cparse User's Guide

V. Visweswaran and C. A. Floudas
Department of Chemical Engineering
Princeton University
Princeton, NJ 08544

VERSION 1.0

March 20, 1996

Copyright © 1995 V. Visweswaran and C. A. Floudas

All Rights Reserved.

NOTE: This is a beta-test version and is intended primarily for internal and limited external distribution. This version has not been extensively tested, and the features described here are likely to be changed in future versions. The opinions and conclusions stated here are those of the authors and do not reflect the opinions of the Department or Princeton University.

1. Introduction

cparse[1] provides a user-friendly interface to **cGOP**. It allows the problems for **cGOP** to be specified in a very high level form without any need for recompilation. It also has several advanced features such as sets, parameters and arbitrary names for variables and constraints. It can be used to specify arbitrary nonlinear expressions. It works by converting the data file to intermediate input files that can be read by **cGOP**.

To use **cparse**, a data file for the problem first needs to be created (the format is described below). If the name of this file is "test.dat", for example, then the command

cparse test.dat

will run the parser on the data file, which will (if everything is correctly specified in the file) produce the file "test.gop". This file can then be used with **cGOP** as described in the **cGOP** manual. If there are nonlinear terms in the objective function, then an additional file ".funobj.bin" will be created that contains binary instructions for recreating the nonlinear terms. This file will be automatically read in by **cGOP** when it is run, so you should not have to do anything with this file.

2. Data File

cparse reads the problem from a high-level data file, and produces an input file that can be read directly by **cGOP**. The data files are simple text files that can be created and edited by any standard text editor. In this section, we discuss the various sections of such a data file.

The **cparse** input files are divided into several sections. Each section contains one or more statements, describing different aspects of the problem. Although the order of the sections in the input files are fixed, the statements in each section can be in any order[2], provided that all the variable and parameter names in the current statements have been previously defined.

2.1. Comments

Comments can be easily added to the input files by adding a '#' or a '$' at the beginning of the lines that are to be treated comments. C-style comment i.e. '/* ... */' are also acceptable. The comment lines which are preceded with '$' are to be echoed to the screen during **cparse** runs.

2.2. Keywords, Identifiers & Expressions

The three most important components of the data files are keywords, identifiers and expressions. Keywords are words in uppercase letters that have special meanings in **cparse**. Keywords have

[1] **cparse** is derived from the front end of MINOPT, a package written by A. Rojnukarin for mixed-integer nonlinear optimization.

[2] An exception to this is in the constraints section; the objective function *must* precede the constraints

fixed meanings and cannot be redefined. Identifiers are "user-defined" names consisting of lowercase letters and numbers (must begin with lower case-letters) that can be used as variable names, parameter (constant) names or indices. Names with mixed lower and uppercase letters are not acceptable in **cparse**. Though keywords and identifiers can be of any length, currently, only the first three letters of keywords and first seven letters of identifiers are meaningful to **cparse**. Parameters and variables can be combined into mathematical expressions. **cparse** understands common mathematical operators, such as $+$, $-$, $*$, / and $**$ (power), as well as some mathematical functions such as sqrt, exp, sin, cos and ln.

2.3. Indices, Sets, Summation & Enumeration

As in C "array" declarations, when a variable or a parameter is to be declared with indices, the number of dimensions, and the size of each dimension are specified. In **cparse** up to four indices (i.e four dimensions) can be used with a particular variable or parameter. Then similar to C, we can refer to a variable, say w, by using variable name and indices, for example $w(0)$, $w(1)$ and so on, and similarly, refer to a multi-dimension variable, say a two-dimension (two indices) variable v, by $v(0,0)$, $v(0,1)$ and so on. Although each w can be declared separately as $w0$, $w1$ and so forth, if the nature of each of the variables is similar, it is more convenient to combine them into one variable to take advantage of "set indices" system to be described in the remains of this section.

cparse employs a rudimentary "set indices" notation system. In **cparse** when a set is declared, only its size is specified; the members of the sets are automatically defined to be 0, 1, 2, and so on up to the size of the set minus one. After a set is defined, it can be used in conjunction with variables, equations or parameters. For example, an index set i when declared to have n members can be used in conjunction with a variable w as $w(i)$ to denote $w(0)$, $w(1)$ up to $w(n-1)$. Two, three or four dimensional variables can also be used with any set indices in a similar manner.

In **cparse**, "Enumeration" or "Summation" operators are used to control the behavior of variables and equations described over sets. The summation operator is denoted as

```
<< set | argument >>
```

and sums up its argument over the given set. For example, the expression

```
<< i | w(i) >>
```

represents the summation of w over the set i, i.e. $w(0) + ... + w(n-1)$. The enumeration operator is denoted as

```
< set | argument >
```

and is used to loop through each member of the given set. It can be used to represent many related expressions (compared to the summation operator which results in a single expression) in a very compact way. Thus, the expression

```
< i | << j | c(i,j) * y(i,j) >> >
```

denotes n expressions: $\sum_j c(0,j)\times y(0,j), \sum_j c(1,j)\times y(1,j)$ and so on. The enumeration operator is very useful in specifying blocks of related constraints in a single statement. The use of summation and enumeration to declare objective function and constraints will be described in subsequent sections that include illustrative examples.

2.4. Input Sections

Each data file can be divided into four sections: Declaration, Option, Variable, and Constraint. The declaration section describes the problem in general, declaring problem name, number of variables, number of constraints and nonlinearity of the problem. The option section is used to specify optional parameters such as tolerances. The variable section contains the declaration of variables, type of variables (x-variables, y-variables), bounds on the variables, set indices, as well as parameters. Lastly, the constraint section describes the objective function and constraints.

Each section is of the form:

```
Section_Name {{
statements ; statements ;
...
}}
```

where `Section_Name` is the name (Keyword) of the corresponding section. The keywords for the four sections are DECLARATIONS, OPTIONS, VARIABLES and CONSTRAINTS respectively. Note that since the section names are also regular keywords, only the first three characters are necessary and meaningful. Thus, DEC, DECL and DECLARATIONS all refer to the Declarations section. All sections are required in the input files, and the order of the sections is fixed. Each section consists one or more statements, which are separated from one another by semi-colons. The details of statements in each section will be described below.

2.4.1. Declaration Section

In this section, important information about the problem must be provided. Statements in this section are of the form:

```
Keyword = String
```

or

```
Keyword = Number
```

where `String` can be any character string (up to 79 characters long) inside a pair of double-quotes. The keywords in this section describe the problem name, type, number of x and y variables and the number of linear and nonlinear constraints, and are described in Table 1. All the keywords *must* be provided. Currently, NNCON *must* be set to 0.

The string associated with "TYPE" keyword consists of two letters, with the first letter representing the type of the primal problem, and the second letter representing the type of the relaxed

Keyword	RHS Type	Meanings
NAME	String	Problem name
TYPE	String	Type of primal & master (see text)
NXVAR	Number	Number of x variables
NYVAR	Number	Number of y variables
NLCON	Number	Number of linear constraints
NNCON	Number	Number of non-linear constraints

Table 1: Keywords in declarations section.

dual problem. The types of the subproblems can be either "L" for linear or "N" for non-linear problems. Thus, the problem type "NL" denotes a nonlinear (i.e. convex) primal problem and a linear relaxed dual problem.

2.4.2. Options Section

This section is used to specify various options for the algorithm. Statements in this section are of the form:

```
Keyword = String
```

Currently, only three keywords are allowed here; these are described below.

EPSA Absolute tolerance for convergence, specified as a real number.

EPSR Absolute tolerance for convergence, specified as a real number.

MAXIT Maximum number of iterations, specified as an integer.

2.4.3. Variable Declaration Section

This section is used to declare variables, bounds and parameters. There are two inputs formats in this section:

```
Keyword List
```

and

```
Keyword Identifier = List
```

A list is basically a group of numbers, or variables. A list can be explicitly defined by using the format

```
{ elem1, elem2, elem3, ... }
```

Another way of defining a list is through the enumeration operator, using the format

Keywords	Format	Meanings
XVAR	1	assign x variables
YVAR	1	assign y variables
STDX	1	assign set index
XUBD	1	x upper bounds
XLBD	1	x lower bounds
YUBD	1	y upper bounds
YLBD	1	y lower bounds
PARA	2	set identifier
UBD	2	set upper bounds of a variable
LBD	2	set lower bounds of a variable

Table 2: Keywords in Variable sections. (Format 1 refers to `Keyword List` and 2 refers to `Keyword Identifier = List`)

```
@< set | expressions >
```

Identifiers in the `expressions` must be parameters (not variables) so that the expressions can be evaluated over a given set at that point and yield a list of size n, where n is the size of the set being enumerated over. These Keywords are defined in Table 2, and the examples of their usages follow.

Here are the examples for variables' declaration:

```
XVAR {temp, prod(4), feed(5,5)};
```

This defines variables `temp`, `prod(0)` to `prod(3)` and `feed(0,0)` to `feed(4,4)` as x variables. Note that indexes go from 0 to $n-1$, where n is the size of the index set.

```
STDX {i(4), i2(4), j(5), k(5)};
```

This defines index `i` to run from 0 to 3, index `j` to run from 0 to 4 and so on. Then these indexes can be used with variables previously declared, for example, `prod(i)` or `feed(j,k)`, provided that their sizes are consistent with each other, i.e `prod(j)` is not allowed. As long as the set indices have the same number of members, they are interchangeable, namely `prod(i)` and `prod(i2)` can be used interchangeably.

Bounds on x-type variables, or y-type variables can also be specified through a list using the first format (See Table 2). The order of bounds specified in the bound list depends on the way the variables are originally defined. The first number in the bound list must refer to the first variable declared through `XVAR` or `YVAR`, or the first element of the variable if the variables have any indices. Within a single variable, row major ordering system is used, that is, indices (0,0,0,0) comes first, then (0,0,0,1), and so on. As the bound (or bounds if the variables have more than one element) on the first variable is specified, the second variable follows and so on. The example which follows may help clarifying this matter.

```
YVAR {x,y,z(2,2)} ;
YLBD {1,2,1,5,1,1} ;
```

This defines x, y and z as y-type variable, and set the lower bounds on x and y variables to 1 and 2 respectively, while the lower bounds on z(0,0) is set to 1, z(0,1) to 5 and so on. Users may opt for another equivalent bound declaration format which puts bounds on one variable at a time, rather than all x-type or y-type variable using the format

```
LBDS z(2,2) = {1,5,1,1};
```

which puts the same bounds on z as the declaration using `YLBD` keyword shown above.

Parameter substitutions can also be used in **cparse** input files. The examples for the definition of parameters are as follows:

```
PARAM c(5) = {1,2,3,4,5};
PARAM q(4) = {1.2,3.4,4.7,5.2};
PARAM f(2,2) = {1,2,3,4};
PARAM d(5) = @<j|c(j)*2>;
```

This first and second lines define a parameter `c` that has five members and can be used as either coefficients or right hand side of any set of constraints (`c(0)`=1,`c(1)=2`, and so on), and a parameter `q`, so that `q(0)` contains 1.2, `q(1)` contains 3.4 and so on. The third lines does the same thing with a two dimensional variable. Again row major ordering system is used. Here `f(0,0)=1`, `f(0,1)`=2, `f(1,0)`=3 and `f(1,1)=4`. The last line assigns `d(5)` to `c(j)*2` where `j` as defined above, runs from 0 to 4. As a result, values that `d` holds are 2,4,6,8 and 10.

2.4.4. Constraint Declarations Section

The objective function is declared by

```
MIN : Expression
```

The expression can be constructed simply using the variable names, or with complex terms involving sets and summations. Note that the expression can only have linear or bilinear terms in x and y variables. When nonlinear terms are to be specified in the objective, then two additional types of terms can be specified after the "MIN" line. These are used to describe the nonlinear terms in x and y respectively in the objective, and are specified with the keywords "GOX" and "GOY". Note that the expression associated with "GOX" *must* be a function of x variables only, and the expression associated with "GOY" *must* be a function of y variables only.

For example, the expression

```
MIN:  - x + xy - y;
```

specifies the objective function to be $-x + xy - y$, while

```
MIN:  xy;
GOX: x^2;
GOY: y^2;
```

specifies the objective function for the CGOP program to be $xy + x^2 + y^2$. When such terms are specified, then the parser will generate an additional file (currently called ".funobj.bin") containing binary instructions for recreating the nonlinear terms. These terms are then read in by the user function when **cGOP** is run; you do not need to recompile **cGOP** in order to use these terms.

Constraints can be declared by

```
Constraint_Name : Expression Relation Right_Hand_Side
```

Constraint names are treated as identifiers, that is, they must be in lowercase letters. Currently, only linear constraints are allowed by the parser; all constraint names must begin with the letter "l". Relations must be "<=", ">=" or "==" to denote less-than, equal-to or greater-than type constraints respectively.

As mentioned above, **cparse** recognizes regular mathematical operators, i.e. `+`, `*` etc. Moreover **cparse** also recognizes some mathematical functions. To distinguish mathematical functions from variable indices, square brackets are used for functions. Mathematical functions that can currently be handled by **cparse** include `log[x]`, `exp[x]`, `sqrt[x]`, `sin[x]` and `cos[x]`. Summation over set indices as well as enumeration can also be used in this section.

Below are a few examples of constraint declarations. All variables, indices and parameters used in the examples are consistent with their declarations in the previous examples in section 2.4.3.

The declaration statement,

```
MIN: q(0)*prod(0) + q(1)*prod(1) + q(2)*prod(2) + q(3)*prod(3) ;
```

simply declares the objective function to be a sum of variable `prod`. In its equivalent notation using the summation operator, the objective function can be defined as:

```
MIN: << i | q(i)*prod(i) >> ;
```

Similarly the statement

```
lin1: prod(0) + prod(1) + prod(2) + prod(3) <= 100 ;
```

which declares a constraint called "lin1" to specify that the sum of prod variables can be no more than 100, can be replaced by

```
lin1: << i | prod(i) >> <= 100 ;
```

On the other hand, the declaration, using the enumeration operator,

```
lin2: < j | << k | feed(j,k) >> > == 1 ;
```

declares, in a single statement, four constraints (size of set j), which specify that for each j, the sum of `feed(j,k)` over the set k should be equal to one. The "1" on the right hand side of the constraints above can be replaced by a parameter, such as c to set the right hand side of "lin2(0)" to be "c(0)", "lin2(1)" to be "c(1)" and so on.

3. Examples

Here we present some example data files that can be read in by **cparse** to provide input files for **cGOP**.

3.1. Example 1: A Bilinear Problem

Consider the following problem:

$$\begin{aligned} \min_{x,y} \quad & -x + xy - y \\ s.t. \quad & -6x + 8y \leq 3 \\ & 3x - y \leq 3 \\ & 0 \leq x, y \leq 5 \end{aligned}$$

It can be specified in a data file as shown in Figure 1. The file can be found as *bilin1.dat* in the cparse subdirectory.

3.2. Example 2: Another Bilinear Problem

Consider the following problem:

$$\min \quad x_1 - x_2 - y_1 - (x_1 - x_2)(y_1 - y_2)$$

$$\begin{aligned} s.t \quad & x_1 + 4x_2 \leq 8 \\ & 4x_1 + x_2 \leq 12 \\ & 3x_1 + 4x_2 \leq 12 \\ & 2y_1 + y_2 \leq 8 \\ & y_1 + 2y_2 \leq 8 \\ & y_1 + y_2 \leq 5 \\ & x_1, x_2, y_1, y_2 \geq 0 \end{aligned}$$

The data for this problem as specified using matrices and sets is shown in Figure 2. The file can be found as *bilin2.dat* in the cparse subdirectory.

3.3. Example 3: Solving an Indefinite Quadratic Problem with cGOP

This example is a large indefinite quadratic programming problem, and is taken from Floudas and Pardalos (1990).

$$\min \quad f(x,y) = f_c - \sum_{i=1}^{10} (\frac{1}{2} Q_i x_i y_i + c_i y_i) + f_1(y)$$

```
DECLARATIONS {{
    NAME  = "bilin1";
    TYPE  = "LL";
    NXVAR = 1;
    NYVAR = 1;
    NLCON = 3;
    NNCON = 0;
}}

OPTIONS {{
    MAXIT = "30";
    EPSA = "1.0e-3";
    EPSR = "1.0e-3";
}}

VARIABLES {{
    XVAR {x1}; YVAR {y1};
    LBD x1 = {0.0};  UBD x1 = {5};
    LBD y1 = {0.0};  UBD y1 = {5};
}}

CONSTRAINTS {{
    MIN: -x1 + x1*y1 - y1;
    lcon1: -6*x1 + 8*y1 <= 3;
    lcon2: -6*x1 + 8*y1 <= 3;
}}
```

Figure 1: Data file for Example 1

```
DECLARATIONS {{
    NAME  = "bilin2";                          # Name of problem
    TYPE  = "LL";                              # Linear primal/dual subproblems
    NXVAR = 2;                                 # Number of X variables
    NYVAR = 2;                                 # Number of Y variables
    NLCON = 6;                                 # Number of linear constraints
    NNCON = 0;                                 # Nonlinear constraints
}}

OPTIONS {{
    MAXIT = "30";                              # Maximum iterations
    EPSA = "1.0e-3";                           # Absolute convergence tolerance
    EPSR = "1.0e-3";                           # Relative convergence tolerance
}}

VARIABLES {{
    STDX  {i(2), j(2), k(3)};                  # Define sets to be used later
    XVAR  {x(2)}; YVAR {y(2)};                 # Declare variables
    LBDS  x(2) = {0};  UBDS x(2) = {3};        # Set all bounds on X
    LBDS  y(2) = {0};  UBDS y(2) = {4};        # Set all bounds on Y
    PARAM ax(3, 2) = {1, 4, 4, 1, 3, 4};      # Coefficients in X constraints
    PARAM ay(3, 2) = {2, 1, 1, 2, 1, 1};      # Coefficients in Y constraints
    PARAM bx(3)    = {8, 12, 12};              # Constraint RHS's
    PARAM by(3)    = {8, 8, 5};                # Constraint RHS's
    PARAM cx(2)    = {1, -1};                  # Linear X terms in objective
    PARAM cy(2)    = {-1, 0};                  # Linear Y terms in objective
    PARAM q(2,2)   = {-1, 1, 1, -1};           # Bilinear terms matrix
}}

CONSTRAINTS {{
    MIN:   << i | cx(i)*x(i) + cy(i)*y(i) + << j | x(i)*q(i,j)*y(j) >> >> ;
    lcon1: < k | << i | x(i)*ax(k, i) >> > <= bx(k);
    lcon2: < k | << i | y(i)*ay(k, i) >> > <= by(k);
}}
```

Figure 2: Data file for Example 2

$$\begin{aligned} s.t. \quad Ay &\leq b \\ x_i - y_i &= 0 \quad i = 1, 2, \ldots, 10 \\ x, y &\geq 0 \end{aligned}$$

where

$$\begin{aligned} f_c &= -363762.0, \\ Q &= \text{Diag}\,(63, 15, 44, 91, 45, 50, 89, 58, 86, 82) \\ c &= (1197, 405, 1012, 4823, 1890, 1300, 2937, 1334, -3526, -1558) \\ b &= (380, 415, 385, 405, 470, 415, 400, 460, 400, 200) \\ f_1(y) &= \sum_{i=11}^{20} d_i (y_i - \overline{y_i})^2 \\ d &= (21, 49, 24, 45.5, 5.5, 31.5, 30.5, 30.5, 19, 13) \\ \overline{y} &= (-52, -3, 81, 30, -85, 68, 27, -81, 97, -73) \end{aligned}$$

and

$$A = \begin{bmatrix} 3 & 5 & 5 & 6 & 4 & 4 & 5 & 6 & 4 & 4 & 8 & 2 & 4 & 1 & 1 & 1 & 2 & 1 & 7 & 3 \\ 5 & 4 & 5 & 4 & 1 & 4 & 4 & 2 & 5 & 2 & 3 & 6 & 1 & 7 & 7 & 5 & 8 & 7 & 2 & 1 \\ 1 & 5 & 2 & 4 & 7 & 3 & 1 & 5 & 7 & 6 & 1 & 7 & 2 & 4 & 7 & 5 & 3 & 4 & 1 & 2 \\ 3 & 2 & 6 & 3 & 2 & 1 & 6 & 1 & 7 & 3 & 7 & 7 & 8 & 2 & 3 & 4 & 5 & 8 & 1 & 2 \\ 6 & 6 & 6 & 4 & 5 & 2 & 2 & 4 & 3 & 2 & 7 & 5 & 3 & 6 & 7 & 5 & 8 & 4 & 6 & 3 \\ 5 & 5 & 2 & 1 & 3 & 5 & 5 & 7 & 4 & 3 & 4 & 1 & 7 & 3 & 8 & 3 & 1 & 6 & 2 & 8 \\ 3 & 6 & 6 & 3 & 1 & 6 & 1 & 6 & 7 & 1 & 4 & 3 & 1 & 4 & 3 & 6 & 4 & 6 & 5 & 4 \\ 1 & 2 & 1 & 7 & 8 & 7 & 6 & 5 & 8 & 7 & 2 & 3 & 5 & 5 & 4 & 5 & 4 & 2 & 2 & 8 \\ 8 & 5 & 2 & 5 & 3 & 8 & 1 & 3 & 3 & 5 & 4 & 5 & 5 & 6 & 1 & 7 & 1 & 2 & 2 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The data file for this problem can be found in the file *indef1.dat* in the cparse directory.