Stanford Circuit Optimization Tool (SCOT) User Guide

Dinesh Patil, Seung-Jean Kim
Stanford University
(ddpatil,sjkim)@stanford.edu

1. Introduction

This document describes the Stanford digital circuits optimization framework. The general purpose of the software is to enable automatic global device sizing, supply and threshold voltage optimization for optimal delay,area or energy. This is achieved by representing the digital circuit netlist as constrained optimization problem (specifically a geometric program - described in Appendix 1). The problem is formulated using analytical delay, area and energy models and solved using a solver called MOSEK (http://www.mosek.com), which is a suite of routines for solving convex optimization problems. A bunch of programs are used as wrappers around this package to enable the assimilation of the device model data into equations, conversion of spice netlist schematic into a mathematical form and back-annotating the results of the optimization in the spice netlist. The models and equations are in the form of generalized posynomials(Appendix 1). The software consists of five main components. Each component includes a collection of perl scripts, C++ code and interface to other stand-alone tools like schematic editors (SUE), IRSIM, MOSEK, SPICE and so on. A brief description of the sections in the order the user is most likely to use them, is as follows:

1. **Schematic Entry**: This includes the schematic editor SUE, used to enter the schematics and design constraints (as annotated comments). The spice file generated from SUE is then modified with perl scripts to interpret all the commands provided by the user in the SUE file. Using SUE, one can also generate SIM file used for activity factor measurements and functional verification of the design in IRSIM prior to optimization. SCOT uses the modified spice file as its main input. Though SUE can generate this file most conveniently, other schematic editors can be used and modified to obtain the required spice format.

2. **Generating Models and Switching Statistics**: This consists of generating analytic delay and energy models for the various channel connected components (CCCs) in the netlist. It uses the data from a file that contains the basic transistor current models for a chain of transistors. This tool section also generates the activity and duty factor of the nets in the netlist by performing switch level simulations using IRSIM.

3. **Problem formulation**: This section contains software to read in the CCC delay and energy model files, the spice netlist and a user specified command file to generate the circuit sizing/Vdd-Vth allocation problem as a geometric program with the constraints specified by the user during schematic entry. The the user can also specify other post analysis operations like drawing PDFs, obtaining vector path delays etc in the command file.

4. **GP solver**: This program takes in the formulated Geometric program (GP) and solves it using MOSEK. This creates a solution file that contains the optimal values of the objective and design variables.

5. **Back-annotation**: This section consists of various scripts to back annotate the MOSEK results in spice file or SUE file, for verification and validation, or viewing the results.

The tool flow and the various components are shown in figure 1 and described in the following sections.

For ease of understanding, we shall consider an example circuit say adder32, which we will take through the tool flow. All the interfaces between the various sections of the design flow is by easily readable text files. Users can modify them in the middle of the flow, if they want to make their own scripts for iterative data generation. Though this is not recommended, it is useful at times to get around some of the constraints that the tool imposes and also to get quick results from a re-run instead

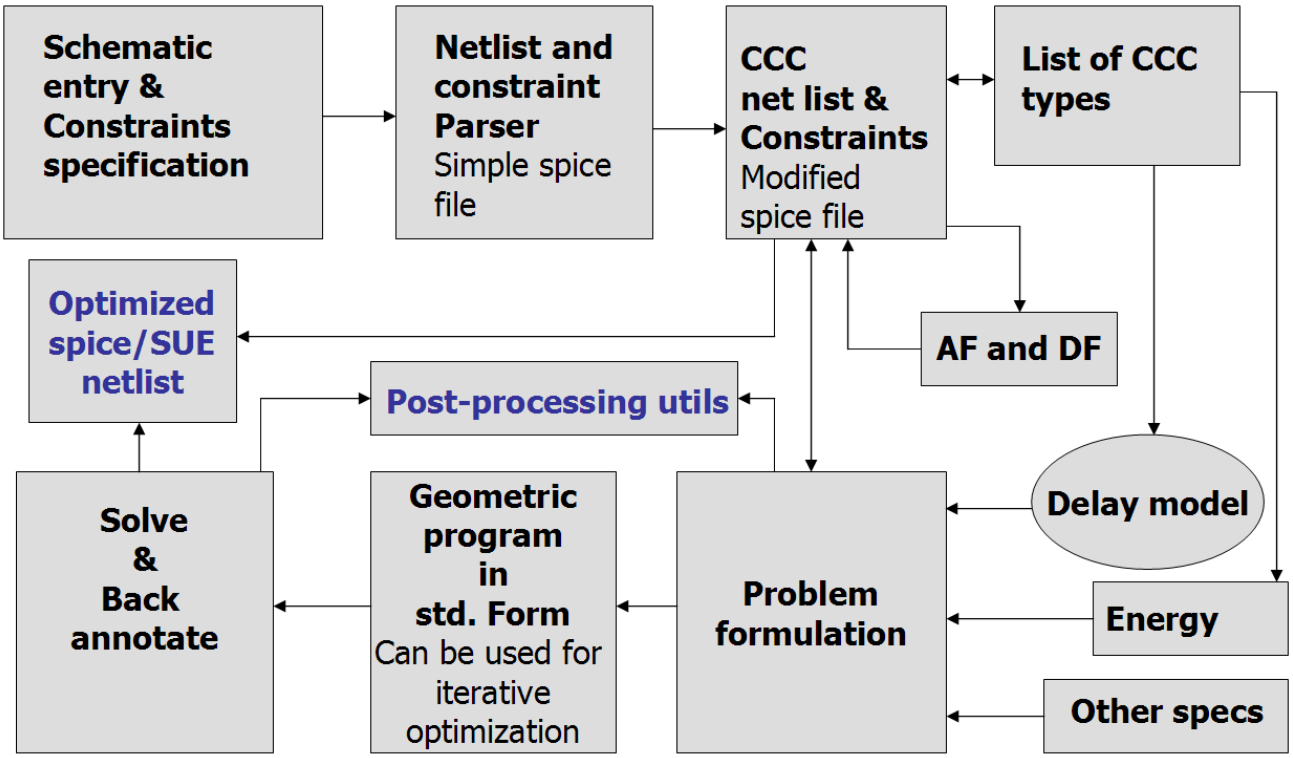


Figure 1. The overall tool flow

of having to go through the entire flow again. Hence it is beneficial to know what information is contained in the interface files. The entire framework is encapsulated in a perl script called **optim.pl** which is run with various options at different points in the tool flow. To obtain a quick summary of the various options, just run optim.pl at the prompt. A snapshot of the result is shown below. Running optim.pl with just the option and no arguments will display the information about its usage.

```
elaine11 ~> optim.pl
Usage: optim.pl [option] respective_fileNames
Run the various optimization commands on the circuit netlist.
Options and the corresponding files needed (in order) are:

  -sp  : convert a typical spice file into the
         modified_spice_file needed by the
         optimizer.

  -mod : obtain the delay and leakage models for
         the CCCs in the netlist. The netlist_file
         is also modified in the process to record
         any transmission gates in the
         netlist.

  -psn : Name all the blocks in the SUE hierarchy.
         Also name all the nets which have the name-net
         hook attached to them. Nets that are buses
         cannot be named. The names help to track mistakes
         and block names are a must for back-annotation in SUE.

  -sol : optimize the netlist using the generated
         models and the optimization and analysis
         tasks specified by user.
         The files produced as a result are
         according to the names given in opt_file.

  -irsim : run IRSIM (provided IRSIM is installed)
         on the netlist to find out the
         activity factors, and include them in the
         optimization file.
         If the user is providing the command file
         for IRSIM commands, then the number of runs
         should be entered correctly by the user.

  -paf   : put the activity factors in the mod-file from
         the already available activity factors file. Used
         when the modified spice file gets over-written or
         otherwise modified and the power section is deleted.

  -pdf   : like paf, put duty factors into the modified file
         from a file containing a list of duty factors.

  -bsu : back annotates the results in the sue file
         and produces a new sue netlist with optimized
         sizes.

  -bsp : back annotate the results in spice file. The
         file used by the optimizer is used.
```

```
  -beldo : same as -bsp except that the spice_file
           is then converted in to an eldo compatible
           spice file.

  -ggp : solve the GGP problem using ggpsol and MOSEK.

  -psvo : Take Vth values from the solution file, snap
          them to discrete values within a specified
          margin and substitute them back into the
          ggpsol input file to make a new problem for
          iterative optimization. Also record the snapped
          Vth values in a separate file.

  -gpIter : Change one constraint and generate multiple solutions
            to obtain a tradeoff curve.

  -extract : Given a file containing a list of solution file names and
             variable names, go thru these solution files and gather the
             optimal values of the specified variables. Dump the values into
             the specified file in a MATLAB loadable format.
elaine11 ~>
```

2. Schematic Entry

2.1 General Rules

As mentioned before, the input file specification for the software is similar to a spice format (described later). We can generate such a file most conveniently with SUE (though other schematic editors with similar capabilities will work as well). Here we describe the design entry using SUE.

The circuits are entered in SUE and can be hierarchical. Each unit of the hierarchy is called a module. The most basic module is a channel connected component (CCC) module(and not a transistor). A CCC is a collection of transistors which are connected through their source-drain channel. Most common logic gates fall in this category. For correctness of the problem specification users are constrained to ensure the following

1. The smallest module in the netlist hierarchy consists entirely of transistors, forms a complete CCC and no net inside the CCC (except the inputs) connects to the gate of any transistor.
2. The CCCs can have multiple outputs. Users can also have multiple parallel CCCs inside the basic unit. For example having two inverters in one CCC is fine. It will make a CCC with two inputs and two outputs.
3. No module can contain other modules and transistors simultaneously. Transistors can only be specified inside a CCC.
4. The input names of all the CCCs should start with letters [i-z] and the output names with letters [a-h]. The higher level modules are allowed to have any suitable input output names.
5. It is recommended that all the nets and modules (including the CCCs) are given instance names. While this is not necessary for optimization, it is a must for back annotation. Perl scripts are provided to help name the instances and nets automatically. For nets the user has to provide the hook (name-net from the menu) on the net to be named. Busses have to be named by the user.
6. All CCCs must be parameterized to have a handle on their sizing variables at the upper level. These will be used by the optimizer to size the CCCs. These parameters are typically for size, but you can have other parameters as well like Vth.

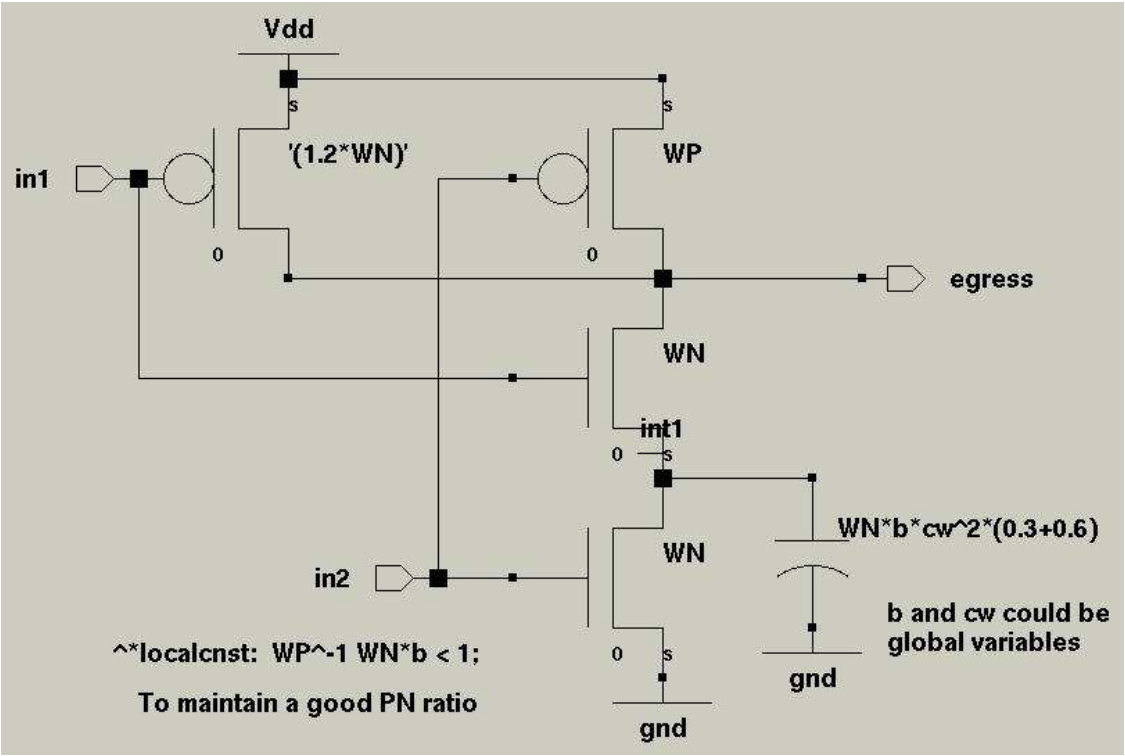


Figure 2. A typical NAND CCC specification

2.2 Features

Following are the design rules for schematic entry for efficient and unambiguous problem specification.

1. The various specifications and optimization constraints are given at various levels in the hierarchy as annotated text. These are prefixed with * in the beginning so that they appear in the resulting spice file as comments. At the CCC level, one can specify the relation between individual transistor widths, which would be useful for limiting the skewing of the inverter, specifying the size of a keeper relative to the pull down chain etc. An example of a CCC specification is shown in figure 2. Using the keyword **localcnst**, we can specify a local constraint involving local and global variables. All the local variables will be prefixed with their hierarchy to make the constraint global. All constraints are specified as a $posynomial < monomial$. Hence in some cases things have to be rearranged.

2. The CCC can be parameterized for each of the transistor width or one can have fewer variables by having the internal transistors sized in a fixed ratio with each other. This depends on the model one is using. In the latter case the width of each transistor can be specified in terms of the parameters by using a monomial inside parenthesis and inverted quotes as shown in figure 2 for the pmos on the left. It is important that this expression for width be in parenthesis since it will be used as a unit (like in the denominator of a delay model).

3. Structures like keepers are included in the same CCC as the dynamic gate (since they are channel connected to it). Similarly one of the back to back inverter in a latch would be in the same CCC as the CCC driving the back to back inverter. In such a case we do not want to have a delay path through such structures since not only is it not a delay path, it will cause a cycle in the problem formulation (rendering it insolvable). This can be avoided by naming the inputs of that CCC which go to such transistors with the prefix “inK”. The optimizer takes care to see that such inputs are not involved in the timing graph, though their loading is considered on the gates driving them. The optimizer however does not take into account any current fight that can occur from using such a structure. It should be included in the model by the user.

4. Some CCCs can be quite big (like barrel shifters for example) and can have long wires inside them. For such cases one can specify a capacitor inside a CCC just like usual. The value of the capacitor can be a posynomial. The software will normally prefix any variable in this posynomial by the CCC's instance name in the global netlist (in order to make the variable global), but if the variable is already global (like bit pitch for example), then it has to be specified at the top level as a global variable.

5. Capacitors can also be included anywhere in the netlist outside the CCC. But outside the CCC, no prefix is added to the variables in the expression for the value of the cap. Hence for such capacitors one has to provide the value in terms of global variables (i.e if the wire caps are expressions, then the capacitance should be specified with its global name of the width).

6. The global specifications about input arrival times, slopes and max input loading capacitances, the output load conditions and other global constraints are given at the top level module, again as annotated text. These are interpreted by the perl scripts that modify the spice file produced by SUE and make it suitable as an input to the optimizer. Figure 3 shows a cartoon example containing all the possible commands.

   The various commands are interpreted as follows:

   (a) Input specifications: This command specifies the name of the inputs, their arrival times, the input rise and fall slopes, and the maximum allowed input capacitance. The maximum capacitance can be a monomial. The input name can be specified as a bus.

   (b) Output specifications: The outputs are specified as names and load. The output load can be a posynomial.

   (c) $A < A_{max}$: The area is assumed to be the sum of the transistor widths. This command results into a constraint which specifies that the sum of the widths is less than A_{max}. We can similarly have $E < E_{max}$ or $D < D_{max}$ where E is a sum of dynamic and leakage energy with E_{max} being the maximum you can have, while D is the overall delay of the circuit given as the maximum of the arrival time of all the outputs.

   (d) MAX_WIDTH and others: These specify the maximum and minimum bounds on width, V_{th} for nmos(pull down) and pmos(pull up) structures inside each CCC and V_{dd}. For specifying the maximum and minimum V_{dd} it is necessary to use the MAX_VDD and MIN_VDD since these also tell the optimizer to use V_{dd} as a variable and not use its default value for energy calculations.

   (e) D(all) : This specifies that the delay per stage is less than a certain number. One can also specify a monomial here. This is done to control the slope of every signal. Since the slope is difficult to model, we constraint the delay of the gate, that is basically correlated with the slope of the output. Instead of "all", one can also more specific delay per stage constraints for a particular cell for a specific rise fall transition. This specific constraint overrides the one provided by D(all).

   (f) DoNotUniquify: When multiple instances of the same module are instantiated, by default they are considered unique and get sized differently. For sizing them the same (for example to size the bit PG generators for all bits the same), the corresponding module name (not the instance name) is specified to not be uniquified using this command.

   (g) GlobalVar: By default, the optimizer appends the name of the CCC to any of the variables found inside the CCC delay or energy model. This is the way to uniquely identify the variable at the top level. But some variables like V_{dd}, bit pitch b, etc are common to all CCCs, i.e. they are global. Such variables are specified using the above command. This prevents their prefixing.

   (h) OnlyFormulateProblem: The problem formulation step prints the optimization problem in a file and automatically calls the GP solver. Many times we want to tweak the problem input file independently for changing a particular constraint to generate a tradeoff curve. This command stops the program after the problem formulation step.

   (i) NoAllEdgeTimeConstraint: In general all the net timings are automatically less than the overall delay variable(represented by the variable POMAX in this framework). But in case of dynamic logic, the dynamic inputs have no bound (mathematically) on their fall edge, since it is not a part of the delay formulation. This can blow up the fall time variable of these nets beyond proportions. Hence by default the optimizer puts in a constraint that limits says that the rise and fall timing of all the nets in the circuit is less than POMAX. Now for self resetting circuits, the reset signals continue to transition even after POMAX since POMAX is the delay of the signal transition of our interest to the output. In such case we have to separately specify to the optimizer not to use the AllEdgeTimings less than POMAX. Syntax shown in figure 4

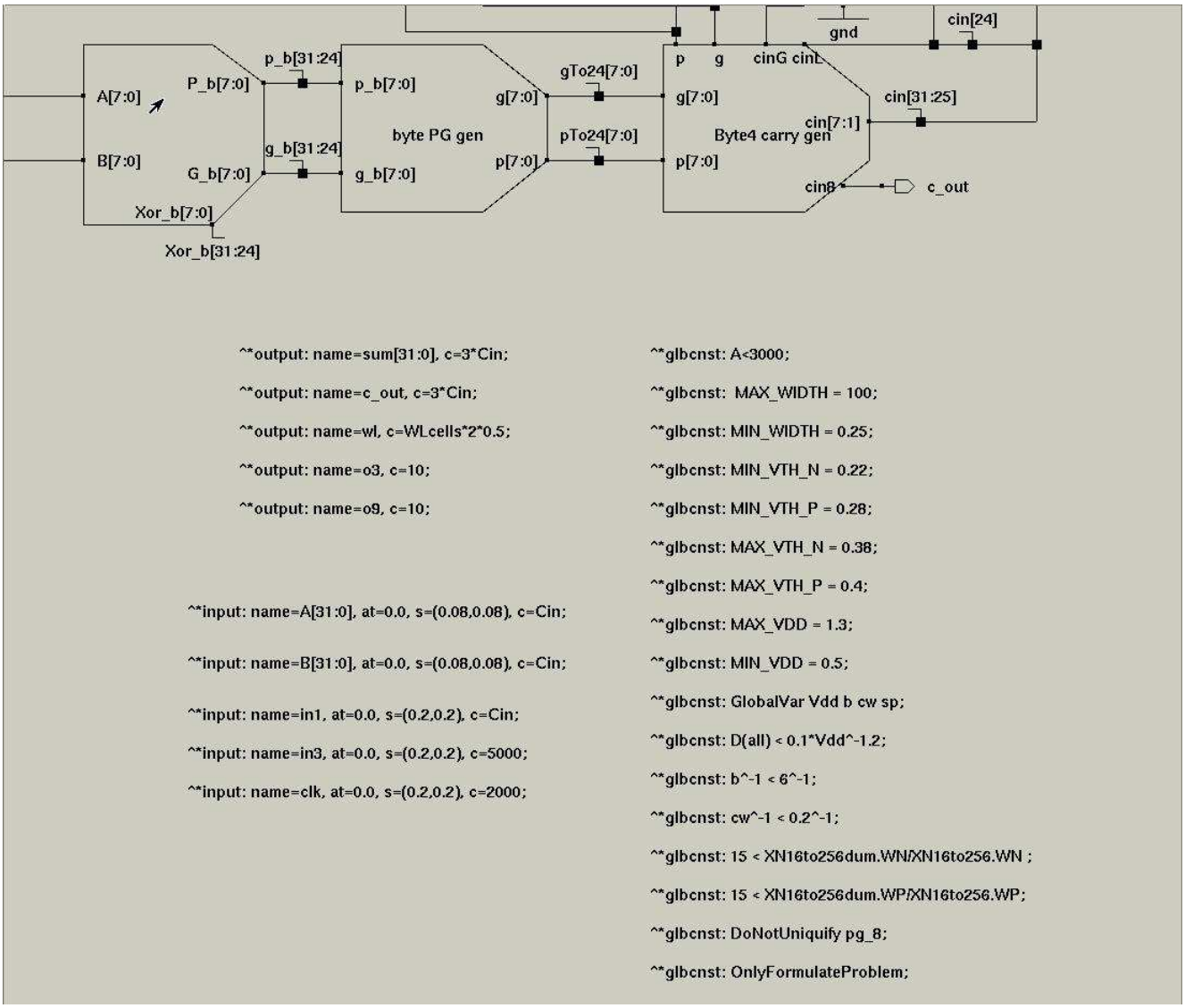


Figure 3. A typical top level specification in SUE

```
.GLBCNSTR
....
NoAllEdgeTimingsConstraint;
....
.ENDS
```

Figure 4. No All Edge Timing Constraint

```
.GLBCNSTR
....
setUniformKappas;
....
.ENDS
```

Figure 5. Set Uniform Kappas Constraint

```
.GLBCNSTR
....
DoNotIncludeLeakage;
....
.ENDS
```

Figure 6. Spec to not include leakage

(j) setUniformKappas : This is used in the context of statistical optimization. As the name suggests it tells the optimizer to use the same kappa (and not scale it according to the path length) for all the nodes. Syntax shown in figure 5

(k) DoNotIncludeLeakage : As the name suggests it is used to tell the optimizer not to include leakage in the energy constraint (if you have one). Since leakage is not very well modelled and may not be significant at all at higher feature sizes. In that case this option can speed up the solver by removing a large expression from the energy constraint. One can always use leakage as a fixed % penalty in energy. Syntax shown in figure 6

(l) LogicDepthFactor : The total energy is equal to the sum of leakage and dynamic energy. While the dynamic energy can be calculate per transition, for leakage energy one has to specify the amount of time between the switching activity. Most logic blocks that are optimized, are supposed to consume one clock cycle and hence the overall delay is the right amount of time to be specified for leakage. But, if we are optimizing only a subset of the logic, say only a single inverter, then the leakage energy of the inverter is roughly logic-depth times the leakage energy spent during its switching delay. Here this command is used. It specifies the ratio of the logic-depth w.r.t. the depth of the circuit you are simulating. By default the value is 1 and is greater than 1 for circuits that are deemed by the user to be smaller than a clock cycle. Syntax is shown in figure 7

(m) UseDefActFact : This command allows the optimizer to use the default activity factor as specified by the user in the global parameter section. Sometimes the user may roughly know the switching factor or may only simulate the activity factor of some nets. This command can be used to attribute the desired activity factors for all other nets. The user should be careful using this command as it can camouflage an error in the switching simulation that did not generate the activity factor or some net. Syntax is shown in figure 8

(n) AntiCorr : This command is used to indicate that a certain pair of inputs to a CCC are anti-correlated, so that the delay modeling routine excludes the transistor stacks containing both the input and its complement. This is useful in dual rail domino case or in other places where a certain relation of the inputs is known. This helps in accurate modeling. Syntax is as follows. “AntiCorr” is a key word. Also at a given time only two mutually complementary

```
.GLBCNSTR
....
LogicDepthFactor = #;
....
.ENDS
```

Figure 7. Specify the Logic Depth Factor

```
.GLBCNSTR
....
UseDefActFact;
....
.ENDS
```

Figure 8. Allow the use of default activity factor for nets

```
.GLBCNSTR
....
AntiCorr CCC_type_name in1 in2;
....
.ENDS
```

Figure 9. Specify the AntiCorr inputs

inputs can be specified. 9

(o) IntPrecharge : Similar to the the AntiCorr command, this command tells the modelling routine to exclude the delay paths that include the precharge transistor. This has to be provided even of the internal precharge input is named as a dummy ("inK*"). Again this is useful mostly in domino logic to avoid the modeling of a fall to rise delay from any of the NMOS only inputs to the output through the intermediate precharge, when the input signals go low. Syntax is as follow. IntPrecharge is a key word. 10

(p) OtherConstraints: In general any global constraint can be written just as is at the top level as far as there is no ambiguity between the constraint and the above special constraint. A couple of such examples are shown in figure 3. Note that all the constraints have to have "$<$" relation so "$>$" has to be inverted. Also "$=$" can be represented by two inequalities, though it is recommended not to use it since it makes the optimizer unstable.

The spice file generated by SUE is then given to a netlist flattener script that removes all the extra hierarchy, preserving only the two level hierarchy of CCCs and main netlist. The scripts include the technology parameters from the technology file into the spice file. The annotated SUE comments are interpreted and expanded into various constraints and specifications. The resulting file is ready to be used for problem formulation. The command line
$\$\$optim.pl\ -sp\ adder32.sp\ glb90param.sp\ adder32\_mod.sp$
does all the above tasks. The resulting file looks likes the one shown in figure 11.

3 Generating Models and Switching Statistics

The delay models for the CCCs are generated from the delay models for chain of transistors. These delay models are based on velocity saturated current models. The delays models the worst case delay from any input to any (excitable) output of the CCC. The modeling software (originally written by Alvin Cheung) takes in the modified spice file adder32_mod.sp, goes through the CCC subcircuit definitions, and makes the delay (mean and standard deviation) models for rising and falling

```
.GLBCNSTR
....
IntPrecharge CCC_type_name in1 in2 in3 ... ;
....
.ENDS
```

Figure 10. Specify the Intermediate Precharge inputs

File Edit Tools Syntax Buffers Window Help

```
*THE CAPS IS IN fF UNLESS MENTIONED OTHERWISE;
*.....
*CURRENT IS IN uA UNLESS MENTIONED OTHERWISE;

.GLBPARAM
CovN = 0.465;
....
....
def_act_factor = 0.25;
.ENDS

.SUBCKT t_2nand egress : in1 in2 WP=0.5 WN=0.3 ...other useful/useless paramemters ;
M_0 egress in1 int1 gnd nmos W=WN L=0.1 GEO=0;
....
M_3 egress in2 Vdd Vdd pmos W=WP L=0.1 GEO=0;
.ENDS

.OTHER CCCs in SUBCIRCUITS
...
.ENDS

**MAINLIST BEGINS HERE;
.CONNECT
Xbuf1__Xt_inv_1 Xbuf1__in_b : cin_9 : t_inv WP=0.5 WN=0.5
Xbuf1__Xt_inv_2 cin[8] : Xbuf1__in_b : t_inv WP=0.5 WN=0.5
...
CXpg_block1__Xpg_8_1_1__C_2 : gnd Xor_b[2] c=15;
....
CXpg_block4__C_1 : gnd Xpg_block4__net_16 c=b*cw*4;
...
Xaoi_nand_1__Xt_aoi_15 g4[7] : p4_b[7] mynet_1 g4_b[7] : t_aoi WP1=0.5 WP2=0.5 WN1=0.5 WN2=0.5;
.ENDS

.PI
name=A[0], at=0.0, s=(0.07,0.07), c=Cin;
....
name=in1, at=0.0, s=(0.07,0.07), c=10000;
...
.ENDS

.PO
name=sum[0], c=g*Cin;
....
name=c_out, c=2*Cin+25;
.ENDS

.GLBCNSTR
Xbuf1__Xt_inv_1.WP^-1 Xbuf1__Xt_inv_1.WN < 1;
...
A<4000;
D(all) < 0.06;
MAX_WIDTH = 10000;
b^-1 < 6^-1;
MIN_VDD = 0.7;
.....
GlobalVar Vdd b cw g;
.ENDS
```

Figure 11. The modified spice file adder32_mod.sp.

transitions of the CCCs. The command
$$optim.pl\ -mod\ adder32\_mod.sp\ bsim90Models.dat\ adder32\_mod.dio$$
is used to generate the delay models. The command also produces as a by product, the parasitic capacitance at the output of a gate (used in the dynamic energy calculations of the circuit) and a crude leakage model for the gate, used for leakage energy calculations. Also the modeling routine finds out all the transmission gates in the CCCs and creates a new section for them in the adder32_mod.sp file. This section is used in their delay specification during problem formulation.

Using the SIM file for the netlist generated from SUE, the activity factors and duty factors can be obtained using the command
$$optim.pl\ -irsim\ adder32\_opt.sp\ adder32\_mod.sp\ scmos90.prm\ 10000\ <testvectors.cmd>$$
The SIM file is also used to functionally verify the design before optimization in the first place.

Since the circuit is not sized yet, this might be a inaccurate. So alternatively one can do a quick optimization w.r.t area or just specifying input and load capacitances (which do not need the activity or duty factors) and obtain a reasonably sized circuit. The SIM file can then be produced from the back-annotated SUE or SPICE file. IRSIM needs a command file containing input stimuli and a specification of the nets to be to be recorded. By default the command produces a command file with random inputs. If such is not the case (like in clocked elements like domino logic or latches) a user generated command file (like the one used in functional verification) can be used. In that case its up to the user to ensure that the number of inputs specified are same as the number of stimuli actually applied. There are default parameters (which can be changed by changing the global parameter section in the adder32_mod.sp file) for minimum activity factor (in case the activity factor of some net is zero), default activity factor (in case the activity factor of some net is not specified) and default duty factor (if the duty factor of some nets is not specified). The script attaches the section of activity factors and duty factors to adder32_mod.sp.

While the model generator accepts inputs going to many places (like the select lines in a barrel shifter for example), it does not handle CCCs in which the inputs are correlated in some way so that one input activates two or more pull down or pull up paths simultaneously (as in a 6T Xor gate for example). The modeling routine considers the output delay as simply the maximum of the delay of the two paths. If this is a huge error, users can write a model themselves in adder32.dio file generated by the modeling routine.

Since the file adder32.dio is in an easy text format, users can change it and add models independently. The model generation routine is just an add on to enable fast modeling of common gates. For funky gates and some other logic families, one has to directly modify the adder32.dio file.

Currently the model generation works well with static CMOS (with transmission gates) and dynamic logic.

4 Problem Formulation

This program (originally written by Sunghee Yun) takes in the adder32_mod.sp and adder32_mod.dio and a file specifying the kind of optimization and post optimization utilities specified by the user and formulates a Geometric Program (GP) file. This specifications file (called adder32.opt below) is created manually by the user and determines the names of the result files generated by the optimizer. Its general structure is shown in the figure 12.

The problem formulator interprets the constraints in adder32_mod.sp file and the user specified tasks from adder32.opt and forms the delay and energy equations accordingly. The file is then solved by internally (unless the command OnlyFormulateProblem is specified in the adder32_mod.sp file) using the command described in the next section. This is achieved using the command:
$$optim.pl\ -sol\ adder32\_mod.sp\ adder32\_mod.dio\ adder32.opt$$

The intermediate GP file generated in the process (lets call it adder32OPT) contains the mathematical problem in a text format. Figure 13 shows an example. Users can easily read the file and modify it at this level too, though it is more confusing to do that. The file contains labeled constraints. The solution file "adder32OPT.out", produced as a result of the optimization contains the optimal values of the variables and also the sensitivity (labeled as DUAL variables) of the constraints to the overall objective. From the labels of the constraints one can trace which constraint in the adder32OPT file is relatively more important for the objective function and its sensitivity to the objective. The problem can be formulated to do delay, area and energy optimization. It can do robust optimization using the method described in [XXX ISQED paper reference]. Besides optimization, it can do timing analysis, draw PDFs, get the $\mu-\sigma$ scatter plot vectors, from optimization result files of previous runs and/or the present one, so that we can do comparison with the older results without having to produce them again. The

Figure 12. The optimization specification file

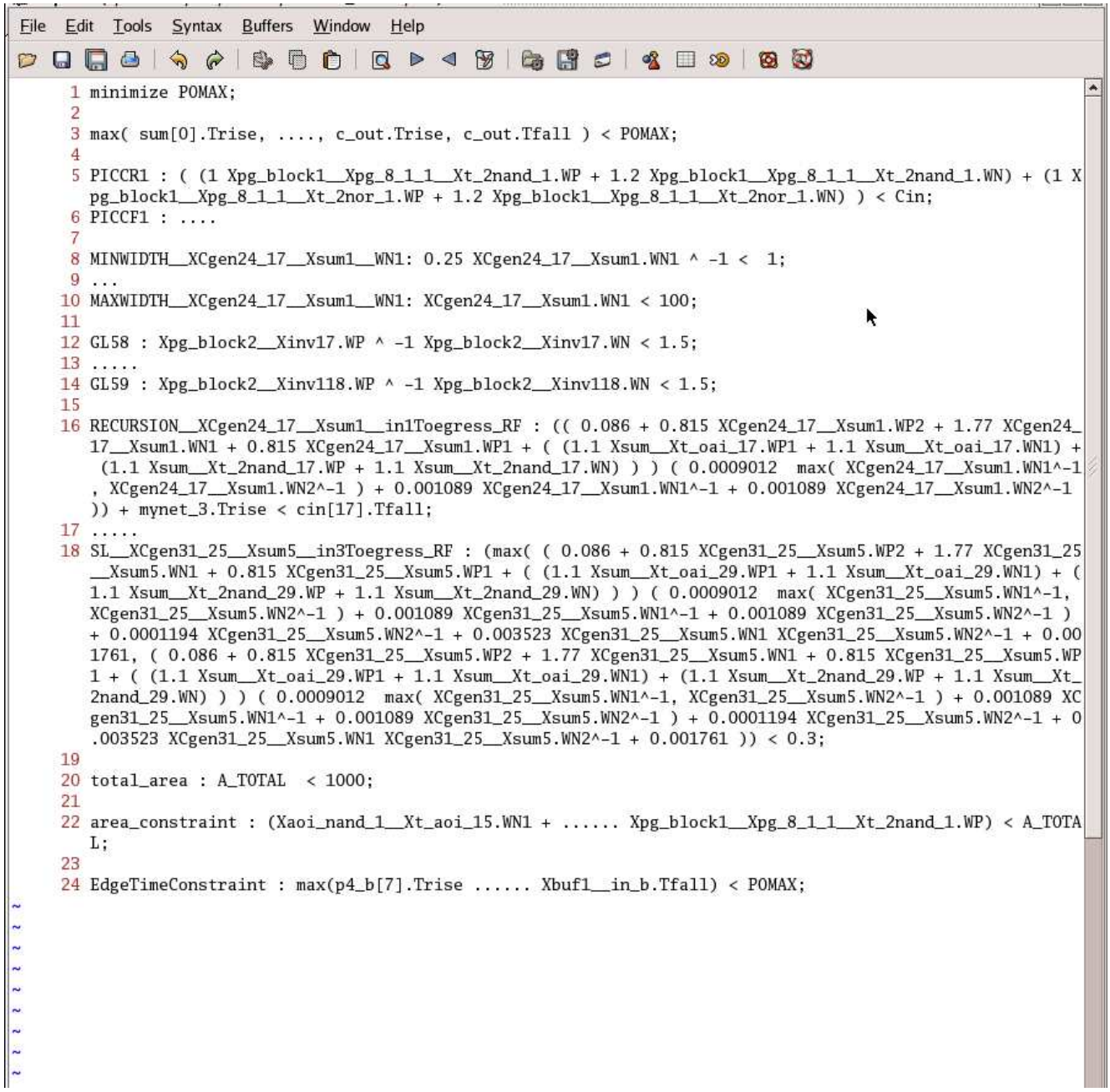


Figure 13. The problem file for optimization

result of some operations is a MATLAB file (like delay PDF generation for example), since one need to view the results as a plot. These files can be executed directly in MATLAB to obtain the desired plots.

5 GP solver

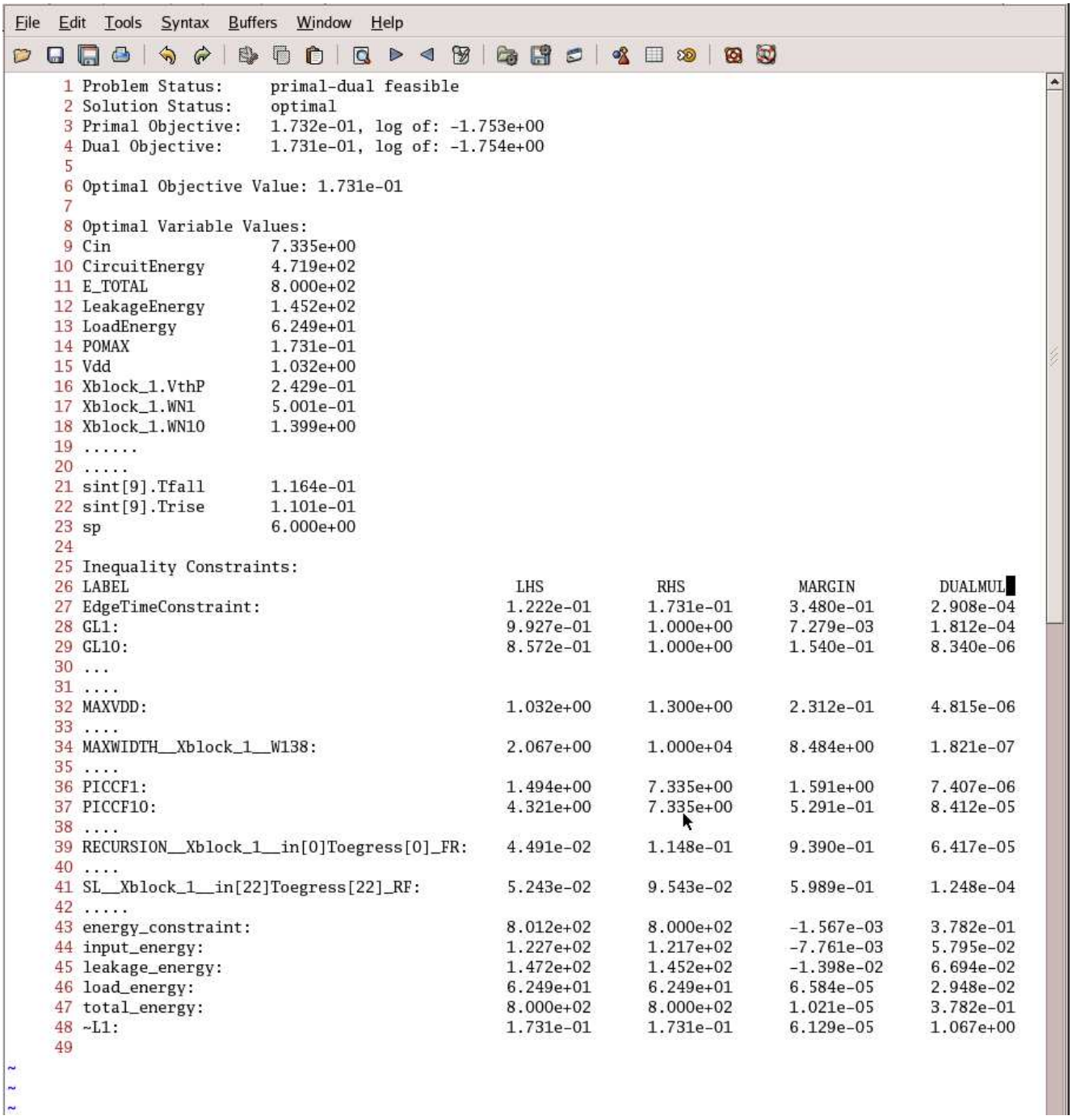
This software (originally written by Sunghee Yun) interfaces with the MOSEK optimization package. To understand this section completely needs an introduction to It takes two input files, one containing the problem and the other containing the various tolerances that determine the termination of the optimization procedure. An excerpt from an example input file is shown in figure 13. The structure of the file is typically as follows:

- The first line specifies a posynomial to minimize. In this case POMAX is our variable for maximum delay of all primary outputs.
- From second line onwards we start the constraints. The first one represents that the maximum of signal arrival time at any of the outputs is less than POMAX.
- Following the above constraint, there are constraints about input capacitance, constraints on width, V_{th} and V_{dd} etc. This is followed by the recursive constraints of the delay propagation of various CCCs. For each CCC we can write $RecursiveConstraintForRiseToFall:\ delay_{inputTooutput} + inputRiseTime = outputFallTime$
  All the stages are assumed to be inverting and both rise and fall delay are considered separately.

The output is written in a ".out" solution file that can be read by the software in the previous section for doing post optimization analysis etc. It is also read by various back annotation scripts. The accuracy of the solution depends on the amount of error tolerated for termination of the iterations. This is decided by the user and is recorded in a file called "paramFile". By changing the tolerances in this file, one can get solutions of arbitrary accuracy.

In general one can just generate such a file directly by hand for other geometric problems like repeater insertion etc. The basic structure of the file being very simple (A minimize statement followed by a bunch of constraints), the GP solver routine can be used independently to solve any generic GP problems. Given such a file, say tmpOPT one can solve for it directly using
$\$\$optim.pl\ -ggp\ tmpOPT$
resulting in the file tmpOPT.out. The structure of this file is shown in figure 14. As shown, the output file contain the optimal objective value, the optimal values of the design variables and the data about the inequality constraints. Some important variables are generated by the problem formulation program. Users needs to have a knowledge of what the variables represent so that they know what to extract. A list of the important variables is as follows.

1. POMAX : Denotes the overall delay of the circuit. It is computed as the maximum of the rise and fall arrival times at all the outputs.
2. E_TOTAL : The total energy of the system. It is the sum of the total dynamic and leakage energy.
3. A_TOTAL The total area of the circuit, calculated as the sum of the widths of all the transistors.
4. LeakageEnergy : Energy leaked by all the gates in the circuit. Expressed as the sum of the leakage energy of each CCC.
5. CircuitEnergy : Dynamic energy dissipated in driving the gates of the circuits (except the primary input gates).
6. InputEnergy : Energy consumed in driving the primary inputs of the netlist.
7. WireEnergy : Energy dissipated in wires in the circuit, including the wires inside the CCC. The wire capacitance is given by adding a capacitor. Note that for the optimizer, any explicitly defined capacitance is wire capacitance.
8. LoadEnergy : The energy dissipated in the load. It is important to separate this value, as in a cascade of systems this is equivalent to the InputEnergy of the next block.
9. Vdd : The supply voltage.



```
File  Edit  Tools  Syntax  Buffers  Window  Help
 1 Problem Status:     primal-dual feasible
 2 Solution Status:    optimal
 3 Primal Objective:   1.732e-01, log of: -1.753e+00
 4 Dual Objective:     1.731e-01, log of: -1.754e+00
 5
 6 Optimal Objective Value: 1.731e-01
 7
 8 Optimal Variable Values:
 9 Cin                 7.335e+00
10 CircuitEnergy       4.719e+02
11 E_TOTAL             8.000e+02
12 LeakageEnergy       1.452e+02
13 LoadEnergy          6.249e+01
14 POMAX               1.731e-01
15 Vdd                 1.032e+00
16 Xblock_1.VthP       2.429e-01
17 Xblock_1.WN1        5.001e-01
18 Xblock_1.WN10       1.399e+00
19 ......
20 .....
21 sint[9].Tfall       1.164e-01
22 sint[9].Trise       1.101e-01
23 sp                  6.000e+00
24
25 Inequality Constraints:
26 LABEL                                          LHS          RHS          MARGIN         DUALMUL
27 EdgeTimeConstraint:                          1.222e-01    1.731e-01    3.480e-01      2.908e-04
28 GL1:                                         9.927e-01    1.000e+00    7.279e-03      1.812e-04
29 GL10:                                        8.572e-01    1.000e+00    1.540e-01      8.340e-06
30 ...
31 ....
32 MAXVDD:                                      1.032e+00    1.300e+00    2.312e-01      4.815e-06
33 ....
34 MAXWIDTH__Xblock_1__W138:                    2.067e+00    1.000e+04    8.484e+00      1.821e-07
35 ....
36 PICCF1:                                      1.494e+00    7.335e+00    1.591e+00      7.407e-06
37 PICCF10:                                     4.321e+00    7.335e+00    5.291e-01      8.412e-05
38 ....
39 RECURSION__Xblock_1__in[0]Toegress[0]_FR:    4.491e-02    1.148e-01    9.390e-01      6.417e-05
40 ....
41 SL__Xblock_1__in[22]Toegress[22]_RF:         5.243e-02    9.543e-02    5.989e-01      1.248e-04
42 .....
43 energy_constraint:                           8.012e+02    8.000e+02    -1.567e-03     3.782e-01
44 input_energy:                                1.227e+02    1.217e+02    -7.761e-03     5.795e-02
45 leakage_energy:                              1.472e+02    1.452e+02    -1.398e-02     6.694e-02
46 load_energy:                                 6.249e+01    6.249e+01    6.584e-05      2.948e-02
47 total_energy:                                8.000e+02    8.000e+02    1.021e-05      3.782e-01
48 ~L1:                                         1.731e-01    1.731e-01    6.129e-05      1.067e+00
49
~
~
~
```

Figure 14. The result file containing optimal variables

10. netname.Trise/netname.Tfall : The rise and fall times at the given net. Note that these might be subject to degeneration as described in section 5.1.

11. meanTransitionName_RF/_FR : The delay from a given input to a given output of a CCC. The variable name consists of the input and output name, the CCC name as well as rise-fall or fall-rise information.

12. CCCname.WN/WP/VthN/VthP : These are as the name suggests the values of the sizing and threshold parameters associated with every CCC.

13. other variables : Users can have their own variables as long as they ensure that they are bounded above and below, else the optimization results in an error. Some examples are bit pitch, spacing, wire capacitance. It is better to have these as variables rather than hard coded numbers, so that you can change them once at the top level to specify a different fabrication environment for example.

If the problem is not feasible, it just indicates that in the first line of the file and no more information is provided. The designer has to figure out by him/herself what the offending constraints was. This is one of the main limitations of the tool.

5.1 Slacking Variables

When there multiple constraints in the problem, some are active and others have a slack. This can create situations where some variable in the entire problem specification can possibly take a range of values and still not change the result or other variable values. In this case there is a degeneracy and the value returned by the optimizer need not be the value you are looking for. For example, if the problem is to find delay under an energy constraint, but the energy is high enough so that the problem is constrained by input and output constraints, so that under these active constraints total energy is E_{tot}. But the variable E_TOTAL can take any value between E_{tot} and E_{max}, where E_{max} is the specified energy constraint. Hence it is important to verify that the constraint is active before using the value of the variable in that constraint. For avoiding this degeneracy, one can run an analysis step that essentially takes the optimal solution you generated and replace "$<$" by "$=$" to remove degeneracy and get the real value of all the variables.

5.2 Iterative solving to obtain tradeoff curve

One typically needs to sweep one of the constraint (say that of total energy) in the adder32OPT file over a range and optimize for the delay to obtain the energy delay tradeoff. This can be done using the command
$\$\$optim.pl\ -gpIter\ tmpOPT\ fooVAL\_MARKbar\ ConstraintName\ logFile$ 800 900 1000 ...$(list\ of\ numbers)$

This command takes the file tmpOPT changes the RHS of the constraint ConstraintName and replaces it by the numbers in the list. It creates new files called fooVAL_MARKbar where VAL_MARK is replaced by the numbers to produce new independent files. The optimization is run on these file to produce new ".out" files.

To get a tradeoff one needs to extract the values of the relevant variables from the solution files. For this the following command can be used.

$\$\$optim.pl\ -extract\ solutionFileNames\ data.dat$

The file solutionFileNames contains the names of all the .out files to read the solutions from and the variables to record. It stores it in a file data.dat in a matrix format that can be read in MATLAB. A format of the solution file is as shown in figure 15

6 Back Annotation

The back annotation perl scripts can put the values back in SPICE or SUE file. Putting them back in SUE is especially helpful since the user can see at a glance, how the devices were sized and can deduce if there is a need for buffer insertion at a point. Along with the optimal values, MOSEK also outputs the sensitivities to various constraints, which can be used to further understand which constraints are most important. This information can be used to snap the widths, Vths etc to discrete values. The back annotation can be done in SUE using the command

$\$ > optim.pl\ -bsu\ top_sue_file\ data_file$

This will generate a whole new hierarchy of SUE files. The filenames are appended with "_opt". If the DoNotUniquify option is not set and there are multiple instances of the same block, the script will create multiple different blocks with

```
Solution Files:
add16ED10000DET.out  add16ED16000DET.out
add16ED2400DET.out add16ED1000DET.out   add16ED1600DET.out
add16ED2600DET.out add16ED1100DET.out add16ED1700DET.out
add16ED2800DET.out
....
....
add16ED12000DET.out
add16ED1800DET.out
add16ED3000DET.out add16ED1200DET.out   add16ED180DET.out
add16ED300DET.out add16ED120DET.out    add16ED1900DET.out
add16ED3300DET.out add16ED1300DET.out   add16ED20000DET.out
add16ED4000DET.out

Variable Names:
POMAX E_TOTAL
CircuitEnergy
LogicEnergy WireEnergy
Cin Vdd
```

Figure 15. The file containing filenames and variable names

different sizes. So with the data file being tmpOPT.out and the top level sue file being adder32.sue, the script will generate adder32_opt.sue file and its schematic hierarchy. For back annotating in spice we use the command
$\$ > optim.pl\ -bsp\ data\_file\ original\_spiceFile\ modified\_spiceFile\ new\_spiceFile$

The new spice file (name specified by user) is produced by back annotating the data from data_file into the modified spice file. The original spice file is used to just get the header and other information if necessary. Note that the new spice file has hierarchy like the modified spice file and not like the original spice file.

7 Limitations and work around

1. Buffer Insertion: For optimal design, it is necessary to have the optimal number of logic stages for driving a particular load. Also in a given topology, a slow gate might be driving a huge fanout, in which case, putting an buffer there would really help. The tool does not have the buffer insertion feature. So the designer has to try out various buffering topologies from the sizing the tool returns to get the most optimal design.

2. Pulse shaping: For optimizing structures like pulsed flip flops, you need to guarantee a minimum pulse and maximum pulse size. At the same time you need to minimize the overall delay. This is easy to specify if one edge of the pulse is the input, but its difficult if the pulse is in the middle of the data path. In such cases, while guaranteing the minimum pulse size is possible, it is not possible to guarantee a maximum pulse size in the canonical geometric form. But we can play tricks to change the constraints in order to give us a maximum pulse timing.

3. Transmission gate inputs: The transmission gate inputs are supposed to arrive within a certain time of each other. This constraint is again non-convex. We cannot guarantee the time between the nMOS and the pMOS input of a transmission gate to be less than a certain value, but we can constraint the optimizer such that it makes the logic between the two inputs to have as small a time as possible. This is done by specifying the timing of the transmission gate as arising from the maximum of the arrival times at either of the inputs. Its a hack but it directs the optimizer in the right direction.

4. Accounting for leakage in the CCC accurately:

5. Specifying the activity factors for nets inside a CCC:

8 A Condensed example

Consider a 32-bit adder to be optimized for delay given energy. The following are the steps to obtain an optimized adder. Since the design flow starts with SUE, a file called adder32.sue is first entered in sue following all the design entry rules mentioned earlier. Lets say we are designing in 0.1μm technology. Following are the steps you would take to get a tradeoff curve.These steps do not cover the entire range of possibilities but give a good flavor of how to work with the tool. We shall use the perl script optim.pl to navigate through the design flow. For quick reference, just typing optim.pl at the prompt returns various options and a brief description of their actions as shown in section 1. Then one can type optim.pl with any one (and only one) of these options and see the syntax and file input arguments for that operation.

1. **Files needed a priori:**
   - Global parameters file(glb90n_par.sp) that definies the various prameters of the technology you are designing in. This file can be produced from characterization scripts like the one in ee371.
   - Basic delay model file(Tech9015var.dat): It defines the delay model for a chain of transistors. This file contains the mean and std. dev. expressions for the delay expressed as a function of sizing (or additionally Vdd, Vth etc). The std. devs. are used if one wishes to do statistical optimization or analysis.
   - IRSIM tech file (scmos90.prm) : This file defines the RC constants for the technology for switch level simulations in IRSIM done to obtain the activity and duty factors.
2. **Schematic entry:** Once the adder32.sue is made in SUE, and the primary constraints on input capacitance, output load, width bounds, slope bounds etc. are mentioned at the top level.

   It is important that all nets be named so that if one needs to generate the spice file again, once can still use the activity factor and duty factor results from IRSIM for the new optimization (for doing energy optimization). Also it is important to name all the cells and blocks for back annotation later on. The following command takes the top level sue file and names all nets (that are attached to the name-net hook) and instances recursively throughout the hierarchy.

   $$ optim.pl -psn adder32.sue

   The spice netlist generated for this schematic (adder32.sp) is used for further processing.
3. **Generate Modified Spice file** The spice file adder32.sp generated by sue is hierarchical and contains commented constraints that have to be extracted for the optimizer. This spice file is converted to a modified spice file which can be interpreted by the optimizer using the following command.

   $$ optim.pl -sp adder32.sp glb90n_par.sp adder32_mod.sp

   The file adder32_mod.sp is the modified spice file produced (like the one shown in figure 11
4. **Model generation:** The adder32_mod.sp file is now the file of relevance. It is used to generate delay models using the command

   $$ optim.pl -mod adder32_mod.sp Tech9015var.dat adder32_mod.dio

   You can generate the AFs and DFs also, but since the circuit is not sized properly, it is possible that the switch level simulation can give wrong answers. Hence one can first optimize the circuit with a reasonable area (or Cin Cout constraints) and then back annotate the sizes in a spice file which can be used for generating the switch statistics. One can use the option "-irsim" to get the AFs and DFs. These will be produces in the adder32.sp.power and adder32.sp.duty files and also included automatically in the adder32_mod.sp file. If you happen to change the adder32_mod.sp file from SUE, you can still just copy paste the date from the .duty and .power files and so do not have to run the switch level simulation again.

5. **Circuit optimization :** We are now ready for doing area-delay optimization. First the user needs to make a file called adder32.opt which specifies the optimization tasks s/he wants to get done. These include optimizing for delay , drawing PDFs using the previous or current solutions, getting the path mean and std. dev. for scatter plots, doing timing analysis using the solution in from a different optimization runs for comparison etc. Having made this file we can use the command

   $$ optim.pl -sol adder32_mod.sp adder32_mod.dio adder32.opt

   to formulate the problem and launch the solver. An intermediate file/s (named according to the specification given in adder32.opt) is/are generated and solved. Lets call one of these files tmpDDET. If the "OnlyFormulateProblem" option is set in adder32_mod.sp, the solver is not launched. The solution is obtained in the file tmpDDET.out(if the solver is launched). The post-analysis continues as specified by the user in adder32.opt. For solving the tmpDDET file separately one can use

   $$ optim.pl -ggp tmpDDET

   to get tmpDDET.out.

6. **Solving the tmpDDET file iteratively:** The file tmpDDET can be solved using the following command.
   $$ optim.pl -ggp tmpDDET

   This will produce the tmpDDET.out file. The user might want to change some constraint and run the optimization repetitively to get a tradeoff curve. This can be done using the following command.

   $$ optim.pl -gpIter tmpDDET fooVAL_MARKbar total_area run.log 800 900 ....

   The command takes in the tmpDDET file, changes the RHS of the constraint labeled "total_area", generates a file foo800bar for say simulation with area spec of 800 (foo900bar etc) and solves it. The simulation statistics are stored in the file run.log and the output file that result are foo800bar.out, foo900bar.out and so on. Note that the token "VAL_MARK" in the argument name is a necessary key and indicates the names of the new files generated. The relevant variables need to be extracted from the solution files for analysis purpose. A file called solutionFileNamesAndVariables is created by the user with the syntax shown in figure 15. Then using the command

   $$ optim.pl -extract solutionFileNamesAndVariables tradeoff.dat

   the user generates the file tradeoff.dat that can be loaded into MATLAB for drawing tradeoff curves.

7. **Back annotation :** After the optimization, the results can be back annotated in the adder32.sue or adder32.sp file using the following two commands.

   $$ optim.pl -bsu adder32.sue tmpDDET.out

   $$ optim.pl -bsp tmpDDET.out adder32.sp adder32_mod.sp adder32_opt.sp

   As explained earlier adder32_opt.sue is the top level cell generated for the new SUE hierarchy that has optimal sizes, while adder32_opt.sp is the spice file with optimal sizes.

Appendix 1: Basic geometric programming

Monomial and posynomial functions

Let $x_1, \ldots, x_n$ denote n real positive variables, and $x = (x_1, \ldots, x_n)$ a vector with components x_i. A real valued function f of x, with the form

$$f(x) = cx_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}, \tag{1}$$

where $c > 0$ and $a_i \in \mathbf{R}$, is called a *monomial function*, or more informally, a *monomial* (of the variables $x_1, \ldots, x_n$). We refer to the constant c as the *coefficient* of the monomial, and we refer to the constants $a_1, \ldots, a_n$ as the *exponents* of the monomial. As an example, $2.3x_1^2 x_2^{-0.15}$ is a monomial of the variables x_1 and x_2, with coefficient 2.3 and x_2-exponent -0.15.

Any positive constant is a monomial, as is any variable. Monomials are closed under multiplication and division: if f and g are both monomials then so are fg and f/g. (This includes scaling by any positive constant.) A monomial raised to any power is also a monomial:

$$f(x)^\gamma = (cx_1^{a_1} x_2^{a_2} \cdots x_n^{a_n})^\gamma = c^\gamma x_1^{\gamma a_1} x_2^{\gamma a_2} \cdots x_n^{\gamma a_n}.$$

The term 'monomial', as used here (in the context of geometric programming) is similar to, but differs from the standard definition of 'monomial' used in algebra. In algebra, a monomial has the form (1), but the exponents a_i must be nonnegative integers, and the coefficient c is one. Throughout this paper, 'monomial' will refer to the definition given above, in which the coefficient can be any positive number, and the exponents can be any real numbers, including negative and fractional.

A sum of one or more monomials, *i.e.*, a function the form

$$f(x) = \sum_{k=1}^{K} c_k x_1^{a_{1k}} x_2^{a_{2k}} \cdots x_n^{a_{nk}}, \tag{2}$$

where $c_k > 0$, is called a *posynomial function* or, more simply, a *posynomial* (with K terms, in the variables $x_1, \ldots, x_n$). The term 'posynomial' is meant to suggest a combination of 'positive' and 'polynomial'.

Any monomial is also a posynomial. Posynomials are closed under addition, multiplication, and positive scaling. Posynomials can be divided by monomials (with the result also a posynomial): If f is a posynomial and g is a monomial, then f/g is a posynomial. If γ is a nonnegative integer and f is a posynomial, then f^γ always makes sense and is a posynomial (since it is the product of γ posynomials).

Let us give a few examples. Suppose x, y, and z are (positive) variables. The functions (or expressions)

$$2x, \quad 0.23, \quad 2z\sqrt{x/y}, \quad 3x^2 y^{-.12} z$$

are monomials (hence, also posynomials). The functions

$$0.23 + x/y, \quad 2(1 + xy)^3, \quad 2x + 3y + 2z$$

are posynomials but *not* monomials. The functions

$$-1.1, \quad 2(1 + xy)^{3.1}, \quad 2x + 3y - 2z, \quad x^2 + \tan x$$

are not posynomials (and therefore, not monomials).

Standard form geometric program

A *geometric program* (GP) is an optimization problem of the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 1, \quad i = 1, \ldots, m, \\ & g_i(x) = 1, \quad i = 1, \ldots, p, \end{array} \tag{3}$$

where f_i are posynomial functions, g_i are monomials, and x_i are the optimization variables. (There is an implicit constraint that the variables are positive, *i.e.*, $x_i > 0$.) We refer to the problem (3) as a geometric program in *standard form*, to

distinguish it from extensions we will describe later. In a standard form GP, the objective must be posynomial (and it must be minimized); the equality constraints can only have the form of a monomial equal to one, and the inequality constraints can only have the form of a posynomial less than or equal to one.

As an example, consider the problem

$$\begin{array}{ll} \text{minimize} & x^{-1}y^{-1/2}z^{-1} + 2.3xz + 4xyz \\ \text{subject to} & (1/3)x^{-2}y^{-2} + (4/3)y^{1/2}z^{-1} \leq 1, \\ & x + 2y + 3z \leq 1, \\ & (1/2)xy = 1, \end{array}$$

with variables x, y and z. This is a GP in standard form, with $n = 3$ variables, $m = 2$ inequality constraints, and $p = 1$ equality constraints.

We can switch the sign of any of the exponents in any monomial term in the objective or constraint functions, and still have a GP. For example, we can change the objective in the example above to $x^{-1}y^{1/2}z^{-1} + 2.3xz^{-1} + 4xyz$, and the resulting problem is still a GP (since the objective is still a posynomial). But if we change the sign of any of the coefficients, or change any of the additions to subtractions, the resulting problem is not a GP. For example, if we replace the second inequality constraint with $x + 2y - 3z \leq 1$, the resulting problem is *not* a GP (since the lefthand side is no longer a posynomial).

The term *geometric program* was introduced by Duffin, Peterson, and Zener in their 1967 book on the topic [**?**]. It's natural to guess that the name comes from the many geometrical problems that can be formulated as GPs. But in fact, the name comes from the geometric-arithmetic mean inequality, which played a central role in the early analysis of GPs.

It is important to distinguish between *geometric programming*, which refers to the family of optimization problems of the form (3), and *geometric optimization*, which usually refers to optimization problems involving geometry. Unfortunately, this nomenclature isn't universal: a few authors use 'geometric programming' to mean optimization problems involving geometry, and vice versa.

Simple extensions of GP

Several extensions are readily handled. If f is a posynomial and g is a monomial, then the constraint $f(x) \leq g(x)$ can be handled by expressing it as $f(x)/g(x) \leq 1$ (since f/g is posynomial). This includes as a special case a constraint of the form $f(x) \leq a$, where f is posynomial and $a > 0$. In a similar way if g_1 and g_2 are both monomial functions, then we can handle the equality constraint $g_1(x) = g_2(x)$ by expressing it as $g_1(x)/g_2(x) = 1$ (since g_1/g_2 is monomial). We can maximize a nonzero monomial objective function, by minimizing its inverse (which is also a monomial).

As an example, consider the problem

$$\begin{array}{ll} \text{maximize} & x/y \\ \text{subject to} & 2 \leq x \leq 3, \\ & x^2 + 3y/z \leq \sqrt{y}, \\ & x/y = z^2, \end{array} \tag{4}$$

with variables $x,\ y,\ z \in \mathbf{R}$ (and the implicit constraint $x,\ y,\ z > 0$). Using the simple transformations described above, we obtain the equivalent standard form GP

$$\begin{array}{ll} \text{minimize} & x^{-1}y \\ \text{subject to} & 2x^{-1} \leq 1, \quad (1/3)x \leq 1, \\ & x^2y^{-1/2} + 3y^{1/2}z^{-1} \leq 1, \\ & xy^{-1}z^{-2} = 1. \end{array}$$

It's common to refer to a problem like (4), that is easily transformed to an equivalent GP in the standard form (3), also as a GP.

Example

Here we give a simple application of GP, in which we optimize the shape of a box-shaped structure with height h, width w, and depth d. 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. Subject to these constraints, we wish to maximize the volume of the structure, hwd. This leads to the problem

$$\begin{array}{ll} \text{maximize} & hwd \\ \text{subject to} & 2(hw+hd) \leq A_{\text{wall}}, \quad wd \leq A_{\text{flr}}, \\ & \alpha \leq h/w \leq \beta, \quad \gamma \leq d/w \leq \delta. \end{array} \tag{5}$$

Here d, h, and w 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). This problem is a GP (in the extended sense, using the simple transformations described above). It can be transformed to the standard form GP

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

How GPs are solved

As mentioned in the introduction, the main motivation for GP modeling is the great efficiency with which optimization problems of this special form can be solved. To give a rough idea of the current state of the art, standard *interior-point* algorithms can solve a GP with 1000 variables and 10000 constraints in under a minute, on a small desktop computer (see [**?**]). For sparse problems (in which each constraint depends on only a modest number of the variables) far larger problems are readily solved. A typical sparse GP with 10000 variables and 1000000 constraints, for example, can be solved in minutes on a desktop computer. (For sparse problems, the solution time depends on the particular sparsity pattern.) It's also possible to optimize a GP solver for a particular application, exploiting special structure to gain even more efficiency (or solve even larger problems).

In addition to being fast, interior-point methods for GPs are also very robust. They require essentially no algorithm parameter tuning, and they require no starting point or initial guess of the optimal solution. They *always* find the (true, globally) optimal solution, and when the problem is infeasible (*i.e.*, the constraints are mutually inconsistent), they provide a certificate showing that no feasible point exists. General methods for NLP can be fast, but are not guaranteed to find the true, global solution, or even a feasible solution when the problem is feasible. An initial guess must be provided, and can greatly affect the solution found, as well as the solution time. In addition, algorithm parameters in general purpose NLP solvers have to be carefully chosen.

In the rest of this section, we give a brief description of the method used to solve GPs. This is not because the GP modeler needs to know how GPs are solved, but because some of the ideas will resurface in later discussions.

The main trick to solving a GP efficiently is to convert it to a nonlinear but *convex optimization problem*, *i.e.*, a problem with convex objective and inequality constraint functions, and linear equality constraints. Efficient solution methods for general convex optimization problems are well developed [**?**]. The conversion of a GP to a convex problem is based on a logarithmic change of variables, and a logarithmic transformation of the objective and constraint functions. In place of the original variables x_i, we use their logarithms, $y_i = \log x_i$ (so $x_i = e^{y_i}$). Instead of minimizing the objective f_0, we minimize its logarithm $\log f_0$. We replace the inequality constraints $f_i \leq 1$ with $\log f_i \leq 0$, and the equality constraints $g_i = 1$ with $\log g_i = 0$. This results in the problem

$$\begin{array}{ll} \text{minimize} & \log f_0(e^y) \\ \text{subject to} & \log f_i(e^y) \leq 0, \quad i = 1, \ldots, m, \\ & \log g_i(e^y) = 0, \quad i = 1, \ldots, p, \end{array} \tag{6}$$

with variables $y = (y_1, \ldots, y_n)$. Here we use the notation e^y, where y is a vector, to mean componentwise exponentiation: $(e^y)_i = e^{y_i}$.

This new problem (6) doesn't look very different from the original GP (3); if anything, it looks more complicated. But unlike the original GP, this transformed version is convex, and so *can be solved very efficiently*. (See [**?**] for convex optimization problems, including methods for solving them; §4.5 gives more details of the transformation of a GP to a convex problem.)

It's interesting to understand what it means for the problem (6) to be convex. We start with the equality constraints. Suppose g is a monomial,

$$g(x) = cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}.$$

Under the transformation above, it becomes

$$\begin{aligned}\log g(e^y) &= \log c + a_1 \log x_1 + \cdots + a_n \log x_n \\ &= \log c + a_1 y_1 + \cdots + a_n y_n,\end{aligned}$$

which is an *affine* function of variables y_i. (An affine function is a linear function plus a constant.) Thus, a monomial equality constraint $g = 1$ is transformed to a *linear equation* in the new variables,

$$a_1 y_1 + \cdots + a_n y_n = -\log c.$$

(In a convex optimization problem, all equality constraint functions must be linear.)

The posynomial inequality constraints are more interesting. If f is a posynomial, the function

$$F(y) = \log f(e^y)$$

is convex, which means that for any $y,\ \tilde{y}$, and any θ with $0 \leq \theta \leq 1$, we have

$$F(\theta y + (1-\theta)\tilde{y}) \leq \theta F(y) + (1-\theta)F(\tilde{y}). \tag{7}$$

The point $\theta y + (1-\theta)\tilde{y}$ is a (componentwise) weighted arithmetic mean of y and $\tilde{y}$. Convexity means that the function F, evaluated at a weighted arithmetic mean of two points, is no more than the weighted arithmetic mean of the function F evaluated at the two points. (For much more on convexity, see [**?**]).

In terms of the original posynomial f and variables x and $\tilde{x}$, the convexity inequality above can be stated as

$$f(x_1^\theta \tilde{x}_1^{1-\theta}, \ldots, x_n^\theta \tilde{x}_n^{1-\theta}) \leq f(x_1, \ldots, x_n)^\theta f(\tilde{x}_1, \ldots, \tilde{x}_n)^{1-\theta}. \tag{8}$$

The point with coefficients $x_i^\theta \tilde{x}_i^{1-\theta}$ is a weighted *geometric* mean of x and $\tilde{x}$. The inequality (8) above means that the posynomial f, when evaluated at a weighted geometric mean of two points, is no more than the weighted geometric mean of the posynomial f evaluated at the two points. This is a very basic property of posynomials, which we'll encounter later.

We emphasize that in most cases, the GP modeler does not need to know how GPs are solved. The transformation to a convex problem is handled entirely by the solver, and is completely transparent to the user. To the GP modeler, a GP solver can be thought of as a reliable black box, that solves any problem put in GP form. This is very similar to the way a numerical linear algebra subroutine, such as an eigenvalue subroutine, is used.

Feasibility, trade-off, and sensitivity analysis

Feasibility analysis

A basic part of solving the GP (3) is to determine whether the problem is feasible, *i.e.*, to determine whether the constraints

$$f_i(x) \leq 1, \quad i = 1, \ldots, m, \qquad g_i(x) = 1, \quad i = 1, \ldots, p \tag{9}$$

are mutually consistent. This task is called the *feasibility problem*. It is also sometimes called the *phase I problem*, since some methods for solving GPs involve two distinct phases: in the first, a feasible point is found (if there is one); in the second, an optimal point is found.

If the problem is infeasible, there is certainly no optimal solution to the GP problem (3), since there is no point that satisfies all the constraints. In a practical setting, this is disappointing, but still very useful, information. Roughly speaking, infeasibility means the constraints, requirements, or specifications are too tight, and cannot be simultaneously met; at least one constraint must be relaxed.

When a GP is infeasible, it is often useful to find a point $\hat{x}$ that is as close as possible to feasible, in some sense. Typically the point $\hat{x}$ is found by minimizing some measure of infeasibility, or constraint violation. (The point $\hat{x}$ is not optimal for the original problem (3) since it is not feasible.) One very common method is to form the GP

$$\begin{array}{ll} \text{minimize} & s \\ \text{subject to} & f_i(x) \leq s, \quad i = 1, \ldots, m, \\ & g_i(x) = 1, \quad i = 1, \ldots, p, \\ & s \geq 1, \end{array} \tag{10}$$

where the variables are x, and a new scalar variable s. We solve this problem (which itself is always feasible, assuming the monomial equality constraints are feasible), to find an optimal $\bar{x}$ and $\bar{s}$. If $\bar{s} = 1$, then $\bar{x}$ is feasible for the original GP; if $\bar{s} > 1$, then the original GP is not feasible, and we take $\hat{x} = \bar{x}$. The value $\bar{s}$ tells us how close to feasible the original problem is. For example, if $\bar{s} = 1.1$, then the original problem is infeasible, but, roughly speaking, only by 10%. Indeed, $\bar{x}$ is a point that is within 10% of satisfying all the inequality constraints.

There are many variations on this method. One is based on introducing independent variables s_i for the inequality constraints, and minimizing their product:

$$\begin{array}{ll} \text{minimize} & s_1 \cdots s_m \\ \text{subject to} & f_i(x) \leq s_i, \quad i = 1, \ldots, m, \\ & g_i(x) = 1, \quad i = 1, \ldots, p, \\ & s_i \geq 1, \quad i = 1, \ldots, m, \end{array} \tag{11}$$

with variables x and $s_1, \ldots, s_n$. Like the problem above, the optimal s_i are all one when the original GP is feasible. When the original GP is infeasible, however, the optimal x obtained from this problem typically has the property that it satisfies most (but not all) of the inequality constraints. This is very useful in practice since it suggests which of the constraints should be relaxed to achieve feasibility. (For more on methods for obtaining points that satisfy many constraints, see [**?**, §11.4].)

GP solvers unambiguously determine feasibility. But they differ in what point (if any) they return when a GP is determined to be infeasible. In any case, it is always possible to set up and solve the problems described above (or others) to find a potentially useful 'nearly feasible' point.

Trade-off analysis

In *trade-off analysis* we vary the constraints, and see the effect on the optimal value of the problem. This reflects the idea that in many practical problems, the constraints are not really set in stone, and can be changed, especially if there is a compelling reason to do so (such as a drastic improvement in the objective obtained).

Starting from the basic GP (3), we form a *perturbed GP*, by replacing the number one that appears on the righthand side of each constraint with a parameter:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & f_i(x) \leq u_i, \quad i = 1, \ldots, m, \\ & g_i(x) = v_i, \quad i = 1, \ldots, p. \end{array} \tag{12}$$

Here u_i and v_i are positive constants. When $u_i = 1$ and $v_i = 1$, this reduces to the original GP (3). We let $p(u, v)$ denote the optimal value of the perturbed problem (12) as a function of u and v. Thus, the value $p(\mathbf{1}, \mathbf{1})$ (where $\mathbf{1}$ denotes a vector with all components one) is equal to the optimal value of the original GP (3).

If $u_i > 1$, then the ith inequality constraint for the perturbed problem,

$$f_i(x) \leq u_i,$$

is *loosened*, compared to the inequality in the standard problem,

$$f_i(x) \leq 1.$$

Conversely, if $u_i < 1$, then the ith inequality constraint for the perturbed problem is *tightened* compared to the ith inequality in the standard problem. We can interpret the loosening and tightening quantitatively: for $u_i > 1$, we can say that the ith inequality constraint has been loosened by $100(u_i - 1)$ percent; for $u_i < 1$, then we can say that the ith inequality constraint has been tightened by $100(1 - u_i)$ percent. Similarly, the number v_i can be interpreted as a *shift* in the ith equality constraint.

It's important to understand what $p(u, v)$ means. It gives the optimal value of the problem, after we perturb the constraints, and then *optimize again*. When u and v change, so does (in general) the associated optimal point. There are several other perturbation analysis problems one can consider. For example, we can ask how sensitive a particular point x is, with respect to the objective and constraint functions (*i.e.*, we can ask how much f_i and g_i change when we change x). But this perturbation analysis is unrelated to trade-off analysis.

In optimal trade-off analysis, we study or examine the function $p(u, v)$ for certain values of u and v. For example, to see the optimal trade-off of the ith inequality constraint and the objective, we can plot $p(u, v)$ versus u_i, with all other u_j and all

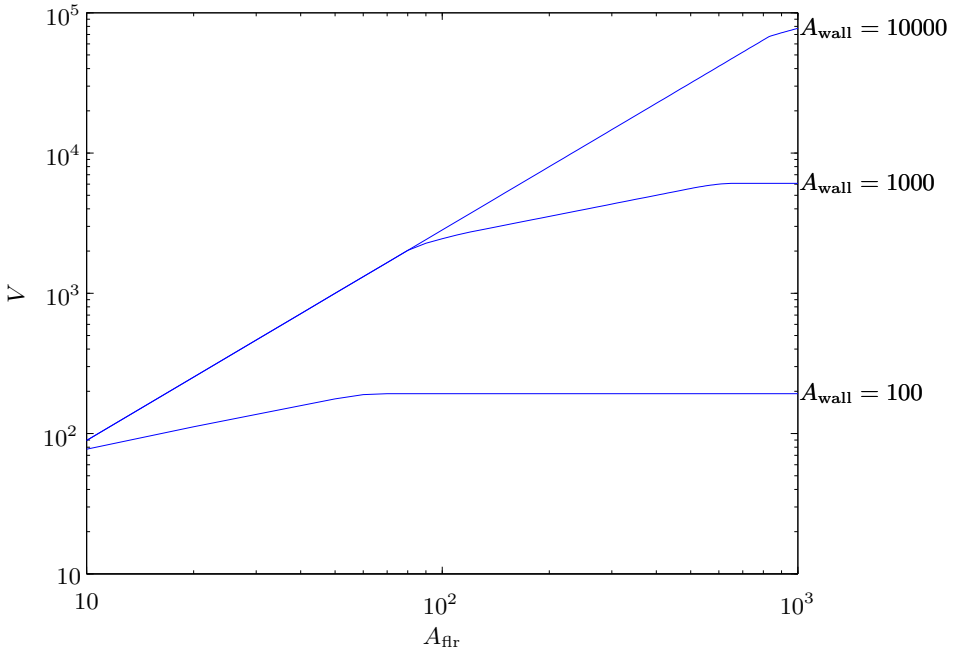


Figure 16. Optimal trade-off curves of maximum volume V versus maximum floor area A_{flr}, for three values of maximum wall area A_{wall}.

v_j equal to one. The resulting curve, called the *optimal trade-off curve*, passes through the point given by the optimal value of the original GP when $u_i = 1$. As u_i increases above one, the curve must decrease (or stay constant), since by relaxing the ith constraint we can only improve the optimal objective. The optimal trade-off curve flattens out when u_i is made large enough that the ith constraint is no longer relevant. When u_i is decreased below one, the optimal value increases (or stays constant). If u_i is decreased enough, the perturbed problem can become infeasible.

When multiple constraints are varied, we obtain an *optimal trade-off surface*. One common approach is to plot trade-off surfaces with two parameters as a set of trade-off curves for several values of the second parameter. An example is shown in figure 16, which shows optimal trade-off curves of optimal (maximum) volume versus A_{flr}, for three values of A_{wall}, for the simple example problem (5) given on page 21. The other problem parameters are $\alpha = 0.5$, $\beta = 2$, $\gamma = 0.5$, $\delta = 2$.

The optimal trade-off curve (or surface) can be found by solving the perturbed GP (12) for many values of the parameter (or parameters) to be varied. Another common method for finding the trade-off curve (or surface) of the objective and one or more constraints is the *weighted sum method*. In this method we *remove* the constraints to be varied, and add positive weighted multiples of them to the objective. (This results in a GP, since we can always add a positive weighted sum of monomials or posynomials to the objective.) For example, assuming that the first and second inequality constraints are to be varied, we would form the GP

$$\begin{array}{ll} \text{minimize} & f(x) + \lambda_1 f_1(x) + \lambda_2 f_2(x) \\ \text{subject to} & f_i(x) \leq 1, \quad i = 3, \ldots, m, \\ & g_i(x) = 1, \quad i = 1, \ldots, p. \end{array} \tag{13}$$

By solving the weighted sum GP (13), we always obtain a point on the optimal trade-off surface. To obtain the surface, we solve the weighted sum GP for a variety of values of the weights. This weighted sum method is closely related to *duality theory*, a topic beyond the scope of this tutorial; we refer the reader to [**?**] for more details.