Numerical Linear Algebra Project 1

Peter Weber

November 13, 2017

The computational tasks are implemented in the attached file Project1_PeterWeber.py. At the end of the file, just after $if\_\_name\_\_ == "\_\_main\_\_"$: the functions, which execute the tasks, are called. All parameters are already set to execute the entire file, but chosen such that the file executes relatively fast i.e. the maximum matrix dimensions are set to relatively small values (for the tasks in which we are supposed to measure execution times). This can be changed by setting the n_max parameter to a different value. In order to set the number of printouts, one has to set the $modulo\_print\_$ parameter. For example, setting $modulo\_print\_ = 10$ means that the results of the computations of the matrices with sizes 10, 20, 30,... are printed. The file also prints the convergence conditions in every iteration.
In order to read the files, my script looks for the folders "$./optpr1/$" and "$./optpr2/$". Another point to mention here is that I needed to rename the $g.dad$ files in optpr1 and optpr2, because my operating system would not distinguish between $G.dad$ and $g.dad$. I renamed $g.dad$ to $g\_.dad$. In principle, my script should handle both cases $g.dad$ and $g\_.dad$. I mention this here just in case.

1 Theoretical task: T1

We have to show, that the nonlinear system of equations

$$\begin{aligned} Gx + g \quad -A\gamma \quad -\lambda \qquad &= 0 \\ -A^T x + b \qquad\qquad &= 0 \\ -C^T x + d \qquad\qquad +s \ &= 0 \\ s_i \lambda_i &= 0, \quad i = 1, ..., m \end{aligned} \tag{1.1}$$

represented as $F(z) = 0$ ($F : \mathbb{R}^N \to \mathbb{R}^N$, $z = (x, \gamma, \lambda, s)$) can be reduced to a linear system of equations given by

$$M_{kkt} \cdot \delta_z = -F(z_0), \tag{1.2}$$

where $\delta_z = (\delta_x, \delta_\gamma, \delta_\lambda, \delta_s) \in \mathbb{R}^N$ is the Newton Step such that $z_1 = z_0 + \delta z$,

$$M_{kkt} = \begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda, \end{bmatrix}$$

and -$F(z_0) = (-r_L, -r_A, -r_C, -r_S)$ the righthand vector.

The standard Newton step in N dimensions is given by

$$z_1 = z_0 - \frac{F(z_0)}{\nabla F(z_0)}. \tag{1.3}$$

This can be brought into the form of Eq. (1.2) by simple algebraic manipulation and setting $\delta_z = z_1 - z_0$. It remains to be shown, that $\nabla F(z_0) = M_{kkt}$ where $\nabla = (\partial_x, \partial_\gamma, \partial_\lambda, \partial_s)$. The gradient of $F(z)$ is then written as

$$\nabla F = \begin{bmatrix} \partial_x F_1 & \partial_\gamma F_1 & \partial_\lambda F_1 & \partial_s F_1 \\ \partial_x F_2 & \ddots & & \vdots \\ \partial_x F_3 & & \ddots & \vdots \\ \partial_x F_4 & \cdots & \cdots & \partial_s F_4 \end{bmatrix} = \begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{bmatrix} = M_{kkt},$$

with F_1 being the first row (F_2 the second etc...) of the non-linear system of equations (Eq. 1.1). On the other, if we start with the Lagrangian given in the assignment sheet, then we realize that the gradient of the Lagrangian $\nabla L(z)$ is $F(z)$ and the Hessia is M_{kkt}

$$\nabla F = \nabla(\nabla L) = \nabla \begin{bmatrix} \partial_x L \\ \partial_\gamma L \\ \partial_\lambda L \\ \partial_s L \end{bmatrix} = \begin{bmatrix} \partial_{xx} L & \partial_{x\gamma} L & \partial_{x\lambda} L & \partial_{xs} L \\ \partial_{\gamma x} L & \partial_{\gamma\gamma} L & & \vdots \\ \partial_{\lambda x} L & & \partial_{\lambda\lambda} & \vdots \\ \partial_{sx} L & \cdots & \cdots & \partial_{ss} L \end{bmatrix} = \begin{bmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{bmatrix} = M_{kkt}.$$

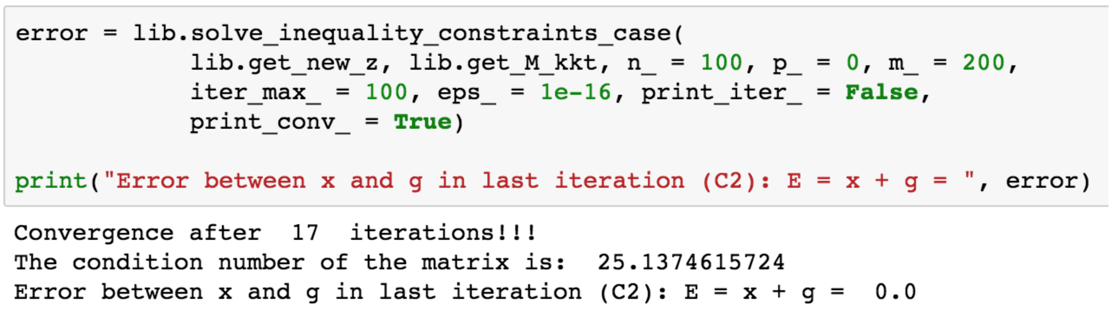
Therefore, solving

$$\nabla(\nabla L) \cdot \delta_z = \nabla F \cdot \delta_z = M_{kkt} \cdot \delta_z = -F(z_0) \tag{1.4}$$

yields the prediction step.

2 Computational task: C1

The stepsize substep is implemented by the function $Newton\_step(lamb0, dlamb, s0, ds)$ provided in the assignment sheet.



```
error = lib.solve_inequality_constraints_case(
            lib.get_new_z, lib.get_M_kkt, n_ = 100, p_ = 0, m_ = 200,
            iter_max_ = 100, eps_ = 1e-16, print_iter_ = False,
            print_conv_ = True)

print("Error between x and g in last iteration (C2): E = x + g = ", error)
```

```
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.1374615724
Error between x and g in last iteration (C2): E = x + g =  0.0
```

Figure 3.1: The output of the computation in task C2.

3 Computational task: C2

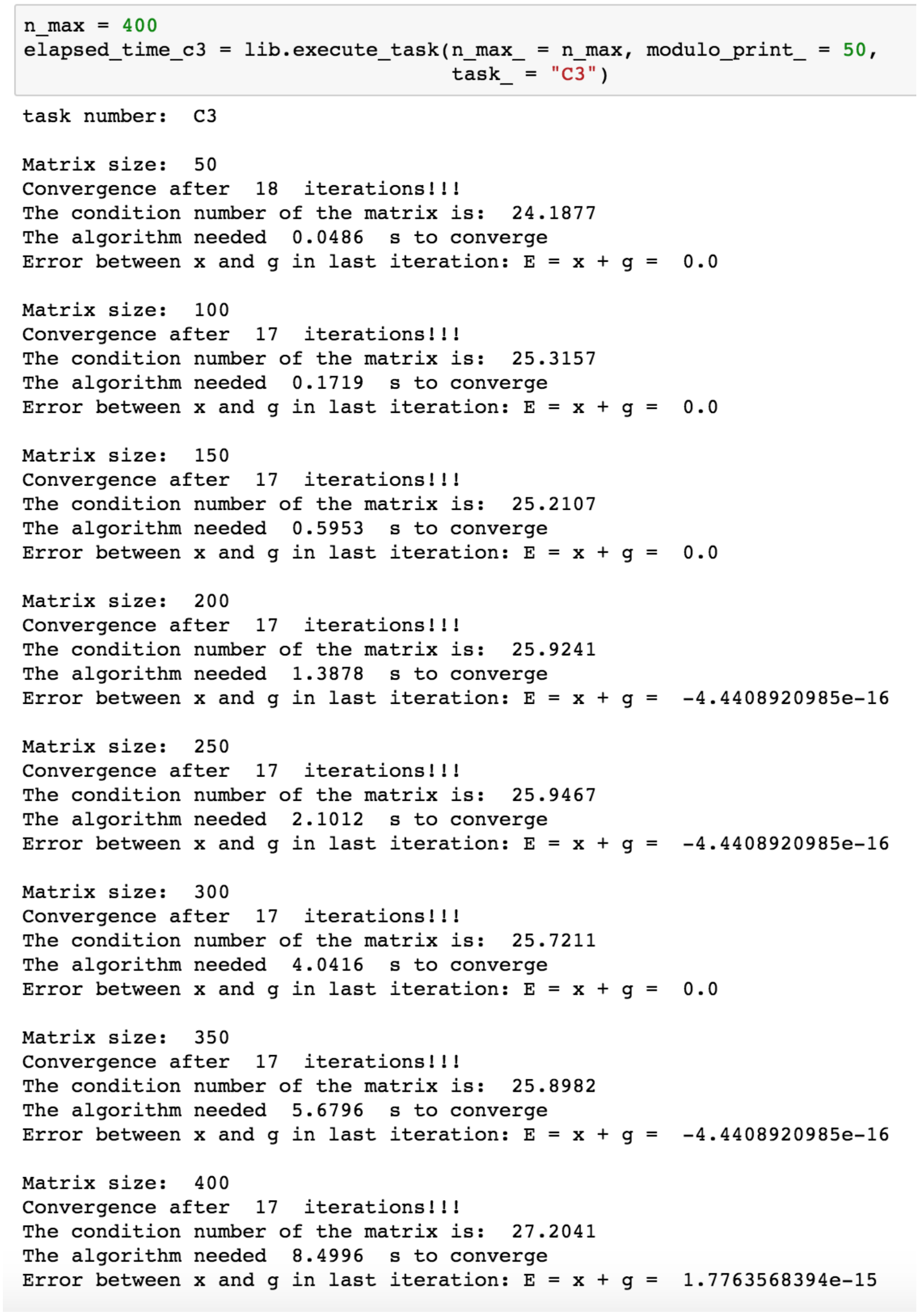
In this part of the assignment, the linear system to solve in every iteration of the algorithm is

$$M_{kkt} \cdot \delta_z = \begin{bmatrix} G & -C & 0 \\ -C^T & 0 & I \\ 0 & S & \Lambda \end{bmatrix} \cdot \begin{bmatrix} \delta_x \\ \delta_\lambda \\ \delta_s \end{bmatrix} = \begin{bmatrix} -r_1 \\ -r_2 \\ -r_3 \end{bmatrix}, \tag{3.1}$$

which is the inequality constraints case. I wrote a programm that executes the 6 steps with the provided input values. Apart from several helper functions, this part is implemented in the functions get_new_z and $solve\_inequality\_constraints\_case$.
The output of the computation for $n = 100$, $p = 0$, and $m = 2n$ is shown in Fig. 3.1, where the linear system is solved using np.linalg.solve and the condition number is computed by np.linalg.cond. As can be seen in the figure, the algorithm converges after 17 iterations i.e. either μ, or r_L, or r_C is below the machine precision $\epsilon = 10^{-16}$, the condition number is about 25, and the expected result for $x = -g$ is achieved with a precision below the machine precision.

4 Computational task: C3

The objective of this task is to compute the time the algorithm needs until it converges vs. the size of the linear system to solve. Figure 4.1 shows a similar output as in the previous task. The output of the computation is plotted only if the matrix size is a multiple of 50. It is also important to mention, that the condition number is only computed when it is printed to the console, otherwise I don't compute the condition number in order to save execution time. In this problem, the condition number is nearly independent of the matrix size indicating that the precision of the solution vector is also nearly independent of the size of the linear system. In fig. 4.2, I plot the algorithm convergence time as function of the matrix size. The fit for large n has the functional form $f(n) \propto n^3$ confirming that Gaussian Elimination with Partial Pivoting, which is used by np.linalg.solve, indeed has n^3 flops.



```
n_max = 400
elapsed_time_c3 = lib.execute_task(n_max_ = n_max, modulo_print_ = 50,
                                   task_ = "C3")
```

```
task number:  C3

Matrix size:  50
Convergence after  18  iterations!!!
The condition number of the matrix is:  24.1877
The algorithm needed  0.0486  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  100
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.3157
The algorithm needed  0.1719  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  150
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.2107
The algorithm needed  0.5953  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  200
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.9241
The algorithm needed  1.3878  s to converge
Error between x and g in last iteration: E = x + g =  -4.4408920985e-16

Matrix size:  250
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.9467
The algorithm needed  2.1012  s to converge
Error between x and g in last iteration: E = x + g =  -4.4408920985e-16

Matrix size:  300
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.7211
The algorithm needed  4.0416  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  350
Convergence after  17  iterations!!!
The condition number of the matrix is:  25.8982
The algorithm needed  5.6796  s to converge
Error between x and g in last iteration: E = x + g =  -4.4408920985e-16

Matrix size:  400
Convergence after  17  iterations!!!
The condition number of the matrix is:  27.2041
The algorithm needed  8.4996  s to converge
Error between x and g in last iteration: E = x + g =  1.7763568394e-15
```

Figure 4.1: The output of the computation in task C3.

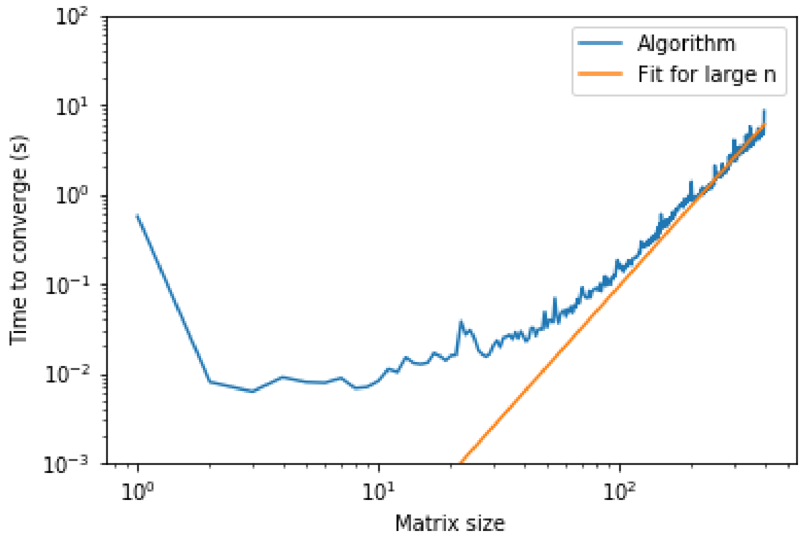


Figure 4.2: GEPP. Elapsed time of the algorithm until convergence as function of the size of the linear system that is solved. The fit has the functional form $f(n) \propto n^3$
.

5 Theoretical task: T2

5.1 Strategy 1

Taking as a starting point equation (3.1) one can isolate $\delta_s = (-r_3 - S\cdot\delta_\lambda)\cdot\Lambda^{-1}$ from the third row $S\cdot\delta_\Lambda + \Lambda\cdot\delta_s = -r_3$ and plug δ_s into the second row:

$$-C^T\delta_x + (-r_3 - S\cdot\delta_\lambda)\lambda^{-1} = -r_2$$

$$\begin{bmatrix}-C^T & -S\Lambda^{-1}\end{bmatrix}\begin{bmatrix}\delta_x\\ \delta_\lambda\end{bmatrix} = -r_2 + r_3\Lambda^{-1}.$$

Combining this with the first row yields the linear system

$$\begin{bmatrix}G & -C\\ -C^T & -S\Lambda^{-1}\end{bmatrix}\cdot\begin{bmatrix}\delta_x\\ \delta_\lambda\end{bmatrix} = -\begin{bmatrix}r_1\\ r_2 - r_3\Lambda^{-1}\end{bmatrix}.$$

The key idea behind the derivations lies in the fact that we obtain a symmetric matrix, which can be factorized using the LDL^T factorization. The linear system is symmetric since G, and $-S\Lambda^{-1}$ are symmetric, and the upper right block C is the transpose of the lower left block C^T. This factorization method is expected to be twice as fast as the PLU factorization (GEPP), and numerically more stable.

5.2 Strategy 2

Taking again as a starting point the system in equation (3.1) one can isolate

$$\delta_s = -r_2 + C^T\cdot\delta_x$$

from the second row and plug this into the third row, which gives

$$\delta_\lambda = S^{-1}(-r_3 + \Lambda r_2) - S^{-1}\Lambda C^T \delta_x.$$

Substituting both δ_s and δ_λ into the first row we obtain the system

$$\hat{G} \cdot \delta_x = -r_1 - \hat{r}$$

with $\hat{G} = G + CS^{-1}\Lambda C^T$ and $\hat{r} = -CS^{-1}(-r_3 + \Lambda r_2)$. Here, the matrix $\hat{G}$ is symmetric positive definite and therefore one can apply Cholesky factorization to solve the system. In order to show that $\hat{G}$ is symmetric positive definite, I first define $S^{-1}\Lambda = D = D^{1/2}D^{1/2}$ (D is diagonal) and $E = CD^{1/2}$. Then one can write $\hat{G} = G + EE^T$, which is symmetric positive definite (SPD) as G is semidefinite positive and a matrix multiplied with its transpose is always SPD. The cost of Cholesky is expected to be similar to the cost of LDL^T.

6 Computational Task: C4

6.1 Strategy 1

In this task, I implemented an LDL^T factorization in Python (function get_LDLT) and the forward and backward substitution to solve the system (function $solve\_LDLT\_system$). The function that implements the 6 steps is $get\_new\_z\_s1$.
Figure 6.1 shows the output of the computations. This implementation is almost two orders of magnitude slower than the PLU factorization from np.linalg.solve, which I attribute to the simple fact that an implementation in Python operating on large matrices will always lack speed compared to an implementation of the same algorithm in a low level language such as C. However, the condition number in this problem is also much larger, which is an indicator that the solution is numerically not stable. The convergence time as function of matrix size is plotted in Fig. 6.2, again with a fit having the functional form n^3.

6.2 Strategy 2

In this task, I used the np.linalg.cholesky implementation in Numpy to factorize the matrix. For the forward and backward substitution I wrote the function $solve\_cholesky\_system$, and the 6 steps are implemented in the function $get\_new\_z\_s2$.
The results of the computation are shown in figures 6.3 and 6.4.
A plot that compares the three methods to solve the test problem is shown in Fig. 6.5. Surprisingly, the Cholesky implementation is slightly slower than the GEPP implementation, which may arise from the fact that I my forward and backward substitution in the Cholesky case is not optimized. I already noted before that the LDL^T implementation is significantly slower than the other two, because it is implemented in Python and not in C.

7 Computational Task: C4

The first step of this subtask consists in reading the files. The functions used for that are called $read\_file$, $get\_matrices$, and $get\_vectors$. When reading the matrices one has to

```
n_max = 100
elapsed_time_c4_s1 = lib.execute_task(n_max_ = n_max, modulo_print_ = 20,
                                       task_ = "C4 strategy 1")
```

```
task number:  C4 strategy 1

Matrix size:  20
Convergence after  15  iterations!!!
The condition number of the matrix is:  3.35003658631e+18
The algorithm needed  0.3534  s to converge
Error between x and g in last iteration: E = x + g =  1.07032438468e-15

Matrix size:  40
Convergence after  16  iterations!!!
The condition number of the matrix is:  6.30147281506e+20
The algorithm needed  1.3487  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  60
Convergence after  18  iterations!!!
The condition number of the matrix is:  2.67500282578e+18
The algorithm needed  5.262  s to converge
Error between x and g in last iteration: E = x + g =  -1.33226762955e-15

Matrix size:  80
Convergence after  17  iterations!!!
The condition number of the matrix is:  4.88170063196e+21
The algorithm needed  9.5385  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  100
Convergence after  17  iterations!!!
The condition number of the matrix is:  4.82466442607e+18
The algorithm needed  12.5324  s to converge
Error between x and g in last iteration: E = x + g =  -4.4408920985e-16
```

Figure 6.1: The output of the computation in task C4 strategy 1.

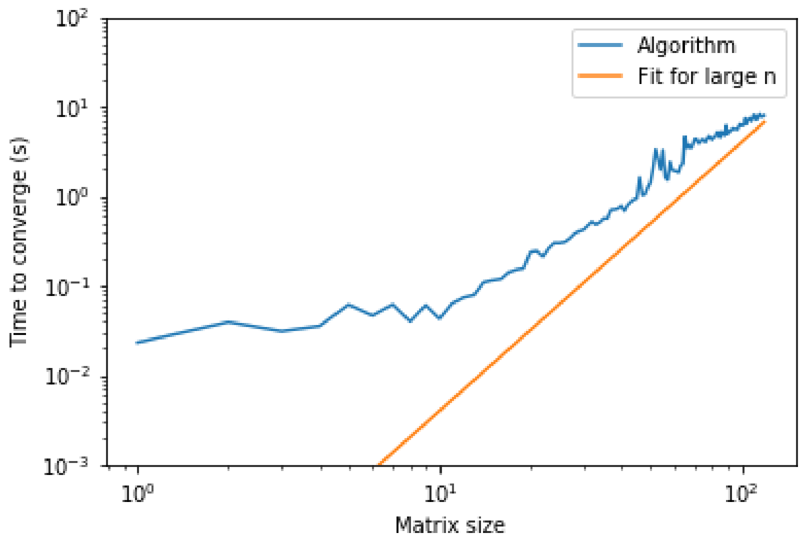


Figure 6.2: LDL^T. Convergence time versus matrix size. The fit has the functional form n^3.

take into account, that G is symmetric, meaning that one has to add to the raw matrix G its transpose and subtract the diagonal of G from the result.
The functions implementing this part are called get_new_z and $solve\_general\_case$. The solutions to both datasets optpr1 and optpr2 are shown in figure 7.1.

8 Theoretical Task: T3

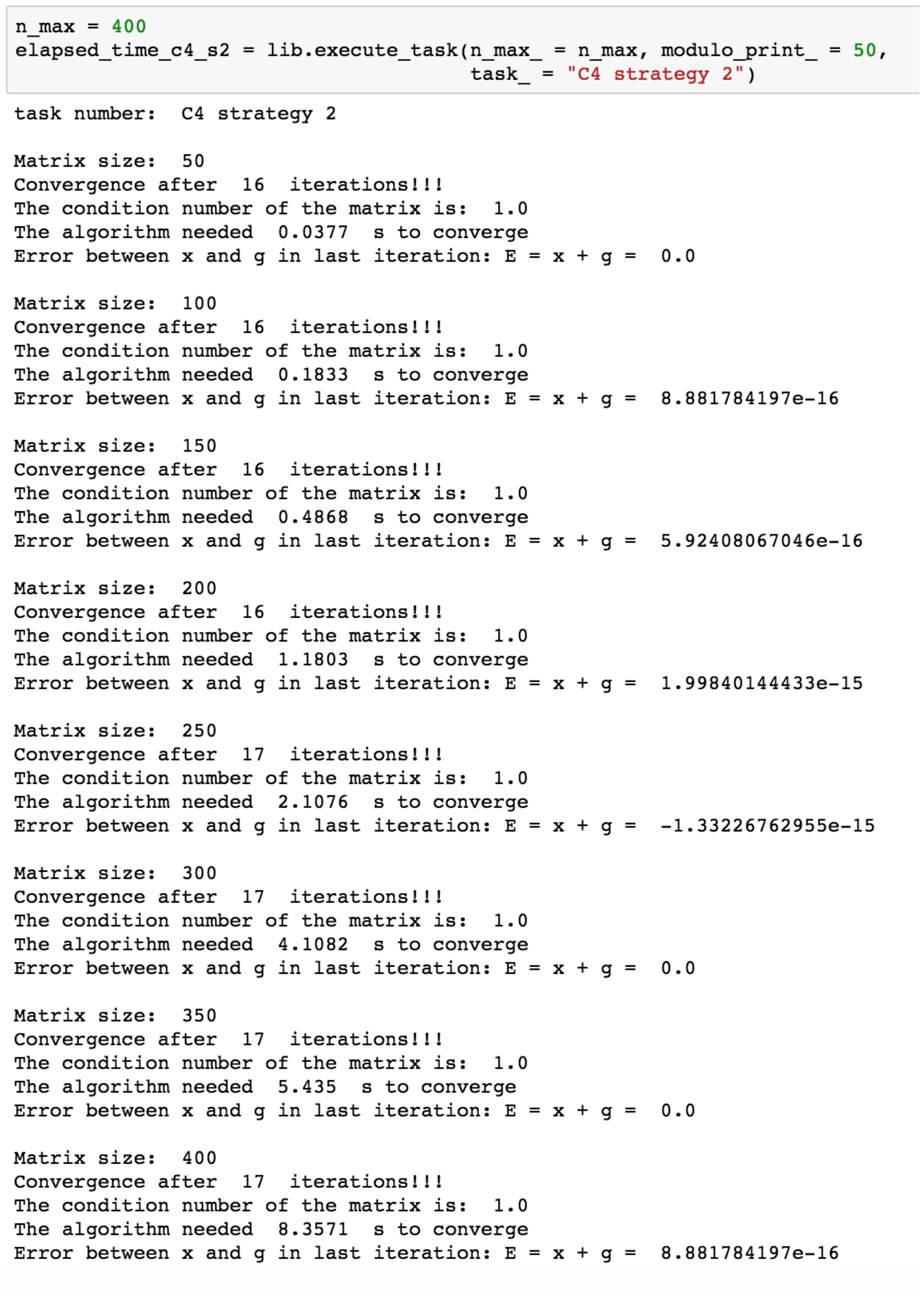
By isolating $\delta_s = \Lambda^{-1}(-r_4 - S \cdot \delta\lambda)$ from the fourth row of M_{kkt} and substituting it into the third row one obtains the following system of equations

$$\begin{bmatrix} G & -A & -C \\ -A^T & 0 & 0 \\ -C^T & 0 & -\Lambda^{-1} \cdot S \end{bmatrix} \cdot \begin{bmatrix} \delta_x \\ \delta_\gamma \\ \delta_\lambda \end{bmatrix} = \begin{bmatrix} -r_1 \\ -r_2 \end{bmatrix}$$

This system is symmetric as G and $-\Lambda^{-1} \cdot S$ are symmetric (the latter is a product of two diagonal matrices, which, of course, is symmetric), and the off diagonal blocks are the transposed when switching row and column indices.

9 Computational Task: C6

In the last part, I have tried to implement a Cholesky factorization of the matrix times its transpose (functions $get\_new\_z\_c6$ and $solve\_general\_case$), but the algorithm blows up after the 8 iterations with a non SPD matrix.



```
n_max = 400
elapsed_time_c4_s2 = lib.execute_task(n_max_ = n_max, modulo_print_ = 50,
                                      task_ = "C4 strategy 2")
```

```
task number:  C4 strategy 2

Matrix size:  50
Convergence after  16  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  0.0377  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  100
Convergence after  16  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  0.1833  s to converge
Error between x and g in last iteration: E = x + g =  8.881784197e-16

Matrix size:  150
Convergence after  16  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  0.4868  s to converge
Error between x and g in last iteration: E = x + g =  5.92408067046e-16

Matrix size:  200
Convergence after  16  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  1.1803  s to converge
Error between x and g in last iteration: E = x + g =  1.99840144433e-15

Matrix size:  250
Convergence after  17  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  2.1076  s to converge
Error between x and g in last iteration: E = x + g =  -1.33226762955e-15

Matrix size:  300
Convergence after  17  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  4.1082  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  350
Convergence after  17  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  5.435  s to converge
Error between x and g in last iteration: E = x + g =  0.0

Matrix size:  400
Convergence after  17  iterations!!!
The condition number of the matrix is:  1.0
The algorithm needed  8.3571  s to converge
Error between x and g in last iteration: E = x + g =  8.881784197e-16
```

Figure 6.3: The output of the computation in task C4 strategy 2.

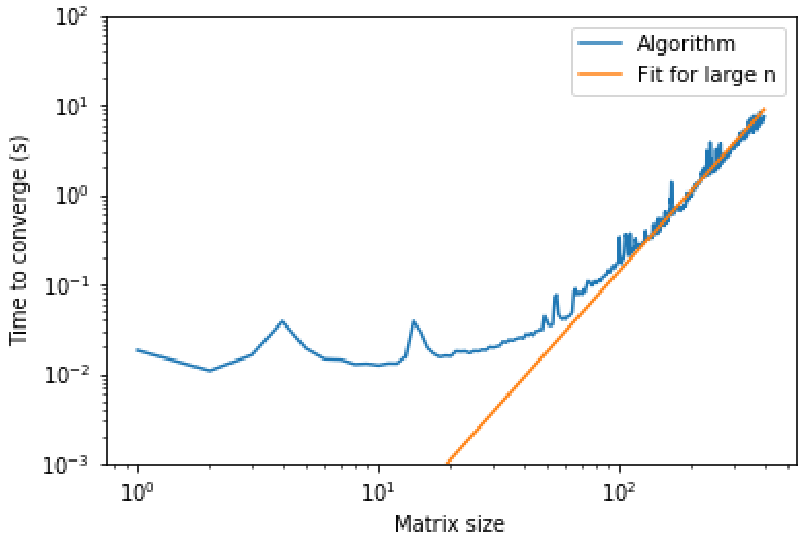


Figure 6.4: Cholesky. Convergence time versus matrix size. The fit has the functional form n^3.

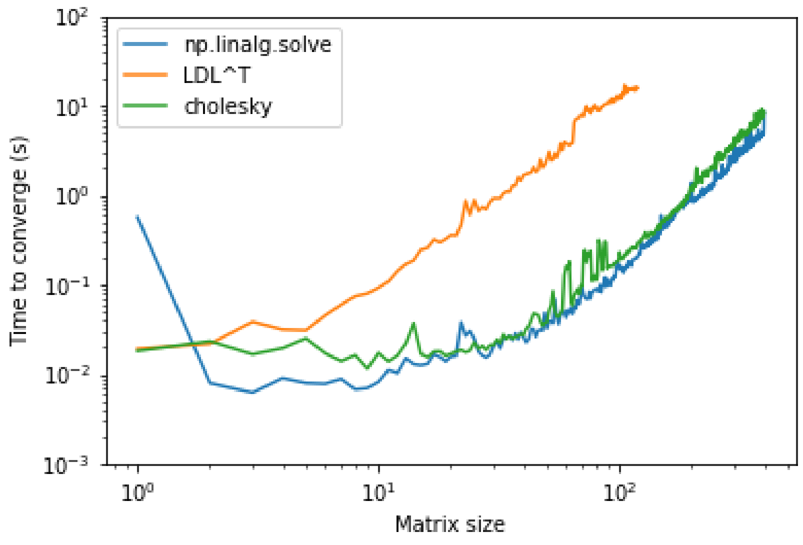
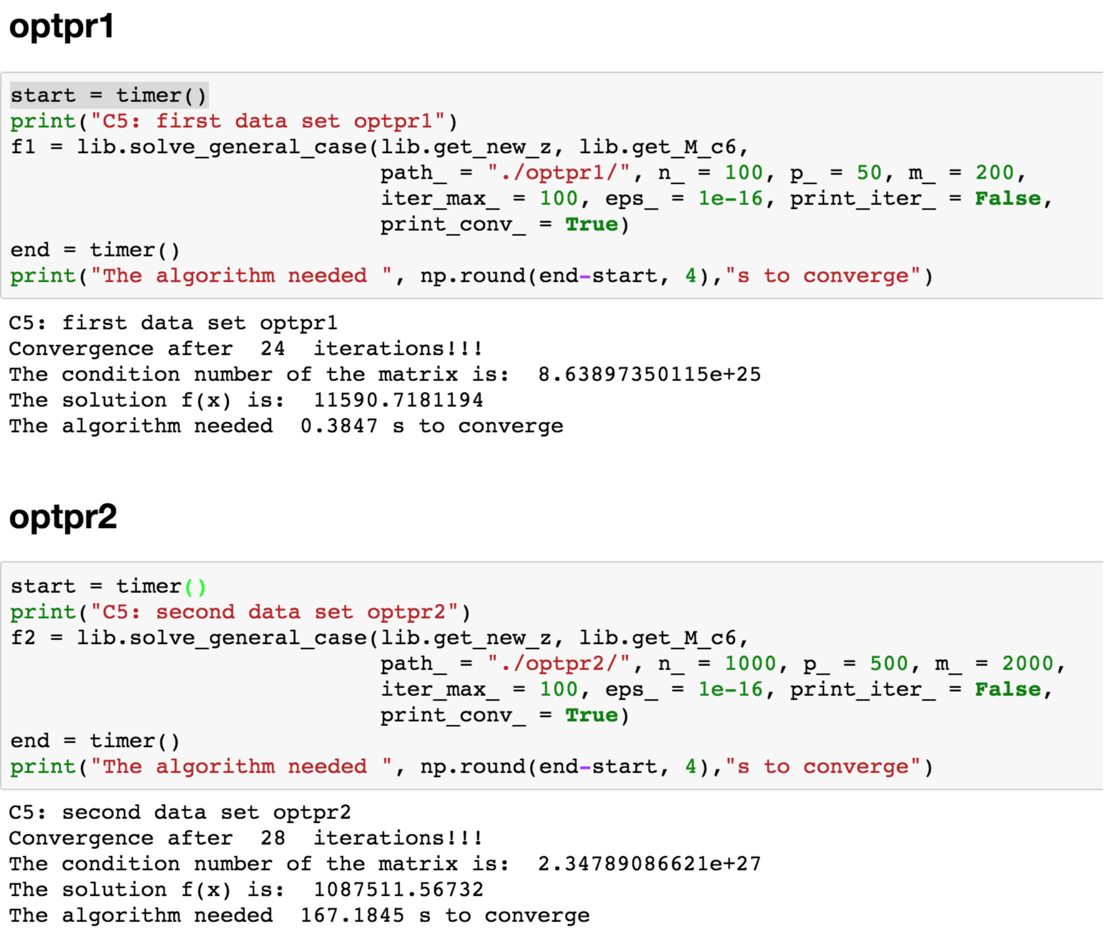


Figure 6.5: Convergence time versus matrix size.



optpr1

```python
start = timer()
print("C5: first data set optpr1")
f1 = lib.solve_general_case(lib.get_new_z, lib.get_M_c6,
                            path_ = "./optpr1/", n_ = 100, p_ = 50, m_ = 200,
                            iter_max_ = 100, eps_ = 1e-16, print_iter_ = False,
                            print_conv_ = True)
end = timer()
print("The algorithm needed ", np.round(end-start, 4),"s to converge")
```

```
C5: first data set optpr1
Convergence after  24  iterations!!!
The condition number of the matrix is:  8.63897350115e+25
The solution f(x) is:  11590.7181194
The algorithm needed  0.3847 s to converge
```

optpr2

```python
start = timer()
print("C5: second data set optpr2")
f2 = lib.solve_general_case(lib.get_new_z, lib.get_M_c6,
                            path_ = "./optpr2/", n_ = 1000, p_ = 500, m_ = 2000,
                            iter_max_ = 100, eps_ = 1e-16, print_iter_ = False,
                            print_conv_ = True)
end = timer()
print("The algorithm needed ", np.round(end-start, 4),"s to converge")
```

```
C5: second data set optpr2
Convergence after  28  iterations!!!
The condition number of the matrix is:  2.34789086621e+27
The solution f(x) is:  1087511.56732
The algorithm needed  167.1845 s to converge
```

Figure 7.1: The output of the computation in task C5.