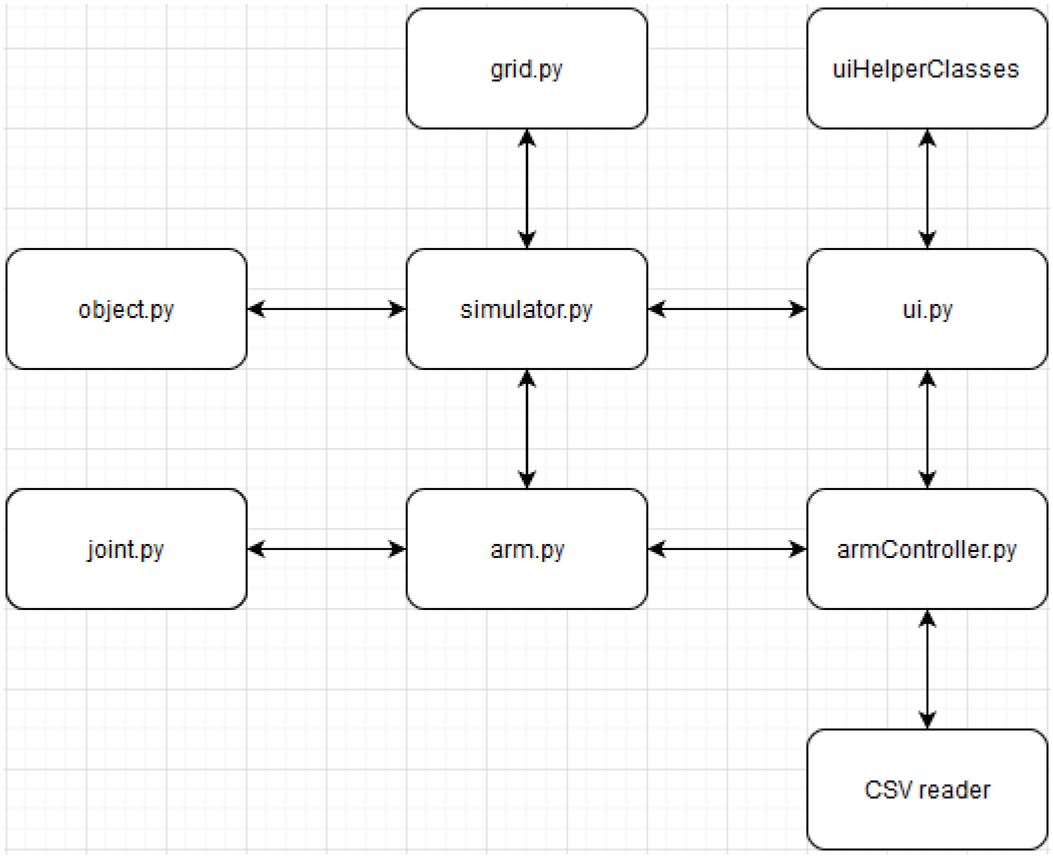
CS-A1121 Technical specifications

Linus Sundström, 432021, Automaatio- ja Informaatioteknologia, starting year 2014

Dated 18.2.2020

Program structure

The project will use the following structure (changes are of course likely later on):



Classes

grid.py

Implements coordinates and a coordinate grid housing the robot arm and objects.
Contains at least the following methods:

- Class position
    - Stores position x and y as floating point numbers
    - addPosition(self, pos1, pos2)
        - returns the sum of the two positions
- checkPosition(self, position)
    - checks whether specified position is occupied.

object.py

models the internals of objects the arm can grab. Contains at least the following methods:

- getGridPosition(self)
    - returns current coordinates
- moveObject(self, newPosition)
    - moves object to specified position

arm.py

The robot arm will be implemented as a linked list. While this is not a particularly effective data structure in python it should be sufficient as long as one arm does not have hundreds of joints. More on this under data structures.

- rotateJoint(self, joint)
    - rotates a joint specified by its index. The root joint has index 0, and further joints increment upwards.
- getArmPosition
    - returns the position of the end of the arm
- grab(self)
    - attempts to grab an object with the arm. Fails if no object in range)
- release(self)
    - releases a held object. Fails if area occupied.
- SetArmPosition(self, position)
    - This may be implemented if time allows. Sets the end of the arm to a specified position.

-

joint.py
contains specifications for individual joints. Each joint is specified by an angle and a length (polar coordinates if you will).

- getAngle(self)
  - returns joint angle
- getLength(self)
  - returns joint length
- setAngle(self, angle)
  - sets the joint to specified angle. Another class may be added later to handle the physics / animation involved
- getJointPosition
  - returns the relative position of the end of the joint.

simulator.py
Handles simulation ticks and makes sure operation is realtime. Something like the brains of the program. I'm not too sure exactly on how this class will end up looking, I expect tht will have to be worked out during development.

- executeTick(self)
  - executes a tick of the simulation, and engages necessary actions in UI, requested changes to joints and so on.

armController.py
Handles requests passed from UI and passes them on to relevant parts of the program. Acts to some extent as an API between front and backend. I'm not sure if this will prove necessary so it might be scrapped down the line.

csvReader.py
parses a csv of predetermined commands and loads them to simulator.

ui.py
Contains all necessities for visuals using pyQT. This will most likely end up having a lot of helper classes, but I'm not to well versed in pyQT as of yet to be able to anticipate what these might be.

User case

Lets have an imaginary situation where a robotic arm is used to unload some cargo daily. Every day a set number of boxes are delivered to a specific spot. These need to be moved somewhere else in the warehouse.

The user would wan't to create a predetermined program to handle this with as little user interaction as possible. As such the user has a csv that he loads into the program daily. This csv triggers control action on specified timestamps in the simulation. The arm is automatically manouvered over each box in turn, grabs it, moves it somewhere else and then drops it. Finally it turns out that one box wasn't in the predetermined location. The operator manually instructs the arm to move over to the last box, instructs it to pick it up, and finally move it to its new location. The operator here would most likely much prefer to give the arm a location than set all joint angles individually.

Lets have a look under the hood. The user first uploads a csv file. This is parsed by the csvParser to a list of commands with timestamps. Next, the simulator runs the arm simulation, and when the timestamp matches one from the csv, triggers that control action, lets say the order is to change the angle of joint 2.

The method setAngle is invoked. This trigger results in the angle changing according to some physics in the background. During all of this, the arm joint new positions are constantly recalculated and rendered on the screen. As the box is picked up and moved, its position is also constantly updated.

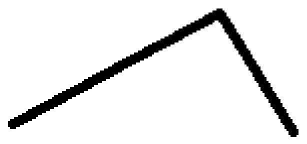
Control actions are made asynchronously, meaning that multiple control actions may take place at once. As such, many joints may be in motion at any one time.

Algorithms

The arm will be modelled as a linked list. The main reason for this is that it allows for some sneaky methods when calculating arm positions and angles using recursive method calls.

The method getAngle will take the angle stored in the joint, and then recursively add the angle of all previous joints to it. The result will be the angle in relation to the grid.

Example: an arm consists of two joints. The first (inner) joint is set to 60^o, the second is set to 90^o. The arm will then look like this:



By adding the first angle to the second angle, we end up at at 150^o, which is the angle of joint 2 in relation to the grid. This works for an arbitrary number of joints.

Using similar logic, calculating the position of each joint will be done by calculating its x and y coordinates in relation to (0,0) using its angle in relation to the grid and Pythagoras theorem, and then adding this position to the position of the previous joint.

Data structures

The arm will be modelled as a linked list. This is mainly because it allows some trickery with recursive functions explained above. I do not believe a linked list in general is the best solution for this kind of a problem, but it seems somewhat applicable and I've been looking for an excuse to use one in a project. The linked list will be manually implemented in arm.py, meaning existing libraries for linked lists will not be used.

A linked list is a data structure where each element of the list contains a link to the previous element in the list. The data structure is not the most efficient in python, as the list objects end up nested in each other, and as such should not be used for any kind of large data. For his application I deem it acceptable however as I don't foresee anyone wanting a robotic arm with hundreds or thousands of joints.

Other data structures that will be used is List or dictionary for the stored controller actions. I'm also sure that further basic data structures will prove necessary down the road.

Libraries

- The project will use PyQT5 for showing the UI.
- Unit tests will be made using unittest
- timestamps will be handled with datetime

Further Library requirements may become apparent over time.

Timetable

My goal is to be finished by 30.4.2020. This leaves some leeway for handling unforeseen problems. I plan to finish the exercise rounds before starting the project, which in practise this leaves just under 2 months for finishing the project. I will preliminarily allocate 2 weeks for each increment specified in the project plan, excluding increment 5. Assuming that I start work in 2 weeks (3.3.2020), the deadlines are as follows:

- Increment 1: 17.3
- Increment 2: 31.3
- Increment 3: 14.4
- Increment 4: 28.4
- Increment 5: any remaining time

Unit testing

The project will be done using Test Driven Development (TDD). As such unit tests a method will be written before the implementation of the method itself. I have been looking for an opportunity to try this approach and this seems like the time to.

TDD is often criticised for being to tedious in the real world, so I'm curious to see what my experience is. If it turns out that it is severely hindering progress, I will forfeit the approach.

For each method I will implement a unit test testing its core functionality, and simple error situations. I will not actively pursue 100% test coverage, but rather attempt to keep a sane approach to testing. When encountering bugs, a test will be written that would have caught that bug as part of solving it.

For instance, tests for the method arm.rotateJoint would include rotating the first, last and some middle joint in the system, as well as attempting to rotate a joint with a higher index than any in the arm. In the last case, an exception should be raised while allowing operation to continue

Unit testing will be supported by more extensive end to end testing done at the end of increments specified in the project plan.

Useful sources:

Course material in A+:
https://plus.cs.aalto.fi/y2/2020/

Python.org pyQT tutorials
https://wiki.python.org/moin/PyQt/Tutorials

Python standard library documentation
https://docs.python.org/3/library/

All the heroes on Stack Exchange
https://stackexchange.com/

This list will most likely grow during the project.