Least Square Estimation using Givens Rotation for QR Factorization

Kartik Tiwari - Ashoka University
Dr. Hari Hablani - IIT, Indore

December 16, 2020

Abstract

In this technical report, I describe the details of the code that I had written to fit a curve for a set of observational data points using the Givens Rotation method of performing QR factorization. This document is structured in a manner where I first describe what Givens Rotation is, then the complete sequence of how I used Givens Rotation to find the R factor and, finally, how I use the R factor to fit a curve that matches the measured data points.

1 Introduction

The crux of the problem that I solve here is that often observational data includes noise that prevents the data points from perfectly aligning with the theoretically modelled equations. What I intend to do is to fit a curve that matches the observed data within the bounds of a reasonable error envelope. This is a form of the least squares squares estimation problem. The name 'LSE' comes from the idea that we are trying to minimize the sum of squares of the difference between observational data and the mathematical models. We square this difference in order to account for the fact that deviations can be either positive or negative.

2 QR Factorization

We are going to fit a curve by finding the three constants in a quadratic equation. One way to solve a system of equations in the matrix form is by finding the Q and R factors of that matrix. For a matrix A of size m x n, the R factor is an upper triangular matrix of the rank m. As we move further in the report, I shall explain how we are going to use the QR factors to fit a curve for our set of data points. Ideally, for three unknown constants, we only need three equations to fit a curve. However, the more data points we have, the closer of an approximation we can make using the LSE. Also, there are multiple ways to solve this problem (Grahm-Schmidt, Householder, Givens etc.) and each method has its own advantages. The one that we are going to use is the Givens Rotation method.

3 Givens Rotation

The principle behind Givens Rotation is that by rotating a vector in a very specific manner, you can transform certain elements of that vector to zero. This technique can, therefore, be used to find the upper triangular matrix R.

Essentially, an element of a vector becomes zero when that vector becomes coincident to one of the coordinate axis in the reference frame. To make a vector coincident to an axis, we need to find the angle between the vector of interest and the axis of our choice. This can be done using basic trigonometry.

We know that a Rotation matrix that performs a rotation by an angle ϕ is of the form,

$$\begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \tag{1}$$

Building on this idea, a transformation matrix that transforms only a specific element in a matrix of dimension n x m would be essentially an identity matrix with 4 of its elements replaced with the elements of our rotation matrix.

To transform the element in the ith row and the kth column of an n x m matrix A, we take an n-dimensional identity matrix U and perform the following replacements.

$$\begin{pmatrix} U_{ii} & U_{ik} \\ U_{ki} & U_{kk} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \tag{2}$$

This U shall be our final rotation matrix.

3.1 Performing Rotation Once

In the code associated with this technical report, I first defined a function `givens_once()` that performs Givens rotation once on a matrix. This means, every time we use this function, one specific element of our choice in the matrix goes to zero.

```
def givens_once(i, k, A):

    #Python stars counting from 0. To avoid confusion, let's switch to our 'natural' index counting
    i_s = i - 1
    k_s = k - 1

    #Shifting to Sympy's Matrix Notations to Book's Matrix Notations
    i = k_s
    k = i_s

    #Count number of rows in Matrix A
    n = A.shape[0]

    #Create Identity Matrix n X n called U
    U = eye(n)

    #Calculating Constants
    c = (1 / ((A[i, i])**2 + (A[k, i]**2))**0.5) * A[i, i]
    s = (1 / ((A[i, i])**2 + (A[k, i]**2))**0.5) * A[k, i]

    #Modifying Matrix U's elements
    U[i, i], U[k, i], U[i, k], U[k, k] = c, -s, s, c

    A_prime = U * A

    return U, A_prime
```

Initial few lines in the code only make the code more intuitive to use. This function creates an identity matrix of rank n (i. e. numbers of rows in input matrix) and swaps out required elements

from the identity matrix as described in Eq. 1 and Eq. 2. It then calculates the constants c and s using the following expression -

$$\begin{pmatrix} c \\ s \end{pmatrix} = \frac{1}{\sqrt{A_{ii}^2 + A_{ki}^2}} \begin{pmatrix} A_{ii} \\ A_{ki} \end{pmatrix} \tag{3}$$

These constants are calculated so as to change A_{ii} and A_{ki} into $A_{ii}^{'}$ and $A_{ki}^{'}$ after the rotation.

$$\begin{pmatrix} A_{ii}^{'} \\ A_{ki}^{'} \end{pmatrix} = \begin{pmatrix} \sqrt{A_{ii}^2 + A_{ki}^2} \\ 0 \end{pmatrix}$$

As an example, let us take the matrix mentioned in Exercise 8.1 *Montenbruck-Gill* and rotate it in a way that transforms the element occupying 2^{nd} row and 1^{st} column to zero.

$$A = \begin{bmatrix} 1 & 0.04 & 0.0016 \\ 1 & 0.32 & 0.1024 \\ 1 & 0.51 & 0.2601 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix}$$

To perform the requried operation, we would give out function the following inputs - `givens_once(2, 1, A)` and we will receive two separate matrices as our output.

$$U = \begin{bmatrix} 0.7071 & 0.7071 & 0 & 0 & 0 & 0 & 0 \\ -0.7071 & 0.7071 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } A^{'} = \begin{bmatrix} 1.4142 & 0.2546 & 0.0735 \\ 0 & 0.198 & 0.0713 \\ 1 & 0.51 & 0.2601 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix}$$

Here, U is our rotation matrix (identity matrix modified for our requirements) and $A^{'}$ is the the rotated matrix A where the element of interest has been transformed to 0.

3.2 Performing Complete Sequence of Rotations

To proceed further with our QR factorization, we need to perform the Givens Rotation multiple times in a very specific order. Essentially, what we will be doing is parsing through each row of our input matrix and then performing the required transformations to convert elements to zero. The goal, by the end of our complete sequence of transformations, is to achieve an upper triangular matrix that shall act as the R factor of our input matrix A.

This action is performed by another function defined in the code called `givens_complete()`. This function performs a couple of other things in addition to finding the R factor but we shall talk about these things in the next subsection.

The part of the code that performs the sequential rotations is a nested-while loop, described below-

```
def givens_complete(A, b):
    ...
    ###################
    ## Solving For R ##
    ###################

    while i <= n:
        while k < i and k <= m:
            R.append(givens_once(i, k, R[rank][1]))
            rank = rank + 1
            k = k + 1                                    #Parsing Column-wise
        k = 1
        i = i + 1                                        #Parsing Row-wise

    #Splicing the matrix so the output is a square upper triangular matrix
    R_out = R[rank][1][0:m, 0:m]

    return ..., R_out,...
```

Essentially, it checks whether the column index of an element is less than its row index. If yes, then it performs givens rotation on that element and moves to the next column. If not, then it moves to the next row and then starts again from the first column. This repeats until the function has transformed all elements of the last row. What we have by the end of all rotations is a matrix that has an upper triangular component and another null component. The last line in the listed code splits the matrix into two and only stores the upper-triangular component of our transformed matrix. This upper triangular component is the R factor.

So the more complete transformation of our input matrix A would look like this -

$$\begin{bmatrix} 1 & 0.04 & 0.0016 \\ 1 & 0.32 & 0.1024 \\ 1 & 0.51 & 0.2601 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \rightarrow \begin{bmatrix} 1.4142 & 0.2546 & 0.0735 \\ 0 & 0.198 & 0.0713 \\ 1 & 0.51 & 0.2601 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \rightarrow \begin{bmatrix} 1.732 & 0.5023 & 0.2102 \\ 0 & 0.198 & 0.0713 \\ 0 & 0.2694 & 0.1699 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} 1.732 & 0.5023 & 0.2102 \\ 0 & 0.3343 & 0.1791 \\ 0 & 0 & 0.0432 \\ 1 & 0.73 & 0.5329 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \rightarrow \begin{bmatrix} 2.0 & 0.8 & 0.4485 \\ 0 & 0.3343 & 0.1791 \\ 0 & 0 & 0.0432 \\ 0 & 0.381 & 0.3564 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \rightarrow \begin{bmatrix} 2.0 & 0.8 & 0.4485 \\ 0 & 0.5069 & 0.386 \\ 0 & 0 & 0.0432 \\ 0 & 0 & 0.1004 \\ 1 & 1.03 & 1.0609 \\ 1 & 1.42 & 2.0164 \\ 1 & 1.6 & 2.56 \end{bmatrix} \ldots$$

until we reach

$$A^{'} = \begin{bmatrix} 2.6458 & 2.1355 & 2.4697 \\ 0 & 1.4049 & 2.372 \\ 0 & 0 & 0.6178 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

That gets split into a zero component and an upper triangular component R,

$$R = \begin{bmatrix} 2.6458 & 2.1355 & 2.4697 \\ 0 & 1.4049 & 2.372 \\ 0 & 0 & 0.6178 \end{bmatrix}$$

IMPORTANT NOTE: *This algorithm isn't the most computationally efficient way to perform the QR factorization. It is not the most mathematically elegant way either. However, it is the most fundamental method that can be employed to solve this problem. We shall discuss more about the memory and computational complexity in the last section of this technical report.*

3.3 Solving the Equations

As mentioned before, the function `givens_complete()` performs others operations in addition to finding the R factor. One of these is calculating the Q factor. Using some clever techniques, we can actually solve our set of equations without even requiring the Q factor. However, as I emphasized in the previous section, the method discussed is not the most efficient technique. It's the easiest one.

Now, since Givens Rotation transforms one element to zero at a time, we have the freedom to process one row after another. `givens_complete()` then lets us exploit the following relation:

$$(U_n U_{n-1} ... U_3 U_2) A = Q^T A = \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{4}$$

$$(U_n U_{n-1} ... U_3 U_2) b = Q^T b = \begin{pmatrix} d \\ r \end{pmatrix} \tag{5}$$

U_i represents the transformations required to transform sub-diagonal elements of i^{th} row to zero. We define any particular U_i matrix as the following:

$$U_i = U_{i,max(i-1,m)} ... U_{i,2} U_{i,1}$$

As one might have noticed, unlike `givens_once()`, `givens_complete()` takes not just a matrix A as its input but also a corresponding vector b. In our case, the vector b is a list of measurements $Z(t)$ taken at different instances of time t. According to the Exercise 8.1 in *Montebruck-Gill*, our vector b is -

$$b = \begin{bmatrix} 2.63 \\ 1.18 \\ 1.16 \\ 1.54 \\ 2.65 \\ 5.41 \\ 7.67 \end{bmatrix}$$

Using this vector b and Eq (5), we can calculate vector d as

$$d = \begin{bmatrix} 8.40576072692871 \\ 4.93399911241531 \\ 3.46317042549133 \end{bmatrix}$$

Finally, we can now solve for the constants required for curve fitting using the following expression

$$\begin{pmatrix} c_o \\ c_1 \\ c_2 \end{pmatrix} = R^{-1}d$$

$$\begin{pmatrix} c_o \\ c_1 \\ c_2 \end{pmatrix} = \begin{bmatrix} 2.6458 & 2.1355 & 2.4697 \\ 0 & 1.4049 & 2.372 \\ 0 & 0 & 0.6178 \end{bmatrix}^{-1} \begin{bmatrix} 8.40576072692871 \\ 4.93399911241531 \\ 3.46317042549133 \end{bmatrix}$$

$$\begin{pmatrix} c_o \\ c_1 \\ c_2 \end{pmatrix} = \begin{bmatrix} 2.748881 \\ -5.952499 \\ 5.605669 \end{bmatrix}$$

4 Conclusions

We can now use the constants calculated in the previous section to achieve the final form of the equation for our required curve.

$$Z(t) = 2.748881 - 5.952499t + 5.605669t^2 \tag{6}$$

4.1 Curve-Fitting

In Fig. 1, I plotted the curve of the final Quadratic Eq. 6 that we identified using the Givens Rotation and QR factorization along with the measured data points. The measured data points were well within the $\pm 2\sigma$ [1] error range. This means the our calculations and the code worked correctly for this problem.

[1] To calculate this error envelope, I took the difference measured $Z(t)$ and the $Z(t)$ that our fitted curve calculated for that instance t. Then, I square all these differences to get rid of the negative signs and took the square root of the average of the squares

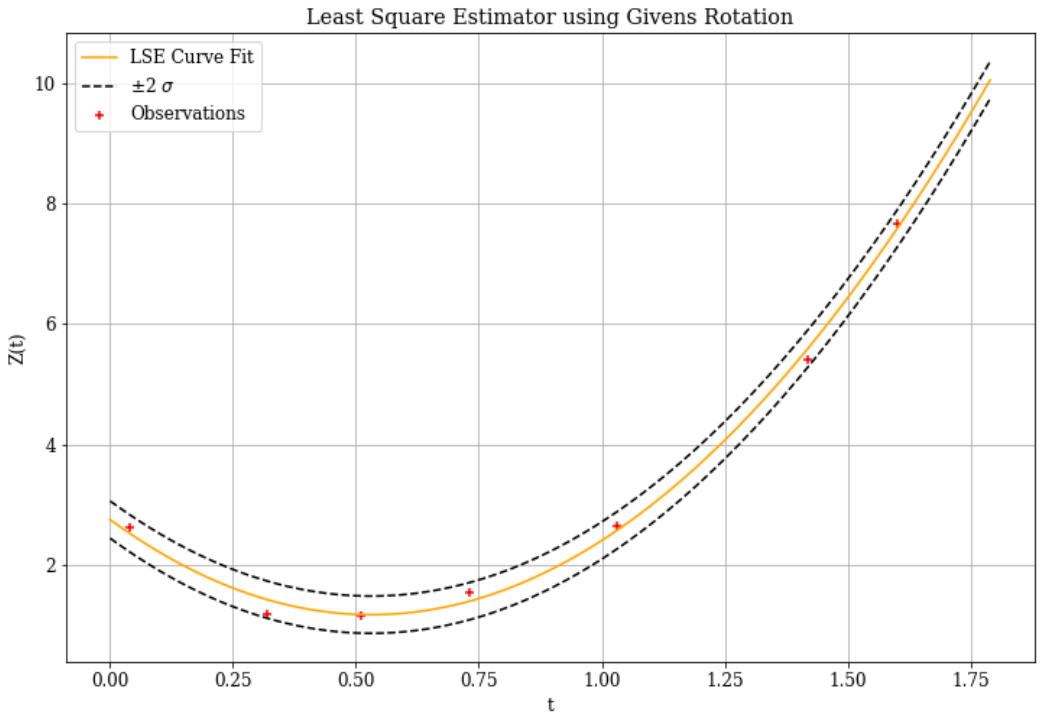

Figure 1: Measured Data points plotted along with the modelled equation and error envelope

4.2 Further Optimization

As I had hinted in the previous sections, the method that my code employs to solve this problem isn't the most computationally efficient one. The reason why Givens Rotation is a powerful tool is because it can be very easy for a computer to calculate. Since my computer is powerful enough that any further optimization of the code would be unnoticeable, I used the most basic algorithm to solve this problem. However, in situations where the volume of calculations are large or the memory/computational capability is low, there are ways to make this algorithm more efficient.

4.2.1 Memory Efficiency

My code maintains a list of all the results that the function `givens_once()` generates during the nested loop. There is no need for this. Further, since Givens Rotation transforms elements one row at a time, we can re-arrange the problem in a way where we use Givens Rotation to compute QR factors with only parts of the design matrix. In this way, at each stage of the sequential accumulation, we would not need the entirety of the design matrix.

4.2.2 Computational Efficiency

This is an even more advanced stage to optimize the performance of this algorithm. Multiple people have worked in order to develop methods that can perform QR factorization using givens rotation without having to use the square-root function during computation. However, the difference that it would make would not even be of the order of nanoseconds for our use.

References

[1] Montenbruck, Gill (2001) *Satellite Orbits- Models, Methods, and Applications*, Springer.