*CLP-1-16-9-standalone-help-text.docx / pdf*

This is a manually reformatted version of the help text which can be generated from a standalone version of CLP 1.16.9. After going to some trouble (I do not have public documentation of how I did it), I was able to build a program along the lines of CLPUnitTest under Windows with the Microsoft VS2013 compiler. I think it is much easier to do this under Linux.

As far as I know, the following contains the complete help text which the program can generate. It seems to be more than just help - I think it reports the current state of various settings in the program.

This text is not available as a standalone document anywhere I know of. The source code for at least some of it can be found at, for instance:

<https://projects.coin-or.org/Clp/browser/trunk/Clp/src/CbcOrClpParam.cpp>

The primary documentation for the standalone program is Chapter 5 of:

http://www.coin-or.org/Clp/userguide/clpuserguide.html

(Set browser View > Text encoding = Western.)

All that follows is covered by CLP’s copyright licence as explained at http://www.coin-or.org/projects/Clp.xml.

The help text which follows may be useful in understanding the operation of the solver itself in the CLP library, even if one does not use the standalone executable.

Most of the settings in the standalone executable are presumably the same as those of the bare solver in the CLP library, but there is one exception I am aware of: pertub(ation) is off by default in the library and on by default in the standalone executable.

Robin Whittle rw@firstpr.com.au 2016-09-25

http://www.firstpr.com.au/linear-programming/

Coin LP version 1.16.9, build May 21 2016

Clp takes input from arguments ( - switches to stdin)

Enter ? for list of commands or help

Clp:?

In argument list keywords have leading - , -stdin or just - switches to stdin

One command per line (and no -)

abcd? gives list of possibilities, if only one + explanation

abcd?? adds explanation, if only one fuller help

abcd without value (where expected) gives current value

abcd value sets value

Commands are:

**Double parameters**:

dualB(ound)

dualT(olerance)

preT(olerance)

primalT(olerance)

primalW(eight)

sec(onds)

zeroT(olerance)

**Int parameters**:

idiot(Crash)

log(Level)

maxF(actor)

maxIt(erations)

output(Format)

randomS(eed)

sprint(Crash)

**Keyword parameters**:

allC(ommands)

chol(esky)

crash

cross(over)

direction

error(sAllowed)

fact(orization)

keepN(ames)

mess(ages)

perturb(ation)

presolve

printi(ngOptions)

scal(ing)

timeM(ode)

**Actions or string parameters**:

allS(lack)

barr(ier)

basisI(n)

basisO(ut)

directory

dualS(implex)

either(Simplex)

end

exit

export

gsolu(tion)

help

import

max(imize)

min(imize)

para(metrics)

primalS(implex)

printM(ask)

quit

restoreS(olution)

saveS(olution)

solu(tion)

solv(e)

stat(istics)

stop

This is only the basic commands.

By using: allC all the ? command produces the full list:

Clp:allC all

Clp:?

In argument list keywords have leading - , -stdin or just - switches to stdin

One command per line (and no -)

abcd? gives list of possibilities, if only one + explanation

**abcd??** adds explanation, if only one fuller help

**abcd** without value (where expected) gives current value

**abcd value** sets value

Commands are:

**Double parameters**:

dualB(ound)

dualT(olerance)

objective(Scale)

preT(olerance)

primalT(olerance)

primalW(eight)

reallyO(bjectiveScale)

rhs(Scale)

sec(onds)

zeroT(olerance)

**Int parameters**:

cpp(Generate)

decomp(ose)

dense(Threshold)

dualize

idiot(Crash)

log(Level)

maxF(actor)

maxIt(erations)

moreS(pecialOptions)

output(Format)

passP(resolve)

pertV(alue)

pO(ptions)

preO(pt)

randomS(eed)

slp(Value)

small(Factorization)

special(Options)

sprint(Crash)

subs(titution)

verbose

**Keyword parameters**:

allC(ommands)

auto(Scale)

biasLU bscale

chol(esky)

crash

cross(over)

direction

dualP(ivot)

error(sAllowed)

fact(orization)

gamma((Delta))

keepN(ames)

KKT

mess(ages)

perturb(ation)

PFI

presolve

primalP(ivot)

printi(ngOptions)

scal(ing)

spars(eFactor)

timeM(ode)

vector

**Actions or string parameters**:

allS(lack)

barr(ier)

basisI(n)

basisO(ut)

directory

dirSample

dirNetlib

dirMiplib

dualS(implex)

either(Simplex)

end

environ(ment)

exit

export

fakeB(ound)

gsolu(tion)

help

import

max(imize)

min(imize)

miplib

netlib

netlibB(arrier)

netlibD(ual)

netlibP(rimal)

netlibT(une)

network

para(metrics)

plus(Minus)

primalS(implex)

printM(ask)

quit

restoreS(olution)

reallyS(cale)

restore(Model)

reverse

saveM(odel)

saveS(olution)

sleep

solu(tion)

solv(e)

stat(istics)

stop

tightLP

unitTest

userClp

Below are the somewhat reformatted full ?? help texts for all these commands. These were generated by pasting a bunch of lines to the console, with each line of the form:

dualB??

dualT??

objective??

**Double parameters**

**-----------------**

**dualB**(ound) : Initially algorithm acts as if no gap between

bounds exceeds this value

The dual algorithm in Clp is a single phase algorithm as opposed to

a two phase algorithm where you first get feasible then optimal.

If a problem has both upper and lower bounds then it is trivial to

get dual feasible by setting non basic variables to correct bound.

If the gap between the upper and lower bounds of a variable is more

than the value of dualBound Clp introduces fake bounds so that it

can make the problem dual feasible. This has the same effect as a

composite objective function in the primal algorithm. Too high a

value may mean more iterations, while too low a bound means the code

may go all the way and then have to increase the bounds. OSL had

a heuristic to adjust bounds, maybe we need that here.

<Range of values is 1e-020 to 1e+012; current 1e+010>

**dualT**(olerance) : For an optimal solution no dual infeasibility

may exceed this value

Normally the default tolerance is fine, but you may want to increase

it a bit if a dual run seems to be having a hard time. One method

which can be faster is to use a large tolerance e.g. 1.0e-4 and dual

and then clean up problem using primal and the correct tolerance

(remembering to switch off presolve for this final short clean up

phase).

<Range of values is 1e-020 to 1e+012; current 1e-007>

**objective**(Scale) : Scale factor to apply to objective

If the objective function has some very large values, you may wish

to scale them internally by this amount. It can also be set by

autoscale.

It is applied after scaling. You are unlikely to need this.

<Range of values is -1e+020 to 1e+020; current 1>

**preT**(olerance) : Tolerance to use in presolve

The default is 1.0e-8 - you may wish to try 1.0e-7 if presolve says

the problem is infeasible and you have awkward numbers and you are

sure the problem is really feasible.

<Range of values is 1e-020 to 1e+012; current 1e-008>

**primalT**(olerance) : For an optimal solution no primal infeasibility

may exceed this value

Normally the default tolerance is fine, but you may want to increase

it a bit if a primal run seems to be having a hard time

<Range of values is 1e-020 to 1e+012; current 1e-007>

**primalW**(eight) : Initially algorithm acts as if it costs this much to

be infeasible

The primal algorithm in Clp is a single phase algorithm as opposed

to a two phase algorithm where you first get feasible then optimal.

So Clp is minimizing this weight times the sum of primal infeasibilities

plus the true objective function (in minimization sense). Too high

a value may mean more iterations, while too low a bound means the

code may go all the way and then have to increase the weight in order

to get feasible. OSL had a heuristic to adjust bounds, maybe we need

that here.

<Range of values is 1e-020 to 1e+020; current 1e+010>

**reallyO**(bjectiveScale) : Scale factor to apply to objective in place

You can set this to -1.0 to test maximization or other to stress code

<Range of values is -1e+020 to 1e+020; current 1>

**rhs**(Scale) : Scale factor to apply to rhs and bounds

If the rhs or bounds have some very large meaningful values, you may

wish to scale them internally by this amount. It can also be set

by autoscale. This should not be needed.

<Range of values is -1e+020 to 1e+020; current 1>

**sec**(onds) : Maximum seconds

After this many seconds clp will act as if maximum iterations had

been reached (if value >=0).

<Range of values is -1 to 1e+012; current -1>

**zeroT**(olerance) : Kill all coefficients whose absolute value is

less than this value

This applies to reading mps files (and also lp files if

KILL\_ZERO\_READLP defined)

<Range of values is 1e-100 to 1e-005; current 1e-020>

**Int parameters**

**--------------**

**cpp**(Generate) : Generates C++ code

Once you like what the stand-alone solver does then this allows you

to generate user\_driver.cpp which approximates the code. 0 gives

simplest driver, 1 generates saves and restores, 2 generates saves

and restores even for variables at default value. 4 bit in cbc generates

size dependent code rather than computed values. This is now deprecated

as you can call stand-alone solver - see Cbc/examples/driver4.cpp.

<Range of values is -1 to 50000; current -1>

**decomp**(ose) : Whether to try decomposition

0 - off, 1 choose blocks >1 use as blocks Dantzig Wolfe if primal,

Benders if dual - uses sprint pass for number of passes

<Range of values is -2147483647 to 2147483647; current 0>

**dense**(Threshold) : Whether to use dense factorization

If processed problem <= this use dense factorization

<Range of values is -1 to 10000; current -1>

**dualize** : Solves dual reformulation

Don't even think about it.

<Range of values is 0 to 4; current 3>

**idiot**(Crash) : Whether to try idiot crash

This is a type of 'crash' which works well on some homogeneous problems.

It works best on problems with unit elements and rhs but will do something

to any model. It should only be used before primal. It can be set

to -1 when the code decides for itself whether to use it, 0 to switch

off or n > 0 to do n passes.

<Range of values is -1 to 99999999; current -1>

**log**(Level) : Level of detail in Solver output

If 0 then there should be no output in normal circumstances. 1 is

probably the best value for most uses, while 2 and 3 give more information.

<Range of values is -1 to 999999; current 1>

**maxF**(actor) : Maximum number of iterations between refactorizations

If this is at its initial value of 200 then in this executable clp

will guess at a value to use. Otherwise the user can set a value.

The code may decide to re-factorize earlier for accuracy.

<Range of values is 1 to 999999; current 200>

**maxIt**(erations) : Maximum number of iterations before stopping

This can be used for testing purposes. The corresponding library

call

setMaximumIterations(value)

can be useful. If the code stops on seconds or by an interrupt this

will be treated as stopping on maximum iterations. This is ignored

in branchAndCut - use maxN!odes.

<Range of values is 0 to 2147483647; current 2147483647>

**moreS**(pecialOptions) : Yet more dubious options for Simplex - see ClpSimplex.hpp

<Range of values is 0 to 2147483647; current -1>

**output**(Format) : Which output format to use

Normally export will be done using normal representation for numbers

and two values per line. You may want to do just one per line (for

grep or suchlike) and you may wish to save with absolute accuracy

using a coded version of the IEEE value. A value of 2 is normal. otherwise

odd values gives one value per line, even two. Values 1,2 give normal

format, 3,4 gives greater precision, while 5,6 give IEEE values.

When used for exporting a basis 1 does not save values, 2 saves values,

3 with greater accuracy and 4 in IEEE.

<Range of values is 1 to 6; current 2>

**passP**(resolve) : How many passes in presolve

Normally Presolve does 10 passes but you may want to do less to make

it more lightweight or do more if improvements are still being made.

As Presolve will return if nothing is being taken out, you should

not normally need to use this fine tuning.

<Range of values is -200 to 100; current 10>

**pertV**(alue) : Method of perturbation

<Range of values is -5000 to 102; current 50>

**pO**(ptions) : Dubious print options

If this is > 0 then presolve will give more information and branch

and cut will give statistics

<Range of values is 0 to 2147483647; current 0>

**preO**(pt) : Presolve options

<Range of values is 0 to 2147483647; current -1>

**randomS**(eed) : Random seed for Clp

This sets a random seed for Clp - 0 says use time of day.

<Range of values is 0 to 2147483647; current 1234567>

**slp**(Value) : Number of slp passes before primal

If you are solving a quadratic problem using primal then it may be

helpful to do some sequential Lps to get a good approximate solution.

<Range of values is -50000 to 50000; current -1>

**small**(Factorization) : Whether to use small factorization

If processed problem <= this use small factorization

<Range of values is -1 to 10000; current -1>

**special**(Options) : Dubious options for Simplex - see ClpSimplex.hpp

<Range of values is 0 to 2147483647; current -1>

**sprint**(Crash) : Whether to try sprint crash

For long and thin problems this program may solve a series of small

problems created by taking a subset of the columns. I introduced

the idea as 'Sprint' after an LP code of that name of the 60's which

tried the same tactic (not totally successfully). Cplex calls it

'sifting'. -1 is automatic choice, 0 is off, n is number of passes

<Range of values is -1 to 5000000; current -1>

**subs**(titution) : How long a column to substitute for in presolve

Normally Presolve gets rid of 'free' variables when there are no more

than 3 variables in column. If you increase this the number of rows

may decrease but number of elements may increase.

<Range of values is 0 to 10000; current 3>

**verbose** : Switches on longer help on single ?

Set to 1 to get short help with ? list, 2 to get long help, 3 for

both. (add 4 to just get ampl ones).

<Range of values is 0 to 31; current 0>

**Keyword parameters**

------------------

**allC**(ommands) : Whether to print less used commands

For the sake of your sanity, only the more useful and simple commands

are printed out on ?.

<Possible options for allCommands are: no more all; current all>

**auto**(Scale) : Whether to scale objective, rhs and bounds of problem if they

look odd

If you think you may get odd objective values or large equality rows

etc then it may be worth setting this true. It is still experimental

and you may prefer to use objective!Scale and rhs!Scale.

<Possible options for autoScale are: off on; current off>

**biasLU** : Whether factorization biased towards U

<Possible options for biasLU are: UU UX LX LL; current LX>

**bscale** : Whether to scale in barrier (and ordering speed)

<Possible options for bscale are: off on off1 on1 off2 on2; current off1>

**chol**(esky) : Which cholesky algorithm

For a barrier code to be effective it needs a good Cholesky ordering

and factorization. The native ordering and factorization is not state

of the art, although acceptable. You may want to link in one from

another source. See Makefile.locations for some possibilities.

<Possible options for cholesky are: native dense fudge(Long\_dummy)

wssmp\_dummy

Uni(versityOfFlorida\_dummy)

Taucs\_dummy Mumps\_dummy;

current native>

**crash** : Whether to create basis for problem

If crash is set on and there is an all slack basis then Clp will flip

or put structural variables into basis with the aim of getting dual

feasible. On the whole dual seems to be better without it and there

are alternative types of 'crash' for primal e.g. 'idiot' or 'sprint'.

I have also added a variant due to Solow and Halim which is as on

but just flip.

<Possible options for crash are: off on so(low\_halim) lots idiot1

idiot2 idiot3 idiot4 idiot5

idiot6 idiot7;

current off>

**cross**(over) : Whether to get a basic solution after barrier

Interior point algorithms do not obtain a basic solution (and the

feasibility criterion is a bit suspect (JJF)). This option will crossover

to a basic solution suitable for ranging or branch and cut. With

the current state of quadratic it may be a good idea to switch off

crossover for quadratic (and maybe presolve as well) - the option

maybe does this.

<Possible options for crossover are: on off maybe presolve; current on>

**direction** : Minimize or Maximize

The default is minimize - use 'direction maximize' for maximization.

You can also use the parameters 'maximize' or 'minimize'.

<Possible options for direction are: min(imize) max(imize) zero;

current min(imize)>

**dualP**(ivot) : Dual pivot choice algorithm

Clp can use any pivot selection algorithm which the user codes as

long as it implements the features in the abstract pivot base class.

The Dantzig method is implemented to show a simple method but its

use is deprecated. Steepest is the method of choice and there are

two variants which keep all weights updated but only scan a subset

each iteration. Partial switches this on while automatic decides at

each iteration based on information about the factorization.

<Possible options for dualPivot are: auto(matic) dant(zig)

partial steep(est);

current auto(matic)>

**error**(sAllowed) : Whether to allow import errors

The default is not to use any model which had errors when reading

the mps file. Setting this to 'on' will allow all errors from which

the code can recover simply by ignoring the error. There are some

errors from which the code can not recover e.g. no ENDATA. This has

to be set before import i.e. -errorsAllowed on -import xxxxxx.mps.

<Possible options for errorsAllowed are: off on; current off>

**fact**(orization) : Which factorization to use

The default is to use the normal CoinFactorization, but other choices

are a dense one, osl's or one designed for small problems.

<Possible options for factorization are: normal dense simple osl;

current normal>

**gamma**((Delta)) : Whether to regularize barrier

<Possible options for gamma(Delta) are: off on gamma delta onstrong

gammastrong deltastrong;

current off>

**keepN**(ames) : Whether to keep names from import

It saves space to get rid of names so if you need to you can set this

to off. This needs to be set before the import of model - so -keepnames

off -import xxxxx.mps.

<Possible options for keepNames are: on off; current on>

**KKT** : Whether to use KKT factorization

<Possible options for KKT are: off on; current off>

**mess**(ages) : Controls if Clpnnnn is printed

The default behavior is to put out messages such as:

Clp0005 2261 Objective 109.024 Primal infeas 944413 (758)

but this program turns this off to make it look more friendly. It

can be useful to turn them back on if you want to be able to 'grep'

for particular messages or if you intend to override the behavior

of a particular message. This only affects Clp not Cbc.

<Possible options for messages are: off on; current off>

**perturb**(ation) : Whether to perturb problem

Perturbation helps to stop cycling, but Clp uses other measures for

this. However large problems and especially ones with unit elements

and unit rhs or costs benefit from perturbation. Normally Clp tries

to be intelligent, but you can switch this off. The Clp library has

this off by default. This program has it on by default.

<Possible options for perturbation are: on off; current on>

**PFI** : Whether to use Product Form of Inverse in simplex

By default clp uses Forrest-Tomlin L-U update. If you are masochistic

you can switch it off.

<Possible options for PFI are: off on; current off>

**presolve** : Whether to presolve problem

Presolve analyzes the model to find such things as redundant equations,

equations which fix some variables, equations which can be transformed

into bounds etc etc. For the initial solve of any problem this is

worth doing unless you know that it will have no effect. on will

normally do 5 passes while using 'more' will do 10. If the problem

is very large you may need to write the original to file using 'file'.

<Possible options for presolve are: on off more file; current on>

**primalP**(ivot) : Primal pivot choice algorithm

Clp can use any pivot selection algorithm which the user codes as

long as it implements the features in the abstract pivot base class.

The Dantzig method is implemented to show a simple method but its

use is deprecated. Exact devex is the method of choice and there

are two variants which keep all weights updated but only scan a subset

each iteration. Partial switches this on while change initially does

dantzig until the factorization becomes denser. This is still a work

in progress.

<Possible options for primalPivot are: auto(matic) exa(ct) dant(zig)

part(ial) steep(est) change sprint;

current auto(matic)>

**printi**(ngOptions) : Print options

This changes the amount and format of printing a solution:

normal - nonzero column variables

integer - nonzero integer column variables

special - in format suitable for OsiRowCutDebugger

rows - nonzero column variables and row activities

all - all column variables and row activities.

For non-integer problems 'integer' and 'special' act like 'normal'.

Also see printMask for controlling output.

<Possible options for printingOptions are: normal integer special rows all

csv bound(ranging) rhs(ranging)

objective(ranging) stats

boundsint boundsall;

current normal>

**scal**(ing) : Whether to scale problem

Scaling can help in solving problems which might otherwise fail because

of lack of accuracy. It can also reduce the number of iterations.

It is not applied if the range of elements is small. When unscaled

it is possible that there may be small primal and/or infeasibilities.

<Possible options for scaling are: off equi(librium) geo(metric) auto(matic)

dynamic rows(only);

current auto(matic)>

**spars**(eFactor) : Whether factorization treated as sparse

<Possible options for sparseFactor are: on off; current on>

**timeM**(ode) : Whether to use CPU or elapsed time

cpu uses CPU time for stopping, while elapsed uses elapsed time. (On

Windows, elapsed time is always used).

<Possible options for timeMode are: cpu elapsed; current cpu>

**vector** : Whether to use vector? Form of matrix in simplex

If this is on ClpPackedMatrix uses extra column copy in odd format.

<Possible options for vector are: off on; current off>

**Actions or string parameters**

----------------------------

**allS**(lack) : Set basis back to all slack and reset solution

Mainly useful for tuning purposes. Normally the first dual or primal

will be using an all slack basis anyway.

**barr**(ier) : Solve using primal dual predictor corrector algorithm

This command solves the current model using the primal dual predictor

corrector algorithm. You may want to link in an alternative ordering

and factorization. It will also solve models with quadratic objectives.

**basisI**(n) : Import basis from bas file

This will read an MPS format basis file from the given file name.

It will use the default directory given by 'directory'. A name of

'$' will use the previous value for the name. This is initialized

to '', i.e. it must be set. If you have libz then it can read compressed

files 'xxxxxxxx.gz' or xxxxxxxx.bz2.

**basisO**(ut) : Export basis as bas file

This will write an MPS format basis file to the given file name.

It will use the default directory given by 'directory'. A name of

'$' will use the previous value for the name. This is initialized

to 'default.bas'.

**directory** : Set Default directory for import etc.

This sets the directory which import, export, saveModel, restoreModel

etc will use. It is initialized to './'

**dirSample** : Set directory where the COIN-OR sample problems are.

This sets the directory where the COIN-OR sample problems reside.

It is used only when -unitTest is passed to clp. clp will pick up

the test problems from this directory. It is initialized to

'../../Data/Sample'

**dirNetlib** : Set directory where the netlib problems are.

This sets the directory where the netlib problems reside. One can

get the netlib problems from COIN-OR or from the main netlib site.

This parameter is used only when -netlib is passed to clp. clp will

pick up the netlib problems from this directory. If clp is built without

zlib support then the problems must be uncompressed. It is initialized

to '../../Data/Netlib'

**dirMiplib** : Set directory where the miplib 2003 problems are.

This sets the directory where the miplib 2003 problems reside. One

can get the miplib problems from COIN-OR or from the main miplib site.

This parameter is used only when -miplib is passed to cbc. cbc will

pick up the miplib problems from this directory. If cbc is built without

zlib support then the problems must be uncompressed. It is initialized

to '../../Data/miplib3'

**dualS**(implex) : Do dual simplex algorithm

This command solves the continuous relaxation of the current model

using the dual steepest edge algorithm. The time and iterations may

be affected by settings such as presolve, scaling, crash and also

by dual pivot method, fake bound on variables and dual and primal

tolerances.

**either**(Simplex) : Do dual or primal simplex algorithm

This command solves the continuous relaxation of the current model

using the dual or primal algorithm, based on a dubious analysis of

model.

**end** : Stops clp execution

This stops execution ; end, exit, quit and stop are synonyms

**environ**(ment) : Read commands from environment

This starts reading from environment variable CBC\_CLP\_ENVIRONMENT.

**exit** : Stops clp execution

This stops the execution of Clp, end, exit, quit and stop are synonyms

**export** : Export model as mps file

This will write an MPS format file to the given file name. It will

use the default directory given by 'directory'. A name of '$' will

use the previous value for the name. This is initialized to 'default.mps'.

It can be useful to get rid of the original names and go over to using

Rnnnnnnn and Cnnnnnnn. This can be done by setting 'keepnames' off

before importing mps file.

**fakeB**(ound) : All bounds <= this value - DEBUG

**gsolu**(tion) : Puts glpk solution to file

Will write a glpk solution file to the given file name. It will use

the default directory given by 'directory'. A name of '$' will use

the previous value for the name. This is initialized to 'stdout'

(this defaults to ordinary solution if stdout). If problem created

from gmpl model - will do any reports.

**help** : Print out version, non-standard options and some help

This prints out some help to get user started. If you have printed

this then you should be past that stage:-)

**import** : Import model from mps file

This will read an MPS format file from the given file name. It will

use the default directory given by 'directory'. A name of '$' will

use the previous value for the name. This is initialized to '', i.e.

it must be set. If you have libgz then it can read compressed files

'xxxxxxxx.gz' or 'xxxxxxxx.bz2'. If 'keepnames' is off, then names

are dropped -> Rnnnnnnn and Cnnnnnnn.

**max**(imize) : Set optimization direction to maximize

The default is minimize - use 'maximize' for maximization.

You can also use the parameters 'direction maximize'.

**min**(imize) : Set optimization direction to minimize

The default is minimize - use 'maximize' for maximization.

This should only be necessary if you have previously set maximization

You can also use the parameters 'direction minimize'.

**miplib** : Do some of miplib test set

**netlib** : Solve entire netlib test set

This exercises the unit test for clp and then solves the netlib test

set using dual or primal.The user can set options before e.g. clp

-presolve off -netlib

**netlibB**(arrier) : Solve entire netlib test set with barrier

This exercises the unit test for clp and then solves the netlib test

set using barrier.The user can set options before e.g. clp -kkt on

-netlib

**netlibD**(ual) : Solve entire netlib test set (dual)

This exercises the unit test for clp and then solves the netlib test

set using dual.The user can set options before e.g. clp -presolve

off -netlib

**netlibP**(rimal) : Solve entire netlib test set (primal)

This exercises the unit test for clp and then solves the netlib test

set using primal.The user can set options before e.g. clp -presolve

off -netlibp

**netlibT**(une) : Solve entire netlib test set with 'best' algorithm

This exercises the unit test for clp and then solves the netlib test

set using whatever works best. I know this is cheating but it also

stresses the code better by doing a mixture of stuff. The best algorithm

was chosen on a Linux ThinkPad using native cholesky with University

of Florida ordering.

**network** : Tries to make network matrix

Clp will go faster if the matrix can be converted to a network. The

matrix operations may be a bit faster with more efficient storage,

but the main advantage comes from using a network factorization.

It will probably not be as fast as a specialized network code.

**para**(metrics) : Import data from file and do parametrics

This will read a file with parametric data from the given file name

and then do parametrics. It will use the default directory given

by 'directory'. A name of '$' will use the previous value for the

name. This is initialized to '', i.e. it must be set. This can not

read from compressed files. File is in modified csv format - a line

ROWS will be followed by rows data while a line COLUMNS will be followed

by column data. The last line should be ENDATA. The ROWS line must

exist and is in the format ROWS, inital theta, final theta, interval

theta, n where n is 0 to get CLPI0062 message at interval or at each

change of theta and 1 to get CLPI0063 message at each iteration.

If interval theta is 0.0 or >= final theta then no interval reporting.

n may be missed out when it is taken as 0. If there is Row data then

there is a headings line with allowed headings - name, number, lower(rhs

change), upper(rhs change), rhs(change). Either the lower and upper

fields should be given or the rhs field. The optional COLUMNS line

is followed by a headings line with allowed headings - name, number,

objective(change), lower(change), upper(change). Exactly one of name

and number must be given for either section and missing ones have

value 0.0.

**plus**(Minus) : Tries to make +- 1 matrix

Clp will go slightly faster if the matrix can be converted so that

the elements are not stored and are known to be unit. The main advantage

is memory use. Clp may automatically see if it can convert the problem

so you should not need to use this.

**primalS**(implex) : Do primal simplex algorithm

This command solves the continuous relaxation of the current model

using the primal algorithm. The default is to use exact devex. The

time and iterations may be affected by settings such as presolve,

scaling, crash and also by column selection method, infeasibility

weight and dual and primal tolerances.

**printM**(ask) : Control printing of solution on a mask

If set then only those names which match mask are printed in a solution.

'?' matches any character and '\*' matches any set of characters.

The default is '' i.e. unset so all variables are printed. This is

only active if model has names.

**quit** : Stops clp execution

This stops the execution of Clp, end, exit, quit and stop are synonyms

**restoreS**(olution) : reads solution from file

This will read a binary solution file from the given file name. It

will use the default directory given by 'directory'. A name of '$'

will use the previous value for the name. This is initialized to

'solution.file'. This reads in a file from saveSolution

**reallyS**(cale) : Scales model in place

**restore**(Model) : Restore model from binary file

This reads data save by saveModel from the given file. It will use

the default directory given by 'directory'. A name of '$' will use

the previous value for the name. This is initialized to 'default.prob'.

**reverse** : Reverses sign of objective

Useful for testing if maximization works correctly

**saveM**(odel) : Save model to binary file

This will save the problem to the given file name for future use by

restoreModel. It will use the default directory given by 'directory'.

A name of '$' will use the previous value for the name. This is initialized

to 'default.prob'.

**saveS**(olution) : saves solution to file

This will write a binary solution file to the given file name. It

will use the default directory given by 'directory'. A name of '$'

will use the previous value for the name. This is initialized to

'solution.file'. To read the file use fread(int) twice to pick up

number of rows and columns, then fread(double) to pick up objective

value, then pick up row activities, row duals, column activities and

reduced costs - see bottom of CbcOrClpParam.cpp for code that reads

or writes file. If name contains '\_fix\_read\_' then does not write

but reads and will fix all variables

**sleep** : for debug

If passed to solver fom ampl, then ampl will wait so that you can

copy .nl file for debug.

**solu**(tion) : Prints solution to file

This will write a primitive solution file to the given file name.

It will use the default directory given by 'directory'. A name of

'$' will use the previous value for the name. This is initialized

to 'stdout'. The amount of output can be varied using printi!ngOptions

or printMask.

**solv**(e) : Solve problem using dual simplex (probably)

Just so can use solve for clp as well as in cbc

**stat**(istics) : Print some statistics

This command prints some statistics for the current model. If log

level >1 then more is printed. These are for presolved model if presolve

on (and unscaled).

**stop** : Stops clp execution

This stops the execution of Clp, end, exit, quit and stop are synonyms

**tightLP** : Poor person's preSolve for now

**unitTest** : Do unit test

This exercises the unit test for clp

**userClp** : Hand coded Clp stuff

There are times e.g. when using AMPL interface when you may wish to

do something unusual. Look for USERCLP in main driver and modify

sample code.