elltool_manual Documentation

Release 0.1

Lukashevichus K.

November 05, 2013

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Research on dynamical and hybrid systems has produced several methods for verification and controller synthesis. A common step in these methods is the reachability analysis of the system. Reachability analysis is concerned with the computation of the reach set in a way that can effectively meet requests like the following:

1. For a given target set and time, determine whether the reach set and the target set have nonempty intersection.
2. For specified reachable state and time, find a feasible initial condition and control that steers the system from this initial condition to the given reachable state in given time.
3. Graphically display the projection of the reach set onto any specified two- or three-dimensional subspace.

Except for very specific classes of systems, exact computation of reach sets is not possible, and approximation techniques are needed. For controlled linear systems with convex bounds on the control and initial conditions, the efficiency and accuracy of these techniques depend on how they represent convex sets and how well they perform the operations of unions, intersections, geometric (Minkowski) sums and differences of convex sets. Two basic objects are used as convex approximations: polytopes of various types, including general polytopes, zonotopes, parallelotopes, rectangular polytopes; and ellipsoids.

Reachability analysis for general polytopes is implemented in the Multi Parametric Toolbox (MPT) for Matlab Kvasnica et al. (2004; "Multi-Parametric Toolbox Homepage"). The reach set at every time step is computed as the geometric sum of two polytopes. The procedure consists in finding the vertices of the resulting polytope and calculating their convex hull. MPT's convex hull algorithm is based on the Double Description method Motzkin et al. (1953) and implemented in the CDD/CDD+ package ("CDD/CDD+ Homepage"). Its complexity is V^n, where V is the number of vertices and n is the state space dimension. Hence the use of MPT is practicable for low dimensional systems. But even in low dimensional systems the number of vertices in the reach set polytope can grow very large with the number of time steps. For example, consider the system,

$$x_{k+1} = Ax_k + u_k,$$

with $A = \begin{bmatrix} \cos 1 & -\sin 1 \\ \sin 1 & \cos 1 \end{bmatrix}$, $u_k \in \{u \in \mathbf{R}^2 \mid \|u\|_\infty \leqslant 1\}$, and $x_0 \in \{x \in \mathbf{R}^2 \mid \|x\|_\infty \leqslant 1\}$.

Starting with a rectangular initial set, the number of vertices of the reach set polytope is $4k + 4$ at the kth step.

In d/dt ("d/dt Homepage"), the reach set is approximated by unions of rectangular polytopes E.Asarin et al. (2000).

The algorithm works as follows. First, given the set of initial conditions defined as a polytope, the evolution in time of the polytope's extreme points is computed (figure 1.1 (a)).

$R(t_1)$ in figure 1.1 (a) is the reach set of the system at time t_1, and $R[t_0, t_1]$ is the set of all points that can be reached during $[t_0, t_1]$. Second, the algorithm computes the convex hull of vertices of both, the initial polytope and $R(t_1)$ (figure 1.1 (b)). The resulting polytope is then bloated to include all the reachable states in $[t_0, t_1]$ (figure 1.1 (c)). Finally, this overapproximating polytope is in its turn overapproximated by the union of rectangles (figure 1.1 (d)). The same procedure is repeated for the next time interval $[t_1, t_2]$, and the union of both rectangular approximations is taken (figure 1.1 (e,f)), and so on. Rectangular polytopes are easy to represent and the number of facets grows linearly

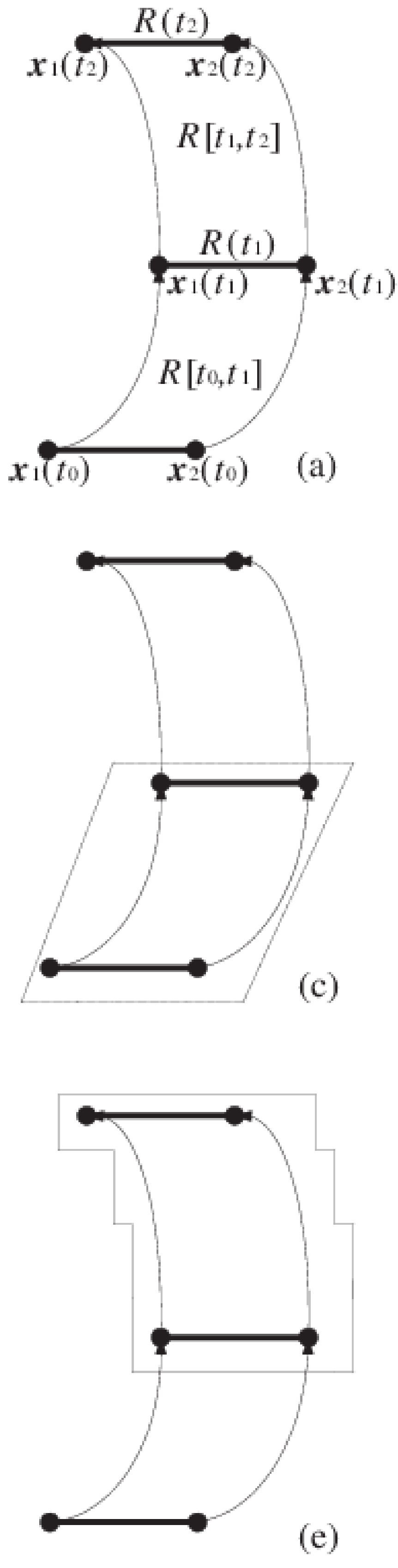


Figure 1.1: Reach set approximation by union of rectangles

with dimension, but a large number of rectangles must be used to assure the approximation is not overly conservative. Besides, the important part of this method is again the convex hull calculation whose implementation relies on the same CDD/CDD+ library. This limits the dimension of the system and time interval for which it is feasible to calculate the reach set.

Polytopes can give arbitrarily close approximations to any convex set, but the number of vertices can grow prohibitively large and, as shown in Avis, Bremner, and Seidel (1997), the computation of a polytope by its convex hull becomes intractable for large number of vertices in high dimensions.

The method of zonotopes for approximation of reach sets Girard (2005; A.Girard, Guernic, and O.Maler 2006; "MATISSE Homepage") uses a special class of polytopes (see ("Zonotope Methods on Wolfgang Kühn Homepage")) of the form,

$$Z = \{x \in \mathbf{R}^n \mid x = c + \sum_{i=1}^{p} \alpha_i g_i,\ -1 \leqslant \alpha_i \leqslant 1\},$$

wherein c and $g_1, ..., g_p$ are vectors in $\mathbf{R}^n$. Thus, a zonotope Z is represented by its center c and 'generator' vectors $g_1, ..., g_p$. The value p/n is called the order of the zonotope. The main benefit of zonotopes over general polytopes is that a symmetric polytope can be represented more compactly than a general polytope. The geometric sum of two zonotopes is a zonotope:

$$Z(c_1, G_1) \oplus Z(c_2, G_2) = Z(c_1 + c_2, [G_1\ G_2]),$$

wherein G_1 and G_2 are matrices whose columns are generator vectors, and $[G_1\ G_2]$ is their concatenation. Thus, in the reach set computation, the order of the zonotope increases by p/n with every time step. This difficulty can be averted by limiting the number of generator vectors, and overapproximating zonotopes whose number of generator vectors exceeds the limit by lower order zonotopes. The benefits of the compact zonotype representation, however, appear to diminish because in order to plot them or check if they intersect with given objects and compute those intersections, these operations are performed after converting zonotopes to polytopes.

CheckMate ("CheckMate Homepage") is a Matlab toolbox that can evaluate specifications for trajectories starting from the set of initial (continuous) states corresponding to the parameter values at the vertices of the parameter set. This provides preliminary insight into whether the specifications will be true for all parameter values. The method of oriented rectangluar polytopes for external approximation of reach sets is introduced in Stursberg and Krogh (2003). The basic idea is to construct an oriented rectangular hull of the reach set for every time step, whose orientation is determined by the singular value decomposition of the sample covariance matrix for the states reachable from the vertices of the initial polytope. The limitation of CheckMate and the method of oriented rectangles is that only autonomous (i.e. uncontrolled) systems, or systems with fixed input are allowed, and only an external approximation of the reach set is provided.

All the methods described so far employ the notion of time step, and calculate the reach set or its approximation at each time step. This approach can be used only with discrete-time systems. By contrast, the analytic methods which we are about to discuss, provide a formula or differential equation describing the (continuous) time evolution of the reach set or its approximation.

The level set method Mitchell and Tomlin (2000; "Level Set Toolbox Homepage") deals with general nonlinear controlled systems and gives exact representation of their reach sets, but requires solving the HJB equation and finding the set of states that belong to sub-zero level set of the value function. The method ("Level Set Toolbox Homepage") is impractical for systems of dimension higher than three.

Requiem ("Requiem Homepage") is a Mathematica notebook which, given a linear system, the set of initial conditions and control bounds, symbolically computes the exact reach set, using the experimental quantifier elimination package. Quantifier elimination is the removal of all quantifiers (the universal quantifier $\forall$ and the existential quantifier $\exists$) from a quantified system. Each quantified formula is substituted with quantifier-free expression with operations $+$, $\times$, $=$ and $<$. For example, consider the discrete-time system

$$x_{k+1} = Ax_k + Bu_k$$

with $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

For initial conditions $x_0 \in \{x \in \mathbf{R}^2 \mid \|x\|_\infty \leqslant 1\}$ and controls $u_k \in \{u \in \mathbf{R} \mid -1 \leqslant u \leqslant 1\}$, the reach set for $k \geqslant 0$ is given by the quantified formula

$$\{x \in \mathbf{R}^2 \mid \exists x_0,\ \ \exists k \geqslant 0,\ \ \exists u_i,\ 0 \leqslant i \leqslant k :\ \ x = A^k x_0 + \sum_{i=0}^{k-1} A^{k-i-1} B u_i\},$$

which is equivalent to the quantifier-free expression

$$-1 \leqslant [1\ \ 0]x \leqslant 1\ \wedge\ -1 \leqslant [0\ \ 1]x \leqslant 1.$$

It is proved in Lafferriere, Pappas, and Yovine (2001) that for continuous-time systems, $\dot{x}(t) = Ax(t) + Bu(t)$, if A is constant and nilpotent or is diagonalizable with rational real or purely imaginary eigenvalues, and with suitable restrictions on the control and initial conditions, the quantifier elimination package returns a quantifier free formula describing the reach set. Quantifier elimination has limited applicability.

The reach set approximation via parallelotopes Kostousova (2001) employs the idea of parametrization described in Kurzhanski and Varaiya (2000) for ellipsoids. The reach set is represented as the intersection of tight external, and the union of tight internal, parallelotopes. The evolution equations for the centers and orientation matrices of both external and internal parallelotopes are provided. This method also finds controls that can drive the system to the boundary points of the reach set, similarly to Varaiya (1998) and Kurzhanski and Varaiya (2000). It works for general linear systems. The computation to solve the evolution equation for tight approximating parallelotopes, however, is more involved than that for ellipsoids, and for discrete-time systems this method does not deal with singular state transition matrices.

Ellipsoidal Toolbox (ET) implements in MATLAB the ellipsoidal calculus Kurzhanski and Vályi (1997) and its application to the reachability analysis of continuous-time Kurzhanski and Varaiya (2000), discrete-time A. A. Kurzhanskiy (2007), possibly time-varying linear systems, and linear systems with disturbances A.B.Kurzhanski and P.Varaiya (2001), for which ET calculates both open-loop and close-loop reach sets. The ellipsoidal calculus provides the following benefits:

- The complexity of the ellipsoidal representation is quadratic in the dimension of the state space, and linear in the number of time steps.
- It is possible to exactly represent the reach set of linear system through both external and internal ellipsoids.
- It is possible to single out individual external and internal approximating ellipsoids that are optimal to some given criterion (e.g. trace, volume, diameter), or combination of such criteria.
- We obtain simple analytical expressions for the control that steers the state to a desired target.

The report is organized as follows. Chapter 2 describes the operations of the ellipsoidal calculus: affine transformation, geometric sum, geometric difference, intersections with hyperplane, ellipsoid, halfspace and polytope, calculation of maximum ellipsoid, calculation of minimum ellipsoid. Chapter 3 presents the reachability problem and ellipsoidal methods for the reach set approximation. Chapter 4 contains *Ellipsoidal Toolbox* installation and quick start instructions, and lists the software packages used by the toolbox. Chapter 5 describes structures and objects implemented and used in toolbox. Also it describes the implementation of methods from chapters 2 and 3 and visualization routines. Chapter 6 describes structures and objects implemented and used in the toolbox. Chapter 6 gives examples of how to use the toolbox. Chapter 7 collects some conclusions and plans for future toolbox development. The functions provided by the toolbox together with their descriptions are listed in appendix A.

CHAPTER

TWO

ELLIPSOIDAL CALCULUS

2.1 Basic Notions

We start with basic definitions. Ellipsoid $\mathcal{E}(q, Q)$ in $\mathbf{R}^n$ with center q and shape matrix Q is the set

$$\mathcal{E}(q, Q) = \{x \in \mathbf{R}^n \mid \langle (x-q), Q^{-1}(x-q)\rangle \leqslant 1\}, \tag{2.1}$$

wherein Q is positive definite ($Q = Q^T$ and $\langle x, Qx\rangle > 0$ for all nonzero $x \in \mathbf{R}^n$). Here $\langle \cdot, \cdot \rangle$ denotes inner product. The support function of a set $\mathcal{X} \subseteq \mathbf{R}^n$ is

$$\rho(l \mid \mathcal{X}) = \sup_{x \in \mathcal{X}} \langle l, x\rangle.$$

In particular, the support function of the ellipsoid (2.1) is

$$\rho(l \mid \mathcal{E}(q, Q)) = \langle l, q\rangle + \langle l, Ql\rangle^{1/2}. \tag{2.2}$$

Although in (2.1) Q is assumed to be positive definite, in practice we may deal with situations when Q is singular, that is, with degenerate ellipsoids flat in those directions for which the corresponding eigenvalues are zero. Therefore, it is useful to give an alternative definition of an ellipsoid using the expression (2.2). Ellipsoid $\mathcal{E}(q, Q)$ in $\mathbf{R}^n$ with center q and shape matrix Q is the set

$$\mathcal{E}(q, Q) = \{x \in \mathbf{R}^n \mid \langle l, x\rangle \leqslant \langle l, q\rangle + \langle l, Ql\rangle^{1/2} \text{ for all } l \in \mathbf{R}^n\}, \tag{2.3}$$

wherein matrix Q is positive semidefinite ($Q = Q^T$ and $\langle x, Qx\rangle \geqslant 0$ for all $x \in \mathbf{R}^n$). The volume of ellipsoid $\mathcal{E}(q, Q)$ is

$$\mathbf{Vol}(E(q, Q)) = \mathbf{Vol}_{\langle x,x\rangle \leqslant 1} \sqrt{\det Q}, \tag{2.4}$$

where $\mathbf{Vol}_{\langle x,x\rangle \leqslant 1}$ is the volume of the unit ball in $\mathbf{R}^n$:

$$\mathbf{Vol}_{\langle x,x\rangle \leqslant 1} = \begin{cases} \frac{\pi^{n/2}}{(n/2)!}, & \text{for even } n, \\ \frac{2^n \pi^{(n-1)/2}((n-1)/2)!}{n!}, & \text{for odd } n. \end{cases} \tag{2.5}$$

The distance from $\mathcal{E}(q, Q)$ to the fixed point a is

$$\mathbf{dist}(\mathcal{E}(q, Q), a) = \max_{\langle l,l\rangle = 1} (\langle l, a\rangle - \rho(l \mid \mathcal{E}(q, Q))) = \max_{\langle l,l\rangle = 1} \left(\langle l, a\rangle - \langle l, q\rangle - \langle l, Ql\rangle^{1/2}\right). \tag{2.6}$$

If $\mathbf{dist}(\mathcal{E}(q, Q), a) > 0$, a lies outside $\mathcal{E}(q, Q)$; if $\mathbf{dist}(\mathcal{E}(q, Q), a) = 0$, a is a boundary point of $\mathcal{E}(q, Q)$; if $\mathbf{dist}(\mathcal{E}(q, Q), a) < 0$, a is an internal point of $\mathcal{E}(q, Q)$.

Given two ellipsoids, $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$, the distance between them is

$$\begin{aligned}
\mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) = & \max_{\langle l,l\rangle=1}\left(-\rho(-l\mid\mathcal{E}(q_1,Q_1))-\rho(l\mid\mathcal{E}(q_2,Q_2))\right) \\
= & \max_{\langle l,l\rangle=1}\left(\langle l,q_1\rangle-\langle l,Q_1l\rangle^{1/2}-\langle l,q_2\rangle-\langle l,Q_2l\rangle^{1/2}\right).
\end{aligned} \tag{2.7}$$

If $\mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) > 0$, the ellipsoids have no common points; if $\mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) = 0$, the ellipsoids have one common point - they touch; if $\mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) < 0$, the ellipsoids intersect.

Finding $\mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2))$ using QCQP is

$$d(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) = \min\langle(x-y),(x-y)\rangle$$

subject to:

$$\begin{aligned}
\langle(q_1-x),Q_1^{-1}(q_1-x)\rangle \leqslant & \quad 1, \\
\langle(q_2-x),Q_2^{-1}(q_2-y)\rangle \leqslant & \quad 1,
\end{aligned}$$

where

$$d(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) = \begin{cases} \mathbf{dist}^2(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) & \text{if } \mathbf{dist}(\mathcal{E}(q_1,Q_1),\mathcal{E}(q_2,Q_2)) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Checking if k nondegenerate ellipsoids $\mathcal{E}(q_1,Q_1),\cdots,\mathcal{E}(q_k,Q_k)$ have nonempty intersection, can be cast as a quadratically constrained quadratic programming (QCQP) problem:

$$\min 0$$

subject to:

$$\langle(x-q_i),Q_i^{-1}(x-q_i)\rangle - 1 \leqslant 0, \quad i=1,\cdots,k.$$

If this problem is feasible, the intersection is nonempty. Given compact convex set $\mathcal{X} \subseteq \mathbf{R}^n$, its polar set, denoted $\mathcal{X}^\circ$, is

$$\mathcal{X}^\circ = \{x\in\mathbf{R}^n \mid \langle x,y\rangle \leqslant 1,\ y\in\mathcal{X}\},$$

or, equivalently,

$$\mathcal{X}^\circ = \{l\in\mathbf{R}^n \mid \rho(l\mid\mathcal{X}) \leqslant 1\}.$$

The properties of the polar set are

- If $\mathcal{X}$ contains the origin, $(\mathcal{X}^\circ)^\circ = \mathcal{X}$;
- If $\mathcal{X}_1 \subseteq \mathcal{X}_2$, $\mathcal{X}_2^\circ \subseteq \mathcal{X}_1^\circ$;
- For any nonsingular matrix $A \in \mathbf{R}^{n\times n}$, $(A\mathcal{X})^\circ = (A^T)^{-1}\mathcal{X}^\circ$.

If a nondegenerate ellipsoid $\mathcal{E}(q,Q)$ contains the origin, its polar set is also an ellipsoid:

$$\begin{aligned}
\mathcal{E}^\circ(q,Q) = & \{l\in\mathbf{R}^n \mid \langle l,q\rangle + \langle l,Ql\rangle^{1/2} \leqslant 1\} \\
= & \{l\in\mathbf{R}^n \mid \langle l,(Q-qq^T)^{-1}l\rangle + 2\langle l,q\rangle \leqslant 1\} \\
= & \{l\in\mathbf{R}^n \mid \langle(l+(Q-qq^T)^{-1}q),(Q-qq^T)(l+(Q-qq^T)^{-1}q)\rangle \leqslant 1+\langle q,(Q-qq^T)^{-1}q\rangle\}.
\end{aligned}$$

The special case is

$$\mathcal{E}^\circ(0, Q) = \mathcal{E}(0, Q^{-1}).$$

Given k compact sets $\mathcal{X}_1, \cdots, \mathcal{X}_k \subseteq \mathbf{R}^n$, their geometric (Minkowski) sum is

$$\mathcal{X}_1 \oplus \cdots \oplus \mathcal{X}_k = \bigcup_{x_1 \in \mathcal{X}_1} \cdots \bigcup_{x_k \in \mathcal{X}_k} \{x_1 + \cdots + x_k\}. \tag{2.8}$$

Given two compact sets $\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathbf{R}^n$, their geometric (Minkowski) difference is

$$\mathcal{X}_1 \dot{-} \mathcal{X}_2 = \{x \in \mathbf{R}^n \mid x + \mathcal{X}_2 \subseteq \mathcal{X}_1\}. \tag{2.9}$$

Ellipsoidal calculus concerns the following set of operations:

- affine transformation of ellipsoid;
- geometric sum of finite number of ellipsoids;
- geometric difference of two ellipsoids;
- intersection of finite number of ellipsoids.

These operations occur in reachability calculation and verification of piecewise affine dynamical systems. The result of all of these operations, except for the affine transformation, is *not* generally an ellipsoid but some convex set, for which we can compute external and internal ellipsoidal approximations.

Additional operations implemented in the *Ellipsoidal Toolbox* include external and internal approximations of intersections of ellipsoids with hyperplanes, halfspaces and polytopes. Hyperplane $H(c, \gamma)$ in $\mathbf{R}^n$ is the set

$$H = \{x \in \mathbf{R}^n \mid \langle c, x\rangle = \gamma\} \tag{2.10}$$

with $c \in \mathbf{R}^n$ and $\gamma \in \mathbf{R}$ fixed. The distance from ellipsoid $\mathcal{E}(q, Q)$ to hyperplane $H(c, \gamma)$ is

$$\mathbf{dist}(\mathcal{E}(q, Q), H(c, \gamma)) = \frac{|\gamma - \langle c, q\rangle| - \langle c, Qc\rangle^{1/2}}{\langle c, c\rangle^{1/2}}. \tag{2.11}$$

If $\mathbf{dist}(\mathcal{E}(q, Q), H(c, \gamma)) > 0$, the ellipsoid and the hyperplane do not intersect; if $\mathbf{dist}(\mathcal{E}(q, Q), H(c, \gamma)) = 0$, the hyperplane is a supporting hyperplane for the ellipsoid; if $\mathbf{dist}(\mathcal{E}(q, Q), H(c, \gamma)) < 0$, the ellipsoid intersects the hyperplane. The intersection of an ellipsoid with a hyperplane is always an ellipsoid and can be computed directly.

Checking if the intersection of k nondegenerate ellipsoids $E(q_1, Q_1), \cdots, \mathcal{E}(q_k, Q_k)$ intersects hyperplane $H(c, \gamma)$, is equivalent to the feasibility check of the QCQP problem:

$$\min 0$$

subject to:

$$\begin{gathered}\langle (x - q_i), Q_i^{-1}(x - q_i)\rangle - 1 \leqslant 0, \quad i = 1, \cdots, k, \\ \langle c, x\rangle - \gamma = 0.\end{gathered}$$

A hyperplane defines two (closed) *halfspaces*:

$$\mathbf{S}_1 = \{x \in \mathbf{R}^n \mid \langle c, x\rangle \leqslant \gamma\} \tag{2.12}$$

and

$$\mathbf{S}_2 = \{x \in \mathbf{R}^n \mid \langle c, x\rangle \geqslant \gamma\}. \tag{2.13}$$

To avoid confusion, however, we shall further assume that a hyperplane $H(c,\gamma)$ specifies the halfspace in the sense (2.12). In order to refer to the other halfspace, the same hyperplane should be defined as $H(-c,-\gamma)$.

The idea behind the calculation of intersection of an ellipsoid with a halfspace is to treat the halfspace as an unbounded ellipsoid, that is, as the ellipsoid with the shape matrix all but one of whose eigenvalues are ∞. Polytope $P(C,g)$ is the intersection of a finite number of closed halfspaces:

$$P = \{x \in \mathbf{R}^n \mid Cx \leqslant g\}, \tag{2.14}$$

wherein $C = [c_1 \ \cdots \ c_m]^T \in \mathbf{R}^{m\times n}$ and $g = [\gamma_1 \ \cdots \ \gamma_m]^T \in \mathbf{R}^m$. The distance from ellipsoid $\mathcal{E}(q,Q)$ to the polytope $P(C,g)$ is

$$\mathbf{dist}(\mathcal{E}(q,Q), P(C,g)) = \min_{y\in P(C,g)} \mathbf{dist}(\mathcal{E}(q,Q), y), \tag{2.15}$$

where $\mathbf{dist}(\mathcal{E}(q,Q),y)$ comes from ([dist:sub:*point*]). If $\mathbf{dist}(\mathcal{E}(q,Q),P(C,g)) > 0$, the ellipsoid and the polytope do not intersect; if $\mathbf{dist}(\mathcal{E}(q,Q),P(C,g)) = 0$, the ellipsoid touches the polytope; if $\mathbf{dist}(\mathcal{E}(q,Q),P(C,g)) < 0$, the ellipsoid intersects the polytope.

Checking if the intersection of k nondegenerate ellipsoids $E(q_1,Q_1),\cdots,\mathcal{E}(q_k,Q_k)$ intersects polytope $P(C,g)$ is equivalent to the feasibility check of the QCQP problem:

$$\min 0$$

subject to:

$$\begin{aligned} \langle (x-q_i), Q_i^{-1}(x-q_i)\rangle - 1 \leqslant 0, \quad & i = 1,\cdots,k, \\ \langle c_j, x\rangle - \gamma_j \leqslant 0, \quad & j = 1,\cdots,m. \end{aligned}$$

2.2 Operations with Ellipsoids

2.2.1 Affine Transformation

The simplest operation with ellipsoids is an affine transformation. Let ellipsoid $\mathcal{E}(q,Q) \subseteq \mathbf{R}^n$, matrix $A \in \mathbf{R}^{m\times n}$ and vector $b \in \mathbf{R}^m$. Then

$$A\mathcal{E}(q,Q) + b = \mathcal{E}(Aq+b, AQA^T). \tag{2.16}$$

Thus, ellipsoids are preserved under affine transformation. If the rows of A are linearly independent (which implies $m \leqslant n$), and $b = 0$, the affine transformation is called *projection*.

2.2.2 Geometric Sum

Consider the geometric sum (2.8) in which $\mathcal{X}_1,\cdots,\mathcal{X}_k$ are nondegenerate ellipsoids $\mathcal{E}(q_1,Q_1),\cdots,\mathcal{E}(q_k,Q_k) \subseteq \mathbf{R}^n$. The resulting set is not generally an ellipsoid. However, it can be tightly approximated by the parametrized families of external and internal ellipsoids.

Let parameter l be some nonzero vector in $\mathbf{R}^n$. Then the external approximation $\mathcal{E}(q,Q_l^+)$ and the internal approximation $\mathcal{E}(q,Q_l^-)$ of the sum $\mathcal{E}(q_1,Q_1)\oplus\cdots\oplus\mathcal{E}(q_k,Q_k)$ are *tight* along direction l, i.e.,

$$\mathcal{E}(q,Q_l^-) \subseteq \mathcal{E}(q_1,Q_1)\oplus\cdots\oplus\mathcal{E}(q_k,Q_k) \subseteq \mathcal{E}(q,Q_l^+)$$

and

$$\rho(\pm l \mid \mathcal{E}(q,Q_l^-)) = \rho(\pm l \mid \mathcal{E}(q_1,Q_1)\oplus\cdots\oplus\mathcal{E}(q_k,Q_k)) = \rho(\pm l \mid \mathcal{E}(q,Q_l^+)).$$

Here the center q is

$$q = q_1 + \cdots + q_k, \tag{2.17}$$

the shape matrix of the external ellipsoid Q_l^+ is

$$Q_l^+ = \left(\langle l, Q_1 l\rangle^{1/2} + \cdots + \langle l, Q_k l\rangle^{1/2}\right)\left(\frac{1}{\langle l, Q_1 l\rangle^{1/2}}Q_1 + \cdots + \frac{1}{\langle l, Q_k l\rangle^{1/2}}Q_k\right), \tag{2.18}$$

and the shape matrix of the internal ellipsoid Q_l^- is

$$Q_l^- = \left(Q_1^{1/2} + S_2 Q_2^{1/2} + \cdots + S_k Q_k^{1/2}\right)^T \left(Q_1^{1/2} + S_2 Q_2^{1/2} + \cdots + S_k Q_k^{1/2}\right), \tag{2.19}$$

with matrices S_i, $i = 2, \cdots, k$, being orthogonal ($S_i S_i^T = I$) and such that vectors $Q_1^{1/2}l, S_2 Q_2^{1/2}l, \cdots, S_k Q_k^{1/2}l$ are parallel.

Varying vector l we get exact external and internal approximations,

$$\bigcup_{\langle l,l\rangle=1} \mathcal{E}(q, Q_l^-) = \mathcal{E}(q_1, Q_1) \oplus \cdots \oplus \mathcal{E}(q_k, Q_k) = \bigcap_{\langle l,l\rangle=1} \mathcal{E}(q, Q_l^+).$$

For proofs of formulas given in this section, see Kurzhanski and Vályi (1997), Kurzhanski and Varaiya (2000).

One last comment is about how to find orthogonal matrices $S_2, \cdots, S_k$ that align vectors $Q_2^{1/2}l, \cdots, Q_k^{1/2}l$ with $Q_1^{1/2}l$. Let v_1 and v_2 be some unit vectors in $\mathbf{R}^n$. We have to find matrix S such that $Sv_2 \upuparrows v_1$. We suggest explicit formulas for the calculation of this matrix (Dariyn and Kurzhanski (2012)):

$$T = I + Q_1(S - I)Q_1^T, \tag{2.20}$$

$$S = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}, \quad c = \langle \hat{v}_1,\ \hat{v}_2\rangle, \quad s = \sqrt{1 - c^2}, \quad \hat{v}_i = \frac{v_i}{\|v_i\|} \tag{2.21}$$

$$Q_1 = [q_1\, q_2] \in \mathbb{R}^{n\times 2}, \quad q_1 = \hat{v}_1, \quad q_2 = \begin{cases} s^{-1}(\hat{v}_2 - c\hat{v}_1), & s \neq 0 \\ 0, & s = 0. \end{cases} \tag{2.22}$$

2.2.3 Geometric Difference

Consider the geometric difference (2.9) in which the sets $\mathcal{X}_1$ and $\mathcal{X}_2$ are nondegenerate ellipsoids $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$. We say that ellipsoid $\mathcal{E}(q_1, Q_1)$ is *bigger* than ellipsoid $\mathcal{E}(q_2, Q_2)$ if

$$\mathcal{E}(0, Q_2) \subseteq \mathcal{E}(0, Q_1).$$

If this condition is not fulfilled, the geometric difference $\mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2)$ is an empty set:

$$\mathcal{E}(0, Q_2) \not\subseteq \mathcal{E}(0, Q_1) \quad \Rightarrow \quad \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) = \emptyset.$$

If $\mathcal{E}(q_1, Q_1)$ is bigger than $\mathcal{E}(q_2, Q_2)$ and $\mathcal{E}(q_2, Q_2)$ is bigger than $\mathcal{E}(q_1, Q_1)$, in other words, if $Q_1 = Q_2$,

$$\mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) = \{q_1 - q_2\} \quad \text{and} \quad \mathcal{E}(q_2, Q_2)\dot{-}\mathcal{E}(q_1, Q_1) = \{q_2 - q_1\}.$$

To check if ellipsoid $\mathcal{E}(q_1, Q_1)$ is bigger than ellipsoid $\mathcal{E}(q_2, Q_2)$, we perform simultaneous diagonalization of matrices Q_1 and Q_2, that is, we find matrix T such that

$$TQ_1T^T = I \quad \text{and} \quad TQ_2T^T = D,$$

where D is some diagonal matrix. Simultaneous diagonalization of Q_1 and Q_2 is possible because both are symmetric positive definite (see Gantmacher (1960)). To find such matrix T, we first do the SVD of Q_1:

$$Q_1 = U_1\Sigma_1V_1^T. \tag{2.23}$$

Then the SVD of matrix $\Sigma_1^{-1/2}U_1^TQ_2U_1\Sigma_1^{-1/2}$:

$$\Sigma_1^{-1/2}U_1^TQ_2U_1\Sigma_1^{-1/2} = U_2\Sigma_2V_2^T. \tag{2.24}$$

Now, T is defined as

$$T = U_2^T\Sigma_1^{-1/2}U_1^T. \tag{2.25}$$

If the biggest diagonal element (eigenvalue) of matrix $D = TQ_2T^T$ is less than or equal to 1, $\mathcal{E}(0, Q_2) \subseteq \mathcal{E}(0, Q_1)$.

Once it is established that ellipsoid $\mathcal{E}(q_1, Q_1)$ is bigger than ellipsoid $\mathcal{E}(q_2, Q_2)$, we know that their geometric difference $\mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2)$ is a nonempty convex compact set. Although it is not generally an ellipsoid, we can find tight external and internal approximations of this set parametrized by vector $l \in \mathbf{R}^n$. Unlike geometric sum, however, ellipsoidal approximations for the geometric difference do not exist for every direction l. Vectors for which the approximations do not exist are called *bad directions*.

Given two ellipsoids $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$ with $\mathcal{E}(0, Q_2) \subseteq \mathcal{E}(0, Q_1)$, l is a bad direction if

$$\frac{\langle l, Q_1l\rangle^{1/2}}{\langle l, Q_2l\rangle^{1/2}} > r,$$

in which r is a minimal root of the equation

$$\mathbf{det}(Q_1 - rQ_2) = 0.$$

To find r, compute matrix T by (2.23)-(2.25) and define

$$r = \frac{1}{\max(\mathbf{diag}(TQ_2T^T))}.$$

If l is *not* a bad direction, we can find tight external and internal ellipsoidal approximations $\mathcal{E}(q, Q_l^+)$ and $\mathcal{E}(q, Q_l^-)$ such that

$$\mathcal{E}(q, Q_l^-) \subseteq \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) \subseteq \mathcal{E}(q, Q_l^+)$$

and

$$\rho(\pm l \mid \mathcal{E}(q, Q_l^-)) = \rho(\pm l \mid \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2)) = \rho(\pm l \mid \mathcal{E}(q, Q_l^+)).$$

The center q is

$$q = q_1 - q_2; \tag{2.26}$$

the shape matrix of the internal ellipsoid Q_l^- is

$$P = \frac{\sqrt{\langle l, Q_1 l\rangle}}{\sqrt{\langle l, Q_2\rangle}};$$
$$Q_l^- = \left(1 - \frac{1}{P}\right) Q_1 + (1-P)\, Q_2.$$

and the shape matrix of the external ellipsoid Q_l^+ is

$$Q_l^+ = \left(Q_1^{1/2} - SQ_2^{1/2}\right)^T \left(Q_1^{1/2} - SQ_2^{1/2}\right). \tag{2.27}$$

Here S is an orthogonal matrix such that vectors $Q_1^{1/2}l$ and $SQ_2^{1/2}l$ are parallel. S is found from (2.20)-(2.22), with $v_1 = Q_2^{1/2}l$ and $v_2 = Q_1^{1/2}l$.

Running l over all unit directions that are not bad, we get

$$\bigcup_{\langle l,l\rangle=1} \mathcal{E}(q, Q_l^-) = \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) = \bigcap_{\langle l,l\rangle=1} \mathcal{E}(q, Q_l^+).$$

For proofs of formulas given in this section, see Kurzhanski and Vályi (1997).

2.2.4 Geometric Difference-Sum

Given ellipsoids $\mathcal{E}(q_1, Q_1)$, $\mathcal{E}(q_2, Q_2)$ and $\mathcal{E}(q_3, Q_3)$, it is possible to compute families of external and internal approximating ellipsoids for

$$\mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) \oplus \mathcal{E}(q_3, Q_3) \tag{2.28}$$

parametrized by direction l, if this set is nonempty ($\mathcal{E}(0, Q_2) \subseteq \mathcal{E}(0, Q_1)$).

First, using the result of the previous section, for any direction l that is not bad, we obtain tight external $\mathcal{E}(q_1 - q_2, Q_l^{0+})$ and internal $\mathcal{E}(q_1 - q_2, Q_l^{0-})$ approximations of the set $\mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2)$.

The second and last step is, using the result of section 2.2.2, to find tight external ellipsoidal approximation $\mathcal{E}(q_1 - q_2 + q_3, Q_l^+)$ of the sum $\mathcal{E}(q_1 - q_2, Q_l^{0+}) \oplus \mathcal{E}(q_3, Q_3)$, and tight internal ellipsoidal approximation $\mathcal{E}(q_1 - q_2 + q_3, Q_l^-)$ for the sum $\mathcal{E}(q_1 - q_2, Q_l^{0-}) \oplus \mathcal{E}(q_3, Q_3)$.

As a result, we get

$$\mathcal{E}(q_1 - q_2 + q_3, Q_l^-) \subseteq \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) \oplus \mathcal{E}(q_3, Q_3) \subseteq \mathcal{E}(q_1 - q_2 + q_3, Q_l^+)$$

and

$$\rho(\pm l \mid \mathcal{E}(q_1 - q_2 + q_3, Q_l^-)) = \rho(\pm l \mid \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) \oplus \mathcal{E}(q_3, Q_3)) = \rho(\pm l \mid \mathcal{E}(q_1 - q_2 + q_3, Q_l^+)).$$

Running l over all unit vectors that are not bad, this translates to

$$\bigcup_{\langle l,l\rangle=1} \mathcal{E}(q_1 - q_2 + q_3, Q_l^-) = \mathcal{E}(q_1, Q_1)\dot{-}\mathcal{E}(q_2, Q_2) \oplus \mathcal{E}(q_3, Q_3) = \bigcap_{\langle l,l\rangle=1} \mathcal{E}(q_1 - q_2 + q_3, Q_l^+).$$

2.2.5 Geometric Sum-Difference

Given ellipsoids $\mathcal{E}(q_1, Q1)$, $\mathcal{E}(q_2, Q_2)$ and $\mathcal{E}(q_3, Q_3)$, it is possible to compute families of external and internal approximating ellipsoids for

$$\mathcal{E}(q_1, Q_1) \oplus \mathcal{E}(q_2, Q_2)\dot{-}\mathcal{E}(q_3, Q_3) \tag{2.29}$$

parametrized by direction l, if this set is nonempty ($\mathcal{E}(0,Q_3)\subseteq\mathcal{E}(0,Q_1)\oplus\mathcal{E}(0,Q_2)$).

First, using the result of section 2.2.2, we obtain tight external $\mathcal{E}(q_1+q_2,Q_l^{0+})$ and internal $\mathcal{E}(q_1+q_2,Q_l^{0-})$ ellipsoidal approximations of the set $\mathcal{E}(q_1,Q_1)\oplus\mathcal{E}(q_2,Q_2)$. In order for the set (2.29) to be nonempty, inclusion $\mathcal{E}(0,Q_3)\subseteq\mathcal{E}(0,Q_l^{0+})$ must be true for any l. Note, however, that even if (2.29) is nonempty, it may be that $\mathcal{E}(0,Q_3)\not\subseteq\mathcal{E}(0,Q_l^{0-})$, then internal approximation for this direction does not exist.

Assuming that (2.29) is nonempty and $\mathcal{E}(0,Q_3)\subseteq\mathcal{E}(0,Q_l^{0-})$, the second step would be, using the results of section 2.2.3, to compute tight external ellipsoidal approximation $\mathcal{E}(q_1+q_2-q_3,Q_l^{+})$ of the difference $\mathcal{E}(q_1+q_2,Q_l^{0+})\dot{-}\mathcal{E}(q_3,Q_3)$, which exists only if l is not bad, and tight internal ellipsoidal approximation $\mathcal{E}(q_1+q_2-q_3,Q_l^{-})$ of the difference $\mathcal{E}(q_1+q_2,Q_l^{0-})\dot{-}\mathcal{E}(q_3,Q_3)$, which exists only if l is not bad for this difference.

If approximation $\mathcal{E}(q_1+q_2-q_3,Q_l^{+})$ exists, then

$$\mathcal{E}(q_1,Q_1)\oplus\mathcal{E}(q_2,Q_2)\dot{-}\mathcal{E}(q_3,Q_3)\subseteq\mathcal{E}(q_1+q_2-q_3,Q_l^{+})$$

and

$$\rho(\pm l\mid\mathcal{E}(q_1,Q_1)\oplus\mathcal{E}(q_2,Q_2)\dot{-}\mathcal{E}(q_3,Q_3))=\rho(\pm l\mid\mathcal{E}(q_1+q_2-q_3,Q_l^{+})).$$

If approximation $\mathcal{E}(q_1+q_2-q_3,Q_l^{-})$ exists, then

$$\mathcal{E}(q_1+q_2-q_3,Q_l^{-})\subseteq\mathcal{E}(q_1,Q_1)\oplus\mathcal{E}(q_2,Q_2)\dot{-}\mathcal{E}(q_3,Q_3)$$

and

$$\rho(\pm l\mid\mathcal{E}(q_1+q_2-q_3,Q_l^{-}))=\rho(\pm l\mid\mathcal{E}(q_1,Q_1)\oplus\mathcal{E}(q_2,Q_2)\dot{-}\mathcal{E}(q_3,Q_3)).$$

For any fixed direction l it may be the case that neither external nor internal tight ellipsoidal approximations exist.

2.2.6 Intersection of Ellipsoid and Hyperplane

Let nondegenerate ellipsoid $\mathcal{E}(q,Q)$ and hyperplane $H(c,\gamma)$ be such that $\mathbf{dist}(\mathcal{E}(q,Q),H(c,\gamma))<0$. In other words,

$$\mathcal{E}_H(w,W)=\mathcal{E}(q,Q)\cap H(c,\gamma)\neq\emptyset.$$

The intersection of ellipsoid with hyperplane, if nonempty, is always an ellipsoid. Here we show how to find it.

First of all, we transform the hyperplane $H(c,\gamma)$ into $H([1\ 0\ \cdots\ 0]^T,0)$ by the affine transformation

$$y=Sx-\frac{\gamma}{\langle c,c\rangle^{1/2}}Sc,$$

where S is an orthogonal matrix found by (2.20)-(2.22) with $v_1=c$ and $v_2=[1\ 0\ \cdots\ 0]^T$. The ellipsoid in the new coordinates becomes $\mathcal{E}(q',Q')$ with

$$\begin{aligned}q' = &\quad q-\frac{\gamma}{\langle c,c\rangle^{1/2}}Sc,\\ Q' = &\qquad SQS^T.\end{aligned}$$

Define matrix $M=Q'^{-1}$; m_{11} is its element in position $(1,1)$, $\bar{m}$ is the first column of M without the first element, and $\bar{M}$ is the submatrix of M obtained by stripping M of its first row and first column:

$$M=\left[\begin{array}{c|c} m_{11} & \bar{m}^T \\ \hline \bar{m} & \bar{M}\end{array}\right].$$

The ellipsoid resulting from the intersection is $\mathcal{E}_H(w', W')$ with

$$\begin{aligned} w' = & \quad q' + q'_1 \begin{bmatrix} -1 \\ \bar{M}^{-1}\bar{m} \end{bmatrix}, \\ W' = & \quad \left(1 - q_1'^2(m_{11} - \langle \bar{m}, \bar{M}^{-1}\bar{m}\rangle)\right) \left[\begin{array}{c|c} 0 & \mathbf{0} \\ \hline \mathbf{0} & \bar{M}^{-1} \end{array}\right], \end{aligned}$$

in which q'_1 represents the first element of vector q'.

Finally, it remains to do the inverse transform of the coordinates to obtain ellipsoid $\mathcal{E}_H(w, W)$:

$$\begin{aligned} w = & \quad S^T w' + \frac{\gamma}{\langle c, c\rangle^{1/2}} c, \\ W = & \quad S^T W' S. \end{aligned}$$

2.2.7 Intersection of Ellipsoid and Ellipsoid

Given two nondegenerate ellipsoids $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$, $\mathbf{dist}(\mathcal{E}(q_1, Q_1), \mathcal{E}(q_2, Q_2)) < 0$ implies that

$$\mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2) \neq \emptyset.$$

This intersection can be approximated by ellipsoids from the outside and from the inside. Trivially, both $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$ are external approximations of this intersection. Here, however, we show how to find the external ellipsoidal approximation of minimal volume.

Define matrices

$$W_1 = Q_1^{-1}, \quad W_2 = Q_2^{-1}.$$

Minimal volume external ellipsoidal approximation $\mathcal{E}(q+, Q^+)$ of the intersection $\mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)$ is determined from the set of equations:

$$Q^+ = \alpha X^{-1}, \tag{2.30}$$

$$X = \pi W_1 + (1 - \pi) W_2, \tag{2.31}$$

$$\alpha = 1 - \pi(1 - \pi)\langle (q_2 - q_1), W_2 X^{-1} W_1 (q_2 - q_1)\rangle, \tag{2.32}$$

$$q^+ = X^{-1}(\pi W_1 q_1 + (1 - \pi) W_2 q_2), \tag{2.33}$$

$$\begin{aligned} 0 = & \alpha(\mathbf{det}(X))^2 \mathbf{trace}(X^{-1}(W_1 - W_2)) - \\ & - n(\mathbf{det}(X))^2 (2\langle q^+, W_1 q_1 - W_2 q_2\rangle + \langle q^+, (W_2 - W_1) q^+\rangle - \\ & - \langle q_1, W_1 q_1\rangle + \langle q_2, W_2 q_2\rangle), \end{aligned} \tag{2.34}$$

with $0 \leqslant \pi \leqslant 1$. We substitute X, α, q^+ defined in (2.31)-(2.33) into (2.34) and get a polynomial of degree $2n-1$ with respect to π, which has only one root in the interval $[0, 1]$, π_0. Then, substituting $\pi = \pi_0$ into (2.30)-(2.33), we obtain q^+ and Q^+. Special cases are $\pi_0 = 1$, whence $\mathcal{E}(q^+, Q^+) = \mathcal{E}(q_1, Q_1)$, and $\pi_0 = 0$, whence $\mathcal{E}(q^+, Q^+) = \mathcal{E}(q_2, Q_2)$. These situations may occur if, for example, one ellipsoid is contained in the other:

$$\mathcal{E}(q_1, Q_1) \subseteq \mathcal{E}(q_2, Q_2) \Rightarrow \pi_0 = 1,$$
$$\mathcal{E}(q_2, Q_2) \subseteq \mathcal{E}(q_1, Q_1) \Rightarrow \pi_0 = 0.$$

The proof that the system of equations (2.30)-(2.34) correctly defines the minimal volume external ellipsoidal approximationi of the intersection $\mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)$ is given in L. Ros (2002).

To find the internal approximating ellipsoid $\mathcal{E}(q^-, Q^-) \subseteq \mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)$, define

$$\beta_1 = \min_{\langle x, W_2 x\rangle = 1} \langle x, W_1 x\rangle, \tag{2.35}$$

$$\beta_2 = \min_{\langle x, W_1 x\rangle = 1} \langle x, W_2 x\rangle, \tag{2.36}$$

Notice that (2.35) and (2.36) are QCQP problems. Parameters β_1 and β_2 are invariant with respect to affine coordinate transformation and describe the position of ellipsoids $\mathcal{E}(q_1, Q_1)$, $\mathcal{E}(q_2, Q_2)$ with respect to each other:

$$\begin{gathered}
\beta_1 \geqslant 1,\ \beta_2 \geqslant 1 \Rightarrow \mathbf{int}(\mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)) = \emptyset, \\
\beta_1 \geqslant 1,\ \beta_2 \leqslant 1 \Rightarrow \mathcal{E}(q_1, Q_1) \subseteq \mathcal{E}(q_2, Q_2), \\
\beta_1 \leqslant 1,\ \beta_2 \geqslant 1 \Rightarrow \mathcal{E}(q_2, Q_2) \subseteq \mathcal{E}(q_1, Q_1), \\
\beta_1 < 1,\ \beta_2 < 1 \Rightarrow \mathbf{int}(\mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)) \neq \emptyset \\
\text{and } \mathcal{E}(q_1, Q_1) \not\subseteq \mathcal{E}(q_2, Q_2) \\
\text{and } \mathcal{E}(q_2, Q_2) \not\subseteq \mathcal{E}(q_1, Q_1).
\end{gathered}$$

Define parametrized family of internal ellipsoids $\mathcal{E}(q^-_{\theta_1\theta_2}, Q^-_{\theta_1\theta_2})$ with

$$q^-_{\theta_1\theta_2} = (\theta_1 W_1 + \theta_2 W_2)^{-1}(\theta_1 W_1 q_1 + \theta_2 W_2 q_2), \tag{2.37}$$

$$Q^-_{\theta_1\theta_2} = (1 - \theta_1\langle q_1, W_1 q_1\rangle - \theta_2\langle q_2, W_2 q_2\rangle + \langle q^-_{\theta_1\theta_2}, (Q^-)^{-1} q^-_{\theta_1\theta_2}\rangle)(\theta_1 W_1 + \theta_2 W_2)^{-1}. \tag{2.38}$$

The best internal ellipsoid $\mathcal{E}(q^-_{\hat\theta_1\hat\theta_2}, Q^-_{\hat\theta_1\hat\theta_2})$ in the class (2.37)-(2.38), namely, such that

$$\mathcal{E}(q^-_{\theta_1\theta_2}, Q^-_{\theta_1\theta_2}) \subseteq \mathcal{E}(q^-_{\hat\theta_1\hat\theta_2}, Q^-_{\hat\theta_1\hat\theta_2}) \subseteq \mathcal{E}(q_1, Q_1) \cap \mathcal{E}(q_2, Q_2)$$

for all $0 \leqslant \theta_1, \theta_2 \leqslant 1$, is specified by the parameters

$$\hat\theta_1 = \frac{1 - \hat\beta_2}{1 - \hat\beta_1\hat\beta_2}, \quad \hat\theta_2 = \frac{1 - \hat\beta_1}{1 - \hat\beta_1\hat\beta_2}, \tag{2.39}$$

with

$$\hat\beta_1 = \min(1, \beta_1), \quad \hat\beta_2 = \min(1, \beta_2).$$

It is the ellipsoid that we look for: $\mathcal{E}(q^-, Q^-) = \mathcal{E}(q^-_{\hat\theta_1\hat\theta_2}, Q^-_{\hat\theta_1\hat\theta_2})$. Two special cases are

$$\hat\theta_1 = 1,\ \hat\theta_2 = 0 \quad \Rightarrow \quad \mathcal{E}(q_1, Q_1) \subseteq \mathcal{E}(q_2, Q_2) \quad \Rightarrow \quad \mathcal{E}(q^-, Q^-) = \mathcal{E}(q_1, Q_1),$$

and

$$\hat\theta_1 = 0,\ \hat\theta_2 = 1 \quad \Rightarrow \quad \mathcal{E}(q_2, Q_2) \subseteq \mathcal{E}(q_1, Q_1) \quad \Rightarrow \quad \mathcal{E}(q^-, Q^-) = \mathcal{E}(q_2, Q_2).$$

The method of finding the internal ellipsoidal approximation of the intersection of two ellipsoids is described in Vazhentsev (1999).

2.2.8 Intersection of Ellipsoid and Halfspace

Finding the intersection of ellipsoid and halfspace can be reduced to finding the intersection of two ellipsoids, one of which is unbounded. Let $\mathcal{E}(q_1, Q_1)$ be a nondegenerate ellipsoid and let $H(c, \gamma)$ define the halfspace

$$\mathbf{S}(c, \gamma) = \{x \in \mathbf{R}^n \mid \langle c, x \rangle \leqslant \gamma\}.$$

We have to determine if the intersection $\mathcal{E}(q_1, Q_1) \cap \mathbf{S}(c, \gamma)$ is empty, and if not, find its external and internal ellipsoidal approximations, $\mathcal{E}(q^+, Q^+)$ and $\mathcal{E}(q^-, Q^-)$. Two trivial situations are:

- $\mathbf{dist}(\mathcal{E}(q_1, Q_1), H(c, \gamma)) > 0$ and $\langle c, q_1 \rangle > 0$, which implies that $\mathcal{E}(q_1, Q_1) \cap \mathbf{S}(c, \gamma) = \emptyset$;
- $\mathbf{dist}(\mathcal{E}(q_1, Q_1), H(c, \gamma)) > 0$ and $\langle c, q_1 \rangle < 0$, so that $\mathcal{E}(q_1, Q_1) \subseteq \mathbf{S}(c, \gamma)$, and then $\mathcal{E}(q^+, Q^+) = \mathcal{E}(q^-, Q^-) = \mathcal{E}(q_1, Q_1)$.

In case $\mathbf{dist}(\mathcal{E}(q_1, Q_1), H(c, \gamma) < 0$, i.e. the ellipsoid intersects the hyperplane,

$$\mathcal{E}(q_1, Q_1) \cap \mathbf{S}(c, \gamma) = \mathcal{E}(q_1, Q_1) \cap \{x \mid \langle (x - q_2), W_2(x - q_2) \rangle \leqslant 1\},$$

with

$$q_2 = (\gamma + 2\sqrt{\overline{\lambda}})c, \tag{2.40}$$

$$W_2 = \frac{1}{4\overline{\lambda}} cc^T, \tag{2.41}$$

$\overline{\lambda}$ being the biggest eigenvalue of matrix Q_1. After defining $W_1 = Q_1^{-1}$, we obtain $\mathcal{E}(q^+, Q^+)$ from equations (2.30)-(2.34), and $\mathcal{E}(q^-, Q^-)$ from (2.37)-(2.38), (2.39).

Remark. Notice that matrix W_2 has rank 1, which makes it singular for $n > 1$. Nevertheless, expressions (2.30)-(2.31), (2.37)-(2.38) make sense because W_1 is nonsingular, $\pi_0 \neq 0$ and $\hat{\theta}_1 \neq 0$.

To find the ellipsoidal approximations $\mathcal{E}(q^+, Q^+)$ and $\mathcal{E}(q^-, Q^-)$ of the intersection of ellipsoid $\mathcal{E}(q, Q)$ and polytope $P(C, g)$, $C \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, such that

$$\mathcal{E}(q^-, Q^-) \subseteq \mathcal{E}(q, Q) \cap P(C, g) \subseteq \mathcal{E}(q^+, Q^+),$$

we first compute

$$\mathcal{E}(q_1^-, Q_1^-) \subseteq \mathcal{E}(q, Q) \cap \mathbf{S}(c_1, \gamma_1) \subseteq \mathcal{E}(q_1^+, Q_1^+),$$

wherein $\mathbf{S}(c_1, \gamma_1)$ is the halfspace defined by the first row of matrix C, c_1, and the first element of vector g, γ_1. Then, one by one, we get

$$\begin{aligned}
\mathcal{E}(q_2^-, Q_2^-) \subseteq \mathcal{E}(q_1^-, Q_1^-) \cap \mathbf{S}(c_2, \gamma_2), \quad \mathcal{E}(q_1^+, Q_1^+) \cap \mathbf{S}(c_2, \gamma_2) \subseteq \mathcal{E}(q_2^+, Q_2^+), \\
\mathcal{E}(q_3^-, Q_3^-) \subseteq \mathcal{E}(q_2^-, Q_2^-) \cap \mathbf{S}(c_3, \gamma_3), \quad \mathcal{E}(q_2^+, Q_2^+) \cap \mathbf{S}(c_3, \gamma_3) \subseteq \mathcal{E}(q_3^+, Q_3^+), \\
\cdots \\
\mathcal{E}(q_m^-, Q_m^-) \subseteq \mathcal{E}(q_{m-1}^-, Q_{m-1}^-) \cap \mathbf{S}(c_m, \gamma_m), \quad \mathcal{E}(q_{m-1}^+, Q_{m-1}^+) \cap \mathbf{S}(c_m, \gamma_m) \subseteq \mathcal{E}(q_m^+, Q_m^+),
\end{aligned}$$

The resulting ellipsoidal approximations are

$$\mathcal{E}(q^+, Q^+) = \mathcal{E}(q_m^+, Q_m^+), \quad \mathcal{E}(q^-, Q^-) = \mathcal{E}(q_m^-, Q_m^-).$$

2.2.9 Checking if one ellipsoid contains another

Theorem of alternatives, also known as S-procedure Boyd and Vandenberghe (2004), states that the implication

$$\langle x, A_1 x\rangle + 2\langle b_1, x\rangle + c_1 \leqslant 0 \quad\Rightarrow\quad \langle x, A_2 x\rangle + 2\langle b_2, x\rangle + c_2 \leqslant 0,$$

where $A_i \in \mathbf{R}^{n\times n}$ are symmetric matrices, $b_i \in \mathbf{R}^n$, $c_i \in \mathbf{R}$, $i = 1, 2$, holds if and only if there exists $\lambda > 0$ such that

$$\left[\begin{array}{cc} A_2 & b_2 \\ b_2^T & c_2 \end{array}\right] \preceq \lambda \left[\begin{array}{cc} A_1 & b_1 \\ b_1^T & c_1 \end{array}\right].$$

By S-procedure, $\mathcal{E}(q_1, Q_1) \subseteq \mathcal{E}(q_2, Q_2)$ (both ellipsoids are assumed to be nondegenerate) if and only if the following SDP problem is feasible:

$$\min 0$$

subject to:

$$\begin{aligned} \lambda > & \quad 0, \\ \left[\begin{array}{cc} Q_2^{-1} & -Q_2^{-1}q_2 \\ (-Q_2^{-1}q_2)^T & q_2^T Q_2^{-1} q_2 - 1 \end{array}\right] \preceq & \quad \lambda \left[\begin{array}{cc} Q_1^{-1} & -Q_1^{-1}q_1 \\ (-Q_1^{-1}q_1)^T & q_1^T Q_1^{-1} q_1 - 1 \end{array}\right] \end{aligned}$$

where $\lambda \in \mathbf{R}$ is the variable.

2.2.10 Minimum Volume Ellipsoids

The minimum volume ellipsoid that contains set S is called *Löwner-John ellipsoid* of the set S. To characterize it we rewrite general ellipsoid $\mathcal{E}(q, Q)$ as

$$\mathcal{E}(q, Q) = \{x \mid \langle (Ax + b), (Ax + b)\rangle\},$$

where

$$A = Q^{-1/2} \quad \text{and} \quad b = -Aq.$$

For positive definite matrix A, the volume of $\mathcal{E}(q, Q)$ is proportional to $\det A^{-1}$. So, finding the minimum volume ellipsoid containing S can be expressed as semidefinite programming (SDP) problem

$$\min \log\det A^{-1}$$

subject to:

$$\sup_{v\in S}\langle (Av + b), (Av + b)\rangle \leqslant 1,$$

where the variables are $A \in \mathbf{R}^{n\times n}$ and $b \in \mathbf{R}^n$, and there is an implicit constraint $A \succ 0$ (A is positive definite). The objective and constraint functions are both convex in A and b, so this problem is convex. Evaluating the constraint function, however, requires solving a convex maximization problem, and is tractable only in certain special cases.

For a finite set $S = \{x_1, \cdots, x_m\} \subset \mathbf{R}^n$, an ellipsoid covers S if and only if it covers its convex hull. So, finding the minimum volume ellipsoid covering S is the same as finding the minimum volume ellipsoid containing the polytope $\mathbf{conv}\{x_1, \cdots, x_m\}$. The SDP problem is

$$\min \log\det A^{-1}$$

subject to:

$$\begin{aligned} A \succ & \quad 0, \\ \langle (Ax_i + b), (Ax_i + b)\rangle \leqslant & \quad 1, \quad i = 1..m. \end{aligned}$$

We can find the minimum volume ellipsoid containing the union of ellipsoids $\bigcup_{i=1}^m \mathcal{E}(q_i, Q_i)$. Using the fact that for $i = 1..m$ $\mathcal{E}(q_i, Q_i) \subseteq \mathcal{E}(q, Q)$ if and only if there exists $\lambda_i > 0$ such that

$$\left[\begin{array}{cc} A^2 - \lambda_i Q_i^{-1} & Ab + \lambda_i Q_i^{-1} q_i \\ (Ab + \lambda_i Q_i^{-1} q_i)^T & b^T b - 1 - \lambda_i (q_i^T Q_i^{-1} q_i - 1) \end{array}\right] \preceq 0.$$

Changing variable $\tilde{b} = Ab$, we get convex SDP in the variables A, $\tilde{b}$, and $\lambda_1, \cdots, \lambda_m$:

$$\min \log\det A^{-1}$$

subject to:

$$\begin{aligned} \lambda_i > & \quad 0, \\ \left[\begin{array}{ccc} A^2 - \lambda_i Q_i^{-1} & \tilde{b} + \lambda_i Q_i^{-1} q_i & 0 \\ (\tilde{b} + \lambda_i Q_i^{-1} q_i)^T & -1 - \lambda_i (q_i^T Q_i^{-1} q_i - 1) & \tilde{b}^T \\ 0 & \tilde{b} & -A^2 \end{array}\right] \preceq & \quad 0, \quad i = 1..m. \end{aligned}$$

After A and b are found,

$$q = -A^{-1} b \quad \text{and} \quad Q = (A^T A)^{-1}.$$

The results on the minimum volume ellipsoids are explained and proven in Boyd and Vandenberghe (2004).

2.2.11 Maximum Volume Ellipsoids

Consider a problem of finding the maximum volume ellipsoid that lies inside a bounded convex set S with nonempty interior. To formulate this problem we rewrite general ellipsoid $\mathcal{E}(q, Q)$ as

$$\mathcal{E}(q, Q) = \{Bx + q \mid \langle x, x\rangle \leqslant 1\},$$

where $B = Q^{1/2}$, so the volume of $\mathcal{E}(q, Q)$ is proportional to $\det B$.

The maximum volume ellipsoid that lies inside S can be found by solving the following SDP problem:

$$\max \log\det B$$

subject to:

$$\sup_{\langle v, v\rangle \leqslant 1} I_S(Bv + q) \leqslant 0,$$

in the variables $B \in \mathbf{R}^{n\times n}$ - symmetric matrix, and $q \in \mathbf{R}^n$, with implicit constraint $B \succ 0$, where I_S is the indicator function:

$$I_S(x) = \begin{cases} 0, & \text{if } x \in S, \\ \infty, & \text{otherwise.} \end{cases}$$

In case of polytope, $S = P(C, g)$ with $P(C, g)$ defined in (2.14), the SDP has the form

$$\min \log\det B^{-1}$$

subject to:

$$\begin{aligned} B \succ & \quad 0, \\ \langle c_i, Bc_i\rangle + \langle c_i, q\rangle \leqslant & \quad \gamma_i, \quad i = 1..m. \end{aligned}$$

We can find the maximum volume ellipsoid that lies inside the intersection of given ellipsoids $\bigcap_{i=1}^m \mathcal{E}(q_i, Q_i)$. Using the fact that for $i = 1..m$ $\mathcal{E}(q, Q) \subseteq \mathcal{E}(q_i, Q_i)$ if and only if there exists $\lambda_i > 0$ such that

$$\left[\begin{array}{cc} -\lambda_i - q^T Q_i^{-1} q + 2q_i^T Q_i^{-1} q - q_i^T Q_i^{-1} q_i + 1 & (Q_i^{-1} q - Q_i^{-1} q_i)^T B \\ B(Q_i^{-1} q - Q_i^{-1} q_i) & \lambda_i I - BQ_i^{-1} B \end{array}\right] \succeq 0.$$

To find the maximum volume ellipsoid, we solve convex SDP in variables B, q, and $\lambda_1, \cdots, \lambda_m$:

$$\min \log\det B^{-1}$$

subject to:

$$\begin{aligned} \lambda_i > & \quad 0, \\ \left[\begin{array}{ccc} 1-\lambda_i & 0 & (q-q_i)^T \\ 0 & \lambda_i I & B \\ q - q_i & B & Q_i \end{array}\right] \succeq & \quad 0, \quad i = 1..m. \end{aligned}$$

After B and q are found,

$$Q = B^T B.$$

The results on the maximum volume ellipsoids are explained and proven in Boyd and Vandenberghe (2004).

CHAPTER

THREE

REACHABILITY

3.1 Basics of Reachability Analysis

3.1.1 Systems without disturbances

Consider a general continuous-time

$$\dot{x}(t) = f(t, x, u), \tag{3.1}$$

or discrete-time dynamical system

$$x(t+1) = f(t, x, u), \tag{3.2}$$

wherein t is time [1], $x \in \mathbf{R}^n$ is the state, $u \in \mathbf{R}^m$ is the control, and f is a measurable vector function taking values in $\mathbf{R}^n$. [2] The control values $u(t, x(t))$ are restricted to a closed compact control set $\mathcal{U}(t) \subset \mathbf{R}^m$. An *open-loop* control does not depend on the state, $u = u(t)$; for a *closed-loop* control, $u = u(t, x(t))$.

The (forward) reach set $\mathcal{X}(t, t_0, x_0)$ at time $t > t_0$ from the initial position (t_0, x_0) is the set of all states $x(t)$ reachable at time t by system (3.1), or (3.2), with $x(t_0) = x_0$ through all possible controls $u(\tau, x(\tau)) \in \mathcal{U}(\tau)$, $t_0 \leqslant \tau < t$. For a given set of initial states $\mathcal{X}_0$, the reach set $\mathcal{X}(t, t_0, \mathcal{X}_0)$ is

$$\mathcal{X}(t, t_0, \mathcal{X}_0) = \bigcup_{x_0 \in \mathcal{X}_0} \mathcal{X}(t, t_0, x_0).$$

Here are two facts about forward reach sets.

1. $\mathcal{X}(t, t_0, \mathcal{X}_0)$ is the same for open-loop and closed-loop control.
2. $\mathcal{X}(t, t_0, \mathcal{X}_0)$ satisfies the semigroup property,

$$\mathcal{X}(t, t_0, \mathcal{X}_0) = \mathcal{X}(t, \tau, \mathcal{X}(\tau, t_0, \mathcal{X}_0)), \quad t_0 \leqslant \tau < t. \tag{3.3}$$

For linear systems

$$f(t, x, u) = A(t)x(t) + B(t)u, \tag{3.4}$$

with matrices $A(t)$ in $\mathbf{R}^{n \times n}$ and $B(t)$ in $\mathbf{R}^{m \times n}$. For continuous-time linear system the state transition matrix is

$$\dot{\Phi}(t, t_0) = A(t)\Phi(t, t_0), \Phi(t, t) = I,$$

[1] In discrete-time case t assumes integer values.

[2] We are being general when giving the basic definitions. However, it is important to understand that for any specific *continuous-time* dynamical system it must be determined whether the solution exists and is unique, and in which class of solutions these conditions are met. Here we shall assume that function f is such that the solution of the differential equation ([ctds1]) exists and is unique in Fillipov sense. This allows the right-hand side to be discontinuous. For discrete-time systems this problem does not exist.

which for constant $A(t) \equiv A$ simplifies as

$$\Phi(t, t_0) = e^{A(t-t_0)}.$$

For discrete-time linear system the state transition matrix is

$$\Phi(t+1, t_0) = A(t)\Phi(t, t_0), \Phi(t, t) = I,$$

which for constant $A(t) \equiv A$ simplifies as

$$\Phi(t, t_0) = A^{t-t_0}.$$

If the state transition matrix is invertible, $\Phi^{-1}(t, t_0) = \Phi(t_0, t)$. The transition matrix is always invertible for continuous-time and for sampled discrete-time systems. However, if for some τ, $t_0 \leqslant \tau < t$, $A(\tau)$ is degenerate (singular), $\Phi(t, t_0) = \prod_{\tau=t_0}^{t-1} A(\tau)$, is also degenerate and cannot be inverted.

Following Cauchy's formula, the reach set $\mathcal{X}(t, t_0, \mathcal{X}_0)$ for a linear system can be expressed as

$$\mathcal{X}(t, t_0, \mathcal{X}_0) = \Phi(t, t_0)\mathcal{X}_0 \oplus \int_{t_0}^{t} \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau \tag{3.5}$$

in continuous-time, and as

$$\mathcal{X}(t, t_0, \mathcal{X}_0) = \Phi(t, t_0)\mathcal{X}_0 \oplus \sum_{\tau=t_0}^{t-1} \Phi(t, \tau+1)B(\tau)\mathcal{U}(\tau) \tag{3.6}$$

in discrete-time case.

The operation '$\oplus$' is the *geometric sum*, also known as *Minkowski sum*. [3] The geometric sum and linear (or affine) transformations preserve compactness and convexity. Hence, if the initial set $\mathcal{X}_0$ and the control sets $\mathcal{U}(\tau)$, $t_0 \leqslant \tau < t$, are compact and convex, so is the reach set $\mathcal{X}(t, t_0, \mathcal{X}_0)$.

The backward reach set $\mathcal{Y}(t_1, t, y_1)$ for the target position (t_1, y_1) is the set of all states $y(t)$ for which there exists some control $u(\tau, x(\tau)) \in \mathcal{U}(\tau)$, $t \leqslant \tau < t_1$, that steers system (3.1), or (3.2) to the state y_1 at time t_1. For the target set $\mathcal{Y}_1$ at time t_1, the backward reach set $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ is

$$\mathcal{Y}(t_1, t, \mathcal{Y}_1) = \bigcup_{y_1 \in \mathcal{Y}_1} \mathcal{Y}(t_1, t, y_1).$$

The backward reach set $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ is the largest *weakly invariant* set with respect to the target set $\mathcal{Y}_1$ and time values t and t_1. [4]

Remark. Backward reach set can be computed for continuous-time system only if the solution of (3.1) exists for $t < t_1$; and for discrete-time system only if the right hand side of (3.2) is invertible [5].

These two facts about the backward reach set $\mathcal{Y}$ are similar to those for forward reach sets.

1. $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ is the same for open-loop and closed-loop control.

[3] Minkowski sum of sets $\mathcal{W}, \mathcal{Z} \subseteq \mathbf{R}^n$ is defined as $\mathcal{W} \oplus \mathcal{Z} = \{w + z \mid w \in \mathcal{W},\ z \in \mathcal{Z}\}$. Set $\mathcal{W} \oplus \mathcal{Z}$ is nonempty if and only if both, $\mathcal{W}$ and $\mathcal{Z}$ are nonempty. If $\mathcal{W}$ and $\mathcal{Z}$ are convex, set $\mathcal{W} \oplus \mathcal{Z}$ is convex.

[4] $\mathcal{M}$ is weakly invariant with respect to the target set $\mathcal{Y}_1$ and times t_0 and t, if for every state $x_0 \in \mathcal{M}$ there exists a control $u(\tau, x(\tau)) \in \mathcal{U}(\tau)$, $t_0 \leqslant \tau < t$, that steers the system from x_0 at time t_0 to some state in $\mathcal{Y}_1$ at time t. If *all* controls in $\mathcal{U}(\tau)$, $t_0 \leqslant \tau < t$ steer the system from every $x_0 \in \mathcal{M}$ at time t_0 to $\mathcal{Y}_1$ at time t, set $\mathcal{M}$ is said to be *strongly* invariant with respect to $\mathcal{Y}_1$, t_0 and t.

[5] There exists $f^{-1}(t, x, u)$ such that $x(t) = f^{-1}(t, x(t+1), u, v)$.

2. $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ satisfies the semigroup property,

$$\mathcal{Y}(t_1, t, \mathcal{Y}_1) = \mathcal{Y}(\tau, t, \mathcal{Y}(t_1, \tau, \mathcal{Y}_1)), \quad t \leqslant \tau < t_1. \tag{3.7}$$

For the linear system (3.4) the backward reach set can be expressed as

$$\mathcal{Y}(t_1, t, \mathcal{Y}_1) = \Phi(t, t_1)\mathcal{Y}_1 \oplus \int_{t_1}^{t} \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau \tag{3.8}$$

in the continuous-time case, and as

$$\mathcal{Y}(t_1, t, \mathcal{Y}_1) = \Phi(t, t_1)\mathcal{Y}_1 \oplus \sum_{\tau=t}^{t_1-1} -\Phi(t, \tau)B(\tau)\mathcal{U}(\tau) \tag{3.9}$$

in discrete-time case. The last formula makes sense only for discrete-time linear systems with invertible state transition matrix. Degenerate discrete-time linear systems have unbounded backward reach sets and such sets cannot be computed with available software tools.

Just as in the case of forward reach set, the backward reach set of a linear system $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ is compact and convex if the target set $\mathcal{Y}_1$ and the control sets $\mathcal{U}(\tau)$, $t \leqslant \tau < t_1$, are compact and convex.

Remark. In the computer science literature the reach set is said to be the result of operator *post*, and the backward reach set is the result of operator *pre*. In the control literature the backward reach set is also called the *solvability set*.

3.1.2 Systems with disturbances

Consider the continuous-time dynamical system with disturbance

$$\dot{x}(t) = f(t, x, u, v), \tag{3.10}$$

or the discrete-time dynamical system with disturbance

$$x(t+1) = f(t, x, u, v), \tag{3.11}$$

in which we also have the disturbance input $v \in \mathbf{R}^d$ with values $v(t)$ restricted to a closed compact set $\mathcal{V}(t) \subset \mathbf{R}^d$.

In the presence of disturbances the open-loop reach set (OLRS) is different from the closed-loop reach set (CLRS).

Given the initial time t_0, the set of initial states $\mathcal{X}_0$, and terminal time t, there are two types of OLRS.

The maxmin open-loop reach set $\overline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0)$ is the set of all states x, such that for any disturbance $v(\tau) \in \mathcal{V}(\tau)$, there exist an initial state $x_0 \in \mathcal{X}_0$ and a control $u(\tau) \in \mathcal{U}(\tau)$, $t_0 \leqslant \tau < t$, that steers system (3.10) or (3.11) from $x(t_0) = x_0$ to $x(t) = x$.

The minmax open-loop reach set $\underline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0)$ is the set of all states x, such that there exists a control $u(\tau) \in \mathcal{U}(\tau)$ that for all disturbances $v(\tau) \in \mathcal{V}(\tau)$, $t_0 \leqslant \tau < t$, assigns an initial state $x_0 \in \mathcal{X}_0$ and steers system (3.10), or (3.11), from $x(t_0) = x_0$ to $x(t) = x$.

In the maxmin case the control is chosen *after* knowing the disturbance over the entire time interval $[t_0, t]$, whereas in the minmax case the control is chosen *before* any knowledge of the disturbance. Consequently, the OLRS do not satisfy the semigroup property.

The terms 'maxmin' and 'minmax' come from the fact that $\overline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0)$ is the subzero level set of the value function

$$\underline{V}(t, x) = \max_v \min_u \{\mathbf{dist}(x(t_0), \mathcal{X}_0) \mid x(t) = x,\ u(\tau) \in \mathcal{U}(\tau),\ v(\tau) \in \mathcal{V}(\tau),\ t_0 \leqslant \tau < t\}, \tag{3.12}$$

i.e., $\overline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0) = \{x \mid \underline{V}(t, x) \leqslant 0\}$, and $\underline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0)$ is the subzero level set of the value function

$$\overline{V}(t, x) = \min_u \max_v \{\mathbf{dist}(x(t_0), \mathcal{X}_0) \mid x(t) = x,\ u(\tau) \in \mathcal{U}(\tau),\ v(\tau) \in \mathcal{V}(\tau),\ t_0 \leqslant \tau < t\}, \tag{3.13}$$

in which $\mathbf{dist}(\cdot,\cdot)$ denotes Hausdorff semidistance. [6] Since $\underline{V}(t,x) \leqslant \overline{V}(t,x)$, $\underline{\mathcal{X}}_{OL}(t,t_0,\mathcal{X}_0) \subseteq \overline{\mathcal{X}}_{OL}(t,t_0,\mathcal{X}_0)$.

Note that maxmin and minmax OLRS imply *guarantees*: these are states that can be reached no matter what the disturbance is, whether it is known in advance (maxmin case) or not (minmax case). The OLRS may be empty.

Fixing time instant τ_1, $t_0 < \tau_1 < t$, define the *piecewise maxmin open-loop reach set with one correction*,

$$\overline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0) = \overline{\mathcal{X}}_{OL}(t,\tau_1,\overline{\mathcal{X}}_{OL}(\tau_1,t_0,\mathcal{X}_0)), \tag{3.14}$$

and the *piecewise minmax open-loop reach set with one correction*,

$$\underline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0) = \underline{\mathcal{X}}_{OL}(t,\tau_1,\underline{\mathcal{X}}_{OL}(\tau_1,t_0,\mathcal{X}_0)). \tag{3.15}$$

The piecewise maxmin OLRS $\overline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0)$ is the subzero level set of the value function

$$\underline{V}^1(t,x) = \max_v \min_u \{\underline{V}(\tau_1,x(\tau_1)) \mid x(t)=x,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ \tau_1\leqslant\tau<t\}, \tag{3.16}$$

with $V(\tau_1,x(\tau_1))$ given by (3.12), which yields

$$\underline{V}^1(t,x) \geqslant \underline{V}(t,x),$$

and thus,

$$\overline{\mathcal{X}}^1_{OL}(t,t_0\mathcal{X}_0) \subseteq \overline{\mathcal{X}}_{OL}(t,t_0,\mathcal{X}_0).$$

On the other hand, the piecewise minmax OLRS $\underline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0)$ is the subzero level set of the value function

$$\overline{V}^1(t,x) = \min_u \max_v \{\overline{V}(\tau_1,x(\tau_1)) \mid x(t)=x,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ \tau_1\leqslant\tau<t\}, \tag{3.17}$$

with $V(\tau_1,x(\tau_1))$ given by (3.13), which yields

$$\overline{V}(t,x) \geqslant \overline{V}^1(t,x),$$

and thus,

$$\underline{\mathcal{X}}_{OL}(t,t_0\mathcal{X}_0) \subseteq \underline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0).$$

We can now recursively define piecewise maxmin and minmax OLRS with k corrections for $t_0 < \tau_1 < \cdots < \tau_k < t$. The maxmin piecewise OLRS with k corrections is

$$\overline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0) = \overline{\mathcal{X}}_{OL}(t,\tau_k,\overline{\mathcal{X}}^{k-1}_{OL}(\tau_k,t_0,\mathcal{X}_0)), \tag{3.18}$$

which is the subzero level set of the corresponding value function

$$\underline{V}^k(t,x) = \max_v \min_u \{\underline{V}^{k-1}(\tau_k,x(\tau_k)) \mid x(t)=x,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ \tau_k\leqslant\tau<t\}.$$

[6] Hausdorff semidistance between compact sets $\mathcal{W},\mathcal{Z}\subseteq\mathbf{R}^n$ is defined as

$$\mathbf{dist}(\mathcal{W},\mathcal{Z}) = \min\{\langle w-z,w-z\rangle^{1/2} \mid w\in\mathcal{W},\ z\in\mathcal{Z}\},$$

where $\langle\cdot,\cdot\rangle$ denotes inner product.

The minmax piecewise OLRS with k corrections is

$$\underline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0)=\underline{\mathcal{X}}_{OL}(t,\tau_k,\underline{\mathcal{X}}^{k-1}_{OL}(\tau_k,t_0,\mathcal{X}_0)), \tag{3.19}$$

which is the subzero level set of the corresponding value function

$$\overline{V}^k(t,x)=\min_u\max_v\{\overline{V}^{k-1}(\tau_k,x(\tau_k))\mid x(t)=x,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ \tau_k\leqslant\tau<t\}.$$

From (3.16), (3.17), (3.1.2) and (3.1.2) it follows that

$$\underline{V}(t,x)\leqslant\underline{V}^1(t,x)\leqslant\cdots\leqslant\underline{V}^k(t,x)\leqslant\overline{V}^k(t,x)\leqslant\cdots\leqslant\overline{V}^1(t,x)\leqslant\overline{V}(t,x).$$

Hence,

$$\underline{\mathcal{X}}_{OL}(t,t_0,\mathcal{X}_0)\subseteq\underline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0)\subseteq\cdots\subseteq\underline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0)\subseteq$$
$$\overline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0)\subseteq\cdots\subseteq\overline{\mathcal{X}}^1_{OL}(t,t_0,\mathcal{X}_0)\subseteq\overline{\mathcal{X}}_{OL}(t,t_0,\mathcal{X}_0).$$

We call

$$\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)=\overline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0),\quad k=\begin{cases}\infty & \text{for continuous-time system}\\ t-t_0-1 & \text{for discrete-time system}\end{cases} \tag{3.20}$$

the *maxmin closed-loop reach set* of system (3.10) or (3.11) at time t, and we call

$$\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)=\underline{\mathcal{X}}^k_{OL}(t,t_0,\mathcal{X}_0),\quad k=\begin{cases}\infty & \text{for continuous-time system}\\ t-t_0-1 & \text{for discrete-time system}\end{cases} \tag{3.21}$$

the *minmax closed-loop reach set* of system (3.10) or (3.11) at time t. Given initial time t_0 and the set of initial states $\mathcal{X}_0$, the maxmin CLRS $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)$ of system (3.10) or (3.11) at time $t>t_0$, is the set of all states x, for each of which and for every disturbance $v(\tau)\in\mathcal{V}(\tau)$, there exist an initial state $x_0\in\mathcal{X}_0$ and a control $u(\tau,x(\tau))\in\mathcal{U}(\tau)$, such that the trajectory $x(\tau|v(\tau),u(\tau,x(\tau)))$ satisfying $x(t_0)=x_0$ and

$$\dot{x}(\tau|v(\tau),u(\tau,x(\tau)))\in f(\tau,x(\tau),u(\tau,x(\tau)),v(\tau))$$

in the continuous-time case, or

$$x(\tau+1|v(\tau),u(\tau,x(\tau)))\in f(\tau,x(\tau),u(\tau,x(\tau)),v(\tau))$$

in the discrete-time case, with $t_0\leqslant\tau<t$, is such that $x(t)=x$. Given initial time t_0 and the set of initial states $\mathcal{X}_0$, the maxmin CLRS $\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)$ of system (3.10) or (3.11), at time $t>t_0$, is the set of all states x, for each of which there exists a control $u(\tau,x(\tau))\in\mathcal{U}(\tau)$, and for every disturbance $v(\tau)\in\mathcal{V}(\tau)$ there exists an initial state $x_0\in\mathcal{X}_0$, such that the trajectory $x(\tau,v(\tau)|u(\tau,x(\tau)))$ satisfying $x(t_0)=x_0$ and

$$\dot{x}(\tau,v(\tau)|u(\tau,x(\tau)))\in f(\tau,x(\tau),u(\tau,x(\tau)),v(\tau))$$

in the continuous-time case, or

$$x(\tau+1,v(\tau)|u(\tau,x(\tau)))\in f(\tau,x(\tau),u(\tau,x(\tau)),v(\tau))$$

in the discrete-time case, with $t_0\leqslant\tau<t$, is such that $x(t)=x$. By construction, both maxmin and minmax CLRS satisfy the semigroup property (3.3).

For some classes of dynamical systems and some types of constraints on initial conditions, controls and disturbances, the maxmin and minmax CLRS may coincide. This is the case for continuous-time linear systems with convex compact bounds on the initial set, controls and disturbances under the condition that the initial set $\mathcal{X}_0$ is large enough to ensure that $\mathcal{X}(t_0+\epsilon,t_0,\mathcal{X}_0)$ is nonempty for some small $\epsilon>0$.

Consider the linear system case,

$$f(t, x, u) = A(t)x(t) + B(t)u + G(t)v, \tag{3.22}$$

where $A(t)$ and $B(t)$ are as in (3.4), and $G(t)$ takes its values in $\mathbf{R}^d$.

The maxmin OLRS for the continuous-time linear system can be expressed through set valued integrals,

$$\begin{aligned}&\overline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, t_0)\mathcal{X}_0 \oplus \int_{t_0}^t \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau\right) \dot{-} \\ &\int_{t_0}^t \Phi(t, \tau)(-G(\tau))\mathcal{V}(\tau)d\tau,\end{aligned} \tag{3.23}$$

and for discrete-time linear system through set-valued sums,

$$\begin{aligned}&\overline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, t_0)\mathcal{X}_0 \oplus \sum_{\tau=t_0}^{t-1} \Phi(t, \tau+1)B(\tau)\mathcal{U}(\tau)\right) \dot{-} \\ &\sum_{\tau=t_0}^{t-1} \Phi(t, \tau+1)(-G(\tau))\mathcal{V}(\tau).\end{aligned} \tag{3.24}$$

Similarly, the minmax OLRS for the continuous-time linear system is

$$\begin{aligned}&\underline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, t_0)\mathcal{X}_0 \dot{-} \int_{t_0}^t \Phi(t, \tau)(-G(\tau))\mathcal{V}(\tau)d\tau\right) \oplus \\ &\int_{t_0}^t \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau,\end{aligned} \tag{3.25}$$

and for the discrete-time linear system it is

$$\begin{aligned}&\underline{\mathcal{X}}_{OL}(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, t_0)\mathcal{X}_0 \dot{-} \sum_{\tau=t_0}^{t-1} \Phi(t, \tau+1)(-G(\tau))\mathcal{V}(\tau)\right) \oplus \\ &\sum_{\tau=t_0}^{t-1} \Phi(t, \tau+1)B(\tau)\mathcal{U}(\tau).\end{aligned} \tag{3.26}$$

The operation '$\dot{-}$' is *geometric difference*, also known as *Minkowski difference*. [7]

Now consider the piecewise OLRS with k corrections. Expression (3.18) translates into

$$: label : ctlsmaxmink$$

$$\begin{aligned}&\overline{\mathcal{X}}_{OL}^k(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, \tau_k)\overline{\mathcal{X}}_{OL}^{k-1}(\tau_k, t_0, \mathcal{X}_0) \oplus \int_{\tau_k}^t \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau\right) \dot{-} \\ &\int_{\tau_k}^t \Phi(t, \tau)(-G(\tau))\mathcal{V}(\tau)d\tau,\end{aligned}$$

in the continuous-time case, and for the discrete-time case into

$$\begin{aligned}&\overline{\mathcal{X}}_{OL}^k(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, \tau_k)\overline{\mathcal{X}}_{OL}^{k-1}(\tau_k, t_0, \mathcal{X}_0) \oplus \sum_{\tau=\tau_k}^{t-1} \Phi(t, \tau+1)B(\tau)\mathcal{U}(\tau)\right) \dot{-} \\ &\sum_{\tau=\tau_k}^{t-1} \Phi(t, \tau+1)(-G(\tau))\mathcal{V}(\tau).\end{aligned} \tag{3.27}$$

Expression (3.19) translates into

$$\begin{aligned}&\underline{\mathcal{X}}_{OL}^k(t, t_0, \mathcal{X}_0) = \\ &\left(\Phi(t, \tau_k)\underline{\mathcal{X}}_{OL}^{k-1}(t, t_0, \mathcal{X}_0) \dot{-} \int_{\tau_k}^t \Phi(t, \tau)(-G(\tau))\mathcal{V}(\tau)d\tau\right) \oplus \\ &\int_{\tau_k}^t \Phi(t, \tau)B(\tau)\mathcal{U}(\tau)d\tau,\end{aligned} \tag{3.28}$$

[7] The Minkowski difference of sets $\mathcal{W}, \mathcal{Z} \in \mathbf{R}^n$ is defined as $\mathcal{W} \dot{-} \mathcal{Z} = \{\xi \in \mathbf{R}^n \mid \xi \oplus \mathcal{Z} \subseteq \mathcal{W}\}$. If $\mathcal{W}$ and $\mathcal{Z}$ are convex, $\mathcal{W} \dot{-} \mathcal{Z}$ is convex if it is nonempty.

in the continuous-time case, and for the discrete-time case into

$$\begin{aligned}&\underline{\mathcal{X}}_{OL}^{k}(t,t_0,\mathcal{X}_0)=\\&\Big(\Phi(t,\tau_k)\underline{\mathcal{X}}_{OL}^{k-1}(\tau_k,t_0,\mathcal{X}_0)\dot{-}\textstyle\sum_{\tau=\tau_k}^{t-1}\Phi(t,\tau+1)(-G(\tau))\mathcal{V}(\tau)\Big)\oplus\\&\textstyle\sum_{\tau=\tau_k}^{t-1}\Phi(t,\tau+1)B(\tau)\mathcal{U}(\tau).\end{aligned} \tag{3.29}$$

Since for any $\mathcal{W}_1,\mathcal{W}_2,\mathcal{W}_3\subseteq\mathbf{R}^n$ it is true that

$$(\mathcal{W}_1\dot{-}\mathcal{W}_2)\oplus\mathcal{W}_3=(\mathcal{W}_1\oplus\mathcal{W}_3)\dot{-}(\mathcal{W}_2\oplus\mathcal{W}_3)\subseteq(\mathcal{W}_1\oplus\mathcal{W}_3)\dot{-}\mathcal{W}_2,$$

from (**??**), (3.28) and from (3.27), (3.29), it is clear that (3.1.2) is true. For linear systems, if the initial set $\mathcal{X}_0$, control bounds $\mathcal{U}(\tau)$ and disturbance bounds $\mathcal{V}(\tau)$, $t_0\leqslant\tau<t$, are compact and convex, the CLRS $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)$ and $\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)$ are compact and convex, provided they are nonempty. For continuous-time linear systems, $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)=\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{X}_0)=\mathcal{X}_{CL}(t,t_0,\mathcal{X}_0)$.

Just as for forward reach sets, the backward reach sets can be open-loop (OLBRS) or closed-loop (CLBRS).

Given the terminal time t_1 and target set $\mathcal{Y}_1$, the maxmin open-loop backward reach set $\overline{\mathcal{Y}}_{OL}(t_1,t,\mathcal{Y}_1)$ of system (3.10) or (3.11) at time $t<t_1$, is the set of all y, such that for any disturbance $v(\tau)\in\mathcal{V}(\tau)$ there exists a terminal state $y_1\in\mathcal{Y}_1$ and control $u(\tau)\in\mathcal{U}(\tau)$, $t\leqslant\tau<t_1$, which steers the system from $y(t)=y$ to $y(t_1)=y_1$.

$\overline{\mathcal{Y}}_{OL}(t_1,t,\mathcal{Y}_1)$ is the subzero level set of the value function

$$\begin{aligned}&\underline{V}_b(t,y)=\\&\max_v\min_u\{\mathbf{dist}(y(t_1),\mathcal{Y}_1)\mid y(t)=y,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ t\leqslant\tau<t_1\},\end{aligned}$$

Given the terminal time t_1 and target set $\mathcal{Y}_1$, the minmax open-loop backward reach set $\underline{\mathcal{Y}}_{OL}(t_1,t,\mathcal{Y}_1)$ of system (3.10) or (3.11) at time $t<t_1$, is the set of all y, such that there exists a control $u(\tau)\in\mathcal{U}(\tau)$ that for all disturbances $v(\tau\in\mathcal{V}(\tau)$, $t\leqslant\tau<t_1$, assigns a terminal state $y_1\in\mathcal{Y}_1$ and steers the system from $y(t)=y$ to $y(t_1)=y_1$. $\underline{\mathcal{Y}}_{OL}(t_1,t,\mathcal{Y}_1)$ is the subzero level set of the value function

$$\begin{aligned}&\overline{V}_b(t,y)=\\&\min_u\max_v\{\mathbf{dist}(y(t_1),\mathcal{Y}_1)\mid y(t)=y,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ t\leqslant\tau<t_1\},\end{aligned}$$

Remark. The backward reach set can be computed for a continuous-time system only if the solution of (3.10) exists for $t<t_1$, and for a discrete-time system only if the right hand side of (3.11) is invertible.

Similarly to the forward reachability case, we construct piecewise OLBRS with one correction at time τ_1, $t<\tau_1<t_1$. The piecewise maxmin OLBRS with one correction is

$$\overline{\mathcal{Y}}_{OL}^{1}(t_1,t,\mathcal{Y}_1)=\overline{\mathcal{Y}}_{OL}(\tau_1,t,\overline{\mathcal{Y}}_{OL}(t_1,\tau_1,\mathcal{Y}_1)), \tag{3.30}$$

and it is the subzero level set of the function

$$\begin{aligned}&\underline{V}_b^1(t,y)=\\&\max_v\min_u\{\underline{V}_b(\tau_1,y(\tau_1))\mid y(t)=y,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ t\leqslant\tau<\tau_1\}.\end{aligned}$$

The piecewise minmax OLBRS with one correction is

$$\underline{\mathcal{Y}}_{OL}^{1}(t_1,t,\mathcal{Y}_1)=\underline{\mathcal{Y}}_{OL}(\tau_1,t,\underline{\mathcal{Y}}_{OL}(t_1,\tau_1,\mathcal{Y}_1)), \tag{3.31}$$

and it is the subzero level set of the function

$$\begin{aligned}&\overline{V}_b^1(t,y)=\\&\min_u\max_v\{\overline{V}_b(\tau_1,y(\tau_1))\mid y(t)=y,\ u(\tau)\in\mathcal{U}(\tau),\ v(\tau)\in\mathcal{V}(\tau),\ t\leqslant\tau<\tau_1\},\end{aligned}$$

Recursively define maxmin and minmax OLBRS with k corrections for $t < \tau_k < \cdots < \tau_1 < t_1$. The maxmin OLBRS with k corrections is

$$\overline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1) = \overline{\mathcal{Y}}_{OL}(\tau_k, t, \overline{\mathcal{Y}}^{k-1}_{OL}(t_1, \tau_k, \mathcal{Y}_1)), \tag{3.32}$$

which is the subzero level set of function

$$\underline{V}^k_b(t, y) = \max_v \min_u \{\underline{V}^{k-1}_b(\tau_k, y(\tau_k)) \mid y(t) = y,\ u(\tau) \in \mathcal{U}(\tau),\ v(\tau) \in \mathcal{V}(\tau),\ t \leqslant \tau < \tau_k\}.$$

The minmax OLBRS with k corrections is

$$\underline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1) = \underline{\mathcal{Y}}_{OL}(\tau_k, t, \underline{\mathcal{Y}}^{k-1}_{OL}(t_1, \tau_k, \mathcal{Y}_1)), \tag{3.33}$$

which is the subzero level set of the function

$$\overline{V}^k_b(t, y) = \min_u \max_v \{\overline{V}^{k-1}_b(\tau_k, y(\tau_k)) \mid y(t) = y,\ u(\tau) \in \mathcal{U}(\tau),\ v(\tau) \in \mathcal{V}(\tau),\ t \leqslant \tau < \tau_k\},$$

From (3.1.2), (3.1.2), (3.1.2) and (3.1.2) it follows that

$$\underline{V}_b(t, y) \leqslant \underline{V}^1_b(t, y) \leqslant \cdots \leqslant \underline{V}^k_b(t, y) \leqslant \overline{V}^k_b(t, y) \leqslant \cdots \leqslant \overline{V}^1_b(t, y) \leqslant \overline{V}_b(t, y).$$

Hence,

$$\underline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1) \subseteq \underline{\mathcal{Y}}^1_{OL}(t_1, t, \mathcal{Y}_1) \subseteq \cdots \subseteq \underline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1) \subseteq \overline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1) \subseteq \cdots \subseteq \overline{\mathcal{Y}}^1_{OL}(t_1, t, \mathcal{Y}_1) \subseteq \overline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1).$$

We say that

$$\overline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1) = \overline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1), \quad k = \begin{cases} \infty & \text{for continuous-time system} \\ t_1 - t - 1 & \text{for discrete-time system} \end{cases} \tag{3.34}$$

is the *maxmin closed-loop backward reach set* of system (3.10) or (3.11) at time t.

We say that

$$\underline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1) = \underline{\mathcal{Y}}^k_{OL}(t_1, t, \mathcal{Y}_1), \quad k = \begin{cases} \infty & \text{for continuous-time system} \\ t_1 - t - 1 & \text{for discrete-time system} \end{cases} \tag{3.35}$$

is the *minmax closed-loop backward reach set* of system (3.10) or (3.11) at time t.

Given the terminal time t_1 and target set $\mathcal{Y}_1$, the maxmin CLBRS $\overline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1)$ of system (3.10) or (3.11) at time $t < t_1$, is the set of all states y, for each of which for every disturbance $v(\tau) \in \mathcal{V}(\tau)$ there exists terminal state $y_1 \in \mathcal{Y}_1$ and control $u(\tau, y(\tau)) \in \mathcal{U}(\tau)$ that assigns trajectory $y(\tau, |v(\tau), u(\tau, y(\tau)))$ satisfying

$$\dot{y}(\tau|v(\tau), u(\tau, y(\tau))) \in f(\tau, y(\tau), u(\tau, y(\tau)), v(\tau))$$

in continuous-time case, or

$$y(\tau + 1|v(\tau), u(\tau, y(\tau))) \in f(\tau, y(\tau), u(\tau, y(\tau)), v(\tau))$$

in discrete-time case, with $t \leqslant \tau < t_1$, such that $y(t) = y$ and $y(t_1) = y_1$.

Given the terminal time t_1 and target set $\mathcal{Y}_1$, the minmax CLBRS $\underline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1)$ of system ([ctds2]) or [dtds2] at time $t < t_1$, is the set of all states y, for each of which there exists control $u(\tau, y(\tau)) \in \mathcal{U}(\tau)$ that for every disturbance $v(\tau) \in \mathcal{V}(\tau)$ assigns terminal state $y_1 \in \mathcal{Y}_1$ and trajectory $y(\tau, v(\tau)|u(\tau, y(\tau)))$ satisfying

$$\dot{y}(\tau, v(\tau)|u(\tau, y(\tau))) \in f(\tau, y(\tau), u(\tau, y(\tau)), v(\tau))$$

in the continuous-time case, or

$$y(\tau+1, v(\tau)|u(\tau, y(\tau))) \in f(\tau, y(\tau), u(\tau, y(\tau)), v(\tau))$$

in the discrete-time case, with $t \leqslant \tau < t_1$, such that $y(t) = y$ and $y(t_1) = y_1$.

Both maxmin and minmax CLBRS satisfy the semigroup property (3.7).

The maxmin OLBRS for the continuous-time linear system can be expressed through set valued integrals,

$$\begin{aligned}&\overline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, t_1)\mathcal{Y}_1 \oplus \int_{t_1}^{t} \Phi(t,\tau)B(\tau)\mathcal{U}(\tau)d\tau\right) \dot{-} \\ &\int_{t}^{t_1} \Phi(t,\tau)G(\tau)\mathcal{V}(\tau)d\tau,\end{aligned} \tag{3.36}$$

and for the discrete-time linear system through set-valued sums,

$$\begin{aligned}&\overline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, t_1)\mathcal{Y}_1 \oplus \sum_{\tau=t}^{t_1-1} -\Phi(t,\tau+1)B(\tau)\mathcal{U}(\tau)\right) \dot{-} \\ &\sum_{\tau=t}^{t_1-1} \Phi(t,\tau+1)G(\tau)\mathcal{V}(\tau).\end{aligned} \tag{3.37}$$

Similarly, the minmax OLBRS for the continuous-time linear system is

$$\begin{aligned}&\underline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, t_1)\mathcal{Y}_1 \dot{-} \int_{t}^{t_1} \Phi(t,\tau)G(\tau)\mathcal{V}(\tau)d\tau\right) \oplus \\ &\int_{t_1}^{t} \Phi(t,\tau)B(\tau)\mathcal{U}(\tau)d\tau,\end{aligned} \tag{3.38}$$

and for the discrete-time linear system it is

$$\begin{aligned}&\underline{\mathcal{Y}}_{OL}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, t_1)\mathcal{Y}_1 \dot{-} \sum_{\tau=t}^{t_1-1} \Phi(t,\tau+1)G(\tau)\mathcal{V}(\tau)\right) \oplus \\ &\sum_{\tau=t}^{t_1-1} -\Phi(t,\tau+1)B(\tau)\mathcal{U}(\tau).\end{aligned} \tag{3.39}$$

Now consider piecewise OLBRS with k corrections. Expression (3.32) translates into

$$\begin{aligned}&\overline{\mathcal{Y}}_{OL}^{k}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, \tau_k)\overline{\mathcal{Y}}_{OL}^{k-1}(t_1,\tau_k,\mathcal{Y}_1) \oplus \int_{\tau_k}^{t} \Phi(t,\tau)B(\tau)\mathcal{U}(\tau)d\tau\right) \dot{-} \\ &\int_{t}^{\tau_k} \Phi(t,\tau)G(\tau)\mathcal{V}(\tau)d\tau,\end{aligned} \tag{3.40}$$

in the continuous-time case, and for the discrete-time case into

$$\begin{aligned}&\overline{\mathcal{Y}}_{OL}^{k}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, \tau_k)\overline{\mathcal{Y}}_{OL}^{k-1}(t_1,\tau_k,\mathcal{Y}_1) \oplus \sum_{\tau=t}^{\tau_k-1} -\Phi(t,\tau+1)B(\tau)\mathcal{U}(\tau)\right) \dot{-} \\ &\sum_{\tau=t}^{\tau_k-1} \Phi(t,\tau+1)G(\tau)\mathcal{V}(\tau).\end{aligned} \tag{3.41}$$

Expression (3.33) translates into

$$\begin{aligned}&\underline{\mathcal{Y}}_{OL}^{k}(t_1, t, \mathcal{Y}_1) = \\ &\left(\Phi(t, \tau_k)\overline{\mathcal{Y}}_{OL}^{k-1}(t_1,\tau_k,\mathcal{Y}_1) \dot{-} \int_{t}^{\tau_k} \Phi(t,\tau)G(\tau)\mathcal{V}(\tau)d\tau\right) \oplus \\ &\int_{\tau_k}^{t} \Phi(t,\tau)B(\tau)\mathcal{U}(\tau)d\tau,\end{aligned} \tag{3.42}$$

in the continuous-time case, and for the discrete-time case into

$$\begin{aligned}&\underline{\mathcal{Y}}_{OL}^{k}(t_1, t, \mathcal{Y}_1) = \\ &(\Phi(t, \tau_k)\overline{\mathcal{Y}}_{OL}^{k-1}(t_1,\tau_k,\mathcal{Y}_1) \dot{-} \sum_{\tau=t}^{\tau_k-1} \Phi(t,\tau+1)G(\tau)\mathcal{V}(\tau)) \oplus \\ &\sum_{\tau=t}^{\tau_k-1} -\Phi(t,\tau+1)B(\tau)\mathcal{U}(\tau).\end{aligned} \tag{3.43}$$

For continuous-time linear systems $\overline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1) = \underline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1) = \mathcal{Y}_{CL}(t_1, t, \mathcal{Y}_1)$ under the condition that the target set $\mathcal{Y}_1$ is large enough to ensure that $\underline{\mathcal{Y}}_{CL}(t_1, t_1 - \epsilon, \mathcal{Y}_1)$ is nonempty for some small $\epsilon > 0$.

Computation of backward reach sets for discrete-time linear systems makes sense only if the state transition matrix $\Phi(t_1, t)$ is invertible.

If the target set $\mathcal{Y}_1$, control sets $\mathcal{U}(\tau)$ and disturbance sets $\mathcal{V}(\tau)$, $t \leqslant \tau < t_1$, are compact and convex, then CLBRS $\overline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1)$ and $\underline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{Y}_1)$ are compact and convex, if they are nonempty.

3.1.3 Reachability problem

Reachability analysis is concerned with the computation of the forward $\mathcal{X}(t, t_0, \mathcal{X}_0)$ and backward $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ reach sets (the reach sets may be maxmin or minmax) in a way that can effectively meet requests like the following:

1. For the given time interval $[t_0, t]$, determine whether the system can be steered into the given target set $\mathcal{Y}_1$. In other words, is the set $\mathcal{Y}_1 \cap \bigcup_{t_0 \leqslant \tau \leqslant t} \mathcal{X}(\tau, t_0, \mathcal{X}_0)$ nonempty? And if the answer is 'yes', find a control that steers the system to the target set (or avoids the target set). [8]
2. If the target set $\mathcal{Y}_1$ is reachable from the given initial condition $\{t_0, \mathcal{X}_0\}$ in the time interval $[t_0, t]$, find the shortest time to reach $\mathcal{Y}_1$,

$$\arg\min_{\tau} \{\mathcal{X}(\tau, t_0, \mathcal{X}_0) \cap \mathcal{Y}_1 \neq \emptyset \mid t_0 \leqslant \tau \leqslant t\}.$$

3. Given the terminal time t_1, target set $\mathcal{Y}_1$ and time $t < t_1$ find the set of states starting at time t from which the system can reach $\mathcal{Y}_1$ within time interval $[t, t_1]$. In other words, find $\bigcup_{t \leqslant \tau < t_1} \mathcal{Y}(t_1, \tau, \mathcal{Y}_1)$.
4. Find a closed-loop control that steers a system with disturbances to the given target set in given time.
5. Graphically display the projection of the reach set along any specified two- or three-dimensional subspace.

For linear systems, if the initial set $\mathcal{X}_0$, target set $\mathcal{Y}_1$, control bounds $\mathcal{U}(\cdot)$ and disturbance bounds $\mathcal{V}(\cdot)$ are compact and convex, so are the forward $\mathcal{X}(t, t_0, \mathcal{X}_0)$ and backward $\mathcal{Y}(t_1, t, \mathcal{Y}_1)$ reach sets. Hence reachability analysis requires the computationally effective manipulation of convex sets, and performing the set-valued operations of unions, intersections, geometric sums and differences.

Existing reach set computation tools can deal reliably only with linear systems with convex constraints. A claim that certain tool or method can be used *effectively* for nonlinear systems must be treated with caution, and the first question to ask is for what class of nonlinear systems and with what limit on the state space dimension does this tool work? Some "reachability methods for nonlinear systems" reduce to the local linearization of a system followed by the use of well-tested techniques for linear system reach set computation. Thus these approaches in fact use reachability methods for linear systems.

3.2 Ellipsoidal Method

3.2.1 Continuous-time systems

Consider the system

$$\dot{x}(t) = A(t)x(t) + B(t)u + G(t)v, \tag{3.44}$$

in which $x \in \mathbf{R}^n$ is the state, $u \in \mathbf{R}^m$ is the control and $v \in \mathbf{R}^d$ is the disturbance. $A(t)$, $B(t)$ and $G(t)$ are continuous and take their values in $\mathbf{R}^{n\times n}$, $\mathbf{R}^{n\times m}$ and $\mathbf{R}^{n\times d}$ respectively. Control $u(t, x(t))$ and disturbance $v(t)$ are measurable functions restricted by ellipsoidal constraints: $u(t, x(t)) \in \mathcal{E}(p(t), P(t))$ and $v(t) \in \mathcal{E}(q(t), Q(t))$. The set of initial states at initial time t_0 is assumed to be the ellipsoid $\mathcal{E}(x_0, X_0)$.

[8] So-called verification problems often consist in ensuring that the system is unable to reach an 'unsafe' target set within a given time interval.

The reach sets for systems with disturbances computed by the Ellipsoidal Toolbox are CLRS. Henceforth, when describing backward reachability, reach sets refer to CLRS or CLBRS. Recall that for continuous-time linear systems maxmin and minmax CLRS coincide, and the same is true for maxmin and minmax CLBRS.

If the matrix $Q(\cdot) = 0$, the system (3.44) becomes an ordinary affine system with known $v(\cdot) = q(\cdot)$. If $G(\cdot) = 0$, the system becomes linear. For these two cases ($Q(\cdot) = 0$ or $G(\cdot) = 0$) the reach set is as given in Definition [def:sub:$o$lrs], and so the reach set will be denoted as $\mathcal{X}_{CL}(t, t_0, \mathcal{E}(x_0, X_0)) = \mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0))$.

The reach set $\mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0))$ is a symmetric compact convex set, whose center evolves in time according to

$$\dot{x}_c(t) = A(t)x_c(t) + B(t)p(t) + G(t)q(t), \quad x_c(t_0) = x_0. \tag{3.45}$$

Fix a vector $l_0 \in \mathbf{R}^n$, and consider the solution $l(t)$ of the adjoint equation

$$\dot{l}(t) = -A^T(t)l(t), \quad l(t_0) = l_0, \tag{3.46}$$

which is equivalent to

$$l(t) = \Phi^T(t_0, t)l_0.$$

If the reach set $\mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0))$ is nonempty, there exist tight external and tight internal approximating ellipsoids $\mathcal{E}(x_c(t), X_l^+(t))$ and $\mathcal{E}(x_c(t), X_l^-(t))$, respectively, such that

$$\mathcal{E}(x_c(t), X_l^-(t)) \subseteq \mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0)) \subseteq \mathcal{E}(x_c(t), X_l^+(t)), \tag{3.47}$$

and

$$\rho(l(t) \mid \mathcal{E}(x_c(t), X_l^-(t))) = \rho(l(t) \mid \mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0))) = \rho(l(t) \mid \mathcal{E}(x_c(t), X_l^+(t))). \tag{3.48}$$

The equation for the shape matrix of the external ellipsoid is

$$\begin{aligned}\dot{X}_l^+(t) = A(t)X_l^+(t) + X_l^+(t)A^T(t) + &\\ \pi_l(t)X_l^+(t) + \frac{1}{\pi_l(t)}B(t)P(t)B^T(t) - &\\ (X_l^+(t))^{1/2}S_l(t)(G(t)Q(t)G^T(t))^{1/2} - &\\ (G(t)Q(t)G^T(t))^{1/2}S_l^T(t)(X_l^+(t))^{1/2}, &\end{aligned}$$

$$X_l^+(t_0) = X_0, \tag{3.49}$$

in which

$$\pi_l(t) = \frac{\langle l(t), B(t)P(t)B^T(t)l(t)\rangle^{1/2}}{\langle l(t), X_l^+(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $S_l(t)$ ($S_l(t)S_l^T(t) = I$) is determined by the equation

$$S_l(t)(G(t)Q(t)G^T(t))^{1/2}l(t) = \frac{\langle l(t), G(t)Q(t)G^T(t)l(t)\rangle^{1/2}}{\langle l(t), X_l^+(t)l(t)\rangle^{1/2}}(X_l^+(t))^{1/2}l(t).$$

In the presence of disturbance, if the reach set is empty, the matrix $X_l^+(t)$ becomes sign indefinite. For a system without disturbance, the terms containing $G(t)$ and $Q(t)$ vanish from the equation (3.2.1).

The equation for the shape matrix of the internal ellipsoid is

$$\begin{aligned}\dot{X}_l^-(t) = A(t)X_l^-(t) + X_l^-(t)A^T(t) + &\\ (X_l^-(t))^{1/2}T_l(t)(B(t)P(t)B^T(t))^{1/2} + &\\ (B(t)P(t)B^T(t))^{1/2}T_l^T(t)(X_l^-(t))^{1/2} - &\\ \eta_l(t)X_l^-(t) - \frac{1}{\eta_l(t)}G(t)Q(t)G^T(t), &\end{aligned}$$

$$X_l^-(t_0) = X_0, \tag{3.50}$$

in which

$$\eta_l(t) = \frac{\langle l(t), G(t)Q(t)G^T(t)l(t)\rangle^{1/2}}{\langle l(t), X_l^+(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $T_l(t)$ is determined by the equation

$$T_l(t)(B(t)P(t)B^T(t))^{1/2}l(t) = \frac{\langle l(t), B(t)P(t)B^T(t)l(t)\rangle^{1/2}}{\langle l(t), X_l^-(t)l(t)\rangle^{1/2}}(X_l^-(t))^{1/2}l(t).$$

Similarly to the external case, the terms containing $G(t)$ and $Q(t)$ vanish from the equation ([fwdint1]) for a system without disturbance.

The point where the external and internal ellipsoids touch the boundary of the reach set is given by

$$x_l^*(t) = x_c(t) + \frac{X_l^+(t)l(t)}{\langle l(t), X_l^+(t)l(t)\rangle^{1/2}}.$$

The boundary points $x_l^*(t)$ form trajectories, which we call *extremal trajectories*. Due to the nonsingular nature of the state transition matrix $\Phi(t, t_0)$, every boundary point of the reach set belongs to an extremal trajectory. To follow an extremal trajectory specified by parameter l_0, the system has to start at time t_0 at initial state

$$x_l^0 = x_0 + \frac{X_0 l_0}{\langle l_0, X_0 l_0\rangle^{1/2}}. \tag{3.51}$$

In the absence of disturbances, the open-loop control

$$u_l(t) = p(t) + \frac{P(t)B^T(t)l(t)}{\langle l(t), B(t)P(t)B^T(t)l(t)\rangle^{1/2}}. \tag{3.52}$$

steers the system along the extremal trajectory defined by the vector l_0. When a disturbance is present, this control keeps the system on an extremal trajectory if and only if the disturbance plays against the control always taking its extreme values.

Expressions (3.47) and (3.48) lead to the following fact,

$$\bigcup_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), X_l^-(t)) = \mathcal{X}(t, t_0, \mathcal{E}(x_0, X_0)) = \bigcap_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), X_l^+(t)).$$

In practice this means that the more values of l_0 we use to compute $X_l^+(t)$ and $X_l^-(t)$, the better will be our approximation.

Analogous results hold for the backward reach set.

Given the terminal time t_1 and ellipsoidal target set $\mathcal{E}(y_1, Y_1)$, the CLBRS $\mathcal{Y}_{CL}(t_1, t, \mathcal{Y}_1) = \mathcal{Y}(t_1, t, \mathcal{Y}_1)$, $t < t_1$, if it is nonempty, is a symmetric compact convex set whose center is governed by

$$y_c(t) = Ay_c(t) + B(t)p(t) + G(t)q(t), \quad y_c(t_1) = y_1. \tag{3.53}$$

Fix a vector $l_1 \in \mathbf{R}^n$, and consider

$$l(t) = \Phi(t_1, t)^T l_1. \tag{3.54}$$

If the backward reach set $\mathcal{Y}(t_1, t, \mathcal{E}(y_1, Y_1))$ is nonempty, there exist tight external and tight internal approximating ellipsoids $\mathcal{E}(y_c(t), Y_l^+(t))$ and $\mathcal{E}(y_c(t), Y_l^-(t))$ respectively, such that

$$\mathcal{E}(y_c(t), Y_l^-(t)) \subseteq \mathcal{Y}(t_1, t, \mathcal{E}(y_1, Y_1)) \subseteq \mathcal{E}(y_c(t), Y_l^+(t)), \tag{3.55}$$

and

$$\rho(l(t) \mid \mathcal{E}(y_c(t), Y_l^-(t))) = \rho(l(t) \mid \mathcal{Y}(t_1, t, \mathcal{E}(y_0, Y_0))) = \rho(l(t) \mid \mathcal{E}(y_c(t), Y_l^+(t))). \tag{3.56}$$

The equation for the shape matrix of the external ellipsoid is

$$\begin{aligned}\dot{Y}_l^+(t) = A(t)Y_l^+(t) + Y_l^+(t)A^T(t) - \\ \pi_l(t)Y_l^+(t) - \frac{1}{\pi_l(t)}B(t)P(t)B^T(t) + \\ (Y_l^+(t))^{1/2}S_l(t)(G(t)Q(t)G^T(t))^{1/2} + \\ (G(t)Q(t)G^T(t))^{1/2}S_l^T(t)(Y_l^+(t))^{1/2},\end{aligned}$$

$$Y_l^+(t_1) = Y_1, \tag{3.57}$$

in which

$$\pi_l(t) = \frac{\langle l(t), B(t)P(t)B^T(t)l(t)\rangle^{1/2}}{\langle l(t), Y_l^+(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $S_l(t)$ satisfies the equation

$$S_l(t)(G(t)Q(t)G^T(t))^{1/2}l(t) = \frac{\langle l(t), G(t)Q(t)G^T(t)l(t)\rangle^{1/2}}{\langle l(t), Y_l^+(t)l(t)\rangle^{1/2}}(Y_l^+(t))^{1/2}l(t).$$

The equation for the shape matrix of the internal ellipsoid is

$$\begin{aligned}\dot{Y}_l^-(t) = A(t)Y_l^-(t) + Y_l^-(t)A^T(t) - \\ (Y_l^-(t))^{1/2}T_l(t)(B(t)P(t)B^T(t))^{1/2} - \\ (B(t)P(t)B^T(t))^{1/2}T_l^T(t)(Y_l^-(t))^{1/2} + \\ \eta_l(t)Y_l^-(t) + \frac{1}{\eta_l(t)}G(t)Q(t)G^T(t),\end{aligned}$$

$$Y_l^-(t_1) = Y_1, \tag{3.58}$$

in which

$$\eta_l(t) = \frac{\langle l(t), G(t)Q(t)G^T(t)l(t)\rangle^{1/2}}{\langle l(t), Y_l^+(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $T_l(t)$ is determined by the equation

$$T_l(t)(B(t)P(t)B^T(t))^{1/2}l(t) = \frac{\langle l(t), B(t)P(t)B^T(t)l(t)\rangle^{1/2}}{\langle l(t), Y_l^-(t)l(t)\rangle^{1/2}}(Y_l^-(t))^{1/2}l(t).$$

Just as in the forward reachability case, the terms containing $G(t)$ and $Q(t)$ vanish from equations (3.2.1) and (3.2.1) in the absence of disturbances. The boundary value problems (3.53), (3.2.1) and (3.2.1) are converted to the initial value problems by the change of variables $s = -t$.

Due to (3.55) and (3.56),

$$\bigcup_{\langle l_1, l_1\rangle = 1} \mathcal{E}(y_c(t), Y_l^-(t)) = \mathcal{Y}(t_1, t, \mathcal{E}(y_1, Y_1)) = \bigcap_{\langle l_1, l_1\rangle = 1} \mathcal{E}(y_c(t), Y_l^+(t)).$$

Remark. In expressions (3.2.1), (3.2.1), (3.2.1) and (3.2.1) the terms $\frac{1}{\pi_l(t)}$ and $\frac{1}{\eta_l(t)}$ may not be well defined for some vectors l, because matrices $B(t)P(t)B^T(t)$ and $G(t)Q(t)G^T(t)$ may be singular. In such cases, we set these entire expressions to zero.

3.2.2 Discrete-time systems

Consider the discrete-time linear system,

$$x(t+1) = A(t)x(t) + B(t)u(t,x(t)) + G(t)v(t), \tag{3.59}$$

in which $x(t) \in \mathbf{R}^n$ is the state, $u(t,x(t)) \in \mathbf{R}^m$ is the control bounded by the ellipsoid $\mathcal{E}(p(t),P(t))$, $v(t) \in \mathbf{R}^d$ is disturbance bounded by ellipsoid $\mathcal{E}(q(t),Q(t))$, and matrices $A(t)$, $B(t)$, $G(t)$ are in $\mathbf{R}^{n\times n}$, $\mathbf{R}^{n\times m}$, $\mathbf{R}^{n\times d}$ respectively. Here we shall assume $A(t)$ to be nonsingular. [9] The set of initial conditions at initial time t_0 is ellipsoid $\mathcal{E}(x_0,X_0)$.

Ellipsoidal Toolbox computes maxmin and minmax CLRS $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)$ and $\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)$ for discrete-time systems.

If matrix $Q(\cdot) = 0$, the system (3.59) becomes an ordinary affine system with known $v(\cdot) = q(\cdot)$. If matrix $G(\cdot) = 0$, the system reduces to a linear controlled system. In the absence of disturbance ($Q(\cdot) = 0$ or $G(\cdot) = 0$), $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)) = \underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)) = \mathcal{X}(t,t_0,\mathcal{E}(x_0,X_0))$, the reach set is as in Definition [def:sub:$o$lrs]. !!! WATCH !!!

Maxmin and minmax CLRS $\overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)$ and $\underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)$, if nonempty, are symmetric convex and compact, with the center evolving in time according to

$$\begin{aligned} x_c(t+1) = A(t)x_c(t) + B(t)p(t) + G(t)v(t), \\ x_c(t_0) = x_0. \end{aligned} \tag{3.60}$$

Fix some vector $l_0 \in \mathbf{R}^n$ and consider $l(t)$ that satisfies the discrete-time adjoint equation, [10]

$$\begin{aligned} l(t+1) = \left(A^T\right)^{-1}(t)l(t), \\ l(t_0) = l_0, \end{aligned} \tag{3.61}$$

or, equivalently

$$l(t) = \Phi^T(t_0,t)l_0.$$

There exist tight external ellipsoids $\mathcal{E}(x_c(t),\overline{X}_l^+(t))$, $\mathcal{E}(x_c(t),\underline{X}_l^+(t))$ and tight internal ellipsoids $\mathcal{E}(x_c(t),\overline{X}_l^-(t))$, $\mathcal{E}(x_c(t),\underline{X}_l^-(t))$ such that

$$\mathcal{E}(x_c(t),\overline{X}_l^-(t)) \subseteq \overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)) \subseteq \mathcal{E}(x_c(t),\overline{X}_l^+(t)), \tag{3.62}$$

$$\rho(l(t) \mid \mathcal{E}(x_c(t),\overline{X}_l^-(t))) = \rho(l(t) \mid \overline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0))) = \rho(l(t) \mid \mathcal{E}(x_c(t),\overline{X}_l^+(t))). \tag{3.63}$$

and

$$\mathcal{E}(x_c(t),\underline{X}_l^-(t)) \subseteq \underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0)) \subseteq \mathcal{E}(x_c(t),\underline{X}_l^+(t)), \tag{3.64}$$

$$\rho(l(t) \mid \mathcal{E}(x_c(t),\underline{X}_l^-(t))) = \rho(l(t) \mid \underline{\mathcal{X}}_{CL}(t,t_0,\mathcal{E}(x_0,X_0))) = \rho(l(t) \mid \mathcal{E}(x_c(t),\underline{X}_l^+(t))). \tag{3.65}$$

[9] The case when $A(t)$ is singular is described in A. A. Kurzhanskiy (2007). The idea is to substitute $A(t)$ with the nonsingular $A_\delta(t) = A(t) + \delta U(t)W(t)$, in which $U(t)$ and $W(t)$ are obtained from the singular value decomposition

$$A(t) = U(t)\Sigma(t)V(t).$$

The parameter δ can be chosen based on the number of time steps for which the reach set must be computed and the required accuracy. The issue of inverting ill-conditioned matrices is also addressed in A. A. Kurzhanskiy (2007).

[10] Note that for (3.61) $A(t)$ must be invertible.

The shape matrix of the external ellipsoid for maxmin reach set is determined from

$$\hat{X}_l^+(t) = (1 + \overline{\pi}_l(t))A(t)\overline{X}_l^+(t)A^T(t) + \left(1 + \frac{1}{\overline{\pi}_l(t)}\right)B(t)P(t)B^T(t), \tag{3.66}$$

$$\begin{aligned}\overline{X}_l^+(t+1) = &\left((\hat{X}_l^+(t))^{1/2} + \overline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}\right)^T \times \\ &\left((\hat{X}_l^+(t))^{1/2} + \overline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}\right),\end{aligned}$$

$$\overline{X}_l^+(t_0) = X_0, \tag{3.67}$$

wherein

$$\overline{\pi}_l(t) = \frac{\langle l(t+1), B(t)P(t)B^T(t)l(t+1)\rangle^{1/2}}{\langle l(t), \overline{X}_l^+(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $\overline{S}_l(t)$ is determined by the equation

$$\begin{aligned}&\overline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}l(t+1) = \\ &\frac{\langle l(t+1), G(t)Q(t)G^T(t)l(t+1)\rangle^{1/2}}{\langle l(t+1), \hat{X}_l^+(t)l(t+1)\rangle^{1/2}}(\hat{X}_l^+(t))^{1/2}l(t+1).\end{aligned}$$

Equation (3.2.2) is valid only if $\mathcal{E}(0, G(t)Q(t)G^T(t)) \subseteq \mathcal{E}(0, \hat{X}_l^+(t))$, otherwise the maxmin CLRS $\overline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0))$ is empty.

The shape matrix of the external ellipsoid for minmax reach set is determined from

$$\begin{aligned}\check{X}_l^+(t) = &\left((A(t)\underline{X}_l^+(t)A^T(t))^{1/2} + \underline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}\right)^T \times \\ &\left((A(t)\underline{X}_l^+(t)A^T(t))^{1/2} + \underline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}\right)\end{aligned}$$

$$\underline{X}_l^+(t+1) = (1 + \underline{\pi}_l(t))\check{X}_l^+(t) + \left(1 + \frac{1}{\underline{\pi}_l(t)}\right)B(t)P(t)B^T(t), \tag{3.68}$$

$$\underline{X}_l^+(t_0) = X_0, \tag{3.69}$$

where

$$\underline{\pi}_l(t) = \frac{\langle l(t+1), B(t)P(t)B^T(t)l(t+1)\rangle^{1/2}}{\langle l(t+1), \check{X}_l^+(t)l(t+1)\rangle^{1/2}},$$

and $\underline{S}_l(t)$ is orthogonal matrix determined from the equation

$$\begin{aligned}&\underline{S}_l(t)(G(t)Q(t)G^T(t))^{1/2}l(t+1) = \\ &\frac{\langle l(t+1), G(t)Q(t)G^T(t)l(t+1)\rangle^{1/2}}{\langle l(t), \underline{X}_l^+(t)l(t)\rangle^{1/2}}(A(t)\underline{X}_l^+(t)A^T(t))^{1/2}l(t+1).\end{aligned}$$

Equations (3.2.2), (3.68) are valid only if $\mathcal{E}(0, G(t)Q(t)G^T(t) \subseteq \mathcal{E}(0, A(t)\underline{X}_l^+(t)A^T(t))$, otherwise minmax CLRS $\underline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0))$ is empty.

The shape matrix of the internal ellipsoid for maxmin reach set is determined from

$$\hat{X}_l^-(t) = \left((A(t)\overline{X}_l^-(t)A^T(t))^{1/2} + \overline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}\right)^T \times \\ \left((A(t)\overline{X}_l^-(t)A^T(t))^{1/2} + \overline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}\right)$$

$$\overline{X}_l^-(t+1) = (1+\overline{\eta}_l(t))\hat{X}_l^-(t) + \left(1 + \frac{1}{\underline{\eta}_l(t)}\right) G(t)Q(t)G^T(t), \quad (3.70)$$

$$\overline{X}_l^-(t_0) = X_0, \quad (3.71)$$

where

$$\overline{\eta}_l(t) = \frac{\langle l(t+1), G(t)Q(t)G^T(t)l(t+1)\rangle^{1/2}}{\langle l(t+1), \hat{X}_l^-(t)l(t+1)\rangle^{1/2}},$$

and $\overline{T}_l(t)$ is orthogonal matrix determined from the equation

$$\overline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}l(t+1) = \\ \frac{\langle l(t+1), B(t)P(t)B^T(t)l(t+1)\rangle^{1/2}}{\langle l(t), \overline{X}_l^-(t)l(t)\rangle^{1/2}}(A(t)\overline{X}_l^-(t)A^T(t))^{1/2}l(t+1).$$

Equation (3.70) is valid only if $\mathcal{E}(0, G(t)Q(t)G^T(t) \subseteq \mathcal{E}(0, \hat{X}_l^-(t))$.

The shape matrix of the internal ellipsoid for the minmax reach set is determined by

$$\check{X}_l^-(t) = (1+\underline{\eta}_l(t))A(t)\underline{X}_l^-(t)A^T(t) + \left(1 + \frac{1}{\underline{\eta}_l(t)}\right) G(t)Q(t)G^T(t), \quad (3.72)$$

$$\underline{X}_l^-(t+1) = \left((\check{X}_l^-(t))^{1/2} + \underline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}\right)^T \times \\ \left((\check{X}_l^-(t))^{1/2} + \underline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}\right),$$

$$\underline{X}_l^-(t_0) = X_0, \quad (3.73)$$

wherein

$$\underline{\eta}_l(t) = \frac{\langle l(t+1), G(t)Q(t)G^T(t)l(t+1)\rangle^{1/2}}{\langle l(t), \underline{X}_l^-(t)l(t)\rangle^{1/2}},$$

and the orthogonal matrix $\underline{T}_l(t)$ is determined by the equation

$$\underline{T}_l(t)(B(t)P(t)B^T(t))^{1/2}l(t+1) = \\ \frac{\langle l(t+1), B(t)P(t)B^T(t)l(t+1)\rangle^{1/2}}{\langle l(t+1), \check{X}_l^-(t)l(t+1)\rangle^{1/2}}(\check{X}_l^-(t))^{1/2}l(t+1).$$

Equations (3.72), (3.2.2) are valid only if $\mathcal{E}(0, G(t)Q(t)G^T(t) \subseteq \mathcal{E}(0, A(t)\underline{X}_l^-(t)A^T(t))$.

The point where the external and the internal ellipsoids both touch the boundary of the maxmin CLRS is

$$x_l^+(t) = x_c(t) + \frac{\overline{X}_l^+(t)l(t)}{\langle l(t), \overline{X}_l^+(t)l(t)\rangle^{1/2}},$$

and the bounday point of minmax CLRS is

$$x_l^-(t) = x_c(t) + \frac{\overline{X}_l^-(t)l(t)}{\langle l(t), \overline{X}_l^-(t)l(t)\rangle^{1/2}}.$$

Points $x_l^{\pm}(t)$, $t \geqslant t_0$, form extremal trajectories. In order for the system to follow the extremal trajectory specified by some vector l_0, the initial state must be

$$x_l^0 = x_0 + \frac{X_0 l_0}{\langle l_0, X_0 l_0\rangle^{1/2}}. \tag{3.74}$$

When there is no disturbance ($G(t) = 0$ or $Q(t) = 0$), $\overline{X}_l^+(t) = \underline{X}_l^+(t)$ and $\overline{X}_l^-(t) = \underline{X}_l^-(t)$, and the open-loop control that steers the system along the extremal trajectory defined by l_0 is

$$u_l(t) = p(t) + \frac{P(t)B^T(t)l(t+1)}{\langle l(t+1), B(t)P(t)B^T(t)l(t+1)\rangle^{1/2}}. \tag{3.75}$$

Each choice of l_0 defines an external and internal approximation. If $\overline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0))$ is nonempty,

$$\bigcup_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), \overline{X}_l^-(t)) = \overline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0)) = \bigcap_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), \overline{X}_l^+(t)).$$

Similarly for $\underline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0))$,

$$\bigcup_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), \underline{X}_l^-(t)) = \underline{\mathcal{X}}_{CL}(t, t_0, \mathcal{E}(x_0, X_0)) = \bigcap_{\langle l_0, l_0\rangle = 1} \mathcal{E}(x_c(t), \underline{X}_l^+(t)).$$

Similarly, tight ellipsoidal approximations of maxmin and minmax CLBRS with terminating conditions $(t_1, \mathcal{E}(y_1, Y_1))$ can be obtained for those directions $l(t)$ satisfying

$$l(t) = \Phi^T(t_1, t)l_1, \tag{3.76}$$

with some fixed l_1, for which they exist.

With boundary conditions

$$y_c(t_1) = y_1, \quad \overline{Y}_l^+(t_1) = \overline{Y}_l^-(t_1) = \underline{Y}_l^+(t_1) = \underline{Y}_l^-(t_1) = Y_1, \tag{3.77}$$

external and internal ellipsoids for maxmin CLBRS $\overline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{E}(y_1, Y_1))$ at time t, $\mathcal{E}(y_c(t), \overline{Y}_l^+(t))$ and $\mathcal{E}(y_c(t), \overline{Y}_l^-(t))$, are computed as external and internal ellipsoidal approximations of the geometric sum-difference

$$A^{-1}(t)\left(\mathcal{E}(y_c(t+1), \overline{Y}_l^+(t+1)) \oplus B(t)\mathcal{E}(-p(t), P(t)) \dot{-} G(t)\mathcal{E}(-q(t), Q(t))\right)$$

and

$$A^{-1}(t)\left(\mathcal{E}(y_c(t+1), \overline{Y}_l^-(t+1)) \oplus B(t)\mathcal{E}(-p(t), P(t)) \dot{-} G(t)\mathcal{E}(-q(t), Q(t))\right)$$

in direction $l(t)$ from (3.76). Section [subsec:sub:$s$umdiff] describes the operation of geometric sum-difference for ellipsoids. !!! WATCH !!!

External and internal ellipsoids for minmax CLBRS $\underline{\mathcal{Y}}_{CL}(t_1, t, \mathcal{E}(y_1, Y_1))$ at time t, $\mathcal{E}(y_c(t), \underline{Y}_l^+(t))$ and $\mathcal{E}(y_c(t), \underline{Y}_l^-(t))$, are computed as external and internal ellipsoidal approximations of the geometric difference-sum

$$A^{-1}(t)\left(\mathcal{E}(y_c(t+1), \underline{Y}_l^+(t+1)) \dot{-} G(t)\mathcal{E}(-q(t), Q(t)) \oplus B(t)\mathcal{E}(-p(t), P(t))\right)$$

and

$$A^{-1}(t)\left(\mathcal{E}(y_c(t+1), \underline{Y}_l^-(t+1)) \dot{-} G(t)\mathcal{E}(-q(t), Q(t)) \oplus B(t)\mathcal{E}(-p(t), P(t))\right)$$

in direction $l(t)$ from (3.76). Section [subsec:sub:$d$iffsum] describes the operation of geometric difference-sum for ellipsoids.

A. A. Kurzhanskiy, P. Varaiya. 2007. "Ellipsoidal Techniques for Reachability Analysis of Discrete-time Linear Systems." *IEEE Transactions on Automatic Control* 52 (1): 26–38.

CHAPTER

FOUR

INSTALLATION

4.1 Additional Software

These packages aren't included in the ET distribution. So, you need to download them separately.

4.1.1 CVX

Some methods of the *Ellipsoidal Toolbox*, namely,

- distance
- intersect
- isInside
- doesContain
- ellintersection_ia
- ellunion_ea

require solving semidefinite programming (SDP) problems. We use CVX (("CVX Homepage")) as an interface to an external SDP solver. CVX is a reliable toolbox for solving SDP problems of high dimensionality. CVX is implemented in Matlab, effectively turning Matlab into an optimization modeling language. Model specifications are constructed using common Matlab operations and functions, and standard Matlab code can be freely mixed with these specifications. This combination makes it simple to perform the calculations needed to form optimization problems, or to process the results obtained from their solution. CVX distribution includes two freeware solvers: SeDuMi(Sturm (1999), ("SeDuMi Homepage")) and SDPT3(("SDPT3 Homepage")). The default solver used in the toolbox is SeDuMi.

4.1.2 MPT

Multi-Parametric Toolbox(("Multi-Parametric Toolbox Homepage")) - a Matlab toolbox for multi-parametric optimization and computational geometry. MPT is a toolbox that defines polytope class used in *ET*. We need MPT for the following methods operating with polytopes.

- distance
- intersect
- intersection_ia
- intersection_ea
- isInside

- hyperplane2polytope
- polytope2hyperplane

4.2 Installation and Quick Start

1. Go to and download the *Ellipsoidal Toolbox*.
2. Unzip the distribution file into the directory where you would like the toolbox to be.
3. Unzip CVX into cvx folder next to products folder.
4. Unzip MPT into mpt folder next to products folder.
5. Read the copyright notice.
6. In MATLAB command window change the working directory to the one where you unzipped the toolbox and type
7. At this point, the directory tree of the *Ellipsoidal Toolbox* is added to the MATLAB path list. In order to save the updated path list, in your MATLAB window menu go to File → Set Path... and click Save.
8. To get an idea of what the toolbox is about, type

   This will produce a demo of basic *ET* functionality: how to create and manipulate ellipsoids.

   Type

   to learn how to plot ellipsoids and hyperplanes in 2 and 3D. For a quick tutorial on how to use the toolbox for reachability analysis and verification, type

"CVX Homepage." cvxr.com/cvx.

"Multi-Parametric Toolbox Homepage." control.ee.ethz.ch/\~mpt.

"SDPT3 Homepage." http://www.math.nus.edu.sg/\~mattohkc/sdpt3.html.

"SeDuMi Homepage." sedumi.mcmaster.ca.

Sturm, J. F. 1999. "Using SeDuMi 1.02, A MATLAB Toolbox for Optimization over Symmetric Cones." *Optimization Methods and Software* 11-12: 625–653.

CHAPTER

FIVE

IMPLEMENTATION

5.1 Operations with ellipsoids

In the *Ellipsoidal Toolbox* we define a new class ellipsoid inside the MATLAB programming environment. The following three commands define the same ellipsoid $\mathcal{E}(q, Q)$, with $q \in \mathbf{R}^n$ and $Q \in \mathbf{R}^{n \times n}$ being symmetric positive semidefinite:

```
centVec = [1 2]';
shMat = eye(2, 2);
ellObj = ellipsoid(centVec, shMat);
ellObj = ellipsoid(shMat) + centVec;
ellObj = sqrtm(shMat)*ell_unitball(size(shMat, 1)) + centVec;
```

For the ellipsoid class we overload the following functions and operators:

- isEmpty(ellObj) - checks if $\mathcal{E}(q, Q)$ is an empty ellipsoid.
- display(ellObj) - displays the details of ellipsoid $\mathcal{E}(q, Q)$, namely, its center q and the shape matrix Q.
- plot(ellObj) - plots ellipsoid $\mathcal{E}(q, Q)$ if its dimension is not greater than 3.
- firstEllObj == secEllObj - checks if ellipsoids $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$ are equal.
- firstEllObj ~= secEllObj - checks if ellipsoids $\mathcal{E}(q_1, Q_1)$ and $\mathcal{E}(q_2, Q_2)$ are not equal.
- [,] - concatenates the ellipsoids into the horizontal array, e.g. ellVec = [firstEllObj secEllObj thirdEllObj].
- [;] - concatenates the ellipsoids into the vertical array, e.g. ellMat = [firstEllObj secEllObj; thirdEllObj fourthEllObj] defines 2×2 array of ellipsoids.
- firstEllObj >= secEllObj - checks if the ellipsoid firstEllObj is bigger than the ellipsoid secEllObj, or equivalently $\mathcal{E}(0, Q_1) \subseteq \mathcal{E}(0, Q_2)$.
- firstEllObj <= secEllObj - checks if $\mathcal{E}(0, Q_2) \subseteq \mathcal{E}(0, Q_1)$.
- -ellObj - defines ellipsoid $\mathcal{E}(-q, Q)$.
- ellObj + bScal - defines ellipsoid $\mathcal{E}(q + b, Q)$.
- ellObj - bScal - defines ellipsoid $\mathcal{E}(q - b, Q)$.
- aMat * ellObj - defines ellipsoid $\mathcal{E}(q, AQA^T)$.
- ellObj.inv() - inverts the shape matrix of the ellipsoid: $\mathcal{E}(q, Q^{-1})$.

All the listed operations can be applied to a single ellipsoid as well as to a two-dimensional array of ellipsoids. For example,

```
% nondegenerate ellipsoid in R^2
firstEllObj = ellipsoid([2; -1], [9 -5; -5 4]);
secEllObj = firstEllObj.polar();% secEll is polar ellipsoid for firstEllObj
% thirdEllObj is generated from secEllObj by inverting its shape matrix
thirdEllObj = secEllObj.getInv();
% 2x2 array of ellipsoids
ellMat = [firstEllObj secEllObj; thirdEllObj ellipsoid([1; 1], eye(2))];
% check if firstEllObj is bigger than each of the ellipsoids in ellMat
ellMat <= firstEllObj

% ans =
%
%      1     0
%      1     0
```

To access individual elements of the array, the usual MATLAB subindexing is used:

```
aMat = [0 1; -2 0];  % aMat - 2x2 real matrix
bVec = [3; 0]; % bVec - vector in R^2
% affine transformation of ellipsoids in the second column of ellMat
aTransMat = aMat * ellMat(:, 2) + bVec;
```

Sometimes it may be useful to modify the shape of the ellipsoid without affecting its center. Say, we would like to bloat or squeeze the ellipsoid:

```
bltEllObj = firstEllObj.getShape(2);  % bloats ellipsoid firstEllObj
sqzEllObj = firstEllObj.getShape(0.5);  % squeezes ellipsoid firstEllObj
```

Since function shape does not change the center of the ellipsoid, it only accepts scalars or square matrices as its second input parameter. Several functions access the internal data of the ellipsoid object:

```
[centVec, shMat] = secEllObj.double()
% centVec =
%
%    -0.5000
%    -0.1667
%
% shMat =
%
%    0.9167     0.9167
%    0.9167     1.5278
```

```
% define new ellipsoid
fourthEllObj = ellipsoid([42 -7 -2 4; -7 10 3 1; -2 3 5 -2; 4 1 -2 2]);
bufEllVec = [ellMat(1, :) fourthEllObj];
bufEllVec.isdegenerate() % check if given ellipsoids are degenerate

% ans =
%
%      0     0     1
```

```
bufEllVec = [ellMat(1, :) fourthEllObj];
[dimVec, rankVec] = bufEllVec.dimension()


% dimVec =
%
%    2     2     4
%
```

```
% rankVec =
%
%    2    2    3
```

One way to check if two ellipsoids intersect, is to compute the distance between them (("Stanley Chan Article Homepage"), Lin and Han (2002)):

```
% distance between thirdEllObj and each of the ellipsoids in ellMat
ellMat.distance(thirdEllObj)

% ans =
%
%          0          0
%          0     0.1683
```

This result indicates that the ellipsoid thirdEllObj does not intersect with the ellipsoid ellMat(2, 2), with all the other ellipsoids in ellMat it has nonempty intersection. If the intersection of the two ellipsoids is nonempty, it can be approximated by ellipsoids from the outside as well as from the inside. See L. Ros, A. Sabater, F. Thomas (2002) for more information about these methods.

```
% external approximation of intersection of firstEllObj and thirdEllObj
externalEllObj = firstEllObj.intersection_ea(thirdEllObj);
% internal approximation of intersection of firstEllObj and thirdEllObj
internalEllObj = firstEllObj.intersection_ia(thirdEllObj);
```

It can be checked that resulting ellipsoid externalEllObj contains the given intersection, whereas internalEllObj is contained in this intersection:

```
% array [firstEllObj secEllObj] should be treated as intersection
externalEllObj.doesIntersectionContain([firstEllObj secEllObj], 'i')

% ans =
%
%      1
```

```
bufEllVec = [firstEllObj thirdEllObj]
bufEllVec.doesIntersectionContain(internalEllObj)

% ans =
%
%      1
```

Function isInside in general checks if the intersection of ellipsoids in the given array contains the union or intersection of ellipsoids or polytopes.

It is also possible to solve the feasibility problem, that is, to check if the intersection of more than two ellipsoids is empty:

```
ellMat.intersect(ellMat(1, 1), 'i')

% ans =
%
%      -1
```

In this particular example the result -1 indicates that the intersection of ellipsoids in ellMat is empty. Function intersect in general checks if an ellipsoid, hyperplane or polytope intersects the union or the intersection of ellipsoids in the given array:

```
bufEllVec = [firstEllObj secEllObj thirdEllObj]
bufEllVec.intersect(ellMat(2, 2), 'i')
```

```

% ans =
%
%       0
```

```
bufEllVec = [firstEllObj secEllObj thirdEllObj];
bufEllVec.intersect(ellMat(2, 2), 'u')

% ans =
%
%       1
```

For the ellipsoids in $\mathbf{R}$, $\mathbf{R}^2$ and $\mathbf{R}^3$ the geometric sum can be computed explicitely and plotted:

```
ellMat.minksum();
```

Figure 5.1: The geometric sum of ellipsoids.

Figure 5.1 displays the geometric sum of ellipsoids. If the dimension of the space in which the ellipsoids are defined exceeds 3, an error is returned. The result of the geometric sum operation is not generally an ellipsoid, but it can be approximated by families of external and internal ellipsoids parametrized by the direction vector:

```
% define the set of directions:
% columns of matrix dirsMat are vectors in R^2
dirsMat = [1 0; 1 1; 0 1; -1 1; 1 3]';
```

```
% compute external ellipsoids for the directions in dirsMat
externalEllVec = ellMat.minksum_ea(dirsMat)

% externalEllVec =
% Array of ellipsoids with dimensionality 1x5

% compute internal ellipsoids for the directions in dirsMat
internalEllVec = ellMat.minksum_ia(dirsMat)

% internalEllVec =
% Array of ellipsoids with dimensionality 1x5

% intersection of external ellipsoids should always contain
% the union of internal ellipsoids:
externalEllVec.doesIntersectionContain(internalEllVec, 'u')
%
% ans =
%
%      1
```

Functions minksum_ea and minksum_ia work for ellipsoids of arbitrary dimension. They should be used for general computations whereas minksum is there merely for visualization purposes.

If the geometric difference of two ellipsoids is not an empty set, it can be computed explicitely and plotted for ellipsoids in $\mathbf{R}$, $\mathbf{R}^2$ and $\mathbf{R}^3$:

```
% ellipsoid defined by squeezing the ellipsoid ellMat(2, 2)
fourthEllObj = ellMat(2, 2).getShape(0.4);
% check if the geometric difference firstEllObj - fourthEllObj is nonempty
firstEllObj >= fourthEllObj
%
% ans =
%
%      1

% compute and plot this geometric difference
firstEllObj.minkdiff(fourthEllObj);
```

Figure 5.2 shows the geometric difference of ellipsoids.

Similar to minksum, minkdiff is there for visualization purpose. It works only for dimensions 1, 2 and 3, and for higher dimensions it returns an error. For arbitrary dimensions, the geometric difference can be approximated by families of external and internal ellipsoids parametrized by the direction vector, provided this direction is not bad:

```
absTol = getAbsTol(firstEllObj);
% find out which of the directions in dirsMat are bad
firstEllObj.isbaddirection(fourthEllObj, dirsMat, absTol)

% ans =
%
%      1     0     0     1     0


% two of five directions specified by dirsMat are bad,
% so, only three ellipsoidal approximations
% can be produced for this dirsMat:
externalEllVec = firstEllObj.minkdiff_ea(fourthEllObj, dirsMat)

% externalEllVec =
```

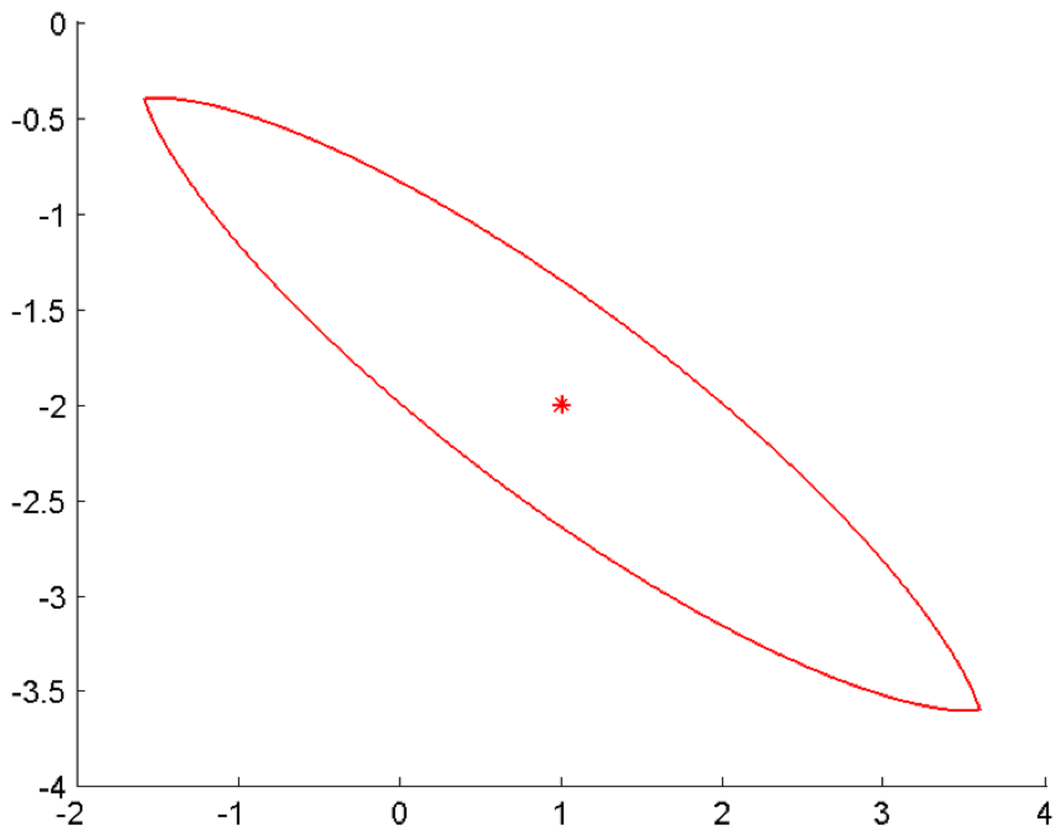
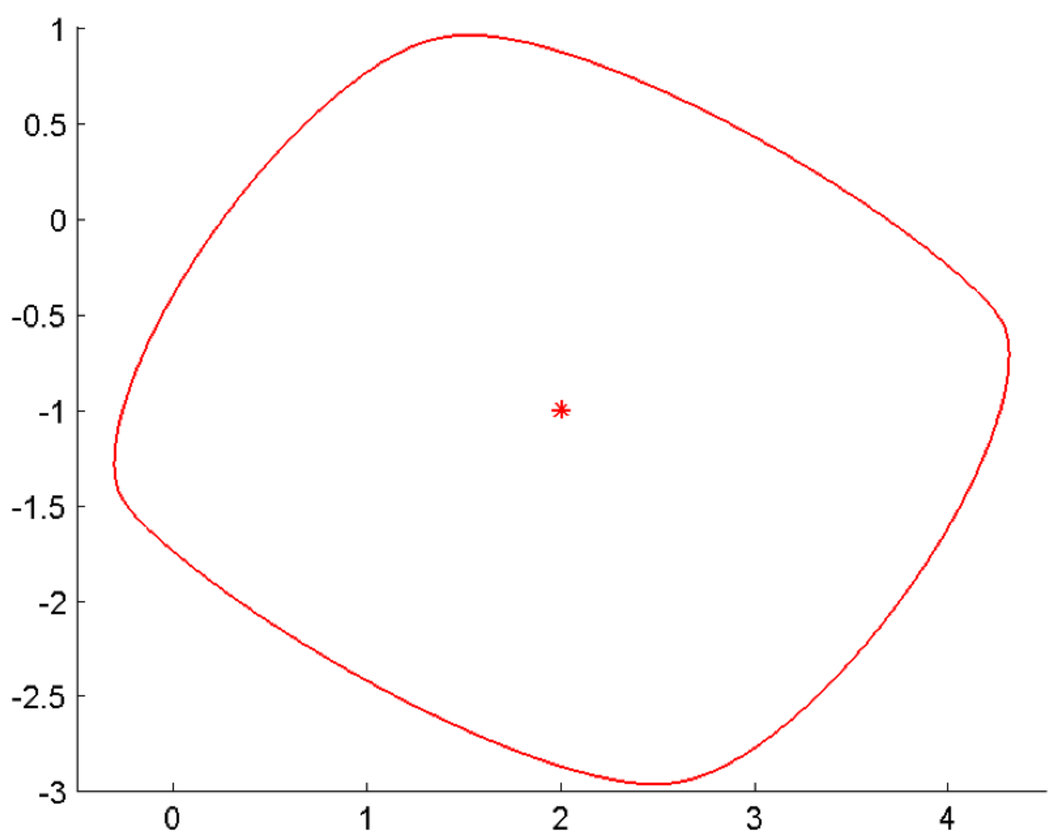


Figure 5.2: The geometric difference of ellipsoids.

```
% Array of ellipsoids with dimensionality 1x3

internalEllVec = firstEllObj.minkdiff_ia(fourthEllObj, dirsMat)

% internalEllVec =
% Array of ellipsoids with dimensionality 1x3
```

Operation 'difference-sum' described in section 2.2.4 is implemented in functions minkmp, minkmp_ea, minkmp_ia, the first one of which is used for visualization and works for dimensions not higher than 3, whereas the last two can deal with ellipsoids of arbitrary dimension.

```
% ellipsoidal approximations for (firstEllObj - thirdEllObj + secEllObj)

% external
externalEllVec = firstEllObj.minkmp_ea(thirdEllObj, secEllObj, dirsMat)
% externalEllVec =
% Array of ellipsoids with dimensionality 1x5

% internal
internalEllVec = firstEllObj.minkmp_ia(thirdEllObj, secEllObj, dirsMat)
% internalEllVec =
% Array of ellipsoids with dimensionality 1x5

% plot the set (firstEllObj - thirdEllObj + secEllObj)
firstEllObj.minkmp(thirdEllObj, secEllObj);
```



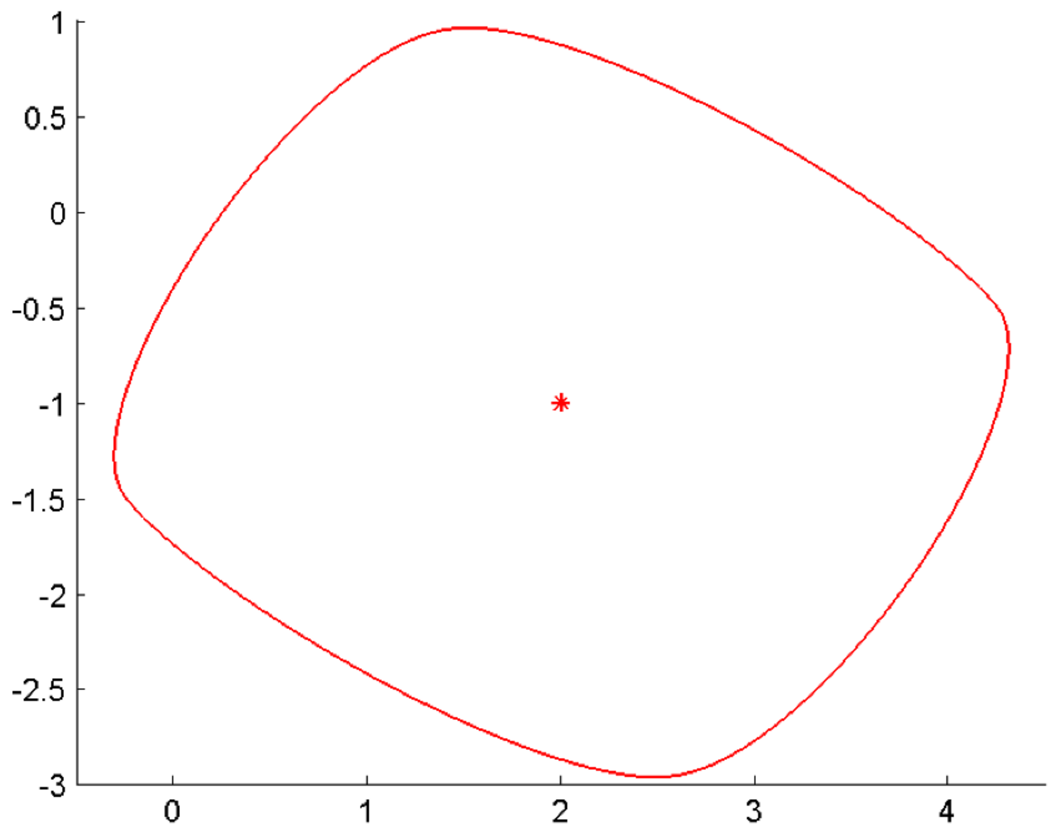

1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
0
1
2
3
4

Figure **??** displays results of the implementation of minkpm and minkmp operations.

Similarly, operation 'sum-difference' described in section 2.2.5 is implemented in functions minkpm, minkpm_ea, minkpm_ia, the first one of which is used for visualization and works for dimensions not higher than 3, whereas the last two can deal with ellipsoids of arbitrary dimension.

```
% ellipsoidal approximations for (firstEllObj + secEllObj - thirdEllObj)
bufEllVec = [firstEllObj secEllObj];
externalEllVec = bufEllVec.minkpm_ea(thirdEllObj, dirsMat)  % external

% externalEllVec =
% Array of ellipsoids with dimensionality 1x5

internalEllVec = bufEllVec.minkpm_ia(thirdEllObj, dirsMat)  % internal

% internalEllVec =
% Array of ellipsoids with dimensionality 1x4

% plot the set (firstEllObj + secEllObj - thirdEllObj)
firstEllObj.minkpm(secEllObj, thirdEllObj)
```

5.2 Operations with hyperplanes

The class hyperplane of the *Ellipsoidal Toolbox* is used to describe hyperplanes and halfspaces. The following two commands define one and the same hyperplane but two different halfspaces:

```
firstHypObj = hyperplane([1; 1], 1);  % defines halfspace x1 + x2 <= 1
firstHypObj = hyperplane([-1; -1], -1);  % defines halfspace x1 + x2 >= 1
```

The following functions and operators are overloaded for the hyperplane class:

- isempty(hypObj) – checks if hypObj is an empty hyperplane.
- display(hypObj) – displays the details of hyperplane $H(c, \gamma)$, namely, its normal c and the scalar γ.
- plot(hypObj) – plots hyperplane $H(c, \gamma)$ if the dimension of the space in which it is defined is not greater than 3.
- firstHypObj == secHypObj – checks if hyperplanes $H(c_1, \gamma_1)$ and $H(c_2, \gamma_2)$ are equal.
- firstHypObj = secHypObj – checks if hyperplanes $H(c_1, \gamma_1)$ and $H(c_2, \gamma_2)$ are not equal.
- [,] – concatenates the hyperplanes into the horizontal array, e.g. hypVec = [firstHypObj secHypObj thirdHypObj].
- [;] – concatenates the hyperplanes into the vertical array, e.g. hypMat = [firstHypObj secHypObj; thirdHypObj fourthHypObj] – defines 2×2 array of hyperplanes.
- -hypObj – defines hyperplane $H(-c, -\gamma)$, which is the same as $H(c, \gamma)$ but specifies different halfspace.

There are several ways to access the internal data of the hyperplane object:

```
[normVec, hypScal] = firstHypObj.double()

% normVec =
%
%    -1
%    -1
%
% hypScal =
```

```
%
%    -1
```

```
firstHypObj.dimension()

% ans =
%
%      2
```

```
% define two hyperplanes passing through the origin
secHypObj = hyperplane([1 -1; 1 1]);
firstHypObj.isparallel(secHypObj)

% ans =
%
%      1     0
```

All the functions of *Ellipsoidal Toolbox* that accept hyperplane object as parameter, work with single hyperplanes as well as with hyperplane arrays. One exception is the function parameters that allows only single hyperplane object.

An array of hyperplanes can be converted to the polytope object of the Multi-Parametric Toolbox (Kvasnica et al. (2004), ("Multi-Parametric Toolbox Homepage")), and back:

```
%define array of four hyperplanes:
hypVec = hyperplane([1 1; -1 -1; 1 -1; -1 1]', [2 2 2 2])

% array of hyperplanes:
% size: [1 4]
%
% Element: [1 1]
% Normal:
%      1
%      1
%
% Shift:
%      2
%
% Hyperplane in R^2.
%
%
% Element: [1 2]
% Normal:
%     -1
%     -1
%
% Shift:
%      2
%
% Hyperplane in R^2.
%
%
% Element: [1 3]
% Normal:
%      1
%     -1
%
% Shift:
%      2
%
```

```
% Hyperplane in R^2.
%
%
% Element: [1 4]
% Normal:
%     -1
%      1
%
% Shift:
%      2
%
% Hyperplane in R^2.

% convert array of hyperplanes to polytope
firstPolObj  = hyperplane2polytope(hypVec);
% covert polytope to array of hyperplanes
convertedHypVec = polytope2hyperplane(firstPolObj);
convertedHypVec == hypVec

% ans =
%
%      1     1     1     1
```

Functions hyperplane2polytope and polytope2hyperplane require the Multi-Parametric Toolbox to be installed.

We can compute distance from ellipsoids to hyperplanes and polytopes:

```
% distance from ellipsoid firstEllObj to each of the hyperplanes in hypVec
firstEllObj.distance(hypVec)

% ans =
%
%      -0.5176    0.8966   -2.6841    0.1444

% distance from each of the ellipsoids in ellMat to the polytope
% firstPolObj
ellMat.distance(firstPolObj)

% ans =
%
%      0     0
%      0     0
```

A negative distance value in the case of ellipsoid and hyperplane means that the ellipsoid intersects the hyperplane. As we see in this example, ellipsoid firstEllObj intersects hyperplanes hypVec(1) and hypVec(3) and has no common points with hypVec(2) and hypVec(4). When distance function has a polytope as a parameter, it always returns non-negative values to be consistent with distance function of the Multi-Parametric Toolbox. Here, the zero distance values mean that each ellipsoid in ellMat has nonempty intersection with polytope firstPolObj.

It can be checked if the union or intersection of given ellipsoids intersects given hyperplanes or polytopes:

```
ellMat.intersect(hypVec, 'u')

% ans =
%
%      1     1     1     1
```

```
ellMat(:, 1).intersect(hypVec, 'i')

```

```
% ans =
%
%     0     0     1     0
```

```
bufEllVec = [firstEllObj secEllObj thirdEllObj];
bufEllVec.intersect(firstPolObj, 'i')

% ans =
%
%     1
```

The intersection of ellipsoid and hyperplane can be computed exactly:

```
% compute the intersections of ellipsoids in the second column of ellMat
% with hyperplane firstHypObj:

intersectEllMat = ellMat(:, 2).hpintersection(firstHypObj)

% intersectEllMat =
% Array of ellipsoids with dimensionality 2x1

intersectEllMat.isdegenerate()  % resulting ellipsoids should lose rank

% ans =
%
%     1
%     1
```

Functions intersection_ea and intersection_ia can be used with hyperplane objects, which in this case define halfspaces and polytope objects:

```
% compute external and internal ellipsoidal approximations
% of the intersections of ellipsoids in the first column of ellMat
% with the halfspace x1 - x2 <= 2:

% get external ellipsoids
firstExternalEllMat = ellMat(:, 1).intersection_ea(firstHypObj(1))
% firstExternalEllMat =
% Array of ellipsoids with dimensionality 2x1

% get internal ellipsoids
firstInternalEllMat = ellMat(:, 1).intersection_ia(firstHypObj(1))
% firstInternalEllMat =
% Array of ellipsoids with dimensionality 2x1

% compute external and internal ellipsoidal approximations
% of the intersections of ellipsoids in the first column of ellMat
% with the halfspace x1 - x2 >= 2:

% get external ellipsoids
secExternalEllMat = ellMat(:, 1).intersection_ea(-firstHypObj(1));

% get internal ellipsoids
secInternalEllMat = ellMat(:, 1).intersection_ia(-firstHypObj(1));
% compute ellipsoidal approximations of the intersection
% of ellipsoid firstEll and polytope firstPol:

% get external ellipsoid
externalEllMat = ellMat(:, 1).intersection_ea(firstPolObj);
```

```
% get internal ellipsoid
internalEllMat = ellMat(:, 1).intersection_ia(firstPolObj);
```

Function isInside can be used to check if a polytope or union of polytopes is contained in the intersection of given ellipsoids:

```
% polytope secPolObj is obtained by affine transformation of firstPolObj
secPolObj = 0.5*firstPolObj + [1; 1];

% check if the intersection of ellipsoids in the first column of ellMat
% contains the union of polytopes firstPolObj and secPolObj:

% equivalent to: doesIntersectionContain(ellMat(:, 1), firstPolObj | secPolObj)
ellMat(:, 1).doesIntersectionContain([firstPolObj secPolObj])

% ans =
%
%      0
```

```
% equivalent to: doesIntersectionContain(ellMat(2, 2),...
%                                    firstPolObj & secPolObj)
ellMat(2, 2).doesIntersectionContain([firstPolObj secPolObj], 'i')

% ans =
%
%      1
```

Functions distance, intersect, intersection_ia and isInside use the CVX interface (("CVX Homepage")) to the external optimization package. The default optimization package included in the distribution of the *Ellipsoidal Toolbox* is SeDuMi (Sturm (1999), ("SeDuMi Homepage")).

5.3 Operations with ellipsoidal tubes

There are several classes in *Ellipsoidal Toolbox* for operations with ellipsoidal tubes. The class gras.ellapx.smartdb.rels.EllTube is used to describe ellipsoidal tubes. The class gras.ellapx.smartdb.rels.EllUnionTube is used to store tubes by the instant of time:

$$\mathcal{X}_U[t] = \bigcup_{\tau \leqslant t} \mathcal{X}[\tau],$$

where $\mathcal{X}[\tau]$ is single ellipsoidal tube. The class gras.ellapx.smartdb.rels.EllTubeProj is used to describe the projection of the ellipsoidal tubes onto time dependent subspaces.There are two types of projection: static and dynamic. Also there is class gras.ellapx.smartdb.rels.EllUnionTubeStaticProj for description of the projection on static plane tubes by the instant of time. Next we provide some examples of the operations with ellipsoidal tubes.

```
nPoints=5;
calcPrecision=0.001;
approxSchemaDescr=char.empty(1,0);
approxSchemaName=char.empty(1,0);
nDims=3;
nTubes=1;
lsGoodDirVec=[1;0;1];
aMat=zeros(nDims,nPoints);
timeVec=1:nPoints;
sTime=nPoints;
```

```
approxType=gras.ellapx.enums.EApproxType.Internal;
qArrayList=repmat({repmat(diag([1 2 3]),[1,1,nPoints])},1,nTubes);
ltGoodDirArray=repmat(lsGoodDirVec,[1,nTubes,nPoints]);
fromMatEllTube=gras.ellapx.smartdb.rels.EllTube.fromQArrays(qArrayList,...
                aMat, timeVec,ltGoodDirArray, sTime, approxType,...
                approxSchemaName, approxSchemaDescr, calcPrecision);
```

```
ellArray(nPoints) = ellipsoid();
approxType=gras.ellapx.enums.EApproxType.Internal;
sTime= 2;
for iElem = 1:nPoints
   ellArray(iElem) = ellipsoid(...
   aMat(:,iElem), qArrayList{1}(:,:,iElem));
end
fromEllArrayEllTube = gras.ellapx.smartdb.rels.EllTube.fromEllArray(...
                ellArray, timeVec,ltGoodDirArray, sTime, approxType,...
                approxSchemaName,approxSchemaDescr, calcPrecision);
```

We may be interested in the data about ellipsoidal tube in some particular time interval, smaller than the one for which the ellipsoidal tube was computed, say $2 \leqslant t \leqslant 4$. This data can be extracted by the cut function:

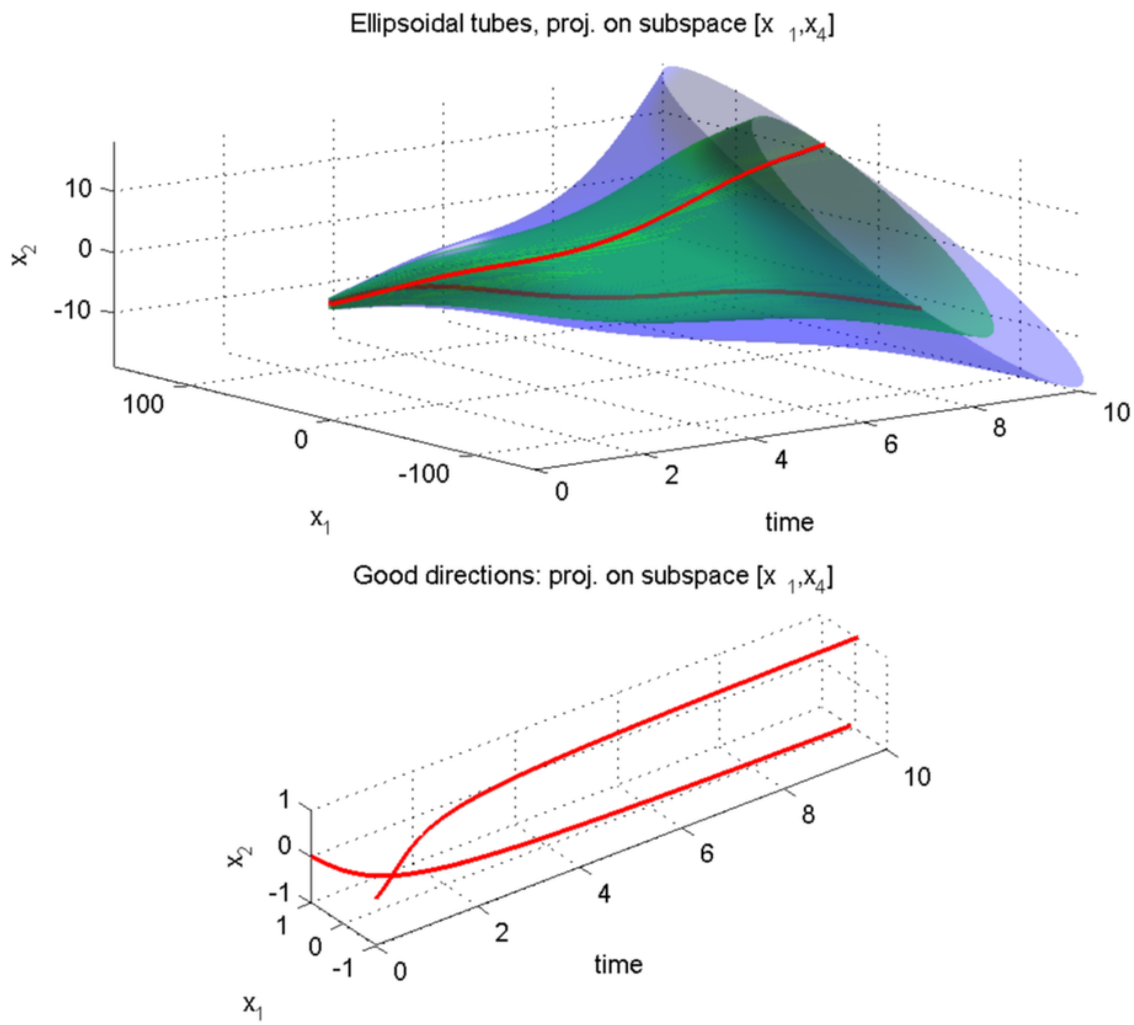
```
cutTimeVec = [2, 4];
cutEllTube = fromMatEllTube.cut(cutTimeVec);
```

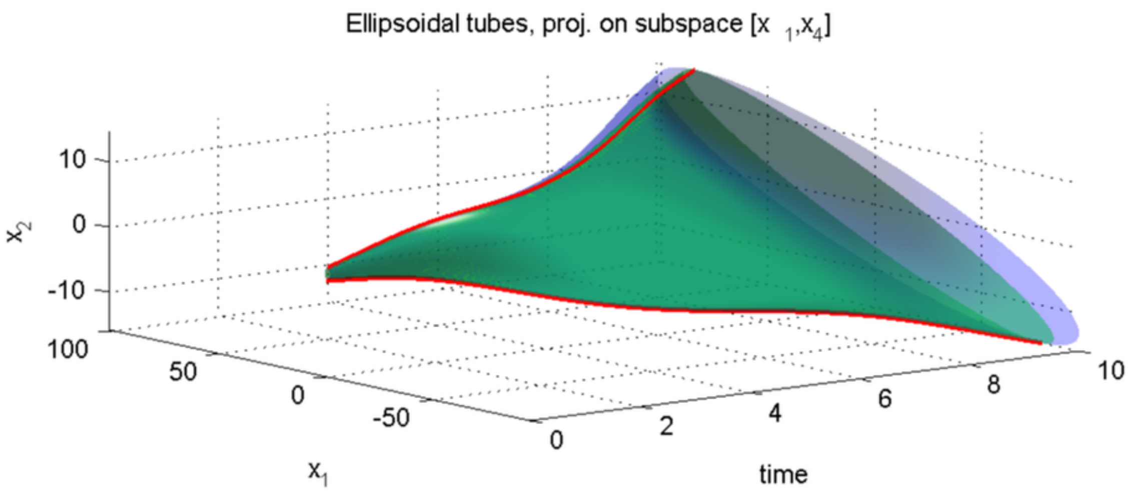
```
function example
   aMat = [0 1; 0 0]; bMat = eye(2);
   SUBounds = struct();
   SUBounds.center = {'sin(t)'; 'cos(t)'};
   SUBounds.shape = [9 0; 0 2];
   sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
   x0EllObj = ell_unitball(2);
   timeVec = [0 10];
   dirsMat = [1 0; 0 1]';
   rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
   ellTubeObj = rsObj.getEllTubeRel();
   projSpaceList = {[1 0;0 1]};
   projType = gras.ellapx.enums.EProjType.Static;
   statEllTubeProj = ellTubeObj.project(projType,projSpaceList,...
      @fGetProjMat);
   projType = gras.ellapx.enums.EProjType.DynamicAlongGoodCurve;
   dynEllTubeProj=ellTubeObj.project(projType,projSpaceList,...
      @fGetProjMat);
   plObj=smartdb.disp.RelationDataPlotter();
   statEllTubeProj.plot(plObj);
   dynEllTubeProj.plot(plObj);

end

function [projOrthMatArray,projOrthMatTransArray]=fGetProjMat(projMat,...
    timeVec,varargin)
  nTimePoints=length(timeVec);
  projOrthMatArray=repmat(projMat,[1,1,nTimePoints]);
  projOrthMatTransArray=repmat(projMat.',[1,1,nTimePoints]);
 end
```

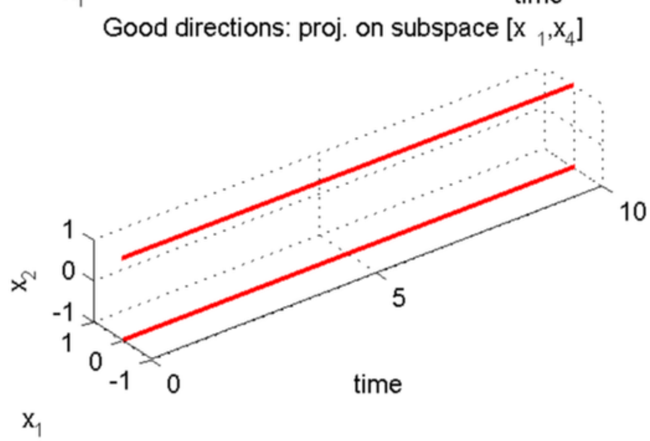
We can compute the projection of the ellipsoidal tube onto time-dependent subspace.

Figure **??** displays static and dynamic projections. Also we can see projections of good directions for ellipsoidal tubes.

We can compute tubes by the instant of time using methodfromEllTubes:


Ellipsoidal tubes, proj. on subspace [x 1,x4]
10
0
-10
x2
100
0
-100
x1
0
2
4
6
8
10
time
Good directions: proj. on subspace [x 1,x4]
1
0
-1
x2
1
0
-1
x1
0
2
4
6
8
10
time


Ellipsoidal tubes, proj. on subspace [x 1,x4]
x2
10
0
-10
100
50
0
-50
x1
0
2
4
6
8
10
time


Good directions: proj. on subspace [x 1,x4]
x2
1
0
-1
1
0
-1
x1
0
5
10
time

```
function example
   aMat = [0 1; 0 0]; bMat = eye(2);
   SUBounds = struct();
   SUBounds.center = {'sin(t)'; 'cos(t)'};
   SUBounds.shape = [9 0; 0 2];
   sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
   x0EllObj = ell_unitball(2);
   timeVec = [0 10];
   dirsMat = [1 0; 0 1]';
   rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
   ellTubeObj = rsObj.getEllTubeRel();
   unionEllTube = ...
    gras.ellapx.smartdb.rels.EllUnionTube.fromEllTubes(ellTubeObj);
   projSpaceList = {[1 0;0 1]};
   projType = gras.ellapx.enums.EProjType.Static;
   statEllTubeProj = unionEllTube.project(projType,projSpaceList,...
      @fGetProjMat);
   plObj=smartdb.disp.RelationDataPlotter();
   statEllTubeProj.plot(plObj);
end

function [projOrthMatArray,projOrthMatTransArray]=fGetProjMat(projMat,...
    timeVec,varargin)
  nTimePoints=length(timeVec);
  projOrthMatArray=repmat(projMat,[1,1,nTimePoints]);
  projOrthMatTransArray=repmat(projMat.',[1,1,nTimePoints]);
 end
```

Figure **??** shows projection of ellipsoidal tubes by the instant of time.

Also we can get initial data from the resulting tube:

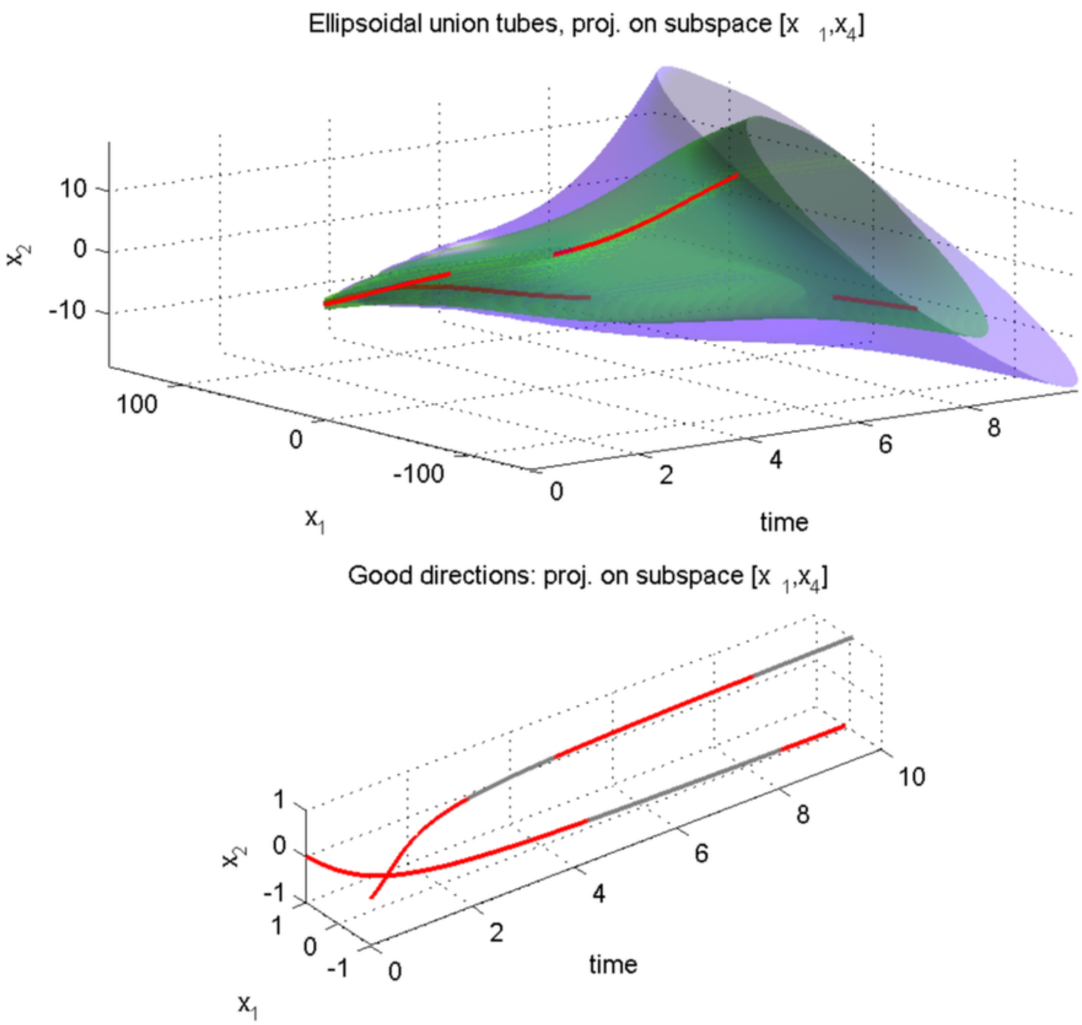
```
approxType=gras.ellapx.enums.EApproxType.Internal;
ellArray = fromEllArrayEllTube.getEllArray(approxType)

% ellArray =
% Array of ellipsoids with dimensionality 5x1
```

There is a method to display a content of ellipsoidal tubes.

```
aMat = [0 1; 0 0]; bMat = eye(2);
SUBounds = struct();
SUBounds.center = {'sin(t)'; 'cos(t)'};
SUBounds.shape = [9 0; 0 2];
sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
x0EllObj = ell_unitball(2);
timeVec = [0 10];
for iElem = 1:5
    dirInitial= rand(2, 1);
    dirInitial = dirInitial ./ norm(dirInitial);
    dirsMat(:, iElem) = dirInitial;
end
rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
ellTubeObj = rsObj.getEllTubeRel();
ellTubeObj.dispOnUI();
```

Figure 5.3 displays all fields of the ellipsoidal tube.

There are several methods to find the tubes with necessary parameters.


Ellipsoidal union tubes, proj. on subspace [x 1,x4]
x2
x1
time
Good directions: proj. on subspace [x 1,x4]
x2
x1
time

	QArray	aMat	scaleFactor	MArray	dim	
1	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
2	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
3	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
4	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
5	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
6	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
7	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
8	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
9	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0
10	double[2x2x100]	double[2x100]	1	double[2x2x100]	2	0

```
newEllTube = fromMatEllTube.getTuplesFilteredBy('sTime', 5);
newEllTube.getNTuples()
%
% ans =
%
%      1
%
newEllTube = fromMatEllTube.getTuplesFilteredBy('sTime', 2);
newEllTube.getNTuples()
%
% ans =
%
%      0
%
```

Also you can use the method display to see the result of the method's work.

```
fromMatEllTube.getNTuples()
%
% ans =
%
%      1
%
fromEllArrayEllTube.getNTuples()
%
% ans =
%
%      1
%
origFromMatEllTube=fromMatEllTube.getCopy();
fromMatEllTube.unionWith(fromEllArrayEllTube);
%
% ans =
%
%      2
%
fromMatEllTube.getNTuples()
isOk=fromMatEllTube.getTuples(1).isEqual(origFromMatEllTube)
%
% isOk =
%
%     1

isOk=fromMatEllTube.getTuples(2).isEqual(fromEllArrayEllTube)

%
% isOk =
%
%     1
```

We can sort our tubes by certain fields:

```
fromMatEllTube.display();
fromMatEllTube.sortBy('sTime');
fromMatEllTube.display();
```

5.4 Reachability

To compute the reach sets of the systems described in chapter 3, we define few new classes in the *Ellipsoidal Toolbox*: class LinSysContinuous for the continuous-time system description, class LinSysDiscrete for the discrete-time system description and classes ReachContinuous\ReachDiscrete for the reach set data. We start by explaining how to define a system using LinSysContinuous object. Also we can use LinSysFactory class for the description of this system. Through it's method create user can get LinSysContinuous or LinSysDiscrete object. For example, description of the system

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \quad u(t) \in \mathcal{E}(p(t), P)$$

with

$$p(t) = \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}, \quad P = \begin{bmatrix} 9 & 0 \\ 0 & 2 \end{bmatrix},$$

is done by the following sequence of commands:

```
aMat = [0 1; 0 0]; bMat = eye(2);  % matrices A and B, B is identity
SUBounds = struct();
% center of the ellipsoid depends on t
SUBounds.center = {'sin(t)'; 'cos(t)'};
SUBounds.shape = [9 0; 0 2]; % shape matrix of the ellipsoid is static
% create linear system object
sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
% is equal to sys = elltool.linsys.LinSysFactory.create(aMat, bMat, SUBounds)
```

If matrices A or B depend on time, say $A(t) = \begin{bmatrix} 0 & 1 - \cos(2t) \\ -\frac{1}{t} & 0 \end{bmatrix}$, then matrix aMat should be symbolic:

```
atMat = {'0' '1 - cos(2*t)'; '-1/t' '0'};
sys_t = elltool.linsys.LinSysFactory.create(atMat, bMat, SUBounds);
```

To describe the system with disturbance

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v(t),$$

with bounds on control as before, and disturbance being $-1 \leqslant v(t) \leqslant 1$, we type:

```
gMat = [0; 1];  % matrix G
vEllObj = ellipsoid(1);  % disturbance bounds: unit ball in R
sys_d = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds,...
    gMat, vEllObj);
```

Control and disturbance bounds SUBounds and vEllObj can have different types. If the bound is constant, it should be described by ellipsoid object. If the bound depends on time, then it is represented by a structure with fields center and shape, one or both of which are symbolic. In system sys, the control bound SUBounds is defined as such a structure. Finally, if the control or disturbance is known and fixed, it should be defined as a vector, of type double if constant, or symbolic, if it depends on time.

To declare a discrete-time system

$$\begin{bmatrix} x_1[k+1] \\ x_2[k+1] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -0.5 \end{bmatrix} \begin{bmatrix} x_1[k] \\ x_2[k] \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u[k], \quad -1 \leqslant u[k] \leqslant 1,$$

we use LinSysDiscrete constructor:

```
adMat = [0 1; -1 -0.5]; bdMat = [0; 1];  % matrices A and B
udBoundsEllObj  = ellipsoid(1);  % control bounds: unit ball in R
% discrete-time system
dtsys = elltool.linsys.LinSysDiscrete(adMat, bdMat, udBoundsEllObj);
% is equal to dtsys = elltool.linsys.LinSysFactory.create(adMat, bdMat,...
%         udBoundsEllObj,...[], [], [], [], 'd');
```

Once the LinSysDiscrete object is created, we need to specify the set of initial conditions, the time interval and values of the direction vector, for which the reach set approximations must be computed:

```
x0EllObj = ell_unitball(2);  % set of initial conditions
timeVec = [0 10];  % time interval
dirsMat = [1 0; 0 1]';  % columns of L specify the directions
```

The reach set approximation is computed by calling the constructor of the ReachContinuous object:

```
% reach set of continuos-time system
firstRsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
```

At this point, variable firstRsObj contains the reach set approximations for the specified continuous-time system, time interval and set of initial conditions computed for given directions. Both external and internal approximations are computed. The reach set approximation data can be extracted in the form of arrays of ellipsoids:

```
externallEllMat = firstRsObj.get_ea()  % external approximating ellipsoids

% externallEllMat =
% Array of ellipsoids with dimensionality 2x100

% internal approximating ellipsoids
[internalEllMat, timeVec] = firstRsObj.get_ia();
```

Ellipsoidal arrays externallEllMat and internalEllMat have 4 rows because we computed the reach set approximations for 4 directions. Each row of ellipsoids corresponds to one direction. The number of columns in externallEllMat and internalEllMat is defined by the nTimeGridPoints parameter, which is available from elltool.conf.Properties static class (see chapter 6 for details). It represents the number of time values in our time interval, at which the approximations are evaluated. These time values are returned in the optinal output parameter, array timeVec, whose length is the same as the number of columns in externallEllMat and internalEllMat. Intersection of ellipsoids in a particular column of externallEllMat gives external ellipsoidal approximation of the reach set at corresponding time. Internal ellipsoidal approximation of this set at this time is given by the union of ellipsoids in the same column of internalEllMat.

We may be interested in the reachability data of our system in some particular time interval, smaller than the one for which the reach set was computed, say $3 \leqslant t \leqslant 5$. This data can be extracted and returned in the form of ReachContinuous object by the cut function:

```
cutObj = firstRsObj.cut([3 5]);  % reach set for the time interval [3, 5]
```

To obtain a snap shot of the reach set at given time, the same function cut is used:

```
cutObj = firstRsObj.cut(5);  % reach set at time t = 5
```

It can be checked if the external or internal reach set approximation intersects with given ellipsoids, hyperplanes or polytopes:

```
ellObj = ellipsoid([-17; 0], [4 -1; -1 1]);  % define ellipsoid
% define 4 hyperplanes
hypVec = hyperplane([1 1; -1 -1; 1 -1; -1 1]', [2 2 2 2]);
polObj = hyperplane2polytope(hypVec) + [2; 10];  % define polytope
% check if ellipsoid ell intersects with external approximation:
cutObj.intersect(ellObj, 'e')
```

```
% ans =
%
%       1
```

```
% check if ellipsoid ellObj intersects with internal approximation:
cutObj.intersect(ellObj, 'i')

%
% ans =
%
%       1
```

```
% check if hyperplanes in hypVec intersect with internal approximation:
cutObj.intersect(hypVec, 'i')

% ans =
%
%       1     1     1     1
```

```
% check if polytope polObj intersects with external approximation:
cutObj.intersect(polObj)

% ans =
%
%       0
```

If a given set intersects with the internal approximation of the reach set, then this set intersects with the actual reach set. If the given set does not intersect with external approximation, this set does not intersect the actual reach set. There are situations, however, when the given set intersects with the external approximation but does not intersect with the internal one. In our example above, ellipsoid ellObj is such a case: the quality of the approximation does not allow us to determine whether or not ellObj intersects with the actual reach set. To improve the quality of approximation, refine function should be used:

```
% define new directions, in this case one, but could be more
newDirsMat = [1; -1];
% compute approximations for the new directions
firstRsObj = firstRsObj.refine(newDirsMat);
% snap shot of the reach set at time t = 5
cutObj = firstRsObj.cut(5);
% check if ellObj intersects the internal approximation
cutObj.intersect(ellObj, 'i')

% ans =
%
%       1
```

Now we are sure that ellipsoid ellObj intersects with the actual reach set. However, to use the refine function, the reach set object must contain all calculated data, otherwise, an error is returned.

Having a reach set object resulting from the ReachContinuous, cut or refine operations, we can obtain the trajectory of the center of the reach set and the good curves along which the actual reach set is touched by its ellipsoidal approximations:

```
[ctrMat, ttVec] = firstRsObj.get_center(); % trajectory of the center
gcCVec = firstRsObj.get_goodcurves()  % get good curves

% gcCVec =
% [2x100 double] [2x100 double] [2x100 double]
```

Variable ctrMat here is a matrix whose columns are the points ofthe reach set center trajectory evaluated at time values returned in the array ttVec. Variable gcCMat contains 4 matrices each of which corresponds to a good curve (columns of such matrix are points of the good curve evaluated at time values in ttVec). The analytic expression for the control driving the system along a good curve is given by formula (**??**).

We computed the reach set up to time 10. It is possible to continue the reach set computation for a longer time horizon using the reach set data at time 10 as initial condition. It is also possible that the dynamics and inputs of the system change at certain time, and from that point on the system evolves according to the new system of differential equations. For example, starting at time 10, our reach set may evolve in time according to the time-variant system sys_t defined above. Switched systems are a special case of this situation. To compute the further evolution in time of the existing reach set, function evolve should be used:

```
% reach set from time 10 to 14 with the same dynamics
secRsObj = firstRsObj.evolve(14);
% reach set from time 10 to 12 with new dynamics
secRsObj = firstRsObj.evolve(12, sys_t);

% not only the dynamics, but the inputs can change as well,
% from time 12 to 13 disturbance is added to the system:

% sys_d - system with disturbance defined above
thirdRsObj = secRsObj.evolve(13, sys_d);
```

Function evolve can be viewed as an implementation of the semigroup property.

To compute the backward reach set for some specified target set, we declare the time interval so that the terminating time comes first:

```
% target set in the form of ellipsoid
yEllObj = ellipsoid([8; 2], [4 1; 1 2]);
tbTimeVec = [10 5];  % backward time interval
% backward reach set
firstBrsObj = elltool.reach.ReachContinuous(sys, yEllObj, dirsMat,...
        tbTimeVec);
firstBrsObj = firstBrsObj.refine(newDirsMat);  % refine the approximation
% further evolution in backward time from 5 to 0;
secBrsObj = firstBrsObj.evolve(0)
```

Reach set and backward reach set computation for discrete-time systems and manipulations with the resulting reach set object are performed using the same functions as for continuous-time systems:

```
timeVec = [0 100];  % represents 100 time steps from 1 to 100
% reach set for 100 time steps
secDtrsObj = elltool.reach.ReachDiscrete(dtsys, x0EllObj, dirsMat, timeVec);
secDtrsObj = secDtrsObj.evolve(200);  % compute next 100 time steps

tbTimeVec = [50 0];  % backward time interval
% backward reach set
dtbrsObj = elltool.reach.ReachDiscrete(dtsys, yEllObj, dirsMat, tbTimeVec);
dtbrsObj = dtbrsObj.refine(newDirsMat);  % refine the approximation
% get external approximating ellipsoids and time values
[externallEllMat, timeVec] = dtbrsObj.get_ea();
% get internal approximating ellipsoids
internalEllMat = dtbrsObj.get_ia()

% internalEllMat =
% Array of ellipsoids with dimensionality 3x51
```

Number of columns in the ellipsoidal arrays externalEllMat and internalEllMat is 51 because the backward reach set is computed for 50 time steps, and the first column of these arrays contains 3 ellipsoids yEllObj - the terminating condition.

When dealing with discrete-time systems, all functions that accept time or time interval as an input parameter, round the time values and treat them as integers.

5.5 Properties

Functions of the *Ellipsoidal Toolbox* can be called with user-specified values of certain global parameters. System of the parameters are configured using xml files, which available from a set of command-line utilities:

```
elltool.setconf('default');
```

Here we list system parameters available from the 'default' configuration:

1. version = '1.4dev' - current version of *ET*.
2. isVerbose = false - makes all the calls to *ET* routines silent, and no information except errors is displayed.
3. absTol = 1e-7 - absolute tolerance.
4. relTol = 1e-5 - relative tolerance.
5. nTimeGridPoints = 200 - density of the time grid for the continuous time reach set computation. This parameter directly affects the number of ellipsoids to be stored in the ReachContinuous\ReachDiscrete object.
6. ODESolverName = ode45 - specifies the ODE solver for continuous time reach set computation.
7. isODENormControl = 'on' - switches on and off the norm control in the ODE solver. When turned on, it slows down the computation, but improves the accuracy.
8. isEnabledOdeSolverOptions = false - when set to false, calls the ODE solver without any additional options like norm control. It makes the computation faster but less accurate. Otherwise, it is assumed to be true, and only in this case the previous option makes a difference.
9. nPlot2dPoints = 200 - the number of points used to plot a 2D ellipsoid. This parameter also affects the quality of 2D reach tube and reach set plots.
10. nPlot3dPoints = 200 - the number of points used to plot a 3D ellipsoid. This parameter also affects the quality of 3D reach set plots.

Once the configuration is loaded, the system parameters are available through elltool.conf.Properties. elltool.conf.Properties is a static class, providing emulation of static properties for toolbox. It has two function types: setters and getters. Using getters we obtain system parameters.

```
elltool.conf.Properties.getAbsTol()
% ans =
%
%    1.0000e-07

elltool.conf.Properties.getNPlot2dPoints()

% ans =
%
%    200
```

Some of the parameters can be changed in run-time via setters.

```
elltool.conf.Properties.setNTimeGridPoints(250);
```

5.6 Visualization

Ellipsoidal Toolbox has several plotting routines:

- ellipsoid/plot - plots one or more ellipsoids, or arrays of ellipsoids, defined in $\mathbf{R}$, $\mathbf{R}^2$ or $\mathbf{R}^3$.
- ellipsoid/minksum - plots geometric sum of finite number of ellipsoids defined in $\mathbf{R}$, $\mathbf{R}^2$ or $\mathbf{R}^3$.
- ellipsoid/minkdiff - plots geometric difference (if it is not an empty set) of two ellipsoids defined in $\mathbf{R}$, $\mathbf{R}^2$ or $\mathbf{R}^3$.
- ellipsoid/minkmp - plots geometric (Minkowski) sum of the geometric difference of two ellipsoids and the geometric sum of n ellipsoids defined in $\mathbf{R}$, $\mathbf{R}^2$ or $\mathbf{R}^3$.
- ellipsoid/minkpm - plots geometric (Minkowski) difference of the geometric sum of ellipsoids and a single ellipsoid defined in $\mathbf{R}$, $\mathbf{R}^2$ or $\mathbf{R}^3$.
- hyperplane/plot - plots one or more hyperplanes, or arrays of hyperplanes, defined in $\mathbf{R}^2$ or $\mathbf{R}^3$.
- reach/plot_ea - plots external approximation of the reach set whose dimension is 2 or 3.
- reach/plot_ia - plots internal approximation of the reach set whose dimension is 2 or 3.

All these functions allow the user to specify the color of the plotted objects, line width for 1D and 2D plots, and transparency level of the 3D objects. Hyperplanes are displayed as line segments in 2D and square facets in 3D. In the hyperplane/plot method it is possible to specify the center of the line segment or facet and its size.

Ellipsoids of dimensions higher than three must be projected onto a two- or three-dimensional subspace before being plotted. This is done by means of projection function:

```
% create two 4-dimensional ellipsoids:
firstEllObj = ellipsoid([14 -4 2 -5; -4 6 0 1; 2 0 6 -1; -5 1 -1 2]);
secEllObj = firstEllObj.getInv();

% specify 3-dimensional subspace by its basis:

% columns of basisMat must be orthogonal
basisMat = [1 0 0 0; 0 0 1 0; 0 1 0 1].';

% get 3-dimensional projections of firstEllObj and secEllObj:
bufEllVec = [firstEllObj secEllObj];
% array ellVec contains projections of firstEllObj and secEllObj
ellVec = bufEllVec.projection(basisMat)

% ellVec =
% Array of ellipsoids with dimensionality 1x2

ellVec.plot();  % plot ellipsoids in ellVec
```

Since the operation of projection is linear, the projection of the geometric sum of ellipsoids equals the geometric sum of the projected ellipsoids. The same is true for the geometric difference of two ellipsoids.

Function projection exists also for the ReachContinuous\ReachDiscrete objects:

```
aMat = [0 1 0 0; -1 0 1 0; 0 0 0 1; 0 0 -1 0];
bMat = [0; 0; 0; 1];
uBoundsEllObj = ellipsoid(1);
% 4-dimensional system
sys = elltool.linsys.LinSysFactory.create(aMat, bMat, uBoundsEllObj);
dirsMat  = [1 1 0 1; 0 -1 1 0; -1 1 1 1; 0 0 -1 1].'; % matrix of directions
% reach set from time 0 to 5
rsObj = elltool.reach.ReachContinuous(sys, ell_unitball(4), dirsMat,...
```

```
    [0 5], 'isRegEnabled', true, 'isJustCheck', false, 'regTol', 1e-4);
basisMat = [1 0 0 1; 0 1 1 0].';  % basis of 2-dimensional subspace

% project reach set rs onto basis basisMat
psObj = rsObj.projection(basisMat);
psObj.plotByEa();  % plot external approximation
hold on;
psObj.plotByIa();  % plot internal approximation
```

The quality of the ellipsoid and reach set plots is controlled by the parameters nPlot2dPoints and nPlot3dPoints, which are available from getters of ellipsoid class.

"CVX Homepage." cvxr.com/cvx.

"Multi-Parametric Toolbox Homepage." control.ee.ethz.ch/\~mpt.

"SeDuMi Homepage." sedumi.mcmaster.ca.

"Stanley Chan Article Homepage." http://videoprocessing.ucsd.edu/~stanleychan/publication/unpublished/Ellipse.pdf.

Kvasnica, M., P. Grieder, M. Baotić, and M. Morari. 2004. "Multi-Parametric Toolbox (MPT)." In *Hybrid Systems: Computation and Control*, edited by R. Alur and G. J. Pappas, 2993:448–462. Springer.

Lin, A., and S. Han. 2002. "On the Distance Between Two Ellipsoids." *SIAM Journal on Optimization* 13 (1): 298–308.

Sturm, J. F. 1999. "Using SeDuMi 1.02, A MATLAB Toolbox for Optimization over Symmetric Cones." *Optimization Methods and Software* 11-12: 625–653.

L. Ros, A. Sabater, F. Thomas. 2002. "An Ellipsoidal Calculus Based on Propagation and Fusion." *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics* 32 (4).

CHAPTER

SIX

EXAMPLES

6.1 Ellipsoids vs. Polytopes

Depending on the particular dynamical system, certain methods of reach set computation may be more suitable than others. Even for a simple 2-dimensional discrete-time linear time-invariant system, application of ellipsoidal methods may be more effective than using polytopes.

Consider the system from chapter 1:

$$\left[\begin{array}{c} x_1[k+1] \\ x_2[k+1] \end{array}\right] = \left[\begin{array}{cc} \cos(1) & \sin(1) \\ -\sin(1) & \cos(1) \end{array}\right] \left[\begin{array}{c} x_1[k] \\ x_2[k] \end{array}\right] + \left[\begin{array}{c} u_1[k] \\ u_2[k] \end{array}\right], \quad x[0] \in \mathcal{X}_0, \quad u[k] \in U, \quad k \geqslant 0,$$

where $\mathcal{X}_0$ is the set of initial conditions, and U is the control set.

Let $\mathcal{X}_0$ and U be unit boxes in $\mathbf{R}^2$, and compute the reach set using the polytope method implemented in MPT ("Multi-Parametric Toolbox Homepage"). With every time step the number of vertices of the reach set polytope increases by 4. The complexity of the convex hull computation increases exponentially with number of vertices. In figure 6.1, the time required to compute the reach set for different time steps using polytopes is shown in red.

To compute the reach set of the system using *Ellipsoidal Toolbox*, we assume $\mathcal{X}_0$ and U to be unit balls in $\mathbf{R}^2$, fix any number of initial direction values that corresponds to the number of ellipsoidal approximations, and obtain external and internal ellipsoidal approximations of the reach set:

```
aMat = [cos(1) sin(1); -sin(1) cos(1)];
uBoundsEllObj = ell_unitball(2);  % control bounds
% define linear discrete-time system
lsys = elltool.linsys.LinSysFactory.create(aMat, eye(2), uBoundsEllObj,...
    [], [], 'd');
x0EllObj = ell_unitball(2);  % set of initial conditions
dirsMat = [cos(0:0.1:pi); sin(0:0.1:pi)];  % 32 initial directions
nSteps  = 100;  % number of time steps

% compute the reach set
rsObj = elltool.reach.ReachDiscrete(lsys, x0EllObj, dirsMat, [0 nSteps]);
```

In figure 6.1, the time required to compute both external and internal ellipsoidal approximations, with 32 ellipsoids each, for different number of time steps is shown in blue.

Figure 6.1 illustrates the fact that the complexity of polytope method grows exponentially with number of time steps, whereas the complexity of ellipsoidal method grows linearly.

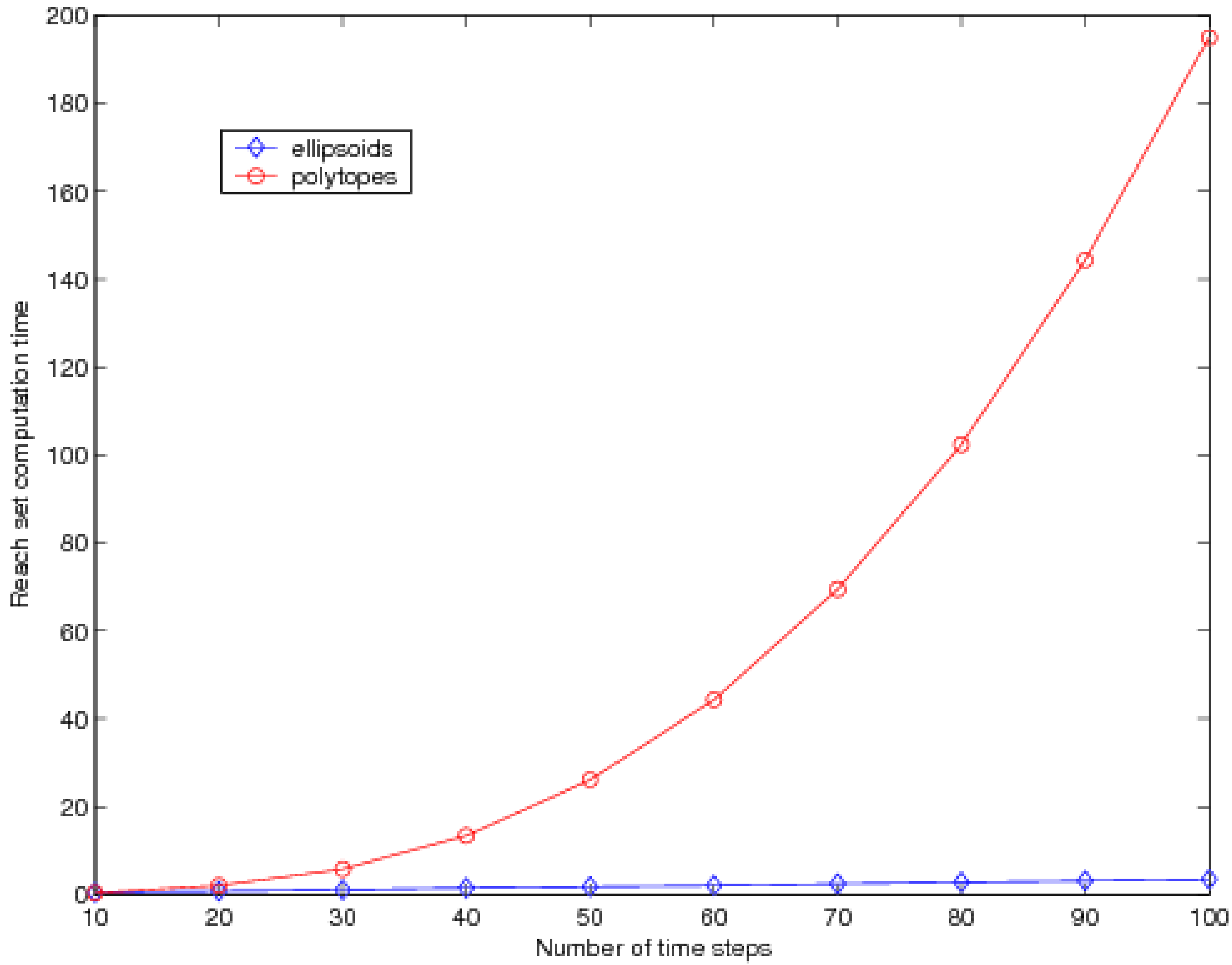


Figure 6.1: Reach set computation performance comparison.

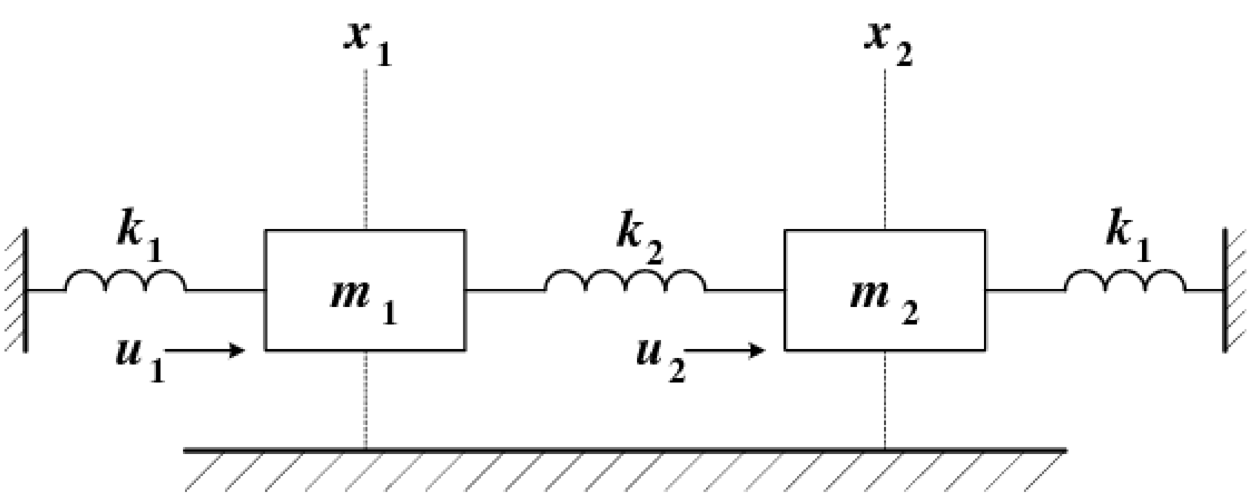


Figure 6.2: Spring-mass system.

6.2 System with Disturbance

The mechanical system presented in figure 6.2, is described by the following system of equations:

$$m_1\ddot{x}_1 + (k_1 + k_2)x_1 - k_2x_2 = u_1, \tag{6.1}$$

$$m_2\ddot{x}_2 - k_2x_1 + (k_1 + k_2)x_2 = u_2. \tag{6.2}$$

Here u_1 and u_2 are the forces applied to masses m_1 and m_2, and we shall assume $[u_1 \;\; u_2]^T \in \mathcal{E}(0, I)$. The initial conditions can be taken as $x_1(0) = 0$, $x_2(0) = 2$. Defining $x_3 = \dot{x}_1$ and $x_4 = \dot{x}_2$, we can rewrite (6.1)-(6.2) as a linear system in standard form:

$$\left[\begin{array}{c} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{array}\right] = \left[\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_1+k_2}{m_1} & \frac{k_2}{m_1} & 0 & 0 \\ \frac{k_2}{m_2} & -\frac{k_1+k_2}{m_2} & 0 & 0 \end{array}\right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right] + \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ \frac{1}{m_1} & 0 \\ 0 & \frac{1}{m_2} \end{array}\right] \left[\begin{array}{c} u_1 \\ u_2 \end{array}\right]. \tag{6.3}$$

Now we can compute the reach set of system (6.1)-(6.2) for given time by computing the reach set of the linear system (6.3) and taking its projection onto (x_1, x_2) subspace.

```
k1 = 24;  k2 = 32;
m1 = 1.5; m2 = 1;
% define matrices aMat, bMat, and control bounds uBoundsEll:
aMat = [0 0 1 0; 0 0 0 1; -(k1+k2)/m1 k2/m1 0 0; k2/m2 -(k1+k2)/m2 0 0];
bMat = [0 0; 0 0; 1/m1 0; 0 1/m2];
uBoundsEllObj = ell_unitball(2);
% linear system
lsys = elltool.linsys.LinSysContinuous(aMat, bMat, uBoundsEllObj);
timeVec = [0 4];  % time interval% initial conditions:
x0EllObj = [0 2 0 0].' + ellipsoid([0.01 0 0 0; 0 0.01 0 0; 0 0 0 0;...
           0 0 0 0]);
% initial directions (some random vectors in R^4):
dirsMat = [1 0 1 0; 1 -1 0 0; 0 -1 0 1; 1 1 -1 1; -1 1 1 0; -2 0 1 1].';
% reach set
rsObj = elltool.reach.ReachContinuous(lsys, x0EllObj, dirsMat, timeVec,...
    'isRegEnabled', true, 'isJustCheck', false, 'regTol', 1e-3);
basisMat = [1 0 0 0; 0 1 0 0]';  % orthogonal basis of (x1, x2) subspace
psObj = rsObj.projection(basisMat);  % reach set projection
% plot projection of reach set external approximation:

psObj.plotByEa('g');  % plot the whole reach tube

%
% ReachContinuous's cut() doesn't work with projections:
psObj = psObj.cut(4);
psObj.plotByEa('g');  % plot reach set approximation at time t = 4
```

Figure 6.3 (a) shows the reach set of the system (6.1)-(6.2) evolving in time from $t = 0$ to $t = 4$. Figure 6.3 (b) presents a snapshot of this reach set at time $t = 4$.

So far we considered an ideal system without any disturbance, such as friction. We introduce disturbance to (6.1)-(6.2) by adding extra terms, v_1 and v_2,

$$m_1\ddot{x}_1 + (k_1 + k_2)x_1 - k_2x_2 = u_1 + v_1, \tag{6.4}$$

$$m_2\ddot{x}_2 - k_2x_1 + (k_1 + k_2)x_2 = u_2 + v_2, \tag{6.5}$$

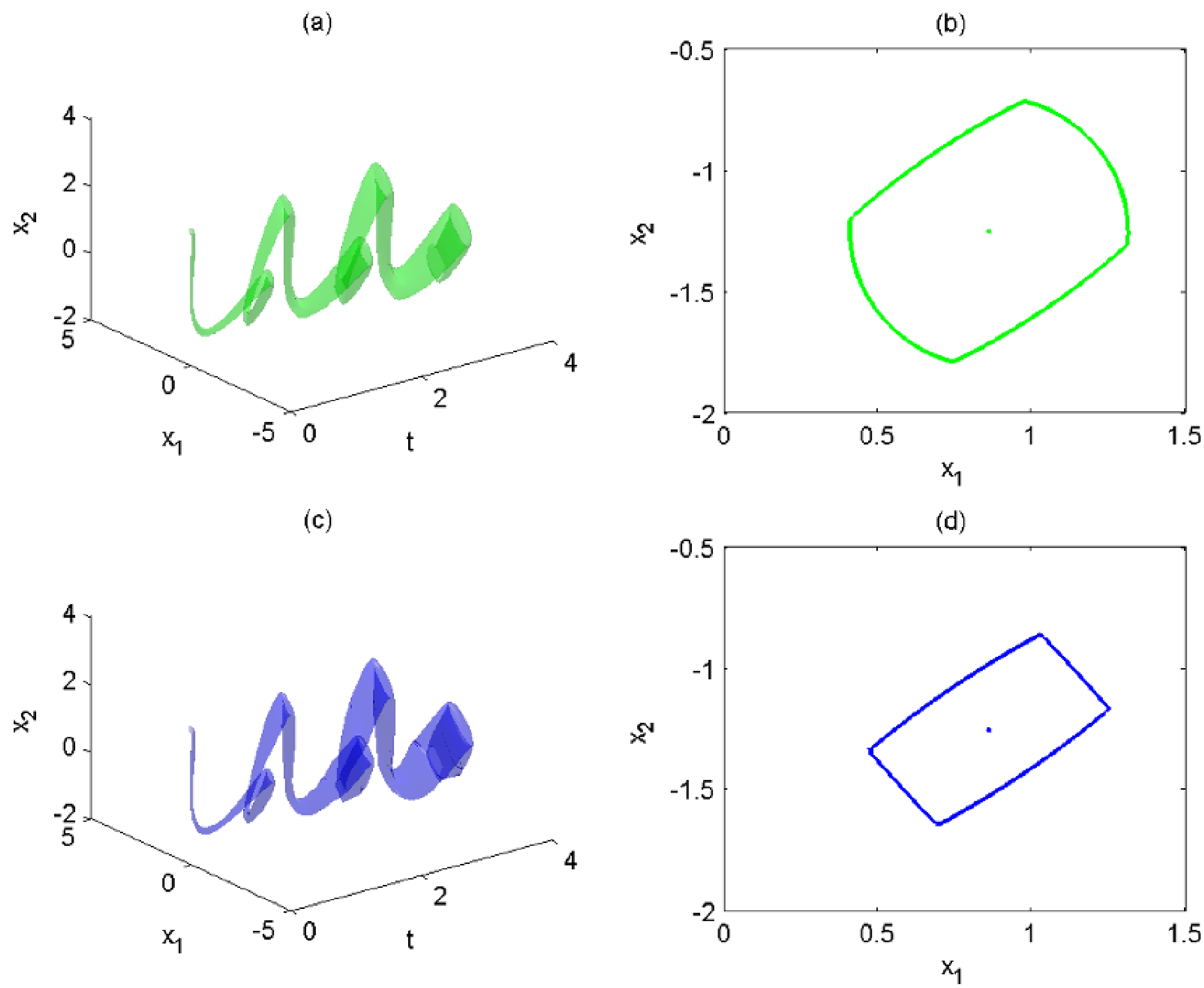


Figure 6.3: Spring-mass system without disturbance: (a) reach tube for time $t \in [0, 4]$; (b) reach set at time $t = 4$. Spring-mass system with disturbance: (c) reach tube for time $t \in [0, 4]$; (d) reach set at time $t = 4$.

which results in equation (6.3) getting an extra term

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}.$$

Assuming that $[v_1 \;\; v_2]^T$ is unknown but bounded by ellipsoid $\mathcal{E}(0, \frac{1}{4}I)$, we can compute the closed-loop reach set of the system with disturbance.

```
% define disturbance:
gMat = [0 0; 0 0; 1 0; 0 1];
vEllObj = 0.05*ell_unitball(2);
% linear system with disturbance
lsysd = elltool.linsys.LinSysContinuous(aMat, bMat, uBoundsEllObj,...
    gMat, vEllObj);
% reach set
rsdObj = elltool.reach.ReachContinuous(lsysd, x0EllObj, dirsMat,...
    timeVec, 'isRegEnabled', true, 'isJustCheck', false, 'regTol', 1e-1);
psdObj = rsdObj.projection(basisMat);  % reach set projection onto (x1, x2)
% plot projection of reach set external approximation:
psdObj.plotEa();  % plot the whole reach tube
psdCutObj = psdObj.cut(4);
psdCutObj.plotEa();  % plot reach set approximation at time t = 4
```

Figure **??**-(6.5) evolving in time from $t = 0$ to $t = 4$. Figure **??**.

6.3 Switched System

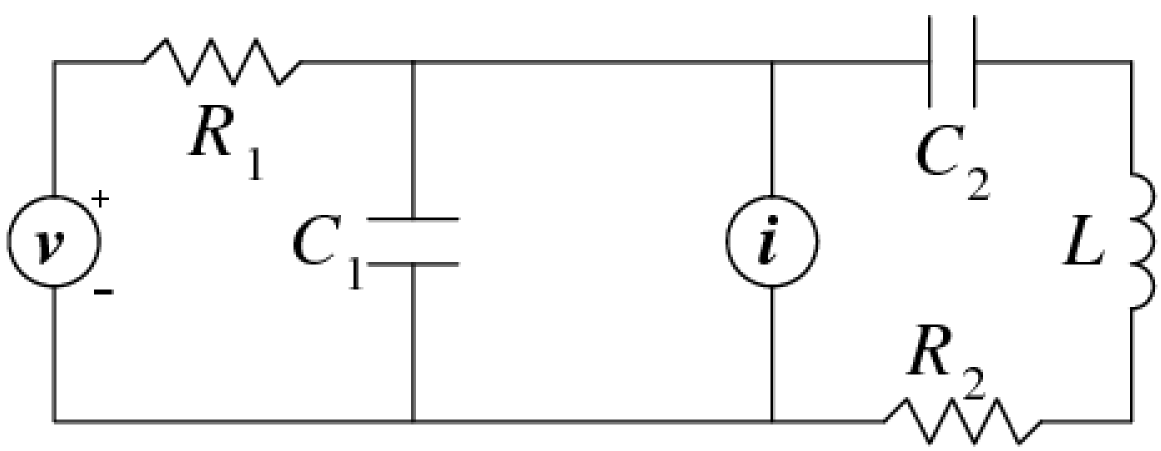


Figure 6.4: RLC circuit with two inputs.

By *switched systems* we mean systems whose dynamics changes at known times. Consider the RLC circuit shown in figure 6.4. It has two inputs - the voltage (v) and current (i) sources. Define

- x_1 - voltage across capacitor C_1, so $C_1\dot{x}_1$ is the corresponding current;
- x_2 - voltage across capacitor C_2, so the corresponding current is $C_2\dot{x}_2$.
- x_3 - current through the inductor L, so the voltage across the inductor is $L\dot{x}_3$.

Applying Kirchoff current and voltage laws we arrive at the linear system,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} -\frac{1}{R_1C_1} & 0 & -\frac{1}{C_1} \\ 0 & 0 & \frac{1}{C_2} \\ \frac{1}{L} & -\frac{1}{L} & -\frac{R_2}{L} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} \frac{1}{R_1C_1} & \frac{1}{C_1} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ i \end{bmatrix}. \tag{6.6}$$

The parameters R_1, R_2, C_1, C_2 and L, as well as the inputs, may depend on time. Suppose, for time $0 \leqslant t < 2$, $R_1 = 2$ Ohm, $R_2 = 1$ Ohm, $C_1 = 3$ F, $C_2 = 7$ F, $L = 2$ H, both inputs, v and i are present and bounded by ellipsoid $\mathcal{E}(0, I)$; and for time $t \geqslant 2$, $R_1 = R_2 = 2$ Ohm, $C_1 = C_2 = 3$ F, $L = 6$ H, the current source is turned off, and $|v| \leqslant 1$. Then, system (6.6) can be rewritten as

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{cases} \begin{bmatrix} -\frac{1}{6} & 0 & -\frac{1}{3} \\ 0 & 0 & \frac{1}{7} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} \frac{1}{6} & \frac{1}{3} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ i \end{bmatrix}, & 0 \leqslant t < 2, \\ \begin{bmatrix} -\frac{1}{6} & 0 & -\frac{1}{3} \\ 0 & 0 & \frac{1}{3} \\ \frac{1}{6} & -\frac{1}{6} & -\frac{1}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} \frac{1}{6} \\ 0 \\ 0 \end{bmatrix} v, & 2 \leqslant t. \end{cases} \tag{6.7}$$

We can compute the reach set of (6.7) for some time $t > 2$, say, $t = 3$.

```
% define system 1
firstAMat = [-1/6 0 -1/3; 0 0 1/7; 1/2 -1/2 -1/2];
firstBMat = [1/6 1/3; 0 0; 0 0];
firstUBoundsEllObj = ellipsoid(eye(2));
firstSys = elltool.linsys.LinSysContinuous(firstAMat, firstBMat,...
       firstUBoundsEllObj);
% define system 2:
secAMat = [-1/6 0 -1/3; 0 0 1/3; 1/6 -1/6 -1/3];
secBMat = [1/6; 0; 0];
secUBoundsEllObj = ellipsoid(1);
secondSys = elltool.linsys.LinSysContinuous(secAMat, secBMat,....
         secUBoundsEllObj);
x0EllObj = ellipsoid(0.01*eye(3));  % set of initial states
dirsMat = eye(3);  % 3 initial directions
switchTime = 2;  % time of switch
termTime = 3;  % terminating time

% compute the reach set:
firstRsObj = elltool.reach.ReachContinuous(firstSys, x0EllObj, dirsMat,...
 [0 switchTime], 'isRegEnabled', true, 'isJustCheck', false,...
 'regTol', 1e-5);  % reach set of the first system
% computation of the second reach set starts
% where the first left off
secRsObj = firstRsObj.evolve(termTime, secondSys);

% obtain projections onto (x1, x2) subspace:
basisMat = [1 0 0; 0 1 0]';  % (x1, x2) subspace basis
firstPsObj = firstRsObj.projection(basisMat);
secPsObj = secRsObj.projection(basisMat);

% plot the results:

firstPsObj.plotByEa('r');  % external apprx. of reach set 1 (red)
hold on;
firstPsObj.plotByIa('g');  % internal apprx. of reach set 1 (green)
secPsObj.plotByEa('y');  % external apprx. of reach set 2 (yellow)
secPsObj.plotByIa('b');  % internal apprx. of reach set 2 (blue)
% plot the 3-dimensional reach set at time t = 3:
```

```
secRsObj = secRsObj.cut(3);
secRsObj.plotByEa('y');
hold on;
secRsObj.plotByIa('b');
```

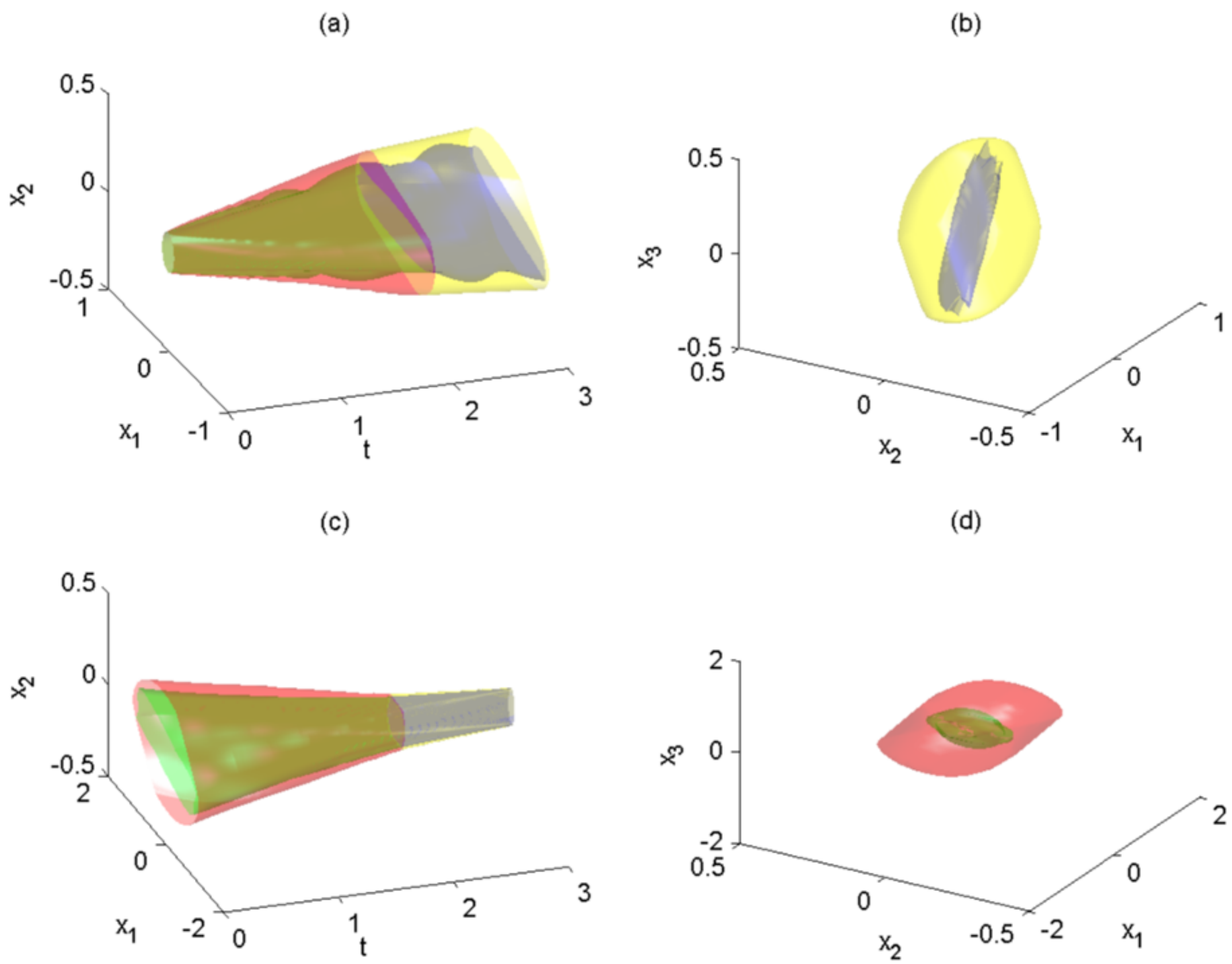


Figure 6.5: Forward and backward reach sets of the switched system (external and internal approximations).

Figure 6.5 (a) shows how the reach set projection onto (x_1, x_2) of system (6.7) evolves in time from $t = 0$ to $t = 3$. The external reach set approximation for the first dynamics is in red, the internal approximation is in green. The dynamics switches at $t = 2$. The external reach set approximation for the second dynamics is in yellow, its internal approximation is in blue. The full three-dimensional external (yellow) and internal (blue) approximations of the reach set are shown in figure 6.5 (b).

To find out where the system should start at time $t = 0$ in order to reach a neighborhood M of the origin at time $t = 3$, we compute the backward reach set from $t = 3$ to $t = 0$.

```
mEllObj  = ellipsoid(0.01*eye(3));  % terminating set
termTime = 3;  % terminating time

% compute backward reach set:
% compute the reach set:
secBrsObj = elltool.reach.ReachContinuous(secondSys, mEllObj, dirsMat,...
 [termTime switchTime], 'isRegEnabled', true, 'isJustCheck', false,...
```

```
 'regTol', 1e-5);  % second system comes first
firstBrsObj = secBrsObj.evolve(0, firstSys);  % then the first system

% obtain projections onto (x1, x2) subspace:
firstBpsObj = firstBrsObj.projection(basisMat);
secBpsObj = secBrsObj.projection(basisMat);

% plot the results:

firstBpsObj.plotByEa('r'); % external apprx. of backward reach set 1 (red)
hold on;
firstBpsObj.plotByIa('g'); % internal apprx. of backward reach set 1 (green)
secBpsObj.plotByEa('y'); % external apprx. of backward reach set 2 (yellow)
secBpsObj.plotByIa('b'); % internal apprx. of backward reach set 2 (blue)

% plot the 3-dimensional backward reach set at time t = 0:

firstBrsObj = firstBrsObj.cut(0);
firstBrsObj.plotByEa('r');
hold on;
firstBrsObj.plotByIa('g');
```

Figure 6.5 (c) presents the evolution of the reach set projection onto (x_1, x_2) in backward time. Again, external and internal approximations corresponding to the first dynamics are shown in red and green, and to the second dynamics in yellow and blue. The full dimensional backward reach set external and internal approximations of system (6.7) at time $t = 0$ is shown in figure 6.5 (d).

6.4 Hybrid System

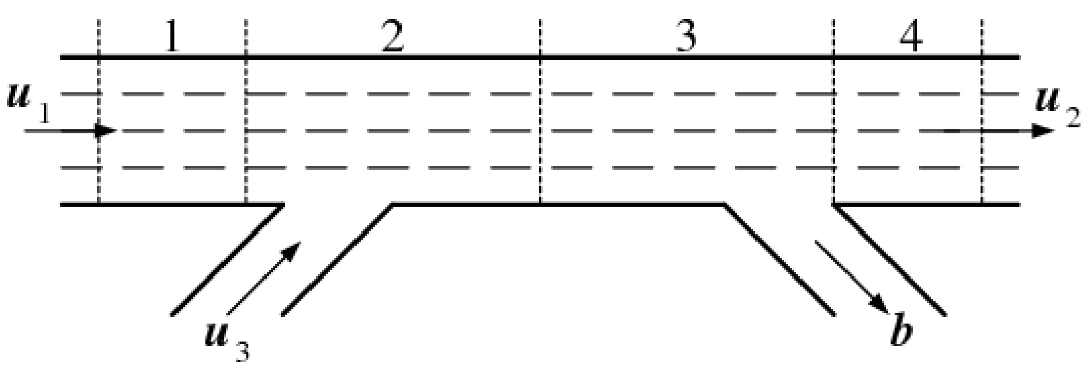


Figure 6.6: Highway model. Adapted from L.Muñoz et al. (2003).

There is no explicit implementation of the reachability analysis for hybrid systems in the *Ellipsoidal Toolbox*. Nonetheless, the operations of intersection available in the toolbox allow us to work with certain class of hybrid systems, namely, hybrid systems with affine continuous dynamics whose guards are ellipsoids, hyperplanes, halfspaces or polytopes.

We consider the *switching-mode model* of highway traffic presented in L.Muñoz et al. (2003). The highway segment is divided into N cells as shown in figure 6.6. In this particular case, $N = 4$. The traffic density in cell i is x_i vehicles per mile, $i = 1, 2, 3, 4$.

Define

- v_i - average speed in mph, in the i-th cell, $i = 1, 2, 3, 4$;
- w_i - backward congestion wave propagation speed in mph, in the i-th highway cell, $i = 1, 2, 3, 4$;
- x_{Mi} - maximum allowed density in the i-th cell; when this velue is reached, there is a traffic jam, $i = 1, 2, 3, 4$;
- d_i - length of i-th cell in miles, $i = 1, 2, 3, 4$;
- T_s - sampling time in hours;
- b - split ratio for the off-ramp;
- u_1 - traffic flow coming into the highway segment, in vehicles per hour (vph);
- u_2 - traffic flow coming out of the highway segment (vph);
- u_3 - on-ramp traffic flow (vph).

Highway traffic operates in two modes: *free-flow* in normal operation; and *congested* mode, when there is a jam. Traffic flow in free-flow mode is described by

$$\begin{bmatrix} x_1[t+1] \\ x_2[t+1] \\ x_3[t+1] \\ x_4[t+1] \end{bmatrix} = \begin{bmatrix} 1-\frac{v_1T_s}{d_1} & 0 & 0 & 0 \\ \frac{v_1T_s}{d_2} & 1-\frac{v_2T_s}{d_2} & 0 & 0 \\ 0 & \frac{v_2T_s}{d_3} & 1-\frac{v_3T_s}{d_3} & 0 \\ 0 & 0 & (1-b)\frac{v_3T_s}{d_4} & 1-\frac{v_4T_s}{d_4} \end{bmatrix} \begin{bmatrix} x_1[t] \\ x_2[t] \\ x_3[t] \\ x_4[t] \end{bmatrix}$$
$$+ \begin{bmatrix} \frac{v_1T_s}{d_1} & 0 & 0 \\ 0 & 0 & \frac{v_2T_s}{d_2} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}.$$

The equation for the congested mode is

$$\begin{bmatrix} x_1[t+1] \\ x_2[t+1] \\ x_3[t+1] \\ x_4[t+1] \end{bmatrix} = \begin{bmatrix} 1-\frac{w_1T_s}{d_1} & \frac{w_2T_s}{d_1} & 0 & 0 \\ 0 & 1-\frac{w_2T_s}{d_2} & \frac{w_3T_s}{d_2} & 0 \\ 0 & 0 & 1-\frac{w_3T_s}{d_3} & \frac{1}{1-b}\frac{w_4T_s}{d_3} \\ 0 & 0 & 0 & 1-\frac{w_4T_s}{d_4} \end{bmatrix} \begin{bmatrix} x_1[t] \\ x_2[t] \\ x_3[t] \\ x_4[t] \end{bmatrix}$$
$$+ \begin{bmatrix} 0 & 0 & \frac{w_1T_s}{d_1} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -\frac{w_4T_s}{d_4} & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$
$$+ \begin{bmatrix} \frac{w_1T_s}{d_1} & -\frac{w_2T_s}{d_1} & 0 & 0 \\ 0 & \frac{w_2T_s}{d_2} & -\frac{w_3T_s}{d_2} & 0 \\ 0 & 0 & \frac{w_3T_s}{d_3} & -\frac{1}{1-b}\frac{w_4T_s}{d_3} \\ 0 & 0 & 0 & \frac{w_4T_s}{d_4} \end{bmatrix} \begin{bmatrix} x_{M1} \\ x_{M2} \\ x_{M3} \\ x_{M4} \end{bmatrix}.$$

The switch from the free-flow to the congested mode occurs when the density x_2 reaches x_{M2}. In other words, the hyperplane $H([0\ 1\ 0\ 0]^T, x_{M2})$ is the guard.

We indicate how to implement the reach set computation of this hybrid system. We first define the two linear systems and the guard.

```
% assign parameter values:
v1 = 65; v2 = 60; v3 = 63; v4 = 65;  % mph
w1 = 10; w2 = 10; w3 = 10; w4 = 10;  % mph
d1 = 2; d2 = 3; d3 = 4; d4 = 2;  % miles
Ts = 2/3600;  % sampling time in hours
xM1 = 200; xM2 = 200; xM3 = 200; xM4 = 200;  % vehicles per lane
b = 0.4;
```

```
firstAMat = [(1-(v1*Ts/d1)) 0 0 0
          (v1*Ts/d2) (1-(v2*Ts/d2)) 0 0
          0 (v2*Ts/d3) (1-(v3*Ts/d3)) 0
          0 0 ((1-b)*(v3*Ts/d4)) (1-(v4*Ts/d4))];
firstBMat = [v1*Ts/d1 0 0; 0 0 v2*Ts/d2; 0 0 0; 0 0 0];
firstUBoundsEllObj = ellipsoid([180; 150; 50], [100 0 0; 0 100 0; 0 0 25]);

secAMat = [(1-(w1*Ts/d1)) (w2*Ts/d1) 0 0
          0 (1-(w2*Ts/d2)) (w3*Ts/d2) 0
          0 0 (1-(w3*Ts/d3)) ((1/(1-b))*(w4*Ts/d3))
          0 0 0 (1-(w4*Ts/d4))];
secBMat = [0 0 w1*Ts/d1; 0 0 0; 0 0 0; 0 -w4*Ts/d4 0];
secUBoundsEllObj = firstUBoundsEllObj;
gMat = [(w1*Ts/d1) (-w2*Ts/d1) 0 0
          0 (w2*Ts/d2) (-w3*Ts/d2) 0
          0 0 (w3*Ts/d3) ((-1/(1-b))*(w4*Ts/d3))
          0 0 0 (w4*Ts/d4)];
vVec = [xM1; xM2; xM3; xM4];
% define linear systems:
% free-flow mode
firstSys = elltool.linsys.LinSysDiscrete(firstAMat, firstBMat,...
    firstUBoundsEllObj);
% congestion mode
secSys = elltool.linsys.LinSysDiscrete(secAMat, secBMat,...
 secUBoundsEllObj, gMat, vVec);
% define guard:
grdHypObj = hyperplane([0; 1; 0; 0], xM2);
```

We assume that initially the system is in free-flow mode. Given a set of initial conditions, we compute the reach set according to dynamics (6.4) for certain number of time steps. We will consider the external approximation of the reach set by a single ellipsoid.

```
%initial conditions:
x0EllObj = [170; 180; 175; 170] + 10*ell_unitball(4);

dirsMat = [1; 0; 0; 0];  % single initial direction
nSteps = 100;  % number of time steps

% free-flow reach set
ffrsObj = elltool.reach.ReachDiscrete(firstSys, x0EllObj, dirsMat, [0 nSteps]);
externalEllMat = ffrsObj.get_ea();  % 101x1 array of external ellipsoids
```

Having obtained the ellipsoidal array externalEllMat representing the reach set evolving in time, we determine the ellipsoids in the array that intersect the guard.

```
% some of the intersections are empty
intersectEllVec = externalEllMat.hpintersection(grdHypObj);
% determine nonempty intersections
indNonEmptyVec = find(~isEmpty(intersectEllVec));
%
min(indNonEmptyVec)

% ans =
%
%       19

max(indNonEmptyVec)

```

```
% ans =
%
%       69
```

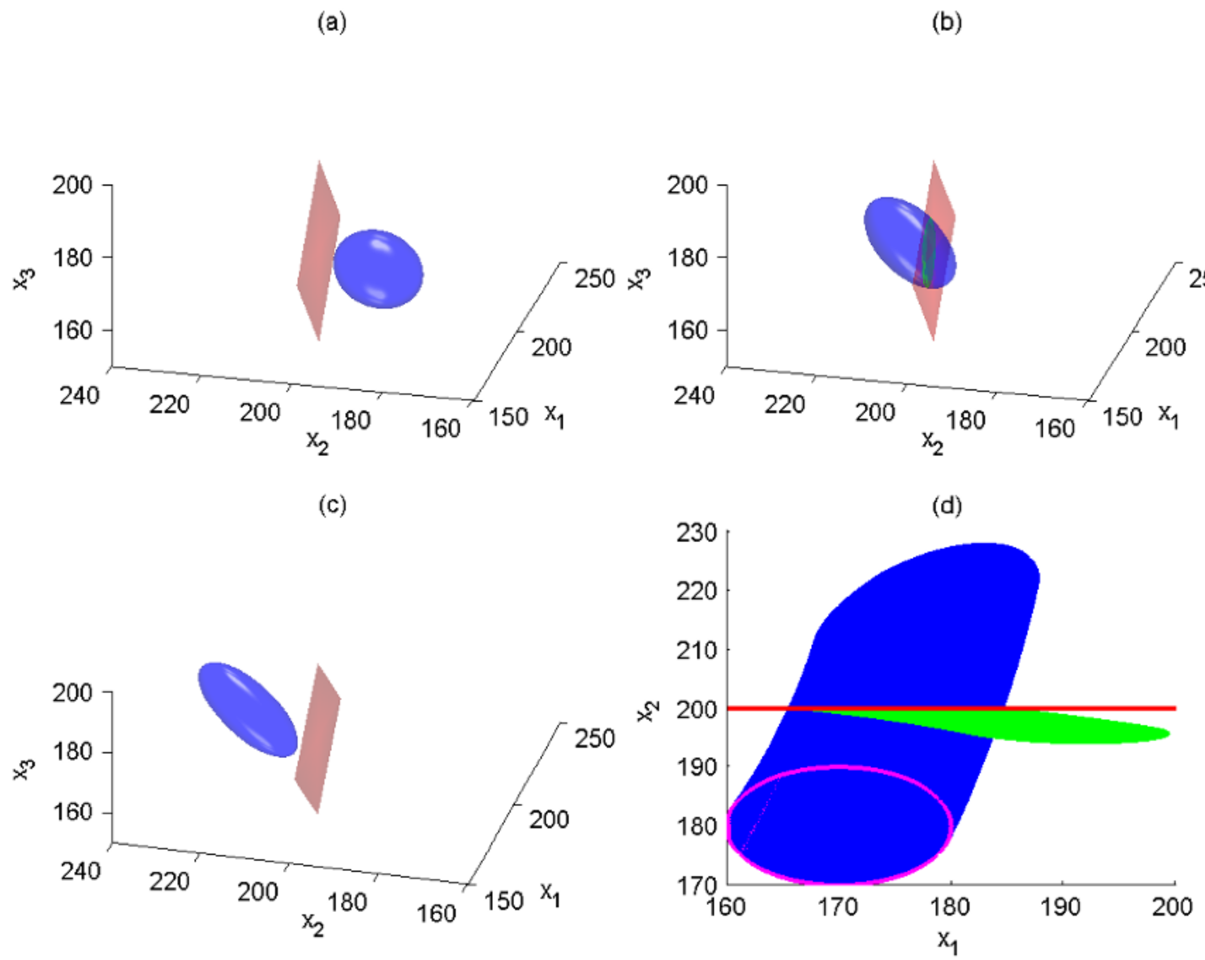


Figure 6.7: Reach set of the free-flow system is blue, reach set of the congested system is green, the guard is red. (a) Reach set of the free-flow system at $t = 10$, before reaching the guard (projection onto (x_1, x_2, x_3)). (b) Reach set of the free-flow system at $t = 50$, crossing the guard. (projection onto (x_1, x_2, x_3)). (c) Reach set of the free-flow system at $t = 80$, after the guard is crossed. (projection onto (x_1, x_2, x_3)). (d) Reach set trace from $t = 0$ to *t=100*, free-flow system in blue, congested system in green; bounds of initial conditions are marked with magenta (projection onto (x_1, x_2)).

Analyzing the values in array dVec, we conclude that the free-flow reach set has nonempty intersection with hyperplane grdHyp at $t = 18$ for the first time, and at $t = 68$ for the last time. Between $t = 18$ and $t = 68$ it crosses the guard. Figure 6.7 (a) shows the free-flow reach set projection onto (x_1, x_2, x_3) subspace for $t = 10$, before the guard crossing; figure 6.7 (b) for $t = 50$, during the guard crossing; and figure 6.7 (c) for $t = 80$, after the guard was crossed.

For each time step that the intersection of the free-flow reach set and the guard is nonempty, we establish a new initial time and a set of initial conditions for the reach set computation according to dynamics (6.4). The initial time is the array index minus one, and the set of initial conditions is the intersection of the free-flow reach set with the guard.

```
crsObjVec = [];
for iInd = 1:size(indNonEmptyVec, 2)
    curTimeLimVec=[indNonEmptyVec(iInd)-1 nSteps];
```

```
    rsObj = elltool.reach.ReachDiscrete(secSys,...
        intersectEllVec(indNonEmptyVec(iInd)), ...
            dirsMat, curTimeLimVec,'isRegEnabled',true);
    crsObjVec = [crsObjVec rsObj];
end
```

The union of reach sets in array crs forms the reach set for the congested dynamics.

A summary of the reach set computation of the linear hybrid system (6.4)-(6.4) for $N = 100$ time steps with one guard crossing is given in figure 6.7 (d), which shows the projection of the reach set trace onto (x_1, x_2) subspace. The system starts evolving in time in free-flow mode from a set of initial conditions at $t = 0$, whose boundary is shown in magenta. The free-flow reach set evolving from $t = 0$ to $t = 100$ is shown in blue. Between $t = 18$ and $t = 68$ the free-flow reach set crosses the guard. The guard is shown in red. For each nonempty intersection of the free-flow reach set and the guard, the congested mode reach set starts evolving in time until $t = 100$. All the congested mode reach sets are shown in green. Observe that in the congested mode, the density x_2 in the congested part decreases slightly, while the density x_1 upstream of the congested part increases. The blue set above the guard is not actually reached, because the state evolves according to the green region.

"Multi-Parametric Toolbox Homepage." control.ee.ethz.ch/\~mpt.

L.Muñoz, X.Sun, R.Horowitz, and L.Alvarez. 2003. "Traffic Density Estimation with the Cell Transmission Model." In *Proceedings of the American Control Conference*, 3750–3755. Denver, Colorado, USA.

CHAPTER

SEVEN

SUMMARY AND OUTLOOK

Although some of the operations with ellipsoids are present in the commercial Geometric Bounding Toolbox Veres et al. (2001; "Geometric Bounding Toolbox Homepage"), the ellipsoid-related functionality of that toolbox is rather limited.

Ellipsoidal Toolbox is the first free MATLAB package that implements ellipsoidal calculus and uses ellipsoidal methods for reachability analysis of continuous- and discrete-time affine systems, continuous-time linear systems with disturbances and switched systems, whose dynamics changes at known times. The reach set computation for hybrid systems whose guards are hyperplanes or polyhedra is not implemented explicitly, but the tool for such computation exists, namely, the operations of intersection of ellipsoid with hyperplane and ellipsoid with halfspace.

"Geometric Bounding Toolbox Homepage." www.sysbrain.com/gbt.

Veres, S. M., A. V. Kuntsevich, I. V. Vályi, S. Hermsmeyer, and D. S. Wall. 2001. "Geometric Bounding Toolbox for MATLAB." *MATLAB/Simulink Connections Catalogue*.

CHAPTER

EIGHT

ACKNOWLEDGEMENT

The authors would like to thank Alexander B. Kurzhanski, Manfred Morari, Johan Löfberg, Michal Kvasnica and Goran Frehse for their support of this work by useful advice and encouragement.

CHAPTER NINE

FUNCTION REFERENCE

9.1 ellipsoid

```
CALCGRID - computes grid of 2d or 3d sphere and vertices for each face
           in the grid with number of points taken from ellObj
           nPlot2dPoints or nPlot3dPoints parameters

CHECKISME - determine whether input object is ellipsoid. And display
            message and abort function if input object
            is not ellipsoid

Input:
  regular:
      someObjArr: any[] - any type array of objects.

Example:
  ellObj = ellipsoid([1; 2], eye(2));
  ellipsoid.checkIsMe(ellObj)

Ellipsoid library of the Ellipsoidal Toolbox.


Constructor and data accessing functions:
-----------------------------------------
 ellipsoid    - Constructor of ellipsoid object.
 double       - Returns parameters of ellipsoid, i.e. center and shape
                matrix.
 parameters   - Same function as 'double'(legacy matter).
 dimension    - Returns dimension of ellipsoid and its rank.
 isdegenerate - Checks if ellipsoid is degenerate.
 isempty      - Checks if ellipsoid is empty.
 maxeig       - Returns the biggest eigenvalue of the ellipsoid.
 mineig       - Returns the smallest eigenvalue of the ellipsoid.
 trace        - Returns the trace of the ellipsoid.
 volume       - Returns the volume of the ellipsoid.


Overloaded operators and functions:
-----------------------------------
 eq      - Checks if two ellipsoids are equal.
 ne      - The opposite of 'eq'.
 gt, ge  - E1 > E2 (E1 >= E2) checks if, given the same center ellipsoid
           E1 contains E2.
 lt, le  - E1 < E2 (E1 <= E2) checks if, given the same center ellipsoid
```

```
          E2 contains E1.
 mtimes  - Given matrix A in R^(mxn) and ellipsoid E in R^n, returns
          (A * E).
 plus    - Given vector b in R^n and ellipsoid E in R^n, returns (E + b).
 minus   - Given vector b in R^n and ellipsoid E in R^n, returns (E - b).
 uminus  - Changes the sign of the center of ellipsoid.
 display - Displays the details about given ellipsoid object.
 inv     - inverts the shape matrix of the ellipsoid.
 plot    - Plots ellipsoid in 1D, 2D and 3D.


Geometry functions:
-------------------
 move2origin       - Moves the center of ellipsoid to the origin.
 shape             - Same as 'mtimes', but modifies only shape matrix of
                     the ellipsoid leaving its center as is.
 rho               - Computes the value of support function and
                     corresponding boundary point of the ellipsoid in
                     the given direction.
 polar             - Computes the polar ellipsoid to an ellipsoid that
                     contains the origin.
 projection        - Projects the ellipsoid onto a subspace specified
                     by  orthogonal basis vectors.
 minksum           - Computes and plots the geometric (Minkowski) sum of
                     given ellipsoids in 1D, 2D and 3D.
 minksum_ea        - Computes the external ellipsoidal approximation of
                     geometric sum of given ellipsoids in given
                     direction.
 minksum_ia        - Computes the internal ellipsoidal approximation of
                     geometric sum of given ellipsoids in given
                     direction.
 minkdiff          - Computes and plots the geometric (Minkowski)
                     difference of given ellipsoids in 1D, 2D and 3D.
 minkdiff_ea       - Computes the external ellipsoidal approximation of
                     geometric difference of two ellipsoids in given
                     direction.
 minkdiff_ia       - Computes the internal ellipsoidal approximation of
                     geometric difference of two ellipsoids in given
                     direction
 minkpm            - Computes and plots the geometric (Minkowski)
                     difference of a geometric sum of ellipsoids and a
                     single ellipsoid in 1D, 2D and 3D.
 minkpm_ea         - Computes the external ellipsoidal approximation of
                     the geometric difference of a geometric sum of
                     ellipsoids and a single ellipsoid in given
                     direction.
 minkpm_ia         - Computes the internal ellipsoidal approximation of
                     the geometric difference of a geometric sum of
                     ellipsoids and a single ellipsoid in given
                     direction.
 minkmp            - Computes and plots the geometric (Minkowski) sum of
                     a geometric difference of two single ellipsoids and
                     a geometric sum of ellipsoids in 1D, 2D and 3D.
 minkmp_ea         - Computes the external ellipsoidal approximation of
                     the geometric sum of a geometric difference of two
                     single ellipsoids and a geometric sum of ellipsoids
                     in given direction.
 minkmp_ia         -  Computes the internal ellipsoidal approximation of
```

```
                      the geometric sum of a geometric difference of
                      two single ellipsoids and a geometric sum of ellipsoids
                      in given direction.
 isbaddirection     - Checks if ellipsoidal approximation of geometric difference
                      of two ellipsoids in the given direction can be computed.
 doesIntersectionContain          - Checks if the union or intersection of
                      ellipsoids or polytopes lies inside the intersection
                      of given ellipsoids.
 isinternal         - Checks if given vector belongs to the union or intersection
                      of given ellipsoids.
 distance           - Computes the distance from ellipsoid to given point,
                      ellipsoid, hyperplane or polytope.
 intersect          - Checks if the union or intersection of ellipsoids intersects
                      with given ellipsoid, hyperplane, or polytope.
 intersection_ea    - Computes the minimal volume ellipsoid containing intersection
                      of two ellipsoids, ellipsoid and halfspace, or ellipsoid
                      and polytope.
 intersection_ia    - Computes the maximal ellipsoid contained inside the
                      intersection of two ellipsoids, ellipsoid and halfspace
                      or ellipsoid and polytope.
 ellintersection_ia - Computes maximum volume ellipsoid that is contained
                      in the intersection of given ellipsoids (can be more than 2).
 ellunion_ea        - Computes minimum volume ellipsoid that contains
                      the union of given ellipsoids.
 hpintersection     - Computes the intersection of ellipsoid with hyperplane.

DIMENSION - returns the dimension of the space in which the ellipsoid is
            defined and the actual dimension of the ellipsoid.

Input:
  regular:
    myEllArr: ellipsoid[nDims1,nDims2,...,nDimsN] - array of ellipsoids.

Output:
  regular:
    dimArr: double[nDims1,nDims2,...,nDimsN] - space dimensions.

  optional:
    rankArr: double[nDims1,nDims2,...,nDimsN] - dimensions of the
           ellipsoids in myEllArr.

Example:
  firstEllObj = ellipsoid();
  tempMatObj = [3 1; 0 1; -2 1];
  secEllObj = ellipsoid([1; -1; 1], tempMatObj*tempMatObj');
  thirdEllObj = ellipsoid(eye(2));
  fourthEllObj = ellipsoid(0);
  ellMat = [firstEllObj secEllObj; thirdEllObj fourthEllObj];
  [dimMat, rankMat] = ellMat.dimension()

  dimMat =

     0     3
     2     1

  rankMat =

     0     2
```

```
     2     0

DISP - Displays ellipsoid object.

Input:
  regular:
    myEllMat: ellipsoid [mRows, nCols] - matrix of ellipsoids.

Example:
  ellObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  disp(ellObj)

  Ellipsoid with parameters
  Center:
      -2
      -1

  Shape Matrix:
       2    -1
      -1     1

DISPLAY - Displays the details of the ellipsoid object.

Input:
  regular:
      myEllMat: ellipsoid [mRows, nCols] - matrix of ellipsoids.

Example:
  ellObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  display(ellObj)

  ellObj =

  Center:
      -2
      -1

  Shape Matrix:
       2    -1
      -1     1

  Nondegenerate ellipsoid in R^2.

DISTANCE - computes distance between the given ellipsoid (or array of
           ellipsoids) to the specified object (or arrays of objects):
           vector, ellipsoid, hyperplane or polytope.

Input:
  regular:
      ellObjArr: ellipsoid [nDims1, nDims2,..., nDimsN] -  array of
         ellipsoids of the same dimension.
      objArray: double / ellipsoid / hyperplane / polytope [nDims1,
          nDims2,..., nDimsN] - array of vectors or ellipsoids or
          hyperplanes or polytopes. If number of elements in objArray
          is more than 1, then it must be equal to the number of elements
          in ellObjArr.

  optional:
```

```
      isFlagOn: logical[1,1] - if true then distance is  computed in
          ellipsoidal metric, if false - in Euclidean metric (by default
          isFlagOn=false).

Output:
  regular:
    distValArray: double [nDims1, nDims2,..., nDimsN] - array of pairwise
          calculated distances.
          Negative distance value means
              for ellipsoid and vector: vector belongs to the ellipsoid,
              for ellipsoid and hyperplane: ellipsoid intersects the
                  hyperplane.
              Zero distance value means for ellipsoid and vector: vector
                  is aboundary point of the ellipsoid,
              for ellipsoid and hyperplane: ellipsoid  touches the
                  hyperplane.
  optional:
      statusArray: double [nDims1, nDims2,..., nDimsN] - array of time of
          computation of ellipsoids-vectors or ellipsoids-ellipsoids
          distances, or status of cvx solver for ellipsoids-polytopes
          distances.

Literature:
 1. Lin, A. and Han, S. On the Distance between Two Ellipsoids.
    SIAM Journal on Optimization, 2002, Vol. 13, No. 1 : pp. 298-308
 2. Stanley Chan, "Numerical method for Finding Minimum Distance to an
    Ellipsoid".
    http://videoprocessing.ucsd.edu/~stanleychan/publication/...
    unpublished/Ellipse.pdf

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  tempMat = [1 1; 1 -1; -1 1; -1 -1]';
  distVec = ellObj.distance(tempMat)

  distVec =

       2.3428    1.0855    1.3799    -1.0000

DOESCONTAIN - checks if one ellipsoid contains the other ellipsoid or
              polytope. The condition for E1 = firstEllArr to contain
              E2 = secondEllArr is
              min(rho(l | E1) - rho(l | E2)) > 0, subject to <l, l> = 1.
              How checked if ellipsoid contains polytope is explained in
              doesContainPoly.
Input:
  regular:
      firstEllArr: ellipsoid [nDims1,nDims2,...,nDimsN]/[1,1] - first
          array of ellipsoids.
      secondObjArr: ellipsoid [nDims1,nDims2,...,nDimsN]/
          polytope[nDims1,nDims2,...,nDimsN]/[1,1] - array of the same
          size as firstEllArr or single ellipsoid or polytope.

   properties:
      mode: char[1, 1] - 'u' or 'i', go to description.
      computeMode: char[1,] - 'highDimFast' or 'lowDimFast'. Determines,
          which way function is computed, when secObjArr is polytope. If
          secObjArr is ellipsoid computeMode is ignored. 'highDimFast'
```

```
          works  faster for  high dimensions, 'lowDimFast' for low. If
          this property is omitted if dimension of ellipsoids is greater
          then 10, then 'hightDimFast' is choosen, otherwise -
          'lowDimFast'

Output:
  isPosArr: logical[nDims1,nDims2,...,nDimsN],
      resArr(iCount) = true - firstEllArr(iCount)
      contains secondObjArr(iCount), false - otherwise.

Example:
  firstEllObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  secEllObj = ellipsoid([-1;0], eye(2));
  doesContain(firstEllObj,secEllObj)

  ans =

       0

DOESINTERSECTIONCONTAIN - checks if the intersection of ellipsoids
                          contains the union or intersection of given
                          ellipsoids or polytopes.

  res = DOESINTERSECTIONCONTAIN(fstEllArr, secEllArr, mode)
      Checks if the union
      (mode = 'u') or intersection (mode = 'i') of ellipsoids in
      secEllArr lies inside the intersection of ellipsoids in
      fstEllArr. Ellipsoids in fstEllArr and secEllArr must be
      of the same dimension. mode = 'u' (default) - union of
      ellipsoids in secEllArr. mode = 'i' - intersection.
  res = DOESINTERSECTIONCONTAIN(fstEllArr, secPolyArr, mode)
       Checks if the union
      (mode = 'u') or intersection (mode = 'i')  of polytopes in
      secPolyArr lies inside the intersection of ellipsoids in
      fstEllArr. Ellipsoids in fstEllArr and polytopes in secPolyArr
      must be of the same dimension. mode = 'u' (default) - union of
      polytopes in secPolyMat. mode = 'i' - intersection.

  To check if the union of ellipsoids secEllArr belongs to the
  intersection of ellipsoids fstEllArr, it is enough to check that
  every ellipsoid of secEllMat is contained in every
  ellipsoid of fstEllArr.
  Checking if the intersection of ellipsoids in secEllMat is inside
  intersection fstEllMat can be formulated as quadratically
  constrained quadratic programming (QCQP) problem.

  Let fstEllArr(iEll) = E(q, Q) be an ellipsoid with center q and shape
  matrix Q. To check if this ellipsoid contains the intersection of
  ellipsoids in secObjArr:
  E(q1, Q1), E(q2, Q2), ..., E(qn, Qn), we define the QCQP problem:
                    J(x) = <(x - q), Q^(-1)(x - q)> --> max
  with constraints:
                    <(x - q1), Q1^(-1)(x - q1)> <= 1   (1)
                    <(x - q2), Q2^(-1)(x - q2)> <= 1   (2)
                    ................................
                    <(x - qn), Qn^(-1)(x - qn)> <= 1   (n)

  If this problem is feasible, i.e. inequalities (1)-(n) do not
```

```
  contradict, or, in other words, intersection of ellipsoids
  E(q1, Q1), E(q2, Q2), ..., E(qn, Qn) is nonempty, then we can find
  vector y such that it satisfies inequalities (1)-(n)
  and maximizes function J. If J(y) <= 1, then ellipsoid E(q, Q)
  contains the given intersection, otherwise, it does not.

  The intersection of polytopes is a polytope, which is computed
  by the standard routine of MPT. How checked if intersection of
  ellipsoids contains polytope is explained in doesContainPoly.

  Checking if the union of polytopes belongs to the intersection
  of ellipsoids is the same as checking if its convex hull belongs
  to this intersection.

Input:
  regular:
      fstEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
          of the same size.
      secEllArr: ellipsoid /
          polytope [nDims1,nDims2,...,nDimsN] - array of ellipsoids or
          polytopes of the same sizes.

          note: if mode == 'i', then fstEllArr, secEllVec should be
              array.

  properties:
      mode: char[1, 1] - 'u' or 'i', go to description.
      computeMode: char[1,] - 'highDimFast' or 'lowDimFast'. Determines,
          which way function is computed, when secObjArr is polytope. If
          secObjArr is ellipsoid computeMode is ignored. 'highDimFast'
          works  faster for  high dimensions, 'lowDimFast' for low. If
          this property is omitted if dimension of ellipsoids is greater
          then 10, then 'hightDimFast' is choosen, otherwise -
          'lowDimFast'


Output:
  res: double[1, 1] - result:
      -1 - problem is infeasible, for example, if s = 'i',
          but the intersection of ellipsoids in E2 is an empty set;
      0 - intersection is empty;
      1 - if intersection is nonempty.
  status: double[0, 0]/double[1, 1] - status variable. status is empty
      if mode == 'u' or mSecRows == nSecCols == 1.

Example:
  firstEllObj = [0 ; 0] + ellipsoid(eye(2, 2));
  secEllObj = [0 ; 0] + ellipsoid(2*eye(2, 2));
  thirdEllObj = [1; 0] + ellipsoid(0.5 * eye(2, 2));
  secEllObj.doesIntersectionContain([firstEllObj secEllObj], 'i')

  ans =

       1

DOUBLE - returns parameters of the ellipsoid.

Input:
```

```
  regular:
      myEll: ellipsoid [1, 1] - single ellipsoid of dimention nDims.


Output:
  myEllCentVec: double[nDims, 1] - center of the ellipsoid myEll.

  myEllShMat: double[nDims, nDims] - shape matrix of the ellipsoid myEll.

Example:
  ellObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  [centVec, shapeMat] = double(ellObj)
  centVec =

      -2
      -1


  shapeMat =

       2    -1
      -1     1
```

```
ELLBNDR_2D - compute the boundary of 2D ellipsoid. Private method.

Input:
  regular:
      myEll: ellipsoid [1, 1]- ellipsoid of the dimention 2.
  optional:
      nPoints: number of boundary points

Output:
  regular:
      bpMat: double[nPoints,2] - boundary points of ellipsoid
  optional:
      fVec: double[1,nFaces] - indices of points in each face of
          bpMat graph
```

```
ELLBNDR_3D - compute the boundary of 3D ellipsoid.

Input:
  regular:
      myEll: ellipsoid [1, 1]- ellipsoid of the dimention 3.

  optional:
      nPoints: number of boundary points

Output:
  regular:
      bpMat: double[nPoints,3] - boundary points of ellipsoid
  optional:
      fMat: double[nFaces,3] - indices of face verties in bpMat
```

```
ELLINTERSECTION_IA - computes maximum volume ellipsoid that is contained
                     in the intersection of given ellipsoids.


Input:
```

```
  regular:
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions.

Output:
  outEll: ellipsoid [1, 1] - resulting maximum volume ellipsoid.

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 2], eye(2);
  ellVec = [firstEllObj secEllObj];
  resEllObj = ellintersection_ia(ellVec)

  resEllObj =

  Center:
      0.1847
      1.6914

  Shape Matrix:
      0.0340   -0.0607
     -0.0607    0.1713

  Nondegenerate ellipsoid in R^2.

ELLIPSOID - constructor of the ellipsoid object.

  Ellipsoid E = { x in R^n : <(x - q), Q^(-1)(x - q)> <= 1 }, with current
      "Properties". Here q is a vector in R^n, and Q in R^(nxn) is positive
      semi-definite matrix

  ell = ELLIPSOID - Creates an empty ellipsoid

  ell = ELLIPSOID(shMat) - creates an ellipsoid with shape matrix shMat,
      centered at 0

   ell = ELLIPSOID(centVec, shMat) - creates an ellipsoid with shape matrix
      shMat and center centVec

  ell = ELLIPSOID(centVec, shMat, 'propName1', propVal1,...,
      'propNameN',propValN) - creates an ellipsoid with shape
      matrix shMat, center centVec and propName1 = propVal1,...,
      propNameN = propValN. In other cases "Properties"
      are taken from current values stored in
      elltool.conf.Properties.
  ellMat = Ellipsoid(centVecArray, shMatArray,
      ['propName1', propVal1,...,'propNameN',propValN]) -
      creates an array (possibly multidimensional) of
      ellipsoids with centers centVecArray(:,dim1,...,dimn)
      and matrices shMatArray(:,:,dim1,...dimn) with
      properties if given.

  These parameters can be accessed by DOUBLE(E) function call.
  Also, DIMENSION(E) function call returns the dimension of
  the space in which ellipsoid E is defined and the actual
  dimension of the ellipsoid; function ISEMPTY(E) checks if
  ellipsoid E is empty; function ISDEGENERATE(E) checks if
  ellipsoid E is degenerate.
```

```
Input:
  Case1:
    regular:
      shMatArray: double [nDim, nDim] /
          double [nDim, nDim, nDim1,...,nDimn] -
          shape matrices array

  Case2:
    regular:
      centVecArray: double [nDim,1] /
          double [nDim, 1, nDim1,...,nDimn] -
          centers array
      shMatArray: double [nDim, nDim] /
          double [nDim, nDim, nDim1,...,nDimn] -
          shape matrices array


  properties:
      absTol: double [1,1] - absolute tolerance with default value 10^(-7)
      relTol: double [1,1] - relative tolerance with default value 10^(-5)
      nPlot2dPoints: double [1,1] - number of points for 2D plot with
          default value 200
      nPlot3dPoints: double [1,1] - number of points for 3D plot with
           default value 200.

Output:
  ellMat: ellipsoid [1,1] / ellipsoid [nDim1,...nDimn] -
      ellipsoid with specified properties
      or multidimensional array of ellipsoids.

Example:
  ellObj = ellipsoid([1 0 -1 6]', 9*eye(4));

ELLUNION_EA - computes minimum volume ellipsoid that contains union
              of given ellipsoids.

Input:
  regular:
      inpEllMat: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions.

Output:
  outEll: ellipsoid [1, 1] - resulting minimum volume ellipsoid.

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 2], eye(2));
  ellVec = [firstEllObj secEllObj];
  resEllObj = ellunion_ea(ellVec)
  resEllObj =

  Center:
     -0.3188
      1.2936

  Shape Matrix:
      5.4573    1.3386
      1.3386    4.1037
```

```
  Nondegenerate ellipsoid in R^2.

FROMREPMAT - returns array of equal ellipsoids the same
             size as stated in sizeVec argument

  ellArr = fromRepMat(sizeVec) - creates an array  size
           sizeVec of empty ellipsoids.

  ellArr = fromRepMat(shMat,sizeVec) - creates an array
           size sizeVec of ellipsoids with shape matrix
           shMat.

  ellArr = fromRepMat(cVec,shMat,sizeVec) - creates an
           array size sizeVec of ellipsoids with shape
           matrix shMat and center cVec.

Input:
  Case1:
      regular:
          sizeVec: double[1,n] - vector of size, have
          integer values.

  Case2:
      regular:
          shMat: double[nDim, nDim] - shape matrix of
          ellipsoids.
          sizeVec: double[1,n] - vector of size, have
          integer values.

  Case3:
      regular:
          cVec: double[nDim,1] - center vector of
          ellipsoids
          shMat: double[nDim, nDim] - shape matrix of
          ellipsoids.
          sizeVec: double[1,n] - vector of size, have
          integer values.

  properties:
      absTol: double [1,1] - absolute tolerance with default
          value 10^(-7)
      relTol: double [1,1] - relative tolerance with default
          value 10^(-5)
      nPlot2dPoints: double [1,1] - number of points for 2D plot
          with default value 200
      nPlot3dPoints: double [1,1] - number of points for 3D plot
          with default value 200.

fromStruct -- converts structure array into ellipsoid array.

Input:
  regular:
      SEllArr: struct [nDim1, nDim2, ...] - array
          of structures with the following fields:

      q: double[1, nEllDim] - the center of ellipsoid
      Q: double[nEllDim, nEllDim] - the shape matrix of ellipsoid
Output:
```

```
      ellArr: ellipsoid [nDim1, nDim2, ...] - ellipsoid array with size of
          SEllArr.

Example:
s = struct('Q', eye(2), 'q', [0 0]);
ellipsoid.fromStruct(s)

-------ellipsoid object-------
Properties:
   |
   |-- actualClass : 'ellipsoid'
   |--------- size : [1, 1]

Fields (name, type, description):
    'Q'    'double'    'Configuration matrix'
    'q'    'double'    'Center'

Data:
   |
   |-- q : [0 0]
   |       -----
   |-- Q : |1|0|
   |       |0|1|
   |       -----

GETABSTOL - gives the array of absTol for all elements in ellArr

Input:
  regular:
      ellArr: ellipsoid[nDim1, nDim2, ...] - multidimension array
          of ellipsoids
  optional
      fAbsTolFun: function_handle[1,1] - function that apply
          to the absTolArr. The default is @min.

Output:
  regular:
      absTolArr: double [absTol1, absTol2, ...] - return absTol for
          each element in ellArr
  optional:
      absTol: double[1,1] - return result of work fAbsTolFun with
          the absTolArr

Usage:
  use [~,absTol] = ellArr.getAbsTol() if you want get only
      absTol,
  use [absTolArr,absTol] = ellArr.getAbsTol() if you want get
      absTolArr and absTol,
  use absTolArr = ellArr.getAbsTol() if you want get only absTolArr

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 2], eye(2));
  ellVec = [firstEllObj secEllObj];
  absTolVec = ellVec.getAbsTol()

  absTolVec =
```

```
     1.0e-07 *

      1.0000    1.0000

GETBOUNDARY - computes the boundary of an ellipsoid.

Input:
  regular:
      myEll: ellipsoid [1, 1]- ellipsoid of the dimention 2 or 3.
  optional:
      nPoints: number of boundary points

Output:
  regular:
      bpMat: double[nPoints,nDim] - boundary points of ellipsoid
  optional:
      fVec: double[1,nFaces]/double[nFacex,nDim] - indices of points in
          each face of bpMat graph

GETBOUNDARYBYFACTOR - computes grid of 2d or 3d ellipsoid and vertices
                      for each face in the grid

GETCENTERVEC - returns centerVec vector of given ellipsoid

Input:
  regular:
     self: ellipsoid[1,1]

Output:
  centerVecVec: double[nDims,1] - centerVec of ellipsoid

Example:
  ellObj = ellipsoid([1; 2], eye(2));
  getCenterVec(ellObj)

  ans =

       1
       2

GETCOPY - gives array the same size as ellArr with copies of elements of
          ellArr.

Input:
  regular:
      ellArr: ellipsoid[nDim1, nDim2,...] - multidimensional array of
          ellipsoids.

Output:
  copyEllArr: ellipsoid[nDim1, nDim2,...] - multidimension array of
      copies of elements of ellArr.

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1; 2], eye(2));
  ellVec = [firstEllObj secEllObj];
  copyEllVec = getCopy(ellVec)
```

```
  copyEllVec =
  1x2 array of ellipsoids.

GETINV - do the same as INV method: inverts shape matrices of ellipsoids
      in the given array, with only difference, that it doesn't modify
      input array of ellipsoids.

Input:
  regular:
    myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids.

Output:
   invEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with inverted shape matrices.

Example:
  ellObj = ellipsoid([1; 1], [4 -1; -1 5]);
  invEllObj = ellObj.getInv()

  invEllObj =

  Center:
       1
       1

  Shape Matrix:
      0.2632    0.0526
      0.0526    0.2105

  Nondegenerate ellipsoid in R^2.

GETMOVE2ORIGIN - do the same as MOVE2ORIGIN method: moves ellipsoids in
      the given array to the origin, with only difference, that it doesn't
      modify input array of ellipsoids.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
          ellipsoids.

Output:
  outEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with the same shapes as in inpEllArr centered at the origin.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  outEllObj = ellObj.getMove2Origin()

  outEllObj =

  Center:
       0
       0

  Shape:
       4    -1
      -1     1
```

```
  Nondegenerate ellipsoid in R^2.

GETNPLOT2DPOINTS - gives value of nPlot2dPoints property of ellipsoids
                   in ellArr

Input:
  regular:
      ellArr: ellipsoid[nDim1, nDim2,...] - mltidimensional array of
          ellipsoids

Output:
      nPlot2dPointsArr: double[nDim1, nDim2,...] - multidimension array
          of nPlot2dPoints property for ellipsoids in ellArr
Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 ;2], eye(2));
  ellVec = [firstEllObj secEllObj];
  ellVec.getNPlot2dPoints()

  ans =

     200   200

GETNPLOT3DPOINTS - gives value of nPlot3dPoints property of ellipsoids
                   in ellArr

Input:
  regular:
      ellArr: ellipsoid[nDim1, nDim2,...] - mltidimensional array  of
         ellipsoids

Output:
      nPlot2dPointsArr: double[nDim1, nDim2,...] - multidimension array
          of nPlot3dPoints property for ellipsoids in ellArr

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 ;2], eye(2));
  ellVec = [firstEllObj secEllObj];
  ellVec.getNPlot3dPoints()

  ans =

     200   200

GETPROJECTION - do the same as PROJECTION method: computes projection of
      the ellipsoid onto the given subspace, with only difference, that
      it doesn't modify input array of ellipsoids.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.
      basisMat: double[nDim, nSubSpDim] - matrix of orthogonal basis
          vectors

Output:
  projEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
```

```
      projected ellipsoids, generally, of lower dimension.

Example:
  ellObj = ellipsoid([-2; -1; 4], [4 -1 0; -1 1 0; 0 0 9]);
  basisMat = [0 1 0; 0 0 1]';
  outEllObj = ellObj.getProjection(basisMat)

  outEllObj =

  Center:
      -1
       4

  Shape:
      1     0
      0     9

  Nondegenerate ellipsoid in R^2.

GETRELTOL - gives the array of relTol for all elements in ellArr

Input:
  regular:
      ellArr: ellipsoid[nDim1, nDim2, ...] - multidimension array
          of ellipsoids
  optional:
      fRelTolFun: function_handle[1,1] - function that apply
          to the relTolArr. The default is @min.
Output:
  regular:
      relTolArr: double [relTol1, relTol2, ...] - return relTol for
          each element in ellArr
  optional:
      relTol: double[1,1] - return result of work fRelTolFun with
          the relTolArr

Usage:
  use [~,relTol] = ellArr.getRelTol() if you want get only
      relTol,
  use [relTolArr,relTol] = ellArr.getRelTol() if you want get
      relTolArr and relTol,
  use relTolArr = ellArr.getRelTol() if you want get only relTolArr

Example:
  firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
  secEllObj = ellipsoid([1 ;2], eye(2));
  ellVec = [firstEllObj secEllObj];
  ellVec.getRelTol()

  ans =

     1.0e-05 *

      1.0000    1.0000

GETSHAPE -  do the same as SHAPE method: modifies the shape matrix of the
   ellipsoid without changing its center, with only difference, that
   it doesn't modify input array of ellipsoids.
```

```
Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.
      modMat: double[nDim, nDim]/[1,1] - square matrix or scalar

Output:
   outEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of modified
      ellipsoids.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  tempMat = [0 1; -1 0];
  outEllObj = ellObj.getShape(tempMat)

  outEllObj =

  Center:
      -2
      -1

  Shape:
      1     1
      1     4

  Nondegenerate ellipsoid in R^2.

GETSHAPEMAT - returns shapeMat matrix of given ellipsoid

Input:
  regular:
     self: ellipsoid[1,1]

Output:
  shMat: double[nDims,nDims] - shapeMat matrix of ellipsoid

Example:
  ellObj = ellipsoid([1; 2], eye(2));
  getShapeMat(ellObj)

  ans =

       1     0
       0     1

HPINTERSECTION - computes the intersection of ellipsoid with hyperplane.

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN]/[1,1] - array
          of ellipsoids.
      myHypArr: hyperplane [nDims1,nDims2,...,nDimsN]/[1,1] - array
          of hyperplanes of the same size.

Output:
  intEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      resulting from intersections.
```

```
  isnIntersectedArr: logical [nDims1,nDims2,...,nDimsN].
      isnIntersectedArr(iCount) = true, if myEllArr(iCount)
      doesn't intersect myHipArr(iCount),
      isnIntersectedArr(iCount) = false, otherwise.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  hypMat = [hyperplane([0 -1; -1 0]', 1); hyperplane([0 -2; -1 0]', 1)];
  ellMat = ellObj.hpintersection(hypMat)

  ellMat =
  2x2 array of ellipsoids.

INTERSECT - checks if the union or intersection of ellipsoids intersects
            given ellipsoid, hyperplane or polytope.

  resArr = INTERSECT(myEllArr, objArr, mode) - Checks if the union
      (mode = 'u') or intersection (mode = 'i') of ellipsoids
      in myEllArr intersects with objects in objArr.
      objArr can be array of ellipsoids, array of hyperplanes,
      or array of polytopes.
      Ellipsoids, hyperplanes or polytopes in objMat must have
      the same dimension as ellipsoids in myEllArr.
      mode = 'u' (default) - union of ellipsoids in myEllArr.
      mode = 'i' - intersection.

  If we need to check the intersection of union of ellipsoids in
  myEllArr (mode = 'u'), or if myEllMat is a single ellipsoid,
  it can be done by calling distance function for each of the
  ellipsoids in myEllArr and objMat, and if it returns negative value,
  the intersection is nonempty. Checking if the intersection of
  ellipsoids in myEllArr (with size of myEllMat greater than 1)
  intersects with ellipsoids or hyperplanes in objArr is more
  difficult. This problem can be formulated as quadratically
  constrained quadratic programming (QCQP) problem.

  Let objArr(iObj) = E(q, Q) be an ellipsoid with center q and shape
  matrix Q. To check if this ellipsoid intersects (or touches) the
  intersection of ellipsoids in meEllArr: E(q1, Q1), E(q2, Q2), ...,
  E(qn, Qn), we define the QCQP problem:
                    J(x) = <(x - q), Q^(-1)(x - q)> --> min
  with constraints:
                     <(x - q1), Q1^(-1)(x - q1)> <= 1   (1)
                     <(x - q2), Q2^(-1)(x - q2)> <= 1   (2)
                     ................................
                     <(x - qn), Qn^(-1)(x - qn)> <= 1   (n)

  If this problem is feasible, i.e. inequalities (1)-(n) do not
  contradict, or, in other words, intersection of ellipsoids
  E(q1, Q1), E(q2, Q2), ..., E(qn, Qn) is nonempty, then we can find
  vector y such that it satisfies inequalities (1)-(n) and minimizes
  function J. If J(y) <= 1, then ellipsoid E(q, Q) intersects or touches
  the given intersection, otherwise, it does not. To check if E(q, Q)
  intersects the union of E(q1, Q1), E(q2, Q2), ..., E(qn, Qn),
  we compute the distances from this ellipsoids to those in the union.
  If at least one such distance is negative,
  then E(q, Q) does intersect the union.
```

```
  If we check the intersection of ellipsoids with hyperplane
  objArr = H(v, c), it is enough to check the feasibility
  of the problem
                      1'x --> min
  with constraints (1)-(n), plus
                    <v, x> - c = 0.

  Checking the intersection of ellipsoids with polytope
  objArr = P(A, b) reduces to checking if there any x, satisfying
  constraints (1)-(n) and
                       Ax <= b.

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
           ellipsoids.
      objArr: ellipsoid / hyperplane /
          / polytope [nDims1,nDims2,...,nDimsN] - array of ellipsoids or
          hyperplanes or polytopes of the same sizes.

  optional:
      mode: char[1, 1] - 'u' or 'i', go to description.

          note: If mode == 'u', then mRows, nCols should be equal to 1.

Output:
  resArr: double[nDims1,nDims2,...,nDimsN] - return:
      resArr(iCount) = -1 in case parameter mode is set
          to 'i' and the intersection of ellipsoids in myEllArr
          is empty.
      resArr(iCount) = 0 if the union or intersection of
          ellipsoids in myEllArr does not intersect the object
          in objArr(iCount).
      resArr(iCount) = 1 if the union or intersection of
          ellipsoids in myEllArr and the object in objArr(iCount)
          have nonempty intersection.
  statusArr: double[0, 0]/double[nDims1,nDims2,...,nDimsN] - status
      variable. statusArr is empty if mode = 'u'.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = firstEllObj + [5; 5];
  hypObj  = hyperplane([1; -1]);
  ellVec = [firstEllObj secEllObj];
  ellVec.intersect(hypObj)

  ans =

       1

  ellVec.intersect(hypObj, 'i')

  ans =

      -1

INTERSECTION_EA - external ellipsoidal approximation of the
                  intersection of two ellipsoids, or ellipsoid and
```

```
                  halfspace, or ellipsoid and polytope.

  outEllArr = INTERSECTION_EA(myEllArr, objArr) Given two ellipsoidal
      matrixes of equal sizes, myEllArr and objArr = ellArr, or,
      alternatively, myEllArr or ellMat must be a single ellipsoid,
      computes the ellipsoid that contains the intersection of two
      corresponding ellipsoids from myEllArr and from ellArr.
  outEllArr = INTERSECTION_EA(myEllArr, objArr) Given matrix of
      ellipsoids myEllArr and matrix of hyperplanes objArr = hypArr
      whose sizes match, computes the external ellipsoidal
      approximations of intersections of ellipsoids
      and halfspaces defined by hyperplanes in hypArr.
      If v is normal vector of hyperplane and c - shift,
      then this hyperplane defines halfspace
              <v, x> <= c.
  outEllArr = INTERSECTION_EA(myEllArr, objArr) Given matrix of
      ellipsoids myEllArr and matrix of polytopes objArr = polyArr
      whose sizes match, computes the external ellipsoidal
      approximations of intersections of ellipsoids myEllMat and
      polytopes polyArr.

  The method used to compute the minimal volume overapproximating
  ellipsoid is described in "Ellipsoidal Calculus Based on
  Propagation and Fusion" by Lluis Ros, Assumpta Sabater and
  Federico Thomas; IEEE Transactions on Systems, Man and Cybernetics,
  Vol.32, No.4, pp.430-442, 2002. For more information, visit
  http://www-iri.upc.es/people/ros/ellipsoids.html

  For polytopes this method won't give the minimal volume
  overapproximating ellipsoid, but just some overapproximating ellipsoid.

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN]/[1,1] - array
          of ellipsoids.
      objArr: ellipsoid / hyperplane /
          / polytope [nDims1,nDims2,...,nDimsN]/[1,1]  - array of
          ellipsoids or hyperplanes or polytopes of the same sizes.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = firstEllObj + [5; 5];
  ellVec = [firstEllObj secEllObj];
  thirdEllObj  = ell_unitball(2);
  externalEllVec = ellVec.intersection_ea(thirdEllObj)

  externalEllVec =
  1x2 array of ellipsoids.

INTERSECTION_IA - internal ellipsoidal approximation of the
                  intersection of ellipsoid and ellipsoid,
                  or ellipsoid and halfspace, or ellipsoid
                  and polytope.

  outEllArr = INTERSECTION_IA(myEllArr, objArr) - Given two
      ellipsoidal matrixes of equal sizes, myEllArr and
      objArr = ellArr, or, alternatively, myEllMat or ellMat must be
      a single ellipsoid, comuptes the internal ellipsoidal
```

```
      approximations of intersections of two corresponding ellipsoids
      from myEllMat and from ellMat.
  outEllArr = INTERSECTION_IA(myEllArr, objArr) - Given matrix of
      ellipsoids myEllArr and matrix of hyperplanes objArr = hypArr
      whose sizes match, computes the internal ellipsoidal
      approximations of intersections of ellipsoids and halfspaces
      defined by hyperplanes in hypMat.
      If v is normal vector of hyperplane and c - shift,
      then this hyperplane defines halfspace
                 <v, x> <= c.
  outEllArr = INTERSECTION_IA(myEllArr, objArr) - Given matrix of
      ellipsoids  myEllArr and matrix of polytopes objArr = polyArr
      whose sizes match, computes the internal ellipsoidal
      approximations of intersections of ellipsoids myEllArr
      and polytopes polyArr.

  The method used to compute the minimal volume overapproximating
  ellipsoid is described in "Ellipsoidal Calculus Based on
  Propagation and Fusion" by Lluis Ros, Assumpta Sabater and
  Federico Thomas; IEEE Transactions on Systems, Man and Cybernetics,
  Vol.32, No.4, pp.430-442, 2002. For more information, visit
  http://www-iri.upc.es/people/ros/ellipsoids.html

  The method used to compute maximum volume ellipsoid inscribed in
  intersection of ellipsoid and polytope, is modified version of
  algorithm of finding maximum volume ellipsoid inscribed in intersection
  of ellipsoids discribed in Stephen Boyd and Lieven Vandenberghe "Convex
  Optimization". It works properly for nondegenerate ellipsoid, but for
  degenerate ellipsoid result would not lie in this ellipsoid. The result
  considered as empty ellipsoid, when maximum absolute velue of element
  in its matrix is less than myEllipsoid.getAbsTol().

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN]/[1,1] - array
          of ellipsoids.
      objArr: ellipsoid / hyperplane /
          / polytope [nDims1,nDims2,...,nDimsN]/[1,1]  - array of
          ellipsoids or hyperplanes or polytopes of the same sizes.

Output:
   outEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of internal
      approximating ellipsoids; entries can be empty ellipsoids
      if the corresponding intersection is empty.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = firstEllObj + [5; 5];
  ellVec = [firstEllObj secEllObj];
  thirdEllObj  = ell_unitball(2);
  internalEllVec = ellVec.intersection_ia(thirdEllObj)

  internalEllVec =
  1x2 array of ellipsoids.

INV - inverts shape matrices of ellipsoids in the given array,
      modified given array is on output (not its copy).
```

```
  invEllArr = INV(myEllArr)  Inverts shape matrices of ellipsoids
      in the array myEllMat. In case shape matrix is sigular, it is
      regularized before inversion.

Input:
  regular:
    myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids.

Output:
   myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with inverted shape matrices.

Example:
  ellObj = ellipsoid([1; 1], [4 -1; -1 5]);
  ellObj.inv()

  ans =

  Center:
       1
       1

  Shape Matrix:
      0.2632    0.0526
      0.0526    0.2105

  Nondegenerate ellipsoid in R^2.

ISEMPTY - checks if the ellipsoid object is empty.

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
           ellipsoids.

Output:
  isPositiveArr: logical[nDims1,nDims2,...,nDimsN],
      isPositiveArr(iCount) = true - if ellipsoid
      myEllMat(iCount) is empty, false - otherwise.

Example:
  ellObj = ellipsoid();
  isempty(ellObj)

  ans =

       1

ISEQUAL - produces logical array the same size as
          ellFirstArr/ellFirstArr (if they have the same).
          isEqualArr[iDim1, iDim2,...] is true if corresponding
          ellipsoids are equal and false otherwise.

Input:
  regular:
      ellFirstArr: ellipsoid[nDim1, nDim2,...] - multidimensional array
          of ellipsoids.
      ellSecArr: ellipsoid[nDim1, nDim2,...] - multidimensional array
```

```
        of ellipsoids.
  properties:
      'isPropIncluded': makes to compare second value properties, such as
      absTol etc.
Output:
  isEqualArr: logical[nDim1, nDim2,...] - multidimension array of
      logical values. isEqualArr[iDim1, iDim2,...] is true if
      corresponding ellipsoids are equal and false otherwise.

  reportStr: char[1,] - comparison report.

ISINSIDE - checks if given ellipsoid(or array of
           ellipsoids) lies inside given object(or array
           of objects): ellipsoid or polytope.

Input:
  regular:
      ellArr: ellipsoid[nDims1,nDims2,...,nDimsN] - array
              of ellipsoids of the same dimension.
      objArr: ellipsoid/
              polytope[nDims1,nDims2,...,nDimsN] of
              objects of the same dimension. If
              ellArr and objArr both non-scalar, than
              size of ellArr must be the same as size of
              objArr. Note that polytopes could be
              combined only in vector of size [1,N].
Output:
  regular:
      resArr: logical[nDims1,nDims2,...,nDimsN] array of
              results. resArr[iDim1,...,iDimN] = true, if
              ellArr[iDim1,...,iDimN] lies inside
              objArr[iDim1,...,iDimN].

Example:
  firstEllObj = [0 ; 0] + ellipsoid(eye(2, 2));
  secEllObj = [0 ; 0] + ellipsoid(2*eye(2, 2));
  firstEllObj.isInside(secEllObj)

  ans =

       1

ISBADDIRECTION - checks if ellipsoidal approximations of geometric
                 difference of two ellipsoids can be computed for
                 given directions.
  isBadDirVec = ISBADDIRECTION(fstEll, secEll, dirsMat) - Checks if
      it is possible to build ellipsoidal approximation of the
      geometric difference of two ellipsoids fstEll - secEll in
      directions specified by matrix dirsMat (columns of dirsMat
      are direction vectors). Type 'help minkdiff_ea' or
      'help minkdiff_ia' for more information.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid. Suppose nDim - space
          dimension.
      secEll: ellipsoid [1, 1] - second ellipsoid of the same dimention.
      dirsMat: numeric[nDims, nCols] - matrix whose columns are
```

```
          direction vectors that need to be checked.
      absTol: double [1,1] - absolute tolerance

Output:
   isBadDirVec: logical[1, nCols] - array of true or false with length
      being equal to the number of columns in matrix dirsMat.
      ture marks direction vector as bad - ellipsoidal approximation
      true marks direction vector as bad - ellipsoidal approximation
      cannot be computed for this direction. false means the opposite.

ISBIGGER - checks if one ellipsoid would contain the other if their
           centers would coincide.

  isPositive = ISBIGGER(fstEll, secEll) - Given two single ellipsoids
      of the same dimension, fstEll and secEll, check if fstEll
      would contain secEll inside if they were both
      centered at origin.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid.
      secEll: ellipsoid [1, 1] - second ellipsoid
          of the same dimention.

Output:
  isPositive: logical[1, 1], true - if ellipsoid fstEll
      would contain secEll inside, false - otherwise.

Example:
  firstEllObj = ellipsoid([1; 1], eye(2));
  secEllObj = ellipsoid([1; 1], [4 -1; -1 5]);
  isbigger(firstEllObj, secEllObj)

  ans =

       0

ISDEGENERATE - checks if the ellipsoid is degenerate.

Input:
  regular:
      myEllArr: ellipsoid[nDims1,nDims2,...,nDimsN] - array of ellipsoids.

Output:
  isPositiveArr: logical[nDims1,nDims2,...,nDimsN],
      isPositiveArr(iCount) = true if ellipsoid myEllMat(iCount)
      is degenerate, false - otherwise.

Example:
  ellObj = ellipsoid([1; 1], eye(2));
  isdegenerate(ellObj)

  ans =

       0

ISINTERNAL - checks if given points belong to the union or intersection
             of ellipsoids in the given array.
```

```
  isPositiveVec = ISINTERNAL(myEllArr,  matrixOfVecMat, mode) - Checks
      if vectors specified as columns of matrix matrixOfVecMat
      belong to the union (mode = 'u'), or intersection (mode = 'i')
      of the ellipsoids in myEllArr. If myEllArr is a single
      ellipsoid, then this function checks if points in matrixOfVecMat
      belong to myEllArr or not. Ellipsoids in myEllArr must be
      of the same dimension. Column size of matrix  matrixOfVecMat
      should match the dimension of ellipsoids.

   Let myEllArr(iEll) = E(q, Q) be an ellipsoid with center q and shape
   matrix Q. Checking if given vector matrixOfVecMat = x belongs
   to E(q, Q) is equivalent to checking if inequality
                   <(x - q), Q^(-1)(x - q)> <= 1
   holds.
   If x belongs to at least one of the ellipsoids in the array, then it
   belongs to the union of these ellipsoids. If x belongs to all
   ellipsoids in the array,
   then it belongs to the intersection of these ellipsoids.
   The default value of the specifier s = 'u'.

   WARNING: be careful with degenerate ellipsoids.

Input:
  regular:
      myEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.
      matrixOfVecMat: double [mRows, nColsOfVec] - matrix which
          specifiy points.

  optional:
      mode: char[1, 1] - 'u' or 'i', go to description.

Output:
   isPositiveVec: logical[1, nColsOfVec] -
      true - if vector belongs to the union or intersection
      of ellipsoids, false - otherwise.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = firstEllObj + [5; 5];
  ellVec = [firstEllObj secEllObj];
  ellVec.isinternal([-2 3; -1 4], 'i')

  ans =

       0     0

  ellVec.isinternal([-2 3; -1 4])

  ans =

       1     1

MAXEIG - return the maximal eigenvalue of the ellipsoid.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
```

```
           ellipsoids.

Output:
  maxEigArr: double[nDims1,nDims2,...,nDimsN] - array of maximal
      eigenvalues of ellipsoids in the input matrix inpEllMat.

Example:
  ellObj = ellipsoid([-2; 4], [4 -1; -1 5]);
  maxEig = maxeig(ellObj)

  maxEig =

      5.6180

MINEIG - return the minimal eigenvalue of the ellipsoid.

Input:
   regular:
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
        ellipsoids.

Output:
   minEigArr: double[nDims1,nDims2,...,nDimsN] - array of minimal
      eigenvalues of ellipsoids in the input array inpEllMat.

Example:
  ellObj = ellipsoid([-2; 4], [4 -1; -1 5]);
  minEig = mineig(ellObj)

  minEig =

      3.3820

MINKCOMMONACTION - plot Minkowski operation  of ellipsoids in 2D or 3D.
Usage:
minkCommonAction(getEllArr,fCalcBodyTriArr,...
   fCalcCenterTriArr,varargin) -  plot Minkowski operation  of
           ellipsoids in 2D or 3D, using triangulation  of output object

Input:
  regular:
      getEllArr:  Ellipsoid: [dim11Size,dim12Size,...,dim1kSize] -
                array of 2D or 3D Ellipsoids objects. All ellipsoids in
                ellArr must be either 2D or 3D simutaneously.
fCalcBodyTriArr - function, calculeted triangulation of output object
   fCalcCenterTriArr - function, calculeted center  of output object
           properties:
      'shawAll': logical[1,1] - if 1, plot all ellArr.
                    Default value is 0.
      'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
               if 1, ellipsoids in 2D will be filled with color.
               Default value is 0.
      'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
                    line width for 1D and 2D plots. Default value is 1.
      'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
                sets default colors in the form [x y z].
               Default value is [1 0 0].
      'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
```

```
               level of transparency between 0 and 1
                  (0 - transparent, 1 - opaque).
               Default value is 0.4.
      'relDataPlotter' - relation data plotter object.

Output:
  centVec: double[nDim, 1] - center of the resulting set.
  boundPointMat: double[nDim, nBoundPoints] - set of boundary
      points (vertices) of resulting set.

MINKDIFF - computes geometric (Minkowski) difference of two
            ellipsoids in 2D or 3D.
 Usage:
MINKDIFF(inpEllMat,'Property',PropValue,...) - Computes
geometric difference of two ellipsoids in the array inpEllMat, if
1 <= min(dimension(inpEllMat)) = max(dimension(inpEllMat)) <= 3,
       and plots it if no output arguments are specified.

   [centVec, boundPointMat] = MINKDIFF(inpEllMat) - Computes
       geometric difference of two ellipsoids in inpEllMat.
       Here centVec is
       the center, and boundPointMat - array of boundary points.
   MINKDIFF(inpEllMat) - Plots geometric differencr of two
   ellipsoids in inpEllMat in default (red) color.
   MINKDIFF(inpEllMat, 'Property',PropValue,...) -
    Plots geometric sum of inpEllMat
       with setting properties.

   In order for the geometric difference to be nonempty set,
   ellipsoid fstEll must be bigger than secEll in the sense that
   if fstEll and secEll had the same centerVec, secEll would be
   contained inside fstEll.
 Input:
   regular:
       ellArr:  Ellipsoid: [dim11Size,dim12Size,...,dim1kSize] -
                array of 2D or 3D Ellipsoids objects. All ellipsoids in ellArr
                must be either 2D or 3D simutaneously.

   properties:
       'shawAll': logical[1,1] - if 1, plot all ellArr.
                    Default value is 0.
       'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
               if 1, ellipsoids in 2D will be filled with color.
               Default value is 0.
       'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
                    line width for 1D and 2D plots. Default value is 1.
       'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
                sets default colors in the form [x y z].
               Default value is [1 0 0].
       'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
                level of transparency between 0 and 1
                   (0 - transparent, 1 - opaque).
                Default value is 0.4.
       'relDataPlotter' - relation data plotter object.
       Notice that property vector could have different dimensions, only
       total number of elements must be the same.

 Output:
```

```
   centVec: double[nDim, 1] - center of the resulting set.
   boundPointMat: double[nDim, nBoundPoints] - set of boundary
       points (vertices) of resulting set.

 Example:
   firstEllObj = ellipsoid([-1; 1], [2 0; 0 3]);
   secEllObj = ellipsoid([1 2], eye(2));
   [centVec, boundPointMat] = minkdiff(firstEllObj, secEllObj);

MINKDIFF_EA - computation of external approximating ellipsoids
              of the geometric difference of two ellipsoids along
              given directions.

  extApprEllVec = MINKDIFF_EA(fstEll, secEll, directionsMat) -
      Computes external approximating ellipsoids of the
      geometric difference of two ellipsoids fstEll - secEll
      along directions specified by columns of matrix directionsMat

  First condition for the approximations to be computed, is that
  ellipsoid fstEll = E1 must be bigger than ellipsoid secEll = E2
  in the sense that if they had the same center, E2 would be contained
  inside E1. Otherwise, the geometric difference E1 - E2
  is an empty set.
  Second condition for the approximation in the given direction l
  to exist, is the following. Given
      P = sqrt(<l, Q1 l>)/sqrt(<l, Q2 l>)
  where Q1 is the shape matrix of ellipsoid E1, and
  Q2 - shape matrix of E2, and R being minimal root of the equation
      det(Q1 - R Q2) = 0,
  parameter P should be less than R.
  If both of these conditions are satisfied, then external
  approximating ellipsoid is defined by its shape matrix
      Q = (Q1^(1/2) + S Q2^(1/2))' (Q1^(1/2) + S Q2^(1/2)),
  where S is orthogonal matrix such that vectors
      Q1^(1/2)l and SQ2^(1/2)l
  are parallel, and its center
      q = q1 - q2,
  where q1 is center of ellipsoid E1 and q2 - center of E2.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid. Suppose
          nDim - space dimension.
      secEll: ellipsoid [1, 1] - second ellipsoid
          of the same dimention.
      directionsMat: double[nDim, nCols] - matrix whose columns
          specify the directions for which the approximations
          should be computed.

Output:
  extApprEllVec: ellipsoid [1, nCols] - array of external
      approximating ellipsoids (empty, if for all specified
      directions approximations cannot be computed).

Example:
  firstEllObj= ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = 3*ell_unitball(2);
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
```

```
  externalEllVec = secEllObj.minkdiff_ea(firstEllObj, dirsMat)

  externalEllVec =
  1x2 array of ellipsoids.

MINKDIFF_IA - computation of internal approximating ellipsoids
              of the geometric difference of two ellipsoids along
              given directions.

  intApprEllVec = MINKDIFF_IA(fstEll, secEll, directionsMat) -
      Computes internal approximating ellipsoids of the geometric
      difference of two ellipsoids fstEll - secEll along directions
      specified by columns of matrix directionsMat.

  First condition for the approximations to be computed, is that
  ellipsoid fstEll = E1 must be bigger than ellipsoid secEll = E2
  in the sense that if they had the same center, E2 would be contained
  inside E1. Otherwise, the geometric difference E1 - E2 is an
  empty set. Second condition for the approximation in the given
  direction l to exist, is the following. Given
      P = sqrt(<l, Q1 l>)/sqrt(<l, Q2 l>)
  where Q1 is the shape matrix of ellipsoid E1,
  and Q2 - shape matrix of E2, and R being minimal root of the equation
      det(Q1 - R Q2) = 0,
  parameter P should be less than R.
  If these two conditions are satisfied, then internal approximating
  ellipsoid for the geometric difference E1 - E2 along the
  direction l is defined by its shape matrix
      Q = (1 - (1/P)) Q1 + (1 - P) Q2
  and its center
      q = q1 - q2,
  where q1 is center of E1 and q2 - center of E2.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid. Suppose
          nDim - space dimension.
      secEll: ellipsoid [1, 1] - second ellipsoid
          of the same dimention.
      directionsMat: double[nDim, nCols] - matrix whose columns
          specify the directions for which the approximations
          should be computed.

Output:
  intApprEllVec: ellipsoid [1, nCols] - array of internal
      approximating ellipsoids (empty, if for all specified directions
      approximations cannot be computed).

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = 3*ell_unitball(2);
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  internalEllVec = secEllObj.minkdiff_ia(firstEllObj, dirsMat)

  internalEllVec =
  1x2 array of ellipsoids.
```

```
MINKMP - computes and plots geometric (Minkowski) sum of the
         geometric difference of two ellipsoids and the geometric
         sum of n ellipsoids in 2D or 3D:
         (E - Em) + (E1 + E2 + ... + En),
         where E = firstEll, Em = secondEll,
         E1, E2, ..., En - are ellipsoids in sumEllArr

Usage:
  MINKMP(firEll,secEll,ellMat,'Property',PropValue,...) -
          Computes (E1 - E2) + (E3 + E4+ ... + En), if
      1 <= min(dimension(inpEllMat)) = max(dimension(inpEllMat)) <= 3,
      and plots it if no output arguments are specified.

  [centVec, boundPointMat] = MINKMP(firEll,secEll,ellMat) - Computes
     (E1 - E2) + (E3 + E4+ ... + En). Here centVec is
      the center, and boundPointMat - array of boundary points.
Input:
  regular:
      ellArr:  Ellipsoid: [dim11Size,dim12Size,...,dim1kSize] -
          array of 2D or 3D Ellipsoids objects. All ellipsoids in ellArr
               must be either 2D or 3D simutaneously.

  properties:
      'showAll': logical[1,1] - if 1, plot all ellArr.
                   Default value is 0.
      'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
              if 1, ellipsoids in 2D will be filled with color.
              Default value is 0.
      'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]-
                   line width for 1D and 2D plots. Default value is 1.
      'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
               sets default colors in the form [x y z].
                  Default value is [1 0 0].
      'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
               level of transparency between 0 and 1
              (0 - transparent, 1 - opaque).
               Default value is 0.4.
      'relDataPlotter' - relation data plotter object.
      Notice that property vector could have different dimensions, only
      total number of elements must be the same.

Output:
  centVec: double[nDim, 1] - center of the resulting set.
  boundPointMat: double[nDim, nBoundPoints] - set of boundary
      points (vertices) of resulting set.

Example:
  firstEllObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj secEllObj ellipsoid([-3; 1], eye(2))];
  minkmp(firstEllObj, secEllObj, ellVec);

MINKMP_EA - computation of external approximating ellipsoids
            of (E - Em) + (E1 + ... + En) along given directions.
            where E = fstEll, Em = secEll,
            E1, E2, ..., En - are ellipsoids in sumEllArr

  extApprEllVec = MINKMP_EA(fstEll, secEll, sumEllArr, dirMat) -
```

```
      Computes external approximating
      ellipsoids of (E - Em) + (E1 + E2 + ... + En),
      where E1, E2, ..., En are ellipsoids in array sumEllArr,
      E = fstEll, Em = secEll,
      along directions specified by columns of matrix dirMat.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid. Suppose
          nDims - space dimension.
      secEll: ellipsoid [1, 1] - second ellipsoid
          of the same dimention.
      sumEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions nDims.
      dirMat: double[nDims, nCols] - matrix whose columns specify the
          directions for which the approximations should be computed.

Output:
  extApprEllVec: ellipsoid [1, nCols] - array of external
      approximating ellipsoids (empty, if for all specified
      directions approximations cannot be computed).

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = 3*ell_unitball(2);
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  bufEllVec = [secEllObj firstEllObj];
  externalEllVec = secEllObj.minkmp_ea(firstEllObj, bufEllVec, dirsMat)

  externalEllVec =
  1x2 array of ellipsoids.

MINKMP_IA - computation of internal approximating ellipsoids
            of (E - Em) + (E1 + ... + En) along given directions.
            where E = fstEll, Em = secEll,
            E1, E2, ..., En - are ellipsoids in sumEllArr

  intApprEllVec = MINKMP_IA(fstEll, secEll, sumEllArr, dirMat) -
      Computes internal approximating
      ellipsoids of (E - Em) + (E1 + E2 + ... + En),
      where E1, E2, ..., En are ellipsoids in array sumEllArr,
      E = fstEll, Em = secEll,
      along directions specified by columns of matrix dirMat.

Input:
  regular:
      fstEll: ellipsoid [1, 1] - first ellipsoid. Suppose
          nDim - space dimension.
      secEll: ellipsoid [1, 1] - second ellipsoid
          of the same dimention.
      sumEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions.
      dirMat: double[nDim, nCols] - matrix whose columns specify the
          directions for which the approximations should be computed.

Output:
  intApprEllVec: ellipsoid [1, nCols] - array of internal
      approximating ellipsoids (empty, if for all specified
```

```
      directions approximations cannot be computed).

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = 3*ell_unitball(2);
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  bufEllVec = [secEllObj firstEllObj];
  internalEllVec = secEllObj.minkmp_ia(firstEllObj, bufEllVec, dirsMat)

  internalEllVec =
  1x2 array of ellipsoids.

MINKPM - computes and plots geometric (Minkowski) difference
         of the geometric sum of ellipsoids and a single ellipsoid
         in 2D or 3D: (E1 + E2 + ... + En) - E,
         where E = inpEll,
         E1, E2, ... En - are ellipsoids in inpEllArr.

  MINKPM(inpEllArr, inpEll, OPTIONS)  Computes geometric difference
      of the geometric sum of ellipsoids in inpEllMat and
      ellipsoid inpEll, if
      1 <= dimension(inpEllArr) = dimension(inpArr) <= 3,
      and plots it if no output arguments are specified.

  [centVec, boundPointMat] = MINKPM(inpEllArr, inpEll) - pomputes
      (geometric sum of ellipsoids in inpEllArr) - inpEll.
      Here centVec is the center, and boundPointMat - array
      of boundary points.
  MINKPM(inpEllArr, inpEll) - plots (geometric sum of ellipsoids
      in inpEllArr) - inpEll in default (red) color.
  MINKPM(inpEllArr, inpEll, Options) - plots
      (geometric sum of ellipsoids in inpEllArr) - inpEll using
      options given in the Options structure.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions 2D or 3D.
      inpEll: ellipsoid [1, 1] - ellipsoid of the same
          dimention 2D or 3D.

  optional:
      Options: structure[1, 1] - fields:
          show_all: double[1, 1] - if 1, displays
              also ellipsoids fstEll and secEll.
          newfigure: double[1, 1] - if 1, each plot
              command will open a new figure window.
          fill: double[1, 1] - if 1, the resulting
              set in 2D will be filled with color.
          color: double[1, 3] - sets default colors
              in the form [x y z].
          shade: double[1, 1] = 0-1 - level of transparency
              (0 - transparent, 1 - opaque).

Output:
   centVec: double[nDim, 1]/double[0, 0] - center of the resulting set.
      centerVec may be empty.
   boundPointMat: double[nDim, ]/double[0, 0] - set of boundary
```

```
      points (vertices) of resulting set. boundPointMat may be empty.

MINKPM_EA - computation of external approximating ellipsoids
            of (E1 + E2 + ... + En) - E along given directions.
            where E = inpEll,
            E1, E2, ... En - are ellipsoids in inpEllArr.

  ExtApprEllVec = MINKPM_EA(inpEllArr, inpEll, dirMat) - Computes
      external approximating ellipsoids of
      (E1 + E2 + ... + En) - E, where E1, E2, ..., En are ellipsoids
      in array inpEllArr, E = inpEll,
      along directions specified by columns of matrix dirMat.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] -
          array of ellipsoids of the same dimentions.
      inpEll: ellipsoid [1, 1] - ellipsoid of the same dimention.
      dirMat: double[nDim, nCols] - matrix whose columns specify
          the directions for which the approximations
          should be computed.

Output:
  extApprEllVec: ellipsoid [1, nCols]/[0, 0] - array of external
      approximating ellipsoids. Empty, if for all specified
      directions approximations cannot be computed.

Example:
  firstEllObj = ellipsoid([2; -1], [9 -5; -5 4]);
  secEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  thirdEllObj = ell_unitball(2);
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  ellVec = [thirdEllObj firstEllObj];
  externalEllVec = ellVec.minkpm_ea(secEllObj, dirsMat)

  externalEllVec =
  1x4 array of ellipsoids.

MINKPM_IA - computation of internal approximating ellipsoids
            of (E1 + E2 + ... + En) - E along given directions.
            where E = inpEll,
            E1, E2, ... En - are ellipsoids in inpEllArr.

  intApprEllVec = MINKPM_IA(inpEllArr, inpEll, dirMat) - Computes
      internal approximating ellipsoids of
      (E1 + E2 + ... + En) - E, where E1, E2, ..., En are ellipsoids
      in array inpEllArr, E = inpEll,
      along directions specified by columns of matrix dirArr.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] -
          array of ellipsoids of the same dimentions.
      inpEll: ellipsoid [1, 1] - ellipsoid of the same dimention.
      dirMat: double[nDim, nCols] - matrix whose columns specify
          the directions for which the approximations
          should be computed.
```

```
Output:
  intApprEllVec: ellipsoid [1, nCols]/[0, 0] - array of internal
      approximating ellipsoids. Empty, if for all specified
      directions approximations cannot be computed.

Example:
  firstEllObj = ellipsoid([2; -1], [9 -5; -5 4]);
  secEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  thirdEllObj = ell_unitball(2);
  ellVec = [thirdEllObj firstEllObj];
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  internalEllVec = ellVec.minkpm_ia(secEllObj, dirsMat)

  internalEllVec =
  1x3 array of ellipsoids.

MINKSUM - computes geometric (Minkowski) sum of ellipsoids in 2D or 3D.

Usage:
  MINKSUM(inpEllMat,'Property',PropValue,...) - Computes geometric sum of
      ellipsoids in the array inpEllMat, if
      1 <= min(dimension(inpEllMat)) = max(dimension(inpEllMat)) <= 3,
      and plots it if no output arguments are specified.

  [centVec, boundPointMat] = MINKSUM(inpEllMat) - Computes
      geometric sum of ellipsoids in inpEllMat. Here centVec is
      the center, and boundPointMat - array of boundary points.
  MINKSUM(inpEllMat) - Plots geometric sum of ellipsoids in
      inpEllMat in default (red) color.
  MINKSUM(inpEllMat, 'Property',PropValue,...) - Plots geometric sum of
  inpEllMat with setting properties.

Input:
  regular:
      ellArr:  Ellipsoid: [dim11Size,dim12Size,...,dim1kSize] -
               array of 2D or 3D Ellipsoids objects. All ellipsoids
               in ellArr must be either 2D or 3D simutaneously.

  properties:
   'showAll': logical[1,1] - if 1, plot all ellArr.
                   Default value is 0.
   'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
              if 1, ellipsoids in 2D will be filled with color. Default
              value is 0.
   'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]-
                   line width for 1D and 2D plots. Default value is 1.
   'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
       sets default colors in the form [x y z]. Default value is [1 0 0].
   'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
     level of transparency between 0 and 1 (0 - transparent, 1 - opaque).
               Default value is 0.4.
      'relDataPlotter' - relation data plotter object.
      Notice that property vector could have different dimensions, only
      total number of elements must be the same.

Output:
  centVec: double[nDim, 1] - center of the resulting set.
  boundPointMat: double[nDim, nBoundPoints] - set of boundary
```

```
      points (vertices) of resulting set.

Example:
  firstEllObj = ellipsoid([-2; -1], [2 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj, secellObj]
  sumVec = minksum(ellVec);

MINKSUM_EA - computation of external approximating ellipsoids
             of the geometric sum of ellipsoids along given directions.

  extApprEllVec = MINKSUM_EA(inpEllArr, dirMat) - Computes
      tight external approximating ellipsoids for the geometric
      sum of the ellipsoids in the array inpEllArr along directions
      specified by columns of dirMat.
      If ellipsoids in inpEllArr are n-dimensional, matrix
      dirMat must have dimension (n x k) where k can be
      arbitrarily chosen.
      In this case, the output of the function will contain k
      ellipsoids computed for k directions specified in dirMat.

  Let inpEllArr consists of E(q1, Q1), E(q2, Q2), ..., E(qm, Qm) -
  ellipsoids in R^n, and dirMat(:, iCol) = l - some vector in R^n.
  Then tight external approximating ellipsoid E(q, Q) for the
  geometric sum E(q1, Q1) + E(q2, Q2) + ... + E(qm, Qm)
  along direction l, is such that
      rho(l | E(q, Q)) = rho(l | (E(q1, Q1) + ... + E(qm, Qm)))
  and is defined as follows:
      q = q1 + q2 + ... + qm,
      Q = (p1 + ... + pm)((1/p1)Q1 + ... + (1/pm)Qm),
  where
      p1 = sqrt(<l, Q1l>), ..., pm = sqrt(<l, Qml>).

Input:
  regular:
      inpEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] - array
          of ellipsoids of the same dimentions.
      dirMat: double[nDims, nCols] - matrix whose columns specify
          the directions for which the approximations
          should be computed.

Output:
  extApprEllVec: ellipsoid [1, nCols] - array of external
      approximating ellipsoids.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj secEllObj firstEllObj.inv()];
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  externalEllVec = ellVec.minksum_ea(dirsMat)

  externalEllVec =
  1x4 array of ellipsoids.

MINKSUM_IA - computation of internal approximating ellipsoids
             of the geometric sum of ellipsoids along given directions.
```

```
  intApprEllVec = MINKSUM_IA(inpEllArr, dirMat) - Computes
      tight internal approximating ellipsoids for the geometric
      sum of the ellipsoids in the array inpEllArr along directions
      specified by columns of dirMat. If ellipsoids in
      inpEllArr are n-dimensional, matrix dirMat must have
      dimension (n x k) where k can be arbitrarily chosen.
      In this case, the output of the function will contain k
      ellipsoids computed for k directions specified in dirMat.

  Let inpEllArr consist of E(q1, Q1), E(q2, Q2), ..., E(qm, Qm) -
  ellipsoids in R^n, and dirMat(:, iCol) = l - some vector in R^n.
  Then tight internal approximating ellipsoid E(q, Q) for the
  geometric sum E(q1, Q1) + E(q2, Q2) + ... + E(qm, Qm) along
  direction l, is such that
      rho(l | E(q, Q)) = rho(l | (E(q1, Q1) + ... + E(qm, Qm)))
  and is defined as follows:
      q = q1 + q2 + ... + qm,
      Q = (S1 Q1^(1/2) + ... + Sm Qm^(1/2))' *
          * (S1 Q1^(1/2) + ... + Sm Qm^(1/2)),
  where S1 = I (identity), and S2, ..., Sm are orthogonal
  matrices such that vectors
  (S1 Q1^(1/2) l), ..., (Sm Qm^(1/2) l) are parallel.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1, nDims2,...,nDimsN] - array
          of ellipsoids of the same dimentions.
      dirMat: double[nDim, nCols] - matrix whose columns specify the
          directions for which the approximations should be computed.

Output:
  intApprEllVec: ellipsoid [1, nCols] - array of internal
      approximating ellipsoids.

Example:
  firstEllObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj secEllObj firstEllObj.inv()];
  dirsMat = [1 0; 1 1; 0 1; -1 1]';
  internalEllVec = ellVec.minksum_ia(dirsMat)

  internalEllVec =
  1x4 array of ellipsoids.

MINUS - overloaded operator '-'

  outEllArr = MINUS(inpEllArr, inpVec) implements E(q, Q) - b
      for each ellipsoid E(q, Q) in inpEllArr.
  outEllArr = MINUS(inpVec, inpEllArr) implements b - E(q, Q)
      for each ellipsoid E(q, Q) in inpEllArr.

  Operation E - b where E = inpEll is an ellipsoid in R^n,
  and b = inpVec - vector in R^n. If E(q, Q) is an ellipsoid
  with center q and shape matrix Q, then
  E(q, Q) - b = E(q - b, Q).

Input:
  regular:
```

```
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
          ellipsoids of the same dimentions nDims.
      inpVec: double[nDims, 1] - vector.

Output:
   outEllVec: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with same shapes as inpEllVec, but with centers shifted by vectors
      in -inpVec.

Example:
  ellVec  = [ellipsoid([-2; -1], [4 -1; -1 1]) ell_unitball(2)];
  outEllVec = ellVec - [1; 1];
  outEllVec(1)

  ans =

  Center:
      -3
      -2

  Shape:
       4    -1
      -1     1

  Nondegenerate ellipsoid in R^2.

  outEllVec(2)

  ans =

  Center:
      -1
      -1

  Shape:
       1     0
       0     1

  Nondegenerate ellipsoid in R^2.

MOVE2ORIGIN - moves ellipsoids in the given array to the origin. Modified
              given array is on output (not its copy).

  outEllArr = MOVE2ORIGIN(inpEll) - Replaces the centers of
      ellipsoids in inpEllArr with zero vectors.

Input:
  regular:
      inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
          ellipsoids.

Output:
  inpEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with the same shapes as in inpEllArr centered at the origin.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  outEllObj = ellObj.move2origin()
```

```
  outEllObj =

  Center:
       0
       0

  Shape:
       4    -1
      -1     1

  Nondegenerate ellipsoid in R^2.

MTIMES - overloaded operator '*'.

  Multiplication of the ellipsoid by a matrix or a scalar.
  If inpEllVec(iEll) = E(q, Q) is an ellipsoid, and
  multMat = A - matrix of suitable dimensions,
  then A E(q, Q) = E(Aq, AQA').

Input:
  regular:
      multMat: double[mRows, nDims]/[1, 1] - scalar or
          matrix in R^{mRows x nDim}
      inpEllVec: ellipsoid [1, nCols] - array of ellipsoids.

Output:
  outEllVec: ellipsoid [1, nCols] - resulting ellipsoids.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  tempMat = [0 1; -1 0];
  outEllObj = tempMat*ellObj

  outEllObj =

  Center:
      -1
       2

  Shape:
       1     1
       1     4

  Nondegenerate ellipsoid in R^2.

PARAMETERS - returns parameters of the ellipsoid.

Input:
  regular:
      myEll: ellipsoid [1, 1] - single ellipsoid of dimention nDims.

Output:
  myEllCenterVec: double[nDims, 1] - center of the ellipsoid myEll.
  myEllShapeMat: double[nDims, nDims] - shape matrix
      of the ellipsoid myEll.

Example:
  ellObj = ellipsoid([-2; 4], [4 -1; -1 5]);
```

```
  [centVec shapeMat] = parameters(ellObj)
  centVec =

      -2
       4

  shapeMat =

      4    -1
     -1     5

PLOT - plots ellipsoids in 2D or 3D.


Usage:
      plot(ell) - plots ellipsoid ell in default (red) color.
      plot(ellArr) - plots an array of ellipsoids.
      plot(ellArr, 'Property',PropValue,...) - plots ellArr with setting
                                              properties.

Input:
  regular:
      ellArr:  Ellipsoid: [dim11Size,dim12Size,...,dim1kSize] -
               array of 2D or 3D Ellipsoids objects. All ellipsoids in ellArr
               must be either 2D or 3D simutaneously.
  optional:
      color1Spec: char[1,1] - color specification code, can be 'r','g',
                              etc (any code supported by built-in Matlab function).
      ell2Arr: Ellipsoid: [dim21Size,dim22Size,...,dim2kSize] -
                                          second ellipsoid array...
      color2Spec: char[1,1] - same as color1Spec but for ell2Arr
      ....
      ellNArr: Ellipsoid: [dimN1Size,dim22Size,...,dimNkSize] -
                                           N-th ellipsoid array
      colorNSpec - same as color1Spec but for ellNArr.
  properties:
      'newFigure': logical[1,1] - if 1, each plot command will open a new figure window.
                   Default value is 0.
      'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
              if 1, ellipsoids in 2D will be filled with color. Default value is 0.
      'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
                   line width for 1D and 2D plots. Default value is 1.
      'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
               sets default colors in the form [x y z]. Default value is [1 0 0].
      'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
               level of transparency between 0 and 1 (0 - transparent, 1 - opaque).
               Default value is 0.4.
      'relDataPlotter' - relation data plotter object.
      Notice that property vector could have different dimensions, only
      total number of elements must be the same.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

Examples:
      plot([ell1, ell2, ell3], 'color', [1, 0, 1; 0, 0, 1; 1, 0, 0]);
      plot([ell1, ell2, ell3], 'color', [1; 0; 1; 0; 0; 1; 1; 0; 0]);
```

```
      plot([ell1, ell2, ell3; ell1, ell2, ell3], 'shade', [1, 1, 1; 1, 1,
      1]);
      plot([ell1, ell2, ell3; ell1, ell2, ell3], 'shade', [1; 1; 1; 1; 1;
      1]);
      plot([ell1, ell2, ell3], 'shade', 0.5);
      plot([ell1, ell2, ell3], 'lineWidth', 1.5);
      plot([ell1, ell2, ell3], 'lineWidth', [1.5, 0.5, 3]);

PLUS - overloaded operator '+'

  outEllArr = PLUS(inpEllArr, inpVec) implements E(q, Q) + b
      for each ellipsoid E(q, Q) in inpEllArr.
  outEllArr = PLUS(inpVec, inpEllArr) implements b + E(q, Q)
      for each ellipsoid E(q, Q) in inpEllArr.

   Operation E + b (or b+E) where E = inpEll is an ellipsoid in R^n,
  and b=inpVec - vector in R^n. If E(q, Q) is an ellipsoid
  with center q and shape matrix Q, then
  E(q, Q) + b = b + E(q,Q) = E(q + b, Q).

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
          of the same dimentions nDims.
      bVec: double[nDims, 1] - vector.

Output:
  outEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids
      with same shapes as ellVec, but with centers shifted by vectors
      in inpVec.

Example:
  ellVec  = [ellipsoid([-2; -1], [4 -1; -1 1]) ell_unitball(2)];
  outEllVec = ellVec + [1; 1];
  outEllVec(1)

  ans =

  Center:
      -1
       0

  Shape:
      4    -1
     -1     1

  Nondegenerate ellipsoid in R^2.

  outEllVec(2)

  ans =

  Center:
       1
       1

  Shape:
      1     0
```

```
      0     1

  Nondegenerate ellipsoid in R^2.

POLAR - computes the polar ellipsoids.

  polEllArr = POLAR(ellArr)  Computes the polar ellipsoids for those
      ellipsoids in ellArr, for which the origin is an interior point.
      For those ellipsoids in E, for which this condition does not hold,
      an empty ellipsoid is returned.

  Given ellipsoid E(q, Q) where q is its center, and Q - its shape matrix,
  the polar set to E(q, Q) is defined as follows:
  P = { l in R^n  | <l, q> + sqrt(<l, Q l>) <= 1 }
  If the origin is an interior point of ellipsoid E(q, Q),
  then its polar set P is an ellipsoid.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.

Output:
  polEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
       polar ellipsoids.

Example:
  ellObj = ellipsoid([4 -1; -1 1]);
  ellObj.polar() == ellObj.inv()

  ans =

      1

PROJECTION - computes projection of the ellipsoid onto the given subspace.
             modified given array is on output (not its copy).

  projEllArr = projection(ellArr, basisMat)  Computes projection of the
      ellipsoid ellArr onto a subspace, specified by orthogonal
      basis vectors basisMat. ellArr can be an array of ellipsoids of
      the same dimension. Columns of B must be orthogonal vectors.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.
      basisMat: double[nDim, nSubSpDim] - matrix of orthogonal basis
          vectors

Output:
  ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of
      projected ellipsoids, generally, of lower dimension.

Example:
  ellObj = ellipsoid([-2; -1; 4], [4 -1 0; -1 1 0; 0 0 9]);
  basisMat = [0 1 0; 0 0 1]';
  outEllObj = ellObj.projection(basisMat)
```

```
  outEllObj =

  Center:
      -1
       4

  Shape:
      1     0
      0     9

  Nondegenerate ellipsoid in R^2.

REPMAT - is analogous to built-in repmat function with one exception - it
         copies the objects, not just the handles

Example:
  firstEllObj = ellipsoid([1; 2], eye(2));
  secEllObj = ellipsoid([1; 1], 2*eye(2));
  ellVec = [firstEllObj secEllObj];
  repMat(ellVec)

  ans =
  1x2 array of ellipsoids.

RHO - computes the values of the support function for given ellipsoid
      and given direction.

      supArr = RHO(ellArr, dirsMat)  Computes the support function of the
      ellipsoid ellArr in directions specified by the columns of matrix
      dirsMat. Or, if ellArr is array of ellipsoids, dirsMat is expected
      to be a single vector.

      [supArr, bpArr] = RHO(ellArr, dirstMat)  Computes the support function
      of the ellipsoid ellArr in directions specified by the columns of
      matrix dirsMat, and boundary points bpArr of this ellipsoid that
      correspond to directions in dirsMat. Or, if ellArr is array of
      ellipsoids, and dirsMat - single vector, then support functions and
      corresponding boundary points are computed for all the given
      ellipsoids in the array in the specified direction dirsMat.

      The support function is defined as
  (1)  rho(l | E) = sup { <l, x> : x belongs to E }.
      For ellipsoid E(q,Q), where q is its center and Q - shape matrix,
  it is simplified to
  (2)  rho(l | E) = <q, l> + sqrt(<l, Ql>)
  Vector x, at which the maximum at (1) is achieved is defined by
  (3)  q + Ql/sqrt(<l, Ql>)

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN]/[1,1] - array
          of ellipsoids.
      dirsMat: double[nDim,nDims1,nDims2,...,nDimsN]/
          double[nDim,nDirs]/[nDim,1] - array or matrix of directions.

Output:
      supArr: double [nDims1,nDims2,...,nDimsN]/[1,nDirs] - support function
      of the ellArr in directions specified by the columns of matrix
```

```
      dirsMat. Or, if ellArr is array of ellipsoids, support function of
      each ellipsoid in ellArr specified by dirsMat direction.

  bpArr: double[nDim,nDims1,nDims2,...,nDimsN]/
          double[nDim,nDirs]/[nDim,1] - array or matrix of boundary points

Example:
  ellObj = ellipsoid([-2; 4], [4 -1; -1 1]);
  dirsMat = [-2 5; 5 1];
  suppFuncVec = rho(ellObj, dirsMat)

  suppFuncVec =

      31.8102    3.5394

SHAPE - modifies the shape matrix of the ellipsoid without
  changing its center. Modified given array is on output (not its copy).

   modEllArr = SHAPE(ellArr, modMat)  Modifies the shape matrices of
      the ellipsoids in the ellipsoidal array ellArr. The centers
      remain untouched - that is the difference of the function SHAPE and
      linear transformation modMat*ellArr. modMat is expected to be a
      scalar or a square matrix of suitable dimension.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.
      modMat: double[nDim, nDim]/[1,1] - square matrix or scalar

Output:
   ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of modified
      ellipsoids.

Example:
  ellObj = ellipsoid([-2; -1], [4 -1; -1 1]);
  tempMat = [0 1; -1 0];
  outEllObj = shape(ellObj, tempMat)

  outEllObj =

  Center:
      -2
      -1

  Shape:
      1     1
      1     4

  Nondegenerate ellipsoid in R^2.

TOPOLYTOPE - for ellipsoid ell makes polytope object represanting the
             boundary of ell

Input:
  regular:
      ell: ellipsoid[1,1] - ellipsoid in 3D or 2D.
  optional:
```

```
      nPoints: double[1,1] - number of boundary points.
               Actually number of points in resulting
               polytope will be ecual to lowest
               number of points of icosaeder, that greater
               than nPoints.

Output:
  regular:
      poly: polytope[1,1] - polytop in 3D or 2D.

toStruct -- converts ellipsoid array into structural array.

Input:
  regular:
      ellArr: ellipsoid [nDim1, nDim2, ...] - array
          of ellipsoids.
Output:
  SDataArr: struct[nDims1,...,nDimsk] - structure array same size, as
      ellArr, contain all data.
  SFieldNiceNames: struct[1,1] - structure with the same fields as SDataArr. Field values
      contain the nice names.
  SFieldDescr: struct[1,1] - structure with same fields as SDataArr,
      values contain field descriptions.

      q: double[1, nEllDim] - the center of ellipsoid
      Q: double[nEllDim, nEllDim] - the shape matrix of ellipsoid

Example:
  ellObj = ellipsoid([1 1]', eye(2));
  ellObj.toStruct()

  ans =

  Q: [2x2 double]
  q: [1 1]

TRACE - returns the trace of the ellipsoid.

   trArr = TRACE(ellArr)  Computes the trace of ellipsoids in
      ellipsoidal array ellArr.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.

Output:
   trArr: double [nDims1,nDims2,...,nDimsN] - array of trace values,
      same size as ellArr.

Example:
  firstEllObj = ellipsoid([4 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj secEllObj];
  trVec = ellVec.trace()

  trVec =
```

```
      5     2

UMINUS - changes the sign of the centerVec of ellipsoid.

Input:
   regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids.


Output:
   outEllArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array of ellipsoids,
       same size as ellArr.

Example:
  ellObj = -ellipsoid([-2; -1], [4 -1; -1 1])

  ellObj =

  Center:
       2
       1

  Shape:
       4    -1
      -1     1

  Nondegenerate ellipsoid in R^2.

VOLUME - returns the volume of the ellipsoid.

   volArr = VOLUME(ellArr)  Computes the volume of ellipsoids in
      ellipsoidal array ellArr.

   The volume of ellipsoid E(q, Q) with center q and shape matrix Q
   is given by V = S sqrt(det(Q)) where S is the volume of unit ball.

Input:
  regular:
      ellArr: ellipsoid [nDims1,nDims2,...,nDimsN] - array
          of ellipsoids.

Output:
   volArr: double [nDims1,nDims2,...,nDimsN] - array of
      volume values, same size as ellArr.

Example:
  firstEllObj = ellipsoid([4 -1; -1 1]);
  secEllObj = ell_unitball(2);
  ellVec = [firstEllObj secEllObj]
  volVec = ellVec.volume()

  volVec =

      5.4414    3.1416
```

9.2 hyperplane

```
CHECKISME - determine whether input object is hyperplane. And display
            message and abort function if input object
            is not hyperplane

Input:
  regular:
      someObjArr: any[] - any type array of objects.

Example:
  hypObj = hyperplane([-2, 0]);
  hyperplane.checkIsMe(hypObj)

CONTAINS - checks if given vectors belong to the hyperplanes.

  isPosArr = CONTAINS(myHypArr, xArr) - Checks if vectors specified
      by columns xArr(:, hpDim1, hpDim2, ...) belong
      to hyperplanes in myHypArr.

Input:
  regular:
      myHypArr: hyperplane [nCols, 1]/[1, nCols]/
          /[hpDim1, hpDim2, ...]/[1, 1] - array of hyperplanes
          of the same dimentions nDims.
      xArr: double[nDims, nCols]/[nDims, hpDim1, hpDim2, ...]/
          /[nDims, 1]/[nDims, nVecArrDim1, nVecArrDim2, ...] - array
          whose columns represent the vectors needed to be checked.

          note: if size of myHypArr is [hpDim1, hpDim2, ...], then
              size of xArr is [nDims, hpDim1, hpDim2, ...]
              or [nDims, 1], if size of myHypArr [1, 1], then xArr
              can be any size [nDims, nVecArrDim1, nVecArrDim2, ...],
              in this case output variable will has
              size [1, nVecArrDim1, nVecArrDim2, ...]. If size of
              xArr is [nDims, nCols], then size of myHypArr may be
              [nCols, 1] or [1, nCols] or [1, 1], output variable
              will has size respectively
              [nCols, 1] or [1, nCols] or [nCols, 1].

Output:
  isPosArr: logical[hpDim1, hpDim2,...] /
      / logical[1, nVecArrDim1, nVecArrDim2, ...],
      isPosArr(iDim1, iDim2, ...) = true - myHypArr(iDim1, iDim2, ...)
      contains xArr(:, iDim1, iDim2, ...), false - otherwise.

Example:
  hypObj = hyperplane([-1; 1]);
  tempMat = [100 -1 2; 100 1 2];
  hypObj.contains(tempMat)

  ans =

       1
       0
       1
```

```
Hyperplane object of the Ellipsoidal Toolbox.


Functions:
----------
 hyperplane - Constructor of hyperplane object.
 double     - Returns parameters of hyperplane, i.e. normal vector and
              shift.
 parameters - Same function as 'double' (legacy matter).
 dimension  - Returns dimension of hyperplane.
 isempty    - Checks if hyperplane is empty.
 isparallel - Checks if one hyperplane is parallel to the other one.
 contains   - Check if hyperplane contains given point.


Overloaded operators and functions:
-----------------------------------
 eq      - Checks if two hyperplanes are equal.
 ne      - The opposite of 'eq'.
 uminus  - Switches signs of normal and shift parameters to the opposite.
 display - Displays the details about given hyperplane object.
 plot    - Plots hyperplane in 2D and 3D.
```

```
DIMENSION - returns dimensions of hyperplanes in the array.

  dimsArr = DIMENSION(hypArr) - returns dimensions of hyperplanes
      described by hyperplane structures in the array hypArr.

Input:
  regular:
      hypArr: hyperplane [nDims1, nDims2, ...] - array
          of hyperplanes.

Output:
      dimsArr: double[nDims1, nDims2, ...] - dimensions
          of hyperplanes.

Example:
  firstHypObj = hyperplane([-1; 1]);
  secHypObj = hyperplane([-1; 1; 8; -2; 3], 7);
  thirdHypObj = hyperplane([1; 2; 0], -1);
  hypVec = [firstHypObj secHypObj thirdHypObj];
  dimsVec  = hypVec.dimension()

  dimsVec =

     2     5     3
```

```
DISPLAY - Displays hyperplane object.

Input:
  regular:
      myHypArr: hyperplane [hpDim1, hpDim2, ...] - array
          of hyperplanes.

Example:
  hypObj = hyperplane([-1; 1]);
  display(hypObj)
```

```
  hypObj =
  size: [1 1]

  Element: [1 1]
  Normal:
      -1
       1

  Shift:
       0

  Hyperplane in R^2.

DOUBLE - return parameters of hyperplane - normal vector and shift.

  [normVec, hypScal] = DOUBLE(myHyp) - returns normal vector
      and scalar value of the hyperplane.

Input:
  regular:
      myHyp: hyperplane [1, 1] - single hyperplane of dimention nDims.

Output:
  normVec: double[nDims, 1] - normal vector of the hyperplane myHyp.
  hypScal: double[1, 1] - scalar of the hyperplane myHyp.

Example:
  hypObj = hyperplane([-1; 1]);
  [normVec, hypScal] = double(hypObj)

  normVec =

      -1
       1

  hypScal =

       0

FROMREPMAT - returns array of equal hyperplanes the same
             size as stated in sizeVec argument

  hpArr = fromRepMat(sizeVec) - creates an array  size
           sizeVec of empty hyperplanes.

  hpArr = fromRepMat(normalVec,sizeVec) - creates an array
           size sizeVec of hyperplanes with normal
           normalVec.

  hpArr = fromRepMat(normalVec,shift,sizeVec) - creates an
           array size sizeVec of hyperplanes with normal normalVec
           and hyperplane shift shift.

Input:
  Case1:
      regular:
          sizeVec: double[1,n] - vector of size, have
          integer values.
```

```
  Case2:
      regular:
          normalVec: double[nDim, 1] - normal of
          hyperplanes.
          sizeVec: double[1, n] - vector of size, have
          integer values.

  Case3:
      regular:
          normalVec: double[nDim, 1] - normal of
          hyperplanes.
          shift: double[1, 1] - shift of hyperplane.
          sizeVec: double[1,n] - vector of size, have
          integer values.

  properties:
      absTol: double [1,1] - absolute tolerance with default
          value 10^(-7)

fromStruct -- converts structural array into hyperplanes array.

Input:
  regular:
  SHpArr: struct [hpDim1, hpDim2, ...] -  structural array with following fields:

       normal: double[nHpDim, 1] - the normal of hyperplane
       shift: double[1, 1] - the shift of hyperplane

Output:
  hpArr : hyperplane [nDim1, nDim2, ...] - hyperplane array with size of
      SHpArr.


Example:
  hpObj = hyperplane([1 1]', 1);
  hpObj.toStruct()

  ans =

  normal: [2x1 double]
  shift: 0.7071

GETABSTOL - gives the array of absTol for all elements in hplaneArr

Input:
  regular:
      ellArr: hyperplane[nDim1, nDim2, ...] - multidimension array
          of hyperplane
  optional
      fAbsTolFun: function_handle[1,1] - function that apply
          to the absTolArr. The default is @min.

Output:
  regular:
      absTolArr: double [absTol1, absTol2, ...] - return absTol for
          each element in hplaneArr
  optional:
      absTol: double[1, 1] - return result of work fAbsTolFun with
```

```
          the absTolArr

Usage:
  use [~,absTol] = hplaneArr.getAbsTol() if you want get only
      absTol,
  use [absTolArr,absTol] = hplaneArr.getAbsTol() if you want get
      absTolArr and absTol,
  use absTolArr = hplaneArr.getAbsTol() if you want get only absTolArr

Example:
  firstHypObj = hyperplane([-1; 1]);
  secHypObj = hyperplane([-2; 5]);
  hypVec = [firstHypObj secHypObj];
  hypVec.getAbsTol()

  ans =

     1.0e-07 *

      1.0000    1.0000

GETCOPY - gives array the same size as hpArr with copies of elements of
          hpArr.

Input:
  regular:
      hpArr: hyperplane[nDim1, nDim2,...] - multidimensional array of
          hyperplanes.

Output:
  copyHpArr: hyperplane[nDim1, nDim2,...] - multidimension array of
      copies of elements of hpArr.

Example:
  firstHpObj = hyperplane([-1; 1], [2 0; 0 3]);
  secHpObj = hyperplane([1; 2], eye(2));
  hpVec = [firstHpObj secHpObj];
  copyHpVec = getCopy(hpVec)

  copyHpVec =
  1x2 array of hyperplanes.

GETPROPERTY - gives array the same size as hpArr with values of
              propName properties for each hyperplane in hpArr.
              Private method, used in every public property getter.

Input:
  regular:
      hpArr: hyperplane[nDim1, nDim2,...] - mltidimensional array
          of hyperplanes
      propName: char[1,N] - name property
  optional:
      fPropFun: function_handle[1,1] - function that apply
          to the propArr. The default is @min.

Output:
  regular:
      propArr: double[nDim1, nDim2,...] - multidimension array of
```

```
          propName properties for hyperplanes in rsArr
  optional:
      propVal: double[1, 1] - return result of work fPropFun with
          the propArr

GETRELTOL - gives the array of relTol for all elements in hpArr

Input:
  regular:
      hpArr: hyperplane[nDim1, nDim2, ...] - multidimension array
          of hyperplanes
  optional:
      fRelTolFun: function_handle[1,1] - function that apply
          to the relTolArr. The default is @min.
Output:
  regular:
      relTolArr: double [relTol1, relTol2, ...] - return relTol for
          each element in hpArr
  optional:
      relTol: double[1,1] - return result of work fRelTolFun with
          the relTolArr

Usage:
  use [~,relTol] = hpArr.getRelTol() if you want get only
      relTol,
  use [relTolArr,relTol] = hpArr.getRelTol() if you want get
      relTolArr and relTol,
  use relTolArr = hpArr.getRelTol() if you want get only relTolArr

Example:
  firsthpObj = hyperplane([-1; 1], 1);
  sechpObj = hyperplane([1 ;2], 2);
  hpVec = [firsthpObj sechpObj];
  hpVec.getRelTol()

  ans =

     1.0e-05 *

      1.0000    1.0000

HYPERPLANE - creates hyperplane structure
             (or array of hyperplane structures).

  Hyperplane H = { x in R^n : <v, x> = c },
  with current "Properties"..
  Here v must be vector in R^n, and c - scalar.

  hypH = HYPERPLANE - create empty hyperplane.

  hypH = HYPERPLANE(hypNormVec) - create
      hyperplane object hypH with properties:
          hypH.normal = hypNormVec,
          hypH.shift = 0.

  hypH = HYPERPLANE(hypNormVec, hypConst) - create
      hyperplane object hypH with properties:
          hypH.normal = hypNormVec,
```

```
          hypH.shift = hypConst.

  hypH = HYPERPLANE(hypNormVec, hypConst, ...
      'absTol', absTolVal) - create
      hyperplane object hypH with properties:
          hypH.normal = hypNormVec,
          hypH.shift = hypConst.
          hypH.absTol = absTolVal

  hypObjArr = HYPERPLANE(hypNormArr, hypConstArr) - create
      array of hyperplanes object just as
      hyperplane(hypNormVec, hypConst).

  hypObjArr = HYPERPLANE(hypNormArr, hypConstArr, ...
      'absTol', absTolValArr) - create
      array of hyperplanes object just as
      hyperplane(hypNormVec, hypConst, 'absTol', absTolVal).

Input:
  Case1:
    regular:
      hypNormArr: double[hpDims, nDims1, nDims2,...] -
          array of vectors in R^hpDims. There hpDims -
          hyperplane dimension.

  Case2:
    regular:
      hypNormArr: double[hpDims, nCols] /
          / [hpDims, nDims1, nDims2,...] /
          / [hpDims, 1] - array of vectors
          in R^hpDims. There hpDims - hyperplane dimension.
      hypConstArr: double[1, nCols] / [nCols, 1] /
          / [nDims1, nDims2,...] /
          / [nVecArrDim1, nVecArrDim2,...] -
          array of scalar.

  Case3:
    regular:
      hypNormArr: double[hpDims, nCols] /
          / [hpDims, nDims1, nDims2,...] /
          / [hpDims, 1] - array of vectors
          in R^hpDims. There hpDims - hyperplane dimension.
      hypConstArr: double[1, nCols] / [nCols, 1] /
          / [nDims1, nDims2,...] /
          / [nVecArrDim1, nVecArrDim2,...] -
          array of scalar.
      absTolValArr: double[1, 1] - value of
          absTol propeties.

    properties:
      propMode: char[1,] - property mode, the following
          modes are supported:
          'absTol' - name of absTol properties.

          note: if size of hypNormArr is
              [hpDims, nDims1, nDims2,...], then size of
              hypConstArr is [nDims1, nDims2, ...] or
              [1, 1], if size of hypNormArr [hpDims, 1],
```

```
              then hypConstArr can be any size
              [nVecArrDim1, nVecArrDim2, ...],
              in this case output variable will has
              size [nVecArrDim1, nVecArrDim2, ...].
              If size of hypNormArr is [hpDims, nCols],
              then size of hypConstArr may be
              [1, nCols] or [nCols, 1],
              output variable will has size
              respectively [1, nCols] or [nCols, 1].

Output:
  hypObjArr: hyperplane [nDims1, nDims2...] /
      / hyperplane [nVecArrDim1, nVecArrDim2, ...] -
      array of hyperplane structure hypH:
          hypH.normal - vector in R^hpDims,
          hypH.shift  - scalar.

Example:
  hypNormMat = [1 1 1; 1 1 1];
  hypConstVec = [1 -5 0];
  hypObj = hyperplane(hypNormMat, hypConstVec);

ISEMPTY - checks if hyperplanes in H are empty.

Input:
  regular:
      myHypArr: hyperplane [nDims1, nDims2, ...] - array
          of hyperplanes.

Output:
  isPositiveArr: logical[nDims1, nDims2, ...],
      isPositiveArr(iDim1, iDim2, ...) = true - if ellipsoid
      myHypArr(iDim1, iDim2, ...) is empty, false - otherwise.

Example:
  hypObj = hyperplane();
  isempty(hypObj)

  ans =

       1

ISEQUAL - produces logical array the same size as
          ellFirstArr/ellFirstArr (if they have the same).
          isEqualArr[iDim1, iDim2,...] is true if corresponding
          ellipsoids are equal and false otherwise.

Input:
  regular:
      ellFirstArr: ellipsoid[nDim1, nDim2,...] - multidimensional array
          of ellipsoids.
      ellSecArr: ellipsoid[nDim1, nDim2,...] - multidimensional array
          of ellipsoids.
  properties:
      'isPropIncluded': makes to compare second value properties, such as
      absTol etc.
Output:
  isEqualArr: logical[nDim1, nDim2,...] - multidimension array of
```

```
      logical values. isEqualArr[iDim1, iDim2,...] is true if
      corresponding ellipsoids are equal and false otherwise.

  reportStr: char[1,] - comparison report.

ISPARALLEL - check if two hyperplanes are parallel.

  isResArr = ISPARALLEL(fstHypArr, secHypArr) - Checks if hyperplanes
      in fstHypArr are parallel to hyperplanes in secHypArr and
      returns array of true and false of the size corresponding
      to the sizes of fstHypArr and secHypArr.

Input:
  regular:
      fstHypArr: hyperplane [nDims1, nDims2, ...] - first array
          of hyperplanes
      secHypArr: hyperplane [nDims1, nDims2, ...] - second array
          of hyperplanes

Output:
  isPosArr: logical[nDims1, nDims2, ...] -
      isPosArr(iFstDim, iSecDim, ...) = true -
      if fstHypArr(iFstDim, iSecDim, ...) is parallel
      secHypArr(iFstDim, iSecDim, ...), false - otherwise.

Example:
  hypObj = hyperplane([-1 1 1; 1 1 1; 1 1 1], [2 1 0]);
  hypObj.isparallel(hypObj(2))

  ans =

       0     1     1

PARAMETERS - return parameters of hyperplane - normal vector and shift.

  [normVec, hypScal] = PARAMETERS(myHyp) - returns normal vector
      and scalar value of the hyperplane.

Input:
  regular:
      myHyp: hyperplane [1, 1] - single hyperplane of dimention nDims.

Output:
  normVec: double[nDims, 1] - normal vector of the hyperplane myHyp.
  hypScal: double[1, 1] - scalar of the hyperplane myHyp.

Example:
  hypObj = hyperplane([-1; 1]);
  [normVec, hypScal] = parameters(hypObj)

  normVec =

      -1
       1


  hypScal =
```

```
       0

PLOT - plots hyperplaces in 2D or 3D.


Usage:
      plot(hyp) - plots hyperplace hyp in default (red) color.
      plot(hypArr) - plots an array of hyperplaces.
      plot(hypArr, 'Property',PropValue,...) - plots hypArr with setting
                                               properties.

Input:
  regular:
      hypArr:  Hyperplace: [dim11Size,dim12Size,...,dim1kSize] -
               array of 2D or 3D hyperplace objects. All hyperplaces in hypArr
               must be either 2D or 3D simutaneously.
  optional:
      color1Spec: char[1,1] - color specification code, can be 'r','g',
                              etc (any code supported by built-in Matlab function).
      hyp2Arr: Hyperplane: [dim21Size,dim22Size,...,dim2kSize] -
                                          second Hyperplane array...
      color2Spec: char[1,1] - same as color1Spec but for hyp2Arr
      ....
      hypNArr: Hyperplane: [dimN1Size,dim22Size,...,dimNkSize] -
                                           N-th Hyperplane array
      colorNSpec - same as color1Spec but for hypNArr.
  properties:
      'newFigure': logical[1,1] - if 1, each plot command will open a new figure window.
                   Default value is 0.
      'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
              if 1, ellipsoids in 2D will be filled with color. Default value is 0.
      'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
                   line width for 1D and 2D plots. Default value is 1.
      'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
               sets default colors in the form [x y z]. Default value is [1 0 0].
      'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
               level of transparency between 0 and 1 (0 - transparent, 1 - opaque).
               Default value is 0.4.
      'size': double[1,1] - length of the line segment in 2D, or square diagonal in 3D.
      'center': double[1,dimHyp] - center of the line segment in 2D, of the square in 3D
      'relDataPlotter' - relation data plotter object.
      Notice that property vector could have different dimensions, only
      total number of elements must be the same.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

toStruct -- converts hyperplanes array into structural array.

Input:
  regular:
      hpArr: hyperplane [hpDim1, hpDim2, ...] - array
          of hyperplanes.

Output:
  ShpArr : struct[nDim1, nDim2, ...] - structural array with size of
      hpArr with the following fields:
```

```
      normal: double[nHpDim, 1] - the normal of hyperplane
      shift: double[1, 1] - the shift of hyperplane

UMINUS - switch signs of normal vector and the shift scalar
         to the opposite.

Input:
  regular:
      inpHypArr: hyperplane [nDims1, nDims2, ...] - array
          of hyperplanes.

Output:
  outHypArr: hyperplane [nDims1, nDims2, ...] - array
      of the same hyperplanes as in inpHypArr whose
      normals and scalars are multiplied by -1.

Example:
  hypObj = -hyperplane([-1; 1], 1)

  hypObj =
  size: [1 1]

  Element: [1 1]
  Normal:
       1
      -1

  Shift:
      -1

  Hyperplane in R^2.
```

9.3 elltool.conf.Properties

```
PROPERTIES - a static class, providing emulation of static properties for
             toolbox.

Example:
  elltool.conf.Properties.checkSettings()

Example:
  elltool.conf.Properties.getAbsTol();

Example:
  elltool.conf.Properties.getConfRepoMgr()

  ans =

    elltool.conf.ConfRepoMgr handle
    Package: elltool.conf

    Properties:
      DEFAULT_STORAGE_BRANCH_KEY: '_default'
```

```
Example:
  elltool.conf.Properties.getIsEnabledOdeSolverOptions();

Example:
  elltool.conf.Properties.getIsODENormControl();

Example:
  elltool.conf.Properties.getIsVerbose();

Example:
  elltool.conf.Properties.getNPlot2dPoints();

Example:
  elltool.conf.Properties.getNPlot3dPoints();

Example:
  elltool.conf.Properties.getNTimeGridPoints();

Example:
  elltool.conf.Properties.getODESolverName();

Example:
  elltool.conf.Properties.getConfRepoMgr.getCurConf()

  ans =

                    version: '1.4dev'
                  isVerbose: 0
                     absTol: 1.0000e-07
                     relTol: 1.0000e-05
            nTimeGridPoints: 200
              ODESolverName: 'ode45'
           isODENormControl: 'on'
  isEnabledOdeSolverOptions: 0
              nPlot2dPoints: 200
              nPlot3dPoints: 200
                    logging: [1x1 struct]

::

Example:
  elltool.conf.Properties.getVersion();

Example:
  elltool.conf.Properties.init()

PARSEPROP - parses input into cell array with values of properties listed
           in neededPropNameList.
           Values are  taken from args or, if there no value for some
           property in args, in current Properties.


Input:
  regular:
      args: cell[1,] of any[] - cell array of arguments that
          should be parsed.
  optional
      neededPropNameList: cell[1,nProp] of char[1,] - cell array of strings
```

```
          containing names of parameters, that output should consist of.
          The following properties are supported:
              version
              isVerbose
              absTol
              relTol
              regTol
              ODESolverName
              isODENormControl
              isEnabledOdeSolverOptions
              nPlot2dPoints
              nPlot3dPoints
              nTimeGridPoints
          trying to specify other properties would be result in error
          If neededPropNameList is not specified, the list of all
          supported properties is assumed.

Output:
  propVal1:  - value of the first property specified
                             in neededPropNameList in the same order as
                             they listed in neededPropNameList
      ....
  propValN:  - value of the last property from neededPropNameList
  restList: cell[1,nRest] - list of the input arguments that were not
      recognized as properties

Example:
    testAbsTol = 1;
    testRelTol = 2;
    nPlot2dPoints = 3;
    someArg = 4;
    args = {'absTol',testAbsTol, 'relTol',testRelTol,'nPlot2dPoints',...
        nPlot2dPoints, 'someOtherArg', someArg};
    neededPropList = {'absTol','relTol'};
    [absTol, relTol,resList]=elltool.conf.Properties.parseProp(args,...
        neededPropList)

    absTol =

         1


    relTol =

         2


    resList =

        'nPlot2dPoints'    [3]    'someOtherArg'    [4]

Example:
  prevConfRepo = Properties.getConfRepoMgr();
  prevAbsTol = prevConfRepo.getParam('absTol');
  elltool.conf.Properties.setConfRepoMgr(prevConfRepo);

Example:
  elltool.conf.Properties.setIsVerbose(true);
```

```
Example:
  elltool.conf.Properties.setNPlot2dPoints(300);

Example:
  elltool.conf.Properties.setNTimeGridPoints(300);

SETRELTOL - set global relative tolerance

Input
relTol: double[1,1]
```

9.4 elltool.core.GenEllipsoid

```
GENELLIPSOID - class of generalized ellipsoids

Input:
  Case1:
    regular:
      qVec: double[nDim,1] - ellipsoid center
      qMat: double[nDim,nDim] / qVec: double[nDim,1] - ellipsoid matrix
          or diagonal vector of eigenvalues, that may contain infinite
          or zero elements

  Case2:
    regular:
      qMat: double[nDim,nDim] / qVec: double[nDim,1] - diagonal matrix or
          vector, may contain infinite or zero elements

  Case3:
    regular:
      qVec: double[nDim,1] - ellipsoid center
      dMat: double[nDim,nDim] / dVec: double[nDim,1] - diagonal matrix or
          vector, may contain infinite or zero elements
      wMat: double[nDim,nDim] - any square matrix


Output:
  self: GenEllipsoid[1,1] - created generalized ellipsoid

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2));
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);

Example:
  firstEllObj = elltool.core.GenEllipsoid([1; 1], eye(2));
  secEllObj = elltool.core.GenEllipsoid([0; 5], 2*eye(2));
  ellVec = [firstEllObj secEllObj];
  ellVec.dimension()

  ans =

       2     2

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);
  ellObj.display()
```

```
     |
     |----- q : [5 2]
     |          -------
     |----- Q : |10|19|
     |          |19|41|
     |          -------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);
  ellObj.getCenter()

  ans =

       5
       2

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);
  ellObj.getCheckTol()

  ans =

     1.0000e-09

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);
  ellObj.getDiagMat()

  ans =

      0.9796         0
           0   50.0204

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], eye(2), [1 3; 4 5]);
  ellObj.getEigvMat()

  ans =

      0.9034   -0.4289
     -0.4289   -0.9034

Example:
  firstEllObj = elltool.core.GenEllipsoid([10;0], 2*eye(2));
  secEllObj = elltool.core.GenEllipsoid([0;0], [1 0; 0 0.1]);
  curDirMat = [1; 0];
  isOk=getIsGoodDir(firstEllObj,secEllObj,dirsMat)

  isOk =

       1

INV - create generalized ellipsoid whose matrix in pseudoinverse
      to the matrix of input generalized ellipsoid
```

```
Input:
  regular:
      ellObj: GenEllipsoid: [1,1] - generalized ellipsoid

Output:
  ellInvObj: GenEllipsoid: [1,1] - inverse generalized ellipsoid

Example:
  ellObj = elltool.core.GenEllipsoid([5;2], [1 0; 0 0.7]);
  ellObj.inv()
     |
     |----- q : [5 2]
     |          -----------------
     |----- Q : |1       |0       |
     |          |0       |1.42857|
     |          -----------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----

MINKDIFFEA - computes tight external ellipsoidal approximation for
             Minkowsky difference of two generalized ellipsoids

Input:
  regular:
      ellObj1: GenEllipsoid: [1,1] - first generalized ellipsoid
      ellObj2: GenEllipsoid: [1,1] - second generalized ellipsoid
      dirMat: double[nDim,nDir] - matrix whose columns specify
          directions for which approximations should be computed
Output:
  resEllVec: GenEllipsoid[1,nDir] - vector of generalized ellipsoids of
      external approximation of the dirrence of first and second
      generalized ellipsoids (may contain empty ellipsoids if in specified
      directions approximation cannot be computed)

Example:
  firstEllObj = elltool.core.GenEllipsoid([10;0], 2*eye(2));
  secEllObj = elltool.core.GenEllipsoid([0;0], [1 0; 0 0.1]);
  dirsMat = [1,0].';
  resEllVec  = minkDiffEa( firstEllObj, secEllObj, dirsMat)
     |
     |----- q : [10 0]
     |          -------------------
     |----- Q : |0.171573|0        |
     |          |0       |1.20557 |
     |          -------------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----

MINKDIFFIA - computes tight internal ellipsoidal approximation for
             Minkowsky difference of two generalized ellipsoids

Input:
  regular:
```

```
      ellObj1: GenEllipsoid: [1,1] - first generalized ellipsoid
      ellObj2: GenEllipsoid: [1,1] - second generalized ellipsoid
      dirMat: double[nDim,nDir] - matrix whose columns specify
          directions for which approximations should be computed
Output:
  resEllVec: GenEllipsoid[1,nDir] - vector of generalized ellipsoids of
      internal approximation of the dirrence of first and second
      generalized ellipsoids

Example:
  firstEllObj = elltool.core.GenEllipsoid([10;0], 2*eye(2));
  secEllObj = elltool.core.GenEllipsoid([0;0], [1 0; 0 0.1]);
  dirsMat = [1,0].';
  resEllVec  = minkDiffIa( firstEllObj, secEllObj, dirsMat)
     |
     |----- q : [10 0]
     |          -------------------
     |----- Q : |0.171573|0       |
     |          |0       |0.544365|
     |          -------------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----

MINKSUMEA - computes tight external ellipsoidal approximation for
            Minkowsky sum of the set of generalized ellipsoids

Input:
  regular:
      ellObjVec: GenEllipsoid: [kSize,mSize] - vector of  generalized
                                          ellipsoid
      dirMat: double[nDim,nDir] - matrix whose columns specify
          directions for which approximations should be computed
Output:
  ellResVec: GenEllipsoid[1,nDir] - vector of generalized ellipsoids of
      external approximation of the dirrence of first and second
      generalized ellipsoids

Example:
  firstEllObj = elltool.core.GenEllipsoid([1;1],eye(2));
  secEllObj = elltool.core.GenEllipsoid([5;0],[3 0; 0 2]);
  ellVec = [firstEllObj secEllObj];
  dirsMat = [1 3; 2 4];
  ellResVec  = minkSumEa(ellVec, dirsMat )

  Structure(1)
     |
     |----- q : [6 1]
     |          -----------------
     |----- Q : |7.50584|0      |
     |          |0      |5.83164|
     |          -----------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----
     O
```

```
  Structure(2)
     |
     |----- q : [6 1]
     |          -----------------
     |----- Q : |7.48906|0      |
     |          |0      |5.83812|
     |          -----------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----
     O

MINKSUMIA - computes tight internal ellipsoidal approximation for
            Minkowsky sum of the set of generalized ellipsoids

Input:
  regular:
      ellObjVec: GenEllipsoid: [kSize,mSize] - vector of  generalized
                                          ellipsoid
      dirMat: double[nDim,nDir] - matrix whose columns specify
          directions for which approximations should be computed
Output:
  ellResVec: GenEllipsoid[1,nDir] - vector of generalized ellipsoids of
      internal approximation of the dirrence of first and second
      generalized ellipsoids

Example:
  firstEllObj = elltool.core.GenEllipsoid([1;1],eye(2));
  secEllObj = elltool.core.GenEllipsoid([5;0],[3 0; 0 2]);
  ellVec = [firstEllObj secEllObj];
  dirsMat = [1 3; 2 4];
  ellResVec  = minkSumIa(ellVec, dirsMat )

  Structure(1)
     |
     |----- q : [6 1]
     |          ---------------------
     |----- Q : |7.45135  |0.0272432|
     |          |0.0272432|5.81802  |
     |          ---------------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----
     O

  Structure(2)
     |
     |----- q : [6 1]
     |          ---------------------
     |----- Q : |7.44698  |0.0315642|
     |          |0.0315642|5.81445  |
     |          ---------------------
     |          -----
     |-- QInf : |0|0|
     |          |0|0|
     |          -----
```

```
     O

PLOT - plots ellipsoids in 2D or 3D.


Usage:
      plot(ell) - plots generic ellipsoid ell in default (red) color.
      plot(ellArr) - plots an array of generic ellipsoids.
      plot(ellArr, 'Property',PropValue,...) - plots ellArr with setting
                                              properties.

Input:
  regular:
      ellArr:  elltool.core.GenEllipsoid: [dim11Size,dim12Size,...,
               dim1kSize] - array of 2D or 3D GenEllipsoids objects.
               All ellipsoids in ellArr  must be either 2D or 3D
               simutaneously.
  optional:
      color1Spec: char[1,1] - color specification code, can be 'r','g',
                              etc (any code supported by built-in Matlab
                              function).
      ell2Arr: elltool.core.GenEllipsoid: [dim21Size,dim22Size,...,
                              dim2kSize] - second ellipsoid array...
      color2Spec: char[1,1] - same as color1Spec but for ell2Arr
      ....
      ellNArr: elltool.core.GenEllipsoid: [dimN1Size,dim22Size,...,
                               dimNkSize] - N-th ellipsoid array
      colorNSpec - same as color1Spec but for ellNArr.
  properties:
      'newFigure': logical[1,1] - if 1, each plot command will open a new .
                   figure window Default value is 0.
      'fill': logical[1,1]/logical[dim11Size,dim12Size,...,dim1kSize]  -
              if 1, ellipsoids in 2D will be filled with color.
              Default value is 0.
      'lineWidth': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
               line width for 1D and 2D plots.
               Default value is 1.
      'color': double[1,3]/double[dim11Size,dim12Size,...,dim1kSize,3] -
               sets default colors in the form [x y z].
               Default value is [1 0 0].
      'shade': double[1,1]/double[dim11Size,dim12Size,...,dim1kSize]  -
               level of transparency between 0 and 1 (0 - transparent,
               1 - opaque).
               Default value is 0.4.
      'relDataPlotter' - relation data plotter object.
      Notice that property vector could have different dimensions, only
      total number of elements must be the same.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

Examples:
  plot([ell1, ell2, ell3], 'color', [1, 0, 1; 0, 0, 1; 1, 0, 0]);
  plot([ell1, ell2, ell3], 'color', [1; 0; 1; 0; 0; 1; 1; 0; 0]);
  plot([ell1, ell2, ell3; ell1, ell2, ell3], 'shade', [1, 1, 1; 1, 1,
    1]);
  plot([ell1, ell2, ell3; ell1, ell2, ell3], 'shade', [1; 1; 1; 1; 1;
```

```
      1]);
  plot([ell1, ell2, ell3], 'shade', 0.5);
  plot([ell1, ell2, ell3], 'lineWidth', 1.5);
  plot([ell1, ell2, ell3], 'lineWidth', [1.5, 0.5, 3]);

Example:
  ellObj = elltool.core.GenEllipsoid([1;1],eye(2));
  dirsVec = [1; 0];
  [resRho, bndPVec] = rho(ellObj, dirsVec)

  resRho =

       2

 bndPVec =

       2
       1
```

9.5 smartdb.relations.ATypifiedStaticRelation

```
ATYPIFIEDSTATICRELATION is a constructor of static relation class
object

Usage: self=AStaticRelation(obj) or
       self=AStaticRelation(varargin)

Input:
  optional
    inpObj: ARelation[1,1]/SData: struct[1,1]
        structure with values of all fields
        for all tuples

    SIsNull: struct [1,1] - structure of fields with is-null
       information for the field content, it can be logical for
       plain real numbers of cell of logicals for cell strs or
       cell of cell of str for more complex types

    SIsValueNull: struct [1,1] - structure with logicals
        determining whether value corresponding to each field
        and each tuple is null or not

  properties:
      fillMissingFieldsWithNulls: logical[1,1] - if true,
          the relation fields absent in the input data
          structures are filled with null values

Output:
  regular:
    self: ATYPIFIEDSTATICRELATION [1,1] - constructed class object

Note: In the case the first interface is used, SData and
      SIsNull are taken from class object obj

ADDDATA - adds a set of field values to existing data in a form of new
          tuples
```

```
Input:
  regular:
     self:ARelation [1,1] - class object

ADDDATAALONGDIM - adds a set of field values to existing data using
                  a concatenation along a specified dimension

Input:
  regular:
      self: CubeStruct [1,1] - the object

ADDTUPLES - adds a set of new tuples to the relation

Usage: addTuplesInternal(self,varargin)

input:
  regular:
      self: ARelation [1,1] - class object
      SData: struct [1,1] - structure with values of all fields  for all
       tuples
  optional:
      SIsNull: struct [1,1] - structure of fields with is-null
        information for the field content, it can be logical for plain
        real numbers of cell of logicals for cell strs or cell of cell of
        str for more complex types

      SIsValueNull: struct [1,1] - structure with logicals determining
        whether value corresponding to each field and each tuple is null
        or not

  properties:
      checkConsistency: logical[1,1], if true, a consistency between the
         input structures is not checked, true by default

APPLYGETFUNC - applies a function to the specified fields as columns, i.e.
               the function is applied to each field as whole, not to
               each cell separately

Input:
  regular:
      hFunc: function_handle[1,1] - function to apply to each of the
         field values
  optional:
      toFieldNameList: char/cell[1,] of char - a list of fields to which
         the function specified by hFunc is to be applied

    Note: hFunc can optionally be specified after toFieldNameList
          parameter

Notes: this function currently has a lots of limitations:
  1) it assumes that the output is uniform
  2) the function is applies to SData part of field value
  3) no additional arguments can be passed
  All this limitations will eventually go away though so stay tuned...

APPLYSETFUNC - applies some function to each cell of the specified fields
               of a given CubeStruct object
```

```
Usage: applySetFunc(self,toFieldNameList,hFunc)
       applySetFunc(self,hFunc,toFieldNameList)

Input:
  regular:
      self: CubeStruct [1,1] - class object

      hFunc: function handle [1,1] - handle of function to be
        applied to fields, the function is assumed to
          1) have the same number of input/output arguments
          2) the number of input arguments should be
             length(structNameList)*length(fieldNameList)
          3) the input arguments should be ordered according to the
          following rule
              (x_struct_1_field_1,x_struct_1_field_2,...,struct_n_field1,
              ...,struct_n_field_m)

  optional:

      toFieldNameList: char or char cell [1,nFields] - list of
        field names to which given function should be applied

        Note1: field lists of length>1 are not currently supported !
        Note2: it is possible to specify toFieldNameList before hFunc in
           which case the parameters will be recognized automatically

  properties:
      uniformOutput: logical[1,1] - specifies if the result
         of the function is uniform to be stored in non-cell
         field, by default it is false for cell fileds and
         true for non-cell fields

      structNameList: char[1,]/cell[1,], name of data structure/list of
        data structure names to which the function is to
             be applied, can be composed from the following values

           SData - data itself

           SIsNull - contains is-null indicator information for data
             values

           SIsValueNull - contains is-null indicators for CubeStruct
              cells (not for cell values)

        structNameList={'SData'} by default

      inferIsNull: logical[1,2] - if the first(second) element is true,
          SIsNull(SIsValueNull) indicators are inferred from SData,
          i.e. with this indicator set to true it is sufficient to apply
          the function only to SData while the rest of the structures
          will be adjusted automatically.

      inputType: char[1,] - specifies a way in which the field value is
         partitioned into individual cells before being passed as an
         input parameter to hFunc. This parameter directly corresponds to
         outputType parameter of toArray method, see its documentation
         for a list of supported input types.
```

```
APPLYTUPLEGETFUNC - applies a function to the specified fields
                    separately to each tuple

Input:
  regular:
      hFunc: function_handle[1,1] - function to apply to the specified
         fields
  optional:
      toFieldNameList: char/cell[1,] of char - a list of fields to which
         the function specified by hFunc is to be applied

  properties:
      uniformOutput: logical[1,1] - if true, output is expected to be
          uniform as in cellfun with 'UniformOutput'=true, default
           value is true

Output:
  funcOut1Arr: <type1>[] - array corresponding to the first output of the
      applied function
          ....
  funcOutNArr: <typeN>[] - array corresponding to the last output of the
      applied function


Notes: this function currently has a lots of limitations:
  1) the function is applies to SData part of field value
  2) no additional arguments can be passed
  All this limitations will eventually go away though so stay tuned...

CLEARDATA - deletes all the data from the object

Usage: self.clearData(self)

Input:
  regular:
    self: CubeStruct [1,1] - class object

CLONE - creates a copy of a specified object via calling
        a copy constructor for the object class

Input:
  regular:
    self: any [] - current object
  optional
    any parameters applicable for relation constructor

Ouput:
  self: any [] - constructed object

COPYFROM - reconstruct CubeStruct object within a current object using the
           input CubeStruct object as a prototype

Input:
  regular:
    self: CubeStruct [n_1,...,n_k]
    obj: any [] - internal representation of the object

  optional:
```

```
    fieldNameList: cell[1,nFields] - list of fields to copy

CREATEINSTANCE - returns an object of the same class by calling a default
                 constructor (with no parameters)

Usage: resObj=getInstance(self)

input:
  regular:
    self: any [] - current object
  optional
    any parameters applicable for relation constructor

Ouput:
  self: any [] - constructed object

DISPONUI - displays a content of the given relation as a data grid UI
           component.

Input:
  regular:
      self:
  properties:
      tableType: char[1,] - type of table used for displaying the data,
          the following types are supported:
          'sciJavaGrid' - proprietary Java-based data grid component
              is used
          'uitable'  - Matlab built-in uitable component is used.
              if not specified, the method tries to use sciJavaGrid
              if it is available, if not - uitable is used.

Output:
  hFigure: double[1,1] - figure handle containing the component
  gridObj: smartdb.relations.disp.UIDataGrid[1,1] - data grid component
      instance used for displaying a content of the relation object

DISPLAY - puts some textual information about CubeStruct object in screen

Input:
 regular:
     self.

FROMSTRUCTLIST - creates a dynamic relation from a list of structures
                 interpreting each structure as the data for
                 several tuples.

Input:
  regular:
      className: name of object class which will be created,
          the class constructor should accept 2 properties:
          'fieldNameList' and 'fieldTypeSpecList'

      structList: cell[] of struct[1,1] - list of structures

Output:
  relDataObj: smartdb.relations.DynamicRelation[1,1] -
     constructed relation
```

```
GETCOPY - returns an object copy

Usage: resObj=getCopy(self)

Input:
  regular:
    self: CubeStruct [1,1] - current CubeStruct object
  optional:
    same as for getData

GETDATA - returns an indexed projection of CubeStruct object's content

Input:
  regular:
      self: CubeStruct [1,1] - the object

  optional:

      subIndCVec:
        Case#1: numeric[1,]/numeric[,1]

        Case#2: cell[1,nDims]/cell[nDims,1] of double [nSubElem_i,1]
              for i=1,...,nDims

          -array of indices of field value slices that are selected
          to be returned; if not given (default),
          no indexation is performed

        Note!: numeric components of subIndVec are allowed to contain
           zeros which are be treated as they were references to null
           data slices

      dimVec: numeric[1,nDims]/numeric[nDims,1] - vector of dimension
          numbers corresponding to subIndCVec

  properties:

      fieldNameList: char[1,]/cell[1,nFields] of char[1,]
          list of field names to return

      structNameList: char[1,]/cell[1,nStructs] of char[1,]
          list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}

      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the cells,
              default value is false

      nullReplacements: cell[1,nReplacedFields]  - list of null
          replacements for each of the fields

      nullReplacementFields: cell[1,nReplacedFields] - list of fields in
         which the nulls are to be replaced with the specified values,
         if not specified it is assumed that all fields are to be
         replaced

         NOTE!: all fields not listed in this parameter are replaced with
         the default values
```

```
      checkInputs: logical[1,1] - true by default (input arguments are
         checked for correctness

Output:
  regular:
    SData: struct [1,1] - structure containing values of
        fields at the selected slices, each field is an array
        containing values of the corresponding type

    SIsNull: struct [1,1] - structure containing a nested
        array with is-null indicators for each CubeStruct cell content

    SIsValueNull: struct [1,1] - structure containing a
       logical array [] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value

GETFIELDDESCRLIST - returns the list of CubeStruct field descriptions

Usage: value=getFieldDescrList(self)

Input:
  regular:
      self: CubeStruct [1,1]
  optional:
      fieldNameList: cell[1,nSpecFields] of char[1,] - field names for
         which descriptions should be returned

Output:
  regular:
    value: char cell [1,nFields] - list of CubeStruct object field
        descriptions

GETFIELDISNULL - returns for given field a nested logical/cell array
                 containing is-null indicators for cell content

Usage: fieldIsNullCVec=getFieldIsNull(self,fieldName)

Input:
  regular:
    self: CubeStruct [1,1]
    fieldName: char - field name
Output:
  regular:
    fieldIsCVec: logical/cell[] - nested cell/logical array containing
       is-null indicators for content of the field

GETFIELDISVALUENULL - returns for given field logical vector determining
                      whether value of this field in each cell is null
                      or not.

BEWARE OF confusing this with getFieldIsNull method which returns is-null
   indicators for a field content

Usage: isNullVec=getFieldValueIsNull(self,fieldName)

Input:
  regular:
```

```
    self: CubeStruct [1,1]
    fieldName: char - field name

Output:
  regular:
    isValueNullVec: logical[] - array of isValueNull indicators for the
       specified field

GETFIELDNAMELIST - returns the list of CubeStruct object field names

Usage: value=getFieldNameList(self)

Input:
  regular:
    self: CubeStruct [1,1]
Iutput:
  regular:
    value: char cell [1,nFields] - list of CubeStruct object field
        names

GETFIELDPROJECTION - project object with specified fields.

Input:
  regular:
      self: ARelation[1,1] - original object
      fieldNameList: cell[1,nFields] of char[1,] - field name list

Output:
  obj: DynamicRelation[1,1] - projected object

GETFIELDTYPELIST - returns list of field types in given CubeStruct object

Usage: fieldTypeList=getFieldTypeList(self)

Input:
  regular:
      self: CubeStruct [1,1]

  optional:
      fieldNameList: cell[1,nFields] - list of field names

Output:
 regular:
  fieldTypeList: cell [1,nFields] of smartdb.cubes.ACubeStructFieldType[1,1]
      - list of field types

GETFIELDTYPESPECLIST - returns a list of field type specifications. Field
                       type specification is a sequence of type names
                       corresponding to field value types starting with
                       the top level and going down into the nested
                       content of a field (for a field having a complex
                       type).

Input:
  regular:
      self:
  optional:
      fieldNameList: cell [1,nFields] of char[1,] - list of field names
```

```
  properties:
      uniformOutput: logical[1,1] - if true, the result is concatenated
         across all the specified fields

Output:
  typeSpecList:
       Case#1: uniformOutput=false
          cell[1,nFields] of cell[1,nNestedLevels_i] of char[1,.]
       Case#2: uniformOutput=true
          cell[1,nFields*prod(nNestedLevelsVec)] of char[1,.]
       - list of field type specifications

GETFIELDVALUESIZEMAT - returns a matrix composed from the size vectors
                       for the specified fields

Input:
  regular:
      self:

  optional:
      fieldNameList: cell[1,nFields] - a list of fileds for which the size
         matrix is to be generated

  properties:
      skipMinDimensions: logical[1,1] - if true, the dimensions from 1 up
          to minDimensionality are skipped

      minDimension: numeric[1,1] - minimum dimension which definies a
         minimum number of columns in the resulting matrix

Output:
  sizeMat: double[nFields,nMaxDims]

GETISFIELDVALUENULL - returns a vector indicating whether a particular
                      field is composed of null values completely

Usage: isValueNullVec=getIsFieldValueNull(self,fieldNameList)

Input:
  regular:
    self: CubeStruct [1,1]

  optional:
    fieldNameList: cell[1,nFields] of char[1,] - list of field names

Output:
  regular:
    isValueNullVec: logical[1,nFields]

GETJOINWITH - returns a result of INNER join of given relation with
              another relation by the specified key fields

LIMITATION: key fields by which the join is peformed are required to form
a unique key in the given relation

Input:
  regular:
      self:
```

```
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT)
          'leftOuter'

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result

GETMINDIMENSIONSIZE - returns a size vector for the specified
                      dimensions. If no dimensions are specified, a size
                      vector for all dimensions up to minimum CubeStruct
                      dimension is returned

Input:
  regular:
      self:
  optional:
      dimNumVec: numeric[1,nDims] - a vector of dimension
          numbers

Output:
  minDimensionSizeVec: double [1,nDims] - a size vector for
     the requested dimensions

GETMINDIMENSIONALITY - returns a minimum dimensionality for a given
                       object

Input:
  regular:
      self

Output:
  minDimensionality: double[1,1] - minimum dimensionality of
     self object

GETNELEMS - returns a number of elements in a given object
Input:
  regular:
     self:

Output:
  nElems:double[1, 1] - number of elements in a given object

GETNFIELDS - returns number of fields in given object

Usage: nFields=getNFields(self)

Input:
  regular:
    self: CubeStruct [1,1]
Output:
  regular:
    nFields: double [1,1] - number of fields in given object
```

```
GETNTUPLES - returns number of tuples in given relation

Usage: nTuples=getNTuples(self)

input:
  regular:
    self: ARelation [1,1] - class object
output:
  regular:
    nTuples: double [1,1] - number of tuples in given  relation

GETSORTINDEX - gets sort index for all tuples of given relation with
               respect to some of its fields

Usage: sortInd=getSortIndex(self,sortFieldNameList,varargin)

input:
  regular:
    self: ARelation [1,1] - class object
    sortFieldNameList: char or char cell [1,nFields] - list of field
       names with respect to which tuples are sorted

  properties:
    Direction: char or char cell [1,nFields] - direction of sorting for
        all fields (if one value is given) or for each field separately;
        each value may be 'asc' or 'desc'
output:
  regular:
   sortIndex: double [nTuples,1] - sort index for all tuples such that if
       fieldValueVec is a vector of values for some field of given
       relation, then fieldValueVec(sortIndex) is a vector of values for
       this field when tuples of the relation are sorted

GETTUPLES - selects tuples with given indices from given relation and
            returns the result as new relation

Usage: obj=getTuples(self,subIndVec)

input:
  regular:
    self: ARelation [1,1] - class object
    subIndVec: double [nSubTuples,1]/logical[nTuples,1] - array of
        indices for tuples that are selected
output:
  regular:
    obj: ARelation [1,1] - new class object containing only selected
        tuples

GETTUPLESFILTEREDBY - selects tuples from given relation such that a
                      fixed index field contains values from a given set
                      of value and returns the result as new relation

Input:
  regular:
    self: ARelation [1,1] - class object
    filterFieldName: char - name of index field
    filterValueVec: numeric/ cell of char [nValues,1] - vector of index
        values
```

```
  properties:
    keepNulls: logical[1,1] - if true, null values are not filteed out,
       and removed otherwise,
          default: false

Output:
  regular:
    obj: ARelation [1,1] - new class object containing only selected
        tuples
    isThereVec: logical[nTuples,1] - contains true for the kept tuples

 GETTUPLESINDEXEDBY - selects tuples from given relation such that fixed
                      index field contains given in a specified order
                      values and returns the result as new relation.
                      It is required that the original relation
                      contains only one record for each field value

 input:
   regular:
     self: ARelation [1,1] - class object
     indexFieldName: char - name of index field
     indexValueVec: numeric or char cell [nValues,1] - vector of index
         values
 output:
   regular:
     obj: ARelation [1,1] - new class object containing only selected
         tuples

TODO add type check

GETTUPLESJOINEDWITH - returns the tuples of the given relation
                      INNER-joined with other relation by the specified
                      key fields

Input:
  regular:
      self:
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT) - inner join
          'leftOuter' - left outer join
          'rightOuter' - right outer join
          'fullOuter' - full outer join

      fieldDescrSource: char[1,] - defines where the field descriptions
         are taken from, can be
          'useOriginal' - field descriptions are taken from the left hand
              side argument of the join operation
          'useOther' - field descriptions are taken from the right hand
              side of the join operation

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result
```

```
GETUNIQUEDATA - returns internal representation for a set of unique
                tuples for given relation

Usage: [SData,SIsNull,SIsValueNull]=getUniqueData(self,varargin)

Input:
  regular:
    self: ARelation [1,1] - class object
  properties
      fieldNameList: list of field names used for finding the unique
          elements; only the specified fields are returned in SData,
          SIsNull,SIsValueNull structures
      structNameList: list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}
      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the tuples
              default value is false

Output:
  regular:

    SData: struct [1,1] - structure containing values of fields in
        selected tuples, each field is an array containing values of the
        corresponding type

    SIsNull: struct [1,1] - structure containing info whether each value
        in selected tuples is null or not, each field is either logical
        array or cell array containing logical arrays

    SIsValueNull: struct [1,1] - structure containing a
       logical array [nTuples,1] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value

    indForward: double[1,nUniqueTuples] - indices of unique entries in
       the original tuple set

    indBackward: double[1,nTuples] - indices that map the unique tuple
       set back to the original tuple set

GETUNIQUEDATAALONGDIM - returns internal representation of CubeStruct

Input:
  regular:
    self:
    catDim: double[1,1] - dimension number along which uniqueness is
       checked

  properties
      fieldNameList: list of field names used for finding the unique
          elements; only the specified fields are returned in SData,
          SIsNull,SIsValueNull structures
      structNameList: list of internal structures to return (by default
          it is {SData, SIsNull, SIsValueNull}
      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all CubeStruct cells
              default value is false
      checkInputs: logical[1,1] - if true, the input parameters are
```

```
         checked for consistency

Output:
  regular:
    SData: struct [1,1] - structure containing values of fields

    SIsNull: struct [1,1] - structure containing info whether each value
        in selected cells is null or not, each field is either logical
        array or cell array containing logical arrays

    SIsValueNull: struct [1,1] - structure containing a
       logical array [nSlices,1] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value

    indForwardVec: double[nUniqueSlices,1] - indices of unique entries in
       the original CubeStruct data set

    indBackwardVec: double[nSlices,1] - indices that map the unique data
       set back to the original data setdata set unique along a specified
       dimension

GETUNIQUETUPLES - returns a relation containing the unique tuples from
                  the original relation

Usage: [resRel,indForwardVec,indBackwardVec]=getUniqueTuples(self,varargin)

Input:
  regular:
    self: ARelation [1,1] - class object
  properties
      fieldNameList: list of field names used for finding the unique
         tuples
      structNameList: list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}
      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the tuples
              default value is false

Output:
  regular:

    resRel: ARelation[1,1] - resulting relation

    indForward: double[1,nUniqueTuples] - indices of unique entries in
       the original tuple set

    indBackward: double[1,nTuples] - indices that map the unique tuple
       set back to the original tuple set

INITBYEMPTYDATASET - initializes cube struct object with null value arrays
                     of specified size based on minDimVec specified.

For instance, if minDimVec=[2,3,4,5,6] and minDimensionality of cube
struct object cb is 2, then cb.initByEmptyDataSet(minDimVec) will create
a cube struct object with element array of [2,3] size where each element
has size of [4,5,6,0]
```

```
Input:
  regular:
      self:
  optional
      minDimVec: double[1,nDims] - size vector of null value arrays

INITBYDEFAULTDATASET - initializes cube struct object with null value
                       arrays of specified size based on minDimVec
                       specified.

For instance, if minDimVec=[2,3,4,5,6] and minDimensionality of cube
struct object cb is 2, then cb.initByEmptyDataSet(minDimVec) will create
a cube struct object with element array of [2,3] size where each element
has size of [4,5,6]

Input:
  regular:
      self:
  optional
      minDimVec: double[1,nDims] - size vector of null value arrays

ISEQUAL - compares current relation object with other relation object and
          returns true if they are equal, otherwise it returns false


Usage: isEq=isEqual(self,otherObj)

Input:
  regular:
    self: ARelation [1,1] - current relation object
    otherObj: ARelation [1,1] - other relation object

  properties:
    checkFieldOrder/isFieldOrderCheck: logical [1,1] - if true, then fields
        in compared relations must be in the same order, otherwise the
        order is not  important (false by default)
    checkTupleOrder: logical[1,1] -  if true, then the tuples in the
        compared relations are expected to be in the same order,
        otherwise the order is not important (false by default)

    maxTolerance: double [1,1] - maximum allowed tolerance

    compareMetaDataBackwardRef: logical[1,1] if true, the CubeStruct's
        referenced from the meta data objects are also compared

    maxRelativeTolerance: double [1,1] - maximum allowed
    relative tolerance

Output:
  isEq: logical[1,1] - result of comparison
  reportStr: char[1,] - report of comparison

 ISFIELDS - returns whether all fields whose names are given in the input
            list are in the field list of given object or not

 Usage: isPositive=isFields(self,fieldList)

 Input:
```

```
   regular:
     self: CubeStruct [1,1]
     fieldList: char or char cell [1,nFields]/[nFields,1] - input list of
         given field names
 Output:
   isPositive: logical [1,1] - true if all gields whose
       names are given in the input list are in the field
       list of given object, false otherwise

   isUniqueNames: logical[1,1] - true if the specified names contain
      unique field values

   isThereVec: logical[1,nFields] - each element indicate whether the
       corresponding field is present in the cube

TODO allow for varargins
```

```
ISMEMBERALONGDIM - performs ismember operation of CubeStruct data slices
                   along the specified dimension
Input:
  regular:
    self: ARelation [1,1] - class object
    other: ARelation [1,1] - other class object
    dim: double[1,1] - dimension number for ismember operation

  properties:
    keyFieldNameList/fieldNameList: char or char cell [1,nKeyFields] -
        list  of fields to which ismember is applied; by default all
        fields of first (self) object are used


Output:
  regular:
    isThere: logical [nSlices,1] - determines for each data slice of the
        first (self) object whether combination of values for key fields
        is in the second (other) object or not
    indTheres: double [nSlices,1] - zero if the corresponding coordinate
        of isThere is false, otherwise the highest index of the
        corresponding data slice in the second (other) object
```

```
ISMEMBER - performs ismember operation for tuples of two relations by key
           fields given by special list

Usage: isTuple=isMemberTuples(self,otherRel,keyFieldNameList) or
       [isTuple indTuples]=isMemberTuples(self,otherRel,keyFieldNameList)

Input:
  regular:
    self: ARelation [1,1] - class object
    other: ARelation [1,1] - other class object
  optional:
    keyFieldNameList: char or char cell [1,nKeyFields] - list of fields
        to which ismember is applied; by default all fields of first
        (self) object are used
Output:
  regular:
    isTuple: logical [nTuples,1] - determines for each tuple of first
        (self) object whether combination of values for key fields is in
```

```
        the second (other) relation or not
    indTuples: double [nTuples,1] - zero if the corresponding coordinate
        of isTuple is false, otherwise the highest index of the
        corresponding tuple in the second (other) relation

ISUNIQUEKEY - checks if a specified set of fields forms a unique key

Usage: isPositive=self.isUniqueKey(fieldNameList)

Input:
  regular:
      self: ARelation [1,1] - class object
      fieldNameList: cell[1,nFields] - list of field names for a unique
          key candidate
Output:
  isPositive: logical[1,1] - true means that a specified set of fields is
     a unique key

REMOVEDUPLICATETUPLES - removes all duplicate tuples from the relation

Usage: [indForwardVec,indBackwardVec]=...
           removeDuplicateTuples(self,varargin)

Input:
  regular:
    self: ARelation [1,1] - class object

  properties:
      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values for all fields uniformly across all
          relation tuples
              default value is false

Output:
  optional:
    indForwardVec: double[nUniqueSlices,1] - indices of unique tuples in
       the original relation

    indBackwardVec: double[nSlices,1] - indices that map the unique
       tuples back to the original tuples

REMOVETUPLES - removes tuples with given indices from given relation

Usage: self.removeTuples(subIndVec)

Input:
  regular:
    self: ARelation [1,1] - class object
    subIndVec: double [nSubTuples,1]/logical[nTuples,1] - array of
       indices for tuples that are selected to be removed

REORDERDATA - reorders cells of CubeStruct object along the specified
              dimensions according to the specified index vectors

Input:
  regular:
      self: CubeStruct [1,1] - the object
      subIndCVec: numeric[1,]/cell[1,nDims] of double [nSubElem_i,1]
```

```
          for i=1,...,nDims array of indices of field value slices that
          are selected to be returned;
          if not given (default), no indexation is performed

  optional:
      dimVec: numeric[1,nDims] - vector of dimension numbers
          corresponding to subIndCVec

SAVEOBJ- transforms given CubeStruct object into structure containing
         internal representation of object properties

Input:
  regular:
    self: CubeStruct [nDim1,...,nDim2]


Output:
  regular:
    SObjectData: struct [n1,...,n_k] - structure containing an internal
       representation of the specified object

SETDATA - sets values of all cells for all fields

Input:
  regular:
    self: CubeStruct[1,1]

  optional:
    SData: struct [1,1] - structure with values of all cells for
        all fields

    SIsNull: struct [1,1] - structure of fields with is-null
       information for the field content, it can be logical for
       plain real numbers of cell of logicals for cell strs or
       cell of cell of str for more complex types

    SIsValueNull: struct [1,1] - structure with logicals
        determining whether value corresponding to each field
        and field cell is null or not

  properties:
      fieldNameList: cell[1,] of char[1,] - list of fields for which data
          should be generated, if not specified, all fields from the
          relation are taken

      isConsistencyCheckedVec: logical [1,1]/[1,2]/[1,3] -
          the first element defines if a consistency between the value
              elements (data, isNull and isValueNull) is checked;
          the second element (if specified) defines if
              value's type is checked.
          the third element defines if consistency between of sizes
              between different fields is checked
            If isConsistencyCheckedVec
              if scalar, it is automatically replicated to form a
                  3-element vector
              if the third element is not specified it is assumed
                  to be true
```

```
      transactionSafe: logical[1,1], if true, the operation is performed
         in a transaction-safe manner

      checkStruct: logical[1,nStruct] - an array of indicators which when
         all true force checking of structure content (including presence
         of required fields). The first element correspod to SData, the
         second and the third (if specified) to SIsNull and SIsValueNull
         correspondingly

      structNameList: char[1,]/cell[1,], name of data structure/list of
        data structure names to which the function is to
             be applied, can be composed from the following values

           SData - data itself

           SIsNull - contains is-null indicator information for data
                values

           SIsValueNull - contains is-null indicators for CubeStruct cells
               (not for cell values)
        structNameList={'SData'} by default

      fieldMetaData: smartdb.cubes.CubeStructFieldInfo[1,] - field meta
         data array which is used for data validity checking and for
         replacing the existing meta-data

      mdFieldNameList: cell[1,] of char - list of names of fields for
         which meta data is specified

      dataChangeIsComplete: logical[1,1] - indicates whether a change
          performed by the function is complete

Note: call of setData with an empty list of arguments clears
   the data

SETFIELDINTERNAL - sets values of all cells for given field

Usage: setFieldInternal(self,fieldName,value)

Input:
  regular:
    self: CubeStruct [1,1]
    fieldName: char - name of field
    value: array [] of some type - field values

  optional:
    isNull: logical/cell[]
    isValueNull: logical[]

  properties:
    structNameList: list of internal structures to return (by default it
      is {SData, SIsNull, SIsValueNull}

    inferIsNull: logical[1,2] - the first (second) element = false
      means that IsNull (IsValueNull) indicator for a field in question
          is kept intact (default = [true,true])

      Note: if structNameList contains 'SIsValueNull' entry,
```

```
       inferIsValueNull parameter is overwritten by false

SORTBY - sorts all tuples of given relation with respect to some of its
         fields

Usage: sortBy(self,sortFieldNameList,varargin)

input:
  regular:
    self: ARelation [1,1] - class object
    sortFieldNameList: char or char cell [1,nFields] - list of field
        names with respect to which tuples are sorted
  properties:
    direction: char or char cell [1,nFields] - direction of sorting for
        all fields (if one value is given) or for each field separately;
        each value may be 'asc' or 'desc'

SORTBYALONGDIM -  sorts data of given CubeStruct object along the
                  specified dimension using the specified fields

Usage: sortByInternal(self,sortFieldNameList,varargin)

input:
  regular:
    self: CubeStruct [1,1] - class object
    sortFieldNameList: char or char cell [1,nFields] - list of field
        names with respect to which field content is sorted
    sortDim: numeric[1,1] - dimension number along which the sorting is
       to be performed
    properties:
    direction: char or char cell [1,nFields] - direction of sorting for
        all fields (if one value is given) or for each field separately;
        each value may be 'asc' or 'desc'

TOARRAY - transforms values of all CubeStruct cells into a multi-
          dimentional array

Usage: resCArray=toArray(self,varargin)

Input:
  regular:
    self: CubeStruct [1,1]

  properties:
    checkInputs: logical[1,1] - if false, the method skips checking the
       input parameters for consistency

    fieldNameList: cell[1,] - list of filed names to return

    structNameList: cell[1,]/char[1,], data structure list
       for which the data is to be taken from, can consist of the
       following values

      SData - data itself
      SIsNull - contains is-null indicator information for data values
      SIsValueNull - contains is-null indicators for CubeStruct cells
         (not for cell values)
```

```
groupByColumns: logical[1,1], if true, each column is returned in a
   separate cell

outputType: char[1,] - method of formign an output array, the
   following methods are supported:
       'uniformMat' - the field values are concatenated without any
               type/size transformations. As a result, this method
               will fail if the specified fields have different types
               or/and sizes along any dimension apart from catDim

       'uniformCell' - not-cell fields are converted to cells
               element-wise but no size-transformations is performed.
               This method will fail if the specified fields have
               different sizes along any dimension apart from catDim

       'notUniform' - this method doesn't make any assumptions about
               size or type of the fields. Each field value is wrapped
               into cell in a such way that a size of resulting cell
               is minDimensionSizeVec for each field. Thus if for
               instance is size of cube object is [2,3,4] and a field
               size is [2,4,5,10,30] its value is splitted into 2*4*5
               pieces with each piece of size [1,1,1,10,30] put it
               its separate cell
       'adaptiveCell' - functions similarly to 'nonUniform' except for
               the cases when a field value size equals
               minDimensionSizeVec exactly i.e. the field takes only
               scalar values. In such cases no wrapping into cell is
               performed which allows to get a more transparent
               output.

catDim: double[1,1] - dimension number for
   concatenating outputs when groupByColumns is false


replaceNull: logical[1,1], if true, null values from SData are
   replaced by null replacement, = true by default

nullTopReplacement: - can be of any type and currently only applicable
  when  UniformOutput=false and of
  the corresponding column type if UniformOutput=true.

  Note!: this parameter is disregarded for any dataStructure different
     from 'SData'.

  Note!: the main difference between this parameter and the following
     parameters is that nullTopReplacement can violate field type
     constraints thus allowing to replace doubles with strings for
     instance (for non-uniform output types only of course)


nullReplacements: cell[1,nReplacedFields]  - list of null
   replacements for each of the fields

nullReplacementFields: cell[1,nReplacedFields] - list of fields in
   which the nulls are to be replaced with the specified values,
   if not specified it is assumed that all fields are to be replaced

   NOTE!: all fields not listed in this parameter are replaced with
```

```
       the default values


Output:
  Case1 (one output is requested and length(structNameList)==1):

      resCMat: matrix/cell[]  with values of all fields (or
        fields selected by optional arguments) for all CubeStruct
        data cells

  Case2 (multiple outputs are requested and their number =
    length(structNameList) each output is assigned resCMat for the
    corresponding struct

  Case3 (2 outputs is requested or length(structNameList)+1 outputs is
  requested). In this case the last output argument is

       isConvertedToCell: logical[nFields,nStructs] -  matrix with true
          values on the positions which correspond to fields converted to
          cells

TOCELL - transforms values of all fields for all tuples into two
         dimensional cell array

Usage: resCMat=toCell(self,varargin)

input:
  regular:
    self: ARelation [1,1] - class object
  optional:
    fieldName1: char - name of first field
    ...
    fieldNameN: char - name of N-th field
output:
  resCMat: cell [nTuples,nFields(N)] - cell with values of all fields (or
      fields selected by optional arguments) for all tuples

FIXME - order fields in setData method

TOCELLISNULL - transforms is-null indicators of all fields for all tuples
               into two dimensional cell array

Usage: resCMat=toCell(self,varargin)

input:
  regular:
    self: ARelation [1,1] - class object
  optional:
    fieldName1: char - name of first field
    ...
    fieldNameN: char - name of N-th field
output:
  resCMat: cell [nTuples,nFields(N)] - cell with values of all fields (or
      fields selected by optional arguments) for all tuples

FIXME - order fields in setData method
```

```
TODISPCELL - transforms values of all fields into their character
             representation

Usage: resCMat=toDispCell(self)

Input:
  regular:
    self: ARelation [1,1] - class object

  properties:
      nullTopReplacement: any[1,1] - value used to replace null values
      fieldNameList: cell[1,] of char[1,] - field name list

Output:
  dataCell: cell[nRows,nCols] of char[1,] - cell array containing the
      character representation of field values

TOMAT - transforms values of all fields for all tuples into two
        dimensional array

Usage: resCMat=toMat(self,varargin)

input:
  regular:
    self: ARelation [1,1] - class object

  optional:
    fieldNameList: cell[1,] - list of filed names to return

    uniformOutput: logical[1,1], true - cell is returned, false - the
       functions tries to return a result as a matrix

    groupByColumns: logical[1,1], if true, each column is returned in a
       separate cell

    structNameList/dataStructure: char[1,], data structure for which the
       data is to be taken from, can have one of the following values

      SData - data itself
      SIsNull - contains is-null indicator information for data values
      SIsValueNull - contains is-null indicators for relation cells (not
         for cell values

    replaceNull: logical[1,1], if true, null values from SData are
       replaced by null replacement, = true by default

    nullTopReplacement: - can be of any type and currently only applicable
      when  UniformOutput=false and of
      the corresponding column type if UniformOutput=true.

      Note!: this parameter is disregarded for any dataStructure different
         from 'SData'.

      Note!: the main difference between this parameter and the following
         parameters is that nullTopReplacement can violate field type
         constraints thus allowing to replace doubles with strings for
         instance (for non-uniform output types only of course)
```

```
    nullReplacements: cell[1,nReplacedFields]  - list of null
       replacements for each of the fields

    nullReplacementFields: cell[1,nReplacedFields] - list of fields in
       which the nulls are to be replaced with the specified values,
       if not specified it is assumed that all fields are to be replaced

       NOTE!: all fields not listed in this parameter are replaced with
       the default values

output:
  resCMat:  [nTuples,nFields(N)] - matrix/cell with values of all fields
      (or fields selected by optional arguments) for all tuples

TOSTRUCT - transforms given CubeStruct object into structure

Input:
  regular:
    self: CubeStruct [nDim1,...,nDim2]


Output:
  regular:
    SObjectData: struct [n1,...,n_k] - structure containing an internal
       representation of the specified object

UNIONWITH - adds tuples of the input relation to the set of tuples of the
            original relation
Usage: self.unionWith(inpRel)

Input:
  regular:
    self: ARelation [1,1] - class object
    inpRel1: ARelation [1,1] - object to get the additional tuples from
      ...
    inpRelN: ARelation [1,1] - object to get the additional tuples from

  properties:
      checkType: logical[1,1] - if true, union is only performed when the
          types of relations is the same. Default value is false

      checkStruct: logical[1,nStruct] - an array of indicators which when
         true force checking of structure content (including presence
         of all required fields). The first element correspod to SData,
         the second and the third (if specified) to SIsNull and
         SIsValueNull correspondingly

      checkConsistency: logical [1,1]/[1,2] - the
          first element defines if a consistency between the value
          elements (data, isNull and isValueNull) is checked;
          the second element (if specified) defines if
          value’s type is checked. If isConsistencyChecked
          is scalar, it is automatically replicated to form a
          two-element vector.
          Note: default value is true
```

```
UNIONWITHALONGDIM - adds data from the input CubeStructs

Usage: self.unionWithAlongDim(unionDim,inpCube)

Input:
  regular:
  self:
      inpCube1: CubeStruct [1,1] - object to get the additional data from
          ...
      inpCubeN: CubeStruct [1,1] - object to get the additional data from

  properties:
      checkType: logical[1,1] - if true, union is only performed when the
          types of relations is the same. Default value is false

      checkStruct: logical[1,nStruct] - an array of indicators which when
         true force checking of structure content (including presence of
all required fields). The first element correspod to SData, the
         second and the third (if specified) to SIsNull and SIsValueNull
         correspondingly

      checkConsistency: logical [1,1]/[1,2] - the
          first element defines if a consistency between the value
          elements (data, isNull and isValueNull) is checked;
          the second element (if specified) defines if
          value's type is checked. If isConsistencyChecked
          is scalar, it is automatically replicated to form a
          two-element vector.
          Note: default value is true

WRITETOCSV - writes a content of relation into Excel spreadsheet file
Input:
  regular:
      self:
      filePath: char[1,] - file path

Output:
  none

WRITETOXLS - writes a content of relation into Excel spreadsheet file
Input:
  regular:
      self:
      filePath: char[1,] - file path

Output:
  fileName: char[1,] - resulting file name, may not match with filePath
      when Excel is not available and csv format is used instead
```

9.6 gras.ellapx.smartdb.rels.EllTube

```
EllTube - class which keeps ellipsoidal tubes

Fields:
  QArray:cell[1, nElem] - Array of ellipsoid matrices
  aMat:cell[1, nElem] - Array of ellipsoid centers
```

```
scaleFactor:double[1, 1] - Tube scale factor
MArray:cell[1, nElem] - Array of regularization ellipsoid matrices
dim :double[1, 1] - Dimensionality
sTime:double[1, 1] - Time s
approxSchemaName:cell[1,] - Name
approxSchemaDescr:cell[1,] - Description
approxType:gras.ellapx.enums.EApproxType - Type of approximation
              (external, internal, not defined)
timeVec:cell[1, m] - Time vector
calcPrecision:double[1, 1] - Calculation precision
indSTime:double[1, 1]  - index of sTime within timeVec
ltGoodDirMat:cell[1, nElem] - Good direction curve
lsGoodDirVec:cell[1, nElem] - Good direction at time s
ltGoodDirNormVec:cell[1, nElem] - Norm of good direction curve
lsGoodDirNorm:double[1, 1] - Norm of good direction at time s
xTouchCurveMat:cell[1, nElem] - Touch point curve for good
                                direction
xTouchOpCurveMat:cell[1, nElem] - Touch point curve for direction
                                  opposite to good direction
xsTouchVec:cell[1, nElem]  - Touch point at time s
xsTouchOpVec :cell[1, nElem] - Touch point at time s

TODO: correct description of the fields in gras.ellapx.smartdb.rels.EllTube
```

See the description of the following methods in section smartdb.relations.ATypifiedStaticRelation for smartdb.relations.ATypifiedStaticRelation:

- smartdb.relations.ATypifiedStaticRelation.addData
- smartdb.relations.ATypifiedStaticRelation.addDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.addTuples
- smartdb.relations.ATypifiedStaticRelation.applyGetFunc
- smartdb.relations.ATypifiedStaticRelation.applySetFunc
- smartdb.relations.ATypifiedStaticRelation.applyTupleGetFunc
- smartdb.relations.ATypifiedStaticRelation.clearData
- smartdb.relations.ATypifiedStaticRelation.clone
- smartdb.relations.ATypifiedStaticRelation.copyFrom
- smartdb.relations.ATypifiedStaticRelation.createInstance
- smartdb.relations.ATypifiedStaticRelation.dispOnUI
- smartdb.relations.ATypifiedStaticRelation.display
- smartdb.relations.ATypifiedStaticRelation.fromStructList
- smartdb.relations.ATypifiedStaticRelation.getCopy
- smartdb.relations.ATypifiedStaticRelation.getFieldIsNull
- smartdb.relations.ATypifiedStaticRelation.getFieldIsValueNull
- smartdb.relations.ATypifiedStaticRelation.getFieldNameList
- smartdb.relations.ATypifiedStaticRelation.getFieldProjection
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeList
- smartdb.relations.ATypifiedStaticRelation.getFieldValueSizeMat

- smartdb.relations.ATypifiedStaticRelation.getIsFieldValueNull
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionSize
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionality
- smartdb.relations.ATypifiedStaticRelation.getNElems
- smartdb.relations.ATypifiedStaticRelation.getNFields
- smartdb.relations.ATypifiedStaticRelation.getNTuples
- smartdb.relations.ATypifiedStaticRelation.getSortIndex
- smartdb.relations.ATypifiedStaticRelation.getTuples
- smartdb.relations.ATypifiedStaticRelation.getTuplesFilteredBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesIndexedBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesJoinedWith
- smartdb.relations.ATypifiedStaticRelation.getUniqueData
- smartdb.relations.ATypifiedStaticRelation.getUniqueDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.getUniqueTuples
- smartdb.relations.ATypifiedStaticRelation.initByEmptyDataSet
- smartdb.relations.ATypifiedStaticRelation.initByDefaultDataSet
- smartdb.relations.ATypifiedStaticRelation.isFields
- smartdb.relations.ATypifiedStaticRelation.isMemberAlongDim
- smartdb.relations.ATypifiedStaticRelation.isMember
- smartdb.relations.ATypifiedStaticRelation.isUniqueKey
- smartdb.relations.ATypifiedStaticRelation.isEqual
- smartdb.relations.ATypifiedStaticRelation.removeDuplicateTuples
- smartdb.relations.ATypifiedStaticRelation.removeTuples
- smartdb.relations.ATypifiedStaticRelation.reorderData
- smartdb.relations.ATypifiedStaticRelation.saveObj
- smartdb.relations.ATypifiedStaticRelation.setData
- smartdb.relations.ATypifiedStaticRelation.setFieldInternal
- smartdb.relations.ATypifiedStaticRelation.sortBy
- smartdb.relations.ATypifiedStaticRelation.sortByAlongDim
- smartdb.relations.ATypifiedStaticRelation.toArray
- smartdb.relations.ATypifiedStaticRelation.toCell
- smartdb.relations.ATypifiedStaticRelation.toCellIsNull
- smartdb.relations.ATypifiedStaticRelation.toDispCell
- smartdb.relations.ATypifiedStaticRelation.toMat
- smartdb.relations.ATypifiedStaticRelation.toStruct
- **smartdb.relations.ATypifiedStaticRelation.unionWith_**

- smartdb.relations.ATypifiedStaticRelation.unionWithAlongDim
- smartdb.relations.ATypifiedStaticRelation.writeToCSV
- smartdb.relations.ATypifiedStaticRelation.writeToXLS

```
CAT  - concatenates data from relation objects.

Input:
  regular:
      self.
      newEllTubeRel: smartdb.relation.StaticRelation[1, 1]/
          smartdb.relation.DynamicRelation[1, 1] - relation object
  properties:
      isReplacedByNew: logical[1,1] - if true, sTime and
          values of properties corresponding to sTime are taken
          from newEllTubeRel. Common times in self and
          newEllTubeRel are allowed, however the values for
          those times are taken either from self or from
          newEllTubeRel depending on value of isReplacedByNew
          property

      isCommonValuesChecked: logical[1,1] - if true, values
          at common times (if such are found) are checked for
          strong equality (with zero precision). If not equal
          - an exception is thrown. True by default.

      commonTimeAbsTol: double[1,1] - absolute tolerance used
          for comparing values at common times, =0 by default

      commonTimeRelTol: double[1,1] - absolute tolerance used
          for comparing values at common times, =0 by default

Output:
  catEllTubeRel:smartdb.relation.StaticRelation[1, 1]/
      smartdb.relation.DynamicRelation[1, 1] - relation object
      resulting from CAT operation
```

```
FROMELLARRAY  - creates a relation object using an array of ellipsoids

Input:
  regular:
    qEllArray: ellipsoid[nDim1, nDim2, ..., nDimN] - array of ellipsoids

  optional:
   timeVec:cell[1, m] - time vector
   ltGoodDirArray:cell[1, nElem] - good direction at time s
   sTime:double[1, 1] - time s
   approxType:gras.ellapx.enums.EApproxType - type of approximation
                 (external, internal, not defined)
   approxSchemaName:cell[1,] - name of the schema
   approxSchemaDescr:cell[1,] - description of the schema
   calcPrecision:double[1, 1] - calculation precision

Output:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1] - constructed relation
       object
```

```
FROMELLMARRAY  - creates a relation object using an array of ellipsoids.
                 This method uses regularizer in the form of a matrix
                 function.

Input:
  regular:
    qEllArray: ellipsoid[nDim1, nDim2, ..., nDimN] - array of ellipsoids
    ellMArr: double[nDim1, nDim2, ..., nDimN] - regularization ellipsoid
        matrices

  optional:
   timeVec:cell[1, m] - time vector
   ltGoodDirArray:cell[1, nElem] - good direction at time s
   sTime:double[1, 1] - time s
   approxType:gras.ellapx.enums.EApproxType - type of approximation
                (external, internal, not defined)
   approxSchemaName:cell[1,] - name of the schema
   approxSchemaDescr:cell[1,] - description of the schema
   calcPrecision:double[1, 1] - calculation precision

Output:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1] - constructed relation
         object

FROMQARRAYS  - creates a relation object using an array of ellipsoids,
               described by the array of ellipsoid matrices and
               array of ellipsoid centers.This method used default
               scale factor.

Input:
  regular:
    QArrayList: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid
        matrices
    aMat: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid centers

Optional:
   MArrayList:cell[1, nElem] - array of regularization ellipsoid matrices
   timeVec:cell[1, m] - time vector
   ltGoodDirArray:cell[1, nElem] - good direction at time s
   sTime:double[1, 1] - time s
   approxType:gras.ellapx.enums.EApproxType - type of approximation
                (external, internal, not defined)
   approxSchemaName:cell[1,] - name of the schema
   approxSchemaDescr:cell[1,] - description of the schema
   calcPrecision:double[1, 1] - calculation precision

Output:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1] - constructed relation
       object

FROMQMARRAYS  - creates a relation object using an array of ellipsoids,
                described by the array of ellipsoid matrices and
                array of ellipsoid centers. Also this method uses
                regularizer in the form of a matrix function. This method
                used default scale factor.

Input:
  regular:
```

```
  QArrayList: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid
        matrices
  aMat: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid centers
  MArrayList: double[nDim1, nDim2, ..., nDimN] - ellipsoid  matrices of
        regularization

 optional:
   timeVec:cell[1, m] - time vector
   ltGoodDirArray:cell[1, nElem] - good direction at time s
   sTime:double[1, 1] - time s
   approxType:gras.ellapx.enums.EApproxType - type of approximation
                (external, internal, not defined)
   approxSchemaName:cell[1,] - name of the schema
   approxSchemaDescr:cell[1,] - description of the schema
   calcPrecision:double[1, 1] - calculation precision

Output:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1] - constructed relation
         object

FROMQMSCALEDARRAYS  - creates a relation object using an array of ellipsoids,
                      described by the array of ellipsoid matrices and
                      array of ellipsoid centers. Also this method uses
                      regularizer in the form of a matrix function.


Input:
  regular:
    QArrayList: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid
        matrices
    aMat: double[nDim1, nDim2, ..., nDimN] - array of ellipsoid centers
    MArrayList: double[nDim1, nDim2, ..., nDimN] - ellipsoid matrices
              of regularization
    scaleFactor:double[1, 1] - tube scale factor

 optional:
   timeVec:cell[1, m] - time vector
   ltGoodDirArray:cell[1, nElem] - good direction at time s
   sTime:double[1, 1] - time s
   approxType:gras.ellapx.enums.EApproxType - type of approximation
                (external, internal, not defined)
   approxSchemaName:cell[1,] - name of the schema
   approxSchemaDescr:cell[1,] - description of the schema
   calcPrecision:double[1, 1] - calculation precision

Output:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1] - constructed relation
         object

GETDATA - returns an indexed projection of CubeStruct object's content

Input:
  regular:
      self: CubeStruct [1,1] - the object

  optional:

      subIndCVec:
```

```
        Case#1: numeric[1,]/numeric[,1]

        Case#2: cell[1,nDims]/cell[nDims,1] of double [nSubElem_i,1]
              for i=1,...,nDims

          -array of indices of field value slices that are selected
          to be returned; if not given (default),
          no indexation is performed

        Note!: numeric components of subIndVec are allowed to contain
           zeros which are be treated as they were references to null
           data slices

      dimVec: numeric[1,nDims]/numeric[nDims,1] - vector of dimension
          numbers corresponding to subIndCVec

  properties:

      fieldNameList: char[1,]/cell[1,nFields] of char[1,]
          list of field names to return

      structNameList: char[1,]/cell[1,nStructs] of char[1,]
          list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}

      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the cells,
              default value is false

      nullReplacements: cell[1,nReplacedFields]  - list of null
          replacements for each of the fields

      nullReplacementFields: cell[1,nReplacedFields] - list of fields in
         which the nulls are to be replaced with the specified values,
         if not specified it is assumed that all fields are to be
         replaced

         NOTE!: all fields not listed in this parameter are replaced with
         the default values

      checkInputs: logical[1,1] - true by default (input arguments are
         checked for correctness

Output:
  regular:
    SData: struct [1,1] - structure containing values of
        fields at the selected slices, each field is an array
        containing values of the corresponding type

    SIsNull: struct [1,1] - structure containing a nested
        array with is-null indicators for each CubeStruct cell content

    SIsValueNull: struct [1,1] - structure containing a
       logical array [] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value
```

```
GETELLARRAY - returns array of matrix's ellipsoid according to
              approxType

Input:
 regular:
    self.
    approxType:char[1,] - type of approximation(internal/external)

Output:
  apprEllMat:double[nDim1,..., nDimN] - array of array of ellipsoid's
          matrices

GETJOINWITH - returns a result of INNER join of given relation with
              another relation by the specified key fields

LIMITATION: key fields by which the join is peformed are required to form
a unique key in the given relation

Input:
  regular:
      self:
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT)
          'leftOuter'

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result

::

ISEQUAL - compares current relation object with other relation object and
          returns true if they are equal, otherwise it returns false


Usage: isEq=isEqual(self,otherObj)

Input:
  regular:
    self: ARelation [1,1] - current relation object
    otherObj: ARelation [1,1] - other relation object

  properties:
    checkFieldOrder/isFieldOrderCheck: logical [1,1] - if true, then fields
        in compared relations must be in the same order, otherwise the
        order is not  important (false by default)
    checkTupleOrder: logical[1,1] -  if true, then the tuples in the
        compared relations are expected to be in the same order,
        otherwise the order is not important (false by default)

    maxTolerance: double [1,1] - maximum allowed tolerance

    compareMetaDataBackwardRef: logical[1,1] if true, the CubeStruct's
        referenced from the meta data objects are also compared
```

```
    maxRelativeTolerance: double [1,1] - maximum allowed
    relative tolerance

Output:
  isEq: logical[1,1] - result of comparison
  reportStr: char[1,] - report of comparsion

PLOT - displays ellipsoidal tubes using the specified RelationDataPlotter


Input:
  regular:
      self:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - plotter
          object used for displaying ellipsoidal tubes

PROJECT - computes projection of the relation object onto given time
          dependent subspase
Input:
  regular:
      self.
      projType: gras.ellapx.enums.EProjType[1,1] -
          type of the projection, can be
          'Static' and 'DynamicAlongGoodCurve'
      projMatList: cell[1,nProj] of double[nSpDim,nDim] - list of
          projection matrices, not necessarily orthogonal
   fGetProjMat: function_handle[1,1] - function which creates
      vector of the projection
            matrices
       Input:
        regular:
          projMat:double[nDim, mDim] - matrix of the projection at the
            instant of time
          timeVec:double[1, nDim] - time interval
        optional:
           sTime:double[1,1] - instant of time
       Output:
          projOrthMatArray:double[1, nSpDim] - vector of the projection
            matrices
          projOrthMatTransArray:double[nSpDim, 1] - transposed vector of
            the projection matrices
Output:
   ellTubeProjRel: gras.ellapx.smartdb.rels.EllTubeProj[1, 1]/
       gras.ellapx.smartdb.rels.EllTubeUnionProj[1, 1] -
          projected ellipsoidal tube

   indProj2OrigVec:cell[nDim, 1] - index of the line number from
            which is obtained the projection

Example:
  function example
   aMat = [0 1; 0 0]; bMat = eye(2);
   SUBounds = struct();
   SUBounds.center = {'sin(t)'; 'cos(t)'};
   SUBounds.shape = [9 0; 0 2];
   sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
   x0EllObj = ell_unitball(2);
   timeVec = [0 10];
```

```
   dirsMat = [1 0; 0 1]';
   rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
   ellTubeObj = rsObj.getEllTubeRel();
   unionEllTube = ...
    gras.ellapx.smartdb.rels.EllUnionTube.fromEllTubes(ellTubeObj);
   projMatList = {[1 0;0 1]};
   projType = gras.ellapx.enums.EProjType.Static;
   statEllTubeProj = unionEllTube.project(projType,projMatList,...
      @fGetProjMat);
   plObj=smartdb.disp.RelationDataPlotter();
   statEllTubeProj.plot(plObj);
end

function [projOrthMatArray,projOrthMatTransArray]=fGetProjMat(projMat,...
    timeVec,varargin)
  nTimePoints=length(timeVec);
  projOrthMatArray=repmat(projMat,[1,1,nTimePoints]);
  projOrthMatTransArray=repmat(projMat.',[1,1,nTimePoints]);
 end

PROJECTTOORTHS - project elltube onto subspace defined by
vectors of standart basis with indices specified in indVec

Input:
  regular:
      self: gras.ellapx.smartdb.rels.EllTube[1, 1] - elltube
          object
      indVec: double[1, nProjDims] - indices specifying a subset of
          standart basis
  optional:
      projType: gras.ellapx.enums.EProjType[1, 1] -  type of
          projection

Output:
  regular:
      ellTubeProjRel: gras.ellapx.smartdb.rels.EllTubeProj[1, 1] -
          elltube projection

Example:
  ellTubeProjRel = ellTubeRel.projectToOrths([1,2])
  projType = gras.ellapx.enums.EProjType.DynamicAlongGoodCurve
  ellTubeProjRel = ellTubeRel.projectToOrths([3,4,5], projType)

SCALE - scales relation object

 Input:
  regular:
     self.
     fCalcFactor - function which calculates factor for
                    fields in fieldNameList
       Input:
         regular:
           fieldNameList: char/cell[1,] of char - a list of fields
                  for which factor will be calculated
        Output:
            factor:double[1, 1] - calculated factor

      fieldNameList:cell[1,nElem]/char[1,] - names of the fields
```

```
 Output:
      none

Example:
  nPoints=5;
  calcPrecision=0.001;
  approxSchemaDescr=char.empty(1,0);
  approxSchemaName=char.empty(1,0);
  nDims=3;
  nTubes=1;
  lsGoodDirVec=[1;0;1];
  aMat=zeros(nDims,nPoints);
  timeVec=1:nPoints;
  sTime=nPoints;
  approxType=gras.ellapx.enums.EApproxType.Internal;
  qArrayList=repmat({repmat(diag([1 2 3]),[1,1,nPoints])},1,nTubes);
  ltGoodDirArray=repmat(lsGoodDirVec,[1,nTubes,nPoints]);
  fromMatEllTube=...
        gras.ellapx.smartdb.rels.EllTube.fromQArrays(qArrayList,...
        aMat, timeVec,ltGoodDirArray, sTime, approxType,...
        approxSchemaName, approxSchemaDescr, calcPrecision);
  fromMatEllTube.scale(@(varargin)2,{});

::
```

9.7 gras.ellapx.smartdb.rels.EllTubeProj

```
EllTubeProj - class which keeps ellipsoidal tube's projection

Fields:
  QArray:cell[1, nElem] - Array of ellipsoid matrices
  aMat:cell[1, nElem] - Array of ellipsoid centers
  scaleFactor:double[1, 1] - Tube scale factor
  MArray:cell[1, nElem] - Array of regularization ellipsoid matrices
  dim :double[1, 1] - Dimensionality
  sTime:double[1, 1] - Time s
  approxSchemaName:cell[1,] - Name
  approxSchemaDescr:cell[1,] - Description
  approxType:gras.ellapx.enums.EApproxType - Type of approximation
                 (external, internal, not defined)
  timeVec:cell[1, m] - Time vector
  calcPrecision:double[1, 1] - Calculation precision
  indSTime:double[1, 1]  - index of sTime within timeVec
  ltGoodDirMat:cell[1, nElem] - Good direction curve
  lsGoodDirVec:cell[1, nElem] - Good direction at time s
  ltGoodDirNormVec:cell[1, nElem] - Norm of good direction curve
  lsGoodDirNorm:double[1, 1] - Norm of good direction at time s
  xTouchCurveMat:cell[1, nElem] - Touch point curve for good
                                direction
  xTouchOpCurveMat:cell[1, nElem] - Touch point curve for direction
                                  opposite to good direction
  xsTouchVec:cell[1, nElem]  - Touch point at time s
  xsTouchOpVec:cell[1, nElem] - Touch point at time s
  projSTimeMat: cell[1, 1] - Projection matrix at time s
  projType:gras.ellapx.enums.EProjType - Projection type
```

```
  ltGoodDirNormOrigVec:cell[1, 1] - Norm of the original (not
                                   projected) good direction curve
  lsGoodDirNormOrig:double[1, 1] - Norm of the original (not
                                  projected)good direction at time s
  lsGoodDirOrigVec:cell[1, 1] - Original (not projected) good
                               direction at time s

TODO: correct description of the fields in
    gras.ellapx.smartdb.rels.EllTubeProj
```

See the description of the following methods in section smartdb.relations.ATypifiedStaticRelation for smartdb.relations.ATypifiedStaticRelation:

- smartdb.relations.ATypifiedStaticRelation.addData
- smartdb.relations.ATypifiedStaticRelation.addDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.addTuples
- smartdb.relations.ATypifiedStaticRelation.applyGetFunc
- smartdb.relations.ATypifiedStaticRelation.applySetFunc
- smartdb.relations.ATypifiedStaticRelation.applyTupleGetFunc
- smartdb.relations.ATypifiedStaticRelation.clearData
- smartdb.relations.ATypifiedStaticRelation.clone
- smartdb.relations.ATypifiedStaticRelation.copyFrom
- smartdb.relations.ATypifiedStaticRelation.createInstance
- smartdb.relations.ATypifiedStaticRelation.dispOnUI
- smartdb.relations.ATypifiedStaticRelation.display
- smartdb.relations.ATypifiedStaticRelation.fromStructList
- smartdb.relations.ATypifiedStaticRelation.getCopy
- smartdb.relations.ATypifiedStaticRelation.getFieldDescrList
- smartdb.relations.ATypifiedStaticRelation.getFieldIsNull
- smartdb.relations.ATypifiedStaticRelation.getFieldIsValueNull
- smartdb.relations.ATypifiedStaticRelation.getFieldNameList
- smartdb.relations.ATypifiedStaticRelation.getFieldProjection
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeList
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeSpecList
- smartdb.relations.ATypifiedStaticRelation.getFieldValueSizeMat
- smartdb.relations.ATypifiedStaticRelation.getIsFieldValueNull
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionSize
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionality
- smartdb.relations.ATypifiedStaticRelation.getNElems
- smartdb.relations.ATypifiedStaticRelation.getNFields
- smartdb.relations.ATypifiedStaticRelation.getNTuples

- smartdb.relations.ATypifiedStaticRelation.getSortIndex
- smartdb.relations.ATypifiedStaticRelation.getTuples
- smartdb.relations.ATypifiedStaticRelation.getTuplesFilteredBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesIndexedBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesJoinedWith
- smartdb.relations.ATypifiedStaticRelation.getUniqueData
- smartdb.relations.ATypifiedStaticRelation.getUniqueDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.getUniqueTuples
- smartdb.relations.ATypifiedStaticRelation.initByEmptyDataSet
- smartdb.relations.ATypifiedStaticRelation.initByDefaultDataSet
- smartdb.relations.ATypifiedStaticRelation.isFields
- smartdb.relations.ATypifiedStaticRelation.isMemberAlongDim
- smartdb.relations.ATypifiedStaticRelation.isMember
- smartdb.relations.ATypifiedStaticRelation.isUniqueKey
- smartdb.relations.ATypifiedStaticRelation.isEqual
- smartdb.relations.ATypifiedStaticRelation.removeDuplicateTuples
- smartdb.relations.ATypifiedStaticRelation.removeTuples
- smartdb.relations.ATypifiedStaticRelation.reorderData
- smartdb.relations.ATypifiedStaticRelation.saveObj
- smartdb.relations.ATypifiedStaticRelation.setData
- smartdb.relations.ATypifiedStaticRelation.setFieldInternal
- smartdb.relations.ATypifiedStaticRelation.sortBy
- smartdb.relations.ATypifiedStaticRelation.sortByAlongDim
- smartdb.relations.ATypifiedStaticRelation.toArray
- smartdb.relations.ATypifiedStaticRelation.toCell
- smartdb.relations.ATypifiedStaticRelation.toCellIsNull
- smartdb.relations.ATypifiedStaticRelation.toDispCell
- smartdb.relations.ATypifiedStaticRelation.toMat
- smartdb.relations.ATypifiedStaticRelation.toStruct
- **smartdb.relations.ATypifiedStaticRelation.unionWith_**
- smartdb.relations.ATypifiedStaticRelation.unionWithAlongDim
- smartdb.relations.ATypifiedStaticRelation.writeToCSV
- smartdb.relations.ATypifiedStaticRelation.writeToXLS

```
GETDATA - returns an indexed projection of CubeStruct object's content

Input:
  regular:
```

```
      self: CubeStruct [1,1] - the object

  optional:

      subIndCVec:
        Case#1: numeric[1,]/numeric[,1]

        Case#2: cell[1,nDims]/cell[nDims,1] of double [nSubElem_i,1]
              for i=1,...,nDims

          -array of indices of field value slices that are selected
          to be returned; if not given (default),
          no indexation is performed

        Note!: numeric components of subIndVec are allowed to contain
           zeros which are be treated as they were references to null
           data slices

      dimVec: numeric[1,nDims]/numeric[nDims,1] - vector of dimension
          numbers corresponding to subIndCVec

  properties:

      fieldNameList: char[1,]/cell[1,nFields] of char[1,]
          list of field names to return

      structNameList: char[1,]/cell[1,nStructs] of char[1,]
          list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}

      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the cells,
              default value is false

      nullReplacements: cell[1,nReplacedFields]  - list of null
          replacements for each of the fields

      nullReplacementFields: cell[1,nReplacedFields] - list of fields in
         which the nulls are to be replaced with the specified values,
         if not specified it is assumed that all fields are to be
         replaced

         NOTE!: all fields not listed in this parameter are replaced with
         the default values

      checkInputs: logical[1,1] - true by default (input arguments are
         checked for correctness

Output:
  regular:
    SData: struct [1,1] - structure containing values of
        fields at the selected slices, each field is an array
        containing values of the corresponding type

    SIsNull: struct [1,1] - structure containing a nested
        array with is-null indicators for each CubeStruct cell content

    SIsValueNull: struct [1,1] - structure containing a
```

```
       logical array [] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value

GETELLARRAY - returns array of matrix's ellipsoid according to
              approxType

Input:
 regular:
    self.
    approxType:char[1,] - type of approximation(internal/external)

Output:
  apprEllMat:double[nDim1,..., nDimN] - array of array of ellipsoid's
           matrices

GETJOINWITH - returns a result of INNER join of given relation with
              another relation by the specified key fields

LIMITATION: key fields by which the join is peformed are required to form
a unique key in the given relation

Input:
  regular:
      self:
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT)
          'leftOuter'

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result

GETREACHTUBEANEPREFIX - return prefix of the reach tube

Input:
  regular:
     self.

GETREGTUBEANEPREFIX - return prefix of the reg tube

Input:
  regular:
     self.

ISEQUAL - compares current relation object with other relation object and
          returns true if they are equal, otherwise it returns false


Usage: isEq=isEqual(self,otherObj)

Input:
  regular:
    self: ARelation [1,1] - current relation object
    otherObj: ARelation [1,1] - other relation object
```

```
  properties:
    checkFieldOrder/isFieldOrderCheck: logical [1,1] - if true, then fields
        in compared relations must be in the same order, otherwise the
        order is not  important (false by default)
    checkTupleOrder: logical[1,1] -  if true, then the tuples in the
        compared relations are expected to be in the same order,
        otherwise the order is not important (false by default)

    maxTolerance: double [1,1] - maximum allowed tolerance

    compareMetaDataBackwardRef: logical[1,1] if true, the CubeStruct's
        referenced from the meta data objects are also compared

    maxRelativeTolerance: double [1,1] - maximum allowed
    relative tolerance

Output:
  isEq: logical[1,1] - result of comparison
  reportStr: char[1,] - report of comparsion

PLOT - displays ellipsoidal tubes using the specified
  RelationDataPlotter

Input:
  regular:
      self:
  optional:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - plotter
          object used for displaying ellipsoidal tubes
  properties:
      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used

Output:
  plObj: smartdb.disp.RelationDataPlotter[1,1] - plotter
          object used for displaying ellipsoidal tubes
```

```
PLOTEXT - plots external approximation of ellTube.


Usage:
      obj.plotExt() - plots external approximation of ellTube.
      obj.plotExt('Property',PropValue,...) - plots external approximation
                                              of ellTube with setting
                                              properties.

Input:
  regular:
      obj:  EllTubeProj: EllTubeProj object
  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      colorSpec: char[1,1] - color specification code, can be 'r','g',
                 etc (any code supported by built-in Matlab function).

  properties:

      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used
      'showDiscrete':logical[1,1]  -
          if true, approximation in 3D will be filled in every time slice
      'nSpacePartPoins': double[1,1] -
          number of points in every time slice.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

PLOTINT - plots internal approximation of ellTube.


Usage:
      obj.plotInt() - plots internal approximation of ellTube.
      obj.plotInt('Property',PropValue,...) - plots internal approximation
                                              of ellTube with setting
```

```
                                                  properties.

Input:
  regular:
      obj:  EllTubeProj: EllTubeProj object
  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      colorSpec: char[1,1] - color specification code, can be 'r','g',
                   etc (any code supported by built-in Matlab function).

  properties:

      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used
      'showDiscrete':logical[1,1]  -
          if true, approximation in 3D will be filled in every time slice
      'nSpacePartPoins': double[1,1] -
          number of points in every time slice.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

::

::
```

9.8 gras.ellapx.smartdb.rels.EllUnionTube

```
EllUionTube - class which keeps ellipsoidal tubes by the instant of
              time

Fields:
  QArray:cell[1, nElem] - Array of ellipsoid matrices
  aMat:cell[1, nElem] - Array of ellipsoid centers
```

```
  scaleFactor:double[1, 1] - Tube scale factor
  MArray:cell[1, nElem] - Array of regularization ellipsoid matrices
  dim :double[1, 1] - Dimensionality
  sTime:double[1, 1] - Time s
  approxSchemaName:cell[1,] - Name
  approxSchemaDescr:cell[1,] - Description
  approxType:gras.ellapx.enums.EApproxType - Type of approximation
                (external, internal, not defined
  timeVec:cell[1, m] - Time vector
  calcPrecision:double[1, 1] - Calculation precision
  indSTime:double[1, 1]  - index of sTime within timeVec
  ltGoodDirMat:cell[1, nElem] - Good direction curve
  lsGoodDirVec:cell[1, nElem] - Good direction at time s
  ltGoodDirNormVec:cell[1, nElem] - Norm of good direction curve
  lsGoodDirNorm:double[1, 1] - Norm of good direction at time s
  xTouchCurveMat:cell[1, nElem] - Touch point curve for good
                                  direction
  xTouchOpCurveMat:cell[1, nElem] - Touch point curve for direction
                                    opposite to good direction
  xsTouchVec:cell[1, nElem]  - Touch point at time s
  xsTouchOpVec :cell[1, nElem] - Touch point at time s
  ellUnionTimeDirection:gras.ellapx.enums.EEllUnionTimeDirection -
                     Direction in time along which union is performed
  isLsTouch:logical[1, 1] - Indicates whether a touch takes place
                            along LS
  isLsTouchOp:logical[1, 1] - Indicates whether a touch takes place
                              along LS opposite
  isLtTouchVec:cell[1, nElem] - Indicates whether a touch takes place
                                along LT
  isLtTouchOpVec:cell[1, nElem] - Indicates whether a touch takes
                                  place along LT opposite
  timeTouchEndVec:cell[1, nElem] - Touch point curve for good
                                   direction
  timeTouchOpEndVec:cell[1, nElem] - Touch point curve for good
                                     direction

TODO: correct description of the fields in
    gras.ellapx.smartdb.rels.EllUnionTube
```

See the description of the following methods in section smartdb.relations.ATypifiedStaticRelation for smartdb.relations.ATypifiedStaticRelation:

- smartdb.relations.ATypifiedStaticRelation.addData
- smartdb.relations.ATypifiedStaticRelation.addDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.addTuples
- smartdb.relations.ATypifiedStaticRelation.applyGetFunc
- smartdb.relations.ATypifiedStaticRelation.applySetFunc
- smartdb.relations.ATypifiedStaticRelation.applyTupleGetFunc
- smartdb.relations.ATypifiedStaticRelation.clearData
- smartdb.relations.ATypifiedStaticRelation.clone
- smartdb.relations.ATypifiedStaticRelation.copyFrom
- smartdb.relations.ATypifiedStaticRelation.createInstance

- smartdb.relations.ATypifiedStaticRelation.dispOnUI
- smartdb.relations.ATypifiedStaticRelation.display
- smartdb.relations.ATypifiedStaticRelation.fromStructList
- smartdb.relations.ATypifiedStaticRelation.getCopy
- smartdb.relations.ATypifiedStaticRelation.getFieldDescrList
- smartdb.relations.ATypifiedStaticRelation.getFieldIsNull
- smartdb.relations.ATypifiedStaticRelation.getFieldIsValueNull
- smartdb.relations.ATypifiedStaticRelation.getFieldNameList
- smartdb.relations.ATypifiedStaticRelation.getFieldProjection
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeList
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeSpecList
- smartdb.relations.ATypifiedStaticRelation.getFieldValueSizeMat
- smartdb.relations.ATypifiedStaticRelation.getIsFieldValueNull
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionSize
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionality
- smartdb.relations.ATypifiedStaticRelation.getNElems
- smartdb.relations.ATypifiedStaticRelation.getNFields
- smartdb.relations.ATypifiedStaticRelation.getNTuples
- smartdb.relations.ATypifiedStaticRelation.getSortIndex
- smartdb.relations.ATypifiedStaticRelation.getTuples
- smartdb.relations.ATypifiedStaticRelation.getTuplesFilteredBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesIndexedBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesJoinedWith
- smartdb.relations.ATypifiedStaticRelation.getUniqueData
- smartdb.relations.ATypifiedStaticRelation.getUniqueDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.getUniqueTuples
- smartdb.relations.ATypifiedStaticRelation.initByEmptyDataSet
- smartdb.relations.ATypifiedStaticRelation.initByDefaultDataSet
- smartdb.relations.ATypifiedStaticRelation.isFields
- smartdb.relations.ATypifiedStaticRelation.isMemberAlongDim
- smartdb.relations.ATypifiedStaticRelation.isMember
- smartdb.relations.ATypifiedStaticRelation.isUniqueKey
- smartdb.relations.ATypifiedStaticRelation.isEqual
- smartdb.relations.ATypifiedStaticRelation.removeDuplicateTuples
- smartdb.relations.ATypifiedStaticRelation.removeTuples
- smartdb.relations.ATypifiedStaticRelation.reorderData

- smartdb.relations.ATypifiedStaticRelation.saveObj
- smartdb.relations.ATypifiedStaticRelation.setData
- smartdb.relations.ATypifiedStaticRelation.setFieldInternal
- smartdb.relations.ATypifiedStaticRelation.sortBy
- smartdb.relations.ATypifiedStaticRelation.sortByAlongDim
- smartdb.relations.ATypifiedStaticRelation.toArray
- smartdb.relations.ATypifiedStaticRelation.toCell
- smartdb.relations.ATypifiedStaticRelation.toCellIsNull
- smartdb.relations.ATypifiedStaticRelation.toDispCell
- smartdb.relations.ATypifiedStaticRelation.toMat
- smartdb.relations.ATypifiedStaticRelation.toStruct
- **smartdb.relations.ATypifiedStaticRelation.unionWith_**
- smartdb.relations.ATypifiedStaticRelation.unionWithAlongDim
- smartdb.relations.ATypifiedStaticRelation.writeToCSV
- smartdb.relations.ATypifiedStaticRelation.writeToXLS

```
FROMELLTUBES - returns union of the ellipsoidal tubes on time

Input:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1]/
      smartdb.relation.DynamicRelation[1, 1] - relation
      object

Output:
ellUnionTubeRel: ellapx.smartdb.rel.EllUnionTube - union of the
            ellipsoidal tubes

GETDATA - returns an indexed projection of CubeStruct object’s content

Input:
  regular:
      self: CubeStruct [1,1] - the object

  optional:

      subIndCVec:
        Case#1: numeric[1,]/numeric[,1]

        Case#2: cell[1,nDims]/cell[nDims,1] of double [nSubElem_i,1]
              for i=1,...,nDims

          -array of indices of field value slices that are selected
          to be returned; if not given (default),
          no indexation is performed

        Note!: numeric components of subIndVec are allowed to contain
           zeros which are be treated as they were references to null
           data slices

      dimVec: numeric[1,nDims]/numeric[nDims,1] - vector of dimension
```

```
          numbers corresponding to subIndCVec

  properties:

      fieldNameList: char[1,]/cell[1,nFields] of char[1,]
          list of field names to return

      structNameList: char[1,]/cell[1,nStructs] of char[1,]
          list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}

      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the cells,
              default value is false

      nullReplacements: cell[1,nReplacedFields]  - list of null
          replacements for each of the fields

      nullReplacementFields: cell[1,nReplacedFields] - list of fields in
         which the nulls are to be replaced with the specified values,
         if not specified it is assumed that all fields are to be
         replaced

         NOTE!: all fields not listed in this parameter are replaced with
         the default values

      checkInputs: logical[1,1] - true by default (input arguments are
         checked for correctness

Output:
  regular:
    SData: struct [1,1] - structure containing values of
        fields at the selected slices, each field is an array
        containing values of the corresponding type

    SIsNull: struct [1,1] - structure containing a nested
        array with is-null indicators for each CubeStruct cell content

    SIsValueNull: struct [1,1] - structure containing a
       logical array [] for each of the fields (true
       means that a corresponding cell doesn't not contain
          any value

GETELLARRAY - returns array of matrix's ellipsoid according to
              approxType

Input:
 regular:
    self.
    approxType:char[1,] - type of approximation(internal/external)

Output:
  apprEllMat:double[nDim1,..., nDimN] - array of array of ellipsoid's
           matrices

GETJOINWITH - returns a result of INNER join of given relation with
              another relation by the specified key fields
```

```
LIMITATION: key fields by which the join is peformed are required to form
a unique key in the given relation

Input:
  regular:
      self:
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT)
          'leftOuter'

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result

ISEQUAL - compares current relation object with other relation object and
          returns true if they are equal, otherwise it returns false


Usage: isEq=isEqual(self,otherObj)

Input:
  regular:
    self: ARelation [1,1] - current relation object
    otherObj: ARelation [1,1] - other relation object

  properties:
    checkFieldOrder/isFieldOrderCheck: logical [1,1] - if true, then fields
        in compared relations must be in the same order, otherwise the
        order is not  important (false by default)
    checkTupleOrder: logical[1,1] -  if true, then the tuples in the
        compared relations are expected to be in the same order,
        otherwise the order is not important (false by default)

    maxTolerance: double [1,1] - maximum allowed tolerance

    compareMetaDataBackwardRef: logical[1,1] if true, the CubeStruct's
        referenced from the meta data objects are also compared

    maxRelativeTolerance: double [1,1] - maximum allowed
    relative tolerance

Output:
  isEq: logical[1,1] - result of comparison
  reportStr: char[1,] - report of comparsion

PROJECT - computes projection of the relation object onto given time
          dependent subspase
Input:
  regular:
      self.
      projType: gras.ellapx.enums.EProjType[1,1] -
          type of the projection, can be
          'Static' and 'DynamicAlongGoodCurve'
      projMatList: cell[1,nProj] of double[nSpDim,nDim] - list of
          projection matrices, not necessarily orthogonal
```

```
   fGetProjMat: function_handle[1,1] - function which creates
      vector of the projection
            matrices
       Input:
        regular:
          projMat:double[nDim, mDim] - matrix of the projection at the
            instant of time
          timeVec:double[1, nDim] - time interval
        optional:
           sTime:double[1,1] - instant of time
       Output:
          projOrthMatArray:double[1, nSpDim] - vector of the projection
            matrices
          projOrthMatTransArray:double[nSpDim, 1] - transposed vector of
            the projection matrices
Output:
   ellTubeProjRel: gras.ellapx.smartdb.rels.EllTubeProj[1, 1]/
       gras.ellapx.smartdb.rels.EllTubeUnionProj[1, 1] -
          projected ellipsoidal tube

   indProj2OrigVec:cell[nDim, 1] - index of the line number from
            which is obtained the projection

Example:
  function example
   aMat = [0 1; 0 0]; bMat = eye(2);
   SUBounds = struct();
   SUBounds.center = {'sin(t)'; 'cos(t)'};
   SUBounds.shape = [9 0; 0 2];
   sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
   x0EllObj = ell_unitball(2);
   timeVec = [0 10];
   dirsMat = [1 0; 0 1]';
   rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
   ellTubeObj = rsObj.getEllTubeRel();
   unionEllTube = ...
    gras.ellapx.smartdb.rels.EllUnionTube.fromEllTubes(ellTubeObj);
   projMatList = {[1 0;0 1]};
   projType = gras.ellapx.enums.EProjType.Static;
   statEllTubeProj = unionEllTube.project(projType,projMatList,...
      @fGetProjMat);
   plObj=smartdb.disp.RelationDataPlotter();
   statEllTubeProj.plot(plObj);
end

function [projOrthMatArray,projOrthMatTransArray]=fGetProjMat(projMat,...
    timeVec,varargin)
  nTimePoints=length(timeVec);
  projOrthMatArray=repmat(projMat,[1,1,nTimePoints]);
  projOrthMatTransArray=repmat(projMat.',[1,1,nTimePoints]);
 end
```

9.9 gras.ellapx.smartdb.rels.EllUnionTubeStaticProj

```
EllUnionTubeStaticProj - class which keeps projection on static plane
                         union of ellipsoid tubes

Fields:
  QArray:cell[1, nElem] - Array of ellipsoid matrices
  aMat:cell[1, nElem] - Array of ellipsoid centers
  scaleFactor:double[1, 1] - Tube scale factor
  MArray:cell[1, nElem] - Array of regularization ellipsoid matrices
  dim :double[1, 1] - Dimensionality
  sTime:double[1, 1] - Time s
  approxSchemaName:cell[1,] - Name
  approxSchemaDescr:cell[1,] - Description
  approxType:gras.ellapx.enums.EApproxType - Type of approximation
                (external, internal, not defined
  timeVec:cell[1, m] - Time vector
  calcPrecision:double[1, 1] - Calculation precision
  indSTime:double[1, 1]  - index of sTime within timeVec
  ltGoodDirMat:cell[1, nElem] - Good direction curve
  lsGoodDirVec:cell[1, nElem] - Good direction at time s
  ltGoodDirNormVec:cell[1, nElem] - Norm of good direction curve
  lsGoodDirNorm:double[1, 1] - Norm of good direction at time s
  xTouchCurveMat:cell[1, nElem] - Touch point curve for good
                                  direction
  xTouchOpCurveMat:cell[1, nElem] - Touch point curve for direction
                                    opposite to good direction
  xsTouchVec:cell[1, nElem]  - Touch point at time s
  xsTouchOpVec :cell[1, nElem] - Touch point at time s
  projSTimeMat: cell[1, 1] - Projection matrix at time s
  projType:gras.ellapx.enums.EProjType - Projection type
  ltGoodDirNormOrigVec:cell[1, 1] - Norm of the original (not
                                    projected) good direction curve
  lsGoodDirNormOrig:double[1, 1] - Norm of the original (not
                                   projected)good direction at time s
  lsGoodDirOrigVec:cell[1, 1] - Original (not projected) good
                                direction at time s
  ellUnionTimeDirection:gras.ellapx.enums.EEllUnionTimeDirection -
                     Direction in time along which union is performed
  isLsTouch:logical[1, 1] - Indicates whether a touch takes place
                            along LS
  isLsTouchOp:logical[1, 1] - Indicates whether a touch takes place
                              along LS opposite
  isLtTouchVec:cell[1, nElem] - Indicates whether a touch takes place
                                along LT
  isLtTouchOpVec:cell[1, nElem] - Indicates whether a touch takes
                                  place along LT opposite
  timeTouchEndVec:cell[1, nElem] - Touch point curve for good
                                   direction
  timeTouchOpEndVec:cell[1, nElem] - Touch point curve for good
                                     direction

  TODO: correct description of the fields in
    gras.ellapx.smartdb.rels.EllUnionTubeStaticProj
```

See the description of the following methods in section smartdb.relations.ATypifiedStaticRelation for smartdb.relations.ATypifiedStaticRelation:

- smartdb.relations.ATypifiedStaticRelation.addData
- smartdb.relations.ATypifiedStaticRelation.addDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.addTuples
- smartdb.relations.ATypifiedStaticRelation.applyGetFunc
- smartdb.relations.ATypifiedStaticRelation.applySetFunc
- smartdb.relations.ATypifiedStaticRelation.applyTupleGetFunc
- smartdb.relations.ATypifiedStaticRelation.clearData
- smartdb.relations.ATypifiedStaticRelation.clone
- smartdb.relations.ATypifiedStaticRelation.copyFrom
- smartdb.relations.ATypifiedStaticRelation.createInstance
- smartdb.relations.ATypifiedStaticRelation.dispOnUI
- smartdb.relations.ATypifiedStaticRelation.display
- smartdb.relations.ATypifiedStaticRelation.fromStructList
- smartdb.relations.ATypifiedStaticRelation.getCopy
- smartdb.relations.ATypifiedStaticRelation.getFieldDescrList
- smartdb.relations.ATypifiedStaticRelation.getFieldIsNull
- smartdb.relations.ATypifiedStaticRelation.getFieldIsValueNull
- smartdb.relations.ATypifiedStaticRelation.getFieldNameList
- smartdb.relations.ATypifiedStaticRelation.getFieldProjection
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeList
- smartdb.relations.ATypifiedStaticRelation.getFieldTypeSpecList
- smartdb.relations.ATypifiedStaticRelation.getFieldValueSizeMat
- smartdb.relations.ATypifiedStaticRelation.getIsFieldValueNull
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionSize
- smartdb.relations.ATypifiedStaticRelation.getMinDimensionality
- smartdb.relations.ATypifiedStaticRelation.getNElems
- smartdb.relations.ATypifiedStaticRelation.getNFields
- smartdb.relations.ATypifiedStaticRelation.getNTuples
- smartdb.relations.ATypifiedStaticRelation.getSortIndex
- smartdb.relations.ATypifiedStaticRelation.getTuples
- smartdb.relations.ATypifiedStaticRelation.getTuplesFilteredBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesIndexedBy
- smartdb.relations.ATypifiedStaticRelation.getTuplesJoinedWith
- smartdb.relations.ATypifiedStaticRelation.getUniqueData
- smartdb.relations.ATypifiedStaticRelation.getUniqueDataAlongDim
- smartdb.relations.ATypifiedStaticRelation.getUniqueTuples

- smartdb.relations.ATypifiedStaticRelation.initByEmptyDataSet
- smartdb.relations.ATypifiedStaticRelation.initByDefaultDataSet
- smartdb.relations.ATypifiedStaticRelation.isFields
- smartdb.relations.ATypifiedStaticRelation.isMemberAlongDim
- smartdb.relations.ATypifiedStaticRelation.isMember
- smartdb.relations.ATypifiedStaticRelation.isUniqueKey
- smartdb.relations.ATypifiedStaticRelation.isEqual
- smartdb.relations.ATypifiedStaticRelation.removeDuplicateTuples
- smartdb.relations.ATypifiedStaticRelation.removeTuples
- smartdb.relations.ATypifiedStaticRelation.reorderData
- smartdb.relations.ATypifiedStaticRelation.saveObj
- smartdb.relations.ATypifiedStaticRelation.setData
- smartdb.relations.ATypifiedStaticRelation.setFieldInternal
- smartdb.relations.ATypifiedStaticRelation.sortBy
- smartdb.relations.ATypifiedStaticRelation.sortByAlongDim
- smartdb.relations.ATypifiedStaticRelation.toArray
- smartdb.relations.ATypifiedStaticRelation.toCell
- smartdb.relations.ATypifiedStaticRelation.toCellIsNull
- smartdb.relations.ATypifiedStaticRelation.toDispCell
- smartdb.relations.ATypifiedStaticRelation.toMat
- smartdb.relations.ATypifiedStaticRelation.toStruct
- **smartdb.relations.ATypifiedStaticRelation.unionWith_**
- smartdb.relations.ATypifiedStaticRelation.unionWithAlongDim
- smartdb.relations.ATypifiedStaticRelation.writeToCSV
- smartdb.relations.ATypifiedStaticRelation.writeToXLS

```
FROMELLTUBES - returns union of the ellipsoidal tubes on time

Input:
   ellTubeRel: smartdb.relation.StaticRelation[1, 1]/
      smartdb.relation.DynamicRelation[1, 1] - relation
      object

Output:
ellUnionTubeRel: ellapx.smartdb.rel.EllUnionTube - union of the
            ellipsoidal tubes
```

```
GETDATA - returns an indexed projection of CubeStruct object's content

Input:
  regular:
      self: CubeStruct [1,1] - the object
```

```
  optional:

      subIndCVec:
        Case#1: numeric[1,]/numeric[,1]

        Case#2: cell[1,nDims]/cell[nDims,1] of double [nSubElem_i,1]
              for i=1,...,nDims

          -array of indices of field value slices that are selected
          to be returned; if not given (default),
          no indexation is performed

        Note!: numeric components of subIndVec are allowed to contain
           zeros which are be treated as they were references to null
           data slices

      dimVec: numeric[1,nDims]/numeric[nDims,1] - vector of dimension
          numbers corresponding to subIndCVec

  properties:

      fieldNameList: char[1,]/cell[1,nFields] of char[1,]
          list of field names to return

      structNameList: char[1,]/cell[1,nStructs] of char[1,]
          list of internal structures to return (by default it
          is {SData, SIsNull, SIsValueNull}

      replaceNull: logical[1,1] if true, null values are replaced with
          certain default values uniformly across all the cells,
              default value is false

      nullReplacements: cell[1,nReplacedFields]  - list of null
          replacements for each of the fields

      nullReplacementFields: cell[1,nReplacedFields] - list of fields in
         which the nulls are to be replaced with the specified values,
         if not specified it is assumed that all fields are to be
         replaced

         NOTE!: all fields not listed in this parameter are replaced with
         the default values

      checkInputs: logical[1,1] - true by default (input arguments are
         checked for correctness

Output:
  regular:
    SData: struct [1,1] - structure containing values of
        fields at the selected slices, each field is an array
        containing values of the corresponding type

    SIsNull: struct [1,1] - structure containing a nested
        array with is-null indicators for each CubeStruct cell content

    SIsValueNull: struct [1,1] - structure containing a
       logical array [] for each of the fields (true
       means that a corresponding cell doesn't not contain
```

```
          any value

GETELLARRAY - returns array of matrix's ellipsoid according to
              approxType

Input:
 regular:
    self.
    approxType:char[1,] - type of approximation(internal/external)

Output:
  apprEllMat:double[nDim1,..., nDimN] - array of array of ellipsoid's
           matrices

GETJOINWITH - returns a result of INNER join of given relation with
              another relation by the specified key fields

LIMITATION: key fields by which the join is peformed are required to form
a unique key in the given relation

Input:
  regular:
      self:
      otherRel: smartdb.relations.ARelation[1,1]
      keyFieldNameList: char[1,]/cell[1,nFields] of char[1,]

  properties:
      joinType: char[1,] - type of join, can be
          'inner' (DEFAULT)
          'leftOuter'

Output:
  resRel: smartdb.relations.ARelation[1,1] - join result

GETREACHTUBEANEPREFIX - return prefix of the reach tube

Input:
  regular:
     self.

GETREGTUBEANEPREFIX - return prefix of the reg tube

Input:
  regular:
     self.

ISEQUAL - compares current relation object with other relation object and
          returns true if they are equal, otherwise it returns false


Usage: isEq=isEqual(self,otherObj)

Input:
  regular:
    self: ARelation [1,1] - current relation object
    otherObj: ARelation [1,1] - other relation object

  properties:
```

```
    checkFieldOrder/isFieldOrderCheck: logical [1,1] - if true, then fields
        in compared relations must be in the same order, otherwise the
        order is not  important (false by default)
    checkTupleOrder: logical[1,1] -  if true, then the tuples in the
        compared relations are expected to be in the same order,
        otherwise the order is not important (false by default)

    maxTolerance: double [1,1] - maximum allowed tolerance

    compareMetaDataBackwardRef: logical[1,1] if true, the CubeStruct's
        referenced from the meta data objects are also compared

    maxRelativeTolerance: double [1,1] - maximum allowed
    relative tolerance

Output:
  isEq: logical[1,1] - result of comparison
  reportStr: char[1,] - report of comparsion

PLOT - displays ellipsoidal tubes using the specified
  RelationDataPlotter

Input:
  regular:
      self:
  optional:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - plotter
          object used for displaying ellipsoidal tubes
  properties:
      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used

Output:
  plObj: smartdb.disp.RelationDataPlotter[1,1] - plotter
          object used for displaying ellipsoidal tubes
```

```
PLOTEXT - plots external approximation of ellTube.


Usage:
      obj.plotExt() - plots external approximation of ellTube.
      obj.plotExt('Property',PropValue,...) - plots external approximation
                                              of ellTube with setting
                                              properties.

Input:
  regular:
      obj:  EllTubeProj: EllTubeProj object
  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      colorSpec: char[1,1] - color specification code, can be 'r','g',
                 etc (any code supported by built-in Matlab function).

  properties:

      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used
      'showDiscrete':logical[1,1]  -
          if true, approximation in 3D will be filled in every time slice
      'nSpacePartPoins': double[1,1] -
          number of points in every time slice.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.

PLOTINT - plots internal approximation of ellTube.


Usage:
      obj.plotInt() - plots internal approximation of ellTube.
      obj.plotInt('Property',PropValue,...) - plots internal approximation
                                              of ellTube with setting
```

```
                                              properties.

Input:
  regular:
      obj:  EllTubeProj: EllTubeProj object
  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      colorSpec: char[1,1] - color specification code, can be 'r','g',
                   etc (any code supported by built-in Matlab function).

  properties:

      fGetColor: function_handle[1, 1] -
          function that specified colorVec for
          ellipsoidal tubes
      fGetAlpha: function_handle[1, 1] -
          function that specified transparency
          value for ellipsoidal tubes
      fGetLineWidth: function_handle[1, 1] -
          function that specified lineWidth for good curves
      fGetFill: function_handle[1, 1] - this
          property not used in this version
      colorFieldList: cell[nColorFields, ] of char[1, ] -
          list of parameters for color function
      alphaFieldList: cell[nAlphaFields, ] of char[1, ] -
          list of parameters for transparency function
      lineWidthFieldList: cell[nLineWidthFields, ]
          of char[1, ] - list of parameters for lineWidth
          function
      fillFieldList: cell[nIsFillFields, ] of char[1, ] -
          list of parameters for fill function
      plotSpecFieldList: cell[nPlotFields, ] of char[1, ] -
          defaul list of parameters. If for any function in
          properties not specified list of parameters,
          this one will be used
      'showDiscrete':logical[1,1]  -
          if true, approximation in 3D will be filled in every time slice
      'nSpacePartPoins': double[1,1] -
          number of points in every time slice.
Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.
```

9.10 elltool.reach.AReach

```
CUT - extracts the piece of reach tube from given start time to given
      end time. Given reach set self, find states that are reachable
      within time interval specified by cutTimeVec. If cutTimeVec
      is a scalar, then reach set at given time is returned.

Input:
  regular:
      self.

   cutTimeVec: double[1, 2]/double[1, 1] - time interval to cut.
```

```
Output:
  cutObj: elltool.reach.IReach[1, 1] - reach set resulting from the CUT
        operation.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  cutObj = rsObj.cut([3 5]);
  dRsObj = elltool.reach.ReachDiscrete(dtsys, x0EllObj, dirsMat, timeVec);
  dCutObj = dRsObj.cut([3 5]);

DIMENSION - returns array of dimensions of given reach set array.

Input:
  regular:
      self - multidimensional array of
             ReachContinuous/ReachDiscrete objects

Output:
  rSdimArr: double[nDim1, nDim2,...] - array of reach set dimensions.
  sSdimArr: double[nDim1, nDim2,...] - array of state space dimensions.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObjArr = rsObj.repMat(1,2);
  [rSdim sSdim] = rsObj.dimension()

  rSdim =

           2


  sSdim =

           2

  [rSdim sSdim] = rsObjArr.dimension()

  rSdim =
          [ 2  2 ]

  sSdim =
          [ 2  2 ]
```

```
DISPLAY - displays the reach set object.

Input:
  regular:
      self.

Output:
  None.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.display()

  rsObj =
  Reach set of the continuous-time linear system in R^2 in the time...
       interval [0, 10].

  Initial set at time t0 = 0:
  Ellipsoid with parameters
  Center:
       0
       0

  Shape Matrix:
       1     0
       0     1

  Number of external approximations: 2
  Number of internal approximations: 2

EVOLVE - computes further evolution in time of the
         already existing reach set.

Input:
  regular:
      self.

      newEndTime: double[1, 1] - new end time.

  optional:
      linSys: elltool.linsys.LinSys[1, 1] - new linear system.

Output:
  newReachObj: reach[1, 1] - reach set on time  interval
        [oldT0 newEndTime].

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
```

```
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  dRsObj = elltool.reach.ReachDiscrete(dsys, x0EllObj, dirsMat, timeVec);
  newRsObj = rsObj.evolve(12);
  newDRsObj = dRsObj.evolve(11);

GETABSTOL - gives the array of absTol for all elements
  in rsArr

Input:
  regular:
      rsArr: elltool.reach.AReach[nDim1, nDim2, ...] -
          multidimension array of reach sets
  optional:
      fAbsTolFun: function_handle[1,1] - function that is
          applied to the absTolArr. The default is @min.

Output:
  regular:
      absTolArr: double [absTol1, absTol2, ...] - return
          absTol for each element in rsArr
  optional:
      absTol: double[1,1] - return result of work fAbsTolFun
          with the absTolArr

Usage:
  use [~,absTol] = rsArr.getAbsTol() if you want get only
      absTol,
  use [absTolArr,absTol] = rsArr.getAbsTol() if you want
      get absTolArr and absTol,
  use absTolArr = rsArr.getAbsTol() if you want get only
      absTolArr

GETCOPY -

Input:
  regular:
      self:
  properties:
      l0Mat: double[nDims,nDirs] - matrix of good
          directions at time s
      isIntExtApxVec: logical[1,2] - two element vector with the
         first element corresponding to internal approximations
        and second - to external ones. An element equal to
         false means that the corresponding approximation type
         is filtered out. Default value is [true,true]

Example:
    aMat = [0 1; 0 0]; bMat = eye(2);
    SUBounds = struct();
    SUBounds.center = {'sin(t)'; 'cos(t)'};
    SUBounds.shape = [9 0; 0 2];
    sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
```

```
    x0EllObj = ell_unitball(2);
    timeVec = [0 10];
    dirsMat = [1 0; 0 1; 1 1;1 2]';
    rsObj = elltool.reach.ReachContinuous(sys, x0EllObj,...
      dirsMat, timeVec);

    copyRsObj = rsObj.getCopy()

    Reach set of the continuous-time linear system in R^2 in
      the time interval [0, 10].

    Initial set at time k0 = 0:
    Ellipsoid with parameters
    Center:
         0
         0

    Shape Matrix:
         1     0
         0     1

    Number of external approximations: 4
    Number of internal approximations: 4

    copyRsObj = rsObj.getCopy('l0Mat',[0;1],'approxType',...
      [true,false])

    Reach set of the continuous-time linear system in R^2 in
      the time interval [0, 10].

    Initial set at time k0 = 0:
    Ellipsoid with parameters
    Center:
         0
         0

    Shape Matrix:
         1     0
         0     1

    Number of external approximations: 1
    Number of internal approximations: 1

GET_EASCALEFACTOR - return the scale factor for external approximation
                    of reach tube

Input:
  regular:
      self.

Output:
  regular:
      eaScaleFactor: double[1, 1] - scale factor.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
```

```
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [10 0];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.getEaScaleFactor()

  ans =

      1.0200

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.getEllTubeRel();

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  getEllTubeUnionRel(rsObj);

GET_IASCALEFACTOR - return the scale factor for internal approximation
                    of reach tube

Input:
  regular:
      self.

Output:
  regular:
      iaScaleFactor: double[1, 1] - scale factor.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [10 0];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.getIaScaleFactor()
```

```
  ans =

      1.0200

GETINITIALSET - return the initial set for linear system, which is solved
                for building reach tube.

Input:
  regular:
      self.

Output:
  regular:
      x0Ell: ellipsoid[1, 1] - ellipsoid x0, which was initial set for
          linear system.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [10 0];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  x0Ell = rsObj.getInitialSet()

  x0Ell =

  Center:
       0
       0

  Shape Matrix:
       1     0
       0     1

  Nondegenerate ellipsoid in R^2.

GETNPLOT2DPOINTS - gives array  the same size as rsArr of
  value of nPlot2dPoints property for each element in rsArr -
  array of reach sets

Input:
  regular:
    rsArr:elltool.reach.AReach[nDims1,nDims2,...] - reach
      set array

Output:
  nPlot2dPointsArr:double[nDims1,nDims2,...] - array of
      values of nTimeGridPoints property for each reach set
      in rsArr

GETNPLOT3DPOINTS - gives array  the same size as rsArr of
  value of nPlot3dPoints property for each element in rsArr
  array of reach sets
```

```
Input:
  regular:
      rsArr:reach[nDims1,nDims2,...] - reach set array

Output:
  nPlot3dPointsArr:double[nDims1,nDims2,...]- array of values
      of nPlot3dPoints property for each reach set in rsArr

GETNTIMEGRIDPOINTS - gives array  the same size as rsArr of
  value of nTimeGridPoints property for each element in rsArr
  array of reach sets

Input:
  regular:
      rsArr: elltool.reach.AReach [nDims1,nDims2,...] - reach
          set array

Output:
  nTimeGridPointsArr: double[nDims1,nDims2,...]- array of
      values of nTimeGridPoints property for each reach set
      in rsArr

GETRELTOL - gives the array of relTol for all elements in
ellArr

Input:
  regular:
      rsArr: elltool.reach.AReach[nDim1,nDim2, ...] -
          multidimension array of reach sets.
  optional
      fRelTolFun: function_handle[1,1] - function that is
          applied to the relTolArr. The default is @min.

Output:
  regular:
      relTolArr: double [relTol1, relTol2, ...] - return
          relTol for each element in rsArr.
  optional:
      relTol: double[1,1] - return result of work fRelTolFun
          with the relTolArr

Usage:
  use [~,relTol] = rsArr.getRelTol() if you want get only
      relTol,
  use [relTolArr,relTol] = rsArr.getRelTol() if you want get
      relTolArr and relTol,
  use relTolArr = rsArr.getRelTol() if you want get only
      relTolArr

GET_CENTER - returns the trajectory of the center of the reach set.

Input:
  regular:
      self.

Output:
  trCenterMat: double[nDim, nPoints] - array of points that form the
      trajectory of the reach set center, where nDim is reach set
```

```
      dimentsion, nPoints - number of points in time grid.

  timeVec: double[1, nPoints] - array of time values.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  [trCenterMat timeVec] = rsObj.get_center();

GET_DIRECTIONS - returns the values of direction vectors for time grid
                 values.

Input:
  regular:
      self.

Output:
  directionsCVec: cell[1, nPoints] of double [nDim, nDir] - array of
      cells, where each cell is a sequence of direction vector values
      that correspond to the time values of the grid, where nPoints is
      number of points in time grid.

  timeVec: double[1, nPoints] - array of time values.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  [directionsCVec timeVec] = rsObj.get_directions();

GET_EA - returns array of ellipsoid objects representing external
         approximation of the reach  tube.

Input:
  regular:
      self.

Output:
  eaEllMat: ellipsoid[nAppr, nPoints] - array of ellipsoids, where nAppr
      is the number of approximations, nPoints is number of points in time
      grid.

   timeVec: double[1, nPoints] - array of time values.
   l0Mat: double[nDirs,nDims] - matrix of good directions at t0
```

```
Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  [eaEllMat timeVec] = rsObj.get_ea();

  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dRsObj = elltool.reach.ReachDiscrete(sys, x0EllObj, dirsMat, timeVec);
  [eaEllMat timeVec] = dRsObj.get_ea();

GET_GOODCURVES - returns the 'good curve' trajectories of the reach set.

Input:
  regular:
      self.

Output:
  goodCurvesCVec: cell[1, nPoints] of double [x, y] - array of cells,
      where each cell is array of points that form a 'good curve'.

  timeVec: double[1, nPoints] - array of time values.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  [goodCurvesCVec timeVec] = rsObj.get_goodcurves();

  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dRsObj = elltool.reach.ReachDiscrete(sys, x0EllObj, dirsMat, timeVec);
  [goodCurvesCVec timeVec] = dRsObj.get_goodcurves();

GET_IA - returns array of ellipsoid objects representing internal
         approximation of the  reach tube.

Input:
  regular:
      self.

Output:
  iaEllMat: ellipsoid[nAppr, nPoints] - array of ellipsoids, where nAppr
      is the number of approximations, nPoints is number of points in time
      grid.

  timeVec: double[1, nPoints] - array of time values.
```

```
Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  [iaEllMat timeVec] = rsObj.get_ia();

GET_SYSTEM - returns the linear system for which the reach set is
             computed.

Input:
  regular:
      self.

Output:
  linSys: elltool.linsys.LinSys[1, 1] - linear system object.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  linSys = rsObj.get_system()

  self =
  A:
       0     1
       0     0


  B:
       1     0
       0     1


  Control bounds:
     2-dimensional ellipsoid with center
      'sin(t)'
      'cos(t)'

     and shape matrix
       9     0
       0     2


  C:
       1     0
       0     1
```

```
  2-input, 2-output continuous-time linear time-invariant system of
          dimension 2:
  dx/dt  =  A x(t)  +  B u(t)
   y(t)  =  C x(t)

  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dRsObj = elltool.reach.ReachDiscrete(sys, x0EllObj, dirsMat, timeVec);
  dRsObj.get_system();

INTERSECT - checks if its external (s = 'e'), or internal (s = 'i')
            approximation intersects with given ellipsoid, hyperplane
            or polytop.

Input:
  regular:
      self.

      intersectObj: ellipsoid[1, 1]/hyperplane[1,1]/polytop[1, 1].

      approxTypeChar: char[1, 1] - 'e' (default) - external approximation,
                                   'i' - internal approximation.

Output:
  isEmptyIntersect: logical[1, 1] -  true - if intersection is nonempty,
                                     false - otherwise.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  ellObj = ellipsoid([0; 0], 2*eye(2));
  isEmptyIntersect = intersect(rsObj, ellObj)

  isEmptyIntersect =

                   1

ISEMPTY - checks if given reach set array is an array of empty objects.

Input:
  regular:
      self - multidimensional array of
             ReachContinuous/ReachDiscrete objects

Output:
  isEmptyArr: logical[nDim1, nDim2, nDim3,...] -
              isEmpty(iDim1, iDim2, iDim3,...) = true - if self(iDim1, iDim2, iDim3,...) is empty,
                                               = false - otherwise.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
```

```
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  dRsObj = elltool.reach.ReachRiscrete(dsys, x0EllObj, dirsMat, timeVec);
  rsObjArr = rsObj.repMat(1,2);
  dRsObjArr = dRsObj.repMat(1,2);
  dRsObj.isEmpty();
  rsObj.isEmpty()

  ans =

       0

  dRsObjArr.isEmpty();
  rsObjArr.isEmpty()

  ans =
      [ 0  0 ]

ISEQUAL - checks for equality given reach set objects

Input:
  regular:
      self.
      reachObj:
          elltool.reach.AReach[1, 1] - each set object, which
           compare with self.
  optional:
      indTupleVec: double[1,] - tube numbers that are
          compared
      approxType: gras.ellapx.enums.EApproxType[1, 1] -  type of
          approximation, which will be compared.
  properties:
      notComparedFieldList: cell[1,k] - fields not to compare
          in tubes. Default: LT_GOOD_DIR_*, LS_GOOD_DIR_*,
          IND_S_TIME, S_TIME, TIME_VEC
      areTimeBoundsCompared: logical[1,1] - treat tubes with
          different timebounds as inequal if 'true'.
          Default: false

Output:
  regular:
      ISEQUAL: logical[1, 1] - true - if reach set objects are equal.
          false - otherwise.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
```

```
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  copyRsObj = rsObj.getCopy();
  isEqual = isEqual(rsObj, copyRsObj)

  isEqual =

          1

ISBACKWARD - checks if given reach set object was obtained by solving
             the system in reverse time.

Input:
  regular:
      self.

Output:
  regular:
      isBackward: logical[1, 1] - true - if self was obtained by solving
          in reverse time, false - otherwise.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [10 0];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.isbackward()

  ans =

       1

ISCUT - checks if given array of reach set objects is a cut of
        another reach set object's array.

Input:
  regular:
      self - multidimensional array of
             ReachContinuous/ReachDiscrete objects

Output:
  isCutArr: logical[nDim1, nDim2, nDim3 ...] -
            isCut(iDim1, iDim2, iDim3,..) = true - if self(iDim1, iDim2, iDim3,...) is a cut of the r
                                          = false - otherwise.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
```

```
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  dRsObj = elltool.reach.ReachRiscrete(dsys, x0EllObj, dirsMat, timeVec);
  cutObj = rsObj.cut([3 5]);
  cutObjArr = cutObj.repMat(2,3,4);
  iscut(cutObj);
  iscut(cutObjArr);
  cutObj = dRsObj.cut([4 8]);
  cutObjArr = cutObj.repMat(1,2);
  iscut(cutObjArr);
  iscut(cutObj);

ISPROJECTION - checks if given array of reach set objects is projections.

Input:
  regular:
      self - multidimensional array of
             ReachContinuous/ReachDiscrete objects

Output:
  isProjArr: logical[nDim1, nDim2, nDim3, ...] -
             isProj(iDim1, iDim2, iDim3,...) = true - if self(iDim1, iDim2, iDim3,...) is projection,
                                             = false - otherwise.


Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  dRsObj = elltool.reach.ReachRiscrete(dsys, x0EllObj, dirsMat, timeVec);
  projMat = eye(2);
  projObj = rsObj.projection(projMat);
  projObjArr = projObj.repMat(3,2,2);
  isprojection(projObj);
  isprojection(projObjArr);
  projObj = dRsObj.projection(projMat);
  projObjArr = projObj.repMat(1,2);
  isprojection(projObj);
  isprojection(projObjArr);

plotByEa - plots external approximation of reach tube.


Usage:
      plotByEa(self,'Property',PropValue,...)
      - plots external approximation of reach tube
           with  setting properties

Input:
  regular:
```

```
      self: - reach tube

  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      charColor: char[1,1]  - color specification code, can be 'r','g',
                     etc (any code supported by built-in Matlab function).
  properties:

      'fill': logical[1,1]  -
              if 1, tube in 2D will be filled with color.
              Default value is true.
      'lineWidth': double[1,1]  -
                   line width for 2D plots. Default value is 2.
      'color': double[1,3] -
               sets default colors in the form [x y z].
                  Default value is [0 0 1].
      'shade': double[1,1]  -
     level of transparency between 0 and 1 (0 - transparent, 1 - opaque).
               Default value is 0.3.

Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
      data plotter object.
```

```
plotByIa - plots internal approximation of reach tube.


Usage:
      plotByIa(self,'Property',PropValue,...)
      - plots internal approximation of reach tube
           with  setting properties

Input:
  regular:
      self: - reach tube

  optional:
      relDataPlotter:smartdb.disp.RelationDataPlotter[1,1] - relation data plotter object.
      charColor: char[1,1]  - color specification code, can be 'r','g',
                     etc (any code supported by built-in Matlab function).
  properties:

      'fill': logical[1,1]  -
              if 1, tube in 2D will be filled with color.
              Default value is true.
      'lineWidth': double[1,1]  -
                   line width for 2D plots. Default value is 2.
      'color': double[1,3] -
               sets default colors in the form [x y z].
                  Default value is [0 1 0].
      'shade': double[1,1]  -
     level of transparency between 0 and 1 (0 - transparent, 1 - opaque).
               Default value is 0.1.

Output:
  regular:
      plObj: smartdb.disp.RelationDataPlotter[1,1] - returns the relation
```

```
      data plotter object.

PLOT_EA - plots external approximations of 2D and 3D reach sets.

Input:
  regular:
      self.

  optional:
      colorSpec: char[1, 1] - set color to plot in following way:
                             'r' - red color,
                             'g' - green color,
                             'b' - blue color,
                             'y' - yellow color,
                             'c' - cyan color,
                             'm' - magenta color,
                             'w' - white color.

      OptStruct: struct[1, 1] with fields:
          color: double[1, 3] - sets color of the picture in the form
                [x y z].
          width: double[1, 1] - sets line width for 2D plots.
          shade: double[1, 1] in [0; 1] interval - sets transparency level
                (0 - transparent, 1 - opaque).
           fill: double[1, 1] - if set to 1, reach set will be filled with
                 color.

Output:
  None.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.plotEa();
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);

  dRsObj = elltool.reach.ReachDiscrete(sys, x0EllObj, dirsMat, timeVec);
  dRsObj.plotEa();

PLOTIA - plots internal approximations of 2D and 3D reach sets.

Input:
  regular:
      self.

  optional:
      colorSpec: char[1, 1] - set color to plot in following way:
                             'r' - red color,
                             'g' - green color,
                             'b' - blue color,
                             'y' - yellow color,
```

```
                             'c' - cyan color,
                             'm' - magenta color,
                             'w' - white color.

      OptStruct: struct[1, 1] with fields:
          color: double[1, 3] - sets color of the picture in the form
                [x y z].
          width: double[1, 1] - sets line width for 2D plots.
          shade: double[1, 1] in [0; 1] interval - sets transparency level
                (0 - transparent, 1 - opaque).
           fill: double[1, 1] - if set to 1, reach set will be filled with
                color.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj.plotIa();
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dRsObj = elltool.reach.ReachDiscrete(sys, x0EllObj, dirsMat, timeVec);
  dRsObj.plotIa();

REFINE - adds new approximations computed for the specified directions
         to the given reach set or to the projection of reach set.

Input:
  regular:
      self.
      l0Mat: double[nDim, nDir] - matrix of directions for new
          approximation

Output:
  regular:
      reachObj: reach[1,1] - refine reach set for the directions
          specified in l0Mat

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  newDirsMat = [1; -1];
  rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
  rsObj = rsObj.refine(newDirsMat);

REPMAT - is analogous to built-in repmat function with one exception - it
         copies the objects, not just the handles
```

```
Input:
  regular:
      self.

Output:
  Array of given ReachContinuous/ReachDiscrete object’s copies.

 Example:
   aMat = [0 1; 0 0]; bMat = eye(2);
   SUBounds = struct();
   SUBounds.center = {’sin(t)’; ’cos(t)’};
   SUBounds.shape = [9 0; 0 2];
   sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
   x0EllObj = ell_unitball(2);
   timeVec = [0 10];
   dirsMat = [1 0; 0 1]’;
   reachObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
   reachObjArr = reachObj.repMat(1,2);

   reachObjArr = 1x2 array of ReachContinuous objects
```

9.11 elltool.reach.ReachContinuous

```
ReachContinuous - computes reach set approximation of the continuous
    linear system for the given time interval.
Input:
    regular:
      linSys: elltool.linsys.LinSys object -
          given linear system .
      x0Ell: ellipsoid[1, 1] - ellipsoidal set of
          initial conditions.
      l0Mat: double[nRows, nColumns] - initial good directions
          matrix.
      timeVec: double[1, 2] - time interval.

    properties:
      isRegEnabled: logical[1, 1] - if it is ’true’ constructor
          is allowed to use regularization.
      isJustCheck: logical[1, 1] - if it is ’true’ constructor
          just check if square matrices are degenerate, if it is
          ’false’ all degenerate matrices will be regularized.
      regTol: double[1, 1] - regularization precision.

Output:
  regular:
    self - reach set object.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {’sin(t)’; ’cos(t)’};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]’;
```

```
rsObj = elltool.reach.ReachContinuous(sys, x0EllObj, dirsMat, timeVec);
```

See the description of the following methods in section elltool.reach.AReach for elltool.reach.AReach:

- elltool.reach.AReach.cut
- elltool.reach.AReach.dimension
- elltool.reach.AReach.display
- elltool.reach.AReach.evolve
- elltool.reach.AReach.getAbsTol
- elltool.reach.AReach.getCopy
- elltool.reach.AReach.getEaScaleFactor
- **elltool.reach.AReach.getEllTubeRel_**?
- **elltool.reach.AReach.getEllTubeUnionRel_**?
- elltool.reach.AReach.getIaScaleFactor
- elltool.reach.AReach.getInitialSet
- elltool.reach.AReach.getNPlot2dPoints
- elltool.reach.AReach.getNPlot3dPoints
- elltool.reach.AReach.getNTimeGridPoints
- elltool.reach.AReach.getRelTol
- **elltool.reach.AReach.getSwitchTimeVec_** ?
- elltool.reach.AReach.get_center
- elltool.reach.AReach.get_directions
- elltool.reach.AReach.get_ea
- elltool.reach.AReach.get_goodcurves
- elltool.reach.AReach.get_ia
- elltool.reach.AReach.get_system
- elltool.reach.AReach.intersect
- elltool.reach.AReach.isEmpty
- elltool.reach.AReach.isEqual
- elltool.reach.AReach.isbackward
- elltool.reach.AReach.iscut
- elltool.reach.AReach.isprojection
- elltool.reach.AReach.plotByEa
- elltool.reach.AReach.plotByIa
- elltool.reach.AReach.plotEa
- elltool.reach.AReach.plotIa
- **elltool.reach.AReach.projection_** ?
- elltool.reach.AReach.refine

- elltool.reach.AReach.repMat

9.12 elltool.reach.ReachDiscrete

```
ReachDiscrete - computes reach set approximation of the discrete linear
                system for the given time interval.


Input:
    linSys: elltool.linsys.LinSys object - given linear system
    x0Ell: ellipsoid[1, 1] - ellipsoidal set of initial conditions
    l0Mat: double[nRows, nColumns] - initial good directions
          matrix.
    timeVec: double[1, 2] - time interval
    properties:
      isRegEnabled: logical[1, 1] - if it is 'true' constructor
          is allowed to use regularization.
      isJustCheck: logical[1, 1] - if it is 'true' constructor
          just check if square matrices are degenerate, if it is
          'false' all degenerate matrices will be regularized.
      regTol: double[1, 1] - regularization precision.
      minmax: logical[1, 1] - field, which:
          = 1 compute minmax reach set,
          = 0 (default) compute maxmin reach set.

Output:
  regular:
    self - reach set object.
Example:
  adMat = [0 1; -1 -0.5];
  bdMat = [0; 1];
  udBoundsEllObj  = ellipsoid(1);
  dtsys = elltool.linsys.LinSysDiscrete(adMat, bdMat, udBoundsEllObj);
  x0EllObj = ell_unitball(2);
  timeVec = [0 10];
  dirsMat = [1 0; 0 1]';
  dRsObj = elltool.reach.ReachDiscrete(dtsys, x0EllObj, dirsMat, timeVec);
```

See the description of the following methods in section elltool.reach.AReach for elltool.reach.AReach:

- elltool.reach.AReach.cut
- elltool.reach.AReach.dimension
- elltool.reach.AReach.display
- elltool.reach.AReach.evolve
- elltool.reach.AReach.getAbsTol
- elltool.reach.AReach.getCopy
- elltool.reach.AReach.getEaScaleFactor
- **elltool.reach.AReach.getEllTubeRel_**?
- **elltool.reach.AReach.getEllTubeUnionRel_**?
- elltool.reach.AReach.getIaScaleFactor
- elltool.reach.AReach.getInitialSet

- elltool.reach.AReach.getNPlot2dPoints
- elltool.reach.AReach.getNPlot3dPoints
- elltool.reach.AReach.getNTimeGridPoints
- elltool.reach.AReach.getRelTol
- **elltool.reach.AReach.getSwitchTimeVec_** ?
- elltool.reach.AReach.get_center
- elltool.reach.AReach.get_directions
- elltool.reach.AReach.get_ea
- elltool.reach.AReach.get_goodcurves
- elltool.reach.AReach.get_ia
- elltool.reach.AReach.get_system
- elltool.reach.AReach.intersect
- elltool.reach.AReach.isEmpty
- elltool.reach.AReach.isEqual
- elltool.reach.AReach.isbackward
- elltool.reach.AReach.iscut
- elltool.reach.AReach.isprojection
- elltool.reach.AReach.plotByEa
- elltool.reach.AReach.plotByIa
- elltool.reach.AReach.plotEa
- elltool.reach.AReach.plotIa
- **elltool.reach.AReach.projection_** ?
- elltool.reach.AReach.refine
- elltool.reach.AReach.repMat

9.13 elltool.reach.ReachFactory

```
Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);

Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  reachObj = rsObj.createInstance();
```

```
Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  dim = rsObj.getDim();

Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  l0Mat = rsObj.getL0Mat()

  l0Mat =

       1     0
       0     1

Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  linSys = rsObj.getLinSys();

Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  tVec = rsObj.getTVec()

  tVec =

       0    10

Example:
  import elltool.reach.ReachFactory;
  crm=gras.ellapx.uncertcalc.test.regr.conf.ConfRepoMgr();
  crmSys=gras.ellapx.uncertcalc.test.regr.conf.sysdef.ConfRepoMgr();
  rsObj =  ReachFactory('demo3firstTest', crm, crmSys, false, false);
  X0Ell = rsObj.getX0Ell()

  X0Ell =

  Center:
       0
       0

  Shape Matrix:
      0.0100         0
           0    0.0100

  Nondegenerate ellipsoid in R^2.
```

9.14 elltool.linsys.ALinSys

```
ALinSys - constructor abstract class of linear system.

Continuous-time linear system:
          dx/dt  =  A(t) x(t)  +  B(t) u(t)  +  C(t) v(t)

Discrete-time linear system:
          x[k+1]  =  A[k] x[k]  +  B[k] u[k]  +  C[k] v[k]

Input:
  regular:
      atInpMat: double[nDim, nDim]/cell[nDim, nDim] - matrix A.

      btInpMat: double[nDim, kDim]/cell[nDim, kDim] - matrix B.

      uBoundsEll: ellipsoid[1, 1]/struct[1, 1] - control bounds
          ellipsoid.

      ctInpMat: double[nDim, lDim]/cell[nDim, lDim] - matrix G.

      vBoundsEll: ellipsoid[1, 1]/struct[1, 1] - disturbance bounds
          ellipsoid.
      discrFlag: char[1, 1] - if discrFlag set:
          'd' - to discrete-time linSys
          not 'd' - to continuous-time linSys.

Output:
  self: elltool.linsys.ALinSys[1, 1] -
      linear system.

DIMENSION - returns dimensions of state, input, output and disturbance
            spaces.
Input:
  regular:
      self: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of
            linear systems.

Output:
  stateDimArr: double[nDims1, nDims2,...] - array of state space
      dimensions.

  inpDimArr: double[nDims1, nDims2,...] - array of input dimensions.

  distDimArr: double[nDims1, nDims2,...] - array of disturbance
        dimensions.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  [stateDimArr, inpDimArr, outDimArr, distDimArr] = sys.dimension()

  stateDimArr =

       2
```

```
  inpDimArr =

       2



  distDimArr =

       0

  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dsys.dimension();

DISPLAY - displays the details of linear system object.

Input:
  regular:
      self: elltool.linsys.ALinSys[1, 1] - linear system.

Output:
  None.

GETABSTOL - gives array the same size as linsysArr with values of absTol
            properties for each hyperplane in hplaneArr.


Input:
  regular:
      self: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of linear
            systems.

Output:
  absTolArr: double[nDims1, nDims2,...] - array of absTol properties for
      linear systems in self.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  sys.getAbsTol();
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dsys.getAbsTol();

GETATMAT -

Input:
  regular:
      self: elltool.linsys.ILinSys[1, 1] - linear system.

Output:
  aMat: double[aMatDim, aMatDim]/cell[nDim, nDim] - matrix A.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
```

```
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  aMat = dsys.getAtMat();

Input:
  regular:
      self: elltool.linsys.ILinSys[1, 1] - linear system.

Output:
  bMat: double[bMatDim, bMatDim]/cell[bMatDim, bMatDim] - matrix B.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  bMat = dsys.getBtMat();

GETCOPY - gives array the same size as linsysArr with with copies of
          elements of self.

Input:
  regular:
      self: elltool.linsys.ALinSys[nDims1, nDims2,...] - an array of
            linear systems.

Output:
  copyLinSysArr: elltool.linsys.LinSys[nDims1, nDims2,...] -  an array of
     copies of elements of self.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  newSys = sys.getCopy();
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  newDSys = dsys.getCopy();

Input:
  regular:
      self: elltool.linsys.ILinSys[1, 1] - linear system.

Output:
  cMat: double[cMatDim, cMatDim]/cell[cMatDim, cMatDim] - matrix C.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
```

```
  gMat = sys.getCtMat();

GETDISTBOUNDSELL -

Input:
  regular:
      self: elltool.linsys.ILinSys[1, 1] - linear system.

Output:
  distEll: ellipsoid[1, 1]/struct[1, 1] - disturbance bounds ellipsoid.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  distEll = sys.getDistBoundsEll();

Input:
  regular:
      self: elltool.linsys.ILinSys[1, 1] - linear system.

Output:
  uEll: ellipsoid[1, 1]/struct[1, 1] - control bounds ellipsoid.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  uEll = dsys.getUBoundsEll();

HASDISTURBANCE - returns true if system has disturbance

Input:
  regular:
      self: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of
            linear systems.
  optional:
      isMeaningful: logical[1,1] - if true(default), treat constant
                    disturbance vector as absence of disturbance

Output:
  isDisturbanceArr: logical[nDims1, nDims2,...] - array such that it's
      element at each position is true if corresponding linear system
      has disturbance, and false otherwise.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  sys.hasDisturbance()
```

```
  ans =

       0
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dsys.hasDisturbance();

ISEMPTY - checks if linear system is empty.

Input:
  regular:
      self: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of linear
            systems.

Output:
  isEmptyMat: logical[nDims1, nDims2,...] - array such that it's element at
      each position is true if corresponding linear system is empty, and
      false otherwise.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  sys.isEmpty()

  ans =

       0
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  dsys.isEmpty();

ISEQUAL - produces produces logical array the same size as
          self/compLinSysArr (if they have the same).
          isEqualArr[iDim1, iDim2,...] is true if corresponding
          linear systems are equal and false otherwise.

Input:
  regular:
      self: elltool.linsys.ILinSys[nDims1, nDims2,...] -  an array of
           linear systems.
      compLinSysArr: elltool.linsys.ILinSys[nDims1,...nDims2,...] - an
           array of linear systems.

Output:
  isEqualArr: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of
      logical values.
      isEqualArr[iDim1, iDim2,...] is true if corresponding linear systems
      are equal and false otherwise.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  newSys = sys.getCopy();
  isEqual = sys.isEqual(newSys)
```

```
  isEqual =

       1
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  newDSys = sys.getCopy();
  isEqual = dsys.isEqual(newDSys)

  isEqual =

       1
```

```
ISLTI - checks if linear system is time-invariant.

Input:
  regular:
      self: elltool.linsys.LinSys[nDims1, nDims2,...] - an array of linear
            systems.

Output:
  isLtiMat: logical[nDims1, nDims2,...] -array such that it's element at
      each position is true if corresponding linear system is
      time-invariant, and false otherwise.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
  isLtiArr = sys.isLti();
  dsys = elltool.linsys.LinSysDiscrete(aMat, bMat, SUBounds);
  isLtiArr = dsys.isLti();
```

9.15 elltool.linsys.LinSysContinuous

```
LINSYSCONTINUOUS - Constructor of continuous linear system object.

Continuous-time linear system:
          dx/dt  =  A(t) x(t)  +  B(t) u(t)  +  C(t) v(t)

Input:
  regular:
      atInpMat: double[nDim, nDim]/cell[nDim, nDim] - matrix A.

      btInpMat: double[nDim, kDim]/cell[nDim, kDim] - matrix B.

  optional:
      uBoundsEll: ellipsoid[1, 1]/struct[1, 1] - control bounds
            ellipsoid.

      ctInpMat: double[nDim, lDim]/cell[nDim, lDim] - matrix G.

      distBoundsEll: ellipsoid[1, 1]/struct[1, 1] - disturbance
            bounds ellipsoid.
```

```
    discrFlag: char[1, 1] - if discrFlag set:
            'd' - to discrete-time linSys,
            not 'd' - to continuous-time linSys.


Output:
  self: elltool.linsys.LinSysContinuous[1, 1] - continuous linear
            system.

Example:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysContinuous(aMat, bMat, SUBounds);
```

See the description of the following methods in section elltool.linsys.ALinSys for elltool.linsys.ALinSys:

- elltool.linsys.ALinSys.dimension
- elltool.linsys.ALinSys.display
- elltool.linsys.ALinSys.getAbsTol
- elltool.linsys.ALinSys.getAtMat
- elltool.linsys.ALinSys.getBtMat
- elltool.linsys.ALinSys.getCopy
- elltool.linsys.ALinSys.getCtMat
- elltool.linsys.ALinSys.getDistBoundsEll
- elltool.linsys.ALinSys.getUBoundsEll
- elltool.linsys.ALinSys.hasDisturbance
- elltool.linsys.ALinSys.isEmpty
- elltool.linsys.ALinSys.isEqual
- elltool.linsys.ALinSys.isLti

9.16 elltool.linsys.LinSysDiscrete

```
LINSYSDISCRETE - constructor of discrete linear system object.

Discrete-time linear system:
          x[k+1]  =  A[k] x[k]  +  B[k] u[k]  +  C[k] v[k]

Input:
  regular:
    atInpMat: double[nDim, nDim]/cell[nDim, nDim] - matrix A.

    btInpMat: double[nDim, kDim]/cell[nDim, kDim] - matrix B.
  optional:
    uBoundsEll: ellipsoid[1, 1]/struct[1, 1] - control bounds
        ellipsoid.

    ctInpMat: double[nDim, lDim]/cell[nDim, lDim] - matrix G.
```

```
      distBoundsEll: ellipsoid[1, 1]/struct[1, 1] - disturbance bounds
          ellipsoid.

      discrFlag: char[1, 1] - if discrFlag set:
           'd' - to discrete-time linSys
           not 'd' - to continuous-time linSys.

Output:
  self: elltool.linsys.LinSysDiscrete[1, 1] - discrete linear system.

Example:
  for k = 1:20
     atMat = {'0' '1 + cos(pi*k/2)'; '-2' '0'};
     btMat =  [0; 1];
     uBoundsEllObj = ellipsoid(4);
     ctMat = [1; 0];
     distBounds = 1/(k+1);
     lsys = elltool.linsys.LinSysDiscrete(atMat, btMat,...
         uBoundsEllObj, ctMat,distBounds);
  end
```

See the description of the following methods in section elltool.linsys.ALinSys for elltool.linsys.ALinSys:

- elltool.linsys.ALinSys.dimension
- elltool.linsys.ALinSys.display
- elltool.linsys.ALinSys.getAbsTol
- elltool.linsys.ALinSys.getAtMat
- elltool.linsys.ALinSys.getBtMat
- elltool.linsys.ALinSys.getCopy
- elltool.linsys.ALinSys.getCtMat
- elltool.linsys.ALinSys.getDistBoundsEll
- elltool.linsys.ALinSys.getUBoundsEll
- elltool.linsys.ALinSys.hasDisturbance
- elltool.linsys.ALinSys.isEmpty
- elltool.linsys.ALinSys.isEqual
- elltool.linsys.ALinSys.isLti

9.17 elltool.linsys.LinSysFactory

```
Factory class of linear system objects of the Ellipsoidal Toolbox.

CREATE - returns linear system object.

Continuous-time linear system:
          dx/dt  =  A(t) x(t)  +  B(t) u(t)  +  C(t) v(t)

Discrete-time linear system:
          x[k+1]  =  A[k] x[k]  +  B[k] u[k]  +  C[k] v[k]
```

```
Input:
  regular:
      atInpMat: double[nDim, nDim]/cell[nDim, nDim] - matrix A.

      btInpMat: double[nDim, kDim]/cell[nDim, kDim] - matrix B.

      uBoundsEll: ellipsoid[1, 1]/struct[1, 1] - control bounds
          ellipsoid.

      ctInpMat: double[nDim, lDim]/cell[nDim, lDim] - matrix G.

      distBoundsEll: ellipsoid[1, 1]/struct[1, 1] - disturbance bounds
          ellipsoid.

      discrFlag: char[1, 1] - if discrFlag set:
          'd' - to discrete-time linSys
          not 'd' - to continuous-time linSys.

Output:
  linSys: elltool.linsys.LinSysContinuous[1, 1]/
      elltool.linsys.LinSysDiscrete[1, 1] - linear system.

Examples:
  aMat = [0 1; 0 0]; bMat = eye(2);
  SUBounds = struct();
  SUBounds.center = {'sin(t)'; 'cos(t)'};
  SUBounds.shape = [9 0; 0 2];
  sys = elltool.linsys.LinSysFactory.create(aMat, bMat,SUBounds);
```

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*