CAQM: Convexity Analysis of Quadratic Maps

Anatoly Dymarsky[*1], Elena Gryazina[†1], Boris Polyak[‡2], and Sergei Volodin[§1]

[1]Skolkovo Institute of Science and Technology
[2]Institute for Control Sciences RAS

CAQM is a MATLAB library designed to analyze geometric properties of images of quadratic maps (which are manifolds of real-valued solutions of systems of quadratic equations).

This file is aimed at both guiding users with the installation steps and enumerating the methods and techniques which constitute the library. The main functionality described below is contained in several MATLAB functions located in the main folder.

Additional procedures which reside in the `library/` folder are described in the accompanying file `library.pdf`.

[*]a.dymarsky@skoltech.ru
[†]e.gryazina@skoltech.ru
[‡]boris@ipu.ru
[§]sergei.volodin@phystech.edu

Installation and Testing

The library requires MATLAB and CVX as prerequisites. Having obtained those, the following steps are required in order to use CAQM.

1. **Obtaining source code** can be done in two ways. First one requires basic knowledge of the GIT version control system. Simply clone the repository:

   `git clone git@github.com:sergeivolodin/CertificateCutting.git`

   The second option is to download the latest snapshot of the repository in form of a .zip archive. To do so, navigate to github.com/sergeivolodin/CertificateCutting, press the green ,,Clone or download" button and then choose ,,Download ZIP" in the drop-down menu. Then unpack the obtained archive. Both cases should give a folder called `CertificateCutting` (or `CertificateCutting-master` in the latter case). This folder must contain a file `README.md`.

2. **Configuring MATLAB** involves adding the folder to the PATH. The following directories (assuming that the root folder is called `CertificateCutting`) should be added:

   `CertificateCutting`, `CertificateCutting/library`, `CertificateCutting/tests`.

   MATLAB gives two ways of adding a folder to MATLAB PATH. The first one involves console commands and the second one uses GUI.

3. **Testing the library** is an essential step which ensures that the library and all of its dependencies were installed correctly. To run the test, open MATLAB,

   navigate to `CertificateCutting/tests` and run the `testCAQM.m` file.

   This file will go through essential parts of the functionality. During that period, a parallel cluster might be started (depending on the MATLAB version) and several GUI dialog boxes might be displayed. If the test succeeded, a message `TEST PASSED` should be displayed in the console output. Otherwise, if the test has failed, a YouTube video demonstrating normal test operations might prove helpful in diagnosing the issue and overcoming installation mistakes. In other cases, feel free to contact the authors.

   Moreover, there exist another group of tests which checks each function individually in a more thorough manner (run in MATLAB console from the `CertificateCutting/tests` directory:

   ```
   >> runtests('testFunctions')
   ```

After installation and testing, the library is ready to use in your applications. The sections below describe the functionality in details.

Notations

1. Real case, the map $f\colon \mathbb{R}^n \to \mathbb{R}^m$

$$f_k(x) = x^T A_k x + 2b_k^T x, \quad A_k = A_k^T, \quad x, b_k \in \mathbb{R}^n, \quad i = k \dots m. \tag{1}$$

2. Complex case, the map $f\colon \mathbb{C}^n \to \mathbb{R}^m$

$$f_k(x) = x^* A_k x + b_i^* x + x^* b_k, \quad A_k = A_k^*, \quad x, b_k \in \mathbb{C}^n, \quad i = k \dots m, \tag{2}$$

where $\cdot^*$ is Hermitian conjugate.

We will use $\mathbb{V}$ to denote $\mathbb{R}^n$ in the real case and $\mathbb{C}^n$ in the complex case. We also use the following notations:

Notation 1. For a vector $c = (c_1, ..., c_m)$ and a tuple of vectors $b = (b_1, ..., b_m),\ b_k \in \mathbb{V}$, or a tuple of $n \times n$ matrices $A = (A_1, ..., A_m),\ A_k \in \mathbb{V}^2$, the dot product is defined as follows,

$$c \cdot b = \sum_{k=1}^{m} c_k\, b_k, \qquad c \cdot A = \sum_{k=1}^{m} c_k\, A_k\ .$$

Notation 2. The image of f is denoted as F,

$$F = f(\mathbb{V}) \subset \mathbb{R}^m\ .$$

Notation 3. The convex hull of F is denoted as G:

$$G = \mathrm{conv}\,(\mathrm{F}) \subset \mathbb{R}^{\mathrm{m}}\ .$$

Notation 4. The boundary points of F touched by a supporting hyperplane with the normal vector $c \in \mathbb{R}^m$,

$$\partial F_c = \arg\min_{y \in F}(c \cdot y)$$

Notation 5. The boundary points of G touched by a supporting hyperplane with the normal vector $c \in \mathbb{R}^m$,

$$\partial G_c = \arg\min_{y \in G}(c \cdot y)$$

Notation 6. Set of normal vectors c, such that ∂F_c is non-convex is denoted as C_-,

$$C_- = \{c \in \mathbb{R}^m \,\big|\, \text{Set } \partial F_c \text{ is non-convex}\}$$

Functionality

The library consists of several functions, each of them is defined in a separate .m file. Input format for specify the quadratic map is as follows.

- The array $A(i, j, k)$ denotes i'th row and j'th column of the $n \times n$ matrix $A_k \in \mathbb{V}^2$
- The array $b(i, k)$ denotes i'th element of the vector $b_k \in \mathbb{V}$

1. **Feasibility membership oracle,** `infeasibility_oracle.m`

   ```
   is_infeasible = infeasibility_oracle(A, b, y)
   ```

   Input:

   - The map f specified by matrices A_k and vectors b_k,
   - A point $y \in \mathbb{R}^m$.

   Output: determines if $y \in G$, returns `is_infeasible`=1 if $y \notin G$, `is_infeasible`=0 if $y \in G$.
   Exceptions: None

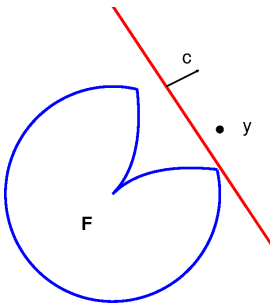

Figure 1: Infeasibility oracle: hyperplane orthogonal to c separates the point y from the convex hull G of F

This function attempts to certify that the point y does not belong to the convex hull G of F by separating y and G by an appropriate hyperplane. This is illustrated in Fig. 1, see Theorem 3.2 in the accompanying article for details.

Procedure returns `is_infeasible`= 1 if the desired hyperplane was found. In this case $y \notin G$ and consequently $y \notin F$, implying there is no $x \in \mathbb{V}$ such that $y = f(x)$, i.e. this point is infeasible. If the hyperplane was not found the function returns `is_infeasible`= 0, which means the feasibility of y with respect to F is uncertain.

2. **Boundary oracle,** `boundary_oracle.m`

```
[t, is_in_F] = boundary_oracle(A, b, y, d)
```

Input:

- The map f specified by matrices A_k and vectors b_k,
- A point $y \in G$,
- A direction $d \in \mathbb{R}^m$.

Output: finds and returns distance t to the boundary of G from the point y inside G in the direction d; verifies if the boundary point belongs to F.
Exception: if the input vector $y \notin G$ or in the case if ∂G is not smooth at the boundary point $y + t\,d \in \partial G$, the function produces an exception.

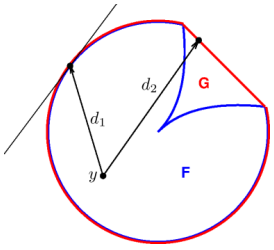

Figure 2: Boundary oracle: distance from y to the boundary ∂G in the direction d. The boundary point of G may or may not belong to ∂F (cases of $d = d_1$ and $d = d_2$ respectively).

This function finds point $y + t\,d$ on the boundary ∂G with the largest $t = \sup\{\tau \big| y + \tau d \in G\}$ and checks if this point belongs to F. This is illustrated in Fig. 2, see Optimization task (3) in the accompanying article for details.

The function returns `t` with the value of t, variable `is_in_F`$= 1$ if the boundary point $y + t\,d$ belongs to F, and variable `is_in_F`$= 0$ if feasibility of $y + t\,d$ with respect to F is uncertain.

3. **Normal vector at the boundary,** `get_c_from_d.m`

```
c = get_c_from_d(A, b, y, d)
```

Input:

- The map f specified by matrices A_k and vectors b_k,
- A point $y \in G$,
- A direction $d \in \mathbb{R}^m$.

Output: finds point $y + t\,d$ at the boundary of G and returns vector c normal to ∂G at that point
Exception: if on the input $y \notin G$ or the normal vector to ∂G at $y + td$ does not exist (because ∂G is not smooth at this point) the function produces an exception.

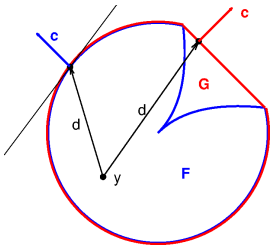

Figure 3: Normal vector at the boundary: vector c normal to ∂G at the boundary point $y + td$. This function only considers the convex hull G and does not distinguish between points $y + td$ which do or do not belong to F.

This function finds the boundary point $y + td \in \partial G$ and calculates the vector c normal to ∂G at that point by using the dual formulation of the optimization problem, see equation (4) of the accompanying article. This is schematically illustrated in Fig. 3.

The function returns vector variable `c` with the value of c.

4. **"Non-convex direction",** `get_c_minus.m`

   `c = get_c_minus(A, b, [y], [k], [DEBUG])`

   Input:

   - The map f specified by matrices A_k and vectors b_k,
   - *(optional)* A point $y \in G$,
   - *(optional)* Number of iterations k.
   - *(optional)* Binary variable adding verbose output.

   Output: finds and returns vector c such that ∂F_c is non-convex through a stochastic algorithm using up to k iterations
   Exception: The functions throws an exception in the case if the vector c was not found during the k iterations

   This function consequently generates up to k random directions d and for each one finds vector c normal to ∂G at the boundary point $y + t\,d \in \partial G$. Next it finds $\partial_c F$, the intersection of F with the supporting hyperplane orthogonal to c and checks if it is non-convex. We note that non-convexity of $\partial_c F$ implies non-convexity of F. This function stops and returns c if non-convexity of $\partial_c F$ was established during one of the iterations. If the vector c was not found, an exception is produced. If y and k are not specified, the function uses default values $y = f(0) = 0$ and $k = 10$.

5. **Nonconvexity certificate,** `nonconvexity_certificate.m`

   `is_nonconvex = nonconvexity_certificate(A, b, [y], [k])`

   Input:

   - The map f specified by matrices A_k and vectors b_k,
   - *(optional)* A point $y \in G$,
   - *(optional)* Number of iterations k.

   Output: attempts to establish if F is convex, returns `is_nonconvex`=1 if F is non-convex, `is_nonconvex`=0 if uncertain. **Exceptions:** None

   This function calls `get_c_minus` and returns `is_nonconvex`=1 if the latter returns a non-trivial c.

6. **Positive-definite** $c \cdot A$**,** `get_c_plus.m`

   `c_plus = get_c_plus(A, [k], [DEBUG])`

   Input:

   - Matrices A_k
   - *(optional)* The number of iterations k
   - *(optional)* Binary variable adding verbose output.

   Output: finds and returns vector c_+ such that $c_+ \cdot A \succ 0$.
   Exception: if c_+ was not found during k iterations of the randomized procedure.

   This function utilizes a randomized algorithm which is used to find c_+ such that $c_+ \cdot A \succ 0$.

   If successful, the function terminates and returns c_+ on the exit, otherwise the search attempt is repeated up to k times. If not specified explicitly, the default value of $k = 10$. If c_+ is not found during k iterations the function produces an exception.

7. **Convex subpart,** `get_z_max.m`

```
z_max = get_z_max(A, b, c_plus, [z_max_guess], [k], [DEBUG])
```

Input:

- The map f specified by matrices A_k and vectors b_k,
- The vector c_+ such that $c_+ \cdot A \succeq 0$,
- *(optional)* The guess value $z_{\max}^{\text{guess}}$,
- *(optional)* The number of iterations k.
- *(optional)* Binary variable adding verbose output.

Output: finds and returns maximal value $z_{\max}$ such that the compact part of F "cut" by the hyperplane $c_+ \cdot (y - y_0) = z_{\max}$, where $y_0 \in \partial_{c_+} F$, is still convex.
Exception: produces an exception if non-convexity of F confined within the half-plane $c_+ \cdot (y - y_0) \leq z_{\max}^{\text{guess}}$ has not been established, i.e. no non-convexities were found in that region.

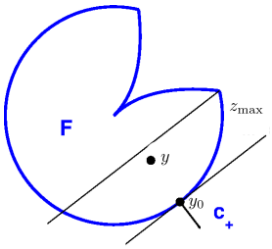

Figure 4: Maximal value of $z_{\max}$ such that the subpart of F, $F_{z_{\max}}^{c_+} \equiv \{y \,|y \in F, c_+ \cdot (y_0 - y) \leq z_{\max}\}$, which is "cut" from F by a hyperplane orthogonal to c_+ does not contain non-convexities.

This function returns maximal value $z_{\max}$ such that the hyperplane perpendicular to c_+ and located distance $z_{\max}$ away from the boundary of F still does not contain non-convexities. More precisely, the compact part of F confined by the half-space $\{y \,|c_+ \cdot (y_0 - y) \leq z_{\max}^{\text{guess}}\}$ is convex. Here $y_0 \in \partial_{c_+} F$, and since $c_+ \cdot A \succ 0$, the set $\partial_{c_+} F$ is a singleton $\{y_0\}$. Moreover, $z_{\max}$ is maximal in a sense that enlarging it would result in the resulting ,,cut" containing at least one boundary non-convexity, which are non-convex subsets of the boundary ∂F_c defined as the points touched by a hyperplane with a normal vector c. The geometric meaning of $z_{\max}$ is illustrated in Fig. 4.

The function first tries to identify "non-convex directions" c_- using `get_c_minus` and then "follow" each non-convexity to the smallest value of z. This is described in detail in the section 4.4 of the accompanying paper.

If no "non-convex directions" found, the function produces an exception. In case the input value of c_+ does not satisfy $c_+ \cdot A \succ 0$, the function produced as exception. If the maximal number of iterations k (to be used with `get_c_minus`) is not specified on input, a heuristic default value $k = 10$ is used.

The guess value $z_{\max}^{\text{guess}}$ is supposed to be substantially large to detect non-convexity of F. If it is not specified on input, a default value of $z_{\max}^{\text{guess}} = 10\text{Tr}(c_+ \cdot A)$ is used.

It is important to keep in mind that the algorithm is heuristic. A non-trivial return value $z_{\max} \neq z_{\max}^{\text{guess}}$ does not guarantee convexity of $F_{z_{\max}} \equiv \{y \,|y \in F, c_+ \cdot (y_0 - y) \leq z_{\max}\}$, but only that $\{y \,|y \in F, c_+ \cdot (y_0 - y) \leq z\}$ for any $z > z_{\max}$ contains boundary non-convexities. Nevertheless, increasing k would increase certainty (in the probabilistic sense) that $F_{z_{\max}}$ is indeed convex.