Speaker theory derivation using SymPy

March 28, 2026

1 Speaker theory derivation using SymPy

Francis Deck

1.1 Summary

Welcome to my speaker design theory notebook! The goal of this notebook is to derive the speaker response equations from the definitions of the Thiele-Small parameters and basic electromechanical laws.

How to read this notebook: At first glance, it might be hard to tell what this "notebook" is. Is it an article? Is it a computer program? The answer to both questions is: Yes.

This notebook uses **jupyter**, which is a framework for creating notebooks that combine text, program code, and results, in a single document. It's like a lab notebook that can do things. Some people have begun using jupyter to create articles, by hiding the code, leaving just the text and results. That would make it look more like a textbook or research paper, but I prefer to leave my code out in the open for all to see. It shows what I actually did.

To make it a bit more complicated, all of my calculations will be done using **sympy**, the symbolic algebra package for Python. As a result, I've hopefully minimized my chance of making mistakes in long drawn-out derivations.

A drawback of **sympy** and most other computer algebra tools, is that they don't necessarily preserve the ordering of symbols, thus it's hard to enforce good notational conventions. The expressions will be harder to read than ones entered by hand.

For casual reading, skim past the equations and program code, and just read the text.

For deeper reading, follow the equations, but understand that they're just points along the way as I guide **sympy** towards a complete model of a speaker.

1.2 Background

Thiele-Small theory has been around for a long time. Thiele's 1963 paper scan be found online:

Thiele 1963 paper, part 1

Thiele 1963 paper, part 2

Thiele's paper uses a lumped element model, where terms of coupled differential equations are shown graphically as resistors, capacitors, etc. The "wires" in the graph represent the couplings. *I am deriving essentially the same theory* but expressing the differential equations in algebraic form.

This is easier for me to understand, and results in formulas that can be translated directly into computer code.

Thiel-Small theory is the basis for virtually all of the freely available speaker design programs. I've checked the result of my calculations against popular programs, though the accuracy of those programs is not uncontroversial – especially regarding the port dimensions. But my point is that this is not "my" theory, but is a representation of mainstream theory, to the best of my understanding.

1.3 Limitations of the theory

1. Small signal levels, meaning that the behavior of the speaker is linear. This avoids dealing with the excursion limit, and with the harmonic distortion generated by nonlinear compression of the air in the box.
2. Low frequencies, up to maybe 250 Hz, give or take. Above those frequencies, off-axis response becomes important, and for full-range drivers, the flexibility of the cone affects the response curve. But higher frequencies are also not affected much by box design, so you can fill them in using the graphs that vendors provide for their drivers.
3. On-axis response only. At higher frequencies, the angular dependence of the response curve becomes important. There are formulas for dealing with off-axis response elsewhere.

1.4 Units of measure

Everything is SI. If you want to make use of my formulas, you have to get your design parameters into SI units.

[120]:
```
import matplotlib.pyplot as plt
import sympy as sp
import numpy as np
from IPython.display import display, Math
import copy

def displayEq(sym, expr):
    '''
    Helper function to display an expression as an equation for readability
    '''
    if __name__ == '__main__':
        display(Math(sym + ' = ' + sp.latex(expr)))
```

Sympy needs to have all symbols defined. I'm putting them all in one place, to keep them from cluttering the rest of the notebook.

[121]:
```
x, v, a, X = sp.symbols('x v a X')
V = sp.symbols('V', real = True)
V_in, omega, t, f, P_in = sp.symbols('V_in omega t f P_in', real=True)
I_in = sp.symbols('I_{in}')
R_e, Z_e, L_e, BL = sp.symbols('R_e Z_e L_e BL', real=True, positive=True)
Q_es, Q_ms, w_s, rho, c, S_d, V_as, F_s = sp.symbols('Q_es Q_ms w_s rho c S_d,␣
 ↪V_as F_s', real=True, positive=True)
```

```
F_mag = sp.symbols('F_mag')
F_spring = sp.symbols('F_spring')
F_damp = sp.symbols('F_damp')
F_inertial = sp.symbols('F_inertial')
M_ms = sp.symbols('M_ms', real=True, positive=True)
R_ms = sp.symbols('R_ms', real=True, positive=True)
C_ms = sp.symbols('C_ms', real=True, positive=True)
C_eff = sp.symbols('C_eff', real=True, positive=True)
C_ms_driver = sp.symbols('C_ms_driver')
gamma, P_atm = sp.symbols('gamma P_atm', real=True)
F_box = sp.symbols('F_box')
r = sp.symbols('r', real=True, positive=True)
V_box, f_port, S_port, w_port = sp.symbols('V_box f_port S_port w_port',␣
 ↪real=True, positive=True)
X_port, m_port, F_cone = sp.symbols('X_port m_port F_cone', real=True)
R_ref, P_ref = sp.symbols('R_ref P_ref', real=True, positive=True)
L_port_m, end_correct = sp.symbols('L_port_m, end_correct')
N_d = sp.symbols('N_d')
```

2 Basic equation of motion for the driver

2.1 Definition of terms, phasor transformation

The phasor transformation is a mathematical tool that allows us to convert a time-domain signal into a frequency-domain signal. It's based on a strong limiting assumption of linear behavior, meaning that f(a + b) = f(a) + f(b). This is also called a "small signal" model.

The magnitudes V and X are assumed to be complex numbers, meaning that they carry both magnitude and phase.

The "physical laws" used here are all approximate. Each one assigns a single parameter to a physical behavior, so that the resulting equations remain simple. Empirically, the laws work pretty well in the small-signal domain, and speakers are designed to obey those laws.

[122]:
```
V_in = sp.nsimplify(V*sp.exp(1j*omega*t))
x = sp.nsimplify(X*sp.exp(1j*omega*t))
v = sp.diff(x, t)
a = sp.diff(v, t)

displayEq('V_{in}', V_in)
displayEq('x', x)
displayEq('v', v)
displayEq('a', a)
```

$V_{in} = Ve^{i\omega t}$

$x = Xe^{i\omega t}$

$v = iX\omega e^{i\omega t}$

$a = -X\omega^2 e^{i\omega t}$

2.2 Ohm's Law and Faraday's Law

Ohm's law $V = IZ_e$

Here, $Z_e = R_e + i\omega\ L_e$, which combines the resistance and inductance of the voice coil.

Faradays Law: $V = Blv$

These are summed together, to get the total voltage on the coil. Faraday's Law is based on the empirical discovery that moving a conductor through a magnetic field induces a voltage across the conductor. This is also how a dynamic microphone works. The voltage due to voice coil motion is referred to as the "back EMF" of the speaker.

Bl is the product of the magnetic field and the length of wire suspended in the field. The general form of Faraday's Law is a vector equation, but the speaker is designed so that the voice coil wire is always perpendicular to the field.

The total voltage across the terminals of the voice coil is the sum of these two terms. We enter the equation for the voltage, then solve it for the current.

Math note: When you see an imaginary number, such as $iBLX\omega$ in the following expression, what you're looking at is a 90 degree phase shift. A good interpretation here is that the *current* which is what produces the driving force on the cone, lags behind the input voltage. It also represents a form of *damping*, which is a loss of energy in the form of heat.

[123]:
```
I_in = sp.symbols('I_{in}')
I_in = sp.solve(sp.Eq(V_in, I_in*Z_e + BL*v), I_in)[0]

displayEq('I_{in}', I_in)
```

$$I_{in} = \frac{\left(-iBLX\omega + V\right) e^{i\omega t}}{Z_e}$$

2.3 Magnetic force law

Current flowing through the coil imposes a force:

$F_{mag} = BlI$

I've substituted the equation for I given above.

[124]:
```
F_mag = sp.expand(BL*I_in)

displayEq('F_{mag}', F_mag)
```

$$F_{mag} = -\frac{iBL^2X\omega e^{i\omega t}}{Z_e} + \frac{BLVe^{i\omega t}}{Z_e}$$

2.4 Hooke's Law

The suspension of the cone (spider and surround) resist its displacement. A single parameter, the "compliance," relates force and displacement. It's the reciprocal of the familiar spring constant

from physics class.

The spring force has no imaginary part, thus it represents an instant reaction to the motion of the cone, and also, results in no energy loss. It's purely an energy storage mechanism.

[125]:
```
F_spring = -x/C_ms

displayEq('F_{spring}', F_spring)
```

$$F_{spring} = -\frac{Xe^{i\omega t}}{C_{ms}}$$

2.5 Mechanical damping

The suspension is not a perfect spring. A small amount of the energy required to move the cone is converted into heat by friction in the materials. This is represented by a velocity-dependent force with a single damping constant.

Note that the velocity is represented by the derivative of displacement. Also, it's a purely imaginary value, meaning that it introduces a phase shift as well as a loss of energy in the form of heat.

[126]:
```
F_damp = -R_ms*v

displayEq('F_{damp}', F_damp)
```

$$F_{damp} = -iR_{ms}X\omega e^{i\omega t}$$

2.6 Inertial force

Newton's Law is the familiar $F = ma$ from physics class. I'm representing it as a force that resists acceleration.

[127]:
```
F_inertial = -1*M_ms*a

displayEq('F_{inertial}', F_inertial)
```

$$F_{inertial} = M_{ms}X\omega^2 e^{i\omega t}$$

2.7 Combined equation of motion

This equation is Newton's third law, which is that the sum of the forces on the cone is zero. It expresses everything we know about the speaker so far.

Now you can see both real and imaginary terms, with both positive and negative signs. This is what makes speaker theory complicated: What seems like it should be a simple matter of an input current producing a force on the cone, has now turned into a whole bunch of reactionary forces, "pointing in all directions" as it were, and all dependent on frequency in different ways.

[128]:
```
eq1 = sp.Eq(0, F_mag + F_spring + F_damp + F_inertial)

eq1
```

[128]:

$$0 = -\frac{iBL^2X\omega e^{i\omega t}}{Z_e} + \frac{BLVe^{i\omega t}}{Z_e} + M_{ms}X\omega^2 e^{i\omega t} - iR_{ms}X\omega e^{i\omega t} - \frac{Xe^{i\omega t}}{C_{ms}}$$

2.8 Solving the equation of motion

[129]:
```
x_driver = sp.solve(eq1, X)[0]

displayEq('X', x_driver)
```

$$X = \frac{BLC_{ms}V}{iBL^2C_{ms}\omega - C_{ms}M_{ms}Z_e\omega^2 + iC_{ms}R_{ms}Z_e\omega + Z_e}$$

2.9 The impedance curve

Impedance is the ratio of voltage to current, both of which have already been expressed as equations.

I don't have much use for the impedance curve in my design explorations. But it has its uses. Perhaps the most important is that electrical measurements are easier than acoustical ones. The impedance curve can be measured on the workbench, and does not require a high quality microphone or an anechoic chamber. But the electrical and acoustical behavior of the speaker are related, and you can learn a lot from the impedance curve, including the resonant frequency, the Q factors, and the port tuning frequency if there is one. Also, plotting the theoretical impedance curve is a good way to check that I've gotten my math right so far.

[130]:
```
z_driver = sp.simplify((V_in/I_in).subs(X, x_driver))

displayEq('Z', z_driver)

# Does it correctly get the DC resistance?

displayEq('Z(0)', z_driver.subs(omega, 0))
```

$$Z = \frac{-iBL^2C_{ms}\omega + C_{ms}M_{ms}Z_e\omega^2 - iC_{ms}R_{ms}Z_e\omega - Z_e}{C_{ms}M_{ms}\omega^2 - iC_{ms}R_{ms}\omega - 1}$$

$$Z(0) = Z_e$$

2.10 Unpack the Thiele-Small parameters

I'm using Wikipedia for my reference on the TS parameters:

https://en.wikipedia.org/wiki/Thiele/Small_parameters

Datasheets usually give the Thiele-Small parameters but not always the electromechanical parameters. I'm going to treat the definitions of the Thiele-Small parameters as 4 equations and solve for the elctromechanical parameters. The result is a set of formulas for computing the EM parameters.

I've elaborated on the constants $c^2\rho$ in the section about the sealed box.

[131]:
```
eq2 = sp.Eq(w_s, 1/sp.sqrt(C_ms*M_ms))
eq3 = sp.Eq(Q_es, R_e/BL**2*sp.sqrt(M_ms/C_ms))
eq4 = sp.Eq(Q_ms, sp.sqrt(M_ms/C_ms)/R_ms)
```

```
eq5 = sp.Eq(V_as, rho*c**2*S_d**2*C_ms)

params = sp.solve((eq2, eq3, eq4, eq5), (C_ms, M_ms, R_ms, BL))[0]
if __name__ == '__main__':
    display(eq2)
    display(eq3)
    display(eq4)
    display(eq5)
    display('Formulas for computing EM parameters from TS parameters')
    displayEq('C_{ms}', params[0])
    displayEq('M_{ms}', params[1])
    displayEq('R_{ms}', params[2])
    displayEq('BL', params[3])

em_parameters = {C_ms: params[0], M_ms: params[1], R_ms: params[2], BL:␣
 ↪params[3]}
```

$$w_s = \frac{1}{\sqrt{C_{ms}}\sqrt{M_{ms}}}$$

$$Q_{es} = \frac{\sqrt{M_{ms}}R_e}{BL^2\sqrt{C_{ms}}}$$

$$Q_{ms} = \frac{\sqrt{M_{ms}}}{\sqrt{C_{ms}}R_{ms}}$$

$$V_{as} = C_{ms}S_d^2c^2\rho$$

```
'Formulas for computing EM parameters from TS parameters'
```

$$C_{ms} = \frac{V_{as}}{S_d^2c^2\rho}$$

$$M_{ms} = \frac{S_d^2c^2\rho}{V_{as}w_s^2}$$

$$R_{ms} = \frac{S_d^2c^2\rho}{Q_{ms}V_{as}w_s}$$

$$BL = \frac{\sqrt{R_e}S_dc\sqrt{\rho}}{\sqrt{Q_{es}}\sqrt{V_{as}}\sqrt{w_s}}$$

2.11 Choosing a driver and box for examples

I've chosen the Eminence DeltaLite 2512-ii driver because I have one in a cabinet that I'm happy with. The data are from here:

https://eminence.com/products/deltalite_ii_2512#specifications

While some of the EM parameters are given, I'm going to compute them from the TS parameters anyway, to check that my calculations work.

I've included all of the parameters from the datasheet. The ones that are commented out are not used yet, or are computed by my code.

In anticipation of getting to the box and port theories, I've added parameters for a basic ported box.

Dealing with multiple drivers and ports, read carefully

My derivations deal with multiple drivers and ports in the following ways:

1. All of the physics computation is done with a single-driver and single-port model. My only personal interest is in building single-driver systems.
2. The parameter N_d sets the number of drivers.
3. The total box volume is divided by the number of drivers, so the compliance is computed on a liters-per-driver basis.
4. The input voltage is computed from the input power and nominal impedance. For instance if you want to model two 8-Ω drivers in parallel at a total input power of 200 W, you should enter a nominal impedance of 4 Ω.
5. *I've done nothing about the number of ports*, since it's your job to enter the correct total port area anyway. My program only cares about the port area, and not the shape of the ports.
6. *I'm not going to worry about impedance.*. The impedance curve will be for a single driver, since I don't know if you want your design to have drivers in series or parallel.
7. There will be a separate notebook for testing whether my treatment of multiple drivers makes sense.

[132]:
```
driver_params = {
    # Thiele-Small parameters from Eminence DeltaLite 2512-ii driver
    'f_s': 53.1, # Resonant frequency, Hz
    w_s: sp.N(53.1*2*sp.pi), # Resonant angular frequency, Hz
    R_e: 5.28, # DC resistance, Ohms
    L_e: 0.31*1e-3, # Voice coil inductance, H
    Q_ms: 2.94, # Mechanical Q factor
    Q_es: 0.64, # Electrical Q factor
    V_as: 67.44*1e-3, # Compliance equivalent air volume, m^3
    'X_max': 4.9*1e-3, # Maximum linear excursion, m
    S_d: 519.5*1e-4, # Diaphragm area, m^2
    # Additional physical constants
    rho: 1.2, # Air density, kg/m^3
    c: 343, # Speed of sound, m/s
    # System parameters
    R_ref: 1.0, # Reference distance for SPL calculation, m
    P_ref: 20e-6, # Reference sound pressure for SPL calculation, Pa
}

def finish_driver_params(driver_params):
    '''
```

```
    These are parameter conversions needed by every analysis
    '''
    # Convert Watts to amplitude in Volts. This is the *amplitude* of the␣
 ↪sinusoid
    # not the RMS value. Also, combine resistance and inductance
    driver_params[Z_e] = driver_params[R_e] + 1j*omega*driver_params[L_e]
    # Frequency to angular frequency
    driver_params[w_s] = sp.N(driver_params['f_s']*2*sp.pi)
    driver_params['Q_ts'] = 1/(1/driver_params[Q_es] + 1/driver_params[Q_ms])

finish_driver_params(driver_params)

'''
I'm putting the box parameters here, even though I won't use them until later,
so that my entire "design" is all in one place.
'''

box_params = {
    N_d: 1, # Number of drivers
    V_box: 32*1e-3, # Box volume, liters converted to m^3
    f_port: 40,
    S_port: 21*3.5*1e-4, # Port area, cm*cm converted to m^2
    end_correct: 0.732,
    'P_in_rms': 100,
    'Znom': 8,
}

box_params[w_port] = 2*sp.pi*box_params[f_port]
```

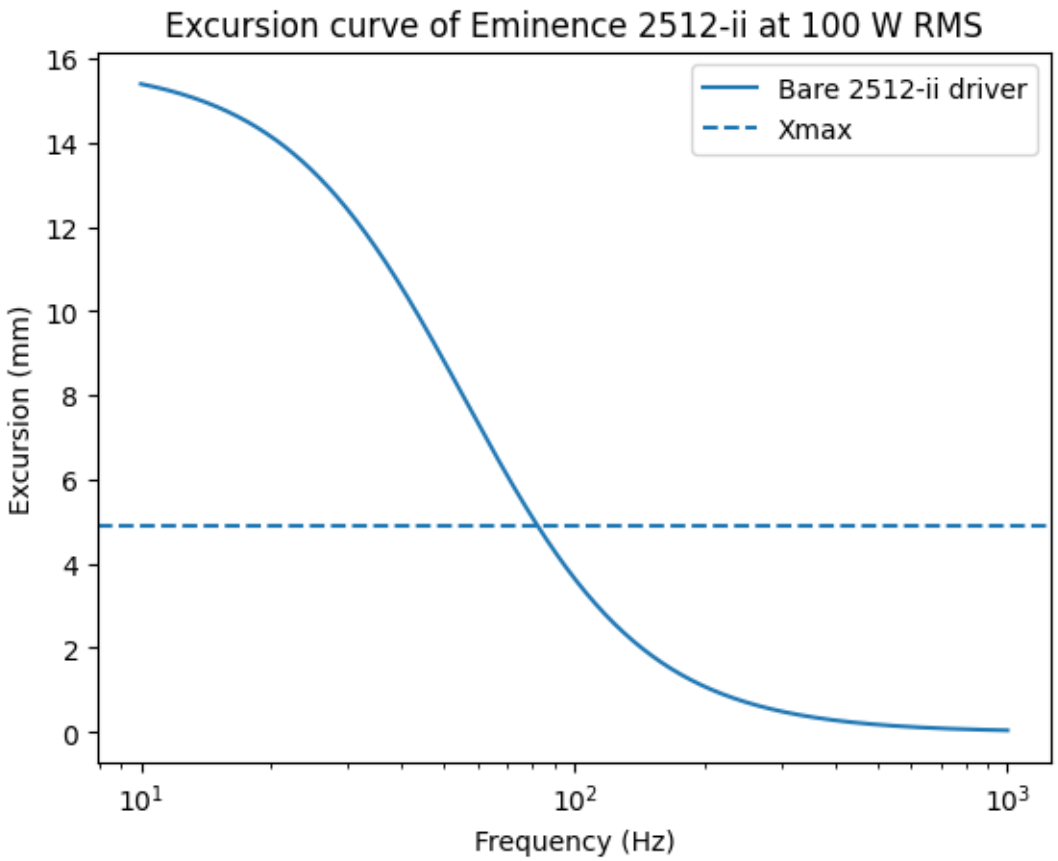
This code creates a set of electromechanical parameters for the driver and box.

[133]:
```
def build_em_params(driver_params, box_params):
    '''
    deepcopy() creates an independent copy, that we can change without changing
    the original.
    '''
    em_params = copy.deepcopy(driver_params)
    # driver_params[V] = sp.
 ↪sqrt(2*driver_params['P_in_rms']*driver_params['Znom'])
    em_params |= {sym: p.subs(driver_params)
            for sym, p in zip([C_ms, M_ms, R_ms, BL], params)}
    em_params[C_ms_driver] = em_params[C_ms]

    em_params |= box_params
    em_params[V] = sp.sqrt(2*em_params['P_in_rms']*em_params['Znom'])
    return em_params
```

```
em_params = build_em_params(driver_params, box_params)
```

2.12 Generate an excursion curve for the bare driver

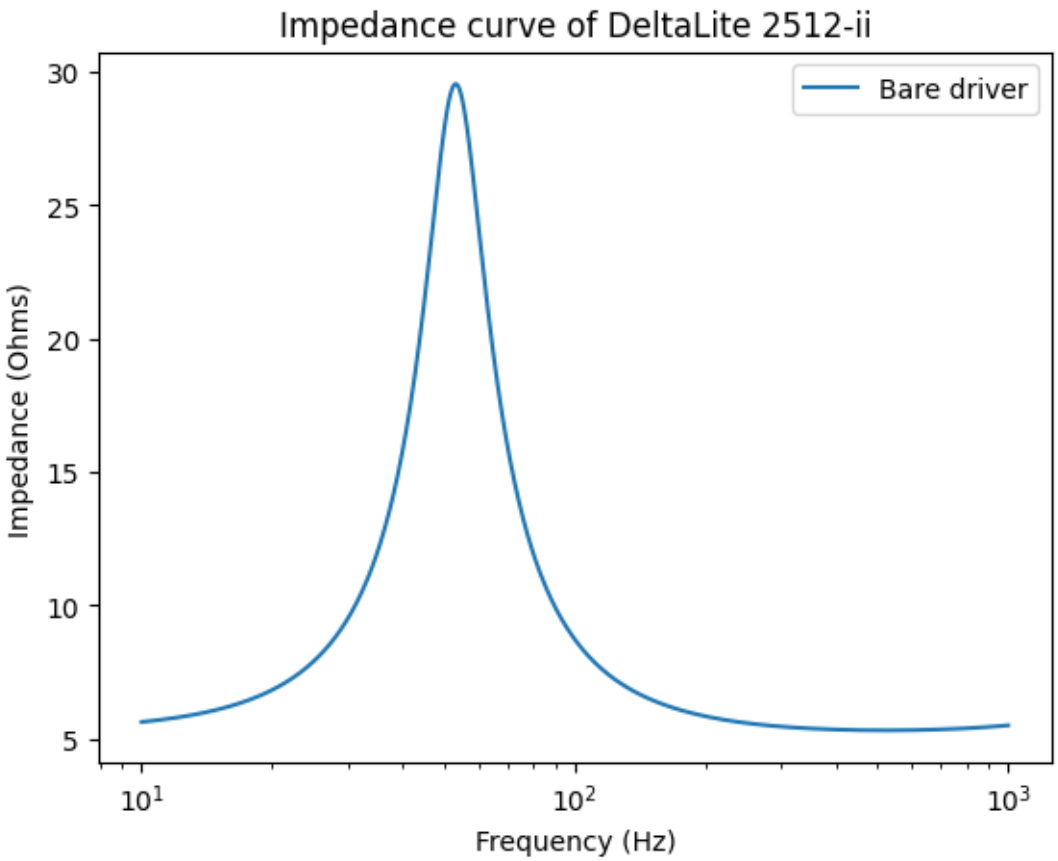
[134]:
```
fa = np.logspace(1, 3, 1000)
wa = 2*np.pi*fa

def excursion_curve(em_params, ax, verbose = False, label = ''):
    '''
    First, substitute the EM parameters into the excursion function. This
    results in an expression that's purely a function of omega. It's ugly,
    but not meant to be readable.
    '''
    xfunc_sp = x_driver.subs(em_params)
    if verbose:
        display(xfunc_sp)
    '''
    Next, convert into a numpy function, for speedy computation. If you've
    got a decent computer, you'll notice that doing all of this math␣
 ↪symbolically
    is barely slowing down the computation at all.
    '''
    xfunc_np = sp.lambdify(omega, xfunc_sp, 'numpy')
    '''
    Finally, plot it.
    '''
    ax.semilogx(fa, np.abs(xfunc_np(wa))*1000, label=label)
    last_line = ax.get_lines()[-1]
    last_line_color = last_line.get_color()
    ax.axhline(em_params['X_max']*1000, color=last_line_color, linestyle='--',␣
 ↪label = 'Xmax')
    ax.set_ylabel('Excursion (mm)')
    ax.legend()

if __name__ == '__main__':
    excursion_curve(em_params, plt.gca(), label = 'Bare 2512-ii driver')
    plt.title('Excursion curve of Eminence 2512-ii at ' +␣
 ↪str(em_params['P_in_rms']) + ' W RMS')
    plt.xlabel('Frequency (Hz)')
    plt.show()
```



2.13 Generate an impedance curve for the bare driver

Same basic schtick.

[135]:
```
def impedance_curve(em_params, ax, label = ''):
    zfunc_sp = z_driver.subs(em_params)
    zfunc_np = sp.lambdify(omega, zfunc_sp, 'numpy')
    zabs = np.abs(zfunc_np(wa))
    ax.semilogx(fa, zabs, label = label)
    ax.set_ylabel('Impedance (Ohms)')
    ax.legend()

if __name__ == '__main__':
    impedance_curve(em_params, plt.gca(), label = 'Bare driver')
    plt.xlabel('Frequency (Hz)')
    plt.title('Impedance curve of DeltaLite 2512-ii')
    plt.show()
```



2.14 The sealed box

Excursion of the cone changes the volume inside the box, and thus its pressure. In turn, pressure imparts a force on the cone. This restoring force, proportional to displacement, works exactly like the restoring force of the suspension. First, the change of volume…

S_d is the frontal area of a single driver.

[136]:
```
dV = (S_d)*X
displayEq('dV', dV)
```

$dV = S_dX$

… resulting in a change of pressure. Some details are given here. I've borrowed the equation for adiabatic compression, and expressed it in this way:

$$P = P_{atm}(\frac{V}{V_{box}})^\gamma$$

where P_atm is the atmospheric pressure and γ is a thermodynamic constant equal to roughly 1.4 for air. And making an approximation:

$$\Delta P = \Delta V \frac{dP}{dV}$$

Thus, $\Delta P = \Delta V \frac{P_{atm}\gamma}{V_{box}}$

However, the constants γP_{atm} can be replaced by ρc^2 where ρ is the density of air and c is the speed of sound in air. I'm using ρc^2 because they were already introduced in the section on the Thiele-Small parameters.

https://en.wikipedia.org/wiki/Adiabatic_process

https://en.wikipedia.org/wiki/Speed_of_sound

The next equation here is where we introduce the number of drivers for the first time.

I'm going to measure the pressure change based on a single driver pushing against the air in a fraction of the box volume determined by V_{box}/N_d. *Remember, if this gets confusing, set N_d equal to 1 and ignore the number of drivers until you understand what's going on.*

[137]:
```
dP = (gamma*P_atm*dV/(V_box/N_d)).subs(gamma*P_atm, c**2*rho)
displayEq('dP', dP)
```

$$dP = \frac{N_d S_d X c^2 \rho}{V_{box}}$$

... resulting in a force.

[138]:
```
F_box = dP*S_d
displayEq('F_{box}', F_box)
```

$$F_{box} = \frac{N_d S_d^2 X c^2 \rho}{V_{box}}$$

What have we got here? A force that's proportional to displacement, just like a spring. Thus we can express the effect of the box as a *compliance* which is the reciprocal of the spring constant from physics class.

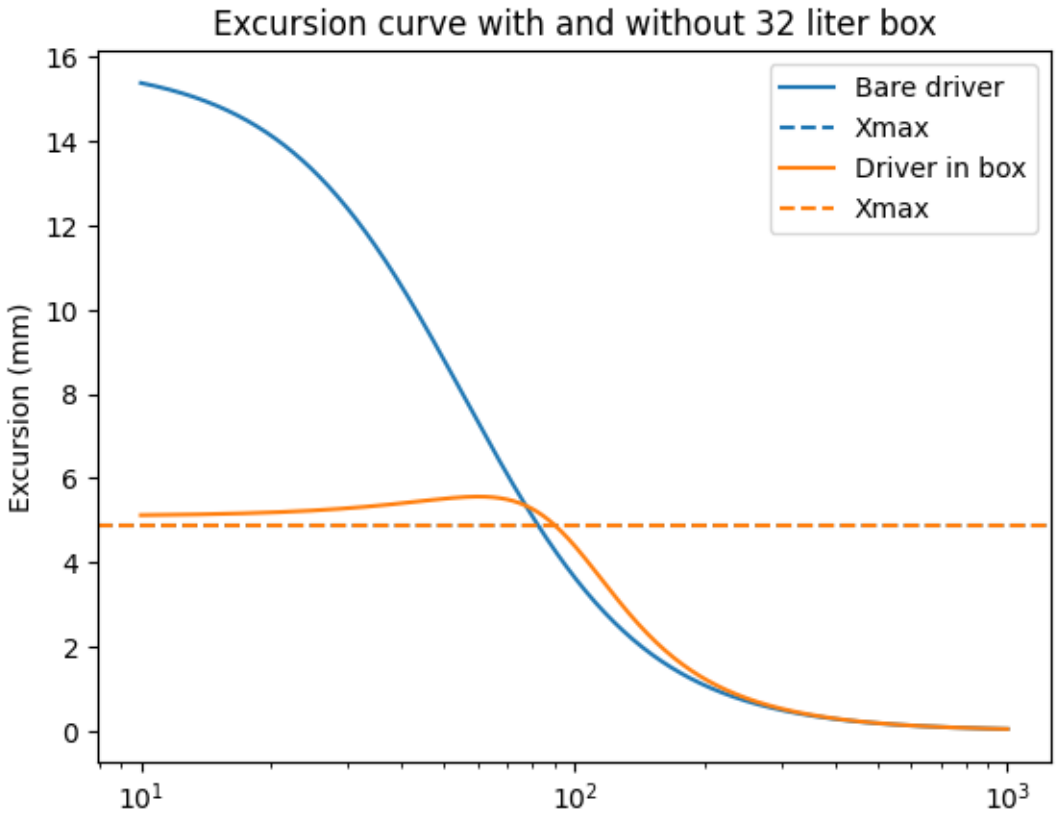
[139]:
```
C_box = X/F_box

displayEq('C_{box}', C_box)
```

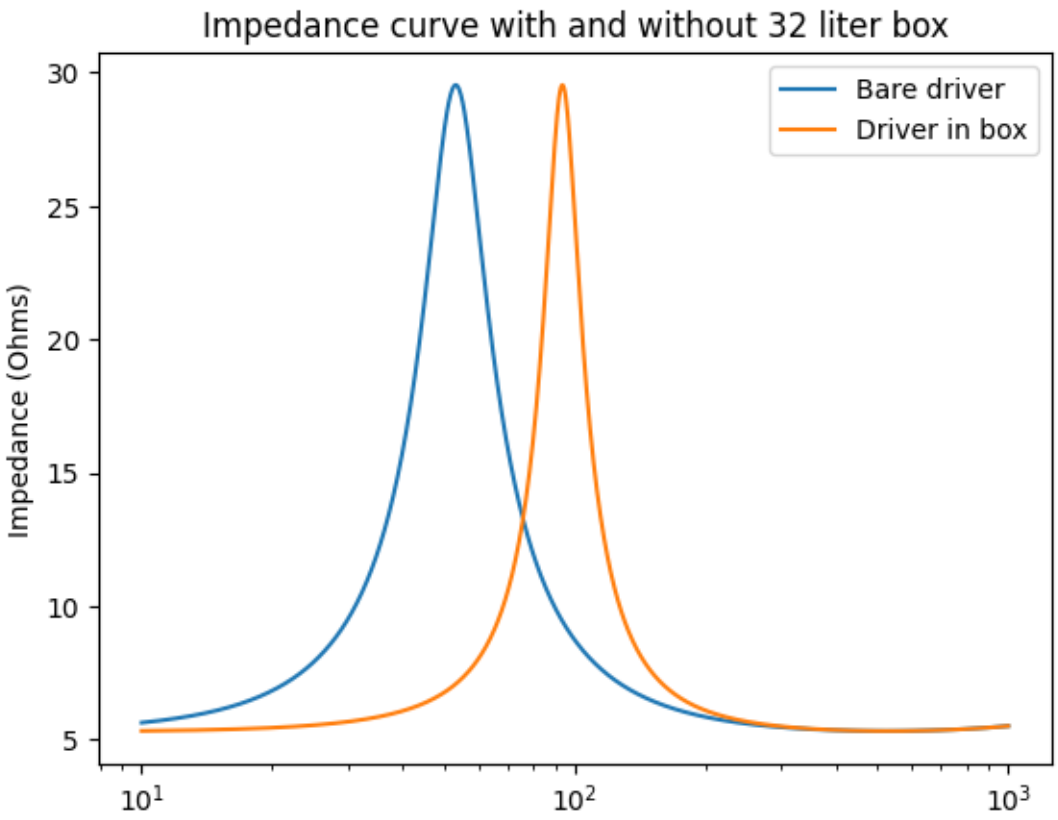
$$C_{box} = \frac{V_{box}}{N_d S_d^2 c^2 \rho}$$

Now we can model the effect of the box. I've just combined the compliances of the driver and the box in parallel. You can see the benefit of the box. Below the resonant frequency, excursion remains roughly constant, which protects the driver from damage due to exceeding its mechanical limit. When we look at the SPL curve, we'll see that we've paid a price in low frequency extension, and gained a small but manageable "hump" in the curve.

[140]:
```
# deepcopy lets us modify em_box_params without changing the contents of␣
 ↪em_params
em_box_params = copy.deepcopy(em_params)
em_box_params[C_ms] = 1/(1/em_params[C_ms] + 1/(C_box.subs(driver_params)))
if __name__ == '__main__':
```

```
excursion_curve(em_params, plt.gca(), label = 'Bare driver')
excursion_curve(em_box_params, plt.gca(), label = 'Driver in box')
plt.title('Excursion curve with and without 32 liter box')
plt.show()
impedance_curve(em_params, plt.gca(), label = 'Bare driver')
impedance_curve(em_box_params, plt.gca() ,label = 'Driver in box')
plt.title('Impedance curve with and without 32 liter box')
plt.show()
```





2.15 Acoustical output

This is where the number of drivers comes back in.

I've found this reference, from a set of acoustics lecture notes. The author derives the "far field" response of a perfect piston radiator in an infinite baffle.

https://jontallen.ece.illinois.edu/uploads/473.F18/Lectures/Chapter_7b.pdf

The equation on p. 23 is:

$$p(r,\Theta,t) = \frac{j\omega\rho_0 a^2 U_o}{2r}$$

$$e^{j(\omega t - kr)}$$

$$\frac{2J_1(ka\sin\Theta)}{ka\sin\Theta}$$

I'm only interested in the amplitude of p,

$$P(r) = \frac{j\omega\rho a^2 U_o}{2r}$$

U_o = Linear velocity of the radiator

a = Radius of cone

r = Distance to listening position

Converting to our symbol convention, we're left with:

$$P = \frac{\omega^2 \rho N_d S_d X}{2\pi r}$$

Further conversion to a dB scale requires a reference pressure. The conventional value is $P_{ref} = 20\mu Pa$. And pressure is an amplitude not a power, so we multiply the log by 20:

$SPL = 20\log_{10}(P/P_{ref})$.

Because the logarithm produces a dimensionless number, you always want to specify an input power and distance for your SPL, such as:

$SPL@1W@1m$

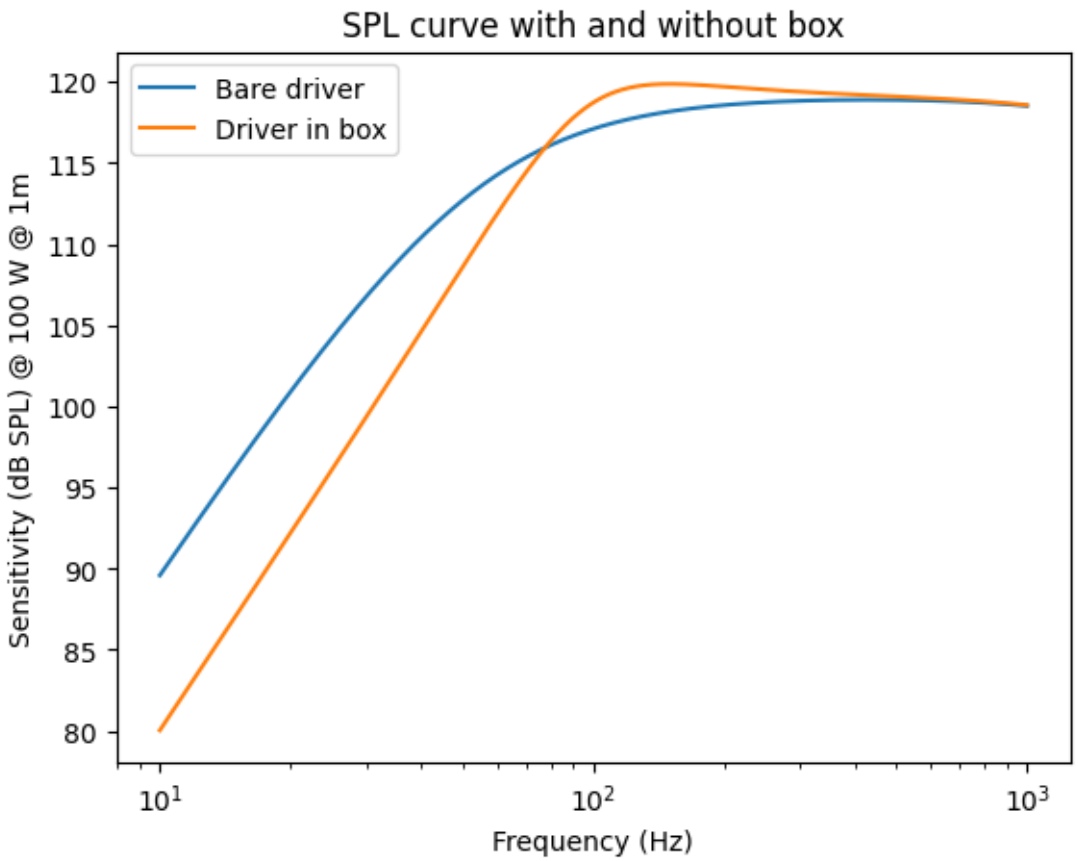
or voltage:

$SPL@2.83VRMS@1m$

Being careless with units turns physics units into marketing units, just saying.

[141]:
```
P_ratio = x_driver*omega**2*rho*N_d*S_d/2/sp.pi/R_ref/P_ref
displayEq('P_{ratio}', P_ratio)

def sensitivity_curve(em_params, ax, label = ''):
    pfunc_sp = P_ratio.subs(em_params)
    pfunc_np = sp.lambdify(omega, pfunc_sp, 'numpy')
    pabs = np.abs(pfunc_np(wa))
    pabs_db = 20*np.log10(pabs)
    ax.semilogx(fa, pabs_db, label = label)
    ax.legend()

if __name__ == '__main__':
    sensitivity_curve(em_params, plt.gca(), 'Bare driver')
    sensitivity_curve(em_box_params, plt.gca(), 'Driver in box')
    plt.title('SPL curve with and without box')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Sensitivity (dB SPL) @ ' + str(em_params['P_in_rms']) + ' W @␣
 ↪1m')
    plt.show()
```

$$P_{ratio} = \frac{BLC_{ms}N_dS_dV\omega^2\rho}{2\pi P_{ref}R_{ref}\left(iBL^2C_{ms}\omega - C_{ms}M_{ms}Z_e\omega^2 + iC_{ms}R_{ms}Z_e\omega + Z_e\right)}$$



2.16 How the port works

Introducing the symbols:

S_d = Frontal area of cone(s)

S_p = Frontal area of port

x = Displacement of the cone.

x_p = Displacement of the "slug" of air inside the port

m_{ms} = Mass of cone

m_p = Mass of air inside the port

The port air mass is modeled as a piston, just as the cone is. The total change of volume is equal to the sum of the volumes displaced by the cone and the port:

[142]:
```
dV = S_port/N_d*X_port + S_d*X

displayEq('dV', dV)
```

$$dV = S_d X + \frac{S_{port} X_{port}}{N_d}$$

Change in pressure within the box. My previous derivation by hand used γP_{atm}, so I'll make the substitution here.

[143]:
```
dP = sp.expand(-gamma*P_atm*dV/(V_box/N_d)).subs(gamma*P_atm, c**2*rho)

displayEq('dP', dP)
```

$$dP = -\frac{N_d S_d X c^2 \rho}{V_{box}} - \frac{S_{port} X_{port} c^2 \rho}{V_{box}}$$

Equation for the force on the cone from the pressure in the box

[144]:
```
eq1 = sp.Eq(F_cone, sp.expand(dP*S_d))

eq1
```

[144]:
$$F_{cone} = -\frac{N_d S_d^2 X c^2 \rho}{V_{box}} - \frac{S_d S_{port} X_{port} c^2 \rho}{V_{box}}$$

I'm going to anticipate the progress of this derivation, based on having done it by hand in the past. I'll define m_p in terms of the port resonant frequency. This is mainly aesthetic, to make the equations look more symmetrical. But also, the resonant frequency is typically what you plug into a speaker design program.

[145]:
```
m_port = (gamma*P_atm*(S_port)**2/(V_box)/w_port**2).subs(gamma*P_atm, c**2*rho)

displayEq('m_{port}', m_port)
```

$$m_{port} = \frac{S_{port}^2 c^2 \rho}{V_{box} w_{port}^2}$$

Equation for the force on the port, including the inertial force on the port mass, where the mass will be defined as above.

[146]:
```
eq2 = sp.Eq(0, sp.expand(omega**2*X_port*m_port + dP*S_port))

eq2
```

[146]:
$$0 = -\frac{N_d S_d S_{port} X c^2 \rho}{V_{box}} + \frac{S_{port}^2 X_{port} c^2 \omega^2 \rho}{V_{box} w_{port}^2} - \frac{S_{port}^2 X_{port} c^2 \rho}{V_{box}}$$

These are two equations in two variables, can be solved for the cone and port motion

[147]:
```
result = sp.solve([eq1, eq2], [X, X_port])

displayEq('X_{cone}', result[X])
displayEq('X_{port}', result[X_port])
```

$$X_{cone} = \frac{-F_{cone} V_{box} \omega^2 + F_{cone} V_{box} w_{port}^2}{N_d S_d^2 c^2 \omega^2 \rho}$$

$$X_{port} = -\frac{F_{cone} V_{box} w_{port}^2}{S_d S_{port} c^2 \omega^2 \rho}$$

Displacement divided by force is a compliance. This compliance can be added in parallel to the cone compliance.

For a bit more explanation, a compliance is the reciprocal of a spring constant. If this were expressed as a spring constant, it would represent how much extra resistance the cone "feels" because of the port on the other side of the box.

[148]:
```
C_ported = sp.simplify(-result[X]/F_cone)

displayEq('C_{ported}', C_ported)
```

$$C_{ported} = \frac{V_{box} \left(\omega^2 - w_{port}^2\right)}{N_d S_d^2 c^2 \omega^2 \rho}$$

Isolate the relationship between the cone and port excursions.

See that the box volume has dropped out. The cone only "cares" about the port tuning frequency!

[149]:
```
port_cone_frac = sp.simplify(result[X_port]/result[X])

displayEq('X_{port}/X', port_cone_frac)
```

$$X_{port}/X = \frac{N_d S_d w_{port}^2}{S_{port} \left(\omega^2 - w_{port}^2\right)}$$

Defining a symbol

$$\kappa = \frac{\omega^2}{\omega^2 - \omega_p^2},$$

compute the total displacement volume

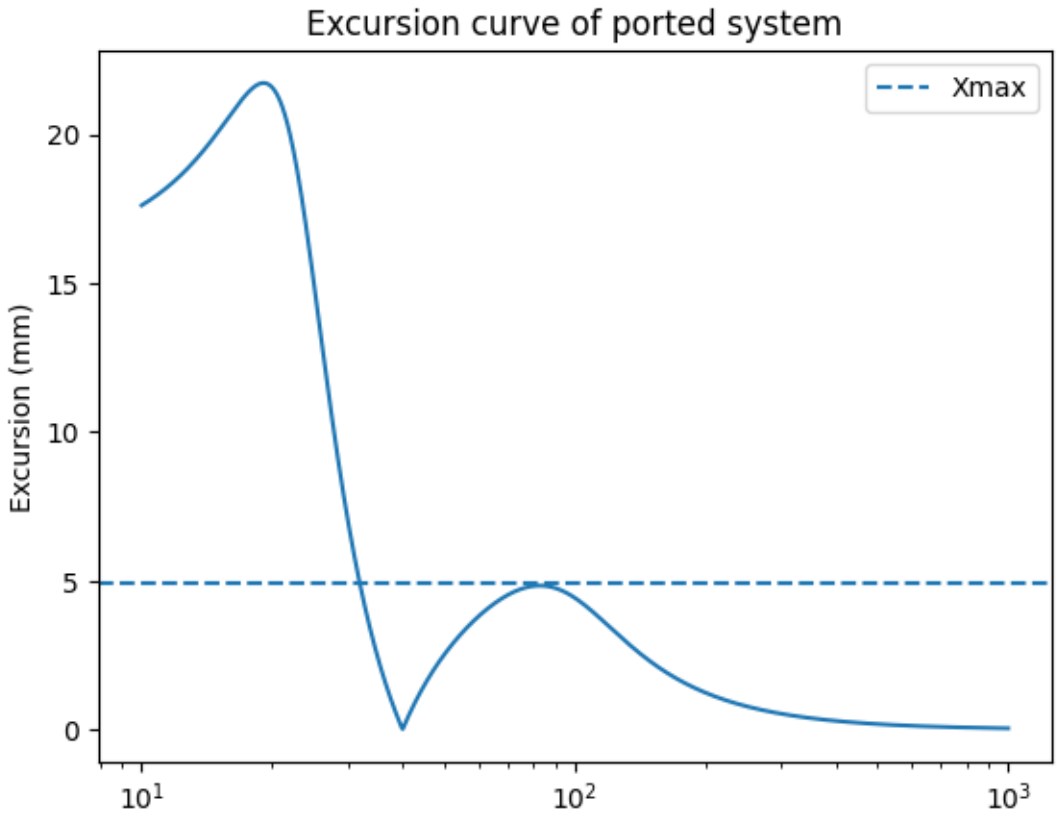
$$S_d x + S_{port} x_{port},$$

which produces the outgoing acoustic wavefront.

Now we get to see if the simulation still works. First, we need to fill in the new parameters added to the model: The port tuning frequency and box volume.

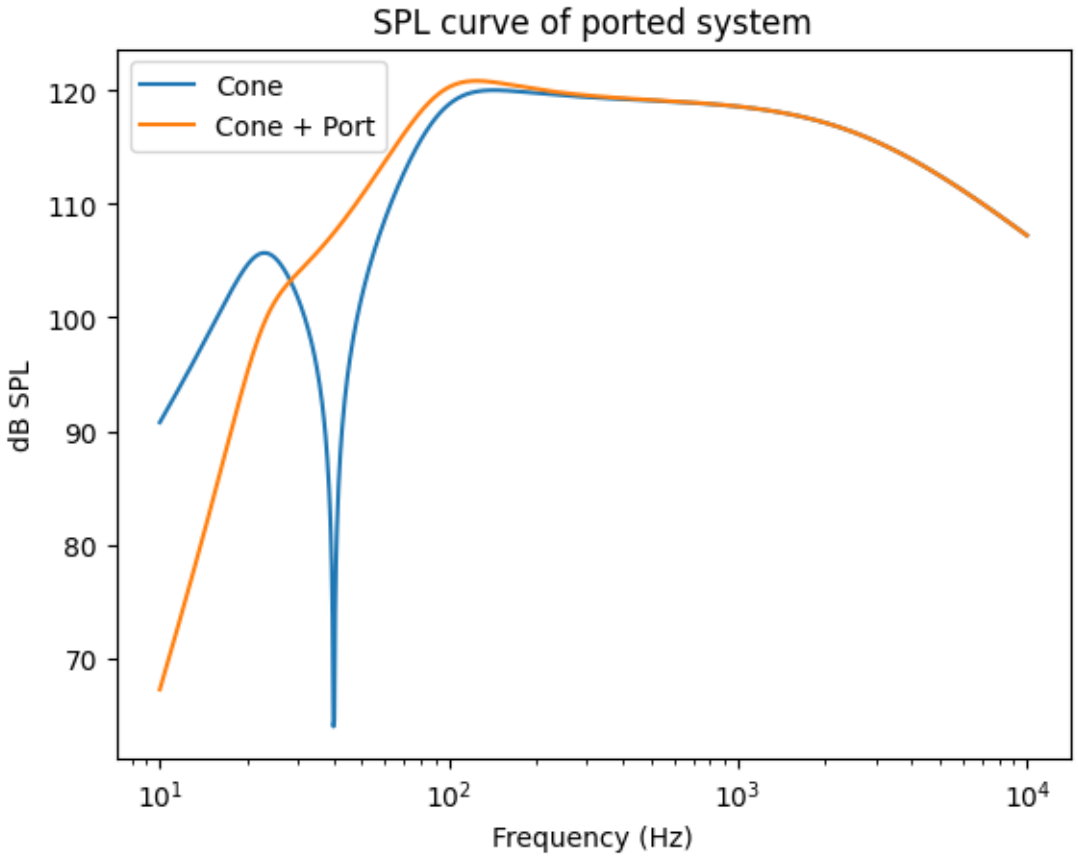
[150]:
```
def build_ported_params(em_params, f_port_hz):
    em_params[w_port] = sp.N(f_port_hz*2*sp.pi)
    em_ported_params = copy.deepcopy(em_params)
    em_ported_params[C_ms] = 1/(1/em_params[C_ms] + 1/C_ported.subs(em_params))
    lport = m_port/rho/S_port - sp.sqrt(4*S_port/sp.
 ↪pi)*em_ported_params[end_correct]
    em_ported_params[L_port_m] = sp.N(lport.subs(em_ported_params))
    return em_ported_params

em_ported_params  = build_ported_params(em_params, 40)
if __name__ == '__main__':
```

```
    excursion_curve(em_ported_params, plt.gca())
    plt.title('Excursion curve of ported system')
    plt.show()
```



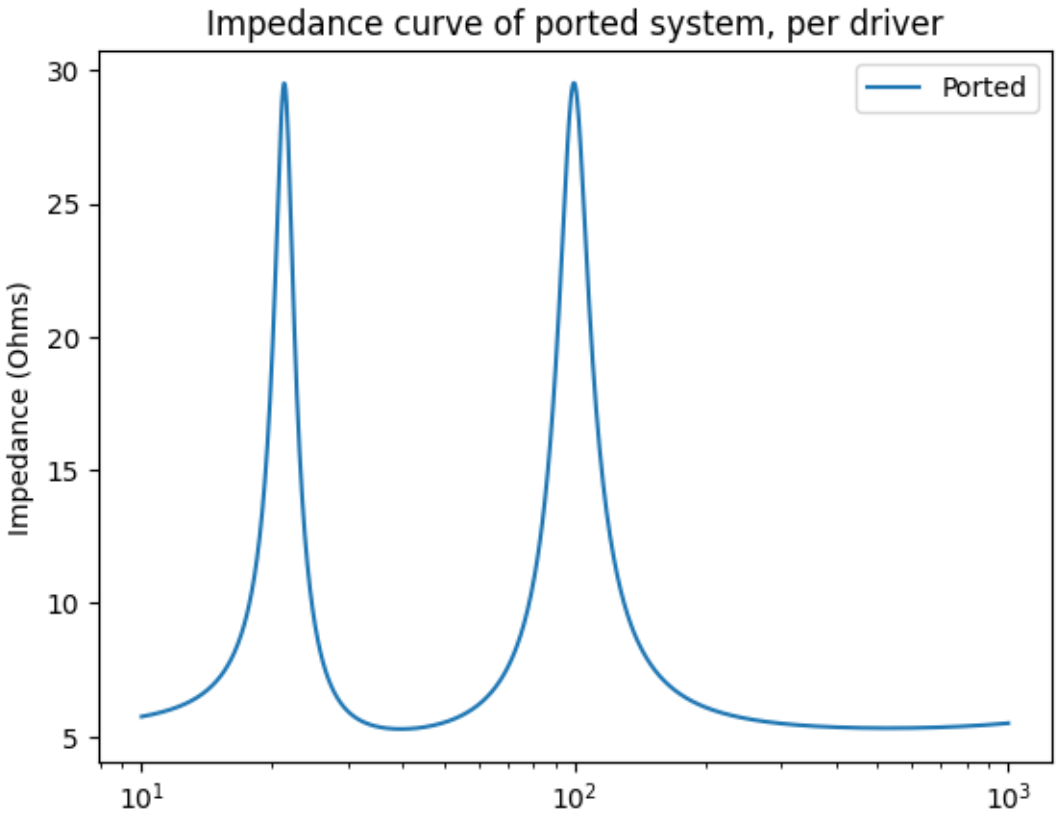
[151]:
```
def sensitivity_curve_ported(em_params, ax, label, verbose = False, show_cone =␣
↪False):
    fa = np.logspace(1, 4, 1000)
    wa = 2*np.pi*fa
    P_driver_ratio = omega**2*rho*S_d*N_d/2/sp.pi/R_ref*x_driver/P_ref
    pfunc_sp = P_driver_ratio.subs(em_params)
    pfunc_np = sp.lambdify(omega, pfunc_sp, 'numpy')
    pabs = np.abs(pfunc_np(wa))
    pabs_db = 20*np.log10(pabs)
    if show_cone:
        ax.semilogx(fa, pabs_db, label = 'Cone')

    kappa = (omega**2/(omega**2 - w_port**2))
    P_total_ratio = (omega**2*rho*S_d*N_d/2/sp.pi/R_ref*x_driver*kappa/P_ref).
↪subs(em_params)
    if verbose:
```

```
        if __name__ == '__main__':
            display(P_total_ratio)
    ptotalfunc_sp = sp.lambdify(omega, P_total_ratio, 'numpy')
    ptotalabs = np.abs(ptotalfunc_sp(wa))
    ptotalabs_db = 20*np.log10(ptotalabs)
    ax.semilogx(fa, ptotalabs_db, label = label)
    ax.set_ylabel('dB SPL')
    ax.legend()

if __name__ == '__main__':
    sensitivity_curve_ported(em_ported_params, plt.gca(), label = 'Cone +␣
 ↪Port', show_cone = True)
    plt.title('SPL curve of ported system')
    plt.xlabel('Frequency (Hz)')
    plt.show()
```



Here's something to notice about the impedance curve. It goes to a minimum at the port tuning frequency. In fact, this is a way to find out what the tuning frequency is, if you can measure the impedance curve.

[152]:
```
if __name__ == '__main__':
    impedance_curve(em_ported_params, plt.gca(), label = 'Ported')
    plt.title('Impedance curve of ported system, per driver')
    plt.show()
```

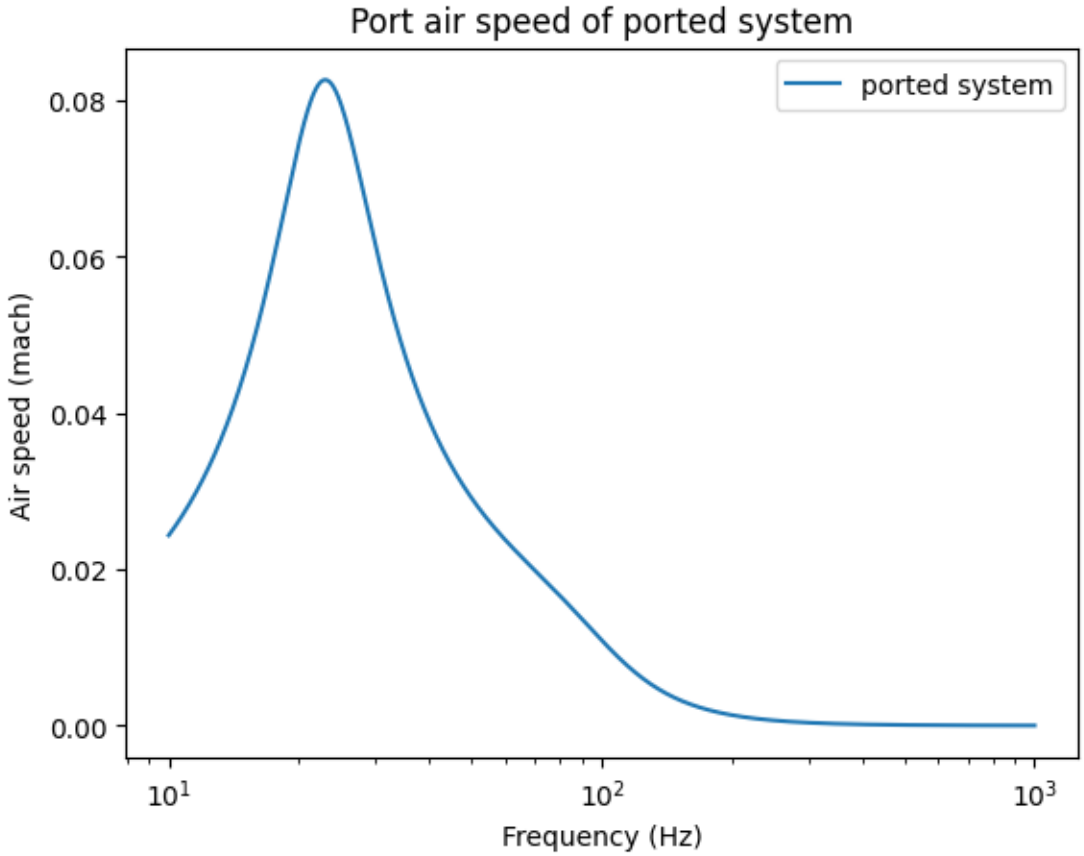


2.17 Port air speed

The number of drivers comes back in here!

[153]:
```
def airspeed_curve_ported(em_params, ax, label, verbose = False):
    vfunc_sp = (1j*omega*x_driver*port_cone_frac/c).subs(em_params)
    if verbose:
        displayEq('v_{func}', vfunc_sp)
    vfunc_np = sp.lambdify(omega, vfunc_sp, 'numpy')
    vabs = np.abs(vfunc_np(wa))
    ax.semilogx(fa, vabs, label = label)
    ax.set_ylabel('Air speed (mach)')
    ax.legend()

if __name__ == '__main__':
    airspeed_curve_ported(em_ported_params, plt.gca(), 'ported system')
```

```
    plt.title('Port air speed of ported system')
    plt.xlabel('Frequency (Hz)')
    plt.show()
```



2.18 Port length, crudely

There are more detailed port length calculators, that take the "end effects" of the port into account. I'm going to leave those for now, and look strictly at the air mass inside the port, which has this volume:

[154]:
```
V_port = m_port/rho
V_port
```

[154]:
$$\frac{S_{port}^2 c^2}{V_{box} w_{port}^2}$$

Thus the length is the volume divided by the area. But the effective air mass moved by the port is slightly longer than the port itself, an end correction factor times the port diameter. I'll model the port diameter as $D = \sqrt{4S_{port}/\pi}$

[155]:
```
L_port = V_port/S_port - end_correct*sp.sqrt(4*S_port/sp.pi)
L_port
```

[155]:
$$-\frac{2\sqrt{S_{port}}end_{correct}}{\sqrt{\pi}} + \frac{S_{port}c^2}{V_{box}w_{port}^2}$$

… and we can hang some numbers on it, in meters of course:

[156]:
```
displayEq('L_{port}', sp.N(L_port.subs(em_ported_params)))
```

$L_{port} = 0.356992544111562$

2.19 Conversion of formulas to Javascript

I've tried a few different approaches to writing my online modeling program. I tried Python, using the **flet** package, but it generates a gigantic pile of files, takes forever to load on the user's machine, and it seems every new version causes my program to break when I try to rebuild it. I'll just say it, that creating small web apps in Python isn't worthwhile.

I also tried Javascript, but it has the inconvenience of lacking good math support, such as an exponentiation operator and complex arithmetic.

Then I learned that **sympy** can print formulas in Javascript syntax. This led me to try writing the online modeling program in Javascript, but with automatically generated math formulas. The following cells generate the formulas that my Javascript program needs, and automatically writes them to a file.

Formulas that are real-valued can just be translated directly. And I've arranged things so that there's only one complex-valued formula: For cone excursion.

[157]:
```
# Box compliance for sealed system. Note that this is always real valued
C_box
```

[157]:
$$\frac{V_{box}}{N_d S_d^2 c^2 \rho}$$

[158]:
```
# This is the effective compliance for a sealed system
C_eff_sealed = 1/(1/C_ms + 1/C_box)
displayEq('C_{eff, sealed}', C_eff_sealed)
```

$$C_{eff,sealed} = \frac{1}{\frac{N_d S_d^2 c^2 \rho}{V_{box}} + \frac{1}{C_{ms}}}$$

[159]:
```
# Likewise, box compliance for a ported system
C_ported
```

[159]:
$$\frac{V_{box}\left(\omega^2 - w_{port}^2\right)}{N_d S_d^2 c^2 \omega^2 \rho}$$

[160]:
```
# The effective compliance for a ported system
C_eff_ported = 1/(1/C_ms + 1/C_ported)
displayEq('C_{eff, ported}', C_eff_ported)
```

$$C_{eff,ported} = \frac{1}{\frac{N_d S_d^2 c^2 \omega^2 \rho}{V_{box}(\omega^2 - w_{port}^2)} + \frac{1}{C_{ms}}}$$

[161]:
```
# Relationship between port and cone displacement. Note that this is always real valued
port_cone_frac
```

[161]:
$$\frac{N_d S_d w_{port}^2}{S_{port}\left(\omega^2 - w_{port}^2\right)}$$

[162]:
```
# Conversion to sound pressure at distance r
sound_pressure_ratio = omega**2*rho*S_d/2/sp.pi/R_ref/P_ref
sound_pressure_ratio
```

[162]:
$$\frac{S_d \omega^2 \rho}{2\pi P_{ref} R_{ref}}$$

[163]:
```
# Excursion formula with voltage set to 1, so my Javascript program can apply two different input voltages.
x_driver_z = x_driver.subs({Z_e: R_e + 1j*omega*L_e, V: 1})
x_squared_real = sp.expand(x_driver_z*sp.conjugate(x_driver_z)).subs(C_ms,C_eff)
```

[164]:
```
from sympy.printing.jscode import jscode

outf = open('speakerjs/generated_code.js', 'w')
def printJs(label, expr, dprint = False):
    if dprint:
        print(label + ' = dprint("' + label + ' = ",' + jscode(expr) + ')',file = outf)
    else:
        print(label + ' = ' + jscode(expr), file = outf)
print('function generated_code_1() {', file = outf)
printJs('C_ms', params[0], dprint = True)
printJs('M_ms', params[1], dprint = True)
printJs('R_ms', params[2], dprint = True)
printJs('BL', params[3], dprint = True)
printJs('P_ref', P_ref.subs(em_params), dprint = True)
printJs('R_ref', R_ref.subs(em_params), dprint = True)
print('}', file = outf)
print('function generated_code_2(omega) {', file = outf)
# printJs('omega', 2*sp.pi*f)
printJs('C_eff_sealed', C_eff_sealed)
printJs('C_eff_ported', C_eff_ported)
print('''if (Fport == 0) {
      C_eff = C_eff_sealed
      }
      else {
```

```
      C_eff = C_eff_ported
      }
      V = VinExc''', file = outf)
printJs('x_squared_real', x_squared_real)
print('xabs = Math.sqrt(x_squared_real)', file = outf)
printJs('port_cone_frac', port_cone_frac)
printJs('sound_pressure_ratio', sound_pressure_ratio)
print('}', file = outf)
outf.close()
```

[165]:
```
# Code for turning this notebook into an import-able Python library
if __name__ == '__main__':
    print('Did you remember to save first?')
    !python3 -m jupyter nbconvert --to python ./speakerTheorySympy.ipynb
```

```
Did you remember to save first?
[NbConvertApp] Converting notebook ./speakerTheorySympy.ipynb to python
[NbConvertApp] Writing 33857 bytes to speakerTheorySympy.py
```

[166]:
```
# Code for converting this document to HTML
# Need to save document first
if __name__ == '__main__':
    print('Did you remember to save first?')
    !python3 -m jupyter nbconvert --to pdf ./speakerTheorySympy.ipynb
```

```
Did you remember to save first?
[NbConvertApp] Converting notebook ./speakerTheorySympy.ipynb to pdf
[NbConvertApp] Support files will be in speakerTheorySympy_files/
[NbConvertApp] Making directory ./speakerTheorySympy_files
[NbConvertApp] Writing 112820 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 393777 bytes to speakerTheorySympy.pdf
```