Readme Document for CRCat:

Complex Rational Catalog of All RLC Networks of Five or Fewer Elements

For the complete theory behind `CRCat`, illustrated with detailed examples, please read my paper, "Direct Least Squares Rational Polynomial Lumped Circuit Model Extraction and Group Theory," in the IEEE Journal of Microwaves, Digital Object Identifier: 10.1109/JMW.2025.3598986, which is available on IEEE Xplore. The published paper is free for all to download. A pdf is included in the same folder as this README file in this `CRCat` distribution, which is at https://github.com/Sonnet-Software/CRCat.

MATLAB™ R2024a was used for the examples that follow.

## Catalog Data Structure

Any of the commands listed below can be copied and pasted directly into MATLAB if desired.

Bring up MATLAB and make sure that the `CRCat` directory is in your path. Type `aCat=CRCat`. This loads the entire default `CRCat` catalog, which includes 623 RLC networks from the base catalog combined with whatever networks you have added to the custom catalog. It is a large data structure and loading it (which `CRCat` does using the MATLAB `load` routine) takes a while. A good project for anyone more skilled than I would be to make this load procedure faster.

`CRCat` has a few options. For example, `CRCat('base')`, or `CRCat('default')` or `CRCat('custom')` loads only the indicated catalog. You can run `CRCat` silently, for example, with `CRCat('custom',0)`. Detailed options are available with `help CRCat/CRCat`.

It is useful to browse through the `CRCat` code in the folder `@CRCat/CRCat.m`. There are many useful routines there for working with `CRCat` data. The '@' in the directory name means `CRCat` is a class folder, which allows methods for that class to be in different files within that folder. `CRCat` is a 'handle' object, much like a pointer in other languages. Normally when you pass a parameter to a routine, MATLAB makes a copy of the entire parameter and passes it by value. As a handle, MATLAB passes only the location in memory of the parameter. This is important because `CRCat` is so large.

As you browse through the code, note that within a function, the names of parameters passed to it begin with `the`, as in `theCat`. Variables that are local to a function begin with `a`, as in `aCat`. The command you typed above loads the local variable `aCat`.

By the time you have read the above and explored some of the `CRCat` code, the loading of `aCat` is likely finished. You should see the following on your screen (a few blank lines have been removed):

```
>> aCat = CRCat
If there are any changes in the base catalog, you must manually
regenerate it.
If there are any changes in the custom catalog, you must manually
regenerate it.
Loading default catalog. This could take a while.
aCat =
  CRCat with properties:
      fUnit: 'GHz'
      fMult: 1.0000e+09
    resUnit: 'Ω'
    indUnit: 'nH'
    capUnit: 'nF'
       list: [1×630 struct]
>>
```

If you made changes in the custom catalog (as described below), you can regenerate it with `CRCat('custom')`. This command automatically replaces the previous catalog with the regenerated catalog. A subsequent call to `aCat=CRCat` refreshes the default catalog with the new information from your custom catalog. It is strongly recommended that you never make changes in the base catalog as regeneration can require multiple runs and require several weeks. No kidding.

You can change units, for example, with `aCat.Units('MHz')`. Only frequency units can be changed in this way. All other units change with the frequency units so that the numbers internal to `aCat` remain unchanged. For example, changing the frequency units from GHz to Hz means the inductance unit changes from nH (nanoHenries) to H (Henries). The numerical value of any inductances stored in `CRCat` data structures remains unchanged. On occasion, for example when labeling circuit schematics, units are selected to conveniently display the numbers. In these cases, the units are always displayed next to the value. Internally, the values remain unchanged. When dealing with measured data (see the `SnP` data type), frequency values and units are stored within the data structure. Otherwise, changing the `CRCat` frequency units leaves the numerical value of all internally stored frequencies and element values unchanged.

There is a massive amount of information in the `list` field. There are 623 networks in the base catalog, which means, in this case, the custom catalog has seven networks that have been appended to the base catalog, yielding 630 networks in this default catalog. For example, the data for the 500$^{th}$ network:

```
>> aCat.list(500)
ans =
  struct with fields:
        symV: [La    Lb    Rc    Ld    Cd]
       symVn: [La    Lb    Rc    Ld    Cd]
        symC: [a3    a2    a1    a0    b4    b3    b2    b1]
        symA: Rc + Lb*s + Ld*s + Cd*Lb*Ld*s^3 + Cd*Ld*Rc*s^2
        symB: s*(La*Rc + Lb*Rc + La*Lb*s + La*Ld*s + Lb*Ld*s + Cd*L…
       symEL: [1×5 sym]
      testEL: [4.3000 13 5.4000 27 0.0920]
     flagSig: '1111-11110'
       nodes: [1 2 2 0 2 3 3 0 3 0]
        name: 'XXXII-39'
      sameId: [164 182 239 243 244 257 261 262 332 … ] (1×29 double)
     symSame: {1×29 cell}
    symZeros: [3×1 sym]
    symPoles: [4×1 sym]
>>
```

Note that structure fields always begin with a lower-case character. Here, `symV` and `symVn` contain the symbolic variables used for the elements (i.e., the RLCs) in this network. The difference between these two fields and their application is described in the section titled "Specifying and Using Fixed Elements". Using equations in the `symEL` field, the `symV` element values are typically determined from numerical values of the rational polynomial coefficients, which are in turn determined by least-squares fitting to measured data.

I have made extensive use of MATLAB's symbolic data type. If you are unfamiliar with this data type, it would be good to do a quick google for the basics. All structure fields that begin with `sym` are symbolic variables.

Field `symC` holds the symbolic variables for the network's complex rational polynomial. Those that begin with `a` are in the numerator, and those that begin with `b` are in the denominator. The corresponding equation in my CR paper is (1), whose coefficients are symbolically stored in `symC`.

Next we have `symA` and `symB`. These are symbolic equations for the numerator and denominator of the complex rational polynomial that yields the admittance of the network. When we substitute actual values for the RLC elements into this equation, and use $s = j\omega;\ \omega = 2\pi f$, the admittance of the network results.

Field `symEL` provides symbolic equations for the synthesis of the numerical RLC values given numerical values for the `symC` coefficients, which are, in turn, determined by a least-squares fit to measured data. Most networks (but not all) that can have synthesis equations do. An excellent research project would be to find a way to solve for synthesis equations that should exist but have not been found by MATLAB's `solve` function.

The five, comma separated, `symEL` symbolic equations for this network are:

```
>> aCat.list(500).symEL
ans =
[(a0*a2*b2 + a0*a3*b1 - a1*a2*b1)/(a0^2*a3), -(a2*(a0*b2 -
a1*b1))/(a0^2*a3), -(a2^2*(a0*b2 - a1*b1))/(a0^2*a3^2), (a2*(a0*a3 -
a1*a2)*(a0*b2 - a1*b1))/(a0^3*a3^2), (a0^2*a3^2)/((a0*a3 -
a1*a2)*(a0*b2 - a1*b1))]
>>
```

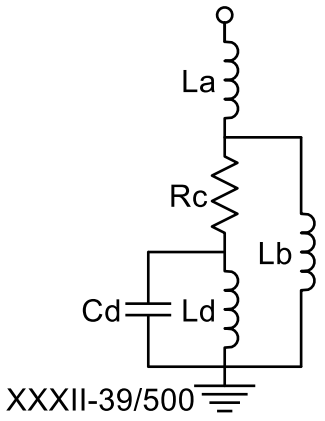
The typical procedure is to least squares fit measured data to the corresponding form of the rational polynomial. Then the resulting numerical values for the $a$ and $b$ coefficients are substituted into the above set of equations. There are five equations, one for each element. The result is the best fit value for each of the five elements forming the network.

In some cases, the MATLAB solve function is unable to solve for the synthesis equations. This becomes especially likely if there are more than five elements. This is where the difference between fields `symV` and `symVn` becomes important. Say we have six elements for a new network in the custom catalog. When we regenerate that catalog, it turns out MATLAB is unable to find the synthesis equations. You can proceed by deleting the last element variable from `symV` while leaving it in `symVn`. Now, CRCat solves synthesis equations only for the five remaining elements in `symV`. But, if successful, those synthesis equations will be a function of the sixth element. Thus, when you do this, you will need to provide a specific value for the sixth element whenever the synthesis equations are used. An example is detailed in the section below titled 'Specifying and Using Fixed Elements'.

The `testEL` field holds RLC values for the `symVn` variables used for testing. The values have been checked to make sure that they have significance in the resulting network admittance, `symA` and `symB`.

Field `flagSig` is a character array that has one character for each term in the network's rational polynomial with a hyphen separating the numerator from the denominator. The character is '1' if the coefficient exists and zero otherwise. This is used when performing the least squares fit. There is no need to least squares fit a coefficient of a particular network if we know in advance that its value must be zero. In this case, `'1111-11110'`, indicates that the zero-frequency coefficient for the denominator, `b0`, is zero. A least squares fitting for this network will be correspondingly constrained. Note that the `symC` field has no `b0` coefficient.

Below is a schematic of XXXII-39 created by typing `aCat.DrawArray(500)`. You can copy and paste schematic drawings into your documents by selecting `Edit->Copy Figure` in the plot window.




Field `nodes` holds a netlist for the network that determines the above schematic. In this case, the first element in `symVn`, `La`, is connected from node 1 to node 2. The second element, `Lb`, is connected between node 2 and ground, etc.

When you add a network to the custom catalog, you specify a unique `name` for the network, the `symVn` symbolic element names and their nodal connections. If `symVn` and `symV` are the same (i.e., you want MATLAB to solve for synthesis equations for all of the elements), you may specify only `symV` if desired. The next time you load the custom catalog, `aCat=CRCat('custom')`, all changes in the custom catalog are automatically evaluated to fill all other fields in the network's structure. Then, the next time you load the default catalog, your new custom catalog is appended to the base catalog, yielding a new default catalog.

Field `name` is a character array used to uniquely identify the network. No other network can have the same name, and this is checked when updating the catalogs. Here, this network uses the XXXII topology. A topology depends only on how the elements are interconnected, it does not depend on what elements occupy those connections. The elements that occupy those interconnections are determined by mapping the `symVn` field to the `nodes` field as illustrated in the above schematic. All possible five-element topologies are illustrated in Figure 6 in my CR paper.

There are typically many ways we can populate a given network topology with elements. This is set by the `symVn` field for each network. This network, which uses the 32$^{nd}$ topology in CRCat, is the 500$^{th}$ network in the catalog, and is the 39$^{th}$ way of populating this topology, hence, `'XXXII-39'` is the network name. The index of a network in the list array is also called its 'Id'. In most cases, the Id and the network name can be used interchangeably, they are equivalent. We expect the Ids of networks in the base catalog to be stable. When referenced, for example, in a publication, and we refer to a network Id, we preface it with 'CR', as in 'CR500' to indicate it is a network in the CRCat catalog. However, Ids in the custom catalog are not expected to be stable. One should take care that there is no ambiguity when referring to any of your custom catalog networks.

As an example, if you add a network to the custom catalog that expands XXXII-39 by adding a sixth element, `Rs`, connected from node 1 to ground, you might add code in `@CRCat/FillCat_Custom.m` similar to:

```
index = index + 1;
if aLen  % XXXII-39-Rs.
  syms La Lb Rc Ld Cd Rs
  theCat.list(index).symV =          [ La  Lb  Rc  Ld  Cd  Rs ];
  theCat.list(index).nodes = uint32( [ 1 2 2 0 2 3 3 0 3 0 1 0 ] );
  theCat.list(index).name =  'XXXII-39-Rs';
end
```

Next, update your custom catalog with `aCustom=CRCat('custom')`. Then, when you bring up the default catalog, `aCat=CRCat`, your new custom catalog will be appended to the default catalog. Note that if MATLAB cannot solve the synthesis equations for this six-element network, you can try removing the sixth element from the solve list as follows:

```
index = index + 1;
if aLen  % XXXII-39+.
  syms La Lb Rc Ld Cd Rs
  theCat.list(index).symV =          [ La  Lb  Rc  Ld  Cd ];
  theCat.list(index).symVn =         [ La  Lb  Rc  Ld  Cd  Rs ];
  theCat.list(index).nodes = uint32( [ 1 2 2 0 2 3 3 0 3 0 1 0] );
  theCat.list(index).name =  'XXXII-39+';
end
```

Now, MATLAB will attempt to find synthesis equations for only the first five elements and those synthesis equations will depend on you specifying a numerical value for the sixth element when needed. An example is given below in the section titled "Specifying and Using Fixed Elements".

Prior to allocating memory for the `list` field, `CRCat` must count the number of networks. That is done by the `index` variable. Then the network is set up in the `list` field after the entire `list` field has been allocated as indicated by the `aLen` variable.

Field `sameId` is an array of the Ids (indices) of all networks that have the same rational polynomial signature (`flagSig`) as this network.

Field `symSame` contains equations for conversion between networks that have the same signature. Networks can have the same signature and not be convertible between each other, in which case an appropriate text is stored indicating why they cannot be converted. Relations between networks can be 1) Equivalent, 2) Degenerate, 3) Regenerate, 4) No relationship as detailed in my CR paper. Regenerate networks were a big surprise. Details on the text descriptions start at line 292 in CRCat.m. Relationships between networks can be found by, for example:

```
>> aCat.Relationship(500,243)
ans =
    'NoXfrm'
>>
```

You may list all the networks that have transform equations with XXXII-39:

```
>> aCat.NetworkRelations(500)
500-> 239 Equiv    413 Equiv    500 Self    624 Equiv    625 Equiv
626 Equiv    627 Equiv
```

Fields `symZeros` and `symPoles` provide symbolic equations for the zeros and poles of the network admittance transfer function as a function of the symbolic `symVn` element variables.

## Handling Measured Data

MATLAB code for handling microwave measured data is found in `SnP.m`. To start, construct an empty `SnP` variable:

```
>> aData = SnP
aData =
  SnP with properties:
    fUnit: ''
    fMult: 0
     freq: []
    nPort: 0
    pType: ''
    dType: ''
    rNorm: 0
    xNorm: 0
      mat: []
>>
```

The fields are described in comments at the beginning of `SnP.m`:

```
properties (SetAccess = protected)
    fUnit;  % Frequency unit text string.
    fMult;  % Multiplier to convert fUnit to Hz.
    freq;   % Vector of frequencies in fUnit units.
    nPort;  % Number of ports.
    pType;  % String indicating parameter type: 'S', 'Y', or 'Z'.
    dType;  % string indicating data type: real/imag ('RI')
            % mag/angle(deg) ('MA') or db/angle(deg) ('DB'). This is
            % the data format in the file. In the SnP.mat member, it
            % is always stored as complex numbers, real and imaginary.
    rNorm;  % R normalization impedance.
    xNorm;  % X normalization impedance, not yet supported.
    mat;    % Data in 3D matrix, first two indices are row and col of
            % n port data and third index is the frequency index.
  end % properties
```

To read in a microwave data file, see `help SnP.Get`:

```
>> aData.Get('CRFitStressTest/hairpin_InBand_Nf251.s2p')
>> aData
aData =
  SnP with properties:
    fUnit: 'GHz'
    fMult: 1.0000e+09
     freq: [3.9500 3.9510 3.9520 3.9530 3.9540 3.9550 … ] (1×251
double)
    nPort: 2
    pType: 'S'
    dType: 'RI'
    rNorm: 50
    xNorm: 0
      mat: [2×2×251 double]
 >>
```

You can also set the data of an existing `SnP` structure with data already in MATLAB using `SnP.Fill`, as described in `help SnP.Fill`.

Conversion of the data (`mat`) between different parameter types (`pType`) are realized with `SnP.Spar`, `SnP.Ypar`, and `SnP.Zpar`.

Function `SnP.AddNoise` is useful for adding noise to noise-free data when checking the performance of the least-squares fit under noisy conditions. For example results, see Figure 7 in my CR paper.

Function `SnP.Pull` pulls desired data from an `SnP` variable, which is useful for plotting results. For example:

```
>> [aS11,aDescr] = aData.Pull('S11')
aS11 =
  Columns 1 through 4
  -0.8402 + 0.5044i  -0.8354 + 0.5115i  -0.8304 + 0.5187i  -0.8253 +
0.5260i
  Columns 5 through 8
  -0.8200 + 0.5334i  -0.8146 + 0.5409i  -0.8089 + 0.5485i  -0.8030 +
0.5562i
  Columns 9 through 12
  -0.7969 + 0.5640i  -0.7906 + 0.5719i  -0.7841 + 0.5799i  -0.7773 +
0.5880i
  Columns 13 through 16
  -0.7703 + 0.5962i  -0.7630 + 0.6045i  -0.7554 + 0.6129i  -0.7475 +
0.6214i
  Columns 17 through 20
  -0.7393 + 0.6300i  -0.7308 + 0.6387i  -0.7220 + 0.6475i  -0.7128 +
0.6564i
…
aDescr =
    'S-Param port 1 to 1'
```

You can pull the actual parameter values, as above, with a capital letter in the character array passed to `SnP.Pull`. For example, `'Y11'` pulls the (1,1) element of the data at each frequency. Lower case, like `'y11'`, yields the admittance of the equivalent shunt-to-ground element of the pi-model. For example, the equivalent pi-model shunt to ground admittance based on 2-port Y-parameters is $y_{11} = Y_{11} - Y_{21}$.

## Components

In `CRCat`, components are networks with values assigned to the network elements. This is where we convert symbolic equations into numbers. If not already loaded, load the default catalog. Then we can construct a component using XXXII-39/CR500:

```
  >> aComp = CRComp(500,aCat)
  aComp =
    CRComp with properties:
       symY: (Rc + Lb*s + Ld*s + Cd*Lb*Ld*s^3 +
  Cd*Ld*Rc*s^2)/(s*(La*Rc + …
       symC: [a3    a2    a1    a0    b4    b3    b2    b1]
      symVn: [La    Lb    Rc    Ld    Cd]
      symEL: [1×5 sym]
      valEL: [4.3000 13 5.4000 27 0.0920]
      flagA: [1 1 1 1]
      flagB: [1 1 1 1 0]
       name: "XXXII-39"
      node1: 0
      node2: 0
>>
```

The fields are described at the top of `CRComp.m`:

```
      properties (SetAccess = protected)
          symY;  % Symbolic equation for entire rational polynomial
                 % in terms of element values.
          symC;  % Symbolic CR Poly coefficients, Ai and Bi.
          symVn; % Symbolic symbols for RLC lumped elements.
          symEL; % Symbolic equations to go from fitted A,B
                 % coefficients to lumped element values.
          valEL; % Values of the RLCs. If empty, ignore this component
                 % during Eval.
          flagA; % Numerator signature.
          flagB; % Denominator signature.
          name;  % Name of component (use aCat.Name2Id to get index
                 % in CRCat catalog).
          node1; % First node between which component will be
                 % connected.
          node2; % Second node between which component will be
                 % connected.
```

Fields `node1` and `node2` are not yet used. These fields will accommodate a future capability to connect components into a nodal network.

Field `symY` is `symA`/`symB` from the network data in the catalog. In other words, it is a symbolic equation for the overall admittance of the component.

Fields `symC`, `symVn`, `symEL`, `flagA`, `flagB`, and `name` are all the same as in `CRCat`. They are included in the component when it is set so that there is no further need to explicitly reference the catalog for information about this component.

The `valEL` field contains the values for the RLC elements in the component. It defaults to the `testEL` values from the catalog.

The frequency response of a component is evaluated using `CRComp.Eval`:

```
>> aFreq = [1:4]
aFreq =
     1     2     3     4
>> aY = aComp.Eval(aFreq)
aY =
   0.0082 - 0.0373i   0.0019 - 0.0186i   0.0008 - 0.0124i   0.0005 -
0.0093i
>>
```

The frequencies and component values can be viewed as being in any consistent units, for example: Radians/sec H, F, Ohms; or Giga-radians/sec, nH, nF, Ohms, etc. The symbolic equation for the admittance of all components in the catalogs have been checked for correctness by evaluating, as above, using the `testEL` values and comparing to the result using the same element values, but with the admittance calculated by a nodal analysis, `CRComp.Nodal`. You can set the component element values manually with `CRComp.SetEL`.

We can convert components between equivalent networks. Check once more to see what networks are equivalent to XXXII-39, CR500:

```
>> aCat.NetworkRelations(500)
500-> 239 Equiv    413 Equiv    500 Self    624 Equiv    625 Equiv
626 Equiv    627 Equiv
```

Let's convert our XXXII-39 component to the equivalent component CR413:

```
>> aCompNew = aComp.Transform(413,aCat)
CRComp with properties:
     symY: (Rb + La*s + Lc*s + Cd*La*Lc*s^3 + Cd*La*Ld*s^3 +
Cd*Lc*Ld*s^3 + Cd*Lc*Rb*s^2 …
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Rb    Lc    Ld    Cd]
    symEL: [1×5 sym]
    valEL: [17.3000 9.5631 53.5379 6.4071 0.0414]
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXI-33"
    node1: 0
    node2: 0
>>
```

To check this result, we can convert our new XXXI-33 back to XXXII-39. `CRComp.Transform` can use either Id or network name, and single or double quotes are accepted. The catalog must be provided as well because `CRComp` does not store the transform equations.

```
>> aCompNew.Transform("XXXII-39",aCat)
ans =
  CRComp with properties:
     symY: (Rc + Lb*s + Ld*s + Cd*Lb*Ld*s^3 +
Cd*Ld*Rc*s^2)/(s*(La*Rc + Lb*Rc + La*Lb*s +…
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Lb    Rc    Ld    Cd]
    symEL: [1×5 sym]
    valEL: [4.3000 13 5.4000 27 0.0920]
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXII-39"
    node1: 0
    node2: 0
```

If you evaluate the admittance of each component with `CRComp.Eval`, you will find that they are identical at all frequencies.

You can perform a least squares fit using `CRComp.Fit`. To demonstrate, we next load a data set generated by a netlist (in Sonnet®) of XXXII-39. That data is loaded into the `SnP` structure that we created in the previous section, `aData`, using `aData.Get` and passing the name of the file to be loaded, in this case: `'CRFitStressTest/XXXII-39.y1p'`. Then `CRComp.Pull` the `y11` data out. We can then use that data to fit our XXXII-39 network to this data and we recover the original component values.

```
>> aData.Get('CRFitStressTest/XXXII-39.y1p')
>> aY = aData.Pull('y11');
>> aRMSError = aComp.Fit(aY,aData.freq)
aRMSError =
   3.1280e-13
>> aComp
aComp =
  CRComp with properties:
     symY: (Rc + Lb*s + Ld*s + Cd*Lb*Ld*s^3 +
Cd*Ld*Rc*s^2)/(s*(La*Rc + …
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Lb    Rc    Ld    Cd]
    symEL: [1×5 sym]
    valEL: [0.5000 0.4000 10.0000 0.2000 0.0030]
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXII-39"
    node1: 0
    node2: 0
>>
```

From the Sonnet netlist, those are the exact values of the Sonnet nodal circuit used to create the data that we fitted:

| Element | Nodes | Description |
|---|---|---|
| Net | 1 | Main Network |
| IND | 1 2 | L=0.5 |
| CAP | 3 | C=3.0 |
| IND | 3 | L=0.2 |
| RES | 2 3 | R=10.0 |
| IND | 2 | L=0.4 |

If there are elements specified in `aComp.symVn` that do not have synthesis equations (i.e., the `symV` list in the catalog is shorter than the `symVn` list), you must provide values for the elements that cannot be fitted because they have no synthesis equations. This is done in the last, optional parameter for `CRComp.Fit`. See `help CRComp.Fit` for details. An example is detailed in the section below titled ' Specifying and Using Fixed Elements'.

On occasion, especially if a component has been assigned negative element values, it is important to determine if the real part of the component admittance transfer function is negative at any frequency. This cannot be rigorously realized by numerically evaluating the transfer function at a finite set of frequencies as a negative real part could be missed. Rather, we evaluate the transfer function symbolically with the specific element values assigned to the component. Then a symbolic expression for the real part is formed and solved for frequencies at which it is zero valued. In addition, the derivative of the real part with respect to frequency is evaluated and if it is also zero, then, while the real part touches zero, it does not cross zero. If there are any zero crossings, or if there are no zero crossings but the real part of the transfer function is negative at high frequency, then the component is not passive.

As an example:

```
>> aZeros = aComp.IsGain
aZeros =
     []
```

When `CRComp.IsGain` returns an empty array, the real part of the component admittance transfer function is greater than or equal to zero at all frequencies. Otherwise, the function returns an array listing all zero-crossing radian frequencies. For the example above, at radian frequency 0.6345…, the real part is zero, but the derivative is also zero. Checking the derivative is important for a rigorous result.

Sometimes it is useful to convert a network name to its index (i.e., Id) in the catalog list. For example:

```
>> aCat.Id2Name(500)
ans =
    "XXXII-39"
>> aCat.Id2Name(ans)
ans =
  uint32
   500
>>
```

CRCat.Id2Name works in both directions. In addition, the parameter can be either a string array of network names or an integer array of network Ids.

## Specifying and Using Fixed Elements

For these examples, we use the custom catalog by itself:

```
>> aCustom = CRCat('custom')
Loading custom catalog.
   Checking to see if stored custom catalog is up to date.
   Stored custom catalog is loaded and up to date.
aCustom =
  CRCat with properties:
      fUnit: 'GHz'
      fMult: 1.0000e+09
    resUnit: 'Ω'
    indUnit: 'nH'
    capUnit: 'nF'
       list: [1×7 struct]
```

If you made any changes to the custom catalog (in the file `@CRCat/FillCatCustom.m`), the updated file will be saved for future use of `CRCat`. The custom catalog is much smaller than the default catalog. Thus, when you can use it for your work, you will find that response time is faster.

With the custom catalog loaded, we work with the first two networks. The pertinent code in `FillCatCustom.m` is:

```
index = index + 1;
if aLen  % An L in series with (an L in parallel with an R in series
with a parallel LC):  XXXII-39.
   syms La Lb Rc Ld Cd
   theCat.list(index).symV =          [ La  Lb  Rc  Ld  Cd ];
   theCat.list(index).nodes = uint32( [ 1 2 2 0 2 3 3 0 3 0 ] );
   theCat.list(index).name =  'XXXII-39_original';
end

index = index + 1;
```

```
if aLen  % An L in series with (an L in parallel with an R in series
with a parallel LC):  XXXII-39.
   syms La Lb Rc Ld Cdd
   theCat.list(index).symV =          [ La  Lb  Rc  Ld ];
   theCat.list(index).symVn =         [ La  Lb  Rc  Ld  Cdd ];
   theCat.list(index).nodes = uint32( [ 1 2 2 0 2 3 3 0 3 0 ] );
   theCat.list(index).name =  'XXXII-39_FixedCdd';
end
```

To remove Cd from being fitted, we specify the member symVn and leave Cdd out of symV in the MATLAB code above. Only elements in symV have synthesis equations solved for them and thus only those elements can be fitted.

Next, we set up two components, with important portions highlighted below. Within the component data structure, we see that the second component, aComp_FixedCd, has Cdd fixed because there are only four symEL synthesis equations, which allows synthesis (from least squares fitted rational polynomial coefficients) of only the first four elements in the component. There is no synthesis equation for Cdd. In fact, the synthesis equations for each of the first four elements are a function of Cdd, in addition to the rational polynomial coefficients. This means the synthesis equations provide no value for Cdd and the 'values' for the other four elements are all functions of Cdd.

To illustrate the use of fixed element values, we first evaluate the admittance of aComp and then use that as though it is measured data to which we will fit the aComp_FixedCdd. To be sure that the fitting has indeed found the correct element values, we create the aComp_FixedCdd component, and then clear the element values.

Below, we create aComp and evaluate its admittance using the (highlighted) default element values followed by creation of the aComp_FixedCdd and changing its (highlighted) element values to an empty array:

```
>> aComp = CRComp(1,aCustom)
aComp =
  CRComp with properties:
     symY: (Rc + Lb*s + Ld*s + Cd*Lb*Ld*s^3 + …
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Lb    Rc    Ld    Cd]
    symEL: [1×5 sym]
    valEL: [4.3000 13 5.4000 27 0.0920]
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXII-39_original"
    node1: 0
    node2: 0

>> aComp_FixedCdd = CRComp(2,aCustom);
>> aComp_FixedCdd.SetEL([])
```

```
aComp_FixedCd =
  CRComp with properties:
     symY: (Rc + Lb*s + Ld*s + Cdd*Lb*Ld*s^3 + …
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Lb    Rc    Ld    Cdd]
    symEL: [1×4 sym]
    valEL: []
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXII-39_FixedCdd"
    node1: 0
    node2: 0
```

We could also do both steps above with one function call:

```
aComp_FixedCdd = CRComp(2,aCustom,[]);
```

Next, we set up frequencies for analysis and analyze `aComp` to generate our 'measured' data.

```
>> aF = [1:20];
>> aY = aComp.Eval(aF)
aY =
  Columns 1 through 4
   0.0082 - 0.0373i   0.0019 - 0.0186i   0.0008 - 0.0124i …
   …
```

Finally, we fit `aComp_FixedCdd` to the admittance we just evaluated from `aComp` to see if we can recover the original element values. Since there is no synthesis equation for `Cdd`, we must provide the fitting routine with a value. In this case, let's use the value that we already know gives the best fit to the 'measured' data: 0.092. The fitting routine returns the RSS error, which, in this case, indicates an essentially perfect fit.

```
>> aRSSError = aComp_FixedCdd.Fit(aY,aF,0.092)
aRSSError =
   2.3910e-09

>> aComp_FixedCdd
aComp_FixedCdd =  CRComp with properties:
     symY: (Rc + Lb*s + Ld*s + Cdd*Lb*Ld*s^3 + …
     symC: [a3    a2    a1    a0    b4    b3    b2    b1]
    symVn: [La    Lb    Rc    Ld    Cdd]
    symEL: [1×4 sym]
    valEL: [4.3000 13.0000 5.4000 27.0000 0.0920]
    flagA: [1 1 1 1]
    flagB: [1 1 1 1 0]
     name: "XXXII-39_FixedCdd"
    node1: 0
    node2: 0
```

We see, above, that the four fitted element values have been recovered exactly, given the correct optimal value of the fifth element.

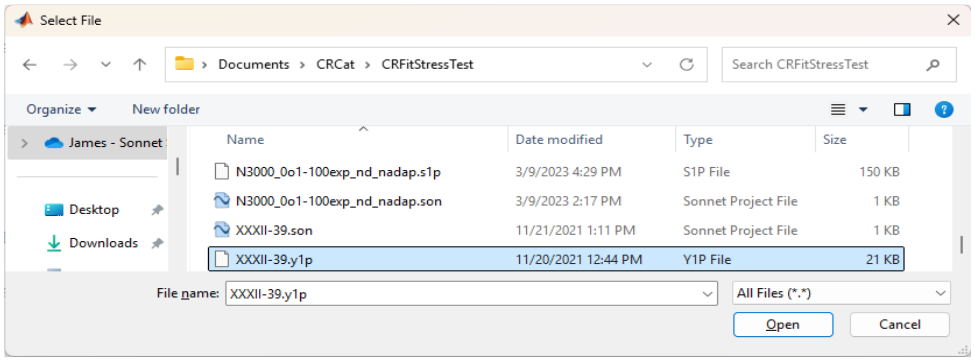
This capability is useful when it is desired to perform fits for networks for which MATLAB cannot solve for all the required synthesis equations. Proceed by introducing fixed elements into the network until MATAB is able to find the synthesis equations for the remaining elements. Perhaps some element values can be determined by additional measurements. Alternatively, the fixed elements can be varied using classical optimization algorithms. For each trial value of `Cdd`, the optimum values for the remaining elements are determined by the synthesis equations. No additional least squares fitting is needed. `Cdd` can then be tuned/optimized for minimum RSS fitting error. Thus, for this example, we reduce a challenging optimization of multiple variables to an optimization of just one variable.

## Extracting Components from Measured Data

On occasion, while we might have some good guesses, we do not know what the best model is for a given instance of measured data. `CRCat` can search through all 623 networks in the base catalog and all networks in the custom catalog and, among those networks, find the best network(s) to fit your data. If your data can be modeled accurately with five or fewer elements, `CRCat` can find that component.

To illustrate, rather than typing out the file name for our 'measured' data, let's use a MATLAB provided GUI that is wrapped up in a `CRCat` m-file:

```
>> aFileName = GetFileName
```




```
aFileName =
    'C:\Documents\CRCat\CRFitStressTest\XXXII-39.y1p'
```

Starting the search with function `CRSeed` with only a file name (or an `SnP` variable loaded with that data) and a catalog name results in an attempt to fit each of the networks in that catalog to that data. Since results are displayed for each attempt, output is extensive. Not all networks can be fitted. For example, it is pointless to fit the data to a network that has no synthesis equations because there is no direct way to go from the fitted rational polynomial coefficients to the corresponding element values, a

situation that is flagged in the CRSeed output. Such networks are likely degenerate, however in some cases, it might be that the MATLAB solve function could not find the synthesis equations. A useful area for future research would be to creatively find such solutions. In addition, in rare cases, it is possible that the matrix to be inverted is singular. This is indicated by a message indicating CRFit error code = −3. A portion of the CRSeed output follows:

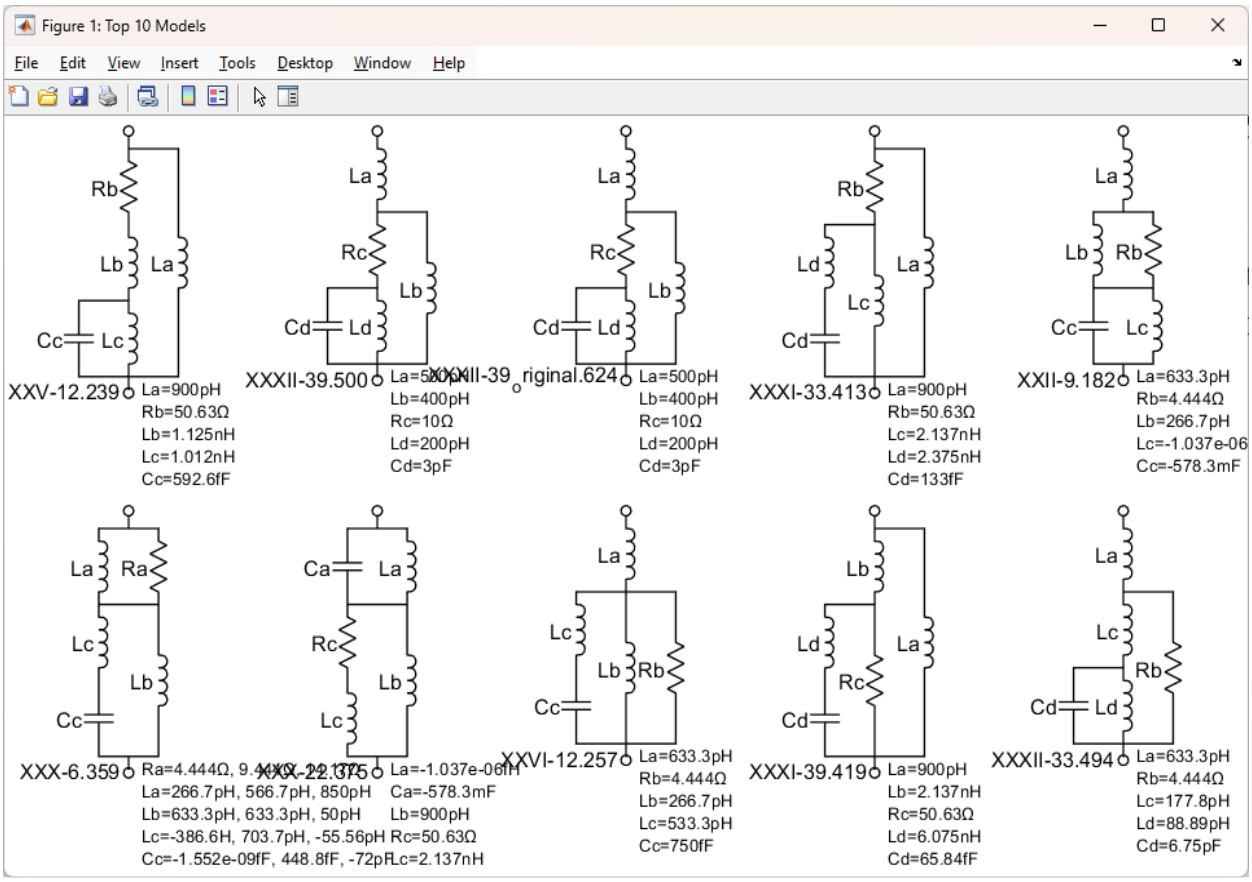
```
>> aComp = CRSeed(aFileName,aCat)
****Fitting Pi admit port 1 to gnd****
File name: C:\Documents\CRCat\CRFitStressTest\XXXII-39.y1p

Id = 1, Name = R,  RSS(y) Error = 0.190120
Component variables:    R
Nodal connection list: 1 0
Fitted RLC Values:   95.17Ω

Id = 2, Name = L,  RSS(y) Error = 0.015825
Component variables:    L
Nodal connection list: 1 0
Fitted RLC Values:   861.5pH
.
.
.
Id = 484, Name = XXXII-23,  RSS(y) Error = 0.033053
Component variables:    Ra    Cb    Lc    Rd    Cd
Nodal connection list: 1 2   2 0   2 3   3 0   3 0
Fitted RLC Values:   -753.5mΩ 511.5fF 804.6pH 819.5mΩ 9.141pF
.
.
.
Id = 499, Name = XXXII-38,  RSS(y) Error = 0.025049
Component variables:    La    Lb    Rc    Rd    Cd
Nodal connection list: 1 2   2 0   2 3   3 0   3 0
Fitted RLC Values:   717.1pH 81.68pH 56.97mΩ 37.34Ω 4.444pF

Id = 500, Name = XXXII-39,  RSS(y) Error = 0.000000
Component variables:    La    Lb    Rc    Ld    Cd
Nodal connection list: 1 2   2 0   2 3   3 0   3 0
Fitted RLC Values:   500pH 400pH 10Ω 200pH 3pF

Id = 501, Name = XXXII-40, No synthesis equations, likely degenerate
network.
.
.
.
```

When complete, CRSeed lists the 10 best fitting components and the fitted element values on a plot of their schematics. Additionally, a list of the 100 best fitting components (passive models indicated with an

asterisk), followed by a list of 20 best fitting components whose transfer function have no negative real parts, i.e., they are passive. Finally, it asks if you want to plot any of the results. `CRSeed` returns a `CRComp` data structure of the last component that you plotted.

```
Up to the 100 Best Fits (ID-Solution, RSS(Pi admit port 1 to gnd)
Error)
File Name: C:\Documents\CRCat\CRFitStressTest\XXXII-39.y1p

 (239* 3.13e-13) (500* 3.13e-13) (624* 3.13e-13) (413* 3.13e-13)
(182* 0.0103) (359* 0.0103) (375* 0.0103) (257* 0.0118) (419*
0.0118) (494* 0.0118)

 (4* 0.0118) (260  0.0144) (497  0.0144) (509  0.0144) (511  0.0144)
(2* 0.0158) (505  0.0183) (573* 0.0196) (604* 0.0196) (332* 0.0197)

 (348* 0.0197) (164* 0.0197) (261* 0.0198) (423* 0.0198) (510*
0.0198) (10* 0.02) (42* 0.0216) (48* 0.0216) (49* 0.0216) (75*
0.0216)
.
.
.

Up to the 20 Best Passive Fits (ID-Solution, RSS(Pi admit port 1 to
gnd) Error)
File Name: C:\Documents\CRCat\CRFitStressTest\XXXII-39.y1p

 (239* 3.13e-13) (500* 3.13e-13) (624* 3.13e-13) (413* 3.13e-13)
(182* 0.0103) (359* 0.0103) (375* 0.0103) (257* 0.0118) (419*
0.0118) (494* 0.0118)

 (4* 0.0118) (2* 0.0158) (573* 0.0196) (604* 0.0196) (332* 0.0197)
(348* 0.0197) (164* 0.0197) (261* 0.0198) (423* 0.0198) (510*
0.0198)

Enter component id for plotting (return to end): 500
Enter component id for next plot (return to end):
```

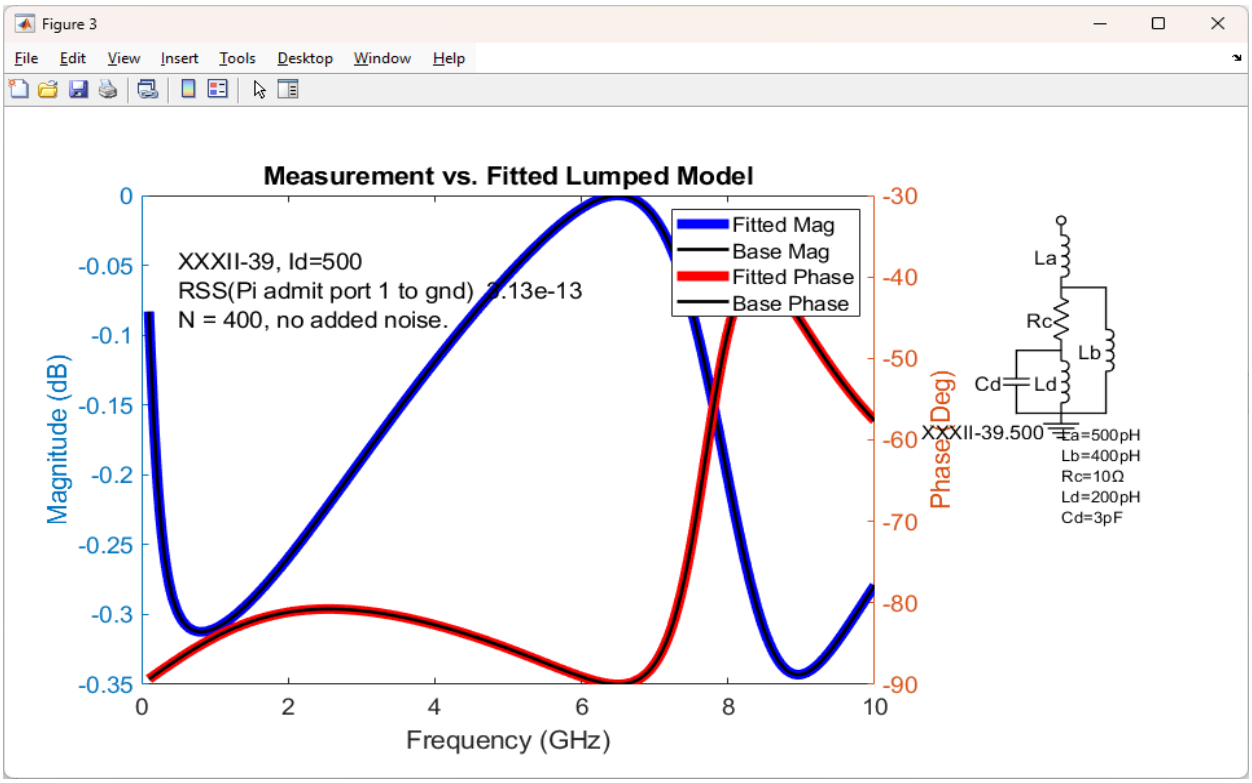
Note that the first four circuits are both equivalent and passive. Note that the third component is from the `custom` catalog, which, except for name, is identical to the second displayed component. You can use `aCat.DrawArray(aCompArray)` where `aCompArray` is an array of `CRComp` components. Each component is listed with a schematic labeled with the name followed by the Id (i.e., its index in `aCat.list`). You can click and drag the separate components and labels around to eliminate overlap, and you can copy and paste them (use the `Edit->Copy Figure` option) as desired into various other applications. All MATLAB plots can be saved for later reference. The first 10 passive components are displayed by line 253 in `CRSeedAddNoise.m` (no noise was added in this case) with a result shown below.




If the schematic used for a component (which is determined by the roman numeral in the network name) is for a network not in the base catalog, you will need to add your own coordinates to locate network nodes for drawing. This is done by adding a case section in the switch statement located in lines 1467 – 1489 in `CRCat.m`.

The bare schematic for a single component can be plotted `aCat.Draw(aName)`, where `aName` can be the network name or Id. The function starts on line 1302. Alternatively, you can call `aCat.DrawArray` for a more elaborate schematic.

The plot, below, of the fitted data compared to the original, noise-free data shows that both are identical. `CRSeedAddNoise` lines 264 – 350 plot the result shown below.




The above plot compares noise-free 'measured' data to the fitted result. If you want to test the performance of fitting to noisy data, you can add noise to noise-free 'measured' data by using `CRSeedAddNoise` instead of `CRSeed`. Figure 7 in my CR paper was generated in this manner. Figure 10 in my CR paper shows a `CRSeed` fit to actual noisy measured data.

## Extracting Multiple Components from Measured Data

Here I describe the results of my preliminary work exploring the challenge of multiple component extraction. I do not consider the present state of this software adequate for main-stream application. However, this work does suggest some future directions that might prove productive. I leave this challenge to future researchers should they find it interesting.

A simple case of multiple component extraction that does see widespread application is Vector Fitting. This technique, in net effect, estimates a partial fractions representation of the complex rational transfer function and fits it, term by term, to one of several lumped models. Network X-2 in Figure 12 of my CR paper is the network usually used to represent a pair of complex poles, although any of the eight networks in that figure could be used as they are all equivalent. When all partial fractions terms have been modeled, they are all connected in parallel to model the measured data.

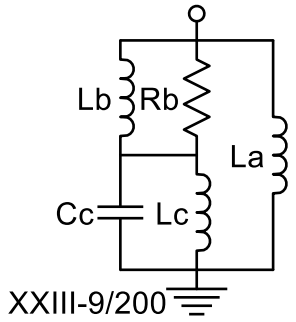
CRSprout routine expands upon this strategy. The circuit model developed by CRSprout is built by successively connecting individual components using some combination of series and shunt connections. An understanding of CRSprout operation can be realized by starting at the end. Assume we have

completed an array of components that model the measured data, `aSprout`. A final call evaluates the result of the transfer function of the entire array:

```
aModeledY = CRSprout(aFreq,aSprout);
```

The admittance of the first component in the array, `aSprout`, is evaluated. The admittance of the second component in the array is then evaluated and added to the admittance of the first component. This connects the two in shunt (i.e., in parallel). Next, the evaluated admittance of the first two components (which are connected in shunt) is inverted and added to the impedance of the third component. This connects the third component in series with the (shunt-connected) first two components. The evaluated impedance of the first three components is inverted and added to the admittance of the fourth component, and so on.

The result is a sequence of components connected in series and parallel to build the entire model. To illustrate, load file `CRSproutTest.m`. Go to test 5, starting on line 136. Here we initialize an array of components so that `CRSprout` will build an XXIII-9 component. After loading the default catalog, `aCat = CRCat`, the result of `aCat.DrawArray(aCat.Name2Id('XXIII-9'))` shows the schematic of the network we are going to build. To copy and paste a MATLAB plot/figure into a document, in the MATLAB plot select Edit->Copy Figure:




To demonstrate using the seed/sprout array of components to build a larger component, test 5 in `CRSproutTest.m` builds XXIII-9 from simple R's, L's, and C's. The test routine starts by creating a component with the complete `CRCat` XXIII-9 network and setting some test values:

```
aFreq = 1:10;
% [        La,       Rb,       Lb,       Lc,       Cc ]
aValEL = [ 1.0000    2.0000    0.3000    4.0000    0.0500 ];
aName = 'XXIII-9';
aCompBase = CRComp( aName,theCat );
aCompBase.SetEL( aValEL );
aOriginalY = aCompBase.Eval( aFreq );
```

The aOriginalY array stores the correct admittance of the network with elements set to the indicated values. Next, the code sets up an array of components that will be used by CRSprout to build XXIII-9 piece-by-piece:

```
% Set Sprout so it sets up an XXIII-9 piece by piece.
clear aSprout; % Clear out any previous data.

% inductor La connected in parallel with rest of circuit, below.
aSprout(4) = CRComp('L',theCat);
aSprout(4).SetEL( aValEL( 1 ) );

% parallel Lc Cc connected in series with parallel RL, below.
aSprout(3) = CRComp('ParLC',theCat);
aSprout(3).SetEL( aValEL( 4:5 ) );

% resistor Rb conected in parallel with Lb, below.
aSprout(2) = CRComp('R',theCat);
aSprout(2).SetEL( aValEL( 2 ) );

% inductor Lb;
aSprout(1) = CRComp('L',theCat);
aSprout(1).SetEL( aValEL( 3 ) );
```

The test then evaluates the RSS difference between what CRSprout has just calculated for its admittance and the known correct answer, aOriginalY.

```
aSproutY = CRSprout(aFreq,aSprout);
aError1 = rmse( aSproutY,aOriginalY );
```

This evaluation is performed in line 31 – 43 of CRSprout.m, where the admittance of the first two components are added together. This connects the first (Lb) and second component (Rb) in parallel, as is the case in the XXIII-9 schematic above.

Next, that resulting admittance is inverted and added to the inverted admittance of the third component, a ParLC, representing Lc and Cc. This connects our initially calculated parallel RL in series with the newly evaluated parallel LC, again, as in the schematic above. That result is inverted once more, so that the admittance of the entire component so far is stored.

Finally, the admittance of the fourth component, representing La, is evaluated and added to the admittance of the above result. This connects the inductor in parallel with the rest of the component, completing the circuit. CRSproutTest then evaluates the difference. If CRSprout is working correctly, the difference will be zero to within numerical precision.

The general rule is that all components with odd indices are connected in series with the entire prior circuit up to that point, and all components with even indices are connected in parallel with the entire

prior circuit. The first component has no prior net circuit, so it remains unchanged. The prior circuit for the second component is just the first component, so it is connected in parallel with the first component. The third component is connected in series with the parallel combination of the first two components, etc. An empty component (i.e., the `valEL` field is empty) is treated as a short circuit if it is to be connected in series, and an open circuit if it is to be connected in parallel, i.e., empty components result in connection of components that have no effect on the final result. This is useful, say, if you want to connect multiple components in series or in parallel. To create an empty component, use `aCompEmpty = CRComp`.

The potential of `CRSprout` becomes apparent when we add a third parameter:

```
aModifiedY = CRSprout(aFreq,aSprout,aOriginalY);
aError2 = rms( aModifiedY );
```

After having evaluated the admittance of the `aSprout` array of components, `CRSprout` subtracts that admittance from the `aOriginalY` admittance that had been passed to it. That admittance is returned in `aModifiedY`, above. When `aSprout` exactly models `aOriginalY`, as is the case here, `aModifiedY` returns all zeros and the `aSprout` model is complete. To view the admittance generated by the `aSprout` model, call `CRSprout` as above, but without the `aOriginalY` parameter.

When the `aSprout` array evaluation does not exactly model the `aOriginalY` admittance, we can see the true power of `CRSprout`. Take the non-zero `aModifiedY` and perform a `CRSeed` extraction, i.e., find another component that best models the left-over admittance, `aModifiedY`. Now, we are faced with a binary choice: Place that component in the next empty `CRSprout` array element with an even index to connect it in parallel, or in the next array element with an odd index to connect it in series with the rest of the network. This should bring the new `aModifiedY` closer to zero. When the `aModifiedY` result is sufficiently close to zero, we have our model in `CRSprout`.

So what could possibly go wrong? Quite a few things. For example, if a wide band result is needed, the `aModifiedY` error could start increasing at high frequency, perhaps drastically, with each added component. A possible solution might be to extract the initial component model from only low frequency data. Then gradually increase the upper frequency limit as additional components, each principally responsible for successively higher frequencies, are extracted from the left-over `aModifiedY`. There is an option in `CRSeed` that facilitates this process, see the comments at the top of `CRSeed.m`.

Another, as yet, unsolved problem derives from the fact that, at any given stage, if we extract one optimal component to model the left-over `aModifiedY`, we have two choices on how to proceed, i.e., should we connect it in series or in parallel. More likely there will be multiple acceptable choices for the next component. If all the acceptable components are equivalent (see section X in my CR paper, Network Relationships), we can completely explore the solution space by selecting only one of the equivalent networks for the next component. If there are multiple non-equivalent networks, then a complete search should evaluate all potential non-equivalent candidates. In cases where a complete model might

require dozens of components, with multiple choices at each stage for each component, a complete search rapidly becomes impractical.

Another obvious avenue for research is to extend `CRCat` to include all six-element networks. The number of such networks will be extremely large. Thus, it will be necessary to make filling the data structures (in `CRCat.list`), and their evaluation when extracting components, much faster. MATLAB required a few days of computer time on my notebook computer to fill the existing list. I would not be surprised if it requires several weeks or even months of computer time to fill a six-element network list. It will also be necessary for everyone to agree on a naming convention for these new `CRCat` networks. We do not want to have different researchers referring to the same networks by different names.

So, these are problems I leave for others, if they are interested, to explore, assuming solutions are even possible.

## Fitting Rational Polynomials to Measured Data

Sometimes, a fitted rational polynomial is the desired final result for measured data especially if the data is wideband. Especially in wide-band cases, the fitted rational polynomial will likely require a large number of terms. Figure 1 and 2 in my CR paper illustrate such a result that uses 1200 polynomial terms, split evenly between the numerator and denominator. There are also numerous examples in the `CRFitStressTest` folder of this software download. Exploring the data sets in this folder will reveal both successful fits and failed fits.

The purpose of `CRFitStressTest` is to explore the limits of the complex rational polynomial fitting routine `CRFit` for large problems. By looking at the results for the various test cases, we can see how to tell when there are not enough polynomial terms and when there is not enough numerical precision. We can also see how these selections affect analysis time. Be aware, however, that analysis times for problems that seem as though they should be the same are sometimes considerably different, so treat timing results with appropriate skepticism.

If the match between measured data and the result from the fitted rational polynomial is good (and you decide what "good" is), our fitting is complete. If it is not good, then we need to know if the problem is the number of terms used in the polynomial or in the number of digits used in the fitting arithmetic. In this case, perform a Fit 2, as described in my CR paper in section IV, *Stress Testing*. To perform a Fit 2, evaluate the rational polynomial that resulted from Fit 1 at the same frequencies as the measured data used for Fit 1. Then perform a fit to that result. The result of this second fitting is Fit 2. If the Fit 2 result is the same as the (bad) Fit 1 result, we are using sufficient numerical precision. We need more terms in the rational polynomial. If the Fit 2 result does not match the Fit 1 result, we need more numerical precision…and possibly more polynomial terms too.

The file `CRFitStressTest.m` in the folder `CRFitStressTest` holds many examples. You will find most of the file is commented out (i.e., a '%' sign is the first character). Many of the tests take a long

time, so it is useful to only run a portion of the file at a time, with the rest commented out. Commenting and un-commenting sections is quickly effected by selecting the desired section of the file and typing control-r (to comment) or control-t (to un-comment) the selected section.

Each section of the routine starts by loading a test data set, for example:

```
aName = 'N100_0o01-10exp_nd_nadap.s1p'; % 100 points 0.1 - 10 GHz,
   nd, no adap.
aDat = SnP; % Allocate a variable to hold the n-port data.
aDat.Get(aName);
fprintf('\n *** File = %s\n',aName);
fprintf(' *** %d freqs, %f-%f %s, no debed, no adaptive
   sweep.\n',length(aDat.freq),aDat.freq(1),aDat.freq(end),aDat.fUn
   t );
```

The line `aDat.Get(aName)` loads the desired data file. The next two lines display useful information.

Immediately following this, the stress test executes its first test:

```
digits(32);
CRStressTestBody( 'Short Stub',5,5,aDat,'y11',[5.996187e-04,
   5.840103e-21],3); % 0.6 0.4s/fit
% Numerator: Number of zeros with positive real parts set to zero
   = 1.
% Denominator: Number of zeros with positive real parts set to zero
   = 0.
% Cleaned Result Error (Fit 2) = 1.452538e-07
```

The call to `digits(32)` sets all vpa (variable precision arithmetic) precision to 32 digits. When we want to change the precision used, we change the argument to the `digits` call.

Next is the call to `CRStressTestBody`. The seven parameters are:

1) `'Short Stub'` : A descriptive name for the test.

2&3) `5,5` : The number of terms in the numerator, followed by the number of terms in the denominator. These two numbers are almost always equal. When we need to change the number of polynomial terms, these are the numbers we change.

4) `aDat`: The data to be fitted is stored in this `SnP` variable. It was loaded in the previous section and stores an entire parameter (S, Y, or Z) matrix, as well as the frequencies for the data.

5) `'y11'`: This is the parameter that will be pulled out of the `aDat` matrix for fitting. This option pulls out the pi-model admittance from port 1 to ground. The data specified by this argument is what gets pulled out of `aDat` regardless of how it is stored in `aDat`. There are many other options. See the `Get` function in `Snp.m`.

6) `[5.996187e-04, 5.840103e-21]` This is the expected error of Fit 1 and Fit 2. If we run the test and the result is significantly different, a warning message displays.

7) `3:` This sets 'tasks' for `CRStressTestBody`. If set to 1, we will see a plot showing three curves: The data being fitted, and the resulting Fit 1 and Fit 2. If set to 2, the positive real parts of any poles and zeros are set to zero. Then, the 'clean' rational polynomial is re-formed and evaluated. If set to 3, both tasks 1 and 2 are performed, and a second plot appears with the Fit 2 data replaced by the cleaned rational polynomial result. Set to zero, or not present, and we have no plots and no pole/zero cleaning.

The code also includes comments about the expected results for each test. One example is the time required for each fitting. Another, if task 2 or 3 is specified, is the error of the cleaned result versus Fit 2. This information is displayed by `CRStressTestBody` when it is executed and was copied over manually into comments in the code. With this information, it can be seen if there is a radical change in, say, execution time or in fitting error.

If your need is for a rational polynomial model of measured data, especially if it is wide-band data, exploring the numerous cases in `CRStressTest` while running a test case and stepping through `CRStressTestBody` will be helpful. It is likely possible to obtain your desired rational polynomial result with simple modification of the code in this procedure. It is also informative to see what cases fail to fit well, and also to explore how fitting time varies with numerical precision, number of polynomial terms, and the number and distribution of data points over frequency.

## Going Forward

Having run Sonnet® Software for over four decades, I am now enjoying retirement. I certainly include doing a bit of research, playing with equations, publishing now and then as enjoyment. But it also includes spending time at our camp in the Adirondacks, kayaking, swimming, hiking, snow shoeing, preparing firewood, gardening, birding, playing cello, photography, amateur radio, and many other pastimes. So, after some 15 years spent on this complex rational modeling project, I do now plan to proceed in other directions.

My hope is that a few younger researchers will find these results to be of sufficient interest to continue, perhaps even to commercialize some aspect of it. And for those who make commercial use of this work or publish research in which these results are found to be useful, I do expect that they shall make appropriate acknowledgement, in return for my having made all of this material freely available. I will certainly be available to give advice, perhaps even solve a bug or two (which, in spite of my most diligent efforts, will certainly be found by creative users!) as we go on. This is why I have made the entire results of my research on this matter available for all to use as we see fit. If we are fortunate, and at least a few pursue this project, I hope we will all cooperate and share our developments and results openly and when we agree, to incorporate the best of the best in the primary distribution. It would be good to avoid, if possible, multiple incompatible versions of CRCat taking off on separate paths. Such a

development would dilute our effort. As someone famous once said, it might be best if we "Put all the wood behind one arrow."

J. Rautio
Big Moose, NY
29 August 2025