Skogestad Python Documentation

Release 1

University of Pretoria CBT students

May 24, 2018

Contents:

1 heading **5**

2 Block diagram algebra **13**

3 State space example **19**

4 Frequency response **21**
4.1 Multivariable Nyquist 23

5 Infinity norm **25**

6 Eigenvalue problem **27**

7 Multivariable Nyquist plot **33**

8 Singular values over frequency **35**

9 Optimisation **39**

10 Robust stability **43**

11 Minimised singular value **47**

12 Uncertainty **51**
12.1 Optimisation of the maximum 54
12.2 Addednum: understanding the use of partial 56

13 Indices and tables **57**
13.1 Default parameters 73
13.2 Example 73

Python Module Index **81**

```
In [9]: 2121
        1 + 1*2 + 5**2

Out[9]: 28

In [16]: a = 2
         b = 2
         c = a

In [17]: a

Out[17]: 2

In [18]: b

Out[18]: 2

In [19]: c

Out[19]: 2

In [20]: a = 3

In [21]: c

Out[21]: 2

In [22]: a = [1, 2, 3]

In [23]: type(c)

Out[23]: int

In [24]: b = a

In [25]: a[0] = 5

In [26]: a

Out[26]: [5, 2, 3]

In [27]: b

Out[27]: [5, 2, 3]

In [32]: 100000 + 2000000

Out[32]: 2100000

In [33]: _

Out[33]: 2100000

In [30]: 34**44

Out[30]: 24270143818756646910938690328635932139549939061012471930088572583936

In [46]: a = 100
         b = 100

In [47]: a is b

Out[47]: True

In [53]: a = [1, "aaa", 3]

In [54]: a + a

Out[54]: [1, 'aaa', 3, 1, 'aaa', 3]

In [56]: a = [[1, 2, 3],
              [10, 20, 30]]

In [62]: b = a[0]
```

```
In [63]:
         b[0]

Out[63]: 1

In [64]: import numpy

In [66]: numpy.gradient?

In [71]: a = numpy.array([1, 2, 3])

In [72]: a

Out[72]: array([1, 2, 3])

In [73]: a.shape

Out[73]: (3,)

In [75]: e = a[0]

In [76]: type(e)

Out[76]: numpy.int64

In [77]: a.shape

Out[77]: (3,)

In [78]: e.shape

Out[78]: ()

In [79]: a = numpy.array([[1, 2, 3], [10, 20, 30]])

In [80]: a

Out[80]: array([[ 1,  2,  3],
                [10, 20, 30]])

In [81]: a.shape

Out[81]: (2, 3)

In [86]: b = a[0]
         b

Out[86]: array([1, 2, 3])

In [85]: b[0]

Out[85]: 1

In [88]: %%timeit

The slowest run took 31.38 times longer than the fastest. This could mean that an intermediate result
1000000 loops, best of 3: 277 ns per loop

In [90]: %%timeit
         a[0, 0]

The slowest run took 32.69 times longer than the fastest. This could mean that an intermediate result
10000000 loops, best of 3: 148 ns per loop

In [92]: a

Out[92]: array([[ 1,  2,  3],
                [10, 20, 30]])

In [97]: a[1, 0:2]

Out[97]: array([10, 20])

In [98]: 0:2
```

```
  File "<ipython-input-98-e66af9ff3e2d>", line 1
    0:2
     ^
SyntaxError: invalid syntax


In [99]: l = [1, 2, 3]

In [101]: l[1:2]

Out[101]: [2]

In [106]: a[:, 1:2].T

Out[106]: array([[ 2, 20]])

In [107]: a[1, :].T

Out[107]: array([10, 20, 30])

In [108]: a

Out[108]: array([[ 1,  2,  3],
                 [10, 20, 30]])

In [112]: A = numpy.matrix(a)

In [113]: A

Out[113]: matrix([[ 1,  2,  3],
                  [10, 20, 30]])

In [123]: a.dot(a.T)

Out[123]: array([[  14,  140],
                 [ 140, 1400]])
```

I can type text here

- bullet
- bullet

CHAPTER 1

heading

$Ax = \beta$

```
In [124]: A = numpy.array([[1, 2], [4, 2]])

In [125]: beta = numpy.array([[2], [2]])

In [132]: numpy.linalg.inv(A).dot(beta)

Out[132]: array([[ 0.],
                 [ 1.]])

In [131]: numpy.linalg.solve(A, beta)

Out[131]: array([[ 0.],
                 [ 1.]])

In [133]: A = numpy.matrix(A)

In [134]: beta = numpy.matrix(beta)

In [136]: A.I*beta

Out[136]: matrix([[ 0.],
                  [ 1.]])

In [137]: import matplotlib.pyplot as plt

In [138]: %matplotlib notebook

In [139]: plt.plot([1, 2, 3], [1, 2, 1])

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[139]: [<matplotlib.lines.Line2D at 0x10b38bba8>]

In [140]: x = numpy.linspace(0, numpy.pi*2)

In [141]: y = numpy.sin(x)

In [142]:
          plt.plot(x, y)
```

```
Out[142]: [<matplotlib.lines.Line2D at 0x10b15b320>]

In [144]: plt.figure()
          plt.plot(x, y)

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[144]: [<matplotlib.lines.Line2D at 0x10ba88cf8>]

In [147]: a = [100, 1, 3]

In [153]: for i in range(0, len(a)):
              print(a[i])

100
1
3

In [159]: for e in a:
              print(e)
              print("another")
          print("done")

100
another
1
another
3
another
done

In [160]: a = 20
          while a > 10:
              a -= 1

In [168]: if a > 10 and a < 3:
              print("Nope")
          else:
              print("Yup")

Yup

In [170]: a = 5

In [172]: 1 < a < 10 < 50 < 100 #=> 1<a and a < 10

Out[172]: True

In [186]: b = 5

In [187]: def f(a):
              return a + b

In [188]: f(2)

Out[188]: 7

In [190]: b = 2

In [191]: f(2)

Out[191]: 4

In [192]: def functionprinter(g):
              print(g(1))

In [194]: functionprinter(f)
```

```
3
```

```
In [196]: len([1, 2, 3])
```

```
Out[196]: 3
```

```
In [197]: a = len
```

```
In [198]: a([1, 2, 3])
```

```
Out[198]: 3
```

```
In [204]: functions = [numpy.sin, f, numpy.cos, 2]
```

```
In [226]: for function in functions:
              print(function(1))
```

```
0.841470984808
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-226-08426fa1b189> in <module>()
      1 for function in functions:
----> 2     print(function(1))

TypeError: 'int' object is not callable
```

```
In [227]: things = []
```

```
In [228]: mylist = range(20)
```

```
In [229]: for i in mylist:
              if i > 10:
                  things.append(i**2)
```

```
In [230]: things
```

```
Out[230]: [121, 144, 169, 196, 225, 256, 289, 324, 361]
```

```
In [231]: [i**2 for i in mylist if i > 10 ]
```

```
Out[231]: [121, 144, 169, 196, 225, 256, 289, 324, 361]
```

```
In [233]: things
```

```
Out[233]: [121, 144, 169, 196, 225, 256, 289, 324, 361]
```

```
In [236]: def f(a):
              return a + 1
```

```
In [237]: [f(i) for i in things]
```

```
Out[237]: [122, 145, 170, 197, 226, 257, 290, 325, 362]
```

$f(x) = 2$

```
In [10]: def f(x):
             return 2
```

```
In [4]: f(2)
```

```
Out[4]: 2
```

```
In [5]: -f(2)
```

```
Out[5]: -2
```

```
In [6]: def functionprinter(g):
            print(g(2))
```

```
In [14]: functionprinter(f)
```

```
*
*
None
```

```
In [12]: def f(x):
             for i in range(x):
                 print('*')
```

```
In [13]: f(5)
```

```
*
*
*
*
*
```

```
In [15]: 1 + 4
```

```
Out[15]: 5
```

```
In [16]: dir(1)
```

```
Out[16]: ['__abs__',
          '__add__',
          '__and__',
          '__bool__',
          '__ceil__',
          '__class__',
          '__delattr__',
          '__dir__',
          '__divmod__',
          '__doc__',
          '__eq__',
          '__float__',
          '__floor__',
          '__floordiv__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getnewargs__',
          '__gt__',
          '__hash__',
          '__index__',
          '__init__',
          '__int__',
          '__invert__',
          '__le__',
          '__lshift__',
          '__lt__',
          '__mod__',
          '__mul__',
          '__ne__',
          '__neg__',
          '__new__',
          '__or__',
          '__pos__',
          '__pow__',
          '__radd__',
          '__rand__',
          '__rdivmod__',
          '__reduce__',
          '__reduce_ex__',
```

```
          '__repr__',
          '__rfloordiv__',
          '__rlshift__',
          '__rmod__',
          '__rmul__',
          '__ror__',
          '__round__',
          '__rpow__',
          '__rrshift__',
          '__rshift__',
          '__rsub__',
          '__rtruediv__',
          '__rxor__',
          '__setattr__',
          '__sizeof__',
          '__str__',
          '__sub__',
          '__subclasshook__',
          '__truediv__',
          '__trunc__',
          '__xor__',
          'bit_length',
          'conjugate',
          'denominator',
          'from_bytes',
          'imag',
          'numerator',
          'real',
          'to_bytes']
```

```
In [18]: (1).__add__(2)
```

```
Out[18]: 3
```

```
In [70]: class MyNumber:
             def __init__(self, value):
                 self.value = value
             def __repr__(self):
                 return 'MyNumber("{}")'.format(self.value)
             def __str__(self):
                 return self.value
             def __add__(self, other):
                 if self.value == 'one' and other.value == 'one':
                     return MyNumber('two')
```

```
In [71]: one = MyNumber('one')
```

```
In [72]: one + one
```

```
Out[72]: MyNumber("two")
```

```
In [73]: one + one + one
```

```
In [74]: a = 1, 2, 3
```

```
In [80]: a = 1, 2, 3
```

```
In [81]: a, b = 1, 2
```

```
In [82]: c = 1, 2
```

```
In [83]: a, b = c
```

```
In [76]: type(a)
```

```
Out[76]: tuple

In [77]: a

Out[77]: (1, 2, 3)

In [78]: b = [1, 2, 3]

In [79]: type(b)

Out[79]: list

In [86]: a = ((((((1))))))

In [87]: type(a)

Out[87]: int

In [89]: a = 1,

In [90]: a

Out[90]: (1,)

In [91]: import numpy

In [93]: t = (1, 2)
         z = numpy.zeros(t)

In [101]: a = [1, 2, 3]
          b = a
          b[1] = 999
          a

Out[101]: [1, 999, 3]

In [99]: a is b

Out[99]: False

In [102]: a = 1
          b = a

In [103]: b is a

Out[103]: True

In [104]: b = 2

In [107]: a = (1, 2, 3)
          b = a
          b[2] = 999

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-107-cf1c42a0b4da> in <module>()
      1 a = (1, 2, 3)
      2 b = a
----> 3 b[2] = 999

TypeError: 'tuple' object does not support item assignment

In [109]: a + a

Out[109]: (1, 2, 3, 1, 2, 3)

In [110]: b = [1, 2, 4]

In [111]: b.append(3)

In [112]: b
```

```
Out[112]: [1, 2, 4, 3]

In [113]: s = '1234'
```

CHAPTER 2

Block diagram algebra

Example 3.1:

The textbook shows the solution as

$$z = (P_{11} + P_{12}K(I - P_{22}K)^{-1}P_{21})w$$

We can do this calculation manually as follows:

$$b = a + P_{22}c \tag{2.1}$$
$$c = Kb \tag{2.2}$$
$$c = Ka + KP_{22}c \tag{2.3}$$
$$c - KP_{22}c = Ka \tag{2.4}$$
$$(I - KP_{22})c = Ka \tag{2.5}$$
$$(I - KP_{22})^{-1}(I - KP_{22})c = (I - KP_{22})^{-1}Ka \tag{2.6}$$
$$c = (I - KP_{22})^{-1}Ka \tag{2.7}$$
$$= K(I - P_{22}K)^{-1}a \text{ (push-through)} \tag{2.8}$$
$$z = f + e \tag{2.9}$$
$$= P_{21}w + P_{12}c \tag{2.10}$$
$$= (P_{21} + P_{12}\underbrace{K(I - P_{22}K)^{-1}P_{12}}_{c})w \tag{2.11}$$

We should also be able to solve this problem symbolically using Sympy

```
In [1]: import sympy
        sympy.init_printing()
```

```
In [5]: (w, a, b, c,
         d, e, f, z,
         P_21, P_22,
         K, P_12, P_11) = sympy.symbols("""w, a, b, c,
                                           d, e, f, z,
                                           P_21, P_22,
```

```
                                        K, P_12, P_11""", commutative=False
                                    )
        eqs = [a - P_21*w,
               b - (a + d),
               c - K*b,
               d - P_22*c,
               e - P_12*c,
               f - P_11*w,
               z - (e + f)]
```

Note this is the right call, unfortunately as of 2018-03-15, this causes an error.

```
In [7]: # Uncomment to check if this solves now
        # sympy.solve(eqs, (a, b, c, d, e, f, z))
```

If we are careful about the ordering, we can still get a sensible answer from SymPy without having to do tedious math ourselves.

```
In [8]: # first we identify which variable can be eliminated by solving each equation
        order = [a, b, c, d, e, f, z]
        # This will store the solutions
        solution = {}
        for var, eq in zip(order, eqs):
            # We solve each equation for the new unknown given all the other solutions
            sol = sympy.solve(eq.subs(solution), var)[0]
            # and add that solution to the list of knowns
            solution[var] = sol
        solution[z].collect(w)
```

$$w\left(P_{11} + P_{12}KP_{21} + P_{12}K\left(1 - P_{22}K\right)^{-1}P_{22}KP_{21}\right)$$

```
In [1]: import sympy
```

```
In [2]: sympy.init_printing()
```

```
In [3]: x = sympy.Symbol('x')
```

```
In [4]: sympy.expand((x + 2)**4)
```

$$x^4 + 8x^3 + 24x^2 + 32x + 16$$

```
In [8]: a = sympy.solve(x**2 + 2, x)
```

```
In [9]: len(a)
```

$$2$$

```
In [10]: [sympy.N(e) for e in a]
```

$$[-1.4142135623731i, \quad 1.4142135623731i]$$

```
In [11]: list(map(sympy.N, a))
```

$$[-1.4142135623731i, \quad 1.4142135623731i]$$

```
In [14]: sympy.laplace_transform?
```

```
In [15]: s, t = sympy.symbols('s t')
```

```
In [16]: sympy.laplace_transform(sympy.sin(t) + 5, t, s, noconds=True)
```

$$\frac{5s^2 + s + 5}{s\left(s^2 + 1\right)}$$

```
In [17]: def L(g):
             return sympy.laplace_transform(g, t, s, noconds=True)
```

```
In [18]: L(sympy.sin(t) + 5)
```

$$\frac{5s^2 + s + 5}{s\left(s^2 + 1\right)}$$

```
In [19]: sympy.inverse_laplace_transform(1/(s + 1), s, t)
```

$$e^{-t}\theta\left(t\right)$$

```
In [20]: G = (s + 1)/((s+2)*(s+3)*(s+4))

In [21]: type(G)

Out[21]: sympy.core.mul.Mul

In [22]: s = sympy.Symbol('s')
         t = sympy.Symbol('t')

In [23]: t = sympy.Symbol('t', positive=True)

In [24]: variables = [s, t]

In [25]: mu = sympy.Symbol('s')

In [26]: mu
```

$$s$$

```
In [27]: for varaiable in variables:
             print(variable)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-25d0663afbe9> in <module>()
      1 for varaiable in variables:
----> 2     print(variable)

NameError: name 'variable' is not defined

In [28]: sympy.inverse_laplace_transform(G, s, t)
```

$$\frac{1}{2e^{4t}}\left(-e^{2t} + 4e^{t} - 3\right)$$

```
In [29]: sympy.apart(G)
```

$$-\frac{3}{2s+8} + \frac{2}{s+3} - \frac{1}{2s+4}$$

```
In [30]: sympy.solve(t**2 + 2, t)
```

$$[]$$

```
In [61]: from sympy.abc import a, b, c

In [62]: eq1 = a + b - c
         eq1
```

$$a + b - c$$

```
In [90]: eq2 = 2*a + b - 5*c*b
         eq2
```

$$2a - 5bc + b$$

```
In [91]: result = sympy.solve(eq2, b)

In [92]: result
```

$$\left[\frac{2a}{5c-1}\right]$$

```
In [ ]: solb = result[0]
        solb, = result
        (solb,) = result
        [solb] = result
```

```
In [84]: eq1.subs({b: result[0]})
```

$$a + \frac{2a}{5c-1} - c$$

```
In [93]: result = sympy.solve([eq1, eq2], [a, b])
```

```
In [94]: result
```

$$\left\{a : \frac{c\,(5c-1)}{5c+1}, \quad b : \frac{2c}{5c+1}\right\}$$

```
In [85]: d = {}
```

```
In [ ]: d['name1'] = 1
        d['name1'] = 2
```

```
In [73]: d = {'name1': 1,
              'name1': 2}
```

```
In [74]: d['name1']
```

$$2$$

```
In [36]: lst = [1, 2, 3]
```

```
In [37]: type(lst)
```

```
Out[37]: list
```

```
In [38]: tup = 1, 2, 3
```

```
In [39]: type(tup)
```

```
Out[39]: tuple
```

```
In [40]: tup2 =(((((1, 2, 3)))))
```

```
In [41]: type(tup2)
```

```
Out[41]: tuple
```

```
In [42]: tup, tup2
```

$$((1,\quad 2,\quad 3)\,,\quad (1,\quad 2,\quad 3))$$

```
In [43]: a, (b, c) = 1, [2, 3]
```

```
In [44]: def f(x):
             return x, x+2
```

```
In [45]: c = f(x)
```

```
In [46]: t = 1, 2, 3, 4, 5, 6
```

```
In [47]: b, *bs = t
```

```
In [48]: b
```

$$1$$

```
In [49]: bs
```

$$[2,\quad 3,\quad 4,\quad 5,\quad 6]$$

```
In [50]: def myfunction(many, many1, many2, params):
             print(many, many1, many2, params)
```

```
In [51]: lst = [1, 2, 3, 4]

In [52]: myfunction(5, *lst[1:])

5 2 3 4

In [53]: def myfunction(*args):
             print(args)

In [54]: myfunction(1, 2, 3)

(1, 2, 3)

In [112]: s = sympy.Symbol('s')
          t = sympy.Symbol('t')

In [165]: def G(s):
              return 1/(s + 1)
              #return (s + 2)/(s**2 + 0.25*s + 1)

In [166]: G(1)
```

$$0.5$$

```
In [167]: sympy.solve(sympy.denom(G(s)), s)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-167-0fb4ff7ceec1> in <module>()
----> 1 sympy.solve(sympy.denom(G(s)), s)

/Users/alchemyst/anaconda3/lib/python3.5/site-packages/sympy/simplify/radsimp.py in denom(expr)
    983
    984 def denom(expr):
--> 985     return fraction(expr)[1]
    986
    987

/Users/alchemyst/anaconda3/lib/python3.5/site-packages/sympy/simplify/radsimp.py in fraction(expr, ex
    948
    949     """
--> 950     expr = sympify(expr)
    951
    952     numer, denom = [], []

/Users/alchemyst/anaconda3/lib/python3.5/site-packages/sympy/core/sympify.py in sympify(a, locals, co
    280         try:
    281             return type(a)([sympify(x, locals=locals, convert_xor=convert_xor,
--> 282                 rational=rational) for x in a])    283         except TypeError:
    284             # Not all iterables are rebuildable with their type.

ValueError: sequence too large; cannot be greater than 32

In [ ]: f = sympy.inverse_laplace_transform(G(s), s, t)

In [168]: sympy.plot(f, (t, 0.1, 10))

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[168]: <sympy.plotting.plot.Plot at 0x10ff52710>

In [137]: import numpy

In [169]: omega = numpy.logspace(-1, 2)
```

```
In [170]: s = 1j*omega

In [175]: mag = numpy.abs(G(s))
          phase = numpy.angle(G(s))

In [176]: import matplotlib.pyplot as plt

In [177]: %matplotlib notebook

In [178]: plt.subplot(2, 1, 1)
          plt.loglog(omega, mag)
          plt.subplot(2, 1, 2)
          plt.semilogx(omega, phase)

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[178]: [<matplotlib.lines.Line2D at 0x110aaa1d0>]

In [181]: plt.figure()
          plt.plot(G(s).real, G(s).imag)
          plt.plot(G(s).real, -G(s).imag)
          plt.axis('equal')

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

$$(0.0, \quad 1.0, \quad -0.6, \quad 0.6)$$

CHAPTER 3

State space example

We take a simple system and integrate it numerically

```
In [2]: A = matrix([[0, 1],
                    [-5, -4]])
        B = matrix([[0],
                    [1]])
        C = matrix([[1, 0]])
        D = matrix([[0]])

        ts = linspace(0, 5, 1000)
        dt = ts[1]

        def u(t):
            if t < 0:
                return matrix([[0]])
            else:
                return matrix([[1]])

        x0 = matrix([[0],
                     [0]])

        ys = zeros_like(ts)
```

```
In [3]: %%timeit
        x = matrix(x0)
        for i, t in enumerate(ts):
            # Evaluate state-space form
            dxdt = A*x + B*u(t)
            y = C*x + D*u(t)

            # Do integration
            x = x + dxdt*dt

            # store result
            ys[i] = y[0,0]
```

```
1 loops, best of 3: 108 ms per loop
```

Then analytically using the matrix exponential

```
In [4]: from scipy.linalg import expm
        y_analytic = zeros_like(ts)
        b0 = solve(A, -B)
```

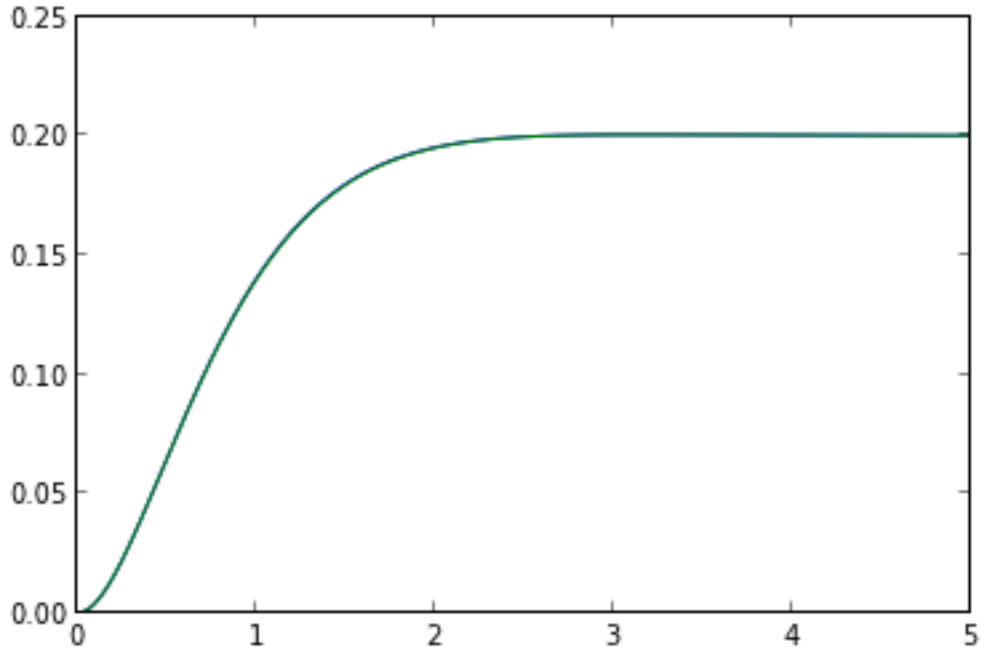
```
In [5]: %%timeit
        for i, t in enumerate(ts):
            y = expm(A*t)*b0
            y_analytic[i] = b0[0] - y[0,0]
```

```
1 loops, best of 3: 217 ms per loop
```

The "analytic" method is slower, but no more accurate

```
In [6]: plot(ts, ys, ts, y_analytic)
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x108f1a450>,
         <matplotlib.lines.Line2D at 0x108f095d0>]
```




```
In [14]: import numpy
         import matplotlib.pyplot as plt
         %matplotlib inline
```

CHAPTER 4

Frequency response

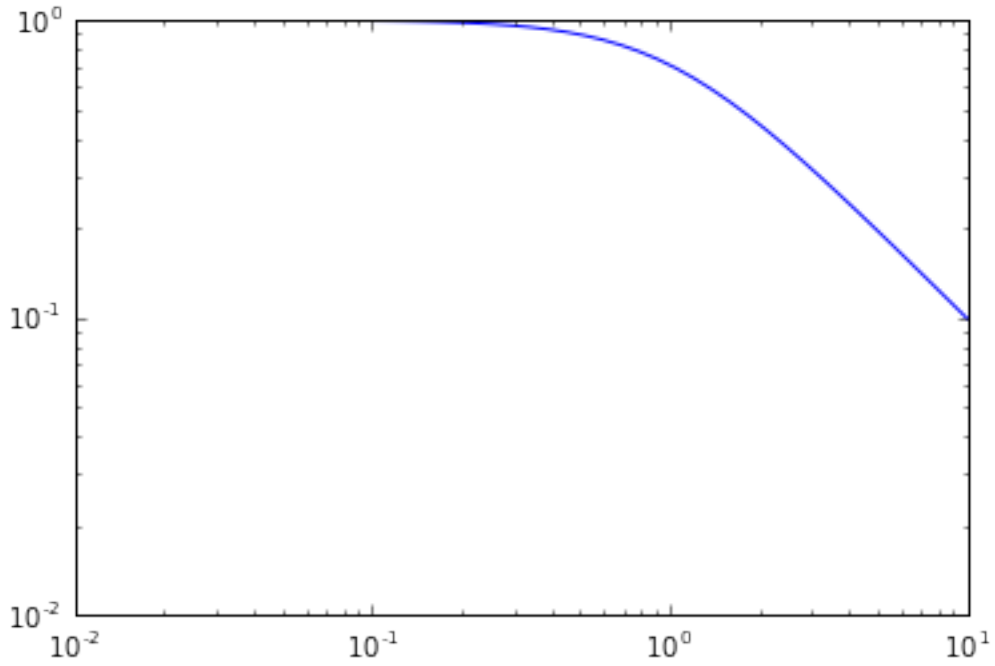
```
In [15]: def G(s):
             return 1/(s + 1)
```

```
In [16]: omega = numpy.logspace(-2, 1)
```

```
In [17]: s = 1j*omega
```

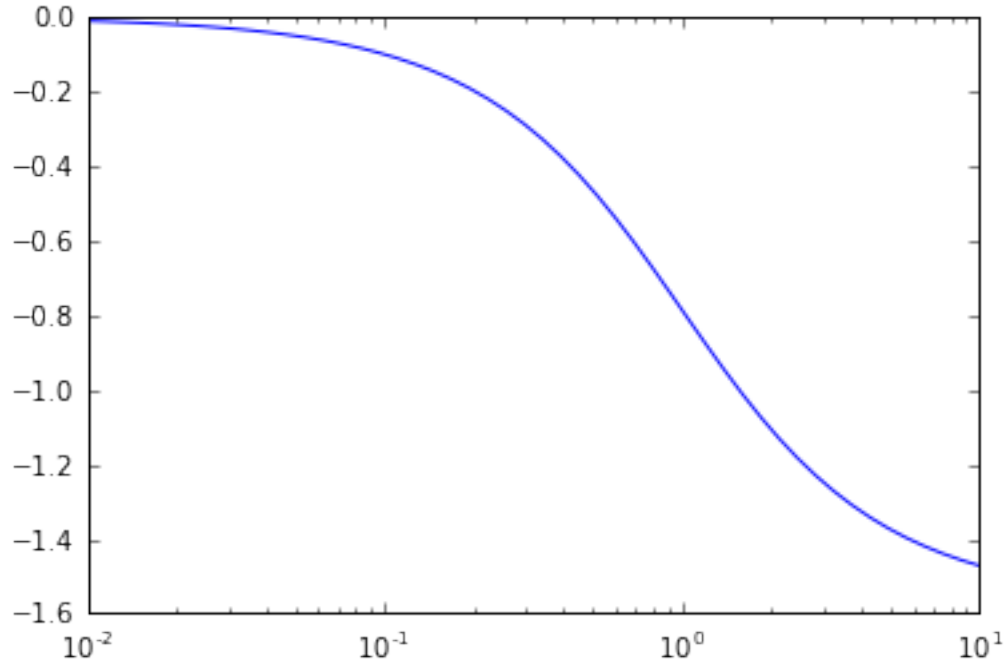
```
In [18]: plt.loglog(omega, numpy.abs(G(s)))
```

```
Out[18]: [<matplotlib.lines.Line2D at 0x10e63a390>]
```

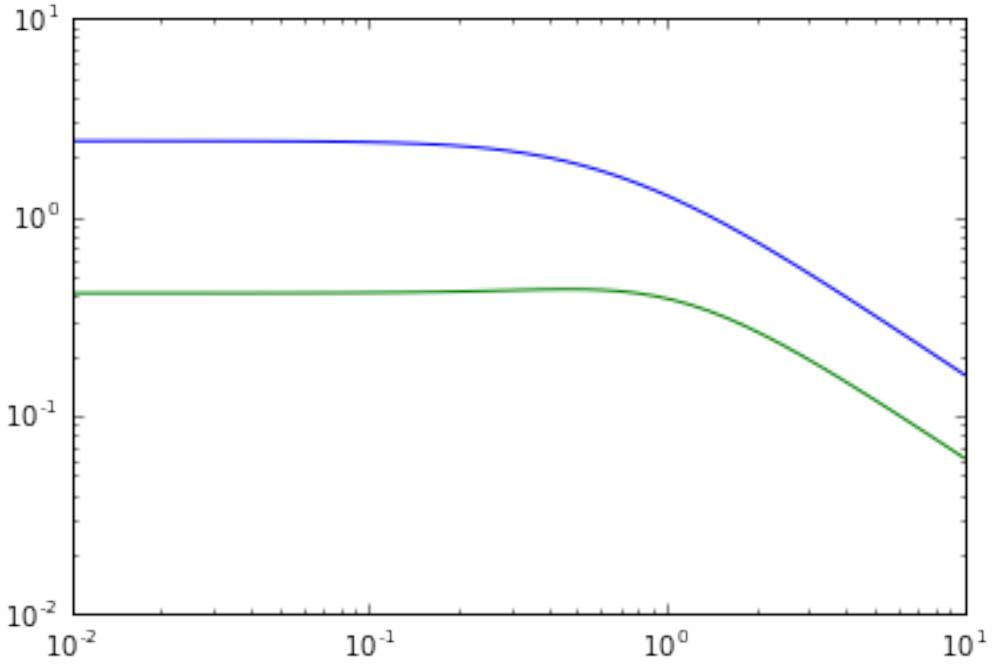



```
In [19]: plt.semilogx(omega, numpy.angle(G(s)))
```

```
Out[19]: [<matplotlib.lines.Line2D at 0x10f553f60>]
```




```
In [20]: def G(s):
             return numpy.matrix([[1/(s + 1), 0],
                                  [2/(2*s + 1), 1/(s + 1)]])

In [21]: G(s[3])

Out[21]: matrix([[ 0.99976706-0.01526062j,  0.00000000+0.j        ],
                 [ 1.99813777-0.06099987j,  0.99976706-0.01526062j]])

In [22]: r = []
         for si in s:
             r.append(G(si))

In [23]: r = [G(si)[0,0] for si in s]

In [24]: r = map(G, s)

In [25]: sigmas = numpy.array([numpy.linalg.svd(G(si))[1] for si in s])

In [26]: plt.loglog(omega, sigmas)

Out[26]: [<matplotlib.lines.Line2D at 0x10f58e860>,
          <matplotlib.lines.Line2D at 0x10ef69f60>]
```

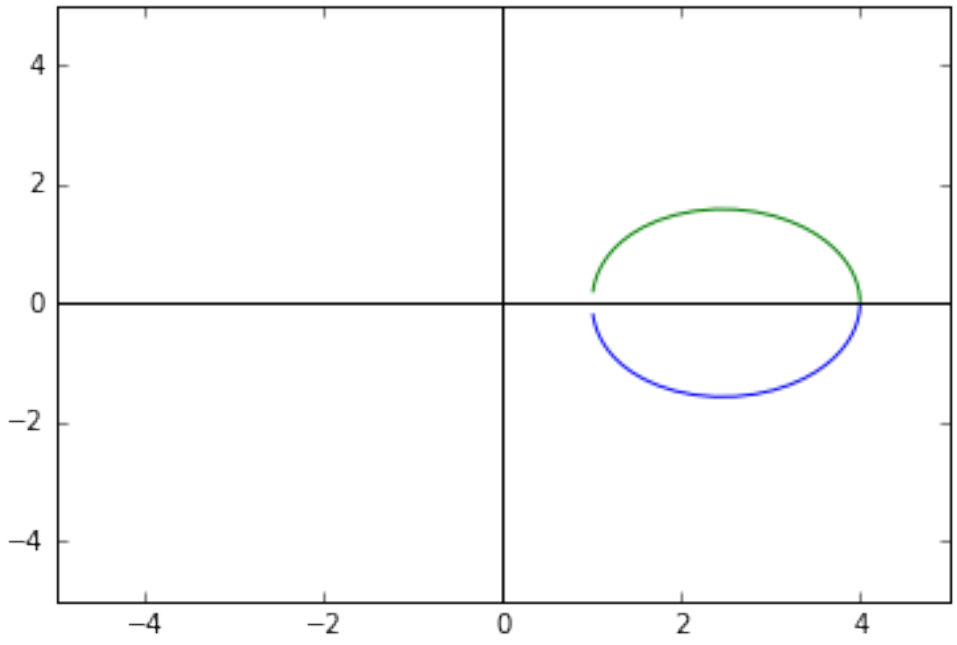


4.1 Multivariable Nyquist

```
In [27]: I = numpy.eye(2)
         dets = [numpy.linalg.det(I + G(si)) for si in s]
```

```
In [28]: plt.plot(numpy.real(dets), numpy.imag(dets))
         plt.plot(numpy.real(dets), -numpy.imag(dets))
         plt.axis([-5, 5, -5, 5])
         plt.axvline(0, color='black')
         plt.axhline(0, color='black')
```

```
Out[28]: <matplotlib.lines.Line2D at 0x10f8b16d8>
```



$$l = \sqrt{a^2 + b^2}$$

```
In [1]: import numpy
        import matplotlib.pyplot as plt
```

```
In [2]: %matplotlib inline
```

CHAPTER 5

Infinity norm

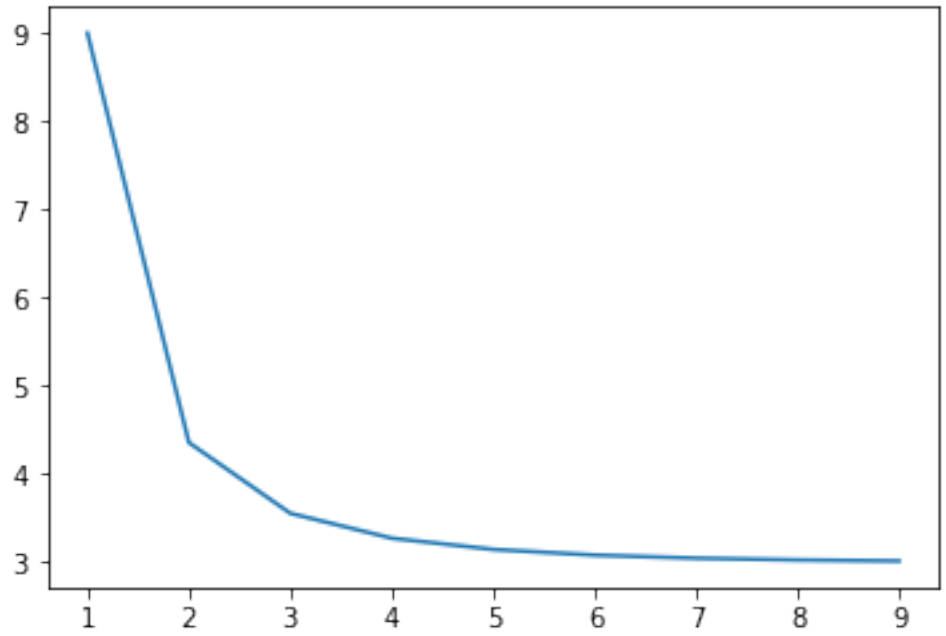
```
In [3]: def norm(v, order):
            return sum([vi**order for vi in v])**(1/float(order))
```

```
In [4]: v = [1, 2, 3, 2, 1]
        norm(v, 10)
```

```
Out[4]: 3.010255858106243
```

```
In [5]: plt.plot(range(1, 10), [norm(v, order) for order in range(1, 10)])
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x11b1295c0>]
```




```
In [6]: G = numpy.matrix([[5, 4],
                          [3, 2]])
```

```
In [7]: d = numpy.array([[1], [1]])

In [8]: G*d

Out[8]: matrix([[9],
                [5]])

In [9]: def gain(theta):
            d = numpy.matrix([[numpy.cos(theta[0])], [numpy.sin(theta[0])]])
            return numpy.linalg.norm(G*d)/numpy.linalg.norm(d)

In [10]: import scipy.optimize

In [11]: scipy.optimize.minimize(lambda theta: -gain(theta), [1])

Out[11]: fun: -7.343420458864692
          hess_inv: array([[ 0.13636409]])
               jac: array([  5.96046448e-08])
           message: 'Optimization terminated successfully.'
              nfev: 18
               nit: 4
              njev: 6
            status: 0
           success: True
                 x: array([ 0.65390079])

In [1]: import numpy
        import matplotlib.pyplot as plt

In [2]: %matplotlib inline
```

CHAPTER 6

Eigenvalue problem

Matrix transformaion is written as

$$\mathbf{y} = A\mathbf{x}$$

A different vector in the same direction can be written as scalar multiplication:

$$\mathbf{y} = \lambda\mathbf{x}$$

Equating these ys yields:

$$A\mathbf{x} = \lambda\mathbf{x} \Rightarrow (A - \lambda I)\mathbf{x} = 0$$

$$\det(A - \lambda I) = 0$$

The eigenvalue problem can also be collected with Λ being a diagonal matrix containing all the eigenvalues and X containing the eigenvectors stacked column-wise. This leads to the eigenvalue decomposition:

$$AX = X\Lambda \Rightarrow A = X\Lambda X^{-1}$$

with

$$\Lambda = diag(\lambda_i)$$

If we try to find a similar decomposition with different constraints, we can write

$$A = UDV^H$$

If D is a diagonal matrix and U and V are unitary, this is the singular value decomposition.

In Skogestad

$$A = U\Sigma V^H$$

$$\Sigma = diag(\sigma_i)$$

```
In [3]: from ipywidgets import interact

In [4]: def plotvector(x, color='blue'):
            plt.plot([0, x[0,0]], [0, x[1,0]], color=color)

In [5]: import matplotlib.patches as patches
```

Let's investigate the properties of this matrix:

```
In [77]: A = numpy.matrix([[4, 3],
                           [2, 1]])
```

The eigenvectors and eigenvalues can be calculated as follows. We also calculate the output vectors associated with a unit vector input in the eigenvector directions.

```
In [78]: A

Out[78]: matrix([[4, 3],
                 [2, 1]])

In [79]: v = numpy.asmatrix(numpy.random.random(2)).T

In [80]: v = A*v

         v = v/numpy.linalg.norm(v)
         v

Out[80]: matrix([[0.89474813],
                 [0.44657115]])

In [81]: lambdas, eigvectors = numpy.linalg.eig(A)
         ev1 = lambdas[0]*eigvectors[:, 0]
         ev2 = lambdas[1]*eigvectors[:, 1]
```

The singular values determine the main axes of the translation ellipse of the matrix. Note that the `numpy.linalg.svd` function returns the conjugate transpose of the input direction matrix.

```
In [82]: U, S, VH = numpy.linalg.svd(A)
         V = VH.H
         ellipseangle = numpy.rad2deg(numpy.angle(complex(*U[:, 0])))

In [83]: def interactive(scale, theta):
             x = numpy.matrix([[numpy.cos(theta)], [numpy.sin(theta)]])
             y = A*x

             plotvector(x)
             plotvector(y, color='red')
             plotvector(ev1, 'green')
             plotvector(ev2, 'green')
             plotvector(V[:, 0], 'magenta')
             plotvector(V[:, 1], 'magenta')
             plt.gca().add_artist(patches.Circle([0, 0], 1,
                                                 color='blue',
                                                 alpha=0.1))
             plt.gca().add_artist(patches.Ellipse([0, 0], S[0]*2, S[1]*2,
                                                  ellipseangle,
                                                  color='red',
                                                  alpha=0.1))
             plt.axis([-scale, scale, -scale, scale])
             plt.axes().set_aspect('equal')
             plt.show()
         interact(interactive, scale=(1., 10), theta=(0., numpy.pi*2))

interactive(children=(FloatSlider(value=5.5, description='scale', max=10.0, min=1.0), FloatSlider(val

Out[83]: <function __main__.interactive>
```

Example 7.9

```
In [2]: import numpy
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [16]: omega = numpy.logspace(-3, 1, 1000)
         s = 1j*omega
```

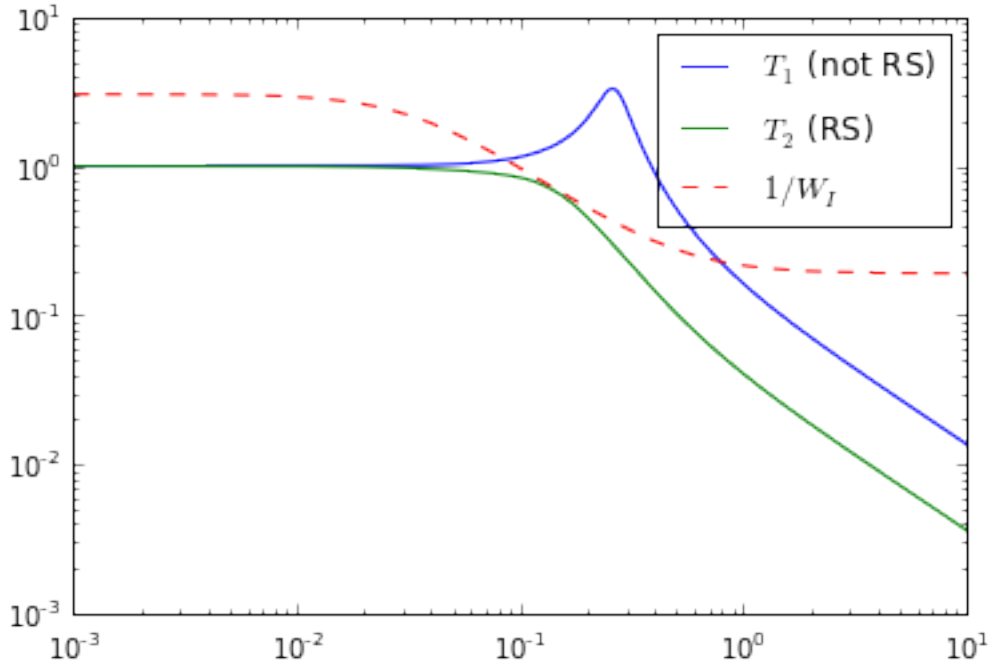
```
In [17]: Kc = 1.13
         G = 3*(-2*s + 1)/((5*s + 1)*(10*s + 1))
         def K(Kc):
             return Kc*(12.7*s + 1)/(12.7*s)
         Gprime = 4*(-3*s + 1)/(4*s + 1)**2
         w_I = (10*s + 0.33)/((10/5.25)*s + 1)
```

```
In [18]: K1 = K(Kc=1.13)
         T1 = G*K1/(1 + G*K1)
```

```
In [19]: K2 = K(Kc=0.3)
         T2 = G*K2/(1 + G*K2)
```

```
In [20]: plt.loglog(omega, numpy.abs(T1), '-', label='$T_1$ (not RS)')
         plt.loglog(omega, numpy.abs(T2), '-', label='$T_2$ (RS)')
         plt.loglog(omega, numpy.abs(1/w_I), '--', label='$1/W_I$')
         plt.legend()
```

```
Out[20]: <matplotlib.legend.Legend at 0x10d7bcdd0>
```




```
In [ ]: e
```

```
In [1]: from sympy import *
        init_printing()
```

```
In [2]: W_P, W_I, G, I, K = var('W_P, W_I, G, I, K', commutative=False)
```

```
In [3]: P = Matrix([[0, 0, W_I],
                    [W_P*G, W_P, W_P*G],
                    [-G, -I, -G]])
```

In [4]: P

$$\begin{bmatrix} 0 & 0 & W_I \\ W_P G & W_P & W_P G \\ -G & -I & -G \end{bmatrix}$$

In [5]: P11 = P[:2, :2]
P11

$$\begin{bmatrix} 0 & 0 \\ W_P G & W_P \end{bmatrix}$$

In [6]: P12 = P[:2,2:]
P12

$$\begin{bmatrix} W_I \\ W_P G \end{bmatrix}$$

In [7]: P21 = P[2:, :2]
P21

$$\begin{bmatrix} -G & -I \end{bmatrix}$$

In [8]: P22 = P[2:,2:]
P22

$$\begin{bmatrix} -G \end{bmatrix}$$

In [9]: Im = Matrix([[I]])

In [10]: (Im + P22*K).inv()

$$\begin{bmatrix} (-GK + I)^{-1} \end{bmatrix}$$

In [11]: N = P11 + P12*K*(Im - P22*K).inv()*P21
N

$$\begin{bmatrix} -W_I K \left(GK + I\right)^{-1} G & -W_I K \left(GK + I\right)^{-1} I \\ W_P G - W_P G K \left(GK + I\right)^{-1} G & W_P - W_P G K \left(GK + I\right)^{-1} I \end{bmatrix}$$

In [50]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
%matplotlib inline
```

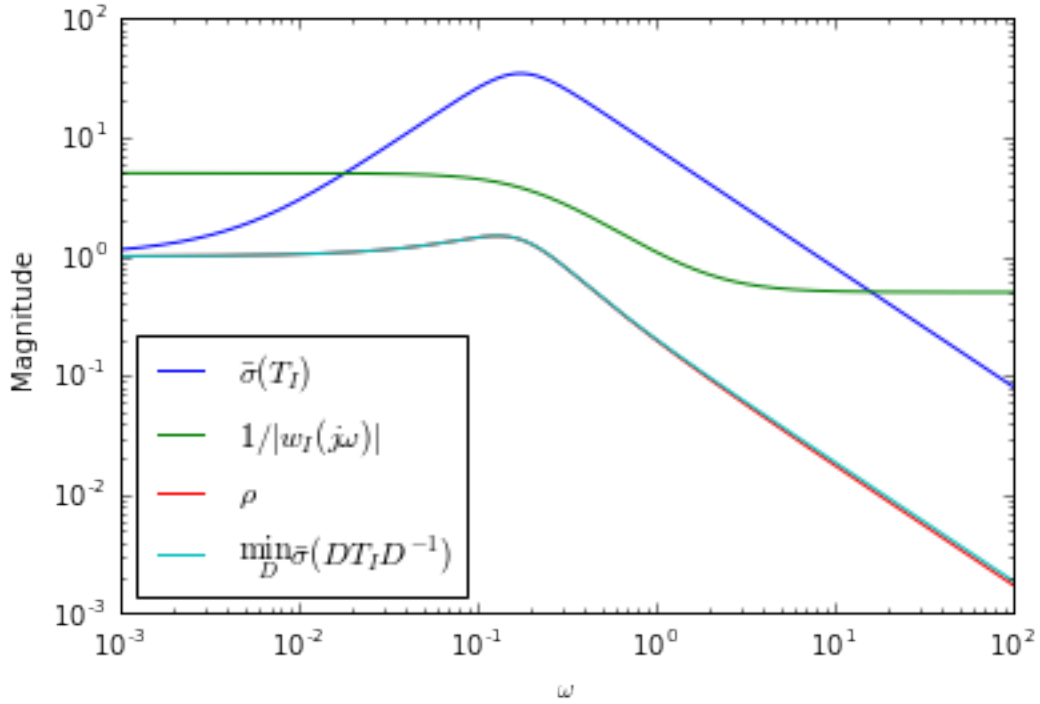
In [51]:
```
I = np.identity(2)
```

In [52]:
```
def KG(s):
    Ggain = np.matrix([[-87.8, 1.4],
                       [-108.2, -1.4]])
    Kgain = np.matrix([[-0.0015, 0],
                       [0, -0.075]])
    return 1/s*Kgain*Ggain

def w_I(s):
    return (s + 0.2) / (0.5 * s + 1)
```

In [53]:
```
def T_I(s):
    return KG(s) * (I + KG(s)).I

def M(s):
    return w_I(s) * T_I(s)
```

In [54]:
```
def maxsigma(G):
    return np.linalg.svd(G)[1].max()

def specrad(G):
    return np.abs(np.linalg.eigvals(G)).max()
```

```
In [58]: def mu_ubound(G):
             def scaled_system(d0):
                 dn = 1  # we set dn = 1 as in note 10 of 8.8.3
                 D = np.asmatrix(np.diag([d0[0], dn]))
                 return maxsigma(D*G*D.I)
             r = scipy.optimize.minimize(scaled_system, 1)
             return r['fun']

In [59]: omega = np.logspace(-3, 2, 1000)
         s = 1j * omega

         T_Is = [T_I(si) for si in s]

In [60]: mu_ubound(T_Is[3])

Out[60]: 1.0043942556993373

In [61]: def F_of_T_I(func):
             return [func(T) for T in T_Is]

In [62]: plt.loglog(omega, F_of_T_I(maxsigma),
                    label=r'$\bar\sigma(T_I)$')
         plt.loglog(omega, 1 / np.abs(w_I(s)),
                    label=r'$1/|w_I(j\omega)|$')
         plt.loglog(omega, F_of_T_I(specrad),
                    label=r'$\rho$')
         plt.loglog(omega, F_of_T_I(mu_ubound),
                    label=r'$\min_{D}\bar\sigma(DT_ID^{-1})$')
         plt.legend(loc='best')
         plt.xlabel(r'$\omega$')
         plt.ylabel('Magnitude')

         plt.show()
```




```
In [10]: import numpy
```

```
        import matplotlib.pyplot as plt
        %matplotlib inline
```

In [11]:
```
        I = numpy.eye(2)
        Kc = 1

        def G(s):
            return numpy.matrix([[3/(3*s + 1), 1/(s + 1)],
                                 [1/(s+1), 3/(4*s + 1)]])

        def K(s):
            return Kc*I

        def L(s):
            return G(s)*K(s)

        def sigmas(G):
            u, s, v = numpy.linalg.svd(G)
            return s

        def RGA(G):
            return numpy.asarray(G)*numpy.asarray(G.I).T

        def RGAnum(G):
            return numpy.abs(RGA(G) - I).sum()
```

In [23]:
```
        omegas = numpy.logspace(-2, 2, 1000)
```

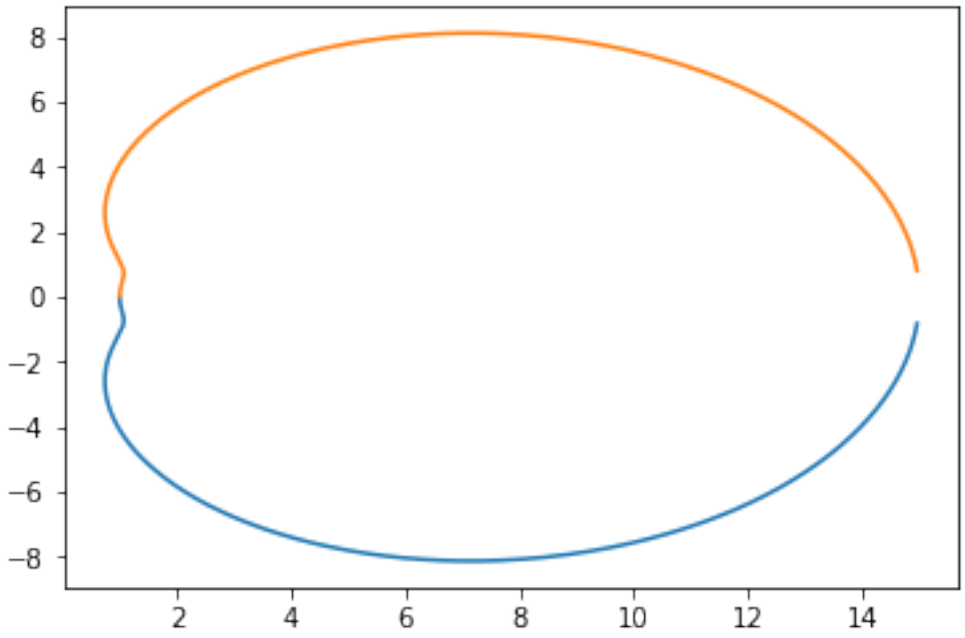
In [17]:
```
        dets = []
        OLsigmas = []
        Tsigmas = []
        RGAnums = []

        for omega in omegass:
            s = 1j*w
            T = L(s)*(I + L(s)).I # Closed loop TF

            OLsigmas.append(sigmas(G(s)))
            Tsigmas.append(sigmas(T))
            RGAnums.append(RGAnum(G(s)))
            dets.append(numpy.linalg.det(L(s) + I))
        dets = numpy.array(dets)
```

CHAPTER 7

Multivariable Nyquist plot

```
In [5]: plt.plot(dets.real, dets.imag, dets.real, -dets.imag)
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x11a44c898>,
         <matplotlib.lines.Line2D at 0x11a44ca20>]
```

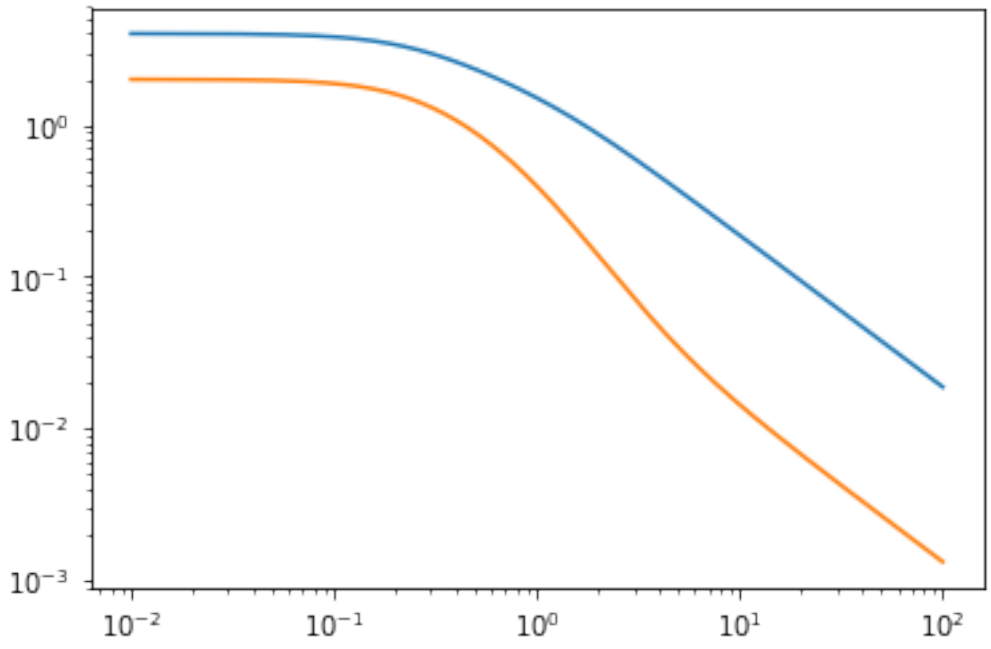


CHAPTER 8

Singular values over frequency

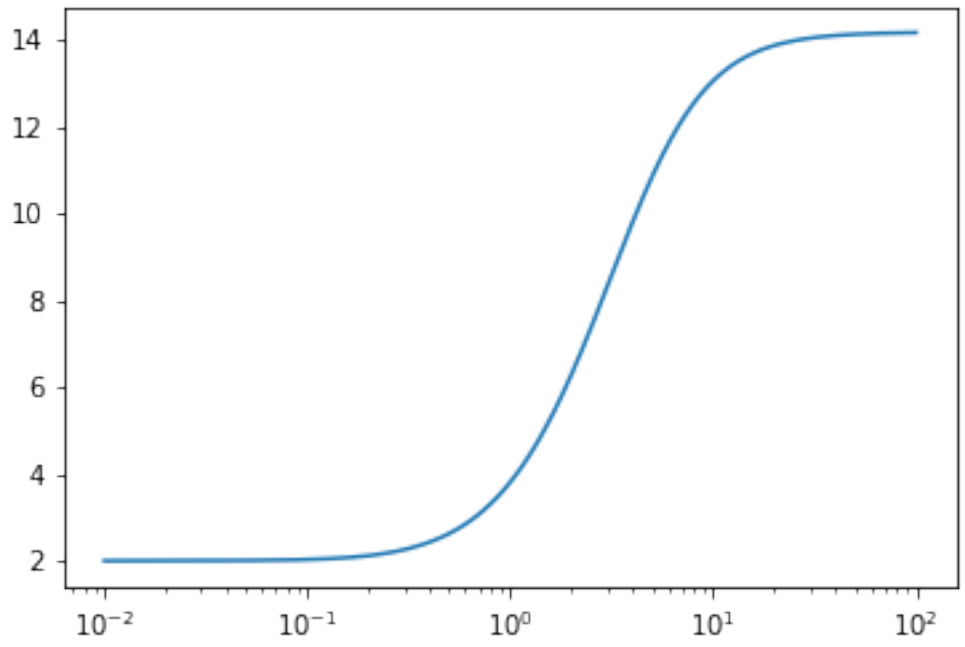
```
In [6]: plt.loglog(ws, numpy.array(OLsigmas))
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x11a53e470>,
         <matplotlib.lines.Line2D at 0x11a552550>]
```



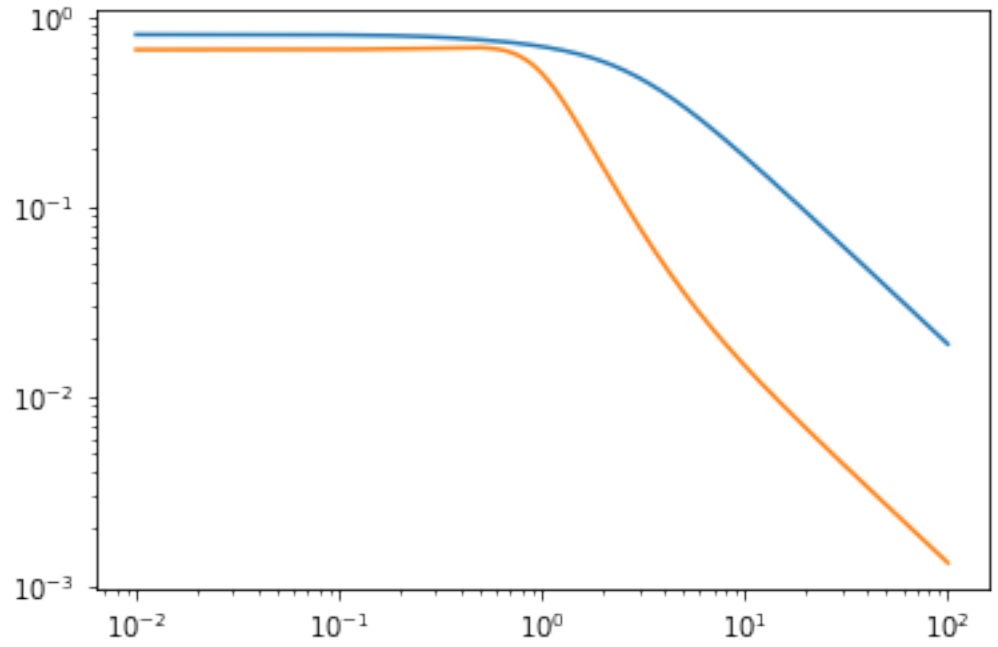

```
In [7]: allsigmas = numpy.array(OLsigmas)
        conditionnumber = allsigmas[:, 0]/allsigmas[:, 1]
        plt.semilogx(ws, conditionnumber)
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x11af367f0>]
```



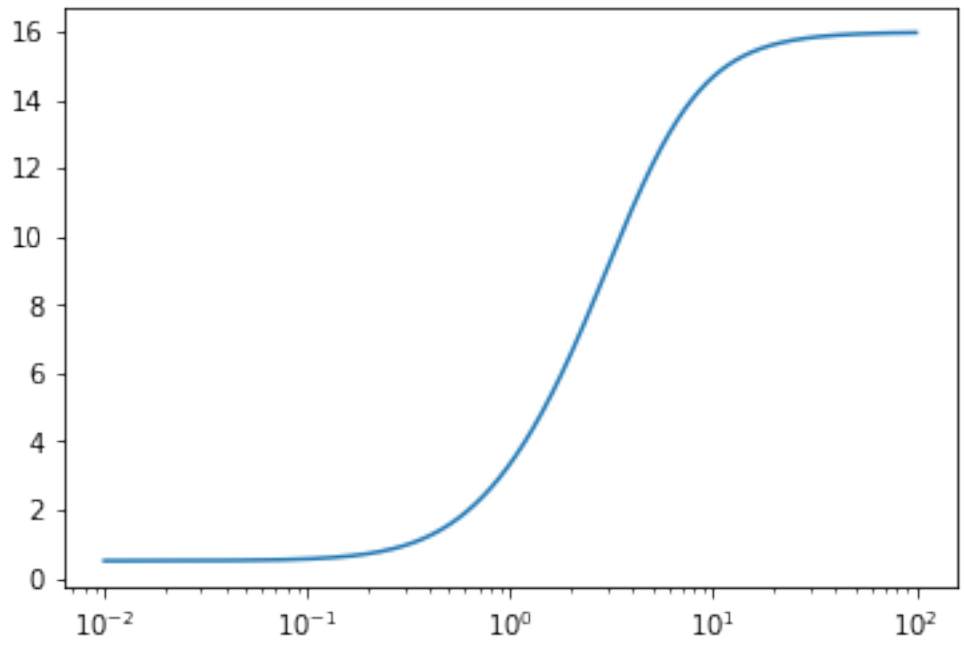

```
In [8]: plt.loglog(ws, numpy.array(Tsigmas))
```

```
Out[8]: [<matplotlib.lines.Line2D at 0x11b1539b0>,
         <matplotlib.lines.Line2D at 0x11b0e4f60>]
```




```
In [9]: plt.semilogx(ws, RGAnums)
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x11b1ca588>]
```


16
14
12
10
8
6
4
2
0
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}

CHAPTER 9

Optimisation

The textbook often contains notation like

$$\min_x f(x)$$

How do we solve these problems in Python? We use `scipy.optimize.minimize`.

```
In [1]: from scipy.optimize import minimize
        import numpy
```

Let's try a one-dimensional example first: $f(x) = 2x^2 + 3x + 2$

```
In [2]: def f(x):
            return 2*x**2 + 3*x + 2
```

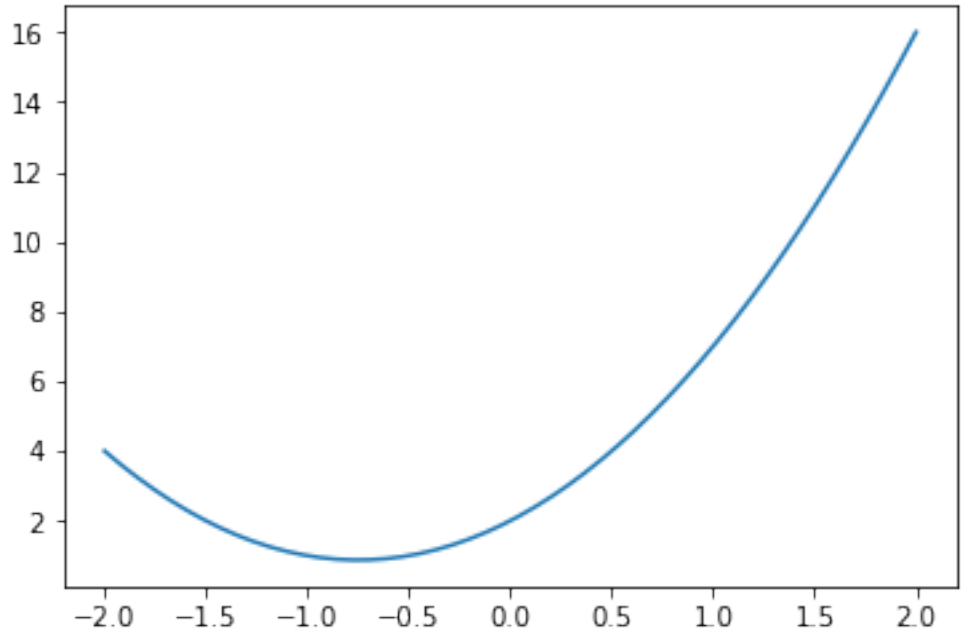
```
In [3]: xx = numpy.linspace(-2, 2)
```

```
In [4]: yy = f(xx)
```

```
In [5]: import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [6]: plt.plot(xx, yy)
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x115491748>]
```



We see the minimum lies between -1 and -0.5, so let's guess -0.5

```
In [7]: result = minimize(f, -0.5)
        result
```

```
Out[7]: fun: 0.8749999999999998
         hess_inv: array([[1]])
              jac: array([  2.98023224e-08])
          message: 'Optimization terminated successfully.'
             nfev: 9
              nit: 1
             njev: 3
           status: 0
          success: True
                x: array([-0.75])
```

The return value is an object. The value of x at which the minimum was found is in the `x` property. The function value is in the `fun` property.

```
In [8]: print("Minimum lies at x =", result.x)
        print("minimum function value: f(x) = ", result.fun)
```

```
Minimum lies at x = [-0.75]
minimum function value: f(x) =  0.8749999999999998
```

```
In [9]: minimize?
```

Let's say we have constraints on x:

$$\min_x f(x), -0.5 \leq x \leq 1$$

```
In [10]: minimize(f, 0.5, bounds=[[-0.5, 1]])
```

```
Out[10]: fun: array([ 1.])
          hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
               jac: array([ 1.00000004])
           message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
              nfev: 4
               nit: 1
```

```
     status: 0
    success: True
          x: array([-0.5])
```

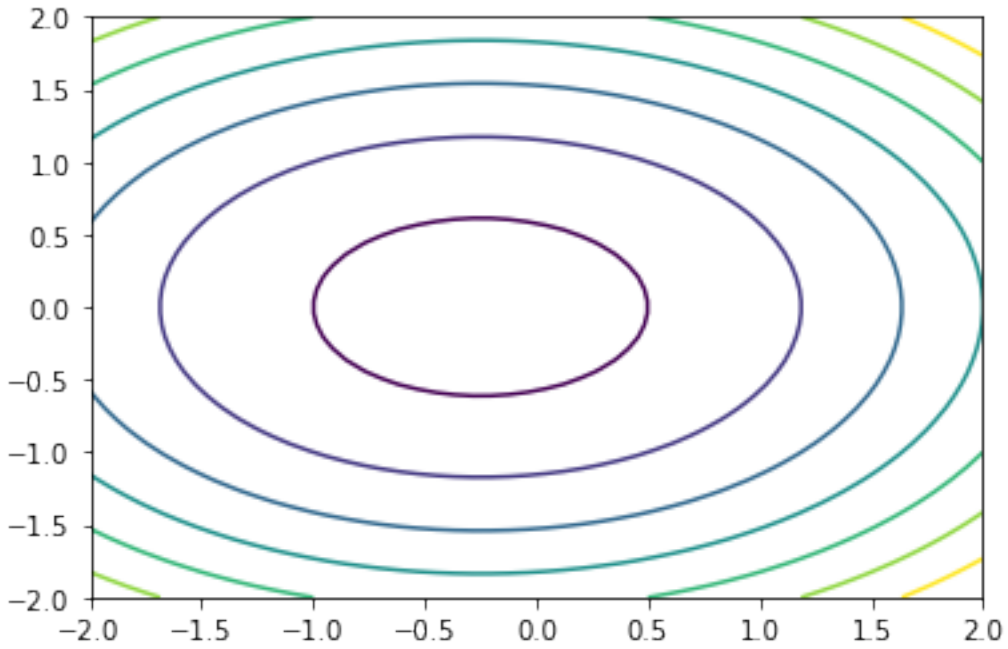
Now let's try a two-dimensional example:

$$\min_{x,y} 2x^2 + 3y^2 + x + 2$$

```
In [11]: def f2(x):
             return 2*x[0]**2 + 3*x[1]**2 + x[0] + 2
```

What does this look like? We can quickly generate values for the two dimensions using `meshgrid`

```
In [12]: x, y = numpy.meshgrid(numpy.arange(3), numpy.arange(0, 30, 10))

In [13]: x + y

Out[13]: array([[ 0,  1,  2],
                [10, 11, 12],
                [20, 21, 22]])

In [14]: xx2d, yy2d = numpy.meshgrid(xx, xx)

In [15]: plt.contour(xx2d, yy2d, f2([xx2d, yy2d]))

Out[15]: <matplotlib.contour.QuadContourSet at 0x1155fd240>
```




```
In [16]: minimize(f2, [1, 1])

Out[16]: fun: 1.8750000000000004
          hess_inv: array([[  2.50000002e-01,  -1.79099489e-09],
                [ -1.79099489e-09,   1.66666665e-01]])
               jac: array([ -1.49011612e-08,   0.00000000e+00])
           message: 'Optimization terminated successfully.'
              nfev: 24
               nit: 4
              njev: 6
            status: 0
           success: True
                 x: array([ -2.50000009e-01,  -7.14132758e-09])
```

As expected, we find x=-0.25 and y=0

Now let's constrain the problem:

```
In [17]: minimize(f2, [1, 1], bounds=[[0.5, 1.5], [0.5, 1.5]])
```

```
Out[17]: fun: 3.75
          hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
               jac: array([ 3.00000003,  3.00000007])
           message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
              nfev: 6
               nit: 1
            status: 0
           success: True
                 x: array([ 0.5,  0.5])
```

CHAPTER 10

Robust stability

```
In [1]: import numpy
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We will build a simple non-diagonal system with a diagonal proportional controller

```
In [2]: W_I = 0.1
        K_c = 1

        def G(s):
            return numpy.matrix([[1/(s + 1), 1/(s+2)],
                                 [0, 2/(2*s + 1)]])
        def K(s):
            return K_c*numpy.asmatrix(numpy.eye(2))

        def M(s):
            """ N_11 from eq 8.32 """
            return -W_I*K(s)*G(s)*(numpy.eye(2) + K(s)*G(s)).I
```

```
In [3]: M(2.0)
```

```
Out[3]: matrix([[-0.025     , -0.01339286],
                [ 0.        , -0.02857143]])
```

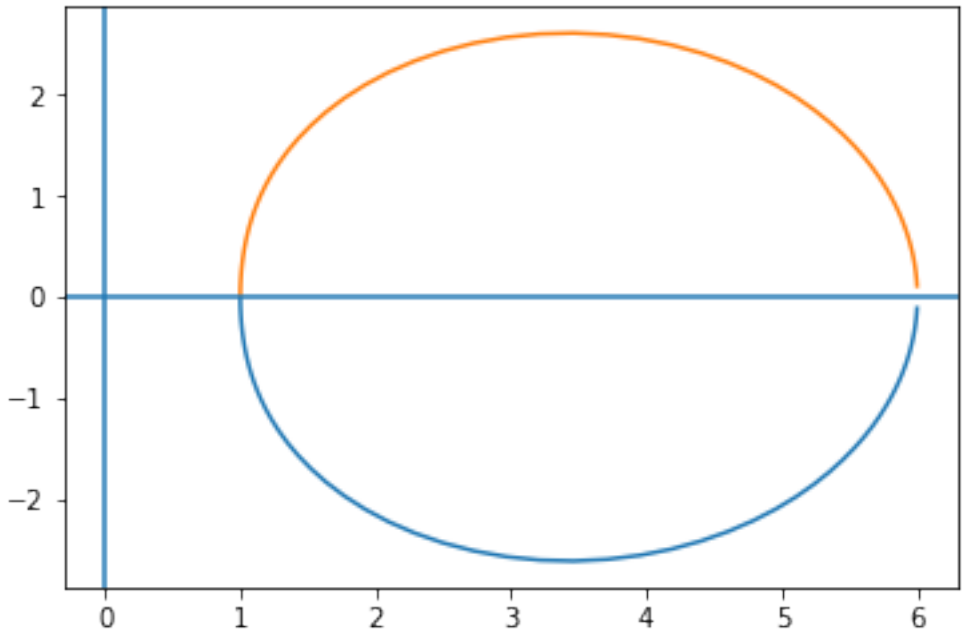
Let's observe the MV Nyquist diagram of this process

```
In [4]: omega = numpy.logspace(-2, 2, 100)
        s = 1j*omega
```

```
In [5]: def mvdet(s):
            return numpy.linalg.det(numpy.eye(2) + G(s)*K(s))

        fr = numpy.array([mvdet(si) for si in s])
```

```
In [6]: plt.plot(fr.real, fr.imag)
        plt.plot(fr.real, -fr.imag)
        plt.axhline(0)
        plt.axvline(0)
```

```
Out[6]: <matplotlib.lines.Line2D at 0x11a960208>
```



This system looks stable since there are no encirclements of 0.

Now, let's add some uncertainty. We will be building an unstructured Δ as well as a diagonal Δ.

```
In [7]: def maxsigma(m):
            _, S, _ = numpy.linalg.svd(m)
            return S.max()

In [8]: def specradius(m):
            return numpy.abs(numpy.linalg.eigvals(m)).max()

In [9]: def unstructuredDelta():
            numbers = numpy.random.rand(4)*2 - 1 + 1j*(numpy.random.rand(4)*2 -1)
            Delta = numpy.asmatrix(numpy.reshape(numbers, (2, 2)))
            return Delta

In [10]: def diagonalDelta():
             numbers = numpy.random.rand(2)*2 - 1 + 1j*(numpy.random.rand(2)*2 -1)
             Delta = numpy.asmatrix(numpy.diag(numbers))
             return Delta

In [11]: def allowableDelta(kind):
             while True:
                 Delta = kind()
                 if maxsigma(Delta) < 1:
                     return Delta
```

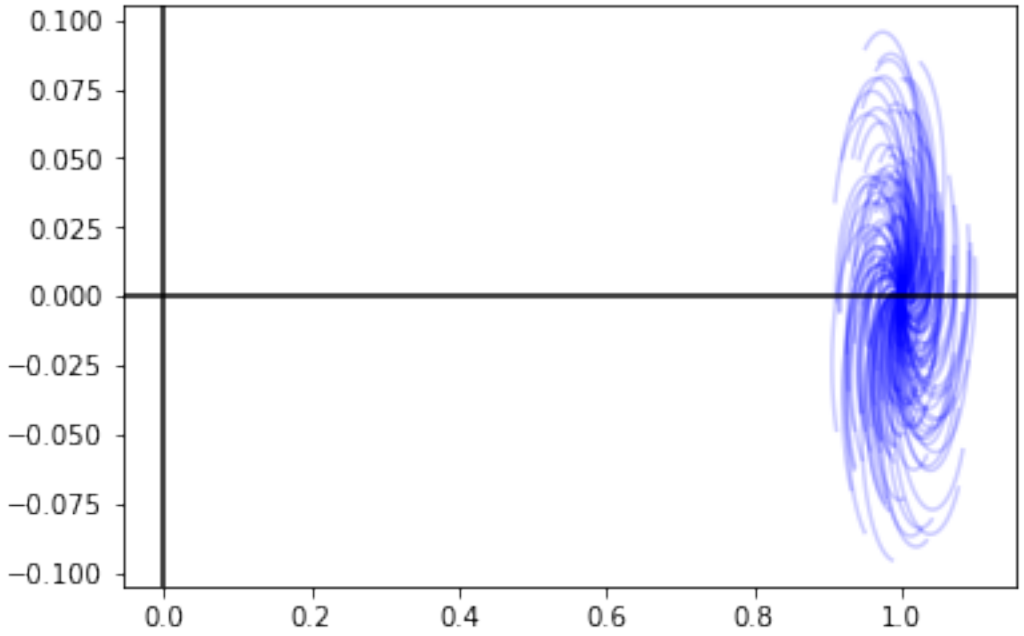
So now we can generate an acceptable Delta.

```
In [12]: allowableDelta(unstructuredDelta)

Out[12]: matrix([[ 0.07539216+0.09878427j, -0.73336265+0.2846334j ],
                 [-0.51959904+0.5188734j , -0.25490849+0.05011251j]])

In [13]: I = numpy.eye(2)

In [14]: def Mdelta(Delta, s):
             return M(s)*Delta
```

```
In [15]: kind = diagonalDelta
```

Let's see what the Nyquist diagrams look like for lots of different allowable Deltas.

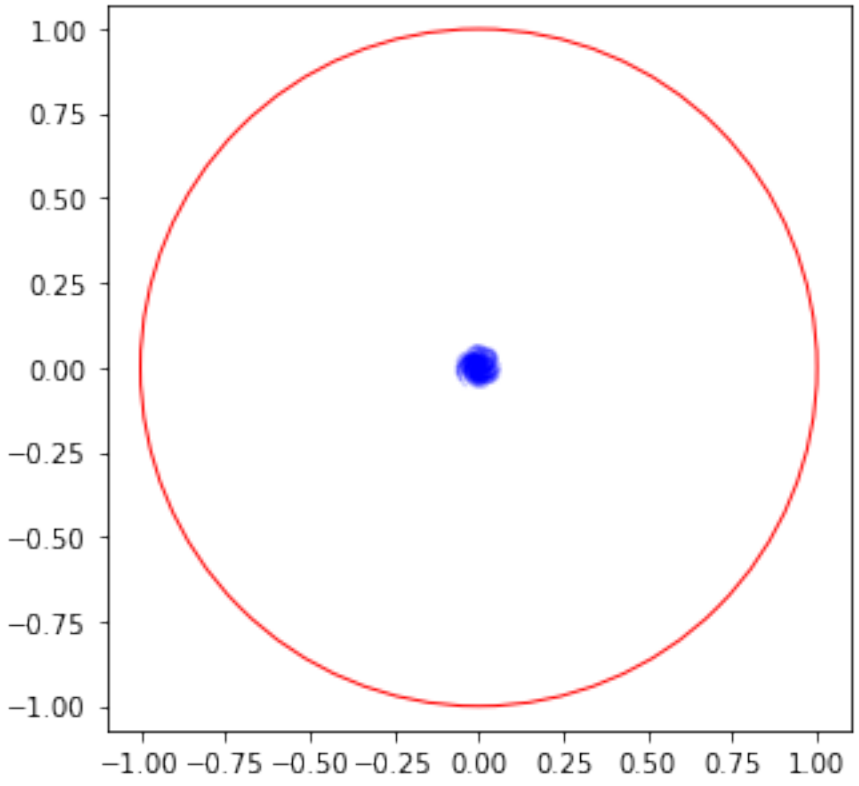
```
In [16]: Ndelta = 200

         for i in range(Ndelta):
             Delta = allowableDelta(kind)
             fr = numpy.array([numpy.linalg.det(I - Mdelta(Delta, si)) for si in s])
             plt.plot(fr.real, fr.imag, color='blue', alpha=0.2)
         plt.axhline(0, color='black')
         plt.axvline(0, color='black')
         plt.show()
```



That looks like no Δ moved us near the zero point, which is where we will become unstable. So this system looks it's robustly stable.

To apply the small gain theorem, we need to check the eigenvalues of $M\Delta$

```
In [17]: Ndelta = 100
         fig = plt.figure(figsize=(5, 5))
         for i in range(Ndelta):
             Delta = allowableDelta(kind)
             fr = numpy.array([numpy.linalg.eigvals(Mdelta(Delta, si)) for si in s])
             plt.plot(fr.real, fr.imag, color='blue', alpha=0.2)
         plt.gca().add_artist(plt.Circle((0, 0), radius=1, fill=False, color='red'))
         plt.axis('equal')
         plt.xlim(-1.1, 1.1)
         plt.ylim(-1.1, 1.1)
         plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00

CHAPTER 11

Minimised singular value

How would be find the minimized singular value? One way is via direct optimization.

```
In [18]: import scipy.optimize
```

$$\bar{\sigma}(DM(s)D^{-1})$$

```
In [19]: def obj(d):
             d1, d2 = d
             D = numpy.asmatrix([[d1, 0],
                                 [0, d2]])
             return maxsigma(D*M(si)*D.I)
```

$$\min_D \bar{\sigma}(DM(s)D^{-1})$$

```
In [20]: r = scipy.optimize.minimize(obj, [1, 1])

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-77439d733717> in <module>()
----> 1 r = scipy.optimize.minimize(obj, [1, 1])

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/_minimize.py in minimize(fun, x0, args, metho
    479         return _minimize_cg(fun, x0, args, jac, callback, **options)
    480     elif meth == 'bfgs':
--> 481         return _minimize_bfgs(fun, x0, args, jac, callback, **options)
    482     elif meth == 'newton-cg':
    483         return _minimize_newtoncg(fun, x0, args, jac, hess, hessp, callback,

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in _minimize_bfgs(fun, x0, args,
    941     else:
    942         grad_calls, myfprime = wrap_function(fprime, args)
--> 943     gfk = myfprime(x0)
    944     k = 0
    945     N = len(x0)

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in function_wrapper(*wrapper_args)
    290     def function_wrapper(*wrapper_args):
```

```
    291         ncalls[0] += 1
--> 292         return function(*(wrapper_args + args))
    293
    294     return ncalls, function_wrapper

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in approx_fprime(xk, f, epsilon,
    701
    702     """
--> 703     return _approx_fprime_helper(xk, f, epsilon, args=args)
    704
    705


~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in _approx_fprime_helper(xk, f, ep
    635     """
    636     if f0 is None:
--> 637         f0 = f(*((xk,) + args))
    638     grad = numpy.zeros((len(xk),), float)
    639     ei = numpy.zeros((len(xk),), float)

~/anaconda3/lib/python3.6/site-packages/scipy/optimize/optimize.py in function_wrapper(*wrapper_args)
    290     def function_wrapper(*wrapper_args):
    291         ncalls[0] += 1
--> 292         return function(*(wrapper_args + args))
    293
    294     return ncalls, function_wrapper

<ipython-input-19-97cd97df0c1e> in obj(d)
      3     D = numpy.asmatrix([[d1, 0],
      4                         [0, d2]])
----> 5     return maxsigma(D*M(si)*D.I)

NameError: name 'si' is not defined
```

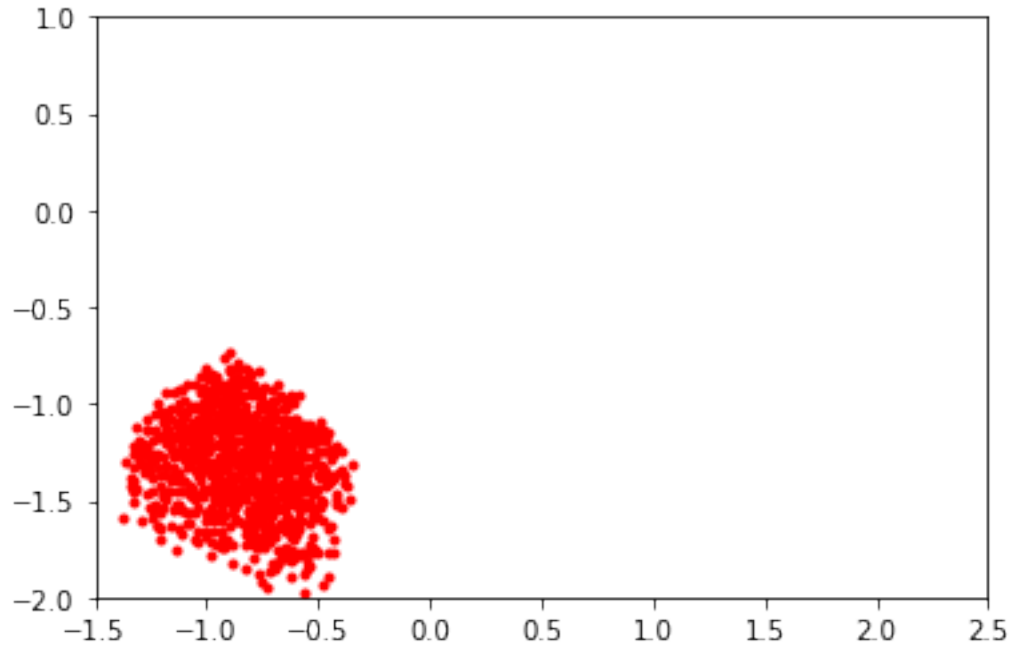
```
In [ ]: b = []
        for si in s:
            r = scipy.optimize.minimize(obj, [4, 4])
            f = r.fun
            b.append(f)
```

Let's plot all the spectral radii along with some bounds. Remember you can go back and change `kind` to do this for the other kind of .

```
In [ ]: Ndelta = 1000

        for i in range(Ndelta):
            Delta = allowableDelta(kind)
            specradiuss = numpy.array([specradius(Mdelta(Delta, si)) for si in s])
            plt.loglog(omega, specradiuss, color='blue', alpha=0.2)

        maxsigmas = numpy.array([maxsigma(M(si)) for si in s])
        specradius_of_m = [specradius(M(si)) for si in s]
        plt.loglog(omega, maxsigmas, color='red', label=r'$\bar{\sigma}(M)$')
        plt.loglog(omega, specradius_of_m, linewidth=6, color='green', label=r'$\rho{M}$')
        plt.loglog(omega, b, linewidth=2, color='yellow', label=r'$\min_D \bar{\sigma}(DMD^{-1})$')
        plt.axhline(1, color='black')
        plt.ylim([1e-2, 1e-1])
        plt.legend()
        plt.show()
```

```
In [1]: import numpy
```

```
In [56]: import matplotlib.pyplot as plt
         %matplotlib inline

In [61]: x = numpy.random.rand(100000)

In [79]: def random_parameter():
             delta = (numpy.random.rand() - 0.5)*2
             alphabar = 2.5
             rp = 0.5/alphabar
             return alphabar*(1 + rp*delta)

In [80]: omega = 0.5

In [82]: for i in range(1000):
             k = random_parameter()
             tau = random_parameter()
             theta = random_parameter()

             s = 1j*omega

             Gp = k/(tau*s + 1)*numpy.exp(-theta*s)
             plt.plot(Gp.real, Gp.imag, 'r.')
         plt.xlim(-1.5, 2.5)
         plt.ylim(-2, 1)

Out[82]: (-2, 1)
```




```
In [4]: import matplotlib.pyplot as plt
        %matplotlib inline

In [5]: import numpy

In [6]: from functools import partial
```

CHAPTER 12

Uncertainty

Let's start by defining a FOPDT function as in the textbook

```
In [7]: def G_P(k, theta, tau, s):
            """ Equation 7.19 """
            return k/(tau*s + 1)*numpy.exp(-theta*s)
```

Let's see what this looks like for particular values of k, τ and θ

```
In [8]: omega = numpy.logspace(-2, 2, 1000)
        s = 1j*omega
```

```
In [9]: Gnom = partial(G_P, 2.5, 2.5, 2.5)
```

```
In [10]: def Gnom(s):
             return G_P(2.5, 2.5, 2.5, s)
```

```
In [11]: Gfr = Gnom(s)
```

```
In [12]: pomega = 0.5
```

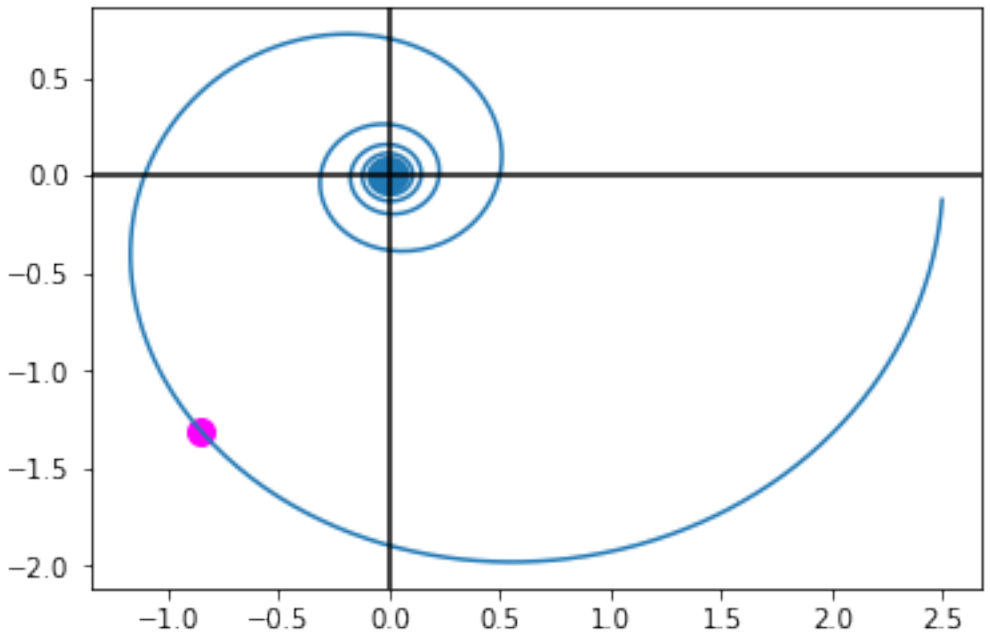
```
In [13]: def nominal_curve():
             plt.plot(Gfr.real, Gfr.imag)
             plt.axhline(0, color='black')
             plt.axvline(0, color='black')
```

```
In [14]: def nominal_point(pomega):
             Gnomp = Gnom(1j*pomega)
             plt.scatter(Gnomp.real, Gnomp.imag, s=100, color='magenta')
```

```
In [15]: Gnom(1j*pomega)
```

```
Out[15]: (-0.8496667432187457-1.3103781193655 34j)
```

```
In [16]: nominal_curve()
         nominal_point(pomega)
```



At a particular frequency, let's look at a couple of possible plants

```
In [17]: varrange = numpy.arange(2, 3, 0.1)
         def cloudpoints(pomega):
             points = numpy.array([G_P(k, theta, tau, pomega*1j)
                                   for k in varrange
                                   for tau in varrange
                                   for theta in varrange])
             nominal_curve()
             nominal_point(pomega)
             plt.scatter(points.real, points.imag, color='red', alpha=0.1)
```

```
In [18]: from ipywidgets import interact
         import ipywidgets as widgets
```

```
In [19]: interact(cloudpoints, pomega=(0.1, 10))
```

```
interactive(children=(FloatSlider(value=5.05, description='pomega', max=10.0, min=0.1), Output()), _
```

```
Out[19]: <function __main__.cloudpoints>
```

Let's try to approximate this region by a disc

```
In [20]: Gnomp = Gnom(1j*pomega)
         points = numpy.array([G_P(k, theta, tau, pomega*1j)
                               for k in varrange
                               for tau in varrange
                               for theta in varrange])

         radius = max(abs(P - Gnomp) for P in points)
         radius
```

```
Out[20]: 0.8068845952466887
```

```
In [21]: def discapprox(pomega, radius):
             Gnomp = Gnom(1j*pomega)
             c = plt.Circle((Gnomp.real, Gnomp.imag), radius, alpha=0.2)
             cloudpoints(pomega)
             plt.gca().add_artist(c)
```

```
            plt.axis('equal')
        interact(discapprox, pomega=(0.1, 5), radius=radius)
```

```
interactive(children=(FloatSlider(value=2.5500000000000003, description='pomega', max=5.0, min=0.1),
```

```
Out[21]: <function __main__.discapprox>
```

The above represents an *additive* uncertainty description,

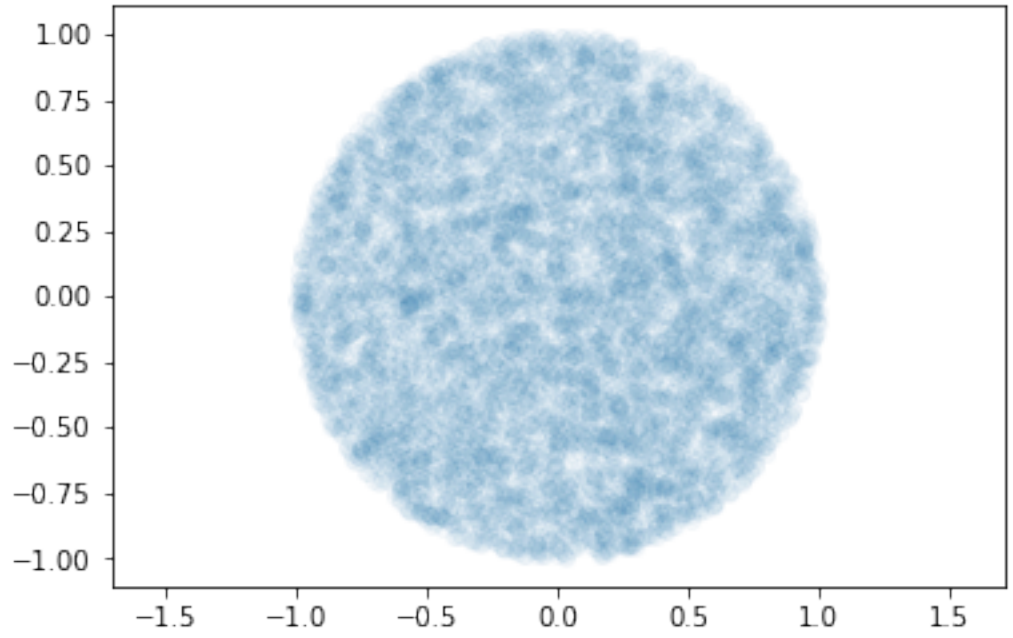
$$|\Delta_A| < 1$$

$$G_p(s) = G(s) + w_A(s)\Delta_A(s); \quad |\Delta_A(j\omega)| \leq 1 \forall \omega | (7.20)$$

```
In [22]: Npoints = 10000
         Delta_As = (numpy.random.rand(Npoints)*2-1 +
                     (numpy.random.rand(Npoints)*2 - 1)*1j)
         valid_values = numpy.abs(Delta_As) < 1
         Delta_As = Delta_As[valid_values]
```

```
In [23]: plt.scatter(Delta_As.real, Delta_As.imag, alpha=0.03)
         plt.axis('equal')
```

```
Out[23]: (-1.1075495875096064,
          1.1094475944764177,
          -1.108437989610789,
          1.1089294077137237)
```



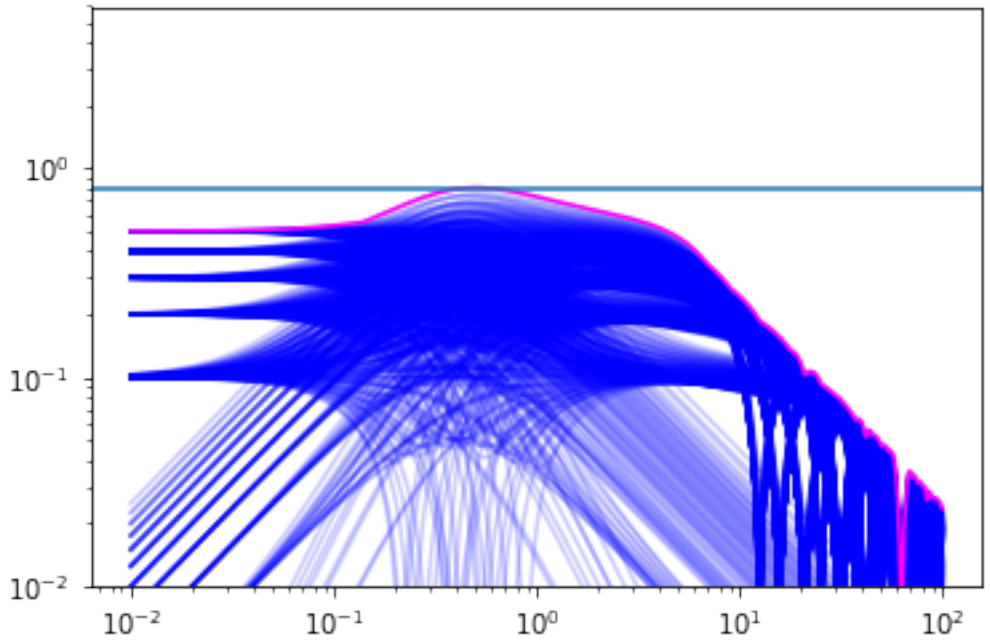

```
In [24]: def discapprox(pomega, radius):
             Gnomp = Gnom(1j*pomega)
             #c = plt.Circle((Gnomp.real, Gnomp.imag), radius, alpha=0.2)
             Gp_A = Gnomp + radius*Delta_As
             plt.scatter(Gp_A.real, Gp_A.imag, alpha=0.03)
             cloudpoints(pomega)
             #plt.gca().add_artist(c)
             plt.axis('equal')
         interact(discapprox, pomega=(0.1, 5), radius=radius)
```

```
interactive(children=(FloatSlider(value=2.5500000000000003, description='pomega', max=5.0, min=0.1),
```

```
Out[24]: <function __main__.discapprox>
```

We now build frequency response for Π (all possible plants).

```
In [25]: Pi = numpy.array([G_P(k, theta, tau, s)
                           for k in varrange
                           for tau in varrange
                           for theta in varrange])

In [26]: deviations = numpy.abs(Pi - Gfr)

In [27]: maxima = numpy.max(deviations, axis=0)

In [28]: maxima.max()

Out[28]: 0.806912987162208

In [29]: plt.loglog(omega, numpy.abs(deviations.T), color='blue', alpha=0.2);
         plt.loglog(omega, maxima, color='magenta')
         plt.axhline(radius)
         #plt.loglog(pomega, radius, 'r.')
         plt.ylim(ymin=1e-2)

Out[29]: (0.01, 5.958602718540838)
```



12.1 Optimisation of the maximum

```
In [30]: from scipy.optimize import minimize

In [31]: def objective(x, omega):
             k, tau, theta = x
             s = 1j*omega
             return -numpy.abs(Gnom(s) - G_P(k, tau, theta, s))

In [32]: objective([2.5, 2.5, 2.5], 0)

Out[32]: -0.0
```

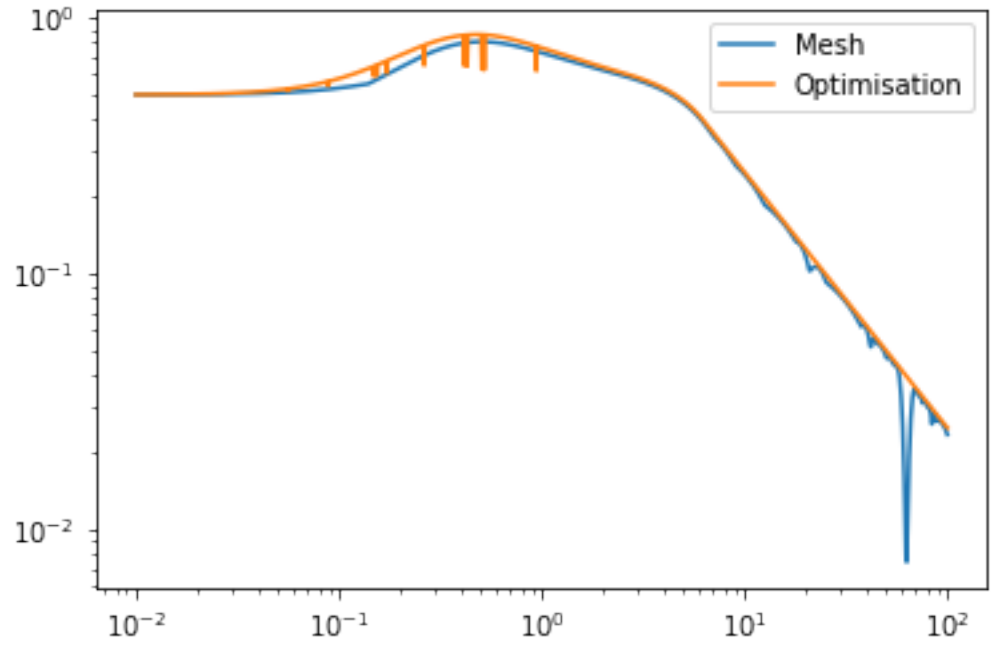
```
In [42]: x0 = numpy.array([2.5, 2.5, 2.5])
         bounds = [[2, 3]]*3
         minimize(objective, x0, args=0, bounds=bounds)
```

```
Out[42]: fun: -0.5
          hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
               jac: array([-0.99999999,  0.        ,  0.        ])
           message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
              nfev: 8
               nit: 1
            status: 0
           success: True
                 x: array([3. , 2.5, 2.5])
```

```
In [59]: vals = []
         starts = 10
         for omegai in omega:
             best = float('-inf')
             # We will use "multi-start" strategy
             for start in range(starts):
                 x0 = numpy.random.uniform(2, 3, size=3)
                 r = minimize(objective,
                              x0,
                              args=omegai,
                              bounds=bounds,
                              method='TNC')  # TNC and L-BFGS-B can handle bounds
                 if -r['fun'] > best:
                     best = -r['fun']
             vals.append(best)
```

```
In [60]: plt.loglog(omega, maxima, label='Mesh')
         plt.loglog(omega, vals, label='Optimisation')
         plt.legend()
```

```
Out[60]: <matplotlib.legend.Legend at 0x11a1b05c0>
```




```
In [61]: def plot_upper(K, tau, plotmax):
```

```
            w_A = K/(tau*s + 1)
            if plotmax:
                plt.loglog(omega, maxima, color='blue')
            else:
                plt.loglog(omega, numpy.abs(deviations.T), color='blue', alpha=0.1);
            plt.loglog(omega, numpy.abs(w_A), color='red')
            plt.ylim(ymin=1e-2)
```

```
In [62]: i = interact(plot_upper, K=(0.1, 2, 0.01), tau=(0.01, 1, 0.01),
                    plotmax=widgets.Checkbox())
```

```
interactive(children=(FloatSlider(value=1.05, description='K', max=2.0, min=0.1, step=0.01), FloatSli
```

```
In [63]: def combined(pomega, K, tau, plotmax):
            f = plt.figure(figsize=(12, 6))
            plt.subplot(1, 2, 1)
            plot_upper(K, tau, plotmax)
            plt.axvline(pomega)
            plt.subplot(1, 2, 2)
            s = 1j*pomega
            radius = numpy.abs(K/(tau*s + 1))
            discapprox(pomega, radius)
```

```
In [64]: interact(combined, pomega=(0.1, 10), K=(0.1, 2, 0.01), tau=(0.01, 1, 0.01),
                 plotmax=widgets.Checkbox())
```

```
interactive(children=(FloatSlider(value=5.05, description='pomega', max=10.0, min=0.1), FloatSlider(v
```

```
Out[64]: <function __main__.combined>
```

12.2 Addednum: understanding the use of partial

In the code above we used the function `functools.partial`. What's going on there?

Let's say we have a function with many arguments, but we want to optimise only one:

```
In [65]: def function_with_many_arguments(a, b, c, d):
            return 1*a + 2*b + 3*c + 4*d
```

We could define a "wrapper" which just calls that function with the single argument we need.

```
In [66]: def wrapper_function(d):
            return function_with_many_arguments(1, 2, 3, d)
```

```
In [67]: wrapper_function(2)
```

```
Out[67]: 22
```

```
In [68]: import scipy.optimize
```

```
In [69]: scipy.optimize.fsolve(wrapper_function, 3)
```

```
Out[69]: array([-3.5])
```

`functools.partial` automates the creation of this wrapper function.

```
In [70]: from functools import partial
```

```
In [71]: wrapper_function = partial(function_with_many_arguments, 1, 2, 3)
```

```
In [72]: wrapper_function(2)
```

```
Out[72]: 22
```

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Created on Jan 27, 2012

@author: Carl Sandrock

utils.**BoundKS**(*G*, *poles*, *up*, *e=1e-05*)

The functions uses equaption 6.24 (p229) to calculate the peak value for KS transfer function using the stable version of the plant.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

poles [numpy array (number of poles)] List of right-half plane poles.

up [numpy array (number of poles)] List of input pole directions.

e [float] Avoid division by zero. Let epsilon be very small (optional).

Returns

KS_max [float] Minimum peak value.

utils.**BoundST**(*G*, *poles*, *zeros*, *deadtime=None*)

This function will calculate the minimum peak values of S and T if the system has zeros and poles in the input or output. For standard conditions Equation 6.8 (p224) is applied. Equation 6.16 (p226) is used when deadtime is included.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

poles [numpy array (number of zeros)] List of poles.

zeros [numpy array (number of zeros)] List of zeros.

deadtime [numpy matrix (n, n)] Deadtime or time delay for G.

Returns

Ms_min [real] Minimum peak value.

utils.**Closed_loop**(*Kz*, *Kp*, *Gz*, *Gp*)
Return zero and pole polynomial for a closed loop function.

Parameters

Kz & Gz [list] Polynomial constants in the numerator.

Kz & Gz [list] Polynomial constants in the denominator.

Returns

Zeros_poly [list] List of zero polynomial for closed loop function.

Poles_poly [list] List of pole polynomial for closed loop function.

utils.**ControllerTuning**(*G*, *method='ZN'*)
Calculates either the Ziegler-Nichols or Tyreus-Luyben tuning parameters for a PI controller based on the continuous cycling method.

Parameters

G [tf] plant model

method [Use 'ZN' for Ziegler-Nichols tuning parameters and] 'TT' for Tyreus-Luyben parameters. The default is to return Ziegler-Nichols tuning parameters.

Returns

Kc [array containing a real number] proportional gain

Taui [array containing a real number] integral gain

Ku [array containing a real number] ultimate P controller gain

Pu [array containing a real number] corresponding period of oscillations

utils.**IterRGA**(*A*, *n*)
Computes the n'th iteration of the RGA.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

Returns

n'th iteration of RGA matrix [matrix] iterated RGA matrix of complex numbers.

utils.**RGA**(*G*)
Computes the RGA (Relative Gain Array) of a matrix.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

Returns

RGA matrix [matrix] RGA matrix of complex numbers.

utils.**RGAnumber**(*G*, *I*)
Computes the RGA (Relative Gain Array) number of a matrix.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

I [numpy matrix] Pairing matrix.

Returns

RGA number [float] RGA number.

utils.**SVD**(*G*)

Returns the singular values (Sv) as well as the input and output singular vectors (V and U respectively).

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

Returns

U [matrix of complex numbers] Unitary matrix of output singular vectors.

Sv [array] Singular values of *Gin* arranged in decending order.

V [matrix of complex numbers] Unitary matrix of input singular vectors.

utils.**Wp**(*wB*, *M*, *A*, *s*)

Computes the magnitude of the performance weighting function. Based on Equation 2.105 (p62).

Parameters

wB [float] Approximate bandwidth requirement. Asymptote crosses 1 at this frequency.

M [float] Maximum frequency.

A [float] Maximum steady state tracking error. Typically 0.

s [complex] Typically *w*1j*.

Returns

'|Wp(s)|' [float] The magnitude of the performance weighting fucntion at a specific frequency (s).

utils.**arrayfun**(*f*, *A*)

Recurses down to scalar elements in A, then applies f, returning lists containing the result.

Parameters

A [array]

f [function]

Returns

arrayfun [list]

>>> def f(x):

... return 1.

>>> arrayfun(f, numpy.array([1, 2, 3]))

[1.0, 1.0, 1.0]

>>> arrayfun(f, numpy.array([[1, 2, 3], [1, 2, 3]]))

[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]

>>> arrayfun(f, 1)

1.0

utils.**astf**(*maybetf*)

Parameters **maybetf** – something which could be a tf

Returns a transfer function object

```
>>> G = tf( 1, [ 1, 1])
>>> astf(G)
tf([1.], [1. 1.])
```

```
>>> astf(1)
tf([1.], [1.])
```

```
>>> astf(numpy.matrix([[G, 1.], [0., G]]))
matrix([[tf([1.], [1. 1.]), tf([1.], [1.])],
        [tf([0.], [1]), tf([1.], [1. 1.])]], dtype=object)
```

utils.**circle**(*cx*, *cy*, *r*)

Return the coordinates of a circle

Parameters

cx [float] Center x coordinate.

cy [float] Center y coordinate.

r [float] Radius.

Returns

x, y [float] Circle coordinates.

utils.**det**(*A*)

Calculate determinant via elementary operations

Parameters **A** – Array-like object

Returns determinant

```
>>> det(2.)
2.0
```

```
>>> A = [[1., 2.],
...      [1., 2.]]
>>> det(A)
0.0
```

```
>>> B = [[1., 2.],
...      [3., 4.]]
>>> det(B)
-2.0
```

```
>>> C = [[1., 2., 3.],
...      [1., 3., 2.],
...      [3., 2., 1.]]
>>> det(C)
-12.0
```

Can handle matrices of tf objects # TODO: make this a little more natural (without the .matrix) >>> G11 = tf([1], [1, 2]) >>> G = mimotf([[G11, G11], [G11, G11]]) >>> det(G.matrix) tf([0.], [1])

```
>>> G = mimotf([[G11, 2*G11], [G11**2, 3*G11]])
>>> det(G.matrix)
tf([ 3. 16. 28. 16.], [ 1. 10. 40. 80. 80. 32.])
```

utils.**distRHPZ**(*G*, *Gd*, *RHP_Z*)

Applies equation 6.48 (p239) For performance requirements imposed by disturbances. Calculate the system's zeros alignment with the disturbacne matrix.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

gd [numpy matrix (n x 1)] The transfer function Gd(s) of the distrurbances.

RHP_Z [complex] Right-half plane zero

Returns

Dist_RHPZ [float] Minimum peak value.

utils.**distRej**(*G*, *gd*)

Convenience wrapper for calculation of ||gd||2 (equation 6.42, p238) and the disturbance condition number (equation 6.43) for each disturbance.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

gd [numpy matrix (m x n)] The transfer function Gd(s) of the distrurbances.

Returns

1/||gd|| :math:'_2' [float] The inverse of the 2-norm of a single disturbance gd.

distCondNum [float] The disturbance condition number σ (G) σ (G $^{-1}$ yd)

yd [numpy matrix] Disturbance direction.

utils.**feedback**(*forward*, *backward=None*, *positive=False*)

Defined for use in connect function Calculates a feedback loop This version is for trasnfer function objects Negative feedback is assumed, use positive=True for positive feedback Forward refers to the function that goes out of the comparator Backward refers to the function that goes into the comparator

utils.**feedback_mimo**(*forward*, *backward=None*, *positive=False*)

Calculates a feedback loop This version is for matrices Negative feedback is assumed, use positive=True for positive feedback Forward refers to the function that goes out of the comparator Backward refers to the function that goes into the comparator

utils.**findst**(*G*, *K*)

Find S and T given a value for G and K.

Parameters

G [numpy array] Matrix of transfer functions.

K [numpy array] Matrix of controller functions.

Returns

S [numpy array] Matrix of sensitivities.

T [numpy array] Matrix of complementary sensitivities.

utils.**freq**(*G*)

Calculate the frequency response for an optimisation problem

Parameters

G [tf] plant model

Returns

Gw [frequency response function]

utils.**gaintf**(*K*)
Transform a gain value into a transfer function.

Parameters

K [float] Gain.

Returns

gaintf [tf] Transfer function.

utils.**kalman_controllable**(*A*, *B*, *C*, *P=None*, *RP=None*)
Computes the Kalman Controllable Canonical Form of the inout system A, B, C, making use of QR Decomposition. Can be used in sequentially with kalman_observable to obtain a minimal realisation.

Parameters

A [numpy matrix] The system state matrix.

B [numpy matrix] The system input matrix.

C [numpy matrix] The system output matrix.

P [(optional) numpy matrix] The controllability matrix

RP [(optional int)]

Returns

Ac [numpy matrix] The state matrix of the controllable system

Bc [nump matrix] The input matrix of the controllable system

Cc [numpy matrix] The output matrix of the controllable system

utils.**kalman_observable**(*A*, *B*, *C*, *Q=None*, *RQ=None*)
Computes the Kalman Observable Canonical Form of the inout system A, B, C, making use of QR Decomposition. Can be used in sequentially with kalman_controllable to obtain a minimal realisation.

Parameters

A [numpy matrix] The system state matrix.

B [numpy matrix] The system input matrix.

C [numpy matrix] The system output matrix.

Q [(optional) numpy matrix] Observability matrix

RQ [(optional) int] Rank of observability matrxi

Returns

Ao [numpy matrix] The state matrix of the observable system

Bo [nump matrix] The input matrix of the observable system

Co [numpy matrix] The output matrix of the observable system

utils.**lcm_of_all_minors**(*G*)
Returns the lowest common multiple of all minors of G

utils.**listify**(*A*)
Transform a gain value into a transfer function.

Parameters

K [float] Gain.

Returns

gaintf [tf] Transfer function.

utils.**margins**(*G*)
Calculates the gain and phase margins, together with the gain and phase crossover frequency for a plant model

Parameters

G [tf] plant model

Returns

GM [array containing a real number] gain margin

PM [array containing a real number] phase margin

wc [array containing a real number] gain crossover frequency where **|G(jwc)|** = 1

w_180 [array containing a real number] phase crossover frequency where angle[G(jw_180] = -180 deg

utils.**marginsclosedloop**(*L*)
Calculates the gain and phase margins, together with the gain and phase crossover frequency for a control model

Parameters

L [tf] loop transfer function

Returns

GM [real] gain margin

PM [real] phase margin

wc [real] gain crossover frequency for L

wb [real] closed loop bandwidth for S

wbt [real] closed loop bandwidth for T

utils.**matrix_as_scalar**(*M*)
Return a scalar from a 1x1 matrix

Parameters **M** – matrix

Returns scalar part of matrix if it is 1x1 else just a matrix

utils.**maxpeak**(*G*, *w_start=-2*, *w_end=2*, *points=1000*)
Computes the maximum bode magnitude peak of a transfer function

class utils.**mimotf**(*matrix*)
Represents MIMO transfer function matrix

This is a pretty basic wrapper around the numpy.matrix class which deals with most of the heavy lifting.

You can construct the object from siso tf objects similarly to calling numpy.matrix:

```
>>> G11 = G12 = G21 = G22 = tf(1, [1, 1])
>>> G = mimotf([[G11, G12], [G21, G22]])
>>> G
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

Some coersion will take place on the elements: >>> mimotf([[1]]) mimotf([[tf([1.], [1.])]])

The object knows how to do:

addition

```
>>> G + G
mimotf([[tf([2.], [1. 1.]) tf([2.], [1. 1.])]
 [tf([2.], [1. 1.]) tf([2.], [1. 1.])]])
```

```
>>> 0 + G
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

```
>>> G + 0
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

multiplication >>> G * G mimotf([[tf([2.], [1. 2. 1.]) tf([2.], [1. 2. 1.])]

[tf([2.], [1. 2. 1.]) tf([2.], [1. 2. 1.])]])

```
>>> 1*G
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

```
>>> G*1
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

```
>>> G*tf(1)
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

```
>>> tf(1)*G
mimotf([[tf([1.], [1. 1.]) tf([1.], [1. 1.])]
 [tf([1.], [1. 1.]) tf([1.], [1. 1.])]])
```

exponentiation with positive integer constants

```
>>> G**2
mimotf([[tf([2.], [1. 2. 1.]) tf([2.], [1. 2. 1.])]
 [tf([2.], [1. 2. 1.]) tf([2.], [1. 2. 1.])]])
```

Methods

__call__(s)	>>> G = mimotf([[1]]) >>> G(0) ↪matrix([[1.]])
inverse()	Calculate inverse of mimotf object
poles()	Calculate poles >>> s = tf([1, 0], [1]) >>> G = mimotf([[(s - 1) / (s + 2), 4 / (s + 2)], ...

cofactor_mat	
det	
mimotf_slice	
zeros	

inverse()

Calculate inverse of mimotf object

```
>>> s = tf([1, 0], 1)
>>> G = mimotf([[(s - 1) / (s + 2),  4 / (s + 2)],
...              [4.5 / (s + 2), 2 * (s - 1) / (s + 2)]])
>>> G.inverse()
matrix([[tf([ 1. -1.], [ 1. -4.]), tf([-2.], [ 1. -4.])],
        [tf([-2.25], [ 1. -4.]), tf([ 0.5 -0.5], [ 1. -4.])]],
       dtype=object)
```

```
>>> G.inverse()*G.matrix
matrix([[tf([1.], [1.]), tf([0.], [1])],
        [tf([0.], [1]), tf([1.], [1.])]], dtype=object)
```

poles()

Calculate poles >>> s = tf([1, 0], [1]) >>> G = mimotf([[(s - 1) / (s + 2), 4 / (s + 2)], ... [4.5 / (s + 2), 2 * (s - 1) / (s + 2)]]) >>> G.poles() array([-2.])

utils.**mimotf2sym**(*G*, *deadtime=False*)

Converts a mimotf object making use of individual tf objects to a transfer function system in sympy.Matrix form.

Parameters

G [mimotf matrix] The mimotf system matrix.

deadtime: boolean Should deadtime be added to sympy matrix or not.

Returns

Gs [sympy matrix] The sympy system matrix

s [sympy symbol] Sympy symbol generated

utils.**minimal_realisation**(*a*, *b*, *c*)

"This function will obtain a minimal realisation for a state space model in the form given in Skogestad second edition p 119 equations 4.3 and 4.4

Parameters

- **a** – numpy matrix the A matrix in the state space model
- **b** – numpy matrix the B matrix in the state space model
- **c** – numpy matrix the C matrix in the state space model

Examples

Example 1:

```
>>> A = numpy.matrix([[0, 0, 0, 0],
...                   [0, -2, 0, 0],
...                   [2.5, 2.5, -1, 0],
...                   [2.5, 2.5, 0, -3]])
```

```
>>> B = numpy.matrix([[1],
...                   [1],
...                   [0],
...                   [0]])
```

```
>>> C = numpy.matrix([0, 0, 1, 1])
```

```
>>> Aco, Bco, Cco = minimal_realisation(A, B, C)
```

Add null to eliminate negatives null elements (-0.)

```
>>> Aco.round(decimals=3) + 0.
array([[-2.038,  5.192],
       [ 0.377, -0.962]])
```

```
>>> Bco.round(decimals=3) + 0.
array([[ 0.   ],
       [-1.388]])
```

```
>>> Cco.round(decimals=3) + 0.
array([[-1.388,  0.   ]])
```

Example 2:

```
>>> A = numpy.matrix([[1, 1, 0],
...                   [0, 1, 0],
...                   [0, 1, 1]])
```

```
>>> B = numpy.matrix([[0, 1],
...                   [1, 0],
...                   [0, 1]])
```

```
>>> C = numpy.matrix([1, 1, 1])
```

```
>>> Aco, Bco, Cco = minimal_realisation(A, B, C)
```

Add null to eliminate negatives null elements (-0.)

```
>>> Aco.round(decimals=3) + 0.
array([[ 1.   ,  0.   ],
       [-1.414,  1.   ]])
```

```
>>> Bco.round(decimals=3) + 0.
array([[-1.   ,  0.   ],
       [ 0.   ,  1.414]])
```

```
>>> Cco.round(decimals=3) + 0.
array([[-1.   ,  1.414]])
```

utils.**minors**(*G*, *order*)
Returns the order minors of a MIMO tf G.

utils.**omega**(*w_start*, *w_end*)
Convenience wrapper Defines the frequency range for calculation of frequency response Frequency in rad/time where time is the time unit used in the model.

utils.**phase**(*G*, *deg=False*)
Return the phase angle in degrees or radians

Parameters

G [tf] Plant of transfer functions.

deg [booleans] True if radians result is required, otherwise degree is default (optional).

Returns

phase [float] Phase angle.

utils.**pole_zero_directions**(*G*, *vec*, *dir_type*, *display_type='a'*, *e=1e-08*, *min_tol=1e-05*, *max_tol=10000.0*)
Crude method to calculate the input and output direction of a pole or zero, from the SVD.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

vec [array] A vector containing all the transmission poles or zeros of a system.

dir_type [string] Type of direction to calculate.

dir_type	Choose
'p'	Poles
'z'	Zeros

display_type [string] Choose the type of directional data to return (optional).

display_type	Directional data to return
'a'	All data (default)
'u'	Only input direction
'y'	Only output direction

e [float] Avoid division by zero. Let epsilon be very small (optional).

min_tol [float] Acceptable tolerance for zero validation. Let min_tol be very small (optional).

max_tol [float] Acceptable tolerance for pole validation. Let max_tol be very small (optional).

Returns

pz_dir [array] Pole or zero direction in the form: (pole/zero, input direction, output direction, valid)

valid [integer array] If 1 the directions are valid, else if 0 the directions are not valid.

utils.**poles**(*G=None*, *A=None*)
If G is passed then return the poles of a multivariable transfer function system. Applies Theorem 4.4 (p135). If G is NOT specified but A is, returns the poles from the state space description as per section 4.4.2.

Parameters

G [sympy or mimotf matrix (n x n)] The transfer function G(s) of the system.

A [State Space A matrix]

Returns

pole [array] List of poles.

utils.**poles_and_zeros_of_square_tf_matrix**(*G*)

Determine poles and zeros of a square mimotf matrix, making use of the determinant. This method may fail in special cases. If terms cancel out during calculation of the determinant, not all poles and zeros will be determined.

Parameters

G [mimotf matrix (n x n)] The transfer function of the system.

Returns

z [array] List of zeros.

p [array] List of poles.

possible_cancel [boolean] Test whether terms were possibly cancelled out in determinant calculation.

utils.**polygcd**(*a*, *b*)

Find the approximate Greatest Common Divisor of two polynomials

```
>>> a = numpy.poly1d([1, 1]) * numpy.poly1d([1, 2])
>>> b = numpy.poly1d([1, 1]) * numpy.poly1d([1, 3])
>>> polygcd(a, b)
poly1d([1., 1.])
```

```
>>> polygcd(numpy.poly1d([1, 1]), numpy.poly1d([1]))
poly1d([1.])
```

utils.**scaling**(*G_hat*, *e*, *u*, *input_type='symbolic'*, *Gd_hat=None*, *d=None*)

Receives symbolic matrix of plant and disturbance transfer functions as well as array of maximum deviations, scales plant variables according to eq () and ()

Parameters

G_hat [matrix of plant WITHOUT deadtime]

e [array of maximum plant output variable deviations] in same order as G matrix plant outputs

u [array of maximum plant input variable deviations] in same order as G matrix plant inputs

input_type [specifies whether input is symbolic matrix or utils mimotf]

Gd_hat [optional] matrix of plant disturbance model WITHOUT deadtime

d [optional] array of maximum plant disturbance variable deviations in same order as Gd matrix plant disturbances

Returns

———-

G_scaled [scaled plant function]

Gd_scaled [scaled plant disturbance function]

utils.**sigmas**(*A*, *position=None*)
Returns the singular values of A

Parameters

A [array] Transfer function matrix.

position [string] Type of sigmas to return (optional).

position	Type of sigmas to return
max	Maximum singular value
min	Minimal singular value

Returns

:math:'sigma' (A) [array] Singular values of A arranged in decending order.

This is a convenience wrapper to enable easy calculation of

singular values over frequency

utils.**ssr_solve**(*A*, *B*, *C*, *D*)
Solves the zeros and poles of a state-space representation of a system.

Parameters

- **A** – System state matrix
- **B** – matrix
- **C** – matrix
- **D** – matrix

For information on the meanings of A, B, C, and D consult Skogestad 4.1.1

Returns: zeros: The system's zeros poles: the system's poles

TODO: Add any other relevant values to solve for, for example, if coprime factorisations are useful somewhere add them to this function's return dict rather than writing another function.

utils.**state_controllability**(*A*, *B*)
This method checks if the state space description of the system is state controllable according to Definition 4.1 (p127).

Parameters

A [numpy matrix] Matrix A of state-space representation.

B [numpy matrix] Matrix B of state-space representation.

Returns

state_control [boolean] True if state controllable

u_p [array] Input pole vectors for the states u_p_i

control_matrix [numpy matrix] State Controllability Matrix

utils.**state_observability_matrix**(*a*, *c*)
calculate the observability matrix

Parameters

- **a** – numpy matrix the A matrix in the state space model
- **c** – numpy matrix the C matrix in the state space model

utils.**sv_dir**(*G*, *table=False*)

Returns the input and output singular vectors associated with the minimum and maximum singular values.

Parameters

G [numpy matrix (n x n)] The transfer function G(s) of the system.

table [True of False boolean] Default set to False.

Returns

u [list of arrays containing complex numbers] Output vector associated with the maximum and minium singular values. The maximum singular output vector is the first entry u[0] and the minimum is the second u[1].

v [list of arrays containing complex numbers] Input vector associated with the maximum and minium singular values. The maximum singular intput vector is the first entry u[0] and the minimum is the second u[1].

table [If table is True then the output and input vectors are summarised] and returned as a table in the command window. Values are reported to five significant figures.

utils.**sym2mimotf**(*Gmat*, *deadtime=None*)

Converts a MIMO transfer function system in sympy.Matrix form to a mimotf object making use of individual tf objects.

Parameters

Gmat [sympy matrix] The system transfer function matrix.

deadtime: numpy matrix of same dimensions as Gmat The dead times of Gmat with corresponding indexes.

Returns

Gmimotf [sympy matrix] The mimotf system matrix

class utils.**tf**(*numerator*, *denominator=1*, *deadtime=0*, *name=''*, *u=''*, *y=''*, *prec=3*)

Very basic transfer function object

Construct with a numerator and denominator:

```
>>> G = tf(1, [1, 1])
>>> G
tf([1.], [1. 1.])
```

```
>>> G2 = tf(1, [2, 1])
```

The object knows how to do:

addition

```
>>> G + G2
tf([1.5 1. ], [1.  1.5 0.5])
>>> G + G # check for simplification
tf([2.], [1. 1.])
```

multiplication

```
>>> G * G2
tf([0.5], [1.  1.5 0.5])
```

division

```
>>> G / G2
tf([2. 1.], [1. 1.])
```

Deadtime is supported:

```
>>> G3 = tf(1, [1, 1], deadtime=2)
>>> G3
tf([1.], [1. 1.], deadtime=2)
```

Note we can't add transfer functions with different deadtime:

```
>>> G2 + G3
Traceback (most recent call last):
    ...
ValueError: Transfer functions can only be added if their deadtimes are the same.␣
↪self=tf([0.5], [1.  0.5]), other=tf([1.], [1. 1.], deadtime=2)
```

Although we can add a zero-gain tf to anything

```
>>> G2 + 0*G3
tf([0.5], [1.  0.5])
```

```
>>> 0*G2 + G3
tf([1.], [1. 1.], deadtime=2)
```

It is sometimes useful to define

```
>>> s = tf([1, 0])
>>> 1 + s
tf([1. 1.], [1.])
```

```
>>> 1/(s + 1)
tf([1.], [1. 1.])
```

Methods

__call__(s)	This allows the transfer function to be evaluated at particular values of s.
exp()	If this is basically "D*s" defined as tf([D, 0], 1), return dead time
inverse()	Inverse of the transfer function
lsim(*args)	Negative step response
step(*args)	Step response

poles	
simplify	
zeros	

exp()

If this is basically "D*s" defined as tf([D, 0], 1), return dead time

```
>>> s = tf([1, 0], 1)
>>> numpy.exp(-2*s)
tf([1.], [1.], deadtime=2.0)
```

inverse()
Inverse of the transfer function

lsim(**args*)
Negative step response

step(**args*)
Step response

utils.**tf2ss**(*H*)
Converts a mimotf object to the controllable canonical form state space representation. This method and the examples were obtained from course work notes available at http://www.egr.msu.edu/classes/me851/jchoi/lecture/Lect_20.pdf which appears to derive the method from "A Linear Systems Primer" by Antsaklis and Birkhauser.

Parameters

H [mimotf] The mimotf object transfer function form

Returns

Ac [numpy matrix] The state matrix of the observable system

Bc [nump matrix] The input matrix of the observable system

Cc [numpy matrix] The output matrix of the observable system

Dc [numpy matrix] The output matrix of the observable system

utils.**tf_step**(*G*, *t_end=10*, *initial_val=0*, *points=1000*, *constraint=None*, *Y=None*, *method='numeric'*)
Validate the step response data of a transfer function by considering dead time and constraints. A unit step response is generated.

Parameters

G [tf] Transfer function (input[u] or output[y]) to evauate step response.

Y [tf] Transfer function output[y] to evaluate constrain step response (optional) (required if constraint is specified).

t_end [integer] length of time to evaluate step response (optional).

initial_val [integer] starting value to evalaute step response (optional).

points [integer] number of iteration that will be calculated (optional).

constraint [real] The upper limit the step response cannot exceed. Is only calculated if a value is specified (optional).

method [['numeric','analytic']] The method that is used to calculate a constrainted response. A constraint value is required (optional).

Returns

timedata [array] Array of floating time values.

process [array (1 or 2 dim)] 1 or 2 dimensional array of floating process values.

utils.**zero_directions_ss**(*A*, *B*, *C*, *D*)
This function calculates the zeros with input and output directions from a state space representation using the method outlined on pg. 140

Parameters

A [numpy matrix] A matrix of state space representation

B [numpy matrix] B matrix of state space representation

C [numpy matrix] C matrix of state space representation

D [numpy matrix] D matrix of state space representation

Returns

zeros_in_out [list] zeros_in_out[i] contains a zero, input direction vector and output direction vector

utils.**zeros**(*G=None*, *A=None*, *B=None*, *C=None*, *D=None*)

Return the zeros of a multivariable transfer function system for with transfer functions or state-space. For transfer functions, Theorem 4.5 (p139) is used. For state-space, the method from Equations 4.66 and 4.67 (p138) is applied.

Parameters

G [sympy or mimotf matrix (n x n)] The transfer function G(s) of the system.

A, B, C, D [numpy matrix] State space parameters

Returns

zero [array] List of zeros.

Common features to plotting functions in this script

13.1 Default parameters

axlim [list [xmin, xmax, ymin, ymax]] A list containing the minimum and maximum limits for the x and y-axis. To autoscale a limit enter 'None' in its placeholder. The default is to allow autoscaling of the axes.

w_start [float] The x-axis valve at which to start the plot.

w_end [float] The x-axis value at which to stop plotting.

points [float] The number of data points to be used in generating the plot.

13.2 Example

def G(s):

return numpy.matrix([[s/(s+1), 1], [s**2 + 1, 1/(s+1)]])

plt.figure('Example 1') your_utilsplot_functionA(G, w_start=-5, w_end=2, axlim=[None, None, 0, 1], more_parameters) plt.show()

plt.figure('Example 2') plt.subplot(2, 1, 1) your_utilsplot_functionB(G) plt.subplot(2, 1, 2) your_utilsplot_functionC(G) plt.show()

utilsplot.**adjust_spine**(*xlabel*, *ylabel*, *x0=0*, *y0=0*, *width=1*, *height=1*)

General function to adjust the margins for subplots.

Parameters

xlabel [string] Label on the main x-axis.

ylabel [string] Label on the main x-axis.

x0 [integer] Horizontal offset of xlabel.

y0 [integer] Verticle offset of ylabel.

width [float] Scaling factor of width of subplots.

height [float] Scaling factor of height of subplots.

Returns

fig [matplotlib subplot area]

utilsplot.**bode**(*G*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *margin=False*)
Shows the bode plot for a plant model

Parameters

G [tf] Plant transfer function.

margin [boolean] Show the cross over frequencies on the plot (optional).

Returns

GM [array containing a real number] Gain margin.

PM [array containing a real number] Phase margin.

Plot [matplotlib figure]

utilsplot.**bodeclosedloop**(*G*, *K*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *margin=False*)
Shows the bode plot for a controller model

Parameters

G [tf] Plant transfer function.

K [tf] Controller transfer function.

margin [boolean] Show the cross over frequencies on the plot (optional).

utilsplot.**complexplane**(*args*, *color=True*, *marker='o'*, *msize=5*)
Plot up to 8 arguments on a complex plane (limited by the colors) Useful when you wish to compare sets of complex numbers graphically or plot your poles and zeros

Parameters

args [A list of the list of numbers to plot]

color [True if every tuple of info must be a different color] False if all must be the same color

marker [Type of marker to use] https://matplotlib.org/api/markers_api.html

msize [Size of the marker]

Example: A = [1+2j, 1-2j, 1+1j, 2-1j] B = [1+2j, 3+2j, 1, 1+2j] complexplane([A, B, [1+3j, 2+5j]], marker='+', msize=8)

utilsplot.**condtn_nm_plot**(*G*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*)
Plot of the condition number, the maximum over the minimum singular value

Parameters

G [numpy matrix] Plant model.

Returns

Plot [matplotlib figure]

utilsplot.**dis_rejctn_plot**(*G*, *Gd*, *S=None*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*)

A subplot of disturbance condition number to check for input saturation (equation 6.43, p238). Two more subplots indicate if the disturbances fall withing the bounds of S, applying equations 6.45 and 6.46 (p239).

Parameters

G [numpy matrix] Plant model.

Gd [numpy matrix] Plant disturbance model.

S [numpy matrix] Sensitivity function (optional, if available).

 # TODO test S condition

Returns

——-

Plot [matplotlib figure]

utilsplot.**freq_step_response_plot**(*G*, *K*, *Kc*, *t_end=50*, *freqtype='S'*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*)

A subplot function for both the frequency response and step response for a controlled plant

Parameters

G [tf] Plant transfer function.

K [tf] Controller transfer function.

Kc [integer] Controller constant.

t_end [integer] Time period which the step response should occur.

freqtype [string (optional)] Type of function to plot:

freqtype	Type of function to plot
S	Sensitivity function
T	Complementary sensitivity function
L	Loop function

Returns

Plot [matplotlib figure]

utilsplot.**input_acceptable_const_plot**(*G*, *Gd*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *modified=False*)

Subbplots for input constraints for accepatable control. Applies equation 6.55 (p241).

Parameters

G [numpy matrix] Plant model.

Gd [numpy matrix] Plant disturbance model.

modified [boolean] If true, the arguments in the equation are change to $\sigma_1(G) + 1 \geq |u_i^H g_d|$. This is to avoid a negative log scale.

Returns

Plot [matplotlib figure]

utilsplot.**input_perfect_const_plot**(*G*, *Gd*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *simultaneous=False*)

Plot for input constraints for perfect control. Applies equation 6.50 (p240).

Parameters

G [numpy matrix] Plant model.

Gd [numpy matrix] Plant disturbance model.

simultaneous [boolean.] If true, the induced max-norm is calculated for simultaneous disturbances (optional).

Returns

Plot [matplotlib figure]

utilsplot.**mimo_bode**(*G*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *Kin=None*, *text=False*, *sv_all=False*)

Plots the max and min singular values of G and computes the crossover frequency.

If a controller is specified, the max and min singular values of S are also plotted and the bandwidth frequency computed (p81).

Parameters

G [numpy matrix] Matrix of plant transfer functions.

Kin [numpy matrix] Controller matrix (optional).

text [boolean] If true, the crossover and bandwidth frequencies are plotted (optional).

sv_all [boolean] If true, plot all the singular values of the plant (optional).

Returns

wC [real] Crossover frequency.

wB [real] Bandwidth frequency.

Plot [matplotlib figure]

utilsplot.**mimo_nyquist_plot**(*L*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*)

Nyquist stability plot for MIMO system.

Parameters

L [numpy matrix] Closed loop transfer function matrix as a function of s, i.e. def L(s).

Returns

Plot [matplotlib figure]

utilsplot.**perf_Wp_plot**(*S*, *wB_req*, *maxSSerror*, *w_start*, *w_end*, *axlim=None*, *points=1000*)

MIMO sensitivity S and performance weight Wp plotting funtion.

Parameters

S [numpy array] Sensitivity transfer function matrix as function of s => S(s)

wB_req [float] The design or require bandwidth of the plant in rad/time. 1/time eg: wB_req = 1/20sec = 0.05rad/s

maxSSerror [float] The maximum stead state tracking error required of the plant.

wStart [float] Minimum power of w for the frequency range in rad/time. eg: for w starting at 10e-3, wStart = -3.

wEnd [float] Maximum value of w for the frequency range in rad/time. eg: for w ending at 10e3, wStart = 3.

Returns

wB [float] The actually plant bandwidth in rad/time given the specified controller used to generate the sensitivity matrix S(s).

Plot [matplotlib figure]

utilsplot.**ref_perfect_const_plot**(*G*, *R*, *wr*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *plot_type='all'*)

Use these plots to determine the constraints for perfect control in terms of combined reference changes. Equation 6.52 (p241) calculates the minimal requirement for input saturation to check in terms of set point tracking. A more tighter bounds is calculated with equation 6.53 (p241).

Parameters

G [tf] Plant transfer function.

R [numpy matrix (n x n)] Reference changes (usually diagonal with all elements larger than 1)

wr [float] Frequency up to witch reference tracking is required

type_eq [string] Type of plot:

plot_type	Type of plot
minimal	Minimal requirement, equation 6.52
tighter	Tighter requirement, equation 6.53
allo	All requirements

Returns

Plot [matplotlib figure]

utilsplot.**rga_nm_plot**(*G*, *pairing_list=None*, *pairing_names=None*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *plot_type='all'*)

Plots the RGA number for a specified pairing

Parameters

G [numpy matrix] Plant model.

pairing_list [List of sparse numpy matrices of the same shape as G.] An array of zeros with a 1. at each required output-input pairing. The default is a diagonal pairing with 1.'s on the diagonal.

plot_type [string] Type of plot:

plot_type	Type of plot
all	All the pairings on one plot
element	Each pairing has its own plot

Returns

Plot [matplotlib figure]

utilsplot.**rga_plot**(*G*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*, *fig=0*, *plot_type='elements'*, *input_label=None*, *output_label=None*)

Plots the relative gain interaction between each output and input pairing

Parameters

G [numpy matrix] Plant model.

plot_type [string] Type of plot.

plot_type	Type of plot
all	All the RGAs on one plot
output	Plots grouped by output
input	Plots grouped by input
element	Each element has its own plot

Returns

Plot [matplotlib figure]

utilsplot.**step**(*G*, *t_end=100*, *initial_val=0*, *input_label=None*, *output_label=None*, *points=1000*)
This function is similar to the MatLab step function.

Parameters

G [tf] Plant transfer function.

t_end [integer] Time period which the step response should occur (optional).

initial_val [integer] Starting value to evaluate step response (optional).

input_label [array] List of input variable labels.

output_label [array] List of output variable labels.

Returns

Plot [matplotlib figure]

utilsplot.**step_response_plot**(*Y*, *U*, *t_end=50*, *initial_val=0*, *timedim='sec'*, *axlim=None*, *points=1000*, *constraint=None*, *method='numeric'*)
A plot of the step response of a transfer function

Parameters

Y [tf] Output transfer function.

U [tf] Input transfer function.

t_end [integer] Time period which the step response should occur (optional).

initial_val [integer] Starting value to evaluate step response (optional).

constraint [float] The upper limit the step response cannot exceed. is only calculated if a value is specified (optional).

method [['numeric','analytic']] The method that is used to calculate a constrained response. A constraint value is required (optional).

Returns

Plot [matplotlib figure]

utilsplot.**sv_dir_plot**(*G*, *plot_type*, *w_start=-2*, *w_end=2*, *axlim=None*, *points=1000*)
Plot the input and output singular vectors associated with the minimum and maximum singular values.

Parameters

G [matrix] Plant model or sensitivity function.

plot_type [string] Type of plot.

plot_type	Type of plot
input	Plots input vectors
output	Plots output vectors

Returns

Plot [matplotlib figure]

Python Module Index

u

utils, 57
utilsplot, 73

Index

A

adjust_spine() (in module utilsplot), 73
arrayfun() (in module utils), 59
astf() (in module utils), 59

B

bode() (in module utilsplot), 74
bodeclosedloop() (in module utilsplot), 74
BoundKS() (in module utils), 57
BoundST() (in module utils), 57

C

circle() (in module utils), 60
Closed_loop() (in module utils), 58
complexplane() (in module utilsplot), 74
condtn_nm_plot() (in module utilsplot), 74
ControllerTuning() (in module utils), 58

D

det() (in module utils), 60
dis_rejctn_plot() (in module utilsplot), 74
distRej() (in module utils), 61
distRHPZ() (in module utils), 61

E

exp() (utils.tf method), 71

F

feedback() (in module utils), 61
feedback_mimo() (in module utils), 61
findst() (in module utils), 61
freq() (in module utils), 61
freq_step_response_plot() (in module utilsplot), 75

G

gaintf() (in module utils), 62

I

input_acceptable_const_plot() (in module utilsplot), 75
input_perfect_const_plot() (in module utilsplot), 75
inverse() (utils.mimotf method), 65
inverse() (utils.tf method), 72
IterRGA() (in module utils), 58

K

kalman_controllable() (in module utils), 62
kalman_observable() (in module utils), 62

L

lcm_of_all_minors() (in module utils), 62
listify() (in module utils), 62
lsim() (utils.tf method), 72

M

margins() (in module utils), 63
marginsclosedloop() (in module utils), 63
matrix_as_scalar() (in module utils), 63
maxpeak() (in module utils), 63
mimo_bode() (in module utilsplot), 76
mimo_nyquist_plot() (in module utilsplot), 76
mimotf (class in utils), 63
mimotf2sym() (in module utils), 65
minimal_realisation() (in module utils), 65
minors() (in module utils), 67

O

omega() (in module utils), 67

P

perf_Wp_plot() (in module utilsplot), 76
phase() (in module utils), 67
pole_zero_directions() (in module utils), 67
poles() (in module utils), 67
poles() (utils.mimotf method), 65
poles_and_zeros_of_square_tf_matrix() (in module utils), 68
polygcd() (in module utils), 68

R

ref_perfect_const_plot() (in module utilsplot), 77
RGA() (in module utils), 58
rga_nm_plot() (in module utilsplot), 77
rga_plot() (in module utilsplot), 77
RGAnumber() (in module utils), 58

S

scaling() (in module utils), 68
sigmas() (in module utils), 68
ssr_solve() (in module utils), 69
state_controllability() (in module utils), 69
state_observability_matrix() (in module utils), 69
step() (in module utilsplot), 78
step() (utils.tf method), 72
step_response_plot() (in module utilsplot), 78
sv_dir() (in module utils), 69
sv_dir_plot() (in module utilsplot), 78
SVD() (in module utils), 59
sym2mimotf() (in module utils), 70

T

tf (class in utils), 70
tf2ss() (in module utils), 72
tf_step() (in module utils), 72

U

utils (module), 57
utilsplot (module), 73

W

Wp() (in module utils), 59

Z

zero_directions_ss() (in module utils), 72
zeros() (in module utils), 73