Mathematical Foundations of Systems Analysis and Optimization

Justin Gray

I. Introduction

Traditionally most practitioners would consider “systems analysis” and “multidisciplinary optimization” to be separate disciplines. However, at their respective mathematical cores, both disciplines are built on top of the same foundation: the manipulation of and techniques for solving nonlinear systems of implicit equations in order to use them for design and understanding of complex interactions.

The primary goal of this lesson is to provide you an introduction to this mathematical foundation, so that you can leverage the various tools and techniques more effectively. It is not critical that you are able to re-derive all of the equations presented here, nor is everything presented in complete detail. Rather, you should focus on understanding the broad concepts presented and how they relate to the type of work you’ll be asked to do as a systems analyst or optimization expert.

II. History of Optimization

Though formal optimization theory is quite old, what we think of today as "optimization" is a fairly young discipline that emerged with the development of general purpose computing.

1740: British mathematician Thomas Simpson described the modern form of a multivariate *Newton’s Method* in “Essays on Several Curious and Useful Subjects, in Speculative and Mix’d Mathematicks”, pointing out its usefulness for solving optimization problems.

1857: J. W. Gibbs shows that chemical equilibrium is attained when the energy is a minimum

1939: William Karush derives the necessary conditions for the inequality constrained problem in his Masters thesis. Harold Kuhn and Albert Tucker rediscover these conditions an publish their seminal paper in 1951. These became known as the Karush–Kuhn–Tucker (KKT) conditions.

1959: Davidon develops the first quasi-Newton method for solving nonlinear optimization problems. Fletcher and Powell publish further developments in 1963.

1963: Wilson invents the sequential quadratic programming method for the first time. Han reinvents it in 1975 and Powell does the same in 1977.

1975: John Holland proposed the first genetic algorithm.

1977: Raphael Haftka publishes one of the first multidisciplinary design optimization (MDO) applications, in a paper entitled “Optimization of flexible wing structures subject to strength and induced drag constraints”.

1984: Narendra Karmarkar starts the age of interior point methods by proposing a more efficient algorithm for solving linear problems. In a particular application in communications network optimization, the solution time was reduced from weeks to days, enabling faster business and policy decisions. Karmarkar’s algorithm stimulated the development of several other interior point methods, some of which are used in current codes for solving linear programs.

1985: The first conference in MDO, the Multidisciplinary Analysis and Optimization (MA&O) conference, takes place.

1995: Kennedy and Eberhart propose the particle swarm optimization algorithm

A. The Advent of Multidisciplinary Optimization

The field of study known as Multidisciplinary Optimization (MDO) — sometimes called Multidisciplinary Design Analysis and Optimization (MDAO) — grew out of the aerospace industry in the 1970's, where there was a growing realization that multiple different subsystems in the aircraft interacted strongly with eachother and that accounting for that interaction was necessary for good design.

The classic application (see Hafka's 1977 paper) was for the aerostructural design of aircraft wings. You can build a much lighter wing if you can account for its deflection in flight, rather than designing it to be rigid. MDO has grown into a broad research field, with extensive research that leverages both gradient-free and gradient-based optimization.

It is through the developments of gradient-based MDO that the deep connection between systems analysis and MDO becomes most readily apparent. Starting in the 1990's there were a series of increasing generalizations that broadened the applicability of gradient-based MDO:

1990: Sobieski developed the global sensitivity equations which enabled computation of analytic derivatives across arbitrary coupled models

1993: Evin Cramer published the Individual Design Feasible optimization architecture to solve MDO problems without direct coupling

2005: Joaquim Martins develops the "coupled adjoint" approach for optimizing coupled high-fidelity analyses (i.e. CFD + FEA)

2013: Joaquim Martins and John Hwang develop the Unified Derivative Equation, unifying all the various different techniques for computing derivatives

III. Why do we need optimization and what does "gradient-based" mean?

Optimization is a technique for finding solutions — the design variables — that maximize or minimize some quantity of interest — the objective function. Fundamentally, the need for optimization arises when you are dealing with under-defined problems; you have more degrees of freedom than equations to constrain them with.

Examples:

- Find the minimum value of a parabola
- Find the angle of attack and chord length needed to get a 5 Newtons of lift for a NACA 0012 airfoil

The objective function provides a means of distinguishing one solution from another, and thus by seeking the max or min of the objective you have fundamentally converted an under-defined problem into a well defined one.

All optimizers perform this task, though they do so in a variety of different ways. There are three main classes of optimization algorithms:

1. Brute Force

If your objective function is cheap enough and the number of design variables is small enough, then you can simply enumerate the entire design space and pick the best design out of the set of solutions you found. If your design variables are continuous, you can just pick a very fine discretization to sample with.

There are some very nice aspects of this approach. For one thing, you get a whole lot more information than just what the best design is, since you know the answer for all the designs. For another, this approach is dead simple and highly parallel. Perhaps most importantly, within your defined design space, this is the only method that can gurantee that you will find the overall best answer, or the global optimum.

If you can afford to use a brute-force technique, you should! As a rule of thumb, if you have 3 or fewer variables and a function that runs in under 0.01 seconds, then consider this approach.

2. Gradient-free
As the name suggests, these methods do not require any gradient information to perform optimization. Some methods, such as the simplex algorithm and the Constrained Optimization by Linear Approximation (COBYLA), are mathematically rigorous approaches to solving optimization problems. Other methods, such as the massive set of evolutionary algorithms (e.g. genetic algorithms and particle swarm algorithms), are heuristic in nature and do not necessarily rest on a rigorous mathematical footing.

Regardless of the mathematical underpinnings of these methods, they can be quite useful. Like the brute force method they are typically very easy to implement and very highly parallel, but they can typically work on problems with as many as 10-20 design variables even if the cost of that computing the objective function is much higher. There are some advanced gradient-free methods that can even be scaled up to 100 variables with reasonable performance. Another important characteristic of gradient-free methods is that they are suitable for use even when some design variables are discrete.

Despite the many good qualities of gradient-free methods, they tend to require a large number of function evaluations to get a reasonable answer. They also commonly include some stochastic qualities that make it so you will get a different answer each time you run them. This is why you will often see results from gradient-free methods presented as an average (or some other statistical quantity) taken over multiple runs.

Here are some good gradient-free methods that you can look into:

- COBYLA: Constrained Optimization by Linear Approximation
- EGO: Efficient Global Optimization
- NSGA-II: Non-dominated Sorting Genetic Algorithm II
- ALPSO: Augmented Lagrangian Particle Swarm Optimization

3. Gradient-based
Gradient-based optimization methods use derivative information to formally transform an under-defined problem into a well-defined one. Recent research has shown that for problems with over 100 design variables, gradient-based methods are 2-4 orders of magnitude faster than gradient-free methods. The improved computational performance is the primary motivation for using gradient-based optimization algorithms.

Of course, there is no free lunch! The downside to using gradient-based optimization algorithm is that it can be quite difficult to compute the gradients that you need. All of the promised performance of gradient-based methods falls apart if you can not compute accurate derivative information in a computationally efficient manner. It is also important to note that, since the definition of a derivative fundamentally assumes a continuous function and a continuous design space, these methods can only work on continuous problems.

Another potential limitation of gradient-based methods is that they can only find "local optima", and hence are less effective at searching bumpy design spaces. In other words, gradient-based methods are very efficient at climbing whatever hill the initial guess has started them on, but they will not search any other hills even if they might have taller peaks.

Despite the challenges associated with computing derivatives, the limitation to continuous design spaces, and the local nature of the algorithms, when properly implemented the performance of gradient-based methods is so great that it justifies their use even for relatively small optimization problems. Gradient-based methods are the only class of optimization algorithms that can handle 1000's, 10,000's, or even 1,000,000's of design variables.

IV. Introduction to Implicit Systems of Equations

The mathematical foundations of both systems analysis and MDO are built on the ability to solve nonlinear, implicit systems of equations. Implicit systems of equations are a collection of inter-related **implicit functions** that must be solved simultaneously. These implicit functions are defined by residual equations, such as

$$r = \mathcal{R}(x, y) = 0 \ . \tag{1}$$

Here x is the vector the input variables and y is the vector of implicit outputs of the function. The length of y will always be the same as the length of the the residual function R. For any given values of x, you must solve for the y values that drive the residual to 0. This is what makes the function “implicit”, because there is no general analytic expression to solve for y. Instead you implicitly find y wherever $R = 0$.

Although you’ve almost certainly run into implicit functions before, you’re likely much more familiar with **explicit functions**. Consider this example,

$$y = 3x^2 + 2x \,, \tag{2}$$

where you have a simple expression to directly compute y given any value of x. Explicit functions are much easier to work with, but unfortunately not every function can be represented explicitly. Both types of functions have inputs and outputs, but the manner in which they compute the outputs is vastly different. As we’ll see the implicit form is more generally useful in for systems analysis and optimization. Importantly, you can convert any explicit function into an implicit form by simply collecting all terms on one side of the equation:

$$R(x, y) = 3x^2 + 2x - y = 0 \,. \tag{3}$$

Consider the example of the Mach-Area relationship for compressible, isentropic flow:

$$\frac{A}{A^*} = \left(\frac{\gamma + 1}{2}\right)^{-\frac{\gamma+1}{2(\gamma-1)}} \frac{1}{MN} \left(1 + \frac{\gamma - 1}{2} MN^2\right)^{\frac{\gamma+1}{2(\gamma-1)}} \,, \tag{4}$$

where A is the flow area, A^* critical flow area at Mach=1, MN is the Mach number of the flow, and γ is the ratio of specific heats that we’ll assume to be a constant ($\gamma = 1.4$ is a very good assumption except at really high temperatures).

Equation (4) is an explicit function that computes the area ratio A/A^* directly, given any value for MN. You can easily plot the relationship between the two — check out the example code given in Figure 1. However, if we flip the question around and ask "For a given area ratio, what is the Mach number?" we will find that there is no longer an explicit relationship because there is no way to isolate MN by itself.

```
import numpy as np
import matplotlib.pylab as plt

gamma = 1.4

gp1 = gamma+1
gm1 = gamma-1

def area_ratio(MN):
    return (gp1/2)**-(gp1/(2*gm1)) * (1 + gm1/2*MN**2)**(gp1/(2*gm1)) / MN


if __name__ == "__main__":
    Machs = np.linspace(0.1, 3, 50)
    area_ratios = area_ratio(Machs)

    fig, ax = plt.subplots()
    ax.plot(Machs, area_ratios)
    ax.set_xlabel("Mach")
    ax.set_ylabel(r"$\frac{A}{A^*}$", rotation="horizontal", fontsize=15)
    fig.savefig("mach_area.pdf")

    plt.show()
```

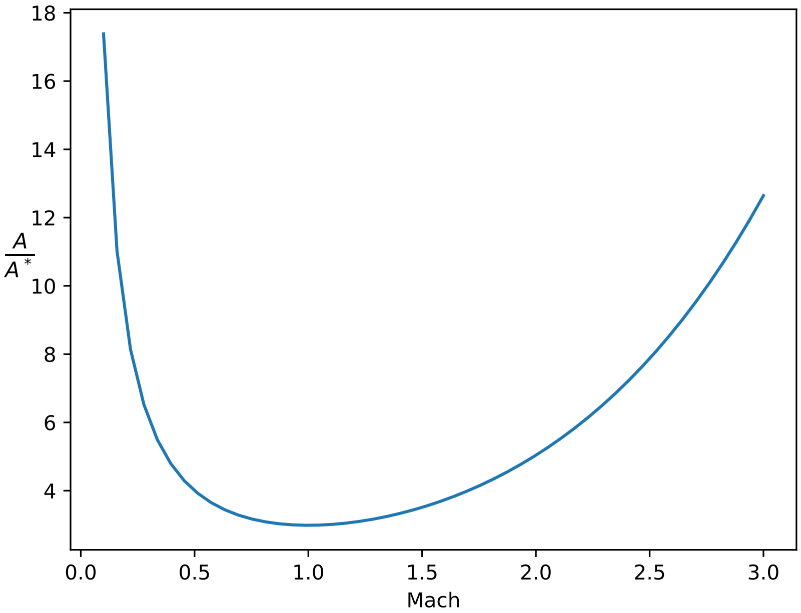

Fig. 1 Python code plotting the Mach-area relationship

So, if you wanted to solve for the Mach number that matches a given area ratio, you need to use a nonlinear solver. In order to do that, you need to define a residual function that must be driven to 0:

$$R(\frac{A}{A^*}, MN) = \frac{A}{A^*} - \left[\left(\frac{\gamma+1}{2}\right)^{-\frac{\gamma+1}{2(\gamma-1)}} \frac{1}{MN} \left(1 + \frac{\gamma-1}{2} MN^2\right)^{\frac{\gamma+1}{2(\gamma-1)}} \right] = 0 \tag{5}$$

Equation (9) is literally just a re-arranging of Equation (4) by moving the right hand side over to the left. So they are exactly the same, but the implicit form is more useful since it can be solved for either area ratio or Mach number, while the explicit form can only be solved for area ratio. Figure 2 give you a short bit of python code that can apply Newton's method to solve the implicit form of this equation. Notice that the answer you get from Newton's method is dependent on the initial guess you give to the

```
from scipy.optimize import newton

from mach_area import area_ratio

def residual(MN, ar_target):
    return ar_target - area_ratio(MN)

# solve for the MN that gives an area ratio of 3, with an initial guess of MN=0.1
solution = newton(residual, 0.1, args=(3.,), maxiter=150)
print('subsonic␣solution:␣', solution)

# solve for the MN that gives an area ratio of 3, with an initial guess of MN=2.0

solution = newton(residual, 2.0, args=(3.,), maxiter=150)
print('supersonic␣solution:␣', solution)
```

Fig. 2 Python code solving the implicit Mach-area equation using newton's method

The trick of re-arranging terms to form an implicit version from explicit function works for any general function. Hence we can generally say that all explicit functions have an implicit form, but not all implicit functions have an explicit form. We're discussing this subtlety because internally and out of sight from the users, OpenMDAO represents all functions in their implicit form. As a user you don't need to worry about this transformation — it is managed automatically for you — but the fact that it occurs becomes relevant when trying to understand how OpenMDAO computes derivatives across complex models. So we'll revisit this subject a bit later.

Disguising an implicit function as an explicit one

Of more direct relevance to you as a user is that you can disguise an implicit function in an explicit form. To be perfectly clear, this is **not** the same thing as converting an implicit function into an equivalent explicit form!

The word "disguise" is used very intentionally here. What you can do is put a black-box wrapper around an implicit function to make it look like an explicit one, even though the internal calculation is still being performed in an implicit way.

Lets go back to our example to see this in action, solving for Mach number when given an area ratio. The implicit form is given in Equation 9. We're **not** going to change that at all. However, if look at the code in Figure 9 you see that we can get solutions with a single call to the 'newton' method. Now, with a few more lines of code, we can wrap that whole process up into something that looks like a regular explicit function, as shown in Figure 3.

```python
from scipy.optimize import newton

from mach_area import area_ratio

def residual(MN, ar_target):
    return ar_target - area_ratio(MN)

# return the supersonic solution
def super_mn_from_ar(ar_target):
    return newton(residual, 2.0, args=(ar_target,), maxiter=150)

# return the subsonic solution
def sub_mn_from_ar(ar_target):
    return newton(residual, 0.01, args=(ar_target,), maxiter=150)

if __name__ == "__main__":
    print('supersonic:␣', super_mn_from_ar(3.))
    print('subsonic:␣', sub_mn_from_ar(3.))
```

Fig. 3 Disguising an implicit function with an explicit wrapper

Even though the functions look explicit now, they aren't really. Why does this matter? For one thing, if the function was really explicit then you would expect to be able to give it any allowable value for area ratio — anything over 1.0 — and expect to get an answer back. However, if you experiment a bit with that function you'll see that the nonlinear solver isn't that stable. It gets an answer for $A/A^* = 3.0$, but not for $A/A^* = 2.5$. A second reason, which is especially important in the context of optimization, is that the way in which you need to compute derivatives across the function changes because of the implicitness. We'll talk about this in more detail in the next sections.

Using Newton's method to solve implicit functions

Given a general implicit vector valued function, $R(x, y)$, we want to solve for the implicit output variables y such that $R(x, y) = 0$.

Lets look at taylor expansion of the vector valued residual equation for small variations in y:

$$R(x, y + \Delta y) = R(x, y) + \left[\frac{\partial R}{\partial y}\right]_x \Delta y + O(\Delta y^2) \tag{6}$$

We seek solutions of the form $R(x, y) = 0$, so finding Δy such that $R(x, y + \Delta y) = 0$ will also suffice.

Disregard the second order terms in Equation (6), Then, by rearranging the remaining terms we get

$$\left[\frac{\partial R}{\partial y}\right]_x \Delta y = -R(x, y) + O(\Delta y^2)\ . \tag{7}$$

. This nonlinear equation is not any more useful than the original Taylor expansion. Yes, if we solved it we would know the exact Δy that would drive $R(x, y) = 0$, but we still have no idea how to do that! However, by dropping the second order terms we get a simple linear system of equations that we can solve:

$$\begin{aligned} \left[\frac{\partial R}{\partial y}\right] \Delta y &= -R(x, y) \\ \Delta y &= -\left[\frac{\partial R}{\partial y}\right]^{-1} R(x, y)\ . \end{aligned} \tag{8}$$

Equation (8) is called "Newton's method". Because we dropped the higher order terms, this solution to this linear system is only an approximation to the roots of $R(x, y)$. So, we can iteratively solve Equation (8) and apply the Δy update until you reach a sufficient level of accuracy. Also, since the terms we disregarded are of $O(\Delta y^2)$, we know that Newton's method should exhibit second order convergence when near the true root.

For the implicit Mach-area relationship:

$$R(\frac{A}{A^*}, MN) = \frac{A}{A^*} - \left[\left(\frac{\gamma+1}{2}\right)^{-\frac{\gamma+1}{2(\gamma-1)}} \frac{1}{MN} \left(1 + \frac{\gamma-1}{2} MN^2\right)^{\frac{\gamma+1}{2(\gamma-1)}} \right] = 0 \tag{9}$$

Newton's method would look as follows:

$$\frac{\partial R}{\partial MN} \Delta MN = -R(\frac{A}{A^*}, MN) \ . \tag{10}$$

Note that since the Mach-area residual has only the scalar MN variable as an implicit output, so Equation (10) is also a scalar equation in this case. We can compute the necessary derivative analytically:

$$\frac{\partial R}{\partial MN} = \left(\frac{\gamma+1}{2}\right)^{-\frac{\gamma+1}{2(\gamma-1)}} \left[\frac{1}{MN^2} \left(1 + \frac{\gamma-1}{2} MN^2\right)^{\frac{\gamma+1}{2(\gamma-1)}} - \frac{\gamma+1}{2(\gamma-1)} \left(1 + \frac{\gamma-1}{2} MN^2\right)^{\frac{\gamma+1}{2(\gamma-1)}-1} (\gamma-1) \right] \tag{11}$$

That is a fairly large equation, which highlights the reality that hand computing derivatives can get end up being a lot of work for large systems of arbitrary equations. So its very common to approximate the partial derivative Jacobian using finite-differences. We'll talk more about the various ways you can get the derivatives you need in further sections, but the key here is to remember how Newton's method work. You can find the roots for any general implicit system of nonlinear equations,by iteratively applying Equation (8).

V. Introduction to the Theory Behind Gradient-Based Optimization

A. Unconstrained Optimization

Consider a simple parabolic function of one variable: less

$$f = F(x_1) = (x_1 - 3)^2 + x_1 - 3 \ . \tag{12}$$

If I asked you to "find the value of f", then you have an under-defined problem. There is an infinite number of solutions for f, corresponding to the infinite number of values that x_1 can take. If instead, I ask you to "find the minimum of value f", you now have a well-defined problem. The formal optimization problem statement for this would be:

min.	f
w.r.t	x

Since this function is parabolic, we know that there is one and only one minimum for all values of x_1. From basic calculus, we know that we can find the minimum of f by taking the derivative and setting the resulting expression equal to 0.

$$\frac{\partial f}{\partial x_1} = 2(x_1 - 3) + 1 = 0 \ . \tag{13}$$

Now lets look at a parabolic function of two variables:

$$f = F(x_1, x_2) = (x_1 - 3)^2 + x_1 x_2 + (x_2 + 4.0)^2 - 3 \ . \tag{14}$$

Since it is still parabolic we know that there is still only one minimum. f is still a scalar, but now the input is 2 scalars: x_1 and x_2 To find the minimum, we take the derivative of f with respect to the two x variables and set the result equal to 0. Because there are two inputs, we get a system of two equations instead of a single equation.

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= 2(x_1 - 3) + x_2 = 0 \\ \frac{\partial f}{\partial x_2} &= x_1 + 2(x_2 + 4.0) = 0 \end{aligned} \tag{15}$$

Now we can solve this system of equations for x_1 and x_2 to find the minimum of the original objective function. This is referred to as an **implicit function**, which would generally be represented as:

$$r = \mathcal{R}(x_1, x_2) = 0 \ . \tag{16}$$

It is implicit because you do not explicitly compute the values x_1 and x_2, but rather we solve the residual equation and when that solution is found we also know the desired values. Conversely, an **explicit function** is one where you can directly compute your objective through $F(x_1, x_2)$, for example. You can then isolate x_1 or x_2 to directly solve for them as well. You can represent any explicit function in an implicit form like that above, but the reverse is not necessarily true. The way you compute derivatives of implicit functions for the gradient is very different from how you would do so for explicit functions; we'll revisit this later when discussing gradient calculations.

Equation (15) can be re-arranged into a linear system of equations:

$$\begin{bmatrix} 2, 1 \\ 1, 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 6 \\ -8 \end{bmatrix} \ . \tag{17}$$

We can now solve this linear system by inverting the matrix on the left hand side, to find the values of x_1 and x_2 that will minimize the original paraboloid objective function. For this specific case, with a parabolic objective function, it turns out the there is also an explicit form for the nonlinear system of equations needed to find the minimum.

What if the objective function is cubic though?

$$f = F(x_1, x_2) = (x_1 - 3)^2 + x_1^2 x_2 + (x_2 + 4.0)^2 - 3 \ . \tag{18}$$

We made it cubic by modifying the cross term ($x_1^2 x_2$), and so now the system of equations to find the minimum will be:

$$\begin{aligned} \frac{\partial f}{\partial x_1} &= 2(x_1 - 3) + 2x_1 x_2 = 0 \\ \frac{\partial f}{\partial x_2} &= x_1^2 + 2(x_2 + 4.0) = 0 \ . \end{aligned} \tag{19}$$

Since this system of equations is nonlinear, we can no longer represent it in a simple matrix form and solve it with a simple matrix inverse. This has two important consequences. First, since this system of equations is now second order we know that there will be two roots which means there are two minimum values. Also, and perhaps more importantly, we now need to employ a nonlinear solution technique to find those roots. For a simple problem like this, perhaps you can find the roots by hand using substitution.

What if your objective is a cubic function of 1000 variables?

$$f = F(x_1, x_2, ..., x_{1000}) = F(\bar{x}) \ . \tag{20}$$

Now, to find the minima we will have to solve a nonlinear system of 1000 equations:

$$r = \mathcal{R}(x_1, x_2, ..., x_{1000}) = \mathcal{R}(\bar{x}) = \frac{\partial f}{\partial \bar{x}} = 0 \ . \tag{21}$$

This is far too large to solve by hand, but the implicit form of the equations can be solved using a numerical technique such as Newton's method. Newton's method is composed of the iterative solution to the following equation:

$$\Delta \bar{x} = -\left[\frac{\partial \mathcal{R}}{\partial \bar{x}}\right]^{-1} \bar{r} \ , \tag{22}$$

where $\left[\frac{\partial \mathcal{R}}{\partial x}\right]$ is a matrix of partial derivatives computed about some $\bar{x}^*$, $[\Delta x]$, $\bar{r}$ is the vector residual values computed at $\bar{x}^*$, and $\Delta \bar{x}$ is the update that should be applied to $\bar{x}^*$ for the next iteration.

Note that we now need $\frac{\partial \mathcal{R}}{\partial \bar{x}}$, but $\mathcal{R}$ was itself defined as a partial derivative of f with respect to $\bar{x}$. So that means that we now need second-derivatives with respect to f:

$$\left[\frac{\partial \mathcal{R}}{\partial \bar{x}}\right] = \left[\frac{\partial^2 f}{\partial \bar{x}^2}\right] \ . \tag{23}$$

$\frac{\partial^2 f}{\partial \bar{x}^2}$ is called the Hessian matrix of f. This is how, for an unconstrained optimization, we can create a well-defined problem (same number of unknowns as equations to constrain them) by taking derivatives of the objective function with respect to the design variables.

If the system of equations we get by taking those derivatives is nonlinear, then we need to use Newton's method to iteratively solve for roots that will minimize the objective. If $\bar{x}$ is a length 1000 vector, The Hessian of f will be 1000×1000, so there are one million terms in the Hessian. We would need to derive a formula for each of those terms in order to implement Newton's method and solve the optimization problem.

Obviously, deriving one million equations isn't feasible. For smaller problems, you could maybe compute the Hessian by hand — in fact, this is what J.W. Gibbs did to solve for chemical equilibrium in 1857. However, for general problems, there is a suite of nonlinear optimization methods (e.g. sequential quadratic programming and interior point methods) that have been developed so that we either don't need the Hessian at all or don't have to explicitly compute it. Some optimization algorithms, called quasi-Newton methods, approximate the Hessian using only first derivative information.

So, while you most likely do not need to explicitly compute the Hessian, you still need to compute the first-derivatives of the objective function with respect to the design variables. So for 1000 variable unconstrained optimization, we would still need to derive 1000 formulas. This is certainly much more achievable, but clearly still a lot of work.

B. Constrained Optimizations

Practically speaking, optimization problems are very rarely unconstrained. Usually, you want to maximize or minimize an objective function, subject to some set of constraints. While an unconstrained optimization problem is under-defined (more unknowns than equations to define them), a constrained optimization problem can actually be under-defined, well-defined, or even over-defined (fewer variables than equations to define them). It all depends on how many constraints you have in your problem, compared to how many design variables. You can have equality constraints ($g(x) = 0$), or inequality constraints ($h(x) < 0$). Both are broadly handled the same way, though there are some slight differences.

1. Equality constraints: $h = H(x) = 0$

First we will modify the optimization problem statement to include the constraints.

min.	f
w.r.t	x
subject to	$h = 0$

Our previous technique for minimizing a function will not work here, because the derivatives of the objective function do not take the constraints into account at all. But we can modify the problem a bit in order to take both the objective and the constraints into account by forming what is called the Lagrangian:

$$\mathcal{L}(x, \lambda) = F(x) - \lambda H(x) \ . \tag{24}$$

The Lagrangian condenses the constrained optimization problem, which was composed of multiple separate equations, back into a single scalar equation which we can apply our existing derivative based technique to! In order to do this, we've introduced a new set of unknown variables, λ, to the problem — one for each equality constraint we want to consider. The set of λ variables are called the Lagrange multipliers, and they provide the necessary additional degrees of freedom to keep the problem well-posed (we'll see why they are needed in a bit). You can take derivatives of the Lagrangian function, with respect to all of the unknown variables (x and λ), and set them equal to 0 for form a system of equations that can be solved to find minima.

The Lagrange multipliers are additional unknown variables who's values must be solved for, but from an engineering perspective they are not new design variables in the problem. So you might be tempted to think they are merely mathematical tricks to maintain the well-posedness of the problem. However, they do have some physical meaning. The Lagrange multipliers represent the sensitivity of the optimum objective value to the constraint. So a very large Lagrange multiplier indicates that the solution is very sensitive to the constraint, and vice versa.

Lets differentiate the Lagrangian with respect to all the unknown variables (λ and x) and set the equations equal to 0:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \lambda} &= H(x) &&= 0 \quad (25) \\ \frac{\partial \mathcal{L}}{\partial x} &= \frac{dF}{dx} - \lambda \frac{dH}{dx} &&= 0 \quad (26) \end{aligned}$$

Because of Equation (25), solutions to this system of equations which clearly have to satisfy the equality constraints. Equation (26) is what actually minimizes the Lagrangian. If we have N_h constraints, then Equation (25) is actually a set of N_h equations. If we have N_x design variables, then Equation (26) is a set of N_x equations. In total, this gives us $N_h + N_x$ equations, for which we need an equal number of degrees of freedom. This is why the Lagrange multipliers are necessary to be able to solve constrained optimization problems. Without them, the system of equations you get from differentiating the Lagrangian would be over-defined.

To solve

2. Inequality constraints: $g = G(x) \geq 0$
Again, lets modify the optimization problem to include the inequality constraint.

min.	f
w.r.t	x
subject to	$g \geq 0$

Inequality constraints are tricky because they may or may not actually constrain the final solution. If your final answer is constrained, inequality constraints actually behave just like an equality constraints. However, if the overall optimum exited at a point where, for example, g=1000, then the inequality constraint actually had no impact on the solution at all!

We can introduce a vector of slack variables, ξ, to the inequality constraints to convert them into equality constraints:

min.	f
w.r.t	x, ξ
subject to	$g + \xi = 0$
	$\xi < 0$

For this modification to be valid, we need to ensure that ξ always takes on whatever value is necessary to enforce the constraint if it is active, but if the constraint is inactive then ξ should have no impact on the solution. It is not obvious that things will work out this way from the given formulation, but luckily they do once we form the Lagrangian.

$$\begin{aligned} \mathcal{L}(x, lambda, \xi) &= F(x) - \lambda(G(x) + \xi) \\ &= F(x) - \lambda G(x) - \lambda \xi \quad (27) \end{aligned}$$

Similar to before, we can take derivatives of the Lagrangian with respect to x and $lambda$, which will yield $N_x + N_g$ equations. However, now we run into a problem because we have introduced an additional set of N_g slack variables to the problem and so we still have an under-defined set of equations. In other words, we don't have any equations to constrain the values for the slack variables, so we can not yet solve for solutions that minimize the Lagrangian.

You might be tempted to simply differentiate with respect to ξ, since thats what we have done in all the previous cases. But, a quick look shows that will not give us a general equation to constrain the value of ξ:

$$\frac{\partial \mathcal{L}}{\partial \xi} = \lambda \quad (28)$$

The only way for Equation (28) to equal zero is if $\lambda = 0$ all the time, which is not what we need. Rather than adding an additional equation to constrain ξ, this additional equation would actually remove the necessary additional degree of freedom that λ introduced in the first place. From a mathematical perspective, avoiding this equation means that we are explicitly not going to be minimizing the Lagrangian with respect to ξ. Rather, we must view ξ as an additional degree of freedom needed to balance things out, but not one that may take any value.

Take another look at the $-\lambda\xi$ term in Equation (27). Since we want to minimize $\mathcal{L}$ with respect to λ this would drive λ toward ∞. This is obviously a degenerate solution that we want to avoid, so we can add a restriction that if λ is non-zero, then ξ must be 0. Interestingly, $\xi = 0$ is the condition for which the inequality constraint becomes active and hence should behave like an equality constraint. What about the situation where the inequality constraint is inactive? In that case, $\xi \neq 0$, so in order to keep the $-\lambda\xi$ from blowing up the Lagrangian, we must force $\lambda = 0$. Basically, we want to ensure that for all cases either $\xi = 0$ or $\lambda = 0$, but never both.

We can achieve that by the following equation:

$$\lambda\xi = 0\,, \tag{29}$$

which is referred to as the complementary slackness condition. The complementary slackness condition gives us an additional N_g equations, which means we now have a well-defined problem formulation that can be solved to find minima:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \lambda} &= G(x) + \xi = 0 \\ \frac{\partial \mathcal{L}}{\partial x} &= \frac{dF}{dx} - \lambda\frac{dG}{dx} = 0 \\ \lambda\xi &= 0 \end{aligned} \tag{30}$$

C. Scaling for Optimization Problems

Equation (30) gives us a general set of equations that, when solved, will provide the local optimum for any general continuous problem. The tick then becomes to solve these equations, and while there are many different techniques to do that, the equations themselves can help us understand how to make that as easy as possible for those various techniques. This brings us to the topic of scaling for optimization problems. The goal of scaling is to make it easier to solve Equation (30), by making sure that all the unknown variables have roughly the same influence on the solution space.

We see the term $G(x)$ showing up, along with λ, ξ, dF/dx, and dG/dx. So as a rough rule of thumb we can say that its our goal to make sure that all of these terms are within 2-3 orders of magnitude of each other. One way to do that is by scaling the design variables (x) and/or the objective (F) and constraint (G) functions. So lets define some scaled versions of these functions, where the $\hat{()}$ indicates a scaled quantity. We'll use a reference value for each quantity that provides the physical value that should be normalized to 1.

$$\hat{x} = x \cdot \text{ref}_x^{-1} \tag{31}$$

$$\hat{F}(\hat{x}) = \frac{F(x \cdot \text{ref}_x^{-1})}{\text{ref}_F} \tag{32}$$

$$\hat{G}(\hat{x}) = \frac{G(x \cdot \text{ref}_x^{-1})}{\text{ref}_G} \tag{33}$$

Equations (31), (32), (33) are now the ones that we'll show to our optimizer. So the nonlinear set of equations that the optimize seeks to solve is now:

$$\begin{aligned} \hat{G}(\hat{x}) + \xi &= 0 \\ \frac{d\hat{F}}{d\hat{x}} - \lambda\frac{d\hat{G}}{d\hat{x}} &= 0 \\ \lambda\xi &= 0 \end{aligned} \tag{34}$$

So how do our three reference values ($\text{ref}_x, \text{ref}_F, \text{ref}_G$) affect these equations? For $\hat{G}(\hat{x})$, the answer is quite straightforward; Equation (33) tells us that ref_G directly affects the magnitude of the scaled constraint equations. What about for the

$d\hat{F}/d\hat{x}$ term?

$$\begin{aligned}
\frac{d\hat{F}}{d\hat{x}} &= \frac{dF}{dx}\frac{dx}{d\hat{x}}\frac{d\hat{F}}{dF} \\
\frac{d\hat{F}}{d\hat{x}} &= \frac{dF}{dx}(\mathrm{ref}_x)\left(\frac{1}{\mathrm{ref}_F}\right) \;.
\end{aligned} \tag{35}$$

Similarly for $d\hat{G}/d\hat{x}$,

$$\begin{aligned}
\frac{d\hat{G}}{d\hat{x}} &= \frac{dG}{dx}\frac{dx}{d\hat{x}}\frac{d\hat{G}}{dG} \\
\frac{d\hat{G}}{d\hat{x}} &= \frac{dG}{dx}(\mathrm{ref}_x)\left(\frac{1}{\mathrm{ref}_G}\right) \;.
\end{aligned} \tag{36}$$

So that means that ref_x and ref_F both have opposite effects on the actual derivative value that the optimizer will see. If you increase ref_x, you will increase the size of both $d\hat{F}/d\hat{x}$ and $d\hat{F}/d\hat{x}$ relative to that of $\hat{G}(\hat{x})$. This would bias the solution path toward minimizing the objective function, instead of satisfying the constraints.

D. Summary

Gradient-based optimization transforms an under-defined problem into a well-defined implicit function that can be solved using nonlinear solution methods by taking derivatives of the Lagrangian function and setting the resulting equations equal to 0. This forms a system of equations that can be solved using nonlinear solution techniques, such as Newton's method.

In reality, not many numerical optimization algorithms actually directly solve this nonlinear system because it requires too much work to compute the Hessian. None the less, even if an optimizer does not directly solve this system of nonlinear equations, it is still searching for solutions to it. Hence, we have gone over this material so that you can understand how an optimizer sees the problem you are posing to it.

VI. Taking Derivatives of Outputs of Implicit Functions

We have seen how gradient-based optimization can create a system of implicit equations by taking derivatives of the objective function and constraints. We've also discussed briefly how Newton's method could be used to solve that system of equations, if you were willing to compute the actual Hessian matrix. If you do not wish to form the exact Hessian, then many gradient-based optimization algorithms exist that don't require it, but they all will still require derivatives of the objective function and constraints with respect to the design variables.

Up to this point, all the examples that we've considered provided the objective and constraint functions as explicit functions. In other words, we say that the objective, f, can be computed directly by evaluating the function $F(x)$. Inherent in that description of the function is the idea that there is an exact, analytic solution for F given any value of x. While this situation is certainly true for simple polynomial expressions, and also for some kinds of more complex engineering analyses, the reality is that the vast majority of modern engineering computations include implicit calculations. For example, CFD and FEA are implicit and turbine engine cycle analysis is highly implicit. Tools such as COMSOL, a popular multiphysics solver, is also implicit in nature.

A general implicit function can be represented as

$$R(x, y) = 0 \,, \tag{37}$$

where R is the residual function, x is the set of fixed inputs, and y is the set of implicit output variables that must be solved for given any specific values for x. There can be any arbitrary number of input variables, but there must always be the same number of residuals as there are implicit variables.

We refer to y as the implicit outputs, because they are the values that actually get solved for when the residuals are all equal to 0. In some cases, your objective function or constraint might be one of those y values. For example, what if you wanted to minimize the mass of a clamped beam with a load applied at the end, by varying the material thickness along the beam, subject to a limit on the displacement of the beam at the tip.

min.	m	mass (kg)
w.r.t	t	material thickness (mm)
subject to	$u < 4$	displacement (cm)

The mass of the beam m can be directly computed as a function of the thickness design variables. But the displacement of the beam (u), which will be used as a constraint, is the implicit output of a FEA. So we have no explicit form for our constraint value, but we still need to compute derivatives of it with respect to the design variables. In other words, we need to compute $\frac{du}{dt}$, when the only definition we have for u is $R(t, u) = 0$.

A. Numerical approximation of derivatives

Numerical approximation of derivatives is by far the most common approach used by engineers in practice. Most optimizers and solvers that require derivative information come with built in numerical approximation methods to use by default. The primary advantage of these numerical approximations is that they are very easy to implement. The primary disadvantage is that they are expensive and can be very inaccurate.

The first-order forward difference approximation is given by:

$$\frac{dF}{dx} \approx \frac{F(x+\epsilon) - F(x)}{\epsilon} \tag{38}$$

This can be derived by looking at a Taylor expansion of $F(x)$

$$F(x+\epsilon) = F(x) + \frac{dF}{dx}\epsilon + \frac{1}{2}\frac{d^2F}{dx^2}\epsilon^2 + ... \tag{39}$$

$$\frac{dF}{dx}\epsilon = F(x+\epsilon) - F(x) - \frac{1}{2}\frac{d^2F}{dx^2}\epsilon^2 - ... \tag{40}$$

$$\frac{dF}{dx} = \frac{F(x+\epsilon) - F(x)}{\epsilon} - \frac{1}{2}\frac{d^2F}{dx^2}\epsilon \tag{41}$$

$$\frac{dF}{dx} \approx \frac{F(x+\epsilon) - F(x)}{\epsilon} \tag{42}$$

In the last step, we neglect any terms multiplied by ϵ because it is small. Because of this, we can assume that these terms are insignificant. Thus we get an approximation for the derivative with error that varies directly with ϵ. This is why its called a first-order approximation.

You have to compute this approximation once for every variable that you want to take a derivative with respect to. So the compute cost scales linearly with the number of input variables you're considering.

What about accuracy? In theory, you can just take ϵ to be really tiny and we can get a very accurate derivative approximation. In practice though, we run into a problem when ϵ gets too small. Lets look at a simple explicit example function: $sin(x)$ at $x = \pi/4$. If we look at a sweep of step sizes from 10^{-2} to 10^{-15}, and compare the FD approximation to the analytic answer ($cos(x)$), then the relative error follows the trend shown in Figure 4.

```
import numpy as np

import matplotlib.pylab as plt

def fd_sin(x, epsilon):

    return (np.sin(x+epsilon) - np.sin(x))/epsilon

steps = np.logspace(-15, -3, base=10, num=50)
fd_derivs = fd_sin(np.pi/4, steps)
true_deriv = np.cos(np.pi/4)
rel_error_derivs = np.abs(fd_derivs-true_deriv)/true_deriv

fig, ax = plt.subplots()
ax.loglog(steps, rel_error_derivs)
ax.set_title('Relative error of the FD approximation')
ax.set_xlabel(r'log($\epsilon$)')
ax.set_ylabel(r'log(rel error)', rotation='horizontal', ha='right')

plt.savefig('sin_fd.pdf', bbox_inches='tight')
```

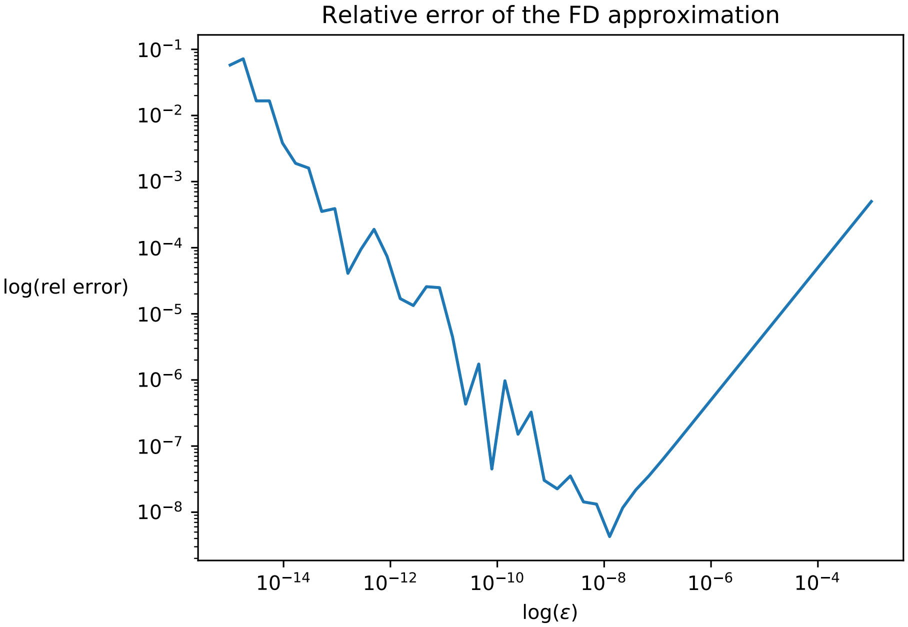

Fig. 4 Relative error of the first order FD approximation for $\frac{dsin(x)}{dx}$ vs step size (ϵ)

Between 10^{-4} and about 10^{-8} we get the linear trend we expect — the error is going down in direct proportion to the decrease in ϵ. For step sizes smaller than 10^{-8} though, the error starts to climb and eventually gets quite large. The theory says the error should continue to decrease as ϵ decreases, but the code says that we're limited in precision after a certain point.

Why is there a difference between theory and numerical reality? The discrepancy arises because computers use finite-precision math (i.e. floating point math), and this phenomenon is called subtractive cancellation. Essentially, because computers don't represent floating point numbers with infinite precision, there is a limit to how small of difference they can resolve in two numbers.

Practically, what this means is that there is always going to be a lower-limit on how accurate your numerical derivative approximation can be. The more nonlinear the function, the smaller the ϵ that you would need in theory, but the subtractive cancellation phenomena is always going to limit how small you can go.

Can we do anything to get around subtractive cancellation?
Yes! You have two options.

1) Use a higher order FD approximation, like central-difference. This will give second-order convergence, which means that the error in the derivative will be proportional to ϵ^2. In other words, for the same ϵ you get a much smaller error.

2) Don't use subtraction! Seriously...

Lets look at a Taylor expansion for $F(x + \epsilon i)$ — so we're looking at complex arithmetic now.

$$F(x+\epsilon i) = F(x) + \frac{dF}{dx}\epsilon i + \frac{1}{2}\frac{d^2F}{dx^2}(\epsilon i)^2 + \frac{1}{6}\frac{d^3F}{dx^3}(\epsilon i)^3 + ... \tag{43}$$

$$F(x+\epsilon i) = F(x) + \frac{dF}{dx}\epsilon i - \frac{1}{2}\frac{d^2F}{dx^2}\epsilon^2 + \frac{1}{6}\frac{d^3F}{dx^3}\epsilon^3 i + ... \tag{44}$$

Notice that in the right hand side of Equation (44) some of the terms are real and some of the terms are complex. In particular, the second order term that includes d^2F/dx^2 is real and negative because $i^2 = -1$. Similarly, the third order term is negative because $i^3 = -i$. We can split Equation (44) into its real and imaginary parts:

$$Re[F(x+\epsilon i)] = F(x) - \frac{1}{2}\frac{d^2F}{dx^2}(\epsilon i)^2 + ... \tag{45}$$

$$Im[F(x+\epsilon i)] = \frac{dF}{dx}\epsilon - \frac{1}{6}\frac{d^3F}{dx^3}\epsilon^3 + ... \tag{46}$$

Equation (46) has the terms we need now. We just need to re-arrange them to solve for $\frac{dF}{dx}$ and discard any higher order terms:

$$\frac{dF}{dx}\epsilon = Im[F(x+\epsilon i)] + \frac{1}{6}\frac{d^3F}{dx^3}\epsilon^3 + ... \tag{47}$$

$$\frac{dF}{dx} = \frac{1}{\epsilon}Im[F(x+\epsilon i)] + \frac{1}{6}\frac{d^3F}{dx^3}\epsilon^2 + ... \tag{48}$$

$$\frac{dF}{dx} \approx \frac{1}{\epsilon}Im[F(x+\epsilon i)] \tag{49}$$

Equation (49) is called the complex-step method. Notice that there is no subtraction at all here, so this method does not suffer from subtractive cancellation. This means that you can take very small steps in the complex plane and not suffer numerical loss of accuracy. Also, the largest term we discarded was on the order of ϵ^2, so the complex-step method is second order accurate. These two things combined — the lack of numerical issues and the second order convergence — combine to make the complex-step method very accurate. If you take $\epsilon = 1e - 40$, we would typically consider the complex-step method to return a numerically-exact derivative.

Note that, similar to the FD calculations, you must still compute one additional function evaluation per design variable you want to differentiate with respect to. So the complex-step method has the same computational scaling as FD. Also note that in order to use the complex-step method, your code must support complex arithmetic. Python, Matlab, and Fortran all natively support complex arithmetic so complex-step is very easy to apply for those languages.

One last thing to be aware of is that not all functions are implemented in a complex-step safe manner. For example, the *np.linalg.norm* method does not work for complex-step correctly. So there are times when complex-step might not give an accurate answer if you have some non compatible functions in your code.

B. Computing analytic derivatives of outputs of implicit functions

If you don't want to use FD or complex-step methods, then your other option is analytic differentiation. For explicit functions, you know how to do this. It's just a lot of basic calculus, algebra, and the chain-rule. However, if you want to analytically differentiate an implicit function things get a bit more interesting.

For a general implicit function, given by $R(x, y) = 0$, we seek dy/dx. For example, in Equation (9), we are talking about $dMN/d\frac{A}{A^*}$. The derivative in question is a total-derivative, which accounts for the change in the converged value of the implicit outputs with respect to changes in the inputs. Since we have no analytic expression for the implicit outputs, there is nothing obvious to differentiate.

Lacking any obvious path... lets differentiate the only equation we have at hand, the residual equation:

$$\frac{dR}{dx} = 0 = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial y}\frac{dy}{dx} \,. \tag{50}$$

The term we are looking for has shown up in this equation, but you might be asking how we know that $dR/dx = 0$? Remember that we are seeking the change in the converged values of the implicit output variables. That means that we are assuming that $R(x, y) = 0$ is always true, and hence we know that its value is never changing and so its derivative must be 0.

Equation (50) can be re-arranged to solve for dy/dx:

$$\begin{aligned} \left[\frac{\partial R}{\partial y}\right]\frac{dy}{dx} &= -\frac{\partial R}{\partial x} \\ \frac{dy}{dx} &= -\left[\frac{\partial R}{\partial y}\right]^{-1}\frac{\partial R}{\partial x} \,. \end{aligned} \tag{51}$$

We now have a linear system of equations, the solution to which gives us dy/dx. Lets examine this linear system more closely. We see the quantity that we are looking for on the left-hand side of Equation (52). On the right-hand side, there are two terms, both composed of partial derivatives of the residual function. The $\partial R/\partial y$ term is a square matrix, of size $N_y \times N_y$. Note that this is the same exact term that would be needed to implement Newton's method. Partial derivatives of the residual function are relatively easy to compute by hand, or they can even be approximated with FD or complex-step.

The $\partial R/\partial x$ term is of size $N_y \times N_x$. So, double checking the sizes in Equation (52) shows us that everything matches up and we get the correct size for the total derivative ($N_y \times N_x$):

$$\underbrace{\frac{dy}{dx}}_{N_y \times N_x} = -\underbrace{\left[\frac{\partial R}{\partial y}\right]^{-1}}_{N_y \times N_y}\underbrace{\frac{\partial R}{\partial x}}_{N_y \times N_x} \,. \tag{52}$$

What Equation (52) is telling us is that we can solve for one column of dy/dx each time we solve that linear system. Each column corresponds to one of the input variables of the implicit function. If you can compute an explicit inverse of $\partial R/\partial y$, as shown in Equation (52), Then each of these linear solves is little more than a simple matrix-vector product, which is very cheap.

However, for complex implicit functions like CFD or FEA you might not be able (or at least might not want to) actually compute the inverse for $\partial R/\partial y$. In these cases, there are iterative linear solvers (e.g. Krylov methods) which can solve the linear system for you. These methods scale to the larger problems that a CFD solver poses, but they are also more expensive. In these cases, you can roughly approximate the cost of a linear solve to be the same as that of the non-linear solve. So in this situation, the computational scaling of this approach requires one linear solve per design variable which we are assuming to be roughly equivalent in cost to one nonlinear function evaluation. So this approach, similar to FD and complex-step methods, also scales linearly with the number of design variables.

VII. Taking derivatives of outputs of mixed implicit/explicit functions

In real applications, your model is likely to be built from a mixture of implicit and explicit functions that are chained together. A really simple example of this would be if you wanted to optimize the thickness distribution of a beam subject to an end load to minimize the average deflection of the beam as predicted by an FEA model.

Admittedly, this is a silly optimization problem formulation. We know the answer here will simply be to pick the thickest structure allowed everywhere. But the purpose of the example is not to come up with a good optimization, but rather a simple mixed explicit/implicit function.

Another example would be to minimize the thrust specific fuel consumption (TSFC) for a jet engine with respect to the design parameters of that engine cycle — engine cycle analysis requires the solution to an implicit nonlinear system of equations to compute implicit outputs which are then fed into the TSFC calculation).

So we want to minimize some objective function, $F(x, y)$, with respect to x where $R(x, y) = 0$. The objective function itself is a simple explicit function of x, y. But one of the inputs to that function is the output of an implicit function, y. The optimizer is going to need df/dx. How do we compute that?

A. Monolithic FD or Complex-step

One option would be to use FD across the whole model. Lets call that the "monolithic numerical approach", because you are treating the entire model as a single monolithic function. In other words, you would be numerically computing derivatives of

$$f = \hat{F}(x)\,, \tag{53}$$

where $\hat{F}(x)$ is the overall function that looks explicit, but really is an implicit function in disguise. This is the approach that is most commonly taken by users of gradient-based optimization. This is what you get by default when you just provide the objective function to matlab's *fmincon* or scipy's *minimize*.

While it is easy to implement, the approach has some important downsides. Just as we discussed before, the computational cost for this approach scales linearly withif x. So if x is large (i.e. you have a lot of design variables), then this will be very expensive.

Perhaps we're willing to pay the computational cost, because we can use some parallel computing resources to keep the wall time down. Even then, we will still potentially run into another challenge. Realizing that this monolithic function is actually an example of an implicit function disguised as an explicit one, we know that approximating derivatives of it with FD will be noisy at best and horribly wrong at worst. We could use complex-step, assuming that $\hat{F}(x)$ was complex-safe, but that would mean that even the nonlinear solver itself would need to be complex-safe. This is doable but somewhat non-trivial in practice.

So our goal is to find another way that is both more computationally efficient and more accurate than the monolithic numerical approximations.

B. The direct analytic derivative method

We're going to derive the direct analytic derivative method, using an approach very similar to what was used in Section VI.B. This method will give us better accuracy than the monolithic numerical approximation, with moderately lower computational cost.

Again, we have arbitrary function $F(x, y)$, where x is a vector of design variables and y is a vector of implicit output variables that also depend on x. From the chain rule, we know that

$$\frac{dF}{dx} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{dy}{dx}. \tag{54}$$

If we use Equation (52), we can directly solve for the dy/dx term and substitute it in:

$$\frac{dF}{dx} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial y}\left(\left[\frac{\partial R}{\partial y}\right]^{-1}\frac{\partial R}{\partial x}\right). \tag{55}$$

Equation 55 is called the "direct analytic method". Notice that the right hand side of Equation 55 is composed only of partial derivatives. Partial derivatives are relatively easy and inexpensive to compute them. If we derived all the partial derivative terms by hand, then we would have a pure analytic method. We could also approximate them numerically using FD or complex-step, which would give us a semi-analytic method.

The semi-analytic method will be less accurate than the pure analytic method, but will still be far cheaper and more accurate than the monolithic numerical approximation approaches.

In Equation 55, we've used an explicit inverse of $\partial R/\partial y$. In some cases, that inverse is not feasible to compute. In that case, the direct method is solved in two steps: First you would solve for the columns of dy/dx one at a time, using an iterative linear solver to find solutions to

$$\left[\frac{\partial R}{\partial y}\right]\frac{dy}{dx} = -\frac{\partial R}{\partial x}\ . \tag{56}$$

Then once you have assembled dy/dx, you can plug that matrix directly into Equation (54) to compute dF/dx. So the compute cost of the direct analytic method scales with the size of x, because that governs how many linear solves are needed to assembled dy/dx.

C. The adjoint analytic derivative method

While the direct analytic derivative method does outperform monolithic numerical differentiation, its computational cost still scales with the number of design variables. So if we formulate design problems with 1000 or 1,000,000 design variables, the direct method is still going to be very expensive.

With some manipulation of the direct method equations, we can reformulate things to give us the same accuracy but with much better computational scaling. As we'll see, the computational cost of the adjoint method scales with the number of outputs that we want to take derivatives of, and is independent of the number of design variables.

Let's re-examine Equation (55), but this time paying attention to the sizes of all the matrices involved. We have a design variable vector, x, that is length n. We have a implicit variable vector, y, that is length m. We'll assume that both n and m are very large. $F(x, y)$ is a scalar function, so it has a size of 1.

$$\underbrace{\frac{dF}{dx}}_{1\times n} = \underbrace{\frac{\partial F}{\partial x}}_{1\times n} - \overbrace{\frac{\partial F}{\partial y}}^{1\times m}\left(\overbrace{\left[\frac{\partial R}{\partial y}\right]^{-1}}^{m\times m}\ \underbrace{\frac{\partial R}{\partial x}}_{m\times n}\right). \tag{57}$$

Lets rearrange the parenthesis on the right most term:

$$\underbrace{\frac{dF}{dx}}_{1\times n} = \underbrace{\frac{\partial F}{\partial x}}_{1\times n} - \underbrace{\left(\overbrace{\frac{\partial F}{\partial y}}^{1\times m}\ \overbrace{\left[\frac{\partial R}{\partial y}\right]^{-1}}^{m\times m}\right)}_{\psi^T}\underbrace{\frac{\partial R}{\partial x}}_{m\times n} \tag{58}$$

$$\frac{dF}{dx} = \frac{\partial F}{\partial x} - \psi^T\frac{\partial R}{\partial x} \tag{59}$$

We just combined $\partial F/\partial y$ and $\partial R/\partial y$ into a single $1 \times m$ vector, which we've labeled ψ^T. Now, if we can find a way to compute ψ, then we can solve for the total derivative we need.

$$\begin{aligned}
\psi^T &= \frac{\partial F}{\partial y}\left[\frac{\partial R}{\partial y}\right]^{-1} \\
\psi &= \left[\frac{\partial F}{\partial y}\left[\frac{\partial R}{\partial y}\right]^{-1}\right]^T \\
\psi &= \left[\frac{\partial R^T}{\partial y}\right]^{-1}\frac{\partial F^T}{\partial y} \\
\left[\frac{\partial R^T}{\partial y}\right]\psi &= \frac{\partial R^T}{\partial y}\left[\frac{\partial R^T}{\partial y}\right]^{-1}\frac{\partial F^T}{\partial y} \\
\underbrace{\left[\frac{\partial R}{\partial y}\right]^T}_{m\times m}\underbrace{\psi}_{m\times 1} &= \underbrace{\left[\frac{\partial F}{\partial y}\right]^T}_{m\times 1}.
\end{aligned} \tag{60}$$

Equation (60) gives us a way to solve for ψ using only partial derivative terms that we already know, but more importantly it only requires one linear solve! This is called the "adjoint analytic method", and it allows us to compute the total derivative we need with only a single linear solve with the following two equations:

$$\begin{aligned}\left[\frac{\partial F}{\partial y}\right]^T \psi &= \frac{\partial F}{\partial y}^T \\ \frac{dF}{dx} &= \frac{\partial F}{\partial x} - \psi^T \frac{\partial R}{\partial x}\ .\end{aligned} \tag{61}$$