Something Mathy

Mikal William Nelson

B.S. Mathematics, University of Minnesota, 2013

Submitted to the graduate degree program in Department of Mathematics and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Arts.

Paul Cazeaux, Chairperson

Committee members

Mat Johnson

Dionyssios Mantzavinos

Yannan Shen

Date defended: July 26, 2021

The Thesis Committee for Mikal William Nelson certifies
that this is the approved version of the following thesis :

Something Mathy

Paul Cazeaux, Chairperson

Date approved: July, 2021

Abstract

Insert Abstract Here

Acknowledgements

I would like to thank all of the little people who made this thesis possible. Sleepy, Dopey, Grumpy, you know who you are.

Contents

List of Figures

List of Tables

Introduction

Chapter 1

Background

1.1 PDE Discretization

Multidimensional topological optimization problems often involve the use of partial differential equations (PDEs) which model the physical properties of the materials involved. Most of these PDEs cannot be uniquely solved analytically, so we turn to numerical methods in order to approximate their solutions. The first step in many of these methods is to discretize our domain; that is, we want to choose some scheme to divide our continuous domain into a finite number of pieces over which we will apply a particular method to approximate solutions to the PDE.

In the SIMP method, the Finite Volume Method is used to discretize and approximate solutions to the heat equation for our heat generating medium. We will introduce the Heat Equation and then proceed to give an overview of the Finite Volume Method.

1.1.1 The Heat Equation

Consider a stationary object of that has heat flowing between its interior regions. The temperature at any point in the interior of the object will depend on the spatial position chosen as well as the time we measure the temperature at that point. Therefore, the temperature (T) at any point in such an object is a function of both space ($\vec{x}$) and time (t) coordinates: $T(\vec{x}, t)$. Physical principles demand that such a temperature function must

satisfy the equation

$$\frac{\partial T}{\partial t} = \nabla \cdot \left(k(\vec{x}) \nabla T \right), \tag{1.1}$$

where ∇ is the gradient operator and the function k represents the thermal diffusivity at a point in our object.

Equation (1.1) is commonly referred to as the Heat or Diffusion Equation. If we were to have a constant thermal diffusivity throughout our object, it would be possible to analytically find a solution to this partial differential equation. However, as in the problems of interest throughout this paper, when k is not constant we must turn to numerical methods to find approximate solutions for the function T.

1.1.2 The Finite Volume Method

For our numerical approximations of PDEs in this paper we used the Finite Volume Method (FVM), which will be described in this section.

As with any other numerical method to solve PDEs, we must first discretize our domain by creating some sort of mesh. One major advantage of the Finite Volume Method is that we have a great amount of freedom in choosing our mesh. In FVM the domain can be discretized into a mesh of arbitrary polygons, but we chose uniform squares in our work to simplify the resulting calculations.

Given a mesh of polygons on our domain Ω with vertices at $\{x_i\} \subset \Omega$, we create a set of *control volumes* around each x_i. The resulting set of control volumes discretize the partial differential equation. The Finite Volume Method has us integrate our PDE over each control volume and then use the Divergence Theorem to convert these volume integrals into surface integrals involving the fluxes across the boundaries of the control volumes. We then approximate those fluxes across the boundaries to calculate approximate solutions to our PDE.

Theorem 1 (The Divergence Theorem)**.** *Suppose that $\mathcal{V}$ is a compact subset of $\mathbb{R}^n$ that has*

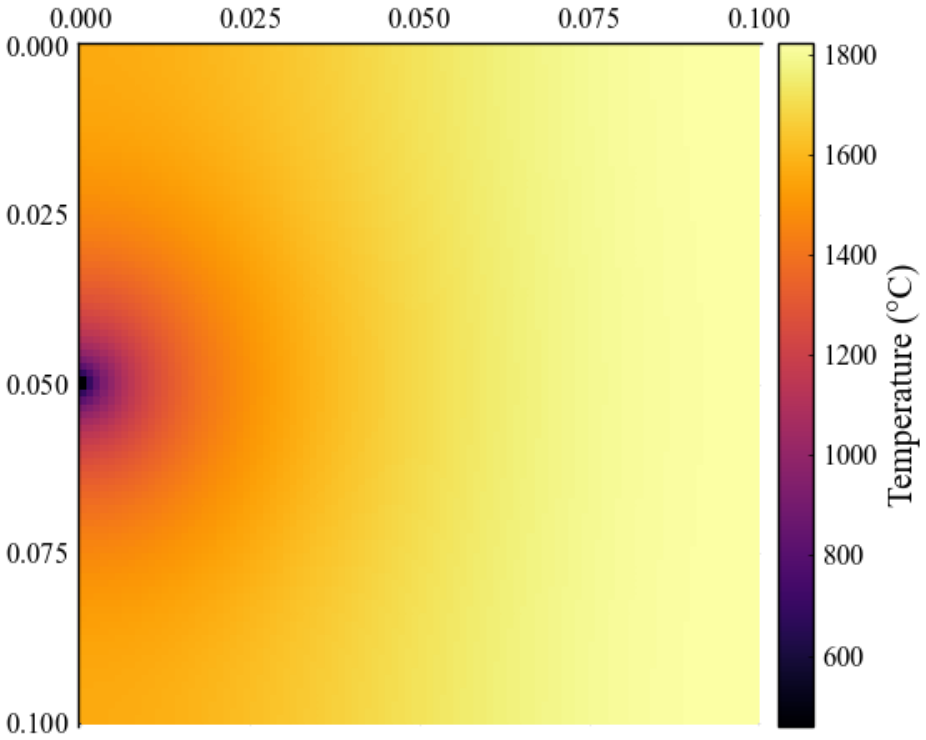

Figure 1.1: Heatmap for a 0.1 m × 0.1 m object with uniform heat generation and a heat sink at the center of its west boundary. This map was produced via the Finite Volume Method using 100 × 100 uniform control volumes.

a piecewise smooth and positively oriented boundary $\mathcal{S}$ (i.e. $\partial\mathcal{V} = \mathcal{S}$). If $\mathbf{F}$ is a continously differentiable vector field defined on a neighborhood of $\mathcal{V}$, then

$$\iiint_{\mathcal{V}} (\nabla \cdot \mathbf{F}) \, \mathrm{d}\mathcal{V} = \oiint_{\mathcal{S}} \mathbf{F} \cdot \mathrm{d}\mathcal{S}. \tag{1.2}$$

The divergence theorem is the key component in the finite volume method because it allows us to look at fluxes across the boundaries of each control volume, rather than the control volume itself.

Let's look at the finite volume method applied to the heat equation in two dimensions. Suppose we have discretized our space by dividing it up into a mesh of control volumes $\{V_i\}$. We integrate the heat equation PDE over each control volume, using the divergence theorem to convert the volume integral into a surface integral:

$$\int_{V_i} \frac{\partial T}{\partial t} \, \mathrm{d}\vec{x} = \int_{V_i} \nabla \cdot \left(k(\vec{x}) \nabla T \right) \mathrm{d}\vec{x} \underset{(1.2)}{=} \int_{\partial V_i} k(\vec{x}) \nabla T \cdot \mathrm{d}s,$$

where s represents the lines that form the boundary of the control volume. Then, applying an approximation scheme to this result, we obtain a sparse and structured linear system.

One other major advantage of the finite volume method is that boundary conditions can easily be taken into account on general domains. For example, adding a heat sink by applying a Dirichlet boundary condition ($T_s = 0\,^\circ\mathrm{C}$) can be thought of as zeroing out our algebraic equations by introducing a ghost cell that, when interpolated with the boundary cell, causes the temperature across the boundary to be zero.

1.2 The Optimization Problem

In much of mathematics, our goal is to seek some sort of solution. In cases where there are multiple solutions, it is desirable to determine the "best" solution judged against some set of criteria. Mathematical optimization is the study of solving such problems. In its simplest case, mathematical optimization is the practice of minimizing or maximizing a given function over a certain set and possibly subject to some constraints.

Definition 1. *An optimization problem (in standard form) has the form*

$$\begin{aligned} &\textit{minimize} && f_0(x) \\ &\textit{subject to} && f_i(x) \leq 0, \quad i = 1, \ldots, m \\ &&& h_i(x) = 0, \quad i = 1, \ldots, p \end{aligned} \tag{1.3}$$

where

- $x = (x_1, \ldots, x_n)$ *are the* ***optimization variables***,
- $f_0 : \mathbb{R}^n \to \mathbb{R}$ *is the* ***objective function***,
- $f_i : \mathbb{R}^n \to \mathbb{R}$ *are the* ***inequality constraint functions***, *and*
- $h_i : \mathbb{R}^n \to \mathbb{R}$ *are the* ***equality constraint functions***.

If there are no constraints ($m = p = 0$), then the problem is called *unconstrained*. (Boyd & Vandenberghe, 2004, p. 127)

We call a vector $x^\star$ **optimal** if it has the smallest objective value among all vectors that satisfy the constraints. That is, for any z with $f_1(z) \leq 0, \ldots, f_m(z) \leq 0$, then $f_0(z) \geq f_0(x^\star)$. A point x that is in the domains of each function f_i and h_i is called **feasible** if it satisfies all the constraints. Finally, the **optimal value** $p^\star$ of the problem is defined as

$$p^\star = \left\{ f_0(x) \mid f_i(x) \leq 0, i = 1, \ldots, m, h_i(x) = 0, i = 1, \ldots, p \right\}.$$

Therefore, $p^\star = f_0(x^\star)$, the objective function value at a feasible, optimal vector $x^\star$.

Notice that the optimization problem in standard form is a minimization problem. We can easily change it into a maximization problem by minimizing the objective function $-f_0$ subject to the same constraints.

The optimization problem is **linear** or called a **linear program** if the objective and constraint functions are all linear. An optimization problem involving a quadratic objective function and linear constraints is **quadratic** or a **quadratic program**. If the optimization problem is not linear or quadratic, it is referred to as a **nonlinear program**.

There are exists efficient methods for solving linear programming and many quadratic programming problems.

1.2.1 Convex Optimization

A set C is **convex** if the line segment between any two points in C lies in C. That is if for any $x_1, x_2 \in C$ and any θ with $0 \leq \theta \leq 1$, we have $\theta x_1 + (1-\theta)x_2 \in C$.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if the domain of f is a convex set and if for all x, y in the domain of f, and θ with $0 \leq \theta \leq 1$, we have

$$f\left(\theta x + (1-\theta)\, y\right) \leq \theta f(x) + (1-\theta) f(y). \tag{1.4}$$

A **convex optimization problem**, therefore, is an optimization problem of the form

$$\begin{aligned} &\text{minimize} && f_0(x) \\ &\text{subject to} && f_i(x) \leq 0, \quad i = 1, \ldots, m \\ &&& a_i^T = b_i, \quad i = 1, \ldots, p \end{aligned} \tag{1.5}$$

where $f_0, \ldots, f_m$ are convex functions.

There are three additional requirements that differentiate a convex optimization problem from a general optimization problem:

- the objective function must be convex
- the inequality constraint functions must be convex
- the equality constraint functions must be **affine**

In a convex optimization problem we minimize a convex objective function over a convex set.

In a convex optimization problem, any locally optimal point is also globally optimal. This is a *very* useful fact!

Why might local optimality implying global optimality be useful? Consider a situation where we have a convex objective function. If we are able to find any minimum value for the objective function, then we know this value is not just a local minimum, but indeed a global minimum. If we do not have a convex function, we cannot be as assured that we have found the global optimal value. For example, consider a function such as the one shown in Figure 1.2. An optimization strategy may find the local minimum near $x = 1$, but since the function is not convex everywhere on its domain, we cannot conclude that this value is the truly optimal value. In fact, we see that the global minimum, and hence the actual optimal value, is between $x = -1$ and $x = -0.5$. On the other hand, if we have a function that is everywhere convex, as soon as we find a local minima, we can be assured that it is also the global minima!

So we can see that convexity is a very powerful and useful property in terms of optimization problems. As a result, any time we can take advantage of convexity or approximate functions using convex functions, we often do so.

1.3 Optimization Methods

In this section I will present a few optimization algorithms.

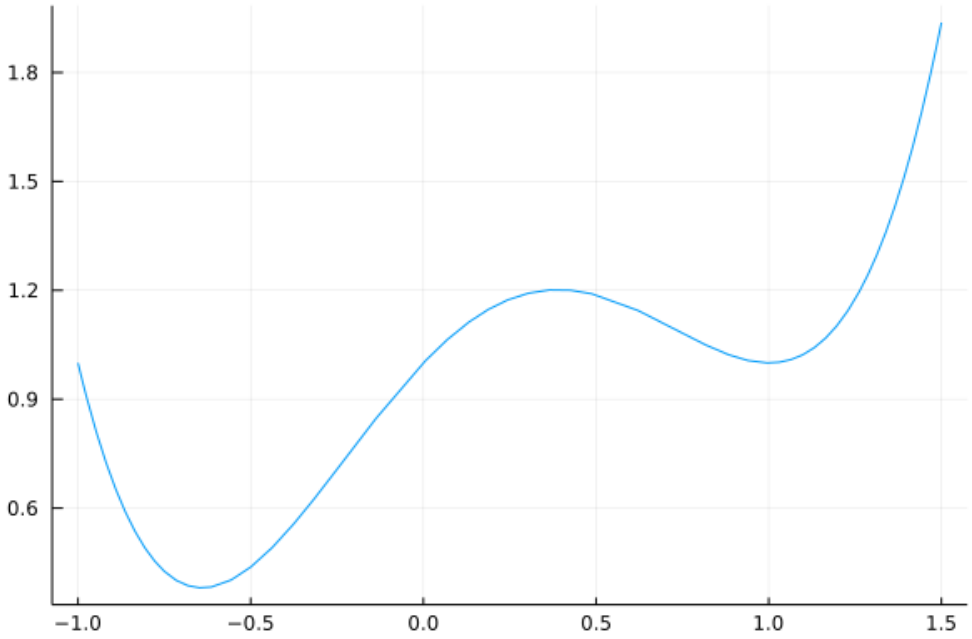

Figure 1.2: A graph of the non-convex function $f(x) = x^4 - x^3 - x^2 + x + 1$. Notice it has two local minima and that the right local minima is not equal to the global minimum.

1.3.1 Line Search Methods

The **line search** is a strategy that selects the step size (commonly represented by t) that determines where along the line $\{x + t\Delta x \mid t \in \mathbb{R}_+\}$ the next iterate in the descent method will be. (Δx represents the *descent direction*, .) Line search strategies can either be *exact* or *inexact*.

Exact Line Search

An **exact line search** chooses the value t along the ray $\{x + t\Delta x \mid t \in \mathbb{R}_+\}$ that exactly minimizes the function of interest f:

$$t = \arg\min_{s \geq 0} f(x + s\Delta x)$$

An exact line search is almost never practical. In very special cases, such as some quadratic optimization problems, where computing the cost of the minimization problem

is low compared to actually calculating the search direction, one might employ an exact line search.

Backtracking Line Search

Most often in practice we use **inexact line searches**. In an inexact line search, we choose t such that f is *approximately* minimized or reduced "enough" along $\{x + t\Delta x \mid t \in \mathbb{R}_+\}$.

One inexact line search strategy is the **Backtracking Line Search**.

Algorithm 1 Backtracking Line Search (Boyd & Vandenberghe, 2004)

given a descent direction Δx for f at $x \in \mathbf{dom} f$, $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$.

$t := 1$.

while $f(x + t\Delta x) > f(x) + \alpha t \nabla f(x)^T \Delta x$ **do**

$t := \beta t$

end while

"Backtracking" in the name is refers to the fact that the method starts with a unit step size ($t = 1$) and then reduces the step size ("backtracks") by the factor β until we meet the stopping criterion $f(x + t\Delta x) \leq f(x) + \alpha t \nabla f(x)^T \Delta x$.

Figure 9.1 (Boyd & Vandenberghe, 2004, p. 465) demonstrates the Backtracking Line Search visually for a parabola.

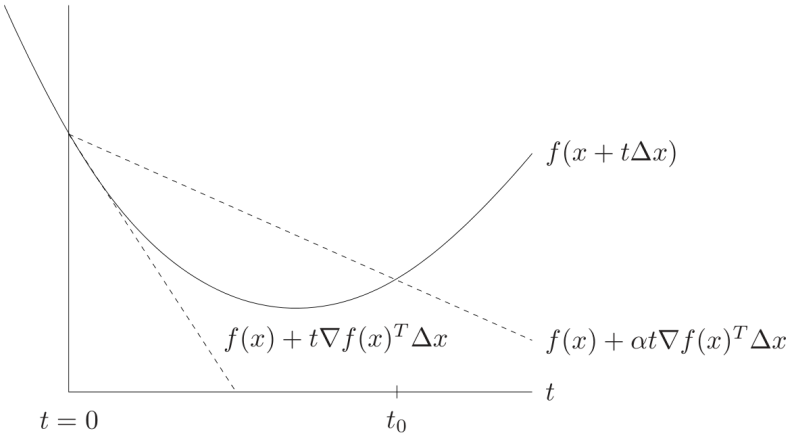

Figure 9.1 *Backtracking line search.* The curve shows f, restricted to the line over which we search. The lower dashed line shows the linear extrapolation of f, and the upper dashed line has a slope a factor of α smaller. The backtracking condition is that f lies below the upper dashed line, *i.e.*, $0 \leq t \leq t_0$.

Notice that the backtracking search will find a step size t such that $0 \leq t \leq t_0$ and that for such t, $f(x + t\Delta x)$ is smaller relative to $f(x)$. However, the step size we choose may not exactly be the minimum of the function, but we have funneled it down to be closer to the minimum of f.

1.3.2 Gradient Descent

The gradient descent method chooses the search direction to be the negative gradient. That is, in this method we set $\Delta x = -\nabla f(x)$, where f is the function we seek to optimize. Since the gradient of a function gives the direction of greatest increase, naturally the negative gradient will give the direction of the most rapid decline.

Algorithm 2 Gradient Descent Method (Boyd & Vandenberghe, 2004)

given a starting point $x \in \mathbf{dom} f$.
repeat

1. $\Delta x := -\nabla f(x)$.
2. *Line search.* Choose step size t via exact or backtracking line search.
3. *Update.* $x := x + t\Delta x$.

until stopping criterion is satisfied.

Notice in Algorithm 2 that essentially a line search is used to determine a step size and then we update our iterate in the direction of the steepest descent. This is repeated until we meet some sort of stopping criterion, typically something of the form $\|\nabla f(x)\|_2 \leq \eta$, where η is small and positive. Another common stopping criterion is to stop the algorithm when no significant progress is made between iterates by stopping the algorithm when $\|f_{k+1} - f_k\| < \eta$.

One thing that is interesting to note about the gradient descent algorithm is that the function f is never actually evaluated, only the gradient of f.

An implementation of the gradient descent algorithm in the Julia language can be found in the appendix.

1.3.3 Nonlinear Conjugate Gradient

The Nonlinear Conjugate Gradient method works similarly to the gradient descent algorithm, but adds the additional requirement that in each iteration gradient is orthogonal to each previous search direction.

Suppose we have a function $f(x)$ of N variables. Let x_0 be an initial guess value for the minimum. The opposite of the gradient will give the direction of steepest descent. Therefore, we start off by setting $\Delta x_0 = -\nabla f(x_0)$.

We set an adjustable step length α and perform a line search in the direction $d_0 = \Delta x_0$ until a minimum of f is reached:

$$\alpha_0 := \arg\min_{\alpha} f(x_0 + \alpha \Delta x_0),$$
$$x_1 = x_0 + \alpha_0 \Delta x_0.$$

Suppose we were to simply iterate this process and for each step i we do the following:

1. Set $\Delta x_i = -\nabla f(x_i)$

2. Calculate $\alpha_i = \arg\min_{\alpha} f(x_i + \alpha \Delta x_i)$

3. Compute $x_{i+1} = x_i + \alpha_i \Delta x_i$

However, there is an issue with this proposed iterative scheme: We have moved α_i in direction Δx_i to find the minimum value in that direction, but by moving α_{i+1} in direction Δx_{i+1} we may have accidentally *undone* the progress made in the previous iteration so that we no longer have a minimum value in direction Δx_i. We can fix this problem by making sure that successive direction vectors have no influence in the directions of previous iterations. That is, we require our directions in each iteration to be conjugate (with respect to the matrix of coefficient for our system) to one another. Therefore, rather than taking Δx_{i+1} to be $-\nabla f(x_i)$, we compute a direction conjugate to all previous directions by some pre-chosen methodology. This suggests the following iterative scheme:

After the first iteration, the following steps constitute one iteration along a conjugate direction:

1. Calculate the new steepest descent direction: $\Delta x_i = -\nabla f(x_i)$,

2. Compute β_i using some formulation. Two options are below:

   - Fletcher–Reeves: $\beta_i^{FR} = \frac{\Delta x_i^T \Delta x_i}{\Delta x_{i-1}^T \Delta x_{i-1}}$
   - Polak–Ribière: $\beta_i^{PR} = \frac{\Delta x_i^T (\Delta x_i - \Delta x_{i-1})}{\Delta x_{i-1}^T \Delta x_{i-1}}$

3. Update the conjugate direction: $d_i = \Delta x_i + \beta_i d_{i-1}$,

4. Line search: Optimize $\alpha_i = \arg\min_{\alpha} f(x_i + \alpha d_i)$,

5. Update iterate value: $x_{i+1} = x_i + \alpha_i d_i$.

Algorithm 3 uses the Newton-Raphson method to find the values of α_i.

To get some intuition of how this algorithm operates, let's look at it applied to a quadratic function.

Algorithm 3 Nonlinear Conjugate Gradient Using Newton-Raphson (Shewchuk, 1994)

Given a function f, a starting value x, a maximum number of CG iterations $i_{\max}$, a CG error tolerance $\epsilon < 1$, a maximum number of Newton-Raphson iterations $j_{\max}$, and a Newton-Raphson error tolerance $\varepsilon < 1$:

$i \Leftarrow 0$
$k \Leftarrow 0$
$r \Leftarrow -f'(x)$
$d \Leftarrow r$
$\delta_{\text{new}} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{\text{new}}$
while $i < i_{\max}$ and $\delta_{\text{new}} > \epsilon^2 \delta_0$ **do**
    $j \Leftarrow 0$
    $\delta_d \Leftarrow d^T d$
    while true **do**
        $\alpha \Leftarrow -\frac{[f'(x)]^T d}{d^T f''(x) d}$
        $x \Leftarrow x + \alpha d$
        $j \Leftarrow j + 1$
        $j < j_{\max}$ and $\alpha^2 \delta_d > \varepsilon^2$ OR **Break**
    end while
    $r \Leftarrow -f'(x)$
    $\delta_{\text{old}} \Leftarrow \delta_{\text{new}}$
    $\delta_{\text{new}} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{\text{new}}}{\delta_{\text{old}}}$
    $d \Leftarrow r + \beta d$
    $k \Leftarrow k + 1$
    if $k = n$ or $r^T d \leq 0$ **then**
        $d \Leftarrow r$
        $k \Leftarrow 0$
    end if
    $i \Leftarrow i + 1$
end while

Example: Rosenbrock Function

1.4 The Method of Moving Asymptotes (MMA)

Chapter 2

SIMP Optimization

2.1 Solid Isotropic Material with Penalization (SIMP)

References

Boyd, S. P. & Vandenberghe, L. (2004). *Convex optimization*. Cambridge Univ. Pr.

Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain.

Appendix A

Julia Codes

A.1 Backtracking Line Search

Here is an implementation of the Backtracking Line Search in Julia with default values for the parameters being $\alpha = 0.25$ and $\beta = 0.5$.

```
function ln_srch(d_dir,x,f,fx,dfx;alpha=0.25,beta=0.5)
t = 1
x1 = x+t*d_dir
y1 = f(x1)
y2 = fx+alpha*t*(dfx)'*d_dir
while y1 > y2
t = beta*t
x1 = x+t*d_dir
y1 = f(x1)
y2 = fx+alpha*t*(dfx)'*d_dir
end
return t
end
```

A.2 Gradient Descent

```
using LinearAlgebra


#Function to Optimize
```

```
f(x)=(x[2])^3-x[2]+(x[1])^2-3x[1]

#Gradient of Function
df(x)=[2x[1]-3,3x[2]^2-1]

#Initial Point
x=[0,0]

#Gradient Descent Algorithm
function grad_d(f,df,x)
d_dir = -df(x)
t = ln_srch(d_dir,x,f,f(x),df(x))
x = x + t*d_dir
return x
end

#Compute Minimum for Defined Tolerance
while norm(df(x))>0.00001
global x = grad_d(f,df,x)
end

display(x)
```