LITE Optimization

Jagannathan Arjun Sathyamoorthy and Walter Unglaub

September 30, 2018

Abstract

We attempt to tackle the problem of automation of the design of CMOS Analog Integrated Circuits using techniques from Simulation Based Learning and Convex Optimization. Our focus is to develop a library of known good designs (KGDs) and then use convex optimization techniques around the known good designs to achieve the desired specifications. As is well known within the community, the optimization problem does not have a convex geometry requiring us to perform data-fitting techniques to leverage convex optimization methods.

Introduction

Digital circuit design has been well automated, but automation of the design of Analog and Analog Mixed Signal (AMS) circuits has evaded us primarily owing to the need of delicate optimization of transistor sizings to meet specifications; which usually requires intuition and design experience. In this project, we attempt at automating/pseudo-automating design of some AMS circuits. In our attempt to tackle this problem, we have chosen a fixed Phase Locked Loop (PLL) architecture, and in the first phase of the optimization we attempt to optimize the Voltage Controlled Oscillator (VCO). With the view of making the project open source we are using LTspice for design and simulation of our circuits, and GNU Octave to perform the optimization. Additionally, all code works with MATLAB as well.

Our optimization approach has two fundamental ideas: (1) Memory is relatively cheap – creating a Library of known good designs, (2) Local optimization is not very computationally expensive and likely to provide a solution. The impetus for this approach is that at every successful optimization we have a new addition to our library and increasing the likelihood of meeting a user desired specification.

Geometric Programming: A geometric program (GP) is a type of mathematical optimization problem characterized by objective and constraint functions that have a special form; which we call the GP format. Formulating a practical problem in the GP format is called GP modeling. However, GP modeling involves some knowledge of the system, as well as creativity, to be done effectively. Moreover, success isn’t guaranteed as many problems can't be represented as GPs. In comparison with Non-Linear Optimization methods, the fundamental difference is where the burden of intelligence is placed. In Non-Linear Optimization methods, modelling of data/system is relatively easy as the objective function and the constraint functions can be non-linear functions, but solving the Non-Linear Optimization problem is very tricky, time consuming and we might have to settle for a compromise (accepting a local solution instead of a global solution). In GP modelling of the data/system the objective function and the constraint equations must be limited to an acceptable GP format, but if we succeed then the GP provides us with the unique global solution effectively, as well as scaling very gracefully.

High-level Optimization flow

We begin the description of our design optimization flow in Fig. [1], which has three stages.

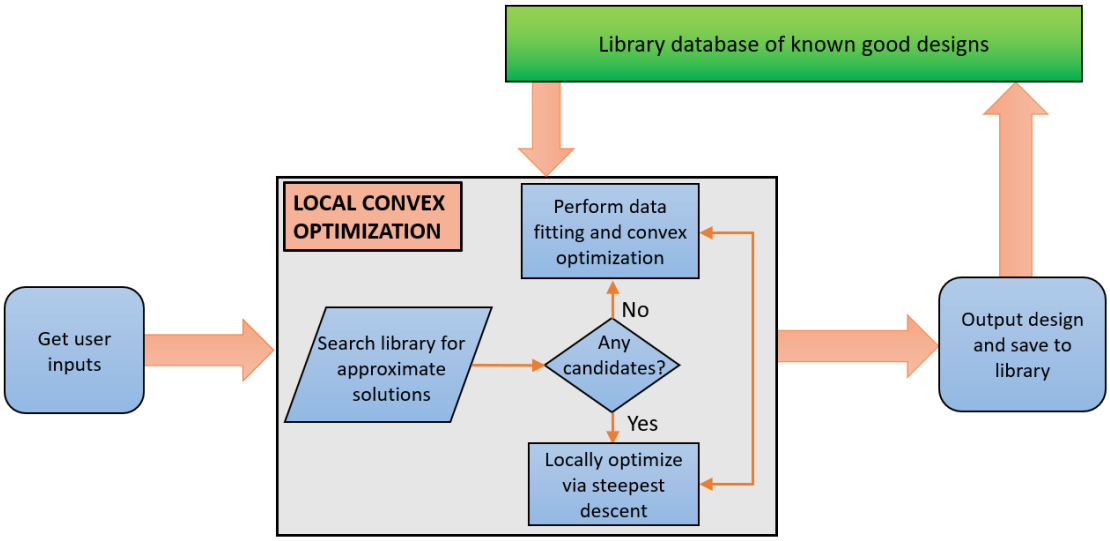

Figure 1: Flowchart

Stage 1:

We take specifications from the user which we call *Inputs* along with their priorities which we address as *Weights* and tolerances *Tol*. Using *Inputs* we search our Library of Known Good designs (KGDs) for solutions, i.e. designs "close" to the user requirements (the notion of distance/closeness is made clear in the next section). If we do find a solution within the specified *Tol* we return the solution, i.e. the *Netlist* to the user. If not, then we proceed to Stage 2.

Stage 2:

We take a heuristically-determined finite number of KGDs "close enough" to the *Inputs* and order them *S*, in an ascending order wrt. the distance. We take the first KGD *S*[1] and perform Steepest Descent aided by Simulation, using it as the starting point. In the Steepest Descent aided by Simulation, the direction of descent is decided by repeated simulations carried out in LTspice. The guiding principle behind this approach is that if we are sufficiently close to the user specs, then we might be able to attain a minimum satisfying *Inputs* locally. If we attain a minimum which satisfies *Tol* post-simulation, we return the *Netlist* to the user and **save the solution to our library**. If the *S*[1] does not yield a solution then we proceed to *S*[2] and so on till we exhaust *S*. If no solutions are found, then proceed to Stage 3.

Stage 3:

We take a uniform sample KGDs from the library and perform GP modelling on the data. Once we have modelled the data to fit the GP format we invoke a convex optimizer to perform the minimization of the cost function using a Geometric Program. Using the minimization parameters given by the GP unit, we simulate the circuit using LTspice. If the solution satisfies *Tol* we return the solution to the user and **save the solution to our library**. If the solution does not meet the target tolerance, we perform Steepest Descent aided by Simulation as in Stage 2, using the solution of the GP as the starting point until a solution is obtained and then **save the solution to our library**. If no solution is yet obtained, however, we inform the user that the Optimizer cannot optimize and produce a circuit meeting the input specifications.

A few remarks are in order. (1) Simulations are time-consuming, and Steepest Descent can be slow in a large parameter space. Hence, Steepest Descent aided by Simulation is a computationally expensive process. This forces us to heuristically determine the size of *S* to optimize the number of steepest descent searches one would have to endure in the worst case. (2) The main impetus for GP modelling is that when we fit our KGD data to the GP format, we can exploit the GP which guarantees a global minimum. This minimum is achieved by data fitting and not raw simulation, and hence we have to simulate to test and validate. Given that GPs can be solved very fast and the existence of global minimum is guaranteed, we can use that as a starting point for our Steepest Descent if need be, rather than Random Steepest Descent search algorithms. (3) Library creation entails two hurdles: (a) a well-organized header with all the specifications of the circuit and the body would could contain the transistor sizing values and (b) a validation function which we also use to validate if a certain circuit from either Stage 2 or Stage 3 satisfy the primary circuit objectives. To initialize the library, we begin with a coarse uniform grid in our parameter space and the validation function is circuit dependent.

Mathematical Framework:

Our variable parameters for design are the widths of the PMOS and NMOS transistors which we represent as W_p^i and W_n^j, respectively. Given a fixed architecture, we have a fixed number of different PMOS and NMOS transistors to optimize, namely $1 \le i \le N_{\text{PMOS}}$ and $1 \le j \le N_{\text{NMOS}}$. We view each of the design specifications f_D as a function from the parameter space to $\mathbb{C}$, i.e.

$$f_D : \mathbb{R}_{\ge L}^{N_{\text{PMOS}}} \times \mathbb{R}_{\ge L}^{N_{\text{NMOS}}} \to \mathbb{C} ,$$

where L is the minimum transistor length for the given process file. Given user *Inputs* = (y_{D_1}, …, y_{D_k}) along with priorities *Weights* = (w_{D_1}, …, w_{D_k}) we define our cost function as

$$l = \sum_{i=1}^{k} w_{D_i} \left(f_{D_i} - y_{D_i}\right)^2 ,$$

where we necessitate that $w_{D_i} \ge 0 \; \forall \; i$ and $\sum_{i=1}^{k} w_{D_i} = 1$. This permits us to view l as

$$l : \mathbb{R}_{\ge L}^{N_{\text{PMOS}}} \times \mathbb{R}_{\ge L}^{N_{\text{NMOS}}} \to \mathbb{R}_{\ge 0} ,$$

and the problem boils down to minimizing the cost function. We attempt to solve this constrained optimization problem, i.e. find $\left(W_{p,\mathrm{sol}}^{i}, W_{n,\mathrm{sol}}^{j}\right) \in \mathbb{R}_{\geq L}^{N_{\mathrm{PMOS}}} \times \mathbb{R}_{\geq L}^{N_{\mathrm{NMOS}}}$ such that $l\left(W_{p,\mathrm{sol}}^{i}, W_{n,\mathrm{sol}}^{j}\right) < Tol$.

Since the cost function *l* has an unknown closed form wrt. W_p^i and W_n^j, it is difficult to verify if conditions such as convexity, continuity, and differentiability are satisfied. The derivative must be calculated numerically using finite difference. In each iteration of the steepest-descent algorithm, one then has to calculate $N = N_{\mathrm{PMOS}} + N_{\mathrm{NMOS}}$ partial derivatives of the function. This implies that in each iteration of the Steepest-Descent algorithm, one would require *N* function evaluations. Since each function evaluation requires a time-consuming simulation, it is clear that as *N* increases, the number of simulations needed increases and consequently just one iteration of steepest descent may take a significant amount of time. This, however, would indeed give us a fool-proof way of determining the solution if it exists, and hence we must make a heuristic trade-off on the size of *S* in the previous section. The distance is measured in terms of the cost function *l*.

Given the constraints of Steepest Descent aided by Simulation, we attempt to fit the data with a function that is compatible with a GP, inspired by Boyd et. al., and Hoburg et. al. We take a large uniform sample of KGDs and evaluate the cost function for each of them. Then we fit the data with max-affine functions. We then improve the max-affine fit using an extended function class based on the softmax affine (SMA) functions and further refine to an implicit class of functions called implicit softmax affine (ISMA) class. Each of the above proposed function classes are directly compatible with Posynomial constrain forms in GP. The fitting problem is formulated as a nonlinear least squares regression and solved locally using a Levenberg-Marquardt algorithm. We take *m* data points from our KGD library and consider the problem of fitting a multivariate function $f(\mathbf{w})$ to the set of *m* data points

$$\{(\mathbf{w}_i, l_i) \in \mathbb{R}^N \times \mathbb{R}, i = 1, \ldots, m\}$$

and compute

$$\min_{f \in \mathcal{F}} \sum_{i=1}^{m} (f(\mathbf{w}_i) - l_i)^2$$

The max-affine (MA) functions with *K* terms would then be of the form

$$f_{\mathrm{MA}}(\mathbf{w}) = \max_{k=1,\ldots,K} [b_k + \mathbf{a}_k^T \mathbf{w}] \,,$$

where $b_k \in \mathbb{R}$ and $\mathbf{a}_k \in \mathbb{R}^N$, giving us $K(N + 1)$ parameters to define *f*. The reason *f* is convex is because it is the supremum over a set of affine functions. The drawback is that we usually need a large number of *K* to have high accuracy. A better approximation is the softmax affine class, where given a max-affine function as above we define f_{SMA} as

$$f_{\text{SMA}}(\mathbf{w}) = \frac{1}{\alpha}\log\sum_{k=1}^{K} e^{\alpha(b_k+\mathbf{a}_k^T\mathbf{w})}$$

where $\alpha \in \mathbb{R}_{>0}$ and we have a total of $K(N + 1) + 1$ parameters. The implicit softmax-affine function goes one step further and expresses a relationship between ($\mathbf{w}$, l) via the zero of an implicit function, i.e.

$$\hat{f}_{\text{ISMA}}(\mathbf{w}) = \log\sum_{k=1}^{K} e^{\alpha_k(b_k+\mathbf{a}_k^T\mathbf{w})}$$

where $\alpha_k \in \mathbb{R}_{>0}$, ergo we have $K(N + 2)$ independent parameters. Owing to its implicit nature we do not have an explicit formula for computing f_{ISMA} and we have to resort to a Newton-Raphson method to solve for f_{ISMA} from $\hat{f}_{\text{ISMA}}(\mathbf{w}) = 0$.

We have been able to observe that the RMS error is at least an order of magnitude better while going from $f_{\text{MA}} \rightarrow f_{\text{SMA}} \rightarrow f_{\text{ISMA}}$. Because all the functions above preserve convexity we can then use convex optimization techniques on them. Once we fit the data to the above function of choice we then convert them to equivalent posynomial functions and then invoke the freely-available *CVX* code (available at http://cvxr.com/cvx/) and use a geometric program to solve the problem. The minimizing $\mathbf{w}$ is then fed to the *Netlist* and simulated in LTspice for validation.

Voltage Controlled Oscillator

Finally, our design example has been a Current-Starved CMOS VCO with 9 stages designed and simulated with a 65-nm Predictive Technologies Model (PTM). Possible user inputs are listed in Table [1] below.

Table 1: Input metrics for circuit optimization

Inputs	**Description**	**Weights**
f_0	Nominal frequency	w_{f_0}
Δf	Tuning range	$w_{\Delta f}$
$J(\sigma_T)$	Jitter	w_J
P_{max}	Max. power	w_P
SL	Load sensitivity	w_{SL}

and hence $Inputs = (f_0, \Delta f, J(\sigma_T), P_{\text{max}}, SL)$. Our parameter space for the VCO model is limited to $\{W_p, W_p, W_{n,\text{bias}}\}$ for the moment.

Software

All software required to execute LITE optimization is freely available online and open source. The circuit simulation software utilized in this effort is LTspice, and the installation executable for Windows 7, 8, and 10 is available for download at the Analog Devices website (http://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html). While all the LITE code for interfacing with LTspice and performing optimization was originally written in and executable in MATLAB, it was modified to be compatible with GNU Octave, which is free to download at the GNU Projects website (https://www.gnu.org/software/octave/). All LITE code was developed using Windows 10, so proper execution is not guaranteed for other operating systems at this moment in time.

Main program script

LITE_MAIN.m

This is the main script, which when run, prompts the user for all relevant inputs. The types of inputs range from parameter values for the circuit being optimized, to technology files used in the circuit, as well as all metrics, constraints, and tolerance criteria used for database and gradient-descent searching. After the user has inputted all required information, the program checks for the existence of a database. If no database is found, it checks for the existence of a metric grid defined in terms of the set of user-specified parameters to vary in the circuit (such as transistor widths, etc.). If no such matrix exists in the folder, the program will prompt the user for specifications to generate a metric grid (using *grid_gen.m*) used for seeding an initial database (generated by *database_gen.m*). Once this is complete, the main *pipeline* function is called and the optimization flow begins. All inputs are saved to data files in the same location as the script, to be used by the various functions listed below.

Functions

grid_gen.m

This subroutine generates a matrix of metrics (for now, just the nominal frequency, f_0) corresponding to a range of values for a set of variable circuit parameters (for now, the set $\{W_p, W_p, W_{n,\text{bias}}\}$). This function makes use of the *edit_extract* subroutine to interface with LTspice, simulate a circuit with a set of parameter values in the given range, and extract the simulation data to compute the metric. This function also utilizes the *evaluator* subroutine to check the validity of the simulation result for each point in the parameter landscape, where "validity" is defined by the user constraints (e.g., acceptable noise margin and jitter). If the output of a simulation is deemed invalid, a value of "NaN" is supplied to the matrix element corresponding to the set of variable coordinates used to yield the invalid output via LTspice. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 1: ***grid_gen.m*** I/O summary

Inputs	Description
switches	Miscellaneous options for plotting/saving figures and saving data.
inputs	A cell array containing all user-inputted information needed for schematic editing and data extraction.

metrics	A cell array containing all user-inputted desired values for circuit optimization, including metrics, tolerances, and bounds.
nsteps	The number of steps along each variable axis to generate the initial metric matrix. The number of elements in the output matrix (and therefore number of simulations to perform) is equal to nsteps^nvars, where nvars is the number of variables.
Outputs	**Description**
metric_grid	The output matrix has dimensionality *nsteps^nvars* for a single metric (here, f_0). Values can be either real numbers or NaN (if output is deemed invalid for a particular set of user-defined constraints).
var_arrays	The variable parameter arrays used for generating the metric matrix; each parameter array has dimensionality [1 × *nsteps*].

evaluator.m

This subroutine uses a combination of the *edit_extract* function as well as user-supplied constraints (e.g., acceptable noise margin and jitter) to evaluate the output of an LTspice simulation. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 2: ***evaluator.m*** I/O summary

Inputs	**Description**
data_in	The output signal generated using LTspice, and extracted using *edit_extract*.
reqs	A cell array containing all user-inputted requirements for the output signal to be deemed valid.
Outputs	**Description**
td	The test decision (valued either "0" if invalid or "1" if valid) for evaluation of the output from an LTspice simulation.

state_levels.m

This subroutine exists as *statelevels* in the latest versions of MATLAB, but is now compatible with GNU Octave. It takes in the VCO output signal that is being evaluated and computes the average values of the high and low. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 3: ***state_levels.m*** I/O summary

Inputs	**Description**
signal	The transient VCO signal being evaluated by *evaluator*.
Outputs	**Description**
levels	A [1 × 2] array containing the [low, high] level values for the VCO output signal. The values are measured in volts (V).

plotter_grid.m

This function is called within *grid_gen* as an option to generate surface (as well as volumetric) plots of the metric vs. variable parameter landscape, allowing for a 3D visualization of the degree of convexity (or non-convexity) that must be faced when performing data-fitting and searching for

an optimal solution. Depending on the values of the input *switches*, all generated figures may be automatically saved to a subdirectory that is created. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 4: ***plotter_grid.m*** I/O summary

Inputs	Description
switches	User-set options for plotting/saving figures and saving data.
nsteps	The number of steps along each variable axis to generate the initial metric matrix.

database_gen.m

This function is called within the main script *LITE_MAIN* after the *grid_gen* function in order to generate an initial database matrix (if one does not already exist). It outputs a matrix in which each row is a KGD parameterized by a nominal frequency and the corresponding variable parameter values used to generate that frequency in LTspice. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 5: ***database_gen.m*** I/O summary

Inputs	Description
metric_grid	The output matrix has dimensionality *nsteps^nvars* for a single metric (here, f_0). Values can be either real numbers or NaN (if output has been deemed invalid by the *evaluator* function for a particular set of user-defined constraints).
var_arrays	The variable parameter arrays used for generating the metric matrix; each parameter array has dimensionality [1 × *nsteps*].
Outputs	**Description**
database	A [(*nsteps^nvars*) × (nvars +1)] matrix containing a set of KGD. Any new KGDs discovered from running the LITE optimization code are automatically appended. Future versions will feature a significantly more sophisticated structure for the database, beyond a simple matrix containing metrics and circuit parameter values.

pipeline.m

This main function is called within the main script *LITE_MAIN* once the user inputs have been stored in the workspace and a database file has either been found or otherwise generated. This function executes the pipeline flow as outlined in Figure [1]. If a solution is found, the database is appended, and a schematic is generated and opened in LTspice with the discovered KGD parameter values, along with a corresponding solution *Netlist*. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 6: ***pipeline.m*** I/O summary

Inputs	Description
switches	User-set options for plotting/saving figures and saving data.
metrics	A cell array containing all user-inputted desired values for circuit optimization, including metrics, tolerances, and bounds.

inputs	A cell array containing all user-inputted information needed for schematic editing and data extraction.

database_search.m

This function is called within the main function *pipeline.m* and as described in the name, searches the existing/generated database for candidate solutions, based on the metrics defined by the user. If any candidates are found in the database, they are automatically sorted in order of closest match to the target metrics and outputted as a matrix to either be used for further evaluation or as a starting point in the Nesterov search function. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 7: ***database_search.m*** I/O summary

Inputs	**Description**
metrics	A cell array containing all user-inputted desired values for circuit optimization, including metrics, tolerances, and bounds.
Outputs	**Description**
candidates	A sorted subset of KGDs extracted from the database file (*database.mat*).

edit_extract.m

This function is called throughout several points along the optimization flow, and is used in various other subroutines. This is essentially the interface code that allows for I/O between Octave and LTspice. As the name suggests, it edits schematic/netlist files, runs LTspice in batch mode, extracts the output at any/all user-specified nodes in the circuit, and in the case of transient analysis, computes the nominal signal frequency. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 8: ***edit_extract.m*** I/O summary

Inputs	**Description**
inputs	A cell array containing all user-inputted information needed for schematic editing and data extraction.
Outputs	**Description**
raw_data	An organized structure containing all the data extracted from the RAW file generated when running LTspice in batch mode, as well as the FFT results for schematic/netlist files containing the transient simulation directive (.tran).
iter	A schematic-generation iteration index.

nesterov_gradient_descent.m

We use Nesterov Gradient Descent to converge to our solution from a close local KGD. This function proceeds by first computing the Lipschitz constant in the neighborhood of the KGD so that the convexity of the local geometry can be estimated. Nesterov's gradient descent takes a *gamble → correction* approach, i.e.,

(1) Nesterov's method first makes a big jump in the direction of the previous accumulated gradient.
(2) It then measures the gradient at the new location and self-adjusts accordingly.

Once the algorithm has converged within the user-specified tolerance (typically, within a few percent of the desired metric(s)), it stops and returns the parameters for which the circuit is optimized to the *pipeline* function. It uses the *grad* function to compute the gradient. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 9: ***nesterov_gradient_descent.m*** I/O summary

Inputs	**Description**
inputs	Schematic file details of VCO model
specs	User-specified metrics (just the target nominal frequency for now)
in_param	Parameters from the closest KGD
param_bounds	Parameter bounds for estimation of the Lipschitz constant
d_param	Differential element for numerical differentiation
tol	Cost function tolerance level for convergence to a local minimum
nit	Maximum number of search iterations at an incorrect local minimum before breaking out of search loop
Outputs	**Description**
in_seed	Parameter array obtained from steepest descent search which yields user-specified metrics
UD	User decision as whether to use local minimum results when descent fails to converge to a minimum within the specified tolerance

grad.m

This subroutine is used exclusively in the *nesterov_gradient_descent* function, taking in the same inputs and calling the *edit_extract* function to execute LTspice and evaluate the gradient of the metric as a function of the variable parameters in the circuit. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 10: ***grad.m*** I/O summary

Inputs	**Description**
inputs	Schematic file details of VCO model
in_param	Last set of parameter values
f_in_param	The initial metric value associated with the parameters in *in_param*
f_s	User-specified metric (here, the target nominal frequency)
d_param	Array of increment sizes for variable parameters
Outputs	**Description**
grad_t	Local gradient in circuit parameter landscape

scpd_editor.m

This subroutine is essentially identical to the schematic-editing code found within *edit_extract*, and is used at the very end of *pipeline* to perform the final edits to the schematic by using the parameters associated with the discovered KGD to generate a KGD schematic, to be outputted by the *pipeline* code (generating the corresponding solution Netlist) at the end of the optimization flow. The function is summarized with the following descriptions of inputs and outputs in the table below.

Table 11: ***scpd_editor.m*** I/O summary

Inputs	**Description**
inputs	Schematic file details (containing the discovered KGD parameter values)
Outputs	**Description**
iter	A schematic-generation iteration index