CAQM: Convexity Analysis of Quadratic Maps

Anatoly Dymarsky[1,2], Elena Gryazina[1,3], Sergei Volodin[1], and Boris Polyak[3]

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

CAQM is a MATLAB library designed to analyze geometric properties of images of quadratic mappings.

This document explains how to install CAQM and describes basic functionality of the library. The corresponding MATLAB functions are located in the main folder.

Additional functionality is provided by the functions located in the folder `library/`. These functions are described in the accompanying file `library.pdf`.

Installation and Testing

The library requires MATLAB and CVX as prerequisites. The following steps are necessary to set up and use CAQM.

1. **Obtaining the library** is possible in two ways. First way is by cloning the GIT repository:

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

   Second way is to download the latest snapshot of the repository from github.com/sergeivolodin/CAQM and unpack the archive. After obtaining the folder `CAQM` (`CAQM-master` in the latter case), no further installation is required.

2. **Configuring the library** requires adding the folder `CAQM` or `CAQM-master` together with its subfolders

   `CAQM/library`, `CAQM/tests`.

   to `MATLAB` path.

3. **Testing the library** is an essential step which ensures that the library and all of its components were installed correctly. To run the test, open MATLAB, navigate to `CAQM/tests` and run `testCAQM.m`.

   This test will check all essential components. Depending on the MATLAB version, it might start a parallel cluster and open several GUI dialog boxes. If the test succeeds, a message `TEST PASSED` should be printed to the console output. The YouTube video demonstrates the normal test operations, which might be useful to identify possible issues should the test fail.

   There is an additional group of tests which checks each function individually in a more thorough manner. To run it, type in MATLAB console from the `CertificateCutting/tests` directory:

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

After installation and testing, the library is ready to use. The rest of this document describes main functionality in detail.

Notations

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

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

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

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

   where $\cdot^*$ stands for 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 full 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:

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

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_-,

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

Notation 7. The image of a unit sphere of f is denoted as H,

$$H = f(\{|x| = 1 \big| x \in \mathbb{V}\}) \subset \mathbb{R}^m\ .$$

Functionality

The library consists of several functions, each of them is defined in a separate .m file. The input format to specify a 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 $b_k \in \mathbb{V}$

In order to interpret the input map as complex, the tensor A or the matrix b should contain at least one complex number. Otherwise, the library assumes the map is real.

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 (the map can be ; in this case all entries of matrix b should be zero, $b_{ki} = 0$)
   - 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

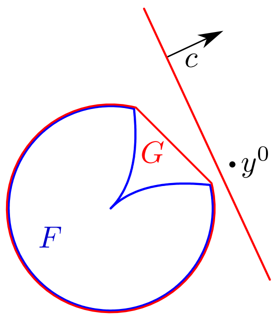

   Figure 1: Infeasibility oracle: hyperplane orthogonal to c separates the point y from the convex hull G of F

   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 4.1 in the accompanying paper for details.

   The function 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 y does belong to G but 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 (the map can be homogeneous; in this case all of the entries of the matrix b should be zero: $b_{ki} = 0$)
   - 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.

   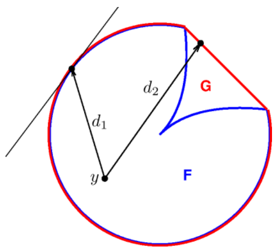

   Figure 2: Boundary oracle: distance from y to the boundary ∂G in the direction d. The boundary point of G may or may not belong to ∂F (cases of $d = d_1$ and $d = d_2$ respectively).

   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 Section 5.2 of the accompanying paper for details.

   The variable `t` on return contains the value of t, the 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 (the map can be homogeneous; in this case all of the entries of the matrix b should be zero: $b_{ki} = 0$)
   - 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.

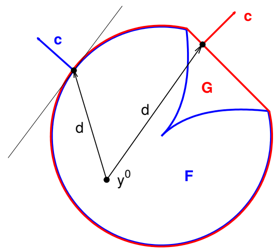

Figure 3: Normal vector at the boundary: vector c normal to ∂G at the boundary point $y + td$. This function only considers the convex hull G and is agnostic to whether the point $y + td$ does or does not belong to F.

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 Section 5.2 of the accompanying paper. This is schematically illustrated in Fig. 3. Variable `c` on return contains 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 (the map can be homogeneous; in this case all of the entries of the matrix b should be zero: $b_{ki} = 0$)
- *(optional)* a point $y \in G$,
- *(optional)* number of iterations k,
- *(optional)* binary variable `DEBUG`=1/0 that turns on/off the verbose output.

Output: finds and returns vector c such that ∂F_c is non-convex by employing up to k iterations/restarts of a stochastic algorithm.
Exception: this functions throws an exception if such a vector c was not found.

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 ∂F_c, the intersection of F with the supporting hyperplane orthogonal to c and checks if it is non-convex. We note that non-convexity of ∂F_c implies non-convexity of F. This function stops and returns c if non-convexity of ∂F_c 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. **"Non-convex direction" for real homogeneous maps and the image of the sphere,** get_c_minus_homog_real_H.m

```
c = get_c_minus_homog_real_H(A)
```

Input: the map f specified by matrices A_k
Output: finds and returns vector c such that ∂H_c is non-convex using a deterministic procedure.
Exception: this functions throws an exception if such a vector c was not found.

The method implements the following convex problem described in the Appendix B of the Article:

$$\max \ \gamma$$
$$\sum_{i=1}^{m} c_i = 1$$
$$\operatorname{Tr}(c \cdot A) = 0$$
$$c \cdot A \succeq \gamma I$$

The intuition is to increase the smallest eigenvalue, avoiding cases $c = 0$ by $\sum c_i = 1$. In addition, matrix $c \cdot A$ is required to have both positive and negative eigenvalues by $\operatorname{Tr}(c \cdot A) = 0$. By increasing the smallest eigenvalue given the constraints, the goal is to make it equal to the second smallest, thus resulting in a degeneracy. In case if the problem was not solved, the function produces an exception. In case if the resulting c does not result in a degenerate lowest eigenvalue, the function produces an exception.

6. **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 (the map can be homogeneous; in this case all of the entries of the matrix b should be zero: $b_{ki} = 0$)
- *(optional)* a point $y \in G$,
- *(optional)* number of iterations k.

Output: attempts to establish if F is non-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.

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

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

Input:

- matrices A_k,
- *(optional)* number of iterations k
- *(optional)* binary variable `DEBUG` that =1/0 that turns on/off the verbose output.

Output: finds and returns vector c_+ such that $c_+ \cdot A \succ 0$ by employing up to k iterations/restarts of a stochastic algorithm.
Exception: if c_+ was not found.

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 is 10. If c_+ is not found during k iterations, the function produces an exception.

8. **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 map *cannot* be homogeneous; at least some entries of the matrix b should be non-zero: $b_{ki} \neq 0$)
- a vector c_+ such that $c_+ \cdot A \succeq 0$,
- *(optional)* guess value $z_{\rm max}^{\rm guess}$,
- *(optional)* the number of iterations k,
- *(optional)* binary variable `DEBUG` that =1/0 that turns on/off the verbose output.

Output: finds and returns maximal value $z_{\rm max}$ such that the compact part of F "cut" by the hyperplane $c_+ \cdot (y - y_0) = z_{\rm max}$, where $y_0 \in \partial F_{c_+}$, is likely convex.
Exception: produces an exception if non-convexity of F confined within the half-plane $c_+ \cdot (y - y_0) \leq z_{\rm max}^{\rm guess}$ has not been established, or if $c_+ \cdot A \not\succ 0$

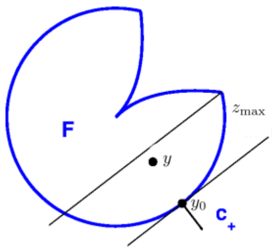

Figure 4: Maximal value of $z_{\rm max}$ such that the subpart of F, $F^{c+}_{z_{\rm max}} \equiv \{y\,|y \in F, c_+ \cdot (y_0 - y) \leq z_{\rm max}\}$, which is "cut" from F by a hyperplane orthogonal to c_+, is (expected to be) convex. The starting point for the non-convexity certificate is $y = y_0 + z_{\rm max}^{\rm guess} c_+$. See section 6 of the accompanying paper for details.

This function returns maximal value $z_{\rm max}$ such that the hyperplane perpendicular to c_+ and located distance $z_{\rm max}$ away from the boundary of F does not contain known non-convexities. Assuming all non-convexities (in the form of corresponding "non-convex directions" c_-) have been identified, this ensures that the compact part of F confined in the half-space $\{y\,|c_+ \cdot (y_0 - y) \leq z_{\rm max}^{\rm guess}\}$ is convex. Here $y_0 \in \partial F_{c_+}$, and since $c_+ \cdot A \succ 0$, the set ∂F_{c_+} consists of only one point $\{y_0\}$. The value $z_{\rm max}$ is maximal in the sense that taking any larger value would result in $F^{c+}_{z_{\rm max}} \equiv \{y\,|y \in F, c_+ \cdot (y_0 - y) \leq z_{\rm max}\}$ becoming non-convex. The geometric meaning of $z_{\rm max}$ is illustrated in Fig. 4.

The function first tries to identify the "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 6 of the accompanying paper.

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

The guess value $z_{\rm max}^{\rm guess}$ is supposed to be substantially large to detect non-convexity of F. If it is not specified on input, a default value of $z_{\rm max}^{\rm guess} = 10{\rm Tr}(c_+ \cdot A)$ is used. It is generally the case that a bigger value of $z_{\rm max}$ results in a lower probability of finding non-convexity. It means that $z_{\rm max}^{\rm guess}$ must be sufficiently large to allow for non-convexity to appear but not too large to still allow algorithm to discover the non-convex directions. The latter restriction can be alleviated by increasing the number of iterations k.

It is important to keep in mind that the algorithm is stochastic and not exact in nature. A non-trivial return value $z_{\max} \neq z_{\max}^{\text{guess}}$ does not guarantee convexity of $F_{z_{\max}}^{c_+} \equiv \{y \,|y \in F, c_+ \cdot (y_0 - y) \leq z_{\max}\}$, but only that $\{y \,|y \in F, c_+ \cdot (y_0 - y) \leq z\}$ for any $z > z_{\max}$ contains boundary non-convexities. Nevertheless, provided all (families of) c_- were identified, Proposition 6.1 of the accompanying paper guarantees convexity of $F_{z_{\max}}^{c_+}$. Hence, increasing k increases certainty (in the probabilistic sense) that $F_{z_{\max}}$ is indeed convex.