

# Wave Digital Filters in circuit simulation

December 9, 2016

*This notebook is a work in progress. Please send notes, comments and other suggestions to [olafur.bogason@mail.mcgill.ca](mailto:olafur.bogason@mail.mcgill.ca)*

## 1 An introduction to Wave Digital Filters in circuit simulation

When starting out Wave Digital Filters can seem cryptic and implementing them time consuming to say the least. In this notebook I hope to shed some light on Wave Digital Filters and how they work in practice. This is not a review article about the current research trends.

Wave Digital Filters (WDFs) were first mentioned in a German patent the late 1960s by Alfred Fettweis. Originally they were used to digitize RCL ladder circuits, that is circuits that can be decomposed into a cascade of series and parallel connections. WDFs can be seen as a [finite difference scheme](#) with some nice numerical properties. Wave Digital theory allow an audio DSP designer to create algorithms that are *physically informed*. That is, the algorithms that retain the underlying structure (topology) of the modelled circuit and approximations (usually lumped elements) of how circuit components physically behave. The Wave part of Wave Digital Filters is a reference to the fact that the independent variables are so called wave-variables, which all EE engineers know from the study of [Two-port networks](#). Having independent variables that are waves allows us to use concepts such as [scattering](#) and impedance matching.

For each lumped element in the Kirchoff/K-domain there exists a *port* in the Wave Digital/WD-domain. Similarly for series and parallel connections in the K-domain there exist series and parallel *adaptors* in the WD-domain. Many circuits in the wild however cannot be split into a cascade of series and parallel adaptors.

That means that the scope of circuits that were tractable via WDF was very limited. Recently researchers have found a way to use WDFs to cover [arbitrary topologies](#). This method proposed to use of [Modified Nodal Analysis](#) along with some clever linear algebra tricks to describe the scattering behaviour of a multi-port R-type (Rigid) adaptor.

---

Examples on how to implement WD-structures will be given in Python code. The reason for choosing Python is that it is an interpreted language and its simplicity allows me to concentrate on the explanation of WD concepts rather than the programming itself. There is a nice real-time C++ library being developed [here](#).

To keep things simple, concepts from Object Oriented Programming are used to model individual WD ports. Simple functions are used for adaptors to show how computation is carried out. Instead of displaying all the code at once snippets of it are interleaved with the text. Concepts discussed are thus put to practice straight away. The results from simulations of WD structure are compared to results obtained by simulating the same circuits in [LTspice](#).

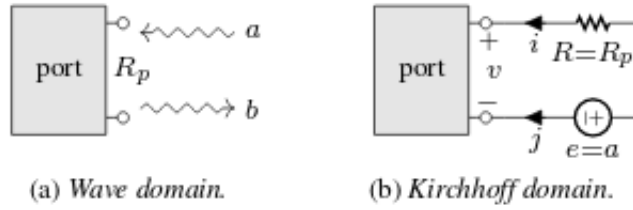


Figure 1: *Instantaneous Thévenin port equivalent.*

## WDF derivation for example 1

In [22]: `%matplotlib inline`

```
import numpy as np
import pandas as pd
from scipy import signal
from sympy import *
import matplotlib.pyplot as plt

FS = 96e3 # Sample frequency.
N = 2**14 # Number of points to simulate.

input = np.zeros(N)
input[0] = 1 # Input is a delta function.
output = np.zeros(input.size)
steps = np.arange(N)

def plot_freqz(x, title="Frequency response"):
    # Plot the frequency response of a signal x.
    w, h = signal.freqz(x, 1, 2048*2)
    H = 20 * np.log10(np.abs(h))
    f = w / (2 * np.pi) * FS
    plt.semilogx(f, H, label="WDF")
    plt.xlim([np.min(f), np.max(f)])
    plt.title(title)
    plt.xlabel('Frequency [Hz]')
    plt.ylabel('Magnitude [dB]')

def plot_ltspice_freqz(filename, title="Frequency response", out_label='V'):
    # Plot the frequency response of a LTspice simulation containing
    # frequency and complex signal.
    def imag_to_mag(z):
        # Returns the magnitude of an imaginary number.
        a, b = map(float, z.split(','))
        return 20*np.log10(np.sqrt(a*a + b*b))
```

```

x = pd.read_csv(filename, delim_whitespace=True)
x['H_dB'] = x[out_label].apply(imag_to_mag)

f = np.array(x['Freq.'])
H_db = np.array(x['H_dB'])
plt.semilogx(f, H_db, label="LTspice")

```

## 2 WD-domain variables

One-port elements in electrical circuits include components such as resistors, capacitors and inductors. They also include ideal and resistive voltage and current sources along with short and open circuits. Many more one-port elements exist, but those are some of the elements that have already been ported over to the WD-domain and are used in practice.

K-domain variables  $v$  and  $i$  are linearly mapped to WD-domain variables  $a$  and  $b$  with the following linear transforms:

$$a = v + i \cdot R_p \quad (1)$$

$$b = v - i \cdot R_p \quad (2)$$

$$v = \frac{1}{2}(a + b) \quad (3)$$

$$i = \frac{1}{2R_p}(a - b) \quad (4)$$

$R_p$  is called a **port resistance**. It is physically introduced into the system, meaning that in practice it can be set to any real value.

An integral part of coming up with a WDF structure is to digitize the reactive components in the reference circuit. Although most text out there will only mention the use of bilinear transform to digitize circuits, it is by no means the only way. There are even certain cases where using it may render the [WD simulation unstable](#). For our case we will still use it for the sake of its simplicity.

$$s \leftarrow \frac{1}{2f_s} \frac{z - 1}{z + 1} \quad (5)$$

There is one thing to keep in mind during the following derivations. For the resulting WDF structure to be computable we cannot have any delay-free loops inside it. That means that for each one-port element we are looking for a function  $f$  that maps  $a \rightarrow b$  but  $b_t$  cannot depend on  $a_t$ . It can only on delayed versions of  $a$ , e.g.  $a_{t-1}$ .

To deal with this issue we have theoretical two tools at our disposal: (a) the port resistance, (b) make one port of a single adaptor reflection free by using the concept of scattering. Making a port reflection free means that we choose the port resistance at that port so that the reflected wave does not depend directly on the incident wave at that same port. Note that the reflected wave can, and often will, depend on incident waves from other ports of the same adaptor.

### 3 One port elements

Starting with the K-domain representation of a one-port element it is simple to derive the most common one-port elements. The linear mappings between the K- and WD-domain are used extensively in this part.

```
In [23]: class WDFOnePort(object):
        def __init__(self):
            # Incident- and reflected wave variables.
            self.a, self.b = 0, 0

        def wave_to_voltage(self):
            voltage = (self.a + self.b) / 2
            return voltage
```

#### 3.0.1 WDF Resistor

$$Z_R = R = \frac{v}{i} \quad (6)$$

$$= \frac{\frac{1}{2}(a+b)}{\frac{1}{2R_p}(a-b)} \quad (7)$$

$$b = \frac{R - R_p}{R + R_p} a$$

If we write that out with explicit time we get:

$$b[n] = \frac{R - R_p}{R + R_p} a[n]$$

so we see that the reflected wave depends instantaneously on the incident wave. We have a delay-free loop. To break that loop we must use our free parameter and set  $R_p = R$ . That choice causes  $Z_R = 0$  and  $b = 0$  for all resistor elements. In other words it means that the reflected wave is always zero independent on the value of incident wave! But what does that really mean? Will the resistors not contribute at all to the WDF structure?

Far from it. Because we set the port resistance equal to the physical resistance of the resistor it directly contributes to the scattering of incident waves at the adaptor that is connected to the resistor. More about that later.

```
In [24]: class Resistor(WDFOnePort):
        def __init__(self, R):
            WDFOnePort.__init__(self)
            self.Rp = R # Port resistance set to physical resistance

        def get_reflected_wave(self, a):
            self.a = a
            self.b = 0
            return self.b
```

### 3.0.2 WDF Capacitor

Now we move on to reactive elements. and describe the capacitor as a function of the Laplace variable  $s$ .

$$Z_C = \frac{1}{sC} = \frac{v}{i} \quad (8)$$

$$= \frac{\frac{1}{2}(a+b)}{\frac{1}{2R_p}(a-b)} \quad (9)$$

$$b = Sa = \frac{1 - R_p C s}{1 + R_p C s} a$$

The bilinear transform is used to digitize the function  $S$

$$b = Sa = \frac{1 - R_p C \frac{1}{2f_s} \frac{z-1}{z+1}}{1 + R_p C \frac{1}{2f_s} \frac{z-1}{z+1}} a \quad (10)$$

$$= \frac{(1 - 2f_s C R_p) + (1 + 2f_s C R_p)z^{-1}}{(1 + 2f_s C R_p) + (1 - 2f_s C R_p)z^{-1}} a \quad (11)$$

If we transform that into the time-domain we have the following difference equation.

$$b[n] = \frac{1}{1 + 2f_s C R_p} \{ (1 - 2f_s C R_p)a[n] + (1 + 2f_s C R_p)a[n-1] - (1 - 2f_s C R_p)b[n-1] \} \quad (12)$$

Again we have a delay-free loop which we must break up. The only way for that to be achieved is if  $R_p = \frac{1}{2f_s C}$ . Then the reflected wave is  $b = z^{-1}a$ . That is, the reflected wave is simply a unit-delayed version of the incident wave. This result will be different for other methods of discretization than the bilinear transform.

```
In [25]: class Capacitor(WDFOnePort):
    def __init__(self, C):
        WDFOnePort.__init__(self)
        self.Rp = 1 / (2 * FS * C)

    def get_reflected_wave(self, a):
        self.b = self.a
        self.a = a
        return self.b

    def set_incident_wave(self, a):
        self.a = a
```

### 3.0.3 WDF Inductor

Following the same approach as above we can easily derive the WD-domain representation for an inductor.

$$Z_L = sL = \frac{v}{i} \quad (13)$$

$$= \frac{\frac{1}{2}(a+b)}{\frac{1}{2R_p}(a-b)} \quad (14)$$

$$b = Sa = \frac{sL - R_p}{sL + R_p}a$$

Now we use the bilinear transform and digitize S.

$$b = Sa = \frac{\frac{1}{2f_s} \frac{z-1}{z+1} L - R_p}{\frac{1}{2f_s} \frac{z-1}{z+1} L + R_p} a \quad (15)$$

$$= \frac{(2f_s L - R_p) - (2f_s L + R_p)z^{-1}}{(2f_s L + R_p) + (2f_s L - R_p)z^{-1}} a \quad (16)$$

Here as well we have a delay-free loop that we can break up by setting  $R_p = 2f_s L$ . Plugging that result into the equation above we get that  $b = -z^{-1}a$ , which means that the reflected wave is a unit-delayed incident wave with its phase inverted.

```
In [26]: class Inductor(WDFOnePort):
    def __init__(self, L):
        WDFOnePort.__init__(self)
        self.Rp = (2 * FS * L)

    def get_reflected_wave(self, a):
        self.b = self.a
        self.a = a
        return -self.b

    def set_incident_wave(self, a):
        self.a = a
```

### 3.0.4 Short circuit

A short circuit can be seen as a resistor with no resistance, that is  $R = 0$ .

$$b = \frac{R - R_p}{R + R_p}a = -a$$

That results essentially tells us that the incident wave gets completely projected back through the port, although with a change in phase.

We see that in this case the reflected wave depends instantaneously on the incident wave. If we want the resulting WDF structure to be computable we cannot have that. Fortunately the port resistance may be tuned so that the incident wave coming from the port will in fact be zero. That way the reflected wave is also always zero.

```
In [27]: class ShortCircuit(WDFOnePort):
    def __init__(self):
        WDFOnePort.__init__(self)

    def get_reflected_wave(self, a):
        self.a = a
        self.b = -a
        return self.b
```

### 3.0.5 Open circuit

A short circuit can most easily be seen as a resistor with infinite resistance, that is  $R \rightarrow \infty$

$$b = \frac{R - R_p}{R + R_p} a \quad (17)$$

$$= \frac{1 - R_p/R}{1 + R_p/R} a \quad (18)$$

$$= a \quad (19)$$

The difference between a short- and open circuit is the additional minus sign, which stands for a 180 degree change in phase.

```
In [28]: class OpenCircuit(WDFOnePort):
    def __init__(self):
        WDFOnePort.__init__(self)

    def get_reflected_wave(self, a):
        self.a = a
        self.b = a
        return self.b
```

### 3.0.6 Switch

Building a simple on/off switch from the last two one-port elements is fairly simple. The only additional thing needed is a variable that holds the port's internal *state*.

```
In [29]: class Switch(WDFOnePort):
    def __init__(self):
        WDFOnePort.__init__(self)
        __state = False

    def get_reflected_wave(self, a):
        if __state: # Switch closed
            self.a = a
            self.b = -a
        else: # Switch open
            self.a = a
            self.b = a
```

```

        return self.b

    def change_state(self, state):
        __state = state

```

### 3.0.7 Ideal Voltage Source

Picture

We can get the relationship in the case of an ideal voltage source with voltage  $v_s$  directly from the definition of wave variables with respect to Kirchoff ones:

$$v = v_s = \frac{1}{2}(a + b) \quad (20)$$

$$b = 2v_s - a \quad (21)$$

```

In [30]: class IdealVoltageSource(WDFOnePort):
    def __init__(self):
        WDFOnePort.__init__(self)

    def get_reflected_wave(self, a, vs=0):
        self.a = a
        self.b = 2*vs - a
        return self.b

```

### 3.0.8 Ideal Current Source

The same is possible for the ideal current source with source current  $i_s$

$$v = \frac{1}{2R_p}(a - b) \quad (22)$$

$$b = a - 2R_p i_s \quad (23)$$

where  $R_p$  is the port resistance and must be matched, which is a concept that we will cover in the next section.

```

In [31]: class IdealCurrentSource(WDFOnePort):
    def __init__(self):
        WDFOnePort.__init__(self)

    def get_reflected_wave(self, a, i_s=0):
        self.a = a
        self.b = a - 2*R_p*i_s
        return self.b

```

The same applies to the ideal (and non-linear) voltage/current sources as the open and closed circuits: Their port resistances need to be matched to the remaining WDF structure, so that the instantaneous incident wave is cancelled out.



### 3.0.9 Resistive Voltage Source

An ideal voltage source with voltage  $v_s$  connected in series with a resistor  $R$  will give us a linear resistive voltage source. Kirchoff's voltage law gives:

$$v = v_s + Ri \quad (24)$$

$$b = 2v - a \quad (25)$$

$$= 2(v_s + Ri) - a \quad (26)$$

$$= 2v_s + \frac{2R}{2R_p}(a - b) - a \quad (27)$$

We can see that again we have a delay-free loop. If  $R = R_p$  is chosen to be the port resistance then the reflected wave is

$$b = v_s$$

```
In [32]: class ResistiveVoltageSource(WDFOnePort):
    def __init__(self, R):
        WDFOnePort.__init__(self)
        self.Rp = R

    def get_reflected_wave(self, a, v_s=0):
        self.a = a
        self.b = v_s
        return self.b
```

### 3.0.10 Resistive Current Source

A resistive current source is essentially an ideal current source with source current  $i_s$  connected in parallel with a resistor  $R$ . The voltage over the resistor is quantified by  $v = (i - i_s)R$  where  $i$  is the current flowing into the port. We now have:

$$b = a - 2R_p i \quad (28)$$

$$= a - 2R_p \frac{v}{R} + i_s R_p \quad (29)$$

$$= a - \frac{R_p}{R}(a + b) + 2i_s R_p \quad (30)$$

Again if we choose to have  $R = R_p$  then we can break up the delay-free loop and obtain the following equation:

$$b = Ri_s$$

```
In [33]: class ResistiveCurrentSource(WDFOnePort):
    def __init__(self, R):
        WDFOnePort.__init__(self)
        self.Rp = R
```

```

def get_reflected_wave(self, a, i_s=0):
    self.a = a
    self.b = self.Rp*i_s
    return self.b

```

## 4 Adaptors

Now that we have a couple of one port elements laid out we need a way to connect them together. Historically circuits that have been simulated using WDFs have been connected in some seperable series and/or parallel conjunction. That is why we start by deriving series and parallel adaptors.

### 4.0.1 Two-port simple adaptors

**Series** Two port series adaptors are described by the following two equations:

$$0 = v_1 + v_2 \quad (31)$$

$$i_1 = i_2 = i_J \quad (32)$$

where  $i_J$  is the current flowing into the junction.

$$0 = v_1 + v_2 = \frac{1}{2}(a_1 + b_1) + \frac{1}{2}(a_2 + b_2) \quad (33)$$

$$b_1 = \underline{-(a_2 + a_1) - b_2} \quad (34)$$

$$0 = i_2 - i_1 = \frac{1}{2R_2}(a_2 - b_2) - \frac{1}{2R_1}(a_1 - b_1) \quad (35)$$

$$= a_2 - b_2 - \frac{R_2}{R_1}(a_1 - b_1) \quad (36)$$

$$b_2 = \frac{R_1 - R_2}{R_1 + R_2}a_2 - \frac{2R_2}{R_1 + R_2}a_1 \quad (37)$$

$$= \underline{-a_1 + \gamma(a_1 + a_2)} \quad (38)$$

where  $\gamma = \frac{R_1 - R_2}{R_1 + R_2}$  is the [reflection coefficient](#) with a negative sign prefixed. Now we can get the reflected wave at port 1.

$$b_1 = -a_2 + \gamma(a_1 + a_2) \quad (39)$$

$$b_2 = -a_1 + \gamma(a_1 + a_2) \quad (40)$$

**Parallel** We can derive the scattering relationship for parallel adaptors in a similar fashion. Conversely to the series adaptor the currents in the parallel adaptor sum up to zero but the voltages are all equal.

$$0 = i_1 + i_2 \quad (41)$$

$$v_1 = v_2 = v_J \quad (42)$$

where  $v_J$  is defined as the voltage at the junction. Now we can use the definition of the wave variables and derive the relationship of reflected waves to the incident ones:

$$0 = i_1 + i_2 \quad (43)$$

$$= \frac{1}{2R_1}(a_1 - b_1) + \frac{1}{2R_2}(a_2 - b_2) \quad (44)$$

$$= R_2(a_1 - b_1) + R_1(a_2 - b_2) \quad (45)$$

$$b_1 = a_1 + \frac{R_1}{R_2}a_2 - \frac{R_1}{R_2}b_2 \quad (46)$$

Next we use the voltage relationship for series adaptors.

$$0 = v_2 - v_1 = \frac{1}{2}(a_2 + b_2) - \frac{1}{2}(a_1 + b_1) \quad (47)$$

$$= -a_1 + a_2 \frac{R_2 - R_1}{2R_2} + b_2 \frac{R_2 + R_1}{2R_2} \quad (48)$$

$$b_2 = \frac{2R_2}{R_1 + R_2}a_1 - \frac{-R_1 + R_2}{R_1 + R_2}a_2 \quad (49)$$

$$= \underline{a_1 + \gamma(a_2 - a_1)} \quad (50)$$

$$b_1 = a_1 + \frac{R_1}{R_2}a_2 - \frac{R_1}{R_2}b_2 \quad (51)$$

$$= a_1(1 - \frac{2R_1}{R_1 + R_2}) + a_2(\frac{R_1}{R_2} - \frac{R_1}{R_2} \frac{R_1 - R_2}{R_1 + R_2}) \quad (52)$$

$$= \underline{a_2 + \gamma(a_2 - a_1)} \quad (53)$$

Which boils down to.

$$b_1 = a_2 + \gamma(a_2 - a_1) \quad (54)$$

$$b_2 = a_1 + \gamma(a_2 - a_1) \quad (55)$$

Putting it together in vector format further shows us that we have a scattering matrix in 2D space. In the case of a parallel adaptor we have:

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \begin{pmatrix} -\gamma & 1 + \gamma \\ 1 - \gamma & \gamma \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \quad (56)$$

And for the series adaptor we have:

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} \gamma & 1 - \gamma \\ 1 - \gamma & \gamma \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \quad (57)$$

```

In [34]: def parallel_adaptor2(a1, Rp1, a2, Rp2):
        gamma = (Rp1 - Rp2) / (Rp1 + Rp2)
        S = np.array((-gamma, 1+gamma), (1-gamma, gamma))
        b1, b2 = np.dot(S, np.array((a1, a2)))

        return b1, b2

def series_adaptor2(a1, Rp1, a2, Rp2):
        gamma = (Rp1 - Rp2) / (Rp1 + Rp2)
        S = np.array((gamma, 1-gamma), (1-gamma, gamma))
        b1, b2 = np.dot(S, np.array((a1, a2)))

        return b1, b2

```

#### 4.0.2 N-port adaptors

Although N-port simple adaptors can always be broken up into 2- and 3-port adaptors we will still carry out the derivation for a variable N-port adaptors.

**Series** As in the case of a 2-port series adaptor the adaptor is described by voltages and currents at each port.

$$0 = v_1 + v_2 + \dots + v_N \quad (58)$$

$$i_1 = i_2 = \dots = i_N \quad (59)$$

So now we can start working...

$$0 = v_1 + v_2 + \dots + v_N = (a_1 - R_1 i_1) + (a_2 - R_2 i_2) + \dots + (a_N - R_N i_N) \quad (60)$$

$$= a_1 + a_2 + \dots + a_N - i \cdot (R_1 + R_2 + \dots + R_N) \quad (61)$$

$$i = i_i = \frac{1}{2R_i}(a_i - b_i) = \frac{1}{R_1 + R_2 + \dots + R_N}(a_1 + a_2 + \dots + a_N) \quad (62)$$

$$b_i = a_i - \frac{2R_i}{R_1 + R_2 + \dots + R_N}(a_1 + a_2 + \dots + a_N) \quad (63)$$

**Parallel** To save space and simplify calculations I put  $G_i = \frac{1}{R_i}$ . The same equations as for the 2-port parallel adaptor apply, only extended to the N-port case.

$$0 = i_1 + i_2 + \dots + i_N \quad (64)$$

$$v_1 = v_2 = \dots = v_N = v \quad (65)$$

Which can be used to find a relationship between incident and reflected waves.

$$0 = i_1 + i_2 + \dots + i_N = G_1(a_1 - v) + G_2(a_2 - v) + \dots + G_N(a_N - v) \quad (66)$$

$$v = v_i = \frac{1}{2}(a_i + b_i) = \frac{1}{G_1 + G_2 + \dots + G_N}(G_1 a_1 + G_2 a_2 + \dots + G_N a_N) \quad (67)$$

The algebra now places out in the following way

$$b_i = \frac{2}{G_1 + G_2 + \dots + G_N} (G_1 a_1 + G_2 a_2 + \dots + G_N a_N) - a_i \quad (68)$$

$$= \gamma_1 a_1 + \gamma_2 a_2 + \dots + \gamma_N a_N - a_i \quad (69)$$

where  $\gamma_i$  is traditionally defined as

$$\gamma_i = \frac{2 \cdot G_i}{G_1 + G_2 + \dots + G_N}$$


---

## 4.1 Adapting adaptor ports

When connecting two adaptors together a question rises. Which port resistance should we choose? The answer is simple. Take the port which is farther away from the root of the resulting SPQR tree and [match the impedance](#) of the port facing root to the other ports of the adaptors. The matching is dependent on the type of adaptor you have. For series adaptors you sum up the resitances and for parallel adaptors you sum the conductances and inverse them.

### 4.1.1 Examples

Say you have a 3-port series adaptor and would like to adapt port 1. Then the port resistance there becomes

$$R_1 = R_2 + R_3 \quad (70)$$

and the reflected wave at that port thus  $b_1 = -a_2 - a_3$ .

The same procedure applied for a 3-port parallel adaptor. The port resistance at port 1 becomes:

$$R_1 = \frac{1}{1/R_2 + 1/R_3} \quad (71)$$

and the reflected wave is then

$$b_1 = \frac{G_2}{G_2 + G_3} a_2 + \frac{G_3}{G_2 + G_3} a_3 \quad (72)$$

```
In [35]: def series_adaptor(A, R):
          Rtot = np.sum(R)
          Atot = np.sum(A)
          return [a - 2*r / Rtot * Atot for a, r in zip(A, R)]

          def parallel_adaptor(A, R):
              G = [1/r for r in R]
              Gtot = np.sum(G)
              Gamma = [2*g / Gtot for g in G]
              aDotGamma = np.dot(A, Gamma)
              return [aDotGamma - a for a in A]
```

## 5 On computability

To simplify the simulation of a WD structure a [SPQR tree](#) has proven to be useful tool. A SPQR tree is an extension of a Binary Computation Tree. It allows for connctions of R-type adaptors which can have more than one parent and child. In a SPQR tree, the leaves represent components or adaptors and branches represent the connections between them.

At each computation cycle the reflected waves from the leaves are calculated and propagated towards the root of the tree. At the root of the tree the resulting incident wave is used to calculate the reflected wave and it is then propagated back down the tree. Then outputs are gathered and values in reactive components are stored for use in next computation cycle.

To see how a derived WDF structure is “shaken” into a SPQR tree it is best to see the examples below.

## 6 Examples

Now that we have the basic bits and pieces laid out we can start simulating real reference circuits.

### 6.1 Resistive voltage source in series with RC

#### 6.1.1 Reference circuit

#### 6.1.2 WDF derivation

#### 6.1.3 SPQR tree

```
In [36]: # One-port parameters.
V1 = ResistiveVoltageSource(1) # 1 Ohm resistive volt. source
C1 = Capacitor(3.5e-5)         # 0.35 uF capacitor.
R1 = Resistor(10)              # 10 Ohm resistor.

# Port resistances.
Rp1, Rp2, Rp3 = C1.Rp, R1.Rp, V1.Rp

# Simulation loop.
b1, b2, b3 = 0, 0, 0
for i in steps:
    # 1. Gather inputs and scatter them.
    a1 = C1.get_reflected_wave(b1)
    a2 = R1.get_reflected_wave(0)
    a3 = V1.get_reflected_wave(0, input[i]) # Read input signal off volta

    a = (a1, a2, a3)
    Rp = (Rp1, Rp2, Rp3)
    b1, b2, b3 = series_adaptor(a, Rp)

    # 2. Gather outputs.
    output[i] = C1.wave_to_voltage() # Output is voltage over C1.
    C1.set_incident_wave(b1)         # Store new input inside Capacitor.
```

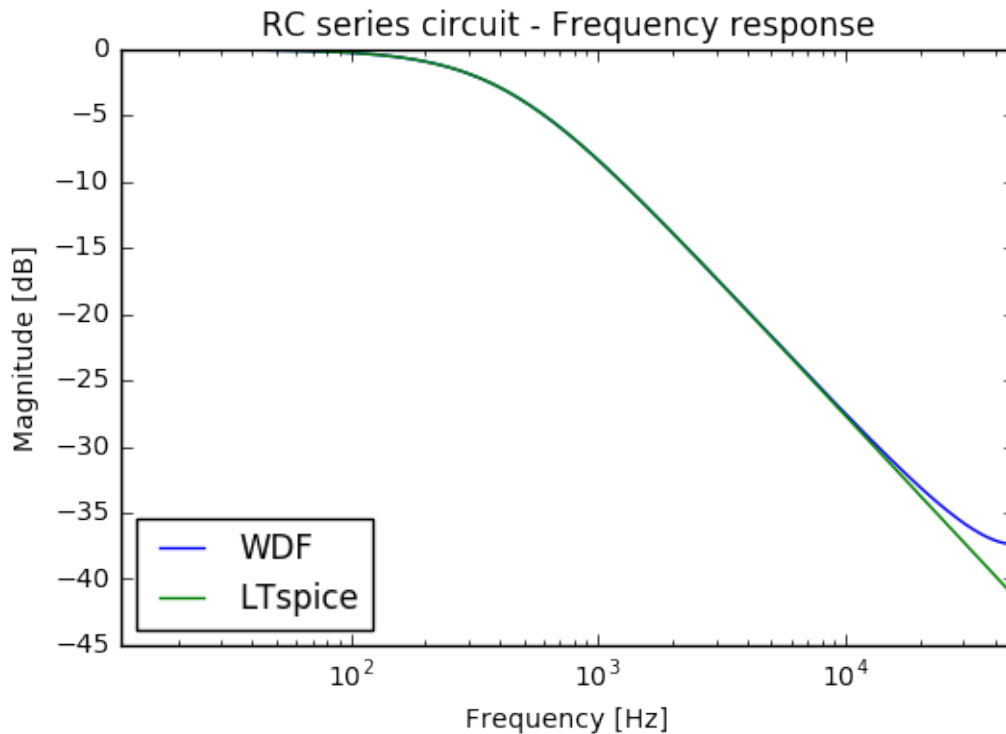
```

# Plot frequency response of the WDF simulation.
plot_freqz(output, title="RC series circuit - Frequency response")

# Plot frequency response from LTspice
plot_ltspice_freqz("data/ex01.txt", title="Frequency response", out_label=

# Show the Frequency response
plt.legend(loc=3)
plt.show()

```



#### 6.1.4 Audio examples

### 6.2 Example - Resistive voltage source connected to RC, RL branches

#### 6.2.1 Reference circuit

#### 6.2.2 WDF derivation

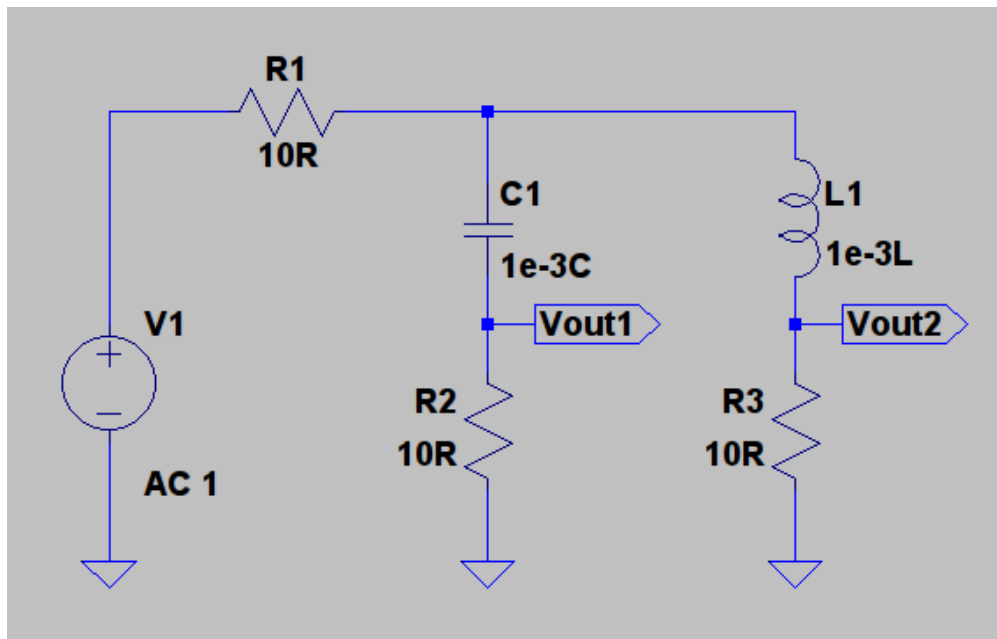
#### 6.2.3 SPQR tree

#### 6.2.4 WDF derivation

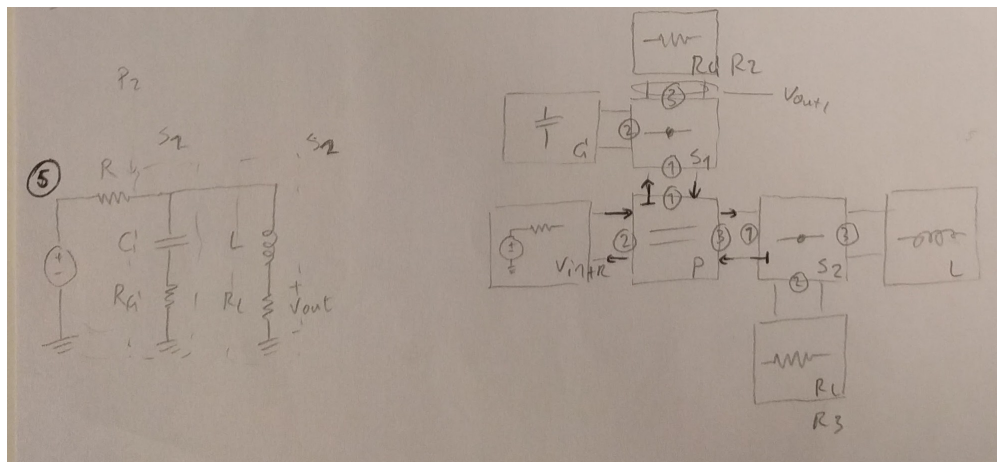
```

In [37]: # Two outputs.
         output_vout1 = np.zeros(input.size)
         output_vout2 = np.zeros(input.size)

```



Reference circuit for example 1



WDF derivation for example 1



```

# One-port parameters.
V1 = ResistiveVoltageSource(10) # Voltage source with 10 Ohm series resis
R2 = Resistor(10) # 10 Ohm resistor.
R3 = Resistor(10) # 10 Ohm resistor.
L1 = Inductor(1e-3) # 1 mH Inductor.
C1 = Capacitor(1e-3) # 1 mF Capacitor.

## Port resistances.
# Series port 2.
Rs22 = R3.Rp
Rs23 = L1.Rp
Rs21 = Rs22 + Rs23 # Matched port.

# Parallel port.
Rp2 = V1.Rp
Rp3 = Rs21
Rp1 = 1 / (1/Rp2 + 1/Rp3) # Matched port.

# Series port 1.
Rs11 = Rp1
Rs12 = C1.Rp
Rs13 = R2.Rp

# Simulation loop.
ap1, ap2, ap3 = 0, 0, 0
as11, as12, as13 = 0, 0, 0
as21, as22, as23 = 0, 0, 0
bs11, bs12, bs13 = 0, 0, 0
bs21, bs22, bs23 = 0, 0, 0
bp3 = 0
for i in steps:
    # 1. Wave up.
    as22 = R3.get_reflected_wave(bs22)
    as23 = L1.get_reflected_wave(bs23)

    as2 = (as21, as22, as23)
    Rs2 = (Rs21, Rs22, Rs23)
    bs21, bs22, bs23 = series_adaptor(as2, Rs2)

    ap2 = V1.get_reflected_wave(bp3, input[i]) # Read signal from voltage
    ap3 = bs21

    ap = (ap1, ap2, ap3)
    Rp = (Rp1, Rp2, Rp3)
    bp1, bp2, bp3 = parallel_adaptor(ap, Rp)

    as11 = bp1

```

```

as12 = C1.get_reflected_wave(bs12)
as13 = R2.get_reflected_wave(bs13)

as1 = (as11, as12, as13)
Rs1 = (Rs11, Rs12, Rs13)
bs11, bs12, bs13 = series_adaptor(as1, Rs1)

# 2. Wave down.
ap1 = bs11
ap = (ap1, ap2, ap3)
Rp = (Rp1, Rp2, Rp3)
bp1, bp2, bp3 = parallel_adaptor(ap, Rp)

as21 = bp3
as2 = (as21, as22, as23)
Rs2 = (Rs21, Rs22, Rs23)
bs21, bs22, bs23 = series_adaptor(as2, Rs2)

output_vout1[i] = R2.wave_to_voltage() # Output is voltage through R2
output_vout2[i] = R3.wave_to_voltage() # Output is voltage through R3

C1.set_incident_wave(bs12)
L1.set_incident_wave(bs23)

# Plot frequency response of the WDF simulation.
plot_freqz(output_vout1, title="RCL circuit, $Vout_1$ - Frequency response")

# Plot frequency response from LTspice
plot_ltspice_freqz("data/ex03_vout1.txt", title="Frequency response", out_

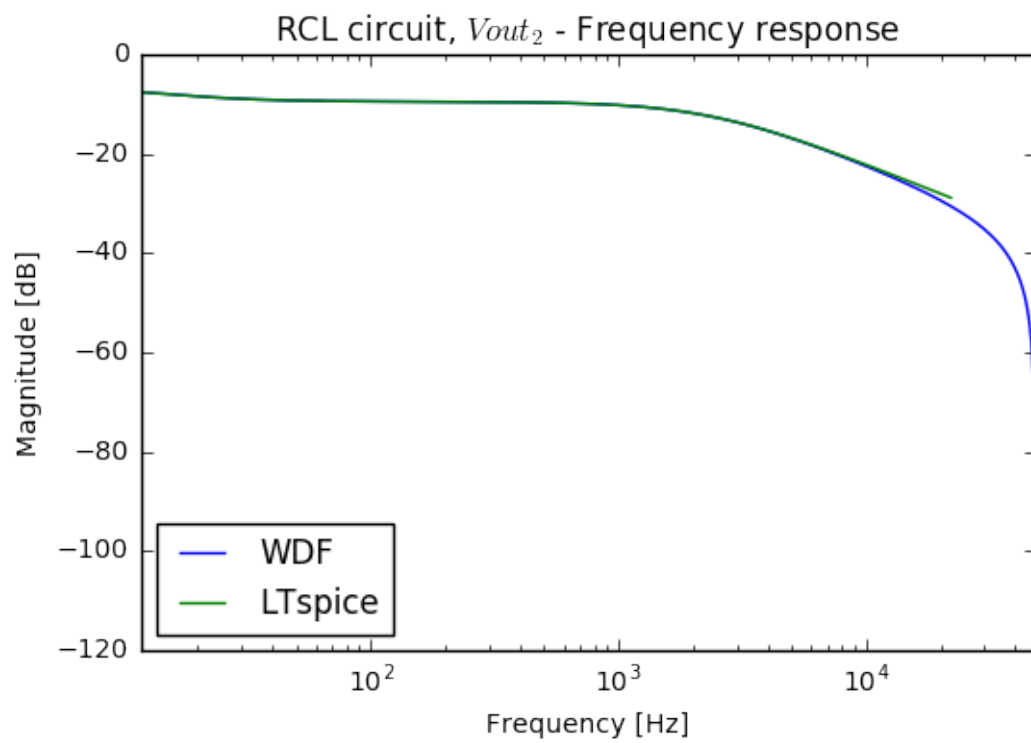
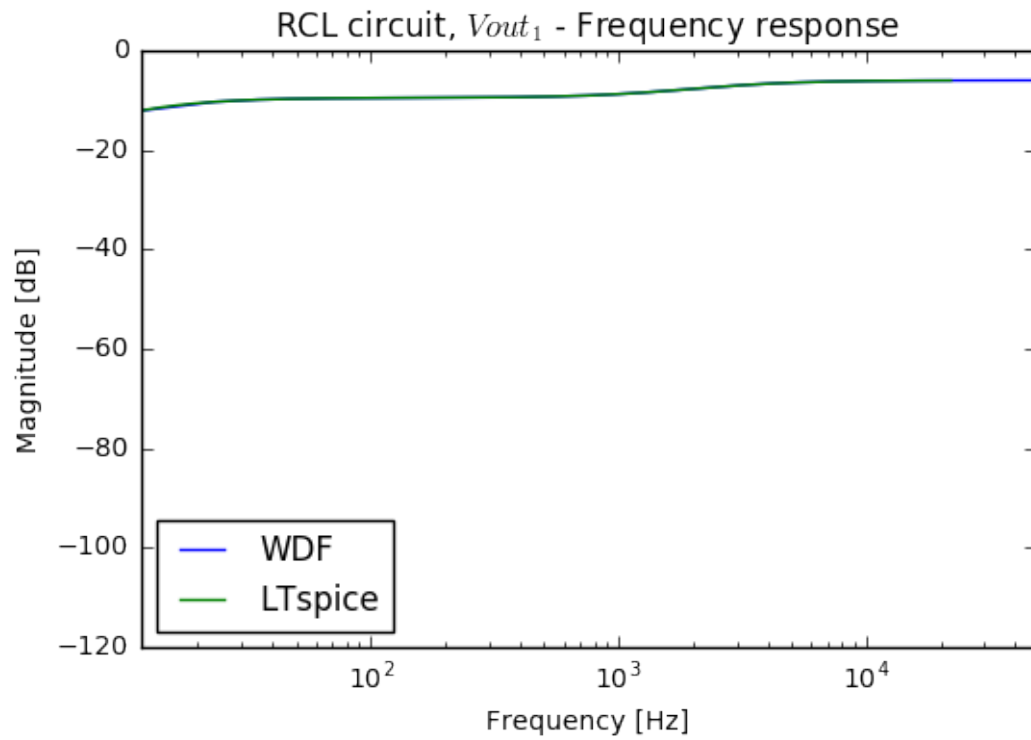
# Show the Frequency responses
plt.legend(loc=0) # Find best location of legend box.
plt.show()

# Plot frequency response of the WDF simulation.
plot_freqz(output_vout2, title="RCL circuit, $Vout_2$ - Frequency response")

# Plot frequency response from LTspice
plot_ltspice_freqz("data/ex03_vout2.txt", title="Frequency response", out_

# Show the Frequency responses
plt.legend(loc=0) # Find best location of legend box.
plt.show()

```



## 7 LTspice files