Design and implementation of an intelligent and adaptive mechanical mattress

Gonzalo Solera Pardo

Thesis director: Conrado Martínez

June 2021

Abstract

In this thesis, we present and implement a radical new concept of mattress, which is both intelligent and capable of mechanically adjust itself to comfortably accommodate any body shape and posture. It does so by controlling many linear actuators and by reading many pressure sensors to obtain updated information of the user's weight distribution and posture. The surface of the mattress is formed by the top side of many linear actuators placed vertically in a grid.

The project can be divided in 4 big components: mechanical, electronic, algorithmic and AI. These parts interact between them to obtain the desired result, but the specific implementation of them is fairly independent while maintaining a compatible interface. We give an explicit implementation of these components and show them in a working prototype.

In the presented implementation, many new and original ideas are introduced to obtain a scalable design that deals with trade-offs regarding cost, complexity, accuracy and agility, among others.

1 Introduction

1.1 Motivation

During the past 10 years, I have suffered a lot of back pain and discomfort while sleeping. Besides discussing these physical problems with doctors and specialists, I have tried many different mattresses with the hope of finding one that allowed me to sleep well and feel well rested in the morning.

I have not yet been able to completely fix this issue, although I have discovered that a good mattress choice was key in order to minimize my pain and discomfort. A good mattress adapts to the body shape and posture by providing a surface that keeps all the body parts in a healthy position. In particular, the backbone needs to be in a natural position, avoiding twists and unnatural curvatures, and the neck also needs to be parallel and aligned to the rest of the backbone, as shown in Figure 1:

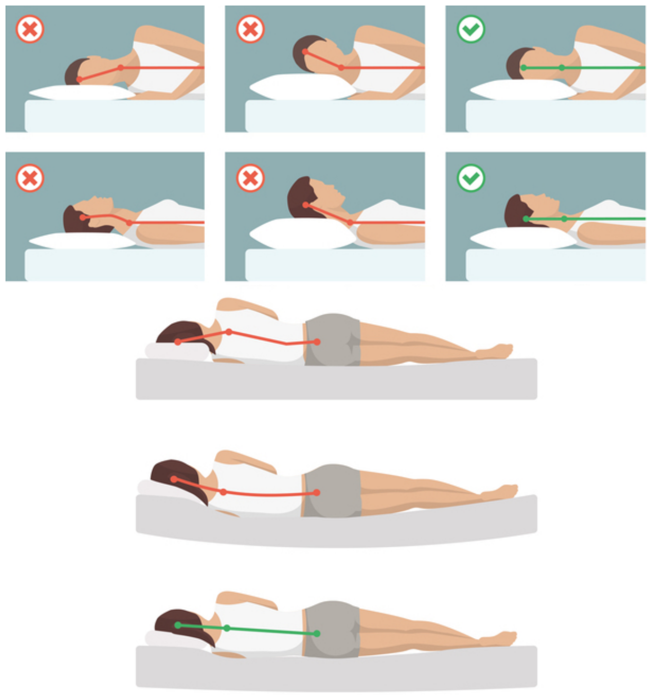


Figure 1: Appropriate back and neck posture while sleeping

However, the few mattresses that initially seemed to work properly for me rapidly stopped working and visible deformations appeared in their surface. My guess is that my weight (95 kg) and my unusual height (195 cm) play a big part in this problem. These deformations accentuated my pain issues and my discomfort. Also, even when there is no deformation present yet, I could not find a mattress that properly adapts to an arbitrary "basic posture". That is, a mattress that properly adapts to my back when facing upward, does not adapt to my body profile when lying sideways, and vice versa. In my case, I change a lot of posture during the night, so I need a mattress that adapts well to these different postures. Furthermore, I obviously need a big pillow when sleeping sideways but it becomes harmful when sleeping upward, and since I am not actively adding and removing the pillow during the night, there are some postures that hurt my body.

Furthermore, my partner Pilar Soriano also suffers from similar back pain problems and discomfort while sleeping, greatly influenced by my weight and the lack of independent support of the mattresses. She is a graduated industrial design engineer with wide knowledge and experience in the field of additive manufacturing and 3D design. Together, we devised and designed the idea that is presented in this thesis, that mainly tries to address the following issues from previous mattresses that we owned:

- Lack of adaptation to an arbitrary body shape and posture
- Performance degradation over time
- Lack of independent support (weight of a user disturbs another)

1.2 The idea

The design that we present is based in the following hypothesis: A person lying on top of of a "bed" of sand (at the beach) that perfectly mimics his body shape is very comfortable, since he can maintain his natural body posture and aligned backbone while distributing his weight in an uniform manner. i.e., close points sustain a similar pressure, and no small cluster of points sustain a specially large nor small pressure, compared to their neighbours.

The main idea of our design is to take this "continuous" sand surface and to "discretize" it by projecting it into a grid, as shown in Figure 2. Then, by actively controlling each one of these discretized parts we can approximate the "ideal" surface of any arbitrary shape/posture. Having control of each of these discretized parts is equivalent to being able to independently move them vertically to arbitrary positions, and that is the main property of our design.

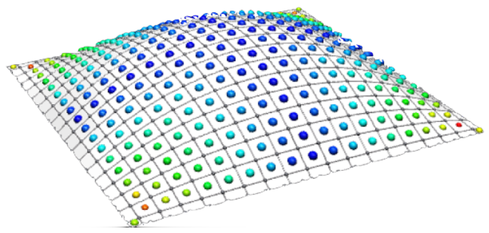


Figure 2: Example of a discretized surface

Any "regular" mattress also tries to follow the same approach of approximating the analogous bed of sand, that is, to accommodate to the user's shape and weight distribution, but they do not have an active control of their surface. Instead, they aim to passively approximate the "ideal" surface configuration using their materials properties (foam, latex, springs, etc), but as I already mentioned, I think they consistently fail to do so for some basic postures, at least for people with marginal physical properties like me (weight and height).

To summarize, our design mainly consists in providing a discretized surface that can be modified to approximate any arbitrary "ideal" surface. We call each discretized part a "lift", and we call their specific placement on their respective z-axis a "configuration", which tries to approximate some continuous surface. Since the discretization is obtained by projecting the surface into a grid, the lifts are arranged in a grid represented by a $n \times m$ matrix. We named our prototype Mattrexx for this reason.

As a side note, our design would remove the need of a pillow: The mattress would change its discretized surface to "create" a pillow by moving up the lifts below the head, until the neck is aligned to the rest of the body, as indicated in Figure 1.

It is left to specify:

- What is this configurable discretized surface physically made of?
  Mechanical component in section 2
- How is it controlled/modified to reach any arbitrary configuration?
  Algorithmic component in section 3
- How to obtain real-time information about the user's body shape and posture?
  Electronic component in section 4
- How to determine what is the most approximate configuration respect to the also unknown "ideal surface"?
  Artificial Intelligence component in section 5

1.3 State of the art

There already exist mattresses that actively change their shape. They are called alternating air pressure mattresses and are used in hospitals to help patients that can not move or that are confined to the bed for more than 15 hours per day. Their main goal is to prevent the generation of pressure ulcers, and advanced ones also can help the patient to change his posture.

As their name indicates, they work by inflating and deflating a mattress that consists in some air cylinders placed parallel to each other, as shown in Figure 3. They do not have individual control of each of these cylinders. Instead, they can only control the pressure (inflate/deflate) of all the cylinders at odd (or even) positions. Also, their surface discretization is different to ours, since there is only one cylinder per row (only one column). In conclusion, they offer a much more limited ability to adapt to arbitrary surfaces than our target design.

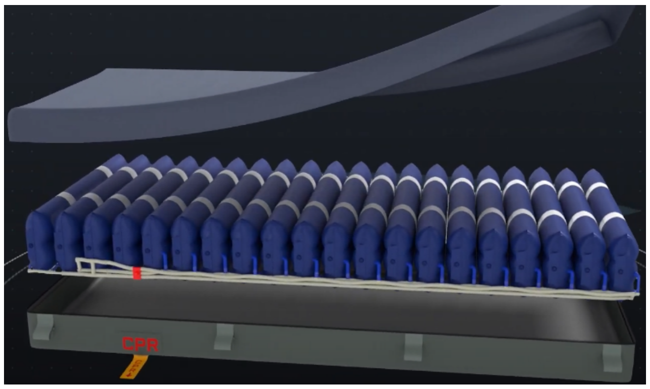


Figure 3: Alternating Air Pressure Mattress

1.4 Scope

Most of our efforts and ideas of this thesis revolve around the scalability of the design respect to the size of the mattress and the number of lifts, that is, the granularity of the discretization. Although this will be discussed deeply in the following sections, it is easy to see that our approach requires a large number of lifts that directly multiplies many variable costs (material, building time, etc). Hence, it is very important to find clever designs that minimize these variable costs. This means that the entire design phase has been focused in fixing scalability issues regarding cost, weight and manufacturing, and we had to ignore other secondary aspects like the noise that the mechanical component generates, at least in this first iteration of the design.

Furthermore, due to the size and complexity of the project, and due to our inexperience in the mechanical engineering field, many design choices were taken by intuitive reasoning rather than a rigorous theoretical analysis. That means that we do not provide a formal justification of why we chose some gear to have 12 teeth instead of 14. Instead, we made those decisions by intuitive and experimental procedures, since there already were many variables to consider and in this first iteration of the design we focused on more general and relevant variables. This implies that there is still much room for improvement, and that our design implementation is just a first approximation of an “optimal” one.

Finally, we want to mention that the initial goal of the project was to physically build a complete mattress following our design. Due to time constraints we have not been able to reach that milestone, although we have built a smaller prototype to demonstrate all the design ideas that we introduce. And although we have not built the entire mattress, all the design process was done considering a full-size mattress. This means that the raw materials and components used to build the small prototype would be the same as to the ones used for the full-size mattress. It would have been much easier to design or to directly buy existing pieces/components for the small prototype without taking into account the huge cost that they would take for a full-size version, that is, ignoring the scalability of the design. Instead, we spent a lot of time and effort thinking of scalable designs and searching for providers that allow the manufacture of a full-size mattress without incurring in costs of tens of thousands of euros. This is explained with more detail in the mechanical and electronic sections.

2 Mechanical component

Depending on the size of the mattress and the granularity of the discretization, the number of lifts will be larger or smaller. A finer discretization will provide a better ability to approximate arbitrary surfaces but at the expense of having a larger number of lifts. They result in more pieces which implies a higher cost, complexity, weight, manufacture time, etc... However, a smaller number of lifts will not approximate arbitrary surfaces properly, although the mattress will be cheaper, simpler and lighter.

The first step is, considering the previous trade-offs, to choose the size s (width and length) of the squared surface that each lift will hold. This determines the density of lifts, or equivalently, the granularity of the discretization. Then, the length of the mattress will be given by $n \cdot s$ and the height by $m \cdot s$ for some arbitrary naturals n and m. We have decided to use $s = 5$cm, since we believe that this size offers a good trade-off of the qualities mentioned above. This means that for a one-person mattress of 200cm length and 70cm width, the total number of lifts would be 560, arranged in a 40×14 grid, each one of them covering a square of 5cm $\times$ 5cm. Furthermore, the vertical travel-length of these lifts needs to be fixed, and we have decided it to be 10cm, which we believe is enough to adapt to the surface of a person in common postures. These are the magnitudes of reference in our design.

2.1 Lift design

So far, we have not defined how exactly these lifts are, but we specified their mission, which is to move a squared surface up and down, while holding a potentially large weight.

A linear actuator is a mechanical device that transforms rotatory movement into linear movement. The most common and basic form of linear actuator, as shown in Figure 4, consists in a leadscrew (threaded rod) that rotates on its own axis and a nut whose rotation is fixed. This forces the nut to move along the leadscrew's axis of rotation, and its direction depends on the direction of such rotation.

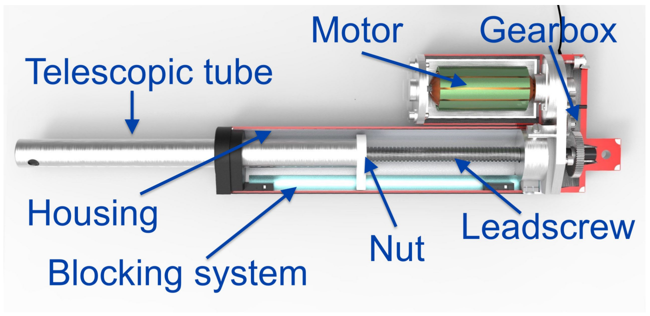


Figure 4: Main components of a linear actuator

A linear actuator can act as a lift, and this is what we use in our design. However, commercially available linear actuators are too expensive for us to use in our design, since we need $n \cdot m$ many. For example, a cheap linear actuator that would work for our use case costs around 25€. That would imply spending $560 \cdot 25 = 14000$€, just on linear actuators.

The main causes of that prohibitively cost are two:

- Many components are replicated for each lift, with no room of sharing resources and thus minimizing the number of components and price.
- Linear actuators are very precise (with precision varying with quality and price), and are able to move the nut sub-millimetric amounts. For our use case, such precision is not really needed since a person won't notice that some lift is off from its target position by 1 millimeter.

There is also another problem with using commercial linear actuators, which is related to how to control them. A commercial linear actuator usually has its own motor with two wires (positive and negative polarities), and one can control it by supplying a voltage to them. However, this would imply a direct connection from some microcontroller to every lift. Having $O(nm)$ wires connecting lifts seemed infeasible for us, both for a prototype and for a real product.

Our idea to solve these previous problems is borrowed from computer engineering, and that is multiplexing.

More specifically, the idea consists in having a single, big and powerful motor shared by all the lifts, and whose traction is transferred to all the lifts by mechanically multiplexing it. A deeper explanation is given in the following subsection, but in relation to the design of the lifts, this means that we don't need 560 motors and 560 gearboxes and thus the price is potentially decreased a lot.

We still have to provide a specific design of the rest of the linear actuator used as a lift. The distinct components that constitute the linear actuators are:

- A leadscrew.
- A nut.
- A system to block the rotation of the nut while allowing it to move freely along the leadscrew's axis.
- The housing of the linear actuator.
- The telescopic structure (tube) that retracts/extends when the linear actuator works, which is moved by the nut.

Our first approach consisted in using a housing and a telescopic tube with a non-circular profile, as for example a squared/rectangular profile, as shown in Figure 5. This way, the housing would not only keep the telescopic tube straight fighting lateral forces (as any housing does), but it would also serve as a blocking mechanism for the nut, since the telescopic tube to which the nut is attached could not rotate inside the also squared housing.

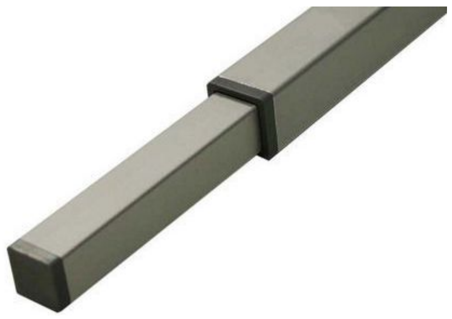


Figure 5: Telescopic squared tube

In our design process we constantly faced a day-to-day reality of an industrial engineer: We can not simply design an arbitrary 3D piece if there is no practical way of manufacture it or build it with commercially available components. Also, the cost of the design will depend greatly in both the manufacturing process and the materials/components used. In this case, we would want the external tube (housing) to be exactly 5cm per external side, while the internal tube's external side to be almost the internal side of the external tube, such that it is well-aligned and also offers freedom of vertical movement. Furthermore, we need these tubes to be made of a light, rigid and cheap material, such as plastic. Unfortunately, we could not find any provider that offered two squared telescopic tubes of such "compatible" dimensions and materials.

However, there are many commercially available circular PVC tubes used mostly for water delivery systems that have "compatible" diameters. That is, we can use two tubes of distinct diameters such that one can be inserted into the other in a telescopic manner. And that is what we have used to build our lifts, as shown in Figure 6.

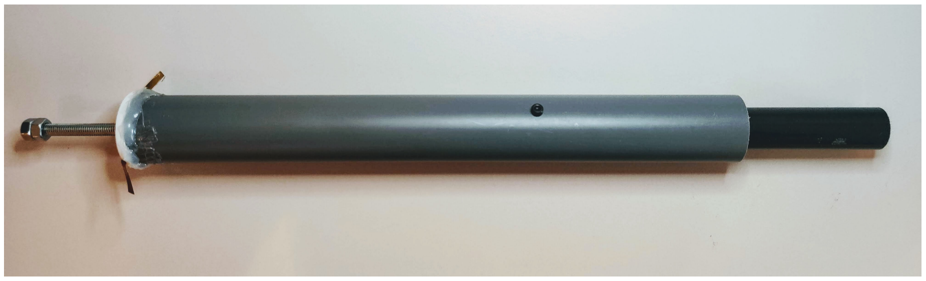
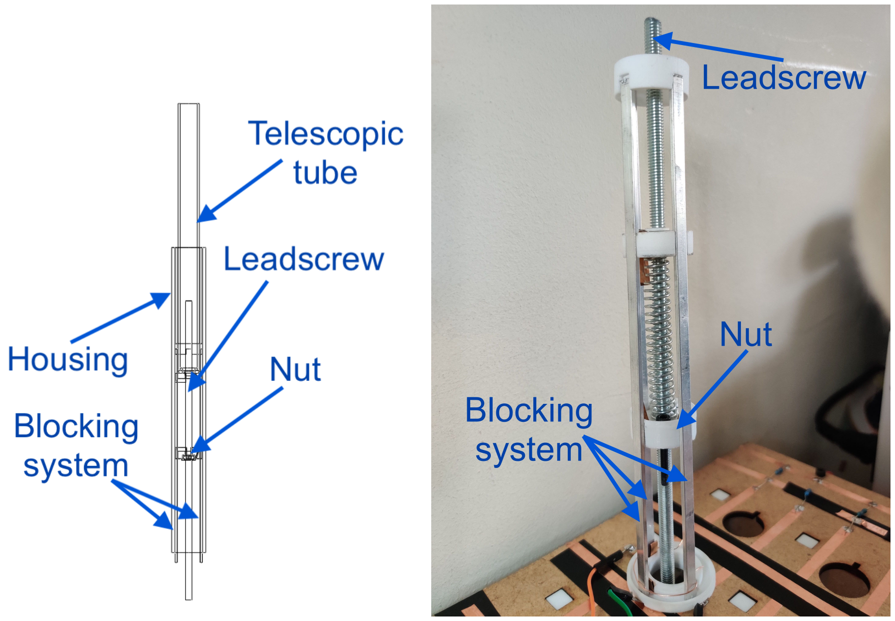


Figure 6: Lift

The only problem is that the nut's rotation is now not blocked by the housing. The solution that we applied consists in using a rail that is fixed across the housing to which the nut is also attached, allowing its free vertical movement but blocking its rotation, as shown in Figure 7:



(a) Blueprint (b) Lift inner parts

Figure 7: Lift design

The use of a rail is a common way of fixing the rotation of the nut, but in our case we use three. They are also used to conduct electricity for the weight sensors, and that will be covered in depth in section 4.

In conclusion, we have used very simple and cheap materials to design the linear actuators, shown in Figure 8. These materials are:

- PVC tube of Ø32mm and 28.5cm length and PVC tube of Ø26mm and 9cm length as external housing.
- PVC tube of Ø20mm and 25cm length as telescopic tube.
- Metric steel threaded rod of Ø6mm and 25cm length.
- Metric nut of Ø6mm.
- 3 aluminum rods of rectangular profile of 5mm $\times$ 2mm and 20cm length.
- Other parts relevant to the weight sensor.

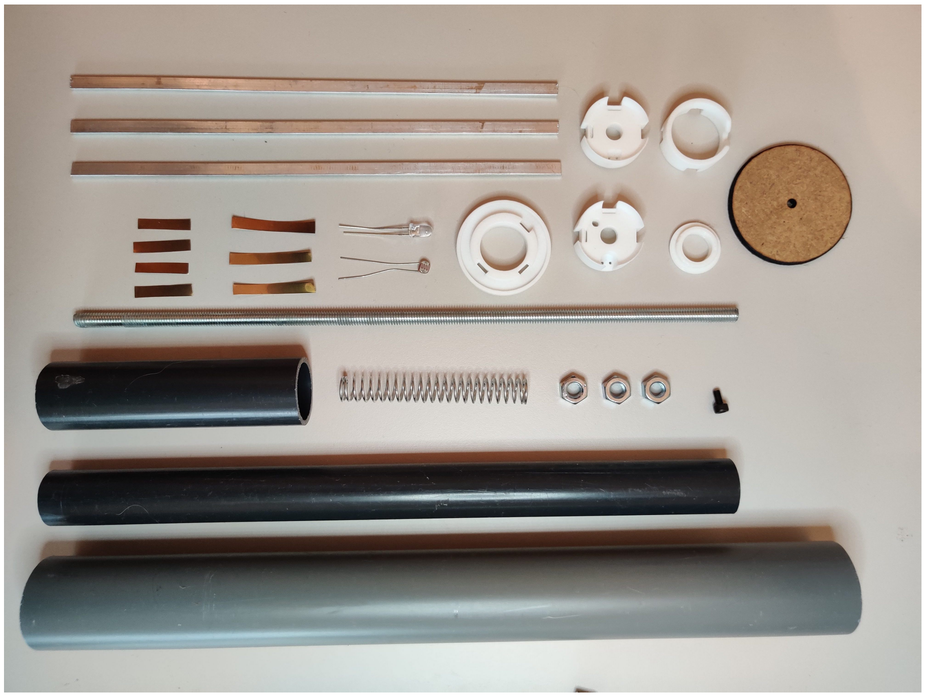


Figure 8: Lift parts

With this design, including the weight sensor, each lift costs less than 1.6€, instead of the 25€ of a commercial linear actuator. In a production environment, where the lifts are produced at a large-scale, the cost would be much smaller.

Regarding the weight that each lift needs to hold, we used 100kg as a reference of the worst case (when a person is only held by a single lift), and around 2kg on average when the person is lying down. In general, depending on the leadscrew and the nut properties, the load that makes the linear actuator to backdrive will vary. When a linear actuator backdrives, the nut is moved because of the load, instead of because of the traction of the gear as it is supposed to. This can be very harmful, both to the rest of gears and to the functionality of the design. The properties of our leadscrew and nut guarantees that it can hold much more than 100kg without backdriving.

However, the load of the lift is transmitted from the telescopic tube to the nut, and from the nut to the leadscrew. This is why we have added a very cheap thrust bearing below each leadscrew, as shown in Figure 9.

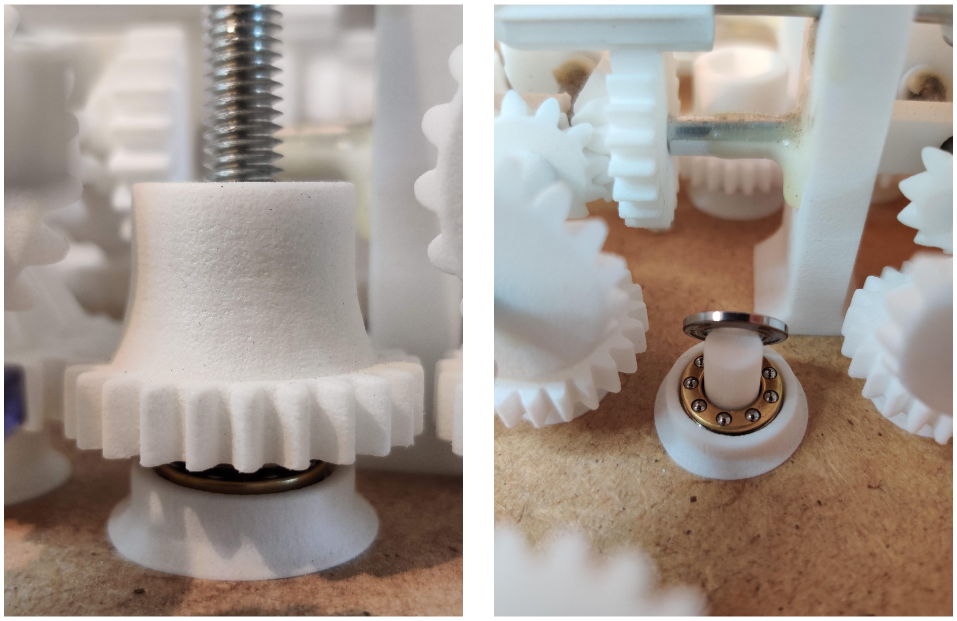

(a) Lift's gear and thrust bearing (b) Thrust bearing

Figure 9: Leadscrew's thrust bearing

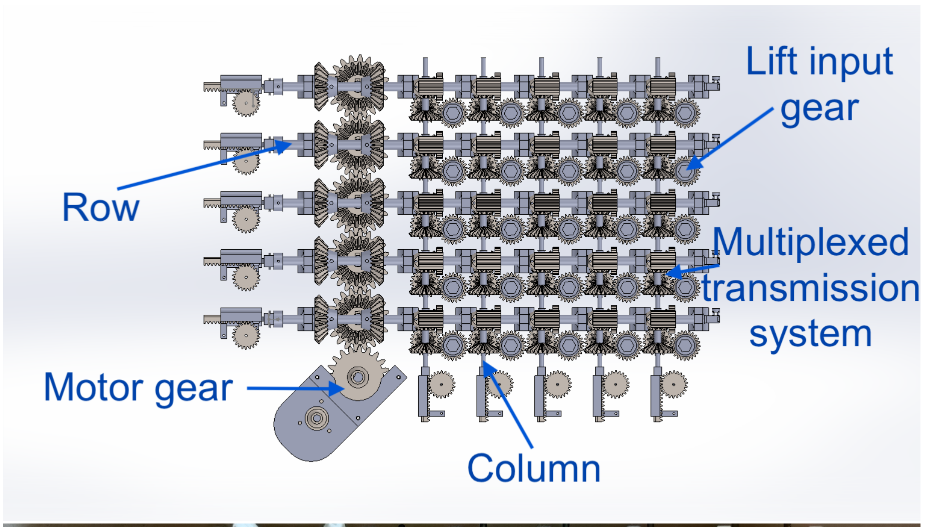
Note that the housing and other elements do not experience this vertical load. However, they might experience a small lateral loads, which are dissipated using a foam layer between the lifts.

2.2 Multiplexed drivetrain

As already mentioned, the solution regarding the scalability of the design consists in multiplexing the traction to the lifts since there are too many of them

($O(nm)$ in general or 560 for our target mattress size). Recall that the justification of the multiplexation is the ability to share a single powerful motor and gearbox among all the lifts, instead of having 560 motors and gearboxes which increases a lot the price and weight. Also, it is infeasible to have a direct connection to that many linear actuators, since there would need to be many wires, motor drivers, microcontrollers, etc.

Hence, we had to design from scratch a system to multiplex the traction to the lifts, in such a way that to move an arbitrary lift of the grid up or down, we have to "activate" its row and its column. In our final design, shown in Figure 10, we have a single big and powerful motor that moves a sequence of large gears, one for each row. Then, for each row there is a small linear actuator built with a servo motor that can engage or disengage its row shaft. This row shaft, when engaged to the drivetrain, starts to rotate clockwise or counterclockwise depending on how the engagement is, since there are two bevel gears on a row that can engage to the drivetrain and each one of them will make the shaft rotate to one or the other direction. Then, a row that is engaged to the drivetrain will make its shaft to rotate. This alone does not move any lift, since a column still needs to also be engaged for that to happen. Another linear actuator can engage a column to the drivetrain, and in all the intersections of the column with an engaged row, the traction will be transferred to the appropriate lifts. It is difficult to describe our design, but the reader can see the Figure 11 to understand how it works:



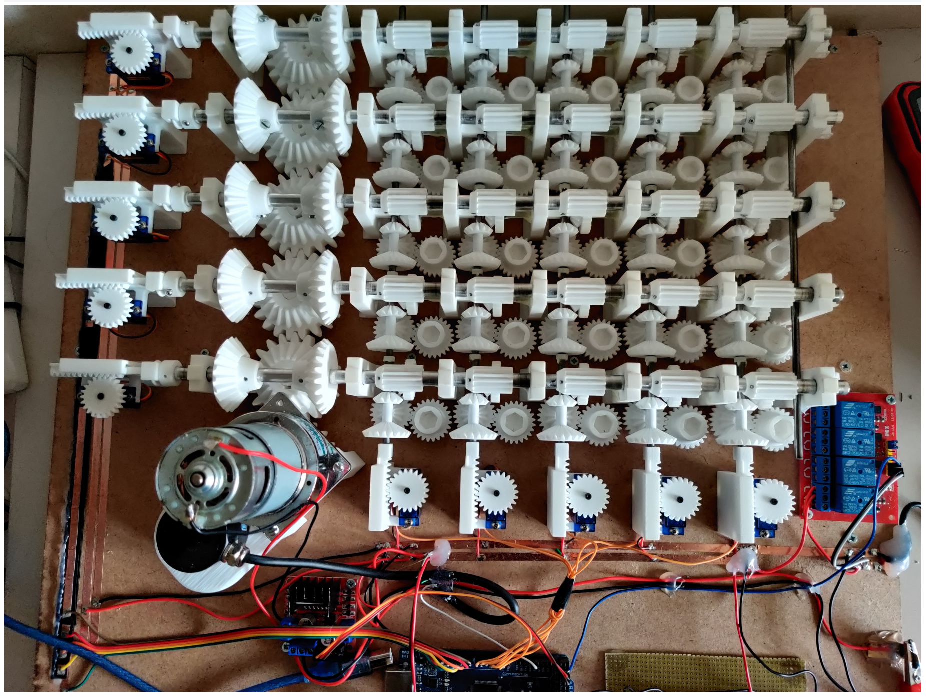


Figure 10: Mechanically multiplexed traction system

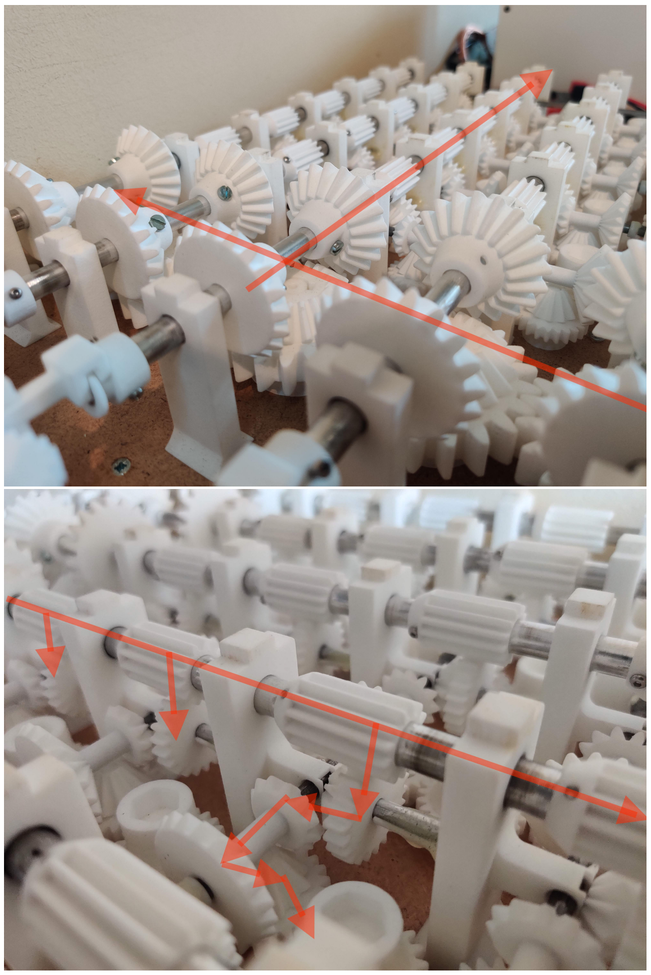


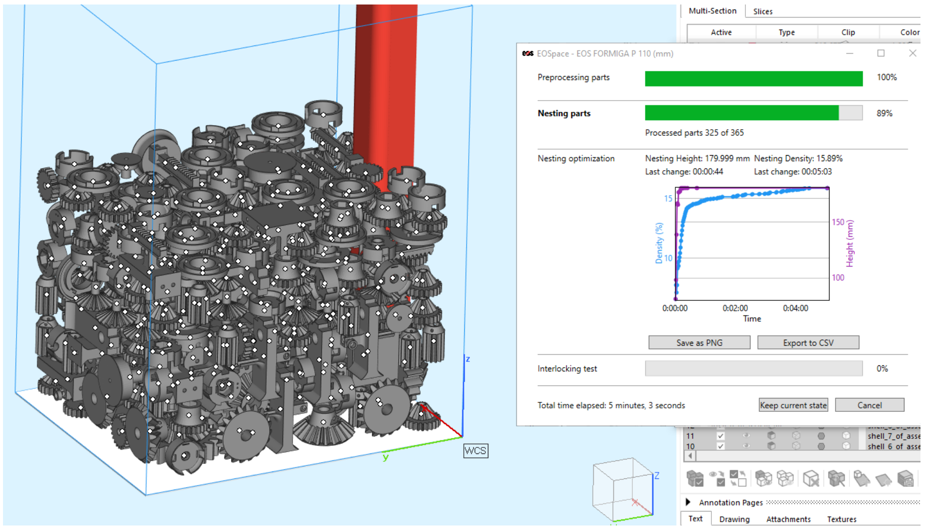
Figure 11: Path of the traction from the motor to a lift

In conclusion, a lift that is placed in the i^{th} row and j^{th} column will move if and only if the row i and column j are both engaged. The direction of the movement (up or down) will depend on the row i. This way, we only need a single powerful motor and $O(n + m)$ commercial linear actuators instead of $O(nm)$ commercial and expensive linear actuators. In our case of $n = 40$ and $m = 14$, we only need 64 commercial linear actuators that also do not need to be powerful, so they can be very cheap as well. We built them with cheap SG90 servos that cost less than 2€ each.

However, one of the problems that multiplexing has is the inability of controlling any arbitrary set of lifts at the same time to arbitrary directions. i.e., when moving two lifts from different rows and columns (i, j and i', j'), then two other lifts will inevitably be moved as well (i, j' and i', j). This issue is dealt algorithmically in section 3.

2.3 Fabrication

Besides the already mentioned components used to build the lifts, we also needed many other parts to build the prototype. Specially, we needed to physically have, in great quantities, the pieces that Pilar Soriano implemented in SolidWorks. We were able to iterate and to validate the design by 3D printing the pieces and seeing how they worked together, thanks to a FDM 3D printer that we own. However, this method does not scale for building a large prototype, since FDM printers take a lot of time to print. Also, the printed pieces have many irregularities due to the nature of FDM printers, and that affected the performance of the gears. This is why, for the final prototype, we purchased a service from a company to print our pieces using a SLS 3D printer, which uses a laser to print and is much more precise. The downside is that this is very expensive, and you pay for the volume that the 3D pieces occupy. On the other hand, you are responsible of placing them in a way such that the total volume is minimized, and we also spent some time using a placement algorithm to place the pieces in the allowed space while satisfying some other fabrication constraints, as shown in Figure 12:


EOSpace - EOS FORMIGA P 110 (mm)

Preprocessing parts 100%

Nesting parts 89%

Processed parts 325 of 365

Nesting optimization

Nesting Height: 179.999 mm Nesting Density: 15.89%

Last change: 00:00:44 Last change: 00:05:03

Interlocking test 0%

Total time elapsed: 5 minutes, 3 seconds

Figure 12: Placement of 3D pieces to print with SLS

The cost of 3D printing all the pieces needed for the 5×5 prototype was 450€. This is a lot, considering that a full-scale mattress of 560 lifts has around 20 times more pieces. This fabrication technique is specially useful for fast-prototyping, since it offers a lot of flexibility to experiment without having to pay much greater fixed-costs for other fabrication techniques that are also slower. In a large-scale production environment, these same pieces would be fabricated with injection techniques, which would decrease the variable-costs a lot. However, the fixed-costs (related to creating the injection molds) are too large to build the prototype with them.

The described design also takes into account the building process, since there are 560 lifts and thus, many components to build. That means that we have discarded many other alternative designs in order to ease the building process of the prototype.

We have built the prototype in a local and collaborative studio named "MadeBCN" that had many tools available for its members, such as laser cutting machines, saws, etc. We had to spend several hours working and learning how to use the machinery to build the prototype. We also had to spend a lot of time designing things to facilitate the fabrication. For example, we built the simple system shown in Figure 13 that allowed us to cut the PVC tubes of the lifts very fast, since for a 40×14 mattress, there would be 1700 cuts and the precision of them is important as well.

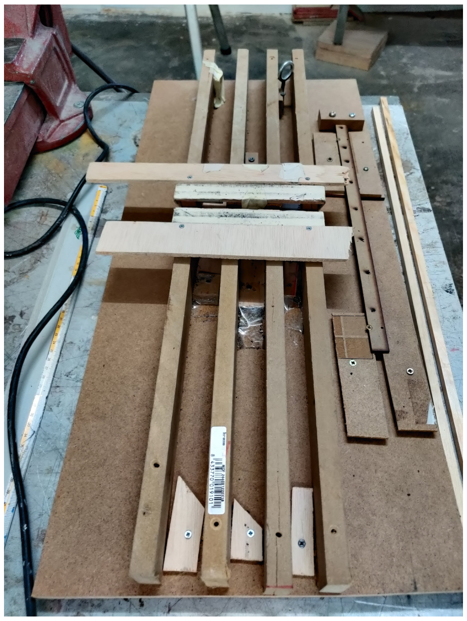


Figure 13: System to cut tubes fast

Another tool that was fundamental for the prototype construction was the laser cutting machine, since that allowed us to obtain a platform with holes to which screw the 3D printer parts, obtaining a correct and precise alignment for the shafts of the prototype. It is very important for the shafts and axles to be perfectly aligned so the friction forces are minimized.

2.4 Final prototype

In Figure 14 we show a picture of the final prototype, and in Figure 15 we show the lifts at their origin and at a some target lift configuration.

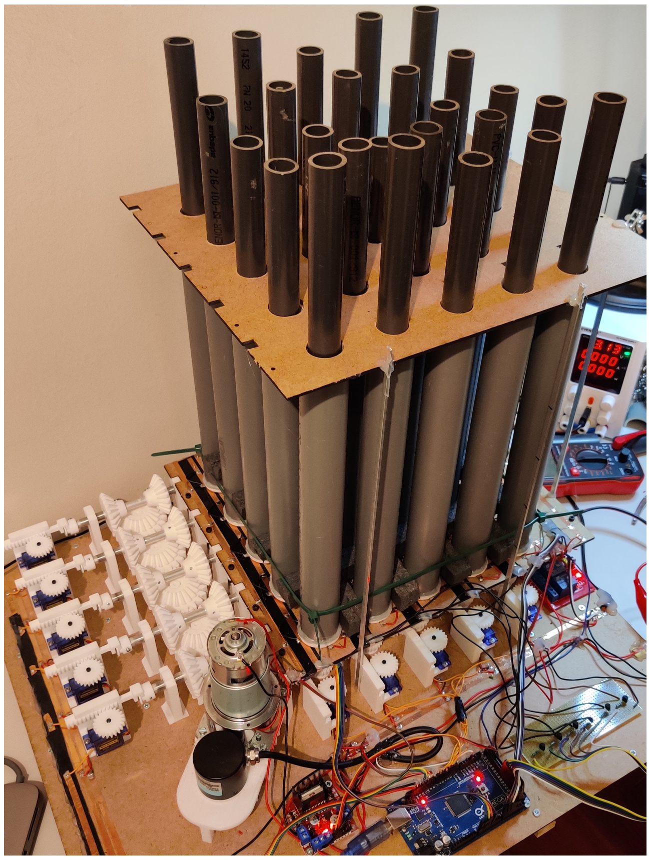
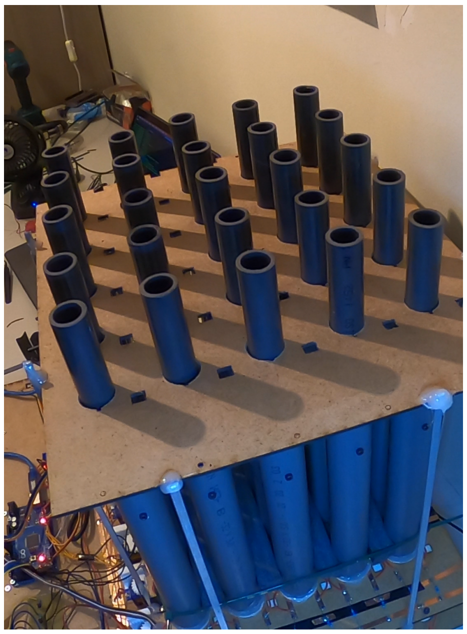


Figure 14: Final prototype


(a)

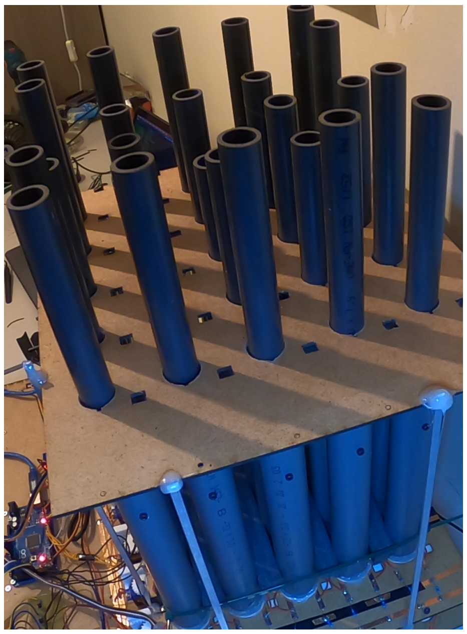

(b)

Figure 15: Two distinct lift configurations

3 Algorithmic component

As already mentioned, this section algorithmically deals with the inherent problem of mechanically multiplexing the movement of the lifts: Not any arbitrary set of lifts can be moved exclusively and concurrently.

3.1 Problem formulation

Let $l_{i,j}$ denote the lift in the i^{th} row and j^{th} column. Its position $C_{i,j}$ is denoted by the current configuration $n \times m$ matrix C.

The aforementioned mechanical multiplexing allows us to determine which lifts to move by deciding which rows and columns to engage to the drivetrain. We encode the engagement of the rows and columns as follows:

- For any row i, the variable $r_i \in \{-1, 0, 1\}$ determines whether the row should be disengaged ($r_i = 0$), engaged with clockwise rotation ($r_i = 1$) or engaged with counter-clockwise rotation ($r_i = -1$).
- For any column j, the variable $c_j \in \{0, 1\}$ determines whether the column should be disengaged ($c_j = 0$) or engaged ($c_j = 1$).

For ease of notation, we will also refer to the vectors $r \in \{-1, 0, 1\}^n$ and $c \in \{0, 1\}^m$ defined as:

$$r = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} \qquad\qquad c = \begin{bmatrix} c_1 & c_2 & \dots & c_m \end{bmatrix}$$

Then, following our multiplexation of the lift movements, the lift $l_{i,j}$ will be moved if and only if $r_i \neq 0$ and $c_j \neq 0$ (i.e., when $r_i c_j \neq 0$). The direction of the movement will be encoded in the variable r_i. More generally, the movement of all the lifts are described by the $n \times m$ matrix Δ defined as:

$$\Delta = rc$$

As already indicated, with this multiplexed setup we are not able to independently and concurrently modify the position of the lifts. i.e., We can move any lift $l_{i,j}$ by engaging its row and column, but we can not move $l_{i,j}$ and $l_{i',j'}$ where $i \neq i'$ and $j \neq j'$ without inevitably be moving $l_{i,j'}$ and $l_{i',j}$ as well. We want to obtain an algorithm that takes into account these dependencies in order to find a sequence of row and column engagements/disengagements $((r^1, c^1), \dots, (r^T, c^T))$ such that the lifts reach the position described by a target configuration C^*, starting from an initial configuration C, while minimizing the time T (denoted as T^*).

Such a sequence of lift movements $(\Delta^1, \Delta^2, \ldots, \Delta^T)$ assumes that each Δ^t is applied during a fixed time interval, equal to the time that it takes to rotate 10 times a lift's leadscrew. This is a first step to simplify the problem, which consists in discretizing the vertical movement of the lifts: A lift moves vertically 1mm per revolution (which is determined by the lead of the leadscrew), and we decided to fix 1cm as the discrete unit that a lift should move. Hence, each time a lift moves one unit up or down, we will rotate its leadscrew 10 times. Since the total travel distance is already fixed as 10cm (by the mechanic components), the z-position of a lift after moving it will be a natural number between 0cm and 10cm respect to its origin. In other words, $C_{i,j} \in \{0, 1, \ldots Z\}$, where $Z = 10$.

We define the (sub)configuration C^t as the lift positions after applying $\Delta^1, \ldots, \Delta^t$ to C. More formally:

$$C^t = C + \sum_{t'=1}^{t} \Delta^{t'} = C^{t-1} + \Delta^t$$

Since by definition:

$$C^* = C^T = C + \sum_{t=1}^{T} \Delta^t$$

We obtain that:

$$\sum_{t=1}^{T} \Delta^t = C^* - C$$

This means that we can ignore the particular initial positions of the lifts C and target positions C^* and just focus on decomposing the matrix $\Delta^* = C^* - C$ by a sum of a deltas $(\Delta^1, \ldots, \Delta^T)$, where each Δ^t is obtained by some r^t and c^t.

We will restrict ourselves with Δ-sequences such that:

$$\Delta^t_{i,j} \in \{0, \operatorname{sign}(\Delta^*_{i,j})\}$$

That is, we force that the distance between the position of any lift respect to the target configuration decreases monotonically. More formally: $|C^T_{i,j} - C^{t-1}_{i,j}| \geq |C^T_{i,j} - C^t_{i,j}|$. This restriction might increase the optimal time T^*, but is justified by two main reasons:

- The problem is further simplified.
- For a user lying on the mattress, it might be uncomfortable to have a lift moving up when it should be moving down, potentially increasing a lot the pressure of that lift (momentarily).

Without this last constraint, our problem could also be nicely described as to find an $n \times k$ matrix A and an $k \times m$ matrix B of minimal k such that $\Delta^* = A \cdot B$ and where $A_{i,j} \in \{-1, 0, 1\}$ and where $B_{i,j} \in \{0, 1\}$.

3.2 Naive algorithms

Note that by only engaging a single row and a single column at any given time (an thus only moving one lift at a time), we would remove the undesired interferences with other lifts. However, this limits our ability to reach a desired configuration of positions fast, since we could not parallelize lift movements. Another trivial approach that is incrementally better consists in modifying the position of the lifts row by row, or column by column. This approach is valid since modifying only lifts of the same row (or column) will never involuntarily move lifts that should not move.

However, these simple strategies might be far away from an optimal one. In the next subsection we review related problems to asses the hardness of finding a Δ-sequence of optimal length T^*.

3.3 Related problems

There are some related problems covered by the existing literature. None of them is exactly the problem that we are tackling here, but they are similar and we are going to use them to asses the complexity of it and to borrow some definitions from them:

- A rectangle R of an $n \times m$ matrix M is the set $R = I \times J \subseteq [1, n] \times [1, m]$.
- A 1-rectangle of M is a rectangle R of M such that $\forall (i, j) \in R,\ M_{i,j} \neq 0$.

Note that the each Δ^t is very similar to a 1-rectangle. More precisely, Δ^t can be covered by the single 1-rectangle $R^t = \{i \mid r_i^t \neq 0\} \times \{j \mid c_j^t = 1\}$, and that R^t is also a 1-rectangle of $C^* - C$ and specially, of $C^* - C^{t-1}$.

3.3.1 Maximum Edge Biclique Problem

Consider an $n \times m$ matrix M. We can interpret M as the adjacency matrix of some bipartite graph $G = ((A, B), E)$, where each row i corresponds to the vertex $i \in A$ and each column j corresponds to the vertex $j \in B$, and where $M_{i,j} \neq 0$ if and only if $(i, j) \in E$.

A biclique of a bipartite graph G is a complete bipartite subgraph of G. i.e., A biclique is represented by a 2-tuple consisting in the subsets $I \subseteq A$ and $J \subseteq B$, such that $I \times J \subseteq E$.

Note that there exists a direct bijection between 1-rectangles of M and bicliques of G: Consider the 1-rectangle $R = (I, J)$ of M. Since R is a 1-rectangle, then $M_{i,j} \neq 0$ (for $i \in I$ and $j \in J$), which implies that the edge $(i, j) \in E$. Hence, by definition (I, J) is a biclique. The inverse is also trivial.

The Maximum Edge Biclique Problem [5] consists in finding a biclique of G with maximum number of edges. That is, to find a biclique (I, J) where $|I| \cdot |J|$

is maximized. This problem (the decisional version) is NP-complete (in contrast to the Maximum Vertex Biclique that maximises $|I|+|J|$, which is in P). Hence, finding a largest 1-rectangle of a binary matrix is NP-hard.

This problem is related to ours, since a natural greedy algorithm would consist in selecting the largest Δ^t of $C^* - C^{t-1}$. However, such task is NP-hard since the problem of finding the largest 1-rectangle can be reduced into it. This algorithm will be discussed in subsection 3.4.1.

3.3.2 Rectangle Partitioning Number

The Rectangle Covering Number [6] (RCN) of an $n \times m$ binary matrix M is the smallest number of 1-rectangles that are needed to cover all the non-zero cells of M, and it is denoted as RCN(M). It is equivalent to the Boolean Rank [9], which is also equivalent to many other problems such as Biclique Edge Covering of bipartite graphs [2, 3]. These problems are NP-hard [7], and they are related to our problem at hand since they ultimately consist in decomposing a binary matrix over boolean algebra [8]. In our case, Δ^* is an integer matrix and we use elementary algebra and further constraints.

There exists a more restricted version of the RCN problem namely Rectangle Partitioning Number (RPN), that as its name suggests, only considers 1-rectangle covers of M without overlaps. This problem is more commonly known as the Binary Rank Over Elementary Algebra, which is also equivalent to the Biclique Edge Partition problem of bipartite graphs [4].

Note that clearly RCN(Δ^*) $\leq T^*$ and RCN(Δ^*) $\leq$ RPN(Δ^*), where we might interpret Δ^* as a binary matrix depending on whether a cell is zero or non-zero. A good example that compares these problems is as follows:

$$\Delta^* = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

RCN(Δ^*) $= 2$ but RPN(Δ^*) $= T^* = 3$. i.e., We can trivially cover Δ^* by two 1-rectangles, but they overlap and such Δ-sequence would not correctly decompose Δ^*. Note that if in this example $\Delta^*_{1,1}$ were 2, then T^* would become 2 because the 1-rectangles could overlap once in $\Delta^*_{1,1}$. However, RPN(Δ^*) would remain being 3.

In general, the RPN problem is not the same as ours unless Δ^* is binary. Hence, we can directly reduce the RPN problem into ours by setting $\Delta^* = M$ and returning T^*. RPN is also NP-hard [4], and since RPN can be reduced to our problem, ours must be NP-hard too.

3.4 Algorithms

By the previous subsection we know that our problem, which can be summarized as to decompose the matrix $\Delta^* = C^* - C$ into a Δ-sequence, is NP-hard.

We deal with this by providing 5 algorithms with different trade-offs and strategies. The first 4, which are greedy algorithms, try to iterativelly find a large Δ^t, measured by the number of lifts that it moves: $|\{l_{i,j} | \Delta^t_{i,j} \neq 0\}|$. This is an heuristic with which we hope to obtain short Δ-sequence, but as we will show, this strategy does not necessarily yield the optimum T^*. Other less-straightforward heuristics have been tried too but they are not included in this thesis. These greedy algorithms differ between them in how to find such Δ^t, since to find the largest one is NP-hard. Our fifth algorithm consists in a reduction into SAT that is able to obtain the optimal T^*.

Note that we can arbitrarily permute any Δ-sequence without violating any restriction nor changing its length T. Knowing this, we always sort the found Δ-sequence in decreasing order of sizes of the Δ^t. The motivation behind is that although the total time to reach the desired configuration does not change, we would like to make as many lift movements as soon as possible, so the user can start being comfortably in his new posture sooner.

Since the greedy algorithms find a Δ^t independently of the past and future $\Delta^{t'}$, we will use Δ^* to refer to $\text{sign}(C^* - C^{t-1})$ when describing the algorithms to ease the notation. That is, the greedy algorithms only focus in finding a large Δ^t to decompose $\text{sign}(C^* - C^{t-1})$, and they do not care about the remaining lift movements $(C^* - C^{t-1})_{i,j}$.

3.4.1 **GreedyLargestRectangle** or GLR

As mentioned before, a natural approach to tackle this problem would be to iteratively and greedily find the largest set of the remaining lifts to move that can be moved together. As previously shown, to find such largest Δ^t is itself NP-hard. Hence, this greedy algorithm would have exponential running time in the worst case. However, for the mattress dimensions that we deal with (40×14 for the full mattress and 5×5 for our prototype), it is possible that a good implementation that prunes the exploration soon might give good practical results in acceptable time.

More in detail, our implementation considers every possible subset of rows $I \subseteq \{1, \dots, n\}$ and direction $d_i \in \{-1, 1\}$ for each $i \in I$. For each of such subset, it finds the largest subset of columns J such that:

$$r_i^t = \begin{cases} d_i, & \text{if } i \in I \\ 0, & \text{otherwise} \end{cases}$$

$$c_j^t = \begin{cases} 1, & \text{if } j \in J \\ 0, & \text{otherwise} \end{cases}$$

is a valid Δ^t for Δ^* $(= \text{sign}(C^* - C^t))$.

To find such largest subset of columns is trivially done in $O(nm)$ time, since it consists in the columns $\{j \in \{1, \dots, m\} \mid \forall_{i \in I} \; \Delta^*_{i,j} = d_i\}$. This step can be implemented in $O(m)$ by reusing computations and we refer the reader to our code to view the details.

The exploration consists in a tree that starts with $I = \{i\}$ for each i and considers adding a new row $i' > i$ with some direction $d_{i'}$ at each level of the tree. It prunes a branch when $|J| = 0$ or when $(|I| + (n - i' + 1)) \cdot |J|$ is less or equal than the largest Δ^t found so far.

This algorithm, although has a running time of $O(2^n m^2)$, does not necessarily reach an optimum Δ-sequence of length T^*, as shown by the following counter-example that compares the Δ-sequence obtained by our GLR algorithm respect to an optimal Δ-sequence:

$$\Delta^* = \begin{bmatrix} 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad \Delta^1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Delta^* - \Delta^1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad \Delta^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\Delta^* - \Delta^1 - \Delta^2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \Delta^3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

An optimal Δ-sequence would instead be:

$$\Delta^* = \begin{bmatrix} 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad \Delta^1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Delta^* - \Delta^1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad \Delta^2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

3.4.2 GreedySweepingSubset or GSS

The goal of this algorithm is to be very fast instead of priorizing finding large Δ^t (unlike GLR that finds the largest possible). In fact, this algorithm basically is the naive algorithm briefly mentioned above that moves rows/columns independently one at a time. However this algorithm tries to find other "compatible" rows/columns that can also be moved without modifying the movements of the initial row/column.

More formally, for each non-empty row i and direction $d_i \in \{-1, 1\}$, it finds all the other rows i' such that $\forall_{\{j|\Delta^*_{i,j}=d_i\}}\ \Delta^*_{i',j} = d_{i'}$ for some $d_{i'}$. In other words, it assumes that all the lifts from row i and direction d_i are going to be moved, that is, all the lifts from row i and columns $J = \{j \mid \Delta^*_{i,j} = d_i\}$. Then it tries to see if there are other rows whose column set J' is a superset of J. These other rows can be engaged at the same time without affecting the already chosen lifts from row i and without moving any undesired lift.

This “row-wise” procedure is repeated for every initial row i and direction d_i. Then, a similar “column-wise” procedure is applied too. Δ^t is chosen to be the largest one found.

Note that if the column set J of row i and direction d_i and the column set J' of row i' and direction $d_{i'}$ satisfy $J \supseteq J'$, then the considered Δ^t when starting with i' would engage rows i and i' with columns $J \cap J'$.

However, if $J \not\supseteq J'$ and $J \not\subseteq J'$, then no considered Δ^t would engage both rows i and i', and this is the price that we pay with this fast algorithm, since we might miss a large Δ^t that engages these two rows and some subset of columns from $J \cap J'$.

This is why this algorithm takes its name, because we only consider adding rows whose column set is a superset of the initial one and because we only try to grow the candidate Δ^t by adding rows or columns in a sweeping fashion (row-wise or column-wise).

This algorithm is carefully implemented using vectorized functions from the `numpy` package and it is very fast. Its time complexity is $O(nm)$ for each t.

3.4.3 GreedySweepingIntersection or GSI

This algorithm is similar to GSS, but it does consider adding rows i and i' when $J \not\supseteq J'$ and $J \not\subseteq J'$ by selecting the columns from the intersection $J \cap J'$, and this is why this algorithm takes its name.

More formally, it also considers potential Δ^t by starting from some initial row i and direction d_i. However, it will add other rows i' iterativelly by selecting the one whose column set J' maximizes $|J \cap J'|$, where J is the currently selected column set (J starts being the column set of the initial row i and direction d_i). This process stops when there are no more rows to add or when adding a row would decrease the size of the candidate Δ^t, that is, when $|I| \cdot |J|$ decreases when adding any other row.

As in the GSS algorithm, a similar procedure is also applied "column-wise" and the largest Δ^t found is selected.

This algorithm can not be implemented in a vectorized fashion and it is noticeably slower than GSS. Its time complexity is $O(n^2m + m^2n)$ for each t.

3.4.4 GreedyCellExpander or GCE

This algorithm does not try to grow a Δ^t exclusively row-wise or column-wise as the previous GSS and GSI algorithms do. Instead, for each lift $l_{i,j}$ that has to move, it starts by only considering engaging rows i and j. Then, it iterativelly chooses another row i' or column j' to engage as well. They need to be legal and we need to make sure that no lift that should not move is moved.

The selection of row i' or column j' is done taking into account the new size of the new candidate Δ^t and the number of rows and columns that can still be added in the next iteration.

We maintain the information of which rows and columns can be legally added in the next iteration to improve its performance. However, this algorithm is considerably slower than GSS and GSI. Its time complexity is $O(n^2m^2)$.

3.4.5 SATReduction or SAT

This algorithm simply reduces the problem into SAT by encoding the question "does some Δ-sequence of length T exist for this Δ^*?" into a CNF boolean formula. The idea is to exploit the performance of greatly optimized SAT solvers hoping that most of our problem instances can be solved fast by them, and that exponential time instances are not very common.

Note that the encoding needs some fixed T, and the common approach consists in guessing different values until finding one T such that the encoded formula is satisfiable for T but it is not for $T-1$. A logic strategy would consist in starting with $T = 1$ and exponentially increase it if the encoded formula is unsatisfiable until it becomes satisfiable, lastly proceeding into a binary search between the last unsatisfiable call and the first satisfiable one, finding the optimal T^* with just $O(\log(T^*))$ calls to SAT.

The problem with the previous strategy is that (usually) it is hard to determine that an unsatisfiable formula is unsatisfiable, while it (usually) is much faster to find a satisfiable assignment of a satisfiable formula. Hence, the process that we follow is to find an initial $T \geq T^*$ using some of the greedy algorithms introduced before, and then iterativelly call the SAT solver by encoding the problem using $T \coloneqq T - 1$ until the formula becomes unsatisfiable. This way, there might be many more calls to SAT but all of them except one will be of a satisfiable formula, answering them much faster in practice.

Note that to determine whether the encoded formula is satisfiable or not can take exponential time in the worst case. We limit each SAT solver call with a timeout of 2 seconds, and if it has not finished in that time, we stop and return the previous found solution. The reasoning is that each Δ^t is applied during 2 seconds, and it is only worth to find a Δ-sequence of length $T-1$ if it takes less than 2 seconds to do so. Otherwise, we would obtain a shorter Δ-sequence that will require 2 less seconds to be applied but at the price of spending more than 2 seconds in computing it.

Our proposed encoding tries to minimize the number of clauses and literals. It uses boolean variables to represent the engagement of each row i and column j for each $t \in \{1, \ldots, T\}$, that way defining the Δ-sequence of length T. However, since r_i^t can take 3 different values (-1, 0 and 1), we use the following two variables to represent the row engagement:

$$r_{i,-1}^t = \begin{cases} 1, & \text{if } r_i^t = -1 \\ 0, & \text{otherwise} \end{cases}$$

$$r_{i,1}^t = \begin{cases} 1, & \text{if } r_i^t = 1 \\ 0, & \text{otherwise} \end{cases}$$

And we create the clauses:

$$\neg r_{i,-1}^t \vee \neg r_{i,1}^t$$

This way, if both $r_{i,-1}^t$ and $r_{i,1}^t$ are false in the assignment, then $r_i^t = 0$. Otherwise, at least one of them will be false, and then the other represents the direction of the row engagement.

The lifts that should never move are the lifts $l_{i,j}$ such that $\Delta_{i,j}^* = 0$. We make sure that they are not moved by adding the clauses:

$$\neg r_{i,-1}^t \vee \neg c_j^t$$
$$\neg r_{i,1}^t \vee \neg c_j^t$$

The lifts $l_{i,j}$ that have to move towards direction $d = \text{sign}(\Delta_{i,j}^*) \in \{-1, 1\}$ can not move towards direction $-d$, and we force that by adding the following clauses:

$$\neg r_{i,-d}^t \vee \neg c_j^t$$

Note that so far we need $O(T(n+m))$ variables and $O(nmT)$ clauses. They are only invalidating wrong lift movements, and the SAT solver has the flexibility to choose which lifts to move to legal directions on each Δ^t. However, we still have to enforce that all the lifts that need to move, does so until they all reach the target configuration. We encode this with a cardinality constraint, but first we need some auxiliary variables $a^t_{i,j}$ that are true iff the lifts $l_{i,j}$ move in Δ^t. That is:

$$\begin{aligned} &\neg a^t_{i,j} \vee r^t_{i,d} \\ &\neg a^t_{i,j} \vee c^t_j \\ &\ a^t_{i,j} \vee \neg r^t_{i,d} \vee \neg c^t_j \end{aligned}$$

Finally, each lift $l_{i,j}$ that needs to move does so until they reach their target configuration thanks to the following cardinality constraint:

$$\sum_t a^t_{i,j} = \Delta^*_{i,j}$$

We have used the Minicard SAT solver and the Cardinality Networks [1] to encode this cardinality constraint, since this combination showed the best empirical results.

3.5 Experimentation

3.5.1 Metrics

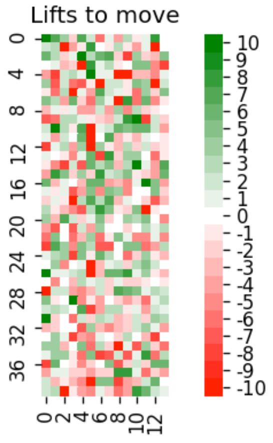
We use the execution time and the length T of the found Δ-sequences as metrics when comparing the proposed algorithms. These two metrics are the quantities that we are most concerned about, since they determine the time needed to reach a target configuration. In fact, the overall time will be the sum of the execution time plus $2T$ seconds, since each Δ^t is applied during 2 seconds, which is the time it takes to move a lift 1cm up or down: The motor rotates at 300rpm and it takes 10 revolutions to move a lift 1cm because the lead of its leadscrew is 1mm.

Hence, we might prefer a fast algorithm that obtains a sub-optimal T over one that obtains the optimal T^* in much greater execution time.

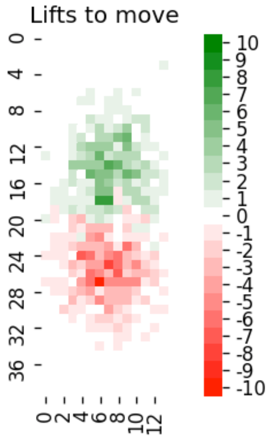
We compare the algorithms using different Δ^* that act as benchmarks, shown in Figure 16. We also use the dimensions of the target full-mattress size: $n = 40$, $m = 14$ and $Z = 10$. These Δ^* used for the comparisons are created as follows:

- **Random:** We generate C and C^* randomly where each $C_{i,j} \sim U[0, Z]$ and $C^*_{i,j} \sim U[0, Z]$. Thus, $\Delta^*_{i,j} \sim U[0, Z] - U[0, Z]$. This Δ^* is useful to check the behaviour of the algorithms to go from an arbitrary initial configuration to an arbitrary target configuration, but it does not reflect real-life scenarios where the configurations are much "smoother"'.
- **Overlapping Gaussian:** We generate initial and target configurations by sampling a multivariate normal distribution parameterized by x and y, which determine the position of the distribution's mean.

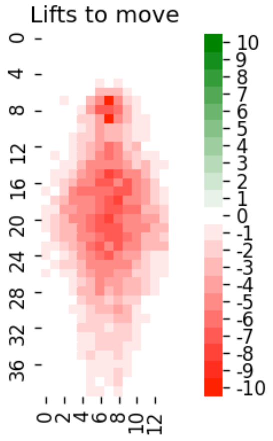
- **Down-to-body:** $C = 0_{n\times m}$ and C^* is built using multiple Gaussian distributions from the previous benchmark in a way such that it roughly resembles the shape of a human body lying upwards. It represents the task of adapting to a user that has just laid down over a previously flat mattress.
- **Body-horizontal-shift:** C is built using multiple Gaussians to resemble a human body lying upwards and C^* is the same as C but sligthly shifted to the right. It represents the task of adapting to a user that has moved slightly to the right while preserving the face upward posture.



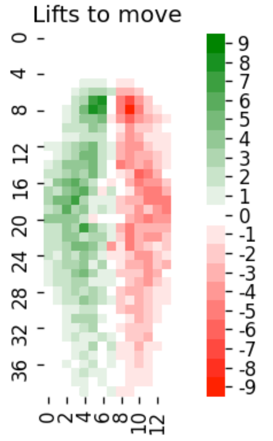
(a) Random



(b) Overlapping-gaussians



(c) Down-to-body

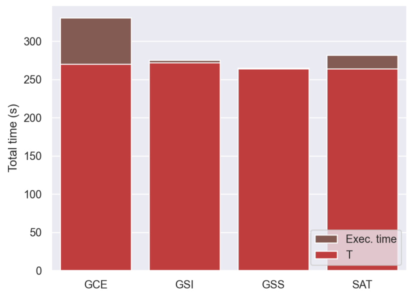


(d) Body-horizontal-shift

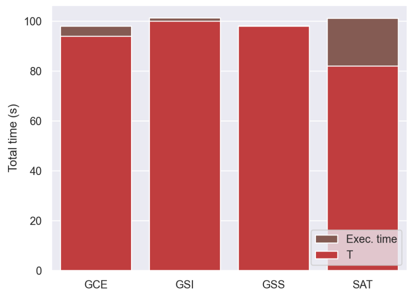
Figure 16: Different Δ^* used to compare the algorithms

3.5.2 Results

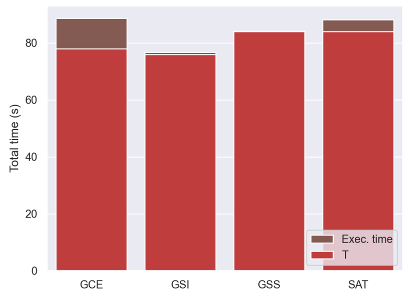
Using the described Δ^*, we get the total time (execution time + $2T$) shown in Figure 17:



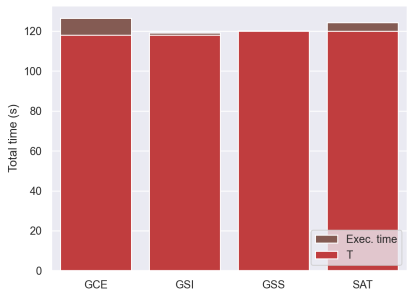
(a) Random



(b) Overlapping-gaussians



(c) Down-to-body

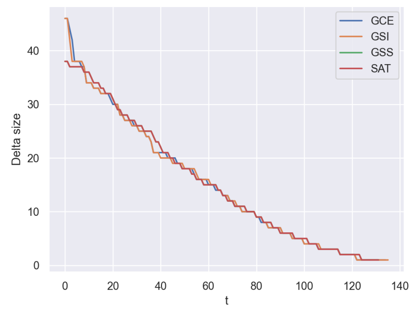


(d) Body-horizontal-shift

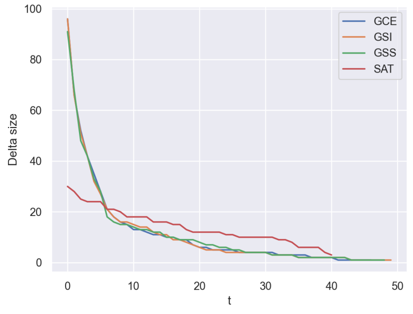
Figure 17: Different Δ^* used to compare the algorithms

As expected, the GLR algorithm takes a lot of execution time compared to the rest with this mattress size (more than 1 hour) and hence, it is not included in these plots. Also, note that SAT has only been able to improve the initial solution of the greedy algorithm in the Overlapping-gaussians benchmark. This is because with the current timeout of 2 seconds, it is not able to find sorter Δ-sequences in the other benchmarks. However, as already indicated, it is not worth to spend more time to find them. Even in the Overlapping-gaussians case where it does find sorter Δ-sequences of length $T = 41$, it does not find the optimal $T^* = 35$ in the allocated time.

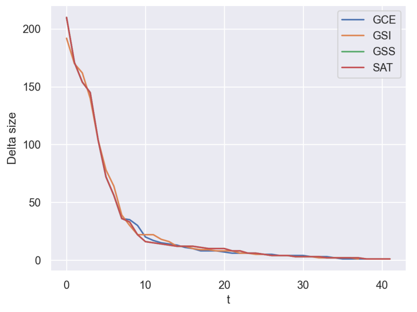
In the case of Overlapping-gaussians where the SAT algorithm does find better solutions, the sizes of the Δ^t are noticeably different respect to the sizes that the other algorithms find, shown in Figure 18:



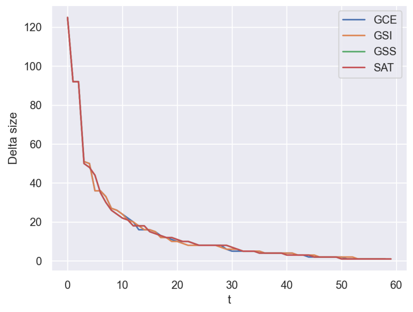
(a) Random



(b) Overlapping-gaussians



(c) Down-to-body

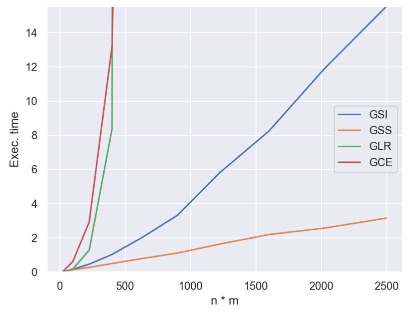


(d) Body-horizontal-shift

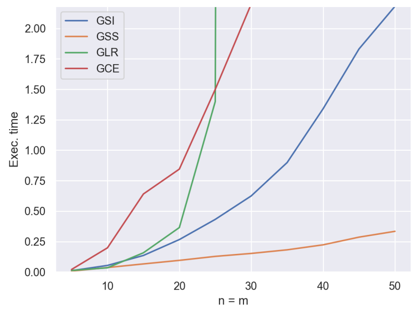
Figure 18: Sizes of Δ^t

This shows that the greedy algorithms try to find Δ^t greedily as large as possible, although a better approach would be to select slightly smaller Δ^t but in greater quantity, reducing the length of the Δ-sequence in the long run. The last Δ^t of the greedy algorithms are very small (only moving a couple of lifts at a time) because they did not forsee that situation when previously selecting Δ^t as large as possible. This is due to the nature of the proposed greedy algorithms, that only try to minimize T by selecting large Δ^t, but as shown in these cases and in the counter-example from the GLR algorithm, this strategy is not always the best one.

The asymptotic time complexity of the greedy algorithms is shown empirically in Figure 19:



(a) Random



(b) Overlapping-gaussians

Figure 19: Execution times for different sizes of squared Δ^*

In these experiments, we generate squared Δ^* of different sizes following the benchmarks Random and Overlapping-gaussians. This way, we obtain problem instances of increasing size and we can evaluate the scalability of the algorithms. GSS is the fastest one by a large difference, followed by GSI. The other two (GCE and GLR) are not scalable and should not be used for large mattress sizes (or larger lift density).

4 Electronic component

The electronic component of this project involves all the electric circuitry to control the motors of the mattress, including the main motor and the servos. But it also involves the design and use of sensors to obtain current information of the user's position and weight distribution.

We have used an Arduino Mega board as microcontroller in order to control all the electronic components. This Arduino is also connected to a computer that runs the higher-level logic, such as the planning algorithms. Hence, a constant communication channel is created between the computer and the Arduino, and instructions and sensor readings are being constantly shared between the two.

4.1 Pressure sensors

One of the main challenges that we faced was related to how to determine the current user's posture and weight distribution. We have considered using a thermal camera centered above the mattress with which to infer the user's position and pressure points using computer vision techniques. Besides the technical challenge of accomplishing this accurately, there is also the potential sense of intimacy violation that a user might feel when there is a camera recording while sleeping.

Instead, we have taken the approach of measuring the weight that each individual lift holds. That is, to have some kind of sensor on each lift to measure its supported weight. However, we had constraints both related to space and price, and this was a difficult task. Since we need 560 sensors (one per lift), the cost of each sensor should not be greater than 0.5€ to avoid the overall price to skyrocket.

Given our constraints, this design phase was a hard challenge, and we contemplated many alternatives that for some reason (mostly price or space) did not work. Some of these discarded alternatives are:

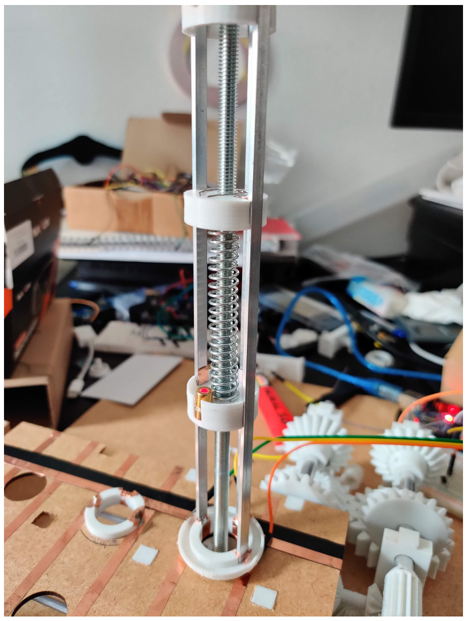
- Force-sensitive sensor: The idea would be to have a force-sensitive sensor between the nut and the telescopic tube. Then, the weight load that the telescopic tube holds would be measured by the sensor. The main problem is the price of such sensors, which is 3.7€ each.
- Spring + sliding potentiometer: By placing a spring between the telescopic tube and the nut, the spring will compress depending on the weight load of the telescopic tube. By knowing its Hooke's constant and by measuring the elongation of the spring, we can infer the weight. This approach uses a sliding potentiometer to measure such elongation. There are cheap sliding potentiometer (for 0.7€) that adjust to our needs, but they are too big and do not fit into the lifts. Also, the fabrication of the lifts becomes highly challenging.

- Spring + ultrasonic distance sensor: This idea replaces the sliding potentiometer by an ultrasonic distance sensor. With some adaptations to common ultrasonic sensors we could fit them into the lifts. However, they are too expensive (3€ each).

As an observation, the sensors that use a spring have an extra advantage: The mattress surface is no longer rigid, and becomes more similar to a regular "soft" mattress that offers a floaty surface to support the body (usually thanks to springs, but also foam, latex, etc).

Fortunately, we found a very simple but perfect idea that also uses a spring to infer the weight load. It adjusts perfectly to our specific constraints and casuistic, and it is much cheaper than the previous alternatives. This idea consists in measuring light intensity to measure the weight.

More in detail, the idea initially consisted in placing a laser diode just next to the nut pointing up. Then, between the spring and the telescopic tube we place a photoresistor that is perfectly aligned respect to the laser (thanks to the rails that fix the nut rotation). Hence, the photoresistor can measure the light intensity of the laser: The more weight that the lift holds, the more compressed the spring will become, so the laser will be closer to the photoresistor, increasing the light intensity. The initial prototype is shown in Figure 20:


(a) Initial prototype

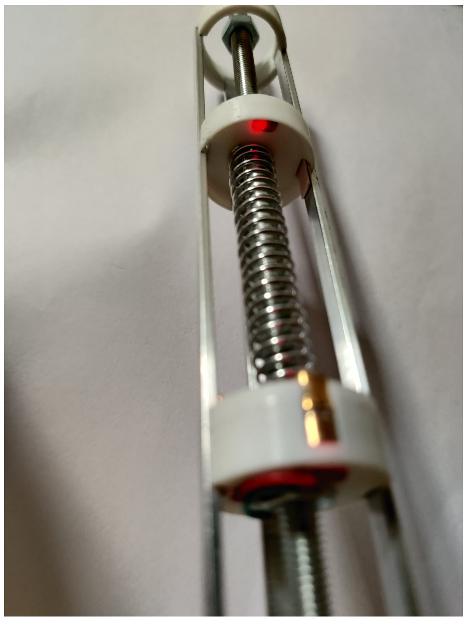

(b) Laser not completely pointing to the photoresistor

Figure 20: Measuring weight with the light intensity of a laser

Although this idea seemed to work well, and we were able to get accurate readings, there were some inconsistent readings too. These were caused by very little oscillations of the nut, which resulted in a misalignment of the laser and the photoresistor, which made the photoresistor to wrongly receive less light intensity than it should for that fixed spring elongation. This behaviour is shown in Figure 20.

To mechanically avoid such misalignments is unfeasible, and what we did instead was to replace the laser by a led, shown in Figure 21. This has several advantages, the first of them being the price, since the cost of a led is much smaller than a laser. Also, a led emits a much broader light beam, instead of the narrow beam that a laser emits. This minimizes a lot the variation of readings due to the alignment of the led and photoresistor, which allows us to obtain more smooth and accurate readings.

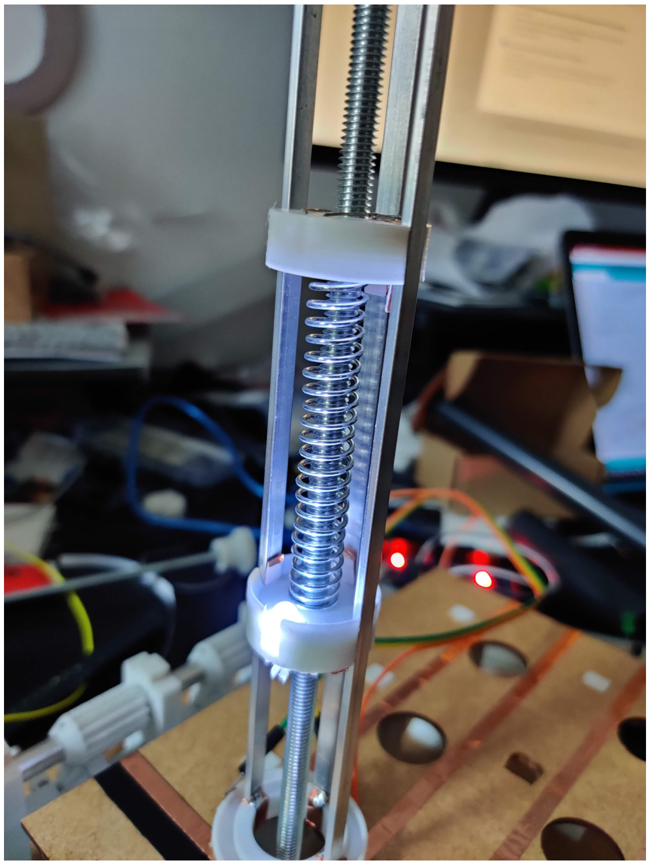


Figure 21: Weight sensor after replacing the laser by a white led

As already mentioned in the mechanical section, some kind of blocking system of the nut was needed since the housing has a circular profile. The common solution is to use a fixed rod that acts as a rail and that allows the vertical displacement of the nut while blocking its rotation. This is the approach that we took, but instead of using a single rail, we use three, because we use them as wires in order to connect the two components of the weight sensor (led and photoresistor). One of the rails is the ground, to which both components are connected, and the other two conduct the current for each component. This was a good idea because we give two functionalities to the same piece: blocking the nut and conducting electricity, and it is much better than simple wires, since they could affect the sensor's readings by covering the light of the led (which could happen when the spring compresses).

However, a seemingly trivial but actually annoying challenge appears: The electric contact from the electronic components to the rails. They can not be soldered because they need to move freely, so the solution consists in using brushes. However, it was very difficult to find a specific design that guaranteed a good conductivity while not creating much friction that affected the movement of the pieces. Several iterations of design were needed to find an appropriate solution, and in Figure 22 a couple of them are shown. Also, the inability of soldering onto the aluminium rails (due to the aluminum oxide) difficulted the connection of the sensors to the rest of the circuit, and we also had to iterate over many different alternatives.

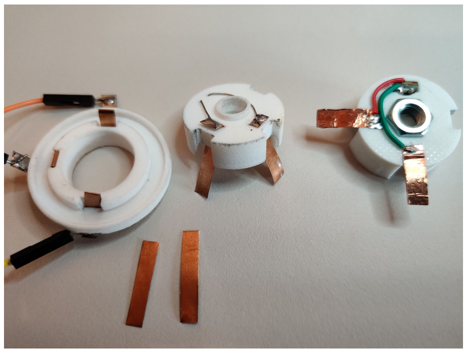


Figure 22: Lift's brushed contacts

4.2 Multiplexing the sensors

Some kind of multiplexation to connect the microcontroller to the weight sensors is needed, since for similar reasons as to the mechanical section, it is infeasible to route $O(nm)$ wires connecting them.

Our solution allows to read an entire row at a time by enabling it while disabling all the others. When a row is enabled, all the leds from that row are turned on. This row activation is done using an NPN transistor on each row.

This way, the photoresistors' resistance takes the respective value depending on the compression of the spring, and we read them individually thanks to a wire that connects all the lifts from the same column. This reading is trivially done thanks to a voltage divider (10 kΩ resistor in series to the active lift on each column) and using an analog converter of the Arduino for each column. The circuit is shown in Figure 23:

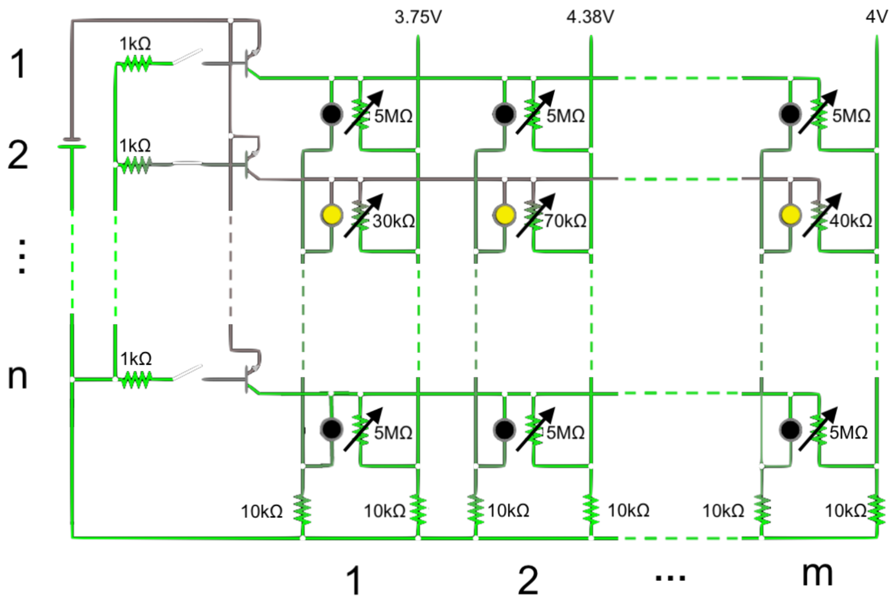


Figure 23: Weight sensor multiplexing circuit while reading row 2

In general, it would not be possible to implement such sensor multiplexation since the resistors are not actually in parallel and they would affect each other. However, we use the fact that the photoresistors' resistance becomes very large when in absence of light. More precisely, in total darkness the resistance of the photoresistors becomes 5 MΩ, which in our case can be approximated to an open circuit. i.e., There is almost no current flowing through "disabled" lifts and thus, they do not interfere in the readings of the active sensors.

The implementation of the circuit from Figure 23 in our prototype is shown in Figure 24. Note that we used a very cheap adhesive copper tape as wires, and this is key in order to scale this for a full-size mattress. This way, each wire line, of which there are $O(n+m)$, only takes $O(1)$ time to place. Using regular wires we would need to spend $O(nm)$ time.

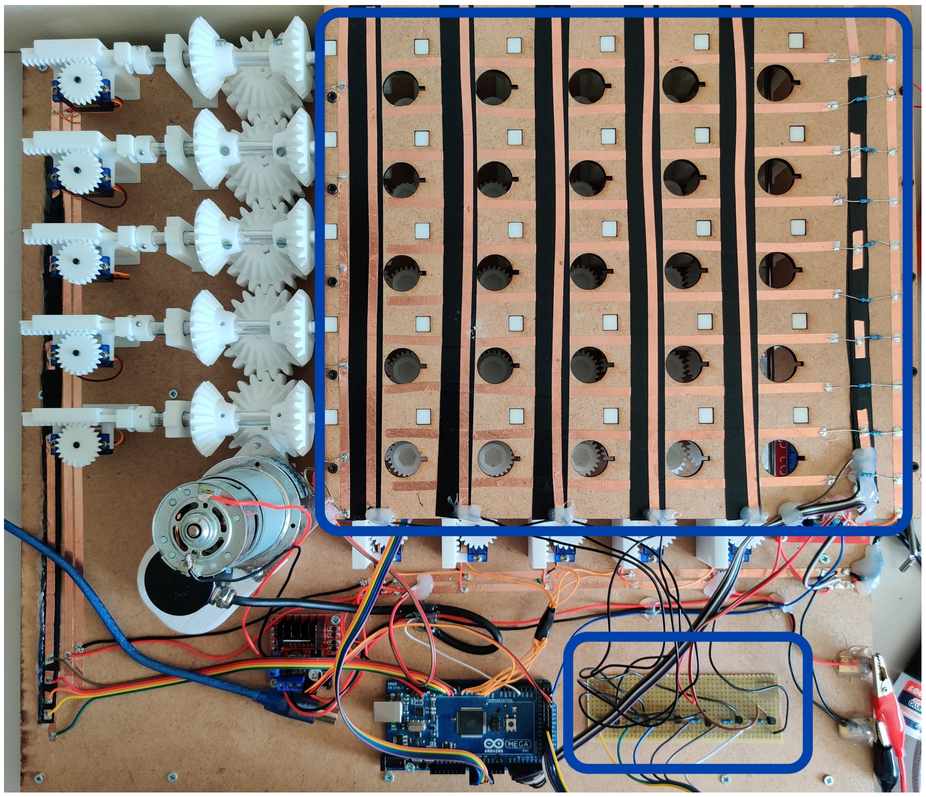


Figure 24: Weight sensor multiplexing circuit in our prototype

4.3 Motor driver and rotary encoder

A common way of driving a motor with a microcontroller is using an intermediate device, called motor driver, that controls the voltage of the motor and thus, its speed. We follow that same scheme, shown in Figure 25, so our microcontroller can choose the voltage of the main motor to be between 0V and 24V.

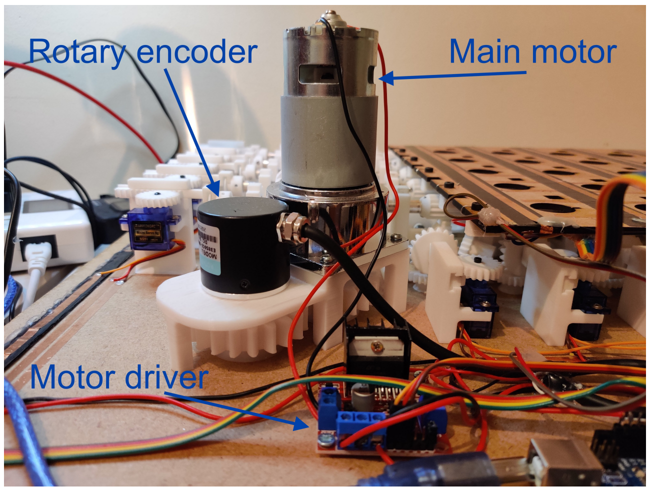


Figure 25: Driving the main motor

However, the actual speed of the motor will be unknown, since in a practical scenario it will be influenced by many factors, including the number and load of the driven lifts. That is why, in order to accurately control the lifts, it is necessary to exactly know how many revolutions the lift's leadscrews have rotated so far. A rotary encoder is used in order to receive such feedback. It consists in a device that by some mechanical system, generates a voltage pulse for each discretized rotation step. Then, using a hardware interruption, the microcontroller can track the rotation of the lift's leadscrews in order to disengage the row and column servos just after 10 leadscrew's rotations (after moving the lifts 1cm).

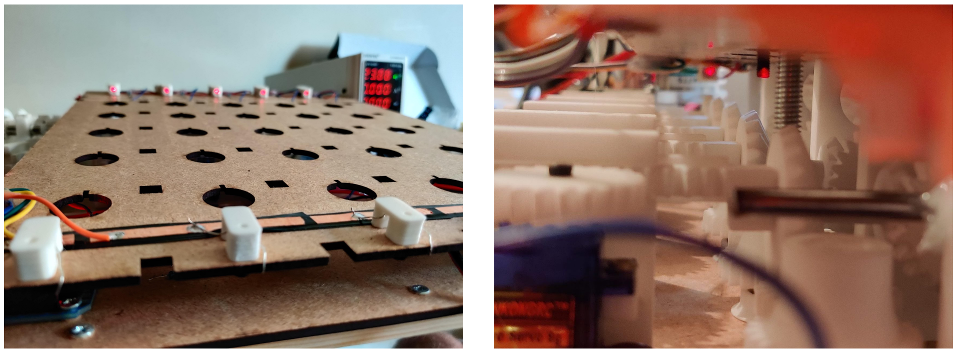
We also use the rotary encoder to measure the speed of the motor in order to briefly adjust it to some low rpm when engaging a row or a column. This is needed because the gears' teeth have to align when engaging, and at high speeds this engagement is very abrupt and damages them, while if it is very slow, there are more chances that the teeth do not engage well.

4.4 End of travel sensors

We also needed to find a scalable way of detecting the end of travel of the lifts. That is, to detect when the vertical position of a lift arrives to a limit, both superior and inferior.

This is needed in order to set the origin of the lifts, making sure that all of them share the same origin. In fact, we have to periodically move all the lifts to their origins in order to re-calibrate them. This is important because even though we try to move each lift exactly 1cm each time we move them, and thus we can keep track of their current position, some minor errors (gear backlash mostly) will accumulate and our tracked position will differ a lot respect to the real position after some time. Hence, after each night, the system should recalibrate in order to reset that error to 0.

The scalable approach that we took consists in placing a laser and a photoresistor for each row and thus, having a cost $O(n)$ instead of $O(nm)$. When a lift arrives to its end of travel, a screw will intercept the laser, covering the photoresistor and thus, allowing the microcontroller to detect it. This is shown in Figure 26:


(a) Lasers (back) and photoresistors (front)

(b) Laser interception

Figure 26: System to detect the end of travel of the lifts

The process of resetting the origin of all the lifts consists in, for each column j:

- Move down all the lifts from column j until each row's laser is intercepted. When a row is intercepted, stop moving that lift.
- Move 1cm up all the lifts from column j.

We can determine that a laser is being intercepted by some lift by toggling the laser while measuring the resistance of the photoresistor. If the resistance is similar in both cases (laser on and off), then we infer that the laser is being intercepted. Otherwise, the laser reaches the photoresistor and the two readings will differ greatly.

The superior end of travel can be detected programatically just using the weight sensors, since thanks to our design, the springs will be compressed when they arrive to the end of travel.

5 AI component

This component is responsible of interpreting the weight sensor's readings in order to infer the current posture of the user and his weight distribution. Then, the most appropriate lift configuration at that moment is determined and the planning algorithms from section 3 are used to actually modify the lift positions as fast as possible. This process should be repeated constantly every time the lifts reach the desired configuration.

The first step is to translate the weight sensor readings into actual weights. Note that we should expect differences between each lift's sensor, mostly due to physical irregularities from the fabrication step and tolerances of the electronic components (resistors, photoresistors, leds, wires, solder joints, etc). We solve this by individually calibrating each sensor. The procedure consists in recording readings for some known (ground-truth) weights, and then fitting a curve in order to accurately predict the weight of future unseen readings.

The next step is considerably harder, and it consists in translating the weight sensor's readings into a model of the user's posture and weight distribution. This should be tackled using computer vision and machine learning techniques.

The final step, related to determining the appropriate lift configuration, can be achieved with heuristics and local search algorithms. For example, an intuitive heuristic could consist in finding a lift configuration that minimizes the maximum weight that a single lift supports. Another potentially good heuristic could be to minimize the gradient of weights.

Unfortunately, due to lack of time, we have not been able to tackle this component of the project.

6 Conclusion

We have provided a radically new mattress design that is intelligent and able to adapt to any person's physical qualities. Such a mattress also has interesting applications in the medical field, since in hospitals there are patients that can not move and they need a mattress that is periodically changing their posture to avoid the generation of ulcers. Our proposed design allows for a finer adaptation to the patient's body than the existing solutions.

This project required dividing a big problem into smaller ones, and solving them in novel ways tacking into account multiple constraints (price, weight, performance, etc). Most of the efforts invested into this project have consisted in ideating the pieces and components of our design from scratch. There is not much formalism or theoretical analysis involved, but intuitive ideas and practical applications. Although this might not seem appropriate for a thesis of a Master program in research, we strongly believe that this project required a lot of inventive mindset and having to discover unusual and new ways of solving novel problems. Also, it forced us to learn a lot from fields very different to our backgrounds, which also allowed us to introduce fresh ideas.

Although we have not been able to build a full-scale prototype as we intended (due to time constraints), the 5×5 version of it shows all the ideas that we introduced and it validates our design. We also have not been able to tackle the artificial intelligence component of the project, but we will work on that in the near future. The project was already too big and we had to invest a lot of time in other fronts.

References

[1] Roberto Javier Asín Achá, R. Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. "Cardinality Networks and Their Applications". In: *SAT*. 2009. DOI: https://doi.org/10.1007/978-3-642-02777-2_18. URL: https://www.cs.upc.edu/~roberto/papers/sat09a.pdf.

[2] J Amilhastre, M.C Vilarem, and P Janssen. "Complexity of minimum biclique cover and minimum biclique decomposition for bipartite domino-free graphs". In: *Discrete Applied Mathematics* 86.2 (1998), pp. 125–144. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(98)00039-0. URL: https://www.sciencedirect.com/science/article/pii/S0166218X98000390.

[3] James Orlin. "Contentment in graph theory: Covering graphs with cliques". In: *Indagationes Mathematicae (Proceedings)* 80.5 (1977), pp. 406–424. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(77)90055-5. URL: https://www.sciencedirect.com/science/article/pii/1385725877900555.

[4] Hideaki Otsuki. “A study of the biclique edge partition and cover problems”. In: 2015. URL: https://nagoya.repo.nii.ac.jp/record/19889/files/o7121_thesis.pdf.

[5] René Peeters. “The maximum edge biclique problem is NP-complete”. In: *Discrete Applied Mathematics* 131.3 (2003), pp. 651–654. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(03)00333-0. URL: http://www.sciencedirect.com/science/article/pii/S0166218X03003330.

[6] Mozhgan Pourmoradnasseri and Dirk Theis. “The Rectangle Covering Number of Random Boolean Matrices”. In: *Electronic Journal of Combinatorics* 24 (June 2017). DOI: 10.37236/5576.

[7] Yuan Sun, Shiwei Ye, Yi Sun, and Tiko Kameda. “Exact and approximate Boolean matrix decomposition with column-use condition”. In: *International Journal of Data Science and Analytics* 1 (Nov. 2016). DOI: 10.1007/s41060-016-0012-3.

[8] Jaideep Vaidya. “Boolean Matrix Decomposition Problem: Theory, Variations and Applications to Data Engineering”. In: *Proceedings - International Conference on Data Engineering* (Apr. 2012), pp. 1222–1224. DOI: 10.1109/ICDE.2012.144.

[9] Valerie L. Watts. “Boolean rank of Kronecker products”. In: *Linear Algebra and its Applications* 336.1 (2001), pp. 261–264. ISSN: 0024-3795. DOI: https://doi.org/10.1016/S0024-3795(01)00338-X. URL: https://www.sciencedirect.com/science/article/pii/S002437950100338X.