Modelica supported automated design

Ion Matei[1] Maksym Zhenirovskyy[1] John Maxwell[1] Saman Mostafavi[1]

[1]Intelligent Systems Laboratory, SRI, United States, {ion.matei, maksym.zhenirovskyy, john.maxwell, saman.mostafavi}@sri.com

Abstract

We propose a component-based, automated, bottom-up method to system design, using models are expressed in the Modelica language. This bottom-up approach is based on a meta-topology that is iteratively refined via optimization. Each topology link is described by a universal component that is defined in terms of atomic components (e.g., resistors, capacitors for the electrical domain) or more complex canonical components with a well defined function (e.g., operational amplifier-based inverters). The activation of such links is done via discrete switches. To address the combinatorial explosion in the resulting mixed-integer optimization problems, we convert the discrete switches into continuous switches that are physically realizable and formulate a parameter optimization problem that learns the component and switch parameters. We encourage topology sparsity through an L_1 regularization term applied to the continuous switch parameters. We improve the time complexity of the optimization problem by reconstructing intermediate design models when components become redundant and by simplifying topologies through collapsing components and removing disconnected ones. To demonstrate the efficacy of our approach, we apply it to the design of various electrical circuits.

Keywords: component-based, design, optimization, nonlinear programming

1 Introduction

In this paper, we describe a general approach for designing physical systems using a bottom-up approach that implements the “change design” process in Figure 1. This type of problem can be formulated as a mixed integer program that includes a combinatorial part to select the component types and a continuous optimization part that selects parameters of components to meet requirements. A brute force approach to solving such an optimization problem suffers from combinatorial explosion, and heuristics based on branch-and-bound methods do not scale with the number of discrete optimization variables (Clausen 2003; Morrison et al. 2016). To limit the effects of combinatorial explosion, we introduce an algorithm that transforms the mixed-integer formulation into a nonlinear program, with physically realizable solutions.

To facilitate the description of the algorithm and of the results, we focus on design problems in the electrical domain. However, the approach can be generalized to other physical domains. We use the Modelica language to describe the basic components and the generated design solutions, which allows subject matter experts to interpret and evaluate the generated designs.

The design models use a universal component that embeds the behavior of basic components in the electrical domain (e.g., resistor, inductor, capacitor, short connection, and open connection) or more complex components based on operational amplifiers (OpAmps) in various configurations. For example, a universal component based on inverting and non-inverting OpAmp configurations is shown in Figure 2. Each branch of the component is activated or deactivated by a switch that controls the current that flows through it. The design problem is to find the correct switch assignments and component parameter values to meet the requirements, which can be specified in terms of the time evolution of certain quantities of interest or the characteristics of a transfer function in the case of filter design. We start with a topology that describes how the universal components are connected and includes points for setting boundary conditions (e.g., voltage/current sources) and taking measurements. The design problem is then formulated as an optimization problem that minimizes a loss function $\mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p},\boldsymbol{s}),\boldsymbol{y}_{0:T})$, where $\boldsymbol{p}$ and $\boldsymbol{s}$ are the parameters and switches of the basic components, respectively, $\boldsymbol{y}_{0:T}$ is a target vector of measurements over time interval $[0,T]$, $\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p},\boldsymbol{s})$ is the model's predicted measurements, and $\mathscr{C}$ is a metric that measures the error between the model predictions and the target measurements (e.g., mean square error). The optimization problem also takes into account dynamic constraints, and bounds on component parameters (e.g., resistances must be non-negative).The main contributions of this paper are as follows:

- *Continuous relaxation with lossless realization:* We developed an optimization algorithm that relaxes the integer constraints on the switches by treating them as continuous variables in the range [0, 1]. The parameters of the components and their associated switches are optimized using gradient-free search algorithms and simulations based on Functional Mockup Units (FMUs) (Blochwitz et al. 2011). To encourage sparsity in the design solution, we also add an L_1 regularization term to the loss function. The non-zero switches are not approximated by 0 or 1, but are realized as electric resistors, ensuring no loss in optimality but a possible

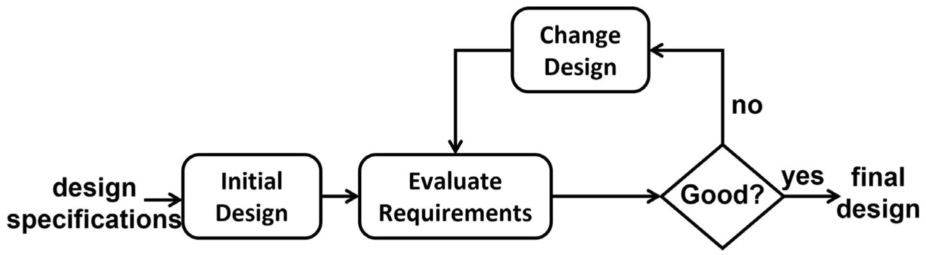


Figure 1. The design optimization process: an initial design is continuously refined until requirements are satisfied.

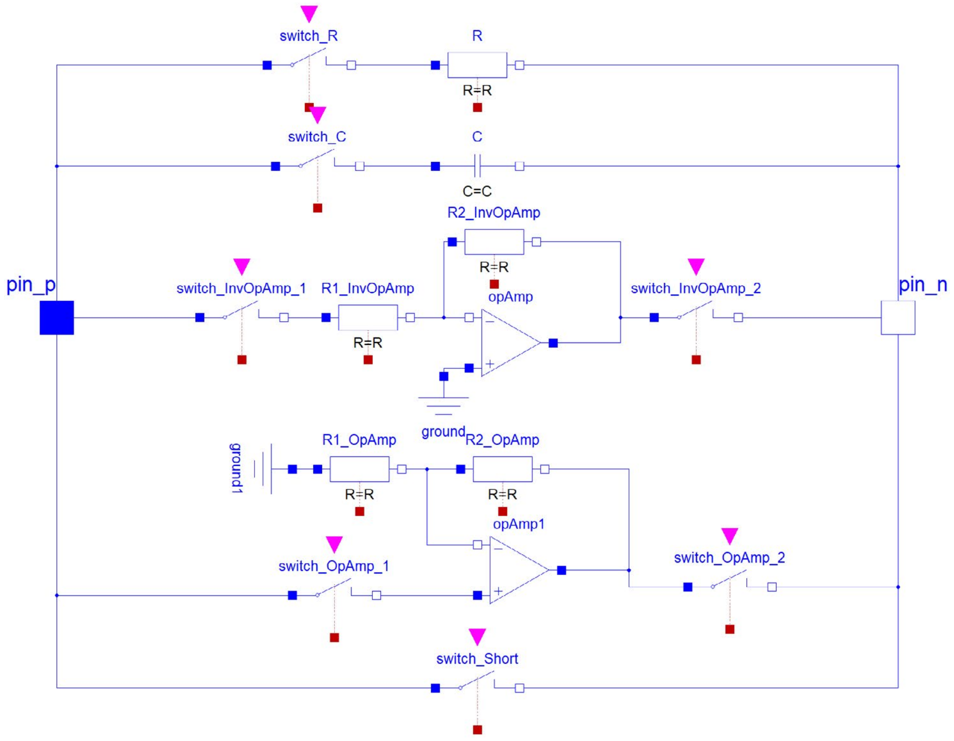


Figure 2. Universal component based on inverting and non-inverting OpAmp configurations.

loss in sparsity. Since we cannot guarantee finding the global optimum, we also use parallel optimization runs with random initial conditions to generate a diverse set of design solutions.

- *Scalability improvement via model simplifications:* During optimization, when certain components are no longer needed (i.e., their switches are set to zero), we eliminate them and reconstruct the design model. This reduces the complexity of the model, as measured by the number of equations, and leads to faster simulation times. In addition, we developed a graph theory-based algorithm that further simplifies the designs generated by the optimization procedure. The algorithm removes unnecessary components, combines compatible components in series and parallel connections into equivalent single components, and annotates the resulting design models for visual representation and simulation in tools that support the Modelica language.

Paper structure: In Section 2, we present an algorithm for automated design that uses continuous relaxation. In Section 3, we discuss how we improve the efficiency of our design algorithm by reducing the complexity of the intermediate design models that are simulated during the design space exploration. Finally, in Section 4, we present the designs generated by the proposed algorithm for various circuit design problems and types of universal components.

2 Design optimization

When using branch-and-bound heuristics to solve mixed integer programs, we may encounter situations where the cost of the relaxed problem is better than the cost obtained by converting the optimization variables into integer values. In this section, we present a method to avoid such a case. The key idea is to interpret the switches in a way that allows for their physical implementation, even when they do not have integer values. In the universal component definition, each branch has a corresponding switch that opens or closes a connection. When the switch is open, no current flows through the component, leading to its exclusion from the design model. In the Modelica electrical library, the switch (`Modelica.Electrical.Analog.Ideal.IdealOpeningSwitch`) is modeled such that when the switch is open, there is a high resistance that blocks the flow of current. When the switch is closed, there is a very low resistance and the current flows freely. This switch is controlled by a boolean input, the value of which determines the switch mode. We draw inspiration from the definition of the Modelica switch to create a continuous switch that is controlled by a parameter that takes values in the range [0, 1]. The switch is defined by the equations

$$v = a((\varepsilon - 1)s + 1), \quad (1)$$
$$i = a((1-\varepsilon)s + \varepsilon), \quad (2)$$

where v is the switch voltage, i is the current through the switch, a is an auxiliary variable, $s \in [0,1]$ is the switch control, and ε is a small hyper-parameter that determines the residual resistance when the switch is closed. The switch equation can be simplified to

$$v = \frac{(\varepsilon - 1)s + 1}{(1-\varepsilon)s + \varepsilon} i,$$

showing that for $s = 0$ we have $v = i/\varepsilon$ and for $s = 1$ we have $v = \varepsilon i$, the expected behavior of a switch. We do not use this simplified representation of the switch for numerical stability reasons. The introduction of the auxiliary variable a prevents the presence of equations with terms that involve divisions by very small numbers. However, the disadvantage is that the resulting system of equations for the design model becomes a differential algebraic equation (DAE) rather than an ordinary differential equation (ODE). This limitation restricts the type of optimization approach that can be used, as we cannot directly utilize platforms that support automatic differentiation (AD) (e.g., the `torchdiffeq` package in Pytorch). In addition to the requirements loss function $\mathscr{C}$, we introduce a sparsity-promoting L_1 regularization term, resulting in the total optimization loss:

$$\mathscr{L}(\boldsymbol{p}, \boldsymbol{s}) = \mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{y}_{0:T}) + \lambda \|\boldsymbol{s}\|_1,$$

where $0 \leq s_i \leq 1$, with $\boldsymbol{s} = (s_i)$, and λ is a positive weight that controls the sparsity strength. If in the optimization solution not all entries of $\boldsymbol{s}$ are zero or one, we map them into electric resistors with equivalent resistances, $\frac{(\varepsilon-1)s_i+1}{(1-\varepsilon)s_i+\varepsilon}$. Thus, we can physically realize them, without affecting the optimal cost function, i.e., the design requirements.

The pseudocode for this algorithm is shown in Algorithm 1. We use a gradual approach to achieve sparsity. We start with a small λ value to make sure that we generate an initial design that satisfies the requirements. Then we gradually increase λ until the requirements cost function is no longer improved. Ideally, for each λ, we would like to obtain the optimal solution. The strategy for updating λ is reminiscent to a primal-dual approach (Bertsekas 1999), where we minimize $\mathscr{C}$ under an L_1 sparsity constraint.

In our approach, we incrementally increase the value of λ until it begins to negatively impact the requirements cost function. At this point, we halt the process and perform a final optimization without the L_1 regularization term. The result of this final optimization will be our design solution. Box constraints are commonly used in our problem setup, but we use variable transformations to eliminate them and use an unconstrained optimization algorithm to minimize $\mathscr{L}$. For example, we can eliminate the constraint $a \leq x \leq b$ by using the transformation $x = a + (\sin(\tilde{x}) + 1)(b-a)/2$, where $\tilde{x}$ is the new unconstrained optimization variable. It is not guaranteed that the optimization will converge to the global minimum, as the cost function's nonlinear dependence on the optimization parameters means we cannot accurately predict the structure of the problem. Ideally, we would find at least a local minimum for each λ value, but it is possible that the optimization algorithm may take too many iterations to converge. As a result, we set a limit on the number of iterations allowed between λ updates for practical reasons.

All optimization algorithms will require the evaluation of the design model. We use a black-box approach to optimization, where the cost evaluation is done by querying a computational model of the design: an FMU (Blochwitz et al. 2011). In the cosimulation version of the FMU, such a representation contains the algorithm used for simulating the model (e.g., CVODE solver (Hindmarsh et al. 2005)), in addition to the design description. FMUs can be integrated in several languages (e.g., Python, C, Java) and computational platforms (e.g., Matlab/Simulink, OpenModelica, Dymola). The optimization algorithms were implemented in Python based on the `Scipy` optimization package. We used a gradient free (i.e., a direct method) optimization algorithm that relies only on the objective function, namely Powell's method (Powell 1964). Empirically, it provides a better convergence rate than other gradient-free algorithms such as Nelder-Mead, and is faster than global, gradient-free optimization algorithms (e.g., genetic algorithms). The integration of FMUs into the optimization algorithms was done using the `PyFMI` library (Andersson, Åkesson, and Führer 2016).

3 Model Construction and Simplification

We automatically construct a Modelica model for a domain given a universal component and a specification of the initial topology. For instance, if the user wanted to use a 5x6 grid, then the program would generate a Modelica

Algorithm 1 Continuous relaxation design algorithm

```
Require: $\delta$: solution tolerance
Require: $\lambda$: $L_1$ loss weight
Require: $\Delta$: $L_1$ loss weight increase rate
Require: FMU of the initial design model
Require: $\boldsymbol{p}$, $\boldsymbol{s}$: initial parameter and switch values
Require: $\boldsymbol{y}_{0:T}$: target measurements
 1: $\mathscr{C}_{prev} = \infty$
 2: while True do
 3:     $\boldsymbol{p}, \boldsymbol{s} \leftarrow \arg\min_{\boldsymbol{p},\boldsymbol{s}} \mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{y}_{0:T}) + \lambda \|\boldsymbol{s}\|_1$
 4:     $\mathscr{C}^* = \mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{y}_{0:T})$
 5:     if $\mathscr{C}^* \leq \mathscr{C}_{prev}$ then
 6:         $\lambda \leftarrow \Delta\lambda$
 7:         $\mathscr{C}_{prev} = \mathscr{C}^*$
 8:         eliminate components corresponding to zero
            switches and reconstruct the model
 9:     else
10:         $\boldsymbol{p}, \boldsymbol{s} \leftarrow \arg\min_{\boldsymbol{p},\boldsymbol{s}} \mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p}, \boldsymbol{s}), \boldsymbol{y}_{0:T})$
11:         return $\boldsymbol{p}, \boldsymbol{s}$
12:     end if
13: end while
```

model with 30 grid points with components connecting pairs of points vertically and horizontally (see Figure 3). This model is embedded in another model which specifies the components that set the boundary conditions, i.e., the voltage source and the resistor load (see Figure 4).

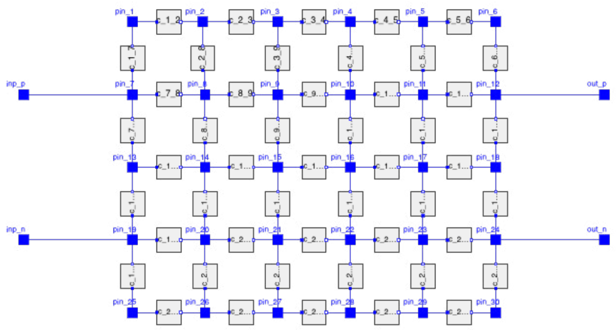


Figure 3. Modelica model for the grid. Universal components connect the grid points.

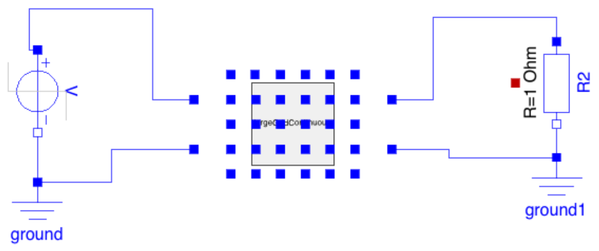
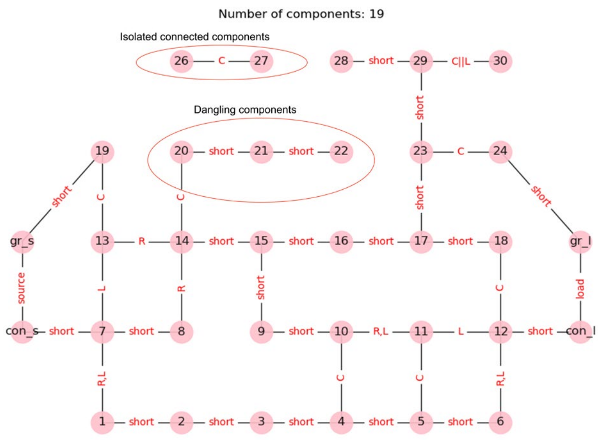


Figure 4. Modelica model for the scenario that gives the boundary conditions of a grid.

In the continuous relaxation approach to optimization, each universal component has switches that allow internal components to be enabled or disabled. These switches can be set from the top level model. When a component is disabled, then the Modelica compiler ignores it when constructing an FMU, thus no equations pertaining to the respective components are added. This process is implemented by conditionally declaring the basic components of the universal component. Consequently, a basic component appears in the instance of a universal component only when a corresponding flag is set to true. The flags of the basic components in all instances of the universal component are continuously updated during the optimization process.

After the optimizer has found a solution (i.e., has determined which components should be enabled and what their parameter values should be), we produce another Modelica model that flattens the universal components and just shows the internal components. At this point we perform two simplification operations: eliminate isolated components and dangling components. These operations are necessary to deal with the cases where switch, resistor or capacitor values are close to zero. Such a situation indicates the presence of open connections. Figure 5 shows a design solution example based a universal component that uses passive components only, and that contains isolated (capacitor between vertices 26 and 27) and dangling (components between vertices 14, 20, 21, 22) components. The design solution can contain isolated com-



Number of components: 19
Isolated connected components
Dangling components
26 — C — 27
28 — short — 29 — C||L — 30
20 — short — 21 — short — 22
23 — C — 24
gr_s
13 — R — 14 — short — 15 — short — 16 — short — 17 — short — 18
gr_l
con_s — short — 7 — short — 8
9 — short — 10 — R,L — 11 — L — 12 — short — con_l
1 — short — 2 — short — 3 — short — 4 — short — 5 — short — 6
short
C
R
L
R,L
source
load

Figure 5. Graph representation of a design solution: vertices are connection points and edges components.

ponents since switches are not exactly zero, meaning that there may be some very small residual currents passing through components. Thus, it may appear that we have components that are isolated but in fact only a small, negligible current passes through them. The isolated components are eliminated by first generating the largest set of connected components that include the boundary conditions (i.e., the voltage source and the resistor load), and discarding the remaining ones. The design solution may also contain components that appear to be dangling, i.e., they are connected at one end only. The reason for such a phenomenon is the same as in the isolated components case: residual currents passing through them. The dangling components are found by looking at the cycles of the design. If a component does not belong to a cycle then it must be dangling, thus it is eliminated. We implemented code that generates a visually interpretable layout for the components based on the and-or graph also of the components that are between two grid points. The layout was achieved by annotating the flattened Modelica design model with Modelica notation that generates the visual effects. Finally, we have code to simplify the model by merging compatible serial or parallel components. The code goes through this process iteratively, until no merging can be achieved. The resulting model has correct equivalent parameter values (i.e., resistances in serial connections are added) and it can be simulated using Modelica.

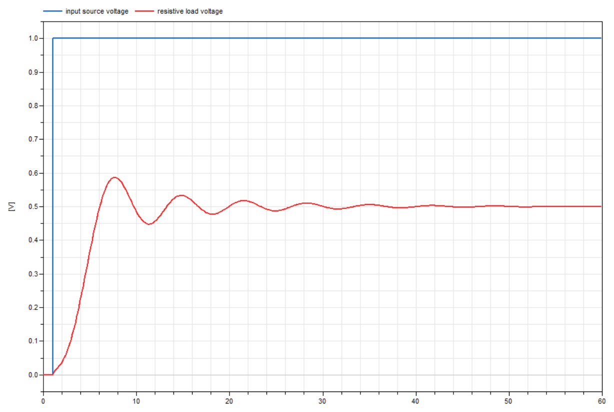


Figure 7. Cauer low pass analog filter: input source voltage vs. resistive load voltage.

4 Results

In this section we present design results based on Algorithm 1 for various design examples.

4.1 Cauer analog low pass filter with passive components

Our goal is to design a filter whose output from a step response matches the output of the Cauer analog low pass filter of the fifth order (see Figure 6). The input voltage versus the load voltage plot is shown in Figure 7. To

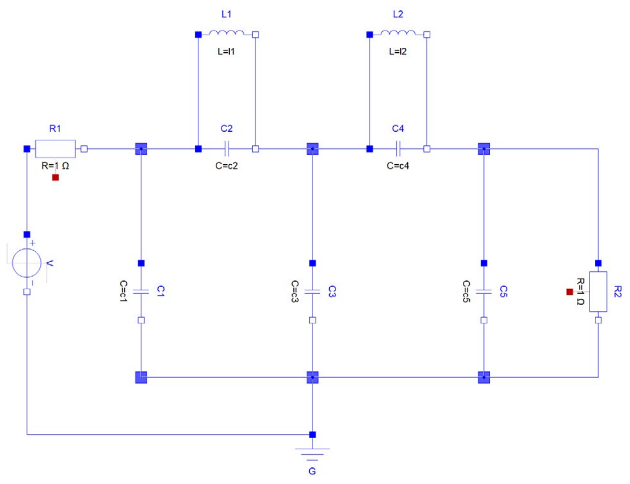


Figure 6. Modelica model of the Cauer analog, low pass filter of the fifth order.

improve the likelihood to find a design solution, we start with a dense initial topology expressed as a 5x6 grid, with a universal component based on passive electrical components. The number of optimization variables corresponding to this initial topology is 343, including component parameters and switch values. The dense initial topology is likely to ensure the existence of several local minima that are close to satisfy the design requirements. To explore multiple of such local minima, we leverage parallel executions of design optimization processes, where each process starts with random initial component parameters, and initial weight for the L_1 cost, and where all switches are initialized to 0.5. We run 20 parallel processes that explore various design solutions. The design optimization algorithm was implemented in Python, and the evaluation of the design loss function was done via FMU-based simulations using the `fmypi` Python package. We refer to each optimization corresponding to an instance of the L_1 loss weight as *outer iteration*. An outer iteration was implemented using the gradient free Powell algorithm, where we limit the execution of the algorithm to 150 (inner) iterations. The limited number of iteration affects only the early outer iterations, since 150 iterations may not be sufficient to converge to a local minima. However, since we use a sequence of outer iterations, where each such outer iteration uses the previous optimization variables as initial values, in practice we do converge to a design that satisfies requirements. More importantly, each outer iteration reduces the time complexity since, after each outer iteration we eliminate redundant components whose switch values are approximately zero. The number of variables drops from 343 at the first iteration to values in the twenties or smaller, at the last iteration. Remarkably, after the first iteration that uses no L_1 regularization term, all processes eliminate more than 250 optimization variables as a result of switches being set to zero. The time per iteration is determined by three factors: the number of iteration of the Powell algorithm, the number of optimization variables and the FMU simulation time. Not unexpectedly, the most expensive outer iteration is the first one, that corresponds to 343 optimization variables. As the design models become simpler, the outer iteration times reduce to tens of seconds. An example of a design solution that realizes the behavior of the Cauer analog filter implemented using passive components is shown in Figure 8.

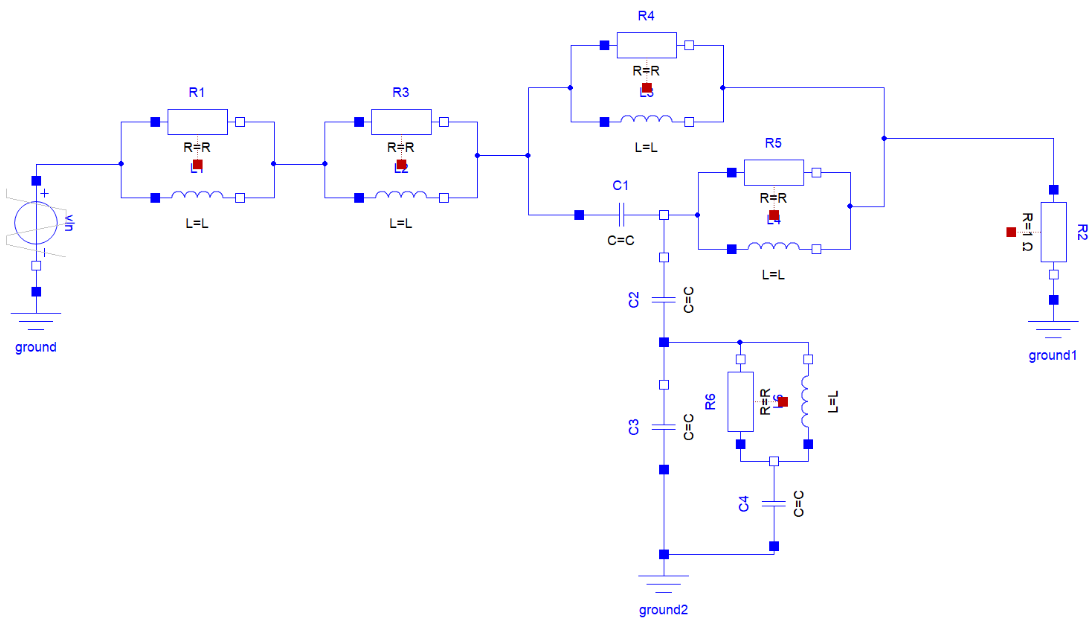


Figure 8. Design solution for the Cauer analog low pass filter based on passive components generated by Algorithm 1.

4.2 Voltage level shifter design with operational amplifiers

We present the results of designing a voltage level shifter (see Figure 9), using Algorithm 1. The universal component employed to generate the initial grid topology consists of a resistor, capacitor, and operational amplifier arranged in a non-inverting configuration, together with open and short connections. We run 10 parallel executions of Algorithm 1 for 150 outer iterations, with a limit of 300 inner iterations for the Powell algorithm in each outer iteration.

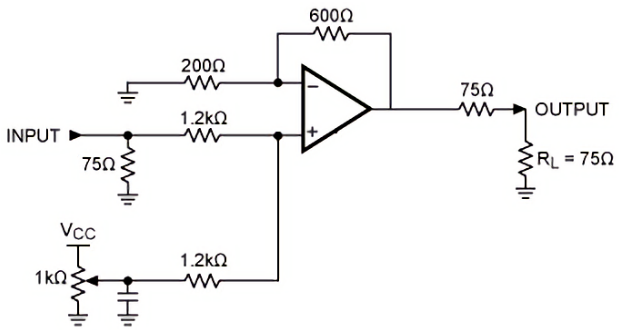


Figure 9. Voltage level shifter circuit used to generate the ground truth data in the form of the voltage across the load resistor (R_L).

Two examples of design solutions produced by Algorithm 1 for the voltage level shifter are depicted in Figures 10 and 11. Notably, both solutions have a component count that is similar to that of the original level shifter depicted in Figure 9, with 10 and 9 components for the two solutions compared to 8 components in the original circuit (not counting the load resistor and the voltage source components). Additionally, both solutions utilize a single OpAmp.

4.3 Cauer analog low pass filter with active filters

We repeated the design optimization problem for the Cauer low pass filter, where the branches of the universal component include first and second-order low and high-pass filters, implemented using operational amplifiers, together with resistor, capacitor, short and open connection components. We started with a 2x6 grid as the initial topology and ran 10 parallel executions of Algorithm 1 for 250 outer iterations, with a limit of 1000 inner iterations for the Powell algorithm in each iteration. After a final simplification, we chose one of the solutions and arrived at a circuit shown in Figure 12 that includes 8 operational amplifiers. The Modelica Standard Library (MSL) has an implementation of the Cauer analog filter that uses only 5 operational amplifiers but also includes 4 negative resistors, where each negative resistor can be implemented using an operational amplifier. Our design solution therefore has a similar number of operational amplifiers as the one in the MSL.

Table 1 summarizes the design results of the above examples in comparison with the original circuits that were used to generate the ground truth. When counting the number of resistors and OpAmps in the MSL active implementation of the Cauer filter, we included the number of resistors and OpAmps needed to implement the negative resistors. The loss function used in the optimization algorithm focuses on behavior and complexity (via the L1 regularization term). The loss function can be augmented with additional objectives that can include component costs, for example. The computational time depends

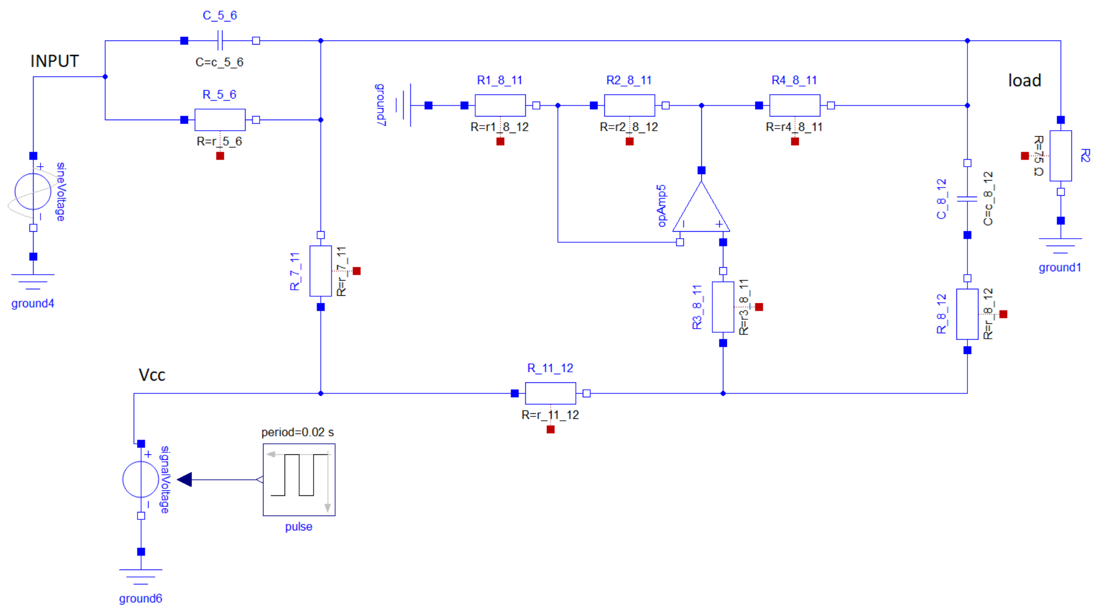


Figure 10. Design of the voltage level shifter with operational amplifiers using Algorithm 1: design solution 1.

on the number of iterations of the optimization algorithms and the FMU simulation time. The latter can be decreased or increased by manipulating the number of collocation points or the solver tolerances. Depending of the complexity of the initial models and the weight of the L1 regularization term, the optimization algorithms can take from tens of minutes to several hours.

5 Differential programming for gradient-based optimization

The algorithm introduced in the previous sections uses gradient-free optimization to search for the component parameters. The advantage of such algorithms is that they work directly with computational representations of the design model (i.e., FMUs). The disadvantage is that they become slower as the number of optimization variables increases. An alternative to gradient-free algorithms is gradient-based algorithms, and the optimization problem would translate into a nonlinear program with dynamical constraints. Solving such a problem would requires having access to the gradients of the objective and constraint functions. When dealing with design models represented as ODEs, we can map the design optimization problem into a framework that supports automatic differentiation (AD) (e.g., Pytorch (Paszke et al. 2017) or Jax (Bradbury et al. 2018)), and solve the problem using gradient descent algorithms. Such platforms are endowed with ODE solvers that support AD (Chen et al. 2018). To formulate the problem in frameworks such as Pytorch or Jax, we first need to extract the equations from the Modelica model of the design. One approach is to generate an XML representation for the DAE using the `dumpXMLDAE` function of the OpenModelica (Fritzson et al. 2010; Open Source Modelica Consortium n.d.) scripting language. Alternatively, we can process the flattened Modelica using a Python Modelica parser such as `modparc` (DongPing 2013). Similar equation extraction can be done using commercial Modelica tools such as `Dymola`, or `SystemModeler`. The extracted equations are converted into symbolic objects such as `Sympy` (Meurer et al. 2017) objects, and mapped into deep-learning platform objects that support automatic differentiation. This process leads to a constrained optimization problem that in the case of the continuous relaxation approach is given by:

$$\min_{\boldsymbol{x},\boldsymbol{p},\boldsymbol{s}} \quad \mathscr{C}(\hat{\boldsymbol{y}}_{0:T}(\boldsymbol{p},\boldsymbol{s}),\boldsymbol{y}_{0:T})+\lambda\|\boldsymbol{s}\|_1 \tag{3}$$

$$\text{subject to:} \quad \dot{\boldsymbol{x}} = f(\boldsymbol{x},\boldsymbol{z};\boldsymbol{p},\boldsymbol{s}), \tag{4}$$

$$g(\boldsymbol{x},\boldsymbol{z};\boldsymbol{p},\boldsymbol{s}) = 0, \tag{5}$$

$$\hat{\boldsymbol{y}} = h(\boldsymbol{x},\boldsymbol{z};\boldsymbol{p},\boldsymbol{s}), \tag{6}$$

where (4)-(5) is the DAE in the semi-explicit form, representing the dynamics of the design model, and $h(\boldsymbol{x},\boldsymbol{z};\boldsymbol{p},\boldsymbol{s})$ is the sensing model.

To solve (3), we can convert (4) into a set of equality constraints using direct collocation methods (Hargraves and Paris 1987; Herman and Conway 1996), or we can use local (e.g., Chebyshev polynomial expansions (Boyd 2001)) or global (e.g., neural networks) parameterizations of the state solution and solve for the representation parameters (e.g., weights and biases of the neural network). For example, if we use neural networks to represent the state $\boldsymbol{x}(t) = NN_x(t;\beta_x)$ and the algebraic variables $\boldsymbol{z}(t) = NN_z(t;\beta_z)$, the optimization problem (3) will be solved in terms of the parameters β_x, β_z, $\boldsymbol{p}$, $\boldsymbol{s}$. In addition, automatic differentiation can be used to evaluate the time derivative of the state. Our attempts to use a differentiable programming paradigm to solve design problems were met with mixed results. In the case where the model is represented

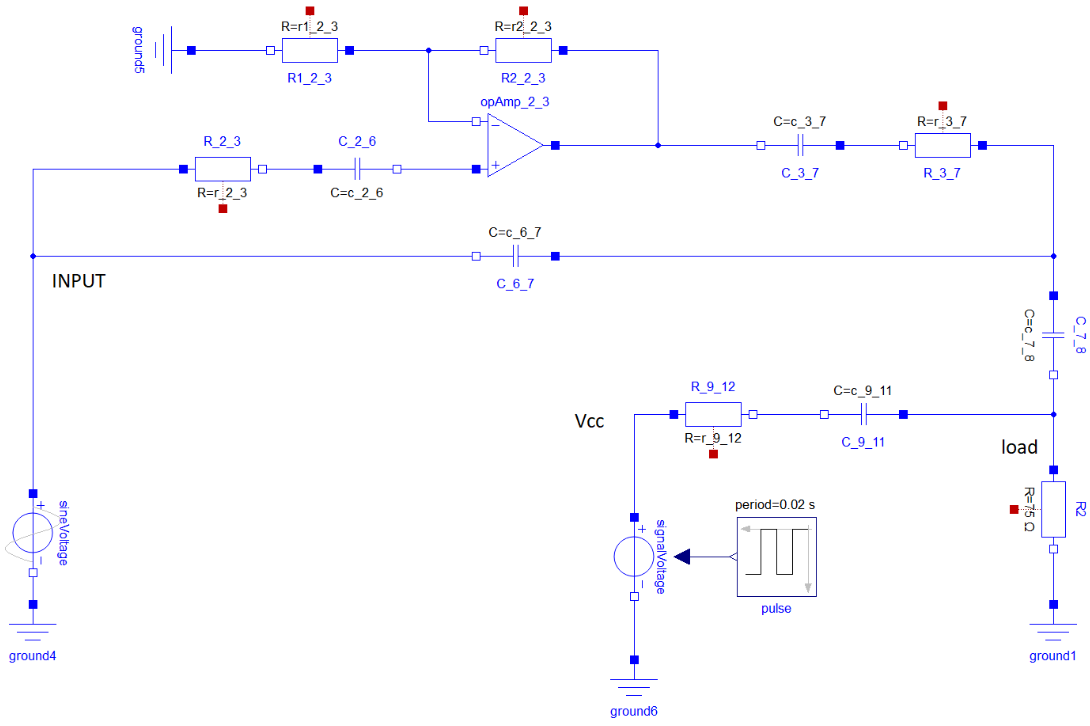


Figure 11. Design of the voltage level shifter with operational amplifiers using Algorithm 1: design solution 2.

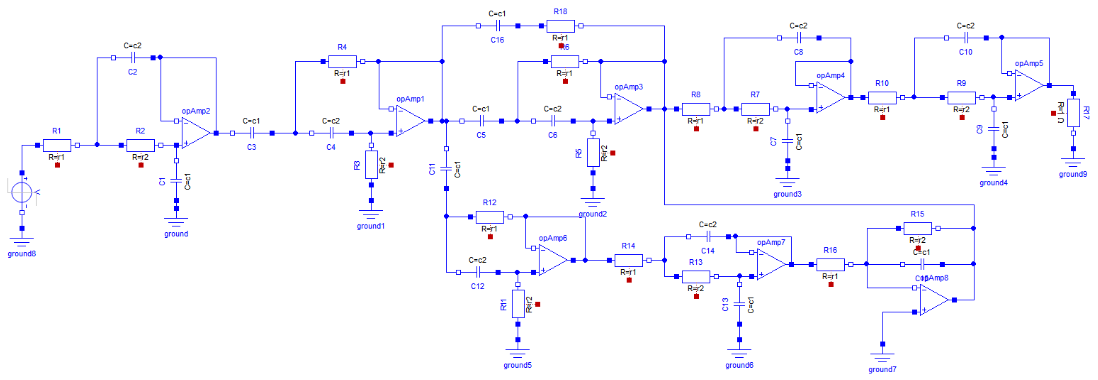


Figure 12. Design solution for the Cauer analog low pass filter with operational amplifiers using Algorithm 1.

as an ODE, we obtained good results. For example, in (Ion Matei et al. 2020) we showed how to learn control policies for an inverted pendulum using a model predictive control approach solved using Pytorch. When dealing with DAEs though, the gradient-based optimization algorithm, when combined with direct collocation methods to approximate time derivatives, tend to converge slowly. In addition, the parameterized DAE solution does not always check against the DAE simulation executed with the optimal component and switch parameters. Unfortunately, we cannot always guarantee that the design model can be represented as an ODE, especially since the model is repeatedly reconstructed and simplified. Thus, the results shown in this paper use a direct method (i.e., Powell algorithm), instead a gradient-based approach. Ideally, we would like to have a sensitivity analysis method embedded in the DAE solvers, so that we can access the Jacobian of the state with respect to the model parameters. Such a method is present for instance in the SUNDIALS software family, introduced in (Gardner et al. 2022; Hindmarsh et al. 2005), with DAE solvers such as CVODES and IDAS that include both direct and adjoint-based approaches to compute sensitivities. Currently though, deep learning platform do not offer such a functionality, except for the case where the DAE can be transformed into an ODE. Moreover, even when dealing with ODE, gradient-descent algorithm that include solvers supporting automatic differentiation tend to slow down as the number of optimization parameters increases. We addressed this challenge in (I. Matei et al. 2023), where we showed that block coordinate descent algorithm in combination with direct collocation method speed up training by several order of magnitude.

Circuit	Number of resistors	Number of capacitors	Number of inductors	Number of OpAmps
Original passive Cauer filter	1	5	2	0
Designed passive Cauer filter	5	3	5	0
Original active Cauer filter	19	8	0	9
Designed active Cauer filter	17	16	0	8
Original voltage level shifter	7	1	0	1
Designed voltage level shifter (sol. 1)	8	2	0	1
Designed voltage level shifter (sol. 2)	5	4	0	1

Table 1. Summary of the design results for various examples.

We are currently working on extending this approach to DAE models. There are Julia libraries that can also be used for a gradient-based approach. For example, ModelingToolkit.jl and its component library, ModelingToolkitStandardLibrary, are modeling languages for symbolic-numeric computation (Ma et al. 2021). They combine symbolic computational algebra systems ideas with causal and acausal equation-based modeling frameworks. We did not use this library in our work because it lacks many components from the Modelica Standard Library, thus requiring a model-transformation component for mapping Modelica models into Julia representations. White the differential programming paradigm is an appealing avenue for dealing with numerical complexity, we cannot always guarantee that the model we use are smooth. It is possible for such models to be hybrid (i.e., include discrete and continues variables) and thus not differentiable.

6 Conclusions

In this paper, we presented an automated design process utilizing a bottom-up approach. The process begins with an initial possibly large topology of universal components that is iteratively refined until a sparse solution is found. The initial design is based on universal components, each of which can exhibit a range of behaviors through basic components. This combination of modes and topology ensures a broad coverage of the design space. We demonstrated an approach for addressing the combinatorial explosion typical of design optimization problems. The approach relaxes discrete variables to continuous variables by transforming discrete switches into continuous switches. These continuous components are physically realizable, resulting in no loss in performance. Additionally, sparsity is induced through an L_1 regularization cost that encourages the parameters of the continuous switches to be zero. The proposed approach is supported by automated model simplification and reconstruction that reduce the complexity of the design model, in turn decreasing the time complexity for the continuous optimization algorithms that require model simulations. The continuous optimization algorithms are gradient-free. We are currently investigating the application of a differential programming paradigm to the design problem described in this paper, which would allow us to utilize gradient-based algorithms. The major challenge we face is extending automatic differentiation support to DAEs that typically require stiff, implicit numerical solvers, while avoiding the need for implementing model-transformation modules to convert Modelica models to new representations.

References

Andersson, C., J. Åkesson, and C. Führer (2016). *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*. Technical Report in Mathematical Sciences 2. Centre for Mathematical Sciences, Lund University.

Bertsekas, D.P. (1999). *Nonlinear Programming*. Athena Scientific.

Blochwitz, T. et al. (2011). "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". In: *In Proceedings of the 8th International Modelica Conference*.

Boyd, John P. (2001). *Chebyshev and Fourier Spectral Methods*. Second. Dover Books on Mathematics. Mineola, NY: Dover Publications. ISBN: 0486411834 9780486411835.

Bradbury, James et al. (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. URL: http://github.com/google/jax.

Chen, Ricky T. Q. et al. (2018). "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc.

Clausen, Jens (2003). "Branch and Bound Algorithms-Principles and Examples". In.

DongPing, X. (2013). *ModParc*. https://github.com/xie-dongping/modparc.

Fritzson, Peter et al. (2010-02). *OpenModelica System Documentation*. URL: https://github.com/OpenModelica/OpenModelica-doc/blob/v1.9.1/OpenModelicaSystem.pdf.

Gardner, David J. et al. (2022). "Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)*.

Hargraves, C. R. and S. W. Paris (1987). "Direct trajectory optimization using nonlinear programming and collocation". In: *Journal of Guidance, Control, and Dynamics* 10.4, pp. 338–342.

Herman, Albert L. and Bruce A. Conway (1996). "Direct optimization using collocation based on high-order Gauss-Lobatto quadrature rules". In: *Journal of Guidance, Control, and Dynamics* 19.3, pp. 592–599.

Hindmarsh, Alan C et al. (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Matei, I. et al. (2023). "Sensitivity-Free Gradient Descent Algorithms". In: *Journal of Machine Learning Research* 24.300, pp. 1–26. URL: http://jmlr.org/papers/v24/22-1002.html.

Matei, Ion et al. (2020). "Deep Learning for Control: a non-Reinforcement Learning View". In: *2020 American Control Conference (ACC)*, pp. 2942–2948. DOI: 10 . 23919 / ACC45564.2020.9147287.

Meurer, Aaron et al. (2017-01). "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3, e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

Morrison, David R. et al. (2016). "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discrete Optimization* 19, pp. 79–102.

Open Source Modelica Consortium (n.d.). *OpenModelica User's Guide*. URL: https : / / openmodelica . org / doc / OpenModelicaUsersGuide/latest/.

Paszke, Adam et al. (2017). "Automatic differentiation in PyTorch". In.

Powell, M. J. D. (1964-01). "An efficient method for finding the minimum of a function of several variables without calculating derivatives". In: *The Computer Journal* 7.2, pp. 155–162.