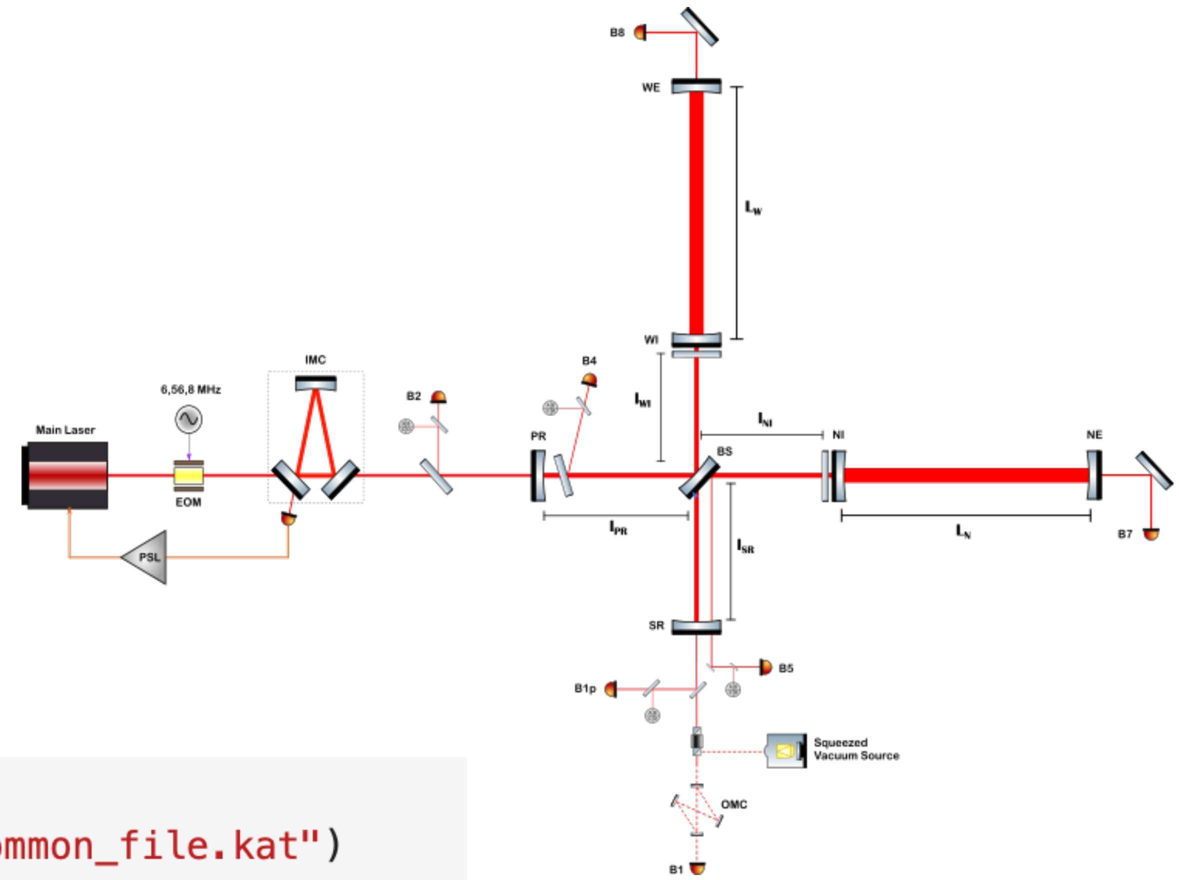
Finding working point of an interferometer with Finesse 3

Enzo Tapia S. for the Finesse team

A model of the detector:

Recommendation: Make a drawing of what you'd like to model.

Then write or import a kat-script that represents the interferometer we want to simulate.



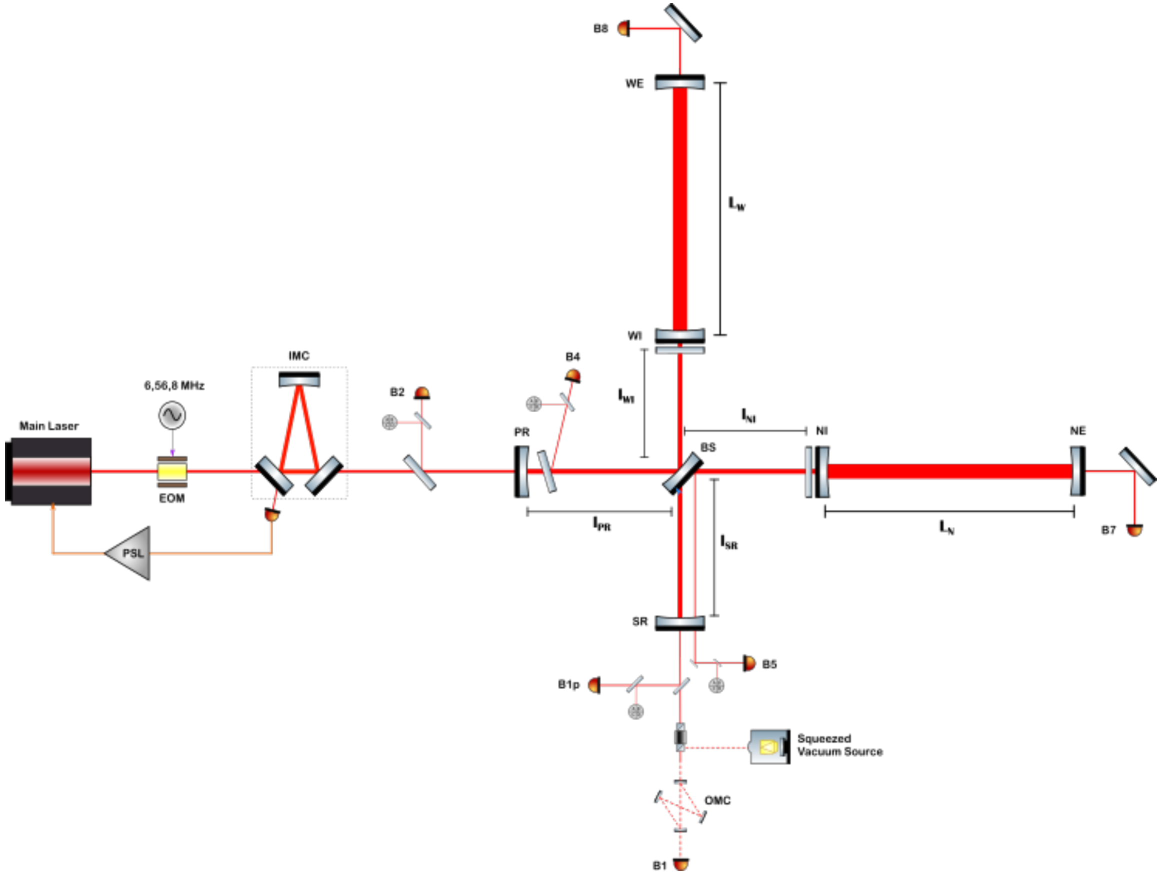

```
model0 = finesse.Model()
model0.parse_file("Virgo_local_katscript/00_virgo_common_file.kat")
model1 = model0.deepcopy()
model1.parse_file("Virgo_local_katscript/01_additional_katscript.kat")
```

Check and modify parameters:

```
model1.i1.P = 25
```

```
model1.i1.parameter_table()
```

Description	Value
Power	25.0 W
Phase	0.0 degrees
Frequency	0.0 Hz
Signals only	False



Adjust recycling cavities

Adjust recycling cavity lengths to fulfil requirement (design).

```
model1.get('lPOP_BS').L
```

```
<lPOP_BS.L=5.9399 @ 0x15dcfbf40>
```

```
adjust_recycling_cavity_length(model1, "PRC", "lPRC", "lPOP_BS", verbose = True)
```

```
— adjusting PRC length
    adjusting lPOP_BS.L by 0.0004736 m
```

```
model1.get('lPOP_BS').L
```

```
<lPOP_BS.L=5.94037359652047 @ 0x15dcfbf40>
```

```
model1.get('lsr').L
```

```
<lsr.L=5.943 @ 0x15ddf9a80>
```

```
adjust_recycling_cavity_length(model1, "SRC", "lSRC", "lsr", verbose = True)
```

```
— adjusting SRC length
    adjusting lsr.L by 0.000883 m
```

```
model1.get('lsr').L
```

```
<lsr.L=5.94388296505047 @ 0x15ddf9a80>
```

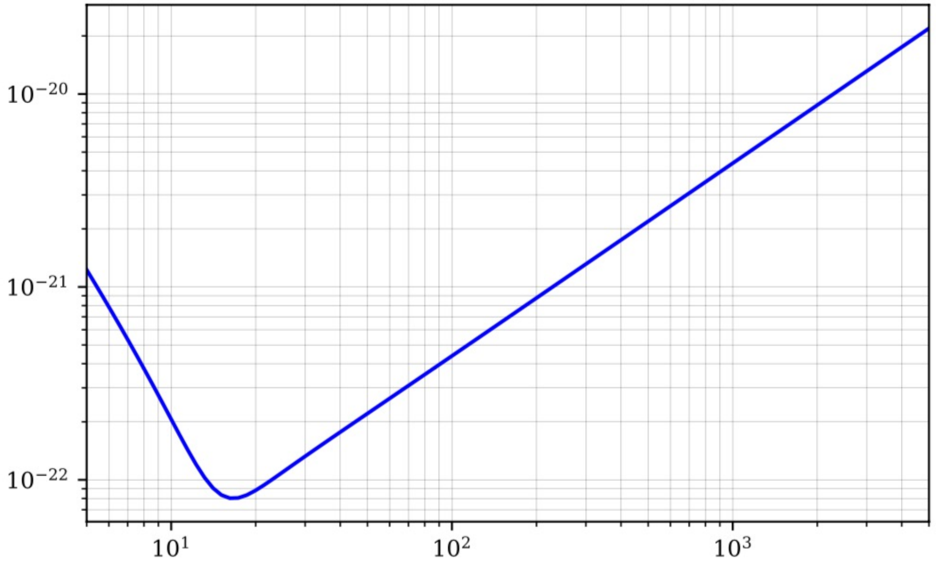
```
def adjust_recycling_cavity_length(
        model, cavity: str, L_in: str, S_out: str, verbose=False
    ):
        """Adjust cavity length so that it fulfils the requirement:

            L = 0.5 * c / (2 * f6), see TDR 2.3 (VIR-0128A-12).

        Parameters
        ----------
        cavity : str
            Name of the cavity being adjusted.
        L_in : str
            Variable used to define the length of the cavity.
            Needed because the common file does not use a variable.
        S_out : str
            Name of the space component used to adjust the cavity.
        """

        # works also for legacy
        f6 = model.get("eom6.f").value

        if verbose:
            print(f"—  adjusting {cavity} length")

        # calculate the required adjustment
        tmp = 0.5 * CONSTANTS["c0"] / (2 * f6)
        delta_l = tmp.eval() - model.get(L_in).value.eval()

        if verbose:
            print(f"    adjusting {S_out}.L by {delta_l:.4g} m")

        # apply the adjustment
        model.get(S_out).L += delta_l
```

Check sensitivity:

```
def get_QNLS(model, axis=[5, 5000, 100]):
        kat = model.deepcopy()
        kat.parse(
            """#kat
            # Differentially modulate the arm lengths
            fsig(1)
            sgen darmx LN.h
            sgen darmy LW.h phase=180

            # Output the full quantum noise limited sensitivity
            qnoised NSR_with_RP B1.p1.i nsr=True
            #qnoised NSR_with_RP SR.p2.o nsr=True

            # Output just the shot noise limited sensitivity
            qshot NSR_without_RP B1.p1.i nsr=True
        """
        )

        return kat.run(f'xaxis(darmx.f, "log", {axis[0]}, {axis[1]}, {axis[2]})')
```

```
out_sensitivity1 = get_QNLS(model1)
plt.loglog(out_sensitivity1.x1, np.abs(out_sensitivity1['NSR_with_RP']))
range_bns1 = inspiral_range.range(out_sensitivity1.x1, np.abs(out_sensitivity1['NSR_with_RP'])**2)
print('BNS range:', np.round(range_bns1, 3), 'Mpc')
```

```
BNS range: 17.262 Mpc
```



But before this step, it is important to set the parameter of propagation of the fundamental mode to 'False'.

```
model1._settings.phase_config.zero_k00=False
```

Note: Here we are experimenting also with the 'inspiral_range' module from Pygwinc. https://git.ligo.org/gwinc/inspiral-range

Check sensitivity:

But before this step, it is important to set the parameter of propagation of the fundamental mode to ‘False’.

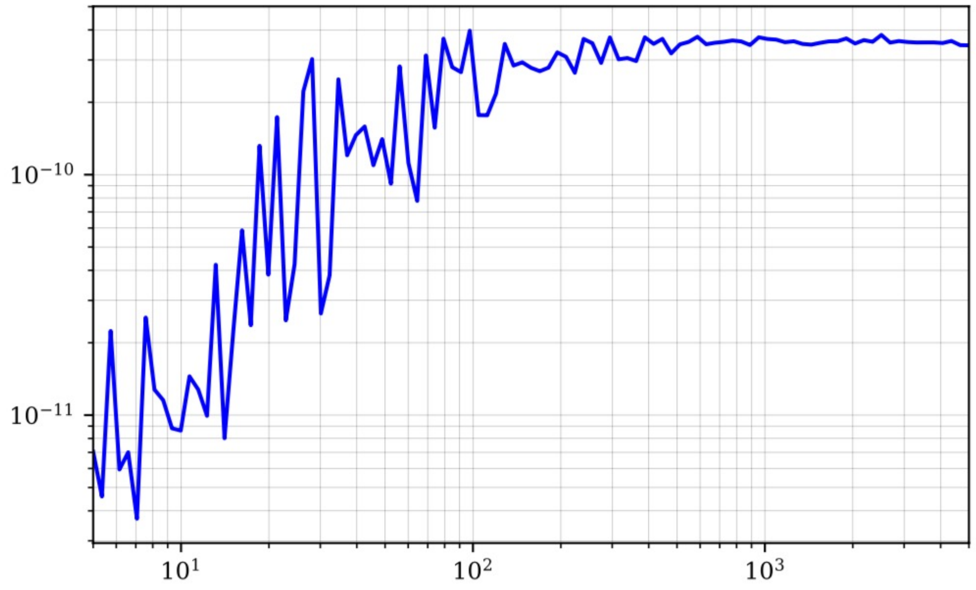
The models have this parameter as ‘True’ by default, thus we get an unwanted working point with a low sensitivity.

```
model1._settings.phase_config.zero_k00
```

```
True
```

```
out_sensitivity1 = get_QNLS(model1)
plt.loglog(out_sensitivity1.x1, np.abs(out_sensitivity1['NSR_with_RP']))
#range_bns1 = inspiral_range.range(out_sensitivity1.x1, np.abs(out_sensitivity1['NSR_with_RP'])**2)
#print('BNS range:', np.round(range_bns1, 3), 'Mpc')
```

```
[<matplotlib.lines.Line2D at 0x2a006f760>]
```



Finding and operating point: Pre-tuning

Following these steps:

- Only use main carrier field in the following steps.
- Remove any cross-coupling between the two arms, so we can tune them independently from each other.
- Arm cavities are on resonance.
- Interferometer at the dark fringe.
- PRC is on resonance.
- SRC is on resonance.
- SRC is in the condition for Resonant Sideband Extraction (RSE).
- Restore the modulators.

```
## Run the model and make this changes in series:
model2 = model1.deepcopy()
out1 = model2.run(
    fac.TemporaryParameters(
        fac.Series(
            # Switch off the modulators and misalign SR and PR mirrors:
            fac.Change({"eom6.midx": 0, "eom8.midx": 0, "eom56.midx": 0,
                        "SR.misaligned": True, "PR.misaligned": True,
                        "SRAR.misaligned": True, "PRAR.misaligned": True}),
            # Maximize arm powers:
            fac.Maximize("B7_DC", "NE_z.DC", bounds=[-180,180], tol=1e-14),
            fac.Maximize("B8_DC", "WE_z.DC", bounds=[-180,180], tol=1e-14),
            # Minimize dark fringe power:
            fac.Minimize("B1_DC", "MICH.DC", bounds=[-180,180], tol=1e-14),
            # Align back PRM:
            fac.Change({"PR.misaligned": False}),
            # Maximize PRC power:
            fac.Maximize("CAR_AMP_BS", "PRCL.DC", bounds=[-180,180], tol=1e-14),
            # Align back SRM:
            fac.Change({"SR.misaligned": False}),
            # Maximize SRC power, then offset by 90 deg:
            fac.Change({"SRCL.DC": 0}),
            fac.Maximize("B1_DC", "SRCL.DC", bounds=[-180,180], tol=1e-14),
            fac.Change({"SRCL.DC": -90}, relative=True)),
        exclude=("PR.phi", "NI.phi", "NE.phi", "WI.phi", "WE.phi", "SR.phi",
                 "NE_z.DC", "WE_z.DC", "MICH.DC", "PRCL.DC", "SRCL.DC")
    )
)
```

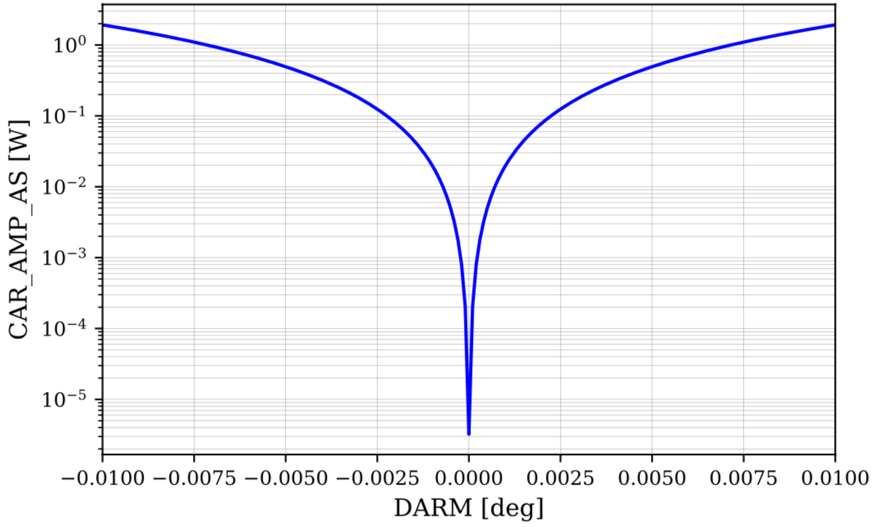
Finding and operating point: Pre-tuning

Check each relevant figure of merit:

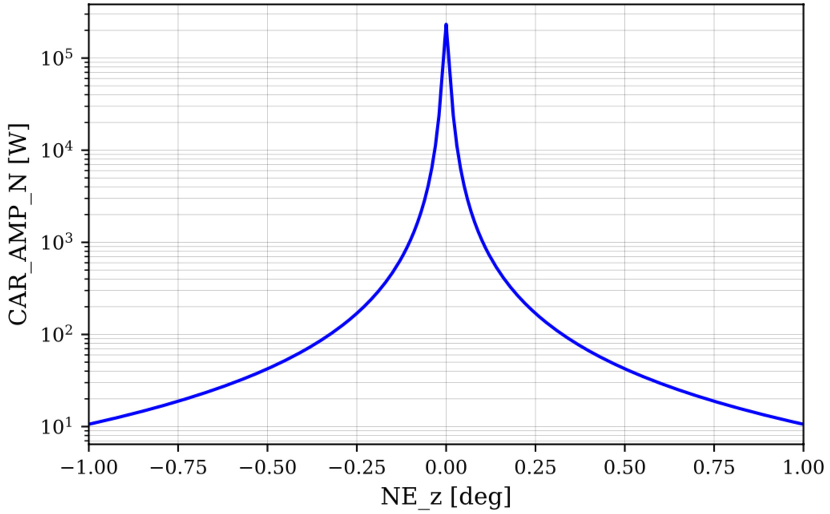
- Maximum power circulating in the arms.
- MICH tuned to dark fringe.
- PRCL tuned for maximum power inside the PRC.
- SRCL tuned for minimum power inside the SRC.
- DARM tuned for minimum power on B1.

```
def dof_plot(model, dof, detector, axis=[-1, 1, 200], show=True):
        """Sweep across a DoF, reading out at the provided (amplitude) detector."""
        axis = np.array(axis, dtype=np.float64)
        #axis[:2] *= xscale
        temp = model
        out = temp.run(
            fac.Xaxis(f"{dof}.DC", "lin", axis[0], axis[1], axis[2], relative=True)
        )

        plt.semilogy(out.x[0], (np.abs(out[detector]))**2) # these are amplitude detectors.
        plt.xlabel(model.get(dof+'.name') + " [deg]")
        plt.ylabel(model.get(detector+'.name') + " [W]")

        return out
```

```
# Plot power at the antisymmetric port:
out6 = dof_plot(model1, "DARM", "CAR_AMP_AS", axis=[-0.01, 0.01, 200])
```




```
# Plot power in North arm
out1 = dof_plot(model1, 'NE_z', 'CAR_AMP_N')
```



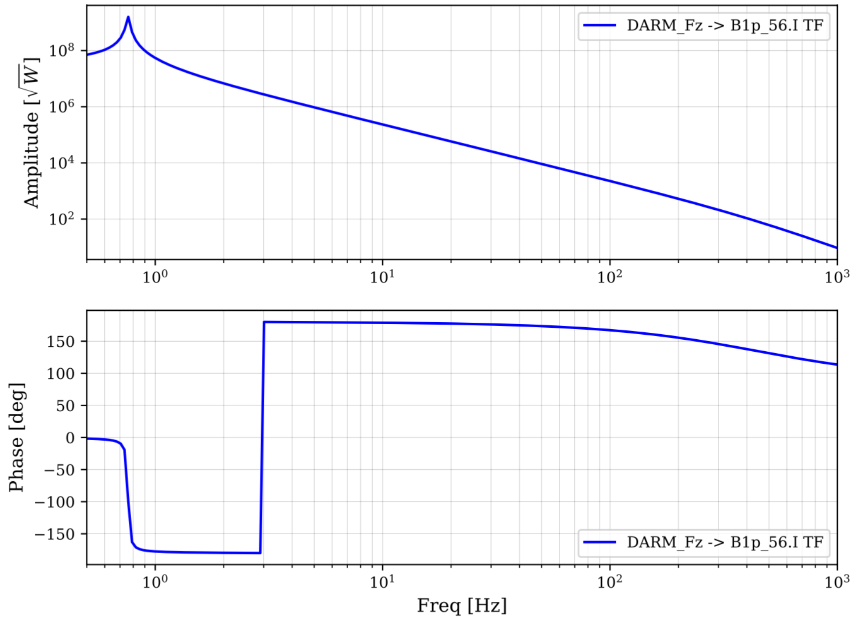
Check sensitivity, plot DARM TF.

The sensitivity can be improved with a locking scheme and not only the pre-tuning process.

We can check the DARM TF too. However, we should be checking more figures of merit.

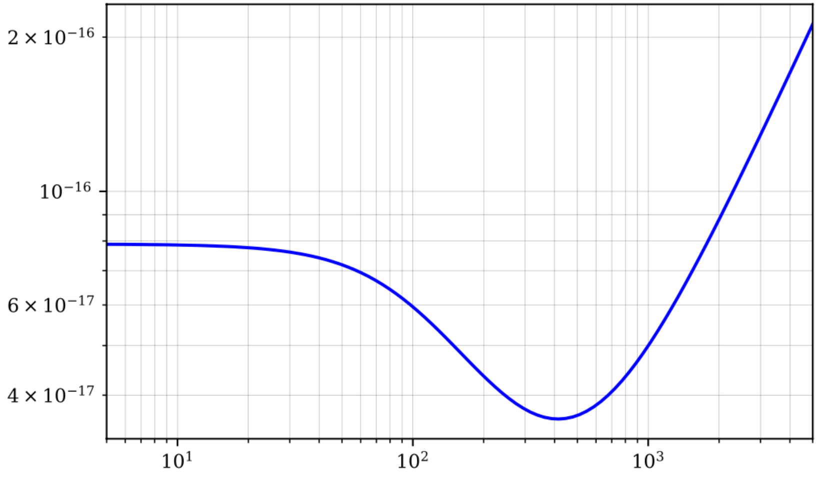
* Other figures of merit? Green lights, sensitivity in Mpc.

```
def get_QNLS(model, axis=[5, 5000, 100]):
        kat = model.deepcopy()
        kat.parse(
            """#kat
            # Differentially modulate the arm lengths
            fsig(1)
            sgen darmx LN.h
            sgen darmy LW.h phase=180

            # Output the full quantum noise limited sensitivity
            qnoised NSR_with_RP B1.p1.i nsr=True
            #qnoised NSR_with_RP SR.p2.o nsr=True

            # Output just the shot noise limited sensitivity
            qshot NSR_without_RP B1.p1.i nsr=True
        """
        )

        return kat.run(f'xaxis(darmx.f, "log", {axis[0]}, {axis[1]}, {axis[2]})')
```




```
out_sensitivity3 = get_QNLS(model3)
plt.loglog(out_sensitivity3.x1, np.abs(out_sensitivity3['NSR_with_RP']))
range_bns3 = inspiral_range.range(out_sensitivity3.x1, np.abs(out_sensitivity3['NSR_with_RP'])**2)
print('BNS range:', np.round(range_bns3, 3), 'Mpc')
```

```
BNS range: 0.0 Mpc
```



Identify and Optimize a Sensing matrix

Define a sensing matrix:

```
sens_matrix_dc = model3.run(fac.sensing.SensingMatrixDC(
                        ['DARM', 'CARM', 'MICH', 'PRCL', 'SRCL'],
                        ['B1p_56', 'B2_6', 'B2_56', 'B2_8'] ))
```

```
# Display sensing matrix:
sens_matrix_dc.display()
```

	B1p_56_I	B1p_56_Q	B2_6_I	B2_6_Q	B2_56_I	B2_56_Q	B2_8_I	B2_8_Q
DARM	71	63	-0.014	0.0024	-0.0037	0.011	0.01	-0.00062
CARM	0.18	0.16	3.8	-0.65	1	-3	-2.8	0.17
MICH	0.25	0.22	-1.2E-05	-4.4E-05	-0.00028	-0.00015	2.4E-05	-1.4E-06
PRCL	0.00027	-0.00022	-0.0018	0.00033	0.00043	0.0024	0.0098	-0.00059
SRCL	2.2E-05	1.1E-05	2.6E-06	-4.5E-07	7.4E-05	-0.00015	-2.7E-09	3.3E-10

Optimize demodulation phases:

```
# new model to be optimized:
model4 = model3.deepcopy()
```

```
for dof in dof_pds.keys():
    model4.run(fac.OptimiseRFReadoutPhaseDC(dof, dof_pds[dof]))
```

```
sens_matrix_dc_4 = model4.run(fac.sensing.SensingMatrixDC(
                        ['DARM', 'CARM', 'MICH', 'PRCL', 'SRCL'],
                        ['B1p_56', 'B2_6', 'B2_56', 'B2_8'] ))
sens_matrix_dc_4.display()
```

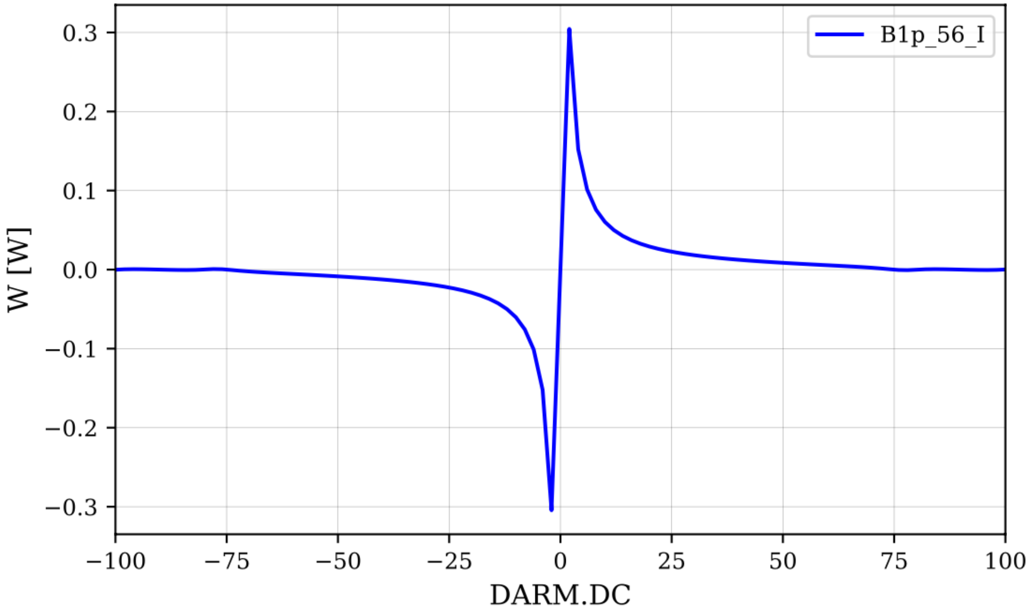
	B1p_56_I	B1p_56_Q	B2_6_I	B2_6_Q	B2_56_I	B2_56_Q	B2_8_I	B2_8_Q
DARM	95	4.5E-08	-0.014	-1.5E-07	-0.011	0.0015	0.01	8.9E-09
CARM	0.24	1.3E-06	3.9	-1E-07	3.1	-0.41	-2.8	-2.4E-06
MICH	0.33	0.00011	-4E-06	-4.5E-05	5.1E-06	-0.00031	2.4E-05	3.6E-08
PRCL	5.5E-05	-0.00035	-0.0018	2.7E-05	-0.002	0.0015	0.0098	6.6E-10
SRCL	2.4E-05	-6.4E-06	2.6E-06	-1.3E-08	0.00016	-7.9E-09	-2.8E-09	1.6E-10

Check error signals

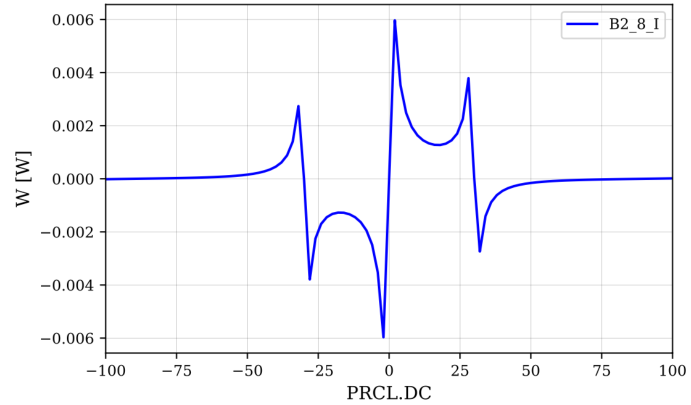
```python
# Longitudinal DoF and readouts:
dof_pds = {'DARM': 'B1p_56_I', 'CARM': 'B2_6_I', 'MICH': 'B2_56_Q', 'PRCL': 'B2_8_I',
```

```python
sol_2 ={}
for dof in dof_pds:
    print(dof_pds[dof])
    sol_2[dof] = model2.run(f'xaxis({dof}.DC, lin, -100, 100, 100, relative=true)')
    sol_2[dof].plot(dof_pds[dof])
```

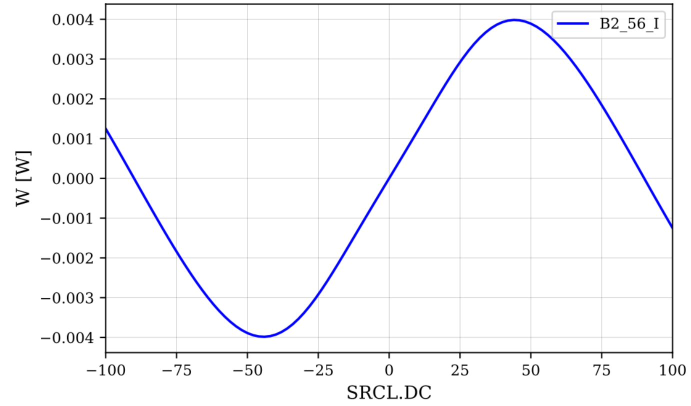
B1p_56_I



B2_8_I



B2_56_I



Define a control scheme

```
dof_pds
```

```
{'DARM': 'B1p_56_I',
 'CARM': 'B2_6_I',
 'MICH': 'B2_56_Q',
 'PRCL': 'B2_8_I',
 'SRCL': 'B2_56_I'}
```

```
model5 = model4.deepcopy()

model5.parse(
    """
    # locks
    lock DARM_lock B1p_56.outputs.I DARM.DC 0 1e-14
    lock CARM_lock B2_6.outputs.I CARM.DC 0 1e-14
    lock MICH_lock B2_56.outputs.Q MICH.DC 0 1e-11
    lock PRCL_lock B2_8.outputs.I PRCL.DC 0 1e-12
    lock SRCL_lock B2_56.outputs.I SRCL.DC 0 50e-11
    # Add also RF and DC locks for DARM:
    #lock DARM_rf_lock B1p_56.outputs.I DARM.DC 1 1e-14
    lock DARM_dc_lock B1.outputs.DC DARM.DC 1 1e-14 offset=4m enabled=false
    """)
```

Define the locking scheme for each DoF.

Note that this step is performed in the initialization of the Virgo model when using the finesse-virgo package.

Optimize lock gains

Here we compute the gain of the PDH signal at the DC value of the TF from the DoF motion to the PDH response.

```
# Longitudinal DoF and readouts:
dof_pds_I_Q = {'DARM': 'B1p_56.I',
               'CARM': 'B2_6.I',
               'MICH': 'B2_56.Q',
               'PRCL': 'B2_8.I',
               'SRCL': 'B2_56.I'}
```

```
sol_4 ={}
for dof in dof_pds_I_Q:
    sol_4[dof] = model3.run(fac.FrequencyResponse(f = np.geomspace(1e-2, 1, 5),
                                                  inputs = (dof),
                                                  outputs = (dof_pds_I_Q[dof])))
```

Then, we adjust the gain to be the reciprocal of the optical gain at DC.

```
## Optimize the gain of the lock:
pdh_gain = {}
lock_gain = {}
for dof in dof_pds_I_Q:
    pdh_gain[dof] = np.abs(sol_5[dof][dof_pds_I_Q[dof], dof][0]) # Gain of the PDH signal in W/m
    lock_gain[dof] = 1/pdh_gain[dof] # Optimal gain of the lock
    lock_gain[dof] *= 2*np.pi/ model5.lambda0*np.rad2deg(1) # scaling the gain from m/W to deg/W
    #model5.get(f'{dof}_lock.gain') = lock_gain[dof]
    model5.set(f'{dof}_lock.gain', lock_gain[dof])
    print(f'{dof} gain: ',  model5.get(f'{dof}_lock.gain'))
```

```
DARM gain:  0.010480253074797585
CARM gain:  0.2606994562525331
MICH gain:  3653.7214307274107
PRCL gain:  98.1426349359976
SRCL gain:  6162.736040814418
```

Now we can run the locks.

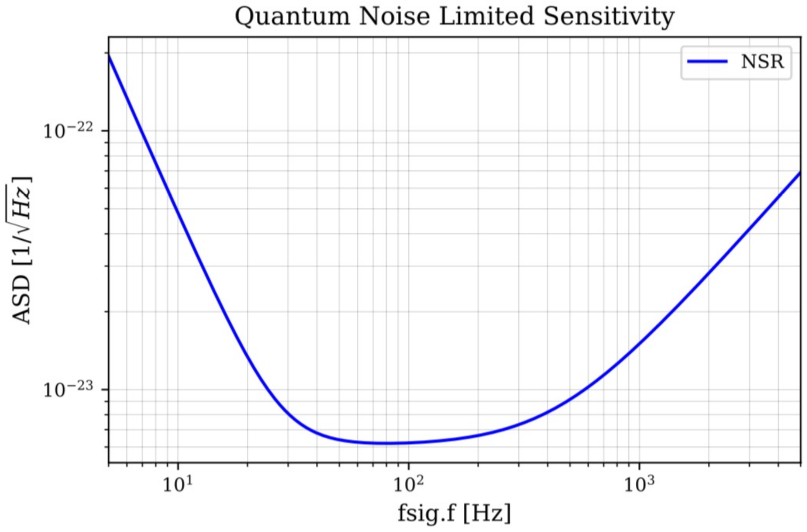
Run locks

We run the locks with the current readout of DARM.

* Other figures of merit? Green lights, sensitivity in Mpc…

```
virgo.plot_QNLS()
out4 = virgo.get_QNLS()
range_bns4 = inspiral_range.range(out4.x1, np.abs(out4['NSR_with_RP'])**2)
print('BNS range:', np.round(range_bns4, 1), 'Mpc')
```

BNS range: 162.9 Mpc



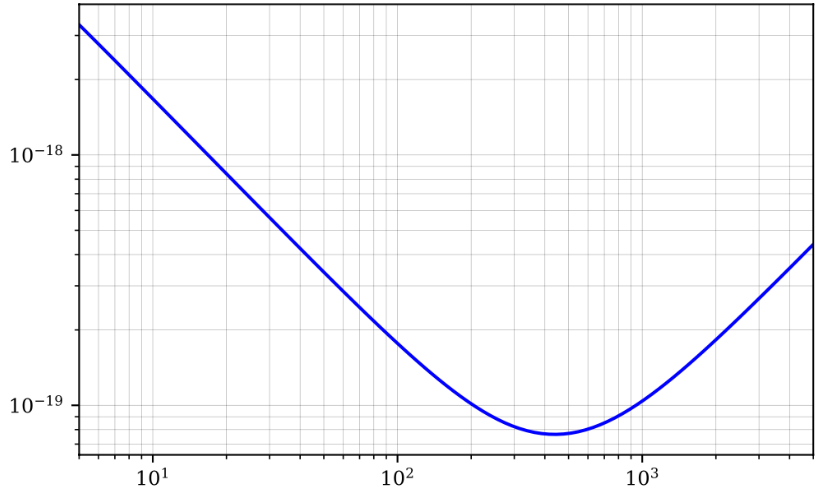

```
model5.run(fac.RunLocks(method='newton'))
```

```
DARM_lock ✔ CARM_lock ✔ MICH_lock ✔ PRCL_lock ✔ SRCL_lock ✔ |██████| 9997/10000
<RunLocksSolution of run locks @ 0x2914659a0 children=0>
```

Now the system is adjusted to an operating point (Heterodyne up to now), defined as the point where the PDH error signals are zero. Now let us check the figures of merit again.

```
out_sensitivity5 = get_QNLS(model5)
plt.loglog(out_sensitivity5.x1, np.abs(out_sensitivity5['NSR_with_RP']))
range_bns5 = inspiral_range.range(out_sensitivity5.x1, np.abs(out_sensitivity5['NSR_with_RP'])**2)
print('BNS range:', np.round(range_bns5, 3), 'Mpc')
```

BNS range: 0.005 Mpc



Change DARM from RF to DC readout

These steps are equivalent to `DARM_RF_to_DC(dc_offset = 0.005)`

from the finesse-virgo package.

```
model6 = model5.deepcopy()
# Check RF locking:
model6.DARM_lock.enabled.value
```

```
True
```

```
## Turn off the RF readout:
model6.DARM_lock.enabled = False
# Check the DC value of DARM:
print(model6.DARM.DC)
```

```
-1.1184696778681427e-07
```

```
# kick lock away from zero tuning for DC lock to grab with
dc_offset = 0.005
model6.DARM.DC += dc_offset
print(model6.DARM.DC)
```

```
0.004999888153032214
```

```
# Check current gain and
print(model6.DARM_dc_lock.gain)
```

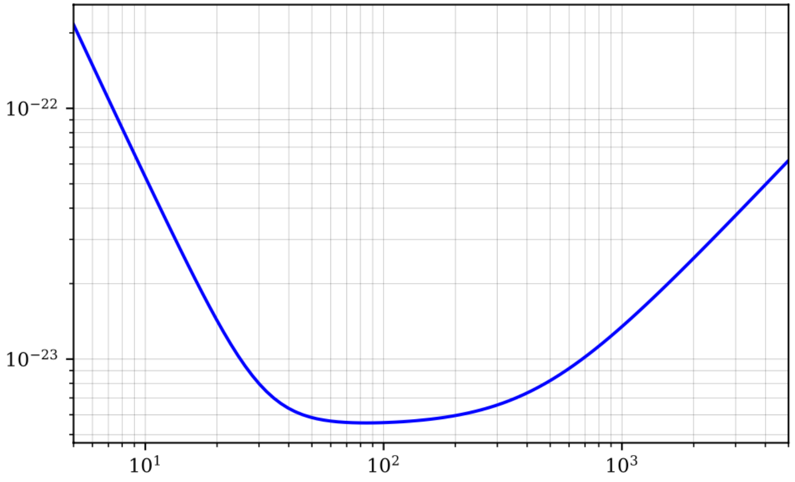
```
1.0
```

```
# take a guess at the gain
lock_gain=-0.01
model6.DARM_dc_lock.gain = lock_gain
model6.DARM_dc_lock.enabled = True
```

Check if this step gives us a good sensitivity level.

```
out_sensitivity6 = get_QNLS(model6)
plt.loglog(out_sensitivity6.x1, np.abs(out_sensitivity6['NSR_with_RP']))
range_bns6 = inspiral_range.range(out_sensitivity6.x1, np.abs(out_sensitivity6['NSR_with_RP'])**2)
print('BNS range:', np.round(range_bns6, 3), 'Mpc')
```

```
BNS range: 169.139 Mpc
```



* Other figures of merit when designing? Green lights, sensitivity in Mpc...

Compute TF and cross couplings.

We can compute these TFs with the FrequencyResponse() from finesse analysis actions.

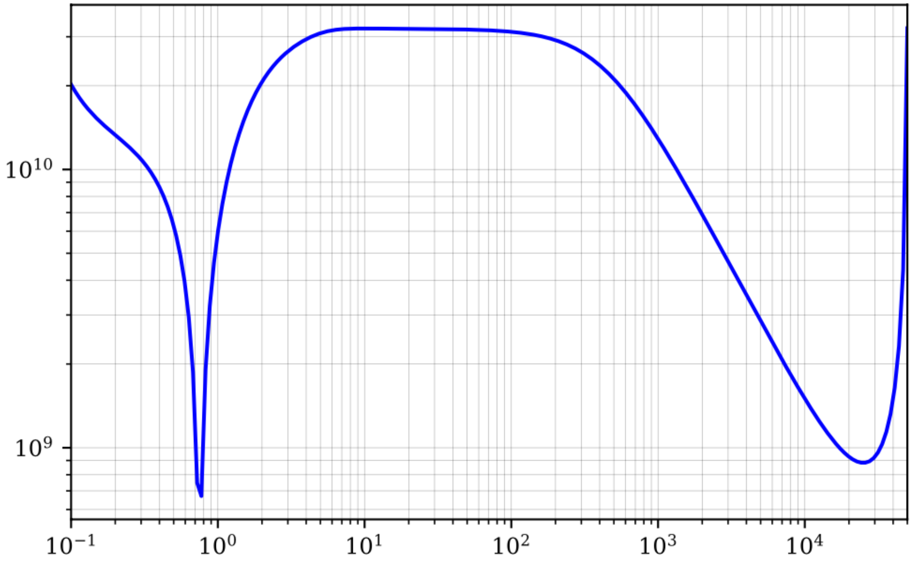
Note that its usage has changed.

```
model7 = model6.deepcopy()
sol7 = model7.run(fac.FrequencyResponse(
                    np.geomspace(0.1, 5e4, 200),
                    ['DARM','CARM','MICH','PRCL','SRCL'],
                    ['B1.DC','B1p_56.I', 'B2_6.I', 'B2_56.Q', 'B2_8.I', 'B2_56.I']))
```

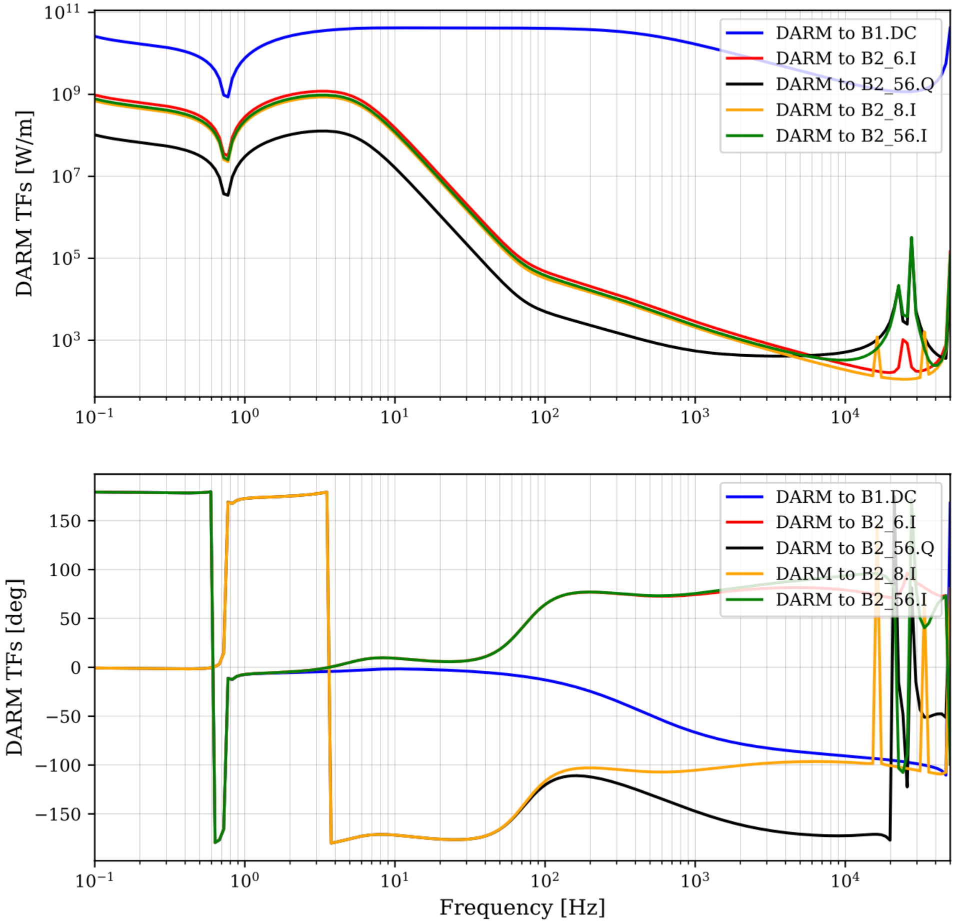
```
plt.loglog(sol7.f, np.abs(sol7['DARM', 'B1p_56.I']))
```

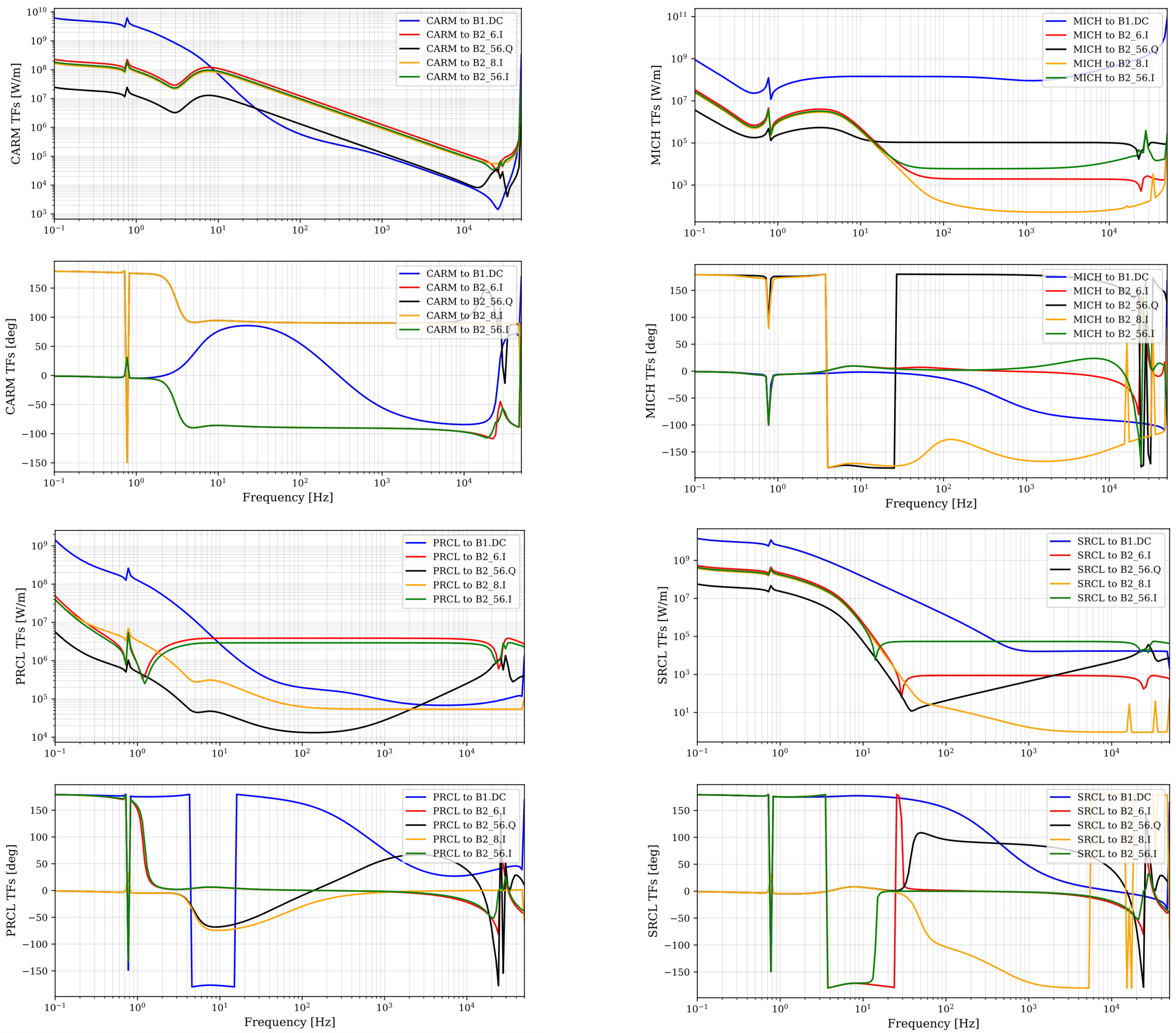
```
/var/folders/vf/trr1b62j04x_tczb622t_vzh0000gn/T/ipykernel_34840/4274949489.py:1: DeprecationWarnin
g: FrequencyResponseSolution has changed to use [output, input], you seemed to have used [input, out
put] so returning that.
  plt.loglog(sol7.f, np.abs(sol7['DARM', 'B1p_56.I']))
```

```
[<matplotlib.lines.Line2D at 0x291b563e0>]
```



Compute TF and cross couplings.





Repository:
https://git.ligo.org/virgo/isc/finesse/enzo_tapia/-/tree/main/2024/LSC_cross-couplings