RT-WDF—A MODULAR WAVE DIGITAL FILTER LIBRARY WITH SUPPORT FOR ARBITRARY TOPOLOGIES AND MULTIPLE NONLINEARITIES

Maximilian Rest [†*], *W. Ross Dunkel* [*], *Kurt James Werner* [*], *Julius O. Smith* [*]

[*]Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, California, USA
[†]Fakultät Elektrotechnik und Informatik, Technische Universität Berlin, Berlin, Germany
m.rest@e-rm.de, [chigi22,kwerner,jos]@ccrma.stanford.edu

ABSTRACT

Wave Digital Filters (WDF) [1] are a popular approach for virtual analog modeling [2]. They provide a computationally efficient way to simulate lumped physical systems with well-studied numerical properties. Recent work by Werner et al. [3, 4] enables the use of WDFs to model systems with complicated topologies and multiple/multiport nonlinearities, to a degree not previously known.

We present an efficient, portable, modular, and open-source *C++* library for real time Wave Digital Filter modeling: *RT-WDF* [5]. The library allows a WDF to be specified in an object-oriented tree with the same structure as a WDF tree and implements the most recent advances in the field. We give an architectural overview and introduce the main concepts of operation on three separate case studies: a switchable attenuator, the Bassman tone stack, and a common-cathode triode amplifier. It is further shown how to expand the existent set of non-linear models to encourage custom extensions.

Index Terms— wave digital filter, software, real time, virtual analog modeling, multiple nonlinearities

1. INTRODUCTION

There are numerous methods for virtual analog modeling of analog audio circuits on a digital system. While some of them operate in the Kirchhoff i–v domain with (non)-linear state space models [6, 7], the framework presented in this paper operates in the *wave* domain.

Though historically developed for the design of digital implementations of analog ladder/lattice filters, Wave Digital Filters (WDF) [1] have in recent years become a popular approach to virtual analog circuit modeling [2]. WDFs benefit from well-studied numerical properties and stability conditions. They have been used to successfully model lumped systems, including mechanical systems, electromechanical systems, and especially electronic circuits.

Among other benefits, they are attractive to algorithm developers due to their modularity, and desirable numerical behavior [1]. The efficiency of WDFs make real time simulation a possibility.

In this paper, we present the modular Real Time Wave Digital Filter *C++* software library: *RT-WDF* [5]. This library allows for more computationally efficient WDF simulation than existing frameworks and, most importantly, incorporates the field's recent theoretical advances. The goal of this paper is not to exhaustively document every feature of the library, but to introduce its main principles of operation and demonstrate its application to representative circuits. Full documentation accompanies the sourcecode (see Section 8).

The rest of the paper is structured as follows: Section 2 reviews recent theoretical advances and existing circuit simulation software packages. Section 3 gives an overview of the *RT-WDF* library. Section 4 details the use of the library to simulate representative circuits: a switchable attenuator, the Fender Bassman tone stack, and a common cathode triode amplifier. Section 5 compares the performance of the *RT-WDF* library with *SPICE* [8] and *Matlab* [9], Section 6 concludes and presents an outlook on future work.

2. PREVIOUS WORK

2.1. Recent Theoretical Advances

Recent work by Werner et al. [4] vastly expanded the class of circuits that could be systematically modeled with WDF to include those with complex topologies as well as multiport linear elements [3]. This approach has been successfully applied to model op-amps at various degrees of complexity [10] and has recently been extended to accommodate multiple non-adaptable linear elements [11].

These topological advances also yielded a general method for handling circuits with multiple nonlinearities [4] with WDFs, previously restricted to a special case. In this formulation, the nonlinearities are solved via table lookup [4] or iteration. Properties of Newton-based iterative approaches are studied in [12] and applied to a complex preamplifier circuit involving four nonlinear triode tubes in [13].

One of the main motivations for the creation of the *RT-WDF* library was to provide a reference implementation of these theoretical advances, which are not represented in existing software packages.

2.2. Existing Software Packages

Apart from generic signal processing environments like *Matlab*, platforms for systematically implementing real time virtual analog and physical modelling algorithms in the wave domain have existed for more than a decade now. A review of some of the packages mentioned here can be found in [14].

Even though it was not specifically designed for WDFs, *BlockCompiler* [15] was one of the first environments which was used for their implementation [16].

An approach more specifically tailored to WDFs was a program called *BCT* with its own GUI to graphically arrange and configure circuit elements in binary connection trees [17].

Both software packages are reviewed together with case studies in [16]. One advantage of *BlockCompiler* is its ability to generate optimized C-code of algorithms whereas an advantage of *BCT*

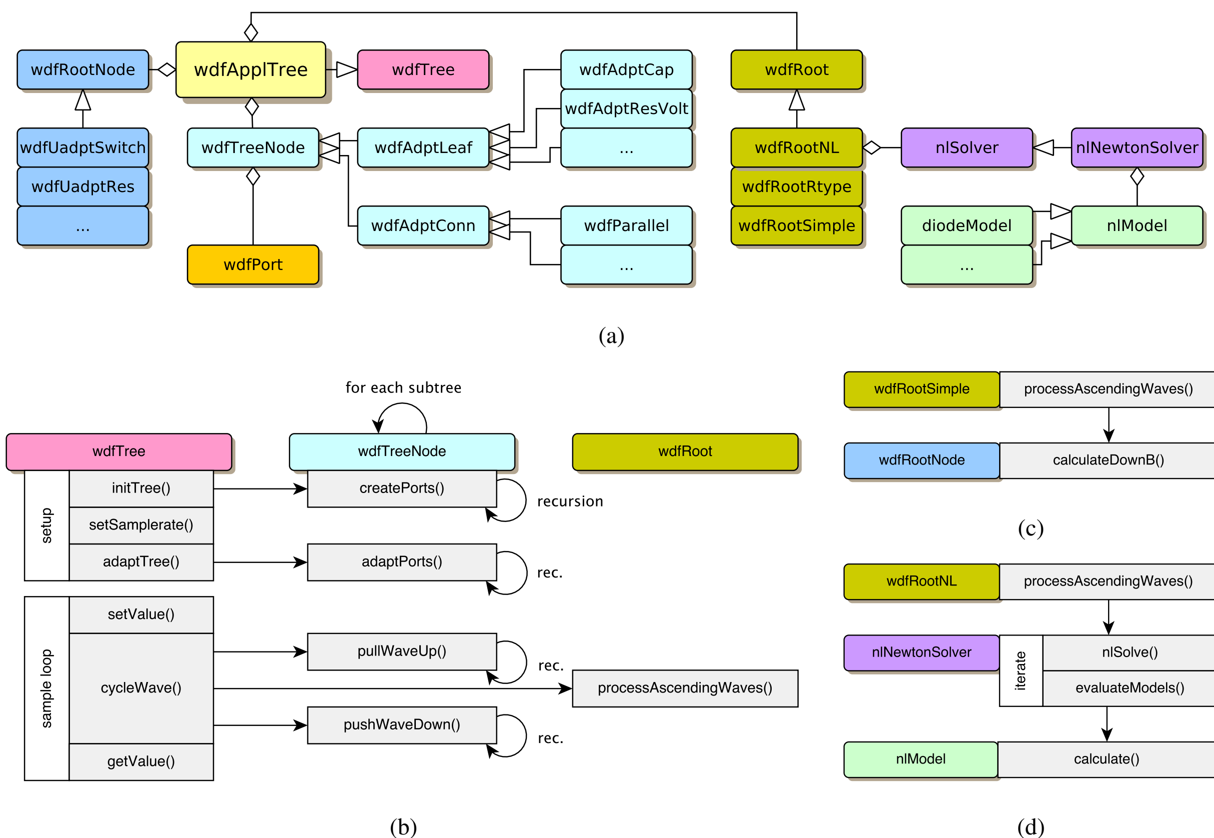


Figure 1: *General RT-WDF framework overview: (a) involved classes and their dependencies, (b) high level functions involved to initialize and run a WDF structure, (c) call graph of a simple WDF root implementation, (d) call graph of a nonlinear root with iterative solver.*

is its intuitive and user-friendly approach. None of them is fully modular in terms of their supported elements, portable to different computer architectures or supports arbitrary topologies and multiple/multiport nonlinearities.

The first public WDF programming library was published in [18] and features a modular, object-oriented class hierarchy written in *Matlab*. It acted as a blueprint for the release of a first public *C++* implementation in the audio developer community around the *JUCE* framework [19]. This library was later refined and published by the same author as *WDF++* [20]. It features architectural modularity and portable code but necessarily could only reflect the state of the art at the time of its development.

Other virtual analog modeling approaches operate in the Kirchhoff i–v domain. *LiveSPICE* [21] builds up on nonlinear state-space models [22] and has support for common nonlinear multiport electrical elements. Another program, *Signaldust 'Salt'* [23], is only available as a closed source preview within the audio DSP developer community and was never fully published.

3. FRAMEWORK OVERVIEW

The framework presented in this paper comes in the form of a publicly available library written in *C++*. This ensures compatibility with popular frameworks for audio applications like *JUCE* [24] and audio plugin APIs such as *LV2*, *VST*, *AU* and *AAX*. It also allows for a structure of hierarchical classes that clearly reflect WDF tree topologies and thus also serves an educational purpose to access the field. Linear algebra functionality is supplied by the third-party *Armadillo* library [25] and all example circuits (see Section 8) are implemented in a standalone host written in *JUCE*.

An overview of the class structure is given in Figure 1a. Within *RT-WDF*, the two main elements in all WDF structures are `wdfTreeNode` and `wdfRoot`. Every WDF *tree* consists of one or more *subtrees* that are formed by *adaptors* and *leafs* as extensions of `wdfTreeNode`. These parallel ($\mathcal{P}$), series ($\mathcal{S}$) and rigid ($\mathcal{R}$) adaptors scatter the waves correctly between the root and the leafs. The leafs represent linear electrical components such as resistors, reactances and non-ideal sources which are the endpoints of each *branch*. The other end of such a subtree is connected to a `wdfRoot` embodying the unadapted circuit components. If these unadapted elements have a closed-form wave domain description they can be implemented as a `wdfRootNode`. Nonlinear elements without closed-form representations are implemented as an `nlModel` which is solved iteratively using an `nlSolver`.

A particular WDF implementation of a circuit with all its adaptors, components and root elements is contained in a user-implemented extension of the `wdfTree` class, the application specific `wdfApplTree`. This class contains all elements and circuit values and provides the API necessary to operate the WDF towards the host application. The required sequence and call graph of these standard methods can be seen in Figure 1b. All functions are initi-

ated from the host by calling methods on such a `wdfApplTree` object as shown in Listing 1.

These methods clearly divide into setup and processing tasks. The first step after instantiation of an `wdfApplTree` is initialization, accomplished by calling `initTree()`. This method itself calls the recursive `createPorts()` function on the entry nodes of the subtrees extending from the root node. This assigns each tree node its up- and down-facing `wdfPort` object which keeps track of wave values and port resistances. Setting the sample rate is necessary for the next step, `adaptPorts()`, as the adaptation of reactive elements depends on the sample rate $f_s = 1/T$.

```
1  //create wdf
2  wdfApplTree myWdfTree( );
3
4  //set up wdf
5  myWdfTree->initTree( );
6  myWdfTree->setSamplerate( Fs );
7  myWdfTree->adaptTree( );
8
9  //process samples
10 for ( int n=0; n<numSamples; n++ ){
11   myWdfTree->setInValue( inSample[n] );
12   myWdfTree->cycleWave( );
13   double outSample[n] = myWdfTree->getOutValue( );
14 }
```

Listing 1: *High level usage of a user-implemented WDF structure from a host.*

Listing 2 illustrates the pattern of recursive function calls that traverse a subtree from the root to the leafs by considering the example of the `adaptPorts()` function.

The adaptation is carried out by first traversing down to the leafs, calculating their up-facing port resistances and then successively passing them on to the parent nodes while keeping also these parent nodes always adapted towards the root. Similar recursive schemes are implemented for example in `pullWaveUp()` and `pushWaveDown()` too.

```
1 double wdfTreeNode::adaptPorts( double T ){
2   for ( wdfPort* dport : downPorts ){
3     dport->Rp = dport->connectedNode->adaptPorts( T );
4   }
5
6   upPort->Rp = calculateUpRes( T );
7   return upPort->Rp;
8 }
```

Listing 2: *Recursive adaptation of the tree.*

After initialization, processing of each audio sample in the WDF is initiated by three function calls: `setInValue()`, `cycleWave()` and `getOutValue()`. The first and the latter have to be overwritten by the user in `wdfApplTree` to correctly assign the input value to the desired source component and collect the output value correctly. Cycling the wave is readily implemented in the `wdfTree` base class as shown in Listing 3.

This listing illustrates the concept of subtrees that hang off the root and again utilizes recursive methods to push and pull wave components to and from all of the leafs of the tree. The tree nodes which are connected to the root are handled as *subtree entry nodes* and act as the starting point of recursive traversals.

Between pulling and pushing, ascending wave components are processed in the root as specified in `wdfApplTree` and the result is returned as descending waves. The different root configurations are explained in detail with examples in Sections 4.1–4.3. The object and method dependencies of a root with a single unadapted one-port (`wdfRootSimple`) and one with multiple nonlinearities (`wdfRootNL`) are shown in Figure 1c/1d respectively. Of course it is worth noting that all these function calls and their dependencies are hidden from the host application and the user-implemented application tree by using a strong hierarchical approach and exposing the internal behaviour of the library only via a few generic high level functions and constructors.

```
1  void wdfTree::cycleWave( ){
2    int treeNo = 0;
3    for ( wdfTreeNode* subtree : subtreeEntryNodes ){
4      (*ascWaves)[treeNo++] = subtree->pullWaveUp( );
5    }
6
7    root->processAscendingWaves( ascWaves, descWaves );
8
9    treeNo = 0;
10   for ( wdfTreeNode* subtree : subtreeEntryNodes ){
11     subtree->pushWaveDown( (*descWaves)[treeNo++] );
12   }
13 }
```

Listing 3: *WDF cycle wave function.*

4. EXAMPLES

This section contains three example circuits that have been modeled using *RT-WDF* to create real time audio algorithms. The host application is based on *JUCE*, which readily provides all audio input and output functionality in a callback function by default and allows the simple creation of graphical user interfaces.

The examples are chosen to highlight several modular concepts of the framework and introduce the three available root types in detail.

4.1. Switchable Attenuator

The first example illustrates the usage of the `wdfRootSimple` object, which supports a single unadapted one-port element at the root. The circuit under examination is a switchable attenuator (Figure 2a) that consists of a voltage source V_{in}, a resistive voltage divider formed by R_1 & R_2 and a switch SW_1 to short the upper resistor R_1. This circuit could obviously also be modeled with less sophisticated approaches than WDF but we chose it here to introduce the library's main concepts on a simple example.

To turn this circuit into its WDF representation, all independent nodes and elements are first labeled with a letter and a digit respectively. These nodes, elements and their interconnections are transformed into a graph representing the circuit (Figure 2b). The graph separation techniques of [26] are applied, after which the graph is transformed into an SPQR tree (Figure 2c). This tree directly yields the WDF representation of the circuit. It consists of adaptors $\mathcal{P}_1$ and $\mathcal{S}_1$ that were introduced by the replacement graphs and the circuit elements SW_1, V_{in}, R_1 and R_2 (Figure 2d). Listing 4 shows the extensions of the `wdfTree` class necessary to model this particular circuit in *RT-WDF*. Such a class always begins with the declaration of pointers for all adapted tree nodes involved, in this case for the resistors R_1, R_2, voltage source V_{in}[1] and adaptors $\mathcal{S}_1$ and $\mathcal{P}_1$. The switch SW_1 is a non-adaptable but linear

[1] Please note that an arbitrary 1Ω resistor in series with the voltage source was necessarily introduced to yield a non-ideal, adaptable voltage source that can serve as a leaf component of the WDF tree. It would alternatively be possible to combine R_2 and the voltage source V_{in} into a non-ideal source and omit $\mathcal{S}_1$.

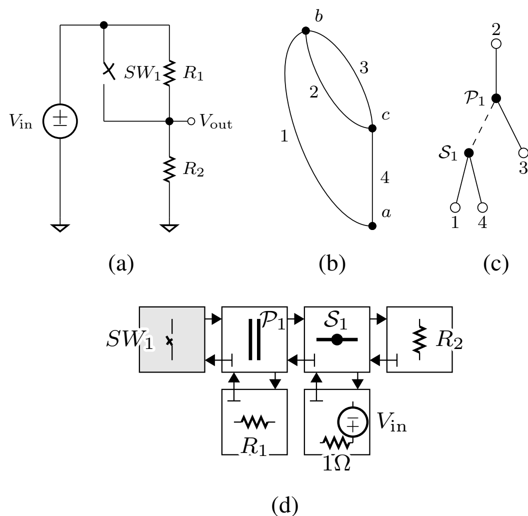


Figure 2: *Deriving a WDF adaptor structure for the switchable attenuator: (a) circuit, (b) graph, (c) SPQR tree, (d) WDF adaptor structure.*

WDF element, which has a closed-form reflection coefficient. It is thus treated as a non-adaptable `rootNode` in this framework and can be implemented using a `wdfRootSimple`

```
1  class wdfAttenTree : public wdfTree
2  {
3  private:
4      wdfAdaptedRes*          R1;
5      wdfAdaptedRes*          R2;
6      wdfAdaptedResVSource*   Vres;
7      wdfAdaptedSeries*       S1;
8      wdfAdaptedParallel*     P1;
9      wdfUnadaptedSwitch*     SW1;
10 public:
11   wdfAttenTree( ){
12     //treeNodes
13     Vres = new wdfAdaptedResVSource( 0,1 );
14     R1 = new wdfAdaptedRes( 250e3 );
15     R2 = new wdfAdaptedRes( 250e3 );
16     S1   = new wdfAdaptedSeries( Vres, R2 );
17     P1   = new wdfAdaptedParallel( S1, R1 );
18     //rootNodes
19     SW1  = new wdfUnadaptedSwitch( 0 );
20
21     subtreeEntryNodes.push_back( P1 );
22     root = new wdfRootSimple( SW1 );
23   }
24   void setInValue( double voltageIn ){
25     Vres->Vs = voltageIn;
26   }
27   double getOutValue( ){
28     return Res2->upPort->getPortVoltage( );
29   }
30   void setParams( std::vector<double> params ){
31     SW1->setSwitch( (int)params[0] );
32   }
33 };
```

Listing 4: *Switchable attenuator tree.*

type root. A pointer for the switch is declared as a private member of the class.

The constructor of the class begins with the creation of the tree and root nodes. Adapted elements are created and initialized according to their physical parameters and the unadapted switch is initialized to be 'open'. The next step is conceptually important: the pointer to the single subtree entry node needs to be stored in a `wdfTree` base class member, `subtreeEntryNodes`. It is used to initiate recursive calls that traverse the subtrees as described in Section 3 and Listing 3. The pointer to the root node `SW1` is handed over to the root's constructor to register it as the root element. The pointer to this root is stored in another member of the base class, the `root` pointer.

The initialization of the WDF elements is followed by the required definitions of the functions `setInValue()` and `getOutValue()`, virtual methods of the `wdfTree` base class. They are used to set and get the input and output samples. Input samples can usually be directly set as voltages or currents of sources. Output samples are retrieved as (a combination of) port voltages or currents for which the port object holds a `getPortVoltage()` and `getPortCurrent()` function. In this case the voltage from the up-facing port of `Res2` is collected, which is the voltage across resistor R_2. The last method demonstrates the ability to further extend the base class to manipulate individual circuit elements: `setParams()` implements the switching functionality of our circuit. It can be called at runtime between samples to effectively configure the reflection coefficient of the root element on the fly. This functionality is necessary to model the circuit in Figure 2a in *RT-WDF*. The resulting `wdfAttenTree` class can now be operated as shown in Listing 1 as a real time algorithm.

4.2. Bassman Tone Stack

The second example makes use of the `wdfRootRtype` class, which allows multiport adaptors with arbitrary topologies in the form of an $\mathcal{R}$-type adaptor at the root. The Fender Bassman tone stack circuit is taken as an example as it is well studied [27] and has a rigid topology that has only recently been supported in WDFs [3]. The circuit's schematic as well as graph-, tree- and WDF-representations are shown in Figure 3. The same process as in the first example is carried out to transform the circuit into an SPQR tree[2]. The additional step here is to capture the rigid connections between the subtree ports of the $\mathcal{R}$-type adaptor in a scattering matrix $\mathbf{S}$ using instantaneous Thévenin port equivalents [3] and modified nodal analysis (MNA) [28].

Listing 5 shows an excerpt of the implementation of the circuit. The setup of the tree nodes is similar to the previous example and not repeated here. Three extra steps must be carried out to set up the $\mathcal{R}$-type root: in contrast to the former example, this time six subtrees need to be registered with their respective entry nodes. These are pointers to the six elements that are directly connected to the $\mathcal{R}$-type adapter at the root, namely `Vin_R3m`, `S2`, `S3`, `C2`, `R4` and `C3`. Secondly, the root object must be created with the number of subtrees as a parameter. This ensures sufficient memory allocation for the scattering matrix $\mathbf{S}$ within the root. In the last step the `setRootMatrData()` function is overwritten. This is an empty function in the `wdfTree` base class and needs to be implemented for specific root class types, including the one used here. Within this function, a `matData` struct is configured to hold the correct values for all required matrices in the root. `setRootMatrData()` is called by `adaptTree()` if the current root requires it. For the $\mathcal{R}$-type root, the `Smat` member of this

[2]The $\mathcal{R}$-type adaptor is chosen here as the root of the tree. It could also reside further down as a tree node adaptor, but this requires an inherent adaptation rule for the up-facing port that depends on the topology and the down-facing port resistances. See [3] for an example.

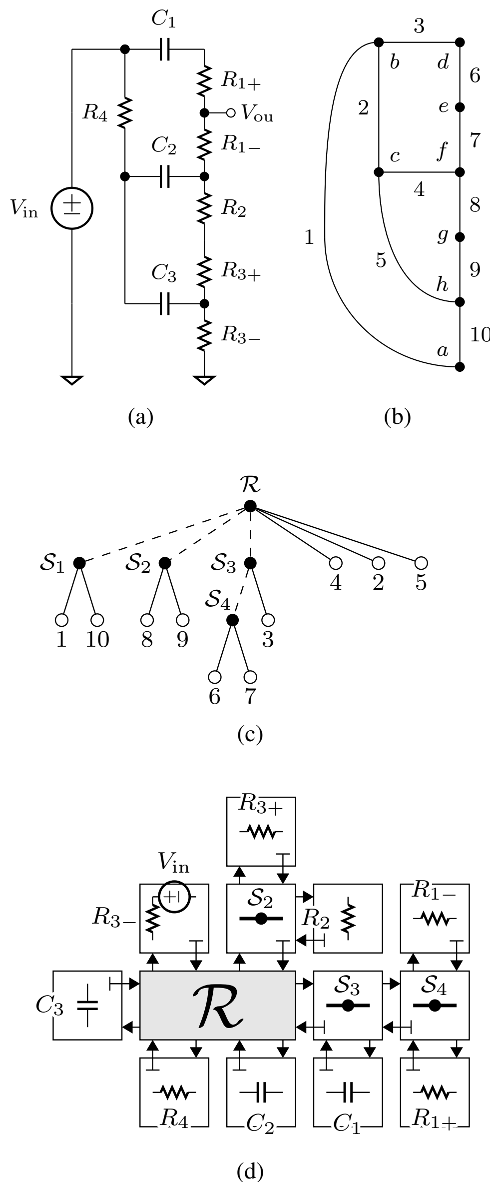


Figure 3: *Deriving a WDF adaptor structure for the Fender Bassman tone stack: a) circuit, b) graph, c) SPQR tree, d) WDF adaptor structure. Modified from [3]*

struct needs to be correctly initialized to embody the scattering behaviour. Dynamic coefficients that depend on the $\mathcal{R}$-type adapter port resistances are supported and enable the user to vary component values in the subtrees (thus affecting the circuit's behaviour) in real time during operation. This functionality is utilized in the code resources of this example.

At the end of the listing, the composition of the output voltage is shown in detail again to demonstrate the flexibility to collect several voltages across circuit elements and adaptors.

```
class wdfTonestackTree : public wdfTree
{
private:
  // pointers for tree elements
  ...
public:
  wdfTonestackTree( ){
    // create tree elements
    ...
    // collect subtree entry points
    subtreeEntryNodes.push_back( Vin_R3m );
    subtreeEntryNodes.push_back( S2 );
    subtreeEntryNodes.push_back( S3 );
    subtreeEntryNodes.push_back( C2 );
    subtreeEntryNodes.push_back( R4 );
    subtreeEntryNodes.push_back( C3 );
    // create new root
    root = new wdfRootRtype( numSubtrees );
  }
  int setRootMatrData( matData* rootMats,
                       double* Rp[] ){

    // populate rootMats->Smat according to
    // R-type scattering behaviour and subtree
    // port resistances Rp[]
    ...
  }
  double getOutValue( ){
    return R1m->upPort->getPortVoltage( ) +
           S2->upPort->getPortVoltage( ) +
           R3m->upPort->getPortVoltage( );
  }
  ...
```

Listing 5: *(partial) Bassman Tone Stack tree class with multiple subtrees and root matrix data update function.*

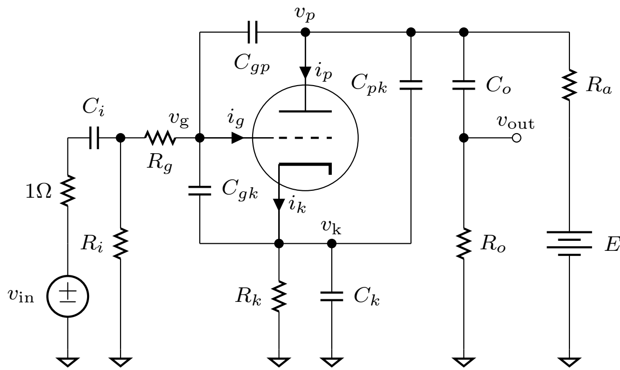
4.3. Common Cathode Triode Amplifier

The final example highlights the ability of the *RT-WDF* library to handle multiple/multiport Kirchhoff-nonlinearities in circuits via the `wdfRootNL` and `nlModel` classes. Here we model the common cathode triode tube amplifier shown in Figure 4a which has been studied for example in [29] as well. The results of recent WDF research [3, 4] to support arbitrary topologies enable us to extend the model from [29] to include the parasitic capacitances C_{gk}, C_{gp} and C_{pk} as well as continuously evaluated triode grid current I_{g}. Deriving the WDF adaptor structure is again accomplished as in the previous two examples, the result of which is shown in Figure 4b. The extension of the `wdfTree` class for this circuit again implements all elements and their topology in the form of tree nodes and registers all subtree entry nodes (not shown).

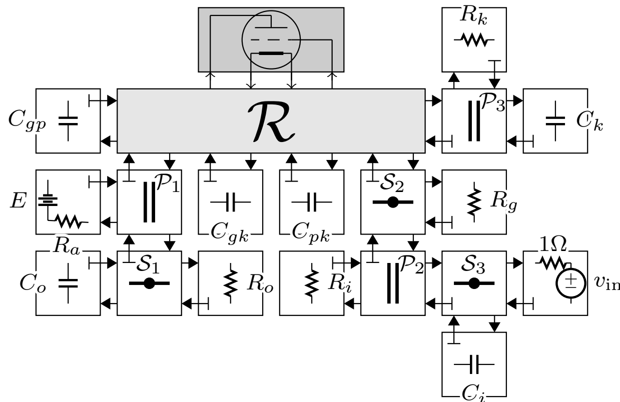
```
wdfCCTATree( ){
...
root = new wdfRootNL( numSubtrees,
                      {12AX7_DW},
                      NEWTON );
}
int setRootMatrData( matData* rootMats,
                     double* Rp[] ){

  // populate rootMats->{Emat,Fmat,Mmat,Nmat}
  // according to R-type scattering behaviour
  // and subtree port resistances Rp[]
  ...
}
...
```

Listing 6: *(partial) Common Cathode Triode Amplifier tree class with nonlinear root element and appropriate root matrix data update function.*

The root is created as a `wdfRootNL` object with the number of subtrees, a vector to specify the nonlinear models and the desired



(a)



(b)

Figure 4: *Common Cathode Triode Amplifier: a) circuit, b) WDF adaptor structure with Kirchhoff nonlinearity. Grey:* `wdfRootNL`*; Dark Grey:* `12AX7DwModel`*.*

solver (Listing 6). In its current state the library supports a multi-dimensional Newton Solver as described in [12] in detail. For this type of root, the `setRootMatrixData()` method must configure the root's system matrices correctly [4]. These matrices implicitly contain a w–K converter to transform the wave variables into the Kirchhoff domain and back. All iterative nonlinear models are currently evaluated in the i–v domain. For this circuit the actual nonlinearity is specified from a user expandable list of models as `12AX7_DW`, a triode model after Dempwolf et al. [30].

$$i_{\mathrm{k}} = G \cdot \left(\log\Big(1 + \exp\big(C \cdot (\frac{1}{\mu} \cdot v_{\mathrm{pk}} + v_{\mathrm{gk}})\big)\Big) \cdot \frac{1}{C} \right)^{\gamma} \tag{1a}$$

$$i_{\mathrm{g}} = G_{\mathrm{g}} \cdot \left(\log\Big(1 + \exp\big(C_{\mathrm{g}} \cdot v_{\mathrm{gk}}\big)\Big) \cdot \frac{1}{C_{\mathrm{g}}} \right)^{\xi} + i_{\mathrm{g0}} \tag{1b}$$

$$i_{\mathrm{p}} = i_{\mathrm{k}} - i_{\mathrm{g}} \tag{1c}$$

The model is described by Equations (1a)–(1c) with perveances G, G_{g}, adaption factors C, C_{g} and positive exponents γ, ξ. The nonlinear two port model is defined in terms of the port voltages $v_{pk} = v_p - v_k$, $v_{gk} = v_g - v_k$ and port currents i_p, i_g. To be used with the Newton solver in this library, it is desirable that the modeling equations are continuously differentiable with respect to their port voltages in a region around the solution and the Jacobian must be invertible [12].

In general, these nonlinear model objects are managed by the `nlNewtonSolver` as shown in Figure 1a / 1d. They always consist of a `calculate()` function that reflects the physical behavior and a `getNumPorts()` method in the base class for housekeeping.

```
1  12AX7DwModel::12AX7DwModel()
2      : nlModel (2){
3
4  }
5  void 12AX7DwModel::calculate( vec* fNL, mat* JNL,
6                                vec* x,   int* port ){
7
8    double Vpk = x->at( *port);
9    double Vgk = x->at((*port)+1);
10
11   // calculate triode currents & their derivatives
12   // and assign them to the vector / matrix entries
13   ...
14
15   fNL->at(   *port )                    = Ip;
16   JNL->at(  (*port),      (*port) )     = dIp_dVpk;
17   JNL->at(  (*port),    ((*port)+1) )   = dIp_dVgk;
18
19   fNL->at(  (*port)+1 )                 = Ig;
20   JNL->at( ((*port)+1),  (*port) )      = dIg_dVpk;
21   JNL->at( ((*port)+1), ((*port)+1) )   = dIg_dVgk;
22
23   (*port) = (*port)+getNumPorts( );
24 }
```

Listing 7: *Implementation of the nonlinear 12AX7 triode model.*

$$\boldsymbol{f}_{\mathrm{NL}}(\boldsymbol{v}) = \begin{bmatrix} i_{\mathrm{p}} \\ i_{\mathrm{g}} \end{bmatrix} \quad \text{with} \quad \boldsymbol{v} = \begin{bmatrix} v_{\mathrm{pk}} \\ v_{\mathrm{gk}} \end{bmatrix} \tag{2}$$

and its Jacobian matrix

$$\mathbf{J}_{\mathrm{NL}} = \begin{bmatrix} \frac{\partial i_{\mathrm{p}}}{\partial v_{\mathrm{pk}}} & \frac{\partial i_{\mathrm{g}}}{\partial v_{\mathrm{pk}}} \\ \frac{\partial i_{\mathrm{p}}}{\partial v_{\mathrm{gk}}} & \frac{\partial i_{\mathrm{g}}}{\partial v_{\mathrm{gk}}} \end{bmatrix}. \tag{3}$$

The Newton Solver iteratively evaluates its specified models for each sample, converging towards a solution of the nonlinear system within a certain tolerance. This solution is then transformed back into the wave domain and returned as descending waves from the root down into the subtrees.

It must be noted that modeling of any nonlinear part in a circuit may introduce drastic aliasing and sufficient oversampling might be necessary to achieve the desired spectral results in the output signal [13]. Also, care must be taken that the selected physical models meet certain criteria in the operating range of the circuit or (fast) convergence of the Newton Solver is not guaranteed [12].

5. PERFORMANCE

All three example circuits from Sections 4.1–4.3 were also implemented in the *SPICE* distribution *LTSpice* and a CCRMA-internal object-oriented Matlab WDF framework. *RT-WDF* and *Matlab* WDF simulations were performed at double precision and a sample rate of $f_s = 44\,100\,\mathrm{Hz}$. *LTSpice* simulations were carried out as a transient analysis with waveform compression disabled. All parameters were left at their default values except `.OPTIONS numdgt=10` to enable internal double precision too. The maximum time step was set to $t_{\mathrm{max}} = 20\,\mu\mathrm{s} \approx 1/f_s$.

All processing times were captured on a laptop computer from 2013 with Intel i7 2,4 GHz CPU (4 cores) and 8GB RAM running OS X 10.11.4. The *RT-WDF* binaries were built with Apple

quantity	input	RT-WDF	SPICE	Matlab	
duration	8.717 s	0.042 s	15.498 s	252.120 s	(1)
norm. dur.	1	0.005	1.778	28.923	
ratio	–	1	369	6003	
duration	8.717 s	0.149 s	31.172 s	670.107 s	(2)
norm. dur.	1	0.017	3.576	76.874	
ratio	–	1	209	4522	
duration	2.961 s	0.272 s	16.411 s	324.652 s	(3a)
norm. dur.	1	0.092	5.543	109.643	
ratio	–	1	60	1192	
duration	2.961 s	1.035 s	50.014 s	1279.034 s	(3b)
norm. dur.	1	0.350	16.893	431.960	
ratio	–	1	48	1235	

(1) switchable attenuator (2) fender tone stack
(3a) triode amplifier (3b) triode amplifier @ $4\times f_s$

Table 1: *Comparison of processing times of WDF and SPICE circuit simulations for all case studies.*

LLVM 7.1 at optimization level `-O3` and ran with a single processing thread without any other considerable applications in the background.

The results of the benchmarking are shown in Table 1. The first row of each simulation holds the absolute processing times in seconds for all three approaches. For *RT-WDF*, the value describes the subsumed processing times of a block-wise operation, for *SPICE*, the value is taken from the logfile. In *Matlab*, the time to finish the sample processing loop is measured with `tic` and `toc`. The second row shows the normalized times with respect to the length of the input signal. This can be seen as an estimate of CPU load for a real-time operation, as it describes the relative amount of time needed by the simulation to process a certain amount of input samples. A normalized duration > 1 could not catch up with the input signal at full CPU load and the algorithm is thus not real-time capable. The last row correlates the performance of all three approaches in terms of "× times slower than" with respect to the least demanding candidate.

It is clear that for all case studies the *RT-WDF* simulation is by far the most performant approach to model this circuit as a modular, physically informed algorithm. As indicated by the normalized processing times, all of them could run in real-time applications, even with $\times 4$ oversampling enabled for the common cathode triode amplifier[3].

6. CONCLUSIONS

We presented the modular WDF *C++* library *RT-WDF* [5] which implements recent research advances in the field. It provides great opportunities for both researchers and audio algorithm developers to approach WDFs for analog modeling of lumped systems.

Due to its custom tailored codebase it greatly decreases computational demands compared to other implementations and is portable to many hardware platforms. Many classic WDF elements (such as resistors, capacitors, parallel and series adapters, etc.) are already implemented in the library and future extensions can be easily added due to its strictly modular, hierarchical approach.

The authors encourage and highly appreciate contributions to the codebase to keep up with current research and further improve the performance of the library.

[3]For the *SPICE* comparison of the $\times 4$ simulation, the maximum time step was set to $t_{\max} = 5\,\mu\text{s} \approx 1/(4 \times f_s)$.

7. ACKNOWLEDGMENTS

We would like to thank Michael Jørgen Olsen for his investigation of the application of Newton's Method to WDFs. Maximilian Rest likes to thank Christoph Hohnerlein for inspiring discussions and CCRMA for supporting the development of *RT-WDF*.

8. RESOURCES

The GNU GPL licensed version of the *RT-WDF* library as well as a reference documentation and examples can be found on GitHub at

`www.github.com/m-rest/rt-wdf`

9. REFERENCES

[1] Alfred Fettweis, "Wave digital filters: Theory and practice," *Proc. of the IEEE*, vol. 74, no. 2, pp. 270–327, 1986.

[2] Giovanni De Sanctis and Augusto Sarti, "Virtual analog modeling in the wave-digital domain," *IEEE Trans. on Audio, Speech and Language Processing*, vol. 18, no. 4, pp. 715–727, 2010.

[3] Kurt J. Werner, Julius O. Smith III, and Jonathan S. Abel, "Wave digital filter adaptors for arbitrary topologies and multiport linear elements," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-15)*, Trondheim, NO, Nov. 30 – Dec. 3 2015.

[4] Kurt J. Werner, Vaibhav Nangia, Julius O Smith III, and Jonathan S. Abel, "Resolving wave digital filters with multiple/multiport nonlinearities," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-15)*, Trondheim, NO, Nov. 30 – Dec. 3 2015.

[5] Maximilian Rest, W. Ross Dunkel, and Kurt J. Werner, "RT-WDF–a modular wave digital filter library," Available at www.github.com/m-rest/rt-wdf, 2016.

[6] David T Yeh, Jonathan S Abel, and Julius O Smith III, "Automated physical modeling of nonlinear audio circuits for real-time audio effects—part i: theoretical development," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 18, no. 4, pp. 728–737, 2010.

[7] Kristjan Dempwolf, Martin Holters, and Udo Zölzer, "Discretization of parametric analog circuits for real-time simulations," in *Proc. of the International Conference on Digital Audio Effects (DAFx-10)*, Graz, AU, Sep. 6 – Sep. 10 2010.

[8] Laurence William Nagel and Donald O. Pederson, *SPICE: Simulation program with integrated circuit emphasis*, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1973.

[9] MATLAB, *version 7.10.0 (R2010a)*, The MathWorks Inc., Natick, Massachusetts, 2010.

[10] Kurt J. Werner, W. Ross Dunkel, Maximilian Rest, Michael J. Olsen, and Julius O. Smith III, "Wave digital filter modeling of circuits with operational amplifiers," in *Proc. of the 24th European Signal Processing Conference. (EUSIPCO-24)*, Budapest, HU, Aug. 29 – Sep. 2 2016.

[11] Kurt J. Werner, W. Ross Dunkel, and François G. Germain, "A computational model of the hammond organ vibrato/chorus using wave digital filters," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-16)*, Brno, CZ, Sep. 5 – Sep. 9 2016.

[12] Michael J. Olsen, Kurt J. Werner, and Julius O. Smith III, "Resolving grouped nonlinearities in wave digital filters using iterative techniques," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-16)*, Brno, CZ, Sep. 5 – Sep. 9 2016.

[13] W. Ross Dunkel, Maximilian Rest, Kurt J. Werner, Michael J. Olsen, and Julius O. Smith III, "The Fender Bassmann 5F6-A family of preamplifier circuits—a wave digital filter case study," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-16)*, Brno, CZ, Sep. 5 – Sep. 9 2016.

[14] Julius O Smith III, "Keynote 2: Recent progress in wave digital audio," Nov. 30 – Dec. 3 2015, Available at https://youtu.be/kUk35_WwTEQ.

[15] Matti Karjalainen, "BlockCompiler: A research tool for physical modeling and DSP," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-03)*, London, UK, Sep. 8 – Sep. 11 2003, pp. 264–269.

[16] Rudolf Rabenstein, Stefan Petrausch, Augusto Sarti, Giovanni De Sanctis, Cumhur Erkut, and Matti Karjalainen, "Block-based physical modeling for digital sound synthesis," *IEEE Signal Processing Magazine*, vol. 24, no. 2, pp. 42–54, March 2007.

[17] Giovanni De Sanctis, Augusto Sarti, Gabriele Scarparo, and Stefano Tubaro, "An integrated system for the automatic block-wise synthesis of sounds," in *Proc. of the 5th European Signal Processing Conference. (EUSIPCO-05)*, Antalya, TR, Sep. 4 – Sep. 8 2005.

[18] Udo Zölzer, Xavier Amatriain, and Daniel Arfib, *DAFX: digital audio effects, Ed. 2*, John Wiley & Sons, 2011.

[19] Maxime Coorevits, "Wave Digital Filter (WDF) with Juce," Available at http://www.juce.com/forum/topic/wave-digital-filter-wdf-juce, Feb. 2013, accessed Feb. 17, 2016.

[20] Maxime Coorevits, "WDF++ - new restructuration & project (audio processor)," Available at http://www.juce.com/forum/topic/wdf-new-restructuration-project-audio-processor, Jul. 2014, accessed Feb. 17, 2016.

[21] Dillon Sharlet, "LiveSPICE: a real time SPICE simulator for audio signals," Available at http://www.livespice.org, Nov. 2013, accessed Feb. 18, 2016.

[22] Dillon Sharlet, "How LiveSPICE works: numerically solving circuit equations," Available at http://www.dsharlet.com/2014/03/28/livespice-numerically-solving-differential-algebraic-equations-circuits, Mar. 2014, accessed Feb. 18, 2016.

[23] Teemu Voipio, "Signaldust 'Salt'," Available at http://www.kvraudio.com/forum/viewtopic.php?f=246&t=398728&hilit=vst+circuit+design, Dec. 2013, accessed Feb. 18, 2016.

[24] "JUCE framework," Available at http://www.juce.com.

[25] Conrad Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," 2010.

[26] Dietrich Fränken, Jörg Ochs, and Karlheinz Ochs, "Generation of wave digital structures for networks containing multiport elements," *IEEE Trans. on Circuits Systems I: Regular Papers*, vol. 52, no. 3, pp. 586–596, 2005.

[27] David T. Yeh and Julius O. Smith III, "Discretization of the '59 Fender Bassman tone stack," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-06)*, Montreal, CA, Sep. 18 – Sep. 20 2006, pp. 18–20.

[28] Chung-Weng Ho, Albert E. Ruehli, and Pierce A. Brennan, "The modified nodal approach to network analysis," *IEEE Trans. on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, 1975.

[29] Stefano D'Angelo, Jyri Pakarinen, and Vesa Välimäki, "New Family of Wave-Digital Triode Models," *IEEE Trans. on Audio, Speech, and Language Processing*, vol. 21, no. 2, pp. 313–321, Feb. 2013.

[30] Kristjan Dempwolf and Udo Zölzer, "A physically-motivated triode model for circuit simulations," in *Proc. of the International Conference on Digital Audio Effects. (DAFx-11)*, Paris, FR, Sep. 19 – Sep. 23 2011, vol. 11.