4.3 Solving Equations Algebraically

Virtually every engineering analysis requires the algebraic solution of an equation or a, more generally, a **system of equations** (i.e., a set of equations) to be solved simultaneously. For the engineer, this set of equations typically encodes a set of design constraints, design heuristics, and physical laws. In general, a system S of m equations in n unknown variables $x_0, \cdots, x_{n-1} \in \mathbb{C}$ and with m functions $f_0, \cdots, f_{m-1}$ can be represented as the set

$$S = \begin{cases} f_0(x_0, \cdots, x_n) = 0 \\ \vdots \\ f_m(x_0, \cdots, x_n) = 0 \end{cases}.$$

A **solution** for S is an n-tuple of values for x_i that satisfies every equation in S. There are three possible cases for a given system S of equations:

1. The system S has no solutions.
2. The system S has exactly one solution, said to be **unique**.
3. The system S has more than one solution (potentially infinitely many).

For some systems, a solution exists, but cannot be expressed in a closed-form or symbolic ("analytic") way. For such systems, a numerical solution is appropriate (see chapter 5). In some cases (e.g., n linear, independent equations and n unknown variables), a unique solution is guaranteed to exist.

There are two high-level SymPy function for solving equations algebraically, `sp.solve()` and `sp.solveset()`. The former is older, but remains the more useful for us; the latter has a simpler interface and is somewhat more mathematically rigorous, but it is often difficult to use its results programmatically. We will focus on `sp.solve()`. Neither function guarantees that it will find a solution, even if it exists, except in special cases.

Representing an equation in SymPy can be done explicitly or an expression can be treated as one side of an equation, with the other side implicitly 0. In other words, the following are equivalent ways of defining the equation $x^2 - y^2 = 2$:

```
x, y = sp.symbols("x, y")
x**2 - y**2 - 2  # == 0 Implicit equation
sp.Eq(x**2 - y**2, 2)  # Explicit equation
```

4.3.1 The `sp.solve()` Function

The `sp.solve()` function has the capability of solving a large class of systems of equations algebraically. The function has many optional arguments, but its basic usage is

```
sp.solve(f, *symbols, **flags)
```

Here is a basic interpretation of each argument:

- `f`: An equation or expression that is implicitly equal to zero or an iterable of equations or expressions.
- `symbols`: A symbol (e.g., variable) to solve for or an iterable of symbols.
- `flags`: Optional arguments, of which there are many. We recommend always using the `dict=True` option because it guarantees a consistent output: a list of dictionaries, one for each solution.

Consider the linear system of 3 equations and 3 unknown variables:

$$3x - 2y + 6z = -9 \tag{4.6a}$$

$$8y + 4z = -1 \tag{4.6b}$$

$$-x + 4y \quad = 0. \tag{4.6c}$$

The `sp.solve()` function can be deployed to solve this system as follows:

```
x, y, z = sp.symbols("x, y, z", complex=True)
S1 = [
  3*x - 2*y + 6*z + 9,  # == 0
  8*y + 4*z + 1, # == 0
  -x + 4*y, # == 0
]  # A system of 3 equations and 3 unknowns
sol = sp.solve(S1, [x, y, z], dict=True); sol
```

```
↳ [{x: 15, y: 15/4, z: -31/4}]
```

Now consider a simpler system of a single equation that includes a symbolic parameter a:

$$x^2 + 3x + a.$$

Applying `sp.solve()`,

```
a = sp.symbols("a", complex=True)
S2 = [x**2 + 2*x + a]  # A system of 1 equation and 1 unknown
sol = sp.solve(S2, [x], dict=True); sol
```

```
↳ [{x: -sqrt(1 - a) - 1}, {x: sqrt(1 - a) - 1}]
```

The quadratic formula has been applied, which yields two solutions, given in the `sol` list. Note that the solver was alerted to which symbolic variable was to be treated as an unknown variable (i.e., `x`) and which was to be treated as a known parameter (i.e., `a`) by the second argument `[x]` (i.e., `symbols`).

Suppose the solutions for x were to be substituted into an expression containing x. The `dict` object returned (here assigned to `sol`) can be used with the `subs()` method. For instance,

```
(x + 5).subs(sol[0])
(x + 5).subs(sol[1])
```

$$4 - \sqrt{1-a}$$

$$\sqrt{1-a} + 4$$

The `sp.solve()` function can solve for expressions and undefined functions, as well. Here we solve for an undefined function:

```
f = sp.Function("f")
eq = sp.Eq(f(x)**2 + 2*sp.diff(f(x), x), f(x))
sol = sp.solve(eq, f(x), dict=True)
```

$$f^2(x) + 2\frac{d}{dx}f(x) = f(x)$$

$$f(x) = \frac{1}{2} - \frac{\sqrt{1 - 8\frac{d}{dx}f(x)}}{2}$$

$$f(x) = \frac{\sqrt{1 - 8\frac{d}{dx}f(x)}}{2} + \frac{1}{2}$$

It can solve for the derivative term, too, as follows:

```
sol = sp.solve(eq, sp.diff(f(x), x), dict=True)
```

$$\frac{d}{dx}f(x) = \frac{(1 - f(x))\, f(x)}{2}$$

Example 4.1

You are designing the truss structure shown in figure 4.2. The external load of $\boldsymbol{f}_F = -f_F\hat{\boldsymbol{j}}$ (we use the standard unit vectors $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$, $\hat{\boldsymbol{k}}$), where $f_F > 0$, is given. As the designer, you are to make the w dimension as long as possible under the following constraints:

- Minimize the dimension h
- The tension in all members is no more than a given T
- The compression in all members is no more than a given C
- The magnitude of the support force at pin A is no more than a given P_A
- The magnitude of the support force at pin C is no more than a given P_C

Use a static analysis and the method of joints to develop a solution for the force in each member F_{AB}, F_{AC}, etc., and the reaction forces using the sign convention that tension is positive and compression is negative. Create a function that determines

design feasibility for a given set of design parameters $\{f_F, T, C, P_A, P_C\}$ and test the function.

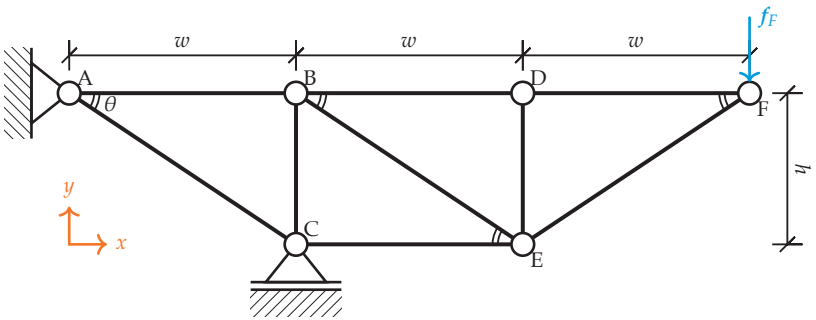

Figure 4.2. A truss with pinned joints, supported by a hinge and a floating support, with an applied force f_F.

Using the method of joints, we proceed through the joints, summing forces in the x- and y-directions. We will assume all members are in tension, and their sign will be positive if this is the case and negative, otherwise. Beginning with joint A, which includes two reaction forces R_{Ax} and R_{Ay} from the support,

$$\Sigma F_x = 0; \qquad R_{Ax} + F_{AB} + F_{AC}\cos\theta = 0 \tag{4.7}$$

$$\Sigma F_y = 0; \qquad R_{Ay} - F_{AC}\sin\theta = 0. \tag{4.8}$$

The angle θ is known in terms of the dimensions w and h as

$$\theta = \arctan\frac{h}{w}.$$

These equations can be encode symbolically as follows:

```
RAx, RAy, FAB, FAC, theta= sp.symbols(
  "RAx, RAy, FAB, FAC, theta", real=True
)
h, w = sp.symbols("h, w", positive=True)
eqAx = RAx + FAB + FAC*sp.cos(theta)
eqAy = RAy - FAC*sp.sin(theta)
theta_wh = sp.atan(h/w)
```

Proceeding to joint B,

$$\Sigma F_x = 0; \qquad -F_{AB} + F_{BD} + F_{BE}\cos\theta = 0 \tag{4.9}$$

$$\Sigma F_y = 0; \qquad -F_{BC} - F_{BE}\sin\theta = 0. \tag{4.10}$$

Encoding these equations,

```
FBD, FBE, FBC = sp.symbols("FBD, FBE, FBC", real=True)
eqBx = -FAB + FBD + FBE*sp.cos(theta)
eqBy = -FBC - FBE*sp.sin(theta)
```

For joint C, the floating support has a vertical reaction force R_C, so the analysis proceeds as follows:

$$\Sigma F_x = 0; \qquad -F_{AC}\cos\theta + F_{CE} = 0 \tag{4.11}$$

$$\Sigma F_y = 0; \qquad F_{AC}\sin\theta + F_{BC} + R_C = 0. \tag{4.12}$$

Encoding these equations,

```
FCE, RC = sp.symbols("FCE, RC", real=True)
eqCx = -FAC*sp.cos(theta) + FCE
eqCy = FAC*sp.sin(theta) + FBC + RC
```

For joint D, we can recognize that DE is a zero-force member:

$$\Sigma F_x = 0; \qquad -F_{BD} + F_{DF} = 0 \tag{4.13}$$

$$\Sigma F_y = 0; \qquad F_{DE} = 0. \tag{4.14}$$

Encoding these equations,

```
FDE, FDF = sp.symbols("FDE, FDF", real=True)
eqDx = -FBD + FDF
eqDy = FDE
```

Proceeding to joint E,

$$\Sigma F_x = 0; \qquad -F_{CE} - F_{BE}\cos\theta + F_{EF}\cos\theta = 0 \tag{4.15}$$

$$\Sigma F_y = 0; \qquad F_{BE}\sin\theta + F_{DE} + F_{EF}\sin\theta = 0. \tag{4.16}$$

Encoding these equations,

```
FEF = sp.symbols("FEF", real=True)
eqEx = -FCE - FBE*sp.cos(theta) + FEF*sp.cos(theta)
eqEy = FBE*sp.sin(theta) + FDE + FEF*sp.sin(theta)
```

Finally, consider joint F, with the externally applied force f_F,

$$\Sigma F_x = 0; \qquad -F_{DF} - F_{EF}\cos\theta = 0 \tag{4.17}$$

$$\Sigma F_y = 0; \qquad -f_F - F_{EF}\sin\theta = 0. \tag{4.18}$$

Encoding these equations,

```
fF = sp.symbols("fF", positive=True)
eqFx = -FDF - FEF*sp.cos(theta)
eqFy = -fF - FEF*sp.sin(theta)
```

In total, we have 12 force equations and 12 unknown forces (9 member forces and three reaction forces). Let's construct the system and solve it for the unkown forces, as follows:

```
S_forces = [
  eqAx, eqAy, eqBx, eqBy, eqCx, eqCy,
  eqDx, eqDy, eqEx, eqEy, eqFx, eqFy,
]  # 12 force equations
forces_unknown = [
  FAB, FAC, FBC, FBD, FBE, FCE, FDF, FDE, FEF,  # 9 member forces
  RAx, RAy, RC, # 3 reaction forces
]  # 12 unkown forces
sol_forces = sp.solve(S_forces, forces_unknown, dict=True); sol_forces
```

```
[{FAB: 2*fF*cos(theta)/sin(theta),
  FAC: -2*fF/sin(theta),
  FBC: -fF,
  FBD: fF*cos(theta)/sin(theta),
  FBE: fF/sin(theta),
  FCE: -2*fF*cos(theta)/sin(theta),
  FDE: 0,
  FDF: fF*cos(theta)/sin(theta),
  FEF: -fF/sin(theta),
  RAx: 0,
  RAy: -2*fF,
  RC: 3*fF}]
```

This solution is in terms of f_F, which is known, and θ. Because w and h are our design parameters, let's substitute `eqtheta` such that our solution is rewritten in terms of f_F, w, and h. Create a list of solutions as follows:

```
forces_wh = []  # Initialize
for force in forces_unknown:
  force_wh = force.subs(
    sol_forces[0]
  ).subs(
    theta, theta_wh
  ).simplify()
  forces_wh.append(force_wh)
  print(f"{force} = {force_wh}")
```

```
FAB = 2*fF*w/h
FAC = -2*fF*sqrt(h**2 + w**2)/h
FBC = -fF
FBD = fF*w/h
FBE = fF*sqrt(h**2 + w**2)/h
FCE = -2*fF*w/h
FDF = fF*w/h
FDE = 0
FEF = -fF*sqrt(h**2 + w**2)/h
RAx = 0
RAy = -2*fF
RC = 3*fF
```

This set of equations is excellent for design purposes. Because $f_F, w, h > 0$, the sign of each force indicates tension (+) or compression (−). For the forces with the factor w/h, clearly increasing w or decreasing h increases the force, proportionally. For the forces with the factor $\sqrt{h^2 + w^2}/h$, things are a bit more subtle. Introducing a new parameter $r = w/h$, we can rewrite these equations in a somewhat simpler manner, as follows:

```
r = sp.symbols("r", positive=True)
forces_r = []  # Initialize
force_r_subs = {}  # For substitutions
for i, force in enumerate(forces_wh):
  force_r = force.subs(w, h*r).simplify()
  forces_r.append(force_r)
  force_r_subs[forces_unknown[i]] = force_r
  print(f"{forces_unknown[i]} = {force_r}")
```

```
FAB = 2*fF*r
FAC = -2*fF*sqrt(r**2 + 1)
FBC = -fF
FBD = fF*r
FBE = fF*sqrt(r**2 + 1)
FCE = -2*fF*r
FDF = fF*r
FDE = 0
FEF = -fF*sqrt(r**2 + 1)
RAx = 0
RAy = -2*fF
RC = 3*fF
```

It is worthwhile investigating the term $\sqrt{r^2 + 1}$. Generate a graph over a reasonable range of $r = w/h$ and compare it to r and $2r$, as follows:

```
r_a = np.linspace(0, 5, 51)
fig, ax = plt.subplots()
ax.plot(r_a, np.sqrt(r_a**2 + 1), label="$\\sqrt{r^2+1}$")
ax.plot(r_a, r_a, label="$r$")
ax.plot(r_a, 2*r_a, label="$2 r$")
ax.set_xlabel("$r = w/h$")
ax.grid()
ax.legend()
```

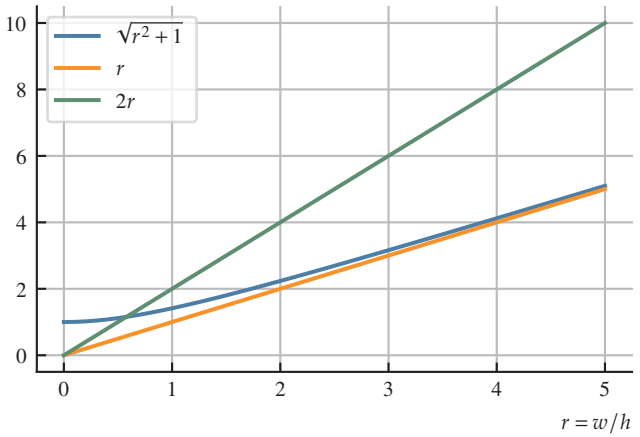

Figure 4.3. A graph of $\sqrt{r^2+1}$, where $r = w/h$.

So we see that $\sqrt{r^2+1} \to r$. That is, $r = w/h$ is the defining parameter and the design requirements are to maximize w and minimize h, which is tantamount to maximizing r. Under the reasonable assumption that $r > 1$, we can see the member with the most tension is AB, with its force $F_{AB} = 2r f_F$, and the member with the most compression is AC, with its force $F_{AC} = -2\sqrt{r^2+1} f_F$. From our design requirements, then,

$$F_{AB} = 2r f_F \le T \tag{4.19}$$

$$-F_{AC} = 2\sqrt{r^2+1} f_F \le C. \tag{4.20}$$

This leads to two constraints on r, call them r_T and r_C, both maxima, which can be solved for automatically as follows:

```
T, C = sp.symbols("T, C", positive=True)
eqrT = FAB.subs(force_r_subs) - T  # <= 0
eqrC = -FAC.subs(force_r_subs) - C  # <= 0
sol_rT = sp.solve(eqrT, r, dict=True)  # Solution for r_T
sol_rC = sp.solve(eqrC, r, dict=True)  # Solution for r_C
r_maxima = {
  "Tension": sol_rT[0],
  "Compression": sol_rC[0],
}
print(r_maxima)
```

```
{'Tension': {r: T/(2*fF)}, 'Compression': {r: sqrt(C**2 -
↪  4*fF**2)/(2*fF)}}
```

Another set of constraints apply to the supports. From the design requirements,

$$|\boldsymbol{R}_A| = \sqrt{R_{Ax}^2 + R_{Ay}^2} \leq P_A \tag{4.21}$$

$$|\boldsymbol{R}_C| = |R_C| \leq P_C \tag{4.22}$$

From our results above, the reaction forces don't depend on r (or w or h), so these constraints are merely to be checked to ensure that the design problem is feasible. Proceeding in a similar manner as above, we obtain two constraints on f_F, maxima f_A and f_C as follows:

```
PA, PC = sp.symbols("PA, PC", positive=True)
eqRA = sp.sqrt(RAx**2 + RAy**2).subs(force_r_subs) - PA  # <= 0
eqRC = sp.Abs(RC).subs(force_r_subs) - PC  # <= 0
sol_fFA = sp.solve(eqRA, fF, dict=True)  # Solution for f_A
sol_fFC = sp.solve(eqRC, fF, dict=True)  # Solution for f_C
load_maxima = {
  "Support A": sol_fFA[0],
  "Support C": sol_fFC[0],
}
print(load_maxima)
```

```
{'Support A': {fF: PA/2}, 'Support C': {fF: PC/3}}
```

Finally, we can create a function to perform the design, given a set of design parameters. First, define an auxilliary function to check the support constraints:

```
def check_supports(load_maxima, design_params, report=""):
  for kLM, vLM in load_maxima.items():
    fF_design = fF.evalf(subs=design_params)
    fF_max = fF.subs(vLM).evalf(subs=design_params)
    if fF_design > fF_max:
      return False, f"Design infeasible due to {kLM} constraint: " \
        f"{fF_design:.4g} </= {fF_max:.4g}."
    else:
      report += f"{kLM} constraint satisfied: " \
        f"{fF_design:.4g} <= {fF_max:.4g}.\n"
  report += "Design feasible for supports."
  return True, report
```

Now define an auxilliary function to maximize r:

```
def maximize_r(r_maxima, design_params, report=""):
  r_maxima_ = []  # Initialize numerical maxima
  for k_max, r_max in r_maxima.items():
    r_max_ = r.subs(r_max).evalf(subs=design_params)
    if np.abs(np.imag(complex(r_max_))) > 0.:  # Ensure real
      report += f"\nNo feasible r for {k_max} constraint."
      return False, None, report
    r_maxima_.append(r_max_)
    report += f"\nMax r for {k_max} constraint: {r_max_:.4g}."
  r_max = min(r_maxima_)  # Min of the maxima if the feasible max
  report += f"\nOverall max r = w/h = {r_max}."
  return True, r_max, report
```

Finally, define the function to design the truss:

```
def truss_designer(load_maxima, r_maxima, design_params):
  """Returns a dict of r=w/h ratio for the truss and a report"""
  satisfied, report = check_supports(load_maxima, design_params)
  if not satisfied:
    return satisfied, report
  satisfied, r_max, report = maximize_r(
    r_maxima, design_params, report
  )
  if not satisfied:
    return satisfied, report
  return r_max, report
```

Define the three sets of design parameters in a dictionary:

```
design_parameters_dict = {
  "1": {fF: 1000, T: 3500, C: 3200, PA: 3500, PC: 3500},
  "2": {fF: 2000, T: 4500, C: 6000, PA: 3500, PC: 3500},
  "3": {fF: 2000, T: 3500, C: 3200, PA: 6500, PC: 6000}
}  # Forces in N
```

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,