A k-d Tree Based Solution Cache for the Non-linear Equation of Circuit Simulations

Martin Holters and Udo Zölzer
Department of Signal Processing and Communications
Helmut Schmidt University – University of the Federal Armed Forces
Hamburg, Germany
Email: {martin.holters,udo.zoelzer}@hsu-hh.de

***Abstract*—In the digital simulation of non-linear audio effect circuits, the arising non-linear equation generally poses the main challenge for a computationally cheap implementation. For any but the simplest circuits, using an iterative solver at execution time will be too slow, while exhaustive look-up tables quickly grow intolerably large. To better cope with the situation, in this paper we propose to store solutions non-uniformly sampled from the parameter space to enable an iterative solver to quickly converge when being started from the closest initial solution. Efficient look-up of this closest solution is realized by using a k-d tree. The method is supported by a step to reduce the dimension of the parameter space and a linear extrapolation from the closest solution stored to the actually needed parameter vector.**

I. Introduction

Digital simulation of analog audio processing circuits is an ongoing research topic. Various methods exist to derive a mathematical model for an analog circuit in a systematic way, most notably wave digital filters [1], [2], port-Hamiltonian approaches [3], and state-space based approaches [4], [5]. Unavoidably, if the circuit to model contains non-linear elements like diodes, transistors, vacuum tubes, or saturating transformers, the resulting model will include a non-linear equation that in general will not have a closed-form solution. Without special measures, solving these non-linear equations at run-time will typically be prohibitively slow for non-trivial circuits. In this paper, we propose a novel scheme to store pre-computed solutions in a k-d tree instead of commonly used linearly indexed look-up tables.

Due to their generality, we consider non-linear equations of the form

$$\boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{D}_\mathrm{q}\bar{\boldsymbol{x}} + \boldsymbol{E}_\mathrm{q}\bar{\boldsymbol{u}} + \boldsymbol{F}_\mathrm{q}\boldsymbol{z}) = \boldsymbol{0} \tag{1}$$

from [5]. Here, $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{u}}$ denote the circuit's states (e.g. capacitor charges) and source values, respectively, after appropriate time discretization, while $\boldsymbol{z}$ is the vector of unknowns to be solved for. Other forms of non-linear equations, e.g. as derived in [4], can be easily rewritten to this form. It should be noted that $\boldsymbol{F}_\mathrm{q}$ is tall, i.e., it has more rows than columns, hence it is not sufficient to find a single permissible $\boldsymbol{q}^*$ such that $\boldsymbol{f}(\boldsymbol{q}^*) = \boldsymbol{0}$ and then solve the linear system $\boldsymbol{F}_\mathrm{q}\boldsymbol{z} = \boldsymbol{q}^* - \boldsymbol{D}_\mathrm{q}\bar{\boldsymbol{x}} - \boldsymbol{E}_\mathrm{q}\bar{\boldsymbol{u}}$ for new values of $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{u}}$, as it will in general not have a solution. Instead, for every change of $\bar{\boldsymbol{x}}$ or $\bar{\boldsymbol{u}}$, i.e. for every time step, the non-linear system has to be solved.

Assuming a good initial estimate $\boldsymbol{z}^{(0)}$ of the solution, Newton iteration or variants thereof are an effective strategy for solving (1). Besides using the same $\boldsymbol{z}^{(0)}$ for every time step, a common strategy is to use the solution found for the previous time step, assuming only small inter-sample differences to occur in $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{u}}$. In [6], a more elaborate scheme is proposed where (1) is simplified such that an explicit solution can be found analytically which is then used to compute $\boldsymbol{z}^{(0)}$.

However, none of these approaches guarantee to deliver a $\boldsymbol{z}^{(0)}$ sufficiently close to the correct solution, and therefore the Newton solver may need a large number of iterations, or worse, not converge at all. To help convergence, different approaches like damped Newton iteration [7], homotopy [4], or heuristical modification of the Newton step [6] have been successfully employed. However, the total number of required iterations may still be prohibitively large, especially for more complex circuits.

An alternative is to use look-up tables [4], [8] which provide constant run-time. Unfortunately, for non-trivial circuits, these tables need to be indexed by vectors, either $\begin{pmatrix}\bar{\boldsymbol{x}}^T & \bar{\boldsymbol{u}}^T\end{pmatrix}^T$ or $\boldsymbol{D}_\mathrm{q}\bar{\boldsymbol{x}} + \boldsymbol{E}_\mathrm{q}\bar{\boldsymbol{u}}$, resulting in memory requirements growing exponentially with the dimension of the index vector. In [8], the memory requirements were successfully reduced by non-uniformly sampling the parameter space. However, with a table based solution, the values to include in the table need to be chosen for each dimension independently. Therefore, in this work, we propose an alternative where the solutions are stored in a tree-like structure, allowing arbitrarily distributed values to be stored and efficiently looked up.

II. Dimension Reduction

Before discussing the tree-based approach, we will reconsider (1) to try to reduce the dimension of the parameter vector. First, it makes sense to decompose $\bar{\boldsymbol{u}}$ into constant entries $\boldsymbol{u}_0$ and time-varying entries $\tilde{\boldsymbol{u}}$ and decompose $\boldsymbol{E}_\mathrm{q}$ accordingly such that $\boldsymbol{E}_\mathrm{q}\bar{\boldsymbol{u}} = \boldsymbol{E}_{\mathrm{q},0}\boldsymbol{u}_0 + \tilde{\boldsymbol{E}}_\mathrm{q}\tilde{\boldsymbol{u}}$. The straight-forward choice of the parameter vector then would be $\boldsymbol{D}_\mathrm{q}\bar{\boldsymbol{x}} + \tilde{\boldsymbol{E}}_\mathrm{q}\tilde{\boldsymbol{u}}$. Apparently, the result is confined to the sub-space spanned by the columns of $\begin{pmatrix}\boldsymbol{D}_\mathrm{q} & \tilde{\boldsymbol{E}}_\mathrm{q}\end{pmatrix}$. The approach in [5] leaves some freedom in the computation of $\boldsymbol{D}_\mathrm{q}$ and $\tilde{\boldsymbol{E}}_\mathrm{q}$, hence they can be chosen so as to minimize the dimension of this sub-space; in particular, they can always be chosen to be orthogonal to $\boldsymbol{F}_\mathrm{q}$.

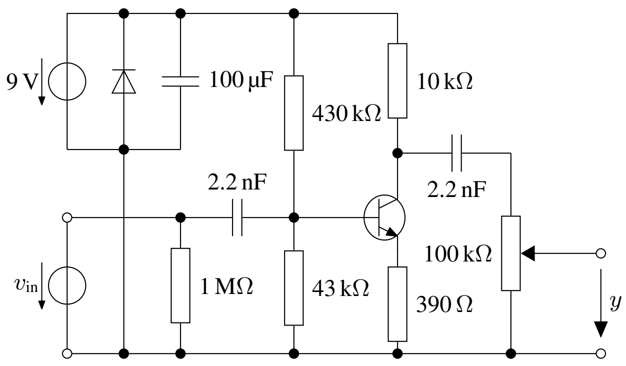


Fig. 1. Exemplary treble booster circuit.

Now, we can factorize $\begin{pmatrix}\boldsymbol{D}_{\text{q}} & \tilde{\boldsymbol{E}}_{\text{q}}\end{pmatrix} = \boldsymbol{Q}\cdot\begin{pmatrix}\hat{\boldsymbol{D}}_{\text{q}} & \hat{\boldsymbol{E}}_{\text{q}}\end{pmatrix}$ such that $\begin{pmatrix}\hat{\boldsymbol{D}}_{\text{q}} & \hat{\boldsymbol{E}}_{\text{q}}\end{pmatrix}$ has full row-rank (using e.g. QR decomposition). Hence, we can rewrite (1) as

$$\boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{q}_0 + \boldsymbol{Q}\boldsymbol{p} + \boldsymbol{F}_{\text{q}}\boldsymbol{z}) = \boldsymbol{0} \tag{2}$$

with

$$\boldsymbol{q}_0 = \boldsymbol{E}_{\text{q},0}\boldsymbol{u}_0 \tag{3}$$

and the parameter vector

$$\boldsymbol{p} = \hat{\boldsymbol{D}}_{\text{q}}\bar{\boldsymbol{x}} + \hat{\boldsymbol{E}}_{\text{q}}\tilde{\boldsymbol{u}} \tag{4}$$

of minimal dimension.

As an example, we shall consider the circuit of Fig. 1. Note that the schematics include a diode anti-parallel to the 9 V supply voltage (to protect the device when connected to a supply voltage of wrong polarity) as well as a stabilizing 100 µF capacitor, both of which are found in typical circuits but could be eliminated from the model without changing its input/output behavior if the supply voltage source is modeled as ideal. However, we include them here to study how well the proposed method handles them.

Invoking the machinery of [5] for a sampling rate of 44.1 kHz and pulling out constant elements from $\boldsymbol{E}_{\text{q}}\bar{\boldsymbol{u}}$ as discussed above, we obtain

$$\boldsymbol{q}_0 = \begin{pmatrix} -9 \\ 0 \\ 3.22\times10^{-7} \\ -1.704\times10^{-7} \\ 8.162\times10^{-4} \\ -8.651\times10^{-4} \end{pmatrix} \quad \tilde{\boldsymbol{E}}_{\text{q}} = \begin{pmatrix} 0 \\ 0 \\ 4.945\times10^{-8} \\ -1.592\times10^{-8} \\ 1.72\times10^{-4} \\ 7.344\times10^{-6} \end{pmatrix} \tag{5}$$

$$\boldsymbol{D}_{\text{q}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -2.248\times10^{1} & 1.521 \\ 0 & 7.234 & -8.1\times10^{-1} \\ 0 & -7.817\times10^{4} & 3.831\times10^{3} \\ 0 & -3.338\times10^{3} & -4.159\times10^{3} \end{pmatrix} \tag{6}$$

$$\boldsymbol{F}_{\text{q}} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -2.917\times10^{-4} & 9.705\times10^{-5} \\ 0 & 9.705\times10^{-5} & -1.054\times10^{-4} \end{pmatrix} \tag{7}$$

and

$$\boldsymbol{f}(\boldsymbol{q}) = \begin{pmatrix} I_{\text{sD}}\cdot(e^{\frac{q_1}{\eta_{\text{D}}v_{\text{T}}}}-1) - q_2 \\ I_{\text{sE}}\cdot(e^{\frac{q_3}{\eta_{\text{E}}v_{\text{T}}}}-1) - \frac{\beta_{\text{r}}}{1+\beta_{\text{r}}} I_{\text{sC}}\cdot(e^{\frac{q_4}{\eta_{\text{C}}v_{\text{T}}}}-1) - q_5 \\ -\frac{\beta_{\text{f}}}{1+\beta_{\text{f}}} I_{\text{sE}}\cdot(e^{\frac{q_3}{\eta_{\text{E}}v_{\text{T}}}}-1) + I_{\text{sC}}\cdot(e^{\frac{q_4}{\eta_{\text{C}}v_{\text{T}}}}-1) - q_6 \end{pmatrix}, \tag{8}$$

where $\begin{pmatrix} q_1 & \cdots & q_6 \end{pmatrix} = \boldsymbol{q}^T$. The nonlinear equation in (8) consists of the Shockley equation for the diode in the first row and the Ebers–Moll equations for the transistor in the second and third row, where v_{T} is the thermal voltage (usually 25 mV), η and I_{s} denote the emission coefficients and the saturation currents, respectively, and β_{f} and β_r are the forward and reverse current gain. The entries in the vector $\boldsymbol{q}$ are, in that order, the diode voltage and current, the base–emitter and base–collector voltage and the emitter and collector current. From the matrices in (5)–(6), it is obvious that diode voltage and current are independent of the capacitor states and input voltage; instead, the voltage is fixed at −9 V and the current needs to be solved for (which is trivial in this case). Likewise, it is apparent that the state of the capacitor parallel to the supply voltage, corresponding to the first entry in $\bar{\boldsymbol{x}}$ and hence the first column of $\boldsymbol{D}_{\text{q}}$, has no influence on the nonlinear equation.

At first glance, it now looks like the nonlinear equation depends on a three-dimensional parameter vector corresponding to the input voltage and the states of the two 2.2 nF capacitors. However, by applying the QR-decomposition to $\begin{pmatrix}\boldsymbol{D}_{\text{q}} & \tilde{\boldsymbol{E}}_{\text{q}}\end{pmatrix}$ as discussed above, we find

$$\boldsymbol{Q} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 2.873\times10^{-4} & -1.094\times10^{-4} \\ -9.247\times10^{-5} & 1.094\times10^{-4} \\ 9.991\times10^{-1} & -4.267\times10^{-2} \\ 4.267\times10^{-2} & 9.991\times10^{-1} \end{pmatrix} \tag{9}$$

$$\hat{\boldsymbol{D}}_{\text{q}} = \begin{pmatrix} 0 & -7.824\times10^{4} & 3.65\times10^{3} \\ 0 & 0 & -4.319\times10^{3} \end{pmatrix} \tag{10}$$

$$\hat{\boldsymbol{E}}_{\text{q}} = \begin{pmatrix} 1.721\times10^{-4} \\ 0 \end{pmatrix} \tag{11}$$

so that, in fact, a two-dimensional parameter vector $\boldsymbol{p} = \hat{\boldsymbol{D}}_{\text{q}}\bar{\boldsymbol{x}} + \hat{\boldsymbol{E}}_{\text{q}}\tilde{\boldsymbol{u}}$ suffices.

III. k-d Trees

Points in a k-dimensional space can be effectively organized in a k-d tree, which is similar to an ordinary (binary) search tree, but branches on different dimensions in different layers of the tree to partition the space [9]. A toy example is shown in Fig. 2, where a tree for four points in two dimensions is shown. At the root, the space is partitioned in the first dimension at $x_1 = 0.59$. In the second level, the space is partitioned in the second dimension, with boundary values depending on the branch taken at the root.

When searching for a point contained in the tree, the correct point is found by traversing the tree from the root to the respective leaf once. However, we are also interested in finding the nearest neighbor to a point not contained in the tree. This is

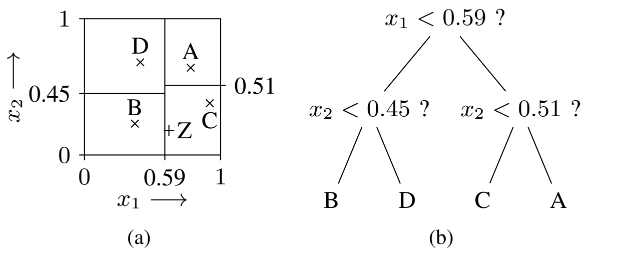


Fig. 2. Example of a k-d tree for four points in $k = 2$ dimensions. (a) Points A–D and induced space partitioning and point Z to search for. (b) Binary tree.

a harder problem as the first answer obtained from traversing the tree may not be the best one. Consider the point marked Z in Fig. 2a. The nearest neighbor we would like to obtain is point B, but the tree search yields point C. An efficient strategy to finding the nearest neighbor is *best bin first* [10], where for every branch taken, the alternative along with the distance of the boundary value to the query point is recorded. After the first candidate has been determined (C in the example), the alternative with the lowest distance is tried next, providing new alternatives itself. In the example, the boundary at $x_1 = 0.59$ is closest to the query point Z, taking the left branch on the second try yields point B.

Additionally, the recorded distances to the boundary provide a stopping criterion: Once the best point found so far is closer to the query point than the boundary of the best untried alternative, no closer match can be found. The distances have to be recorded for every dimension individually, resulting in the following algorithm:

```
**Require:** $k$-d tree $T$, query point $p$
  $A \leftarrow \{\}$
  $a \leftarrow (\mathrm{Root}(T), \mathbf{0})$
  $\boldsymbol{p}^* \leftarrow$ any point from $T$
  **while** $\|a[2]\| < \|\boldsymbol{p} - \boldsymbol{p}^*\|$ **do**
    $n \leftarrow a[1]$
    **while not** $\mathrm{IsLeaf}(n)$ **do**
      $d \leftarrow \mathrm{Dim}(n)$
      $v \leftarrow \mathrm{Val}(n)$
      $\boldsymbol{\Delta} \leftarrow a[2]$
      $\Delta_d \leftarrow p_d - v$
      **if** $p_d \leq v$ **then**
        $n \leftarrow \mathrm{LeftChild}(n)$
        $A \leftarrow A \cup \{(\mathrm{RightChild}(n), \boldsymbol{\Delta})\}$
      **else**
        $n \leftarrow \mathrm{RightChild}(n)$
        $A \leftarrow A \cup \{(\mathrm{LeftChild}(n), \boldsymbol{\Delta})\}$
      **end if**
    **end while**
    **if** $\|\boldsymbol{p} - \mathrm{Point}(n)\| < \|\boldsymbol{p} - \boldsymbol{p}^*\|$ **then**
      $\boldsymbol{p}^* \leftarrow \mathrm{Point}(n)$
    **end if**
    $a \leftarrow \mathrm{PopBestAlternative}(A)$
  **end while**
  **return** $\boldsymbol{p}^*$
```

Here A keeps the alternatives yet to consider as pairs a, where the first component $a[1]$ is the node in the tree from where to restart the search and the second component $a[2]$ holds a vector $\boldsymbol{\Delta}$ of distances. In particular, for every "wrong" branch taken, the distance of the decision boundary to the query point $\boldsymbol{p}$ is recorded for the respective dimension. The operation $\mathrm{PopBestAlternative}(A)$ removes the entry a with the lowest $\|a[2]\|$ from A and returns it. Hence, A is best implemented as a heap-based priority queue. Further obvious optimizations include not adding to A alternatives where Δ is already too large and pruning from A every time $\boldsymbol{p}^*$ is updated.

Oftentimes, the assumption of small inter-sample differences is valid, so the solution from the previous time-step should also be taken into consideration. So, instead of initializing $\boldsymbol{p}^*$ with any point from the cache as in the algorithm above, it makes sense to use the parameter vector $\boldsymbol{p}$ from the previous time-step. This may serve to both limit the tree search and provide a solution that is even closer to the current $\boldsymbol{p}$ than the best one cached.

IV. Extrapolation

Once a $\boldsymbol{p}^*$ close to $\boldsymbol{p}$ along with the corresponding solution $\boldsymbol{z}^*$ has been retrieved, instead of using $\boldsymbol{z}^*$ directly, we can extrapolate by linearizing $\boldsymbol{f}(\boldsymbol{q})$ as

$$\begin{aligned}\boldsymbol{f}\big(\boldsymbol{q}_0 + \boldsymbol{Q}(\boldsymbol{p}^* + \boldsymbol{\Delta p}) + \boldsymbol{F}_{\mathrm{q}}(\boldsymbol{z}^* + \boldsymbol{\Delta z})\big) \approx \\ \underbrace{\boldsymbol{f}\big(\boldsymbol{q}_0 + \boldsymbol{Q}\boldsymbol{p}^* + \boldsymbol{F}_{\mathrm{q}}\boldsymbol{z}^*\big)}_{=\mathbf{0}} + \boldsymbol{J}\boldsymbol{Q}\boldsymbol{\Delta p} + \boldsymbol{J}\boldsymbol{F}_{\mathrm{q}}\boldsymbol{\Delta z}\end{aligned} \quad (12)$$

where $\boldsymbol{J}$ denotes the Jacobian of $\boldsymbol{f}$ at $\boldsymbol{q}_0 + \boldsymbol{Q}\boldsymbol{p}^* + \boldsymbol{F}_{\mathrm{q}}\boldsymbol{z}^*$. Letting $\boldsymbol{\Delta p} = \boldsymbol{p} - \boldsymbol{p}^*$ and $\boldsymbol{\Delta z} = \boldsymbol{z}^{(0)} - \boldsymbol{z}^*$ and setting the right-hand-side of (12) to zero, we obtain

$$\boldsymbol{z}^{(0)} = \boldsymbol{z}^* - (\boldsymbol{J}\boldsymbol{F}_{\mathrm{q}})^{-1}\boldsymbol{J}\boldsymbol{Q}(\boldsymbol{p} - \boldsymbol{p}^*). \quad (13)$$

Note that as $\boldsymbol{J}$ is rectangular, its two occurrences do not cancel. The $\boldsymbol{z}^{(0)}$ thus obtained can now be used as the initial solution for a Newton solver.

V. Populating the Cache

Now that we have an efficient means of retrieving pre-computed solutions and extrapolating them to the current parameter vector, the question remains how to choose which solutions to store. Unfortunately, finding the optimal set of solutions to store remains an open issue. We propose here a simple heuristic where the solution cache is gradually build up while trying to solve the non-linear equation for many possible values of $\boldsymbol{p}$. For every $\boldsymbol{p}$ we run the following steps:

1) Retrieve closest solution $\boldsymbol{p}^*$ from cache (or the previous time-step) as described in section III.
2) Extrapolate to current $\boldsymbol{p}$ as described in section IV.
3) Try to solve (1) using Newton's method with the initial solution $\boldsymbol{z}^{(0)}$ thus determined.
4) a) If Newton's method converged in no more than a predefined number of iterations $N_{\max}$, proceed to next $\boldsymbol{p}$.

b) If Newton's method required more than $N_{\max}$ to converge, add solution to cache and proceed to next $\boldsymbol{p}$.

c) If Newton's method did not converge, use a homotopy approach to find solutions on the line from $\boldsymbol{p}^*$ to $\boldsymbol{p}$ by applying steps 2–4 at the intermediate points, gradually moving towards $\boldsymbol{p}$ while potentially adding additional points to the cache. Add the solution finally found to the cache and proceed to next $\boldsymbol{p}$.

For simple circuits, where the dimensionality of $\boldsymbol{p}$ is low enough and bounds on its possible values can be derived, the cache can be filled is a systematic fashion. To do so, the volume containing the possible values of $\boldsymbol{p}$ is divided into a fine grid and the above steps executed at each grid point. To avoid asymmetries or otherwise skewed distributions of the cached solutions, randomizing the order in which the solutions are tried is recommended. Note that sometimes new cached solutions actually increase the number of needed iterations at near-by points—the closest $\boldsymbol{p}^*$ does not necessarily yield the best $\boldsymbol{z}^{(0)}$—so the whole procedure should be repeated until no new solutions need to be cached. Assuming a sufficiently fine grid, Newton's method now should never require more than $N_{\max}$ iterations when running the model.

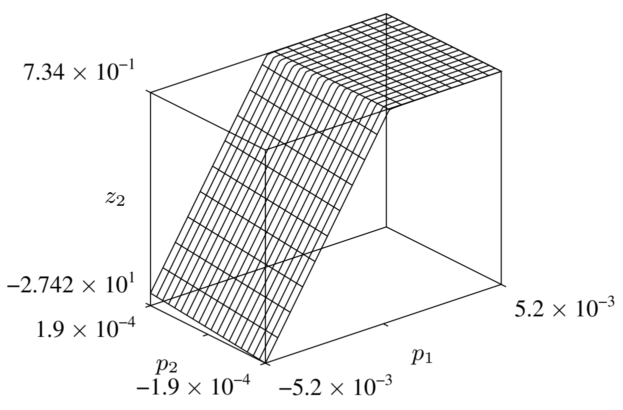
For more complex circuits, using the gridded approach quickly becomes infeasible due to the exponential growth of the number of points to consider when the dimensionality of $\boldsymbol{p}$ increases. In fact, just determining the range of values the parameter vector $\boldsymbol{p}$ can assume for a bounded input $\bar{\boldsymbol{u}}$ is a surprisingly tough problem. Therefore, we employ a heuristic where solutions are stored while the model is executed on actual data, gradually building the solutions cache. After a sufficiently long training phase with varying input stimuli, Newton's method should never require more than $N_{\max}$ iterations. However, how best to choose the input signals and how to determine when the training can safely by stopped are still open research questions.

VI. Experimental Results

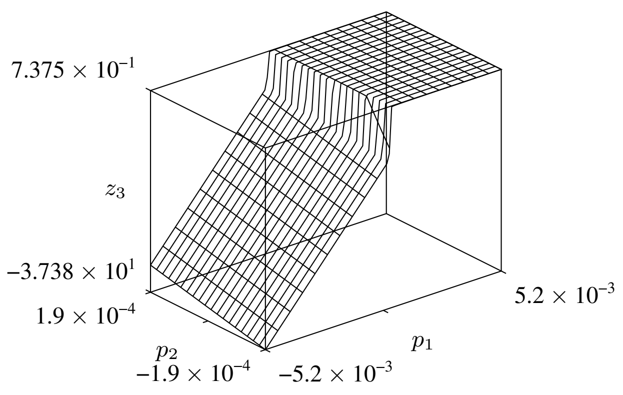
To evaluate the proposed method, it has been implemented as part of the *ACME.jl*[1] project, a package for circuit simulation in the Julia programming language. The following examples again use a sampling rate of 44.1 kHz throughout.

We first consider the simple circuit of Fig. 1. If we limit the input to $\pm 10\,\mathrm{V}$, we can safely assume the capacitor voltages to stay within $\pm 19\,\mathrm{V}$, which after generously rounding leads to $-5.2\times10^{-3} \le p_1 \le 5.2\times10^{-3}$ and $-1.9\times10^{-4} \le p_2 \le 1.9\times10^{-4}$. While z_1 (the diode current) is constant at −350 pA, the values of the solution components z_2 and z_3 in this range are shown in Fig. 3. We observe linear behavior for negative values of p_1, a transitional region around $p_1 = 0$ slightly shifting with the value of p_2, and saturation effects occurring for positive p_1.

[1]Available at https://github.com/HSU-ANT/ACME.jl.



(a)



(b)

Fig. 3. Components (a) z_2 and (b) z_3 of the solution of (1) for the circuit of Fig. 1.

For this simple circuit, the gridded cache filling strategy can be employed. In particular, we subdivide the range of p_1 into 10 000 points and the range of p_2 into 400 points. We then solve the nonlinear equation at all 4 000 000 points in randomized order using the procedure of section V with $N_{\max} = 5$ to fill the cache. Depending on the randomized order, typically two–three repetitions are required until no more new solutions have to be cached, making the training take a couple of minutes on the author's Xeon E5-1620 PC.

The cached solutions thus determined are marked in Fig. 4. We clearly see the high number of required points in the transitional region of the solution, while no solutions need to be cached for the linear part thanks to the extrapolation mechanism. In the mildly curved saturation region, only few extra points are required. In total, the cache holds 388 points, with the exact number and position again varying with the randomized order during cache population. As can also be seen in Fig. 4, this is sufficient to even keep the required number of iterations well below the maximum allowed $N_{\max} = 5$ over wide areas.

As a second example, we consider a circuit of moderate complexity, namely *Der SuperOver*[2], a clone of the *Boss*

[2]Schematics are available at http://diy.musikding.de/wp-content/uploads/2013/06/superoverschalt.pdf.

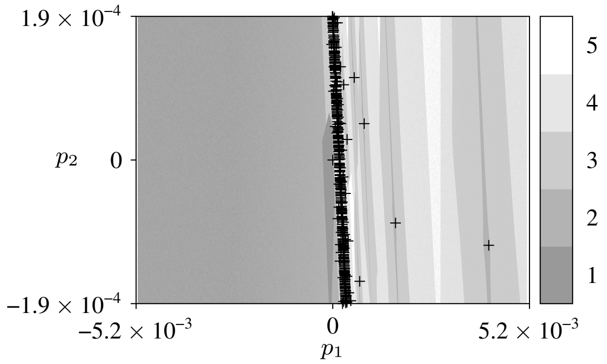


Fig. 4. Number of iterations needed to solve (1) for the circuit of Fig. 1 using the closest of the cached solution marked with + as initial solution $\boldsymbol{z}^{(0)}$.

SD-1 pedal. We model the operational amplifiers as ideal, but include the diode anti-parallel to the supply voltage as for the treble booster. This gives a total of 4 diodes and 2 transistors to model, so that $\boldsymbol{f}(\boldsymbol{q})$ is a system of 8 equations in 16 variables. The number of entries in the state vector $\bar{\boldsymbol{x}}$, corresponding to the number of capacitors in the circuit, is 11. The dimension reduction step of section II leads to a 7-dimensional parameter space.

A systematic pre-computation step on a fixed grid is no longer feasible for 7 dimensions, so a training phase with varying input stimuli is employed instead. In particular, the following stimuli have been applied:

- a linear sine-sweep from 0 Hz to 22.05 kHz and back to 0 Hz with a total duration of 20 s, repeated with amplitudes varying from 10 mV to 1 V in 10 mV steps
- white noise with a total duration of 120 s, linearly increasing in amplitude from 0 V to 1 V
- two short (clean) electric guitar tracks with a total duration of 143 s

Due to the larger amount of data to process and the increased model complexity, the training in this case takes almost an hour, filling the solution cache with 27 039 solutions.

The model with the trained solution cache is then fed with another guitar track of 35 s length. Unfortunately, the training does not suffice to guarantee convergence with $N_{\text{max}} = 5$ iterations for the whole track. For 2172 samples (corresponding to 0.14 % of the track), more iterations are needed and the found solutions subsequently added to the cache. Only for 88 samples, more than 15 iterations are needed, however, in the worst case, 500 iterations are required. Nevertheless, on average, convergence is achieved after only 2.3081 iterations.

We compare to using only the solution from the previous time step as $\boldsymbol{p}^*$ and $\boldsymbol{z}^*$ instead of other cached solutions. By employing the same extrapolation scheme of section IV to obtain the initial solution $\boldsymbol{z}^{(0)}$ and the same homotopy Newton solver, the average number of iterations only slightly increases to 2.5862, proving the efficacy of the extrapolation scheme. However, the number of samples which now require more than 15 iterations increases significantly to 4357.

VII. Conclusion and Outlook

The proposed method allows efficient storage and retrieval of solutions non-uniformly distributed in a multi-dimensional parameter space. Using these stored solutions to initialize a Newton solver can greatly reduce the required number of iterations to converge. The most important open question is how to choose the stored solutions so as to guarantee a certain maximum number of iterations will always suffice to achieve convergence.

It should be noted that the proposed method is not meant to replace, but rather complement other approaches to speed up solving the non-linear equation. Especially the decomposition in (almost) independent sub-circuits [11] is still very attractive. Not only does it speed up the individual Newton iterations, it usually also helps reducing the iterations needed. For the proposed method, it would also mean replacing a single k-d tree of high dimensionality with multiple trees of lower dimensionality. Furthermore, the total number of solutions stored could be reduced: Assuming for simplicity the first circuit component depends solely on p_1 and requires N_1 solutions stored and the second component depends solely on p_2 and requires N_2 solutions stored, a combined cache would have to store all $N_1 \cdot N_2$ combinations of p_1 and p_2, instead of just $N_1 + N_2$ for two individual caches. Therefore, automating such a decomposition is a highly interesting future task.

References

[1] G. De Sanctis and A. Sarti, "Virtual analog modeling in the wave-digital domain," *IEEE Audio, Speech, Language Process.*, vol. 18, no. 4, pp. 715–727, May 2010.

[2] A. Bernardini, K. J. Werner, A. Sarti, and J. O. Smith, "Modeling a class of multi-port nonlinearities in wave digital structures," in *23rd European Signal Process. Conf. (EUSIPCO)*, Nice, France, 2015, pp. 664–668.

[3] A. Falaize-Skrzek and T. Hélie, "Simulation of an analog circuit of a wah pedal: a port-Hamiltonian approach," in *135th Audio Eng. Soc. Conv.*, New York, 2013.

[4] D. T. Yeh, J. S. Abel, and J. O. Smith, "Automated physical modeling of nonlinear audio circuits for real-time audio effects – part I: Theoretical development," *IEEE Audio, Speech, Language Process.*, vol. 18, no. 4, pp. 728–737, May 2010.

[5] M. Holters and U. Zölzer, "A generalized method for the derivation of non-linear state-space models from circuit schematics," in *23rd European Signal Process. Conf. (EUSIPCO)*, Nice, France, 2015, pp. 1078–1082.

[6] B. Holmes and M. van Walstijn, "Improving the robustness of the iterative solver in state-space modelling of guitar distortion circuitry," in *Proc. of the 18th Int. Conf. on Digital Audio Effects (DAFx-15)*, Trondheim, Norway, 2015.

[7] K. Dempwolf and U. Zölzer, "Discrete state-space model of the fuzz-face," in *Proc. of Forum Acusticum*, Aalborg, Denmark, 2011, pp. 455–460.

[8] J. Mačák, M. Holters, and J. Schimmel, "Simulation of fender type guitar preamp using approximation and state-space model," in *Proc. of the 15th Int. Conf. on Digital Audio Effects DAFx-12*, York, UK, 2012, pp. 209–216.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[10] J. Beis and D. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Proc. of IEEE Comput. Soc. Conf. on Comput. Vision and Pattern Recognition*, 1997, pp. 1000–1006.

[11] J. Mačák and J. Schimmel, "Real-time guitar tube amplifier simulation using an approximation of differential equations," in *Proc of the 13th Int. Conf. on Digital Audio Effects DAFx-10*, Graz, Austria, 2010.