BME Medical Robotics Report

Maria Michailidou: Team A

As a primary contributor to the testing phase, my responsibilities included ensuring all code components integrated smoothly with each other, organizing and leading meetings to keep the team informed and aligned and testing basic functionalities, error handling, and creating robust tests to ensure system reliability for future development.

All of my personal contributions to the project can be found here: BME-Medical-Robotics-Team-A-Testing

Problems Encountered and Solutions

Problem 1: Initial integration and planning.

- Charalampos had gone ahead and established how the inverse and forward kinematics were solved for the system using Matlab, while Nikos had done similar work using the Robotics System Toolbox and establishing MQTT communication.
    - A mixed development approach was established as a solution, where some of the basic kinematic functions from Charalampos were used, while Nikos developed the 4 Degree of Freedom model and the MQTT communication.

Problem 2: Team Organization

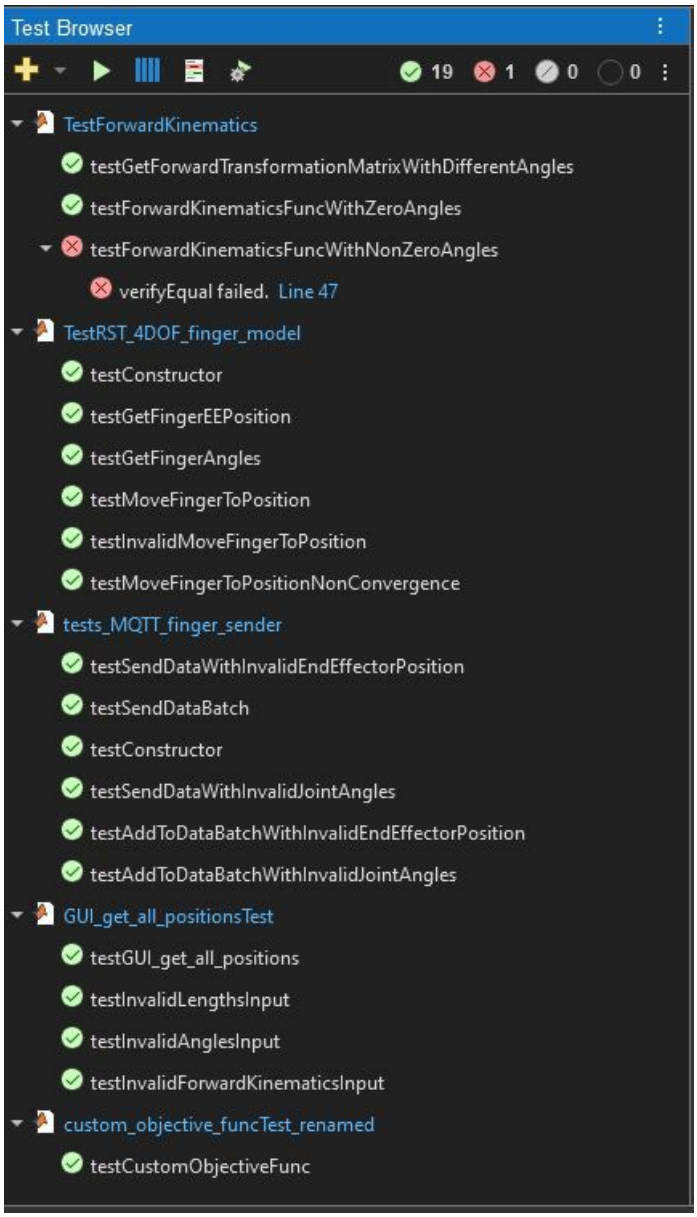
- Due to a busy schedule because of examinations, work, and other factors which lead to different productive hours for each individual, I went ahead and organized two general group meetings of the entire team, one meeting for the mathematical description subgroup comprised of Euanthia, Charalampos, and Chrysanthi, and two other meetings of the programming subgroup comprised of myself, Magdalini, Nikos and Charalampos. Some efforts to reach team B were made to see if I could perhaps provide testing for some of their code, or at least check the MQTT connection between the two applications but until now, unfortunately this wasn't possible.
    - In order to solve that, I scheduled regular meetings and written updates, plus special communication channels for our group on Messenger and Discord, where individuals could easily plan and communicate for the project, while also sometimes providing written notes about the meetings, meant for individuals who missed them to catch up to speed. Also I went ahead and created the team's Github repository, and collaborated to keep it updated.

Problem 3: Writing tests for existing code

- Trying to connect code from a lot of different people together was a challenge, so in order to familiarize myself with the functions used in the application and make the entire project easier to be adapted and upgraded in the future, I created tests cases for the most critical functions.
    - Specifically, I created specific test cases for error handling, incorrect inputs and validation of proper output for the system. In total **24** tests cases were created, **19** which were running normally, and **5** test cases which failed or errored during execution. The failed test cases are included in the current code base but not in the documentation, in case anyone wishes to debug the system further, at a later date.

System Testing

Here, a short description of all the test files is given, along with a brief explanation on what each test case is testing for. For a closer look at the test functions, appropriate documentation is available in the github repository for my personal contribution.



List of Test Cases per file:

TestRST_4DOF_finger_model

- **testConstructor**: Verifies the correct construction of the 4DOF finger model by checking that the properties `P_ph`, `M_ph`, and `D_ph` are set to `0.05`, and ensuring that `fingerRobotModel` is an instance of `rigidBodyTree` and `ikSolver` is an instance of `generalizedInverseKinematics`.
- **testGetFingerEEPosition**: Tests the `getFingerEEPosition` method by moving the finger to a new position `[0.2, 0.2, 0.2]` and verifying that the end-effector position matches this new position within a specified tolerance (`1e-6`).
- **testGetFingerAngles**: Checks the initialization of joint angles by verifying that the `getFingerAngles` method returns a vector of zeros `[0; 0; 0; 0]`.
- **testMoveFingerToPosition**: Validates the `moveFingerToPosition` method by moving the finger to the position `[0.1, 0.05, 0]` and ensuring that the end-effector position matches this target position within a specified tolerance (`1e-6`).
- **testInvalidMoveFingerToPosition**: Ensures proper error handling for invalid positions by attempting to move the finger to an invalid position `[0.9, 0.9, 5]`. The test verifies that an error with the identifier `MATLAB:invalidPosition` is thrown.
- **testMoveFingerToPositionNonConvergence**: Simulates non-convergence scenarios by attempting to move the finger to an invalid position `[0.9, 0.9, 5]` with the maximum iterations set to 100. The test verifies that the inverse kinematics solver reaches the maximum iterations, confirming non-convergence.

TestForwardKinematics

- **testGetForwardTransformationMatrixWithDifferentAngles**: Verifies the function `get_forward_transformation_matrix` with different angles, ensuring the output is a 4x4 matrix and the last row is `[0, 0, 0, 1]`.
- **testForwardKinematicsFuncWithZeroAngles**: Tests the `forward_kinematics_func` function with zero joint angles. It checks that the function returns a 3x1 position vector and verifies that the end-effector position is `[L1+L2+L3; 0; 0]` when all angles are zero.
- **testForwardKinematicsFuncWithNonZeroAngles**: Checks the `forward_kinematics_func` function with non-zero joint angles, ensuring the function returns a 3x1 position vector. The test calculates the expected end-effector position for given angles and link lengths, verifying that the function's output matches this expected position within a specified tolerance (`1e-6`).

tests_MQTT_finger_sender

- **testSendDataWithInvalidEndEffectorPosition**: This test verifies the `sendData` method's ability to handle invalid `endEffectorPosition` input. It checks if an appropriate error is thrown and that the error identifier contains `MATLAB:invalidType`.
- **testSendDataBatch**: This test validates the `sendDataBatch` method by adding a valid set of data to the batch and ensuring that calling `sendDataBatch` does not produce any warnings.
- **testConstructor**: This test checks the constructor of the `MQTT_finger_sender` class. It verifies that the properties `clientID`, `brokerAddresses_OverUnTCP`, `port_OverUnTCP`, `topic`, `mqClient`, and `dataBatch` are correctly initialized.
- **testSendDataWithInvalidJointAngles**: This test ensures that the `sendData` method handles invalid `jointAngles` input properly. It verifies that the method can accept invalid input without causing unexpected behavior, assuming `sendData` is void and does not return an output.
- **testAddToDataBatchWithInvalidEndEffectorPosition**: This test verifies that the `addToDataBatch` method handles invalid `endEffectorPosition` input by checking if it throws the expected `MATLAB:invalidType` error.
- **testAddToDataBatchWithInvalidJointAngles**: This test ensures that the `addToDataBatch` method handles invalid `jointAngles` input correctly by verifying that it throws the expected `MATLAB:invalidType` error.

custom_objective_funcTest_renamed

- **Test 1: Zero Desired Position**:
  - An input is given for a zero desired position (`x_des = L1 + L2 + L3`, `y_des = 0`, `z_des = 0`) and zero effective angle (`theta_eff = 0`). The expected output is a zero error vector.
  - Checking to verify if the output error has the correct size (`[3, 1]`) and is equal to the expected zero error within a tolerance of `1e-6`.
- **Test 2: Non-Zero Desired Position**:
  - An input is set for a non-zero desired position (`x_des = 2`, `y_des = 1`, `z_des = 0`) and a zero effective angle (`theta_eff = 0`).
  - Here the expected output is a on-zero error vector.
  - To verify the result if correct we check that the output error is not equal to the expected zero error vector.

Small note:

The tests confirm the correctness of the `custom_objective_func` objective function under different input scenarios. The function accurately computes the error between the desired end effector position and the actual position based on the given joint angles and finger segment lengths. The validation ensures the reliability of the objective function for use in our system.

TestAnalyticalandNumericalSolver

- **Analytical Solver**: Evaluates the analytical inverse kinematics solver's performance and defines input parameters to call the GUI-based analytical solver to compute joint angles for a given target end effector position.
- **Numerical Solver**: Used to assess the numerical inverse kinematics solver's accuracy by initializing the `RST_4DOF_finger_model` with specified finger segment lengths and call the numerical solver to move the finger to the target position and then, retrieve the end effector position and joint angles.

Here, it should be noted that this isn't an actual numerical solver that Nikos has implemented in the RST_4DOF function, but an iterative one, and the naming of the solver is a mishap on my part, one that Madga and Nikos were kind enough to point out to me when they reviewed my code. The numerical name of the function, was kept, but the clarification is important in this report to avoid any confusion.

- **Analytical Solver Results**:
  - Analyze the joint angles computed by the analytical solver.
  - Print the calculated values of `theta1`, `theta2`, `theta3`, and `theta4` in radians.
- **Numerical Solver Results**:
  - Examines the performance of the numerical solver.
  - Prints the end effector position and joint angles obtained from the numerical solver.
  - Needs around 50 iterations for the solver to produce accurate results.

TestRST_4DOF_finger_model.m | tests_MQTT_finger_sender.m | forward_kinematics_func.m | MQTT_finger_sender.m | TestAnalyticalandNumericalSolver.m

/MATLAB Drive/project2 (1)/project2/TestAnalyticalandNumericalSolver.m

```
% Define input parameters
lengths = [10, 10, 10]; % Lengths of the finger segments (e.g., in cm)
targetPosition = [15, 5, 10]; % Target end-effector position (e.g., in cm)

% Run analytical solver
try
    [theta1_analytical, theta2_analytical, theta3_analytical, theta4_analytical, angles_info_analytical] = ...
        GUI_inverse_kinematics(lengths, targetPosition(1), targetPosition(2), targetPosition(3), 0);

    % Check if the angles are valid
    if angles_info_analytical.message == 'Invalid angles'
        error('Analytical solver produced invalid angles.');
    end

    % Display analytical solver results
    fprintf('Analytical Solver Results:\n');
    fprintf('Theta1: %.4f rad\n', theta1_analytical);
    fprintf('Theta2: %.4f rad\n', theta2_analytical);
    fprintf('Theta3: %.4f rad\n', theta3_analytical);
    fprintf('Theta4: %.4f rad\n', theta4_analytical);
catch ME
```

Command Window

New to MATLAB? See resources for Getting Started.

```
>> TestAnalyticalandNumericalSolver
Invalid value detected in theta_4: -1.9985 (deg: -114.5064)
Valid range for theta_4: [-31, 97.5]
Error in analytical solver: Scalar structure required for this assignment.

Numerical Solver Results:
Theta1: 0.3218 rad
Theta2: -0.1745 rad
Theta3: 0.8849 rad
Theta4: 0.7854 rad
```

Small note:

The comparison between the analytical and numerical inverse kinematics solvers provides insights like how the analytical solver offers precise solutions, but is limited by complex mathematical expressions and assumptions. On the other hand, the numerical solver, even though it is computationally needy, it offers flexibility and robustness.