We then record its position in `NrInSim` to give fast access to it later through the data structure `zJs`. If exactly one vertex, say `sver[i]` $=\mathbf{x}_i^{\mathbf{z}\mathcal{J}\sigma}$, is in $K^\circ$, i.e. `NrInKi == 1`, then we modify this simplex and add it to the simplicial fan. We first copy `sver[0]` $=\mathbf{x}_0^{\mathbf{z}\mathcal{J}\sigma}$ to `sver[i]` and then the zero vector `ZV` to `sver[0]`. Then we add it to `Sim` and record its position in `Fan`.

Now that we have constructed the simplicial complex we tidy up and prepare the simplicial complex for efficient application. This is implemented as follows:

```
// CODE BLOC 2 - IMPLEMENTATION
// record all vertices
list<int> lv;
for_each(Sim.begin(),Sim.end(),
  [&](vector<int> &v){
    for_each(v.begin(),v.end(),
      [&](int iv){
        lv.push_back(iv);
      }
    );
  }
);
lv.sort();
lv.unique();
vector<int> vID(lv.size());
vID.assign(lv.begin(),lv.end());
Nr0=equal_range(vID.begin(),vID.end(),
  G.V2I(ZV)).first - vID.begin();
// let the simplices in "Sim" refer to the
// vertices by their positions in vID rather
// than their ID-number from "Grid G"
for_each(Sim.begin(),Sim.end(),
  [&](vector<int> &v){
    for_each(v.begin(),v.end(),
      [&](int &iv){
      iv=equal_range(vID.begin(),vID.end(),
        iv).first - vID.begin();
      }
    );
  }
);
// record the vertices in "Ver" in the same
// order as in vID
for_each(vID.begin(),vID.end(),
  [&](int vID){
    Ver.push_back(G.I2V(vID));
  }
);
// sort "NrInSim" and "Fan" for binary search
sort(NrInSim.begin(),NrInSim.end());
sort(Fan.begin(),Fan.end());
```

To record all the vertices in the complex we first add all vertex ID-numbers of all simplices to the list `lv`, sort it and use `unique` to remove duplicates. Then we copy the contents of `lv` to the vector `vID` to be able to apply efficient binary search, i.e. the STL function `equal_range`. Further, we record the position `Nr0` of the zero vertex in `vID`.

Then we go through all vertices of all the simplices and replace the ID-number of every vertex by its position in `vID`. Then we actually construct the vertices as `ivec` and write them in `Ver` in the same order as in `vID`. Thus `Ver[Sim[k][i]]` is the $i$-th vertex of the $k$-th simplex in `Sim`. Finally, we sort `NrInSim` and `Fan`, again to be able to efficiently apply the STL function `equal_range`. Thus given `z,J,` and `sigma` for a simplex $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ we can efficiently locate it in `Sim` and given the position of a simplex in `Sim` we can efficiently check whether it is in the simplicial fan, i.e. in

`Fan`, or not. We take advantage of the former property in the function `int InSimplexNr(vec x)`, discussed in the next section. The second property is not important for the applications described in this paper, but is useful for other applications.

# 6 ALGORITHMS FOR T_std_NK

If the data structure `T_std_NK` is to be useful for serving as a basis for CPA functions, we have to be able to efficiently solve the following problem: For an arbitrary $\mathbf{x} \in N$ find a simplex $\mathfrak{S} \in \mathcal{T}^{\text{std}}_{N,K}$ such that $\mathbf{x} \in \mathfrak{S}$. In this section we implement this efficiently given that the simplicial fan contains a small fraction of the total number of simplices in the complex. If this is not the case a different strategy should be used, e.g. storing an appropriate `zJs` for simplices in the simplicial fan and project $\mathbf{x} \in K$ to the boundary of $K$ and search for this appropriate `zJs`. For demonstrating our ideas the following is, however, more informative, because it includes ideas necessary to solve this problem when $\mathcal{T}^{\text{std}}_{N,K}$ has been deformed as explained in Section 2.

Given a simplex $\mathfrak{S} = \text{co}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$ and a vector $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \in \mathfrak{S}$ if and only if $\mathbf{x}$ can be written as a convex combination of the vertices of the simplex. This means that there are nonnegative numbers $\lambda_0, \lambda_1, \ldots, \lambda_n$, such that

$$\mathbf{x} = \sum_{i=0}^{n} \lambda_i \mathbf{v}_i, \text{ where } \sum_{i=0}^{n} \lambda_i = 1, \tag{4}$$

which in turn is equivalent to

$$\mathbf{x} - \mathbf{v}_0 = \sum_{i=1}^{n} \lambda_i (\mathbf{v}_i - \mathbf{v}_0), \text{ where } \sum_{i=1}^{n} \lambda_i \le 1. \tag{5}$$

We construct the matrix $X$ by writing the vectors $\mathbf{v}_1 - \mathbf{v}_0, \mathbf{v}_2 - \mathbf{v}_0, \ldots, \mathbf{v}_n - \mathbf{v}_0$ in its columns consequently. Because the vertices of a simplex are affinely independent, the equation $\mathbf{x} - \mathbf{v}_0 = X\mathbf{L}$ always has a solution $\mathbf{L} = (\lambda_1, \lambda_2, \ldots, \lambda_n)^T$. If the solution fulfills $\lambda_i \ge 0$ for all $i = 1, 2, \ldots, n$ and $\sum_{i=1}^{n} \lambda_i \le 1$, then $\mathbf{x} \in \mathfrak{S}$, otherwise $\mathbf{x} \notin \mathfrak{S}$. Thus we can implement `bool T_std_NK::InSimp(vec x,int ind)`, which returns `true` if `vec x` is in the simplex `Sim[ind]` and `false` otherwise, as follows:

```
bool T_std_NK::InSimp(vec x,int ind){
  vector<vec> v(n+1);
  for(int i=0;i<=n;i++){
    ivec t=Ver[Sim[ind][i]];
    v[i]=conv_to<vec>::from(t);
  }
  mat X(n,n);
  for(int i=1;i<=n;i++){
    X.col(i-1)=v[i]-v[0];
  }
  vec L=solve(X,x-v[0]);
  return min(L) >= 0 && sum(L) <= 1;
}
```

The code is self explaining. The connection to CPA functions $f : N \to \mathbb{R}$, defined by giving its values $f(\mathbf{v})$ at every vertex $\mathbf{v}$ of every simplex of $\mathcal{T}^{\text{std}}_{N,K}$ is as follows: If $\mathbf{x} = \sum_{i=0}^{n} \lambda_i \mathbf{v}_i \in \text{co}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n)$, then

$$f(\mathbf{x}) = f(\sum_{i=0}^{n} \lambda_i \mathbf{v}_i) = \sum_{i=0}^{n} \lambda_i f(\mathbf{v}_i), \tag{6}$$

as can be easily verified. Because $\lambda_0 = 1 - \sum_{i=1}^{n} \lambda_i$ the solution $\mathbf{L}$ thus gives us a formula for the function value.

Going through all simplices $\mathfrak{S} \in \mathcal{T}^{\text{std}}_{N,K}$ to check whether a given $\mathbf{x} \in \mathfrak{S}$ is not very efficient and if $\mathbf{x} \in N \setminus K^\circ$ we can do much better. In this case we know that $\mathbf{x} \in \mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ for some $\mathbf{z} \in \mathbb{N}_0^n$, $\mathcal{J} \subset \{1, 2, \ldots, n\}$, and $\sigma \in S_n$, and if we calculate $\mathbf{z}$, $\mathcal{J}$, and $\sigma$ directly from $\mathbf{x}$ we can find the simplex $\mathfrak{S}_{\mathbf{z}\mathcal{J}\sigma}$ using binary search in the vector `NrInSim`. To compute $\mathbf{z}$ and $\mathcal{J}$ we first construct a vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)^T$ and an integer $J$. We do this by going through the entities $x_i$ of $\mathbf{x}$. If $x_i \ge 0$ we set $y_i = x_i$ and the $i$-th bit of the binary representation of $J$ equal to zero. If $x_i < 0$ we set $y_i = -x_i$ and the $i$-th bit in the binary representation of $J$ equal to one. After this procedure the integer $J$ characterizes the set $\mathcal{J}$ as discussed in Section 4 and $\mathbf{z} = (z_1, z_2, \ldots, z_n)^T$ can be computed by $z_i = \lfloor y_i \rfloor$ (largest integer $\le y_i$) for $i = 1, 2, \ldots, n$. Now clearly $\mathbf{y} \in \mathfrak{S}_{\mathbf{z}\emptyset\sigma}$, i.e. $\mathbf{y} \in \mathfrak{S}_{\mathbf{z}\mathcal{J}^*\sigma}$ with $\mathcal{J}^* = \emptyset$ the empty set, and it is easily verified that

$$\mathbf{w} := \mathbf{y} - \mathbf{z} = \sum_{i=1}^{n} \lambda_i \sum_{j=1}^{i} \mathbf{e}_{\sigma(j)} = \sum_{i=1}^{n} \left( \sum_{j=i}^{n} \lambda_j \right) \mathbf{e}_{\sigma(i)}. \tag{7}$$

Because all the $\lambda_j$ are nonnegative we have the relation

$$1 \ge w_{\sigma(1)} \ge w_{\sigma(2)} \ge \cdots \ge w_{\sigma(n)} \ge 0 \tag{8}$$

for $\mathbf{w} = (w_1, w_2, \ldots, w_n)^T$. Thus, if we sort the entities of $\mathbf{w}$ in decreasing order and record the corresponding indices we have computed $\sigma$. The function `sort_index(w,1)` in Armadillo does exactly this (the optional argument `1` specifies descending sort).

In the implementation of `int T_std_NK::InSimpNr(vec x)`, that returns the position in `Sim` of a simplex containing `vec x` if possible and $-1$ otherwise, we first check if `x` is in the domain of the simplicial complex. Then we check if `x` is (only) in the domain of the simplicial fan. If it is we go through all simplices in the fan to find a simplex containing `x`. If `x` is in the domain of the simplicial complex, but not in the fan, we use the efficient strategy of computing a simplex containing it as described above. The code has, obviously, to be adapted to the fact that Armadillo indexes vectors from zero. The implementation is as follows:

```
int T_std_NK::InSimpNr(vec x){
  if(!(min(Np-x)>0 && min(x-Nm)>0)){
    return -1;
  }
  if(min(Kp-x)>0 && min(x-Km)>0){
    for(int i=0;i<Fan.size();i++){
      if(InSimp(x,Fan[i])){
        return Fan[i];
      }
    }
  }
  // WE CAN COMPUTE THE POSITION OF THE SIMPLEX
  int J=0;
  ivec z(n),sig; // sig=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));
  }
  sig=conv_to<ivec>::from(sort_index(x-z,1));
  return equal_range(NrInSim.begin(),
      NrInSim.end(),zJs(z,J,sig)).first->Pos;
}
```

We have a few comments on this implementation. The command `J |= 1<<i;` sets the $(i+1)$-th bit of the binary representation of $J$ equal to 1. Because the entities of `x` are all nonnegative when we want to compute their floor, we can simply cast from `double` to `int`. The Armadillo function `sort_index` returns a vector of unsigned integers that describes the sorted order of the given vector's elements. The optional second parameter can be set to 1 to let `sort_index` use descending sort, otherwise it uses the default, which is ascending sort.

# 7 SUMMARY

We described the implementation of a simplicial complex with a simplicial fan at the origin. Such complexes allow for the parameterizations of continuous, piecewise affine (CPA) functions, with an arbitrary rich structure at a singularity. Such CPA functions have been shown to be irreplaceable in the computation of true CPA Lyapunov functions for general nonlinear systems. We used C++11 and the Armadillo linear algebra library for the implementation and we discussed some of the advantages of doing so in the paper. Thus, the paper might be of interest to scientists and engineers interested in modern numerical programming in C++11 under Windows, even if they are not necessarily interested in our particular problem of implementing simplicial complexes for CPA functions.

# REFERENCES

Eghbal, N., Pariz, N., and Karimpour, A. (2012). Discontinuous piecewise quadratic Lyapunov functions for planar piecewise affine systems. *J. Math. Anal. Appl.*, 399, pp. 586–593.

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

Giesl, P. and Hafstein, S. (2012). Existence of piecewise linear Lyapunov functions in arbitary dimensions, *Discrete Contin. Dyn. Syst.*, 32, pp. 3539–3565.

Giesl, P. and Hafstein, S. (2013). Revised CPA method to compute Lyapunov functions for nonlinear systems. *(submitted)*

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

Hafstein, S. (2005). A constructive converse Lyapunov theorem on asymptotic stability for nonlinear autonomous ordinary differential equations. *Dynamical Systems*, 20, pp. 281–299

Johansen, T. (2000). Computation of Lyapunov Functions for Smooth Nonlinear Systems using Convex Optimization. *Automatica*, 36, pp. 1617–1626.

Johansson, M. and Rantzer, A. (1997). On the computation of piecewise quadratic Lyapunov functions. In: *Proceedings of the 36th IEEE Conference on Decision and Control.*

Julian, P., Guivant, J., and Desages, A. (1999). A parametrization of piecewise linear Lyapunov function via linear programming *Int. Journal of Control*, 72, pp. 702–715.

Marinosson, S. (2002a). Lyapunov function construction for ordinary differential equations with linear programming. *Dynamical Systems*, 17, pp. 137–150.

Marinosson, S. (2002b). *Stability analysis of nonlinear systems with linear programming: A Lyapunov functions based approach*. Ph.D. Thesis: Gerhard-Mercator-University, Duisburg, Germany.

Peet, M. and Papachristodoulou, A. (2010). A converse sum-of-squares Lyapunov result: An existence proof based on the Picard iteration. In: *49th IEEE Conference on Decision and Control*, pp. 5949–5954.

Rezaiee-Pajand, M. and Moghaddasie, B. (2012). Estimating the Region of Attraction via collocation for autonomous nonlinear systems. *Struct. Eng. Mech.*, 41-2, pp. 263–284.

Sanderson, C. (2010). *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report, NICTA.