Loop through the designs, run `truss_designer()`, and print each report:

```
for design, design_params in design_parameters_dict.items():
  r_max, rep = truss_designer(load_maxima, r_maxima, design_params)
  rep = f"Design {design} report:\n\t" + rep.replace("\n", "\n\t")
  print(rep)
```

```
Design 1 report:
  Support A constraint satisfied: 1000 <= 1750.
  Support C constraint satisfied: 1000 <= 1167.
  Design feasible for supports.
  Max r for Tension constraint: 1.750.
  Max r for Compression constraint: 1.249.
  Overall max r = w/h = 1.24899959967968.
Design 2 report:
  Design infeasible due to Support A constraint: 2000 </= 1750.
Design 3 report:
  Support A constraint satisfied: 2000 <= 3250.
  Support C constraint satisfied: 2000 <= 2000.
  Design feasible for supports.
  Max r for Tension constraint: 0.8750.
  No feasible r for Compression constraint.
```

4.4 From Symbolics to Numerics

An engineering analysis typically requires that a symbolic solution be applied via the substitution of numbers into a symbolic expression. In section 4.2.7, we considered how to subsitute numerical values into expressions using SymPy's `evalf()` method. This is fine for a single value, but frequently an expression is to be evaluated at an array of numerical values. Looping through the array and applying `evalf()` is cumbersome and computationally slow. An easier and computationally efficient technique using the `sp.lambdify()` function is introduced in this section. The function `sp.lambdify()` creates an efficient, numerically evaluable function from a SymPy expression. The basic usage of the function is as follows:

```
x = sp.symbols("x", real=True)
expr = x**2 + 7
f = sp.lambdify(x, expr)
f(2)
```

```
↳ 11
```

By default, if NumPy is present, `sp.lambdify()` vectorizes the function such that the function can be provided with NumPy array arguments and return NumPy array values. However, it is best to avoid relying on the function's implicit behavior,

which can change when different modules are present, it is best to provide the numerical module explicitly, as follows:

```
f = sp.lambdify(x, expr, modules="numpy")
f(np.array([1, 2, 3.5]))
```

```
array([ 8.  , 11.  , 19.25])
```

Multiple arguments are supported, as in the following example:

```
x, y = sp.symbols("x, y", real=True)
expr = sp.cos(x) * sp.sin(y)
f = sp.lambdify([x, y], expr, modules="numpy")
f(3, 4)
```

```
0.7492287917633428
```

All the usual NumPy broadcasting rules will apply for the function. For instance,

```
X = np.array([[1], [2]])  # 2x1 matrix
Y = np.array([[1, 2, 3]])  # 1x3 matrix
f(X, Y)
```

```
array([[ 0.45464871,  0.4912955 ,  0.07624747],
       [-0.35017549, -0.37840125, -0.05872664]])
```

Example 4.2

You are designing the circuit shown in figure 4.4. Treat the source voltage V_S, the source resistance R_S, and the overall circuit topology as known constants. The circuit design requires the selection of resistances R_1, R_2, and R_3 such that the voltage across R_3, $v_{R_3} = V_{R_3}$, and the current through R_1, $i_{R_1} = I_{R_1}$, where V_{R_3} and I_{R_1} are known constants (i.e., design requirements). Proceed through the following steps:

1. Solve for all the resistor voltages v_{R_k} and currents i_{R_k} in terms of known constants and R_1, R_2, and R_3 using circuit laws
2. Apply the constraints $v_{R_3} = V_{R_3}$ and $i_{R_1} = I_{R_1}$ to obtain two equations relating R_1, R_2, and R_3
3. Solve for R_2 and R_3 as functions of R_1 and known constants
4. Create a design graph for the selection of R_1, R_2, and R_3 given the following design parameters: $V_S = 10$ V, $R_S = 50\ \Omega$, $V_{R_3} = 1$ V, and $I_{R_1} = 20$ mA.

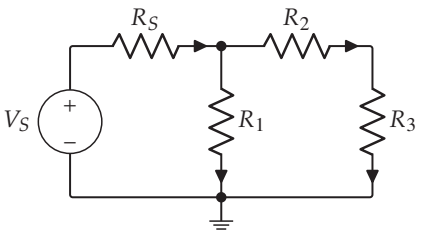

Figure 4.4. A resistor circuit design for example 4.2.

Solve for the Resistor Voltages and Currents Each resistor has an unknown voltage and current. We will develop and solve a system of equations using circuit laws. Begin by defining symbolic variables as follows:

```
v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3 = sp.symbols(
  "v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3", real=True
)
viR_vars = [v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3]
R1, R2, R3 = sp.symbols("R1, R2, R3", positive=True)
V_S, R_S, V_R3, I_R1 = sp.symbols("V_S, R_S, V_R3, I_R1", real=True)
```

There are 4 resistors, so there are $2 \cdot 4 = 8$ unknown voltages and currents; therefore, we need 8 independent equations. The first circuit law we apply is Ohm's law, which states that the ratio of voltage over current for a resistor is approximately constant. Applying this to each resistor, we obtain the following 4 equations:

```
Ohms_law = [
  v_RS - R_S*i_RS,  # == 0
  v_R1 - R1*i_R1,  # == 0
  v_R2 - R2*i_R2,  # == 0
  v_R3 - R3*i_R3,  # == 0
]
```

The second circuit law we apply is Kirchhoff's current law (KCL), which states that the sum of the current into a node must equal 0. Applying this to the upper-middle and upper-right nodes, we obtain the following 2 equations:

```
KCL = [
  i_RS - i_R1 - i_R2,  # == 0
  i_R2 - i_R3,  # == 0
]
```

The third circuit law we apply is Kirchhoff's voltage law (KVL), which states that the sum of the voltage around a closed loop must equal 0. Applying this to the left and right inner loops, we obtain the following 2 equations:

```
KVL = [
  V_S - v_R1 - v_RS,  # == 0
  v_R1 - v_R3 - v_R2, # == 0
]
```

Our collection of 8 equations are independent because none can be derived from another. They make a linear system of equations, which can be solved simultaneously as follows:

```
viR_sol = sp.solve(Ohms_law + KCL + KVL, viR_vars, dict=True)[0]
print(viR_sol)
```

$$i_{R1} = \frac{V_S\left(R_2 + R_3\right)}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$i_{R2} = \frac{R_1V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$i_{R3} = \frac{R_1V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$i_{RS} = \frac{V_S\left(R_1 + R_2 + R_3\right)}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$v_{R1} = \frac{R_1V_S\left(R_2 + R_3\right)}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$v_{R2} = \frac{R_1R_2V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$v_{R3} = \frac{R_1R_3V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$v_{RS} = \frac{R_SV_S\left(R_1 + R_2 + R_3\right)}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

Apply the Requirement Constraints The requirements that $v_{R_3} = V_{R_3}$ and $i_{R_1} = I_{R_1}$ can be encoded symbolically as two equations as follows:

```
constraints = {v_R3: V_R3, i_R1: I_R1}  # Design constraints
constraint_equations = [
  sp.Eq(v_R3.subs(constraints), v_R3.subs(viR_sol)),
  sp.Eq(i_R1.subs(constraints), i_R1.subs(viR_sol)),
]
print(constraint_equations)
```

$$V_{R3} = \frac{R_1R_3V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

$$I_{R1} = \frac{R_2V_S + R_3V_S}{R_1R_2 + R_1R_3 + R_1R_S + R_2R_S + R_3R_S}$$

Solve for Resistances The system of 2 constraint equations and 3 unkowns (R_1, R_2, and R_3) is underdetermined, which means there are infinite solutions.

The two equations can be solved for R_1 and R_2 in terms of R_3 and parameters as follows:

```
constraints_sol = sp.solve(
  constraint_equations, [R1, R2], dict=True
)[0]
print(constraints_sol)
```

$$R_1 = -R_S + \frac{V_S}{I_{R1}} - \frac{R_S V_{R3}}{I_{R1} R_3}$$

$$R_2 = \frac{-R_3 \left(I_{R1} R_S + V_{R3} - V_S\right) - R_S V_{R3}}{V_{R3}}$$

Create a Design Graph Applying the design parameters and defining numerically evaluable functions for R_1 and R_2 as functions of R_3,

```
design_params = {V_S: 10, R_S: 50, V_R3: 1, I_R1: 0.02}
R1_fun = sp.lambdify(
  [R3],
  R1.subs(constraints_sol).subs(design_params),
  modules="numpy",
)
R2_fun = sp.lambdify(
  [R3],
  R2.subs(constraints_sol).subs(design_params),
  modules="numpy",
)
```

And now we are ready to create the design graph, as follows:

```
R3_ = np.linspace(10, 100, 101)  # Values of R3
fig, ax = plt.subplots()
ax.plot(R3_, R1_fun(R3_), label="$R_1$ ($\\Omega$)")
ax.plot(R3_, R2_fun(R3_), label="$R_2$ ($\\Omega$)")
ax.set_xlabel("$R_3$ ($\\Omega$)")
ax.legend()
ax.grid()
plt.show()
```

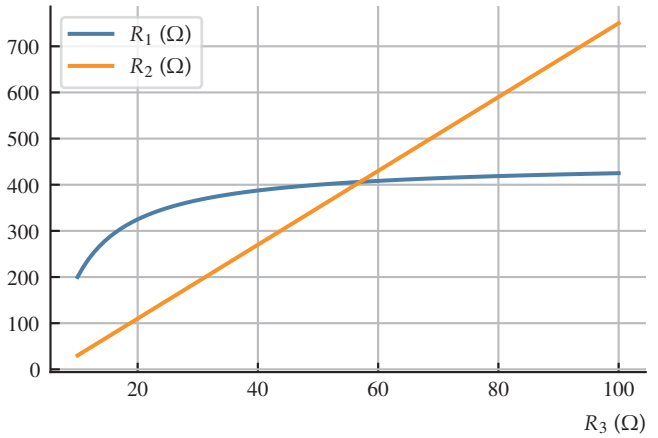

Figure 4.5. A design graph for resistors R_1, R_2, and R_3.

4.5 Vectors and Matrices

RD

Symbolic vectors and matrices can be constructed, manipulated, and operated on with SymPy. Basic vectors and matrices are represented with the mutable `sp.matrices.dense.MutableDenseMatrix` class and can be constructed with the `sp.Matrix` constructor, as follows:

```
u = sp.Matrix([[0], [1], [2]])  # 3×1 column vector
v = sp.Matrix([[3, 4, 5]])  # 1×3 row vector
A = sp.Matrix([[0, 1, 2], [3, 4, 5], [6, 7, 8]])  # 3×3 matrix
```

Without loss of generality, we can refer to vectors and matrices as matrices.

Symbolic variables can be elements of symbolic matrices; for instance, consider the following:

```
x1, x2, x3 = sp.symbols("x1, x2, x3")
x = sp.Matrix([[x1], [x2], [x3]])  # 3×1 vector
```

Symbolic matrix elements can be accessed with the same slicing notation as `lists` and NumPy arrays; for insance:

```
A[:,0]
A[0,:]
A[1,1:]
x[0:,0]
```