ggplab version 0.8 (beta)
A Matlab Toolbox for Geometric Programming

Almir Mutapcic
almirm@stanford.edu

Kwangmoo Koh
deneb1@stanford.edu

Seungjean Kim
sjkim@stanford.edu

Stephen Boyd
boyd@stanford.edu

September 26, 2005

ggplab is a Matlab-based toolbox for specifying and solving geometric programs (GPs) and generalized geometric programs (GGPs). It is intended to complement the survey paper, *A Tutorial on Geometric Programming* [BKVH], and the book *Convex Optimization* [BV04].

ggplab consists of

- gpcvx, a primal-dual interior-point solver for GP (in convex form) and a wrapper, gpposy, that accepts GPs given in posynomial form.
- A library of objects, such as monomials, posynomials, and generalized posynomials, to support the specification of GPs and GGPs.
- A variety of examples taken from [BKVH, BV04, BKPH, BKM].

Some caveats:

- The solver supports sparse problems, but is not optimized for very large scale problems.
- Object manipulation overhead can make ggplab slow on large problems.
- ggplab does not include support for dual variables. (The solver, however, does.)
- Our intention is to incorporate support for GP and GGP in the convex optimization toolbox cvx [GBY05]; after that, development of and support for ggplab will be minimal.
- We will be posting frequent updates of ggplab until version 1, so check the web site for new versions often.

ggplab was designed and implemented by Almir Mutapcic (object library), Kwangmoo Koh (solver), Seungjean Kim (solver), Lieven Vandenberghe (original version of solver), and Stephen Boyd (general trouble maker). Please send your feedback or report any bugs

to Almir Mutapcic (almirm@stanford.edu) for the object library, or to Kwangmoo Koh (deneb1@stanford.edu) for the solver.

The rest of this document describes the object library; the GP solver gpcvx, and the posynomial form wrapper gpposy, are described in separate documents.

1 Installing ggplab

To use ggplab you need Matlab 6.1 or later. Unpack the ggplab distribution in some convenient location, and add the ggplab directory to Matlab's path, as in

```
>> addpath /home/username/ggplab
```

(assuming you've unpacked the distribution in /home/username/).

2 GP objects

ggplab includes a library of objects for GP variables, monomials, posynomials, generalized posynomials, and GP constraints. In most cases where standard Matlab operators and functions make sense, they have been overloaded to work with GP objects.

2.1 GP variables

The gpvar command declares scalar GP variables and arrays of GP variables. For example, to create three scalar GP variables x, y, and z, and an array w of 25 GP variables, we use

```
>> gpvar x y z w(25);
```

The array constructor always creates a *column* vector of GP variables. Individual components of an array of variables can be accessed using the standard parenthesis notation. For example, w(20) is a scalar GP variable. These variables and arrays of variables are not numbers; they are Matlab objects. Typing a variable name to the Matlab prompt gives its name and dimension, if it is an array. The Matlab command gpvars lists all GP variables defined in the workspace.

2.2 Monomials, posynomials, and generalized posynomials

GP variables can be used to construct monomials, posynomials, and generalized posynomials. Monomials are created using multiplication, division, and powers (including sqrt), starting from positive constants, GP variables, or other monomials. For example, we construct monomials m1 and m2 as

```
>> m1 = 2*x^3*y^4/z^5;
>> m2 = 1/2*sqrt(m1)*w(1)^2*w(2)^(-1)*w(20)^4.5;
```

Posynomials can be constructed from positive constants, GP variables, monomials, and other posynomials, using addition and multiplication. Division is not allowed between posynomials, but a posynomial can be divided by a monomial to produce another posynomial. For example,

```
>> p1 = 1 + x^1.5*y^(-4)*z^2 + m1 + 4*x*y;
>> p2 = p1/m2;
```

creates two posynomials `p1` and `p2`.

You can form generalized posynomials from positive constants, GP variables, monomials, posynomials, and other generalized posynomials using addition, multiplication, positive powers, and maximum (using `max` function). You can also divide a generalized posynomial by a monomial. For example,

```
>> g1 = (4 + x^5.1 + m1)^2.4;
>> g2 = max( 1/x^2, m1^3, p1 );
```

The first line forms a generalized posynomial `g1` as a positive power of a posynomial. The second line defines a generalized posynomial `g2` as the maximum of two monomials (which are also posynomials) and a posynomial.

A positive integer positive power of a posynomial is another posynomial (when expanded). But `ggplab` treats all positive powers (integer or not) of posynomials as generalized posynomials. For example, `(x+y)^2` gives a generalized posynomial in `ggplab`, even though it can also be written as a posynomial, *i.e.*, `x^2+2*x*y+y^2` (which is indeed recognized as a posynomial in `ggplab`).

The commands `monomial`, `posynomial`, and `gposynomial` create an empty monomial, posynomial, and generalized posynomial object, respectively. They can be used when initializing loops that recursively build one of these functions. For example, the following loop creates the posynomial $p = \sum_{k=1}^{25} w_k^k$.

```
>> p = posynomial;  % create an empty posynomial
>> for k = 1:25
>>   p = p + w(k)^k;
>> end
```

2.3 Vectors and arrays

You can construct arrays of GP objects. For example, `[m1, m2]` forms a row vector of two monomials, and `[w(1) w(2); w(3) w(4)]` forms a 2×2 matrix of GP variables. `ggplab` supports Matlab's array operators and functions when they make sense in the context of GP. Standard (matrix) multiplication, and pointwise multiplication, of arrays are overloaded to work with appropriately sized arrays of GP objects. For example, if `w` is a column vector of GP variables, and `A` is an elementwise nonnegative matrix (with a column dimension equal to the dimension of `w`), then `s = A*w` defines `s` as an array of posynomials. As another example,

if `B` is an array of generalized posynomials, and `C` is an array (with the same dimensions) of monomials, then `D=B./C` defines an array of generalized posynomials.

Functions such as `max`, `sum`, `sqrt`, and `inv` work with GP object arrays just as you would expect them to work with double arrays. For example, consider the posynomial p defined above using a for loop. We can form the same posynomial using elementwise powers, and the `sum` function:

```
>> powers = [1:25]';
>> p = sum( w.^powers );
```

For matrices, the functions `max` and `sum` return row vectors containing the maximum and the sum of the GP objects from each column; `sqrt` and `inv` act elementwise.

Typing the name of a defined monomial, posynomial, and generalized posynomial to the Matlab prompt will give its name and expression, if it is a scalar, and its name and dimension, if it is an array (or a matrix).

2.4 GP constraints

There are two types of constraints in `ggplab`:

- *Generalized posynomial inequality constraints* have the form `f<=g`, where `f` is a generalized posynomial and `g` is a monomial. (`ggplab` does not use the `>=` operator, so you have to express `g>=f` as `f<=g`.)
- *Monomial equality constraints* have the form `g1==g2`, where `g1` and `g2` are monomials.

Of course you can use positive constants or GP variables as momonials, and positive constants, GP variables, monomials, or posynomials as generalized posynomials. For example, `x^2+sqrt(y)<=z` is a valid inequality constraint, and `x*y*z==1` is a valid equality constraint.

Inequality operators between GP objects (when valid) return *GP constraint objects*, which can be assigned. For example,

```
>> c1 = (x*y)^4 <= z^2;
>> c2 = p1 <= 1;
>> c3 = m1 == m2;
```

forms three valid GP constraints `c1`, `c2`, and `c3`.

You can form arrays of GP constraints, as in `constr = [c1; c2; c3]`, which creates a column vector of the three constraints defined above. You can add to an array of constraints, or change a specific constraint in array, using standard Matlab array operations. For example,

```
>> constr = [constr; max( x, y ) <= z];
```

adds a new constraint to our array, and

```
>> constr(3) = x*y == 2;
```

changes the third constraint in the array to `x*y == 2`.

Equality and inequality operators can be used on arrays of GP objects, to create arrays of GP inequalities. The arrays must have the same dimension, and the constraints are formed componentwise. For example,

```
>> lbound = ones(25,1) <= w; ubound = w <= 2*ones(25,1);
>> bounds = [lbound; ubound];
```

first creates two arrays of GP constraints, then combines them into one. The constraint array `bounds` consists of the 50 constraints that limit each variable `w(k)` to lie between one and two.

3 Solving GPs and GGPs

The `gpsolve` command calls the primal-dual interior-point solver `gpcvx` to solve GP and GGP problems. Its usage is

```
>> [obj_value, solution, status] = gpsolve(obj, constr_array, type)
```

`gpsolve` takes as input arguments

- a GP function `obj`, the objective function of the problem,
- an array of GP constraints `constr_array`,
- a string that specifies the problem type, which is either `'min'` (if the objective is to be minimized) or `'max'` (if the objective is to be maximized). This is an optional input; the default is `'min'`. If the problem type is `'max'`, then the objective must be a monomial.

`gpsolve` returns

- the optimal objective value `obj_value` (a number),
- a cell array of GP variable names and their optimal values, `solution`,
- the problem status flag, which can be `'Solved'` (if the optimization is successful, and an optimal point was found), `'Infeasible'` (if the problem was determined to be infeasible), or `'Failed'` (if the optimization was not successful).

The inputs can also be empty arrays. If the objective is an empty array or a constant, then `gpsolve` solves a feasibility problem. If the constraint array is empty, we have an unconstrained GP problem.

If the problem status is `'Solved'`, then `solution` contains an optimal solution. If the problem status is `'Infeasible'`, then `solution` contains the phase I solution found. This is *not* a solution of the original GGP that was specified; instead, it is the point found in

phase I of the optimization that came as close as possible to being feasible. When the status is `'Failed'`, the solution is an empty array.

The solution is returned as a $n \times 2$ cell array, where n is the number of GP variables in the problem. The first column consists of strings, which give the GP variable names; the second column consists of the values found by the solver. This is convenient for just looking at the optimal values of the variables; if you want to do some further processing, however, you might want to use the `eval` or `assign` commands described below.

3.1 Evaluating GP functions

GP functions (variables, monomials, posynomials, and generalized posynomials) do not have numerical values associated with them. However, we can evaluate a GP function for specific (numerical) values of GP variables using the `eval` command as in

```
result = eval(gp_object, gpvar_values_cell_array)
```

The second input is a Matlab cell array with two columns that specifies values of GP variables. The first column of the cell array contains the name of GP variable or array as a string, while the second column contains numeric value of the variable or the array. (You can think of this data structure as a hash table of GP variable names and their associated values.) An example cell array is

```
>> gpvar_values = {'x' 2; 'y' 5.44; 'w' ones(25,1)};
```

If the cell array does not specify values of all GP variables that appear in the GP object, then a reduced version of the GP object is returned. For example, the command `eval( x + z , gpvar_values )` returns the posynomial `2+z`. If the cell array does specify values for all the GP variables appearing in the GP object, then the object is evaluated, and `result` becomes an ordinary number, or a numerical array.

The `eval` command is useful after you've solved a GGP, to get a hold of the optimal values of some variables, or the values of some GP function. For example, the code

```
>> [obj_value, solution, status] = gpsolve(obj, constr_array, type);
>> ov = eval(obj,solution);
```

solves a GGP, then evaluates the obective at the optimal point found and assigns this number ot `ov`. (This should give the same result as `obj_value`, if the problem status is feasible.)

The Matlab command `assign` takes a cell array of specified GP variables, along with their values, as input and assigns each of the values to the variables. (In particular, it causes each of the GP variables to become a double.) For example, the command

```
>> assign(gpvar_values);
```

transforms GP variables `x`, `y`, and `w` into doubles (in the case of `w`, into a column vector of doubles). This command is useful to convert a solution of a GGP problem into numeric variables that can be used for further computation or plotting.

3.2 An example

We consider a simple GP given in [BKVH]. The problem is to maximize the volume of a box with height h, width w, and depth d, subject to some constraints. We have a limit on the total wall area $2(hw + hd)$, and the floor area wd, as well as lower and upper bounds on the aspect ratios h/w and w/d. This leads to the GP

$$\begin{array}{ll} \text{maximize} & hwd \\ \text{subject to} & 2(hw + hd) \leq A_{\text{wall}}, \quad wd \leq A_{\text{flr}}, \\ & \alpha \leq h/w \leq \beta, \quad \gamma \leq d/w \leq \delta. \end{array} \tag{1}$$

Here h, w, and d are the optimization variables, and the problem parameters are A_{wall} (the limit on wall area), A_{flr} (the limit on floor area), and $\alpha,\ \beta,\ \gamma,\ \delta$ (the lower and upper limits on the wall and floor aspect ratios).

The following Matlab script solves the box volume optimization problem (for some specific values of the problem data):

```
% problem data
Awall  = 10000; Afloor = 1000;
alpha = 0.5; beta = 2; gamma = 0.5; delta = 2;

% GP variables
gpvar h w d;

% objective function is the box volume
volume = h*w*d;

% set of constraints expressed as an array
constr = [ 2*(h*w + h*d) <= Awall;        % wall area limit
           w*d <= Afloor;                 % floor area limit
           alpha <= h/w; h/w <= beta;     % h/w aspect ratio limits
           gamma <= d/w; d/w <= delta;];  % d/w aspect ratio limits

% solve the GP
[max_volume,solution,status] = gpsolve(volume, constr, 'max')
% no semicolon after the gpsolve command, so
% max_volume, solution, and status will be printed

% convert the GP variables to doubles, the optimal values found
assign(solution);
```

Since we did not append a semicolon after the `gpsolve` command, we get the following output:

```
>> max_volume = 7.7458e+04
```

```
solution =

    'd'    [25.8187]
    'h'    [77.4588]
    'w'    [38.7312]

status = Solved
```

The `assign` command converts `d`, `h`, and `w` into doubles, each containing the optimal value found by the solver.

This code can be found in the `examples_ggplab` subdirectory of your `ggplab` distribution, along with many others. Other examples include: maximizing volume of a box with changing constraints [BKVH], floor planning [BKVH], Frobenious norm diagonal scaling [BV04], cantilever beam design [BV04], minimizing Peron-Frobenious norm [BV04], digital circuit sizing [BKPH], and optimal design of a two pole lowpass filter [BKM].

3.3 Solver options

By default, `ggplab` uses `gpcvx`, the primal-dual interior point solver included with `ggplab`. `ggplab` can also be use the commercial MOSEK GP solver `mskgpopt` [MOS05]. This is specified using the global variable `GP_SOLVER`. The lines

```
>> global GP_SOLVER
>> GP_SOLVER = 'mosek';
```

set the default solver to MOSEK. (Of course, you must have MOSEK installed, and a valid license, for `ggplab` to work with MOSEK.) To switch back to the default GP solver `gpcvx`, you need to clear the global variable `GP_SOLVER`,

```
>> clear global GP_SOLVER
```

or set it to an empty string using `GP_SOLVER = ''`.

References

[BKM] S. Boyd, S.-J. Kim, and S. Mohan. Geometric programming and its applications to EDA problems. Design and Test in Europe (DATE) 2005 tutorial. Available at `www.stanford.edu/~boyd/date05.html`.

[BKPH] S. Boyd, S.-J. Kim, D. Patil, and M. Horowitz. Digital circuit optimization via geometric programming. To appear in *Operations Research 2005*. Available at `www.stanford.edu/~boyd/gp_digital_ckt.html`.

[BKVH] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. To appear in *Optimization and Engineering, 2005*. Available at `www.stanford.edu/~boyd/gp_tutorial.html`.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. Available at `www.stanford.edu/~boyd/cvxbook.html`.

[GBY05] M. Grant, S. Boyd, and Y. Ye. *CVX: Matlab Software for Disciplined Convex Programming, version 0.8 alpha (July 2005)*, July 2005.

[MOS05] MOSEK ApS. *MOSEK (software package)*, 2005.