Northeastern University

EECE 5659 Control System Engineering

Project Report

ELEVATOR CONTROL SYSTEM OPTIMIZATION FOR A TWO ELEVATOR BUILDING

Presented by Jorge Ortega

INDEX

1. Introduction

Elevators are a highly important part of transportation within buildings, especially in buildings that have a large number of floors. Optimizing elevator operations will result in a significant reduction in waiting times and energy consumption. This project seeks to simulate the logic that governs the movement of one or more elevators in a multiple-floor building.

The main objective of this project is to design and develop efficient and optimized control system algorithms for the operation and coordination of two elevators in a multi-floored building. To achieve this goal the control systems will have to solve the following problems:

1.1. Determine which elevator should respond based on the current positions and direction in which they are moving.

1.2. Conduct time and/or energy effective responses to the user's requests.

1.3. Coordinate both elevators to avoid conflicts and obtain quick responses.

1.4. Avoid the overloading of the elevators.

2. Background and Motivation

Elevators have become a must in every modern building. They are the most efficient way of transportation invented for high-rise buildings. Motivated by the challenges of developing a control system to optimize elevator operations, this project explores the design of simple and complex control strategies. The simulation of these systems allows us to compare and evaluate the results ensuring an advanced solution for real-world implementation.

3. State of Art and Main Contribution.

The earliest references found of elevators date back to 236 B.C. when it is believed that the Greek mathematician and engineer Archimedes invented the first elevator in history. This model worked by wrapping a rope over a large drum that was turned by several people pulling at once. This model evolved over the centuries where they were included in large buildings such as the Colosseum of Rome where they were used to move animals.[1]

The modern versions of the elevators that we know appeared in the 19th century with the invention of the safety brake, created by Elisha Graves Otis in 1852. This emergency brake system set the beginning of the development of modern elevators that are used nowadays in every multiple-floor building.[1]

Different elevator control systems have been developed to satisfy the diverse needs of commercial and residential buildings. Below are four primary types of elevator controls, emphasizing their functionality and typical applications:

- **Selective Collective Operation:** This is the most common system used in commercial elevators. It allows the clients to indicate the desired direction (up or down), and the elevator stops sequentially at the requested floors. This system is efficient for moderate-traffic situations[2].
- **Constant pressure:** In this system the elevator moves only while the buttons are continuously pressed. If the pressure on the buttons is released, the elevator is stopped immediately. Constant pressure controls are mainly used for wheelchair lifts[2].
- **Nonselective Collective Operation:** This control system is commonly used in residential elevators. It responds to the calls in the order they are made. Thus, the elevator will travel through the floors in the exact order of requests, regardless of efficiency[2].
- **Single Automatic Operation:** This system executes one command at a time. It will wait to receive a new input only after completing the first request. It lacks collective functionality, so it is frequently used for vertical reciprocating conveyor applications[2].

This project builds on these existing systems by exploring three different advanced strategies. Each of these three controllers will be simulated in MATLAB to evaluate its time effectiveness. Below is a brief introduction to each of them:

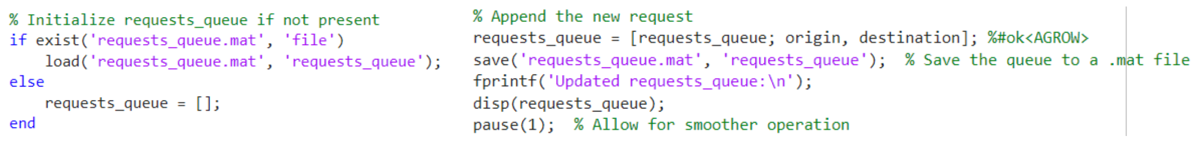
1. **First Call Controller (FCC):** This controller coordinates the elements in Nonselective Collective Operation in two elevators that allow the users to specify the destination before entering the elevator. The requests are stored in a matrix and the elevators fulfill all the requests in the same order in which they were saved allowing simultaneous operation.
2. **Closest Call Controller (CCC):** This algorithm extends the Nonselective Collective Operation concept by incorporating proximity-based request assignment. As a result, the distances traveled by each elevator are reduced. Destinations are selected before entering the elevator.
3. **Optimized Multi-Elevator Controller (OMEC):** Optimization of Selective Collective Operation by introducing grouping logic for requests in the same direction to maximize efficiency and minimize redundant trips. This Controller is coordinated simultaneously by two elevators.

4. Analysis of Work

The first step to design the control systems previously introduced was figuring out how to simulate in MATLAB the movement of one individual elevator. The main idea was to represent the decisions made by the elevator during the whole work process in a simple way. The work process of the elevator includes receiving a request, moving to the floor in which the request was originated and transporting the client to the desired destination. To make the simulation easy to visualize, the elevator must give constant feedback of its current state through the MATLAB Command Window.

The first major challenge encountered was including the possibility of generating the input at the same time as the output. In other words, to make the simulation realistic the algorithm should be able to let new clients make a request while the elevator is working on a previous request. This implementation allows real time operation of the elevator as the input can be edited while the elevator simulation is in progress.

To solve this problem the simulation was divided into two different codes that are executed at the same time in two different MATLAB instances. The first code was named *“Elevator_input.m”* and it is used in the three controllers that are developed in this report. It governs the storage of requests and is edited by the user and the elevator simulations at the same time.



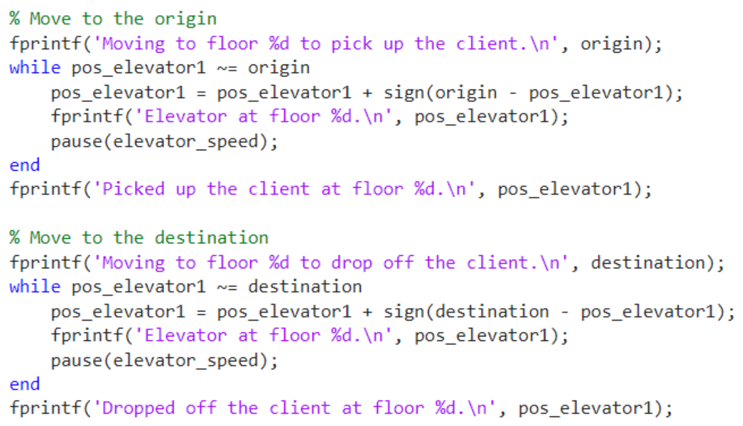
```
% Initialize requests_queue if not present
if exist('requests_queue.mat', 'file')
    load('requests_queue.mat', 'requests_queue');
else
    requests_queue = [];
end
```

```
% Append the new request
requests_queue = [requests_queue; origin, destination]; %#ok<AGROW>
save('requests_queue.mat', 'requests_queue');  % Save the queue to a .mat file
fprintf('Updated requests_queue:\n');
disp(requests_queue);
pause(1);  % Allow for smoother operation
```

Figure 1:Request storage in file.mat

The requests, which consist of an origin and a destination, are saved in a matrix called *“requests_queue”* which is uploaded to a file.mat. The *“requests_queue”* matrix is made of two columns (origin and destination) and *n* rows which corresponds to the number of requests stored in the matrix. The use of the *file.mat* allows the code that simulates the movement of the elevators to read and update the *“requests_queue”* matrix. Every time a new request is registered the input code adds a new row to the matrix. Meanwhile, every time a request is assigned to an elevator it is deleted from the matrix to make sure that it is only accomplished just one time.

Once the input is isolated in one individual script the second code can be entirely used to simulate the request assignation and movement of the elevators. This process starts by loading the request queue and assigning the values of the first row to two variables: origin and destination. Then the request is deleted from the matrix and the updated queue is saved into the *file.mat*. Finally, the simulation of the elevator’s movement was carried out in two loops. The first loop will bring the elevator from its original position to the origin of the request previously assigned. On the other hand, the second loop will bring the client from the origin to the destination. During this process, the elevator uses its current position and the difference between this value and the floor that is trying to reach. This can be seen in figure 2.



```
% Move to the origin
fprintf('Moving to floor %d to pick up the client.\n', origin);
while pos_elevator1 ~= origin
    pos_elevator1 = pos_elevator1 + sign(origin - pos_elevator1);
    fprintf('Elevator at floor %d.\n', pos_elevator1);
    pause(elevator_speed);
end
fprintf('Picked up the client at floor %d.\n', pos_elevator1);

% Move to the destination
fprintf('Moving to floor %d to drop off the client.\n', destination);
while pos_elevator1 ~= destination
    pos_elevator1 = pos_elevator1 + sign(destination - pos_elevator1);
    fprintf('Elevator at floor %d.\n', pos_elevator1);
    pause(elevator_speed);
end
fprintf('Dropped off the client at floor %d.\n', pos_elevator1);
```

Figure 2: Movement loops

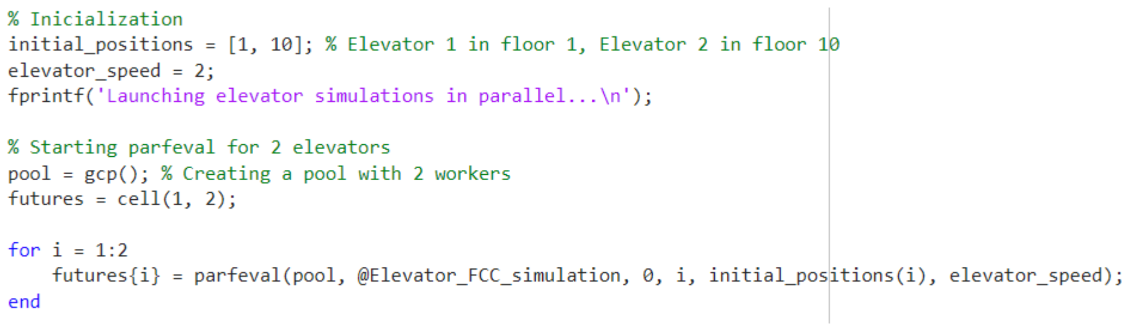
As mentioned before, it is continuously displayed through the Command Window the state of the elevator.

4.1. First Call Controller (FCC)

A second major problem was faced at this point. Once the simulation for a single elevator worked properly the first controller designed for this project consisted in transitioning the single elevator simulation to a dual elevator system where both elevators cooperated working at the same time. The problem arose because of the inability to execute two different loops of the simulation code (one for each elevator) concurrently. The sequential execution model used in MATLAB prevented both elevators from operating independently, as one elevator's loop blocks the other from starting until it is completed.

To overcome this problem, I used **parfeval**. The parfeval function from MATLAB's Parallel Computing Toolbox enables asynchronous execution of functions. This means that it is possible to run the simulation code in multiple instances. Each instance is assigned to a separate worker (elevator 1, elevator 2…) in a parallel pool, ensuring simultaneous operation.

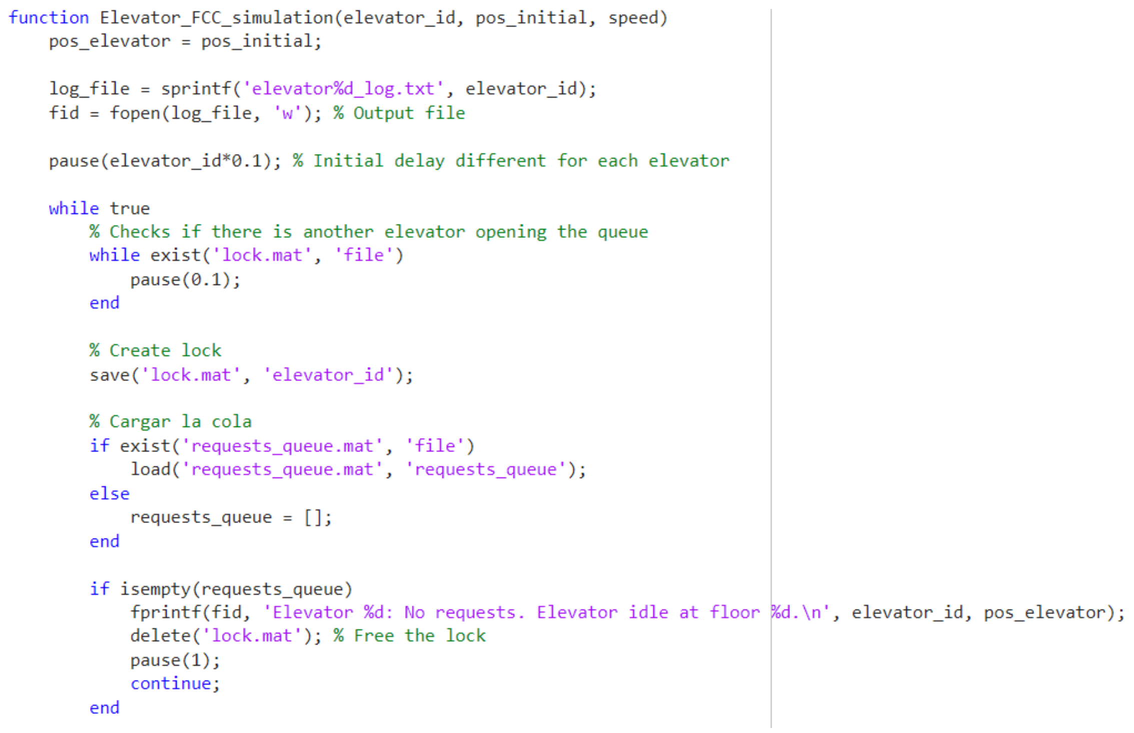
Consequently, in the design of the First Call Controller two scripts were developed. The first one is the parfeval function which is in charge of the individual elevator process. The second script creates a pool of two workers (2 elevators) that simultaneously execute the function.



```
% Inicialization
initial_positions = [1, 10]; % Elevator 1 in floor 1, Elevator 2 in floor 10
elevator_speed = 2;
fprintf('Launching elevator simulations in parallel...\n');

% Starting parfeval for 2 elevators
pool = gcp(); % Creating a pool with 2 workers
futures = cell(1, 2);

for i = 1:2
    futures{i} = parfeval(pool, @Elevator_FCC_simulation, 0, i, initial_positions(i), elevator_speed);
end
```

Figure 3: Initialization of the parfeval function with 2 workers

To monitor and analyze each elevator’s operation in real time, each worker displays the output in a different *txt* file. This approach provided clear visibility into the elevator´s movement and processing the requests enabling real time debugging and performance evaluation.



```
function Elevator_FCC_simulation(elevator_id, pos_initial, speed)
    pos_elevator = pos_initial;

    log_file = sprintf('elevator%d_log.txt', elevator_id);
    fid = fopen(log_file, 'w'); % Output file

    pause(elevator_id*0.1); % Initial delay different for each elevator

    while true
        % Checks if there is another elevator opening the queue
        while exist('lock.mat', 'file')
            pause(0.1);
        end

        % Create lock
        save('lock.mat', 'elevator_id');

        % Cargar la cola
        if exist('requests_queue.mat', 'file')
            load('requests_queue.mat', 'requests_queue');
        else
            requests_queue = [];
        end

        if isempty(requests_queue)
            fprintf(fid, 'Elevator %d: No requests. Elevator idle at floor %d.\n', elevator_id, pos_elevator);
            delete('lock.mat'); % Free the lock
            pause(1);
            continue;
        end
```

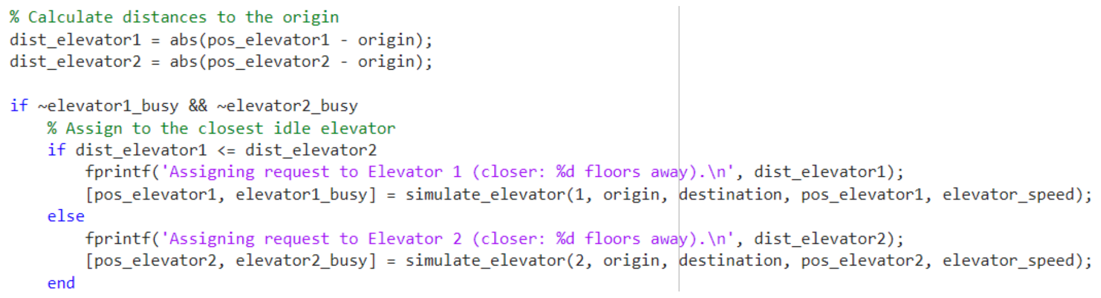
Figure 4: FCC simulation code

Finally, to prevent both elevators from fulfilling the same request, a lock file is created when an elevator is acceding to the file.mat and additionally, there is a brief pause proportional to the index of the elevator at the beginning of the loop. This pause ensures that in the first iteration the elevators assign their first request at different times. Otherwise, the first request after a time when both elevators are idle is assigned to both workers.

These solutions successfully enabled the simultaneous operation of both elevators, closely replicating real-world behavior and paving the way for further optimization and coordination.

4.2. Closest Call Controller (CCC)

The Closest Call Controller's goal was to reduce the distances travelled by the elevators compared to the First Call Controller. To do so, this implementation assigns incoming requests to the elevator that is physically closest to the origin of the request. Unlike the FCC and the Optimized Multi-Elevator Controller that use parfeval for simultaneous operation, this controller operates sequentially, meaning that when one elevator is processing a request, the other remains idle.



```
% Calculate distances to the origin
dist_elevator1 = abs(pos_elevator1 - origin);
dist_elevator2 = abs(pos_elevator2 - origin);

if ~elevator1_busy && ~elevator2_busy
    % Assign to the closest idle elevator
    if dist_elevator1 <= dist_elevator2
        fprintf('Assigning request to Elevator 1 (closer: %d floors away).\n', dist_elevator1);
        [pos_elevator1, elevator1_busy] = simulate_elevator(1, origin, destination, pos_elevator1, elevator_speed);
    else
        fprintf('Assigning request to Elevator 2 (closer: %d floors away).\n', dist_elevator2);
        [pos_elevator2, elevator2_busy] = simulate_elevator(2, origin, destination, pos_elevator2, elevator_speed);
    end
```

Figure 5: Request assignation in CCC

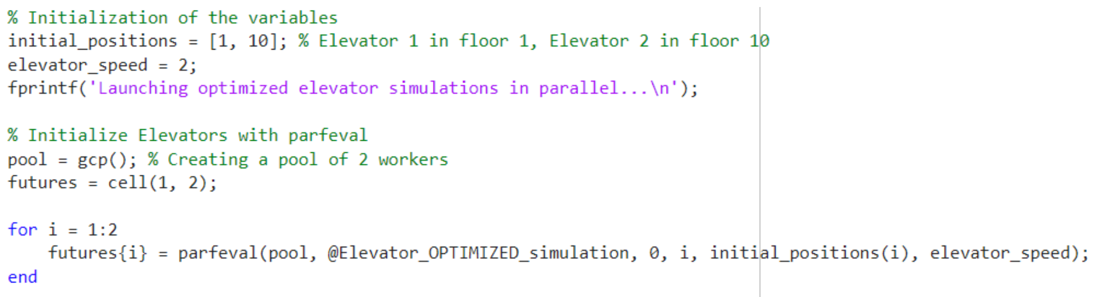
The Closest Call Controller evaluates each incoming request by calculating the distance between the origin of the request and the current position of each elevator. Elevator 1 was provided with default priority for cases in which both elevators are equidistant to a request.

This simpler logic ensures efficient allocation of requests based on proximity. However, the sequential nature of this controller limits the overall efficiency since only one elevator can be active at a time.

4.3. Optimized Multi-Elevator Controller (OMEC)

The Optimized Multi-Elevator Controller is the most complex of all the controllers developed in this project. This optimized controller uses an intelligent request grouping and direction-based fulfillment strategy to maximize time-efficiency.

To maintain the simultaneous operation of two elevators, the OMEC follows the same structure as the First Call Controller. A parfeval function is used in a pool of two workers who represent the work of each elevator. This function is called in a MATLAB script named *"Elevator_OPTIMIZED_run.m"* as shown in figure 6.

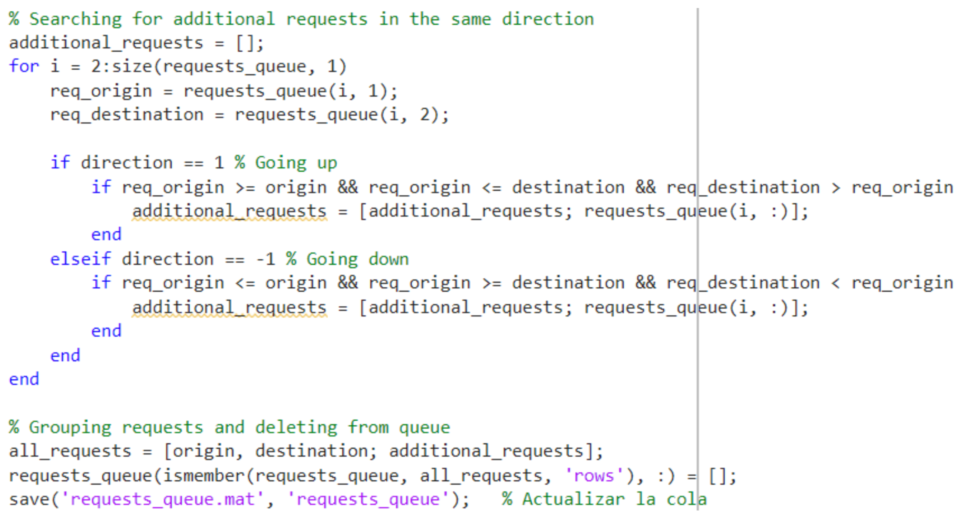


```
% Initialization of the variables
initial_positions = [1, 10]; % Elevator 1 in floor 1, Elevator 2 in floor 10
elevator_speed = 2;
fprintf('Launching optimized elevator simulations in parallel...\n');

% Initialize Elevators with parfeval
pool = gcp(); % Creating a pool of 2 workers
futures = cell(1, 2);

for i = 1:2
    futures{i} = parfeval(pool, @Elevator_OPTIMIZED_simulation, 0, i, initial_positions(i), elevator_speed);
end
```

Figure 6: Initialization of the Optimized Controller function with 2 workers

As previously explained, all the requests are managed in a separate file and, just as the other two controllers, they are loaded from requests_queue.mat. However, a great improvement is added in the request handling of this Optimized Multi-Elevator Controller.

Unlike basic controllers that handle one request at a time, this controller evaluates the entire queue and identifies the requests that align with the current direction of the elevator. In other words, while fulfilling a request, the elevator searches in the requests matrix for additional requests in the same direction and within its current trajectory. Requests that share overlapping floors or fall within the same travel path are grouped and processed in the same trip.

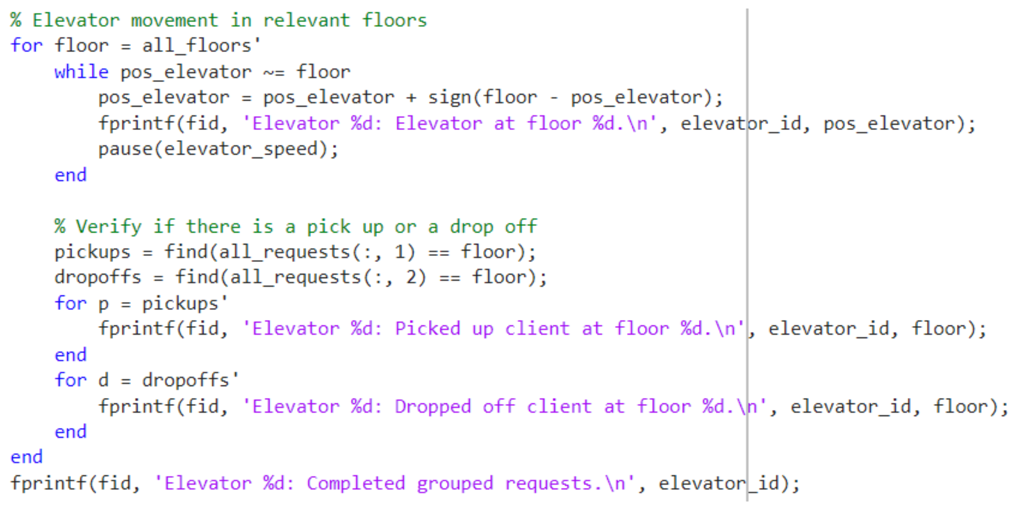


```
% Searching for additional requests in the same direction
additional_requests = [];
for i = 2:size(requests_queue, 1)
    req_origin = requests_queue(i, 1);
    req_destination = requests_queue(i, 2);

    if direction == 1 % Going up
        if req_origin >= origin && req_origin <= destination && req_destination > req_origin
            additional_requests = [additional_requests; requests_queue(i, :)];
        end
    elseif direction == -1 % Going down
        if req_origin <= origin && req_origin >= destination && req_destination < req_origin
            additional_requests = [additional_requests; requests_queue(i, :)];
        end
    end
end

% Grouping requests and deleting from queue
all_requests = [origin, destination; additional_requests];
requests_queue(ismember(requests_queue, all_requests, 'rows'), :) = [];
save('requests_queue.mat', 'requests_queue');   % Actualizar la cola
```

Figure 7: Dynamic grouping of requests

Before the elevator starts moving the grouped requests are removed from the queue to avoid being assigned for the other elevator and the updated requests matrix is loaded to the *file.mat*.

The movement simulation in this controller brought another big challenge. Once the requests are grouped the elevator needs to design a route to travel through all the relevant floors without changing the previously selected direction. To handle this problem the group of requests are reorganized in ascending order for upward requests and in descending order for downward requests.



```
% Elevator movement in relevant floors
for floor = all_floors'
    while pos_elevator ~= floor
        pos_elevator = pos_elevator + sign(floor - pos_elevator);
        fprintf(fid, 'Elevator %d: Elevator at floor %d.\n', elevator_id, pos_elevator);
        pause(elevator_speed);
    end

    % Verify if there is a pick up or a drop off
    pickups = find(all_requests(:, 1) == floor);
    dropoffs = find(all_requests(:, 2) == floor);
    for p = pickups'
        fprintf(fid, 'Elevator %d: Picked up client at floor %d.\n', elevator_id, floor);
    end
    for d = dropoffs'
        fprintf(fid, 'Elevator %d: Dropped off client at floor %d.\n', elevator_id, floor);
    end
end
fprintf(fid, 'Elevator %d: Completed grouped requests.\n', elevator_id);
```

Figure 8: Pickup and Drop-off logic

During the elevator movement, the loop checks at each floor for both pickups and drop-offs, ensuring that no client is missed. A locking mechanism is used again to ensure safe access to the shared queue by preventing both elevators from modifying it simultaneously.

In addition, the elevators operations are monitored in separate *txt* files using the same method explained in the First Call Controller.

This approach minimizes idle time by fulfilling multiple requests in the same trip which reduces the operating time of the elevator and the distance traveled by each elevator. Overall, the Optimized Multi-Elevator Controller ensures a very effective load distribution making it ideal for high-traffic scenarios.

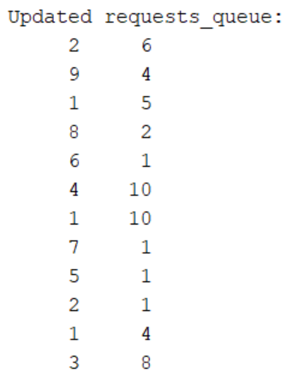
5. Results

The analysis in performance of the different elevator control strategies implemented was carried out in a 12-request simulation. For each controller efficiency is evaluated by tracking key metrics such as the position of the elevators over time, the number of fulfilled requests by each elevator, the number of floors traveled, etc.

The timing mechanism used to track the elapsed time between floors is the MATLAB command **tic** and **toc**. Saving the timestamps during the elevator simulation allows the representation of the movement of each elevator over time.

5.1. Analysis Test

The analysis test consists of 12 requests that are given at the same time to all the controllers. The following figures show the results obtained in the three simulations where elevator 1 begins idle on the first floor and elevator starts also idle on the top floor. The elevator speed is set to 2 seconds between floors and the pause to pick up or drop off a client is 3 seconds long.



```
Updated requests_queue:
     2     6
     9     4
     1     5
     8     2
     6     1
     4    10
     1    10
     7     1
     5     1
     2     1
     1     4
     3     8
```

Figure 9: Requests queue for Test 1

	CCC	FCC	OMEC
Total time (s)	226.9	126.7	90.5
Floors travelled by Elevator 1	49	45	27
Floors travelled by Elevator 2	28	38	11
Total floors travelled	77	83	38
Requests fulfilled by Elevator 1	8	6	6
Requests fulfilled by Elevator 2	4	6	6
Total Requests fulfilled	12	12	12

Table 1: Simulation Results

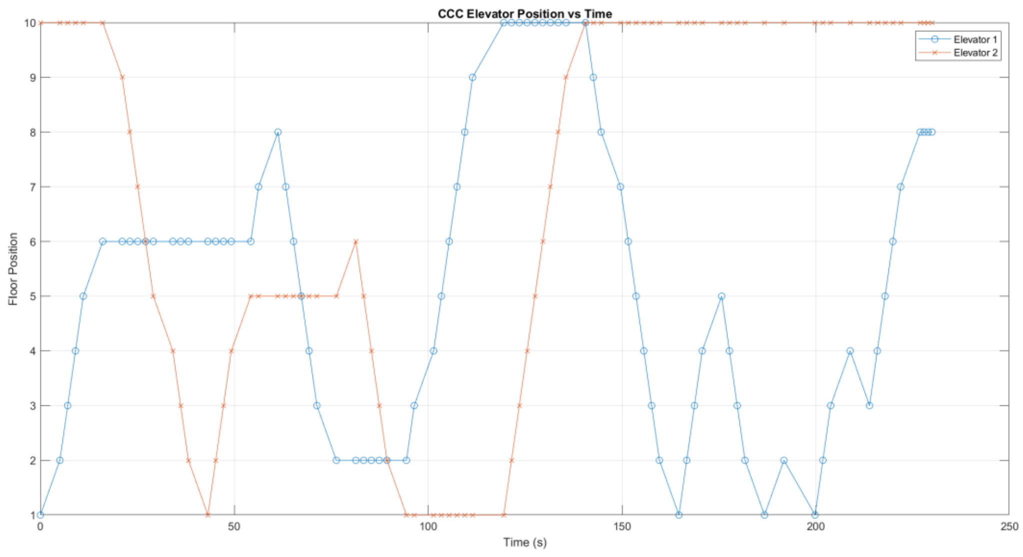


Figure 10: CCC Elevator Position vs Time

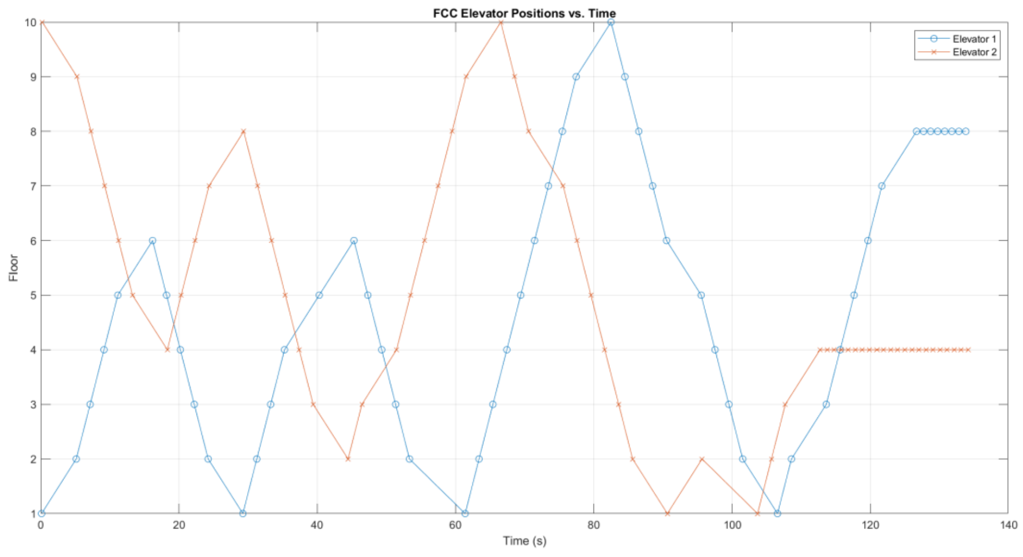


Figure 11: FCC Elevator Position vs Time

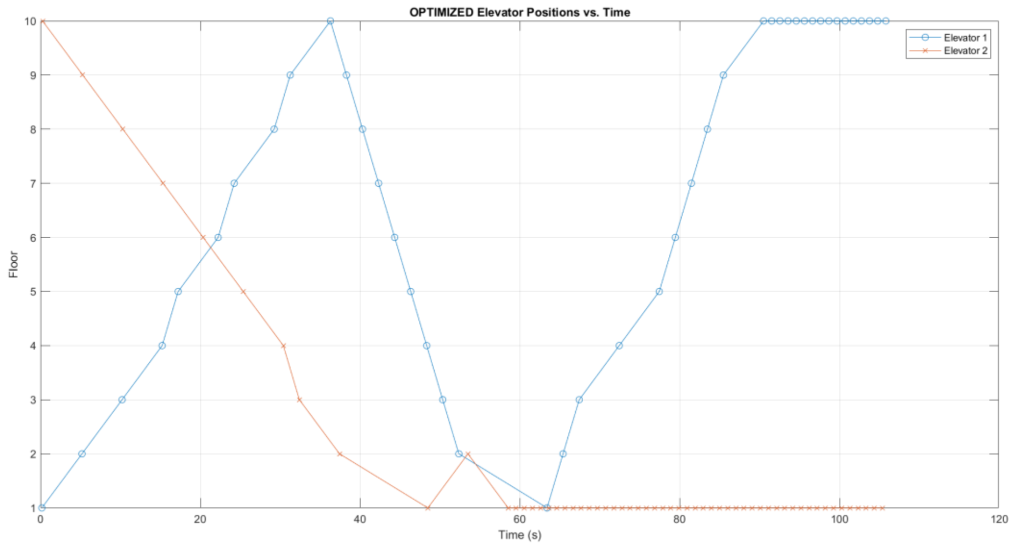


Figure 12: OMEC Elevator Position vs Time

The comparison of the three implemented control systems of elevators highlights clear differences in time efficiency and workload distribution. As it was expected, CCC took the longest total time to complete the 12 requests (226.9 seconds) as shown in figure 10. This is mainly due to the sequential processing of the requests, leaving always one elevator idle. However, the CCC reduced the number of floors collectively traveled in comparison to the FCC (77 vs 83), which achieves the goal set for this elevator control strategy.

The parallel processing used in the FCC significantly improved time efficiency by reducing the total time to 126.7 seconds. Moreover, the workload was perfectly distributed unlike the CCC, in which two thirds of the workload was carried out by Elevator 1.

Nevertheless, the most efficient approach was clearly the Optimized Multi-Elevator Controller which minimized the total time to just 90.5 seconds by grouping and optimizing requests based on their travel direction. The OMEC workload distribution was perfect, and the number of floors travelled was also substantially optimized, with the elevators travelling less than half of CCC and FCC.

6. Conclusions

Through this project, I obtained valuable knowledge about the challenges of designing and implementing Rea time control systems. Developing the Closest Call Controller, the First Call Controller and the Optimized Multi-Elevator Controller strategies provided me with a deep understanding in parallel processing and work optimization.

Comparing the results of the simulations emphasized the importance of optimization and reducing operational time. The OMEC approach proved to be the most time and energy efficient. This optimization was achieved through iterative development, testing, and refinement of the control algorithms, illustrating the value of incremental improvements.

For future work, I would plan to explore the combination of the Closer Call Controller with the OMEC strategies. I would also aim to simulate larger-scale systems with more elevators, which I know is possible by adding more workers to the parfeval function, and buildings with a higher number of floors.

7. References

[1] Nationwide Lifts, “The Invention and History of the Elevator.”

[2] Cibes, “Elevator control system operations & codes.”