Task-based control

Lab 1: visual servoing

1 Goals

The goal of this lab is to observe the image-space and 3D-space behaviors that are induced when using various visual features.
The ones that are considered are:

- Cartesian coordinates of image points
- Polar coordinates of image points
- $2\frac{1}{2}$ visual servo
- 3D translation (either ${}^{c}\mathbf{t}_{o}$ or ${}^{c*}\mathbf{t}_{c}$)
- 3D rotation (either ${}^{c}\mathbf{R}_{c*}$ or ${}^{c*}\mathbf{R}_{c}$)

Several initial / desired poses are available, and some features that lead to a good behavior in some cases may lead to a undesired one in other cases.

2 Running the lab

The lab should be done on Ubuntu (either P-robotics room or the Virtual Machine).
Dependencies: ['log2plot'](https://github.com/oKermorgant/log2plot), ViSP

2.1 Download and setup

You can download the lab from github. Navigate to the folder you want to place it and call:

```
git clone https://github.com/oKermorgant/ecn_visualservo
cd ecn_visualservo
ideconf
```

This will download the git repository and prepare everything for the compilation.
You can then open QtCreator and load the `CMakeLists.txt` file of this project.

2.2 Lab files

Here are the files for this project:

```
ecn_visualservo
├── include
│   ├── feature_stack.h
│   └── simulator.h
├── src
│   ├── feature_stack.cpp
│   └── simulator.cpp
├── CMakeLists.txt
├── config.yaml
└── main.cpp
```

Only two files are to be modified:

- `main.cpp`: to build the feature stack and control law
- `config.yaml` : configuration file to change which features you want to use

Feel free to have a look at the other files to see how everything is done under the hood.

The simulation runs with the green `Run` button on the bottom left corner of QtCreator.
Of course, as no visual feature are used initially, the camera does not move.

3 Implementation

Only a few things are to be implemented in C++ to get a valid simulation: adding features to the stack, and computing the control law.

3.1 Implement the choice of the features

In the `main.cpp`file, a few design variables are loaded from the configuration (`useXY, usePolar` , etc.).

They are used to tell which features we want to use for this simulation. All these features are already available in the ViSP library, and we just have to register them inside a custom feature stack.

The first part of this lab is hence to add features to the stack depending on the configuration variables.
We assume that:

- If `useXY` is `true`, we want to stack all points from the simulation with Cartesian coordinates.
- If `usePolar` is `true`, we want to stack all points from the simulation with polar coordinates.
- If `use2Half` is `true`, we want to stack:
  - the center of gravity of the simulation with Cartesian coordinates
  - the center of gravity of the simulation with depth coordinate

  - a 3D rotation (either ${}^{c}\mathbf{R}_{c*}$ or ${}^{c*}\mathbf{R}_{c}$).
    Use `stack.setRotation3D("cdRc")` or `stack.setRotation3D("cRcd")`

- The 3D translation and rotation are automatically used if they have a valid value in the configuration file:
  - Translation can be `"cdTc"` (${}^{c*}\mathbf{t}_{c}$) or `"cTo"` (${}^{c}\mathbf{t}_{o}$)
  - Rotation can be `"cdRc"` (${}^{c*}\mathbf{R}_{c}$) or `"cRcd"` (${}^{c}\mathbf{R}_{c*}$)

Documentation on the `FeatureStack` class is available in Appendix A.1. It describes how to add new features to the stack.

From this point, the stack is able to update the features from a given pose ${}^{c}\mathbf{M}_{o}$ and give the current features and their interaction matrix.

3.2 Implement the control law

In the main control loop, you should retrieve the current feature values `s` from the stack, and their corresponding interaction matrix `L`.
Then compute `v` as the classical control $\mathbf{v} = -\lambda \mathbf{L}^{+}(\mathbf{s} - \mathbf{s}^{*})$.

This velocity twist will be sent to the simulation and a new pose will be computed.
Some gains and parameters are loaded from the configuration: `lambda, err_min`, etc.
Observe what is their influence on the simulation.

4 Combining features

Test that your simulation works for all types of features.

Initially it starts from a position tagged as `cMo_t` in the configuration file, while the desired position is tagged as `cdMo`.
Try various combinations of starting / desired poses, together with various choices on the feature set. Typically:

- Large translation / rotation error, using XY or Polar
- 180 error using XY
- Very close poses (`cMo_visi`)

What happens if many features are used at the same time, such as Cartesian + polar coordinates, or 2D + 3D features?

4.1 On Z-estimation

The `z_estim` keyword of the configuration file allows changing how the Z-depth is estimated in 2D visual servoing.
Compare the behavior when using the current depth (negative `z_estim`), the one at desired position (set `z_estim` to 0) or an arbitrary constant estimation (positive `z_estim`).

4.2 Output files

All simulations are recorded so you can easily compare them. They are placed in the folder:

`ecn_visualservo/results/<start>-<end>/<feature set>/`

All files begin with the Z-estimation method and the lambda gain in order to compare the same setup with various gains / estimation methods.

A Class helpers

Below are details the useful methods of the main classes of the project. Other methods have an explicit name and can be listed from auto-completion.

A.1 The `FeatureStack` class

This class is instanciated under the variable `stack` in the main function. It has the following methods:

Adding feature to the stack

- `void addFeaturePoint(vpPoint P, PointDescriptor descriptor)` add a point to the stack, where `descriptor` can be:
  - `PointDescriptor::XY`
  - `PointDescriptor::Polar`
  - `PointDescriptor::Depth`
- `void setTranslation3D(std::string descriptor)`: adds the 3D translation feature (cTo, cdTc or none)
- `void setRotation3D(std::string descriptor)`: adds the 3D rotation feature (cRcd, cdRc or none)
- `void updateFeatures(vpHomogeneousMatrix cMo)`: updates the features from the passed `cMo`

Stack information

- `void summary()`: prints a summary of the added features and their dimensions
- `vpColVector s()`: returns the current value of the feature vector
- `vpColVector sd()`: returns the desired value of the feature vector
- `vpMatrix L()`: returns the current interaction matrix

A.2 The `Simulator` class

This class is instanciated under the variable `sim`. It has the following methods:

- `std::vector<vpPoint> observedPoints()`: returns the list of considered 3D points in the simulation
- `vpPoint cog()`: returns the center of gravity as a 3D point

For example, to add all points of the simulation as XY features, plus the depth of the CoG, just write:

```
for(auto P: sim.observedPoints())
    stack.addFeaturePoint(P, PointDescriptor::XY);

stack.addFeaturePoint(sim.cog(), PointDescriptor::Depth);
```