Lab 7: SLAM Visualisation

MCHA4400

Semester 2 2023

Introduction

In this lab, you will develop a visualisation of the marginal distributions of camera and landmark positions that will serve as a foundation for the landmark SLAM rendering aspects of Assignment 1.

LLM Tip

You are strongly encouraged to explore the use of Large Language Models (LLMs), such as GPT, PaLM or LLaMa, to assist in completing this activity. Bots with some of these LLMs are available to use via the UoN Mechatronics Slack team, which you can find a link to from Canvas. If you are are unsure how to make the best use of these tools, or are not getting good results, please ask your lab demonstrator for advice.

Before beginning this lab, merge your earlier work into the Lab 7 template as follows:

- Merge the following in `src/Camera.cpp` from Lab 3:
  - `Camera::calibrate`
  - `Camera::vectorToPixel`
  - `Camera::worldToVector`
  - `Camera::pixelToVector`
  - `Camera::isVectorWithinFOV`
  - `Camera::calcFieldOfView`
  - `ChessboardImage::ChessboardImage`
  - `ChessboardImage::drawBox`
- Merge the following in `src/Gaussian.hpp` from Lab 6:
  - `Gaussian::marginal`
  - `Gaussian::conditional`
  - `Gaussian::log` (all three variants)

Be mindful that some other changes have been made to these classes, so double check before overwriting any code (or use your favourite merge tool).

1. Rotations

In this task you will create templated rotation functions, within `src/rotation.hpp`, to transform between a rotation matrix and its RPY Euler angle parameterisation.

Tip

Refer to the Week 3 slides on parameterisations of $\mathsf{SO}(3)$.

Tasks

a) Complete the templated function `rotx`, which returns $\mathbf{R}_x(\phi)$.

b) Complete the templated function `roty`, which returns $\mathbf{R}_y(\theta)$.

c) Complete the templated function `rotz`, which returns $\mathbf{R}_z(\psi)$.

d) Complete the templated function `rpy2rot`, which returns $\mathbf{R}(\boldsymbol{\Theta}) = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)$ from the vector of Euler angles $\boldsymbol{\Theta} = \begin{bmatrix}\phi & \theta & \psi\end{bmatrix}^\mathsf{T}$.

e) Complete the templated function `rot2rpy`, which returns a vector of Euler angles $\boldsymbol{\Theta} = \begin{bmatrix}\phi & \theta & \psi\end{bmatrix}^\mathsf{T}$ for the given $\mathbf{R} \in \mathsf{SO}(3)$.

f) Build the application and ensure the unit tests in `test/src/rotation.cpp` pass.

Terminal

```
nerd@basement:~/MCHA4400/lab7$ cmake -G Ninja -B build && cd build
nerd@basement:~/MCHA4400/lab7/build$ ninja
[Lots of failing unit tests, pay attention to the relevant ones only]
```

Tip

If the non-relevant unit tests are too distracting, you can temporarily move them out of the `test/src` directory and move them back in as needed for the following parts.

2. Landmarks to features

The world-to-pixel model we developed in Lab 3 maps the location of the world point $\mathbf{r}^n_{P/N}$ to its location in the image $\mathbf{r}^i_{Q/O}$. This was previously implemented with the help of `cv::projectPoints`; however, now we will need to compute the derivatives of this function with respect to the state variables in a SLAM solution.

We will consider the same rational radial distortion, decentering distortion and thin prism distortion models supported by OpenCV, and let $\boldsymbol{\theta}_\text{dist} \in \mathbb{R}^m$ be the distortion coefficients of the planar camera model,

$$\begin{bmatrix}u'\\v'\end{bmatrix} = \underbrace{c\begin{bmatrix}u\\v\end{bmatrix}}_{\text{radial distortion}} + \underbrace{\begin{bmatrix}2p_1uv + p_2(r^2+2u^2)\\p_1(r^2+2v^2)+2p_2uv\end{bmatrix}}_{\text{decentering distortion}} + \underbrace{\begin{bmatrix}s_1r^2+s_2r^4\\s_3r^2+s_4r^4\end{bmatrix}}_{\text{thin prism distortion}}, \tag{1}$$

where

$$r^2 = u^2+v^2, \qquad u = \frac{x}{z}, \qquad v = \frac{y}{z}, \qquad \mathbf{r}^c_{P/C} = \begin{bmatrix}x\\y\\z\end{bmatrix}. \tag{2}$$

The radial distortion coefficient is given by the following rational model:

$$c = \frac{1+\alpha}{1+\beta}, \tag{3}$$

where

$$\alpha = k_1 r^2 + k_2 r^4 + k_3 r^6, \tag{4}$$

$$\beta = k_4 r^2 + k_5 r^4 + k_6 r^6. \tag{5}$$

Using the above expressions, the mapping from world points with respect to C in camera in the camera basis $\{c\}$ by

$$\mathbf{r}^i_{Q/O} = \mathtt{v2p}\big(\mathbf{r}^c_{P/C}; \boldsymbol{\theta}\big) = \begin{bmatrix} f_x u' + c_x \\ f_y v' + c_y \end{bmatrix}, \tag{6}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\theta}_{\text{cam}}, \boldsymbol{\theta}_{\text{dist}}\}$[1] are the camera intrinsic parameters and $\boldsymbol{\theta}_{\text{cam}} = \{f_x, f_y, c_x, c_y\}$ are the nontrivial elements of the camera matrix

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

The world-to-pixel mapping is defined by composing a kinematic transformation from the world point $\mathbf{r}^n_{P/N}$ to the camera vector $\mathbf{r}^c_{P/C}$ with the vector to pixel mapping, which yields

$$\mathbf{r}^i_{Q/O} = \mathtt{v2p}\big((\mathbf{R}^n_c)^\mathsf{T}(\mathbf{r}^n_{P/N} - \mathbf{r}^n_{C/N}); \boldsymbol{\theta}\big) \tag{8a}$$

$$= \mathtt{w2p}\big(\mathbf{r}^n_{P/N}; \mathbf{T}^n_c, \boldsymbol{\theta}\big), \tag{8b}$$

where camera pose $\mathbf{r}^n_{C/N}$ and $\mathbf{R}^n_c$ are the camera position and orientation, respectively, which can be parameterised by $\mathbf{T}^n_c \in \mathsf{SE}(3)$.

For SLAM using point landmarks, we can consider the following state vector:

$$\mathbf{x} = \begin{bmatrix} \boldsymbol{\nu} \\ \boldsymbol{\eta} \\ \mathbf{r}^n_{1/N} \\ \mathbf{r}^n_{2/N} \\ \vdots \\ \mathbf{r}^n_{n_L/N} \end{bmatrix}, \tag{9}$$

where

$$\boldsymbol{\nu} = \begin{bmatrix} \mathbf{v}^b_{B/N} \\ \boldsymbol{\omega}^b_{B/N} \end{bmatrix}, \tag{10}$$

$$\boldsymbol{\eta} = \begin{bmatrix} \mathbf{r}^n_{B/N} \\ \boldsymbol{\Theta}^n_b \end{bmatrix}, \tag{11}$$

are the body-fixed velocities and world-fixed pose, respectively, $\mathbf{r}^n_{j/N}$ is the position of the j^{th} landmark and n_L is the number of landmarks in the map. The body orientation is parameterised by RPY Euler angles, $\boldsymbol{\Theta}^n_b = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^\mathsf{T}$ such that

$$\mathbf{R}^n_b = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi) = e^{\psi \mathbf{S}(\mathbf{e}_3)} e^{\theta \mathbf{S}(\mathbf{e}_2)} e^{\phi \mathbf{S}(\mathbf{e}_1)}. \tag{12}$$

[1]See the source code for `cvProjectPoints2Internal` for more details.

Let $\mathbf{h}_j(\mathbf{x}) \in \mathbb{R}^2$ denote the pixel coordinates of the feature corresponding to the j^{th} landmark as a function of the state $\mathbf{x}$, i.e.,

$$\mathbf{h}_j(\mathbf{x}) = \mathtt{w2p}(\mathbf{r}^n_{j/N}; \mathbf{T}^n_c, \boldsymbol{\theta}). \tag{13}$$

To propagate the uncertainties in the landmark position and camera pose through to uncertainty in the predicted feature location in an image using an affine transform, we will need to compute the Jacobian of $\mathbf{h}_j(\mathbf{x})$ with respect to $\mathbf{x}$. This has the following block structure:

$$\frac{\partial \mathbf{h}_j}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{0}_{2\times3} & \mathbf{0}_{2\times3} & \frac{\partial \mathbf{h}_j}{\partial \mathbf{r}^n_{B/N}} & \frac{\partial \mathbf{h}_j}{\partial \mathbf{\Theta}^n_b} & \mathbf{0}_{2\times3} & \cdots & \frac{\partial \mathbf{h}_j}{\partial \mathbf{r}^n_{j/N}} & \cdots & \mathbf{0}_{2\times3} \end{bmatrix} \in \mathbb{R}^{2\times(12+3n_L)}. \tag{14}$$

Expressions for the analytical derivatives of the non-zero blocks can be found in Appendix B.

Tasks

a) Complete the implementation of the `StateSLAM::predictFeature` template member function in `src/StateSLAM.h` and the member function in `src/StateSLAM.cpp`.

> **Info**
> `StateSLAM::predictFeature` is used within `StateSLAM::predictFeatureDensity`, which uses `Gaussian::transform` to propagate the state distribution through the world to pixel map.

These functions implement $\mathbf{h}_j(\mathbf{x})$ as the return value and the latter function also writes the Jacobian to `J`, which can be evaluated by one of the following methods:

i) Implement the function
`Eigen::Vector2d Camera::vectorToPixel(const Eigen::Vector3d & rPCc, Eigen::Matrix23d & J) const`
within `src/Camera.cpp` that returns $\mathbf{r}^i_{Q/O}$ and computes $\mathtt{J} = \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}}$ using the expressions for the analytical Jacobian given in Appendix A. Call this function within `StateSLAM::predictFeature` to obtain $\frac{\partial \mathbf{h}_j}{\partial \mathbf{r}^n_{j/N}}$ and then assemble the blocks of the Jacobian $\frac{\partial \mathbf{h}_j}{\partial \mathbf{x}}$ using the results given in Appendix B.

ii) Implement the `Camera::vectorToPixel` template member function within `src/Camera.h` that returns $\mathbf{r}^i_{Q/O}$, ensuring that all scalar values are of type given by the template parameter `Scalar`. Then, compute the Jacobian using automatic differentiation within `StateSLAM::predictFeature` to obtain $\frac{\partial \mathbf{h}_j}{\partial \mathbf{r}^n_{j/N}}$ and then assemble the blocks of the Jacobian $\frac{\partial \mathbf{h}_j}{\partial \mathbf{x}}$ using the results given in Appendix B.

iii) Implement the `Camera::vectorToPixel` template member function within within `src/Camera.h` that returns $\mathbf{r}^i_{Q/O}$, ensuring that all scalar values are of type given by the template parameter `Scalar`. Then, implement the `StateSLAM::predictFeature` template member function `src/StateSLAM.h` that implements $\mathbf{h}_j(\mathbf{x})$, ensuring that all scalar values are of type given by the template parameter `Scalar`. Finally, in the `StateSLAM::predictFeature` member function in `src/StateSLAM.cpp`, compute the Jacobian of the template function above using automatic differentation, directly yielding $\frac{\partial \mathbf{h}_j}{\partial \mathbf{x}}$.

Tip

To compute a Jacobian matrix using forward-mode autodifferentiation, see the `jacobian` function.

To compute a Jacobian matrix using reverse-mode autodifferentiation, we can compute each row of the Jacobian using the `gradient` function as follows, since autodiff doesn't provide a convenient reverse-mode Jacobian:

```
Eigen::VectorXd C::f_jac_rev(const Eigen::VectorXd & x, Eigen::MatrixXd & J) const
{
    Eigen::VectorX<autodiff::var> xvar = x.cast<autodiff::var>();
    Eigen::VectorX<autodiff::var> fvar = f(xvar);   // Build expression tree
    J.resize(fvar.size(), xvar.size());
    for (Eigen::Index i = 0; i < fvar.size(); ++i)
        J.row(i) = gradient(fvar(i), xvar);      // Evaluate derivatives from tree
    return fvar.cast<double>();                  // cast return value to double
}
```

b) Run `ninja` and ensure the code builds (especially for those using automatic differentiation). Note that there are no unit tests to validate your solution.

c) (Optional) If you have implemented any of the analytical Jacobians and wish to check your work, consider writing unit tests that use automatic differentiation to generate the oracle data.

3. Confidence regions

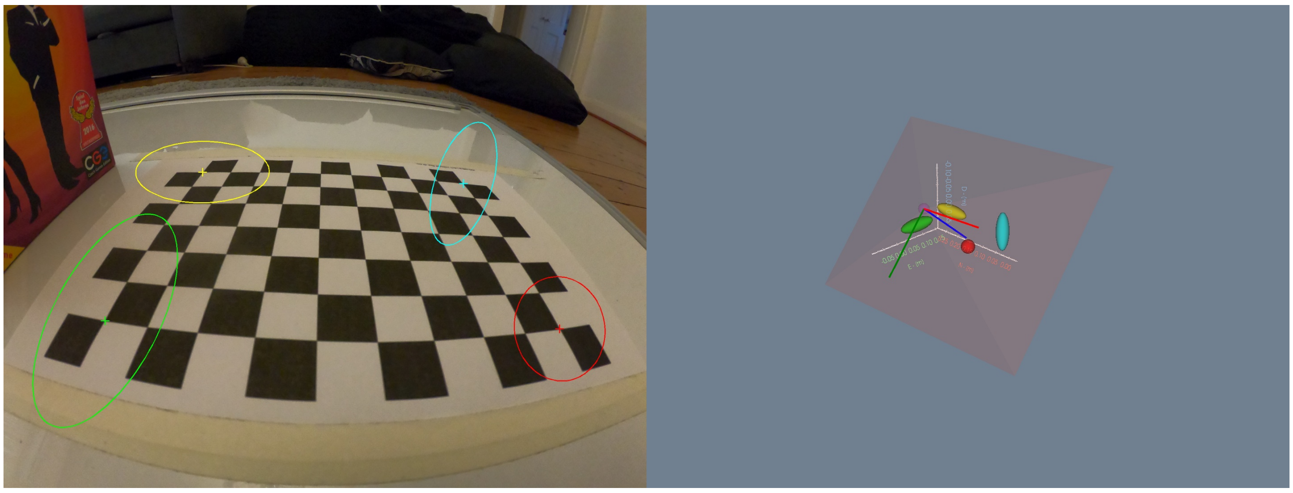

Figure 1: Rendering output for confidence regions of features (left) and camera and landmark positions (right).

The Bayesian credible region (BCR) or *confidence region* of a pdf $p(\cdot)$ is given by the set

$$\mathcal{R} = \{\mathbf{x} \in \mathcal{X} \mid p(\mathbf{x}) \geq t\}, \tag{15a}$$

where the density threshold, t, is the unique solution to

$$\int_{p(\mathbf{x})\geq t} p(\mathbf{x})\,\mathrm{d}\mathbf{x} = c, \tag{15b}$$

where $0 < c < 1$ is the desired probability mass to be contained within the BCR. Common choices include $c = 0.95$ or $c = 0.997$, which correspond to the $\mu \pm 2\sigma$ and $\mu \pm 3\sigma$ regions of a Gaussian distribution, respectively.

If $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \mathbf{P})$ where $\mathbf{x} \in \mathbb{R}^n$, then the BCR (15) is an n-dimensional hyperellipsoid that defined by the points $\mathbf{x}$ that satisfy the following inequality:

$$(\mathbf{x} - \boldsymbol{\mu})^{\mathsf{T}} \mathbf{P}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \leq \chi^2_n(c), \tag{16}$$

where $\chi^2_n(c)$ is the inverse cumulative distribution function for probability c of the chi-squared distribution with n degrees of freedom[2].

The points $\mathbf{x}$ that lie on the boundary of (16) satisfy the following equation:

$$\underbrace{\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}^{\mathsf{T}}}_{\mathbf{p}^{\mathsf{T}}} \underbrace{\begin{bmatrix}\mathbf{P}^{-1} & -\mathbf{P}^{-1}\boldsymbol{\mu}\\ -\boldsymbol{\mu}^{\mathsf{T}}\mathbf{P}^{-1} & \boldsymbol{\mu}^{\mathsf{T}}\mathbf{P}^{-1}\boldsymbol{\mu} - \chi^2_n(c)\end{bmatrix}}_{\mathbf{Q}} \underbrace{\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}}_{\mathbf{p}} = 0, \tag{17}$$

which admits the following compact form in homogeneous coordinates:

$$\mathbf{p}^{\mathsf{T}}\mathbf{Q}\mathbf{p} = 0. \tag{18}$$

If we parameterise the covariance with an upper triangular matrix $\mathbf{S}$ such that $\mathbf{S}^{\mathsf{T}}\mathbf{S} = \mathbf{P}$, then (16) can be written as

$$\begin{aligned}(\mathbf{x} - \boldsymbol{\mu})^{\mathsf{T}}(\mathbf{S}^{\mathsf{T}}\mathbf{S})^{-1}(\mathbf{x} - \boldsymbol{\mu}) &\leq \chi^2_n(c)\\ (\mathbf{x} - \boldsymbol{\mu})^{\mathsf{T}}\mathbf{S}^{-1}\mathbf{S}^{-\mathsf{T}}(\mathbf{x} - \boldsymbol{\mu}) &\leq \chi^2_n(c)\\ \left(\mathbf{S}^{-\mathsf{T}}(\mathbf{x} - \boldsymbol{\mu})\right)^{\mathsf{T}}\mathbf{S}^{-\mathsf{T}}(\mathbf{x} - \boldsymbol{\mu}) &\leq \chi^2_n(c)\\ \mathbf{w}^{\mathsf{T}}\mathbf{w} &\leq \chi^2_n(c),\end{aligned} \tag{19}$$

where $\mathbf{w}$ is the solution to the following triangular system of equations:

$$\mathbf{S}^{\mathsf{T}}\mathbf{w} = \mathbf{x} - \boldsymbol{\mu}. \tag{20}$$

The matrix that appears in the homogeneous equation (18) is given by the following:

$$\mathbf{Q} = \begin{bmatrix}\mathbf{S}^{-1}\mathbf{S}^{-\mathsf{T}} & -\mathbf{y}\\ -\mathbf{y}^{\mathsf{T}} & \mathbf{z}^{\mathsf{T}}\mathbf{z} - \chi^2_n(c)\end{bmatrix}, \tag{21}$$

where $\mathbf{z}$ and $\mathbf{y}$ are the solutions to the following triangular systems of equations:

$$\begin{aligned}\mathbf{S}^{\mathsf{T}}\mathbf{z} &= \boldsymbol{\mu},\\ \mathbf{S}\mathbf{y} &= \mathbf{z}.\end{aligned}$$

[2] In MATLAB, this is equivalent to $\chi^2_n(c) = \texttt{chi2inv}(c, n)$.

Tasks

a) Complete the implementation of $\chi^2_\nu(p)$ in the static function `Gaussian::chi2inv`, which is the inverse cumulative distribution function for probability p of the chi-squared distribution with ν degrees of freedom. To do this, you can use `boost::math::gamma_p_inv`, noting that

$$\texttt{chi2inv}(p, \nu) \equiv 2 \cdot \texttt{gamma\_p\_inv}\left(\frac{\nu}{2}, p\right). \tag{22}$$

Include appropriate `boost` header file(s) within `src/Gaussian.hpp`.

b) Complete the implementation of the static function `Gaussian::normcdf`, which evaluates the cumulative density function for a standard normal distribution[3]. To do this, you can use `boost::math::erfc`, noting that

$$\texttt{normcdf}(w) \equiv \frac{1}{2} \cdot \texttt{erfc}\left(\frac{-w}{\sqrt{2}}\right). \tag{23}$$

Include appropriate `boost` header file(s) within `src/Gaussian.hpp`.

c) Implement (19) in `Gaussian::isWithinConfidenceRegion`.

Hint

The probability mass, c, contained within k standard deviations of the mean of a Gaussian pdf is given by

$$c = \int_{\mu-k\sigma}^{\mu+k\sigma} \mathcal{N}(x; \mu, \sigma^2)\,\mathrm{d}x = \int_{-k}^{k} \mathcal{N}(x; 0, 1)\,\mathrm{d}x, \tag{24}$$

which can be integrated numerically or computed using the normal cumulative distribution function.

d) Complete the implementation of `Gaussian::confidenceEllipse`, which returns a set of points on the elliptical boundary of a given confidence region for a bivariate ($n = 2$) Gaussian distribution.

Hint

For $n = 2$, $\mathbf{w} \in \mathbb{R}^2$ and the boundary of (19) becomes the equation of a circle with radius $\sqrt{\chi^2_n(c)}$. Points on this circle can be found in $\mathbf{w}$-coordinates and then transformed back to $\mathbf{x}$-coordinates using (20) to obtain points on the boundary of the confidence ellipse.

e) Complete the implementation of (21) in `Gaussian::quadricSurface`, which calculates the homogeneous matrix representation for the quadric surface $\mathbf{Q}$, for a trivariate ($n = 3$) Gaussian.

f) Build the application and ensure the unit tests in `test/src/GaussianConfidence.cpp` pass.

g) Within `src/Plot.cpp`, complete the implementation of the `QuadricPlot::update` function, which calls `Gaussian::quadricSurface` to obtain $\mathbf{Q}$ and then populates the coefficients of the `vtkQuadric` instance via `quadric->SetCoefficients(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9)`.

[3]zero mean and unit variance

Tip

The `vtkQuadric` class expects a quadric in the following form

$$F(x,y,z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9 = 0$$

the above expression can be represented in matrix form as

$$F(x,y,z) = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \underbrace{\begin{bmatrix} a_0 & \frac{1}{2}a_3 & \frac{1}{2}a_5 & \frac{1}{2}a_6 \\ \cdot & a_1 & \frac{1}{2}a_4 & \frac{1}{2}a_7 \\ \cdot & \cdot & a_2 & \frac{1}{2}a_8 \\ \cdot & \cdot & \cdot & a_9 \end{bmatrix}}_{\mathbf{Q}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where $\mathbf{Q}$ is a symmetric matrix.

h) Copy `config.xml`, `camera.xml` and all the image files within `lab3/data` to `lab7/data`.

i) Run the application and ensure the confidence ellipses and quadric surfaces match those in Figure 1.

Terminal

```
nerd@basement:~/MCHA4400/lab7/build$ ninja && ./lab7
[Interactive VTK window is displayed]
```

The 3D view can be manipulated by dragging the mouse or using the mouse scroll. Try also holding down `Ctrl` or `Shift` while dragging the mouse. The mouse interactions are documented in the detailed description section of the `vtkInteractorStyleTrackballCamera` class and the keyboard commands are documented in the detailed description section of the `vtkInteractorStyle` class.

j) Export all of the calibration images without blocking on the interactor as follows:

Terminal

```
nerd@basement:~/MCHA4400/lab7/build$ ninja && ./lab7 -e -i=0
```

Inspect the files written to the `out` directory.

Tip

You may need to modify the plotting code for the Assignment(s). Make use of the VTK C++ examples to familiarise yourself with the approach for plotting with VTK. You should familiarise yourself with the following class definitions and documentation to strengthen your rendering knowledge.

- vtkActor — represents an object rendered in the scene, including its properties (colour, shading type, etc.) and position in the worlds coordinate system.
- vtkCamera — defines the view position, focal point and other viewing properties of the

scene.

- vtkImageMapper — provides 2D image display support for VTK.
- vtkInteractorStyle — the controller for window interactivity.
- vtkMapper — the geometric representation for an actor. More than one actor may refer to the same mapper.
- vtkProperty — defines the appearance properties of an actor including colour, transparency, and lighting properties such as specular and diffuse. Also representational properties like wireframe and solid surface.
- vtkRenderer — coordinates the rendering process involving lights, cameras, and actors.
- vtkRenderWindow — manages a window on the display device; one or more renderers draw into an instance of vtkRenderWindow.

A. Jacobian of `vectorToPixel`

The gradients of u and v with respect to the input $\mathbf{r}^c_{P/C}$ are given by

$$\frac{\partial u}{\partial \mathbf{r}^c_{P/C}} = \begin{bmatrix} \frac{1}{z} & 0 & -\frac{x}{z^2} \end{bmatrix}, \tag{25}$$

$$\frac{\partial v}{\partial \mathbf{r}^c_{P/C}} = \begin{bmatrix} 0 & \frac{1}{z} & -\frac{y}{z^2} \end{bmatrix}. \tag{26}$$

The gradients of the radius with respect to u and v are given by

$$\frac{\partial r}{\partial u} = \left(u^2 + v^2\right)^{-\frac{1}{2}} u, \tag{27}$$

$$\frac{\partial r}{\partial v} = \left(u^2 + v^2\right)^{-\frac{1}{2}} v. \tag{28}$$

The gradient of the radial distortion model with respect to r is given by

$$\frac{\partial \alpha}{\partial r} = 2k_1 r + 4k_2 r^3 + 6k_3 r^5. \tag{29}$$

The gradient of the rational distortion model with respect to r is given by

$$\frac{\partial \beta}{\partial r} = 2k_4 r + 4k_5 r^3 + 6k_6 r^5. \tag{30}$$

The gradient of the c with respect to r is given by

$$\frac{\partial c}{\partial r} = \frac{\frac{\partial \alpha}{\partial r}(1+\beta) - (1+\alpha)\frac{\partial \beta}{\partial r}}{(1+\beta)^2}. \tag{31}$$

The gradient of u' with respect to u is given by

$$\frac{\partial u'}{\partial u} = \frac{\partial c}{\partial r}\frac{\partial r}{\partial u} u + c + 2p_1 v + p_2\left(2r\frac{\partial r}{\partial u} + 4u\right) + 2s_1 r\frac{\partial r}{\partial u} + 4s_2 r^3\frac{\partial r}{\partial u}. \tag{32}$$

The gradient of u' with respect to v is given by

$$\frac{\partial u'}{\partial v} = \frac{\partial c}{\partial r}\frac{\partial r}{\partial v} u + 2p_1 u + p_2\left(2r\frac{\partial r}{\partial v}\right) + 2s_1 r\frac{\partial r}{\partial v} + 4s_2 r^3\frac{\partial r}{\partial v}. \tag{33}$$

The gradient of v' with respect to u is given by

$$\frac{\partial v'}{\partial u} = \frac{\partial c}{\partial r}\frac{\partial r}{\partial u} v + 2p_2 v + p_1\left(2r\frac{\partial r}{\partial u}\right) + 2s_3 r\frac{\partial r}{\partial u} + 4s_4 r^3\frac{\partial r}{\partial u}. \tag{34}$$

The gradient of v' with respect to v is given by

$$\frac{\partial v'}{\partial v} = \frac{\partial c}{\partial r}\frac{\partial r}{\partial v} v + c + 2p_2 u + p_1\left(2r\frac{\partial r}{\partial v} + 4v\right) + 2s_3 r\frac{\partial r}{\partial v} + 4s_4 r^3\frac{\partial r}{\partial v}. \tag{35}$$

The Jacobian of (6) with respect to $\mathbf{r}^c_{P/C}$ is given by

$$\frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} = \begin{bmatrix} f_x \left(\frac{\partial u'}{\partial u} \frac{\partial u}{\partial \mathbf{r}^c_{P/C}} + \frac{\partial u'}{\partial v} \frac{\partial v}{\partial \mathbf{r}^c_{P/C}} \right) \\ f_y \left(\frac{\partial v'}{\partial u} \frac{\partial u}{\partial \mathbf{r}^c_{P/C}} + \frac{\partial v'}{\partial v} \frac{\partial v}{\partial \mathbf{r}^c_{P/C}} \right) \end{bmatrix}. \tag{36}$$

B. Jacobian of `worldToPixel`

Applying the chain rule to (8a) yields the Jacobians of `w2p` with respect to $\mathbf{r}^n_{P/N}$, $\mathbf{r}^n_{B/N}$ and $\mathbf{\Theta}^n_b$ as follows:

$$\frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^n_{P/N}} = \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} \frac{\partial \mathbf{r}^c_{P/C}}{\partial \mathbf{r}^n_{P/N}} = \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} (\mathbf{R}^b_c)^\mathsf{T} (\mathbf{R}^n_b)^\mathsf{T}, \tag{37a}$$

$$\frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^n_{B/N}} = \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} \frac{\partial \mathbf{r}^c_{P/C}}{\partial \mathbf{r}^n_{B/N}} = -\frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} (\mathbf{R}^b_c)^\mathsf{T} (\mathbf{R}^n_b)^\mathsf{T}, \tag{37b}$$

$$\begin{aligned} \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{\Theta}^n_b} &= \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} \frac{\partial \mathbf{r}^c_{P/C}}{\partial \mathbf{\Theta}^n_b} \\ &= \frac{\partial \mathbf{r}^i_{Q/O}}{\partial \mathbf{r}^c_{P/C}} (\mathbf{R}^b_c)^\mathsf{T} \left[\left(\frac{\partial \mathbf{R}^n_b}{\partial \phi}\right)^\mathsf{T} (\mathbf{r}^n_{P/N} - \mathbf{r}^n_{B/N}) \quad \left(\frac{\partial \mathbf{R}^n_b}{\partial \theta}\right)^\mathsf{T} (\mathbf{r}^n_{P/N} - \mathbf{r}^n_{B/N}) \quad \left(\frac{\partial \mathbf{R}^n_b}{\partial \psi}\right)^\mathsf{T} (\mathbf{r}^n_{P/N} - \mathbf{r}^n_{B/N}) \right]. \end{aligned} \tag{37c}$$

Note

When finding the Jacobian of (13) with respect to Euler angles, we must take the Jacobian of the rotation matrix with respect to its parameters. The rotation matrix can be parametrised by Euler angles as follows:

$$\mathbf{R}(\mathbf{\Theta}^n_b) = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi) = e^{\psi \mathbf{S}(\mathbf{e}_3)} e^{\theta \mathbf{S}(\mathbf{e}_2)} e^{\phi \mathbf{S}(\mathbf{e}_1)}.$$

The Jacobians with respect to each Euler angle are given by

$$\begin{aligned} \frac{\partial \mathbf{R}(\mathbf{\Theta}^n_b)}{\partial \psi} &= \frac{\partial \mathbf{R}_z(\psi)}{\partial \psi} \mathbf{R}_y(\theta)\mathbf{R}_x(\phi) = \mathbf{S}(\mathbf{e}_3)\mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi) = \mathbf{R}_z(\psi)\mathbf{S}(\mathbf{e}_3)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi), \\ \frac{\partial \mathbf{R}(\mathbf{\Theta}^n_b)}{\partial \theta} &= \mathbf{R}_z(\psi) \frac{\partial \mathbf{R}_y(\theta)}{\partial \theta} \mathbf{R}_x(\phi) = \mathbf{R}_z(\psi)\mathbf{S}(\mathbf{e}_2)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi) = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{S}(\mathbf{e}_2)\mathbf{R}_x(\phi), \\ \frac{\partial \mathbf{R}(\mathbf{\Theta}^n_b)}{\partial \phi} &= \mathbf{R}_z(\psi)\mathbf{R}_y(\theta) \frac{\partial \mathbf{R}_x(\phi)}{\partial \phi} = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{S}(\mathbf{e}_1)\mathbf{R}_x(\phi) = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi)\mathbf{S}(\mathbf{e}_1). \end{aligned}$$