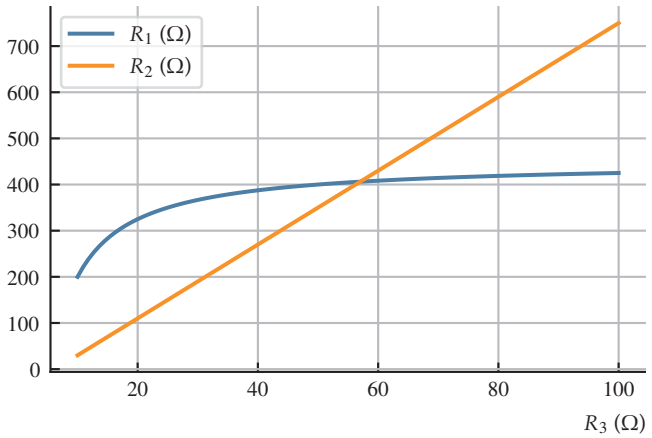

Figure 4.5. A design graph for resistors R_1, R_2, and R_3.

4.5 Vectors and Matrices

RD v1

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]
```

```
↳ ⎡0⎤
  ⎢3⎥
  ⎣6⎦
↳ ⎡0  1  2⎤
↳ ⎣4  5⎦
  ⎡x₁⎤
↳ ⎢x₂⎥
  ⎣x₃⎦
```

As with `list`s and contrary to arrays, these slices return a copy and not a view of the original matrix. Elements and slices can be overwritten with the same notation as `list`s and arrays, as follows:

```
A[0,0] = 7; A  # A is changed
A[:,1] = sp.Matrix([[8], [8], [8]]); A  # A is changed
```

```
↳ ⎡7  1  2⎤
  ⎢3  4  5⎥
  ⎣6  7  8⎦
↳ ⎡7  8  2⎤
  ⎢3  8  5⎥
  ⎣6  8  8⎦
```

Matrix row `i` or column `j` can be deleted with the `row_del(i)` or `col_del(j)` method. These methods operate in place. For instance,

```
A.row_del(2); A
A.col_del(1); A
```

```
↳ ⎡7  8  2⎤
  ⎣3  8  5⎦
↳ ⎡7  2⎤
  ⎣3  5⎦
```

Conversely, a row can be inserted at index `i` or a column can be inserted at index `j` with the method `row_insert(i, row)` or `col_insert(j, col)`. These methods do not operate in place. For instance,

```
A.row_insert(2, sp.Matrix([[9, 9]]))  # A is unchanged
A.col_insert(1, sp.Matrix([[9], [9]]))  # A is unchanged
```

```
↳ ⎡7  2⎤
  ⎢3  5⎥
  ⎣9  9⎦
```

```
↳ Matrix([[7, 9, 2], [3, 9, 5]])
```

Addition and subtraction works element-wise, in accordance with the matrix mathematics, as follows:

```
A = sp.Matrix([[0, 1], [2, 3]])  # 2×2 matrix
B = sp.Matrix([[4, 5], [6, 7]])  # 2×2 matrix
A + B
A - B
```

```
↳ Matrix([[4, 6], [8, 10]])
↳ Matrix([[-4, -4], [-4, -4]])
```

Matrix multiplication is in accordance with mathematical matrix multiplication (i.e., not element-wise), as follows:

```
A*B
B*A
```

```
↳ Matrix([[6, 7], [26, 31]])
↳ Matrix([[10, 19], [14, 27]])
```

The matrix inverse, if it exists, can be computed by raising the matrix to the power `-1`, as follows:

```
A**-1
B**-1
```

```
↳ Matrix([[-3/2, 1/2], [1, 0]])
↳ Matrix([[-7/2, 5/2], [3, -2]])
```

The matrix transpose can be accessed as an attribute `T`, which returns a transposed copy, as follows:

```
A.T
B.T
```

```
↳ Matrix([[0, 2], [1, 3]])
↳ Matrix([[4, 6], [5, 7]])
```

An n-by-n identity matrix can be constructed via the `eye(n)` function, as follows:

```
sp.eye(3)
```

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

An n-by-m matrix with all `0` compenents can be constructed via the `zeros(n, m)` function, as follows:

```
sp.zeros(2,4)
```

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

Similarly, an n-by-m matrix with all `1` compenents can be constructed via the `ones(n, m)` function, as follows:

```
sp.ones(2,8)
```

$$\begin{bmatrix}1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\end{bmatrix}$$

A diagonal or block-diagonal matrix can be constructed by providing the diagonal elements to the `diag()` function, as follows:

```
D = sp.diag(1, 2, 3); D
```

$$\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 3\end{bmatrix}$$

The determinant of a matrix can be computed via the `det()` method, as follows:

```
D.det()
```

6

The eigenvalues and eigenvectors of a matrix can be computed via the `eigenvects()` method, which returns a list of tuples, one for each eigenvalue, of the form `(eval, m, evec)`, where `eval` is the eigenvalue, `m` is the corresponding algebraic multiplicity of the eigenvalue, and `evec` is the corresponding eigenvector. For instance,

```
A.eigenvects()
```

```
[(3/2 - sqrt(17)/2,
  1,
  [Matrix([
   [-sqrt(17)/4 - 3/4],
   [                1]])]),
 (3/2 + sqrt(17)/2,
  1,
  [Matrix([
   [-3/4 + sqrt(17)/4],
   [                1]])])]
```

4.6 Calculus

Engineering analysis regularly includes calculus. Derivatives with respect to time and differential equations (i.e., equations including derivatives) are the key mathematical models of rigid-body mechanics (e.g., statics and dynamics), solid mechanics (e.g., mechanics of materials), fluid mechanics, heat transfer, and electromagnetism. Integration is necessary for solving differential equations and computing important quantities of interest. Limits and series expansions are frequently used to in the analytic process to simplify equations and to estimate unkown quantities. In other words, calculus is central to the enterprise of engineering analysis.

4.6.1 Derivatives

In SymPy, it is possible to compute the derivative of an expression using the `diff()` function and method, as follows:

```
x, y = sp.symbols("x, y", real=True)
expr = x**2 + x*y + y**2
expr.diff(x)  # Or sp.diff(expr, x)
expr.diff(y)  # Or sp.diff(expr, y)
```

$2x + y$

$x + 2y$

Higher-order derivatives can be computed by adding the corresponding integer, as in the following second derivative:

```
expr.diff(x, 2)  # Or sp.diff(expr, x, 2)
```

2

We can see that the partial derivative is applied to a multivariate expression. The differentiation can be mixed, as well, as in the following example:

```
expr = x * y**2/(x**2 + y**2)
expr.diff(x, 1, y, 2).simplify()  # ∂³/∂x∂y²
```