Theoretical Part

Task 1:

As the mechanical and electrical engineering is already done, now I have to implement a compatible architecture for the brain of the robot. As the robot is equipped with gear motors, PWM-controlled motor drivers for wheel and servo motors for wheel, a rotary encoder, GPS, IMU, and a stereo camera, I'm going to choose a single highly computation powered SBC for the architecture of the brain. In this case, The Raspberry Pi can be go to option because of this system's performance and capabilities.

The Architecture: As a single Raspberry Pi board has reasonably enough computational power to implement and run all of the above mentioned components, I've chosen a single SBC for this architecture. The Raspberry Pi can effortlessly handle various computational heavy tasks like image processing from the stereo camera, sensor fusion with IMU and GPS data while running the necessary motor drivers. In this architecture, the gear motors can be connected to the PWM-controlled motor drivers like L298N. With the in-built hardware PWM hardware capabilities, the rotary encoder can be integrated in the architecture. Thus, using many other modules or library function, all of the components can be connected and integrated in the architecture.

Reason: In this single SBC architecture, all of the components can be connected in a single board quite easily and the communication as well as data sharing between different sensors and actuators can be done easily too. This can effectively enhance the performance and overall balance of the robot. Also, the low cost and space efficiency motivated me to choose this single SBC architecture. As it has only one Raspberry Pi controller, the cost as well as the space can be minimized and the whole system overall, can be simplified.

Task 2:

If one Arduino Mega and one Arduino Nano are on the same PCB board, I would choose I2C as the communication protocol. To establish my argument first a brief overview of I2C, UART and SPI is given below:

- **UART (Universal Asynchronous Receiver/Transmitter)**: UART is a communication protocol which is really simple to implement. It uses two wires for communication, one for receiving and one for transmitting data. It is Asynchronous. It has multiple mode options (Simplex, Half-Duplex, Full-Duplex) and it supports different electronic devices and microcontrollers.
- **SPI (Serial Peripheral Interface):**
  This is a synchronous communication protocol which is widely used. In this, the master and slave devices share a common clock signal ensuring simultaneous transfer. It

operates at a highspeed making data transfer fast. Also it allows multiple slave devices.

- **I2C (Inter-Integrated Circuit):**
  This also uses two wires for communication, a data line and a clock line. I2C buses allow multiple device support by using addresses for devices. This allows communication without interference. It supports multi- master communication.

Reason for choosing I2C:

I2C is a synchronous protocol unlike UART. It uses only 2 wires which is better than using 4 wires for SPI.

UART cannot simultaneously transfer data which I2C can. Also I2C has built-in error-checking. I2C is great for designs with space constrictions as it requires few pins compared to SPI.

Both Arduinos have a built in I2C module which means these devices will be efficient in implementing I2C.

UART is also a great option for this as it is specifically designed for communication between two devices. As I want Synchronous transfer, I am choosing I2C but if it's not an issue then UART can be chosen as well. But UART is less immune to noise than I2C.

Implementing SPI is quite difficult compared to I2C. It is mainly used for one master and multiple slave devices. So, for highspeed data transfer between different sensors or displays,

communication modules such as radio transceivers, SPI is a good choice.

As I2C allows multi-master and multiple devices so it is great for communication between multiple devices, so if I want to further extend my circuit, I can do so if I use I2C. That is why I am choosing I2C as my communication protocol.

CODE SNIPPET FOR ARDUINO MEGA (MASTER):

```
#include <Wire.h>
void setup() {
   Wire.begin
   Serial.begin(9600);
}
void loop() {
   Wire.requestFrom(9, 5);
   while (Wire.available()) {
      int data = Wire.read();
      Serial.print("Received data: ");
      Serial.println(data);
   }
   delay(1000);
}
```

CODE SNIPPET FOR ARDUINO NANO (SLAVE):

```
#include <Wire.h>

void setup() {
   Wire.begin(9);
   Wire.onRequest(SendData);
   Serial.begin(9600);

}

void loop() {
   delay(50);
}

void SendData() {
   int data = 2023;
   Wire.write(data);
   Serial.print("Sending data: 2023");
}
```

Task 3A:

- **DC Motor:**
  DC motors convert electrical energy into mechanical energy. These motors are simple and efficient as well as easy to use. DC motors provide continuous rotation. DC motors can be of various kinds (Brushed, Brushless). Their speed can be controlled by controlling the voltage. A higher voltage when applied provides a higher speed and reducing the voltage reduces the speed. Using Pulse Width modulation technique the power loss while changing voltage in a DC motor can be controlled.
  DC motors are used in various fields like Robotics, vehicles, appliances, machines etc. These motors provide speed control, which is highly appreciated in making moving parts of robots. DC motors are also used in fans, vacuum cleaners, blenders, pumps, electric cars etc.
- **Stepper Motor:**
  Stepper motor is a brushless DC electric motor. The motor divides the full rotation into particular steps. Each step is at a fixed angular displacement. These are mainly used for precise angular control. These motors can be Unipolar or Bipolar.
  Stepper motors are usually controlled using a microcontroller. The electrical pulses sent to the motor driver gives it instructions on how to do the rotation.

Stepper motors are used in Camera systems, optical equipment, laser, robotic joints, floppy disk drives, printers, plotters, CNC machines, 3D printers.
(Source: Wikipedia)

Task 3B:

Motor drivers are a robotic component which provides an interface between the control system and the motor. Microcontrollers and motors work on different voltage ranges. So, the motor driver provides a higher current level and controls the voltage supply. Specific motor driver chips exist for every kind of motor.
It controls the motion of the motor and other factors like power delivery. The output usually uses PWM which also amplifies input signals from the microcontroller. A driver maintains the logic levels in a circuit.
Different types of drivers for different motors are:

1. H-Bridge Drivers, Brushed motor drivers for DC motors (A4988, L293, DRV8833).
2. Unipolar and Bipolar Stepper Motor Drivers for Stepper Motors (A4988, L293, DRV8833, ULN2003).
3. Servo Motor Drivers for Servo Motors (L293).

(Source: OurPCB)

Task 3C:

PWM or Pulse Width Modulation is a modulation technique. It turns digital signals to analog by generating variable-width pulses which represent the amplitude of the analog inputs. This is used in DC motors, LED, power regulation, audio amplifiers etc. PWM is used to control the average voltage, speed as well as for the higher efficiency, the precise control and digital compatibility.

PWM works by creating a varying duty cycle while keeping the frequency constant. For example, if a duty cycle is 50% the device will remain on for half the time and will remain off the rest half of the time. If it is 75% the device will stay on for longer than off. This ensures the frequency remains same while outputting more power. As the frequency is quite high, the device receives an average power of the duty cycle. This is how PWM is used to control motors.

Task 3D:

How to implement a motor feedback using encoders:

1. Choosing and attaching encoder to motor.
2. Connecting the encoder to the controller.
3. Receiving signal from the encoder.
4. Processing the encoded data.
5. Using feedback to control system

Task 4:

For motor control, STM32 will do the job. It has a great performance that can handle multiple motors with it's high processing power. STM32F103C8T6- bluepill is a great option.

For communication the ESP32 is ideal for IoT application and monitoring. It has a built-in Wi-Fi and Bluetooth system as well as a dedicated library for long range communication, so it is suitable as a transceiver.

The STM32 can handle Real-Time operations as well. The ARM Cortex-M cores process fast with minimal latency and RTOS support makes it possible to finish tasks within a specified time frame.

For sensor integration the Arduino Mega 2560 can be used. It supports different sensor libraries, it is easy to use and also the GPIO pins can be easily used for multiple sensors.

The ESP32 is extremely power efficient so it can extend battery life. It has 5 low power modes allowing it to conserve energy when it is not being used.

Task 5:

For an outdoor environment in Mars, the Robot needs to process and calculate several forms of data which cannot be done with only one sensor. And for a few tasks it needs backup sensors so if one sensor fails to provide correct information, it

can still work properly, Thus, it'll need a combination of sensors for its motion planning. The sensors I'll choose are:

1. Lidar:
   This sensor will help the rover measure distances between it and other objects. The data can be collected and processed to create a 3D environment of the scenario around the robot. It will allow the robot to detect obstacles in almost real time even in low light. The robot will be able to navigate its path safely with this.
2. Odometer:
   An odometer will measure the distance that the robot has traveled. It may be integrated into the robot's rotatory part like wheels.
3. IMU:
   IMU is an Inertial Measurement Unit. It will control the robot's speed and movement. For correct orientation, acceleration, estimating position it is important as sometimes it might not be possible to rely on outside information for this.

Choosing multiple sensors provides a backup source of information in case any sensor fails at any time. Also, these sensors perform best in different environments so the robot will be able to navigate its way no matter what outside condition it is in. And in the best conditions, the robot will be able to merge the data from all the sensors to have an even perfect sense of atmosphere allowing it to make better decisions.

Logical Part

Task 1A:

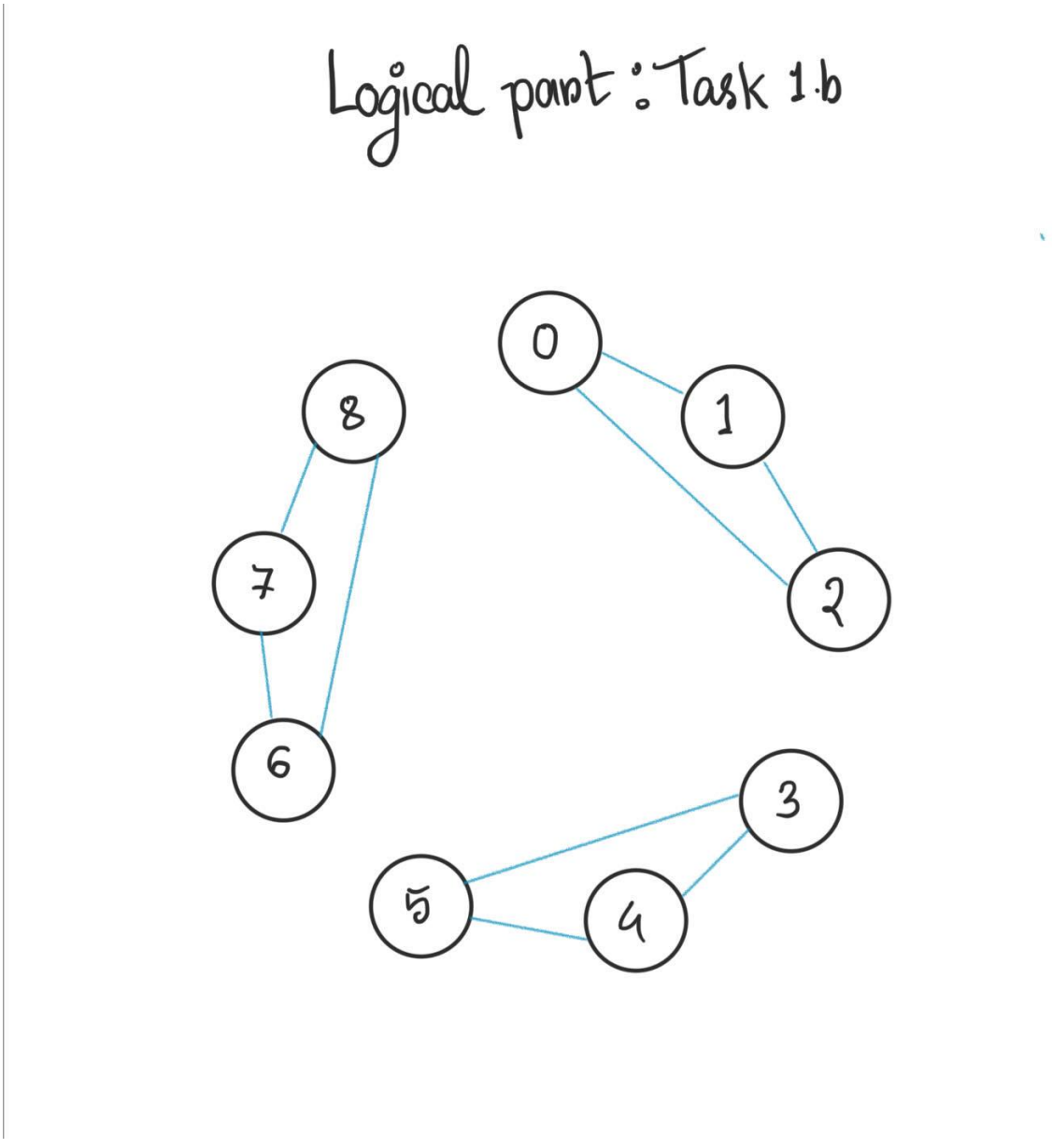
code link: https://github.com/Fa-riya/Altair-Software-Subteam-Selection/blob/main/210041103_Fariya%20Ahmed_week%201_Logical_Task%201_A.cpp

I used Depth-First Search Traversal method to traverse the graph in this code. First, I created an adjacency list of the graph using an array of integer vectors named edges. The vector at a particular index contains the other vertices that specific vertex is connected to.

Then I created a bool array named visited_vertex which holds true or false depending on whether that vertex has been visited or not. At first, I set all values to false. Then started dfs traversal to visit these vertices starting with vertex 0 as it is the first vertex in the edge list. After setting visited_vertex of that index to true, the function then traverses its adjacency list to mark the adjacent vertices as visited. As it is a recursive call, the function keeps getting called until it finds a vertex that has been visited before or if all the neighboring vertices of a certain vertex has been traversed through recursion. Then the function returns and continues traversing the adjacency list of the previous vertex. If any vertex is not connected anyhow to the first vertex ,then the algorithm won't mark it as visited. After the traversal ends, the function returns.

In the path function I called the dfs function. After traversal the path function then checks if all values in the visited_vertex have turned to true. If yes that means all the vertices can be visited so the function returns true. Otherwise, there is at least one vertex that cannot be visited so the function returns false. Finally, I printed this value as answer.

Task 1B: Graph-



Task 2:

1. Dijkstra Algorithm:

Dijkstra's algorithm is a shortest path algorithm which works by repeatedly selecting the vertex with the minimum distance from the source. It is used in various fields such as navigation systems, network routing, GPS etc to find shortest path in maps or telephone networks.
The time complexity of this algorithm is O ((V+E) log V) where E is the number of edges and V is the number of vertices. The Space complexity for this algorithm is O(V).

- Highly efficient for small or medium graphs which have fewer edges. However, it's not recommended for larger graphs
- It works with graphs with non-negative edge weights, it does not work properly if there are negative edges in the graph.
- It is the best shortest path algorithm for graphs with weighted edges.

Pseudocode for Dijkstra algorithm:

```
Dijkstra (Graph , Source)

Create vertex set Q

For each vertex V in the graph

    Distance[V] = infinity

    Add V to Q
```

```
Distance[Source] =0
While Q is not empty
    u = Extract-min[Q]
    for each neighbor V of u
        Relax(u,V)
```

2. Bellman Ford Algorithm:

Bellman Ford Algorithm is another shortest path algorithm which can work with negative edges. It first overestimates the length of path to all vertices, then iterates to relax those estimates by finding new paths that are shorter than the paths found or estimated before. It can work on graphs with negative edges which is why it is more versatile.
The time complexity for this algorithm is O(VE) however the best case complexity is O(E). The space complexity is O(V).

- Can handle negative edge graphs and detect negative cycles, can be used when Dijkstra fails.
- It is slower and requires more iterations.
- Good for small or medium graphs but inefficient for larger graphs because of the time complexity.

Pseudocode of Bellman Ford:

```
Bellman Ford (G, V, E, S)
For each vertex v
    Distance[v] = infinity
Distance [Source]=0
For i=1 to v-1
  For each edge (u,v) belongs to G
    Relax (u,v,w)
For each edge (u,v) belongs to G
  If (distance[u] + w(u,v) < distance[v] )
    Return presence of negative edge
Return distance
```

3. Floyd-Warshall Algorithm:

It is an all-pair SP algorithm that works on graphs with non-negative and negative edges. The time complexity for this algorithm is O(V^3) and space complexity is O(V^2).

- Works for graphs with both positive and negative weighted edges.
- Inefficient for larger graphs.
- Consumes large amount of memory because of the 3D matrix created to track paths between all pairs of vertices.

- Great for ASPS.
- Can be used to check whether an undirected graph is bipartite.

Code snippet of Floyd-Warshall algorithm:

```
#include <bits/stdc++.h>
using namespace std;
#define V 4
#define INF INT_MAX
void floydwarshall( int distance[][V]){
   int I, j, k;
   for( k=0;k < V ; k++){
      for( i=0; i < V; i++){
         for( j=0; j < V; j++){
            if (dist[i][j] > (dist[i][k] + dist[k][j])  && (dist[k][j] !=INF
               && dist[i][k] != INF))
             dist[i][j] = dist[i][k] + dist[k][j];
          }
      }
   }
}
```

4. A* Algorithm:

A* algorithm is like a smarter version of the Dijkstra algorithm which can work on graphs with negative paths. However, it might not always produce the shortest path. It is commonly used in various applications, games, robotics and GPS navigation systems. The time complexity for A* is O(E) and space complexity is O(V).

- Works on both unweighted and weighted graphs with positive and negative edges.
- Highly efficient, outperforms other algorithms.
- Great for larger graphs.
- May not always find the shortest path.

Pseudocode of A* algorithm:

A* (Graph, Edges, Vertices)

unvisited_list={start}

visited_list= {}

distance(start)= 0

Est_distance(start)= heuristic_function (start, end)

Total_distance(start)= distance(start) + Est_distance(start)

while unvisited_list is not empty

```
least= Node that has the least total_distance,on top of unvisited_list
if least == end
        return
remove least from unvisited_list and add to visited_list
for each vertex v in adjacent_vertices[least]
         if v is in visited_list
              continue
        cost = distance(least) + distance_between(least, v)
        if v is in unvisited_list and cost< distance(v)
             remove v from unvisited_list
        if v is in visited_list and cost< distance(v)
             remove v from visited_list
        if v is not in unvisited_list and visited_list
             add v to unvisited_list
             distance(v)= cost
             Est_distance(v)= heuristic_function (v, end)
             Total_distance(v)= distance(v) + Est_distance(v)
```

To summarize, if there is one source and one destination then it is best to use A* algorithm. If there are multiple destinations,

Dijkstra is great for graphs with non-negative edges which are weighted. But for negative edges, Bellman Ford is used in this case. For ASPS (path between every pair of nodes) it is best to use Floyd-Warshall algorithm.

Microcontroller Part

Task 1:

Tinkercad link: https://www.tinkercad.com/things/2IaUWSEYpZU-exquisite-uusam/editel?sharecode=FCO3kH3Q7FwdN-mc60ousqjF7I0Dt0VJeqwxiTw7i0Q

Task description:

Using two Arduino Uno, I used I2C to communicate between the boards. I used Wire.h library for this. During transmission the strings are converted into bytes.

The master firsts sends data to the slave “Hi from Master” which is printed on the serial monitor on the slave Arduino. Then master sends a request to slave to send “Successful”. Master receives the message and prints it on the serial monitor.

The slave Arduino first receives “Hi from Master” without any request. The data is printed on the serial monitor after being received by slave. Then the master requests data from slave and slave sends the message “Successful” through .

Problems I faced during the task:

1. As this was my first time using Arduino, I had to first learn the basic setups
2. During running simulation, Tinkercad was showing “Invalid Header file” error even though all the header files were valid. This was due to an error in function parameters which took me one hour to resolve.
3. Transmitted datatype was byte so I had to convert my message into bytes and after receiving reconvert them into a string to print on the serial monitor.

Task 2:

Tinkercad link: https://www.tinkercad.com/things/4fTwfquDbkJ-terrific-leelo/editel?sharecode=WvOHSF1bZYaM96U6i7WozbBMbwyzf-4_l9K3_T-KQLs

Task description:

Using an Arduino Uno and a DC motor with encoder I controlled the motor with the Arduino and printed the speed in rpm in the serial monitor. To calculate speed in rpm I took the position of the motor every 1 second and the time elapsed in simulation using millis() function. Then I printed the result in the serial monitor.

Problems I faced during the task:

1. I wanted to calculate the instantaneous speed in rpm however an error occurred which I couldn’t debug ,

printing unrealistic values. So I printed the average speed in rpm as the simulation ran

2. The rpm is shown in integer format even though the calculations are done in float format. This was done to simplify the look. But there is a slight error/deviation from the original value which can be ignored.

Github repository link: https://github.com/Fariya/Altair-Software-Subteam-Selection