# ME 575: Homework 3

PJ Stanley

February 18, 2016

## 1 Introduction

In this paper, I will use provided Python code (which I have heavily modified) to explore different methods of obtaining gradients for a function. This code takes returns the mass, stress, and derivatives of both mass and stress of a ten bar truss arrangement, given the cross sectional area of each bar. Within this function, I implemented finite difference, complex step, and the adjoint method to obtain $\frac{\delta m}{\delta A}$ and $\frac{\delta \sigma}{\delta A}$.

I will also discuss the minimization of the total mass of the system using a built in Python optimizer with constraints of maximum stress imposed on each bar.

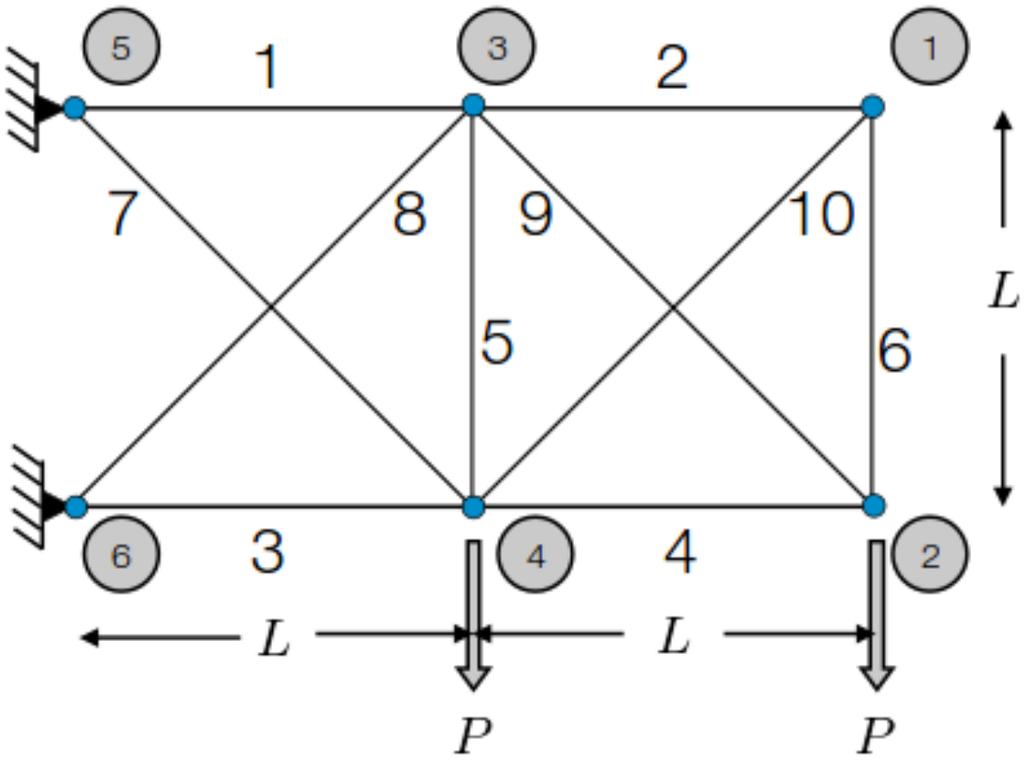


Figure 1: The truss set up that will be analyzed in this paper. The maximum stress allowed on each bar is 25,000 psi, with the exception of bar 9, which has a maximum stress of 75,000 psi.

Lastly, I will discuss the use of automatic differentiation to determine partial derivatives for a problem specific to my team project.

## 2 3.1: Truss Derivatives

Partial derivatives $\frac{\delta m}{\delta A}$ and $\frac{\delta \sigma}{\delta A}$ were obtained using three different methods. The central difference method was used as a finite difference method, as well as complex step and the adjoint method. Because there are 10 bars, and consequentially the partial derivatives of stress with respect to the cross sectional area of each bar is a 10x10 matrix, I will only report $\frac{\delta m}{\delta A}$ from the area of the first bar, and the first entry of the first vector of $\frac{\delta \sigma}{\delta A}$. These values will only be used to compare each of the methods. Know that this is representative of the all the other values, and for further validation please refer to the attached code. Each of the derivatives is evaluated with a cross sectional area of 2 $in^2$ for each of the bars.

| Method | $\frac{\delta m}{\delta A}$ | $\frac{\delta \sigma}{\delta A}$ |
|---|---|---|
| Finite Difference | 36.00000002 | -43131.02470769 |
| Complex Step | 36. | -43131.02474161 |
| Adjoint | 36. | -43131.02474161 |

Table 1: Selections of partial derivatives from the truss function, A = 2 $in^2$.

As can be seen from the table, the values of the derivatives are practically identical for this application. Each value is the same for at least 9 digits of accuracy. We see a similar pattern even when using a different cross sectional area. When we use an area of 3 $in^2$, you get the following results.

| Method | $\frac{\delta m}{\delta A}$ | $\frac{\delta \sigma}{\delta A}$ |
|---|---|---|
| Finite Difference | 36.00000002 | -19169.34433393 |
| Complex Step | 36. | -19169.3443296 |
| Adjoint | 36. | -19169.3443296 |

Table 2: Selections of partial derivatives from the truss function, A = 3 $in^2$.

The finite difference method that I used, central difference requires two function calls per derivative calculation, and is therefore very computationally expensive. Also, finite difference methods only give an approximate value of the gradient. When the step size used in it's calculation is too large, the gradient calculation is not very accurate. Additionally, there is a subtraction used in this method, and therefore if the step size used is too small and the function values that are compared are not different enough, machine epsilon will become a problem. I used a step size of $10^{-6}$. It is interesting that the derivatives using complex step and the adjoint method are identical to the precision that is given. Both calculate the exact gradient. The real advantage comes in the number of function calls required. Complex step is still as computationally expensive as finite difference, but it does not use a subtraction so machine epsilon is no longer an issue. The adjoint method is extremely powerful because provides an analytic solution, and is not as computationally expensive, as long as it is done correctly. The computation expense of the adjoint method scales with the number of outputs.

# 3 3.2 Truss Optimization

I used a Scipy optimizer to minimize the mass of the truss given the constraints on maximum stress. Specifically, I used scipy.optimize.minimize with the "SLSQP" method. To do so, I needed to provide a function that returned mass and $\frac{\delta m}{\delta A}$ as the objective function. Constraints were given in the form $c \geq 0$, and included the the stress for each beam could not exceed the yield stress, in tension or compression. Additionally, the partial derivative of the constraints with respect to area were provided, as they were calculated in part 1. I minimized the mass with starting areas of all the bars being 3 $in^2$. Also, I ran this optimization using 2 different methods to find gradients: finite difference and adjoint.

The optimal mass and corresponding cross sectional areas are as follows:

| Mass | 1497.60000001 lbs |
|---|---|
| Areas | [ 7.9, 0.1, 8.1, 3.9, 0.1, 0.1, 5.79827561, 5.51543289, 3.67695526, 0.14142136] $in^2$ |

Table 3: Optimized cross sectional areas for minimal mass. This optimal value was the same for both derivative finding methods, and was found to a tolerance of $10^{-6}$

these results can be confirmed by checking the stress of each bar given these areas.

| Stress | [25,000, 25,000, -25,000, -25,000, 3.17304512e-07, 25,000, 25,000, -25,000, 37,500 -25,000] psi |
|---|---|

Table 4: Optimized cross sectional areas for minimal mass. This optimal value was the same for both derivative finding methods, and was found to a tolerance of $10^{-6}$

As we see, none of the bars exceed their yield stress. We follow the optimizer through it's progressive steps with a convergence plot.

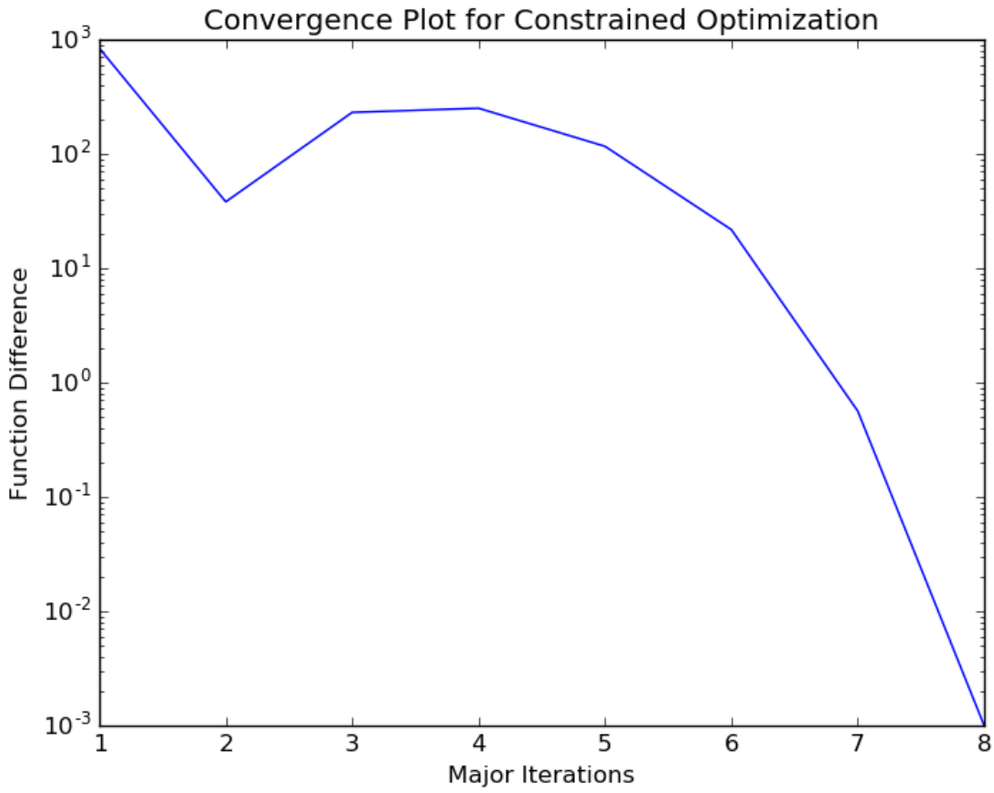


Figure 2: This figure represents the difference in function value from the previous iteration as a function of major iterations of the optimizer. We can see that only 8 objective call functions were required to converge.

The method use to calculate the gradients is only significant for the number of calls required. If our assumptions about the computational expense of the finite difference method and the adjoint method as discussed above are correct, the finite difference method should require many more function calls. This was measured by putting a counter variable in one of the intermediate functions called "truss". The results are as follows:

| Method | Function Calls |
|---|---|
| Finite Difference | 3738 |
| Adjoint | 73 |

Table 5: The number of truss function calls required for a finite difference method and an exact adjoint method to converge.

It is clear from the table, that using finite difference to calculate gradients is much worse than using the adjoint method. It is not as exact, but also requires many more function calls, about 50 times as were required with adjoint. The only downside to the adjoint method is the initial set up is more complex, and some manipulation of the internal code is required. This is not always given, therefore it is not always possible to use this method.

# 4 3.3 Automatic Differentiation

Automatic differentiation is a useful tool to quickly and relatively easily obtain derivatives. For my team project, we are using an overlap function to determine what portion of a turbine is in the wake of another turbine. I used automatic differentiation to calculate how the x and y coordinates of two turbines affect the overlap fraction. In other words, I calculated $\frac{\delta overlap}{\delta position}$. The method that was used is called operator overloading. Certain mathematical functions were redefined using a library called algopy, which returned both the function value and the derivative. I also calculated the same derivatives using finite difference to be able to compare.

| Method | Partial Derivative |
|---|---|
| Finite Difference | [-0.01509328, 0.01509328, 0.00746602, -0.00746602] |
| Automated Differentiation | [-0.01509328, 0.01509328, 0.00746602, -0.00746602] |

Table 6: Partials obtained using finite difference and automatic differentiation.

It is clear to see that both results are the same. Automatic differentiation required an extra library in Python, and therefore required some additional initial set up. Also, it results in exactly the same results, which leads me to question it's effectiveness for simple problems like I had in this situation. However, for

more complex derivatives, it is clear that automatic differentiation is far superior because it requires very little knowledge of what is inside the code, and returns the same result. For the a small excerpt of the code used to calculate the derivatives using automatic differentiation, see the Appendix. For the full code, see the attached Python zip folder.

# 5 Appendix

```
#define parameters
params = np.array([r_0, alpha])

#define values that we want to take a derivative with respect to
xin = np.array([x_r[3], x_r[6], y_r[3], y_r[6]])

#Finite Differencing
step = 1e-6
p1 = np.array([step, 0, 0, 0])
p2 = np.array([0, step, 0, 0])
p3 = np.array([0, 0, step, 0])
p4 = np.array([0, 0, 0, step])
p = np.array([p1, p2, p3, p4])

derivative_FD = np.zeros(4)
for i in range(len(p)):
    derivative_FD[i] = (overlap(xin+p[i], params)-overlap(xin, params))/step

print "FD: ", derivative_FD

#Automatic Differentiation

x_algopy = UTPM.init_jacobian(xin)

overlap_fraction = overlap(x_algopy, params)

derivative_auto = UTPM.extract_jacobian(overlap_fraction)

print "Automatic: ", derivative_auto
```

Figure 3: This is a selection of code that was used to calculate derivatives using automatic differentiation.