bensolve tools

Calculus of Convex Polyhedra

Calculus of Polyhedral Convex Functions

Global Optimization

Vector Linear Programming

for Octave and Matlab

April 26, 2019

Supported by German Research Foundation, grant number LO 1379/7-1

Contents

1. Overview

The package *bensolve tools* contains the following components:

- `polyh` is a class for calculus of convex polyhedra. It is supplemented by the commands `ball`, `bensolvehedron`, `cart`, `chunion`, `cone`, `cube`, `emptyset`, `intsec`, `msum`, `origin`, `point`, `simplex`, `space`, see Section 5 for details.
- `polyf` is a class for calculus of polyhedral convex functions. It is supplemented by the commands `affine`, `fenv`, `finfc`, `fmax`, `fsum`, `gauge`, `indicator`, `maxnorm`, `sumnorm`, `translative`, see Section 6 for explanation.
- `lpsolve` solves linear programs (GLPK interface), see Section 7.
- `pcpsolve` solves polyhedral convex programs, see Section 8.
- `qcsolve` is a solver for global optimization problems with quasi-concave objective function and linear constraints, see Section 9.
- `molpsolve` solves multiple objective linear programs, see Section 10.
- `vlpsolve` solves vector linear programs, see Section 11.

2. Theoretical Background

The package *bensolve tools* is based on the vector linear program solver *bensolve* [10]. As shown in [9], vector linear programming is equivalent to polyhedral projection, which is the basis of all polyhedral calculus tools. The global optimization solver, see Section 9, is based on a modified version of *bensolve* using the theoretical results in [3].

In this exposition, we use several standard concepts from (polyhedral) Convex Analysis without any further explanation. More information can be found in standard books on Convex Analysis, such as [11], [6], [2].

For details about the algorithms used in *bensolve*, see e.g. [1], [4], [7], [5].

3. Citation Policy

Please cite reference [10] if you use *bensolve tools* in scientific papers. In case you use the global optimization solver, please cite reference [3].

4. Installation and Testing

4.1. Installation for Matlab

We assume that Matlab version 2015b or newer is installed.

1. Go to `http://bensolve.org/tools/download.html`.
2. Download and unpack the file `bt-X.Y.zip`

3. Run Matlab and change into the *bensolve tools* directory `.../bt-X.Y`.

Note that *bensolve tools* for Matlab uses a pre-compiled mex-file, see also Section 4.4.

4.2. Installation for Octave

It is necessary to generate a mex-file named `bensolve.mex`. Please follow the instructions depending on your operating system.

Ubuntu

1. Open a terminal (Shift+Ctrl+T)
2. To install Octave (in case Octave is already installed, make sure 'mkoctfile' is available), run:
   ```
   sudo apt-get update
   sudo apt-get install liboctave-dev
   ```
3. Go to `http://bensolve.org/tools/download.html`.
4. Download the file `bt-X.Y.tgz`.
5. Change into the folder where `bt-X.Y.tgz` is located by typing into the terminal:
   ```
   cd path_to_your_folder
   ```
6. Unpack the files by typing into the terminal:
   ```
   tar -xz < bt-X.Y.tgz
   ```
7. Change into the generated subdirectory:
   ```
   cd bt-X.Y
   ```
8. Run:
   ```
   mkoctfile --mex src/*.c -lglpk -O3 -o bensolve
   ```

MacOS

The following instructions are based on an installation of GNU Octave using the package manager Homebrew. There are many other possibilities to install Octave, see Section 4.4 for possible problems.

1. Install Homebrew: `https://brew.sh`
2. Install GNU Octave. Open a terminal and enter:
   ```
   brew install octave
   ```
3. Go to `http://bensolve.org/tools/download.html`.
4. Download and unpack `bt-X.Y.zip`
5. Change into the directory `bt-X.Y` by typing into the terminal:
   ```
   cd your_download_location/bt-X.Y
   ```

6. Run Octave by entering:
   `octave`

7. In the octave terminal, run:
   `make_oct`

Windows

1. Go to `https://ftp.gnu.org/gnu/octave/windows/`.

2. Download and run the octave-X.Y.Z-w64-installer.exe to install Octave (we recommend the latest version).

3. Go to `http://bensolve.org/tools/download.html`.

4. Download and unpack `bt-X.Y.zip`

5. Run Octave and move into the *bensolve tools* directory `bt-X.Y`

6. In the Octave terminal, run:
   `make_oct`

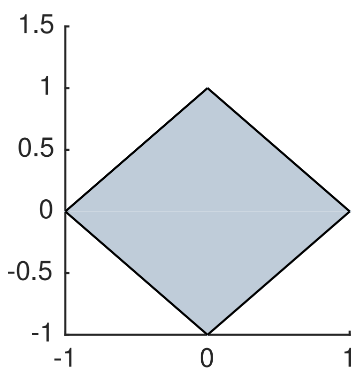
4.3. Test your installation

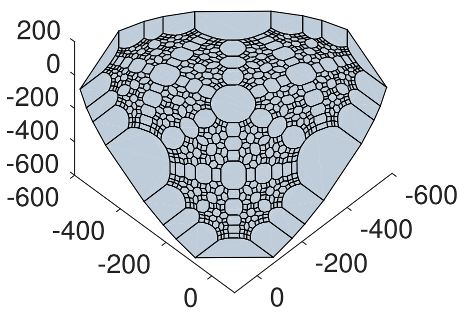
To test your installation, enter

```
P=ball(2);
plot(P);
```

You can also try to generate a more complex image by entering the following commands:

```
P=bensolvehedron(3,3);
Q=cone(3);
R=P+Q;
opt.dirlength=100;
plot(R,opt);
```





4.4. Trouble shooting

If *bensolve tools* does not work correctly, you can try to install a newer version of Matlab or Octave.

Note that *bensolve tools* for Matlab uses a mex-file with the name `bensolve.mexmaci64` (or similar, depending on the operating system). If the pre-compiled mex-file is not working on your operating system, you can try to compile your own mex-file using the source code in the subdirectory `src`. Some hints are given in the file `generate_mex_file_matlab.txt` in the subdirectory `doc`.

If the mex file generation for Octave does not work, we recommend to check whether the GLPK library is installed properly. The usual way to generate a mex file is to run the following in a terminal (exit octave before):

```
cd your_path/bt-X.Y
mkoctfile --mex src/*.c -lglpk -O3 -o bensolve
```

Note that the `O` in `-O3` stands for `Optimization`, 'zero' does not work. Sometimes `glpk.h` is not found. For instance, if it is located in the directory `/usr/include`, enter:

```
mkoctfile --mex src/*.c -I/usr/include -lglpk -O3 -o bensolve
```

Sometimes it can be necessary to specify the glpk library `libglpk.a`. For instance, if it is located in the directory `/usr/lib`, enter:

```
mkoctfile --mex src/*.c -I/usr/include /usr/lib/libglpk.a -O3 -o bensolve
```

4.5. Getting help

Use the `help` command to get help for a class or command (for classes in Matlab also the `doc` command can be useful). For instance:

```
help msum
```

```
 -- P = msum({P1,...,Pn})    Minkowski sum of n polyhedra

    Input:
      {P1,...,Pn}: finitely many polyhedra (cell array of polyh objects)
    Output:
      P: Minkowski sum (polyh object)

    see also: polyh/plus, intsec, chunion, cart
```

5. polyh – Calculus of Convex Polyhedra

`polyh` is a class for computations with convex polyhedra. Throughout this section, a convex polyhedron is called *polyhedron* for short. The most important operations are:

- computing vertices and extreme directions (V-representation)
- computing an inequality representation (H-representation)

- image under linear transformation (in particular projection)
- inverse image under linear transformation
- lineality space, affine hull, dimension, recession cone
- cone generated by a polyhedron
- adjacency list, facet-vertex incidence list
- Minkowski sum of n polyhedra
- intersection of n polyhedra
- closed convex hull of the union of n polyhedra
- cartesian product of n polyhedra
- polar of a polyhedron
- polarcone of a polyhedron
- normal cone of a polyhedron at a point
- comparing polyhedra: subset, proper subset, equality
- plotting 2d and 3d polyhedra

5.1. Representing a convex polyhedron

A convex polyhedron P can be defined in three ways:

- An *H-represenatation*

  $$P = \{x \in \mathbb{R}^n |\ a \leq Bx \leq b,\ l \leq x \leq u\}$$

  means that P is given by finitely many linear inequalities. Here B is an $(m \times n)$-matrix. The lower bound vector a has m components being either a real number or $-\infty$. If a is not specified, all of its components are $-\infty$ by default. The remaining bounds have a similar meaning.

- A *V-representation*

  $$P = \{x \in \mathbb{R}^n |\ x = V\lambda + D\mu + L\eta,\ \lambda \geq 0,\ \mu \geq 0,\ e^\top \lambda = 1\}$$

  means that P is given by a generalized convex hull of finitely many points (the columns of the matrix V), finitely many directions (the columns of the matrix D) and finitely many lineality directions (the columns of the matrix L). Here $e = (1, \dots, 1)^\top$ denotes the all-one-vector.

- A *P-representation*

$$P = \{Mx \in \mathbb{R}^q |\ a \le Bx \le b,\ l \le x \le u\}$$

  means that another polyhedron $Q = \{x \in \mathbb{R}^n |\ a \le Bx \le b,\ l \le x \le u\}$ is given by an H-representation and P is the image of Q under the linear transformation $x \mapsto Mx$, where M is a $(q \times n)$-matrix. The 'P' stands for "projection" and is motivated by the following reformulation of P:

$$P = \{y \in \mathbb{R}^q |\ \exists x \in \mathbb{R}^n :\ Mx - y = 0,\ a \le Bx \le b,\ l \le x \le u\},$$

  which shows that P is a projection of the polyhedron

$$Q = \{((x, y) \in \mathbb{R}^n \times \mathbb{R}^q |\ Mx - y = 0,\ a \le Bx \le b,\ l \le x \le u\}$$

  onto the space $\mathbb{R}^q$. Both an H-representation and a V-representation are special cases of a P-representation. The concept of a P-representation is often related to the term "lifting" in the literature.

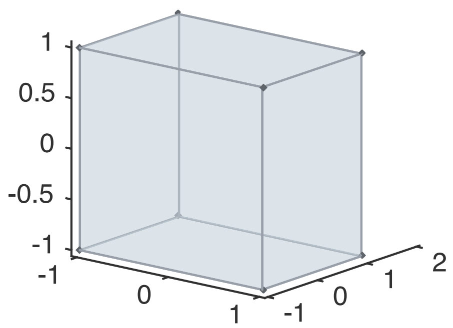
5.1.1. H-representation examples

Example 5.1 *Let a cube in $\mathbb{R}^3$ be defined by*

$$P = \{x \in \mathbb{R}^3 |\ -1 \le x_1 \le 1,\ -1 \le x_2 \le 1,\ -1 \le x_3 \le 1\}.$$

A corresponding `polyh` *instance is obtained and plotted by the following commands:*

```
clear rep;
rep.l=[-1;-1;-1];
rep.u=[1;1;1];
P=polyh(rep,'h');
plot(P);
```



`rep` *is a structure to store a representation of the polyhedron. The command* `rep.l=[-1;-1;-1];` *sets lower bounds $x_1 \ge -1$, $x_2 \ge -1$, $x_3 \ge -1$. Likewise,* `rep.u=[1;1;1];` *sets upper bounds $x_1 \le 1$, $x_2 \le 1$, $x_3 \le 1$. The command* `P=polyh(rep,'h');` *defines a* `polyh` *instance. The option* `'h'` *is required to indicate that the polyhedron is given as an H-representation.*

To define a polyhedron by an H-representation, a structure (named `rep` here) with the following fields is used.

```
rep.B
rep.a
rep.b
rep.l
rep.u
```

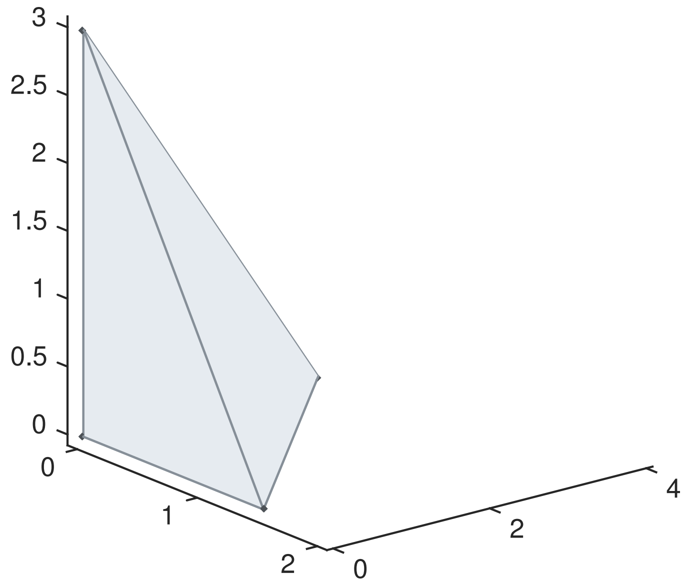
Some fields are optional. Default values for `a`, `l` and `b`, `u` are vectors with entries $-\infty$ and $+\infty$, respectively. At least one of the fields `B`, `l`, `u` is required.

Example 5.2 *Let*

$$P = \{x \in \mathbb{R}^3 | \; x_1 \geq 0, \, x_2 \geq 0, \, x_3 \geq 0, \, 2x_1 + x_2 + x_3 \leq 3\}.$$

The corresponding `polyh` *instance is defined and plotted by the commands:*

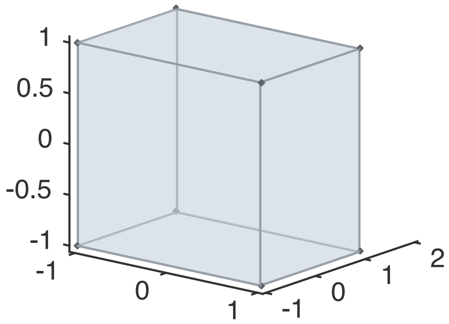
```
clear rep;
rep.B=[2 1 1];
rep.b=3;
rep.l=[0;0;0];
P=polyh(rep,'h');
plot(P);
```



5.1.2. V-representation examples

Example 5.3 *A cube in $\mathbb{R}^3$, see Example 5.1, is also given by its eight vertices:* $(0,0,0)^\top$, $(0,0,1)^\top$, $(0,1,0)^\top$, $(0,1,1)^\top$, $(1,0,0)^\top$, $(1,0,1)^\top$, $(1,1,0)^\top$, $(1,1,1)^\top$. *A corresponding* `polyh` *instance is obtained and plotted by*

```
clear rep;
rep.V=[0 0 0 0 1 1 1 1;0 0 1 1 0 0 1 1;0 1 0 1 0 1 0 1];
P=polyh(rep,'v');
plot(P);
```



To define a polyhedron by a V-representation, a structure (named `rep` here) with the following fields must be given.

`rep.V`
`rep.D`
`rep.L`

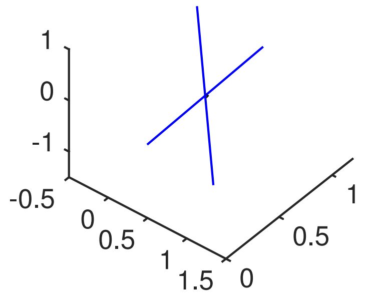
The columns of `rep.V` refer to given points, the columns of `rep.D` refer to given directions and the columns of `rep.L` refer to given lineality directions. At least one point is required to be given.

Example 5.4 *A* `polyh` *instance of the hyperplane*

$$H = \left\{ \begin{pmatrix} 3 \\ 2 \\ 5 \end{pmatrix} + \eta_1 \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix} + \eta_2 \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} \;\middle|\; \eta_1, \eta_2 \in \mathbb{R} \right\}$$

can be generated by the commands:

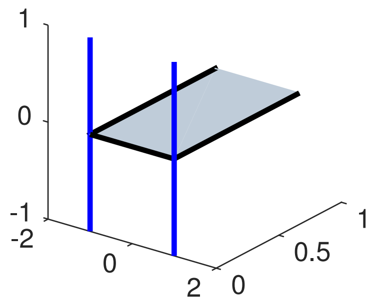
```
clear rep;
rep.V=[3;2;5];
rep.L=[2 1 4;-2 1 0]';
P=polyh(rep,'v');
plot(P);
```



Note that in case of a nontrivial lineality space, we plot only the polyhedron $P \cap U$ (which is a single point in the previous example), where U is a space complementary to the lineality space. Additionaly, lineality directions are drawn by blue lines. This principle becomes more clear by the next example.

Example 5.5 *Plotting the set* $P = \{x \in \mathbb{R}^3 | -1 \leq x_1 \leq 1,\ x_2 \geq 0,\ x_3 \in \mathbb{R}\}$*:*

```
P=ball(1):cone(1):space(1);
plot(P);
```

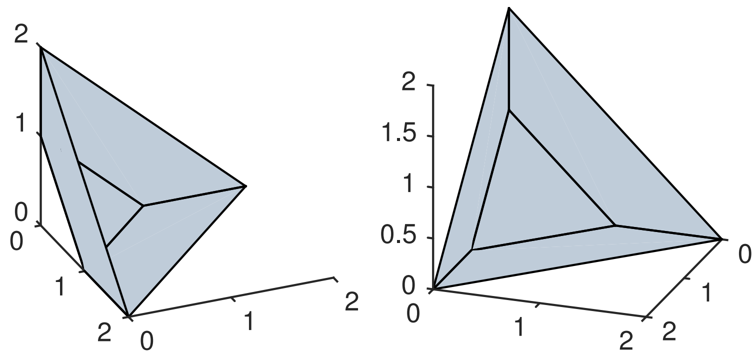


The set P is composed as the cartesian product of three one-dimensional sets: `ball(1)` *defines the interval* $[-1, 1]$*,* `cone(1)` *defines the nonnegative real numbers* $\mathbb{R}_+$*, and space(1) defines* $\mathbb{R}$*.*

Example 5.6 *Let the polyhedron P be given as:*

$$P := \text{conv}\left\{\begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\1\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix}\right\} + \text{cone}\left\{\begin{pmatrix}1\\0\\0\end{pmatrix} \begin{pmatrix}0\\1\\0\end{pmatrix} \begin{pmatrix}0\\0\\1\end{pmatrix}\right\}.$$

```
clear rep;
rep.V=eye(3);
rep.D=eye(3);
P=polyh(rep,'v');
plot(P);
```



5.1.3. P-representation examples

Example 5.7 *Let P be the image of the 3-dimensional unit cube mapped onto $\mathbb{R}^2$ by the mapping*

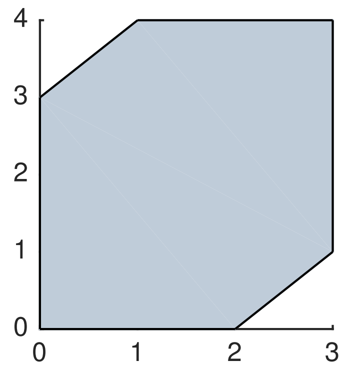
$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \mapsto \begin{pmatrix} 2x_1 + x_3 \\ 3x_2 + x_3 \end{pmatrix},$$

that is,

$$P = \left\{ \begin{pmatrix} 2 & 0 & 1 \\ 0 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \;\middle|\; \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}.$$

A corresponding `polyh` *instance is obtained and plotted by the following commands:*

```
clear rep;
rep.l=zeros(3,1);
rep.u=ones(3,1);
rep.M=[2 0 1;0 3 1];
P=polyh(rep);
plot(P);
```

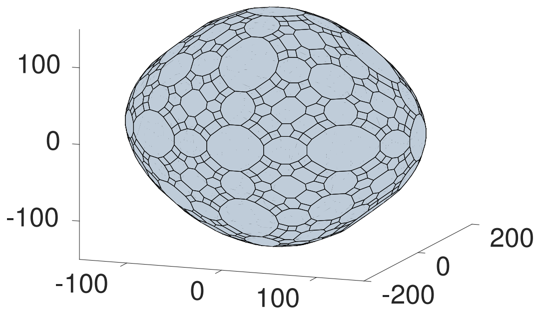


An alternative way to describe this polyhedron is first to define an H-representation of the 3-dimensional unit cube Q and then to compute the image of Q under the linear transformation M by the `im` *command.*

```
clear rep;
rep.l=zeros(3,1);
rep.u=ones(3,1);
Q=polyh(rep,'h');
M=[2 0 1;0 3 1];
P=im(Q,M);
```

Example 5.8 *Another example can be found in the file* `bensolvehedron.m`*. The following command yields the image of a 125-dimensional hypercube with respect to a* (3×125)*-matrix which consists of all possible column-wise arrangements of the numbers* $-2, -1, -0, -1, -2$*.*

```
P=bensolvehedron(3,2);
plot(P);
```



5.2. Retrieving representations and evaluation of polyhedra

For a `polyh` instance `P`, an H-, V- or P-representation is returned as a structure, respectively, by the commands:

```
hrep(P)
vrep(P)
prep(P)
```

A `polyh` instance stores (at least) a P-representation of the polyhedron. To obtain an H- or an V-representation, an evaluation of the polyhedron is required. Evaluation can become expensive, in particular, when the dimension increases.

Example 5.9 *Evaluating the 2-dimensional standard cone*

$$P = \mathbb{R}^2_+ := \{x \in \mathbb{R}^2 |\; x_1 \geq 0,\; x_2 \geq 0\} :$$

```
P=cone(2);
a=iseval(P)
P=eval(P);
b=iseval(P)
```

```
a = 0
b = 1
```

The following commands automatically evaluate a polyhedron if necessary:

```
eval
reinit
hrep
vrep
adj
inc
adj01
inc01
le / <=
ge / >=
eq / ==
ne / ~=
lt / <
gt / >
plot
```

All remaining operations do not require evaluation of the polyhedron. If more than one of the listed functions is used, it is more efficient to store the evaluated polyhedron before calling the commands. For instance

```
P=bensolvehedron(3,2);
P=eval(P);
hrep(P);
vrep(P);
```

is more efficient than

```
P=bensolvehedron(3,2);
hrep(P);
vrep(P);
```

because the first variant requires only one evaluation while the second variant requires two.

A P-representation of a `polyh` instance can be obtained directly by the `prep` command. An evaluation is not necessary.

Example 5.10 *Retrieving a P-representation of the 2-dimensional standard cone:*

```
P=cone(2);
rep=prep(P)
```

```
scalar structure containing the fields:
B = [](0x2)
a = [](0x1)
b = [](0x1)
l =
    0
    0
u =
    Inf
    Inf
```

Example 5.11 *Retrieving an H-representation of the 2-dimensional standard cone:*

```
P=cone(2);
rep=hrep(P)
```

```
ans =
scalar structure containing the fields:

Beq = [](0x2)
beq = [](0x1)
B =
    -1   0
     0  -1
b =
     0
     0
```

Note the `hrep` *returns an H-representation of the form*

$$Bx \leq b, \quad B_{eq}x = b_{eq}.$$

Example 5.12 *Retrieving an V-representation of the 2-dimensional standard cone:*

```
P=cone(2);
rep=vrep(P)
```

```
ans =
scalar structure containing the fields:

L = [](2x0)
V =
    0
    0
D =
    1  0
    0  1
```

5.3. Adjacency lists and incidence lists

After evaluation of a `polyh` instance, adjacency information for vertices and extremal directions is available. The command `adj` returns an adjacency list as a cell array, where indexing starts from 1. Note that the indices of the adjacency list refer to the columns of matrix:

```
[vrep(P).V, vrep(P).D]
```

Moreover, in case of a nontrivial lineality space (that is `vrep(P).L` is nonempty), the adjacency list of $P \cap U$ is returned, where U is a linear space complementary to the lineality space L, that is $L \cap U = \{0\}$, $L + U = \mathbb{R}^n$.

The command `adj01` returns a sparse matrix that contains the same information stored by zero and one entries.

Example 5.13 *The 2-dimensional standard cone has one vertex (index* 1*), which is adjacent to its two extremal directions (indices* 2 *and* 3*):*

```
P=cone(2);
P=eval(P);
a=adj(P)
b=adj01(P)
```

```
a =
{
  [1,1] = 2   3
  [1,2] = 1
  [1,3] = 1
}

b =
Compressed Column Sparse (rows = 3, cols = 3, nnz = 4 [44%])
(2, 1) -> 1
(3, 1) -> 1
(1, 2) -> 1
(1, 3) -> 1
```

After evaluation of a `polyh` instance, "facet-vertex" incidence information is available. Vertices and extremal directions are indexed in the same way as for adjacency lists.

Example 5.14 *The 2-dimensional standard cone shifted by the vector* $(5, 13)^\top$ *has one vertex in* $(5, 13)^\top$*, which is adjacent to its two extremal directions (the two unit vectors):*

```
P=cone(2);
P=P+[5;13];
P=eval(P);
inc(P)
```

```
ans =
{
  [1,1] = 1   3
  [1,2] = 1   2
}
```

The result tells us that the first facet contains "vertex 1" (which is indeed a vertex) and "vertex 3" (which is an extremal direction). Let us verify this information, just for demonstration reasons:

```
...
normalvector=hrep(P).B(1,:)
rhs=hrep(P).b(1)
M=[vrep(P).V,vrep(P).D];
vert1=M(:,1)
vert3=M(:,3)
a=(normalvector * vert1 == rhs)
b=(normalvector * vert3 == 0)
```

```
normalvector =  -1   0
rhs = -5
vert1 =
   5
  13
vert3 =
   0
   1
a = 1
b = 1
```

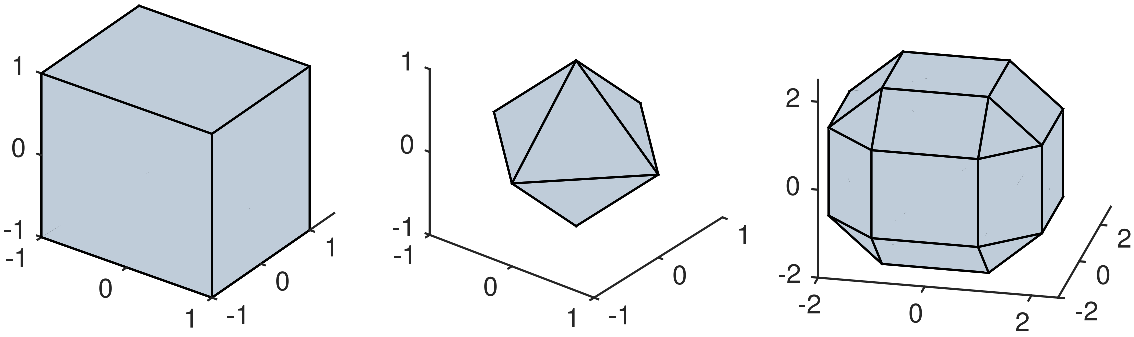
5.4. Calculus examples

The Minkowski sum of two polyhedra $P, Q \subseteq \mathbb{R}^n$ is defined as

$$P + Q = \{x \in \mathbb{R}^n |\ \exists y \in P, \exists z \in Q : \ x = y + z\}.$$

Example 5.15 *The Minkowski sum can be computed by the + operator:*

```
P=cube(3);
Q=ball(3);
R=P+Q;
plot(P);
plot(Q);
plot(R);
```



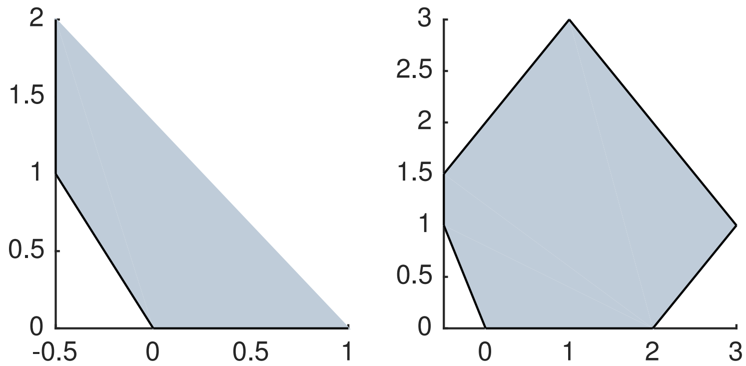
Likewise one can compute

- the *intersection* of two polyhedra by the command: `R=P&Q;`
- the *closed convex hull of the union* of two polyhedra by the command: `R=P|Q;`
- a polyhedron *scaled* by a factor `k`: `R=k*P;`
- a polyhedron *shifted* by a (column) vector `v`: `R=P+v;`

The following example demonstrates how these operations can be combined.

Example 5.16 *In line 3, we compute the closed convex hull of the 2-dimensional standard cone and the point* $(-1/2, 1)^\top$. *In line 6, the result* `R` *is intersected (*`&`*operator) by the sum-norm unit ball (generated by the command* `ball(2)`*), which is scaled by 2 and shifted by the vector* $(1, 1)^\top$*:*

```
1 P=cone(2);
2 Q=point([-1/2;1]);
3 R=Q|P;
4 S=R&(2*ball(2)+[1;1]);
5 plot(R);
6 plot(S);
```

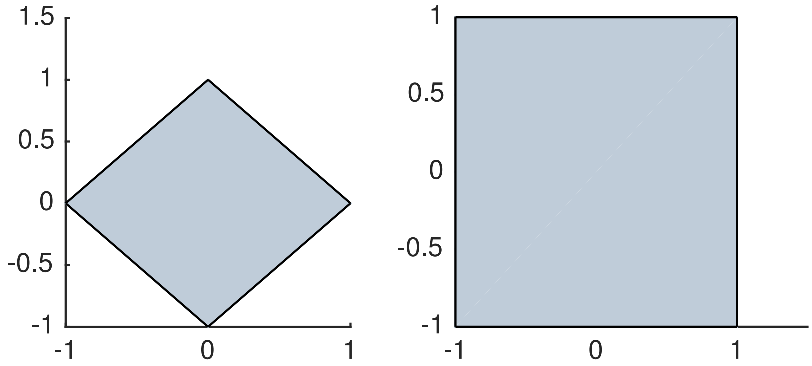


The *polar* of a polyhedron $P \subseteq \mathbb{R}^n$ is defined as

$$P^\circ = \{y \in \mathbb{R}^n | \, \forall x \in P : \; y^\top x \leq 1\}.$$

Example 5.17 *The polar of the 2-dimensional sum-norm unit ball is a square:*

```
P=ball(2);
R=polar(P);
plot(P);
plot(R);
```

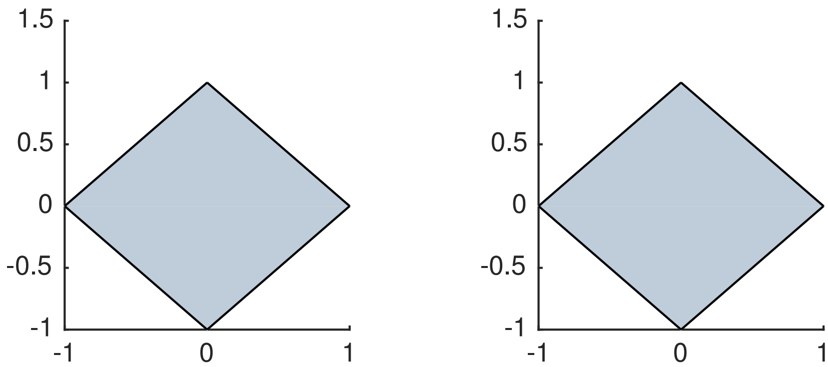


An alternative way to express `R=polar(P)` *is*

```
R=P';
```

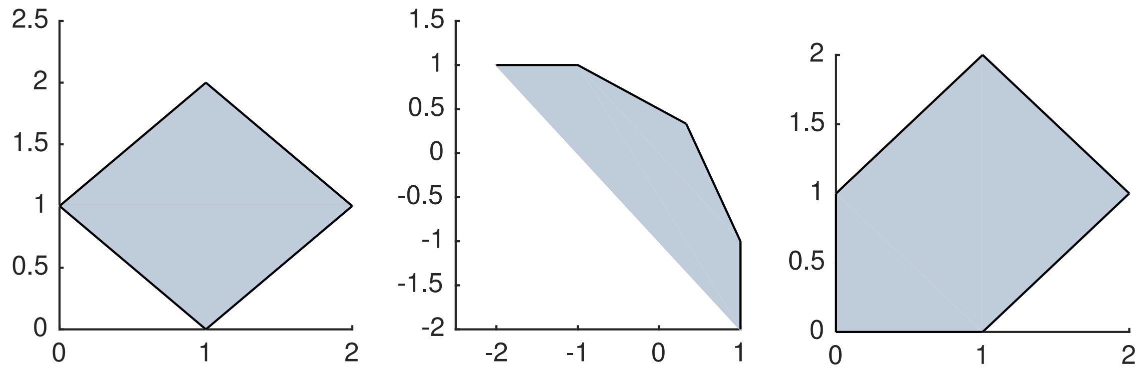
If a polyhedron contains the origin, then the polar of the polar results into the original polyhedron.

```
P=ball(2);
R=P'';
plot(P);
plot(R);
```



Otherwise it is the closed convex hull of the origin and the original polyhedron.

```
P=ball(2)+[1;1];
R=P';
S=R';
plot(P);
plot(R);
plot(S);
```



5.5. Class polyh – list of methods

```
-- polyh : class for calculus of convex polyhedra
```

5.5.1. Initialization and evaluation

```
-- P = polyh(REP,REPTYPE)    constructor

   constructor of polyh class

   Input:
     REP: representation of the polyhedron (struct)
   Optional input:
     REPTYPE: type of representation (char: 'v', 'h', 'p')
            :   reptype 'v':
            :     V matrix of points
            :     D matrix of directions
            :     L matrix of lineality directions
            :   reptype 'h'
            :     fields B, a, b, l, u according to the H-representation
            :       P = { x | a <= Bx <= b, l <= x <= u}
            :   reptype 'p' (default)
            :     fields M, B, a, b, l, u according to the P-representation
            :       P = { Mx | a <= Bx <= b, l <= x <= u}
   Output:
     P: polyhedron (polyh object)
```

```
-- R = eval(P)    evaluate polyh object

   i.e. compute H- and V-representation, adjacency list, incidence list

   Input:
     P: polyhedron (polyh object)
   Output:
     R: evaluated polyhedron (polyh object)


-- R = reinit(P,REPTYPE)    re-initialize polyh object

   re-initialize polyh object using its V- or H-representation
   (can simplify further computations but requires evaluation)

   Input:
     P: polyhedron (polyh object)
   Optional input:
     REPTYPE: type of representation: 'v' (default) or 'h' (char)
   Output:
     R: polyhedron (polyh object)
```

5.5.2. Polyhedral calculus

```
-- R = plus(P,Q)    sum
   R = P + Q

   compute the Minkowski sum of two polyhedra or
   sum of polyhedron and vector

   Input:
     P, Q: two polyhedra (polyh objects)
         : or one polyhedron and one vector
   Output:
     R: Minkowski sum (polyh object)


-- R = mtimes(k,P)    scaling
   R = k * P

   scaling of polyhedron

   Input:
     k: scaling factor (number)
     P: polyhedron (polyh object)
   Output:
```

```
    R: scaled polyhedron (polyh object)


-- R = minus(P,Q)    difference
   R = P - Q

   compute the Minkowski difference of two polyhedra or
   sum of polyhedron and vector

   Input:
     P, Q: two polyhedra (polyh objects)
         : or one polyhedron and one vector
   Output:
     R: Minkowski difference (polyh object)


-- R = uminus(P)    negative of
   R = -P

   compute negative of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     R: polyhedron (polyh object)


-- R = and(P,Q)    intersection
   R = P & Q

   compute the intersection of two polyhedra

   Input:
     P, Q: two polyhedra (polyh objects)
   Output:
     R: intersection (polyh object)


-- R = or(P,Q)    closed convex hull of union of two polyhedra
   R = P | Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Output:
     R: closed convex hull of union (polyh object)
```

```
-- R = colon(P,Q)    cartesian product of two or three polyhedra
   R = colon(P,Q,S)
   R = P : Q
   R = P : Q : S

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional input:
     S: third polyhedron (polyh object)
   Output:
     R: cartesian product (polyh object)


-- R = im(P,M)    image under linear transformation

   Input:
     P: polyhedron (polyh object)
     M: linear transformation (matrix)
   Output:
     R: image M(P) (polyh object)


-- R = inv(P,M)    inverse image of polyhedron under linear transformation

   Input:
     P: polyhedron (polyh object)
     M: linear transformation (matrix)
   Output:
     R: inverse image {x | Mx in P} (polyh object)
```

5.5.3. Retrieving properties and related objects

```
-- d = sdim(P)    space dimension

   Input:
     P: polyhedron (polyh object)
   Output:
     d: space dimension of P (number)


-- d = dim(P)    dimension of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     d: dimension of P (number)
```

```
-- d = ldim(P)    lineality space dimension

   Input:
     P: polyhedron (polyh object)
   Output:
     d: lineality space dimension of P (number)


-- v = getpoint(P)    point belonging to polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     v: if P is nonempty: point v belonging to polyhedron P (column vector)
      : if P is empty: (spacedim x 0) matrix


-- v = rint(P)    relative interior point

   Input:
     P: polyhedron (polyh object)
   Output:
     v: if P is nonempty: relative interior point v of P (column vector)
      : if P is empty: (spacedim x 0) matrix


-- [R,d] = affine(P)    affine hull

   Input:
     P: polyhedron (polyh object)
   Output:
     R: affine hull of P (polyh object)
     d: dimension of P (number)


-- R = lin(P)    lineality space

   Input:
     P: polyhedron (polyh object)
   Output:
     R: lineality space of P (polyh object)


-- R = polar(P)    polar set of nonempty polyhedron
   R = P'

   Input:
```

```
    P: polyhedron (polyh object)
  Output:
    R: polar set of P (polyh object)
```

```
-- R = polarcone(P)    polar cone of nonempty polyhedron

  Input:
    P: polyhedron (polyh object)
  Output:
    R: polar cone of P (polyh object)
```

```
-- R = conic(P)    closed cone generated by polyhedron

  Input:
    P: polyhedron (polyh object)
  Output:
    R: closed conic hull of P (polyh object)
```

```
-- R = ncone(P,v)    normal cone of polyhedron P at point v

  Input:
    P: polyhedron (polyh object)
    v: column vector
  Output:
    R: normal cone of P at v (polyh object)
```

```
-- R = recc(P)    reccesion cone of polyhedron

  Input:
    P: polyhedron (polyh object)
  Output:
    R: recession cone of P (polyh object)
```

5.5.4. Property checking

```
-- flag = isempty(P)    test whether polyhedron is empty

  Input:
    P: polyhedron (polyh object)
  Output:
    flag: nonzero if P is empty (number)
```

```
-- flag = iselem(P,v)    test whether point belongs to polyhedron

   Input:
     P: polyhedron (polyh object)
     v: point (column vector)
   Output:
     flag: nonzero if v belongs to P (number)


-- flag = iseval(P)    check whether polyhedron is evaluated

   Input:
     P: polyhedron (polyh object)
   Output:
     flag: nonzero if polyhedron is evaluated (number)


-- flag = isbounded(P,C,POLARCONE_D,POLARCONE_L)    check boundedness

   test polyhedron P for being bounded (w.r.t. cone C,
   i.e. there is a bounded set B such that P is contained in B+C)

   Input:
     P           : polyhedron (polyh object)
   Optional input:
     C           : cone (polyh object)
     POLARCONE_D : see remark
     POLARCONE_L : see remark
   Output:
     flag: nonzero if P is bounded or C-bounded (number)

   Remark: since the cone C needs to be evaluated, it can be more
     efficient to enter a V-representation of the polar cone (if known)
```

5.5.5. Retrieving representations

```
-- REP = hrep(P)    H-representation

   retrieve H-representation of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     REP: H-representation of P (struct)
```

```
-- REP = vrep(P)    V-representation

   retrieve V-representation of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     REP: V-representation of P (struct)


-- REP = prep(P)    P-representation

   retrieve P-representation of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     REP: P-representation of P (struct)
```

5.5.6. Retrieving adjacency and incidence lists

```
-- A = adj(P)    adjacency list

   retrieve adjacency list (cell array) of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     A: adjacency list of P (cell array)

   Remark: In case of a nontrivial lineality space, the list
           corresponds to the intersection of P with
           a complement of the lineality space.


-- M = adj01(P)    adjacency list

   retrieve 0-1 adjacency list of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     M: adjacency list of P (sparse matrix)

   Remark: In case of a nontrivial lineality space, the list
           corresponds to the intersection of P with
```

```
          a complement of the lineality space.


-- A = inc(P)    incidence list

   retrieve facet-vertex incidence list (cell array) of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     A: incidence list of P (cell array)

   Remark: In case of a nontrivial lineality space, the list
           corresponds to the intersection of P with
           a complement of the lineality space.


-- M = inc01(P)    incidence list

   retrieve 0-1 incidence list of polyhedron

   Input:
     P: polyhedron (polyh object)
   Output:
     M: incidence list of P (sparse matrix)

   Remark: In case of a nontrivial lineality space, the list
           corresponds to the intersection of P with
           a complement of the lineality space.
```

5.5.7. Comparison of polyhedra

```
-- f = le(P,Q,epsilon)    subset
   P <= Q

   test: P subset of Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              default: 1e-6
   Output:
     f: flag to indicate whether P is subset of Q (number)
```

```
-- f = ge(P,Q,epsilon)    superset
   P >= Q

   test: P superset of Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              see polyh/le for details
   Output:
     f: flag to indicate whether P is superset of Q (number)


-- f = eq(P,Q,epsilon)    equal
   P == Q

   test: P equal to Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              see polyh/le for details
   Output:
     f: flag to indicate whether P is equal to Q (number)


-- f = ne(P,Q,epsilon)    unequal
   P ~= Q

   test: P unequal to Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              see polyh/le for details
   Output:
     f: flag to indicate whether P is unequal to Q (number)


-- f = lt(P,Q,epsilon)    proper subset
   P < Q

   test: P proper subset of Q
```

```
   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              see polyh/le for details
   Output:
     f: flag to indicate whether P is proper subset of Q (number)
```

```
-- f = gt(P,Q,epsilon) proper superset
   P > Q

   test: P proper superset of Q

   Input:
     P, Q: two polyhedra (polyh objects)
   Optional Input:
     epsilon: tolerance (number)
              see polyh/le for details
   Output:
     f: flag to indicate whether P is proper superset of Q (number)
```

5.5.8. Plotting of polyhedra

```
-- plot(P,OPT)    plot

   plot polyhedron

   Input:
     P: polyhedron (polyh object)
   Optional Input:
     OPT: options (struct)

   ------------------------------------------------------------------
   option          default value       explanation
   ------------------------------------------------------------------
   color           [3/4 4/5 17/20]     color as [r,g,b]
   color2d         color               color of 2d faces
   color1d         0.7*color           color of 1d faces
   color0d         0.4*color           color of 0d faces
   edgewidth       1                   edge width
   dirlength       1                   lentgh of extremal directions
   vertexsize      1.3                 vertex size
   alpha           0.4                 transparency
   ------------------------------------------------------------------
```

5.6. Class polyh – list of supplementary functions in alphabetical order

```
-- P = ball(d)    sum-norm unit ball

   Input:
     d: dimension (number)
   Output:
     P: unit ball (polyh object)


-- P = bensolvehedron(d,m)    bensolvehedron

   that is, a projection of a hypercube of dimension (2m+1)^d
   where the columns of the projection matrix consist of all
   possible arrangements of the set {-m,-(m-1),...,-1,0,1,...,m-1,m}

   Input:
     d: dimension
     m: parameter, e.g., 1,2,3,... (number)
   Output:
     P: bensolvehedron (polyh object)


-- P = cart({P1,...,Pn})    cartesian product of n polyhedra

   Input:
     {P1 ,..., Pn}: finitely many polyhedra (cell array of polyh objects)
   Output:
     P: cartesian product (polyh object)


-- P = chunion({P1,...,Pn})    closed convex hull of union of n polyhedra

   Input:
     {P1,...,Pn}: finitely many polyhedra (cell array of polyh objects)
   Output:
     P: closed convex hull of union (polyh object)


-- P = cone(d)    standard cone

   Input:
     d: dimension
   Output:
     P: standard cone (polyh object)


-- P = cube(d)    max-norm unit ball
```

```
   Input:
     d: dimension
   Output:
     P: max-norm unit ball (polyh object)


-- P = emptyset(d)    empty set

   Input:
     d: space dimension
   Output:
     P: empty set (polyh object)


-- P = intsec({P1,...,Pn})    intersection of n polyhedra

   Input:
     {P1,...,Pn}: finitely many polyhedra (cell array of polyh objects)
   Output:
     P: intersection (polyh object)


-- P = msum({P1,...,Pn})    Minkowski sum of n polyhedra

   Input:
     {P1,...,Pn}: finitely many polyhedra (cell array of polyh objects)
   Output:
     P: Minkowski sum (polyh object)


-- P = origin(d)    origin

   Input:
     d: space dimension
   Output:
     P: origin (polyh object)


-- P = point(v)    point

   Input:
     v: column vector
   Output:
     P: point (polyh object)


-- P = simplex(d)    regular simplex
```

```
   Input:
     d: dimension (number)
   Output:
     P: simplex (polyh object)


-- P = space(d)    whole space polyhedron

   Input:
     d: space dimension
   Output:
     P: space (polyh object)


-- carr = faces(P,k)   l-faces of P

   Input:
     P: polyhedron (polyh object)
   Optional input:
     k: dimension of faces (number)
        default: return all proper faces
   Output:
     carr: cell array of faces


-- G = hasse(P,k,inv)    Hasse diagram of polyhedron

   Input:
     P: polyhedron (polyh object)
   Optional input:
     k:   dimension (number, default: dim of P)
     inv: flag to indicate inverse order (bool, default: false)
   Output:
     G: Hasse diagram (cell array)

   The nodes of G correspond to the faces of P. An arc from node
   A to node B means that A subset B and dim A + 1 = dim B. If
   inv==true, an arc from node A to node B means that A supset B
   and dim A = dim B + 1.

   G is stored as cell array of nodes, each cell has 4 entries:
     1: successor nodes
     2: vertex indices of the face
     3: polyh object of the face
     4: dimension of the face

   For details of the algorithm, see V. Kaibel, M. E. Pfetsch:
```

`Computing the face lattice of a polytope from its vertex-facet`
`incidences, Computational Geometry 23 (2002) 281-290`

6. polyf – Calculus of Polyhedral Convex Functions

`polyf` is a class for computations with polyhedral convex functions. As we only consider convex functions, we say *polyhedral function* for short. The most important operations for polyhedral functions are:

- pointwise maximum of n polyhedral functions
- lower closed convex envelope of n polyhedral functions
- infimal convolution of n polyhedral functions
- pointwise sum of n polyhedral functions
- conjugate of a polyhedral function
- recession function of a polyhedral function
- computation of domain, range, level sets, recession cone of a polyhedral function
- computation of the subdifferential at some point
- test for pointwise ordering and equality of two polyhedral functions
- plot of polyhedral functions with one or two variables

6.1. Representing polyhedral convex functions

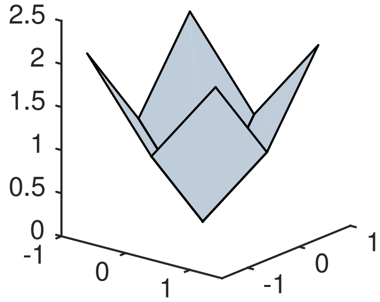
A polyhedral convex function $f : \mathbb{R}^n \to [-\infty, +\infty]$ is represented by its epigraph

$$\operatorname{epi} f = \{(x, r) \in \mathbb{R}^n \times \mathbb{R} | \; r \geq f(x)\},$$

which is a convex polyhedron. The epigraph of f is stored as a `polyh` object, see Section 5.

Example 6.1 *Consider the sum norm $\|x\|_1 := \sum_{i=1}^n |x_i|$ in $\mathbb{R}^n$. Its epigraph is the cone generated by the set $B \times \{1\}$, where $B := \{x \in \mathbb{R}^n | \; \|x\|_1 \leq 1\}$. To create a* `polyh` *instance of B, we can use the command* `ball`*, see Section 5.*

```
epi=conic(ball(2):point(1));
f=polyf(epi);
plot(f)
```



Alternatively the sum norm can be defined by composition of affine functions:

```
f1=affine([1;0],0);
f2=affine([0;1],0);
f3=affine([-1;0],0);
f4=affine([0;-1],0);
g=fmax(f1,f3) + fmax(f2,f4);
```

The easiest way is to use the `sumnorm` *command:*

```
h=sumnorm(2);
```

All three definitions generate the same function:

```
f==g
g==h
```

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

The value of f at x, say at $x = (1, 2)^{\mathsf{T}}$, can be computed as:

```
f=sumnorm(2);
val(f,[1;2])
```

```
ans =
     3
```

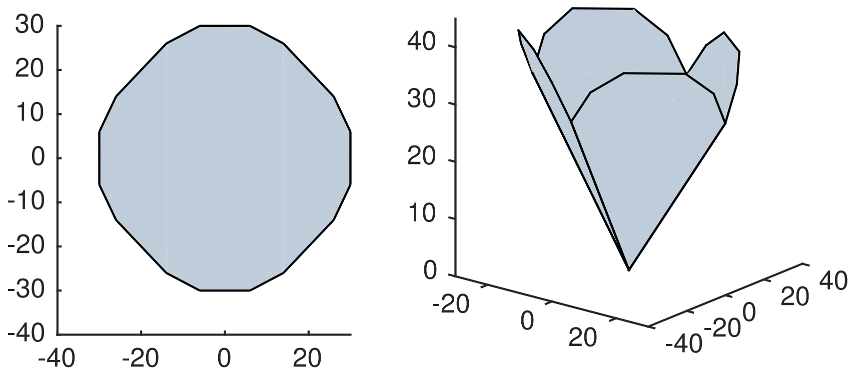
Note that the value of f at x is obtained from the epigraph of f by solving the linear program

$$\min r \quad \text{s.t.} \quad (x, r)^{\mathsf{T}} \in \operatorname{epi} f.$$

Since epi f is stored as a P-representation, compare Section 5, the linear program has more than one variable in general.

A `polyf` instance with one ore two variables can be plotted. By default, the plotting region is the interval $[-1, 1]$ and the square $[-1, 1] \times [-1, 1]$, respectively. Other plotting regions are possible:

```
f=sumnorm(2)
R=bensolvehedron(2,2);
plot(R);
plot(f,R);
```

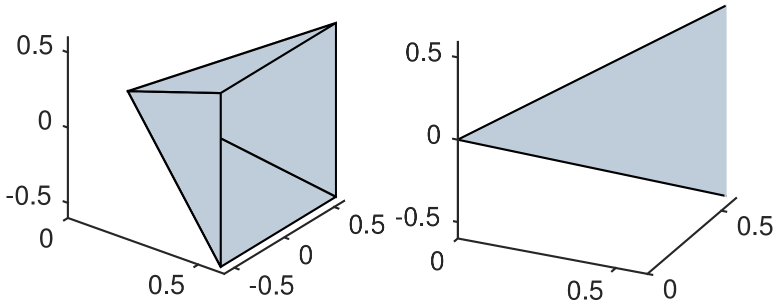


6.2. Calculus examples

Example 6.2 *Subdifferential: We compute the subdifferential of the indicator function of the sum-norm unit ball* B *at* $x = (1,0,0)^{\mathsf{T}}$ *and, which is known to be the same, the normal cone of* B *at* x*. Then we compute the subdifferential at* $x = (1/2,1/2,0)^{\mathsf{T}}$*:*

```
B=ball(3);
f=indicator(B);
P=subdiff(f,[1;0;0]);
Q=ncone(B,[1;0;0]);
P==Q
plot(P);
R=subdiff(f,[1/2;1/2;0]);
plot(R);
```

```
ans =
     1
```

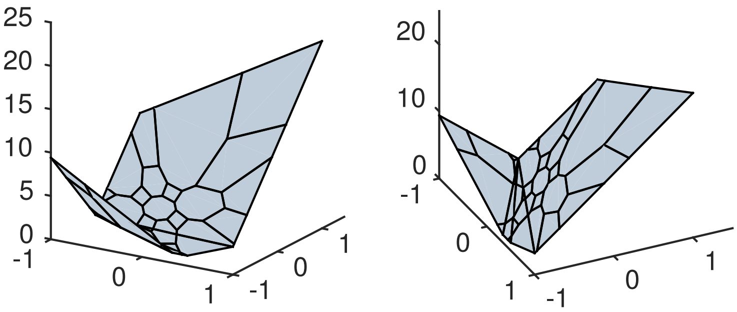


Example 6.3 *Assume we want to compose a polyhedral function of the form*

$$f(x) = \sum_{i=1}^{k} d_i(A^i x + b^i)$$

where $d_i : \mathbb{R}^m \to (-\infty, \infty]$ is a gauge function of a polytope $B_i \in \mathbb{R}^m$ with $0 \in B_i$, $A_i \in \mathbb{R}^{m \times n}$, $b^i \in \mathbb{R}^m$, for $i = 1, \ldots, k$. To illustrate the idea, we use a small sample instance with $m = 3$, $n = 2$, $k = 2$:

```
A1=[1 2;2 3;3 4]; b1=[1;2;3];
A2=[3 2;1 4;3 2]; b2=[2;1;0];
B1=ball(3);
B2=bensolvehedron(3,1);
f=precomp(gauge(B1),A1,b1)+precomp(gauge(B2),A2,b2);
plot(f);
```



6.3. Class polyf – list of methods

```
 -- polyh : class for calculus of polyhedral convex functions

    Public properties:
      nvar      : number of variables
```

6.3.1. Initialization and evaluation

```
-- f = polyf(P)    constructor

   constructor of polyf class

   Input:
     P: epigraph of polyhedral convex function (polyh object)
   Output:
     f: polyhedral convex function (polyf object)


-- h = eval(f)    evaluate polyf object

   i.e. evaluate epigraph

   Input:
     f: function (polyf object)
   Output:
     h: evaluated function (polyf object)


-- h = reinit(f,REPTYPE)    re-initialize function

   re-initialize polyf object using the V- or H-representation of the epigraph
   (can simplify further computations but requires evaluation)

   Input:
     f: function (polyf object)
   Optional input:
     REPTYPE: type of representation: 'v' (default) or 'h' (char)
   Output:
     h: function (polyh object)
```

6.3.2. Basic operations and retrieving related objects

```
-- y = val(f,x)    value y=f(x)

   Input:
     f: function (polyf object)
     x: argument (column vector)
   Output:
     y: value y=f(x) (number)

   Remark: command requires to solve one linear program
```

```
-- P = epi(f)    epigraph

   epigraph of function f

   Input:
     f: function (polyf object)
   Output:
     P: epigraph of f (polyh object)


-- P = dom(f)    domain

   domain of function f

   Input:
     f: function (polyf object)
   Output:
     P: domain of f (polyh object)


-- P = level(f,a)    sublevel set

   sublevel set of function of f w.r.t. level a

   Input:
     f: function (polyf object)
     a: level (number)
   Output:
     P: sublevel set (polyh object)


-- P = recc(f)    recession cone of polyf object

   i.e. the recession cone of nonempty sublevel sets

   Input:
     f: function (polyf object)
   Output:
     P: recession cone of f (polyh object)


-- P = subdiff(f,x)    subdifferential

   subdifferential of proper function f at argument x

   Input:
     f: function (polyf object)
```

```
    x: argument (column vector)
  Output:
    P: subdifferential (polyh object)
```

6.3.3. Property checking

```
-- flag = iseval(f)    check whether function is evaluated

   Input:
     f: function (polyf objects)
   Output:
     flag: nonzero if function is evaluated (number)


-- flag = isimproper(f)    check whether function is improper

   Input:
     f: function (polyf objects)
   Output:
     flag:
        0 = proper
        1 = improper with nonempty domain
        2 = improper with empty domain
```

6.3.4. Calculus and composition of polyhedral convex functions

```
-- h = fmax(f,g)    pointwise maximum of two functions
   h = f & g

   Input:
     f,g: two functions (polyf objects)
   Output:
     h: pointwise maximum function (polyf object)


-- h = finfc(f,g)    infimal convolution of two functions

   Input:
     f,g: two functions (polyf objects)
   Output:
     h: infimal convolution (polyf object)


-- h = fenv(f,g)    lower closed convex envelope of two functions
   h = f | g
```

```
   Input:
     f,g: two functions (polyf objects)
   Output:
     h: lower closed convex envelope (polyf object)


-- h = fsum(f,g)    sum of two functions
   h = f + g

   Input:
     f,g: two functions (polyf objects)
   Output:
     h: sum (polyf object)


-- h = mtimes(k,f)    scaling
   h = k * f

   nonnegative scaling of function values

   Input:
     k: scaling factor (nonnegative number)
     f: function (polyf object)
   Output:
     h: scaled function (polyf object)


-- h = precomp(f,M,v)    pre-composition with affine transformation

   h(x) = f(M x + v)

   Input:
     f: function (polyf object)
     M: matrix
     v: column vector
   Output:
     h: function (polyf object)


-- h = conj(f)    conjugate function

   Input:
     f: function (polyf object)
   Output:
     h: conjugate function (polyf object)
```

```
-- h = recf(f)    recession function

   recession function of f

   Input:
     f: function (polyf object)
   Output:
     h: recession function of f (polyf object)
```

6.3.5. Comparing polyhedral convex functions

```
-- flag = le(f,g)    <= for two functions
   f <= g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)<=g(x) for all x (number)


-- flag = ge(f,g)    >= for two functions
   f >= g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)>=g(x) for all x (number)


-- flag = eq(f,g)    == for two functions
   f == g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)==g(x) for all x (number)


-- flag = ne(f,g)    ~= for two functions
   f ~= g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)~=g(x) for some x (number)
```

```
-- flag = lt(f,g)    (<= and ~=)  for two functions
   f < g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)<=g(x) for all x and f(x)<g(x) for some x (number)


-- flag = gt(f,g)    (>= and ~=)  for two functions
   f > g

   Input:
     f,g: two functions (polyf objects)
   Output:
     flag: nonzero if f(x)>=g(x) for all x and f(x)>g(x) for some x (number)
```

6.3.6. Plotting polyhedral convex functions

```
-- plot(f,opt)    plot function

   Input:
     f: function (polyf object)
     opt: optional options (struct)

   ---------------------------------------------------------------------------
   option          default value      explanation
   ---------------------------------------------------------------------------
   color          [3/4 4/5 17/20]     color as [r,g,b]
   color2d        color               color for 2d faces
   color1d        0.7*color           color for 1d faces
   color0d        0.4*color           color for 0d faces
   edgewidth      1                   edge width
   vertexsize1    1.3                 vertex size for singleton domain
   alpha          0.4                 transparency (Matlab only)
   range          cube(sdim-1)        plot range for functions (polyh object)
   showrange      true                show the range for function plots
   rangecolor     [14/15 14/15 14/15] color of range for function plots
   ---------------------------------------------------------------------------
```

6.4. Class polyf – list of supplementary functions in alphabetical order

```
-- f = affine(a,b)    affine function

   f(x)=a^T x + b
```

```
   Input:
     a: column vector a
   Optional input:
     b: number b
        default: 0
   Output:
     f: affine function (polyf object)


-- f = fenv({f1,...,fn})    lower closed convex envelope

   lower closed convex envelope of n polyhedral functions

   corresponds to convex hull of the union of epigraphs

   Input:
     {f1,...,fn}: n polyhedral convex functions with same number
                  of variables (cell array of polyf objects)
   Output:
     f: lower closed convex envelope (polyf object)


-- f = finfc({f1,...,fn})    infimal convolution

   infimal convolution of n polyhedral functions

   corresponds to Minkowski sum of epigraphs

   Input:
     {f1,...,fn}: polyhedral convex functions with same number
                  of variables (cell array of polyf objects)
   Output:
     f: infimal convolution (polyf object)


-- f = fmax({f1,...,fn})    pointwise maximum

   pointwise maximum of n polyhedral functions

   corresponds to intersection of epigraphs

   Input:
     {f1,...,fn}: polyhedral convex functions with same number
                  of variables (cell array of polyf objects)
   Output:
     f: pointwise maximum (polyf object)
```

```
-- f = fsum({f1,...,fn})    pointwise sum of n polyhedral functions

   Input:
     {f1,...,fn}: n polyhedral convex functions with same number
                  of variables (cell array of polyf objects)
   Output:
     f: pointwise sum (polyf object)


-- f = gauge(P)    gauge function

   gauge function f of a polyhedral set P,
   where zero must be contained in P

   f(x) = inf{r > 0 | x \in r P}

   Input:
     P: polyhedon (polyh object)
   Output:
     f: gauge function (polyf object)


-- f = indicator(P)    indicator function

   indicator function f of a polyhedral set P

   Input:
     P: polyhedon (polyh object)
   Output:
     f: indicator function (polyf object)


-- f = maxnorm(n)    maximum norm

   Input:
     n: number of variables
   Output:
     f: maximum norm (polyf object)


-- f = sumnorm(n)    sum norm

   Input:
     n: number of variables
   Output:
     f: sum norm (polyf object)
```

```
-- f = support(P)    support function

   Input:
     P: polyhedon (polyh object)
   Output:
     f: support function (polyf object)


-- f = translative(P,C,k)    translative function

   translative function of polyhedron P, polyhedral cone C, vector k in C

     f(x)=inf{r : x + r*k in P+C}

   Input:
     P: polyhedron (polyh object)
     C: polyhedral cone (polyh object)
     k: column vector
   Output:
     f: translative function (polyf object)
```

7. lpsolve – Solving Linear Programs

lpsolve can be used to solve linear programs. It is based on the GNU linear programming kit (GLPK). The input data consist of the objective function c, which is a column vector, the feasible set S, which is a polyh object, and an optimization direction.

```
-- [optval,sol_p,sol_d,status] = lpsolve(c,S,optdir)    solve linear program

   min c^T y  s.t.  y in S

   where S is given by a P-representation:

   S = {Mx : l <= x <= u, a <= Bx <= b}

   Input:
     c         objective function (column vector)
     S         feasible set (polyh object)
     optdir    'min' (default) or 'max'
   Output:
     optval     optimal value of the problem (number)
     sol_p      primal solution (column vector)
     sol_d      dual solution (column vector)
     status     solution status (string)
```

```
Remark:
  the dual solution refers to the P-representation of S and consists of
  row variables:     the first m entries
  column variables:  the last n entries
  where [m,n] = size(S.prep.B)
```

Example 7.1 *Consider the minimization problem with* $c^\mathsf{T} = (0, \dots, 0, 1, \dots, 1)$ *where the feasible set is the sum-norm unit ball.*

```
n=5;
k=2;
c=[zeros(n-k,1);ones(k,1)];
S=ball(n);
[optval,sol_p]=lpsolve(c,S)
```

```
optval = -1
sol_p =
   0
   0
   0
   0
  -1
```

The following code computes the set of all optimal solutions:

```
...
rep.B=c';
rep.b=optval;
H=polyh(rep,'h');
P=S&H;
vrep(P)
```

An alternative variant using the `polyf` *class is:*

```
...
P=S&level(affine(c,0),optval);
vrep(P)
```

```
ans =
  scalar structure containing the fields:
    L = [](5x0)
    V =
       0   0
       0   0
       0   0
      -1   0
       0  -1
    D = [](5x0)
```

Note that the last command (computation of a V-representation) requires evaluation and is therefore expensive in larger example. Less expensive is, for instance, the computation of the dimension of the set of all solutions:

```
...
dim(P)
```

```
ans =  1
```

8. pcpsolve – Solving Polyhedral Convex Programs

The command pcpsolve provides a convenient way to solve polyhedral convex programs. Internally, the polyhedral convex program is reformulated as a linear program and solved by GLPK.

```
 -- [optval, sol]=pcpsolve(f,S)    solve polyhedral convex program

    minimize f(x)  s.t.  x in S

    where S is given by a P-representation:

    S = {Mx : a <= Bx <= b , l <= x <= u}

    Input:
      f     polyhedral convex objective function (polyf object)
      S     feasible set (polyh object)
    Output:
      optval: optimal value
      sol: an optimal solution (column vector)
```

Example 8.1 *Locational analysis: Given five points in the plane: $a^{(1)} = (1,4)^\mathsf{T}$, $a^{(2)} = (2,2)^\mathsf{T}$, $a^{(3)} = (3,3)^\mathsf{T}$, $a^{(4)} = (1,2)^\mathsf{T}$, $a^{(5)} = (6,5)^\mathsf{T}$, we are looking for $x \in \mathbb{R}^2$ minimizing the function*

$$\sum_{i=1}^{5} \|x - a^{(i)}\|_1$$

subject to the constraint $x_1 + x_2 \leq 2$:

```
A=[1 2 3 1 6;4 2 3 2 5];
m=size(A,2);
C=cell(m,1);
for i=1:m
  C{i,1}=precomp(sumnorm(2),eye(2),-A(:,i));
end
S=level(affine([1;1],0),2);
[optval,sol]=pcpsolve(fsum(C),S)
```

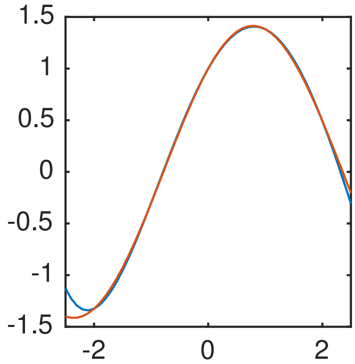
```
optval =  19
sol =
   0
   2
```

Example 8.2 *Chebyshev Approximation: The function* $f(x) = cos(x) + sin(x)$ *is approximated by a polynomial* $g_a(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ *of degree* 4 *at a finite set* $X = [-2, -1, 0, 1, 2]$ *by solving the polyhedral convex optimization problem*

$$\min_{a \in \mathbb{R}^5} \max_{x \in X} |f(x) - g_a(x)|.$$

```
X=-2:1:2;
m=length(X);
f=@(x)cos(x)+sin(x);
g=@(x)([x.^4;x.^3;x.^2;x.^1;x.^0]);
C=cell(2*m,1);
for i=1:m
  C{i,1}  = affine( g(X(i)),-f(X(i)));
  C{i+m,1}= affine(-g(X(i)), f(X(i)));
end
[approx_error,a]=pcpsolve(fmax(C),space(5))
h=@(x)(a'*g(x));
x=-2.5:.1:2.5;
plot(x,h(x),x,f(x));
```

```
approx_error = 0
a =
   0.035220
  -0.128941
  -0.494918
   0.970412
   1.000000
```



9. qcsolve – Global Optimization Solver

The command qcsolve provides a solver for a class of global optimization problems with quasi-concave objective function. For more information about the theoretical background, the reader is referred to [3].

```
 -- [fmin,x] = qcsolve (S,fname,P,args)    solve quasi-concave program

    solve quasi-concave global optimization problem

    minimze f(Px) s.t. x in S

     where

     -- f is a function from R^p to [-inf,Inf) which is quasi-concave
        (i.e. f has convex super-level sets)
     -- P is a p times n matrix
     -- S is a polyhedral feasible set in R^n such that P[S] is bounded

    If f is monotone with respect to a polyhedral pointed convex cone C
    on dom f = {y | f(y) > -Inf}, then C can be used as optional input
    argument. Specifying such a cone can speed up the algorithm.
    Moreover, boundedness of P[S] can be weakened to C-boundedness of
    P[S]. C-monotonicity is not checked by the program and has to be
    ensured by the user.

    Input:
      S:      feasible set S (polyh object)
      fname:  name of the objective function (string)
              for requirements of the function itself, see below.
      P:      matrix
      args:   optional arguments:
        args.C:             monotonicity cone C (polyh object)
        args.opt.display    flag to display solution
    Output:
      fmin:  optimal value
      x:     optimal solution

    Remark: The objective function f is required to be given as
            Matlab/Octave function. It is not allowed to use functions of
            bensolve tools in the definition of f. A single argument
            of f is a column vector. It is important to guarantee, that
            multiple arguments are possible: If the input for f is a
            matrix X the output of f is expected to be a row vector
            the entries of which are the functions values of the
            corresponding columns of X.
```

Example 9.1 *Concave quadratic programming: Let $Q \in \mathbb{R}^{n\times n}$ be a positive semi-definite symmetric matrix. Then, the function $g\colon \mathbb{R}^n \to \mathbb{R}$ with $g(x) = -x^\mathsf{T} Q x$ is concave, hence quasi-concave. In order to minimize g under linear constraints, Q can be factorized, i.e. $Q = P^\mathsf{T} P$ for some matrix $P \in \mathbb{R}^{p\times n}$. We obtain an appropriated problem instance*

$$\min f(Px) \quad \text{s.t.} \quad x \in S$$

for `qcsolve` *with $f : \mathbb{R}^p \to \mathbb{R}$ with $f(y) := -y^\mathsf{T} y$.*

Let us solve this global optimization problem for the choice (compare [8, 3])

$$P_{ij} = \lfloor p \cdot \sin((j-1)\cdot p + i)\rfloor,$$

where $\lfloor x \rfloor := \max\{z \in \mathbb{Z} \mid z \le x\}$ and $S = \{x \in \mathbb{R}^n \mid -e \le x \le e\}$.

First, we need to store the objective function f in a file, say in `f.m`*:*

```
function y=f(x)
  y=-sum(x.^2);
end
```

To illustrate the preceding remark about multiple arguments of f, consider:

```
X=[1 2 3;3 4 5]
Y=f(X)
```

```
X =
    1     2     3
    3     4     5
Y =
  -10   -20   -34
```

The correct result is obtained with our function, whereas `y=-X'*X` *yields a wrong result in case* `X` *has more than one column.*

Next we generate the matrix P and the feasible region S and call `qcsolve`*:*

```
n=1000; p=6;
P=floor(p*sin(reshape(1:p*n,p,n)));
S=cube(n);
qcsolve(S,'f',P)
```

In order to run the dual algorithm described in [3], an option for bensolve *must be set:*

```
set_bensolve_option('a','dual');
```

See Section 12 for more details.

10. molpsolve – Multiple Objective Linear Programming Solver

The command `molpsolve` can be used to solve a multiple objective linear program of the form

$$\min Px \quad \text{s.t.} \quad x \in S. \tag{MOLP}$$

where the feasible set S is a polyhedron given as a `polyh` object, see Section 5, and P is a $q \times n$ matrix. For more information see e.g. [10].

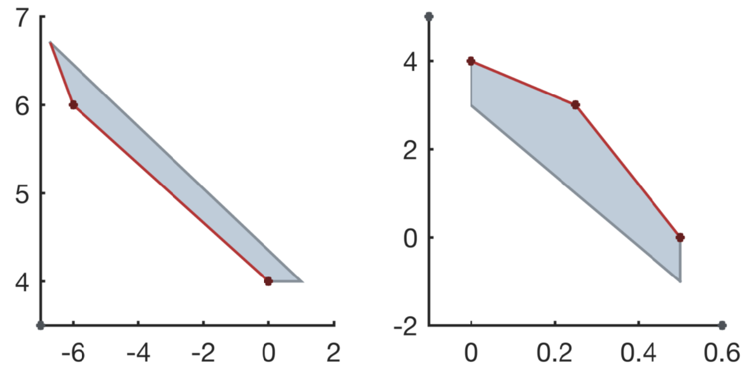
```
 -- [img_p,img_d,sol_p,sol_d]=molpsolve(P,S,optdir)    solve MOLP

    solve multiple objective linear program

    minimize Px  s.t  x in S

    where S is given by a P-representation:

    S = {Mx : l <= x <= u, a <= Bx <= b}

    Input:
      P:  objective matrix
      S:  feasible set (polyh object)
      optdir:  'min' (default) or 'max'
    Output:
      img_p:  extended image of the primal problem (polyh object)
      img_d:  extended image of the dual problem (polyh object)
      sol_p:  primal solution (matrix)
      sol_d:  dual solution (matrix)

    Remark:
      the dual solution refers to the P-representation of S and consists of
      row variables:     the first m entries
      column variables:  the next n entries
      weight variables:  the last q entries
      where [m,n] = size(S.prep.B) and q is the number of objectives
      see Section 1.7 at http://bensolve.org/files/manual.pdf
```

Example 10.1 *Consider the following MOLP with two objectives:*

$$\min \begin{pmatrix} x_1 - x_2 \\ x_1 + x_2 \end{pmatrix} \quad \textit{s.t.} \quad 6 \le 2x_1 + x_2,\ 6 \le x_1 + 2x_2,\ x_1 \ge 0,\ x_2 \ge 0$$

```
clear('rep');
rep.B=[2 1;1 2];
rep.a=[6;6];
rep.l=[0;0];
S=polyh(rep,'h');
P=[1 -1; 1 1];
[PP,DD,sol]=molpsolve(P,S);
ploteff(PP)
ploteff(DD)
```

The picture shows the upper image of (MOLP) *and the lower image of its dual problem. The Pareto frontier is shown in red. Note that, as usual, maximization w.r.t. the ordering cone* $K = \{x \in \mathbb{R}^2 \mid x_1 = 0,\ x_2 \geq 0\}$ *is done in the dual problem.*




```
sol =
  0   2   0   0
  6   2   1   0
```

To interpret the solution we need to know the number of vertices of the V-representation of `PP`*:*

```
size(vrep(PP).V,2)
```

```
ans =  2
```

There are two vertices, hence the first two columns of `sol` *are points and the third column is a direction. Zero directions are not part of the solution of* (MOLP)*. Here the fourth column of* `sol` *corresponds to*

```
[vrep(PP).V, vrep(PP).D](:,4)
```

```
ans =
   1
   0
```

The vector $(1, 0)^\mathsf{T}$ belongs to the ordering cone $\mathbb{R}^2_+$. Therefore it is not a minimal direction and does not belong to a solution, see e.g. [10] for further explanation.

11. vlpsolve – Vector Linear Programming Solver

The command `vlpsolve` provides a convenient way to use the VLP solver *bensolve*.

```
 -- [img_p,img_d,c,sol_p,sol_d]=vlpsolve(P,S,optdir,C,c)    solve VLP

    solve vector linear program

    minimize Px  s.t  x in S  w.r.t  ordering cone C

    where S is given by a P-representation:

    S = {Mx : l <= x <= u, a <= Bx <= b}

    Input:
      P        objective matrix
      S        feasible set (polyh object)
      optdir   'min' (default) or 'max'
      C        ordering cone (polyh object)
      c        duality parameter vector
    Output:
      img_p:    extended image of the primal problem (polyh object)
      img_d:    extended image of the dual problem (polyh object)
      c_ret:    duality parameter corresponding to dual solution
      sol_p:    primal solution (matrix)
      sol_d:    dual solution (matrix)

    Remark:
      the dual solution refers to the P-representation of S and consists of
      row variables:     the first m entries
      column variables:  the next n entries
      weight variables:  the last q entries
      where [m,n] = size(S.prep.B) and q is the number of objectives
      see Section 1.7 at http://bensolve.org/files/manual.pdf
```

Example 11.1 *Let the following code define the feasible set S of a vector linear program:*

```
clear('rep');
rep.B=[ones(1,3);1 2 2;2 2 1;2 1 2];
rep.a=[1;3/2;3/2;3/2];
rep.l=[0;0;0];
S=polyh(rep,'h');
```

The objective matrix of the vector linear program is:

```
P=[1  0  1;  1  1  0;  0  1  1];
```

Let the ordering cone C be generated by the vectors

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\\0\end{pmatrix} \quad \begin{pmatrix}-1\\0\\2\end{pmatrix} \quad \begin{pmatrix}0\\-1\\2\end{pmatrix},$$

which can be entered as:

```
clear ('rep');
rep.V=[0 0 0]';
rep.D=[1 0 0; 0 1 0; -1 0 2; 0 -1 2]';
C=polyh(rep,'v');
```

It is possible to specify a geometric duality parameter vector c. It must belong in the interior of C and must have a nonzero last component. If c is not specified, it is computed by the solver.

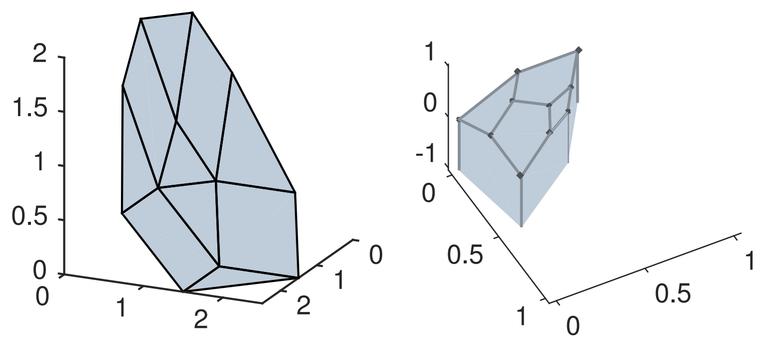
```
c=[2;2;2];
```

Now the vector linear programming solver is called.

```
[img_p,img_d,c,sol_p]=vlpsolve(P,S,'min',C,c);
```

We plot the upper image of the primal problem and the lower image of the dual problem:

```
plot(img_p);
plot(img_d);
```



Now we display a V-representation of the upper image of the primal problem:

```
vrep(img_p)
```

```
V =
   1.5   0.5   0.5   1
   1.5   1     0.5   0.5
   0     0.5   1     0.5
D =
   0     1    -0.5   0
   1     0     0    -0.5
   0     0     1     1
L = [](3x0)
```

The duality parameter vector is returned by the solver. Displaying the returned vector shows that c is scaled by the solver. The dual lower image is defined with respected to this scaled vector c.

```
c =
   1
   1
   1
```

Now a solution of the vector linear program is displayed:

```
sol_p
```

```
sol_p =
   1.5   0.5   0     0.5   0   0   0   0
   0     0.5   0.5   0     0   0   0   0
   0     0     0.5   0.5   0   0   0   0
```

A solution has to be interpreted together with the V-representation of the upper image of the primal problem. The upper image has four vertices, hence the first four columns of the solution are points and the remaining columns are directions. The V-representation of the optimal solution has four extremal directions, however all four belong to the ordering cone. Thus the last four zero columns are just place holders, they do not belong to the solution.

Further details can be found in [10] and in the *bensolve* reference manual[1].

Faces of the upper and lower images can be checked for being efficient (i.e. all their points are minimal or maximal w.r.t. some cone) be using the following command.

```
 -- flag = isefficient(F,P,C)   efficiency test

    Input:
      F: polyhedron (polyh object)
      P: polyhedron (polyh object)
    Optional input:
      C: pointed ordering cone (polyh object)
         default: standard cone
    Output:
      flag

    Test whether the polyhedron F consists of only efficient points
    of the polyhedron P w.r.t. the cone C.
```

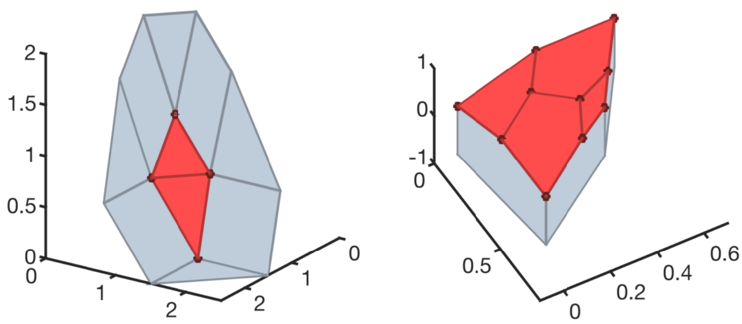
The efficient frontier can be visualized by the `ploteff` command.

[1] http://bensolve.org/files/manual.pdf

```
-- plotefficient(P,C,OPT)   plot efficient faces in different color

   Input:
     P: polyhedron (polyh object)
   Optional input:
     C: pointed polyhedral ordering cone (polyh object)
        default: standard cone
     OPT: options (struct)

   -----------------------------------------------------------------------
   option          default value       explanation
   -----------------------------------------------------------------------
   color           [3/4 4/5 17/20]     color nonefficient faces as [r,g,b]
   effcolor        [1 1/3 1/3]         color efficient faces as [r,g,b]
   edgewidth       1.2                 edge width
   dirlength       1                   lentgh of extremal directions
   vertexsize      1.3                 vertex size
   alpha           0.4                 transparency
   -----------------------------------------------------------------------

   Plot a polyhedron, where efficient faces are displayed in red.
```

In the above example we can highlight the C-minimal faces of the upper image and the K-maximal faces of the lower image of the dual problem, where as usual for the dual problem the cone $K = \{x \in \mathbb{R}^3 \mid x_1 = x_2 = 0,\ x_3 \geq 0\}$ is used.

```
ploteff(img_p,C);
K=origin(2):cone(1);
ploteff(img_d,-K);
```



12. Bensolve options

bensolve tools is based on the VLP solver *bensolve*. The command qcsolve is based on a modified variant of *bensolve*. An option for *bensolve* can be set globally using the command set_bensolve_option.

```
 -- set_bensolve_option(fn,val)    set options for bensolve

    Input:
      fn: fieldname (string)
      val: value (of different type)
    No input:
      reset of options

    -------------------------------------------------------------------------
    fn   val              explanation
    -------------------------------------------------------------------------
    'b'  0 1              assume VLP to be bounded (can be faster)
    'g'  0 1              enable global optimization mode (for internal use)
    's'  0 1              enable outut of solutions (pre-image information)
    'k'  * (see below)    simplex type in phase 0 of Benson's algorithm
    'L'  * (see below)    simplex type in phase 1 of Benson's algorithm
    'l'  * (see below)    simplex type in phase 2 of Benson's algorithm
    'm'  '0' '1' '2' '3'  display less or more messages
    'M'  '0' '1' '2' '3'  display less or more messages of internal lp solver
    'A'  'primal' 'dual'  type of Benson algorithm in phase 1
    'a'  'primal' 'dual'  type of Benson algorithm in phase 2
    'E'  e.g. '1e-6'      epsilon for Benson algorithm in phase 1
    'e'  e.g. '1e-6'      epsilon for Benson algorithm in phase 2
    -------------------------------------------------------------------------

    * 'primal_simplex' 'dual_simplex' 'dual_primal_simplex'
```

Example 12.1 *Increasing the message level of* bensolve*:*

```
set_bensolve_option('m','3');
A=eval(ball(2));
```

Choosing the dual variant of Benson's algorithm:

```
set_bensolve_option('a','dual');
A=eval(ball(2));
```

Reset to default options:

```
set_bensolve_option();
```

References

[1] H. Benson. An outer approximation algorithm for generating all efficient extreme points in the outcome set of a multiple objective linear programming problem. *Journal of Global Optimization*, 13:1–24, 1998.

[2] D. P. Bertsekas. *Convex optimization theory.* Belmont, MA: Athena Scientific, 2009.

[3] D. Ciripoi, A. Löhne, and B. Weißing. A vector linear programming approach for certain global optimization problems. *submitted to Journal of Global Optimization*, 2017. http://arxiv.org/abs/1705.02297.

[4] M. Ehrgott, A. Löhne, and L. Shao. A dual variant of Benson's outer approximation algorithm. *Journal of Global Optimization*, 52(4):757–778, 2012.

[5] A. H. Hamel, A. Löhne, and B. Rudloff. A Benson type algorithm for linear vector optimization and applications. *Journal of Global Optimization*, 59(4):811–836, 2014. see also http://arxiv.org/abs/1302.2415.

[6] J.-B. Hiriart-Urruty and C. Lemaréchal. *Fundamentals of convex analysis.* Berlin: Springer, 2001.

[7] A. Löhne. *Vector Optimization with Infimum and Supremum*. Vector Optimization. Springer, Berlin, 2011.

[8] A. Löhne and A. Wagner. Solving DC programs with a polyhedral component utilizing a multiple objective linear programming solver. *J. Global Optim.*, 2017. DOI: 10.1007/s10898-017-0519-8, see also http://arxiv.org/abs/1610.05470.

[9] A. Löhne and B. Weißing. Equivalence between polyhedral projection, multiple objective linear programming and vector linear programming. *Mathematical Methods of Operations Research*, 84(2):411–426, 2016. see also http://arxiv.org/abs/1507.00228.

[10] A. Löhne and B. Weißing. The vector linear program solver Bensolve – notes on theoretical background. *European Journal of Operational Research*, 2016. DOI: 10.1016/j.ejor.2016.02.039, see also http://arxiv.org/abs/1510.04823.

[11] R. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, 1972.

A. APPENDIX: GNU GENERAL PUBLIC LICENSE

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
   d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
   e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.