Models and State Representation in Scene: Generalised Software for Sequential Map-Building and Localisation

Andrew J. Davison
Robotics Research Group
Department of Engineering Science
University of Oxford
UK
ajd@robots.ox.ac.uk
http://www.robots.ox.ac.uk/~ajd/

March 8, 2004

1 Introduction

A major goal of Scene is to permit the uniform use of sequential estimation methods across the widest possible range of robot, camera or other mobile sensor platforms. To this end, a modular **model** system is used to represent the specifics of a particular system, so that applying Scene to a new application is a simple matter of creating new model classes which can be "plugged in" to the main system.

In this document we will discuss in general the meaning of using mathematical models to represent real-world systems, and then look specifically at how models are used in Scene to describe motion and measurements in one, two and three dimensions, explaining the procedure needed to use Scene in a new application domain by building new model classes.

This document is best read in combination with a study of the source code of the example programs using the Scene Library which are supplied in the SceneApp package: the specific implementations there will clarify the general explanations given here.

See [1] for more general information on the Sequential Localisation and Map-Building theory behind Scene.

2 Models

A model is a mathematical representation of a real-world process. If we wish to make sense of any data obtained from measurements the real world, a model is essential: it is what connects the numbers we put into and get out of a system.

The way we prefer to define models, however, is as follows:

- **A model is a simplification of reality.**

In the real world, systems are never made from pure geometrical objects interacting in perfect ways. Even saying something like "the side of this object is flat" is, if we examine the object closely enough, an approximation to its non-smooth structure on a microscopic or molecular scale — while this is a rather extreme way to think about things, we feel that it is important to realise what is going on when we talk about models: they are always simplifications.

Constructing a model is a process of forming a description of a real-world system which reduces its actual complexity to equations and a number of **parameters** describing the specifics of a simplified mathematical process. The amount of complexity in the real-world system which we attempt to represent in the model is always a choice. Think of building a model for an articulated robot arm: the choice taken here would usually be to represent it as a chain of rigid components joined by hinged couplings with motors and encoders. A more complex model, though, might also take into account factors such as the flexibility of the "rigid" components. A line must be drawn somewhere — usually in a position such that the model remains mathematically convenient while representing reality with a high enough degree of realism to satisfy our needs.

As well as representing what we know about a system in a model, we must also acknowledge the finer details that we don't know about the system: this is what is usually referred to as noise, or uncertainty in the model. **Noise accounts for the things that we don't attempt to model.** There is of course no such thing as random noise in (classical!) physics: bodies move and interact deterministically. If we had a perfect description of a system then we would not need measurements to tell us what was happening to it: all we would need to do would be to set up the "perfect model" and let it run. However, while in theory we could model everything (slipping wheels; human joints, muscles, chemical reactions!), in reality we can't know all the details of what is happening to a system. Models have to stop somewhere: the rest of what is going on we call noise, and estimate the size of its effect. It is this measure of the amount a model potentially differs from reality which means that **measurements** are able to improve our knowledge of a system. Measurement processes themselves have models with uncertainty, and thus can be combined in a weighted sense with the estimates from our system model to get improved estimates.

3 Using Models in Scene

Scene's current domain of applicability can be defined as follows:

- Scene can be applied to any situation where we wish to estimate the states of a generally moving robot and many stationary features of which it is able to make measurements via one or more sensors which are mounted in a fixed position relative to the robot.

(Note that there are several interesting ways of using Scene which fit these criteria although they might not initially seem to do so. One is that a moving, active sensor such as a pair of cameras mounted on a motorised platform can be used as long as the whole setup is considered as a single sensor (with no internal state) whose base is fixed rigidly to the main robot. Another is that Scene has successfully been applied to a pair of cooperating robots in the case that only one of them carries sensors able to make measurements of world features: the second robot can be thought of as simply an extension to the first, in rather the same way as we would naturally model a two-armed robot as a single system. It does not matter that the two cooperating robots are not physically connected.)

We will often refer to the "state" of a system: this is the vector of parameters which, along with the equations specifying our model, represent what is currently known about the system. In Scene, the total state vector $\mathbf{x}$ can be partitioned as follows:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_v \\ \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \end{pmatrix} , \tag{1}$$

where $\mathbf{x}_v$ is the state of the robot and $\mathbf{y}_i$ is the state of the ith feature. Computation of how this state vector changes over time is the goal of all the processing in Scene. The two main operations which take place on the state vector are:

1. The process equation, which describes how the state of the robot changes over time (the motion model):

   $$\mathbf{x}_v(t + \Delta t) = \mathbf{f}_v(\mathbf{x}_v(t), \mathbf{u}, \Delta t) + \mathbf{q} \ . \tag{2}$$

   $\mathbf{q}$ is a zero-mean Gaussian noise vector with covariance `Q` (the process noise). Since the features are assumed to be stationary, their states are not involved in the process equation.

2. The measurement equation, which describes how a measurement is made of a feature (the feature measurement model):

   $$\mathbf{z}_i = \mathbf{h}_i(\mathbf{x}_v, \mathbf{y}_i) + \mathbf{r} \ . \tag{3}$$

   $\mathbf{r}$ is a zero-mean Gaussian noise vector with covariance `R` (the measurement noise).

Which parameters belong in the state vector? Scene allows a free choice to be made about what to choose as the representation for the robot state x_v and that of each feature type state y_i. As discussed above, parameters are numbers which describe the specifics of a mathematical model of a system. However, the equations of a model might have parameters which need not be in the state vector: they can be treated as constant, for instance, like the length of the jointed sections of a robot arm. On a first reflection then, one might define state parameters as "the parameters that change over time."

However, sometimes it is worthwhile to include in a state vector a parameter which represents something in the system which is expected to remain constant. While that real quantity stays constant, our estimate of it can change (in fact it can only improve) over time as more measurements are made. This if a form of on-line self-calibration.

As an example, in previous work on robot localisation it was found that our model included one particularly significant systematic error: when a velocity command was sent to the robot, the actual velocity achieved was not normally distributed about the commanded value, but actually about some other mean which differed significantly; e.g. when sending the command $1ms^{-1}$ the robot was actually running at $0.9 \pm 0.1ms^{-1}$ rather than $1.0 \pm 0.1ms^{-1}$: there was a scaling between the two which was not equal to unity. To overcome this problem, we included in the state vector a parameter representing the scaling between velocity commands sent and the real velocity of the robot. This is not something which expected to change over time; rather it is a constant characteristic of a particular robot about which we are not very certain. Therefore, taking a first guess of the parameter's value (e.g. 1), and a measure of the standard deviation of that initial guess (e.g. 0.2), we initialise the parameter into the state vector with non-zero covariance. Although the parameter does not get changed in the model's process equation, its involvement in various equations and Jacobians means that over time and with measurements its estimated value will change and its uncertainty will decrease. The estimate should converge towards the true value.

In general, if we write down equations to model a system, **any parameter which appears in the equations could be placed in the state vector.** Parameters which are placed in the state vector will have their estimates improved by measurements, while those left out will be be constants whose estimated value will not change. A choice must be made as to which parameters should be placed in the state vector. The downside of including many parameters is that this causes computational complexity to be increased. However, if the value of a parameter is relatively uncertain, it is worth considering including it in the state vector.

A final third type of parameter concerns the effect of **known outside influences** on the modelled system. Since physics is deterministic, a model of any enclosed system will be deterministic: it will be possible to predict its state all future times. However, when we model a real object such as a robot, we do not usually have a truly enclosed system: there may be something which is influencing it, for instance via purposive commands and inputs, whether human operator or onboard computer. Whichever it is, its effect cannot be included in a deterministic model: we do not try to represent the state of the computer or a human brain in our state vector; but its influence is not constant either. Nevertheless, we may have information about the inputs the robot is receiving and these affect how the robot's state will evolve. We call these input parameters **control parameters** and in equations use the vector $\mathbf{u}$ to refer to them.

Note: there may also be various **unknown outside influences** on a system. In this case, their effect must be included as noise in the process equation. For instance, if instead of a robot we have mounted a sensor on a human head (such as a forehead-mounted, outward-looking camera) and aim to estimate the head's motion in the same way as which we would estimate the motion of a robot equipped with the same sensor, the way in which the person moves is unknown, but contributes uncertainty to the camera's motion in a way which can be modelled. Again, all we are doing here is accepting that noise in a model represents the things we don't attempt to model.

Summarising, parameters can be divided into three types:

1. **State parameters**, which are parameters describing moving aspects of the system, or stationary aspects of which we wish to improve initially uncertain estimates.

2. **Constant parameters**, of which we have an estimate with which we are happy and which we are content not to improve.

3. **Control parameters**, which are quantities which affect the future state of the model but which are not known; they come from outside the system under consideration, such as control inputs to a robot. They are like one-off constants which appear in the system equations transiently.

4 Position State

We introduce here the concept of **position state** $\mathbf{x}_p$: this is a standard, minimal way to represent the raw geometrical position and pose of a robot in 1, 2 or 3D space.

Most implementations of robot localisation have made no distinction between the concepts of state and position state: usually, what we would call the position state — the minimal description of robot location — is what goes directly into the state vector. However, as discussed above, modelling systems sometimes calls for parameters additional to those purely describing position to be part of the state vector — in systems with redundant degrees of freedom for example. While it may be the case that the additional parameters appear in the state vector simply in addition to those representing pure position, more generally the position state is something which is functionally derived from all the state parameters:

$$\mathbf{x}_p = \mathbf{x}_p(\mathbf{x}_v) \tag{4}$$

For instance, if our system is a robot arm, the parameters which we store in the state vector might be the angles of articulation at each joint. A sensor mounted at the end of the arm has 3D position which is a rather complicated function of all of these angles and the constant characteristics of the arm.

In Scene, we gain greatly from the concept of position state because it allows **separation** of the details of a particular robot motion model from the essential

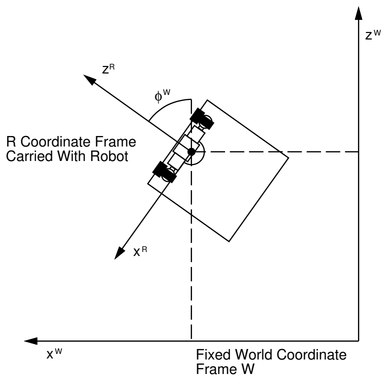

Figure 1: Coordinate frames in two dimensions.

position information which allows that object to interact with other object in the world. Specifically, we can separate the concepts of motion model and feature measurement model.

When specifying a feature measurement model, we need to define functions such as $\mathbf{h}_i(\mathbf{x}_v, \mathbf{y}_i)$ (the measurement obtained of a feature as a function of the robot state and the feature's state). By abstracting this to $\mathbf{h}_i(\mathbf{x}_p, \mathbf{y}_i)$ (the measurement as a function of **robot position** and feature state), where we know from the definition of position state $\mathbf{x}_p(\mathbf{x}_v)$, we are able to use measurement models which are decoupled from specific robot motion models. For instance, we could use the same model of a camera whether it is fixed rigidly to a robot vehicle or mounted at the end of an articulated robot arm.

It should be noted that the concept of position state, while convenient at the current time, is only a step towards what we would like to achieve with the future design of Scene, in which a robot models could have multiple "hooks" into position-state-like quantities. This would be necessary if, for instance, a robot consisted of multiple non-rigid parts, more than one of which was equipped with a sensor.

4.1 Position State in One Dimension

$$\mathbf{x}_p = (z) \tag{5}$$

In 1D, the position state consists of a single parameter z describing displacement along a straight line.

4.2 Position State in Two Dimensions

$$\mathbf{x}_p = \begin{pmatrix} z \\ x \\ \phi \end{pmatrix} \tag{6}$$

In 2D movement, position on a plane can be specified by 3 parameters: the cartesian coordinates z and x and angle ϕ (restricted to the range $-\pi < \phi <= \pi$) representing orientation relative to the z axis. We apologise somewhat here for our choice of z and x as our 2D coordinates rather than the more usual x and y — this choice stems from our original application domain in computer vision, where traditionally the z axis of coordinate frames are aligned with the optic axis of cameras: for a robot with a forward-facing camera, this makes z horizontal. See Figure 1 for clarification of this.

4.3 Position State in Three Dimensions

Representing position and pose in 3D is substantially more complicated than the 1D and 2D cases. There are several different ways to represent 3D orientation, but each has its disadvantages. Minimally, three parameters are needed to represent 3D orientation (as in for example the Euler angle parameterisation), with a further three required for cartesian position, leading to a total of six parameters. However, we have chosen to add one parameter to this minimal six for our standard representation of 3D position and pose and use a **quaternion** of 4 parameters to represent orientation.

$$\mathbf{x}_p = \begin{pmatrix} x \\ y \\ z \\ q_0 \\ q_x \\ q_y \\ q_z \end{pmatrix} \tag{7}$$

Quaternions are a well-established way to represent 3D orientation. Any 3D rotation can be described by a single rotation about an appropriately placed axis. In a quaternion, a unit vector $\mathbf{u}$ (with elements u_x, u_y, u_z) representing the axis of this rotation and its angular magnitude θ are stored as follows:

$$\begin{pmatrix} q_0 \\ q_x \\ q_y \\ q_z \end{pmatrix} = \begin{pmatrix} \cos\frac{\theta}{2} \\ u_x \sin\frac{\theta}{2} \\ u_y \sin\frac{\theta}{2} \\ u_z \sin\frac{\theta}{2} \end{pmatrix} . \tag{8}$$

Quaternions are defined in this way because they have an algebra which allows rotations to be conveniently composed (applied in sequence). In Scene, in the 3D position state vector a quaternion represents the rotation of the robot with respect to the fixed world coordinate frame. The following is a summary of the relevant properties of quaternions:

- The magnitude of a quaternion, defined as the square root of the sum of the squares of the elements, is always 1:

$$q_0^2 + q_x^2 + q_y^2 + q_z^2 = 1^2 . \tag{9}$$

- Quaternion $\mathbf{q}$ has a conjugate $\bar{\mathbf{q}}$ which represents a rotation about the same axis but with negative magnitude, and which is defined as:

$$\bar{\mathbf{q}} = \begin{pmatrix} q_0 \\ -q_x \\ -q_y \\ -q_z \end{pmatrix} . \tag{10}$$

- The rotation matrix $\mathtt{R}$ associated with quaternion $\mathbf{q}$ is defined as follows:

$$\mathtt{R}\mathbf{v} = \mathbf{q} \times \mathbf{v} \times \bar{\mathbf{q}} \tag{11}$$

  where $\mathbf{v}$ is an arbitrary 3×1 column vector. We can calculate that:

$$\mathtt{R} = \begin{bmatrix} q_0^2 + q_x^2 - q_y^2 - q_z^2 & 2(q_x q_y - q_0 q_z) & 2(q_x q_z + q_0 q_y) \\ 2(q_x q_y + q_0 q_z) & q_0^2 - q_x^2 + q_y^2 - q_z^2 & 2(q_y q_z - q_0 q_x) \\ 2(q_x q_z - q_0 q_y) & 2(q_y q_z + q_0 q_x) & q_0^2 - q_x^2 - q_y^2 + q_z^2 \end{bmatrix} . \tag{12}$$

  So, if we have quaternion $\mathbf{q}$ as part of the position state of a robot and form rotation matrix $\mathtt{R}$ as above, this $\mathtt{R}$ will relate vectors in the **fixed world coordinate frame** W and the **robot coordinate frame** R (which is a coordinate frame carried around by the robot) as follows:

$$\mathbf{v}^W = \mathtt{R}\mathbf{v}^R . \tag{13}$$

  We prefer to be clear in our descriptions of rotation matrices, and refer to this $\mathtt{R}$ as $\mathtt{R}^{WR}$ since it relates vectors in frames W and R: the equation becomes:

$$\mathbf{v}^W = \mathtt{R}^{WR}\mathbf{v}^R . \tag{14}$$

  Note that it is very difficult to get confused about rotation matrices when this notation is used: in $\mathbf{v}^W = \mathtt{R}^{WR}\mathbf{v}^R$ the two frame suffices of the rotation matrix are on the sides closest to the vector specified in those frames.

  (When referring to vectors, we use the plain notation $\mathbf{v}$ to mean a purely spatial vector, a directed segment in space not associated with any coordinate frame. The notation $\mathbf{v}^A$ means the vector of coordinate parameters representing that spatial vector in frame A.)

- Composing rotations: if quaternion $\mathbf{q}_1$ represents the rotation $\mathtt{R}^{AB}$ and quaternion $\mathbf{q}_2$ represents the rotation $\mathtt{R}^{BC}$, then the composite rotation $\mathtt{R}^{AC} = \mathtt{R}^{AB}\mathtt{R}^{BC}$ is represented by the *product* of the two quaternions, defined as:

$$\mathbf{q}_3 = \mathbf{q}_1 \times \mathbf{q}_2 = \begin{pmatrix} q_{10}q_{20} - (q_{1x}q_{2x} + q_{1y}q_{2y} + q_{1z}q_{2z}) \\ q_{10}\begin{pmatrix} q_{2x} \\ q_{2y} \\ q_{2z} \end{pmatrix} + q_{20}\begin{pmatrix} q_{1x} \\ q_{1y} \\ q_{1z} \end{pmatrix} + \begin{pmatrix} q_{1y}q_{2z} - q_{2y}q_{1z} \\ q_{1z}q_{2x} - q_{2z}q_{1x} \\ q_{1x}q_{2y} - q_{2x}q_{1y} \end{pmatrix} \end{pmatrix} \tag{15}$$

The disadvantage of the quaternion representation is that it is redundant, using four parameters to represent something which can be described minimally with three. This leads to the minor factor of extra computational complexity in calculations due to the extra parameter, but more importantly it means that care must be taken to make sure that the four parameters in the quaternion part of the state vector represent a true quaternion: that is to say that they satisfy the magnitude criterion above.

For this reason, generalised normalisation functionality has been built into the models in Scene: a model class should know how to normalise its state vector if necessary. In a model containing a quaternion, this involves enforcing the magnitude constraint described above.

5 Model Classes

In Scene, modularity is implemented via a system of `Model` classes. There are three types:

1. `Motion_Model` classes describing the movement of a robot system of other sensor platform.

2. `Feature_Measurement_Model` classes each describing the process of measuring a feature with a particular sensor.

3. `Internal_Measurement_Model` classes describing measurements that a system makes of itself.

`Motion_Model`, `Feature_Measurement_Model`, `Internal_Measurement_Model` base classes are defined in the source file `SceneLib/Scene/models_base.h` within the SceneLib distribution. When building a Scene application, specific model classes are derived from these base classes. The derived classes will share the interfaces of their parents but of course provide specific instatiation of the functionality.

In a program using Scene, each specific model class to be used is instantiated only once and then a pointer to that class is passed around as needed. There will be just one motion model, zero, one or several feature measurement models (several if there is more than one type of feature of which it is possible to make measurements), and then optionally one internal measurement model. (In future versions of Scene it is planned to increase generality by providing greater choice here.) The specific model classes need to be defined by the creator of a new Scene application, and then instantiated in a program so that pointers can be passed to Scene's main classes which will make use of the models' functions to perform calculations.

Functions in model classes are used in a uniform way. A function named `func_A_and_B(C, D)` calculates the result vectors or matrices `A` and `B` from input vectors or matrices `C` and `D`. The results will be stored in pre-allocated spaces `CRES` and `DRES` in the model classes, from which they should be copied promptly after calling the function.

Several of the functions in model classes calculate Jacobians. A Jacobian is a matrix of derivatives of one vector with respect to the other. If vector $\mathbf{a}(\mathbf{b})$ is a function of vector $\mathbf{b}$, the Jacobian $\frac{\partial \mathbf{a}}{\partial \mathbf{b}}$ is defined as follows:

$$\frac{\partial \mathbf{a}}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial a_1}{\partial b_1} & \frac{\partial a_1}{\partial b_2} & \frac{\partial a_1}{\partial b_3} & \cdots \\ \frac{\partial a_2}{\partial b_1} & \frac{\partial a_2}{\partial b_2} & \frac{\partial a_2}{\partial b_3} & \cdots \\ \frac{\partial a_3}{\partial b_1} & \frac{\partial a_3}{\partial b_2} & \frac{\partial a_3}{\partial b_3} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \tag{16}$$

In the following detailed look at each of the types of model class, vectors and matrices are named as they appear in plain text form within the source code of Scene: thus $\mathbf{x}_v$ for example becomes `xv`.

5.1 Defining a Motion Model

Two levels of derivation are used with motion models. From the base class `Motion_Model`, motion model classes specific to a particular dimensionality of movement `OneD_Motion_Model`, `TwoD_Motion_Model` are `ThreeD_Motion_Model` are derived (again defined in the file `SceneLib/Scene/models_base.h`). One of these should then be used as the parent for a specific motion model to describe a particular system.

In a specific motion model class, the following constants must be defined:

1. `STATE_SIZE`, the integer state vector size.
2. `CONTROL_SIZE`, the integer control vector size.

And the following functions:

1. `func_fv_and_dfv_by_dxv(xv, u, delta_t)`, the main motion model function (process equation) which calculates a new state `fv` as a function of the old state `xv`, control vector `u` and time interval `delta_t`. Also calculates the Jacobian `dfv_by_dxv`.
2. `func_Q(xv, u, delta_t)`, which forms the covariance matrix `Q` of the process noise associated with `fv`.
3. `func_xp(xv)`, the function which defines the position state `xp` in terms of `xv`.
4. `func_dxp_by_dxv(xv)` which is the Jacobian for `func_xp`.
5. `func_fv_noisy(xv_true, u_true, delta_t)`, a noisy version of the process equation which is used in simulation only to This function should follow the same basic form as `func_fv_and_dfv_by_dxv` but incorporate random noise to represent uncertainty in the model.

6. `func_xvredef_and_dxvredef_by_dxv_and_dxvredef_by_dxpdef(xv, xpdef)` specifies how to redefine the robot state `xv` to `xvredef` when axes are redefined such that `xpdef`, a given position state, becomes the new zero of coordinates.

These should optionally be defined depending on the system:

1. `func_xvnorm_and_dxvnorm_by_dxv(xv)`, a function which defines how to **normalise** the parameters in `xv`, forming a new state `xvnorm`.
2. `navigate_to_waypoint(xv, xv_goal, u, delta_t)`, which can be used in systems with control parameters to set them automatically with the aim of reaching a goal state `xv_goal`.

5.2 Defining a Feature Measurement Model

With feature measurement models, two levels of derivation are again used. From the base class `Feature_Measurement_Model`, models specific to particular geometrical feature types are derived like `Point_Feature_Measurement_Model` and `Line_Feature_Measurement_Model` (in `SceneLib/Scene/models_base.h`). Then a specific feature measurement model class should be derived from one of these types.

In a specific feature measurement model class, the following constants need to be defined:

1. `MEASUREMENT_SIZE`, the number of parameters representing a measurement of the feature.
2. `FEATURE_STATE_SIZE`, the number of parameters to represent the state of the feature.

And the following functions:

1. `func_yipose_and_Pyiyipose(yi, Pyiyi)`, a function which takes the state and covariance of the feature and calculates its **pose state**: this is the basic representation of that geometrical feature type. This function is used for graphics only.
2. `func_yi_and_dyi_by_dxp_and_dyi_by_dhi_and_Ri(hi, xp)`, the feature initialisation function and its Jacobians, which calculates a feature's state `yi` from a measurement `hi` and the robot position state `xp`.
3. `func_hi_and_dhi_by_dxp_and_dhi_by_dyi(yi, xp)`, the main measurement function and its Jacobians, which calculates a measurement `hi` from the feature state `yi` and the robot position state `xp`.
4. `func_Ri(hi)`, which calculates the covariance `Ri` of the measurement noise for measurement `hi`.

5. `func_nui(hi, zi)`, a function which calculates the innovation `nui` of a measurement with predicted value `hi` and actual value `zi`. Normally this function should perform the simple subtraction `nui` = `zi` − `hi` but the function is left as user-definable because in some cases that is not the case: for instance, if a measurement parameter is an angle in the range $-\pi \to \pi$, cases where `zi` and `hi` lie either side of π would give an incorrectly large innovation if a simple subtraction was performed: in this case the user-defined innovation function can normalise the angle to the $-\pi \to \pi$ range.

6. `func_hi_noisy(yi_true, xp_true)`, a noisy measurement function for use in simulation, producing a measurement with random noise added.

7. `func_zeroedyi_and_dzeroedyi_by_dxp_and_dzeroedyi_by_dyi(yi, xp)`, a function which defines how to redefine the state of a feature in the case of a redefinition of axes. The position state `xp` at which the axes are to be redefined is given, and a new feature state `zeroedyi` is calculated from the current value `yi`, plus the relevant Jacobians.

8. `visibility_test(xp, yi, xp_orig, hi)`, which is used in active selection of measurements to decide whether the feature should be attempted to be measured from robot position `xp`. As well as the feature state `yi` and current predicted measurement `hi`, the position state `xp_orig` of the robot when the feature was first observed is passed. A criterion for the expected measurability of the feature should be defined based on these relative positions: for instance, if with vision we are attempting to match a template patch via image correlation matching, success could only be expected from within a limited range of robot motion away from the position from which the feature was first seen and the template stored. This function is different from other model functions in that it returns a single integer value representing success (0) or failure (other value).

9. `selection_score(Si)` is a function which calculates a score for a feature representing its value for immediate measurement based on the innovation covariance `Si`. This criterion will be used within Scene to compare candidate measurements of different features and allow resources to be devoted to where they are most useful. In general, measurements with a high `Si` should be favoured because it makes sense to make a measurement of a feature where the result is uncertain rather than of one where it is possible to accurately predict the result.

5.3 Defining an Internal Measurement Model

The derivation hierarchy for internal measurement models does not have any intermediate steps and all specific models should be derived directly from the base class `Internal_Measurement_Model`. The constants and functions which must be defined for an internal measurement model are very similar to those for a feature measurement model. Note however, the important difference that while a feature measurement model depends only on robot position state and is thus potentially compatible with many different motion models, an internal measurement model is specific to a particular motion model.

The following constant must be defined:

1. `MEASUREMENT_SIZE`, the number of parameters in the measurement vector.

And the following functions:

1. `func_hv_and_dhv_by_dxv(xv)`, the measurement function calculating measurement `hv` and Jacovian `dhv_by_dxv` from the robot state `xv`.
2. `func_Rv(hv)`, which calculates measurement noise `Rv` from predicted measurement `hv`.
3. `func_nuv(hv, zv)`, which calculates the innovation `nuv` from predicted measurement `hv` and actual measurement `zv`.
4. `func_hv_noisy(xv_true)`, the noisy measurement function for use in simulation.
5. `feasibility_test(xv, hv)`, which calculates the feasibility of the measurement based on the current robot state `xv` and predicted measurement `hv`. Analogous to function `visibility_test` in feature measurement models, this function returns 0 if the measurement is possible.

6 Conclusion

We have discussed the meaning of making mathematical models of real-world systems, and shown how model classes should be defined for use in Scene. This document will evolve as the design of Scene changes, hopefully towards a more powerful generic modelling framework supporting systems where multiple moving objects can be equipped with multiple independent sensors.

References

[1] A. J. Davison and N. Kita. Sequential localisation and map-building in computer vision and robotics. In *Proceedings of the 2nd Workshop on Structure from Multiple Images of Large Scale Environments (SMILE), in conjunction with ECCV 2000, Dublin, Ireland.* Springer-Verlag LNCS, 2000.