# Control perspective on distributed optimization

Sam Farkhooi

2023

## Abstract

In the intersection between machine learning, artificial intelligence and mathematical computation lies optimization. A powerful tool that enables us to solve a variety of large scale problems. The purpose of this work is to explore optimization in the distributed setting. We will then touch on factors that contribute to a faster and more stable algorithm while solving a distributed optimization problem. The main factor we will look into is how we can integrate control.

## Preface

This work is a bachelor thesis in mathematics worth 15 credits. All of the coding is done in the programming language Python using Visual studio code. I want to thank my supervisor Sérgio Pequito whose help has been invaluable for this work. I also want to thank my subject reviewer Lauri Viitasaari.

# Contents

# 1 Introduction to distributed optimization

Throughout the recent years there have been massive advancements around the world in fields such as artificial intelligence and machine learning. This explosion of improvements has brought with it the need for solving a variety of large scale problems. A machine learning algorithm tries to find the best possible model to a dataset. There are several ways to approach this task but it all boils down to the same core idea [8].

When we talk about best possible model we have to define what makes a model good in order to compare and find the best possible one. When we find a model we evaluate its performance by looking at the error it yields [6]. Consider a linear regression problem solved using machine learning. Here, we start with a number of training data points which we use to fit a line to predict future outputs.

A model here means a candidate for which regression line to be used. What we call error is the difference between the predicted output produced by the machine and the true output revealed by the user. The best possible model is defined as the model which yields the smallest error. This means that the problem of finding the best possible model is the same as finding a model while solving the sub-problem of minimizing the error.

This can be expressed mathematically as finding an objective function $f : \mathbb{R}^n \to \mathbb{R}$ which is a function of the error. For the linear regression case the function would be of the form $f(x) = y(x) - g(x)$. Here $y(x)$ is the observed output from the user and $g(x) = x^T\theta+\gamma$ is the predicted output by the machine, where $\theta$ is the parameter space and $\gamma$ is a constant. Now the goal is to find a model while finding $\min_{x\in\mathbb{R}^n} f(x)$.

The process of finding the best solution by minimizing such an objective function is what we call optimization. Most machine learning problems can in this way be expressed as optimization problems so mathematically optimization lies in the foundation of machine learning.

The reason we call it the best possible model and not simply the best model is because sometimes it might not be feasible to find the actual best model. It might only be possible to consider a subset of all models, then the goal is to find the best one among those. Optimization comes in a few different forms and we are going to cover two types, centralized and distributed. The heaviest focus in this paper will be on the latter.

## 1.1 Centralized optimization

Consider a network of connected entities, called agents, where each agent $i$ has access to a local function $f_i : \mathbb{R}^n \to \mathbb{R}$ and the goal is to minimize the objective function $f = \sum_i^M f_i$. One way to tackle this is to have one "main agent" which has access to each of the local functions and hence the sum of the functions too. The "main agent" then has all the information needed to optimize the network. This is what is called centralized optimization [7].

In Figure 1, we have illustrated how centralized optimization looks in a simple graph. The red node is the "main agent" which can collect data from all the other agents, which are the smaller gray nodes.

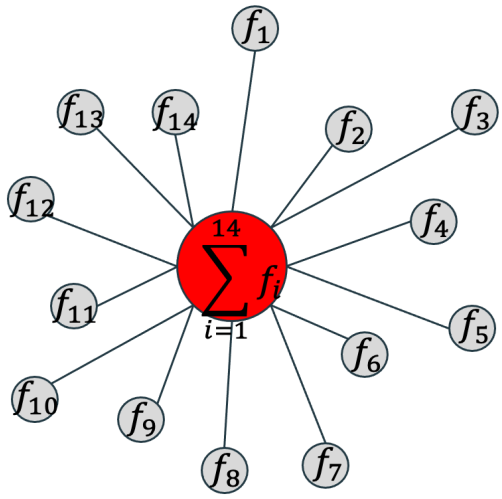


Figure 1: A simple network where centralized optimization can be applied.

Some of the strengths of having a system using centralized optimization is that it is fast, easy to set up and has a low maintenance cost. The optimization is fast since the node with all the information can optimize the system effectively. The low maintenance cost comes from the fact that the system is run through a single main server.

Some of the weaknesses of having such a system is that there is a low fault tolerance and the throughput is limited. The low fault tolerance is a consequence of the fact that the whole system fails if the main node fails. The main node thus acts as a single point of failure. The throughput in the system is limited by how much processing power the main server has.

## 1.2 Distributed optimization

As centralized optimization is relatively fast and easy it is desirable to have a network using it. However, it is not always possible to make decisions in a centralized manner. Sometimes having information about how the agents are connected may be required. It can also be that the optimization problem is so substantial that the processing power of the main node is too limiting to perform the operations effectively.

As an answer to these obstacles we introduce distributed optimization [12]. The idea, as the name suggests, is to distribute the optimization process over

multiple entities. So in the network there is no main node with all the information to perform the optimization. Instead each node holds part of the information and all the nodes need to communicate in order to see the bigger picture.

Again the global objective is to minimize the objective function $f = \sum_i^M f_i$ where $f_i$ is the local function available to node $i$. Now, the approach is to let each node $i$ have the local objective to minimize their own function $f_i$ and then the nodes communicate – to find the shared solution. This way the workload is shared among the nodes. So, instead of directly solving the problem of optimizing $\sum_i^M f_i$, we divide it into smaller sub-problems and solve each of them.

Figure 2 illustrates distributed optimization in a network. Note that all pairs of nodes are at least indirectly connected so the information is shared through the whole network.

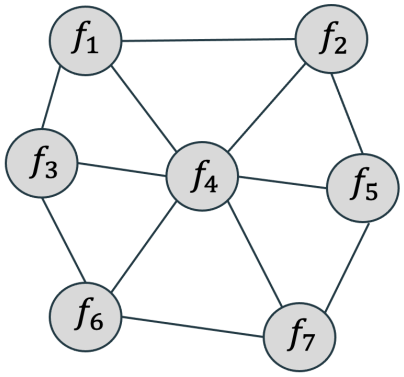


Figure 2: A simple network of 7 connected agents where distributed optimization can be applied.

An advantage of having a system using distributed optimization is that there is a high fault tolerance with no single point of failure. Another advantage is that the workload is shared among the nodes which prevents overloading of one machine.

Some of the disadvantages compared to using a network with centralized optimization is that it is more complex. Also the nodes need to cooperate to reach the same values to optimize the objective function. Another disadvantage is that using multiple nodes to perform the optimization comes with a higher maintenance cost [7].

An important thing to note about distributed optimization is that the nodes need to reach a consensus for the global solution. This is done by the nodes tracking the state of its neighbors.

In this paper we are going to dive deeper into distributed optimization. We

will solve a distributed optimization problem with coding and discuss many of the factors that impact the speed and stability of the algorithm.

# 2 Terminology

In this section, we go through how the following terminology will be used throughout the paper. Let $f : \mathbb{R}^n \to \mathbb{R}$ be an arbitrary – convex function[10].

$x^{opt}$: $x^{opt} \in \mathbb{R}^n$ is the value such that $f$ attains its global minimum for $f(x^{opt})$.

$x^*$: $x^* \in \mathbb{R}^n$ is an approximated value of the optimal value $x^{opt}$.

$[x_1 \ \dots \ x_n]^T$: A column vector in $\mathbb{R}^n$, our usage of $T$ denotes the transpose.

$x(k)$: In an algorithm this will denote the value of $x \in \mathbb{R}^n$ at iteration $k$.

$G_f(x)$: The gradient of the function $f$ at the point $x$.

$\mathcal{N}_i$: The set containing all neighbors to node $i$ in a graph.

Setpoint: The desired output value of a process.

# 3 Numerical methods for optimization

Explicitly finding the solution to an objective function in a network can be a huge task. To calculate the sought optimal value is often not feasible at all. That is because, in large networks we have no proper way to solve a distributed optimization problem. This is why we instead use numerical methods for optimization. So, instead of explicitly finding the solution we settle for numerically approximating the solution.

There are several different numerical methods such as coordinate descent, Newton-Raphson and the conjugate gradient method [4]. The numerical method that we will make use of in this paper is one of the most commonly used in machine learning called gradient descent.

## 3.1 Gradient descent

Gradient descent is a method used, for a function $f$, to find $\min_{x\in\mathbb{R}^n} f(x)$. Generally speaking, the method for finding a minimum is to initiate at a starting point and compute the gradient in that point to see where the function is increasing the most and then take a step in the opposite direction. The algorithm is iterative, and finds the next approximation based on the current one in the following way: [6]

$$x^*_{k+1} = x^*_k - \alpha G_f(x^*_k). \tag{1}$$

This means, a step is taken in the direction where the function is decreasing the most. Then, one iteration has been made and a new approximation is obtained. The process is then repeated from the new point and iterations occur until some stopping criterion is met. Like the distance between two consecutive points being smaller than some constant i.e $|x^*_{k+1} - x^*_k| < \epsilon$ for some small constant $\epsilon$. When this occurs the last point obtained is the approximated minimum point of the function.

Here, we call $\alpha$ the learning rate. It is a parameter $\alpha \in \mathbb{R}$ such that $0 < \alpha < 1$ which is multiplied with the gradient in each iteration to make the step of adequate length. In Figure 3, we have performed gradient descent in the simple case, where the function is $f(x) = x^2$ and the starting point is $x^*_0 = 5$ with $\alpha = 0.2$. Each red dot represents an approximation $x^*_k$. For this function it is clear that $x^{opt} = 0$. We will now show how we implemented the algorithm, which performs 1-dimensional gradient descent, in pseudo-code.

**Algorithm 1** 1-dimensional gradient descent

**Input:** Initial $x$, function $f : \mathbb{R} \to \mathbb{R}$, learning rate $\alpha$, constant $\epsilon$
**Output:** List containing the $x^*$ obtained in each step.

Set $x\_history \leftarrow [x]$
**while** len($x\_history$) == 1 **or** $|x^*_{i+1} - x^*_i| > \epsilon$ **do**
    Update $x \leftarrow x - \alpha f'(x)$
    Update $x\_history$.append($x$)
**end while**
**return** $x\_history$

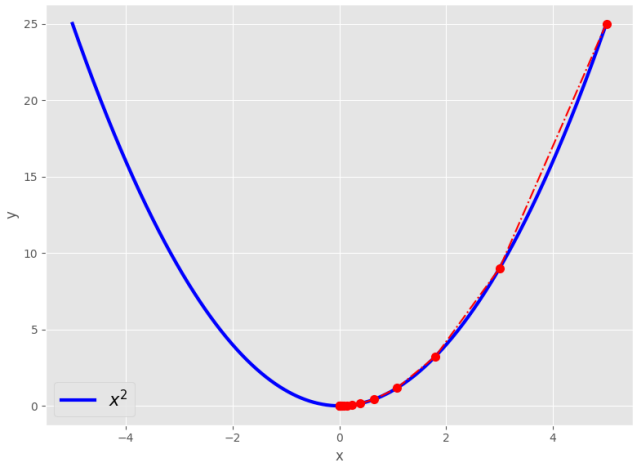


Figure 3: Gradient descent to find the minimum of the function $f(x) = x^2$.

## 3.2 Adjusting the learning rate

However, the application of this method is not always straightforward. We will only be studying – convex functions, where the objective is to find the global minimum. But as mentioned the method relies on taking steps in the direction towards the minimum, and the step size is regulated by the learning rate $\alpha$. So the problem of finding an appropriate step size arises i.e finding a suitable value for $\alpha$.

For example, in Figure 3, we used the learning rate $\alpha = 0.2$ which is a good choice since the minimum is found and the algorithm is not too slow, the process stops after 16 iterations. But, if we lower–$\alpha$ to 0.01, we can see the problem that occurs when the learning rate is too small. Namely, the algorithm takes unnecessarily long to complete the task. In this case, it takes 229 iterations to achieve what we could easily achieve in 16 iterations before, – see Figure 4.

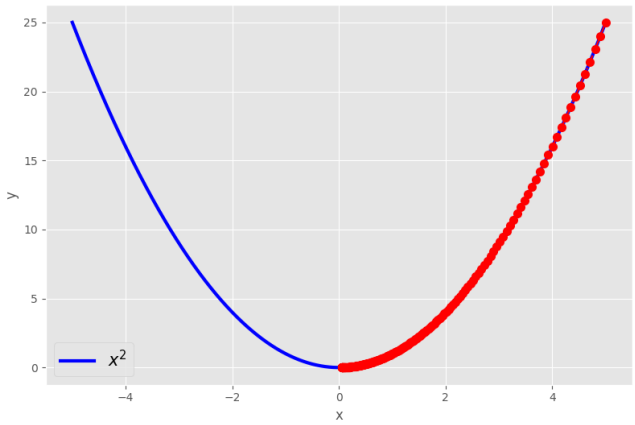


Figure 4: Gradient descent where the learning rate was decreased from 0.2 to 0.01.

Finally, we illustrate the problem with the learning rate being too large. What can happen is that the steps taken are so large that we go past the minimum. This creates an oscillating behavior where you jump back and forth past the desired point. This can either lead to a slower convergence because there is less progress towards the minimum. If–$\alpha$ is too large, the method might not even converge at all.

In Figure 5, we have illustrated the behavior with a higher learning rate, $\alpha = 0.95$. Here, we can see that even though the learning rate is higher, it still converges slower than the case where $\alpha = 0.2$. We still have convergence, however – this time the algorithm stops after 89 iterations. If we had for example tried $\alpha = 1$, then we would oscillate between the approximations $x = 5$ and $x = -5$ without converging.

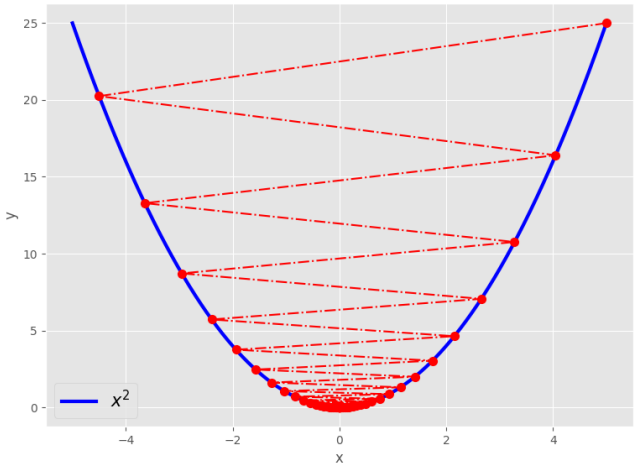


Figure 5: Gradient descent with the learning rate $\alpha = 0.95$ gives an oscillating behavior.

### 3.3 The Wolfe conditions

How do we know which values for–$\alpha$ are good and how do we choose them? This is where the so–called Wolfe conditions come into play. They are two conditions for $\alpha$ which, if both satisfied, yield a satisfactory $\alpha$. This check for a satisfactory $\alpha$ is done in each iteration of the optimization. So the goal is to have a good step size for every single step and for each iteration $k$ we have a different $\alpha_k$. The first condition: [1]

$$f(x_k - \alpha_k G_f(x_k)) \leq f(x_k) - c_1 \alpha_k G_f(x_k). \tag{2}$$

The intuition behind it is that if we choose a large value for $\alpha_k$ the value has to decrease enough to justify the step length. The constant $c_1$ is chosen such that $0 < c_1 < 1$ and the closer to zero it is the more lenient the condition becomes. The second condition: [2]

$$\frac{G_f(x_k - \alpha_k G_f(x_k))}{G_f(x_k)} \leq c_2. \tag{3}$$

So the second condition requires that the gradient after taking the step must have decreased by a factor $c_2$. The constant $c_2$ is chosen such that $0 < c_1 < c_2 < 1$ and the condition is more strict as $c_2$ gets closer to 0. The first condition limits $\alpha_k$ from above and–hence–ensures that the step taken is not too large.

On the contrary, the second condition functions as a lower limit and makes sure $\alpha_k$ is not too small. Therefore, if both conditions are satisfied – we have a step size which is not too large and not too small.

In Figure 6 – we illustrate both conditions for the same gradient descent problem as we have covered so far. Specifically – we are looking at the first iteration, so we will get a lower and upper bound for $\alpha_0$. Remember that we have that $x_0 = 5$. To make the plot easier to follow we denote the terms the following way:

$$f_1^k := f(x_k - \alpha_k G_f(x_k)), \qquad \gamma_1^k := f(x_k) - c_1 \alpha_k G_f(x_k),$$

$$f_2^k := \frac{G_f(x_k - \alpha_k G_f(x_k))}{G_f(x_k)} \text{ and } \gamma_2^k := c_2$$

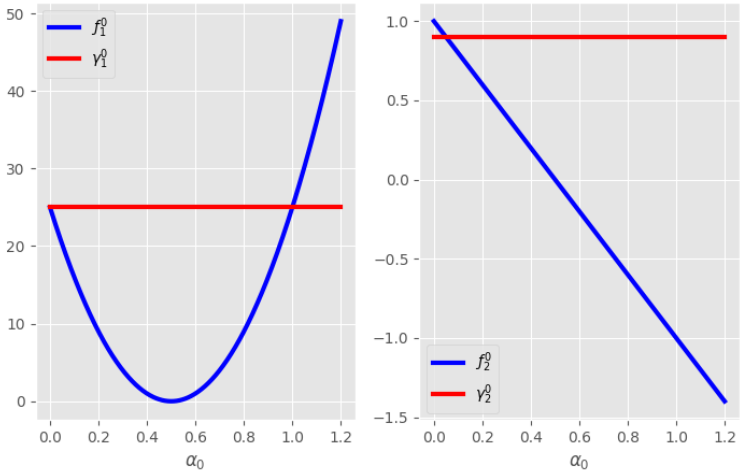


Figure 6: The left plot shows the first Wolfe condition, which is satisfied when $f_1^0 \leq \gamma_1^0$. The right plot shows the second condition, which is satisfied when $f_2^0 \leq \gamma_2^0$.

In this example, we have used $c_1 = 0.0001$ and $c_2 = 0.9$. Note that $c_1$ corresponds to the negative slope of the red line in the left plot, albeit a very small slope. The constant $c_2$ corresponds to the value of the red line in the right plot. What we can gather from this is that any $\alpha_0$ such that $0.05 < \alpha_0 < 0.99999$ fulfills both conditions.

Imposing those conditions do not make any significant restrictions for $\alpha_0$, – however, say that we have taken $k$ steps and have that $x_k = 0.0002$. – The conditions for $\alpha_k$ in this case are illustrated in Figure 7. In this iteration the Wolfe conditions are satisfied if $0.05 < \alpha_k < 0.75$.

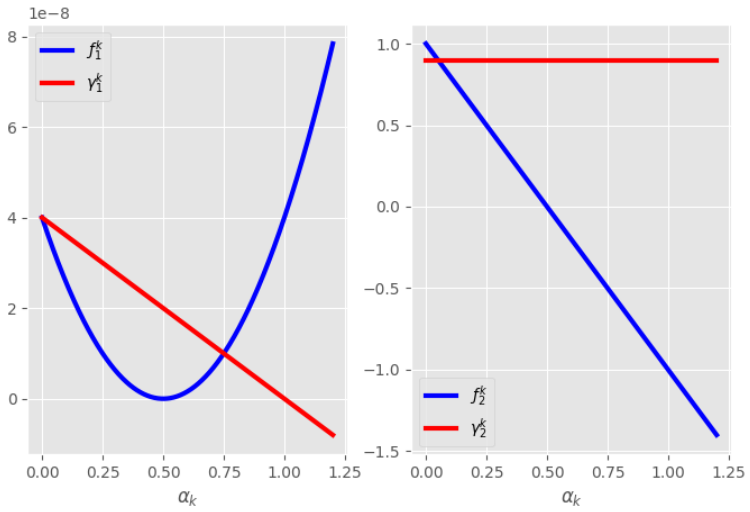


Figure 7: As $x_k$ has reached a smaller value the first condition becomes stricter compared to Figure 6. The slope of $\gamma_1^k$ becomes more prevalent as $x_k$ decreases.

So now, we will make an attempt to use the Wolfe conditions to solve the optimization faster than what we did before using $\alpha = 0.2$. Algorithm 2 shows how we integrated the Wolfe conditions in the gradient descent algorithm.

**Algorithm 2** Gradient descent with the Wolfe conditions

**Input:** Initial $x$, function $f : \mathbb{R} \to \mathbb{R}$, constant $\epsilon$
**Output:** List containing the $x^*$ obtained in each step.

Set $x\_history \leftarrow [x]$
**while** len($x\_history$) == 1 **or** $|x_{i+1}^* - x_i^*| > \epsilon$ **do**
    Set $a \leftarrow$ the value of $\alpha_k$ such that $f_1^k = \gamma_1^k$ with $a \neq 0$
    Set $b \leftarrow$ the value of $\alpha_k$ such that $f_2^k = \gamma_2^k$
    Set $\alpha \leftarrow \frac{a+b}{2}$
    Update $x \leftarrow x - \alpha f'(x)$
    Update $x\_history$.append($x$)
**end while**
**return** $x\_history$

So here, we have chosen to take the mean value for $\alpha_k$ in each iteration so in the example we just mentioned we would get $\alpha_0 = \frac{0.99999+0.05}{2} \approx 0.525$ and $\alpha_s = \frac{0.75+0.05}{2} = 0.400$. Using this we did not even have to choose a learning rate, other than choosing how lenient the constants $c_1$ and $c_2$ should be.

The result was that the algorithm required much fewer steps as the approximation was completed after 4 iterations – see Figure 8. However, now the algorithm requires time to compute $\alpha_k$ for every iteration so there is a trade-off

between how many iterations are needed and how long it takes to make the iterations. In this case the algorithm actually takes longer to converge than it did with the fixed $\alpha = 0.2$.

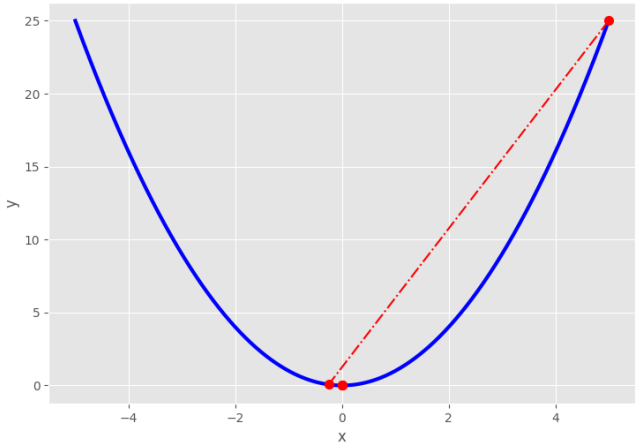


Figure 8: Gradient descent using the Wolfe conditions to determine $\alpha$ in each iteration.

### 3.4 Adjusting the stopping condition

Another parameter that can affect the result of the method is the stopping condition, which is a constant $\epsilon \in \mathbb{R}$. As we can see in line 2 of Algorithm 1, a new step is taken as long as the absolute value between the two last approximations is greater than $\epsilon$. So, when we say that the algorithm has "converged" – we mean that taking additional steps from that point will not make a significant difference to $x^*$. In all cases above the stopping condition is $\epsilon = 0.001$.

The consequences of not choosing a proper $\epsilon$ are not as dire as they are for $\alpha$, but it is still worth giving consideration. If $\epsilon$ is too large, then the algorithm will be satisfied too easily and stop iterating while $x^*$ is still far from $x^{opt}$. On the other hand, if $\epsilon$ is too small it slows down the process since the algorithm has to take more steps before the stopping condition is met.

For both $\alpha$ and $\epsilon$ there are no general values, instead they must be chosen with both the function and the starting point in mind. In this paper, we will be looking into the problem of finding these parameters, focusing mostly on $\alpha$. We will also make an attempt at solving gradient descent problems with adaptive parameters.

### 3.5 Higher dimensions

So far we have covered gradient descent for functions $f : \mathbb{R} \to \mathbb{R}$, but the method is often applied for higher dimensional cases. For example for functions of the form $f : \mathbb{R}^n \to \mathbb{R}$. In this paper, we will limit ourselves to the case $f : \mathbb{R}^2 \to \mathbb{R}$. Fundamentally, it works the same way, so it is important to have an understanding of the lower dimensional cases before increasing dimensions.

To introduce the higher dimension setting, we will again consider a simple case where $f(x_1, x_2) = x_1^2 + x_2^2$. Clearly $x^{opt} = [0 \quad 0]^T$ and in this case the solution will be found by updating both $x_1^*$ and $x_2^*$ in every iteration. Instead of plotting the function as before – we will now use a more general strategy where we plot one curve for $x_1^*$ and one curve for $x_2^*$ – see Figure 9. This way of plotting works for visualisation in any dimension.

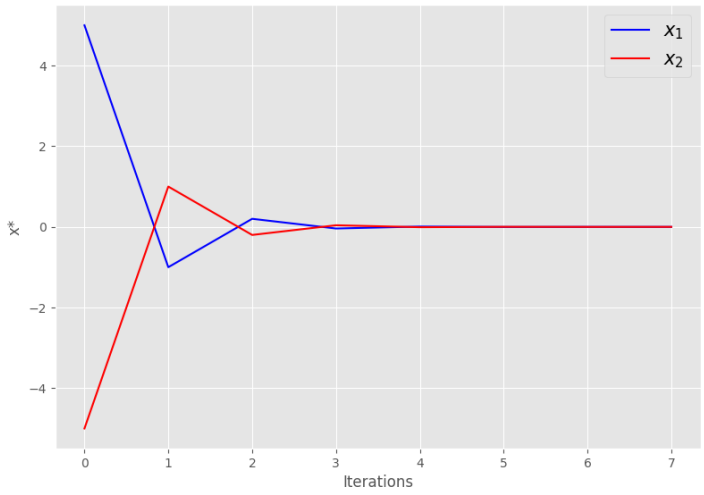


Figure 9: Using gradient descent to approximate the global minimum of $x$. Here $\alpha = 0.6$, $\epsilon = 0.001$ and the initial condition is $x = [5 \quad -5]^T$.

# 4 Proportional integral control

## 4.1 PI control

Optimization is done iteratively, in order to improve the result in each iteration step, the error term needs to be driven to zero. To help us achieve that, we introduce what is called control. One of the most commonly used control strategies today is called proportional integral control. This strategy is what we will be using throughout this work [5].

To understand what control is in general, let us start by considering a problem solved without using control. Suppose we have created a self–driving car that can drive a specific route from house $A$ to house $B$. If this is solved by calculating a route based on the distance and the car's speed then the drive would be successful every time in theory. However – this might not always be the case in practice.

Imagine that the air pressure is lower than expected in one of the tyres causing the car to drift in that direction. Since the car has only been programmed to drive a certain speed in a predetermined direction it will not make adjustments for this drift to the side and will drive off the road. Another thing that could go wrong could be just slightly overestimating the speed of the car which could cause it to stop too early and not reach house $B$.

In this case, it would be of great benefit to handle these errors using control. It would allow the errors to be countered while the system is running, i.e while the car is driving. The car would use a feedback system to notice that it is going off the road or too slow and the control scheme would describe how to handle these error terms. The goal of the control scheme is to make the total error go toward zero with time.

So how is this achieved using PI control? The strategy consists of two components, the proportional term and the integral term. These serve as a function of the error at each time step. If we let $x_t \in \mathbb{R}^2$ be the car's position at time $t$, with $x_0 = A$ and $x^{opt} = B$. Then the error at time $t$, denoted by $l_t$, would be $l_t = x_t - B$. The function of the error $h(l_t) : \mathbb{R}^2 \to \mathbb{R}^2$ is given by

$$h(l_t) = \underbrace{k \cdot l_t}_{P} + \underbrace{\int_0^T l_t}_{I}. \tag{4}$$

Proportional control (corresponding to the $P$ term) is the simplest form of control. It reacts immediately to the current error and makes an effort to combat it. The larger the current error is, the greater the action taken to reduce it will be. However, proportional control lacks in setpoint accuracy and, therefore, does not alone achieve the desired result.

Improving the setpoint accuracy is where the integral $I$ term comes in. As the name suggests, integral control tracks the error as an integral and, hence, aims to reduce smaller but constant error. The integral term will not react as quickly to the immediate error as the proportional term. Instead it will help achieve better accuracy long term, since the integral of a small error will build up to a greater value. The integral term will then put more weight into handling that error.

Relating this to the example above, when the car started veering off the road the proportional term would immediately react. It would then turn this error into a command and start to steer the car back towards the road. The integral term would afterwards play a larger role in fine adjusting the car to be perfectly on the road again after some time. Similarly, if the car is too slow the proportional term would speed up the car and the integral term would make sure that the speed reaches the setpoint value.

This is why control is important since in practice there will almost always be error terms that arise and control serves as a tool for handling these. In Figure 10, we show how the two terms in PI control reacts to the same error.

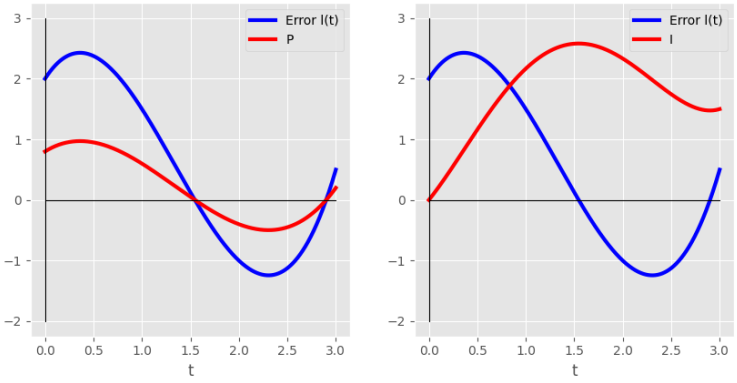


Figure 10: Illustration of the proportional and the integral term – in PI control.

Note that the proportional term reacts faster to the large initial error and the change in value of the error. On the other hand, the integral term is only going to be 0 when the area above the $x$-axis is the same as the area below the $x$-axis. This is how the proportional term handles larger errors fast and how the integral term does the fine tuning to achieve the setpoint value.

## 4.2 Gradient descent using PI control

To show how this can be integrated into optimization – we will now perform centralized optimization using PI control. When we are looking at functions that are convex and differentiable the gradient serves as a function of the error. This is because $G_f(x^{opt}) = 0$ and the gradient increases with the error.

This means that the normal algorithm for gradient descent is already using the proportional term. So the algorithm we have been using so far has been proportional control (P control). With this in mind, we update the algorithm to also include the integral term. The following algorithm show how we will do gradient descent with PI control:

$$x^*_{k+1} = x^*_k - \underbrace{\alpha G_f(x^*_k)}_{P} + \underbrace{\beta \int_{k_0}^{k} G_f(x^*_k)}_{I}.$$

We can rewrite the integral term using the fundamental theorem of calculus: [11]

$$x^*_{k+1} = x^*_k - \alpha G_f(x^*_k) + \beta(f(x^*_k) - f(x^*_{k_0})). \tag{5}$$

The real valued constant $\beta$ tells us how much weight to put on the integral term. Another choice here that affects the performance is the lower bound of the integral, $k_0 \in \mathbb{N} \cup \{0\}$. This value tells us how big the interval we integrate over will be. In other words, if $k_0$ is close to 0 we will have a long memory and consider the past a lot. If $k_0$ is close to $k$ then we will forget the past much quicker. After implementing this in code it was time to compare the performance of this algorithm compared to the gradient descent we previously used.

We used $\beta = \alpha = 0.8$ which means we're weighing the two terms equally. We also let $k_0 = k - 10$ so we are considering the last 10 iterations. If there has been less than 10 iterations then we consider all of them. The function that we are going to optimize here is

$$f(x_1, x_2) = 0.5x_1^2 + 0.05x_2^2. \tag{6}$$

To make the comparison we are going to try a number of initial conditions $x_0 \in \mathbb{R}^2$. We will try 12 initial conditions spread out on the unit circle with respect to the $x_1$–$x_2$ plane. So the angle is 30 degrees between each point.

In Figure 11, the function is plotted in blue and the unit circle is plotted in purple. So, the starting points we are considering are those along the intersection of the plots. To the right the same plot is shown from above to only show the $x_1$–$x_2$ plane. Figure 12 shows how many iterations was needed from each starting position to converge, this time with the stopping condition $\epsilon = 0.002$. In this figure, we show the comparison with and without PI control.

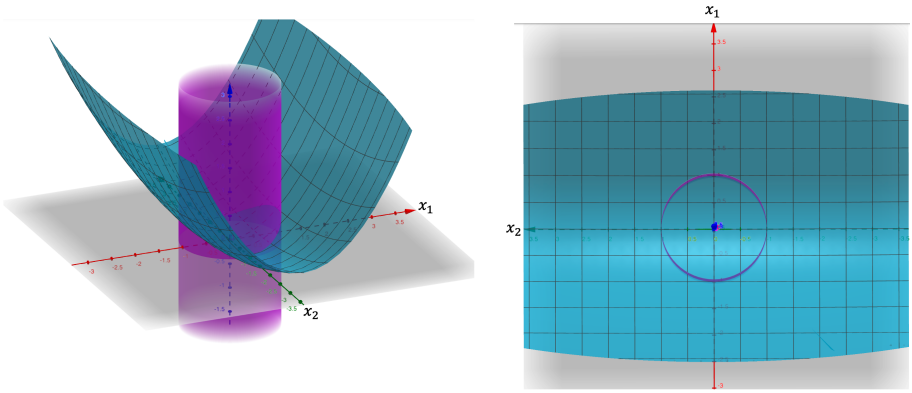


Figure 11: To the left we have the function $f(x_1, x_2) = 0.5x_1^2 + 0.05x_2^2$ in blue and the unit circle with respect to the $x_1, x_2$ plane in purple. On the right is the same plot viewed from above.

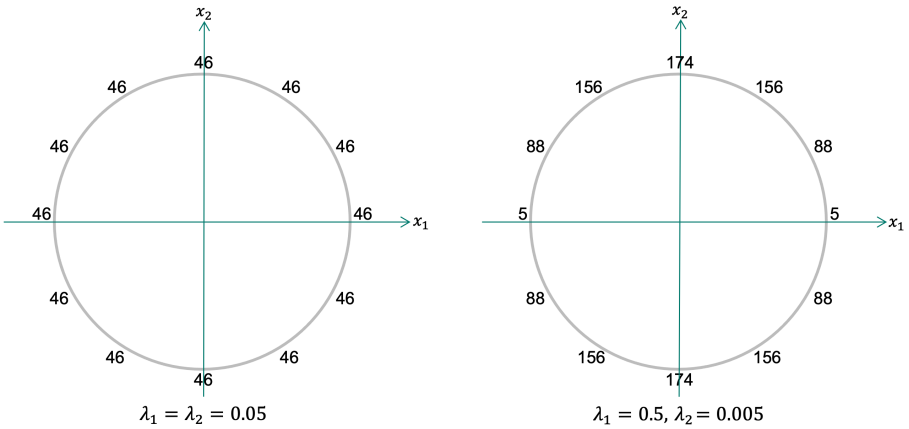


Figure 12: The numbers of iterations needed to converge to the minimum when starting from different points on the unit circle. We are also showing the impact of using PI control compared to normal gradient descent.

As we can see in Figure 12, – using PI control with the parameters described before seems to reduce the number of iterations needed. Though, similarly to the problem with the Wolfe conditions, there is a trade-off when it comes to running time since we now have to compute another term.

However, after also timing the algorithm from these initial conditions we found that the algorithm was faster on average using PI control. The average time it took for the algorithm to converge from one of these starting points in the case without PI control was $7.48 \cdot 10^{-4}$ seconds. When using PI control that time was instead $6.30 \cdot 10^{-4}$ seconds.

# 5 Control perspective on distributed optimization

## 5.1 Consensus

Let us suppose that each agent in a network has reached an estimate of the value to minimize an objective function. By leveraging consensus, all agents will agree on one common value, which becomes the global solution. The way that the nodes reach a consensus is by exchanging information with their neighbors about their current state. This is done using a consensus term, which tracks the difference between neighboring nodes' states.

To understand how the consensus term works, we will look at a network consisting of four nodes in a undirected ring graph – see Figure 14. For now we let the functions have one dimensional inputs but after we will increase the dimension as showed in Figure 12. We will let the nodes communicate their states using the following algorithm:

$$x_i(k+1) = x_i - \alpha \sum_{j \in \mathcal{N}_i} a_{ij}(x_j(k) - x_i(k)). \tag{7}$$

Where $0 \leq a_{ij} \leq$ is the weight of the edge between nodes $i$ and $j$. Here – we set $a_{ij} = 1$ for each pair $i, j$. The trajectories of the four nodes is shown in Figure 13, note how they start in different positions but converge towards the same value.

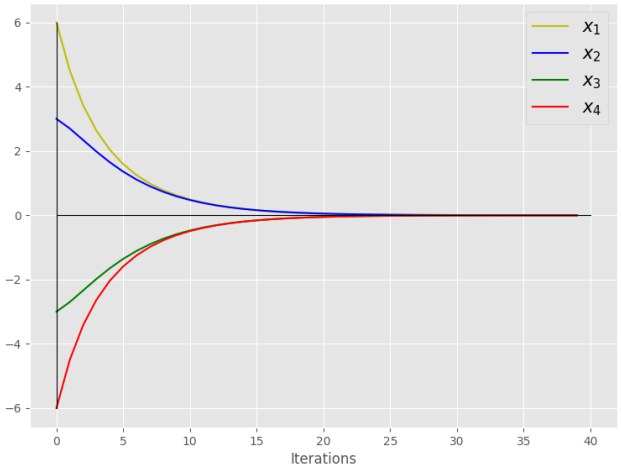


Figure 13: A plot illustrating consensus between 4 nodes with different starting values.

As we proceed to the implementation the network will still be a ring graph with four nodes. The only difference now is the dimension of the inputs of the functions. We have local functions $f_i : \mathbb{R}^2 \to \mathbb{R}$ and an objective function $f : \mathbb{R}^2 \to \mathbb{R}$ such that $f = \sum_{i=1}^{4} f_i$.

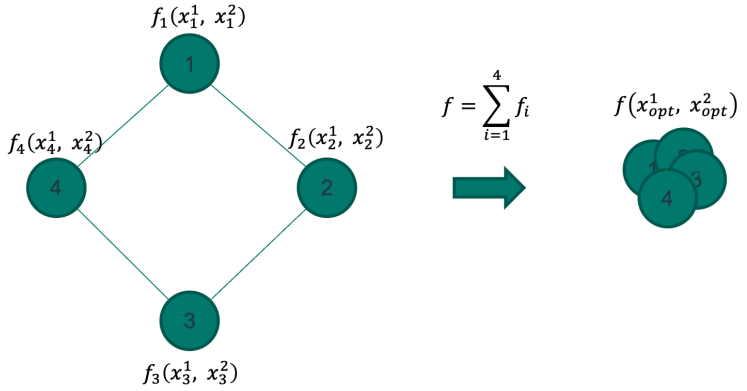


Figure 14: Illustration of the consensus in a simple network using distributed optimization.

## 5.2 Implementation

By combining all the theory that we have covered, we will now solve an optimization problem using control and gradient descent. We will start by solving it with centralized optimization – see Section 5.3 before moving to the distributed setting – see Section 5.4. When we have successfully solved that we will then move on to research about the parameters in the system. Finally, we will see different ways to implement the control scheme.

The implementation will start with the goal of reproducing the results from [9]. In this setting – we are considering a distributed optimization case where the optimization is spread out among 4 entities just like above. In the paper they proposed the following algorithm for distributed optimization with control

$$\begin{cases} x_i(k+1) = x_i(k) + \underbrace{\beta \sum_{j \in \mathcal{N}_i} a_{ij}(x_j(k) - x_i(k))}_{\text{Consensus } P} + \underbrace{\beta \sum_{j \in \mathcal{N}_i} a_{ij}(z_j(k) - z_i(k))}_{\text{Consensus } I} \\ \qquad\qquad \underbrace{-\beta\alpha G_{f_i}(x_i(k))}_{P}, \\ z_i(k+1) = z_i(k) + \underbrace{\beta \sum_{j \in \mathcal{N}_i} a_{ij}(x_i(k) - x_j(k))}_{I}. \end{cases} \tag{8}$$

Algorithm (8) will serve as a backbone for our algorithms. In each iteration – we are going to update the values for $x$ as well as $z$. These terms correspond to proportional and integral terms, both for the global and the local objectives simultaneously. The consensus terms work toward the global objective and the other terms work towards local objectives. For node $i$, we have the local function $f_i : \mathbb{R}^2 \to \mathbb{R}$ defined as $f_i(x) = x^T P_i x + b_i x$, where $x = [x_1 \;\; x_2]^T$ and

$$P_1 = \begin{bmatrix} 0.2 & 0.1 \\ 0.1 & 0.2 \end{bmatrix}, \; P_2 = \begin{bmatrix} 0.4 & 0.1 \\ 0.2 & 0.4 \end{bmatrix}, \; P_3 = \begin{bmatrix} 0.3 & 0.1 \\ 0.1 & 0.2 \end{bmatrix}, \; P_4 = \begin{bmatrix} 0.5 & 0.1 \\ 0.1 & 0.2 \end{bmatrix},$$

$$b_1 = [1 \;\; 8]^T, \qquad b_2 = [1 \;\; 1]^T, \qquad b_3 = [3 \;\; 1]^T \text{ and } \quad b_4 = [5 \;\; 1]^T.$$

So, the goal is to find the global minimum point for the sum of these 4 functions. A positive with recreating the results of others is that we have a clear reference point. Some calculations show that the objective function is given by

$$f(x) = \sum_{i=1}^{4} f_i(x_1, x_2) = 1.4x_1^2 + x_2^2 + 0.9x_1x_2 + 10x_1 + 11x_2. \tag{9}$$

In the paper, they concluded that the sought minimum point is $x^{opt} = [-\frac{7}{3} \;\; -\frac{13}{3}]^T$.

### 5.3 Centralized optimization with 4 entities

We start by solving (9) in the more simple centralized way before moving to distributed optimization. This corresponds to simply letting one node use gradient descent to optimize the sum of the functions. The code is similar to Algorithm 2 but adjusted to handle higher dimensions. The learning rate is chosen by the algorithm itself using the Wolfe conditions, again with $c_1 = 0.0001$ and $c_2 = 0.9$.

The plot is shown in Figure 15. Something to note is that this algorithm converged toward the values $x^{opt} = [-\frac{1010}{479} \;\; -\frac{2180}{479}]^T$. These values differ slightly from what was proposed in the paper, but that is not necessarily significant when it comes to a numerical solution. After manually checking the values we found that the true minimum of the function is in fact $[-\frac{1010}{479} \;\; -\frac{2180}{479}]^T$. Even though the function's value is not very different we will still consider this point as our $x^{opt}$. The function had the following values:

$$f\left(-\frac{3}{7} \;\; -\frac{11}{7}\right) = -35 \text{ and } f\left(-\frac{1010}{479} \;\; -\frac{2180}{479}\right) \approx -35.57.$$

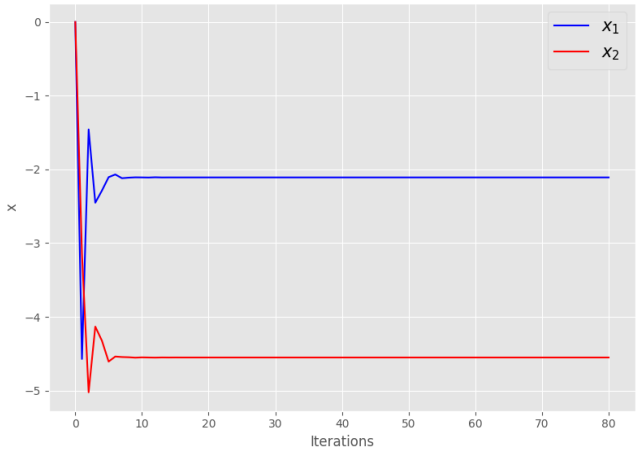


Figure 15: Centralized optimization using gradient descent in a network of 4 nodes. The parameter $\alpha$ is adaptively chosen using the Wolfe conditions.

## 5.4 Distributed optimization with 4 entities

Now, we are again going to solve (9) using distributed optimization. In this case we are going to incorporate algorithm (8) to update the values of the four nodes simultaneously. Since we already computed the minimum it will be clear to see if the algorithm is working correctly. We choose $\beta = 0.2$ and $\alpha = 3$. – The function which solves the main problem of this thesis is described in Algorithm 3. The starting points for the 4 nodes are given by the following:

$$x_1(0) = [0 \;\; 0]^T, x_2(0) = [0 \;\; -5]^T, x_3(0) = [-8 \;\; -3]^T \text{ and } \; x_4(0) = [5 \;\; 10]^T.$$

The starting points were arbitrarily chosen. Note that the choice for these initial conditions have insignificant impact in the long run for the algorithm. That is because, with working code, the algorithm should converge to the same values regardless of the initial conditions.

**Algorithm 3** Distributed optimization on 4 entities with control

**Input:** List of arrays $x$, function $f : \mathbb{R}^2 \to \mathbb{R}$, constants $\alpha, \beta, \epsilon$
**Output:** List containing the $x^* \in \mathbb{R}^2$ obtained in each step.

Set $t \leftarrow array([[0], [0]])$
Set $x\_history \leftarrow [x[0], x[1], x[2], x[3]]$
Set $z\_history \leftarrow [t, t, t, t]$
set $x_i = x[i-1]$
**while** len$(x_history) == 4$ **or** $\sum_{i=1}^{4} ||x\_history[-i] - x\_history[-i-4]|| > \epsilon$ **do**
    Update $x_i \leftarrow x_i + \beta \sum_{j \in \mathcal{N}_i}(x_j - x_i) + \beta \sum_{j \in \mathcal{N}_i}(z_j - z_i) - \alpha\beta G_{f_i}(x_i)$
    Update $z_i \leftarrow z_i + \beta \sum_{j \in \mathcal{N}_i}(x_i - x_j)$
    **for** a **do** in $[x_1, x_2, x_3, x_4]$
        Update $x\_history$.append(a)
    **end for**
**end while**
**return** $x\_history$

With Algorithm 3, the solution to the problem can be shown by plotting the trajectories of the nodes. In Figure 16, we have plotted the evolution of the estimates of $\min_{x \in \mathbb{R}^2} f(x)$ from node 1. The trajectories of the other nodes behave similarly, and in particular, converge to the same values.

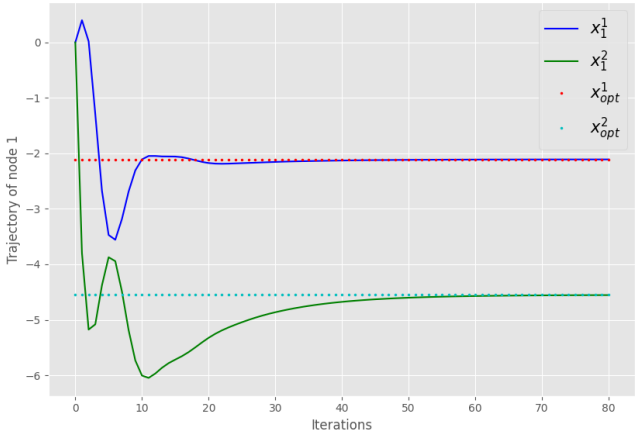


Figure 16: Distributed optimization in a network of 4 nodes. The parameters are $\alpha = 3, \beta = 0.2$ and $\epsilon = 0.001$.

Since, $x_1^1$ and $x_1^2$ converge to the correct values, we conclude that the implementation is correct.

# 6 Discussion

## 6.1 Results

We have successfully solved an optimization problem in a simple network. We have implemented both centralized optimization as well as distributed optimization. In the centralized case, we managed to automate the task of choosing $\alpha$ in each step using the Wolfe conditions. In both cases, we implemented PI control. We also saw the effect that PI control had on the performance.

However, the problem does not end here. There are a multitude of different settings to try in order to further improve the performance of the algorithm. Furthermore, analyzing the performance is not always straightforward. This is because decreasing the iterations needed for the algorithm to converge may still increase the time needed for the computer. Another factor is that a faster algorithm might not achieve the desired setpoint precision. Some things that may affect the performance will be discussed in the following sections.

## 6.2 Parameters

Our results suggest that choosing ample parameters is crucial when solving optimization problems. When using gradient descent, the choice for the learning rate $\alpha$ can make a massive difference. The Wolfe conditions arise as a solution to this but these conditions have their own parameters $c_1$ and $c_2$. Additionally, after making the choices for $c_1$ and $c_2$ there are still multiple $\alpha$ which satisfy the conditions. Therefore, there are still many things to explore down this path.

The stopping condition $\epsilon$ may not have as great of an impact as $\alpha$ but still serves as a constant regulating the trade-off between speed and accuracy. When we introduced PI control we also got a new constant $\beta$ which can be altered to change the weight of the integral term. Since for example for each $\alpha$ we can try different $\beta$ and $\epsilon$ choosing parameters becomes a sort of grid search. In this work we have limited ourselves and not optimized the parameters for the PI control.

## 6.3 Adjusting the PI control

The PI control scheme can also be explored further to improve the performance of the algorithm. As we discussed earlier, changing how much memory equation (5) should consider has an impact. We chose to consider 10 iterations which was quite an arbitrary choice. For example, if we look at the starting point

$[\frac{\sqrt{3}}{2} \quad \frac{1}{2}]^T$, we managed to get the iterations needed down from 37 to 31. But, if we look closer at the integral term – we can do even better [3].

We looked at how many iterations are needed to converge from this point for different levels of memory on the integral term. We could then see that having a memory which considers between 14 and 20 past iterations mean we only need 27 iterations to converge which was the lowest possible. However, if we had instead looked at the starting point $[\frac{1}{2} \quad \frac{\sqrt{3}}{2}]^T$ then the lowest amount of iterations needed would be 24. This is achieved if the memory is set to consider between 12 and 17 past iterations.

Recall that these values are calculated assuming that $\beta = \alpha = 0.8$. With different values for the parameters the optimal memory might change as well. This shows that there is a lot of potential to decrease the iterations and the time needed to converge.

## 6.4 Properties of the functions

Another factor to consider is the properties of the objective function and the local functions themselves. We will conclude this work by illustrating how the iterations needed to converge in Figure 12 changes with the properties of Function (6). First, we rewrite the function on the form

$$f(x) = x^T P x, \text{ where } P = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.05 \end{bmatrix}.$$

For this matrix $P$, we have the eigenvalues $\lambda_1 = 0.5$ and $\lambda_2 = 0.05$, now we look at the relationship between $\lambda_1$ and $\lambda_2$. Let us consider $\lambda_1 = \lambda_2 \cdot 10$, – we are going to change this relationship by dividing the eigenvalues by 10 one–by–one. So, in the first case – we will have $\lambda_1 = \lambda_2 = 0.05$ and in the second case we will have $\lambda_1 = 0.5$ and $\lambda_2 = 0.005$. How many iterations were needed to converge in these cases is shown in Figure 17.

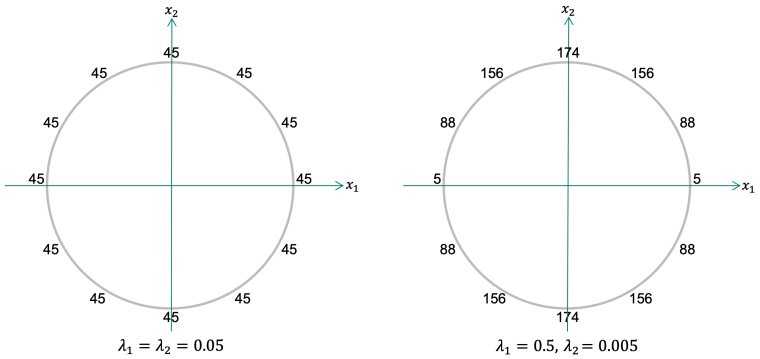


Figure 17: The iterations needed to converge from different starting points after we have changed the relationship between $\lambda_1$ and $\lambda_2$.

## 6.5 Conclusion

Distributed optimization serves as a powerful tool which can be used in large networks. It has a rather intricate implementation. In order to get the best possible performance one will have to spend time tuning the parameters of the numerical method used. Furthermore, using control is helpful to achieve a better performance, but control also comes with components in need of tuning.

# 7 References


[1] Michel Bierlaire. Descent methods and line search: first wolfe condition, 2019. Youtube.

[2] Michel Bierlaire. Descent methods and line search: second wolfe condition, 2019. Youtube.

[3] Sarthak Chatterjee, Subhro Das, and Sérgio Pequito. Neo: Neuro-inspired optimization—a fractional time series approach. *Frontiers in physiology*, page 1551, 2021.

[4] Jean-Pierre Corriou and Jean-Pierre Corriou. *Numerical Methods of Optimization*. Springer, 2021.

[5] Carl Knospe. Pid control. *IEEE Control Systems Magazine*, 26(1):30–31, 2006.

[6] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B Schön. *Machine learning: a first course for engineers and scientists*. Cambridge University Press, 2022.

[7] Tharinda Dilshan Piyadasa. Centralized vs distributed systems in a nutshell. 2020.

[8] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

[9] Jing Wang and Nicola Elia. Control approach to distributed optimization. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 557–561. IEEE, 2010.

[10] Jing Wang and Nicola Elia. A control perspective for centralized and distributed convex optimization. In *2011 50th IEEE conference on decision and control and European control conference*, pages 3800–3805. IEEE, 2011.

[11] Wikipedia contributors. Fundamental theorem of calculus — Wikipedia, the free encyclopedia, 2023. [Online; accessed 8-June-2023].

[12] Tao Yang, Xinlei Yi, Junfeng Wu, Ye Yuan, Di Wu, Ziyang Meng, Yiguang Hong, Hong Wang, Zongli Lin, and Karl H Johansson. A survey of distributed optimization. *Annual Reviews in Control*, 47:278–305, 2019.

# 8 Appendix

```
#This code produced figures 3,4 and 5 by alternating alpha.
import numpy as np
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])

#The basic function for gradient descent.
def gradient_descent(x, alpha=0.2, epsilon=0.001):
    x_history=[x]
    while len(x_history) == 1 or \
    abs(x_history[-1] - x_history[-2]) > epsilon:
        x = x - alpha*(2*x)
        x_history.append(x)
    return x_history

x = 5
x_history = gradient_descent(x)
# m = len(x_history)
# print(m)
# print(x_history[-1])

#Plot
fig,ax = plt.subplots(figsize=(12,8))
ax.set_ylabel('y')
ax.set_xlabel('x')
x = np.linspace(-5,5,1000)
y = x**2
plt.plot(x,y, 'b', linewidth = 3)
plt.plot(x_history, list(map(lambda x : x**2, x_history)),\
     'r' '.-.', markersize = 14)
plt.legend(['$x^2$'], fontsize=15)
plt.show()
```

```
#The code for figures 6 and 7
import matplotlib.pyplot as plt
import numpy as np
plt.style.use(['ggplot'])

#For figure 7 we changed x to 0.0002
x = 5
c1 = 0.0001
c2 = 0.9

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xlabel(f'${chr(945)}_k$')
ax2.set_xlabel(f'${chr(945)}_k$')
alpha = np.linspace(0,1.2,500)
x1 = (x - alpha*x*2)**2
x2 = x**2 - 2*c1*x*alpha
ax1.plot(alpha ,x1, 'b', linewidth = 3)
ax1.plot(alpha ,x2, 'r', linewidth = 3)
ax1.legend(['$f_1^k$','$\gamma_1^k$'])

y1 = 2*(x - alpha*x*2)/(2*x)
y2 = c2*np.ones(500)
ax2.plot(alpha ,y1, 'b', linewidth = 3)
ax2.plot(alpha ,y2, 'r', linewidth = 3)
ax2.legend(['$f_2^k$','$\gamma_2^k$'])
plt.show()
```

```
#This code integrates the Wolfe conditions as in Figure 8
import numpy as np
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])

def gradient_descent(x, epsilon=0.001):
    x_history = [x]
    while len(x_history) == 1 or \
    abs(x_history[-1] - x_history[-2]) > epsilon:
        a = np.array([[4*x**2]])
        b = np.array([4*x**2-0.0002*x])
        alpha = float((np.linalg.solve(a, b)+(1-c2)/2)/2)
        x = x - alpha*(2*x)
        x_history.append(x)
    return x_history

x = 5
c1 = 0.0001
c2 = 0.9
x_history = gradient_descent(x)

#Plot
fig,ax = plt.subplots(figsize=(12,8))
ax.set_ylabel('y')
ax.set_xlabel('x')
x = np.linspace(-5,5,1000)
y = x**2
plt.plot(x,y, 'b', linewidth = 3)
plt.plot(x_history, list(map(lambda x: x**2, x_history)), \
'r' '.-.', markersize = 14)
plt.legend(['$x^2$'], fontsize=15)
plt.show()
```

```
#This code performs 2 dimensional gradient descent
import numpy as np
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])


def gradient_descent(x, learning_rate=0.6, epsilon=0.001):
    t = np.array([[0.0], [0.0]])
    x_history = [t, x]

    x1 = x[0]
    x2 = x[1]
    while learning_rate > epsilon:
        if np.linalg.norm(x_history[-1] - x_history[-2])\
        > epsilon:
            x1 = x1  - learning_rate*(2*x1)
            x2 = x2  - learning_rate*(2*x2)

            x_history.append(np.array([[x1],[x2]]))
        else:
            learning_rate = learning_rate/2
    return x_history

#Now x is an array consisting of x1 and x2.
x = np.array([[5.0], [-5.0]])

x_history = gradient_descent(x)
m = len(x_history)
print()
print(m)
print(x_history[-1])

x1_hat = np.array([x_history[i+1][0] for i in range(m-1)]\
,dtype=object)
x2_hat = np.array([x_history[i+1][1] for i in range(m-1)]\
,dtype=object)

#""""""
#Plot
fig,ax = plt.subplots(figsize=(12,8))
ax.set_ylabel('x*')
ax.set_xlabel('Iterations')
ax.plot(range(m-1), x1_hat, 'b', x2_hat, 'r')
plt.legend(['$x_1$', '$x_2$'], fontsize=15)
plt.show()
#""""""
```

```
#The code which illustrates PI control in Figure 10
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])
import numpy as np

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xlabel('t')
ax2.set_xlabel('t')
x = np.linspace(0,3,1000)
x2 = np.linspace(-2,3,1000)
y = x**3-4*x**2+2.5*x+2

y1 = 0.4*(x**3-4*x**2+2.5*x+2)
y2 = (x**4/4)-(4*x**3/3)+1.25*x**2+2*x

ax1.plot(x,y,'b', linewidth = 3)
ax1.plot(x,y1, 'r', linewidth = 3)
ax2.plot(x,y,'b', linewidth = 3)
ax2.plot(x,y2, 'r', linewidth = 3)

ax1.plot(x,x*0, 'black', linewidth = 0.8)
ax1.plot(x2*0,x2, 'black', linewidth = 0.8)
ax2.plot(x,x*0, 'black', linewidth = 0.8)
ax2.plot(x2*0,x2, 'black', linewidth = 0.8)

ax1.legend(['Error_l(t)','P'])
ax2.legend(['Error_l(t)','I'])
plt.show()
```

```
#This code performs gradient descent for the function \
#f(x_1, x_2) = 0.5x_1**2 + 0.05x_2**2
#From this code we obtain the values illustrated \
# to the left in Figure 12.
import numpy as np
import matplotlib.pyplot as plt
from time import perf_counter as pc
plt.style.use(['ggplot'])

def gradient_descent(x, alpha=0.8, epsilon=0.002):
    x_history = [x]
    x1 = x[0]
    x2 = x[1]
    while True:
        if len(x_history) == 1 or np.linalg.norm( \
        x_history[-1] - x_history[-2]) > epsilon:
            x1 = x1 - alpha*(1.0*x1)
            x2 = x2 - alpha*(0.1*x2)

            x_history.append(np.array([[x1],[x2]]))
        else:
            break
    return x_history
#Changing lambda1 or lambda2 is how we obtained \
#the values in Figure 17.
lambda1, lambda2 = 0.5, 0.05

#Here, we can input different initial conditions.
x = np.array([[1], [0]])

# m represents the iterations needed to converge
x_history = gradient_descent(x)
m = len(x_history)-1
print(m)

#Checking the time in seconds
def timer():
    tstart = pc()
    gradient_descent(x)
    tend = pc()
    return tend-tstart
L=[]
for t in range(100):
    L.append(timer())
print(sum(L)/len(L))
```

```
#This code integrates PI control and corresponds \
#to the values illustrated to the right in Figure 12.
import numpy as np
import matplotlib.pyplot as plt
from time import perf_counter as pc
plt.style.use(['ggplot'])

# q determines how many past iterations to consider.
q = 10

def gradient_descent(x,beta=0.8,alpha=0.8,epsilon=0.002):
    x_history = [x]
    x1 = x[0]
    x2 = x[1]
    while True:
        if len(x_history) == 1 or np.linalg.norm( \
        x_history[-1] - x_history[-2]) > epsilon:

            z1 = (0.5*x1**2 - \
            0.5*x_history[max(-len(x_history),-q)][0]**2)
            z2 = (0.05*x2**2 - \
            0.05*x_history[max(-len(x_history),-q)][1]**2)

            x1 = float(x1 - alpha*(1.0*x1) + beta*z1)
            x2 = float(x2 - alpha*(0.1*x2) + \
            np.sign(x[1])*beta*z2)

            x_history.append(np.array([[x1],[x2]]))
        else:
            break
    return x_history

x = np.array([[1], [0]])
x_history = gradient_descent(x)
m = len(x_history)-1
print(m)

def timer():
    tstart = pc()
    gradient_descent(x)
    tend = pc()
    return tend-tstart
L=[]
for t in range(100):
    L.append(timer())
print(sum(L)/len(L))
```

```
#This code explores consensus and plots Figure 13.
import numpy as np
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])

def gradient_descent(x, alpha=0.1):
    x1, x2, x3, x4 = x[0], x[1], x[2], x[3]
    x_history = x
    for i in range(40):
        x12, x14 = x1 - x2, x1 - x4
        x23, x34 = x2 - x3, x3 - x4

        x1 = x1 - alpha*(x12 + x14)
        x2 = x2 + alpha*(x12 - x23)
        x3 = x3 + alpha*(x23 - x34)
        x4 = x4 + alpha*(x14 + x34)

        for k in [x1,x2,x3,x4]:
            x_history.append(k)
    return x_history

#Arbitrarily chosen distinct starting points
x = [6, 3, -3, -6]
x_history = gradient_descent(x)

x1_hat = np.array([x_history[4*i] for i in range(40)],
    dtype=object)
x2_hat = np.array([x_history[4*i+1] for i in range(40)],
    dtype=object)
x3_hat = np.array([x_history[4*i+2] for i in range(40)],
    dtype=object)
x4_hat = np.array([x_history[4*i+3] for i in range(40)],
    dtype=object)

#Plot
fig,ax = plt.subplots(figsize=(12,8))
x = np.linspace(0,40,1000)
x2 = np.linspace(-6,6,1000)
ax.set_xlabel('Iterations')
ax.plot(range(40), x1_hat, 'y', x2_hat, 'b',
    x3_hat, 'g', x4_hat, 'r', markersize = 3)
plt.legend(['$x_1$','$x_2$','$x_3$','$x_4$'], fontsize=15)
plt.plot(x,x*0, 'black', linewidth = 0.8)
plt.plot(x2*0,x2, 'black', linewidth = 0.8)
plt.show()
```

```
#This code solves the centralized optimization
# problem as illustrated in Figure 15.
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])

#To find the upper limit for alpha.
def f1(a, x1,x2):
    return c1*a*(np.sqrt((2.8*x1+0.9*x2+10)**2+
    (0.9*x1+2*x2+11)**2))-2.8*a*x1*(2.8*x1+0.9*x2+10) \
    + 1.4*a**2*(2.8*x1+0.9*x2+10)**2 - 2*a*x2*\
    (0.9*x1+2*x2+11) + a**2*(0.9*x1+2*x2+11)**2 \
    - 0.9*(a*x1*(0.9*x1+2*x2+11)+a*x2*(2.8*x1+0.9*x2+10)-\
    a**2*(2.8*x1+0.9*x2+10)*(0.9*x1+2*x2+11)) \
    - 10*a*(2.8*x1+0.9*x2+10) - 11*a*(0.9*x1+2*x2+11)

#To find the lower limit for alpha.
def f2(a, x1,x2):
    return np.sqrt((2.8*(x1-a*(2.8*x1+0.9*x2+10))+\
    0.9*(x2-a*(0.9*x1+2*x2+11))+10)**2 \
        +(0.9*(x1-a*(2.8*x1+0.9*x2+10))+\
        2*(x2-a*(0.9*x1+2*x2+11))+11)**2)-c2

def gradient_descent(x, epsilon=0.001):
    t = np.array([[epsilon], [epsilon]])
    x_history = [t,x]

    x1, x2 = x[0], x[1]
    alpha = (fsolve(f1, 1, (x1,x2)) + \
    fsolve(f2, 1, (x1,x2)))/2
    for i in range(80):
        alpha = ((fsolve(f1, alpha, (x1,x2)) + \
        fsolve(f2, alpha, (x1,x2)))/2) % 1
        x1 = x1 - alpha*(2.8*x1 + 0.9*x2 + 10)
        x2 = x2 - alpha*(0.9*x1 + 2.0*x2 + 11)

        x_history.append(np.array([[x1],[x2]]))
    return x_history

c1 = 0.0001
c2 = 0.9

x = np.array([[0.0], [0.0]])
x_history = gradient_descent(x)
```

```
x1_hat = np.array([x_history[i+1][0] for i in range(80)],
    dtype=object)
x2_hat = np.array([x_history[i+1][1] for i in range(80)],
    dtype=object)

#Plot
fig,ax = plt.subplots(figsize=(12,8))
ax.set_ylabel('x')
ax.set_xlabel('Iterations')
ax.plot(range(80), x1_hat, 'b', x2_hat, 'r')
plt.legend(['$x_1$','$x_2$'],fontsize=15)
plt.show()
```

```
#This code solves the main problem of the thesis.
#It implements distributed optimization and control.
import numpy as np
import matplotlib.pyplot as plt
plt.style.use(['ggplot'])


#This is the function described in algorithm 3
def gradient_descent(x, beta=0.2, learning_rate=3,
epsilon=0.001):
    t = np.array([[epsilon], [epsilon]])
    x_history = [t,t,t,t, x[0], x[1], x[2], x[3]]
    z_history = [t,t,t,t]

    #x_ij means coordinate i for node j
    x_11 , x_21= x[0][0], x[0][1]
    x_12 , x_22= x[1][0], x[1][1]
    x_13 , x_23= x[2][0], x[2][1]
    x_14 , x_24= x[3][0], x[3][1]
    z_11, z_21, z_12, z_22, z_13, z_23, z_14, z_24 = \
    0,0,0,0,0,0,0,0
    for i in range(100):
        if np.linalg.norm(x_history[-4]-x_history[-8]) +\
            np.linalg.norm(x_history[-3]-x_history[-7])+\
            np.linalg.norm(x_history[-2]-x_history[-6])+\
            np.linalg.norm(x_history[-1]-x_history[-5])\
                < epsilon:
            break

        x_112, x_212 = x_11 - x_12, x_21 - x_22
        x_114, x_214 = x_11 - x_14, x_21 - x_24
        x_123, x_223 = x_12 - x_13, x_22 - x_23
        x_134, x_234 = x_13 - x_14, x_23 - x_24

        z_112, z_212 = z_11 - z_12, z_21 - z_22
        z_114, z_214 = z_11 - z_14, z_21 - z_24
        z_123, z_223 = z_12 - z_13, z_22 - z_23
        z_134, z_234 = z_13 - z_14, z_23 - z_24

        #Updating x and z according to equation (8)
        y = x_14
        x_11 = x_11 - beta*(x_112 + x_114) - beta*(z_112+\
         z_114) - beta*learning_rate*(0.4*x_11+0.2*x_21+1)

        x_21 = x_21 - beta*(x_212 + x_214) - beta*(z_212+\
         z_214) - beta*learning_rate*(0.4*x_21 + \
```

```
        0.2*(x_112+x_12) + 8)

    x_12 = x_12 + beta*(x_112 - x_123) + beta*(z_112-\
     z_123) - beta*learning_rate*(0.8*x_12+0.3*x_22+1)

    x_22 = x_22 + beta*(x_212 - x_223) + beta*(z_212-\
     z_223) - beta*learning_rate*(0.8*x_22 + \
        0.3*(x_123+x_13) + 1)

    x_13 = x_13 + beta*(x_123 - x_134) + beta*(z_123-\
     z_134) - beta*learning_rate*(0.6*x_13+0.2*x_23+3)

    x_23 = x_23 + beta*(x_223 - x_234) + beta*(z_223-\
     z_234) - beta*learning_rate*(0.4*x_23 + \
        0.2*(x_134+x_14) + 1)

    x_14 = x_14 + beta*(x_114 + x_134) + beta*(z_114+\
     z_134) - beta*learning_rate*(x_14 + 0.2*x_24 + 5)

    x_24 = x_24 + beta*(x_214 + x_234) + beta*(z_214+\
        z_234) - beta*learning_rate*(0.4*x_24 + \
            0.2*y + 1)

    z_11 = z_11 + beta*(x_112 + x_114)
    z_21 = z_21 + beta*(x_212 + x_214)
    z_12 = z_12 - beta*(x_112 - x_123)
    z_22 = z_22 - beta*(x_212 - x_223)
    z_13 = z_13 - beta*(x_123 - x_134)
    z_23 = z_23 - beta*(x_223 + x_234)
    z_14 = z_14 - beta*(x_114 + x_134)
    z_24 = z_24 - beta*(x_214 + x_234)

    for k in [np.array([[x_11],[x_21]]),
    np.array([[x_12],[x_22]]),
    np.array([[x_13],[x_23]]),
    np.array([[x_14],[x_24]])]:
        x_history.append(k)

    for k in [np.array([[z_11],[z_21]]),
    np.array([[z_12],[z_22]]),
    np.array([[z_13],[z_23]]),
    np.array([[z_14],[z_24]])]:
        z_history.append(k)

return x_history
```

```
x_1 = np.array([[0.0], [0.0]])
x_2 = np.array([[0.0], [-5.0]])
x_3 = np.array([[-8.0], [-3.0]])
x_4 = np.array([[5.0], [10.0]])
x = [x_1, x_2, x_3, x_4]

x_history = gradient_descent(x)
m = len(x_history)//4
print(m)
print(x_history[-1])

#Choose which node's coordinates to plot by choosing\
# j in {1,2,3,4}
j = 1
x1_hat = np.array([x_history[4*(i+1)+j-1][0] for i in
    range(m-1)], dtype=object)
x2_hat = np.array([x_history[4*(i+1)+j-1][1] for i in
    range(m-1)], dtype=object)
x1_opt = np.array([-1010/479 for i in range(m-1)],
dtype=object)
x2_opt = np.array([-2180/479 for i in range(m-1)],
dtype=object)

#Plot
fig,ax = plt.subplots(figsize=(12,8))
ax.set_ylabel('Trajectory of node 1')
ax.set_xlabel('Iterations')
ax.plot(range(m-1), x1_hat, 'b', x2_hat, 'g', x1_opt,
    '.' 'r', x2_opt, '.' 'c', markersize = 3)
plt.legend(['$x^1$','$x^2$','$x^1_{opt}$', '$x^2_{opt}$'],
    fontsize=15)
plt.show()
```