Efficient Implementation of Piecewise Quadratic Lyapunov Function Computations for Switched Linear Systems*

Stefania Andersen[1][a], Sigurdur Hafstein[1][b], Juan Javier Palacios Roman[2][c]
and Sebastiaan J.A.M. van den Eijnden[2][d]

[1]*Faculty of Physical Sciences, University of Iceland, Dunhagi 5, 107 Reykjavik, Iceland*

[2]*Eindhoven University of Technology, Department of Mechanical Engineering, Groene Loper 3, 5612 AE Eindhoven, Netherlands*

Keywords: Common Lyapunov Function, Switched Linear Systems, Linear Programming, C++, MATLAB.

Abstract: We describe a linear programming (LP) problem to parameterize continuous and piecewise quadratic (CPQ) Lyapunov functions for switched linear systems. We discuss some algorithms and data-structures for its implementation in C++ and compare the computational efficiency of our implementation to an analogous implementation in MATLAB.

1 INTRODUCTION

Switched systems play an important role in modelling in science and engineering (Davrazos and Koussoulas, 2001; Liberzon, 2003; Shorten et al., 2007; Sun and Ge, 2011). In control theory the stability of an equilibrium point is commonly of central importance and is most conveniently dealt with using the Lyapunov stability theory of dynamical systems (Hahn, 1967; Sastry, 1999; Khalil, 2002; Vidyasagar, 2002). It is well known that the existence of a Lyapunov function for a switched system, which is a common Lyapunov function for all the subsystems, is equivalent to its stability and suitable classes of potential Lyapunov functions is a thoroughly researched subject (Dayawansa and Martin, 1999; Goebel et al., 2006; Mason et al., 2006; Shorten et al., 2007; Mason et al., 2022). In this paper we consider the algorithm from (Palacios Roman et al., 2024) to parameterize continuous and piecewise quadratic (CPQ) Lyapunov functions for switched linear systems along with its efficient implementation in the programming language C++. The class of CPQ Lyapunov functions have been considered in (Johansson and Rantzer, 1998), and have recently been successfully used to study the stability of, hybrid integrator-gain systems (HIGS), see e.g. (van den Eijnden et al., 2020; Deenen et al., 2021; van den Eijnden et al., 2022).

The main contribution of this work is the efficient computation of CPQ Lyapunov functions for switched linear systems based on a linear program, where the classical constraints for CPQ Lyapunov functions, which are often expressed in terms of linear matrix inequalities, are formulated as a linear programming (LP) problem.

This paper is organized as follows: In Section 2 we describe how we write the LP problem in our implementation, before we discuss in Section 3 triangulations, CPQ functions, and how we parameterize them. Section 4 deals with the linear constraints we use to compute a suitable parameterization for a CPQ Lyapunov function for a given switched linear system. Then, in Section 5, we give some details on the implementation of the linear constraints and compare the numerical efficiency of our implementation in C++ to a corresponding implementation in MATLAB. We conclude the paper in Section 6.

2 WRITING OUR LP PROBLEM

An LP feasibility problem can be characterized using a matrix $A \in \mathbb{R}^{r\times c}$, a vector $\mathbf{b} \in \mathbb{R}^r$, and an `enum` vector $\mathbf{s}$, $s_i \in$ {`LE`,`GE`,`EQ`}, having r elements. A feasible

[a] https://orcid.org/0000-0001-6747-775X
[b] https://orcid.org/0000-0003-0073-2765
[c] https://orcid.org/0009-0004-3145-1119
[d] https://orcid.org/0000-0003-1273-109X

*This work was supported in part by the Icelandic Research Fund under Grant 228725-051.

Andersen, S., Hafstein, S., Roman, J. and van den Eijnden, S.
Efficient Implementation of Piecewise Quadratic Lyapunov Function Computations for Switched Linear Systems.
DOI: 10.5220/0012927500003822
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 21st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2024) - Volume 1*, pages 277-284
ISBN: 978-989-758-717-7; ISSN: 2184-2809
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

solution to the LP problem is a vector $\mathbf{x} \in \mathbb{R}^c$ that satisfies

$$[A\mathbf{x}]_i \;\; [s_i] \;\; b_i, \quad \text{for all } i = 1, 2, \ldots, r,$$

where $[A\mathbf{x}]_i$ is the i-th component of the vector $A\mathbf{x}$. This means that

$$\begin{aligned} & [A\mathbf{x}]_i \leq b_i, \;\; \text{if } s_i \text{ is } \texttt{LE}, \\ & [A\mathbf{x}]_i = b_i, \;\; \text{if } s_i \text{ is } \texttt{EQ}, \\ \text{and} \quad & [A\mathbf{x}]_i \geq b_i, \;\; \text{if } s_i \text{ is } \texttt{GE}. \end{aligned}$$

Since the matrix A is usually sparse, we implement it using two vectors, `rn` and `cn`, of row and column indices, respectively, and one vector of values `val`. The vector $\mathbf{b}$ is implemented by the vector `b` and the vector of symbols $\mathbf{s}$ by the `enum TypeCon {LE,GE,EQ}` vector `con`.

The vectors `rn`, `cn` and `val` will all have the same number of elements, namely the number of non-zero elements of A, and to set the (i, j)-th element of A to some non-zero value `x`, i.e. $a_{i,j}$:=`x`, we write

```
rn[k]=i; cn[k]=j; val[k]=x;
```

A constraint of the LP problem is written in a row of the matrix A. A column of A corresponds to a variable. More exactly, the columns of A contain the coefficients with which the corresponing variable in the vector `Variables` appears in the constrains of the LP problem. The `Variables` vector is described below.

2.1 Storing the Variables

To implement a variable of the LP problem we use an Armadillo (Sanderson, 2010) integer vector `ivec`. For example, the variable $\varphi^{\nu}_{k,\ell}$ is `ivec {'P',nu,k,l}`, where `nu`, `k` and `l` are some numbers of type `bint` (big integer, i.e. `long long`). All the variables of the problem are stored together in the sorted vector `Variables`.

In order to sort the `Variables` vector we need to define an order relation. We first let the variable's length dictate the relation. If the variables have the same length we compare the first non-equal elements or last elements, which ever comes first. This is implemented with the following binary function:

```
bool varCmp(const ivec &x, const ...
    ivec &y) {
  if (x.size() < y.size()) {
    return true;
  }
  else if (x.size() > y.size()) {
    return false;
  }
  int i, len = x.size();
  for(i=0; i < len-1 && x(i) == ...
    y(i); i++) {};
  return x(i) < y(i);
}
```

We also implement a function, `VarID`, that obtains the index of a variable in `Variables`. That is, if `var=Variables(i)` then `VarID(var)` returns `i`. If the variable is not in `Variables` we return the impossible value -1. The function is as follows:

```
bint VarID(const ivec &v){
  auto found = ...
    equal_range(Variables.begin(), ...
    Variables.end(), v, varCmp);
  if( found.first == found.second ){
    return -1;
  }
  else{
    return found.first - ...
    Variables.begin();
  }
}
```

2.2 Construction of the Matrix A

Since each row of the matrix A corresponds to a constraint, it is preferable to write this matrix in a row-by-row manner. We implement the function, `new_aij`, that writes a new nonzero element of A or modifies a nonzero value, and a function, `close_constr`, that "closes" the constraint, such that the next call to `new_aij` will write an element in the next row. We store the current index of `rn`, `cn` and `val` in the `bint` variable `Index`, and the current row number in the `int` variable `ConstrNr`. The corresponding functions are as follows:

```
void new_aij(const ivec ...
    &variable, double value){
  bint VID = VarID(variable);
  assert(VID !=-1);
  for (bint i = Index - 1; i >= 0 ...
    && rn[i] == ConstrNr; i--) {
    if (cn[i] == VID) {
      val[i] += value;
      return;
    }
    rn.push_back(ConstrNr);
    cn.push_back(VID);
    val.push_back(value);
    Index++;
  }
}
```

```
void close_constr(TypeCon Type, ...
    double _b){
  con.push_back(Type);
  b.push_back(_b);
  ConstrNr++;
}
```

2.3 Minimal LP Example

As a minimal example for our format, consider the constraints

$$\begin{aligned} 3x - y &\leq 5 \\ x + 2y &= 7. \end{aligned} \tag{1}$$

We need the following variables:

```
using bint = long long;
using namespace std;
using namespace arma;
bint Index;
int ConstrNr;
vector<ivec> Variables
vector<bint> rn, cn;
vector<double> val, b;
enum TypeCon {LE, EQ, GE};
vector<TypeCon> con;
```

The constraints can be implemented as:

```
vector<ivec> Variables={ ...
    ivec{'x'}, ivec{'y'}};
rn[0]=0; cn[0]=0; val[0]=3;
rn[1]=0; cn[1]=1; val[1]=-1;
con[0]=LE; b[0]=5;
rn[2]=1; cn[2]=0; val[2]=1;
rn[3]=1; cn[3]=1; val[3]=2;
con[1]=EQ; b[1]=7;
```

or alternatively with `new_aij` and `close_constr` as:

```
vector<ivec> Variables={ ...
    ivec{'x'}, ivec{'y'}};
new_aij(ivec {'x'},3);
new_aij(ivec {'y'},-1);
close_constr(LE,5);
new_aij(ivec {'x'},1);
new_aij(ivec {'y'},2);
close_constr(EQ,7);
```

Note that `'x'` and `'y'` are interpreted as their ASCII `dec` equivalents, that is 120 and 121 respectively.

The purpose of the structure discussed above is to act as a parser between the problem formulation and the linear solver we use to solve the LP problems, Gurobi (Gurobi Optimization, LLC, 2023). The same example using the Gurobi interface for C would be given by

```
int vars    = 2;
int constrs = 2;
size_t vbeg[]  = {0, 2};
int vlen[]  = {2, 2};
int vind[]  = {0, 1, 0, 1};
double vval[] = {3.0, 1.0, -1.0, ...
    2.0};
char sense[] = ...
    {GRB_LESS_EQUAL,GRB_EQUAL};
double rhs[]   = {5.0, 7.0};
error = GRBXloadmodel(env, ...
    &model, "example", vars, ...
    constrs, 1, 0.0, NULL, sense, ...
    rhs, vbeg, vlen, vind, vval, ...
    NULL, NULL, NULL, NULL, NULL);
```

The syntax of this interface has two main drawbacks; it is rather opaque and one can easily overwrite a previously declared nonzero element of the matrix A accidentally. If, for example, we obtained the $3x$ in (1) via a sum $(x+2x)$ and wrote

```
int vlen[]={3, 2};
int vind[]={0, 0, 1, 0, 1};
double vval[]={1.0, 2.0, 1.0, ...
    -1.0, 2.0};
```

we would overwrite the first instance $1.0 \cdot x$ with $2.0 \cdot x$ rather than adding them together. A parser is easily implemented:

```
// var_size = Variables.size()
// rncnval_size = rn.size()
for(int v=0,j=0;v < var_size;v++){
  vbeg[v]=j;
  for(int k=0;k < rncnval_size;k++){
    if(cn[k] == v){
      vval[j]=val[k];
      vind[j]=rn[k];
      j++;
    }
  }
}
```

3 CPQ FUNCTIONS

In order to parameterize a CPQ function, we use a triangulation of the domain of the function. We will first discuss the specifics of a suitable triangulation $\mathcal{T}$. Then, a prelude on the parameterization of continuous and piecewise affine (CPA) functions will be given, from which the parameterization of CPQ functions using the triangulation $\mathcal{T}$ will follow naturally. Note that our parameterization of CPQ Lyapunov functions largely follows (Johansson, 1999).

3.1 Triangulation

A triangulation $\mathcal{T}$ with vertices $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p\} \subset \mathbb{R}^n$ is a subdivision of a subset of $\mathbb{R}^n$ into simplices $\mathfrak{S}$. A simplex $\mathfrak{S}_\nu$ is defined as

$$\begin{aligned} \mathfrak{S}_\nu &:= \operatorname{co}\{\mathbf{x}_0^\nu, \mathbf{x}_1^\nu, \ldots, \mathbf{x}_n^\nu\} \\ &= \left\{ \mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=0}^{n} \lambda_i \mathbf{x}_i^\nu, \lambda_i \geq 0, \sum_{i=0}^{n} \lambda_i = 1 \right\}, \end{aligned}$$

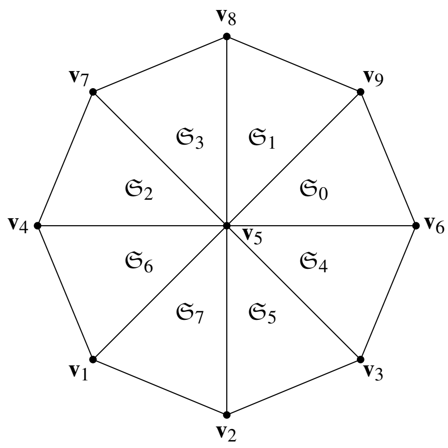

Figure 1: Triangulation $\mathcal{T}_1$ with simplices and vertices indexed.

where $\mathbf{x}_i^\nu = \mathbf{v}_{g_\nu(i)}$ for $i = 0,1,\ldots,n$ and $g_\nu : \{0,1,\ldots,n\} \to \{1,2,\ldots,p\}$ is an index-function. For our purposes, we require the triangulation $\mathcal{T}$ to be shape-regular, i.e., every two different simplices $\mathfrak{S}_\nu, \mathfrak{S}_\mu \in \mathcal{T}$ either intersect in a common lower-dimensional face or do not intersect at all. Further, we demand that $\mathbf{x}_0^\nu = \mathbf{0}$ and that the vectors $\mathbf{x}_1^\nu, \mathbf{x}_2^\nu, \ldots, \mathbf{x}_n^\nu$ are linearly independent for all $\mathfrak{S}_\nu \in \mathcal{T}$. Finally, the set theoretic union of all $\mathfrak{S}_\nu \in \mathcal{T}$, denoted $\mathcal{D}_\mathcal{T}$, must be a neighbourhood of the origin of $\mathbb{R}^n$.

An efficient implementation of a triangulation that satisfies these requirements is the triangular fan $(\mathcal{T}_{K,\text{fan}}^{\text{std}})^{\mathbf{F}}$ discussed in (Hafstein, 2019), from here on denoted simply by $\mathcal{T}_K$, where a formula for all $\mathbf{x}_i^\nu$ is given. The vertices $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p\}$ of the triangulation $\mathcal{T}_K$ are

$$\{\mathbf{0}\} \cup \left\{ \frac{K}{\|\mathbf{z}\|_2}\mathbf{z} : \mathbf{z} \in \mathbb{Z}^n, \|\mathbf{z}\|_\infty = K \right\},$$

where the scaling parameter $K \in \mathbb{N}$ determines the fineness of the triangulation around zero. The number of simplices in the triangulation $\mathcal{T}_K$ is given by the formula $2^n \cdot K^{n-1} \cdot n!$. The triangulation $\mathcal{T}_1$ in $\mathbb{R}^2$ is depicted in Figure 1.

3.2 Prelude on CPA Functions

To parameterize CPA functions on the whole space $\mathbb{R}^n$, we associate a cone $\mathfrak{C}_\nu$ to each simplex $\mathfrak{S}_\nu \in \mathcal{T}$, defined as

$$\begin{aligned} \mathfrak{C}_\nu &:= \text{cone}\{\mathbf{x}_1^\nu, \mathbf{x}_2^\nu, \ldots, \mathbf{x}_n^\nu\} \\ &= \left\{ \mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^n \lambda_i \mathbf{x}_i^\nu, \lambda_i \geq 0 \right\}. \end{aligned}$$

We refer to the vector $\boldsymbol{\lambda} = (\lambda_i)_{i=1,2,\ldots,n} \in \mathbb{R}_+^n$ as the *cone coordinates* of $\mathbf{x}$ in $\mathfrak{C}_\nu$. Additionally, we define the matrix $X_\nu := [\mathbf{x}_1^\nu, \ldots, \mathbf{x}_n^\nu] \in \mathbb{R}^{n \times n}$. Note that for this definition we must assume that the order of the vertices of each $\mathfrak{S}_\nu$ is fixed.

For a CPA function $W: \mathbb{R}^n \to \mathbb{R}$ defined using the triangulation $\mathcal{T}$ we have for every $\mathbf{x} \in \mathfrak{C}_\nu$ that $W(\mathbf{x}) = \mathbf{w}_\nu^T \mathbf{x}$, where

$$\mathbf{w}_\nu^T = \Big[W(\mathbf{x}_1^\nu), W(\mathbf{x}_2^\nu), \ldots, W(\mathbf{x}_n^\nu)\Big] X_\nu^{-1}.$$

Just note that for all $\mathbf{x} \in \mathfrak{C}_\nu$ we have $\mathbf{x} = X_\nu \boldsymbol{\lambda}$ and thus

$$\begin{aligned} W(\mathbf{x}) &= \mathbf{w}_\nu^T \mathbf{x} \qquad (2) \\ &= \Big[W(\mathbf{x}_1^\nu), W(\mathbf{x}_2^\nu), \ldots, W(\mathbf{x}_n^\nu)\Big] X_\nu^{-1} X_\nu \boldsymbol{\lambda} \\ &= \sum_{i=1}^n \lambda_i W(\mathbf{x}_i^\nu). \end{aligned}$$

To implement the computation of W using LP, it is advantageous to define the vector of variables $\Phi \in \mathbb{R}^p$, where $\varphi_j = W(\mathbf{v}_j)$, $j = 1,2,\ldots,p$. Then $W(\mathbf{x}_i^\nu) = \varphi_{g_\nu(i)}$ for every vertex $\mathbf{x}_i^\nu = \mathbf{v}_j$, $i = 0,1,\ldots,n$, of some $\mathfrak{S}_\nu \in \mathcal{T}$. Next, we define the vector

$$\Psi^\nu := \left(\varphi_{g_\nu(1)}, \varphi_{g_\nu(2)}, \cdots, \varphi_{g_\nu(n)}\right)^T$$

for every $\mathfrak{S}_\nu \in \mathcal{T}$. Then

$$W(\mathbf{x}) = (\Psi^\nu)^T \boldsymbol{\lambda}$$

by (2), where $\boldsymbol{\lambda} \in \mathbb{R}_+^n$ are the cone coordinates of $\mathbf{x} \in \mathfrak{C}_\nu$.

That W is well-defined and continuous now follows easily from the fact that the vertices $\mathbf{x}_1^\nu, \ldots, \mathbf{x}_n^\nu$ of every $\mathfrak{S}_\nu \in \mathcal{T}$ are linearly independent and that $\mathcal{T}$ is shape-regular: Since $\mathcal{T}$ is shape-regular, $\mathfrak{C}_\nu \cap \mathfrak{C}_\mu = \text{cone}\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_j\}$, where $j < n$ and the $\mathbf{y}_i$ are the common nonzero vertices of $\mathfrak{S}_\nu$ and $\mathfrak{S}_\mu$, i.e., $\mathbf{y}_i = \mathbf{x}_{k_i}^\nu = \mathbf{x}_{\ell_i}^\mu$ for some $k_i, \ell_i \in \{1,\ldots,n\}$ and $g_\nu(k_i) = g_\mu(\ell_i)$ for all $i = 1,\ldots,j$. Then, for all $\mathbf{x} \in \mathfrak{C}_\nu \cap \mathfrak{C}_\mu$

$$\mathbf{x} = \sum_{i=1}^j \lambda_i \mathbf{y}_i = \sum_{i=1}^j \lambda_i \mathbf{x}_{k_i}^\nu = \sum_{i=1}^j \lambda_i \mathbf{x}_{\ell_i}^\mu,$$

for some unique $\boldsymbol{\lambda}$, because of the linear independence of the vertices. Hence

$$(\Psi^\nu)^T \boldsymbol{\lambda} = \sum_{i=1}^j \lambda_i \varphi_{g_\nu(k_i)} = \sum_{i=1}^j \lambda_i \varphi_{g_\mu(\ell_i)} = (\Psi^\mu)^T \boldsymbol{\lambda}$$

and W is well-defined on $\mathfrak{C}_\nu \cap \mathfrak{C}_\mu$ and continuous, since it is the restriction of the continuous functions $\mathbf{x} \mapsto \mathbf{w}_\nu^T \mathbf{x}$ and $\mathbf{x} \mapsto \mathbf{w}_\mu^T \mathbf{x}$ to $\mathfrak{C}_\nu \cap \mathfrak{C}_\mu$. Hence, the vector $\boldsymbol{\varphi} \in \mathbb{R}^p$ and the functions g_ν connect the different formulas $W(\mathbf{x}) = \mathbf{w}_\nu^T \mathbf{x}$, $\mathfrak{S}_\nu \in \mathcal{T}$, such that the resulting function

$$W(\mathbf{x}) = \mathbf{w}_\nu^T \mathbf{x} \quad \text{if } \mathbf{x} \in \mathfrak{C}_\nu$$

is well-defined and continuous.

3.3 Representation of CPQ Functions

For representing a CPQ function $V\colon \mathbb{R}^n \to \mathbb{R}$, i.e., V is continuous and $V(\mathbf{x}) = \mathbf{x}^T P^\nu \mathbf{x}$ for a symmetric matrix $P^\nu \in \mathbb{R}^{n\times n}$ on each $\mathfrak{S}_\nu \in \mathcal{T}$, we can proceed similarly as in the CPA case. However, now we need a $p \times p$ matrix $\Phi \in \mathbb{R}^{p\times p}$ of variables, such that for a simplex $\mathfrak{S}_\nu \in \mathcal{T}$ we have

$$P^\nu = (X_\nu^{-1})^T \Psi^\nu X_\nu^{-1},$$

where the (k,ℓ)-th entry of the matrix $\Psi^\nu \in \mathbb{R}^{n\times n}$ is equal to $\varphi_{g_\nu(k),g_\nu(\ell)}$, which is the $(g_\nu(k),g_\nu(\ell))$-th entry of Φ.

Then, for an $\mathbf{x} = \sum_{i=1}^n \lambda_i \mathbf{x}_i^\nu \in \mathfrak{C}_\nu$, i.e., $\mathbf{x} = X^\nu \boldsymbol{\lambda}$ with $\boldsymbol{\lambda} \in \mathbb{R}_+^n$, we have

$$\begin{aligned} V(\mathbf{x}) &= \mathbf{x}^T P^\nu \mathbf{x} \\ &= \boldsymbol{\lambda}^T X_\nu^T (X_\nu^{-1})^T \Psi^\nu X_\nu^{-1} X_\nu \boldsymbol{\lambda} \\ &= \boldsymbol{\lambda}^T \Psi^\nu \boldsymbol{\lambda} \qquad (3) \\ &= \sum_{k,\ell=1}^n \lambda_k \lambda_\ell \varphi_{g_\nu(k),g_\nu(\ell)}. \end{aligned}$$

That V is well-defined and continuous follows similarly as for the CPA case: For all $\mathbf{x} \in \mathfrak{C}_\nu \cap \mathfrak{C}_\mu$

$$\mathbf{x} = \sum_{i=1}^j \lambda_i \mathbf{y}_i = \sum_{i=1}^j \lambda_i \mathbf{x}_{k_i}^\nu = \sum_{i=1}^j \lambda_i \mathbf{x}_{\ell_i}^\mu,$$

for some $k_i, \ell_i \in \{1,\dots,n\}$ and a unique $\boldsymbol{\lambda}$. Since

$$\begin{aligned} \mathbf{x}^T P^\nu \mathbf{x} &= \sum_{r=1}^j \lambda_r (\mathbf{x}_{k_r}^\nu)^T (X_\nu^{-1})^T \Psi^\nu X_\nu^{-1} \sum_{s=1}^j \lambda_s \mathbf{x}_{k_s}^\nu \\ &= \sum_{r=1}^j \lambda_r \mathbf{e}_{k_r}^T \Psi^\nu \sum_{s=1}^j \lambda_s \mathbf{e}_{k_s} \\ &= \sum_{r,s=1}^j \lambda_r \lambda_s \varphi_{g_\nu(k_r),g_\nu(k_s)}, \end{aligned}$$

where $\mathbf{e}_i$ is the i-th unit vecotor of $\mathbb{R}^n$. Likewise, $\mathbf{x}^T P^\mu \mathbf{x} = \sum_{r,s=1}^j \lambda_r \lambda_s \varphi_{g_\mu(\ell_r),g_\mu(\ell_s)}$ and it can be concluded that

$$\mathbf{x}^T P^\nu \mathbf{x} = \mathbf{x}^T P^\mu \mathbf{x}$$

because $g_\nu(k_i) = g_\mu(\ell_i)$ for $i = 1,2,\dots,j$. Thus, V is well-defined and continuous.

Remark 1. Note that $g_\nu(0)$, i.e. the index of the zero vertex $\mathbf{v}_k = \mathbf{0}$, is never needed in the formulas above.

4 COMPUTING A CPQ LYAPUNOV FUNCTION

Consider a switched linear system with arbitrary switching

$$\dot{\mathbf{x}}(t) = A(t)\mathbf{x}(t), \quad A(t) \in \mathcal{A} := \{A_1, A_2, \dots, A_N\},$$

where $A: \mathbb{R}_+ \to \mathcal{A}$ is an arbitrary right-continuous piecewise constant mapping and $N \in \mathbb{N}$ is finite. As discussed in the Introduction, the origin is asymptotically stable for the system, if and only if there exists a Lyapunov function for the system, i.e., a locally Lipschitz function $V\colon \mathbb{R}^n \to \mathbb{R}$ that satisfies

$$\begin{aligned} V(\mathbf{x}) &> 0 && \forall \mathbf{x} \in \mathbb{R}^n \backslash \{\mathbf{0}\}, \qquad (4) \\ V(\mathbf{0}) &= 0, && \\ \langle \nabla V(\mathbf{x}), A_i \mathbf{x} \rangle &< 0 && \forall \mathbf{x} \in \mathbb{R}^n \backslash \{\mathbf{0}\}, \\ & && \forall i \in \{1,2,\dots,N\}, \end{aligned}$$

where $\langle \nabla V(\mathbf{x}), A_i \mathbf{x} \rangle$ denotes the inner product of the vectors $\nabla V(\mathbf{x})$ and $A_i \mathbf{x}$. Strictly speaking $\nabla V(\mathbf{x})$ is the Clarke's subdifferential (Clarke, 1990), which in our CPQ case means that for $\mathbf{x} \in \mathfrak{C}_\nu \cap \mathfrak{C}_\mu$ we need

$$\left\langle \nabla(\mathbf{x}^T P^\nu \mathbf{x}), A_i \mathbf{x} \right\rangle < 0 \text{ and } \left\langle \nabla(\mathbf{x}^T P^\mu \mathbf{x}), A_i \mathbf{x} \right\rangle < 0.$$

Going into details would go beyond the scope of this paper and we refer the interested reader to the literature (Clarke et al., 1998; Baier et al., 2012).

If V is CPQ and parameterized as in (3), then conditions (4) are equivalent to

$$\begin{aligned} \mathbf{x}^T P^\nu \mathbf{x} &> 0 && \forall \mathbf{x} \in \mathfrak{C}_\nu \backslash \{\mathbf{0}\} \\ \mathbf{x}^T Q_i^\nu \mathbf{x} &< 0 && \forall \mathbf{x} \in \mathfrak{C}_\nu \backslash \{\mathbf{0}\} \\ & && \forall i \in \{1,2,\dots,N\} \end{aligned}$$

for all $\mathfrak{S}_\nu \in \mathcal{T}$, where $Q_i^\nu := A_i^T P^\nu + P^\nu A_i$. We will refer to these conditions as P^ν and Q_i^ν being positive definite (p.d.) and negative definite (n.d.), respectively, on the cone $\mathfrak{C}_\nu$.

4.1 Reexpression of Constraints

Now it is desired to reexpress the condition on Q_i^ν into one that can be verified with LP. Note that for $\mathbf{x} \in \mathfrak{C}_\nu$ we can write $\mathbf{x} = X_\nu \boldsymbol{\lambda}$, where $\boldsymbol{\lambda} \in \mathbb{R}_+^n$ are the cone coordinates of $\mathbf{x}$. Thus we can define a matrix

$$\begin{aligned} B_i^\nu &= X_\nu^T Q_i^\nu X_\nu \\ &= X_\nu^T (A_i^T (X_\nu^{-1})^T \Psi^\nu X_\nu^{-1} + (X_\nu^{-1})^T \Psi^\nu X_\nu^{-1} A_i) X_\nu \\ &= \left(\Psi^\nu X_\nu^{-1} A_i X_\nu\right)^T + \Psi^\nu X_\nu^{-1} A_i X_\nu \\ &= (\Psi^\nu \hat{A}_i^\nu)^T + \Psi^\nu \hat{A}_i^\nu, \end{aligned}$$

where $\hat{A}_i^\nu = X_\nu^{-1} A_i X_\nu$. Then we define a matrix C_i^ν such that $c_{k,k}^{\nu,i} = b_{k,k}^{\nu,i}$ and $c_{k,\ell}^{\nu,i} = \max\{0, b_{k,\ell}^{\nu,i}\}$ for all $k,\ell \in \{1,2,\dots,n\}$, where $b_{k,\ell}^{\nu,i}$ and $c_{k,\ell}^{\nu,i}$ are the (k,ℓ)-th entries of B_i^ν and C_i^ν, respectively.

Consider the next proposition.

Proposition 1. *If the sum* $\sum_{\ell=1}^n c_{k,\ell}^{\nu,i} < 0$ *for all* $k \in \{1,2,\dots,n\}$*, then* Q_i^ν *is n.d. on the cone* $\mathfrak{C}_\nu$.

Sketch of proof. We will provide a sketch of the proof here; a detailed proof will be presented in an upcoming paper (Palacios Roman et al., 2024).

Because the off-diagonal elements of C_i^ν are positive, the conditions $\sum_{\ell=1}^n c_{k,\ell}^{\nu,i} < 0$ force the matrix C_i^ν to be a diagonally dominant and n.d. matrix. Because of how C_i^ν is defined from B_i^ν, $\boldsymbol{\lambda}^T C_i^\nu \boldsymbol{\lambda} \geq \boldsymbol{\lambda}^T B_i^\nu \boldsymbol{\lambda}$ for all $\boldsymbol{\lambda} \in \mathbb{R}_+^n$. Thus, B_i^ν is n.d. on the cone $\mathbb{R}_+^n$, from which it follows that Q_i^ν is n.d. on $\mathfrak{C}_\nu$. □

Finally, we can implement the max function in $c_{k,\ell}^{\nu,i} = \max\{0, b_{k,\ell}^{\nu,i}\}$ with the constraints $c_{k,\ell}^{\nu,i} \geq 0$ and $c_{k,\ell}^{\nu,i} \geq b_{k,\ell}^{\nu,i}$. Then, the conditions on C_i^ν become

$$\begin{aligned} c_{k,\ell}^{\nu,i} &\geq \sum_{r=1}^n \left(\psi_{\ell,r}\hat{a}_{r,k}^{\nu,i} + \psi_{k,r}\hat{a}_{r,\ell}^{\nu,i}\right) \\ c_{k,\ell}^{\nu,i} &\geq 0 \end{aligned}$$

for all $k,\ell \in \{1,2,\dots n\}$ and $k \neq \ell$ and

$$2\sum_{r=1}^n \psi_{k,r}\hat{a}_{r,k}^{\nu,i} + \sum_{\substack{\ell=1 \\ \ell\neq k}}^n c_{k,\ell} < 0.$$

for all $k \in \{1,2,\dots,n\}$.

4.2 Constraints of the LP Problem

Equipped with the above results, the final constraints for the LP problem can now be summarized. On every simplex $\mathfrak{S}_\nu := \text{co}\{\mathbf{0}, \mathbf{x}_1^\nu, \mathbf{x}_2^\nu, \dots, \mathbf{x}_n^\nu\}$, we have $X_\nu = [\mathbf{x}_1^\nu, \mathbf{x}_2^\nu, \dots, \mathbf{x}_n^\nu]$ and index-function g_ν such that $\mathbf{x}_i^\nu = \mathbf{v}_{g_\nu(i)}$. The constraints are then given by:

1. $V(\mathbf{v}_i) > 0$ for all $\mathbf{v}_i \in \mathcal{T}$.
2. For each system define $\hat{A}_i^\nu := X_\nu^{-1} A_i X_\nu$ and C_i^ν such that

$$c_{k,k}^{\nu,i} = 2\sum_{r=1}^n \varphi_{g_\nu(r),g_\nu(k)}\hat{a}_{r,k}^{\nu,i}, \quad (5)$$

for all $k \in \{1,\dots,n\}$ and

$$\begin{aligned} c_{k,\ell}^{\nu,i} &\geq \sum_{r=1}^n \left(\varphi_{g_\nu(\ell),g_\nu(r)}\hat{a}_{r,k}^{\nu,i} + \varphi_{g_\nu(k),g_\nu(r)}\hat{a}_{r,\ell}^{\nu,i}\right), \quad (6) \\ c_{k,\ell}^{\nu,i} &\geq 0 \end{aligned}$$

for all $k,\ell \in \{1,\dots,n\}$ and $k \neq \ell$. Then require for each $\mathfrak{S}_\nu \in \mathcal{T}$ that

$$\sum_{\ell=1}^n c_{k,\ell}^{\nu,i} < 0 \quad \forall k \in \{1,2,\dots,n\}.$$

The problem variables are the elements of the matrices Φ and C_i^ν.

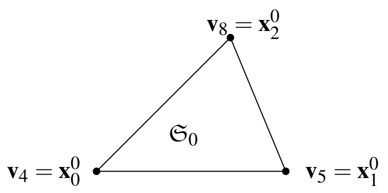
$\mathbf{v}_8 = \mathbf{x}_2^0$
$\mathfrak{S}_0$
$\mathbf{v}_4 = \mathbf{x}_0^0$
$\mathbf{v}_5 = \mathbf{x}_1^0$

Figure 2: Simplex $\mathfrak{S}_0$ with its vertices indexed. For example, with `simNo=0` and `vertNo=1` the functions `VertexNr(simNo,VertNo)` and `Vertex(simNo,VertNo)` return the `int` 5 and Armadillo vector $[1,0]^T$ respectively.

Remark 2. Note that $V(\mathbf{v}_i) > 0$ for all nonzero vertices $\mathbf{v}_i$ of $\mathcal{T}$ does not ensure that $V(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$. This can however be checked a posteriori to solving the LP problem by checking whether there exists a solution to the linear matrix inequality (LMI)

$$P^\nu - (X_\nu^{-1})^T U_\nu X_\nu^{-1} \succ 0,$$

where $U_\nu \in \mathbb{R}_+^{n\times n}$ is the problem variable and $A \succ B$ means that $A - B$ is a symmetric and positive definite matrix. This a posteriori check was proposed in (Johansson, 1999, Proposition 4.4).

5 IMPLEMENTATION AND COMPARISON

In our implementation the object corresponding to the simplicial complex $\mathcal{T}_K$ provides the functions `VertexNr(simNo,VertNo)`, to obtain the vertex number analogous to the indexing function $g_\nu(i)$, and `Vertex(simNo,VertNo)` to obtain the vertex vector analogous to $\mathbf{x}_i^\nu$ (see Figure 2). With these functions we can define our variables and construct our constraints discussed in Section 4.2.

Note that since Φ is a sparse symmetric matrix, we only need to define variables for the upper triangular half. Further, we only need $\varphi_{k,\ell}$ if $\mathbf{v}_k$ and $\mathbf{v}_\ell$ are nonzero vertices of the same simplex and then we define `var = ivec{'P', min(k,l), max(k,l)}`. For example for the simplex in Figure 2 we only need $\varphi_{5,8}$, $\varphi_{5,5}$ and $\varphi_{8,8}$. To avoid unnecessary checks and multiple definitions of variables, we first define the variables above the diagonal in this way, and subsequently the diagonal $\varphi_{k,k}$, $k = 1,\dots,p$. Note that where the index for the zero vertex can be skipped, as it is not used in the constraints.

Furthermore, each C_i^ν is a symmetric $n \times n$ matrix so we only need the upper triangular half of the elements. In fact, we only need the elements above the diagonal since the diagonal is equal to B_i^ν, so rather than using $c_{k,k}^{\nu,i}$ in the constraint we can directly use the right hand side of (5).

In this way we can define each variable, push into the `Variables` vector and then sort it.

All that is left to complete the setup of the LP problem is defining the constraints. This is were the benefit of our structure defined in Section 2 is most apparent since there is no need to keep track of whether a variable appears twice in a sum, as in (6), or not. We can simply go through each constraint using `new_aij` to add the coefficient of each summand and `close_constr` before moving to the next constraint.

As an alternative, we could use a parser such as YALMIP (Löfberg, 2004) in MATLAB (The MathWorks Inc., 2023) to solve for Φ. This would admittedly entail a less convoluted set-up process since we could create matrix variables and don't need to implement the max function with an intermediate variable like C_i^ν. However, since MATLAB is an interpreted language it doesn't benefit from the efficiency associated with compiled languages, in our case the efficiency of the GNU Compiler `g++`.

For comparison consider the arbitrarily switched linear system discussed in (Andersen et al., 2023; Polanski, 1997; Pyatnitskii, 1971; Brockett, 1966) with subsystems given by

$$A_1 = \begin{bmatrix} 0 & 1 \\ -0.01 & -2 \end{bmatrix} \quad \text{and} \quad A_2 = \begin{bmatrix} 0 & 1 \\ -11.7 & -2 \end{bmatrix}.$$

We can compute a CPQ Lyapunov function with a scaling factor $K \geq 4$ and run the computations twenty times for each K. As can be seen in Figure 3, the average time for each scaling factor using the C++ implementation is roughly $1/200$th of the time needed when using MATLAB. We used the SDPT3 (Toh et al., 1999) solver in the MATLAB implementation and Gurobi version 11 for the C++ implementation. Additional MATLAB tooling to speed up the computations like the MATLAB Coder can not be utilized to minimize computation time without major changes to YALMIP.

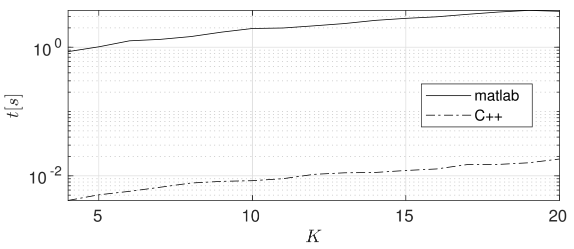

Figure 3: Comparison of the computational time for parameterizing Lyapunov functions with our method in MATLAB and C++; run on i9900K (8 cores) under Linux Mint. Average computation time of twenty tests as a function of triangulation scaling factor K in $\mathcal{T}_K$. Note that the C++ code is ca. 200 times faster than the MATLAB code.

6 CONCLUSIONS

We presented the implementation of a method to compute continuous and piecewise quadratic (CPQ) Lyapunov functions for switched linear systems in C++; such a Lyapunov function is a common Lyapunov function for all the linear subsystems. We compared our implementation to an analogous implementation in MATLAB and showed that the C++ implementation is more than 100 times faster for an example system from the literature. We expect this difference in computational speed to be representative for the general case, as the setup of the linear programming (LP) problem used to parameterize the CPQ Lyapunov function is quite involved and an interpreted computer language like MATLAB is at a great disadvantage in comparison to a compiled language like C++.

REFERENCES

Andersen, S., Giesl, P., and Hafstein, S. (2023). Common Lyapunov functions for switched linear systems: Linear programming-based approach. *IEEE Control Systems Letters*, 7:901–906.

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.

Brockett, R. (1966). The status of stability theory for deterministic systems. *IEEE Trans. Automat. Control*, 11(3):596–606.

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

Clarke, F., Ledyaev, Y., and Stern, R. (1998). Asymptotic stability and smooth Lyapunov functions. *J. Differential Equations*, 149:69–114.

Davrazos, G. and Koussoulas, N. (2001). A review of stability results for switched and hybrid systems. In *Proceedings of 9th Mediterranean Conference on Control and Automation*, Dubrovnik, Croatia.

Dayawansa, W. and Martin, C. (1999). A converse Lyapunov theorem for a class of dynamical systems which undergo switching. *IEEE Trans. Automat. Control*, (44):751–760.

Deenen, D., Sharif, B., van den Eijnden, S., Nijmeijer, H., Heemels, M., and Heertjes, M. (2021). Projection-based integrators for improved motion control: Formalization, well-posedness and stability of hybrid integrator-gain systems. *Automatica*, 133:109830.

Goebel, R., Hu, T., and Teel, A. (2006). *Current Trends in Nonlinear Systems and Control. Systems and Control: Foundations & Applications*, chapter Dual Matrix Inequalities in Stability and Performance Analysis of Linear Differential/Difference Inclusions, pages 103–122. Birkhauser.

Gurobi Optimization, LLC (2023). Gurobi Optimizer Reference Manual.

Hafstein, S. (2019). *Simulation and Modeling Methodologies, Technologies and Applications*, volume 873 of *Advances in Intelligent Systems and Computing*, chapter Fast Algorithms for Computing Continuous Piecewise Affine Lyapunov Functions, pages 274–299. Springer.

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

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.

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

Liberzon, D. (2003). *Switching in systems and control*. Systems & Control: Foundations & Applications. Birkhäuser.

Löfberg, J. (2004). YALMIP: A toolbox for modeling and optimization in MATLAB. In *In Proceedings of the CACSD Conference*, Taipei, Taiwan.

Mason, P., Boscain, U., and Chitour, Y. (2006). Common polynomial Lyapunov functions for linear switched systems. *SIAM J. Control Optim.*, 45(1):226–245.

Mason, P., Chitour, T., and Sigalotti, M. (2022). On universal classes of Lyapunov functions for linear switched systems. arXiv:2208.09179.

Palacios Roman, J., Hafstein, S., Giesl, P., van den Eijnden, S., Andersen, S., and Heemels, M. (2024). Constructing continuous piecewise quadratic Lyapunov functions with linear programming. [Manuscript in preparation].

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

Pyatnitskii, Y. (1971). Absolute stability criterion for the 2nd order nonlinear controlled systems with one nonlinear nonstationary element. *Autom. Remote Control*, 32(1):1–11.

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

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

Shorten, R., Wirth, F., Mason, O., Wulff, K., and King, C. (2007). Stability criteria for switched and hybrid systems. *SIAM Review*, 49(4):545–592.

Sun, Z. and Ge, S. (2011). *Stability Theory of Switched Dynamical Systems*. Communications and Control Engineering. Springer.

The MathWorks Inc. (2023). MATLAB version: 23.2.0 (r2023b).

Toh, K. C., Todd, M. J., and Tütüncü, R. H. (1999). SDPT3 - a matlab software package for semidefinite programming, version 1.3. *Optimization Methods and Software*, 11(1-4):545–581.

van den Eijnden, S., Heemels, M., Nijmeijer, H., and Heertjes, M. (2022). Stability and performance analysis of hybrid integrator–gain systems: A linear matrix inequality approach. *Nonlinear Analysis: Hybrid Systems*, 45:101192.

van den Eijnden, S., Heertjes, M., Heemels, M., and Nijmeijer, H. (2020). Hybrid integrator-gain systems: A remedy for overshoot limitations in linear control? *IEEE Control Systems Letters*, PP:1–1.

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