Modeling and solving bundle adjustment problems

C Angla, Jean Bigeon, D Orban

Les Cahiers du GERAD ISSN: 0711–2440

Modeling and solving bundle adjustment problems

C. Angla,
J. Bigeon, D. Orban

G–2020–42

August 2020

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2020
– Bibliothèque et Archives Canada, 2020

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2020
– Library and Archives Canada, 2020

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Modeling and solving bundle adjustment problems

Célestine Angla [a,b,c]

Jean Bigeon [a,b,c]

Dominique Orban [a,d]

[a] *GERAD, Montréal (Québec), Canada, H3T 2A7*

[b] *Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées, Université Grenoble-Alpes, 38000 Grenoble, France*

[c] *Laboratoire G-SCOP, Université Grenoble-Alpes, 38031 Grenoble, France*

[d] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal (Québec) Canada, H3C 3A7*

jean.bigeon@grenoble-inp.fr
dominique.orban@gerad.ca

August 2020
Les Cahiers du GERAD
G–2020–42

Abstract: We present a modeling of bundle adjustment problems in Julia, as well as a solver for nonlinear least square problems (including bundle adjustment problems). The modeling uses NLPModels Julia's library and computes sparse Jacobians analytically. The solver is based on the Levenberg-Marquardt algorithm and uses QR or LDL factorization, with AMD or Metis permutation algorithm. The user can choose to use normalization and line search. Our experimental results contain comparison of the several versions of the solver and comparison with Scipy's least_square function and Ceres solver on the test problems given in [26]. We show that our solver is quite competitive with Scipy's solver and Ceres solver in terms of convergence, and that it is in average two times faster than Scipy's solver and three times slower than Ceres. However, the advantage of our solver is that it is coded is Julia and thus allows the user to run it in several precisions in a very efficient way, in order to gain time and energy (in small precisions) or accuracy (in big precisions).

Acknowledgments: This work constitutes the final report of C. Angla's internship at the G–SCOP laboratory in collaboration with GERAD. The supervisor at G–SCOP was J. Bigeon and the supervisor at GERAD was D. Orban.

Introduction

Given a set of images depicting a number of 3D points from different viewpoints, bundle adjustment problem consists in refining the 3D coordinates describing the scene geometry together with the parameters of relative motion and the optical characteristics of the camera(s) [9]. These problems are widely used in computer vision, as they are often the last step of feature-based 3D reconstruction. For instance, bundle adjustment is used to reconstruct the scenes taken by the Google Car, and give the rendering of Google Street View.

The goal of this internship was to model and solve bundle adjustment problems in Julia, to benefit of Julia's type system and have a code that can be run in a precision chosen by the user. Most languages (apart from Fortran and C++) only have simple (Float32) and double (Float64) precision, while in Julia other precisions, such as half and quadratic, are also available. Furthermore, the advantage of Julia over other languages is that, if a function is well written, and if you call it with arguments in a given precision, a version of the function for this given precision is automatically compiled. Thus, if used on a computer with a simple precision processor, operations will automatically be performed in simple precision (on standard computers simple precision is just a truncated double precision). So my solver could be ran in simple precision to obtain less accurate solutions, but faster and cheaper in terms of heat emissions by processors (which is more ecological). Or, on an architecture where several precisions are available, one may run my solver in simple precision until the stopping criteria defined in simple precision are satisfied, and then run again my solver, starting from the solution obtained before, until double precision stopping criteria are met, once again time and energy will be gained.

The team of my supervisor at GERAD (Polytechnique Montréal) has created several tools for modeling and solving continuous optimization problems in Julia [11]. Coding in Julia also allowed me to have access and make use of these tools which are very useful to model problems, perform matrix factorization, make benchmarks, etc.

Bundle adjustment problems can be modeled as a sum of the squares of the errors between the projection of the 3D points on the cameras and the observed 2D points. This kind of problem is called a non-linear least-square problem. The Levenberg-Marquardt algorithm has proven to be one of the most efficient method to solve this kind of problem, while quite easy to implement. This algorithm uses a damping factor, which is adjusted at each iteration. If the residuals (in our case, the vector of errors between the projections and the actual 2D points) decrease fast, a smaller value of the damping parameter can be used, bringing the algorithm closer to the Gauss–Newton algorithm, whereas if an iteration gives insufficient reduction in the residuals, the damping parameter can be increased, giving a step closer to the gradient-descent direction [16].

Because of its efficiency and ease of implementation, the Levenberg-Marquardt algorithm has been widely used to solve bundle adjustment problems [2, 7, 27]. Bundle adjustment problems can also be solved using the reduced camera system [4, 10, 27] which consists in rewriting the problem using the sparse block structure of the matrix J^TJ (where J is the Jacobian) into two smaller problems. Although it has proven to be very fast, we did not use this method, as we wanted to have a general solver for non-linear least square problems. In the Levenberg-Marquardt method, one can use a Cholesky matrix factorization [7, 22] or a QR factorization to solve what are called the "normal equations" of Levenberg-Marquardt [7, 22]. Those factorizations avoid to compute the inverse of big matrices, which is very long. Computing the factorization of those matrices can also be very long, but one can exploit their sparse structure and use sparse factorization methods such as [3] or [5].

In [27], Google researchers describe the Ceres solver [18], used in Google Street View. This is a solver for non-linear least-square problems, including bundle adjustment problems, coded in C++. It provides different methods (which are compared in [27]) to solve these problems. These researchers have also published datasets for bundle adjustment problems [26]. I will use Ceres solver to compare my solver with, as it is considered as a reference for solving bundle adjustment problems. I will also use their datasets to test my solver.

My objectives for this internship were the following:

- Understand the datasets given in [26] and create an interface in Julia to read them.
- Model those problems as non-linear least-square problems using Julia's libraries JuMP or NLPModels.
- Code a Julia solver for those problems, and compare it with other solvers (including Ceres-solver).

The sections of my master thesis are articulated around those objectives. In the first section, I will provide a description of the bundle adjustment problems datasets from [26]. In Section 2, I will explain how I modeled bundle adjustment problems, and in particular how I computed the Jacobian. In Section 3, I will describe my solver based on the Levenberg-Marquardt algorithm, and the two factorizations I used. Finally, in the last section, I will present the results of my solver and compare them with those of other solvers.

My code can be found on github at: `https://github.com/CelestineAngla/BundleAdjustment.jl`.

1 Understanding the datasets

The website [26] contains the bundle adjustment problem library, created by the Google researchers. The first step of my internship was to understand those datasets and then to create an interface to read them in Julia.

This problem library provides us with five datasets: Ladybug, Trafalgar Square, Dubrovnik, Venice, Final. And each one of those datasets contains several problems. Those datasets were obtained from two sources of data. The first source uses images captured at a regular rate using a Ladybug camera mounted on a moving vehicle. Image matching was done by exploiting the temporal order of the images and the GPS information captured at the time of image capture. The second source of data uses images downloaded from Flickr.com and matched to find common points. Those images were taken in Trafalgar Square and in the cities of Dubrovnik, Venice, and Rome.

Each bundle adjustment problem is given as a bzip2 compressed file. Each file contains the following:

- The first line contains the number of cameras, the number of points and the number of observations.
- The second block of lines contains, for each observation, the index of the camera used for this observation, the index of the 3D point observed, and the x and y coordinates of the 2D projection of this point on the camera.
- The third block of lines contains, for each camera, the vector $(r_x, r_y, r_z, t_x, t_y, t_z, f, k_1, k_2)$, where (r_x, r_y, r_z) is the Rodrigues vector representing the rotation of the camera, (t_x, t_y, t_z) are the coordinates of the translation of the camera, f is the focal length of the camera and (k_1, k_2) are radial distortion parameters of the camera. These are initial values.
- The fourth block of lines contains, for each point, its 3D coordinates (x, y, z). These are initial values.

The structure of those datasets is detailed in Appendix A.

I created a bash script that downloads all the datasets for the user, and separates them into five folders: Dubrovnik, Final, LadyBug, Trafalgar and Venice. I also coded a Julia function to read the files and store the data into matrices.

From the datasets, one can extract five matrices:

- $O \in \mathbf{R}^{N_{obs} \times 2}$ the matrix of observations where each line contains the 2D coordinates of the observed point.
- $CI \in \mathbf{R}^{N_{obs}}$ the vector of camera indices: for each observation k, $CI[k]$ gives the index of the camera used for this observation.

- $XI \in \mathbf{R}^{N_{obs}}$ the vector of point indices: for each observation k, $XI[k]$ gives the index of the 3D point observed in this observation.
- $C \in \mathbf{R}^{N_{cam} \times 9}$ the camera matrix where each line contains the parameters $(r_x, r_y, r_z, t_x, t_y, t_z, f, k_1, k_2)$ of the camera.
- $X \in \mathbf{R}^{N_{points} \times 3}$ the point camera where each line contains the 3D coordinates of the point.

2 Modeling bundle adjustment problems

In this part is described my two attempts to model bundle adjustment problems in Julia: with JuMP and then with NLPModels. To understand my modeling, the first two subsections provide a brief explanation of the camera projection formula and a mathematical definition of bundle adjustment problems.

2.1 Camera projection

A camera can be described by a vector $C = (r_x, r_y, r_z, t_x, t_y, t_z, k_1, k_2, f) \in \mathbf{R}^9$, where:

- $R = (r_x, r_y, r_z)$ is the Rodrigues rotation vector [17],
- $T = (t_x, t_y, t_z)$ is the translation vector,
- k_1 and k_2 are distortion coefficients,
- f is the focal length.

The rotation vector and the translation vector give us the relative position of the camera, while k_1, k_2 and f are its optical parameters.

Given a 3D point $X = (x, y, z)$ and a camera $C = (R, T, k_1, k_2, f)$, the 2D projection P of point X on camera C is given by [26] $P = P_3 \circ P_2 \circ P_1$, with:

$$\begin{cases} P_1(R, X, T) = rot(R, X) + T \\ P_2(X) = -\frac{1}{X.z} \begin{bmatrix} X.x \\ X.y \end{bmatrix} \\ P_3(X, f, k_1, k_2) = f \times r(X, k_1, k_2) \times X \end{cases},$$

where $rot(R, X)$ is the point X rotated using the Rodrigues vector $R = (r_x, r_y, r_z)$ [17]:

$$rot(R, X) = cos(\theta)X + sin(\theta)k \times X + (1 - cos(\theta))(k.X)k,$$

where $\theta = ||R||$ and $k = \frac{R}{\theta}$.

And $r(X, k_1, k_2) = 1.0 + k_1||X||^2 + k_2||X||^4$ is a function that computes a scaling factor to undo the radial distortion.

The first line of the projection formula computes the coordinates of the point in the camera frame. The second line transforms the 3D coordinates into 2D coordinates (the coordinates on the image of the camera). Finally, the last step of the projection undoes the radial distortion of the image, and takes into account the focal length of the camera.

2.2 Optimization problem

A bundle adjustment problem consists in finding the optimal camera parameters and 3D point coordinates that fit the observed 2D points. Thus, it can be written as a non-linear least-square problem like this [9]:

$$\min_{(X,C)} \sum_{i=1}^{N_{points}} \sum_{j=1}^{N_{cam}} v_{i,j} ||P(X_i, C_j) - x_{i,j}^{obs}||^2,$$

where

$$\begin{cases} X \text{ is the point matrix in which each line } X_i \text{ contains the 3D coordinates of the } i\text{-th point} \\ C \text{ is the camera matrix in which each line } C_j \text{ contains the parameters of the } j\text{-th camera} \\ v_{i,j} = 1 \text{ if point } i \text{ is observed on camera } j \text{ and 0 otherwise} \\ P \text{ is the projection of a point on a camera as descibed in the previous subsection} \\ x_{i,j}^{obs} \text{ is the 2D observation of point } i \text{ on camera } j. \end{cases}$$

Using the matrices described in the first section, we can rewrite the problem as:

$$\min_{(X,C)} \sum_{k=1}^{N_{obs}} ||p(X[XI[k]], C[CI[k]]) - O[k]||^2$$

Instead of summing over the cameras and 3D points, we sum over all observations, by getting the camera index and 3D point index from the matrices of indices.

2.3 Modeling with JuMP

The first way I tried to model these problems in Julia is using Julia's JuMP library [1]. I tried two types of modeling:

- $\min_x \frac{1}{2}||f(x)||^2$ ("direct modeling")
- $\min_x \frac{1}{2}||r||^2$ under $f(x) + r = 0$ ("residual modeling")

with $f(x) = ||p(X[XI[k]], C[CI[k]]) - O[k]||$.

The first problem I encountered when modeling the problems with JuMP is that it is not possible to use functions with non-scalar arguments or expressions with non-scalar variables. So it was quite complicated to use, as these problems are very large and thus it is inconvenient to model them without vectors and matrices. Moreover, functions like the norm function or the square root function are not easy to manipulate in JuMP. A good way to model these problems is to model the residuals as a 1D vector of size $2 \times nobs$ (one residual for the x coordinate and one for the y coordinate for each observation). This way of modeling makes more sense and would have avoided the norm function in f. Unfortunately, I did not manage to implement it this way as I would have needed to manipulate vectors.

The first way of modeling bundle adjustment problems with JuMP did not work, as JuMP does not allow to use square roots or norms in the objective function. The second way worked, and I managed to run an optimization algorithm (Ipopt) on these models. The first value of the objective seemed right (by comparing with an Python code I found that solves bundle adjustmet problems with Scipy [6]), and the objective decreased for a few iterations but then it began to rise again (Ipopt might not be a good algorithm to use on these problems).

Although JuMP does not seem do be adapted to handle bundle adjustment problems, it ensured me that I had well understood the problems and the datasets.

2.4 Modeling with NLPModels

I chose to use NLPModels [12] (and to abandon JuMP) to have more freedom for modeling and simpler models to use when building optimization algorithms. I build a new model type `BALNLPModel` (included in AbstractNLPModel). The constructor `BALNLPModels` takes the path of a dataset (from [26]) as input and models the bundle adjustment problems like this:

$$\begin{cases} min\ 0 \\ \text{under } r(x) = 0 \end{cases}$$

where $r \in \mathbf{R}^{2nobs}$ is the vector of residuals given by, $\forall k \in \{1, ..., nobs\}$: $r[2k-1:2k] = P(x) - (x_k^{obs}, y_k^{obs})$,

and $x = \begin{bmatrix} X_1 & ... & X_{npnts} & C_1 & ... & C_{ncam} \end{bmatrix} \in \mathbf{R}^{3\times npnts+9\times ncam}$

where $X_k = (x, y, z)$ and $C_k = (r_x, r_y, r_z, t_x, t_y, t_z, k_1, k_2, f)$.

When running an optimization algorithm on a `BALNLPModel`, I transform it into an non-linear least square problem (NLSModel [13]) using a function available in NLPModels called "FeasibilityResidual".

2.4.1 A new function to read the datasets

In order to create `BALNLPModels` faster, I rewrote a function to read the datasets to avoid making useless allocations. This function does not return a matrix for observations, a matrix for the camera parameters and a matrix for the 3D points coordinates anymore. Instead, it directly returns a 1D vector pt2d of size $2\times nobs$ (in which line $2k-1$ contains the x coordinate of observation k and line $2k$ contains the y coordinate of observation k) and a 1D vector x_0 containing the initial cameras parameters and 3D point coordinates as described above ($x = \begin{bmatrix} X_1 & ... & X_{npnts} & C_1 & ... & C_{ncam} \end{bmatrix} \in \mathbf{R}^{3\times npnts+9\times ncam}$). It still returns the camera indices and points indices vectors.

2.4.2 Computing the Jacobian of the residuals by hand

The main methods associated to my `BALNLPModel` are methods to compute the vector of residuals and the Jacobian. The function to compute the residuals is straightforward, as it is based on the camera projection formula described at the very begining of this section. The methods to compute the Jacobian are a bit more complex as I decided to compute the Jacobian analytically, that is to say: by hand. I made that choice as I wanted to have a sparse Jacobian and the only Julia module I found for sparse automatic differentiation was buggy.

In order to compute a sparse Jacobian for my new type `BALNLPModel`, I coded the functions "jac_structure!" (that computes the sparsity structure of the Jacobian) and "jac_coord!" (that computes the values to store in this structure). The function "jac_structure!" fills two vectors "rows" and "cols" with the indices of the non-zero values of the Jacobian. That is to say, if $row[k] = i$ and $cols[k] = j$ then $J_{i,j} \neq 0$. The function "jac_coord!" fills a vector "vals" which contains the values of the non-zero elements of the Jacobian. That is to say: $J_{row[k],cols[k]} = vals[k]$.

The residuals of the bundle adjustment problem are defined by, $\forall k \in \{1, ..., nobs\}$:

$$\begin{aligned} r_{2k-1}(x) &= P(x).x - x_k^{obs} = P(C_k, X_k).x - x_k^{obs} \\ r_{2k}(x) &= P(x).y - y_k^{obs} = P(C_k, X_k).y - y_k^{obs}, \end{aligned}$$

where $P(C_k, X_k)$ is the projection of the 3D point X_k on camera C_k (here k is the number of the observation and not the actual index of the camera or point, that is to say C_k is assimilated to $C_{cam\_index[k]}$, and similarly for X_k).

We have $P = P_3 \circ P_2 \circ P_1$, with:

$$\begin{cases} P_1(x, y, z, r_x, r_y, r_z, t_x, t_y, t_z) = cos(\theta) \begin{bmatrix} x \\ y \\ z \end{bmatrix} + sin(\theta) \begin{bmatrix} k_y z - k_z y \\ k_z x - k_x z \\ k_x y - k_y x \end{bmatrix} \\ \qquad\qquad + (1 - cos(\theta))(k_x x + k_y y + k_z z) \begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\ P_2(x, y, z) = -\frac{1}{z} \begin{bmatrix} x \\ y \end{bmatrix} \\ P_3(f, k_1, k_2, x, y) = f(1 + k_1(x^2 + y^2) + k_2(x^2 + y^2)^2) \begin{bmatrix} x \\ y \end{bmatrix} \end{cases}$$

with $\theta = \sqrt{r_x^2 + r_y^2 + r_z^2}$ and $\begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} = \frac{1}{\theta} \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}$.

The Jacobian of the residuals, J_r, is a block matrix like this:

$$J_r = J_P = \begin{bmatrix} JX_{1,1} & \dots & JX_{1,npnts} & JC_{1,1} & \dots & JC_{1,ncam} \\ \vdots & & \vdots & \vdots & & \vdots \\ JX_{nobs,1} & \dots & JX_{nobs,npnts} & JC_{nobs,1} & \dots & JC_{nobs,ncam} \end{bmatrix},$$

where:

- $JX_{i,j} = \begin{bmatrix} \frac{\partial P_i.x}{\partial x_j} & \frac{\partial P_i.x}{\partial y_j} & \frac{\partial P_i.x}{\partial z_j} \\ \frac{\partial P_i.y}{\partial x_j} & \frac{\partial P_i.y}{\partial y_j} & \frac{\partial P_i.y}{\partial z_j} \end{bmatrix}$
- $JC_{i,j} = \begin{bmatrix} \frac{\partial P_i.x}{\partial rx_j} & \frac{\partial P_i.x}{\partial ry_j} & \frac{\partial P_i.x}{\partial rz_j} & \frac{\partial P_i.x}{\partial tx_j} & \frac{\partial P_i.x}{\partial ty_j} & \frac{\partial P_i.x}{\partial tz_j} & \frac{\partial P_i.x}{\partial k1_j} & \frac{\partial P_i.x}{\partial k2_j} & \frac{\partial P_i.x}{\partial f_j} \\ \frac{\partial P_i.y}{\partial rx_j} & \frac{\partial P_i.y}{\partial ry_j} & \frac{\partial P_i.y}{\partial rz_j} & \frac{\partial P_i.y}{\partial tx_j} & \frac{\partial P_i.y}{\partial ty_j} & \frac{\partial P_i.y}{\partial tz_j} & \frac{\partial P_i.y}{\partial k1_j} & \frac{\partial P_i.y}{\partial k2_j} & \frac{\partial P_i.y}{\partial f_j} \end{bmatrix}$

J_r is sparse as $JC_{i,j}$ is non-zero if and only if $j = idx\_cam[i]$ and $JX_{i,j}$ is non-zero if and only if $j = idx\_pnt[i]$. For a given observation k, let $\tilde{J}_P[k]$ be the dense Jacobian of $J_P[k]$, that is to say the line number k of J_P where all the blocks of zeros have been removed:

$$\tilde{J}_P[k] = \begin{bmatrix} JX_{k,idx\_pnt[k]} & JC_{k,idx\_cam[k]} \end{bmatrix}.$$

As $P = P_3 \circ P_2 \circ P_1$, we have $\tilde{J}_P[k] = (\tilde{J}_{P_3} \circ P_2 \circ P_1) \times (\tilde{J}_{P_2} \circ P_1) \times \tilde{J}_{P_1}$, where $\tilde{J}_P[k] \in \mathbf{R}^{2\times 12}$, $\tilde{J}_{P_1} \in \mathbf{R}^{6\times 12}$, $\tilde{J}_{P_2} \in \mathbf{R}^{5\times 6}$, $\tilde{J}_{P_3} \in \mathbf{R}^{2\times 5}$, and:

- $\tilde{J}_{P_1} = \begin{bmatrix} \frac{\partial P_1.x}{\partial x} & \frac{\partial P_1.x}{\partial y} & \frac{\partial P_1.x}{\partial z} & \frac{\partial P_1.x}{\partial r_x} & \frac{\partial P_1.x}{\partial r_y} & \frac{\partial P_1.x}{\partial r_z} & & \\ \frac{\partial P_1.y}{\partial x} & \frac{\partial P_1.y}{\partial y} & \frac{\partial P_1.y}{\partial z} & \frac{\partial P_1.y}{\partial r_x} & \frac{\partial P_1.y}{\partial r_y} & \frac{\partial P_1.y}{\partial r_z} & I_3 & O_3 \\ \frac{\partial P_1.z}{\partial x} & \frac{\partial P_1.z}{\partial y} & \frac{\partial P_1.z}{\partial z} & \frac{\partial P_1.z}{\partial r_x} & \frac{\partial P_1.z}{\partial r_y} & \frac{\partial P_1.z}{\partial r_z} & & \\ 0 & 0 & 0 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & O_3 & I_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & & \end{bmatrix},$
- $\tilde{J}_{P_2} = \begin{bmatrix} \frac{\partial P_2.x}{\partial x} & \frac{\partial P_2.x}{\partial y} & \frac{\partial P_2.x}{\partial z} & 0 & 0 & 0 \\ \frac{\partial P_2.y}{\partial x} & \frac{\partial P_2.y}{\partial y} & \frac{\partial P_2.y}{\partial z} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$
- $\tilde{J}_{P_3} = \begin{bmatrix} \frac{\partial P_3.x}{\partial x} & \frac{\partial P_3.x}{\partial y} & \frac{\partial P_3.x}{\partial k_1} & \frac{\partial P_3.x}{\partial k_2} & \frac{\partial P_3.x}{\partial f} \\ \frac{\partial P_3.y}{\partial x} & \frac{\partial P_3.y}{\partial y} & \frac{\partial P_3.y}{\partial k_1} & \frac{\partial P_3.y}{\partial k_2} & \frac{\partial P_3.y}{\partial f} \end{bmatrix}.$

The detailed computations of the elements of these matrices are in Appendix B.

So to compute the sparse Jacobian of the residuals J_r, one simply needs to compute, for each observation k, the small dense Jacobian $\tilde{J}_P[k]$ and to store the JX and JC blocks at the right place.

To check my computations for the Jacobian, I used the function "Jacobian_check" from the NLP-Models package. This function computes the Jacobian of the constraints of a NLPModel using finite differences, and returns a dictionary containing the errors of the Jacobian that are greater than a chosen threshold. I also compared my Jacobian with the one computed with the CUTEst library [15].

2.4.3 Use of multiple threads

I used multi-threading to improve the performance of my model, and thus of the optimization algorithms I use or write for them. Indeed, most of optimization algorithms will need to compute the

Jacobian (only the values, as the structure does not change) and the residuals of my model at each iteration. Thus, the faster these functions are, the faster the optimization will get.

The functions cons! (that computes the residuals), jac_structure! and jac_coord! all iterate on the observations using a for loop. The content of this for loop is independent (as the observations are independent from one another), so it is quite straightforward to split the loop between the threads.

3 Solving bundle adjustment problems

I created a solver based on Levenberg-Marquardt algorithm that takes as input an AbstractNLSModels (this a type included in NLPModels that models a non-linear least square problem). Thus, my solver can solve bundle adjustment problems, by creating a BALNLPModel (the type I have created for bundle adjustment problems) and using the function FeasibilityResidual which wrapsd it into an NLSModel.

The first subsection explains the standard Levenberg-Marquardt algorithm. My implementation is described in the next subsections.

3.1 The Levenberg-Marquardt algorithm

Let us consider a least square optimization problem:

$$\min_{x\in\mathbf{R}^n} \sum_{i=1}^{m} r_i(x)^2 = \min_{x\in\mathbf{R}^n} ||r(x)||^2,$$

where $r : \mathbf{R}^n \to \mathbf{R}^m$ is the residual function.

The Jacobian of r is $J = (\frac{\partial r_i}{\partial x_j}) \in \mathbf{R}^{m\times n}$. Let $\delta \in \mathbf{R}^n$. We have $r(x+\delta) \simeq r(x) + J(x)\delta$, so:

$$\begin{aligned} ||r(x+\delta)||^2 &\simeq ||r(x) + J(x)\delta||^2 \\ &\simeq (r(x) + J(x)\delta)^T (r(x) + J(x)\delta) \\ &\simeq r(x)^T r(x) + 2r(x)^T J(x)\delta + \delta^T J(x)^T J\delta. \end{aligned}$$

To find the search direction δ, let us take the derivative of $||r(x+\delta)||^2$ with respect to δ:

$$\frac{\partial ||r(x+\delta)||^2}{\partial \delta} = 2J^T r + 2J^T J\delta.$$

Thus, to first order, the minimum of $||r(x+\delta)||^2$ is reached for δ verifying:

$$J^T J\delta = -J^T r.$$

This is the Gauss-Newton method. When J is rank deficient (not of full rank), this algorithm can diverge. The idea of Levenberg was to add a damping parameter to avoid this. So in the Levenberg-Marquardt algorithm, the previous equation is replaced by a "damped" version:

$$(J^T J + \lambda D^2)\delta = -J^T r,$$

where $\lambda > 0$ and D^2 is a diagonal matrix such that $J^T J + \lambda D^2$ is non singular. Often $D^2 = I$ or $D^2 = Diag(J^T J)$.

When the damping parameter λ is close to 0, the algorithm is close to the Gauss-Newton method, whereas when $\lambda \to \infty$, $\frac{\delta}{\lambda} \to -J^T r$, so the algorithm is close to a gradient descent.

3.2 Update of the damping parameter λ

The idea of Marquardt was to refine the damping parameter λ at each iteration. Indeed, the step δ found at each iteration may not always result in a decrease of the cost function $\frac{1}{2}||r||^2$. So the step δ is accepted if and only if the actual reduction is larger than ϵ times the predicted reduction:

$$||r_k||^2 - ||r_{k+1}||^2 \geq \epsilon \, (||r_k||^2 - ||J_k\delta + r_k||^2), \, \epsilon > 0.$$

If δ is not accepted, we update it as: $\lambda_{k+1} = max(\lambda_k, \frac{1}{||\delta||}) \times \nu_m$, where $\nu_m > 1$. This way to update λ is a bit more sophisticated than the classic way ($\lambda_{k+1} = \lambda_k \times \nu_m$). It is inspired from Moré's formulation of Levenberg-Marquardt algorithm, which is based on trust-region methods [22]:

$$\min \frac{1}{2}||J\delta + r||^2, \text{ under } ||\delta|| \leq \Delta.$$

In our formulation, λ is similar to $\frac{1}{\Delta}$. So the larger λ is, the more $||\delta||$ is constrained, and the lower λ is, the more $||\delta||$ is free. So if a step δ is rejected, the radius Δ is decreased. But if the new Δ is still larger than $||\delta||$, it is a waste of time, since the same δ will be computed. To avoid that, we need to make sure that the new Δ is smaller than $||\delta||$. In our formulation, it means that we have to increase λ at least by a factor $\frac{1}{||\delta||}$. That is why we update λ this way: $\lambda_{k+1} = max(\lambda_k, \frac{1}{||\delta||}) \times \nu_m$.

If δ is accepted, the classic way to update λ is $\lambda_{k+1} = \frac{\lambda_k}{\nu_d}$, where $\nu_d > 1$. That is what I do. But, in addition, if the actual reduction is bigger than 0.9 time the predicted reduction (which means that δ is a very successful step), I update λ this way: $\lambda_{k+1} = \frac{\lambda_k}{\nu_d^2}$. And to avoid very low values of λ, I make sure that λ is higher than a lower bound ($1.0e^{-8}$).

It is also important to find a good initial value for λ. One can choose an initial value that works well for several problems, or one can choose a formula or a heuristic that adapts the initial value of λ to the problem. Once again, I got inspiration from trust region methods, where they initialize Δ like this $\Delta_0 = min(10, \frac{||\nabla f(x_0)||}{10})$. So I decided to initialize λ like this:

$$\lambda_0 = max(\lambda'_0, \frac{l}{||J^T r(x_0)||}),$$

where $\lambda'_0 > 0$ and $l > 0$ are constants left to the choice of the user, and for which I found good default values.

Another way to initialize λ is presented in [23]. The authors claim it is reasonable to relate the initial value of λ to the size of the eigenvalues of the symmetric positive definite matrix $J^T J$. The maximum of the diagonal elements of this matrix has the same order of magnitude than the biggest eigenvalue, so one can initialize λ this way:

$$\lambda_0 = \tau \times \max J^T J(x_0)_{i,i},$$

where $\tau > 0$ is chosen small if x_0 is close to x^*.

But computing $J^T J$ for large problems such as bundle adjustment problems is quite expensive, so we did not use this method.

3.3 Stopping criteria

The algorithm has several stopping criteria:

- First order criterion (that is to say that the norm of the gradient of the cost function is smaller than a given bound): $||J^T r|| < atol + rtol||J^T r||_0$ ("first order").
- Small objective change criterion: $\frac{1}{2}||r_{k-1}||^2 - \frac{1}{2}||r_k||^2 < oatol + ortol \times \frac{1}{2}||r_{k-1}||^2$ ("acceptable").
- Small step criterion: $||\delta|| < satol + srtol||x||$ ("small step").
- Small residuals criterion: $||r|| < restol$ ("small residuals").
- Tired criterion: $k > ite\_max$ ("max iter").

3.4 Factorization and permutation

In my solver I use two kinds of factorization (QR and LDL) of matrices that are computed from the Jacobian (they contain blocks that are either multiple of I, J or J^T). These matrices are sparse, thus one needs to be careful as the factorization of a sparse matrix can be dense. That is why, the standard factorization algorithms use permutations to transform the original matrix in such a way that its factorization will be as sparse as possible.

As the structure of the Jacobian of a bundle adjustment problem does not change between iterations, one does not need to compute the permutation each time the factorization is computed. That is why, when the algorithm that computes the factorization allows it, I compute the permutation at the beginning and pass it as an argument to the factorization algorithm. Even if the factorization algorithm does not allow to pass the permutation vector as argument, it is possible to choose the permutation method the algorithm uses. Thus I added an option to my solver for the user to choose the permutation he would like to use.

To compute the permutations I used two types of algorithms: AMD and Metis. The Approximate Minimum Degree ordering algorithm (AMD) [25] pre-orders a symmetric sparse matrix prior to numerical factorization. It uses techniques based on the quotient graph for matrix factorization that allows to obtain computationally cheap bounds for the minimum degree. These bounds are often equal to the actual degree. The Julia interface for AMD, takes as input a matrix A and computes a fill-reducing permutation based on the sparsity pattern of $A + A^T$, so the input matrix can be anything (even non-symmetric or rectangular matrices). The Metis algorithm [21] finds good partitioning of highly unstructured graphs. It has several applications, including computing fill-reducing permutations for sparse matrices. The main drawback of the Julia's Metis interface is that it only takes symmetric square matrices as input so one needs to compute $A + A^T$ if A is not symmetric and square.

3.5 The QR version

One way to implement the Levenberg-Marquardt algorithm is to notice that the equations $(J^TJ + \lambda I)\delta = -J^Tr$ are just the normal equations for the following linear least-squares problem [24]:

$$\min_{\delta} || \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} \delta + \begin{bmatrix} r \\ 0 \end{bmatrix} ||^2.$$

Thus, one only need to solve a linear least-square problem at each iteration. The QR version of my implementation of Levenberg-Marquardt is based on this.

In order to do that efficiently, one can use a QR factorization of the matrix $A = \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$. That is to say find Q orthogonal and R upper triangular such that $A = QR$.

Let us rewrite the QR factorization of A: $A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix}$. Thus:

$$\begin{aligned} ||A\delta + \begin{bmatrix} r \\ 0 \end{bmatrix} || &= || \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} (\begin{bmatrix} R \\ 0 \end{bmatrix} \delta + \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}^T \begin{bmatrix} r \\ 0 \end{bmatrix})|| \\ &= || \begin{bmatrix} R \\ 0 \end{bmatrix} \delta + \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}^T \begin{bmatrix} r \\ 0 \end{bmatrix} || \\ &= || \begin{bmatrix} R\delta + Q_1^T \begin{bmatrix} r \\ 0 \end{bmatrix} \\ Q_2^T \begin{bmatrix} r \\ 0 \end{bmatrix} \end{bmatrix} ||. \end{aligned}$$

As the $Q_2^T \begin{bmatrix} r \\ 0 \end{bmatrix}$ part does not depend on δ, the best we can do, if R is invertible, is to take $\delta^* = -R^{-1}Q_1^T \begin{bmatrix} r \\ 0 \end{bmatrix}$. So finally, $||A\delta^* + \begin{bmatrix} r \\ 0 \end{bmatrix}|| = ||Q_2^T \begin{bmatrix} r \\ 0 \end{bmatrix}||$.

In addition, as A is a sparse matrix (since J is sparse), one can use a sparse QR factorization which is even faster. In my implementation, I used the "SuiteSparseQR" library [5] (with a Julia wrapper), which is a C library that computes the QR factorization of sparse matrices.

The QR version of my solver looks like this:

Algorithm 1 QR version

Compute residuals r, $obj = \frac{1}{2}||r||^2$ and create $b = \begin{bmatrix} -r \\ 0 \end{bmatrix}$
Compute Jacobian J
Compute $||J^T r||$
$\lambda = max(\lambda, \frac{1e10}{||J^T r||})$
Create $A = \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$
while none of the stopping criteria is verified **do**
  Compute QR factorization of A
  Find $\delta = argmin(||A\delta + b||^2)$
  $\delta r2 = \frac{1}{2}||J\delta + r||^2$
  $\rho = \frac{obj - obj\_suiv}{obj - \delta r2}$
  if $\rho \geq 1e^{-4}$ **then**
    $\lambda = max(\lambda, \frac{1}{||\delta||}) \times \nu_m$
    Update the $\sqrt{\lambda} I$ part of A
  else
    $\lambda = \frac{\lambda}{\nu_d}$
    if $\rho \geq 0.9$ **then**
      $\lambda = \frac{\lambda}{\nu_d}$
    end if
    $\lambda = max(1e^{-8}, \lambda)$
    Update r, J, A, b
  end if
end while

3.6 Givens rotations

This section briefly explains what Givens rotations are, in order to help the reader understand the next section.

Givens rotations are used to create zeros in matrices. They are stored in matrix $G(i, j, c, s)$, similar to the identity matrix, except that two rows and two columns are changed:

- there is a c in positions (i, i) and (j, j);
- there is a s in position (i, j) $(i < j)$,
- there is a $-s$ in position (j, i).

Thus, if we have a matrix A in which we want to remove the element $A[j, i] = x$, by taking $r = \sqrt{x^2 + y^2}$ (where $x = A[i, i]$), $c = \frac{x}{r}$ and $s = \frac{y}{r}$, we can apply the Givens rotation $G(i, j, c, s)$ on A.

For instance:

$$G(i = 1, j = 2, c, s)A = \begin{bmatrix} 0.7682 & 0.6492 & 0 \\ -0.6492 & 0.7682 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix} = \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & -2.4327 & 3.0729 \\ 0 & 4 & 3 \end{bmatrix},$$

as $r = \sqrt{6^2 + 5^2} = 7.8102$, $c = \frac{6}{7.8102}$ and $s = \frac{5}{7.8102}$.

A zero has been created at position $(2, 1)$. A Givens rotation modifies the whole rows i and j. We notice that a non-zero element has been created at position $(1, 3)$.

3.7 Improvements to the QR version

One can avoid to compute the full QR factorization of A at each iteration. Indeed one may compute the QR factorization of J only when the Jacobian changes (that is to say when the iteration is accepted) and compute at each iteration the QR factorization of A from the one of J using Givens rotations [22, 24].

Indeed, if $J = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$, then $\begin{bmatrix} R \\ 0 \\ \sqrt{\lambda} I \end{bmatrix} = \begin{bmatrix} Q^T & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$.

The matrix $\begin{bmatrix} R \\ 0 \\ \sqrt{\lambda} I \end{bmatrix}$ is almost triangular. One can eliminate the elements of $\sqrt{\lambda} I$ by performing at most $\frac{n(n+1)}{2}$ Givens rotations. Indeed, one can eliminate the element in position (n, n) in $\sqrt{\lambda} I$ by rotating row n of $\sqrt{\lambda} I$ with row n of R. Then, one can eliminate the element in position $(n-1, n-1)$ in $\sqrt{\lambda} I$ by rotating row $n-1$ of $\sqrt{\lambda} I$ with row $n-1$ of R. If the element in position $(n-1, n)$ of R is a non-zero it will create a non-zero element at position $(n-1, n)$ in $\sqrt{\lambda} I$, one can eliminate it by rotating row $n-1$ of $\sqrt{\lambda} I$ with row n of R, and so on.

If we store the Givens rotations in a matrix $\tilde{Q}_\lambda^T$,then $\tilde{Q}_\lambda^T \begin{bmatrix} R \\ 0 \\ \sqrt{\lambda} I \end{bmatrix} = \begin{bmatrix} R_\lambda \\ 0 \\ 0 \end{bmatrix}$.

Let $Q_\lambda = \begin{bmatrix} Q & 0 \\ 0 & I \end{bmatrix} \tilde{Q}_\lambda$, then $\begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} = Q_\lambda \begin{bmatrix} R_\lambda \\ 0 \\ 0 \end{bmatrix}$.

Actually, as explained in Section 3.5, permutations of rows and columns are performed during the factorization process in order to get sparse matrices. Thus, J is not equal to $Q \begin{bmatrix} R \\ 0 \end{bmatrix}$. Most of the time, one actually has:

$$P_1 J P_2 = Q \begin{bmatrix} R \\ 0 \end{bmatrix},$$

where P_1 and P_2 are permutation matrices.

It follows that:

$$\begin{aligned}
\| \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} \delta + \begin{bmatrix} r \\ 0 \end{bmatrix} \| &= \| \begin{bmatrix} J P_2 \\ \sqrt{\lambda} P_2 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} r \\ 0 \end{bmatrix} \| \\
&= \| \begin{bmatrix} P_1^T & \\ & I \end{bmatrix} (\begin{bmatrix} P_1 J P_2 \\ \sqrt{\lambda} P_2 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} P_1 r \\ 0 \end{bmatrix}) \| \\
&= \| \begin{bmatrix} P_1 J P_2 \\ \sqrt{\lambda} P_2 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} P_1 r \\ 0 \end{bmatrix} \| \qquad \text{as } \begin{bmatrix} P_1^T & \\ & I \end{bmatrix} \text{ is orthogonal} \\
&= \| \begin{bmatrix} Q^T P_1 J P_2 \\ \sqrt{\lambda} P_2 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix} \| \quad \text{as } \begin{bmatrix} Q & \\ & I \end{bmatrix} \text{ is orthogonal} \\
&= \| \begin{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} \\ \sqrt{\lambda} P_2 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix} \| \qquad \text{as } Q^T P_1 J P_2 = \begin{bmatrix} R \\ 0 \end{bmatrix} \\
&= \| \begin{bmatrix} I & \\ & P_2 \end{bmatrix} \tilde{Q}_\lambda \begin{bmatrix} \begin{bmatrix} R_\lambda \\ 0 \end{bmatrix} \\ 0 \end{bmatrix} (P_2^T \delta) + \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix} \| \text{ as } \tilde{Q}_\lambda^T \begin{bmatrix} I & \\ & P_2^T \end{bmatrix} \begin{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} \\ \sqrt{\lambda} P_2 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} R_\lambda \\ 0 \end{bmatrix} \\ 0 \end{bmatrix} \\
&= \| \begin{bmatrix} R_\lambda \\ 0 \\ 0 \end{bmatrix} (P_2^T \delta) + \tilde{Q}_\lambda^T \begin{bmatrix} I & \\ & P_2^T \end{bmatrix} \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix} \| \quad \text{as } \begin{bmatrix} I & \\ & P_2 \end{bmatrix} \tilde{Q}_\lambda \text{ is orthogonal} \\
&= \| \begin{bmatrix} R_\lambda \\ 0 \\ 0 \end{bmatrix} (P_2^T \delta) + \tilde{Q}_\lambda^T \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix} \|.
\end{aligned}$$

Thus, if R_λ is invertible, the solution of the linear problem is $\delta^* = -P_2 R_\lambda^{-1} \tilde{Q}_\lambda^T \begin{bmatrix} Q^T P_1 r \\ 0 \end{bmatrix}$.

My algorithm takes as input a copy of R and performs the Givens rotations in place, without forming the big matrix $\begin{bmatrix} R \\ 0 \\ \sqrt{\lambda} I \end{bmatrix}$. And to avoid memory allocations, it does not compute explicitly the matrix Q_λ. It keeps in memory a list of the Givens rotations (for one rotation, it only stores four elements: the indices of the two rows to rotate, the cosinus and sinus of the rotation) that were performed on R. I created a function that computes $\tilde{Q}_\lambda^T \begin{bmatrix} Q^T & 0 \\ 0 & I \end{bmatrix} x$ given the list of the of the Givens rotations, the matrix Q (from the QR factorization of J) and a vector x. The algorithms that computed the Givens rotations, together with some explanations about them, can be found in Annexe C.

3.8 The LDL version

One can notice that the equations $(J^T J + \lambda D^2)\delta = -J^T r$ are equivalent to:

$$\begin{bmatrix} I & J \\ J^t & -\lambda I \end{bmatrix} \begin{bmatrix} \delta r \\ \delta \end{bmatrix} = \begin{bmatrix} -r \\ 0 \end{bmatrix}.$$

So in this version of the algorithm, one considers the sparse symmetric indefinite matrix $A = \begin{bmatrix} I & J \\ J^t & -\lambda I \end{bmatrix}$ (as it is symmetric one only needs to store the upper triangular part).

As A is symmetric, one can perform a LDL factorization (which is a variant of the Cholesky decomposition for matrices that are not necessarily positive definite) of it and then solve $Ax = b$ with $x = \begin{bmatrix} \delta r \\ \delta \end{bmatrix}$ and $b = \begin{bmatrix} -r \\ 0 \end{bmatrix}$. If $A = LDL^T$, with L upper triangular and D diagonal, solving $Ax = b$ amounts to solving $Ly = b$ (triangular problem), then $Dz = y$ (diagonal problem), then $L^T x = z$ (triangular problem).

I used the Julia package "LDLFactorizations" [8] that performs LDL factorization of sparse matrices and solves linear systems $Ax = b$ given the LDL factorization of A.

3.9 Improvements to the LDL version

To improve the LDL version, I split the LDL Factorisation into two parts: the symbolic analysis and the numeric factorisation. The symbolic analysis only depends on the structure of the matrix, so I only need to perform it once in the whole algorithm, while the numerical factorization is performed at each iteration.

The LDL version of my solver looks like this:

Algorithm 2 LDL version

Compute residuals r, $obj = \frac{1}{2}||r||^2$ and create $b = \begin{bmatrix} -r \\ 0 \end{bmatrix}$
Compute Jacobian J
Compute $||J^T r||$
$\lambda = max(\lambda, \frac{1e10}{||J^T r||})$
Create $A = \begin{bmatrix} I & J \\ J^T & -\lambda I \end{bmatrix}$
Compute permutation and perfom LDL analysis
while none of the stopping criteria is verified **do**
  Perform LDL factorization of A
  Find δ such that $Ax = b$
  $\delta r2 = \frac{1}{2}||J\delta + r||^2$
  $\rho = \frac{obj - obj\_suiv}{obj - \delta r2}$
  if $\rho \geq 1e^{-4}$ **then**
    $\lambda = max(\lambda, \frac{1}{||\delta||}) \times \nu_m$
    Update the $\sqrt{\lambda} I$ part of A
  else
    $\lambda = \frac{\lambda}{\nu_d}$
    if $\rho \geq 0.9$ **then**
      $\lambda = \frac{\lambda}{\nu_d}$
    end if
    $\lambda = max(1e^{-8}, \lambda)$
    Update r, J, A, b
  end if
end while

3.10 Normalization

Sometimes, in the QR version, due to computation errors, the decomposition of the matrix $A = \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$ (which is normally of full rank because of the $\sqrt{\lambda} I$ part), gives zeros on the diagonal of R and thus the resolution of the linear problem fails. To avoid that, the whole matrix A or only J can be normalized before the factorization. I added a parameter for the user to choose whether he wants to normalize A, J or if he does not want to use normalization.

In the QR version we have:

$$|| \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} \delta + \begin{bmatrix} r \\ 0 \end{bmatrix} || = || \begin{bmatrix} JD \\ \sqrt{\lambda} D \end{bmatrix} (D^{-1}\delta) + \begin{bmatrix} r \\ 0 \end{bmatrix} ||$$

Where D is the diagonal matrix such that $D_{i,i} = \frac{1}{||J[:,i]||}$ if we decide to normalize only J. Or, if we decide to factorize the whole matrix $A = \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$, $D_{i,i} = \frac{1}{||A[:,i]||}$.

In the LDL version I did not obtain errors without normalization, but it is a good thing to normalize J to have a better conditioned problems. The LDL version with normalization of J is given by the following equations:

$$\begin{aligned}
\begin{bmatrix} I & J \\ J^T & -\lambda I \end{bmatrix} \begin{bmatrix} \delta r \\ \delta \end{bmatrix} = \begin{bmatrix} -r \\ 0 \end{bmatrix} &\iff \begin{bmatrix} I & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & JD \\ (JD)^T & -\lambda D^2 \end{bmatrix} \begin{bmatrix} \delta r \\ D^{-1}\delta \end{bmatrix} = \begin{bmatrix} -r \\ 0 \end{bmatrix} \\
&\iff \begin{bmatrix} I & JD \\ (JD)^T & -\lambda D^2 \end{bmatrix} \begin{bmatrix} \delta r \\ D^{-1}\delta \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} -r \\ 0 \end{bmatrix} \\
&\iff \begin{bmatrix} I & JD \\ (JD)^T & -\lambda D^2 \end{bmatrix} \begin{bmatrix} \delta r \\ D^{-1}\delta \end{bmatrix} = \begin{bmatrix} -r \\ 0 \end{bmatrix}
\end{aligned}$$

Where D is the diagonal matrix such that $D_{i,i} = \frac{1}{||J[:,i]||}$.

To have an even better conditioning of the matrix A in the LDL version, we can write the following equivalent system [28]:

$$\begin{bmatrix} \sqrt{\lambda}I & J \\ J^T & -\sqrt{\lambda}I \end{bmatrix} \begin{bmatrix} \delta r \\ \sqrt{\lambda}\delta \end{bmatrix} = \begin{bmatrix} -\sqrt{\lambda}r \\ 0 \end{bmatrix}$$

The LDL version with a normalization of A consists actually of normalizing J and then use the preceding equations to get the same coefficient $\sqrt{\lambda}$ in the diagonal blocks .

3.11 The line search strategy

In order to avoid computing too many factorizations, one can use the fact that the step δ computed at a given iteration is a descent direction. Thus, even if δ is rejected, one can try and divide it by a constant $\delta_d > 1$ until the step is accepted. This strategy is an Armijo line search.

In my implementation, δ_d can be chosen by the user and its default value is 2. If after dividing δ by δ_d four times, the step is still not accepted, the step is definitely rejected. The way we update λ changes a bit with the line search strategy. Let $ntimes$ be the number of times the step δ has been divided by δ_d, we have $0 \leq ntimes \leq 4$. If the step is rejected, we update λ like this: $\lambda = max(\lambda, \frac{1}{||\delta||}) \times \nu_m^{ntimes+1}$. Thereby, if no line search is performed, we have $ntimes = 0$ and $\lambda = max(\lambda, \frac{1}{||\delta||}) \times \nu_m$, just like before. Moreover, it allows to increase λ proportionally to the number of times the step has been rejected. Now, if the step is accepted and $ntimes > 0$ (ie: line search has been performed), we update λ this way: $\lambda = \frac{\lambda}{\nu_d^(ntimes-1)}$ (the power $ntimes-1$ has been chosen empirically). If $ntimes = 0$, λ is updated like before: $\lambda = \frac{\lambda}{\nu_d}$.

3.12 Global convergence of the Levenberg-Marquardt algorithm

There are many ways to implement the Levenberg-Marquardt algorithm, and many ways to prove the global convergence of this method. The proof presented in this section is inspired from [29] and is based on the line search version of the algorithm. Let us suppose that $r \in \mathbf{C}^1$.

From the algorithm, one easily deduces that $||r(x_k)||$ is monotonically decreasing and bounded below by 0. Thus, $||r(x_k)|| \to \gamma \geq 0$, when k tends towards infinity.

From the Armijo line search, one has [29]: $||r(x_{k+1})||^2 \leq ||r(x_k)||^2 - \beta \frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2}$, with $\beta > 0$.

Thus, $\frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2} \leq \frac{1}{\beta}(||r(x_k)||^2 - ||r(x_{k+1})||^2)$, and the sum:

$$\sum_{k=1}^{\infty} \frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2} \leq \frac{1}{\beta} \sum_{k=1}^{\infty} (||r(x_k)||^2 - ||r(x_{k+1})||^2) = \frac{1}{\beta}(||r_1||^2 - \gamma^2) < \infty.$$

By definition, $(J_k^T J_k + \lambda_k I)\delta_k = -J_k^T r_k$. Thus, $r_k^T J_k = (J_k^T r_k)^T = \delta_k^T (J_k^T J_k + \lambda_k I)$. So:

$$(r_k^T J_k \delta_k)^2 = (\delta_k^T (J_k^T J_k + \lambda_k I)\delta_k)^2 \geq (\delta_k^T \lambda_k I \delta_k)^2 \geq \alpha^2 ||\delta_k||^4,$$

as $\lambda_k \geq \alpha = 1e^{-8}$. Thus $\frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2} \geq \alpha^2 ||\delta_k||^2$.

As $\frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2} \to 0$ when k tends towards infinity (as the term of a convergent serie) and $\frac{(r_k^T J_k \delta_k)^2}{||\delta_k||^2} \geq \alpha^2 ||\delta_k||^2$, one has:

$$\lim_{k \to \infty} ||\delta_k|| = 0$$

This limit, $(J_k^T J_k + \lambda_k I)\delta_k = -J_k^T r_k$ and the continuity of $J(x)$ imply that at any accumulation point x^* of $\{x_k\}$, we have that $J(x^*)^T r(x^*) = 0$ which says that x^* is a stationary point of $\frac{1}{2}||r(x)||^2$. This proves the global convergence of the Levenberg-Marquardt algorithm.

3.13 Use of several precisions

I modified my solver so that it preserves the type of the variables. It is only possible for the LDL version, as the LDL decomposition is coded in Julia and preserves the types, but not the QR version as SuiteSparseQR is coded in C. This preservation of types allows the user to choose the precision he wants to use by choosing the precision of x_0. He can also choose to read the datasets in a lower precision than Float64 if we wants to. Using simple (Float32) or half (Float16) precision will allow to have faster and less precise optimization, whereas using quadratic precision (Float128) will produce slower and more precise optimization.

In addition to the gain of time, doing computations in lower precisions can save energy. Indeed, the heat emitted by a processor is more or less proportional the processor surface, and the length of the processor is proportional to the number of significant digits, thus to the precision. As a result, dividing the precision by a factor k will decrease the heat emitted by the processor by a factor k^2.

Most algorithms for non-linear least square problems only compute the first order derivatives of the residuals, based on the hypothesis that the residuals are almost null or linear at a solution. But this hypothesis is not always verified. In order to improve the convergence of the solver, an improvement would be to test this hypothesis at each iteration, and to compute the second order derivatives of the residuals, when the hypothesis is not verified and the number of iteration is big enough (so that we are close to convergence).In addition, in order to improve the convergence without a big waste of time, the second derivatives could be computed in simple precision. This improvement has not been coded in this internship, as computing the second order derivatives analytically, as I did for the first order derivatives, would be very long and tedious, and the module for automatic differentiation in Julia [14] is still buggy.

4 Experimental results

4.1 Improving the execution time with multi-threading

As explained in Section 2.4.3, I used multiple threads to run the residuals and Jacobian (both jac_structure and jac_coord) methods. I tested my methods on the computer I was given access to at Polytechnique Montréal, which possesses 48 cores: 8 sockets with 6 cores on each. I tried with various number of threads. The following table presents the execution time (in ms) of the three methods (residuals!, jac_structure! and jac_coord!) on problem LadyBug-49 on the left hand side and on problem LadyBug-138 on the right hand side:

nthreads	res!	jac_struct!	jac_coord!	res!	jac_struct!	jac_coord!
1	100	91	312	291	265	860
2	45	43	196	159	132	541
3	33	31	188	147	93	521
4	26	24	206	92	72	598
5	23	18	230	75	58	706
12	12	9	303	41	29	828
16	10	7	327	40	22	872
24	11	6	345	33	18	967
36	11	6	409	51	19	1116
48	16	10	423	39	25	1190

As expected, the execution time of residuals! and jac_structure! decreases with the number of threads for both problems. It decreases a bit in the end, but one can assume that it is due to the fact that the communication time will at some point be higher than the time gained using multithreading. However, the execution time of jac_coord is optimal with only three threads for both problems. It can be explained by the fact that this method needs more memory access than the others so adding too many threads does not improve the execution time. Nonetheless, as it is not the main topic of this internship, I did not spend more time on this matter and I only limited the number of threads used in jac_coord to 3, so that the user can choose how many threads he wants to use to improve the execution time of the two other methods without altering the execution time of jac_coord.

4.2 Results of the Givens strategy

I first tested the results of the Givens strategy on small matrices. It was very vast and efficient. But when I tested it on a first bundle adjustment problem (LadyBug-49), it appeared to be very slow. Although it gave the correct results, it took much more time to perform the Givens strategy than to compute the QR factorization of the whole matrix $\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix}$. It can be explained by the fact that it is a very expensive algorithm as it has a complexity in $O(\frac{n^2(n+1)}{2})$ (at most $\frac{n(n+1)}{2}$ Givens rotations are performed and each rotations is between two rows so at most $2 \times n$ elements need to be computed). Although, I improved my first try by taking into account the sparsity pattern of the matrix R (I decreased the execution time by 10), it was coded in Julia, whereas the SuiteSparseQR library is coded in C, so it will still be faster. The execution time of my Givens strategy was 129.82s on the LadyBug-49 problem, while the execution time of the QR factorization of the whole matrix $\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix}$ was 2.96s.

However, coding a sparse QR factorization in Julia was not the purpose of this internship so we decided to abandon this strategy. Nonetheless, I learnt a lot trying to implement this Givens strategy, and it helped me better understand QR factorizations.

Other methods could have been explored, such as the one proposed in [20]. In this paper the authors claim that Givens rotations become slow as the number of columns increases (which is the case for bundle adjustment problems). Instead, they propose to permute the rows of the matrix $\begin{bmatrix} R \\ \sqrt{\lambda} \end{bmatrix}$ so as to insert the $\sqrt{\lambda}$ into the R part to obtained what they call a "block banded matrix", as shown on the left hand side of the figure below.

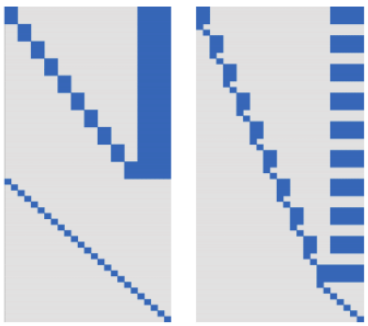

Figure 1: $\begin{bmatrix} R \\ \sqrt{\lambda} \end{bmatrix}$ on the left and $P\begin{bmatrix} R \\ \sqrt{\lambda} \end{bmatrix}$ on the right (Image taken from [20])

The "block banded" structure is a good one, as they present an efficient way to perform the QR factorization on those matrices. They compute operations on small dense blocks sequentially and use the compressed WY representation [19] of Householder QR.

4.3 Comparison of QR and LDL version with AMD and Metis permutations

4.3.1 Sparsity structure of the factors in QR and LDL factorizations

Using the command "spy" in Julia allows to display the sparsity pattern of a sparse matrix. I used it to check if the matrices R (from the QR factorization of $\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix}$) and L (from the LDL factorization of $\begin{bmatrix} I & J \\ J^T & -\lambda I \end{bmatrix}$) are sparse and to compare their sparsity structure with the two kinds of permutations I used: AMD and Metis.

In the following figure are shown the sparsity pattern of the Jacobian J and of the matrix R (computed for the QR factorization of $\begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$) with AMD and then Metis permutation, for the problem LadbyBug-49.

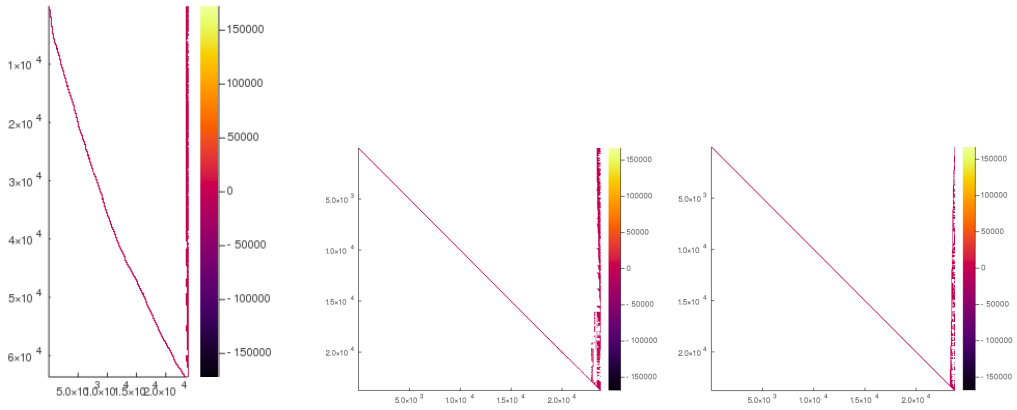

Figure 2: Sparsity structure of the Jacobian (left) and of the matrix R with AMD permutation (center) and with Metis permutation (right) for problem LadyBug-49

In the next figure are shown the sparsity pattern of the Jacobian J and of the matrix L (computed for the LDL factorization of $\begin{bmatrix} I & J \\ J^T & -\lambda I \end{bmatrix}$) with AMD and then Metis permutation, for the problem LadbyBug-49.

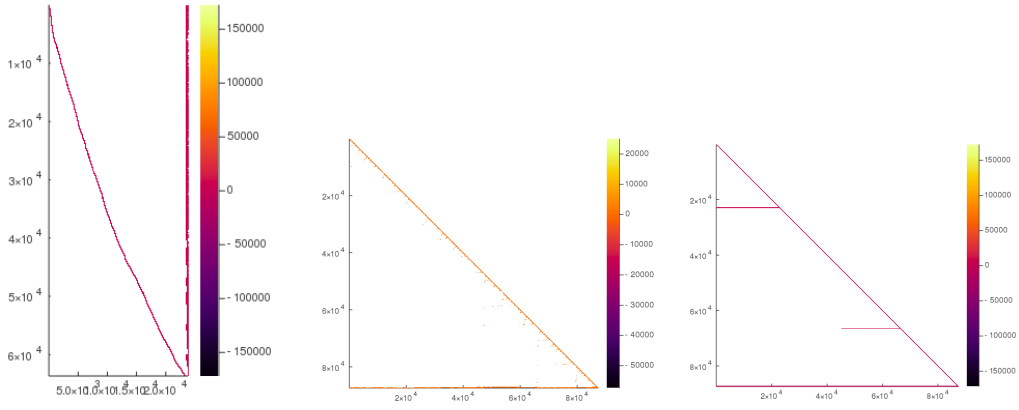

Figure 3: Sparsity structure of the Jacobian (left) and of the matrix L with AMD permutation (center) and with Metis permutation (right) for problem LadyBug-49

The next two figures present the same results but with the problem LadyBug-73.

Those figures confirm that the factorization methods (both QR and LDL) are sparse methods as the factors computed are indeed sparse. The sparse QR factorization produces similar sparsity pattern of R with AMD and Metis: the non-zero elements are gathered around the diagonal and in the last column. For the LDL factorization, the structure of the matrix L seem to differ between AMD and Metis. For both problems, while AMD gathers the non-zero elements of L around the diagonal and in the last rows, Metis tends to produce rows of non-zero elements randomly distributed in the matrix.

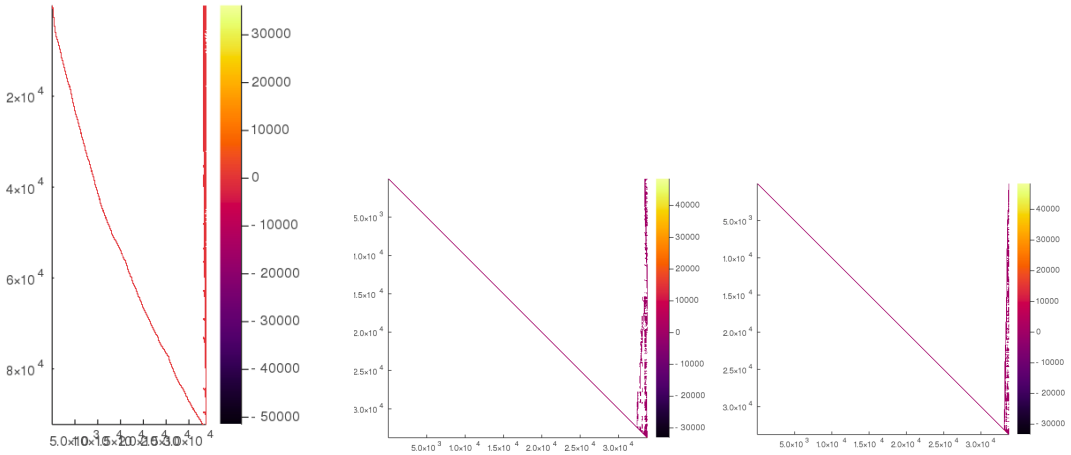

Figure 4: Sparsity structure of the Jacobian (left) and of the matrix R with AMD permutation (center) and with Metis permuatation (right) for problem LadyBug-73

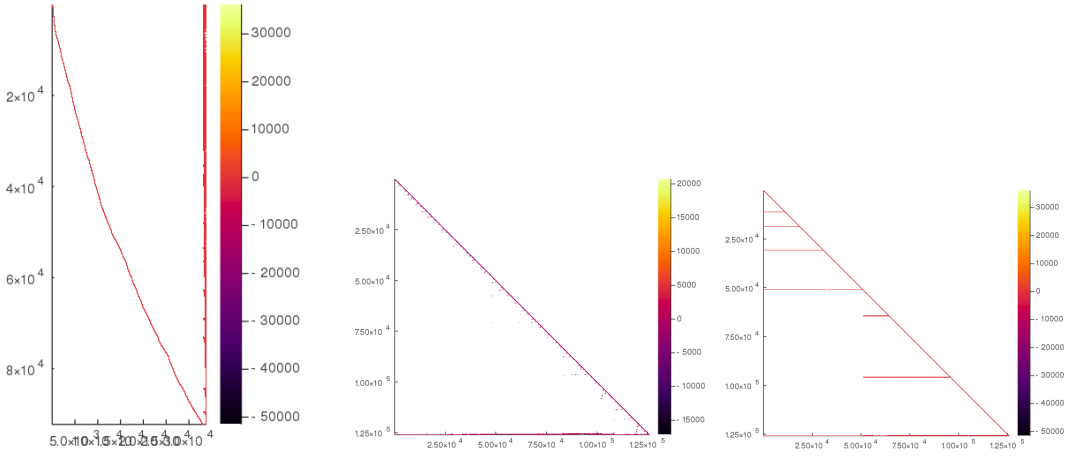

Figure 5: Sparsity structure of the Jacobian (left) and of the matrix L with AMD permutation (center) and with Metis permuatation (right) for problem LadyBug-73

The following table compares the number of non-zero elements in R (in the QR factorization) with AMD and Metis:

Problem	LB-49	LB-138	LB-318	LB-646
AMD	1000153	2958750	7716888	1637406
Metis	998640	3002580	7835553	1656254

For the QR version, the two permutations seem to be almost equivalent, although AMD seems to be a bit better.

The following table compares the number of non-zero elements in L (in the LDL factorization) with AMD and Metis:

Problem	LB-49	LB-138	LB-318	LB-646	LB-1031	LB-1723	D-202	D-356	V-427
AMD	1732781	4864731	11692048	24411696	37948213	69672478	41357606	69589108	93778918
Metis	1738679	5087030	13663002	29217289	45856499	71737270	40193169	69460530	94540821

As seen in the graphs of the sparsity structure above, AMD seems to be better than Metis for the LDL factorization. The difference between the two permutation algorithms gets smaller as the problems gets bigger.

4.3.2 Results of the solver with LDL and QR factorizations and with AMD and Metis permutations

In this section are presented the results of the simplest version of my solver, without normalization nor line search. This will allow us to compare the permutation algorithms AMD and Metis in the LDL and QR versions. The QR version does not work on very big problems, as the sparse QR factorization fails when $nvar \times ncon$ becomes higher than $6e^{11}$. That is why my results are split into two: results on relatively small problems (to compare QR and LDL factorizations), results on bigger problems (to compare LDL with AMD and Metis on bigger problems).

The following tables present the results of the simplest version of my solver, on "small problems". The first two tables contain the results of the QR version (with AMD and then Metis permutation algorithm). The columns contain: the name of the problem, the number of variables, the number of constraints (ie: the length of the residual vector), the final objective value, the elapsed time, the number of iterations, the status in which the solver finished, and the final dual feasibility value (ie: the final value of the gradient of the objective $||J^t r||$).

Table 1: Results of the solver QR-AMD

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	2.2e+02	57	acceptable	2.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.3e+02	17	first_order	7.1e+01
LadyBug-138-19878-feasres	60876	170434	6.0246e+04	2.5e+03	95	acceptable	3.5e+02
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	2.3e+03	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	7.3e+03	26	acceptable	2.0e+05
LadyBug-646-73584-feasres	226566	654594	∞	∞	0	exception	∞
LadyBug-810-88814-feasres	273732	787550	∞	∞	0	exception	∞
LadyBug-1031-110968-feasres	342183	1000530	∞	∞	0	exception	∞
LadyBug-1235-129634-feasres	400017	1152572	∞	∞	0	exception	∞
Dubrovnik-202-132796-feasres	400206	1503304	∞	∞	0	exception	∞

Table 2: Results of the solver QR-Metis

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	2.2e+02	57	acceptable	2.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.3e+02	17	first_order	7.1e+01
LadyBug-138-19878-feasres	60876	170434	6.0246e+04	2.6e+03	95	acceptable	3.5e+02
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	2.4e+03	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	7.1e+03	26	acceptable	2.0e+05
LadyBug-646-73584-feasres	226566	654594	∞	∞	0	exception	∞
LadyBug-810-88814-feasres	273732	787550	∞	∞	0	exception	∞
LadyBug-1031-110968-feasres	342183	1000530	∞	∞	0	exception	∞
LadyBug-1235-129634-feasres	400017	1152572	∞	∞	0	exception	∞
Dubrovnik-202-132796-feasres	400206	1503304	∞	∞	0	exception	∞

The next two tables contain the results of the LDL version (with AMD and then Metis permutation algorithm):

Table 3: Results of the solver LDL-AMD

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	4.3e+01	57	acceptable	1.6e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.8e+01	17	first_order	7.1e+01
LadyBug-138-19878-feasres	60876	170434	6.0246e+04	2.6e+02	95	acceptable	3.6e+02
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	1.5e+02	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	4.9e+02	26	acceptable	2.0e+05
LadyBug-646-73584-feasres	226566	654594	3.0126e+05	5.8e+01	1	first_order	6.8e+08
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	5.9e+02	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7524e+05	1.4e+03	26	first_order	1.1e+05
LadyBug-1235-129634-feasres	400017	1152572	3.2273e+05	1.3e+03	17	first_order	1.4e+07
Dubrovnik-202-132796-feasres	400206	1503304	3.3014e+05	4.0e+02	10	first_order	2.2e+03

Table 4: Results of the solver LDL-Metis

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	5.1e+01	57	acceptable	2.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	2.7e+01	17	first_order	7.1e+01
LadyBug-138-19878-feasres	60876	170434	6.0246e+04	3.3e+02	95	acceptable	3.5e+02
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	2.6e+02	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	9.1e+02	26	acceptable	2.0e+05
LadyBug-646-73584-feasres	226566	654594	1.8672e+05	6.4e+02	12	first_order	1.0e+09
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	1.4e+03	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7532e+05	3.1e+03	37	acceptable	1.6e+08
LadyBug-1235-129634-feasres	400017	1152572	3.2184e+05	2.1e+03	18	first_order	1.6e+07
Dubrovnik-202-132796-feasres	400206	1503304	3.3014e+05	3.4e+02	10	first_order	2.2e+03

From those tables of results we can draw several conclusions. First, the QR version (with AMD and Metis) does not terminate half of the time. As explained in Section 3.10, this is due to rank deficiency of the matrix $\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix}$ because of computations errors. That is why I added a normalization option to the solver. Second, the results of the various versions on the first five datasets are very alike. Indeed, apart from the elapsed time, all versions converge to the same objective in the same number of iterations, status, and nearly the same dual feasibility value. This is normal, as all versions solve the same linear problem at each iteration ($(J^TJ + \lambda I)\delta = -J^Tr$). However, the results of the two LDL versions differ on the last problems. This is also normal, because the various versions solve the same linear problem but in different ways, and the accumulation of numerical approximations (which bigger when the matrices and vectors are bigger, that is to say when the problem is bigger) can lead to such differences in the results.

The QR-AMD and QR-Metis versions give very similar results, there are a few differences on the elapsed time, but none on them seems to be really faster than the other. QR-AMD has a lower dual feasibility in the first problem, but it would be a bit arbitrary to choose based on only one problem. However, in Section 4.3.1, we have seen that the AMD algorithm uses less memory than Metis, as the matrix R computed by the QR factorization after AMD permutation is a bit sparser than with Metis. Thus, in the next versions we will keep QR-AMD. In terms of time performance, the LDL versions are much faster than the QR versions. On the first problem, the LDL versions are 5 times faster than the QR ones, and this factor increases and reaches 15 in the last problems. We will nonetheless have a look at the results of the QR normalized versions to see if it fixes the issue that causes the exception in the last five problems. The slowness of the QR methods can be explained by the fact that Julia's interface to SuiteSparseQR does not allow the user to give the permutation vector as parameter. Thus, in the QR versions, the permutation is computed at each iteration whereas in the LDL versions the permutation is only computed once, at the beginning of the solver.

If we compare the LDL-AMD and LDL-Metis versions, LDL-AMD seems to have better time performances (apart from the last problem), and better optimality performances. Indeed, for all problems it has a lower or equal value of dual feasibility than LDL-Metis. On problem LadyBug-646-73584, LDL-AMD reaches a lower objective value than LDL-Metis, but a higher dual feasibility. In that case, we cannot say that one of them is better than the other on that problem. Indeed, what is the most important proof of convergence: small residuals or a small gradient of the residuals (the dual feasibility is $||\nabla(\frac{1}{2}||r||^2)|| = ||J^Tr||$) ? So overall, LDL-AMD seems to be better than LDL-Metis in terms of execution time and also in terms of convergence.

The following graph is called a performance profile. Let us consider a set of algorithms $\{A_i\}$ and a set of problems $\{P_j\}$. Let $S_{P_j,A_i} \geq 0$ be a statistic (for instance the execution time of the algorithm) corresponding to the solution of P_j by A_i, and suppose that the smaller the statistic the better the algorithm. Furthermore, let $S_{P_j} = min\{S_{P_j,A_i}\}$. The performance profile of algorithm A_i is defined as:

$$\pi_i(\chi) = \frac{|\{P_j|\frac{S_{P_j,A_i}}{S_{P_j}} \leq \chi\}|}{|\{P_j\}|}, \chi \geq 1$$

where the ratio $\frac{S_{P_j,A_i}}{S_{P_j}}$ is set to infinity if A_i fails in solving P_j. So the performance profile π_i of algorithm A_i at χ gives us the proportion of problems for which A_i is better than χ times worse than the best algorithm. Thus $\pi_i(1)$ gives the percentage of problems for which A_i is the best, while the percentage of problems that are successfully solved by A_i is $\lim_{\chi\to\infty} \pi_i(\chi)$.

The following performance profile compares the four versions of the solver that we have seen in this section (LDL factorization with AMD permutation, LDL factorization with Metis permutation, QR factorization with AMD permutation, QR factorization with Metis permutation) in terms of: execution time (plot on the left hand side), number of evaluation of the residuals (plot in the center), number of evaluation of the Jacobian (plot on the right hand side).

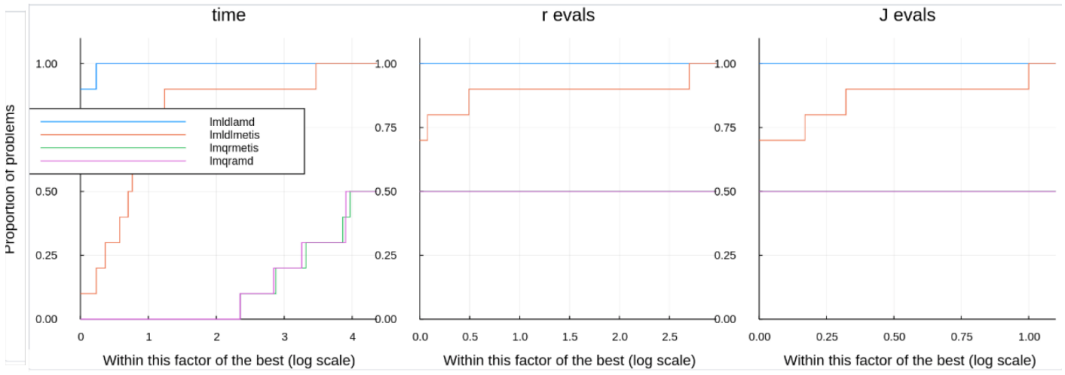

Figure 6: Profile of the simplest version on relatively small problems (the horizontal axis is a log scale so x means 2^x

As seen in the tables, the QR versions only solve half of the given problems. The LDL versions are, as expected, much faster than the QR ones, and LDL-AMD has better time performances than LDL-Metis, as it is faster on 90% of the problems. In terms of function evaluations LDL-AMD is better than LDL-Metis. This is due to the fact that LDL-Metis sometimes needs more iterations to reach optimality.

The following tables present the results of the LDL version (with AMD and then Metis permutation algorithm), on bigger problems:

Table 5: Results of the solver LDL-AMD on bigger problems

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-1723-156502-feasres	485013	1357436	6.3315e+05	1.1e+04	34	small_step	1.2e+11
Dubrovnik-273-176305-feasres	531372	1885940	3.4564e+05	1.2e+03	30	first_order	4.3e+03
Dubrovnik-356-226730-feasres	683394	2510536	4.9482e+05	1.2e+03	19	small_step	3.8e+03
Venice-427-310384-feasres	934995	3398290	1.0621e+06	1.7e+03	20	small_step	1.2e+05
Venice-1350-894716-feasres	2696298	9034252	2.6558e+06	1.9e+03	1	small_step	3.7e+15

Table 6: Results of the solver LDL-Metis on bigger problems

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-1723-156502-feasres	485013	1357436	3.7932e+05	5.1e+03	20	first_order	1.0e+08
Dubrovnik-273-176305-feasres	531372	1885940	3.4564e+05	1.2e+03	30	first_order	4.3e+03
Dubrovnik-356-226730-feasres	683394	2510536	4.9482e+05	1.2e+03	19	small_step	3.8e+03
Venice-427-310384-feasres	934995	3398290	1.0621e+06	1.8e+03	20	small_step	1.2e+05
Venice-1350-894716-feasres	2696298	9034252	2.6558e+06	1.9e+03	1	small_step	3.7e+15

These tables show that LDL-AMD and LDL-Metis give very similar results on big problems. Apart from the first problem, the results nearly exactly the same. In the first problem, LDL-Metis converges faster (in less iterations) to a lower objective value and a lower dual feasibility than LDL-AMD.

The profile below compares the LDL verion with AMD and Metis on the "bigger" problems:

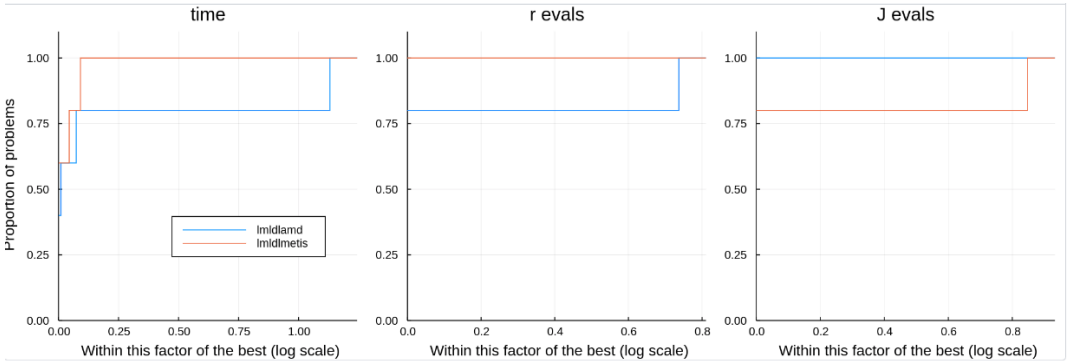

Figure 7: Profile of the LDL version on "big problems"

On the profile, LDL-Metis seems to be a bit better than LDL-AMD, but not that much as the horizontal axis is very spread out. From the tables we know that this difference is only due to one problem, so we will not take it into account.

In the following versions we will use the AMD permutation algorithm, as it has proven to be faster and more efficient on small problems than Metis in the LDL version, and a bit better (in terms of memory allocations) than Metis in the QR version.

4.4 Results of the normalized versions

4.4.1 Results of the QR normalized versions

In the QR version, the normalizations used are: None (no normalization is used), J (the matrix J is normalized) and A (the whole matrix $\begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$ is normalized). For each one of these versions, the AMD permutation algorithm is used, because it has proven better performance in the previous section.

The following two tables present the results of the QR-AMD version, with A normalization and then J normalization (see the Table 1 to see the results without normalization):

Table 7: Results of the solver QR-AMD with A normalization

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	2.0e+02	57	acceptable	2.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.1e+02	17	first_order	7.1e+01
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	2.0e+03	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	6.5e+03	26	acceptable	2.0e+05
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	8.1e+03	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7532e+05	2.7e+04	36	acceptable	1.6e+08
Dubrovnik-202-132796-feasres	400206	1503304	3.3014e+05	6.2e+03	10	first_order	2.2e+03

Table 8: Results of the solver QR-AMD with J normalization

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	2.1e+02	57	acceptable	2.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.2e+02	17	first_order	7.1e+01
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	2.0e+03	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	6.5e+03	26	acceptable	2.0e+05
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	8.4e+03	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7532e+05	2.7e+04	36	acceptable	1.6e+08
Dubrovnik-202-132796-feasres	400206	1503304	∞	∞	0	exception	∞

First of all, we are relieved to notice that the normalized versions are able to solve more problems than the non-normalized versions. The solver with the A normalization solves all the problems, so on this matter it is the best version of the three QR versions. In addition, it seems faster than the J-normalized and non-normalized version (and it gives the same results in terms of convergence). Thus, the A-normalized versions is the best of all three QR versions, as it does not suffer from bad contionning, and is hence able to solve all problems, and it is faster than the other versions.

This profile compares the QR-AMD version with the three types of normalization: A, J and None:

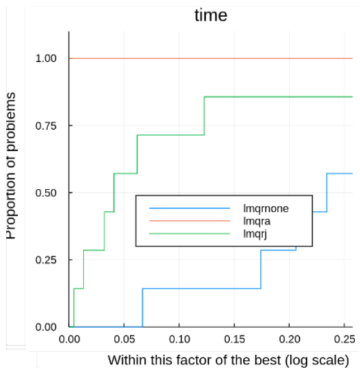

Figure 8: Profile of the QR normalized versions

The profile confirms what we concluded from the table. Indeed, the values at the abscissa 0 tell us that the A-normalized version is always faster than the others, and the values at infinity tell us that the A-normalized version solves all problems, unlike the other versions.

4.4.2 Results of the LDL normalized versions

In the LDL version, the normalizations used are: None (no normalization is used), J (the matrix J is normalized) and A (J is normalized and the matrix A is replaced by $\begin{bmatrix} \sqrt{\lambda}I & \hat{J} \\ \hat{J}^T & -\sqrt{\lambda} \end{bmatrix}$ as detailed in Section 3.10). For each one of these versions, the AMD permutation algorithm is used, because it has proven better performance in the previous section.

The following two tables present the results of the LDL-AMD version, with A normalization and then J normalization (see the Tables 3 and 5 to see the results without normalization):

Table 9: Results of the solver LDL-AMD with A normalization

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	4.4e+01	57	acceptable	4.1e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.9e+01	17	first_order	7.1e+01
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	1.5e+02	14	acceptable	4.5e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	4.6e+02	26	acceptable	2.0e+05
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	5.6e+02	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7532e+05	1.9e+03	36	acceptable	1.6e+08
Dubrovnik-202-132796-feasres	400206	1503304	3.3014e+05	4.2e+02	10	first_order	2.2e+03
Dubrovnik-273-176305-feasres	531372	1885940	3.4564e+05	1.3e+03	30	first_order	4.3e+03
Dubrovnik-356-226730-feasres	683394	2510536	4.9482e+05	1.3e+03	19	small_step	3.7e+03
Venice-427-310384-feasres	934995	3398290	1.0621e+06	1.8e+03	20	small_step	1.2e+05
Venice-1350-894716-feasres	2696298	9034252	2.6558e+06	1.7e+03	1	small_step	3.7e+15

The results of the three LDL-AMD versions are quite similar. However, one can notice some differences. To begin with, on the first problem, all three versions have a different value of the dual feasibility, the one of the non-normalized version being the lowest, and the one from the A-normalized version being the highest. Then, on problem LabyBug-1031-110868, the non-normalized version terminates with "first order", in less iterations and with a much lower dual feasibility value than the two other versions. In addition, the non-normalized version seems to be faster than the others overall.

Table 10: Results of the solver LDL-AMD with J normalization

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3364e+04	4.0e+01	57	acceptable	2.8e+02
LadyBug-73-11032-feasres	33753	92244	1.7101e+04	1.9e+01	17	first_order	7.1e+01
LadyBug-318-41628-feasres	127746	359838	8.6309e+04	1.6e+02	14	acceptable	4.6e+03
LadyBug-460-56811-feasres	174573	483754	1.2878e+05	4.8e+02	26	acceptable	2.0e+05
LadyBug-810-88814-feasres	273732	787550	2.0347e+05	5.7e+02	13	acceptable	4.8e+08
LadyBug-1031-110968-feasres	342183	1000530	2.7532e+05	1.8e+03	36	acceptable	1.6e+08
Dubrovnik-202-132796-feasres	400206	1503304	3.3014e+05	4.0e+02	10	first_order	2.2e+03
Dubrovnik-273-176305-feasres	531372	1885940	3.4564e+05	1.2e+03	30	first_order	4.3e+03
Dubrovnik-356-226730-feasres	683394	2510536	4.9482e+05	1.3e+03	19	small_step	3.7e+03
Venice-427-310384-feasres	934995	3398290	1.0621e+06	1.8e+03	20	small_step	1.2e+05
Venice-1350-894716-feasres	2696298	9034252	2.6558e+06	2.0e+03	1	small_step	3.7e+15

This profile compares the LDL-AMD version with the three types of normalization: A, J and None:

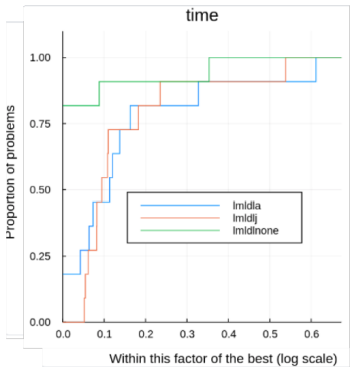

Figure 9: Profile of the LDL normalized versions

The profile confirms the fact that the non-normalized version is faster than the others, while the two other versions seem to be more or less equivalent. However there is not a big time difference between the three versions. Indeed, the horizontal axis is quite spread out, and one can notice that the curves are very close at abscissa 0.35: the three solvers are at least as fast as $2^{0.35} \simeq 1.27$ times slower than the fastest solver on 90% of the problems. Thus, the non-normalized is overall a bit faster than the other versions and gives better results on some problems, but one can use either of the two other versions to avoid bad conditioning problems (even if we have not seen such scenario in our test problems), without suffering a great loss of time and optimality.

4.5 Results with line search

In this section, we compare LDL-AMD with and without line search. The following table presents the results of the LDL-AMD version without normalization and with line search (the results without line search are in Table 3:

As expected, the version with line search seems to be faster, as it performs, overall, less iterations, apart from problem Dubrovnik-356-226730 where the version without line search preforms 5 less iterations than the version with line search. In terms of convergence, the version with line search almost always reaches a bit smaller or equal objective value. On the other hand, the final dual feasibility value often differs in the two versions, with a factor of 10 (in one sens or another), and sometimes with a bigger factor (in problem Ladybug-810-8814 the dual feasibility of the line search version in $1e4$ times smaller, in problems Dubrovnik-273-176305 and Dubrovnik-356-226730 the dual feasibility of the line search version is $1e2$ and $1e3$ bigger). All in all, the two versions seem pretty equivalent in terms of convergence, even if the dual feasibility is often very different (not in the favor of any versions) because, unlike the previous versions we have seen before, the linear problems solved at each iteration differ between the two versions because of the line search. But in terms of time the version with line

Table 11: Results of the solver LDL-AMD without normalization and with linesearch

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3361e+04	2.2e+01	26	acceptable	2.4e+03
LadyBug-73-11032-feasres	33753	92244	1.7099e+04	1.7e+01	13	acceptable	2.5e+02
LadyBug-318-41628-feasres	127746	359838	8.6326e+04	9.0e+01	7	first_order	7.9e+02
LadyBug-460-56811-feasres	174573	483754	1.2855e+05	3.3e+02	17	acceptable	2.0e+05
LadyBug-810-88814-feasres	273732	787550	2.0348e+05	6.0e+02	13	acceptable	7.6e+04
LadyBug-1031-110968-feasres	342183	1000530	2.7523e+05	1.0e+03	19	first_order	3.1e+06
Dubrovnik-202-132796-feasres	400206	1503304	3.3015e+05	5.1e+02	10	small_step	7.6e+02
Dubrovnik-273-176305-feasres	531372	1885940	3.4477e+05	1.5e+03	30	acceptable	4.8e+05
Dubrovnik-356-226730-feasres	683394	2510536	4.9394e+05	1.6e+03	24	acceptable	1.8e+06
Venice-427-310384-feasres	934995	3398290	1.0621e+06	1.3e+03	13	small_step	3.0e+04
Venice-1350-894716-feasres	2696298	9034252	2.6558e+06	1.6e+03	1	small_step	3.7e+15

search seems overall faster because it needs less iterations (because of the iterations that are "hidden" in the line search but that do not need to perform factorization).

The following profile compares the LDL-AMD version with and without line search.

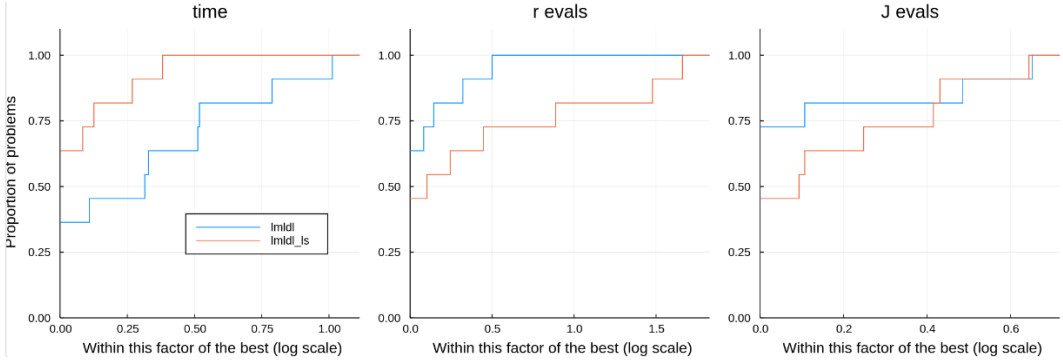

Figure 10: Profile of the LDL-AMD-None version with and without line-search (the version with line-search is "lmldl_ls")

As seen in the tables, the version with line search is faster than the other one (on around 65% of the problems). The profiles comparing the number of residuals and Jacobian evaluation are quite interesting, as we notice that the versions with line search performs more evaluations of the residuals but only a few more evaluations of the Jacobian than the version without line search. This is due to the fact that in the line search algorithm, the residuals can be computed several times in one iteration in order to know whether the new step $\frac{\delta}{\delta_d}$ is accepted, while the Jacobian is computed only one time per iteration (but it is still computed more times in the line search version as a bigger proportion of the iterations are accepted and thus need to compute a new Jacobian).

4.6 Comparison with Scipy's least_square function and Ceres solver

In this section, we compare the results of my solver (the LDL-AMD version) with open source solvers: Scipy's least_square function and Ceres solver.

I created a Python script that run Scipy's least_square function on bundle adjustment problems, using the modeling code given at [6]. It uses sparse finite differences to compute the Jacobian and trust region methods to solve the problems. I used the same tolerances as in my solver and ran the script on the same computer. The following table presents the results of Scipy's least_square function:

Let us compare the results from Scipy's solver with the best version of my solver (LDL-AMD without normalization and with line search). The final objective values of both solvers are quite similar, with often better values obtained from my solver (except on the last the problem, where the objective value of my solver is nearly two times bigger than the one from Scipy). On the other hand, the dual feasibility values are often quite different, but not to any of the two solvers advantage

Table 12: Results of Scipy's least_square function

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3398e4	294	76	acceptable	8.6383e1
LadyBug-73-11032-feasres	33753	92244	1.7112e4	56	8	acceptable	1.7921e2
LadyBug-318-41628-feasres	127746	359838	8.6308e4	165	8	acceptable	2.4758e5
LadyBug-460-56811-feasres	174573	483754	1.2861e5	399	12	acceptable	2.7606e3
LadyBug-810-88814-feasres	273732	787550	2.1239e5	1547	24	acceptable	1.6922e5
LadyBug-1031-110968-feasres	342183	1000530	2.7637e5	831	16	small step	6.5953e24
Dubrovnik-202-132796-feasres	400206	1503304	3.2937e5	1085	10	acceptable	1.9758e3
Dubrovnik-273-176305-feasres	531372	1885940	3.4529e5	1481	12	acceptable	9.2544e3
Dubrovnik-356-226730-feasres	683394	2510536	4.9689e5	4034	15	acceptable	1.60050e4
Venice-427-310384-feasres	934995	3398290	1.0716e6	3875	15	acceptable	1.1900e5
Venice-1350-894716-feasres	2696298	9034252	1.4956e6	18002	22	acceptable	7.3690e11

overall. One notices the very high dual feasibility value ($1e18$ times bigger than the one of my solver) of Scipy's solver on problem LadyBug-1031-110968, and also the high dual feasibility of my solver on the last problem ($1e3$ times bigger than the one of Scipy's solver). However, the high objective and high dual feasibility of my solver on the last problem is due to the fact that my solver only performs one iteration on this problem. Thus, by relaxing the step tolerances, one can reach similar results as the ones of Scipy's solver (here we have chosen the default tolerance values on all problems, but by tuning those tolerances one can obtain better results). So overall, my solver seems a bit better in terms of convergence. In terms of number of iterations, the two solvers seem more or less equivalent, and in terms of time (as the number of iterations varies a lot between the two solvers, the time is compared in terms of mean time per iteration), my solver is almost always faster (except on the last problem) than Scipy's, by at least a factor 2. This comparison with Scipy's solver is quite satisfying as my solver has proven to have equal or better convergence than a standard optimization library, and most importantly, it has proven to be faster.

I downloaded the binary files of Ceres solver [18] and used the file called "bundle_adjuster". It models a bundle adjustment problem from a file and solves it with Ceres. A lot of options are available in Ceres, I chose the solver the most similar to mine. It is called "SPARSE_CHOLESKY", and it computes the Cholesky decomposition of $J^TJ + \lambda I$. If $R^TR = J^TJ + \lambda I$ then $\delta^* = -R^{-1}R^TJ^Tr$. The matrix R is the same as in the QR factorization of $\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix} = QR$. Indeed, $J^TJ + \lambda I = \begin{bmatrix} J^T & \sqrt{\lambda}I \end{bmatrix} \begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix} = R^TQ^TQR = R^TR$, as Q is orthonormal. Ceres solver also has a QR version very similar to mine, but not comparable in terms of time as they use a dense QR factorization which is much more longer than SuiteSparseQR. So I chose "SPARSE_CHOLESKY", as it is somehow equivalent to my QR version but not with the same factorization type, and because it uses a Cholesky factorization which is close to a LDL factorization, even if they do not factorize the same matrix as in my LDL version. Once again, I used the same tolerances as in my solver and ran the script on the same computer. Table 13 contain the results of Ceres solver:

Table 13: Results of Ceres solver

name	nvar	nequ	objective	elapsed_time	iter	status	dual_feas
LadyBug-49-7776-feasres	23769	63686	1.3345e4	7.86	25	acceptable	2.48e1
LadyBug-73-11032-feasres	33753	92244	1.7100e4	4.02	8	acceptable	6.65e1
LadyBug-318-41628-feasres	127746	359838	8.6341e4	27.66	10	acceptable	4.57e4
LadyBug-460-56811-feasres	174573	483754	1.2870e5	62.94	14	acceptable	2.56e3
LadyBug-810-88814-feasres	273732	787550	2.0471e5	546.00	56	acceptable	1.80e4
LadyBug-1031-110968-feasres	342183	1000530	2.7525e5	148.20	13	acceptable	2.09e6
Dubrovnik-202-132796-feasres	400206	1503304	3.2513e5	289.25	33	acceptable	7.16e5
Dubrovnik-273-176305-feasres	531372	1885940	3.4477e5	162.84	13	acceptable	7.82e4
Dubrovnik-356-226730-feasres	683394	2510536	4.9393e5	183.18	10	acceptable	6.68e5
Venice-427-310384-feasres	934995	3398290	1.0713e6	373.50	14	acceptable	8.19e5
Venice-1350-894716-feasres	2696298	9034252	1.5005e6	1827.87	11	small step	8.46e13

Let us compare the results of Ceres solver on these problems with the ones of my solver with line search. The final objective values of the two solvers are very close on each problem (apart from the last one, but once again this is due to the default tolerances). As for the dual feasibility values, they are often quite different, but they are equivalent overall. The two solvers also seem equivalent in terms of number of iterations. Now, in terms of execution time (per iteration), Ceres solver is in average three times faster than my solver. Once again the convergence results of my solver are quite satisfying, but this time the execution time is a bit disappointing. However, Ceres solver is a library specialized in non-linear solvers (contrary to Scipy), and has been developed by a whole team of brilliant researchers and engineers during probably more than four months (contrary to my solver).

4.7 Use of several precisions

This section presents the results of the solver used as described below:

- first, the solver is launched in simple precision until one of the stopping criteria (with tolerances in simple precision) is met, we keep the solution x_{sol} in memory,
- then, the solver is launched in double precision with $x_0 = x_{sol}$ until one of the stopping criteria (with tolerances in double precision) is met.

As the computer used does not have a simple precision processor, we will not observe a decrease of the execution time. However, we know that the computations in simple precision will be twice as slow as the computations in double precision. Thus, by comparing the number of iterations in simple precision and in double precision, we will be able to compute a time saving, as well as an energy saving. Indeed, we saw in Section 3.13 that dividing the precision by 2, divides the energy produced by 4. Let us denote 4α the energy produced by an iteration in double precision. If a problem needs k iterations in double precision to be solved, or k_1 iterations in simple precision then k_2 iterations in double precision, the percentage of energy saved is:

$$E = 1 - \frac{k_1\alpha + 4k_2\alpha}{4k\alpha} = 1 - \frac{k_1 + 4k_2}{4k}.$$

The following table presents the results of the solver used in simple then double precision on a few problems. k_1 denotes the number of iterations in simple precision, k_2 the number of iterations in double precision, and k the number of total iterations when the solver (LDL-AMD without normalization, line search is not used as the time spent for one iteration is variable when line search is used so it would have made the computations less accurate) is ran in double precision from the start. E is the percentage of energy saved, computed as explained above, and T is the percentage of time saved (following the same reasoning as for the energy $T = 1 - \frac{k_1+2k_2}{2k}$). The tolerances used for the simple precision are: $oatol = ortol = atol = 1e-4$, $rtol = 1e-3$, $satol = 1e-6$ and $srtol = 1e-7$.

Table 14: Results of my solver using simple precision then double precision

name	objective	k_1	k_2	k	T	E
LadyBug-73-11032-feasres	1.7168e4	17	2	17	38%	63%
LadyBug-138-41628-feasres	6.0103e4	49	2	27	2%	67%
LadyBug-318-41628-feasres	8.6403e4	17	2	14	25%	55%
LadyBug-460-56811-feasres	1.2988e5	27	1	26	44%	71%

The results show that, by well tuning the tolerances of the solver in simple precision, most iterations can be performed in simple precision (in the problems in the table, no more than 2 iterations have been performed in double precision). This leads to a save up to 44% of time and 71% of energy. Moreover, even when the time gain is not that big (for instance 2% in the second problem), the save of energy is huge (more than the half on the four problems tested). The final objective values are a bit bigger than without this strategy, but by refining the tolerances of the solver in double precision (here the default values are used), one can obtain the same results as before.

Conclusion

During this internship, I created an interface to read the bundle adjustment datasets given in [26] and modeled such problems using NLPModels, which was way easier to use than JuMP as they allow vector modeling. As I did not found a working sparse automatic differentiation module in Julia, I computed the Jacobian by hand, and used multithreading to speed up the computations. I coded a solver based on the Levenberg-Marquardt algorithm, using QR and LDL factorizations, AMD and Metis permutation algorithms, normalization and line search.

The experimental results proved that the QR version is much slower than the LDL, and this speed difference increases when the problem gets bigger. This slowness is due to the need to compute the permutation at each iteration in the QR version. The normalized versions allow a better conditioning of the problems, which provides a stabler solver (especially for the QR version, which produces errors due to numerical approximations when used without normalization). Finally, the LDL version with AMD permutation has shown to be the faster version of my solver, with the same convergence properties as the other versions, and the speed can be increased again by activating the line search option.

The comparison with other solvers has shown that my solver is quite competitive with Scipy's solver and Ceres solver in terms of convergence, and that it is in average two times faster than Scipy's solver and three times slower than Ceres. However, the advantage of my solver is that it is coded is Julia and thus allows the user to run it in several precisions in a very efficient way, in order to gain time and energy (in small precisions) or accuracy (in big precisions). The experimental results have shown that we can save up to 44% of time and 71% of energy by running the solver in simple precision first before refining the results with double precision.

There is much scope for further development of the solver I created during this internship. Indeed, the comparison with other solvers have shown that my solver converges well, but that it could be faster. The QR version could be improved by changing Julia's interface to SuiteSparseQR to allow the user to give the permutation vector as parameter (which is allowed in the C++ version) which should make this version much more faster. The Givens strategy could be improved, or the version with block-banded matrices discussed in [20] could be implemented or interfaced, to avoid computing the full QR factorization at each iteration. Moreover, the linear problems solved at each iteration could be solved approximately, like in [2], and the line search strategy could be refined. All those improvements will increase the execution time of each iteration, but another way to gain time is to perform less iterations (and still converge to the same point). One way to reduce the number of iterations would be to use the improvement discussed in Section 3.13. It consists in using second order derivatives, which will allow to find a better descent direction at each iteration, and at low cost if we compute the second order derivatives in simple precision for instance.

During this internship, I have learnt a lot about bundle adjustment problems. I have been able to understand the datasets given in [26] and to create an interface to read them in Julia. I have also learnt a lot about Julia, a language that I had never used before, and its usefulness to model and solve optimization problems, with libraries such as JuMP and NLPModels. I also learnt a lot about optimization, especially about the Levenberg-Marquardt algorithm. I used LDL and QR factorizations for my solver, but more than that I went quite deep in understanding how sparse QR factorizations work by spending some time trying to implement the Givens strategy. I also got used to standard techniques such as normalization and line search.

Appendices

Appendix A Structure of the datasets

The datasets from [26] look like this

$$
\begin{bmatrix}
ncam & npoints & nobs & \\
camera\_index & point\_index & x_1^{obs} & y_1^{obs} \\
. & . & . & . \\
. & . & . & . \\
. & . & . & . \\
camera\_index & point\_index & x_{nobs}^{obs} & y_{nobs}^{obs} \\
rx_1 & & & \\
ry_1 & & & \\
rz_1 & & & \\
tx_1 & & & \\
ty_1 & & & \\
tz_1 & & & \\
f_1 & & & \\
k1_1 & & & \\
k2_1 & & & \\
. & & & \\
. & & & \\
. & & & \\
rx_{ncam} & & & \\
ry_{ncam} & & & \\
rz_{ncam} & & & \\
tx_{ncam} & & & \\
ty_{ncam} & & & \\
tz_{ncam} & & & \\
f_{ncam} & & & \\
k1_{ncam} & & & \\
k2_{ncam} & & & \\
x_1 & & & \\
y_1 & & & \\
z_1 & & & \\
. & & & \\
. & & & \\
. & & & \\
x_{npnts} & & & \\
y_{npnts} & & & \\
z_{npnts} & & &
\end{bmatrix}
$$

Appendix B Computations for the jacobian of the residuals

First of all, we have:

$$\forall a \in \{x, y, z\},\ \frac{\partial \theta}{\partial r_a} = \frac{r_a}{\theta} = k_a$$

And:

$$\frac{\partial k_a}{\partial r_a} = \frac{\partial}{\partial r_a}\left(\frac{r_a}{\sqrt{r_x^2 + r_y^2 + r_z^2}}\right) = \frac{\theta - r_a k_a}{\theta^2} = \frac{1 - k_a^2}{\theta}$$

Then:

$$\begin{aligned}
\frac{\partial}{\partial r_x}(k_x^2 x + k_y k_x y + k_z k_x z) &= \frac{\partial}{\partial r_x}\left(\frac{r_x^2}{r_x^2 + r_y^2 + r_z^2}x + \frac{r_x r_y}{r_x^2 + r_y^2 + r_z^2}y + \frac{r_x r_z}{r_x^2 + r_y^2 + r_z^2}z\right) \\
&= 2\frac{r_x\theta^2 - r_x^3}{\theta^4}x + \frac{r_y\theta^2 - 2r_x^2 r_y}{\theta^4}y + \frac{r_z\theta^2 - 2r_x^2 r_z}{\theta^4}z \\
&= 2x\frac{k_x - k_x^3}{\theta} + y\frac{k_y - 2k_y k_x^2}{\theta} + z\frac{k_z - 2k_z k_x^2}{\theta} \\
&= \frac{1}{\theta}(2xk_x(1 - k_x^2) + yk_y(1 - 2k_x^2) + zk_z(1 - 2k_x^2))
\end{aligned}$$

And:

$$\begin{aligned}
\frac{\partial}{\partial r_y}(k_x^2 x + k_y k_x y + k_z k_x z) &= \frac{\partial}{\partial r_y}\left(\frac{r_x^2}{r_x^2 + r_y^2 + r_z^2}x + \frac{r_x r_y}{r_x^2 + r_y^2 + r_z^2}y + \frac{r_x r_z}{r_x^2 + r_y^2 + r_z^2}z\right) \\
&= -\frac{2r_y r_x^2}{\theta^4}x + \frac{r_x\theta^2 - 2r_x r_y^2}{\theta^4}y - \frac{2r_x r_y r_z}{\theta^4}z \\
&= -2x\frac{k_y k_x^2}{\theta} + y\frac{k_x - 2k_x k_y^2}{\theta} - 2z\frac{k_x k_y k_z}{\theta} \\
&= \frac{1}{\theta}(-2xk_x^2 k_y + yk_x(1 - 2k_y^2) - 2zk_x k_y k_z)
\end{aligned}$$

Similarly:

$$\begin{aligned}
\frac{\partial}{\partial r_z}(k_x^2 x + k_y k_x y + k_z k_x z) &= \frac{\partial}{\partial r_z}\left(\frac{r_x^2}{r_x^2 + r_y^2 + r_z^2}x + \frac{r_x r_y}{r_x^2 + r_y^2 + r_z^2}y + \frac{r_x r_z}{r_x^2 + r_y^2 + r_z^2}z\right) \\
&= -\frac{2r_z r_x^2}{\theta^4}x - \frac{2r_x r_y r_z}{\theta^4}y + \frac{r_x\theta^2 - 2r_x r_z^2}{\theta^4}z \\
&= -2x\frac{k_z k_x^2}{\theta} - 2y\frac{k_x k_y k_z}{\theta} + z\frac{k_x - 2k_x k_z^2}{\theta} \\
&= \frac{1}{\theta}(-2xk_x^2 k_z - 2yk_x k_y k_z + zk_x(1 - 2k_z^2))
\end{aligned}$$

By similar computations we get: $\frac{\partial}{\partial r_x}(k_x k_y x + k_y^2 y + k_z k_y z) = \frac{1}{\theta}(xk_y(1 - 2k_x^2) - 2yk_x k_y^2 - 2zk_x k_y k_z)$

$\frac{\partial}{\partial r_y}(k_x k_y x + k_y^2 y + k_z k_y z) = \frac{1}{\theta}(xk_x(1 - 2k_y^2) + 2yk_y(1 - k_y^2) + zk_z(1 - 2k_y^2))$

$\frac{\partial}{\partial r_z}(k_x k_y x + k_y^2 y + k_z k_y z) = \frac{1}{\theta}(-2xk_x k_y k_z - 2yk_y^2 k_z + zk_y(1 - 2k_z^2))$

And: $\frac{\partial}{\partial r_x}(k_x k_z x + k_y k_z y + k_z^2 z) = \frac{1}{\theta}(xk_z(1 - 2k_x^2) - 2yk_x k_y k_z - 2zk_x k_z^2)$

$\frac{\partial}{\partial r_y}(k_xk_zx + k_yk_zy + k_z^2z) = \frac{1}{\theta}(-2xk_xk_yk_z + yk_z(1-2k_y^2) - 2zk_xk_z^2)$

$\frac{\partial}{\partial r_z}(k_xk_zx + k_yk_zy + k_z^2z) = \frac{1}{\theta}(xk_x(1-2k_z^2) + yk_y(1-2k_z^2) + 2zk_z(1-k_z^2))$

Now, let us compute the derivatives of $P_1.x$:

- $\frac{\partial P_1.x}{\partial x} = cos(\theta) + (1-cos(\theta))k_x^2$
- $\frac{\partial P_1.x}{\partial y} = -sin(\theta)k_z + (1-cos(\theta))k_yk_x$
- $\frac{\partial P_1.x}{\partial z} = sin(\theta)k_y + (1-cos(\theta))k_zk_x$
- $\frac{\partial P_1.x}{\partial r_x} = -sin(\theta)xk_x + cos(\theta)k_x(k_yz - k_zy) + sin(\theta)\frac{-k_yk_xz+k_zk_xy}{\theta} + sin(\theta)k_x^2(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(2xk_x(1-k_x^2) + yk_y(1-2k_x^2) + zk_z(1-2k_x^2))$
- $\frac{\partial P_1.x}{\partial r_y} = -sin(\theta)xk_y + cos(\theta)k_y(k_yz - k_zy) + sin(\theta)\frac{(1-k_y^2)z+k_zk_yy}{\theta} + sin(\theta)k_xk_y(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(-2xk_x^2k_y + yk_x(1-2k_y^2) - 2zk_xk_yk_z)$
- $\frac{\partial P_1.x}{\partial r_z} = -sin(\theta)xk_z + cos(\theta)k_z(k_yz - k_zy) + sin(\theta)\frac{-k_yk_zz-(1-k_z^2)y}{\theta} + sin(\theta)k_xk_z(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(-2xk_x^2k_z - 2yk_xk_yk_z + zk_x(1-2k_z^2))$

Then the derivatives of $P_1.y$:

- $\frac{\partial P_1.y}{\partial x} = sin(\theta)k_z + (1-cos(\theta))k_yk_x$
- $\frac{\partial P_1.y}{\partial y} = cos(\theta) + (1-cos(\theta))k_y^2$
- $\frac{\partial P_1.y}{\partial z} = -sin(\theta)k_x + (1-cos(\theta))k_zk_y$
- $\frac{\partial P_1.y}{\partial r_x} = -sin(\theta)yk_x + cos(\theta)k_x(k_zx - k_xz) + sin(\theta)\frac{-k_zk_xx-(1-k_x^2)z}{\theta} + sin(\theta)k_yk_x(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(xk_y(1-2k_x^2) - 2yk_xk_y^2 - 2zk_xk_yk_z)$
- $\frac{\partial P_1.y}{\partial r_y} = -sin(\theta)yk_y + cos(\theta)k_y(k_zx - k_xz) + sin(\theta)\frac{-k_zk_yx+k_xk_yz}{\theta} + sin(\theta)k_yk_y(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(xk_x(1-2k_y^2) + 2yk_y(1-k_y^2) + zk_z(1-2k_y^2))$
- $\frac{\partial P_1.y}{\partial r_z} = -sin(\theta)yk_z + cos(\theta)k_z(k_zx - k_xz) + sin(\theta)\frac{(1-k_z^2)x+k_xk_zz}{\theta} + sin(\theta)k_yk_z(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(-2xk_xk_yk_z - 2yk_y^2k_z + zk_y(1-2k_z^2))$

Then the derivatives of $P_1.z$:

- $\frac{\partial P_1.z}{\partial x} = -sin(\theta)k_y + (1-cos(\theta))k_zk_x$
- $\frac{\partial P_1.z}{\partial y} = sin(\theta)k_x + (1-cos(\theta))k_zk_y$
- $\frac{\partial P_1.z}{\partial z} = cos(\theta) + (1-cos(\theta))k_z^2$
- $\frac{\partial P_1.z}{\partial r_x} = -sin(\theta)zk_x + cos(\theta)k_x(k_xy - k_yx) + sin(\theta)\frac{(1-k_x^2)y+k_xk_yx}{\theta} + sin(\theta)k_zk_x(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(xk_z(1-2k_x^2) - 2yk_xk_yk_z - 2zk_xk_z^2)$
- $\frac{\partial P_1.z}{\partial r_y} = -sin(\theta)zk_y + cos(\theta)k_y(k_xy - k_yx) + sin(\theta)\frac{-k_xk_yy-(1-k_y^2)x}{\theta} + sin(\theta)k_zk_y(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(-2xk_xk_yk_z + yk_z(1-2k_y^2) - 2zk_xk_z^2)$
- $\frac{\partial P_1.z}{\partial r_z} = -sin(\theta)zk_z + cos(\theta)k_z(k_xy - k_yx) + sin(\theta)\frac{-k_xk_zy+k_zk_yx}{\theta} + sin(\theta)k_zk_z(k_xx + k_yy + k_zz) + \frac{1-cos(\theta)}{\theta}(xk_x(1-2k_z^2) + yk_y(1-2k_z^2) + 2zk_z(1-k_z^2))$

Now, let us compute the derivatives of $P_2.x$ and $P_2.y$:

- $\frac{\partial P_2.x}{\partial x} = -\frac{1}{z}$
- $\frac{\partial P_2.x}{\partial y} = 0$
- $\frac{\partial P_2.x}{\partial z} = \frac{x}{z^2}$
- $\frac{\partial P_2.y}{\partial x} = 0$

- $\frac{\partial P_2.y}{\partial y} = -\frac{1}{z}$
- $\frac{\partial P_2.y}{\partial z} = \frac{y}{z^2}$

And finally, let us compute the derivatives of $P_3.x$ and $P_3.y$:

- $\frac{\partial P_3.x}{\partial x} = f(1 + k_1(x^2 + y^2) + k_2(x^2 + y^2)^2) + f(2k_1x + k_2(4x^3 + 4xy^2))x$
- $\frac{\partial P_3.x}{\partial y} = f(2k_1y + k_2(4y^3 + 4yx^2))x$
- $\frac{\partial P_3.x}{\partial f} = (1 + k_1(x^2 + y^2) + k_2(x^2 + y^2)^2)x$
- $\frac{\partial P_3.x}{\partial k_1} = f(x^2 + y^2)x$
- $\frac{\partial P_3.x}{\partial k_2} = f(x^2 + y^2)^2x$

- $\frac{\partial P_3.y}{\partial x} = f(2k_1x + k_2(4x^3 + 4xy^2))y$
- $\frac{\partial P_3.y}{\partial y} = f(1 + k_1(x^2 + y^2) + k_2(x^2 + y^2)^2) + f(2k_1y + k_2(4y^3 + 4yx^2))y$
- $\frac{\partial P_3.y}{\partial f} = (1 + k_1(x^2 + y^2) + k_2(x^2 + y^2)^2)y$
- $\frac{\partial P_3.y}{\partial k_1} = f(x^2 + y^2)y$
- $\frac{\partial P_3.y}{\partial k_2} = f(x^2 + y^2)^2y$

Appendix C Algorithms for the Givens strategy

Here are the algorithms I coded for the Givens strategy. The main one is called fullQR_Givens!, and the one that computes the Givens rotations is called apply_givens!.

In fullQR_Givens!, we go through each row k of the matrix R (starting from the last one), we perform a Givens rotation with the diagonal element of R to eliminate a $\sqrt{\lambda}$ and store the rotation in a list. The fuction apply_givens stores the elements created in $\sqrt{\lambda}I$ by the rotation in a 1D vector called news (at iteration k, news contains the k-th row of $\sqrt{\lambda}I$. As we know that R will have all his non-zero elements around the last columns, we do not want to go through the whole row of R. Thus, we look for the second non-zero element (the first one is the diagonal element) of row k of R. As R is stored in compressed sparse columns format, it is not easy to access the elements of a whole row, that is why R^T is an argument of the function, so that we have access to row k of R by the column k of R^T. Then we perform the Givens rotations the eliminate the new elements and store them in the list of rotations.

Algorithm 3 fullQR_Givens!

```
counter = 0
for k = n → 1 do
    G, r = givens(R[k, k], √λ, k, m + k)
    min_news = apply_givens!(R, Rt, G, r, news, n, m, true, 0)
    counter+ = 1
    G_list[counter] = G
    if Rt.colptr[k] < nnz_R then
        vbeg = Rt.rowval[Rt.colptr[k] + 1]
    else
        beg = n + 1
    end if
    for col = beg → n do
        if news[col] ≠ 0 then
            G, r = givens(R[col, col], news[col], col, m + k)
            min_news = apply_givens!(R, Rt, G, r, news, n, m, false, min_news + 1)
            counter+ = 1
            G_list[counter] = G
        end if
    end for
end for
```

The function apply_givens! takes as input the Givens rotation to perform. There are two cases: either this rotation is performed to eliminate a $\sqrt{\lambda}$ and we know that the news vector is empty so far, either this rotation is performed to eliminate the new elements in $\sqrt{\lambda}I$. In the first case, we look for the second non-zero element of the row just like before, and we perform the Givens rotation starting from this column. We keep in memory the index of the first new element created. In the second case, we start from the first new element that has not yet been eliminated and we keep in memory what will be the next one.

Algorithm 4 apply_givens!

$min\_news = 0$
$min\_found = false$
if $diag$ **then**
  $R[G.i1, G.i1] = r$
  for $k = Rt.colptr[G.i1] + 1 : Rt.colptr[G.i1 + 1] - 1$ **do**
    $col = Rt.rowval[k]$
    $R[G.i1, col], news[col] = G.c * R[G.i1, col], -G.s * R[G.i1, col]$
    if $!min_found \;\&\&\; news[col]! = 0$ **then**
      $min\_found = true$
      $min\_news = col$
    end if
  end for
else
  $R[G.i1, G.i1] = r$
  $news[G.i1] = 0$
  if $old\_min\_news! = 0$ **then**
    $beg = old\_min\_news$
  else
    $beg = n + 1$
  end if
  for $col = beg : n$ **do**
    $R[G.i1, col], news[col] = G.c * R[G.i1, col] + G.s * news[col], -G.s * R[G.i1, col] + G.c * news[col]$
    if $!min\_found \;\&\&\; news[col]! = 0$ **then**
      $min\_found = true$
      $min\_news = col$
    end if
  end for
end if

References

[1] Jump. https://www.juliaopt.org/JuMP.jl/stable/.

[2] An inexact levenberg-marquardt method for large sparse non-linear least squares. https://www.cambridge.org/core/journals/anziam-journal/article/an-inexact-levenbergmarquardt-method-for-large-sparse-nonlinear-least-squres/C92147BBF93B355F317369800FF8CF6A, 1983.

[3] Algorithm 849: A concise sparse cholesky factorization package. https://dl.acm.org/doi/10.1145/1114268.1114277, 2005.

[4] Pushing the envelope of modern methods for bundle adjustment. https://www.microsoft.com/en-us/research/wp-content/uploads/2010/06/Jeong-CVPR10.pdf, 2010.

[5] Suitesparseqr. https://github.com/PetterS/SuiteSparse, 2012.

[6] Large-scale bundle adjustment in scipy. https://scipy-cookbook.readthedocs.io/items/bundle_\adjustment.html, 2016.

[7] Course on slam. https://www.iri.upc.edu/people/jsola/JoanSola/objectes/toolbox/courseSLAM.pdf, 2017.

[8] Ldlfactorizations. https://github.com/JuliaSmoothOptimizers/LDLFactorizations.jl, 2017.

[9] Bundle adjustment. https://en.wikipedia.org/wiki/Bundle_adjustment, 2019.

[10] Bundle adjustment revisited. https://arxiv.org/pdf/1912.03858.pdf, 2019.

[11] Julia smooth optimizers. https://github.com/JuliaSmoothOptimizers, 2019.

[12] Nlpmodels. https://github.com/JuliaSmoothOptimizers/NLPModels.jl, 2019.

[13] Nlsmodels. https://github.com/JuliaSmoothOptimizers/NLPModels.jl/blob/master/src/NLSModels.jl, 2019.

[14] Sparsedifftools. https://github.com/JuliaDiff/SparseDiffTools.jl, 2019.

[15] Cutest. https://github.com/JuliaSmoothOptimizers/CUTEst.jl, 2020.

[16] Levenberg-marquardt algorithm. https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_\algorithm, 2020.

[17] Rodrigues'rotation formula. https://en.wikipedia.org/wiki/Rodrigues'_rotation_formula, 2020.

[18] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. http://ceres-solver.org.

[19] Charles Van Loaner and Christian Bischof. The WY representation for products of Householder matrices. Cornell University, 1987.

[20] Andrew Fitzgibbon, Jan Svoboda, Thomas Cashman. Qrkit: Sparse, composable qr decompositions for efficient and stable solutions to problems in computer vision. https://arxiv.org/pdf/1802.03773.pdf, 2018.

[21] George Karypis. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0. 2013.

[22] Jorge J. Moré. Lecture Notes in Mathematics. G.A Watson, 1978.

[23] Hans Bruun Nielsen. damping parameter in marquardt's method. http://www2.imm.dtu.dk/documents/ftp/tr99/tr05_99.pdf.

[24] Jorge Nocedal and Stephen J. Wright. Numerical Optimization. Springer, 2006.

[25] Iain S. Duff, Patrick R. Amestoy, Timothy A. Davis. An approximate minimum degree ordering algorithm. http://faculty.cse.tamu.edu/davis/publications_files/An_Approximate_Minimum_Degree_Ordering_Algorithm.pdf, 1996.

[26] Steven M. Seitz, Richard Szeliski, Sameer Agarwal, Noah Snavely. Bundle adjustment in the large. https://grail.cs.washington.edu/projects/bal, 2010.

[27] Steven M. Seitz, Richard Szeliski, Sameer Agarwal, Noah Snavely. Bundle adjustment in the large. 2010.

[28] Micheal A. Saunders. Solution of sparse rectangular systems using lsqr and craig, 1995.

[29] Jin yan Fan and Ya xiang Yuan. On the quadratic convergence of the levenberg-marquardt method without nonsingularity assumption. https://www.researchgate.net/publication/220261230_On_the_Quadratic_Convergence_of_the_Levenberg-Marquardt_Method_without_Nonsingularity_Assumption, 2001.