Assignment 4

Johannes Holler
Philipp Temmel

February 2022

0 General information

The assignment was developed using a local conda environment using the following library versions:

python=3.7
matplotlib=3.5
scipy=1.7

Upon running the uploaded main.py file all tasks are executed and the figures pdf file is created for all methods and our final choice of parameters exactly as found in the report. We added lots of comments within the source code to make it simpler following our thoughts.

In total we used four instead of two different methods for task 1 and 2. We analytically derived the projected gradient method (see method __projected_gradient_method() in the source code) and the Frank Wolfe method (using the predefined diminishing step size as defined in algorithm 2 of the assignment sheet) as required. Additionally, we developed our own projected gradient method which we explain here in the report. While this method works differently than the provided algorithm provided for the regular projected gradient method in the assignment sheet, we found our algorithm to perform nearly identical to the original projected gradient method. The solutions were numerically extremely close (like 10^-15 or similar in maximal dimension deviations of the solution vector and the cost objective function was pretty much identical). Please refer to the method __selfmade_projection_method() in the python source code if you want to have a look at the implementation itself. Also, we added the bonus task, i.e. exact line search for the Frank Wolfe method, as a 4th method. You can find the adaptations within the method __frank_wolfe_method() in the source code. We also added the comparison in all plots and discuss the differences here as well.

0.1 Bonus: Our own customly developed projected gradient method

As noted previously, in addition to the provided algorithms in the assignment sheet we also tried to develop our own projected gradient method from scratch on our own and added this method to all other comparisons within this assignment. So every time when we refer to our custom projected gradient method we are talking about our own developed version in comparison to the one defined in algorithm 1 of the assignment sheet. In python you can find the implementation of our own custom version in the method __selfmade_projection_method(). Overall we can say that our custom version performed nearly identical to the projected gradient method of the assignment sheet both in terms of convergence, the absolute difference in values of each dimension in the solution vectors x_k as well as the final values of the cost objective functions we try to minimize in this assignment.

Let's discuss the key ideas behind our custom solution going through the method step by step:

The main iteration looks similar to the regular projected gradient method. The choice for the step size is performed exactly the same, i.e. step size = 2 / lipschitz constant - epsilon. Our choice of epsilon was 10^{-4} and ensures that our step size stays within the allowed interval due to numerical reasons in order to ensure the requirements for the sufficient decrease lemma.

Next we define a hyperplane in which our constrained space (i.e. the unit simplex) is contained. All vectors on this defined hyperplane already fulfill the requirement that the sum over all dimensions of the vector x_k is 1. However, the additional constraint of all dimensions having to be positive (i.e. ¿= 0) is only fulfilled on the unit simplex which denotes our constrained space (in constrast to the full plane = hyperplane). The hyperplane is defined by choosing a point p on the hyperplane and adequate linear independent direction vectors. The problem resides in a d-dimensional vector space while the hyperplane defines a d-1 dimensional subspace within that vector space. For the choice of p we may choose any point on the hyperplane, we decided to choose a point p with its first dimensional component being 1 and all other dimensional vector components set to 0. The d-1 direction vectors are constructed by setting their first dimension to 1 and one of their dimensions to -1. (i.e Figure 1) The intuition was to find d-1 linearly independent direction vectors which lie on the hyperplane, i.e. the linearity of the vector space is fulfilled. If starting from our point p we add lambda times any of our direction vectors the projected point denotes a linear projection and is again part of the hyperplane (x_projected = x + lambda * direction_vector) since we add and subtract the same factor lambda which overall results in the same vector coordinate sum, e.g. in 2D: p(1, 0) + 3 * (1, -1) = (4, -3) = x_projected (the total sum of the vector coordinates always remains 1). Note that this does not denote the projection back to the

constrained set which is discussed next, but only to fulfill the unit simplex constrain in our constructed hyperplane.

Next we start the iterations of our custom projected gradient method. In each of the iterations we perform the following steps:

1. Perform steepest gradient descent (exactly like in the regular projected gradient method, hence we don't go into details here)

2. Project our approximated solution vector z of the current iteration on the unconstrained previously constructed hyperplane:
   The analytical way to project a point onto a hyperplane can be found in section 3
   (see method __projection_on_unconstrained_hyperplane() in the source code)

3. Project our intermediate result from the unconstrained hyperplane on the unit simplex
   This section can found in the method __projection_from_hyperplane_to_unit_simplex(). First we create a binary projection mask by initiliazing all its values with 1 to store which dimensions still need a projection from the negative to the positive half space in the respective dimension. Next we loop as long as any negative vector coordinates exist (i.e. if all vector coordinates are already positive the projection is unnecessary). Inside the loop we only need to take negative vector coordinates into consideration (hence we check whether the current vector coordinate is positive in an early escape inside the code). Then we set the respective vector coordinate in our binary projection mask to 0 in order to denote that this dimension has been fixed already (our projection works in a way that doesn't violate already fixed previously negative dimensions). Then we project the current negative vector coordinate to 0 (which fulfills our unit simplex constraint) and all other not yet fixed dimensions (i.e. all dimensions which still have value 1 in our binary projection mask) are projected by a corresponding uniform factor in the opposite direction. This factor is defined by the value of the currently modified dimension divided by the number of still modifiable dimensions.

	: 0	: 1	: 2	: 3	: 4	: 5	: 6	: 7	: 8	: 9	: 10	: 11	: 12	: 13
0	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
1	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
2	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
9	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000	0.00000
10	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000	0.00000
11	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000	0.00000
12	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000	0.00000
13	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000	0.00000
14	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	-1.00000

Figure 1: Matrix of direction vectors

1 Task 1

1.1 Gradient and Hessian of f(x), Lipschitz constant

$$\min_{\boldsymbol{x}} \frac{1}{2}||\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}||_2^2 \quad \text{s.t. } x \in \Delta = \left\{ \boldsymbol{x} \in \mathbb{R}^d | x_j \geq 0, \sum_{j=1}^{d} x_j = 1 \right\} \tag{1}$$

$$\begin{aligned} f(\boldsymbol{x}) &= \frac{1}{2}||\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}||_2^2 && (2)\\ &= \frac{1}{2}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}) && (3)\\ &= \frac{1}{2}(\boldsymbol{x}^T\boldsymbol{A}^T-\boldsymbol{b}^T)(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}) && (4)\\ &= \frac{1}{2}(\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}^T\boldsymbol{A}\boldsymbol{x}-\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b}+\boldsymbol{b}^T\boldsymbol{b}) && (5)\\ &= \frac{1}{2}(\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}-2\boldsymbol{b}^T\boldsymbol{A}\boldsymbol{x}+\boldsymbol{b}^T\boldsymbol{b}) && (6) \end{aligned}$$

$$\begin{aligned} \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} &= \frac{1}{2}(2\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}-2\boldsymbol{A}^T\boldsymbol{b}) && (7)\\ &= \boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}-\boldsymbol{A}^T\boldsymbol{b} && (8)\\ &= \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}) && (9) \end{aligned}$$

$$\frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}\partial \boldsymbol{x}} = \boldsymbol{A}^T\boldsymbol{A} \tag{10}$$

Equation (10) is the Hessian of $f(\boldsymbol{x})$. Since the Lipschitz constant is defined as the norm of the Hessian of a function, and the norm of a matrix is the largest Eigen-value the Lipschitz constant for $f(\boldsymbol{x})$ is $\lambda_{max}(\boldsymbol{A}^T\boldsymbol{A})$

1.2 optimal solution of y_k

$$\arg\min_{\boldsymbol{y}} \nabla f(\boldsymbol{x}_k)^T\boldsymbol{y} \quad \text{s.t. } \boldsymbol{y} \in \boldsymbol{C} \tag{11}$$

Let i* be the index, where $\nabla f(\boldsymbol{x}_k)$ has the smallest value. The $\boldsymbol{y}$ which minimizes this function is a vector with the value one at position i* and zero at every other position.
Proof by contradiction:

Assumption :

$$\exists \boldsymbol{y'} : y'_j > 0 \quad , j \neq i^* \quad \forall i \neq i^* \wedge i \neq j : \quad y'_i = 0, \qquad \boldsymbol{y'} \in \boldsymbol{C} \tag{12}$$

and

$$\nabla f(\boldsymbol{x}_k)^T \boldsymbol{y'} < \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y} \qquad , y_{i^*} = 1, y_i = 0 \quad \forall i \neq i^* \tag{13}$$

then

$$\nabla f(\boldsymbol{x}_k) = \begin{pmatrix} (\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_1 \\ (\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_2 \\ \dots \\ (\nabla f(\boldsymbol{x}_k))_{i^*} \\ \dots \\ (\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_j \\ \dots \\ (\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_d \end{pmatrix} \qquad , \forall i : \quad \epsilon_i > 0 \tag{14}$$

$$\nabla f(\boldsymbol{x}_k)^T \boldsymbol{y} = (\nabla f(\boldsymbol{x}_k))_{i^*} y_{i^*} \tag{15}$$

$$\nabla f(\boldsymbol{x}_k)^T \boldsymbol{y'} = (\nabla f(\boldsymbol{x}_k))_{i^*} y'_{i^*} + ((\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_j) y'_j \tag{16}$$

$$= (\nabla f(\boldsymbol{x}_k))_{i^*} (y'_{i^*} + y'_j) + \epsilon_j y'_j \tag{17}$$

$$\sum_{i=1}^{d} y'_i = 1 \quad \Rightarrow \quad y'_{i^*} + y'_j = 1 \tag{18}$$

$$\Rightarrow \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y'} = (\nabla f(\boldsymbol{x}_k))_{i^*} + \epsilon_j y'_j \tag{19}$$

$$\epsilon_j > 0 \wedge y'_j > 0 \quad \Rightarrow \quad \epsilon_j y'_j > 0 \tag{20}$$

$$\Rightarrow \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y'} > \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y} \tag{21}$$

(13) and (21) contradict each other

$$\Rightarrow \nexists \boldsymbol{y'} : \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y'} < \nabla f(\boldsymbol{x}_k)^T \boldsymbol{y} \tag{22}$$

1.3 Dictionary Matrix A

The python implementation of the dictionary matrix A can be found in the method __project_on_unit_simplex where first the dictionary matrix is created, then we use this dictionary matrix to calculate the required matrix A.T * A as we analytically derived in task 1.1 and finally call the provided power method which calculates us the largest eigenvalue in an efficient manner. This largest eigenvalue denotes our lipschitz constant which we used in algorithm 1 to calculate the upper boundary in order to select a good step size.

1.4 Projected Gradient Method

In the method __projected_gradient_method() you can find the implementation of algorithm 1 (projected gradient method). Please note that we decided to choose the step size slightly below the upper bound by subtracting epsilon =

10^{-4} from the upper boundary, i.e. our step size t = 2 / lipschitz_constant - epsilon. The reason for subtracting a suitable small epsilon here is so that due to our numerical calculations (using float values) we might otherwise by accident get a step size for which the sufficient decrease lemma discussed in the lectures is not valid any more. All our stated results in this report in the upcoming tasks are based on this step size choice for the projected gradient method.

1.5

You can find the implementation of the Frank Wolfe method (both versions, i.e. using a predefined diminishing step size and also the bonus task version using an exact line search instead) in the method __frank_wolfe_method().

1.6 & 1.7 4 individual experiments

In order to ensure stable comparisions between different test runs (e.g. finding good values for the number of iterations), i.e. deterministic behavior across repeated algorithm runs, we included a random seed of value 42 in the code which was used to initialize np.random.seed(42) at the beginning of our Python implementation. Hence, our noisy signal vector b was the same across all four experiment, across repeated runs of finding values for k or when the project is rerun for other reasons (e.g. code changes).

Since we used a random seed for the Gaussian noise, the clean signal as well as the noisy signal are identical across all four experiments. The assignment sheet does not state exactly what kind of information should be plotted, only stating "Create a plot for each experiment showing the clean signal, the noisy signal and the results from both algorithms.". Hence, we decided to create one plot showing the difference between the clean and noisy signal in all its dimensions since the noisy signal is identical four all experiments. This gives an idea about how noisy the noisy signal is. Then we added one plot for each method where we show the convergence of the cost function for all four methods and all four experiments, i.e. one plot for each experiment using four different colours (the same colors as in task 2 for easy guidance) for the respective methods.

Please note that since the projected gradient method and our custom projected gradient method are extremely close in performance, their reconstructed signals are shown on top of each other in the four experiment plots and thus the green line for the signals of the projected gradient methods are hidden directly below the red reconstructed signal of our custom projected gradient method.

1.6.1 Remarks regarding comparision of different experiments:

Additionally we compared the solution vector (hence the optimal found solution by each method) and compared them with each other. In order to do so we computed the element-wise difference between the solution vectors, took the absolute value (in order to get the true difference) and chose the largest value. Hence, the reported values denote the largest difference found in any dimension between the two solution vectors. These largest differences are denoted at the end of each experiment before we start the corresponding discussion section.

Suitable number of iterations (value k) was evaluated independently for each of the three methods. Then we compared the method outcomes based on these chosen k values. The chosen values for k were based directly on finding suitable convergence of the objective function (i.e. after which point the objective function does not change drastically any more).

For all methods and four experiments we tracked the history of the vector x_k in each iteration, the sum over all dimensions of x (in order to ensure that the unit simplex constraint is maintained over all iterations) and the value of the objective function (to verify convergence). We decided to also include the Frank Wolfe algorithm using the exact line search from task 2 and our custom developed projected gradient method in the comparison here.

The start vectors always have value 1 in their first dimension and the rest zeros. We kept them the same over all experiments in order to have fair comparisons (same principle as to why we chose to use a fixed random seed for the gaussian noise added to the signal).

Since the assignment sheet does not specify clearly whether the different values for the iterations k should be chosen individually only for the different methods or also for the different experiments we decided to evaluate suitable values for k for all methods and experiments individually (i.e. 4 methods and 4 experiments results in a total of 16 different k values). A reasonable comparison between experiments is still possible when different k values are chosen since we carefully tried out different values for k for all 16 permutations to choose the smallest values after which the algorithm hardly shows any further improvements to the cost function.

1.6.2 experiment a:

$$x_{k_0} = \begin{pmatrix} 1. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \end{pmatrix} \tag{23}$$

- projected gradient method:
  k = 30 cost = 0.09183285852195043
- custom projected gradient method:
  k = 30 cost = 0.0918328585219498
- Frank Wolfe method:
  k = 150 cost = 0.01078178422547727
- Frank Wolfe using exact line search:
  k = 30 cost = 0.009450083852713264
- Discussion of experiment a:
  The first experiment denotes our baseline after which we will investigate the different effects of noise and number of iterations. The regular Frank Wolfe method had the highest cost function and needed way longer to converge compared to its counterparts. Both the projected gradient methods behave nearly identical, only very slightly varying in their objective cost function value. The Frank Wolfe variation using exact line search converges way quicker than the regular Frank Wolfe which is something to be expected based on the convergence discussed in the lectures. The cost function value is only slightly lower.

1.6.3 experiment b:

$$x_{k_0} = \begin{pmatrix} 1. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \end{pmatrix} \tag{24}$$

- projected gradient method:
  k = 6 cost = 0.08088440817654877
- custom projected gradient method:
  k = 6 cost = 0.08088440817654861
- Frank Wolfe method:
  k = 6 cost = 0.06854315465292803
- Frank Wolfe using exact line search:
  k = 5 cost = 0.02613355450256772
- Discussion of experiment b:
  In this experiment the deviation is trippled (hence the variation is 9 times as high). Contrary to our expectations convergence happened way quicker after very few iterations. All algorithms converged similarly quickly, but the Frank Wolfe method using exact line search is able to lower the cost function value significantly lower. Even very high values of k (e.g. 300 and above) did not decrease the cost function values in any reasonable way. While higher variance seems to lead to the method converging way quicker, the cost function value could not be lowered as much when using much lower variance in experiment a. Latter described effect is something we expected, i.e. higher signal noise leading to the optimization methods not performing as well on minimizing the objective function. Overall, Frank Wolfe using exact line search showed by far the best result in reconstructing the signal.

1.6.4 experiment c:

$$x_{k_0} = \begin{pmatrix} 1. \\ 0. \\ \dots \\ 0. \\ 0. \end{pmatrix} \tag{25}$$

- projected gradient method:
  k = 30 cost = 0.09492405235996453
- custom projected gradient method:
  k = 30 cost = 0.09492405235996415
- Frank Wolfe method:
  k = 20 cost = 0.025115206060776652
- Frank Wolfe using exact line search:
  k = 15 cost = 0.01556845882560676
- Discussion of experiment c:
  Experiment c denotes a high dimensional version of experiment a, i.e. low variance with high dimension. Both projected gradient methods showed convergence behavior similar to experiment a and could not match the fast convergence in experiment b when we used a high variance. Their values of the objective cost function also were extremely close to experiment a. Using very high k values hardly improved the result, so we decided to keep k=30 here. Up to this number of iterations the objective cost function improved notably. The projected gradient methods don't seem to benefit from adding more dimensions for a simple problem like this. We assume that this would change if we were to use more dimensions to a much more complex problem than discussed here.

  The higher dimensions help the regular Frank Wolfe algorithm to converge much faster than in experiment a where we observed huge improvements until 150 iterations. Surprisingly, adding more dimensions increased the value of the cost objective function here compared to experiment a by doubling its value. We didn't manage to get a similar low cost objective function value in experiment c using any other number of iterations either. We assumed that higher dimensions would only impact algorithm performance but not show visible negative impacts on the cost objective function. However, having both lower noise (i.e. variance) and higher dimensions results in a much lower cost objective function value than in experiment b which makes sense (less deviation from the clean signal and more room for optimization in higher dimensions).

  Frank Wolfe using exact line search takes similarly long to converge compared to experiment a (k=15 vs k=30 but cost objective function values

hardly changed there, something we take into account here), also the cost objective function value is quite similar. Increasing the iterations here to k=30 does not match the same low cost objective function value like in experiment a but it's not that far off. Hence, adding dimensions to this variant of Frank Wolfe used on this simple task does not show any improvements. But again as we discussed in the case of the projected gradient methods here this is heavily problem dependent. Even though Frank Wolfe using exact line search takes a bit longer to converge to a good cost objective value (k=15 vs k=5) compared to the higher variance experiment b, we obtain a very low cost objective function here. More dimensions grant room for better optimization if necessary but we believe that the major impact here is simply having much lower amounts noise applied since the other methods don't seem to benefit really from the added dimensions for this simple task. Also note that even if you lower the iterations to k=5 here as well like in experiment b we would still get a cost = 0.019992237228846462 which is much lower than observed in experiment b. We simply chose k=15 here since the cost objective function still improves greatly up to this point.

Both Frank Wolfe algorithms perform really well, but the version using exact line search outperforms using a predefined diminishing step size.

1.6.5 experiment d:

$$x_{k_0} = \begin{pmatrix} 1. \\ 0. \\ 0. \\ 0. \\ 0. \end{pmatrix} \tag{26}$$

- projected gradient method:
  k = 6 cost = 0.23738813312003557
- custom projected gradient method:
  k = 6 cost = 0.2373881331200357
- Frank Wolfe method:
  k = 30 cost = 0.1643405808737765
- Frank Wolfe using exact line search:
  k = 6 cost = 0.16280832197168832
- Discussion of experiment d:
  This last experiment denotes a scenario where we investigate how algorithm performance is affected when lowering the dimensions to a very low value. Based on the problem at hand performance might be affected heavily. We can immediately note that all four methods are strongly negatively impacted by using only 5 dimensions. Their cost objective function values

are much worse than in all other experiments. It seems like d=15 like in the first two experiments might be a good sweet spot for choosing dimensionality which is adequate for the task at hand. This result can not be transferred to other problems using the same methods and can only be said here. Convergence happens really fast, in general optimization methods using low dimensionality for function approximation converges in fewer iterations. This is exactly what we are seeing here. Even if we choose hundreds of iterations the objective value function shows hardly any decrements any more. Summarized we can note that d=5 is insufficient for this signal reconstruction.

2 Task 2

2.1 Exact line search

$$\tau_k \in \arg\min_{\tau\in[0,1]} f(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)) \tag{27}$$

From (6):

$$\begin{aligned}
f(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)) &= \frac{1}{2}(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k))^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)) \\
&\quad - 2\boldsymbol{b}^T \boldsymbol{A}(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)) + \boldsymbol{b}^T\boldsymbol{b} && (28)\\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}\tau(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x}_k \\
&\quad + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}\tau(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad - 2\boldsymbol{b}^T \boldsymbol{A}\boldsymbol{x}_k - 2\boldsymbol{b}^T \boldsymbol{A}\tau(\boldsymbol{y}_k - \boldsymbol{x}_k) + \boldsymbol{b}^T\boldsymbol{b}) && (29)\\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x}_k + 2\boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}\tau(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad + \tau^2(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad - 2\boldsymbol{b}^T \boldsymbol{A}\boldsymbol{x}_k - 2\boldsymbol{b}^T \boldsymbol{A}\tau(\boldsymbol{y}_k - \boldsymbol{x}_k) + \boldsymbol{b}^T\boldsymbol{b}) && (30)\\
&= \frac{1}{2}(\boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{b}^T\boldsymbol{b} - 2\boldsymbol{b}^T \boldsymbol{A}\boldsymbol{x}_k \\
&\quad + 2\tau(\boldsymbol{x}_k^T \boldsymbol{A}^T - \boldsymbol{b}^T)\boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad + \tau^2(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)) && (31)
\end{aligned}$$

$$\begin{aligned}
\nabla_\tau f(\boldsymbol{x}_k + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)) &= \frac{1}{2}(2(\boldsymbol{x}_k^T \boldsymbol{A}^T - \boldsymbol{b}^T)\boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad + 2\tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)) && (32)\\
&= (\boldsymbol{x}_k^T \boldsymbol{A}^T - \boldsymbol{b}^T)\boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) \\
&\quad + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) && (33)
\end{aligned}$$

$$(\boldsymbol{x}_k^T \boldsymbol{A}^T - \boldsymbol{b}^T)\boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) + \tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) = 0 \tag{34}$$

$$\tau(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) = \boldsymbol{b}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) - \boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) \tag{35}$$

$$\tau = \frac{\boldsymbol{b}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) - \boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)}{(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)} \tag{36}$$

$$\tau_k = \begin{cases} 0 & \tau < 0 \\ \dfrac{\boldsymbol{b}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k) - \boldsymbol{x}_k^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)}{(\boldsymbol{y}_k - \boldsymbol{x}_k)^T \boldsymbol{A}^T \boldsymbol{A}(\boldsymbol{y}_k - \boldsymbol{x}_k)} & 0 \leq \tau \leq 1 \\ 1 & \tau > 1 \end{cases} \tag{37}$$

2.2 Image Representation

Apart from the three required plots for the ground truth image, the projected gradient method and the Frank Wolfe method (using the predefined diminishing step size as defined in algorithm 2 of the assignment sheet) we added two additional plots, one for our custom derived projected gradient method and one for the Frank Wolfe method using the exact line search. This allows simple visual inspection in the report figures. Our plot for the progression of the objective function also shows the progression of the objective function value of all four methods (projected gradient method, our custom projected gradient method, frank wolfe using a predefined diminishing step size, frank wolfe using exact line search). We denoted them using different colors and using their respective names in the label.

We were asked to find the lowest dimension d for which the different methods are able to create a good reconstruction of the clean image. We needed d = 33 in order to do so. Any values below did not provide a sufficient reconstruction for the two projected gradient methods (the frank wolfe ones started to look somewhat decent starting from d = 20). Any values above d = 33 did not yield any noticeable improvements. Even d = 50 and above looked pretty much identical to the naked eye but computation times increased dramatically. Increasing the parameter d influences the algorithm computation times quadratically based on the dimensions used in the matrices and vectors.

When looking at the convergence of the different methods we can note that Frank Wolfe using the exact line search converges extremely quickly (as noted in the lecture: exact line search might be difficult to implement and to compute its necessary step depending on the objective function which needs to be minimized). However, the overall minimization of the objective cost function over the predefined number of iterations k = 1500 actually is behind the reconstruction result of the projected gradient methods. This can also be noted clearly

when looking at the reconstructed images themselves. The reconstructed versions of the projected gradient methods look much cleaner in comparison and are extremely close to the original clean ground truth image.

The projected gradient methods converge slower in the beginning compared to the exact line search approach but steadily improve when more iterations are used resulting in the overall best reconstructed image. We tracked some additional metrics in the python code such as the x_k solution vectors over all iterations, the differences between these and the different methods and whether the unit simplex requirements are fulfilled. There we can not only see that no conditions are violated but also that the two projected gradient methods perform pretty much identical. Please note that only one line can be visually seen in the plot since both projected gradient methods performed numerically pretty identical, hence the green progression line of the objective function can not be seen here right under the dark orange one. This is not a mistake of not plotting the green one. The projection itself requires more effort regarding its implementation and also the projection step can be computationally very costly. However, here we only were asked to plot the progression over the number of iterations not taking into account the exact time the methods need to calculate one iteration.

In comparison the Frank Wolfe method using the predefined diminishing step size denoted in algorithm 2 of the assignment sheet both convergence of the minimization of the cost function as well as the final optimization result after the fixed number of iterations k = 1500 performed worst. As mentioned in the discussion about choosing an adequate parameter for the dimension d we should note that the Frank Wolfe method required lower dimensions in order to be able to provide good reconstruction results.

3 Projection on Hyperplane

This section show how the projection from a d-dimensional space to a hyperplane is computed. This is needed for the custom projected gradient.

$$\boldsymbol{p} = \text{point on hyperplane} \tag{38}$$

$$\boldsymbol{z} = \text{point to project on hyperplane} \tag{39}$$

$$\boldsymbol{w} = \text{vector normal to hyperplane} \tag{40}$$

$$\boldsymbol{h}_i = \text{the direction vectors of the hyperplane} \tag{41}$$

The closest point to $\boldsymbol{z}$ on the hyperplane is reached by following $\boldsymbol{w}$ until the hyperplane is reached.

$$\boldsymbol{z} + \lambda_w \boldsymbol{w} = \boldsymbol{p} + \lambda_1 \boldsymbol{h}_1 + \lambda_2 \boldsymbol{h}_2 + \cdots + \lambda_{d-1} \boldsymbol{h}_{d-1} \tag{42}$$

$$\boldsymbol{z} - \boldsymbol{p} = \lambda_1 \boldsymbol{h}_1 + \lambda_2 \boldsymbol{h}_2 + \cdots + \lambda_{d-1} \boldsymbol{h}_{d-1} - \lambda_w \boldsymbol{w} \tag{43}$$

$$\begin{pmatrix} z_1 - p_1 \\ z_2 - p_2 \\ \dots \\ z_d - p_d \end{pmatrix} = \begin{pmatrix} | & | & & | & | \\ \boldsymbol{h}_1 & \boldsymbol{h}_2 & \dots & \boldsymbol{h}_{d-1} & -\boldsymbol{w} \\ | & | & & | & | \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_{d-1} \\ \lambda_w \end{pmatrix} \tag{44}$$

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_{d-1} \\ \lambda_w \end{pmatrix} = \begin{pmatrix} | & | & & | & | \\ \boldsymbol{h}_1 & \boldsymbol{h}_2 & \dots & \boldsymbol{h}_{d-1} & -\boldsymbol{w} \\ | & | & & | & | \end{pmatrix}^{-1} \begin{pmatrix} z_1 - p_1 \\ z_2 - p_2 \\ \dots \\ z_d - p_d \end{pmatrix} \tag{45}$$

$$\boldsymbol{z}_{projected} = \boldsymbol{z} + \lambda_w \boldsymbol{w} \tag{46}$$