Poles/Zeros/Gain from Look-Up Tables (Prototyping)

BSM Oct 2020

In [1]:
```
import numpy as np
from scipy import signal

import vectfit

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
%matplotlib inline
```

In [2]:
```
# NIMS FILTERS
#   These values are from here:
#   http://service.iris.edu/fdsnws/station/1/query?net=EM&sta=IDA11&cha=
*&level=response&format=xml&nodata=404
#   ALSO can pull these from an sp file (gains are clearer in sp files)

# B fields, one filter:
zeros_Blowpass = []
poles_Blowpass = [-6.28319 + 10.8825j, -6.28319 - 10.8825j, -12.5664]
gain_Blowpass = 1984.31
NIMS_Magnetic_3PoleLowpass = \
    signal.ZerosPolesGain(zeros_Blowpass, poles_Blowpass, gain_Blowpass)

# E fields, two filters:
zeros_Ehighpass = [0.]
poles_Ehighpass = [-1.66667E-4]
gain_Ehighpass = 1.
NIMS_Electric_Highpass = \
    signal.ZerosPolesGain(zeros_Ehighpass, poles_Ehighpass,
                          gain_Ehighpass)

zeros_Elowpass = []
poles_Elowpass = [-3.88301 + 11.9519j, -3.88301 - 11.9519j,
                  -10.1662 + 7.38651j, -10.1662 - 7.38651j,
                  -12.5664]
gain_Elowpass = 313384.
NIMS_Electric_5PoleLowpass = \
    signal.ZerosPolesGain(zeros_Elowpass, poles_Elowpass, gain_Elowpass)
```

In [3]:
```
# ZEN COIL RESPONSE
# angular frequency, real, imaginary
zen_response = np.array(
    [[0.0006283185307179586, 0.00016610681887368792, 0.2785499504730251
],
     [0.0012566370614359172, 0.009607482581442478, 0.5968226757408328],
     [0.0031415926535897933, 0.0340338255267764, 1.1534980271851412],
     [0.006283185307179587, 0.10326125512223676, 2.345528075549423],
     [0.012566370614359173, 0.2694037321803452, 4.648009077571525],
     [0.024504422698000385, 0.8674625452867567, 9.250988152485366],
     [0.04900884539600077, 3.3996002487875288, 18.026842965656627],
     [0.09801769079200154, 12.288173305808309, 32.68634912660055],
     [0.19666370011472106, 35.77374451709642, 47.65907329465832],
     [0.39269908169872414, 68.80622670880321, 45.83081507358207],
     [0.7853981633974483, 89.45957701017464, 29.675744159172037],
     [1.5707963267948966, 96.71915794659112, 15.82688714795047],
     [3.141592653589793, 98.73350517756762, 7.627472850645089],
     [6.283185307179586, 99.24219816830174, 2.958293251509568],
     [12.566370614359172, 99.30832240704096, -0.2879949423266931],
     [25.132741228718345, 99.24383148030137, -3.7035126269049026],
     [50.26548245743669, 98.97856371276684, -8.952157997609012],
     [100.53096491487338, 97.62982613999657, -18.64457201832332],
     [201.06192982974676, 92.02668043950737, -37.226970582652726],
     [402.1238596594935, 69.87783035350839, -70.17219305236156],
     [804.247719318987, -13.629323102806389, -89.21119896277264],
     [1608.495438637974, -26.895281950721987, 4.2573962015260145],
     [3216.990877275948, -2.138888186949132, 2.88837534502174],
     [6433.981754551896, -0.09177331053666367, 0.4396128837660938]]
)
zen_angular_frequencies = zen_response[:, 0]
zen_period = 2. * np.pi / zen_angular_frequencies
zen_complex_response = zen_response[:, 1] + 1.j * zen_response[:, 2]
zen_amplitude_response = np.absolute(zen_complex_response)
zen_phase_response = np.angle(zen_complex_response, deg=True)
```

In [4]:
```python
def plot_response(w_obs=None, resp_obs=None, zpk_obs=None,
                  zpk_pred=None, w_values=None, xlim=None):

    fig = plt.figure(figsize=(14, 4))
    gs = GridSpec(2, 3)
    ax_amp = fig.add_subplot(gs[0, :2])
    ax_phs = fig.add_subplot(gs[1, :2])
    ax_pz = fig.add_subplot(gs[:, 2], aspect='equal')

    if w_obs is not None and resp_obs is not None:
        ax_amp.plot(2. * np.pi / w_obs, np.absolute(resp_obs),
                    color='tab:blue', linewidth=1.5, linestyle='-',
                    label='True')
        ax_phs.plot(2. * np.pi / w_obs, np.angle(resp_obs, deg=True),
                    color='tab:blue', linewidth=1.5, linestyle='-')
    elif zpk_obs is not None:
        w_obs, resp_obs = signal.freqresp(zpk_obs, w=w_values)
        ax_amp.plot(2. * np.pi / w_obs, np.absolute(resp_obs),
                    color='tab:blue',  linewidth=1.5, linestyle='-',
                    label='True')
        ax_phs.plot(2. * np.pi / w_obs, np.angle(resp_obs, deg=True),
                    color='tab:blue', linewidth=1.5, linestyle='-')
        ax_pz.scatter(np.real(zpk_obs.zeros), np.imag(zpk_obs.zeros),
                      s=75, marker='o', ec='tab:blue', fc='w',
                      label='True Zeros')
        ax_pz.scatter(np.real(zpk_obs.poles), np.imag(zpk_obs.poles),
                      s=75, marker='x', ec='tab:blue', fc='tab:blue',
                      label='True Poles')

    if zpk_pred is not None:
        w_pred, resp_pred = signal.freqresp(zpk_pred, w=w_values)
        ax_amp.plot(2. * np.pi / w_pred, np.absolute(resp_pred),
                    color='tab:red',linewidth=3, linestyle=':',
                    label='Fit')
        ax_phs.plot(2. * np.pi / w_pred, np.angle(resp_pred, deg=True),
                    color='tab:red', linewidth=3, linestyle=':')
        ax_pz.scatter(np.real(zpk_pred.zeros), np.imag(zpk_pred.zeros),
                      s=35, marker='o', ec='tab:red', fc='w',
                      label='Fit Zeros')
        ax_pz.scatter(np.real(zpk_pred.poles), np.imag(zpk_pred.poles),
                   s=35, marker='x', ec='tab:red', fc='tab:blue',
                   label='Fit Poles')

    if xlim is not None:
        ax_amp.set_xlim(xlim)
        ax_phs.set_xlim(xlim)

    ax_amp.set_xscale('log')
    ax_amp.set_yscale('log')
    ax_amp.set_ylabel('Amplitude Response')
    ax_amp.grid()
    ax_amp.legend()

    ax_phs.set_ylim([-180., 180.])
    ax_phs.set_xscale('log')
    ax_phs.set_ylabel('Phase Response')
    ax_phs.set_xlabel('Period (s)')
    ax_phs.grid()

    ax_pz.set_xlabel('Re(z)')
    ax_pz.set_ylabel('Im(z)')
    max_lim = max([abs(ax_pz.get_ylim()[0]), abs(ax_pz.get_ylim()[1]),
                   abs(ax_pz.get_xlim()[0]), abs(ax_pz.get_xlim()[0])])
    ax_pz.set_ylim([-1.25 * max_lim, 1.25 * max_lim])
    ax_pz.set_xlim([-1.25 * max_lim, 1.25 * max_lim])
    ax_pz.grid()
    ax_pz.legend()

    plt.show()
```

Non-Linear Least Squares Fitting

Using nonlinear least squares with Levenberg-Marquardt step length damping to fit poles and zeros representation to look-up table of complex system response. Nominally, the target function is of the form

$$zpk(s) = k\frac{(s - z_1)(s - z_2)(s - z_3)\ldots(s - z_m)}{(s - p_1)(s - p_2)(s - p_3)\ldots(s - p_n)}$$

where s is complex frequency ($s = i\omega$, where ω is angular frequency), k is the overall system gain, z represents the m system zeros, and p represents the n system poles.

HOWEVER. We're actually going to fit the look-up table with the transfer function form of the system response function:

$$h(s) = \frac{a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0}{b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s^1 + b_0}$$

where a_x and b_x are the (real) coefficients on the numerator and denominator polynomials, m is the number of system zeros (also the degree of the numerator polynomial), n is the number of system poles (also the degree of the denominator polynomial), and s is complex frequency ($s = i\omega$, where ω in angular frequency).

This form has several benefits. First of all, because the coefficients are real, the roots of these polynomials (and therefore the poles/zeros) will occur as complex conjugate pairs, which is what we expect/require for a real linear system operating on a real time series (I think...). Having real coefficients in this polynomial form also makes the math a bit easier.

Then, to find the poles/zeros, we just have to factor the numerator and denominator of this rational function. Operationally, we'll just use Python tools to do so.

For the nonlinear least squares fitting, the Jacobian is fairly simple to calculate:

$$\frac{\partial h(s)}{\partial a_m} = \frac{s^m}{b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s^1 + b_0} = \frac{s^m}{d(s)}$$

(Other partial derivatives with respect to the numerator terms are similar.)

$$\frac{\partial h(s)}{\partial b_n} = -\frac{a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0}{(b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s^1 + b_0)^2} s^n = -\frac{n(s)}{d(s)^2} s^n$$

(Other partial derivatives with respect to the denominator terms are similar.)

Here, $n(s)$ and $d(s)$ are the numerator and denominator polynomials, respectively.

In the matrices here, all the numerator terms will be listed first, then all the denominator terms.

In [5]:
```
def evaluate_transfer_function_response(num, den, w):
    """
    num, array-like : numerator coefficients, in decreasing order
    den, array-like : denominator coefficients, in decreasing order
    w, array-like   : angular frequencies at which to evalute
                      the function
    """
    tf = signal.TransferFunction(num, den)
    w, resp = signal.freqresp(tf, w=w)
    return resp
```

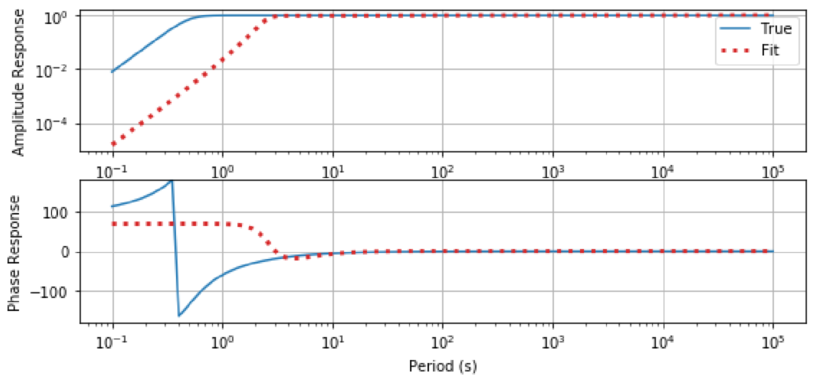
In [6]:
```
def get_jacobian(num, den, w):
    """
    num, array-like : numerator coefficients, in decreasing order
    den, array-like : denominator coefficients, in decreasing order
    w, array-like   : angular frequencies at which to evalute
                      the function
    """
    jac = np.zeros((w.size, num.size + den.size), dtype=np.complex128)
    s = 1.j * w
    n_s = np.polyval(num, s)
    d_s = np.polyval(den, s)
    for i in range(num.size):
        jac[:, i] = s**(num.size - 1 - i) / d_s
    for i in range(den.size):
        jac[:, i + num.size] = - s**(den.size - 1 - i) * n_s / d_s**2.
    return jac
```

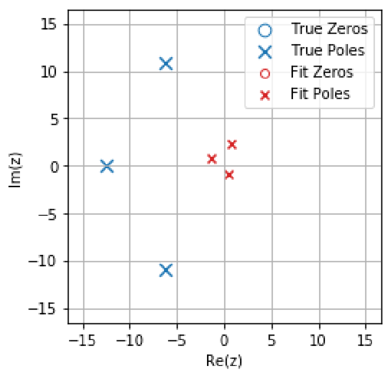
In [7]:
```python
def Fit_ZerosPolesGain_toFrequencyResponse_LM(w, resp, m, n):
    """
    w (array)    : (real) angular frequencies at which we
                       have the system response
    resp (array) : complex system response at angular frequencies w
    m (int)      : number of zeros to use in the fit; number of
                       numerator coefficients will be m + 1
    n (int)      : number of poles to use in the fit; number of
                       denominator coefficients will be n + 1
    """

    # set up initial guess.
    tf = np.zeros(m + 1 + n + 1, dtype=np.complex128)
    tf[:] = 1.   # for now, just set initial guess at all ones
    print("Initial Numerator Guess: ", tf[:m+1])
    print("Initial Denominator Guess: ", tf[m+1:])

    # evaluate initial misfit
    resp_pred = \
        evaluate_transfer_function_response(tf[:m+1], tf[m+1:], w)
    resid = resp - resp_pred
    misfit = np.mean(np.absolute(resid))
    print("Initial misfit: ", misfit)

    max_iter = 10
    l = 1
    for i in range(max_iter):

        misfit_previous = misfit

        # (1) get Jacobian matrix
        jac = get_jacobian(tf[:m+1], tf[m+1:], w)

        # (2) solve the least squares system, with L-M damping
        hes = np.matmul(jac.conj().T, jac)
        hes += l * np.eye(m + 1 + n + 1)
        d_tf = np.matmul(np.matmul(np.linalg.inv(hes), jac.conj().T),
                         resid)
        # PROBLEM the polynomial coefficients are becoming complex

        # (3) update model parameter
        tf += d_tf

        # (4) evaluate misfit
        resp_pred = \
            evaluate_transfer_function_response(tf[:m+1], tf[m+1:], w)
        resid = resp - resp_pred
        misfit = np.mean(np.absolute(resid))
        print("Misfit Step {:d}: ".format(i+1), misfit)

        # (5) adjust damping term l
        if misfit > misfit_previous: l *= 2
        else: l /= 2

    return signal.TransferFunction(tf[:m+1], tf[m+1:]).to_zpk()
```

Nonlinear LM fit to NIMS B response

```
In [8]: print("PARAMETERIZATION: 0 zeros, 3 poles")
        test_period = np.logspace(-1, 5, num=100)
        test_w = 2. * np.pi / test_period
        NIMS_w, NIMS_Bresp = signal.freqresp(NIMS_Magnetic_3PoleLowpass,
                                             w=test_w)    # w is in rad/sec
        NIMS_LM_ZPK = Fit_ZerosPolesGain_toFrequencyResponse_LM(NIMS_w,
                                                                NIMS_Bresp,
                                                                0, 3)
        plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_LM_ZPK,
                      w_values=test_w)
```

```
PARAMETERIZATION: 0 zeros, 3 poles
Initial Numerator Guess:  [1.+0.j]
Initial Denominator Guess:  [1.+0.j 1.+0.j 1.+0.j 1.+0.j]
Initial misfit:  0.3905301610061775
Misfit Step 1:  0.3403990610683085
Misfit Step 2:  0.364175109332949
Misfit Step 3:  0.3060996107464751
Misfit Step 4:  0.355562825616186
Misfit Step 5:  0.28887850705540613
Misfit Step 6:  0.24457928512349647
Misfit Step 7:  0.21663084876634714
Misfit Step 8:  0.1827350637992361
Misfit Step 9:  0.1545248385960636
Misfit Step 10:  0.15382987812226132
```





Ok, so the non-linear least squares approach isn't working so well. Apparently there's no guarantee that the model updates will be purely real (as I had hoped), and it's not converging very well. This could be because I used a bad initial guess, or because I'm doing something wrong. But let's try something else...

Linear Least Squares

We can do some equation manipulation to turn this into a linear least squares problem. Again starting with the transfer function representation of the response function:

$$h(s) = \frac{a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0}{b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s^1 + b_0} = \frac{n(s)}{d(s)}$$

where a_x and b_x are the (real) coefficients on the numerator and denominator polynomials, m is the number of system zeros (also the degree of the numerator polynomial), n is the number of system poles (also the degree of the denominator polynomial), and s is complex frequency ($s = i\omega$, where ω in angular frequency).

Rearranging this equation...

$d(s) \cdot h(s) = n(s)$

$(b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s^1 + b_0) \cdot h(s) = a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0$

If we require $b_0 = 1$, then we can get this into a convenient linear least squares form. (Note that $b_0 = 1$ just means that all the other polynomial coefficients are effectively normalized by whatever b_0 would've been before.)

$b_n s^n \cdot h(s) + b_{n-1} s^{n-1} \cdot h(s) + \ldots + b_1 s^1 \cdot h(s) + 1 \cdot h(s) = a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0$

$h(s) = a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s^1 + a_0 - b_n s^n \cdot h(s) - b_{n-1} s^{n-1} \cdot h(s) - \ldots - b_1 s^1 \cdot h(s)$

So now we can have the complex response values $h(s)$ in the RHS vector $\mathbf{d}$; the coefficients a_m and b_m as the model parameter $\mathbf{m}$; s^m and $-s^n \cdot h(s)$ in the design matrix $\mathbf{G}$.

$\mathbf{d} = \mathbf{G}\mathbf{m}$

$$\begin{bmatrix} h(s_0) \\ h(s_1) \\ \vdots \\ h(s_k) \end{bmatrix} = \begin{bmatrix} s_0^m & s_0^{m-1} & \ldots & s_0^1 & 1 & -s_0^n \cdot h(s_0) & -s_0^{n-1} \cdot h(s_0) & \ldots & -s_0^1 \cdot h(s_0) \\ s_1^m & s_1^{m-1} & \ldots & s_1^1 & 1 & -s_1^n \cdot h(s_1) & -s_1^{n-1} \cdot h(s_1) & \ldots & -s_1^1 \cdot h(s_1) \\ & & & & \vdots & & & & \\ s_k^m & s_k^{m-1} & \ldots & s_k^1 & 1 & -s_k^n \cdot h(s_k) & -s_k^{n-1} \cdot h(s_k) & \ldots & -s_k^1 \cdot h(s_k) \end{bmatrix} \begin{bmatrix} a_m \\ a_{m-1} \\ \vdots \\ a_1 \\ a_0 \\ b_n \\ b_{n-1} \\ \vdots \\ b_1 \end{bmatrix}$$

This formulation of a linear least-squares problem is sometimes called the "error-equation" method or the Levy Method in the signal processing literature. However, this form is suboptimal, as the denominator coefficients b_n are effectively weighted by the response values $h(s)$. A simple method to deal with this bias is to iteratively weight (divide) the linear least squares problem by the denominator term $d(s_k)$ from the previous iteration until convergence. This is called Sanathanan-Koerner iteration.

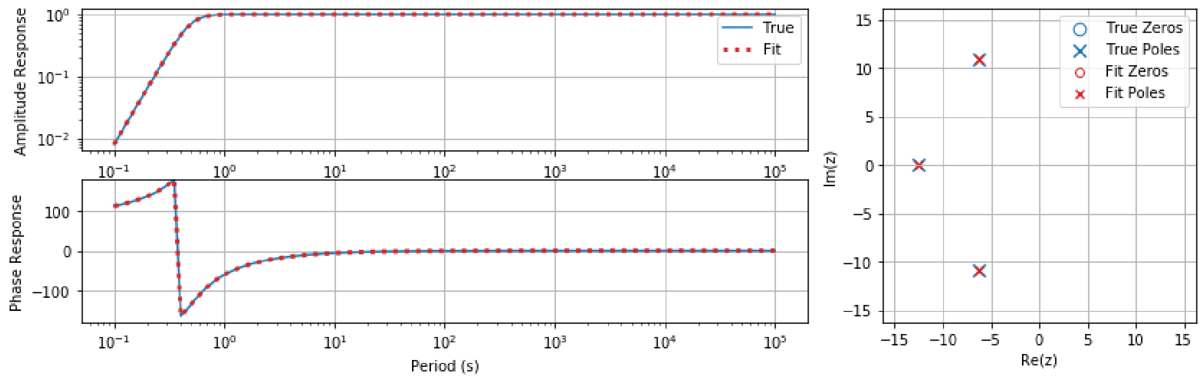
```
In [9]: def Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(w, resp, m, n,
                                                        useSKiter=False,
                                                        regularize=False):
            """
            USING Levy Method ("Error-Equation" Method) TO MAKE THIS A LINEAR
            LEAST SQUARES PROBLEM. THEN OPTIONALLY USING Sanathanan-Koerner (SK)
            Iteration TO DEAL WITH THE BIAS INTRODUCED BY THE WAY THE PROBLEM
            IS SET UP. ALSO INCLUDES OPTIONAL REGULARIZATION.

            w (array)    : (real) angular frequencies at which we
                              have the system response
            resp (array) : complex system response at angular frequencies w

            m (int)      : number of zeros to use in the fit; number of
                              numerator coefficients will be m + 1
            n (int)      : number of poles to use in the fit; number of
                              denominator coefficients will be n + 1

            useSKiter (bool)   : flips on/off Sanathanan-Koerner (SK) iteration

            regularize (bool)  : flips on/off regularization

            """

            # CONVERT TO COMPLEX (LAPLACE) FREQUENCY, DEFINE USEFUL VALUES
            s = 1.j * w
            num = m + 1
            den = n + 1

            # SET UP ITERATION CONTROLS
            if useSKiter:
                iter_max = 100
            else:
                iter_max = 1
            tf = None
            iter_count = 0
            for it in range(iter_max):

                # (1) SET UP DESIGN MATRIX
                # Because of the way this problem is framed, the constant term
                # in the denominator is not included here. See notes above.
                # Basically, the constant term from the denominator is over
                # in the RHS vector.
                g = np.zeros((w.size, num + den - 1), dtype=np.complex128)
                for i in range(num):
                    g[:, i] = s**(num - 1 - i)
                for i in range(den - 1):
                    g[:, i + num] = - resp * s**(den - 1 - i)

                # (2) SET UP WEIGHTS
                # In S-K iteration, the system is weighted by the denominator
                # of theprevious iteration. During the first iteration,
                # set weights = 1.
                if tf is None:
                    weights = np.eye(resp.size)
                else:
```

```
            weights = np.diag(np.polyval(tf[m+1:], w))

        # (3) COMPUTE THE SVD
        u, sing, vh = np.linalg.svd(np.matmul(weights, g),
                                    full_matrices=True)

        # (4) REGULARIZE, IF NECESSARY
        if regularize:
            alpha = np.logspace(-4, 4, num=100)
            sing_full = np.zeros(u.shape[0])
            sing_full[:sing.size] = sing
            filt = sing_full[np.newaxis, :]**2. / \
                   (sing_full[np.newaxis, :]**2. +
                    alpha[:, np.newaxis]**2.)
            d_bar = np.matmul(u.conj().T, np.matmul(weights, resp))
            err = np.matmul((1. - filt)**2., d_bar**2.) / \
                  (resp.size * np.sum(1. - filt, axis=1)**2.)
            alpha_optimal = alpha[np.argmin(err)]
            print("Regularization Parameter: ", alpha_optimal)
            filter_factors = sing**2. / (sing**2. + alpha_optimal**2.)
            print("Max/Min Filter Factors: ", np.amax(filter_factors),
                  " / ", np.amin(filter_factors))
        else:
            filter_factors = np.ones(sing.size)

        # (4) CALCULATE THE MODEL PARAMETER
        # Need to concatenate a value of [1] on the end of the model
        # parameter in order to get the full list of transfer function
        # coefficients. The [1] is the constant term in the
        # denominator polynomial.
        model = \
    np.matmul(vh.conj().T,
              np.matmul(np.diag(filter_factors),
                        np.matmul(np.diag(sing**-1.),
                                  np.matmul(u[:, :num+den-1].conj().T,
                                            np.matmul(weights, resp)))))
        tf_previous = tf
        tf = np.concatenate((model, np.array([1.])))

        # (5) EVALUATE THE CHANGE FROM THE LAST MODEL PARAMETER
        # Break iteration if the percent change in mean magnitude is <1%
        if tf_previous is not None:
            mean_value = np.mean(np.absolute(tf_previous))
            mean_change = np.mean(np.absolute(tf - tf_previous))
            if mean_change / mean_value < 0.01:
                break
        iter_count += 1

    if useSKiter:
        print("Number of S-K iterations: ", iter_count)

    # (6) RETURN THE RESULT IN POLES-ZEROS FORM
    return signal.TransferFunction(tf[:m+1], tf[m+1:]).to_zpk()
```

Fitting NIMS B response with linear approach, no S-K iteration and no regularization

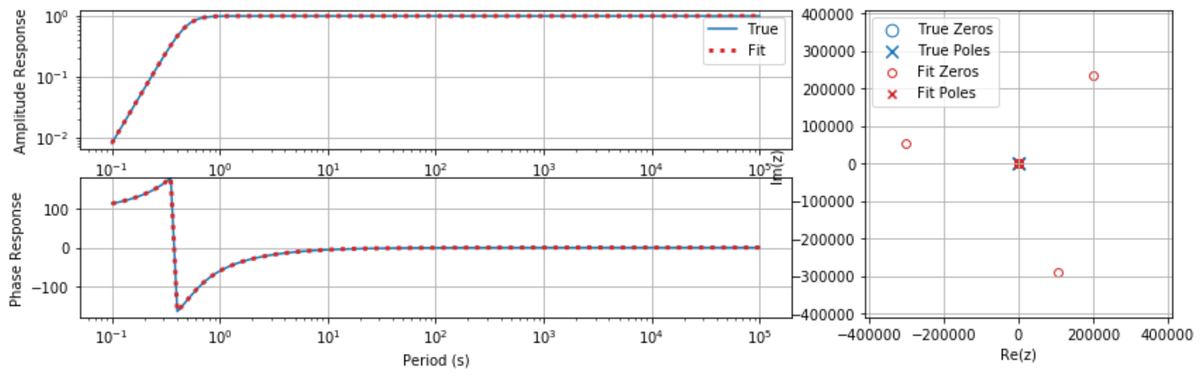
In [10]:
```
print("PARAMETERIZATION: 0 zeros, 3 poles")
test_period = np.logspace(-1, 5, num=100)
test_w = 2. * np.pi / test_period
NIMS_w, NIMS_Bresp = signal.freqresp(NIMS_Magnetic_3PoleLowpass,
                                     w=test_w)
NIMS_Fit_ZPK = Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w,
                                                           NIMS_Bresp,
                                                           0, 3)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

```
PARAMETERIZATION: 0 zeros, 3 poles
```



In [11]:
```
print("PARAMETERIZATION: 7 zeros, 7 poles")
NIMS_Fit_ZPK = Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w,
                                                           NIMS_Bresp,
                                                           7, 7)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

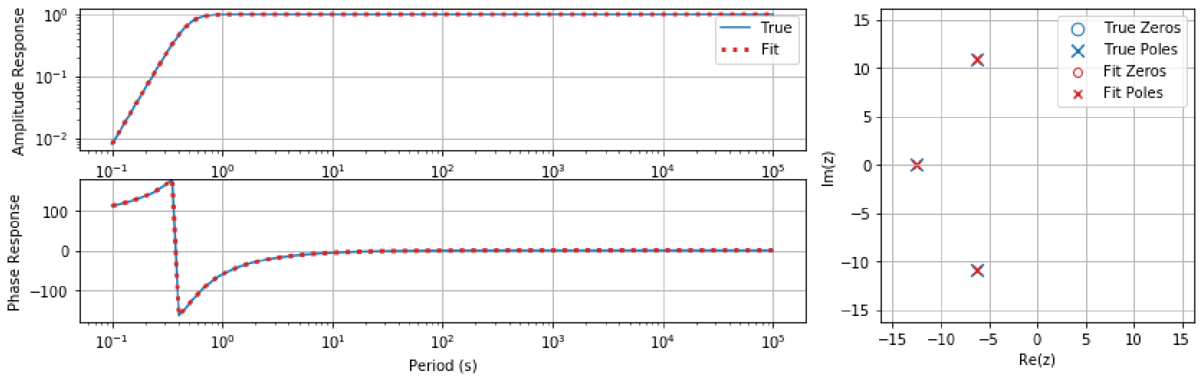
```
PARAMETERIZATION: 7 zeros, 7 poles
```



Fitting NIMS B response with linear approach, using S-K iteration

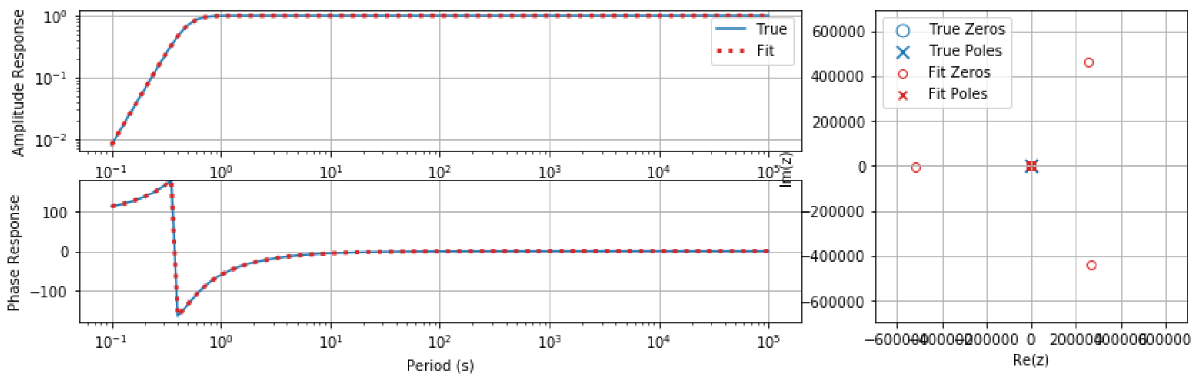
In [12]:
```
print("PARAMETERIZATION: 0 zeros, 3 poles")
NIMS_Fit_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w, NIMS_Bresp,
                                                0, 3,
                                                useSKiter=True)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

```
PARAMETERIZATION: 0 zeros, 3 poles
Number of S-K iterations:  1
```



In [13]:
```
print("PARAMETERIZATION: 7 zeros, 7 poles")
NIMS_Fit_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w, NIMS_Bresp,
                                                7, 7,
                                                useSKiter=True)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

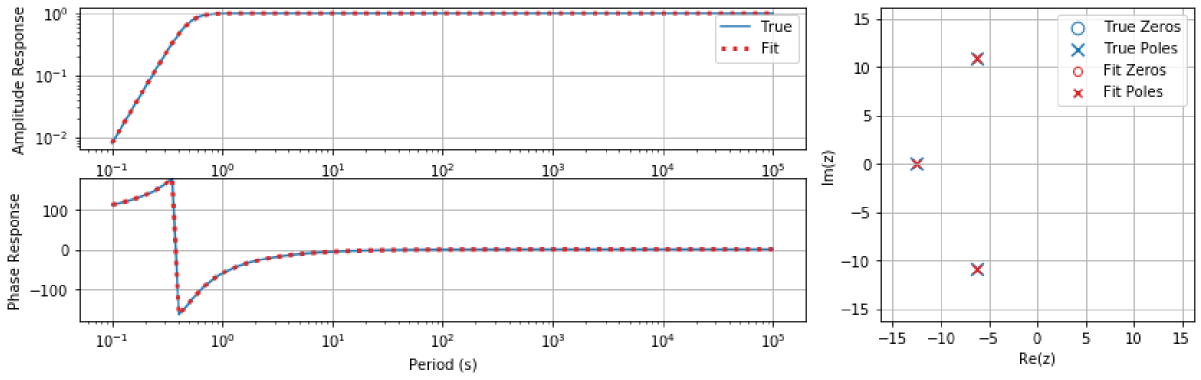
```
PARAMETERIZATION: 7 zeros, 7 poles
Number of S-K iterations:  4
```



Fitting NIMS B response with linear approach, no S-K iteration, regularized

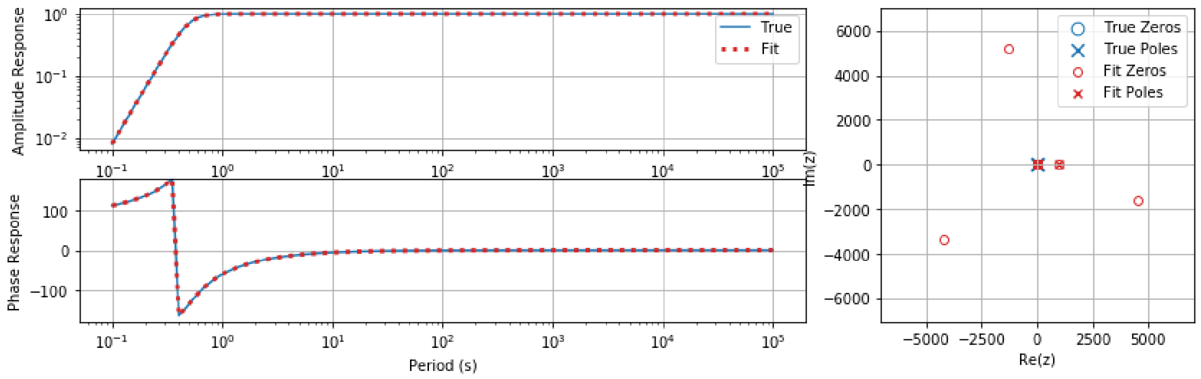
In [14]:
```
print("PARAMETERIZATION: 0 zeros, 3 poles")
NIMS_Fit_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w, NIMS_Bresp,
                                                0, 3, regularize=True)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

```
PARAMETERIZATION: 0 zeros, 3 poles
Regularization Parameter:  0.0001
Max/Min Filter Factors:  0.9999999999999999  /  0.9999999997738326
```



In [15]:
```
print("PARAMETERIZATION: 7 zeros, 7 poles")
NIMS_Fit_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w, NIMS_Bresp,
                                                7, 7, regularize=True)
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_Fit_ZPK,
              w_values=test_w)
```

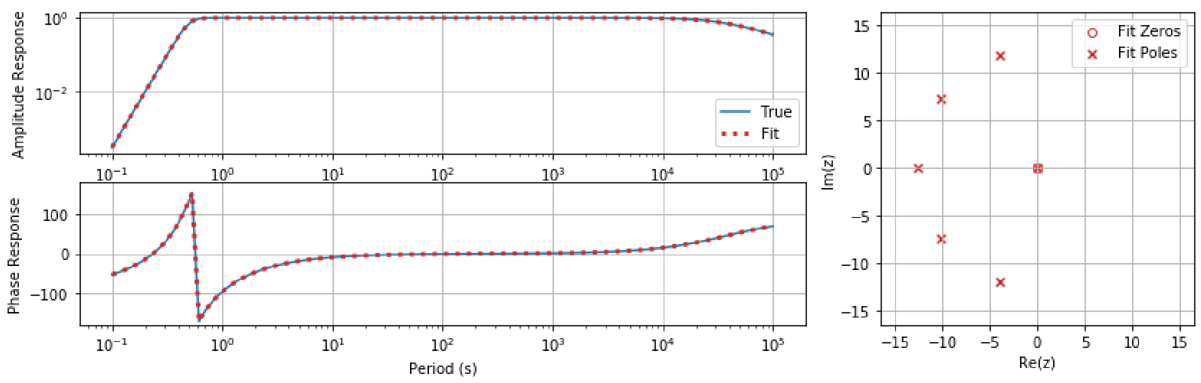
```
PARAMETERIZATION: 7 zeros, 7 poles
Regularization Parameter:  0.0001
Max/Min Filter Factors:  1.0  /  4.879234765713976e-21
```



Fitting NIMS E response (total) with linear approach, using S-K iteration

In [16]:
```
print("PARAMETERIZATION: 1 zero, 6 poles")
test_period = np.logspace(-1, 5, num=100)
test_w = 2. * np.pi / test_period
NIMS_w_highpass, NIMS_Ehigh = signal.freqresp(NIMS_Electric_Highpass,
                                              w=test_w)
NIMS_w_lowpass, NIMS_Elow = signal.freqresp(NIMS_Electric_5PoleLowpass,
                                            w=test_w)
NIMS_w = NIMS_w_lowpass
NIMS_Eresp = NIMS_Ehigh * NIMS_Elow
NIMS_EFit_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(NIMS_w, NIMS_Eresp,
                                                1, 6,
                                                useSKiter=True)
plot_response(w_obs=NIMS_w, resp_obs=NIMS_Eresp, zpk_pred=NIMS_EFit_ZPK,
              w_values=test_w)
print("True Zeros: ", NIMS_Electric_Highpass.zeros)
print("Fit Zeros: ", NIMS_EFit_ZPK.zeros)
print("True Poles: ", NIMS_Electric_Highpass.poles,
      NIMS_Electric_5PoleLowpass.poles)
print("Fit Poles: ", NIMS_EFit_ZPK.poles)
```

```
PARAMETERIZATION: 1 zero, 6 poles
Number of S-K iterations:  1
```

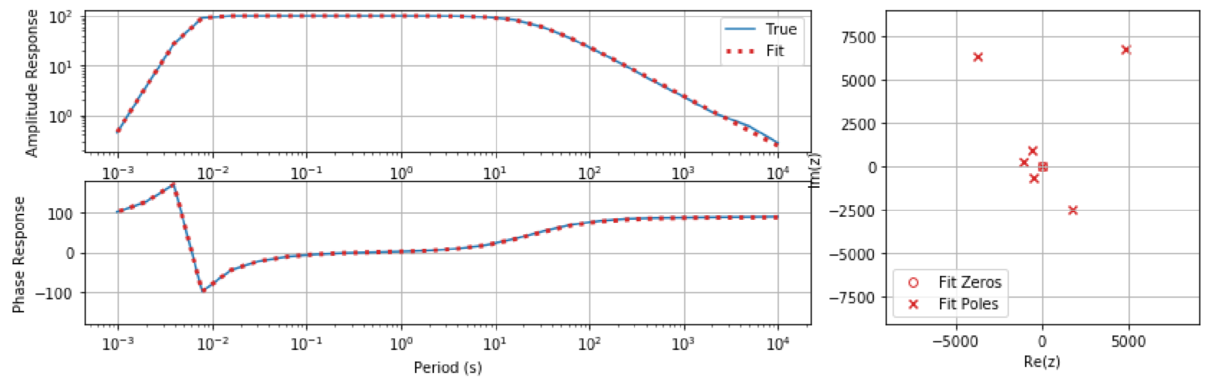


```
True Zeros:  [0.]
Fit Zeros:  [-4.63714494e-08+5.7406444e-08j]
True Poles:  [-0.00016667] [ -3.88301+11.9519j   -3.88301-11.9519j  -1
0.1662  +7.38651j
 -10.1662  -7.38651j -12.5664  +0.j      ]
Fit Poles:  [-3.88301000e+00-1.19519000e+01j -1.01662000e+01-7.38651000
e+00j
 -3.88301000e+00+1.19519000e+01j -1.25664000e+01+1.95846361e-10j
 -1.01662000e+01+7.38651000e+00j -1.66713332e-04+5.74402212e-08j]
```

Now trying fitting the Zen coil response with the linear approach, no S-K iteration

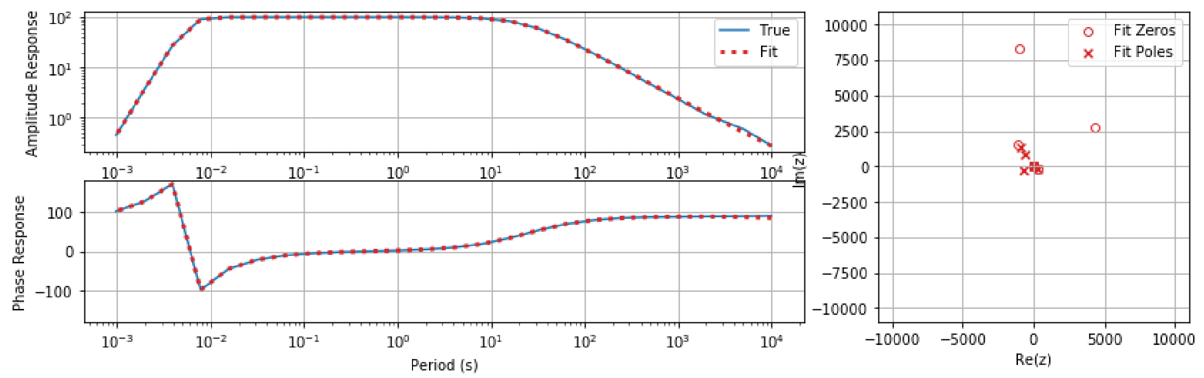
In [17]:
```
print("PARAMETERIZATION: 1 zero, 7 poles")
Zen_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                zen_complex_response,
                                                1, 7, useSKiter=False)
plot_response(w_obs=zen_angular_frequencies,
              resp_obs=zen_complex_response,
              zpk_pred=Zen_ZPK, w_values=zen_response[:, 0])
```

```
PARAMETERIZATION: 1 zero, 7 poles
```



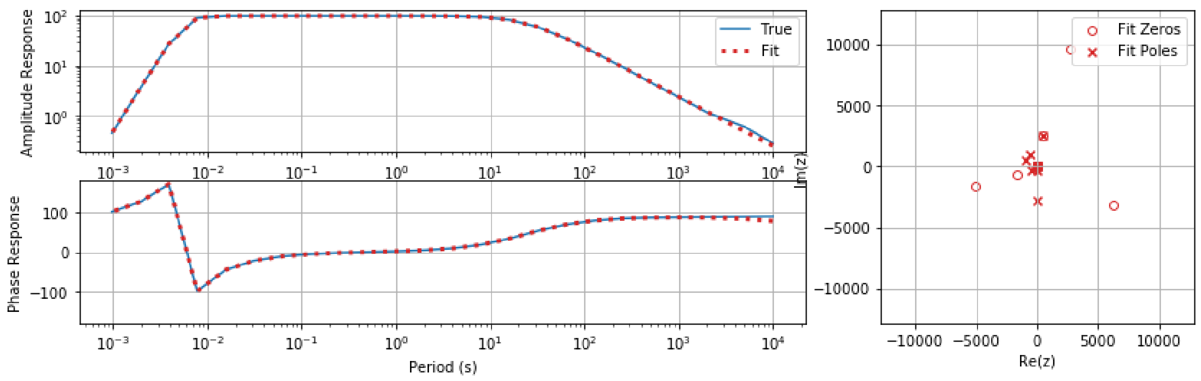
In [18]:
```
print("PARAMETERIZATION: 7 zeros, 7 poles")
Zen_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                zen_complex_response,
                                                7, 7, useSKiter=False,
                                                regularize=False)
plot_response(w_obs=zen_angular_frequencies,
              resp_obs=zen_complex_response,
              zpk_pred=Zen_ZPK, w_values=zen_response[:, 0])
```

```
PARAMETERIZATION: 7 zeros, 7 poles
```



```
In [19]: print("PARAMETERIZATION: 10 zeros, 10 poles")
         Zen_ZPK = \
             Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                          zen_complex_response,
                                                          10, 10, useSKiter=False,
                                                          regularize=False)
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response,
                       zpk_pred=Zen_ZPK, w_values=zen_response[:, 0])
```

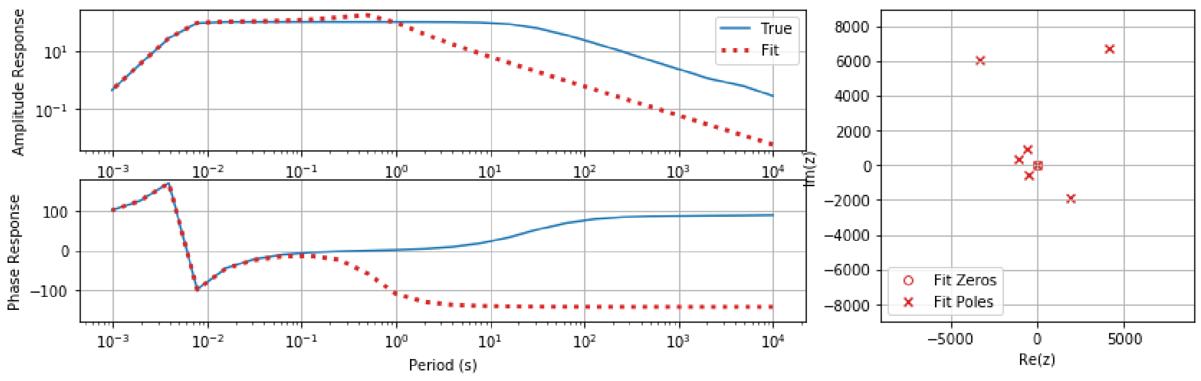
```
PARAMETERIZATION: 10 zeros, 10 poles
```



Now trying fitting the Zen coil response with the linear approach with S-K iteration

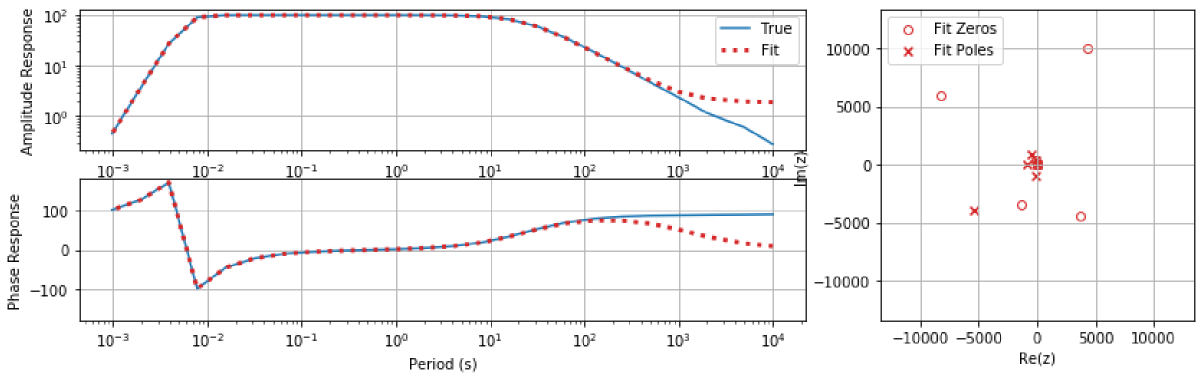
```
In [20]: print("PARAMETERIZATION: 1 zero, 7 poles")
         Zen_ZPK = \
             Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                          zen_complex_response,
                                                          1, 7, useSKiter=True)
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response,
                       zpk_pred=Zen_ZPK, w_values=zen_response[:, 0])
```

```
PARAMETERIZATION: 1 zero, 7 poles
Number of S-K iterations:  100
```



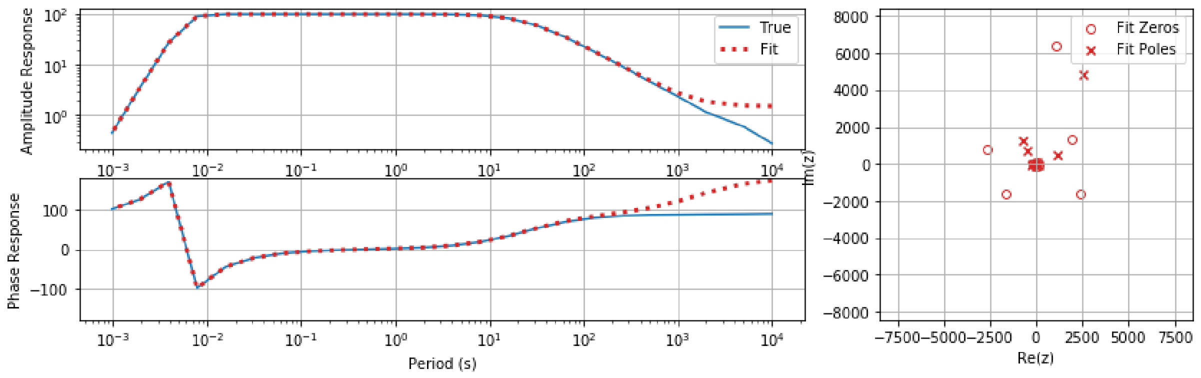
```
In [21]: print("PARAMETERIZATION: 7 zeros, 7 poles")
         Zen_ZPK = \
             Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                         zen_complex_response,
                                                         7, 7,
                                                         useSKiter=True)
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response,
                       zpk_pred=Zen_ZPK,
                       w_values=zen_response[:, 0])
```

```
PARAMETERIZATION: 7 zeros, 7 poles
Number of S-K iterations:  10
```



```
In [22]: print("PARAMETERIZATION: 10 zeros, 10 poles")
         Zen_ZPK = \
             Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                         zen_complex_response,
                                                         10, 10,
                                                         useSKiter=True)
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response,
                       zpk_pred=Zen_ZPK,
                       w_values=zen_response[:, 0])
```

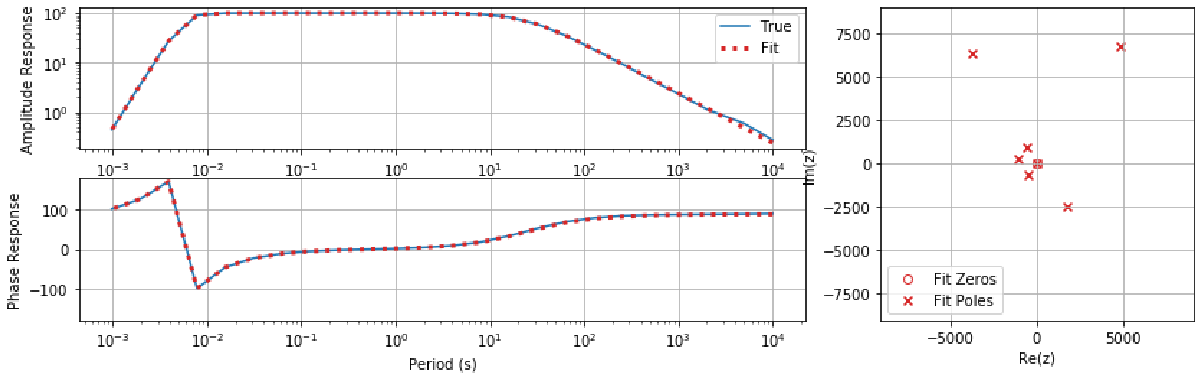
```
PARAMETERIZATION: 10 zeros, 10 poles
Number of S-K iterations:  13
```



Trying to fit Zen coil response with regularized linear least squares, no S-K iteration

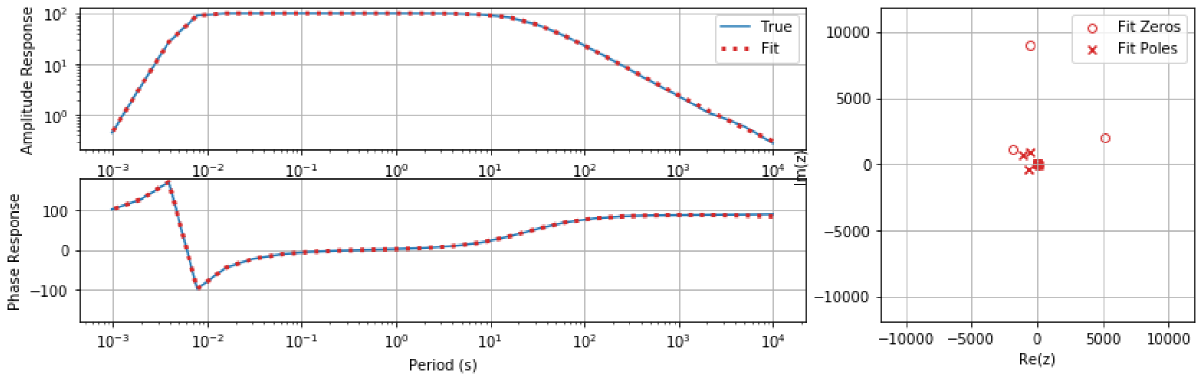
In [23]:
```
print("PARAMETERIZATION: 1 zero, 7 poles")
Zen_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                zen_complex_response,
                                                1, 7, regularize=True)
plot_response(w_obs=zen_angular_frequencies, zpk_pred=Zen_ZPK,
              resp_obs=zen_complex_response, w_values=zen_response[:,0])
```

```
PARAMETERIZATION: 1 zero, 7 poles
Regularization Parameter:  0.0001
Max/Min Filter Factors:  1.0  /  0.999999944819405
```



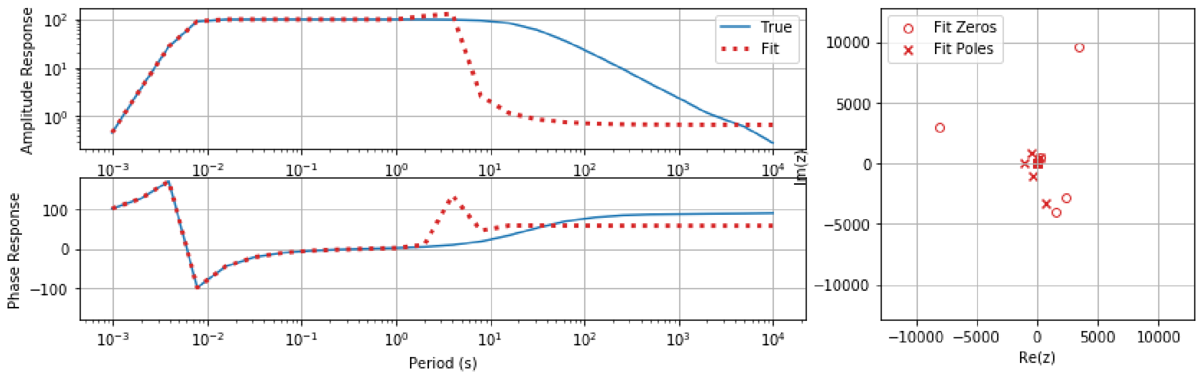
In [24]:
```
print("PARAMETERIZATION: 7 zeros, 7 poles")
Zen_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                zen_complex_response,
                                                7, 7, regularize=True)
plot_response(w_obs=zen_angular_frequencies, zpk_pred=Zen_ZPK,
              resp_obs=zen_complex_response, w_values=zen_response[:,0])
```

```
PARAMETERIZATION: 7 zeros, 7 poles
Regularization Parameter:  0.007220809018385471
Max/Min Filter Factors:  1.0  /  0.3243396474213219
```



```
In [25]: print("PARAMETERIZATION: 10 zeros, 10 poles")
         Zen_ZPK = \
             Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(zen_angular_frequencies,
                                                          zen_complex_response,
                                                          10, 10, useSKiter=False,
                                                          regularize=True)
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response, zpk_pred=Zen_ZPK,
                       w_values=zen_response[:, 0])
```

```
PARAMETERIZATION: 10 zeros, 10 poles
Regularization Parameter:  14.849682622544666
Max/Min Filter Factors:  1.0  /  1.3403626783077374e-10
```



Using S-K iteration with the Zen coil response skrews up the fit. The regularization doesn't really seem to help.

And regardless of the use of S-K iteration or regularization, for the Zen coil response, the poles here aren't coming in conjugate pairs, and some of the poles are plotting on the positive side of the real axis. **This might be a problem...?** My impression is that, for stable real systems (no runaway gains, real time series comes in, real time series comes out), the poles/zeros should come in complex conjugate pairs (or fall along the real axis) and should be restricted to the left half plane (-Re). But maybe I'm wrong about that...

Vector Fitting

Vector fitting is another method in the signal processing world to fit a function to complex response data. Instead of using the transfer function (or factored poles/zeros) representation of the system response, vector fitting uses the partial fractions expansion of the rational function:

$$h(s) = \sum_{n=1}^{N} \frac{c_n}{s - p_n} + d + s \cdot h$$

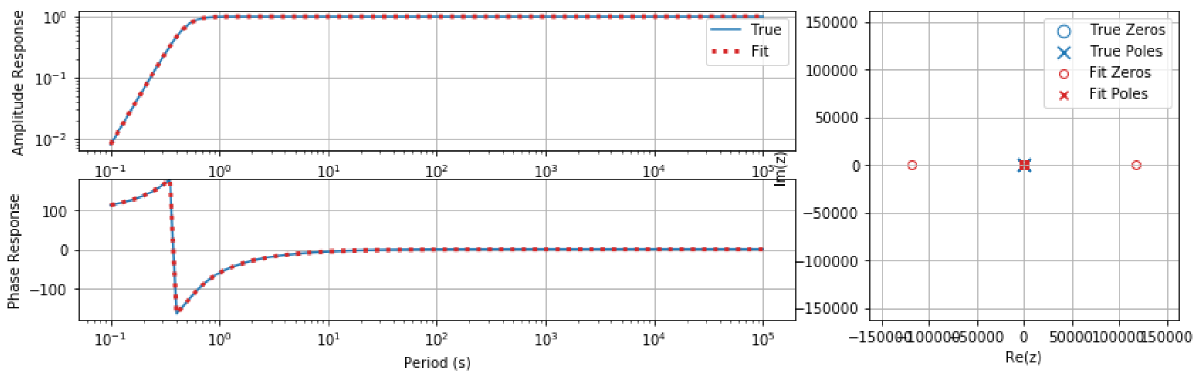
where c_n are the system residues, p_n are the system poles, d and h are constants, and s is complex frequency.

I'm using someone else's implementation of the algorithm here, so it's sort of a black box, and it also is perhaps overkill for our relatively simple systems. (VF seems to be used on complicated functions, not the simple functions that we deal with: flat in some pass band and then falling off with frequency on either side.)

Vector Fitting with NIMS B Response

In [26]:
```
test_period = np.logspace(-1, 5, num=100)
test_w = 2. * np.pi / test_period
NIMS_w, NIMS_Bresp = signal.freqresp(NIMS_Magnetic_3PoleLowpass,
                                     w=test_w)
poles, residues, d, h = vectfit.vectfit_auto(NIMS_Bresp, 1.j*NIMS_w, n_p
oles=3)
num_, den_ = signal.invres(residues, poles, [h, d])
NIMS_VF_ZPK = signal.TransferFunction(num_, den_).to_zpk()
plot_response(zpk_obs=NIMS_Magnetic_3PoleLowpass, zpk_pred=NIMS_VF_ZPK,
              w_values=NIMS_w)
print("# poles: ", NIMS_VF_ZPK.poles.size, ";   # zeros: ",
      NIMS_VF_ZPK.zeros.size)
```

```
poles: -12.5666 + 0j, -11.5852 + 0j, -6.34538 + -10.8346j, -6.34538 + 1
0.8346j, -6.28322 + -10.8827j, -6.28322 + 10.8827j
residues: 12.5644 + 0j, 0.00181701 + 0j, 0.00128419 + 0.0151252j, 0.001
28419 + -0.0151252j, -6.28442 + 3.61257j, -6.28442 + -3.61257j
offset: -1.31964e-08
slope: 1.03931e-10
```

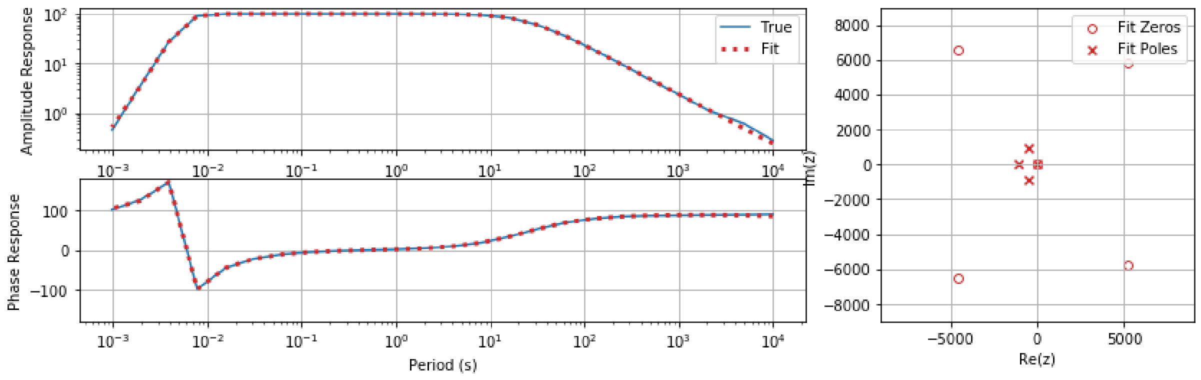


```
# poles:  6 ;   # zeros:  5
```

Vector Fitting with Zen coil response

```
In [27]: poles, residues, d, h = \
             vectfit.vectfit_auto(zen_complex_response,
                                  1.j * zen_angular_frequencies,
                                  n_poles=2)
         num_, den_ = signal.invres(residues, poles, [h, d])
         Zen_VF_ZPK = signal.TransferFunction(num_, den_).to_zpk()
         plot_response(w_obs=zen_angular_frequencies,
                       resp_obs=zen_complex_response,
                       zpk_pred=Zen_VF_ZPK, w_values=zen_response[:, 0])
         print("# poles: ", Zen_VF_ZPK.poles.size, ";   # zeros: ",
               Zen_VF_ZPK.zeros.size)
```

```
poles: -1072.29 + 0j, -538.767 + -919.537j, -538.767 + 919.537j, -0.261
917 + 0j
residues: 111274 + 0j, -55091.6 + 29648.9j, -55091.6 + -29648.9j, -26.0
316 + 0j
offset: -0.107995
slope: 3.11315e-05
```



```
# poles:  4 ;   # zeros:  5
```

Although I'm telling the VF algorithm to start with just a few poles, it is automatically adding more poles based on some internal criteria. This is resulting in overfitting the NIMS B response curve with too many poles (and zeros). For the fit to the Zen coil response, the poles are at least coming out in conjugate pairs, but the algorithm is also producing zeros in the right half of the complex plane.

However, this is again at least producing a poles-zeros representation of the response function that captures the amplitude and phase behavior correctly...

Automated Poles/Zeros Fitting

This is a simple code snippet that could be used to pick out a parameterization for a poles-zeros-gain representation of a complex response lookup table. Basic idea in my implementation here:

1. Loop over a number of poles and zeros (up to an arbitrary total number).
2. Fit the given number of poles/zeros to the lookup table complex values.
3. Evalute misfit of the fit via some metric.
4. Once the misfits from the all poles/zeros combinations have been evaluated, pick the fit that has the least number of zeros while also reaching some target misfit level.

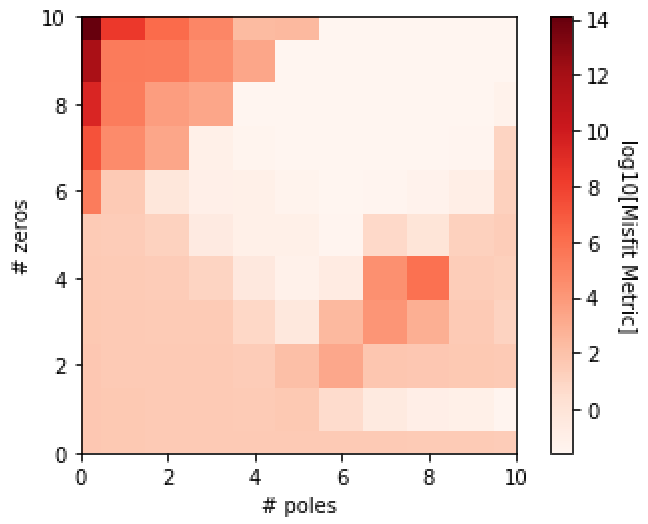
Picking the parameterization with the least number of zeros makes the most sense to me. With the types of system responses we generally deal with (flat in some pass band, then steeply rolling of on the sides), we're most likely going to have more poles than zeros (e.g., Buttersworth 3 pole, 5 pole, etc.). Here, I'm just using the simplest implementation of the linear least squares problem (no iterative reweighting, no regularization).

In [28]:
```
misfit_grid = np.zeros((11, 11))
for m in range(11):
    for n in range(11):
        Zen_ZPK = \
    Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(
        zen_angular_frequencies, zen_complex_response, m, n)
        w_pred, resp_pred = signal.freqresp(Zen_ZPK,
                                            w=zen_angular_frequencies)
        misfit_grid[m, n] = np.mean(np.absolute(
                                zen_complex_response - resp_pred))
```

```
/Users/bmurphy/.conda/envs/geo/lib/python3.7/site-packages/scipy/signa
l/filter_design.py:1619: BadCoefficients: Badly conditioned filter coef
ficients (numerator): the results may be meaningless
  "results may be meaningless", BadCoefficients)
```

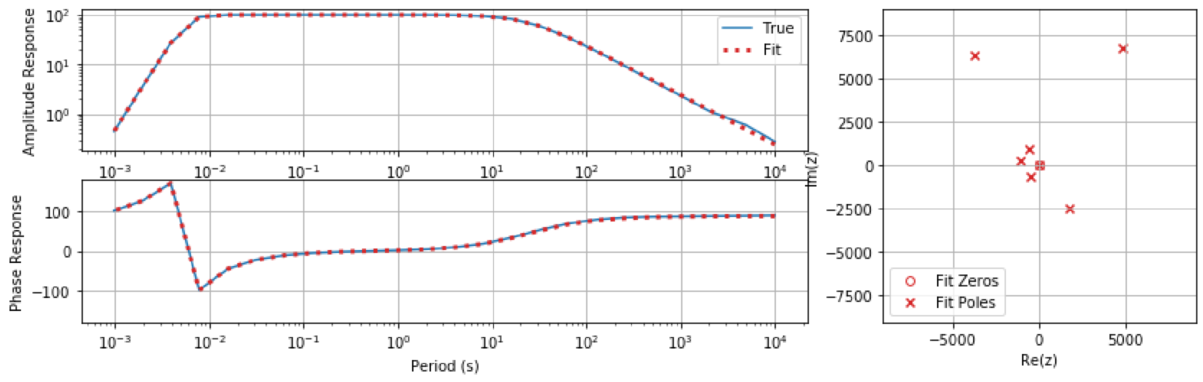
In [29]:
```
def find_optimal_parameterization(misfit, target_misfit_value=1.):
    for m in range(11):
        for n in range(11):
            if misfit[m, n] <= target_misfit_value:
                return m, n
```

In [30]:
```python
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
cax = ax.pcolormesh(np.arange(12)-0.5, np.arange(12)-0.5,
                    np.log10(misfit_grid), cmap='Reds')
ax.set_xlabel('# poles')
ax.set_ylabel('# zeros')
ax.set_xlim([0, 10])
ax.set_ylim([0, 10])
cbar = fig.colorbar(cax)
cbar.set_label('log10[Misfit Metric]', rotation=270, labelpad=12)
plt.show()
```

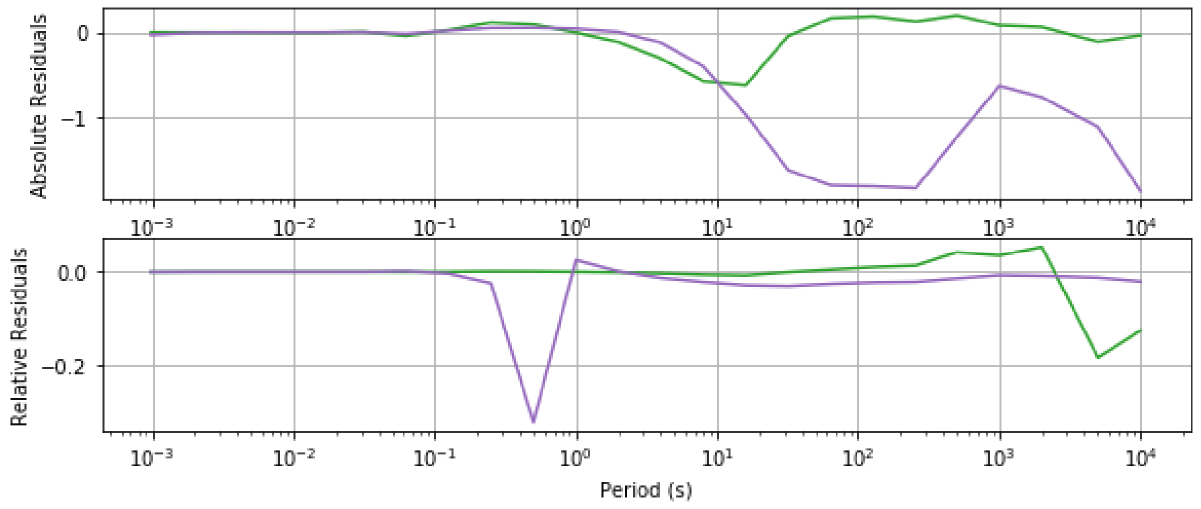


In [31]:
```python
m, n = \
    find_optimal_parameterization(misfit_grid, target_misfit_value=1.)
print("CHOSEN PARAMETERIZATION TO REACH DESIRED MISFIT: {:d} zeros, "
      "{:d} poles".format(m, n))
Zen_ZPK = Fit_ZerosPolesGain_toFrequencyResponse_LLSQ(
    zen_angular_frequencies, zen_complex_response, m, n,
    useSKiter=False, regularize=False)
plot_response(w_obs=zen_angular_frequencies,
              resp_obs=zen_complex_response, zpk_pred=Zen_ZPK,
              w_values=zen_angular_frequencies)
```

```
CHOSEN PARAMETERIZATION TO REACH DESIRED MISFIT: 1 zeros, 7 poles
```



In [33]:
```
w_pred, resp_pred = signal.freqresp(Zen_ZPK, w=zen_angular_frequencies)
amplitude_residuals = \
    np.absolute(resp_pred) - np.absolute(zen_complex_response)
phase_residuals = \
    np.angle(resp_pred, deg=True) - np.angle(zen_complex_response,
                                              deg=True)
fig = plt.figure(figsize=(10, 4))
ax_abs = fig.add_subplot(211)
ax_abs.plot(zen_period, amplitude_residuals, c='tab:green')
ax_abs.plot(zen_period, phase_residuals, c='tab:purple')
ax_abs.set_xscale('log')
ax_abs.set_ylabel('Absolute Residuals')
ax_abs.grid()
ax_rel = fig.add_subplot(212)
ax_rel.plot(zen_period,
            amplitude_residuals/np.absolute(zen_complex_response),
            c='tab:green')
ax_rel.plot(zen_period,
            phase_residuals/np.angle(zen_complex_response, deg=True),
            c='tab:purple')
ax_rel.set_xscale('log')
ax_rel.set_xlabel('Period (s)')
ax_rel.set_ylabel('Relative Residuals')
ax_rel.grid()
plt.show()
```



Concluding Thoughts/Questions

So, using either linear least squares or vector fitting, we can derive a poles-zeros-gain representation of the Zen coil response lookup table. The question is, does it matter if the poles/zeros aren't coming out in conjugate pairs and if any poles/zeros appear in the right half (+Re) of the complex plane. My (perhaps erroneous) impression is that, for the real systems we deal with (real-valued time series comes in, real-valued time series comes out, no instabilities or runaway gains in the system), poles/zeros should come in complex conjugate pairs (or fall along the real axis) and should be restricted to the left half plane (-Re). But maybe I'm wrong about that.

One way or another, we have a poles/zeros/gain representation at some level of misfit to the lookup table response, and maybe (hopefully) that's good enough for archiving with IRIS (even if the fit is overparameterized with too many poles/zeros, or even if the fit isn't 100% perfect).

References

https://cdn.intechopen.com/pdfs/57667.pdf (https://cdn.intechopen.com/pdfs/57667.pdf)

Linear LS Approach

https://www.dsprelated.com/freebooks/filters/Filter_Design_Minimizing_L2.html (https://www.dsprelated.com/freebooks/filters/Filter_Design_Minimizing_L2.html)

https://www.dsprelated.com/showcode/20.php (https://www.dsprelated.com/showcode/20.php)

Vector Fitting

https://arxiv.org/pdf/1908.08977.pdf (https://arxiv.org/pdf/1908.08977.pdf)

https://github.com/PhilReinhold/vectfit_python (https://github.com/PhilReinhold/vectfit_python)