RF-LinkBudget Documentation

Release 0

Alexander Ott

Dec 12, 2021

Contents:

CHAPTER 1

Motivation

RF-LinkBudget is a software package to aid RF Hardware Developer in defining and comparing LinkBudget's of RF Ciruits.

It calculates key parameter's like cumulative Noise Figure, cumulates Gain, Intermodulation Signal Amplitudes and so on.

Something I often saw was that RF Designer use excel tables for calculating the link budget. For simple circuits without switches and configurable attenuators that works okay. But when you have switches which will add or remove an LNA in a signal chain depending on the power level at the input of a specific device, it gets complicated. You can add this to an excel table but you end up in "if-else" formula hell... The excel tables itself have limitations: Try to compare two linkbudgets or show a plot with NF over input power (with switched LNA's).

This package will simplify the simulation with link-budgets where attenuators, switches and even mixers are present.

CHAPTER 2

Limitations

It does this with the (to RF Designers) well known approximation formulas. RF-LinkBudget does **not** simulate a hole RF-Circuit with all S-Parameters and non-linearities.

2.1 Introduction

This tool is used in three phases.

1. First we define a circuit containing multiple devices.
   we connect those and maybe define some callback functions.
2. In the second phase we simulate the circuit, define plots of interest.
3. And in the third phase we optimize the circuit.

2.1.1 A simple example

First we define a simple circuit
We use two amplifiers in series
A Low Noise Amp and a Driver

```
import rf_linkbudget as rf
import matplotlib.pyplot as plt
import numpy as np

cr = rf.Circuit('SimpleEx')

lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2)],
                   NF=0.7,
```

(continues on next page)

(continued from previous page)

```
                    OP1dB=21.5,
                    OIP3=40)

drv = rf.Amplifier("Driver TQP3M9028",
                    Gain=[(0, 14.9)],
                    NF=1.7,
                    OP1dB=21.4,
                    OIP3=40)

src = rf.Source("Source")
sink = rf.Sink("Sink")
```

Now we have to define connections between the devices

```
src['out'] >> lna['in']
lna['out'] >> drv['in']
drv['out'] >> sink['in']
```

that we can simulate the circuit correctly, we have to define where the signal is applied to.
A Port as example.
We do this by using a callback function.
First we define it with an inline function, then connect this function to the Port by using the memberfunction called **regCallback**.

```
# create callback function
def cb_src(self, f, p):
    return {'f': f, 'p': p, 'Tn': rf.RFMath.T0}


src['out'].regCallback(cb_src)  # connect callback to Port
```

after defining all the callbacks we can finalize the circuit

```
cr.finalise()
```

and simulate the circuit

```
sim = cr.simulate(network=cr.net,
                  start=cr['Source'],
                  end=cr['Sink'],
                  freq=[100e6],
                  power=np.arange(-50, -10, 1.0))
```

(continues on next page)

(continued from previous page)

```

h = sim.plot_chain(['p'])

plt.show()
```

We see that we have to define a start port and an end port.
In this case its intelligent enough to choose the cr[‘Source’][‘out’] port and the cr[‘Sink’][‘in’] port.
Also we have to define the frequency and input power range of the simulation

And here we see our output!
In this case we see the signal power **p** from the source to the sink.
We can also show other parameters like noisefigure, signal-to-noise ratio, spurious-free-dynamic-range and others.

```
import rf_linkbudget as rf
import matplotlib.pyplot as plt
import numpy as np

cr = rf.Circuit('SimpleEx')

lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2)],
                   NF=0.7,
                   OP1dB=21.5,
                   OIP3=40)

drv = rf.Amplifier("Driver TQP3M9028",
                   Gain=[(0, 14.9)],
                   NF=1.7,
                   OP1dB=21.4,
                   OIP3=40)

src = rf.Source("Source")
sink = rf.Sink("Sink")


src['out'] >> lna['in']
lna['out'] >> drv['in']
drv['out'] >> sink['in']


# create callback function
def cb_src(self, f, p):
    return {'f': f, 'p': p, 'Tn': rf.RFMath.T0}


src['out'].regCallback(cb_src)  # connect callback to Port

cr.finalise()


sim = cr.simulate(network=cr.net,
```

(continues on next page)

(continued from previous page)

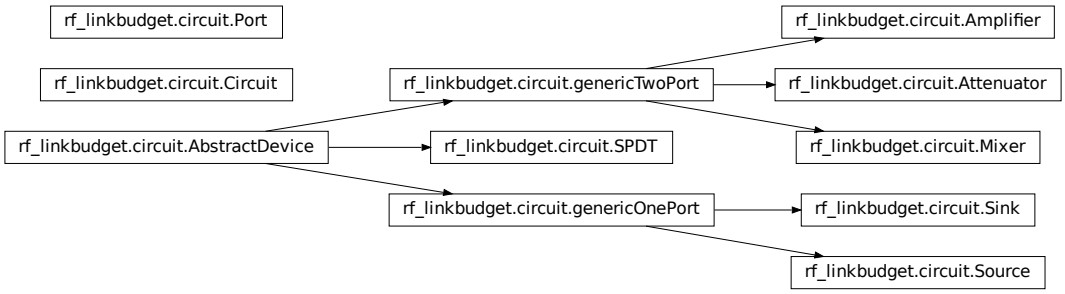
```
                    start=cr['Source'],
                    end=cr['Sink'],
                    freq=[100e6],
                    power=np.arange(-50, -10, 1.0))

h = sim.plot_chain(['p'])

plt.show()
```

2.2 Circuit and Devices

2.2.1 Class diagram



2.2.2 Circuit

The first step in every rf-linkbudget calculation is the creation of the circuit object

```
import rf_linkbudget as rf
cr = rf.Circuit("name of the circuit")
```

after that we have to create devices which are automatically added to the currently active circuit.

```
lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2)],
                   NF=0.7,
                   OP1dB=21.5,
                   OIP3=40)
```

We can access the amplifier in this example with the circuit object. This is just syntax sugar, but will be usefull when we want to configure plots

```
print(cr["LNA TQL9066"])
>>> <rf_linkbudget.circuit.Amplifier object at 0x7f1466e2b400>
```

As in the example we populate our circuit and define connections and register callback functions.

But when we're done with that we want to finalise the circuit.

```
cr.finalise()
```

This function makes some checks and creates a network object. A network of ports. We will see more in the next chapter about this. Lets just say for the moment, that we can model our circuit as a network of input and output ports which have a beginning and an end.

And after we finalised our circuit we want to simulate our results

```
sim = cr.simulate(network=cr.net,
                  start=cr['Source'],
                  end=cr['Sink'],
                  freq=[100e6],
                  power=np.arange(-50, -10, 1.0))
```

We have to deliver our circuit as a network object, a starting port, an ending port a list of frequency points and a list of input power levels. Here we see the usefullness of the syntax sugar for referencing our devices by the circuit object.

The simulation makes a two dimensional matrix with frequency and power rows resp. columns. For every entry in this matrix there will be a full set of values calculated. like signal-to-noise ration, spurious-free-dynamic-range, but also concatenated Intermodulation point IP3 and compression point P1dB. But there will be an own chapter just for the calculations.

Lets just say for the moment, that the return object is from class simResult and does the hard-work...

2.2.3 Port

Every device we will use has at least one Port. Imagine the port as a **model port** from network theory. An amplifier will have two ports, an in and an out port. A source only has one port and 3dB hybrid will have 3 ports.

We can access the ports of a component by the number or for some devices which are non-reciproke with a string 'in' or 'out'

```
lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2)],
                   NF=0.7,
                   OP1dB=21.5,
                   OIP3=40)

 print(lna['in'])
 >>> LNA TQL9066 Port 0
 print(lna['in'] == lna[0])
 >>> True
 print(lna['in'] == lna[1])
 >>> False
 print(lna['out'] == lna[1])
 >>> True
```

As seen in the example we register a callback functions to a port. You probably assumed that the register callback function would be registered to the device object, but I decided to do it differently. As described before a circuit is a network of ports not a network of objects. This way around the calculations are more generalized.

Every port could be the start-or the end-port of a simulation. The network abstracts the point between ports as graphes and for every simulation an algorithm searches for the nearest distance between two ports and returns a directed graph from the start to the end port.

By defining the callback function to a port, we could also simulate the isolation of an amplifier in a generalized way and we would have to push our stimuli on the output port of this amplifier.

Callback Functions

We have to types of callback functions :

1. <port>.regCallback
2. <port>.regCallback_preIteration

The first **regCallback** is used to generate stimuli for a simulation. We can also use it to define the values of a step attenuator depending on the input power level

```
# create callback function
def cb_att1(self, f, p):
    tb = {-21: 2, -20: 3, -19: 4, -18: 5, -17: 6, -16: 7, -15: 8, -14: 9, -13: 10, -
↪12: 11, -11: 12, -10: 13, }
    if(p in tb):
        self.setAttenuation(tb[p])
    else:
        self.setAttenuation(1.6)
    return {}

att1['in'].regCallback(cb_att1)
```

this way the attenuator can be regulated for the simulation. The regCallback gets called every frequency and power level the simulation was specified for. The Callback gets called just before calulating the input / output values of a device

And the second callback **regCallback_preIteration** is used to define settings for each iteration before calculating the individual device parameters. With this callback we can define system settings like if a mixer is an up- or downconverter, the mixer frequency and the noisefigure of the device depending on the LO frequency or power level.

We can refence inside the callback function to a temporary variable called **Circuit.currentSimParams**. Inside this variable all currently set Simulation Parameters are availlable.

```
# create callback function
def cb_att1(self, f, p):
    ...
    freq = Circuit.currentSimParams['freq']
    start_node = Circuit.currentSimParams['start']
    ...
```

We can use this feature to simulate more complex control schemes.

2.2.4 Devices

- *AbstractDevice*
- *genericOnePort*
- *genericTwoPort*
- *Source*
- *Sink*
- *Amplifier*
- *Attenuator*

- *Filter*
- *SPDT*
- *Mixer*
- *Duplexer*

AbstractDevice

The AbstractDevice class is an Abstract Class. Other "real" devices will inherit this class.

It defines that every device has a name and at least one port. It calculates the internal port network. It also implements the behaviour that you can access the port by a key.

It defines an abstract function **calcCurrentEdge** which gets called by the Circuit class to calculate the output values. Every device has to implement this function by its own. And in this function only the "non-dependend" values are calculated.

It defines another function called **calcAdditionalParameters** which calculates all the other values based on the "non-dependend" values.

genericOnePort

The genericOnePort is an Abstract class itself, but also inherits AbstractDevice. It defines that device which inherit this class will exactly have one port and can be called by 'in', 'out' or '0'

genericTwoPort

The genericTwoPort is an Abstract class itself, but also inherits AbstractDevice. It defines that device which inherit this class will exactly have two ports where the input is called 'in' or '0' and the output is 'out' or '1' This class also implements the possibility to use a S2P file to import the device data directly. But be aware this script only extracts the S21 values.

classmethod genericTwoPort.**fromSParamFile**(*name*, *filename*, *Tn*, *P1*, *IP3*, *patchString='S12DB'*)

classmethod to create a two port device from a Touchstone S2P file
only S21 is regarded

Parameters

- **name** (*str*) – device name
- **filename** (*str*) – S2P filename
- **Tn** (*list of [Tn @ Port 1, Tn @ Port 2]*) – noisetemperature of object, represented at the "target" port [°K]
- **P1** (*list of [P1 @ Port 1, P1 @ Port 2]*) – Signal Compression Point of object, represented at the "target" port in [dB]
- **IP3** (*list of [P3 @ Port 1, P3 @ Port 2]*) – Signal Intermodulation Point 3 of object, represented at the "target" port in [dB]

Other Parameters **patchString** (*str*) – default 'S12DB'

Returns **cls** – object

Return type genericTwoPort

The parameter patchString is a little bit odd. The author of the scikit-rf project which implements the touchstone importer actually only knows the generic n-port touchstone file format. The S2P touchstone format does arange the columns differently, thats why we need for extracting the S21 data of a S2P file the column "S12" which is at the same position for a n-port touchstone format as for the S21 values in an S2P touchstone file

Source

The **Source** is a one port device. It has an 'out' port and a name. Usually we use it as a starting point for our simulation

```
src = rf.Source("Source")
...

src['out'] >> ...
```

Sink

The **Sink** is a one port device. It has an 'in' port and a name. Usually we use it as an ending point for our simulation

```
sink = rf.Sink("Sink")
...

... >> sink['in']
```

Amplifier

The amplifier is a two port device. We can define it as following:

```
lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2), (100, 18.0), (200, 17.8)],
                   NF=0.7,
                   OP1dB=21.5,
                   OIP3=40)
```

We have to define at least a name, Gain, Noisefigure, OP1dB and OIP3 level. For the Gain we can also use a S2P file to import the data

```
lna = rf.Amplifier.fromSParamFile("LNA TQL9066",
                                  'data/TQL9066.s2p',
                                  NF=0.7,
                                  OP1dB=21.5,
                                  OIP3=40)
```

The S2P importer only uses the S21 : Gain Information. (No embedded Noisefigure)

We can define the gain as a list of a tuple of frequency and gain.

Attenuator

An attenuator is also a classical two port device. It can be static or variable, defined by a range.

```
att_fix = rf.Attenuator("Att Fix1",
                        Att=[1.5])
```

First we see here the fixed attenuator case.

```
dsa = rf.Attenuator("AFE79xx DSA",
                        Att=np.arange(1.0, 29, 1.0))
```

And here a version with a defined range. We can now use a callback to parameterise.

```
dsa = rf.Attenuator("AFE79xx DSA",
                        Att=np.arange(1.0, 29, 1.0))
...

# create callback function
def cb_dsa(self, f, p):

  if(p > -20):
     self.setAttenuation(0.0)
  else:
     self.setAttenuation(15.6)
  return {}

  dsa['in'].regCallback(cb_dsa)
```

As we see in this example, we set the attenuator to 15.6dB for all input powers **p** below or equal -20dBm and set it to zero above it. The function **setAttenuation** will set it to the nearest attenuation value defined in the constructor. In this case the zero value will get set to effectively 1.0dB and the 15.6dB will be rounded up to 16.0dB.

Filter

An filter is a classical two port device. It has a frequency response

```
filter = rf.Filter("Filter 1",
                        Att=[(100, 1.5),(200, 4.5),(300, 55)])
```

As we see, we can define the frequency response like the frequency response of an Amplifier.

SPDT

```
sw1 = rf.SPDT("SW 1",
                Att=0.3)
...

sw1['S-1'] >> att1['in']
att1['out'] >> sw2['S-1']
sw1['S-2'] >> sw2['S-2']
```

Has ports ‘S’, ‘S-1’, ‘S-2’

```
# create callback function
  def cb_sw1(self, f, p):
      if(p < -21):
          self.setDirection('S-2')
```

(continues on next page)

(continued from previous page)

```
      else:
          self.setDirection('S-1')
      return {}

  sw1['S'].regCallback(cb_sw1)
  sw2['S'].regCallback(cb_sw1)
```

We can also specify the isolation in case of a malconfigured switch

```
sw1 = rf.SPDT("SW 1",
              Att=0.3, Iso=60)
...
```

Mixer

```
mix = rf.Mixer("MIY SYM-18+",
                   Gain=[(0, -8.61), (100, -8.41), (200, -8.64)],
                   OP1dB=14,
                   OIP3=30)
```

The single-sideband noisefigure is equal to the insertion loss of the mixer. We will add the sideband-noise in the callback function

As we will see there is the therm : *(10**(Att/10) * rf.RFMath.T0 - rf.RFMath.T0)* which adds the sideband noise term. In this case the sideband-noise is suppressed down to the noisefloor and will add 3dB noisefigure. We can also simulate the influence of higher sideband noise the same way.

```
# create callback function
def cbI_mix(self, data, f, p):
    # upconverter with flo = 100MHz
    data.update({'f': f + 100e6})
    # add 3dB noise Figure due to sideband noise
    data.update({'Tn': data['Tn'] + (10**(3/10) * rf.RFMath.T0 - rf.RFMath.T0)})
    return data

mix['in'].regCallback_preIteration(cbI_mix)  # connect callback to Port
```

There are two important things to notice here:

1. The callback function we use is calleg **regCallback_preIteration()** this callback will be called each iteration for each component
2. The callback function must be attached at the '**in**' port

In the callback function we manipulate the data['f'] parameter. We can simulate the influence of the frequency conversion with it. Usually the Gain and Noise Values are referenced by data['f']. But in the case, where we mix to a fixed output frequency this is leads to a wrong output values. (because we would use always the same gain / noise values because the output frequency, where we reference on is always the same) We can define in the callback function the following code to define an *own* reference frequency:

```
# create callback function
def cbI_mix(self, data, f, p):
    ...
    data.update({'f_ref': f + 100e6})
    ...
    return data
```

This way the code will use 'f_ref' as refence for the calculations.

Duplexer

TODO

2.3 Math Models

2.3.1 Constants

In our RF-Math Module there are some constants defined:

```
T0 = np.float(290)
N0 = np.float(10*np.log10(constants.k * 290 * 1 * 1000))
```

2.3.2 Parameter

Gain

The gain gets calculated from the given parameter from each device. The values are linear interpolated between the given frequency points.

Noise Temperature / Noise Figure

The noise parameter are internally calculated as noise-temperature. All other noise parameter are converter from the noise-temperature to their scale.

RFMath.**convert_T_n**(*B=1*)

Convert Noise Temperature to Noise Power

Parameters **T** (*float*) – Noise Temperature in [°K]

Returns **n** – noise power in [dBm/Hz] / [dBm/(B*Hz)]

Return type numpy.float

RFMath.**convert_T_NF**(*Gain*)

Convert Noise Temperature to NoiseFigure

Parameters

- **T** (*float*) – Noise Temperature in [°K]
- **Gain** (*float*) – Accumulated Gain in [dB]

Returns **NF** – Noisefigure in [dB]

Return type numpy.float

Notes

Output Power

The output power is simply the input power multiplied by the gain

P1dB

The output P1 parameter is simplistic modeled. It's just a saturating upper bound.

OIP3

The OIP3 resp. IP3 in general is calculated by the famous equation:

```
IP3 = P - (2*(IP3) - P + 6)
```

It is assumed a single CW signal (thus the +6dB)

SNR

The signal-to-noise ratio is exactly the signal - noise in dB.

SFDR

The spurious-dynamic range uses a simple approach and is the distance between output power and intermodulation 3 product.

Dynamic Range

The dynamic range is defined as the smaller of the SNR or SFDR.

2.4 Analyse and Plotting

We go back to our first example:

```
import rf_linkbudget as rf
import matplotlib.pyplot as plt
import numpy as np

cr = rf.Circuit('SimpleEx')

lna = rf.Amplifier("LNA TQL9066",
                   Gain=[(0, 18.2)],
                   NF=0.7,
                   OP1dB=21.5,
                   OIP3=40)

drv = rf.Amplifier("Driver TQP3M9028",
                   Gain=[(0, 14.9)],
                   NF=1.7,
                   OP1dB=21.4,
                   OIP3=40)

src = rf.Source("Source")
sink = rf.Sink("Sink")


src['out'] >> lna['in']
lna['out'] >> drv['in']
drv['out'] >> sink['in']


# create callback function
def cb_src(self, f, p):
    return {'f': f, 'p': p, 'Tn': rf.RFMath.T0}


src['out'].regCallback(cb_src)  # connect callback to Port

cr.finalise()


sim = cr.simulate(network=cr.net,
                  start=cr['Source'],
                  end=cr['Sink'],
                  freq=[100e6],
                  power=np.arange(-50, -10, 1.0))

h = sim.plot_chain(['p'])

plt.show()
```

We focus now on the last part after finalise()

```
sim = cr.simulate(network=cr.net,
              start=cr['Source'],
              end=cr['Sink'],
              freq=[100e6],
```

(continues on next page)

(continued from previous page)

```
                power=np.arange(-50, -10, 1.0))

h = sim.plot_chain(['p'])

plt.show()
```

We see the call to the Circuit.simulate() function. With this we create a simulation-result type. For the calculation we have our circuit **cr** and as first parameter we have to define a route or network through our circuit. We have to explicit state with which node we start and where to end. And then we can define the frequency and input power points we want to calculate our results.

The next command *h = sim.plot_chain(['p'])* is a plotting command and will be discussed later. The *plt.show()* is a special plotting command which commands that the plotting windows are getting drawn. But it blocks the current instance. Thats why we always set this command to the end of the script.

Here we see a simple plot. a so called *sim.plot_chain()*. "chained" because we see all the components in a signal chain, device after device. We see on the right side also some text information:

- *p*
- 100.00MHz
- -50.0dBm

the p is the current viewed variable, the middle is the current active frequency and the last one the current input power level. On some plots these items are clickable and will change when clicked or the mouseweel is used.

We will see more infos in a second.

2.4.1 Plotting

plot_chain

As we already saw, one of the plotting commands is called **plot_chain**. With this plot we see all the data-chain at a specific input frequency and at a specific power. This is a good view to analyse a system by scrolling through all the states and observe the hole component chain.

`simResult.`**`plot_chain`**(*param=None*, *freq=None*, *power=None*, *keys=None*)

plots a window with responsive fields which can be switched between (click or mousewheel)

plot_chain plots all data from the start to the end, by all ports between
we can select which ports to show, be defining the keys
when param, freq or power is not defined, all parameters will be shown

Parameters

- **param** (*str or list of str, optional*) – a single parameter or a list of parameters which shall be plotted
- **freq** (*float or list of floats, optional*) – a single frequency or a list of frequencies which shall be plotted
- **power** (*float or list of floats, optional*) – a single power value or a list of power values which shall be plotted
- **keys** (*str or list of str, optional*) – a single device port or a list of device ports. only these values are extracted.

Notes

return handler must remain, otherwise the plot will not be responsive anymore

Returns **handler**

Return type callback handlers

plot_total

In contrast to plot_chain we have the **plot_total** function. With this we see all the values at the end of the chain. The *conclusion* in some kind. We see a hole input power level sweep at a give frequency or a frequency sweep at a given input power.

With this kind of plot we can easily see the performance over the hole input range for a system.

`simResult.`**`plot_total`**(*param=None*, *freq=None*, *power=None*)

plots a window with responsive fields which can be switched between (click or mousewheel)
plot_total plots all "and" data in respect to frequency or power
we can select which ports to show, be defining the keys
when param, freq or power are not defined, all parameters will be shown

Parameters

- **param** (*str or list of str, optional*) – a single parameter or a list of parameters which shall be plotted
- **freq** (*float or list of floats, optional*) – a single frequency or a list of frequencies which shall be plotted
- **power** (*float or list of floats, optional*) – a single power value or a list of power values which shall be plotted

Notes

return handler must remain, otherwise the plot will not be responsive anymore

Returns **handler**

Return type callback handlers

plot_total_simple

Whereas the plot_total is an interactive window with clickable buttons, this **plot_total_simple** is static, more simplistic but gives the possibility to compare two different systems with each other.

For many RF application this plot together with the values 'SNR' and 'SFDR' resp. 'Dynamic' is a very informative plot to see where we loose SNR or Dynamic and vice versa where we got a sweetspot for both parameters.

`simResult.`**`plot_total_simple`**(*param*, *freq=None*, *power=None*, *axis=None*)

plots a window with the defined parameter
plot_total_simple is a simpler variant of the other plots
it is not a responvice view, but it is easy to show different parameters at once

Parameters

- **`param`** (*`str or list of str`*) – a single parameter or a list of parameters which shall be plotted
- **`freq`** (*`float or list of floats, optional`*) – a single frequency or a list of frequencies which shall be plotted
- **`power`** (*`float or list of floats, optional`*) – a single power value or a list of power values which shall be plotted
- **`axis`** (*`axis object, optional`*) – currently not in use

Notes

return handler must remain, otherwise the plot will not be responsive anymore

Examples

```
>>> t = sim1.plot_total_simple(['SFDR', 'SNR', 'DYN'], freq=430e6)
>>> plt.show()
```

Returns handler

Return type callback handlers

plot_surface

The **plot_surface** is used as a hole view to see the system at all input frequencies versus all input power levels at the same time.

`simResult.`**`plot_surface`**(*param=None*, *freq=None*, *power=None*, *keys=None*)

plots a window with responsive fields which can be switched between (click or mousewheel)
plot_surface plots a surface plot with all data in respect to al ports and power or frequency
we can select which ports to show, be defining the keys
when param, freq or power are not defined, all parameters will be shown

Parameters

- **param** (*str or list of str, optional*) – a single parameter or a list of parameters which shall be plotted
- **freq** (*float or list of floats, optional*) – a single frequency or a list of frequencies which shall be plotted
- **power** (*float or list of floats, optional*) – a single power value or a list of power values which shall be plotted
- **keys** (*str or list of str, optional*) – a single device port or a list of device ports. only these values are extracted.

Notes

return handler must remain, otherwise the plot will not be responsive anymore

Returns **handler**

Return type callback handlers

2.4.2 setNoiseBandwidth

We can adjust the noise-bandwidth of our system to adapt to our simulation needs. The script calculates internally in **1Hz** resolution but shall be adjusted with this function.

`simResult.`**`setNoiseBandwidth`**(*B*)

set the noisebandwidth
changes the following parameters

- n : noise power
- SNR : signal to noise ratio
- DYN : dynamic

2.4.3 extract measurement data

extractValues

`simResult.`**`extractValues`**(*param*, *freq*, *power*, *keys=None*)

convencience function to access the data in a structured way

Parameters

- **param** (*str or list of str*) – a single parameter or a list of parameters which shall be extracted
- **freq** (*float or list of floats*) – a single frequency or a list of frequencies which shall be extracted
- **power** (*float or list of floats*) – a single power value or a list of power values which shall be extracted

- **keys** (*str or list of str, optional*) – a single device port or a list of device ports. only these values are extracted.

Returns **params** – a tuple with all ports and the corresponding data

Return type (ports, data)

extractLastValues

simResult.**extractLastValues**(*key*, *freq=None*, *power=None*)

convencience function to access the data at the last device port.
an easy way to export all "end" values to compare systems with each other

Parameters

- **key** (*str*) – a single device port or a list of device ports. only these values are extracted.
- **freq** (*float, optional*) – a single frequency or a list of frequencies which shall be extracted
- **power** (*float, optional*) – a single power value or a list of power values which shall be extracted

Returns **params** – data

Return type pandas.DataFrame

2.5 Complex Example

Here we have a more complex example

```
import rf_linkbudget as rf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


def example():

    cr = rf.Circuit('Example')

    dup = rf.Attenuator("Duplexer",
                        Att=[1.5])

    lim = rf.Attenuator("Limiter",
                        Att=np.array([0.5, 6.5, 12.5, 18.5]),
                        IIP3=60,
                        OP1dB=30)

    sw1 = rf.SPDT("SW 1",
                  Att=0.3)

    lna = rf.Amplifier("LNA",
                       Gain=[(0, 18.2)],
                       NF=0.7,
```

(continues on next page)

(continued from previous page)

```
                        OP1dB=21.5,
                        OIP3=40)

    sw2 = rf.SPDT("SW 2",
                  Att=0.3)

    att_fix = rf.Attenuator("Att Fix1",
                            Att=[1.5])

    rxfilt = rf.Attenuator("Rx Filter",
                           Att=[2.0])

    att_fix2 = rf.Attenuator("Att Fix2",
                             Att=[1.5])

    driver = rf.Amplifier("Driver",
                          Gain=[(0, 14.9)],
                          NF=1.7,
                          OP1dB=21.4,
                          OIP3=40)

    dsa = rf.Attenuator("DSA",
                        Att=np.arange(1.0, 29, 1.0))

    adc = rf.Amplifier("ADC",
                       Gain=[(0, 0)],
                       OP1dB=-1,
                       OIP3=34,
                       NF=19)

    src = rf.Source("Source")
    sink = rf.Sink("Sink")

    src['out'] >> dup['in']
    dup['out'] >> lim['in']
    lim['out'] >> sw1['S']

    sw1['S-1'] >> lna['in']
    lna['out'] >> sw2['S-1']
    sw1['S-2'] >> sw2['S-2']

    sw2['S'] >> att_fix['in']
    att_fix['out'] >> rxfilt['in']
    rxfilt['out'] >> att_fix2['in']
    att_fix2['out'] >> driver['in']
    driver['out'] >> dsa['in']
    dsa['out'] >> adc['in']
    adc['out'] >> sink['in']

    # create callback function
    def cb_src(self, f, p):
        return {'f': f, 'p': p, 'Tn': rf.RFMath.T0}

    src['out'].regCallback(cb_src)

    # create callback function
    def cb_lim(self, f, p):
```

(continues on next page)

(continued from previous page)

```
        tb = {-31: 6, -30: 6, -29: 6, -28: 6, -27: 6, -26: 6, -25: 12, -24: 12, -23: 12, -22: 12, -21: 12, -20: 12, -19: 18, -18: 18, -17: 18, -16: 18, -15: 18, -14: 18, -13: 18, -12: 18, -11: 18, -10: 18,}
        if(p in tb):
            self.setAttenuation(tb[p])
        else:
            self.setAttenuation(0.0)
        return {}

    lim['in'].regCallback(cb_lim)

    # create callback function
    def cb_dsa(self, f, p):
        tb = {-37: 0, -36: 1, -35: 2, -34: 3, -33: 4, -32: 5, -31: 0, -30: 1, -29: 2, -28: 3, -27: 4, -26: 5, -25: 0, -24: 1, -23: 2, -22: 3, -21: 4, -20: 5, -19: 0, -18: 1, -17: 2, -16: 3, -15: 4, -14: 5, -13: 6, -12: 7, -11: 8, -10: 9,}
        if(p in tb):
            self.setAttenuation(tb[p])
        else:
            self.setAttenuation(0)
        return {}

    dsa['in'].regCallback(cb_dsa)

    # create callback function
    def cb_sw1(self, f, p):
        if(p > -11):
            self.setDirection('S-2')
        else:
            self.setDirection('S-1')
        return {}

    sw1['S'].regCallback(cb_sw1)
    sw2['S'].regCallback(cb_sw1)

    cr.finalise()

    return cr


if __name__ == "__main__":

    # define circuit
    cr1 = example()

    # simualte
    sim1 = cr1.simulate(network=cr1.net,
                        start=cr1['Source'],
                        end=cr1['Sink'],
                        freq=[0],
                        power=np.arange(-50, -10, 1.0))

    # key's of interest
    # only these keys show up in the plots
    keys1 = [cr1['Source']['out'],
             cr1['Duplexer']['out'],
             cr1['Limiter']['out'],
```

(continues on next page)

(continued from previous page)

```
            cr1['SW 1']['S'],
            cr1['SW 1']['S-1'],
            cr1['LNA']['out'],
            cr1['SW 2']['S'],
            cr1['Att Fix1']['out'],
            cr1['Rx Filter']['out'],
            cr1['Att Fix2']['out'],
            cr1['Driver']['out'],
            cr1['DSA']['out'],
            cr1['ADC']['out'],
            cr1['Sink']['in']]

    # set noise bandwidth to smallest subband
    sim1.setNoiseBandwidth(15e3)

    # plot system parameter
    h = sim1.plot_chain(keys=keys1)
    k = sim1.plot_total(['NF', 'DYN'])
    t = sim1.plot_total_simple(['SFDR', 'SNR'], freq=0)
    k = sim1.plot_total(['NF', 'DYN'])

```

With the **plot_total** function we can show the NoiseFigure and the Dynamic of the system.

We see steps in the noisefigure over the input power level. We see that these steps are from the limiter attenuator called **lim / Limiter**. The limiter is in front of the LNA which is not optimal.

We also see that the overall dynamic has losses exactly at the same input levels. Which is not surprising.

When we compare the SNR and the SFDR view in a **plot_total_simple** we see something interesting. We see the sweet spot between SNR and the spurious free dynamic range. To have an optimal system means we have to optimize the circuit in a way, so that SFDR and SNR are equal over the hole input power range.

2.5.1 SNR / SFDR optimisation

To optimize the SNR / SFDR we have to keep the signal power of each component in view.

Here in this plot we got an input power of -32dBm. The Driver Amplifier is at an output power of approx -6dBm. And the ADC level is approx at -12dBm which leads to approx -13dBFS. But already with the next higher input power level we have to adjust the ADC input level, otherwise the IM3 components will dominate the hole dynamic. At -31dBm the limiter was activated and will attenuate approx 6.5dB. We see this in the other figures as the krack in the performance. But it is necessary. Otherwise the spurious would dominate and our signal would have an overall worse dynamic.

2.5.2 The End

Thats it for now.
Have fun with this python package!

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index