A Comparison of SciPy's and MATLAB's Optimization Toolboxes

Joanna Bitton
Courant Institute of Mathematical Sciences
New York University
New York, NY 10003
`jtb470@nyu.edu`

Abstract

In this study, `scipy`, an open source Python library, is compared to Matlab within the domain of optimization. A number of attributes are considered in this comparison: performance, robustness, function availability, organization, and documentation. To test performance and understand the behavior of each library's functions, three optimization problem areas were considered: one-dimensional zero finding, linear programming, and unconstrained optimization.

1 Introduction

With software engineering in high demand, there is a surplus of programming languages, libraries, and frameworks. In the domain of optimization, however, there are two top languages: Matlab and Python [5]. Although Matlab has a large scientific computing community, Python is the fastest-growing language for data science, machine learning and academic research [12]. Python is now the gold standard of machine learning development, and thus there is an abundance of open source scientific computing libraries, such as `numpy` and `scipy`.

Numerical Python, also known as `numpy`, is a library for matrix manipulation and provides numerical routines such as finding the eigenvalues of a matrix [11]. Meanwhile, `scipy` builds upon `numpy` and adds functionality for solving differential equations, computing integrals, and optimization [6].

The purpose of this study is to compare Matlab's proprietary optimization toolbox to `scipy`'s open source version. Various attributes will be considered in the analysis, such as performance, availability and robustness. Previous comparisons of optimization software will be discussed in section 2. Sections 3 and 4 contain the problem setup, experiments and their results. Finally, section 5 provides a summary of the findings and suggests avenues for future work.

2 Related Work

Due to the increasing popularity and availability of open source libraries, many researchers have explored the possibility of using Python as a language for scientific computing. Pérez et al. dictates that there is a need to integrate numerical software into large applications [3]. For example, numerical software may require access to an online database in order to perform computations before the results are displayed on a website. Since Python is a general purpose programming language, this is much easier to accomplish in comparison to Matlab or R. Additionally, the article suggests an ecosystem of open source packages for scientific computing in Python including `numpy` and `scipy` [3].

Matott et al. compared the Matlab optimization toolbox to `scipy` with two case studies in the domain of geosciences. From the two benchmark problems, `scipy`'s implementation of Hooke-Jeeves produced the best performance in terms of finding the least-cost solutions [8].

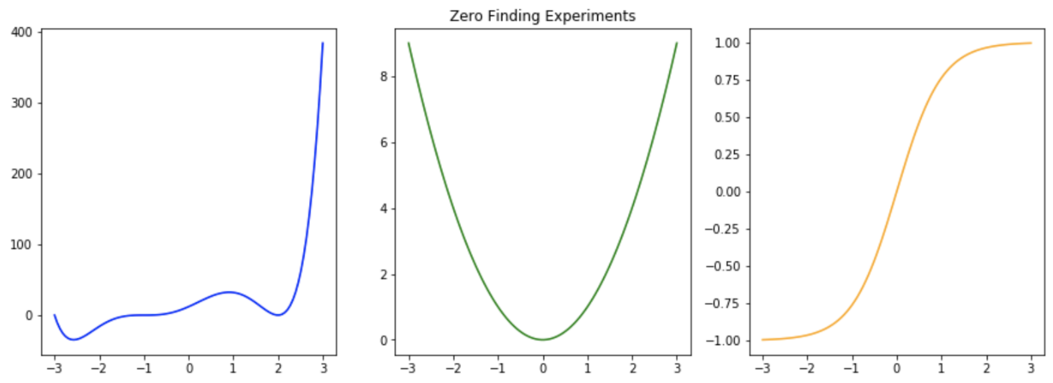

Figure 1: Three functions were utilized during the one-dimensional zero finding experiments. From left to right: equation (1), equation (2) and equation (3).

Multiple studies have proposed methods for benchmarking optimization software and algorithms. Dolan et al. proposes analyzing the distribution functions of a performance metric as a tool for profiling optimization functions. Dolan et al. claims that significant performance characteristics are revealed when the distribution of a performance metric is plotted [2]. Moré et al. suggests a tool for benchmarking derivative-free optimization algorithms under a computational budget [10].

3 Experiments

An analysis was conducted on various routines available in both toolboxes. For each routine analyzed, the execution time, number of function calls or iterations, correctness, and robustness was evaluated. Additionally, a comparison of function organization, function existence, and documentation was completed. Below are the various experiments done on each procedure.

3.1 Setup

All of the following experiments were conducted on a 2016 MacBook Pro with an Intel core i7 processor with a processing speed of 2.6GHz and 16GB of RAM. Matlab version "R2018b Update 3" was used. Additionally, Python version 3.7.3 was used along with `numpy` version 1.16.3 and `scipy` version 1.2.1. Python experiments were profiled using `cProfile` and the `time` library function `perf_counter`. Matlab experiments were profiled using the `profile` viewer.

3.2 One-Dimensional Zero Finding

To test the abilities of Matlab and Python's one-dimensional zero finding, the function `fzero` was compared to `scipy.optimize.root_scalar`. The `root_scalar` function was used with the option `method='brentq'`, an implementation of Brent's method [1]. Three tests were carried out on the following equations, also displayed in Figure 1.

$$f(x) = (x+3)(x-2)^2(x+1)^3 \tag{1}$$

$$\bar{f}(x) = x^2 \tag{2}$$

$$\hat{f}(x) = \tanh(x) \tag{3}$$

Equation (1) was chosen since it has three zeros all of different multiplicities. The interval of uncertainty $[-2.573, 1.999]$ was selected because $x = -2.573$ is a stationary point and $x = 1.999$ narrowly excludes the root $x = 2$.

Equation (2) was selected as the function has a root, but the range is nonnegative. The interval of uncertainty $[-1, 1000]$ was chosen because more than one step of bisection is required to get to the final solution.

Equation (3) was chosen since it has horizontal asymptotes. The interval $[-100, 100000]$ was selected such that more than one step of bisection would be necessary to find the final solution.

3.3 Linear Programming

To compare linear programming functionality, `scipy`'s and Matlab's `linprog` functions were assessed. Both `linprog` methods were used with the simplex method setting and all constraints were given using all-inequality form. The linear programming experiments are described below.

3.3.1 Diet Problem

Given a set of available foods, their nutritional values, and cost, determine the number of servings per food item such that the nutritional constraints are satisfied and the cost is minimized. A simple variant of this problem was taken from [14].

Assume that there are three available food items: corn, milk, and wheat bread. The amount of calories consumed must be between 2,000 and 2,250, and the amount of vitamin A must be between 5,000 and 50,000. The following table presents the nutritional values and costs per serving.

Food	**Cost**	**Calories**	**Vitamin A**
Corn	\$0.18	72	107
Milk	\$0.23	121	500
Bread	\$0.05	65	0

This problem will be used to assess the performance of each `linprog` method.

3.3.2 Klee-Minty

A version of the Klee-Minty problem was tested to evaluate the exponential "worst-case" performance of the simplex method [7].

$$\begin{aligned}
&\text{minimize} && \sum_{j=1}^{n} 10^{n-j} x_j \\
&\text{subject to} && 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1} \texttt{ for } 1 \leq i \leq n \\
& && \text{all } x_j \geq 0
\end{aligned}$$

In this experiment $n = 9$, meaning x was a nine-dimensional vector. A value larger than nine causes precision errors due to the exponentially large numbers present in the linear program.

3.3.3 Cycling

The final experiment on `linprog` examines a linear program that induces cycling when following the textbook rules. The example below was constructed by Harold Kuhn [15]:

$$\begin{aligned}
&\text{minimize} && -2x_1 - 3x_2 + x_3 + 12x_4 \\
&\text{subject to} && x_1, x_2, x_3, x_4 \geq 0 \\
& && 2x_1 + 9x_2 - x_3 - 9x_4 \geq 0 \\
& && -\tfrac{1}{3}x_1 - x_2 + \tfrac{1}{3}x_3 + 2x_4 \geq 0
\end{aligned}$$

3.4 Unconstrained Optimization

In order to test multi-dimensional unconstrained optimization functionality, `scipy`'s `minimize` procedure was compared to Matlab's `fminunc`. The `minimize` function was used with the BFGS implementation option. Meanwhile, `fminunc` was used with the default quasi-Newton algorithm option and the BFGS Hessian update strategy. Thus, `scipy`'s and Matlab's BFGS strategies were compared using the following two equations.

3.4.1 Rosenbrock Function

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right]$$

$$\text{where the minimum is: } f(\underbrace{1, \ldots, 1}_{n \text{ times}}) = 0$$

The Rosenbrock function is commonly used as a performance test for unconstrained optimization [13]. In this study, a two-dimensional variant tested the number of iterations and the runtime of each routine. In addition, a thirty-dimensional variant examined how each implementation handled higher-dimensional problems in regards to runtime and solution accuracy.

3.4.2 Ill-Conditioned Hessian

$$f(\boldsymbol{x}) = x_1 + x_2 + x_3 + x_4 + \frac{1}{2}(22202x_1^2 + x_2^2 + 10^{-1}x_3^2 + 10^{-3}x_4^2)$$

In addition to purely testing performance, a variant of a problem from [16] with a highly ill-conditioned Hessian was examined. Since BFGS includes updating the Hessian, the method is affected by ill-conditioning [4]. Thus, along with examining performance, correctness of the approximate Hessian was analyzed. For the problem shown above the Hessian has a condition number of 22,202,000, a relatively large value. The Hessian of this function is shown below.

$$\begin{pmatrix} 22202 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 10^{-1} & 0 \\ 0 & 0 & 0 & 10^{-3} \end{pmatrix}$$

4 Results

4.1 One-Dimensional Zero Finding

During experimentation, some interesting aspects of one-dimensional zero finding were uncovered in both libraries. For instance, `scipy` provides various zero finding methods such as bisection, secant, Newton's method, Brent's method and many more [6]. However, Matlab's optimization toolbox only provides Brent's method [9]. There are publicly available implementations of other methods in Matlab; however, they are not maintained by a professional cohort of software engineers and thus may be unreliable.

Experimentation revealed an additional distinction between the two procedures: `fzero` is capable of taking one point as input for the interval and invoking a "search" method to find another point of opposite sign [9]. At the time of writing this article, `root_scalar` does not have this functionality [6].

Lastly, `fzero` has an insightful feature which displays the utilized procedure (bisection, interpolation, etc.) at each iteration [9]. This is not a built-in feature for `root_scalar`. It is possible to pass in a callback function that prints out the results [6], but this must be manually implemented by the user which could affect performance due to the extra function calls. The results of the experiments are described below.

4.1.1 Experiment #1

Language	Time(s)	Function Calls	Converged?
Matlab	2.300e-2	12	✓
Python	1.709e-4	102	✓

The first experiment considered the function $f(x) = (x+3)(x-2)^2(x+1)^3$ in the interval $[-2.573, 1.999]$. Both methods were able to converge to the root $x = -1$ within the interval of uncertainty. However, Python was able to converge faster by two orders of magnitude. The number of function calls is substantially higher compared to Matlab, but this may be due to the fact that Matlab's profiler does not count function calls to $f(x)$ while Python's does.

4.1.2 Experiment #2

Language	Time(s)	Function Calls	Converged?
Matlab	`1.400e-2`	12	X
Python	`1.188e-4`	21	X

The second experiment examined the function $\bar{f}(x) = x^2$ within the interval $[-1, 1000]$. Both methods were unable to converge to the only root of this function, $x = 0$. This is due to the fact that no two points in the range of $\bar{f}(x)$ differ in sign. Execution time for Python remains two orders of magnitude lower than that of Matlab. As before, Matlab reports fewer function calls due to differences in profiling conventions.

4.1.3 Experiment #3

Language	Time(s)	Function Calls	Converged?
Matlab	`2.900e-2`	12	✓
Python	`1.257e-4`	23	✓

The last experiment inspected the function $\hat{f}(x) = \tanh(x)$ in the interval [-100, 100000]. Although the function flattens out as x approaches positive and negative infinity, the methods were able to converge to the root $x = 0$, with `root_scalar` achieving convergence two orders of magnitude faster than `fzero`.

4.2 Linear Programming

Both `linprog` methods provide two algorithms to solve linear programs: the simplex method and the interior-point algorithm. However, Matlab's implementation solves the dual simplex problem as opposed to `scipy`, which solves the primal problem. Both methods accept as input inequality and equality constraints [6, 9]. The results of the experiments are described below.

4.2.1 Diet Problem

Language	Time(s)	Number of Iterations	Converged?
Matlab	`1.860e-1`	5	✓
Python	`9.742e-3`	11	✓

Similar to the results seen in the previous section, `scipy`'s implementation is orders of magnitude faster than the Matlab implementation despite having double the iterations. Both methods however were able to converge to the correct solution of 1.94 servings of corn, 10 servings of milk, and 10 servings of bread.

4.2.2 Klee-Minty

Language	Time(s)	Number of Iterations	Converged?
Matlab	`1.770e-1`	9	✓
Python	`2.589e-1`	511	✓

Remarkably, with the Klee-Minty problem, the Matlab implementation achieved better performance than `scipy` in both time and number of iterations. In fact, `scipy`'s runtime increased by two orders of magnitude and the number of iterations increased by 500 in comparison to the diet problem. Since both implementations of `linprog` do not accept a starting vertex as input, both methods must execute a Phase-1 search for a vertex. The performance of the Klee-Minty problem is highly dependent on the starting vertex [7], so it is possible that the Matlab implementation of Phase 1 found an initial vertex closer to the optimal solution.

4.2.3 Cycling

Language	Time(s)	Number of Iterations	Converged?
Matlab	`1.730e-1`	4	✓
Python	`4.287e-1`	1000	X
Python w/Bland	`4.088e-3`	4	✓

Using the default options on `linprog`, the `scipy` implementation was unable to converge and hit the default maximum iteration limit. This is due to the fact that by default, `scipy` does not activate the Bland's rules option [6]. On the contrary, Matlab's function using default parameters exited after four iterations upon discovering that the linear program is unbounded. When `scipy`'s method is run with the Bland's rules option, the function is also able to exit in four iterations and its runtime is two orders of magnitude lower than Matlab's.

4.3 Unconstrained Optimization

Matlab provides two overarching algorithms for unconstrained minimization: the quasi-Newton and trust-region methods. The quasi-Newton method has three options for Hessian update strategies: BFGS, DFP, and steepest descent. The trust-region method has two options for calculating the iteration step: conjugate gradient or Cholesky factorization [9]. In contrast, `scipy` has nine different unconstrained minimization strategies such as the Nelder-Mead algorithm, Powell's method, BFGS, and four trust-region variants. Beyond the provided algorithms, a custom minimization method may also be passed in as input [6]. While Matlab approximates the exact Hessian in the BFGS update strategy, `scipy` approximates the inverse Hessian [6, 9].

4.3.1 Rosenbrock Function: Two-Dimensional

Language	Time(s)	Number of Iterations	Converged?
Matlab	`1.540e-1`	48	✓
Python	`8.844e-3`	58	✓

The two-dimensional Rosenbrock function reinforces what has been observed in previous experiments. `scipy` again converged to the final solution two orders of magnitude faster than Matlab, but has a higher number of iterations.

4.3.2 Rosenbrock Function: Thirty-Dimensional

Language	Time(s)	Number of Iterations	Converged?
Matlab	`3.920e-1`	67	X
Matlab w/modified tols	`2.973e+0`	844	✓
Python	`2.722e+0`	1279	✓

The thirty-dimensional Rosenbrock function had a significant effect on `scipy`'s performance. It took just under three seconds and over 1,200 iterations to find the optimal x that minimizes the equation. In contrast, using the default optimality tolerance, maximum function evaluations, and maximum iteration values for `fminunc` resulted in an incorrect solution to the optimization problem. However, after modifying the optimality tolerance to `1e-30`, the maximum evaluations to 100,000, and maximum iterations to 1,000, the method was able to converge to the correct solution. It took Matlab 0.221 seconds longer to complete compared to `scipy` despite having fewer iterations.

4.3.3 Ill-Conditioned Hessian

Language	Time(s)	Number of Iterations	Converged?
Matlab	`1.050e-1`	53	✓
Python	`4.462e-3`	23	✓

$$\texttt{matlab\_approximation} = \begin{pmatrix} 22202 & 0 & -3.81e-7 & 0 \\ 0 & 1 & -7.62e-7 & 0 \\ -3.81e-7 & -7.62e-7 & 1e-1 & 0 \\ 0 & 0 & 0 & 1e-3 \end{pmatrix}$$

$$\texttt{scipy\_approximation} = \begin{pmatrix} 22083 & -2.95 & 3.46e-1 & 3.82e-2 \\ -2.95 & 1.11 & 4.84e-2 & -4.49e-4 \\ 3.46e-1 & 4.84e-2 & 1.24e-1 & 1.24e-4 \\ 3.82e-2 & -4.49e-4 & 1.24e-4 & 1e-3 \end{pmatrix}$$

Despite the function having an ill-conditioned Hessian, both BFGS implementations converged faster compared to the other unconstrained optimization experiments. Interestingly, the runtime *and* number of iterations was lower for `scipy` compared to Matlab. However, Matlab computes a more accurate approximation of the Hessian compared to `scipy`, whose approximation includes some noise. As mentioned previously, `scipy`'s BFGS method approximates the inverse Hessian, meaning `scipy` must take the inverse of the inverse in order to approximate the original Hessian. These additional operations may induce precision errors in the final result.

4.4 Documentation

Documentation is fundamental part of any software package. After analyzing both `scipy`'s and Matlab's optimization documentation, it is clear that both development groups have heavily invested in documentation efforts. Both sets of documentation have a single main page that contains all available optimization functions. The pages are organized by problem area, such as linear programming or nonlinear optimization, and give a one-line description of each function. Unlike Matlab, `scipy` shows the required parameters and some optional arguments of each function on its main page [6, 9]. This is convenient because a user does not need to click on the documentation for the specific function to know the valid input parameters.

For both libraries, in the function-specific documentation page, there are detailed explanations of each input parameter, output parameter, and method. Matlab also provides a description of what occurs when certain parameter configurations are passed into a function. Moreover, they provide an extensive number of examples showcasing different ways the function can be used. `scipy`, on the other hand, provides about two examples per function [6, 9].

4.5 Function Organization

Matlab and `scipy` have different approaches to routine organization. `scipy` tends to consolidate problem areas into single functions. For instance, all multivariate nonlinear optimization problems are solved using the `minimize` function [6]. Matlab, however, divides its multivariate nonlinear optimization solvers into the following categories: unconstrained optimization, derivative-free unconstrained optimization, constrained optimization, and semi-infinitely constrained optimization [9]. From a software engineering point of view, `scipy`'s method encourages code reuse for all minimization strategies. In addition, `scipy`'s code is extensible and modular since custom minimization methods can be passed in as input [6]. This is most likely due to the object-oriented nature of the Python language. On the contrary, Matlab's routine separation may be a technique to avoid constructing a monolithic function.

Some users may prefer the separation of functions, as it reduces the number of optional parameters. However, others may prefer learning only one function that can be used to solve a variety of problems.

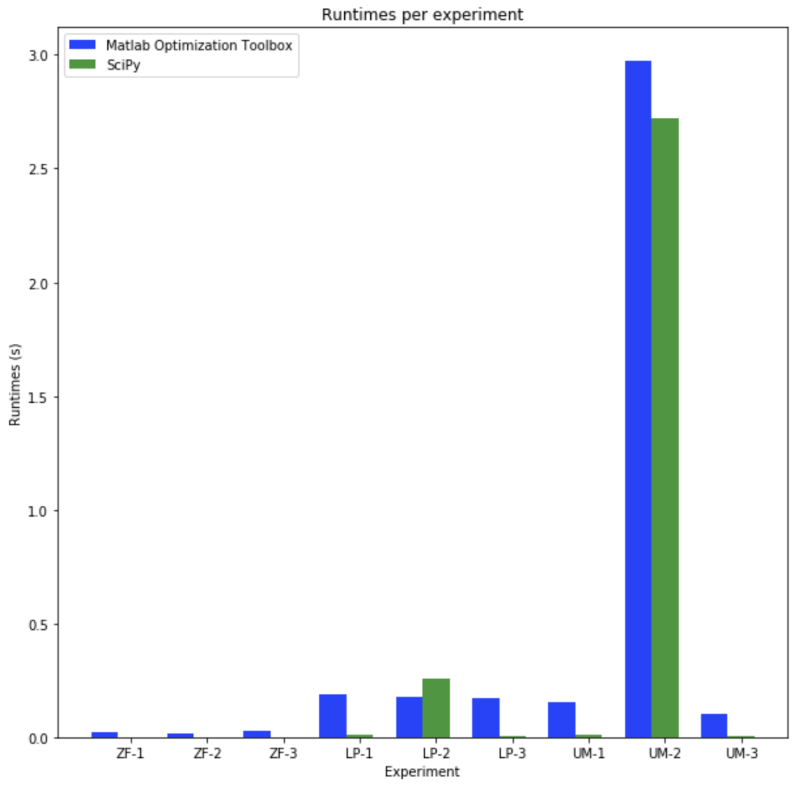

Figure 2: The figure above compares the runtime of each experiment in seconds. "ZF" refers to one-dimensional zero finding, "LP" refers to linear programming, and "UM" refers to unconstrained minimization. The digit following each prefix is the experiment number, corresponding to the order of appearance in this paper.

5 Conclusion

5.1 Summary

Overall, both `scipy` and Matlab have the necessary functionality to solve optimization problems. However, as seen in Figure 2, all `scipy` experiment runtimes are faster in comparison to Matlab, with the exception of one linear programming test. It is worth noting that `scipy` may sacrifice accuracy for speed. In the final unconstrained optimization experiment, Matlab's approximation to the Hessian was far more accurate than `scipy`'s.

For one-dimensional zero finding and unconstrained optimization, the Python package provides a larger variety of algorithms[1]. The parabola zero finding experiment demonstrated the advantage of including several different algorithms; a root could have been found if Newton's method was used instead of Brent's method. With `scipy`, this is simple to achieve since it implements a variety of methods, but in Matlab a user would have to either implement the routine or seek out other resources. Many of these algorithms work well for certain problems and are worse for others. Thus, having several options is an advantage.

In contrast, both `linprog` procedures provided the same algorithms, though Matlab solved the dual simplex problem and `scipy` solved the primal. Although Python's procedure is faster in two-thirds of the linear programming experiments, Matlab's routine is more robust to edge cases such as cycling and the exponential-runtime Klee-Minty problem without modifying optional parameters.

As for documentation, both are detailed and provide sufficient guidance to the user. In addition, both function organizations are intuitive for users.

[1]See Appendix for a table comparing their available functions.

5.2 Future Work

As with any other study, there is a plethora of other tests that could be done to evaluate `scipy` and Matlab's optimization toolbox. This study focused on one-dimensional zero finding, linear programming, and unconstrained optimization. There are a variety of other methods that could be evaluated, such as nonlinear constrained optimization, linear least squares, and multi-objective optimization. Aside from testing other functions, it is worth revisiting previous problem areas with higher-dimensional experiments. These libraries could also be compared by testing each method on different hardware. Matlab is optimized for certain architectures, so the performance of a method on one device might not necessarily translate to another [9].

References

[1] Richard P. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, 1973.

[2] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *CoRR*, cs.MS/0102001, 2001.

[3] Brian E Granger Fernando Pérez and John D. Hunter. Python: An ecosystem for scientific computing. *Computing in Science and Engineering*, 13:13–21, 2010.

[4] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 1987.

[5] Pablo Guerrón Jesús Fernández-Villaverde and David Z. Valencia. Scientific computing languages, 2019.

[6] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[7] Victor Klee and George J Minty. How good is the simplex method? *Inequalities-III*, page 159–175, 1972.

[8] Kenny Leung Loren S. Matott and Junyoung Sim. Application of matlab and python optimizers to two case studies involving groundwater flow and contaminant transport modeling. *Computers & Geosciences*, 37:1894–1899, 2011.

[9] MATLAB. *version R2018b update 3*. The MathWorks Inc., Natick, Massachusetts, 2018.

[10] Jorge J. Moré and Stefan M. Wild. Benchmarking derivative-free optimization algorithms. *Society of Industrial and Applied Mathematics*, 20:172–191, 2009.

[11] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed <today>].

[12] David Robinson. Why is python growing so quickly?, 2017.

[13] Howard H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3:175–184, 1960.

[14] Joseph Czyzyk Timothy J. Wisniewski. The diet problem: A www-based interactive case study in linear programming. 1996.

[15] Margaret H. Wright. Numerical optimization: Homework 2, 2019.

[16] Margaret H. Wright. Numerical optimization: Homework 5, 2019.

6 Appendix: Matlab vs Python Functions

Available Matlab and Python Functions[6, 9]

Problem Area	Matlab	Python
1. One-Dimensional Zero Finding		
• Brent's method	✓	✓
• Brent's method with hyperbolic extrapolation		✓
• Ridder's method		✓
• Bisection		✓
• Newton-Raphson's method		✓
• Secant method		✓
• TOMS Algo 748		✓
• Halley's method		✓
2. Linear Programming		
• Simplex method	✓	✓
• Interior-point algorithm	✓	✓
3. Unconstrained Minimization		
• quasi-Newton w/BFGS	✓	✓
• quasi-Newton w/DFP	✓	
• quasi-Newton w/steepest descent	✓	
• trust-region w/conjugate gradient	✓	✓
• trust-region w/factorization	✓	
• Nelder-Mead method		✓
• Powell's method		✓
• Nonlinear Conjugate Gradient		✓
• Truncated Newton Method		✓
• dog-leg trust-region algorithm		✓
• Newton GLTR trust-region algorithm		✓
• Nearly-exact trust-region algorithm		✓

7 Appendix: Matlab Code

```
% zero finding - experiment 1
f = @(x) (x+3)*(x-2)^2*(x+1)^3
fzero(f, [-2.573, 1.999])

% zero finding - experiment 2
f = @(x) x^2
fzero(f, [-1, 1000])

% zero finding - experiment 3
fzero(@tanh, [-100, 100000])

% linear program - diet LP
A = [
    -1, 0, 0;
    0, -1, 0;
    0, 0, -1;
    1, 0, 0;
    0, 1, 0;
    0, 0, 1;
    -72, -121, -65;
    72, 121, 65;
    -107, -500, 0;
    107, 500, 0
]
b = [0, 0, 0, 10, 10, 10, -2000, 2250, -5000, 50000]
c = [0.18, 0.23, 0.05]
[x, fval, exitflag, output] = linprog(c, A, b)

% linear program - klee-minty LP
n = 9
A = [eye(n) * -1; zeros(n,n)]
for i = 1 : 1:n
   for j = 1 : 1:(i-1)
       A(i+n,j) = 2 * (10^(i-j))
   end
   A(i+n,i) = 1
end
b = [zeros(1,n) arrayfun(@(i) 100^(i-1), 1:n)]
c = arrayfun(@(j) -1*(10^(n-j)), 1:n)
[x, fval, exitflag, output] = linprog(c, A, b)

% linear program - cycling LP
A = [
    -1, 0, 0, 0;
    0, -1, 0, 0;
    0, 0, -1, 0;
    0, 0, 0, -1;
    -2, -9, 1, 9;
    1/3, 1, -1/3, -2
]
b = [0, 0, 0, 0, 0, 0]
c = [-2, -3, 1, 12]
[x, fval, exitflag, output] = linprog(c, A, b)
```

8 Appendix: Matlab Code, Continued

```
% bfgs - rosenbrock - 2-dim
n = 2
fun = @(x) sum(arrayfun(@(i) 100*(x(i) - x(i-1)^2)^2 + (1 - x(i-1))^2, 2:n))
[x, fval, exitflag, output, grad, hessian] = fminunc(fun, [5 -10])

% bfgs - rosenbrock - 30-dim
n = 30
fun = @(x) sum(arrayfun(@(i) 100*(x(i) - x(i-1)^2)^2 + (1 - x(i-1))^2, 2:n))
x0 = [ 149  -64  283  214  273 -199  222  -64  -60 -118   28 -179  288  108 ...
   28 -102  298 -157   54 -114   30  135  207 -103   84 -166  196 -201 ...
  -23  162]
[x, fval, exitflag, output, grad, hessian] = fminunc(fun, x0)
[x, fval, exitflag, output, grad, hessian] = fminunc(
        fun,
        x0,
        optimoptions(
                @fminunc,
                'OptimalityTolerance', 1e-30,
                'MaxFunctionEvaluations', 100000,
                'MaxIterations', 1000
        )
)

% bfgs - ill conditioned hessian problem
fun = @(x) x(1) + x(2) + x(3) + x(4) + 0.5*(22202*x(1)^2 + ...
      x(2)^2 + 0.1*x(3)^2 + 0.001*x(4)^2)
[x, fval, exitflag, output, grad, hessian] = fminunc(fun, [1,1,1,1])
```

9 Appendix: Python Code

```python
import math
import time
import cProfile
import numpy as np
import scipy.optimize as opt

"""
root_scalar vs fzero

problem: experiment 1
"""

f = lambda x: (x+3)*(x-2)**2*(x+1)**3
cProfile.run("opt.root_scalar(f, bracket=[-2.573, 1.999], method='brentq')")
start = time.perf_counter()
opt.root_scalar(f, bracket=[-2.573, 1.999], method='brentq')
end = time.perf_counter()
print(f"time: {end-start}")

"""
root_scalar vs fzero

problem: experiment 2
"""

f = lambda x: x**2
cProfile.run("opt.root_scalar(f, bracket=[-1, 1000], method='brentq')")
start = time.perf_counter()
opt.root_scalar(f, bracket=[-1, 1000], method='brentq')
end = time.perf_counter()
print(f"time: {end-start}")

"""
root_scalar vs fzero

problem: experiment 3
"""

f = lambda x: math.tanh(x)
cProfile.run("opt.root_scalar(f, bracket=[-100, 100000], method='brentq')")
start = time.perf_counter()
opt.root_scalar(f, bracket=[-100, 100000], method='brentq')
end = time.perf_counter()
print(f"time: {end-start}")
opt.root_scalar(f, bracket=[-100, 100000], method='brentq')
```

10 Appendix: Python Code, Continued

```
"""
linprog vs linprog

problem: classic diet problem
"""

c = np.array([0.18, 0.23, 0.05])
A = np.array([
    [-1, 0, 0],  [0, -1, 0],  [0, 0, -1], [1, 0, 0],  [0, 1, 0], [0, 0, 1],
     [-72, -121, -65], [72, 121, 65], [-107, -500, 0], [107, 500, 0]
])
b = np.array([0, 0, 0, 10, 10, 10, -2000, 2250, -5000, 50000])

start = time.perf_counter()
opt.linprog(c, A_ub=A, b_ub=b, method='simplex')
end = time.perf_counter()
print(f"time: {end-start}")
opt.linprog(c, A_ub=A, b_ub=b, method='simplex')

"""
problem: klee-minty variation problem
"""
n = 9
c = np.array([-1*(10**(n-j-1)) for j in range(0, n)])
A = np.zeros((2*n,n))
np.fill_diagonal(A, -1)
for i in range(0, n):
    for j in range(0, i):
        A[i+n][j] = 2 * (10**(i-j))
    A[i+n][i] = 1
b = np.append(np.zeros(n), np.array([100**i for i in range(0, n)]))

start = time.perf_counter()
opt.linprog(c, A_ub=A, b_ub=b, method='simplex')
end = time.perf_counter()
print(f"time: {end-start}")
opt.linprog(c, A_ub=A, b_ub=b, method='simplex')

"""
problem: cycling
"""

c = np.array([-2, -3, 1, 12])
A = np.array([
    [-1, 0, 0, 0],  [0, -1, 0, 0],  [0, 0, -1, 0],
    [0, 0, 0, -1], [-2, -9, 1, 9], [1/3, 1, -1/3, -2]
])
b = np.array([0, 0, 0, 0, 0, 0])

start = time.perf_counter()
opt.linprog(c, A_ub=A, b_ub=b, method='simplex', options={'bland': True})
end = time.perf_counter()
print(f"time: {end-start}")
opt.linprog(c, A_ub=A, b_ub=b, method='simplex', options={'bland': True})
```

11 Appendix: Python Code, Continued

```
"""
minimize vs fminunc (unconstrained optimization using BFGS)

problem: rosenbrock, 2-dim
"""
n = 2
f = lambda x: sum([100*(x[i] - x[i-1]**2)**2 + (1 - x[i-1])**2 for i in range(1,n)])
x0 = np.array([[5, -10]])

start = time.perf_counter()
opt.minimize(f, x0, method='BFGS')
end = time.perf_counter()
print(f"time: {end-start}")
opt.minimize(f, x0, method='BFGS')

"""
minimize vs fminunc (unconstrained optimization using BFGS)

problem: rosenbrock, 30-dim
"""
n = 30
f = lambda x: sum([100*(x[i] - x[i-1]**2)**2 + (1 - x[i-1])**2 for i in range(1,n)])
x0 = np.array([ 149,  -64,  283,  214,  273, -199,  222,  -64,  -60, -118,   28,
       -179,  288,  108,   28, -102,  298, -157,   54, -114,   30,  135,
        207, -103,   84, -166,  196, -201,  -23,  162])

start = time.perf_counter()
opt.minimize(f, x0, method='BFGS')
end = time.perf_counter()
print(f"time: {end-start}")
opt.minimize(f, x0, method='BFGS')

"""
minimize vs fminunc (unconstrained optimization using BFGS)

bad problem: f(x) = x1^2*x4 + 11101*x2^2 + x2*x3^2 + x3*x4
  - cond = 22202000
"""

f = lambda x: x[0] + x[1] + x[2] + x[3] + 0.5*(22202*(x[0]**2) \
              + x[1]**2 + 0.1*(x[2]**2) + 0.001*(x[3]**2))

x0 = np.array([1, 1, 1, 1])

start = time.perf_counter()
opt.minimize(f, x0, method='BFGS')
end = time.perf_counter()
print(f"time: {end-start}")
opt.minimize(f, x0, method='BFGS')
```