Navigation and Control with pNav and pHelm

Paul Newman

March 17, 2009

Abstract

This document will describe two control and navigation processes which are part of the standard releases of MOOS, namely pHelm and pNav . Both these processes have a marine flavour to them (which speaks to the origins of MOOS); nevertheless, both applications can be used on land vehicles.

1 The Helm - pHelm

Along with pNav , the Helm process Helm is one of the most important high-level processes that are typically run on a given mission. The Helm's job is to take NAV_* from the navigator and, given a set of mission goals, decide on the most suitable actuation commands. The multiple mission goals take the form of prioritized tasks within the Helm. For example, a "snap-shot" of the helm might reveal five active tasks: follow a track line, stay at constant depth, limit depth, limit altitude and limit mission length. The first two of these are conventional trajectory control tasks that could be expected to be found in any mobile robot lexicon. The last three are safety tasks that take some action if a limiting condition specified within them is violated.

The Helm is designed to allow "at sea" reloading of missions. If the mission file (specifying what tasks to run) is edited while the Helm is running, the Helm can be commanded via a MOOS message (RESTART_HELM) to clean down and rebuild itself. This makes for very rapid turnaround of missions in a research-oriented field trip.

The Helm has two modes: online and off-line. When off-line no tasks are run and the Helm makes no attempt to control actuators. In this mode the vehicle can be controlled via iRemote (see the document on iRemote for more information). When online, the vehicle's motors are under the control of the Helm – autopilot is engaged. As would be expected, iRemote is used to send the signal that relinquishes the manual control of the vehicle and puts the Helm online. Manual control can be regained at any time by pressing "space" on the iRemote console.

The details of tasks that can be run by the Helm and topics relating to their management are now discussed.

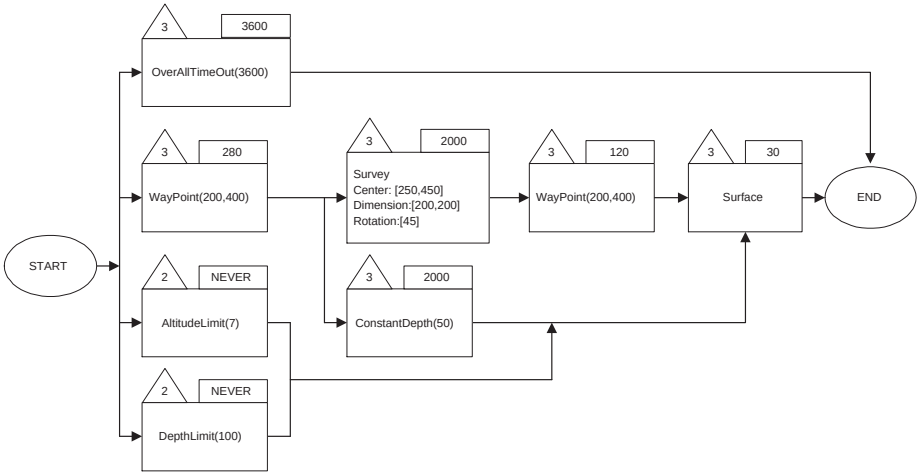

Figure 1: A typical mission plan. Traverse to (200,400), dive to 50m and perform a survey. Impose safety limits of 100m depth and 7m altitude. When the survey is complete (or timed out) return to the dive co-ordinate and surface. If something terrible has happened and the mission is still running after one hour, abort immediately. The task priorities are written in triangles and the timeouts in boxes above the task descriptions.

1.1 Tasks

As might be expected, the Helm and its constituent tasks rely heavily on the communications provided by the lower level MOOS API. The tasks use the communications apparatus to coordinate themselves. The parent Helm application is derived from `CMOOSApp` and as such has access to the MOOS communications system. The Helm application handles all communications on behalf of its owned tasks. All received `CMOOSMsg` s are offered to all active tasks. Each task is queried for any newly required subscriptions that are subscribed for by the Helm. In a similar fashion, each task is queried for any messages that it requires to be emitted. This mechanism gives each task the illusion that it has its own `CMOOSCommClient` object even though it is in fact sharing it with other active tasks.

All tasks are derived from a common base class that provides all descendants with a shared heritage, in particular the ability to specify in the mission file some key named properties which are described in Table 1. Perhaps the most important thing to understand is that tasks use MOOS messages to synchronize themselves. As one task finishes it emits one or more messages which may be "just the thing" that other tasks were watching for to signal that they should go active. Note that this has two important consequences:

- One task can activate any number of other tasks.
- A task can be activated by activity *outside* of the helm. For example, a command received by an acoustic modem could start a task. This open architecture is a powerful concept within MOOS.

Table 1: Base Task Properties

Property	Use
`Name`	The name of the task - for example "Leg7" or "InitialDelay"
`StartFlags`	A list of messages names that, if received, will spur the tasks into action (turn it on).
`FinishFlags`	A list of messages that are emitted when the task completes or when it starves because it is not receiving notifications on one or more of its subscribed variables.
`EventFlags`	A list of messages that are emitted when some event happens but the task does not complete. A typical example of this would be a depth limit task emitting event messages when the vehicle exceeds a specified limit. The last thing you want to happen is the task quitting just when this has happened. Instead the event flags are sent and the task keeps running.
`TimeOut`	The maximum time the task should run for. If the task does not complete naturally before this timeout the FinishFlags are sent and the task retires itself automatically. The value of "NEVER" can be specified to indicated that a task should never timeout — useful for safety tasks.
`InitialState`	A task can be on initially, in which case it does not listen for start-flags. Or it can be off, in which case it must receive a start-flag before it goes active.
`Priority`	Each task is assigned a priority from 1 to ∞ (0 is reserved for the special `EndMission` task). The lower the value, the more important the task is.

1.1.1 Task Completion

A given task completes when any of the following conditions are met.

- It has completed successfully – fulfilled its goal criteria. For example, it has steered the vehicle close (enough) to a way point or has driven the vehicle to a specified depth.
- It has timed out before achieving its goal.
- It has starved. This occurs when it is does not receive the data it needs at regular enough intervals. This is a crucial safety feature. Say, for example, the navigation has failed – `NAV_*` will not be being emitted by the navigator and a motion control task requiring navigation information to function should not be allowed to continue to run with blithe disregard. Starvation of tasks is a sure sign that something is wrong with the system configuration or navigation.

1.1.2 Task Arbitration

The Helm employs the simplest of strategies in deciding which task wins when two concurrent active tasks are both trying to control the same actuator — the

one with highest priority (lowest numerical value) wins. To illustrate this point, envision the case where a track-line task is running at priority level three. Then a way-point task with priority level two goes active – perhaps after reception of data via an acoustic modem. The way-point task will take control of the rudder actuator until it completes. At this point the original track-line task will resume. Throughout the episode the track-line task is unaware that it does not have control of the vehicle. In the case that one or more equal priority tasks controlling the same actuator(s) are active, the Helm simply chooses to give the first task processed (in its list of active tasks) control.

1.2 Task Configuration and *.hoof Files

Tasks are configured in a similar way to processes — within a named, brace delimited block of text. Any number of tasks can be specified in this fashion. A task configuration block always begins with `Task` =*TaskType*, where *TaskType* is one of the named types in Table 1 followed by the opening brace of the block. Figure 2 shows typical task configuration blocks for two tasks. In this

```
Task = ConstantHeading
{
    Priority = 3
    Name = South
    Heading = 180
    TimeOut = 300
    InitialState = OFF
    StartFlag = MissionStart
    FinishFlag = GoNorth
    LOGPID = true
}

Task = ConstantHeading
{
    Priority = 3
    Name = North
    Heading = 0
    TimeOut = 120
    InitialState = OFF
    StartFlag = GoNorth
    FinishFlag = EndMission
    LOGPID = true
}
```

Figure 2: A two task example

case two tasks are being configured — one will drive south for 300 seconds, and when this task completes another will drive north for 120 seconds — note the pairing of Start and Finish flags. Each task derived from the base task class is likely to add its own specialization parameters (like `Heading` in the case of `ConstantHeading` Tasks). The semantics and specifications of these

additions are found in the task documentation (see the MOOS web-pages `http://www.robots.ox.ac.uk/~pnewman/TheMOOS`).

1.2.1 Task Specification Redirection

If the Helm configuration block contains the line "`TaskFile` = *filename.hoof*" then at startup the helm builds all the tasks it finds in the specified `*.hoof` file. This indirection is useful for building a library of missions that can be loaded into the Helm at any time, simply by changing a single line in the Mission file.

Table 2 gives a brief summary of the most commonly used tasks found in the `MOOSTaskLib` library. This list is not exhaustive but serves to illustrate the kind of functionality provided. The configuration parameters for each task can be found on the MOOS documentation web-pages.

1.3 Third-party Task Execution

An interesting component of the Helm is its ability to dynamically launch a task on behalf of another client – "a third-party request". Obviously some security needs to be in place to stop wayward software requesting tasks to be performed that endanger other aspects of the mission or even the vehicle itself. The issue here is that third-party requests are simply MOOS messages with a certain string format issued from the innards of some unknown application. The third-party request security mechanism requires that the human creating the mission file actively grants permissions (by placing certain text in the Helm's configuration block) for the Helm to launch a specified task on behalf of a specified (named) client. Any third-party requests not mentioned in the configuration block will be denied.

1.3.1 Common Usage

One may wonder why, given the event-driven nature of the Helm, such a scheme is needed. The architecture was implemented by extending the model of Navigator and Helm pair to include a 'Guest Scientist' — a client application which, on the fly, makes requests to the Captain (Helm) to alter course to accommodate the needs of some scientific mission. However,

> "The Captain reserves the right to deny the request on the grounds that it may jeopardize the safety of the vessel or contradict more important mission orders."

This clause summarizes the interacting priorities of existing tasks and the extent of dynamic task generation the Helm is allowed to grant to third-parties.

1.3.2 Granting Permissions

The format of the permission specification in the Helm configuration block is as follows.

`Allow` =*jobname*@*clientname*:*tasktype* | `SessionTimeOut` =*Timeout*

The fields have the following meaning:

Jobname Describes the kind of work that the application requesting the launch of a task wants to do. It can be any name, for example, it might be "Explore" or "Detect Mine" – something that is meaningful to humans.

Clientname is the name of the application making the request, for example, `iAcousticModem` . It is the name that is used by the client when communicating with the DB.

Tasktype is the string name of a task supported by the Helm application. For example `XYPattern` .

SessionTime is the time the client has after completion of a requested third-party task during which it can request another task and be guaranteed that it will be accepted (provided it has the relevant permissions). This is a subtle point: imagine a client has requested the vehicle traverse to a distant way-point; having got there we do not want a competing client to request some other destination before the first task has had a chance to perform whatever it wanted to do at the "distant" way-point. Typically this value is set to a few seconds.

1.3.3 Request Generation

A client can generate a third-party request by using the `CThirdPartyRequest` class provided in `MOOSGenLib` . The class's methods allow specification of the task type and job name. For every line that one would usually see in a Task configuration block one call to `CThirdPartyRequest::AddProperty()` is made specifying the property name and value. This done, a call to the `ExecuteTask` function returns a string that can be transmitted via the `CMOOSCommClient` or through a call to `MOOSApp::Notify()` under the name of `THIRDPARTY_REQUEST` . The string returned by the `ExecuteTask` function simply packs all relevant information in a string in a manner that is understood by the Helm.

1.4 Dynamic Controllers

We shall now discuss how each task maps navigation data and task goals to desired actuation settings.

Each task possesses two standard PID controllers (Proportional Integral Differential) – one for yaw control and one for depth control. All tasks that use navigation data as feedback for motion control use one or both of these controllers. The yaw axis controller is simple and is encapsulated in the class `CScalarPID` found in `MOOSGenLib` . The output of the controller is the task "vote" on `DESIRED_RUDDER` .

1.4.1 Scalar PID

The scalar PID has the structure shown in Figure 3. It has integral limits to prevent integral wind up and is also output limited to bound the output values. Hence three gains and two limits specify the characteristics of the controller.

It is important to note that the controller does not require that it be run at precisely regular time intervals. Instead it uses the sequence of time stamps on the input navigation data to perform the differentiation and integration as required. The differentiation is performed using a 5-tap FIR filter. This leads to smooth performance. However, no phase advance is performed – a point that could be improved upon.

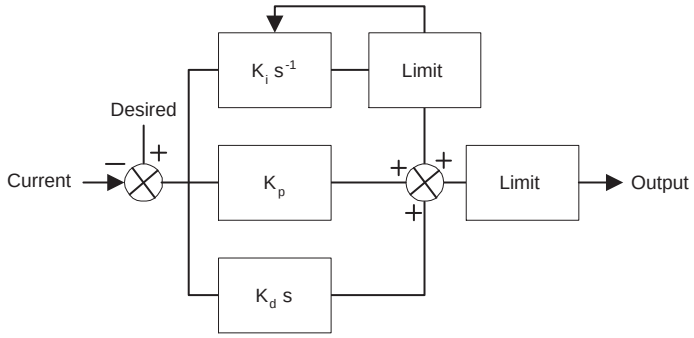

Figure 3: The structure of the scalar PID

1.4.2 Vertical Control via Pitch Control

The control of motion in the vertical direction is only applicable to the subsea case. Here we choose to control depth via pitch. The overall controller is fourth order and consists of an inner scalar PID loop controlling vehicle pitch. This loop maps error in pitch to desired elevator. The set point for the pitch loop is derived from the error in the vertical direction [1]. This topology is shown in Figure 4.

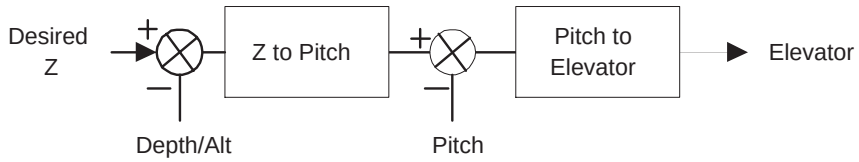

Figure 4: The structure of Z axis dynamic controller

1.4.3 Track-line Control

The `TrackLine` task is an interesting case study of the use of the dynamic controllers. It is derived from the `WayPoint` task which uses yaw control to drive in a straight line to a goal position. The `TrackLine` is more sophisticated in that it draws the vehicle onto a line defined between a start and goal point and then heads for the goal point along the track-line direction. Beneath the hood the task is continually changing the goal coordinates of the underlying way-point task – a "carrot and donkey" approach. At each time step the goal coordinate is set to be the projection of the vehicle location onto the track-line plus some "lead distance" along it.

[1] Note altitude control and depth control have sign inversion between them. Depth is in the negative Z direction.

1.4.4 The Independence Assumption

Implicit in all this is that the dynamics in the XY plane and Z axis are decoupled. Clearly this assumption may be too strong for some vehicles[2].In this case XY controlling tasks will also need to place a vote on elevator as well as rudder commands. This is no big problem, but they will need to be extended to control in the vertical plane at the same time. In other words, such a vehicle will not be able to tolerate the division of tasks into XY and vertical control types as is currently the case.

[2]Although two AUVs have been successfully controlled using this strategy.

2 Navigation – pNav

The pNav process is perhaps the most complex of all the MOOS processes. Its job requirement is simple –

> *"taking asynchronous, anachronistic and inconsistent input from sensors, provide a single, up-to-date estimate of vehicle pose and velocity.".*

The most important output of pNav is the NAV_* (pronounced "Nav Star") family where the wild card is any one of *X, Y, Z, DEPTH, PITCH, ROLL, YAW, SPEED, X_VEL, Y_VEL, Z_VEL or YAW_VEL.* Note that velocities are in earth coordinates not body coordinates. The NAV_* family should be taken to be the best estimate of vehicle state. Indeed the Helm application uses this family to deduce actuation commands.

2.1 Priority Queues

pNav allows the route by which NAV_* information is derived to be specified in its control block. This is achieved by the use of Priority Queues or Stacks. The idea is simple – each NAV_* output has its own line in the configuration block. Reading from left to right specifies the preferences for deriving state estimation. For example, it might be preferable to have an AUV use pure GPS fixes for X,Y navigation when on the surface. In this case the GPS_X and GPS_Y sensor outputs are mapped straight through to NAV_X and NAV_Y and the first entry on the RHS of the "X" line in the pNav configuration block would be "GPS@TIMEOUT". Here the value of "TIMEOUT" would be the maximum time between GPS fixes that could be tolerated before assuming the sensor has stopped publishing valid data. For example, if no GPS_X data arrived for TIMEOUT seconds then the GPS sensor would no longer be fed through to NAV_X and the navigator would shift attention to the next "SOURCE@TIMEOUT" pair in the priority queue. A likely second entry would be DVL . When the vehicle dives GPS fixes will cease and it may be desirable to dead-reckon using a DVL in its place. In this case, the second entry would be "DVL@TIMEOUT". As soon as the vehicle surfaces and GPS fixes resume, the stack/queue would "pop" and NAV_X would once more derive itself from GPS data. Figure 5 gives an example definition of navigation priority queues.

Of course this scenario assumes that the DVL sensor can produce DVL_X/Y measurements. It is more likely to produce reliable body velocity measurements. Now we must ask how to derive position estimates from sensors that do not measure positions. The pNav process solves this problem by providing two kinds of real-time navigation filters: a ten state non-linear Kalman filter and a five state non-linear Least Squares filter. Collectively these filters provide the following functionality:

- Derivation of rate from position sensors
- Derivation of pose and rate from LBL ranges
- Optimal fusion of rate and position measurements
- Estimation of tide-height

```
////////////////////////////////////
//   routing priority stack
////////////////////////////////////
X       = GPS @ 2.0, EKF @ 2.5  ,LSQ @ 5.0,
Y       = GPS @ 2.0, EKF @2.0   ,LSQ @ 5.0,
Z       = EKF @ 2.0  ,LSQ @ 5.0,
Depth   = EKF @ 2.0  ,LSQ @ 5.0,DEPTH @ 1.0
Yaw     = EKF @ 2.0  ,LSQ @ 5.0, INS @ 1.0
Pitch   = INS Speed  = EKF
```

Figure 5: Specifying NAV_* derivations. Here for example NAV_X is derived preferentially from GPS (i.e. straight from GPS_X) unless no GPS data is received for 2 seconds. In this case EKF_X is used in its place. If the EKF (extended Kalman filter) fails to produce any output for 2.5 seconds then as a last resort LSQ_X is used (non-linear least squares). However, as soon as GPS data begins to be delivered once more the system switches back to using GPS – popping the stack.

- Automatic robust rejection of outliers
- Self diagnostics

These navigation filters reside in the MOOSNavLib library, which is linked with pNav. They are discussed in greater detail in Section 2.2.

2.2 Stochastic Navigation Filters

The navigation filters are moderately complicated software components and require a good understanding of stochastic estimation — the full extent of which is beyond the remit of this paper.

There are two kinds of navigation filters used for online navigation: a non-linear Kalman Filter (EKF) and a non-linear Least Squares Filter (LSQ). [3]

Least Squares Filter(LSQ) The LSQ filter is a conventional non-linear least squares filter using the Newton-Raphson iteration scheme. It can only produce estimates of pose (no rate states) and process measurements that are proportional to pose (no velocity measurements). Internally measurements are accrued (up to a certain time span) until the state space becomes observable, at which point a solution is found iteratively by successive linearizations. The accumulation of observations then begins again until the next estimate can be formed. Heuristically speaking, the LSQ filter "starts from scratch" after every published estimate and has no concept of the flow of time across successive estimates. This implies that it is not easy to use measurements of rate in deriving pose estimates. On the other hand, it does mean the LSQ filter is robust and immune to errors in previous estimates.

[3] The top down calibration filter within MOOS is used for automatic calibration of an LBL net using GPS and acoustic ranges and is not considered to be online.

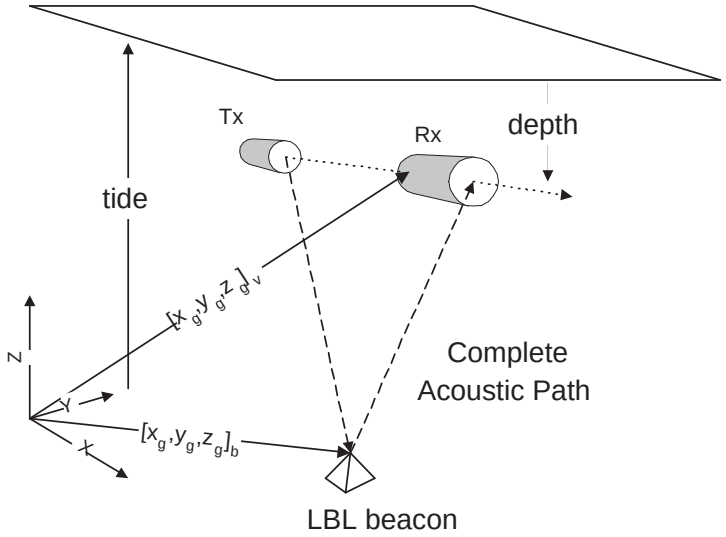

Figure 6: The relationship between tide, Z and depth. The navigation filters understand LBL time of flights as a function of two vehicle positions (to be estimated) and the beacon's location (known).

Extended Kalman Filter (EKF) The EKF is the most complex of the two filters and is also the most versatile and accurate. It fuses predictions derived from a simple vehicle model with measurements supplied by sensors. Internally it uses a fine discretization of time to faithfully interpret and model sensor measurements. The filter is recursive and so, heuristically speaking, possesses a concept of time. This enables it to produce estimates of velocity in addition to pose using pose only measurements.

The filters are designed to be fully configurable in an intuitive way from text in the `pNav` configuration block in the mission file. A typical `pNav` block is given in the Appendix. This section will use this listing as an example and discuss the meanings and use of some important constructs within it.

2.2.1 Defining Sensors

The filter needs to know what sensors are in use and their geometry with respect to the vehicle's coordinate frame – this is achieved with the `SENSOR_*` keyword where "*" is any of XY, LBL, ORIENTATION, DEPTH, BODY_VEL or ALTITUDE. The general syntax is as follows:

$$\texttt{SENSOR\_*} \quad = \textit{iSource} \rightarrow \textit{sensorname} \ @ \ x,y,z,[twist] \quad \textit{noise}$$

Anthropomorphically, the syntax can be read as:

> Declare a sensor of type (*) which is managed by a process called *iSource*. Call it *sensorname* and understand it to be located at the vector (x, y, z) in body coordinates with a twist of *twist* about the body z axis (useful for compass offsets). Expect the sensor to produce data corrupted by noise with standard deviation *noise*.

Note there is no limit on the number of sensors that can be used. For example, you can have two GPS sensors (XY). In fact on a large vessel this will allow estimation of yaw without an orientation sensor — for "free".

2.2.2 Mobile and Static Vehicles

The EKF allows the vehicle to be defined as static or mobile. A mobile vehicle includes velocity (x,y,z and yaw) estimates in its state estimate. A static vehicle has only pose estimates. The LSQ filter implicitly uses a static vehicle model as it has no concept of prior history and recalculates fixes from scratch whenever possible.

2.2.3 Defining Vehicle Dynamics

The EKF uses a linear, constant velocity dynamic model for the vehicle. It assumes that the vehicle will continue to move in a straight line with constant velocity across time steps. There is however a noise model associated with this model that dilutes the precision of the state estimate over time (in the absence of data from sensors). The degree to which this happens is governed by the `EKF_*_DYNAMICS` variables, which vary from 0 to 10. A setting of zero implies the vehicle is immutable in that direction and never changes. For example, a planet-sized vehicle might have a setting of zero for its yaw dynamics. A setting of 10 implies that the vehicle is very mobile in a given degree of freedom, and in the absence of observations from sensors over a few seconds we expect to have a large uncertainty in our state estimate. Essentially these numbers are parameterizing the expected error in our state model. Heuristically, a larger number means that we are not so sure the vehicle always obeys a constant velocity model.

Note that we do *not* inflict the constraint that the vehicle translates along the direction it is pointing (although when moving at speed this may be the case). Intentionally the vehicle model does not couple angular and cartesian degrees of freedom. Of course that is not to say that models explaining sensor data do the same — in particular the use of DVL body velocity observations correlates yaw and position estimates.

2.2.4 Start Conditions

The LSQ filter can be used to initialize the EKF. If however the EKF is being run in stand-alone mode, it needs an initial guess to start up. The initial state guess is provided in the mission file via the `EKF_*` and the uncertainty (1 standard deviation bound) by the `EKF_SIGMA_*` variables. If the vehicle is type "mobile", velocity estimates are internally initialized to zero with a suitable uncertainty.

2.2.5 Logging

The performance of the navigation filters can be logged to file for post-mission analysis[4]. The location and stem name of these log files can be specified using the `NAV_LOG_*` variables. If time stamping is required, the file name includes

[4]The `MOOSMat` matlab script is designed to do this graphically.

the creation time of the log in standard MOOS format. Two kinds of files are created: **.olog* and **.xlog*. The former logs all observations presented to the filter and the outcome of their processing — rejection/acceptance etc. The latter logs the evolution of the state estimate and its covariance. Both files are in text format and intended for simple quick parsing in Matlab.

2.2.6 Data Rejection

Both the EKF and the LSQ filters possess moderately sophisticated machinery, which allows them to discriminate between good and out of bound (outliers) sensor data.

The LSQ filter builds a temporal history of sensor data and applies some robust statistics tests to it to identify outliers. The underlying assumption here is that sensor noise is high frequency whereas the vehicle motion and hence reliable sensor data are low frequency processes. The `CSensorChannel` class tries to find the best (in terms of consensus) linear relationship between recent sensor data points. This done, it is able to identify outliers and remove them from the input stream feeding into the filter. The `LSQ_REJECTION` entries specify sensors (and acoustic channel in the case of LBL) which should be piped through this process. Each such statement declares the number of data points to be analyzed (the extent of the temporal history) and tolerable/expected deviation of point from the fitted line. In the case of LBL sensors, this is a time measured in seconds.

The EKF filter is a little smarter and uses its current state estimate and uncertainty to identify that the current crop of observations are not mutually consistent either with themselves or with the last state estimate (a common technique in Kalman filtering). However, having detected that at least one of the current sensor measurements makes no sense it employs geometric projective techniques to identify the true outlier.

2.2.7 EKF Lag

The EKF can be told to run "behind time". That is, only process sensor data up until η seconds ago. Having done this, the EKF then forward predicts to the current time to produce an external navigation estimate (which is used to make dynamic control decisions). The next iteration will use the "un-altered" internal estimate as a prior (i.e. the η second old one). One might ask "why bother?". The reason stems from the fact that sensors cannot (should not) be relied upon to produce up-to-date measurements. Typically they are always out of date by the time they are processed. Things are exacerbated in the case of LBL data processing. Envision the case where an LBL transceiver ranges to two beacons, one of which is close, say 150m away, with a reply delay of 0.1 seconds, and the other one is distant, say 1.5km away, with a reply delay of 1 second. The reply from the first beacon will come in 0.3 seconds after the interrogation but the remaining beacon's reply will not be heard for 3 seconds. Only when all replies are in (or the receive window has timed out) will the time of flight be transmitted by the sensor hardware. This means that by the time the data is processed it is at least 3 seconds old. Now, to properly "explain" these time measurements the filter must use the state estimates of the vehicle valid at the time-stamp of the data — in this case 3 seconds ago. Hence the filter needs

to run behind time by an amount specified by the `EKF_LAG` parameter. The filter buffers all sensor data in a queue (which is self-sorting on time stamp) and extracts data from it valid at the current time minus whatever the lag is set to.

2.2.8 Fixed Observations

It is possible to force a constraint on the navigation by declaring persistent observations. Every time the filter "ticks" these artificial observations are added to a list of observations stemming from real sensors. This, for example, would allow a surface craft with no depth sensor to use LBL navigation by declaring a zero depth artificial observation. Another case is when a heading measuring device is not available but a constant heading observation can be added to take its place. (Recall that the vehicle model does not correlate heading and position, so with a constant heading observation the vehicle will simply translate).

2.2.9 LSQ/EKF Interaction

If both the LSQ and EKF filters are run at the same time, the LSQ filter can be used to initialize and monitor the performance of the EKF. The fact that although they have lower precision LSQ estimates have no dependency on prior estimates can be exploited. This can be advantageous for several reasons.

Robustness A poorly tuned/setup EKF can diverge — that is, produce a state estimate so far from the truth and with sufficient confidence that it rejects all incoming sensor data. There is no reason that this should happen given the data rejection schemes employed; however, navigation is stochastic and unlikely things will happen given time. A deployed system should be robust to such occurrences and the checking of estimates against the LSQ filter offers one (of several employed) way to do this. If the estimates emerging from the LSQ differ significantly from those emerging from the EKF over a sustained interval `pNav` takes the EKF off-line and attempts to re-initialize it to the LSQ estimates. We require a lack of consensus over several seconds/fixes to prevent spurious LSQ fixes pulling the plug on the EKF prematurely. This also accords with the observation that once the EKF has diverged it is unlikely to recover, and so waiting a few seconds is not going to change anything.

Observability The LSQ estimate essentially solves a nonlinear set of equations to derive state observation data. No state estimate can be derived until the system of equations is observable — i.e. enough of the right kind of measurements have been accrued. If no LSQ solution can be produced for an extended period, it provides *prima-facie* evidence that the navigation problem is or has become (via sensor failure) unobservable and unsolvable. Note that the EKF does not require full observability to operate — unobservable states just become more uncertain with each iteration[5]. The navigation control block allows a LSQ timeout to be specified. If no LSQ fixes are derived for a time exceeding this, an navigation failure is declared and all filters are taken off-line. This mechanism has two applications. Firstly, it prevents an autonomous mission from being started on

[5] Until specified limits are reached, at which point the filter is taken off-line.

a vehicle that cannot be navigated. Secondly, it serves to monitor sensor failure and prevent motion control decisions being taken on the basis of at least partially "predict only" EKF navigation estimates.

Automatic Initialization The LSQ does not need a prior to produce a state estimate. Hence it does not need to be primed with an initial guess. Hence once a stable/repeatable stream of solutions begins to flow from the LSQ filter they can be used to initialize the EKF, removing the need for approximate starting conditions to be specified.

2.2.10 Hidden State Estimation

Both the EKF and the LSQ estimate the tide height — the distance between the XY navigation plane and the surface. The EKF can optionally also estimate a bias term in an orientation sensor (one bias is applied to all orientation sensors). Great care has to be taken in ensuring that this term is in fact observable given the system configuration — heading bias estimation should only be enabled when operating with a body velocity sensor *and* a position sensor.

2.2.11 Navigation Failure

Limits can be placed on the permissible navigation uncertainty. If the one sigma bound of the estimates exceed these limits a *Navigation Failure* is declared. All relevant filters are taken off-line and the `EndMission` variable is published. This is the last line of defence against navigation failure — if this happens things have gone badly wrong and the mission should be terminated. The EKF also watches the numerical value of the pose derivative states. If they exceed sensible limits (10m/s or $\pi rads^{-1}$) the states are reset to zero. This is essentially a coordinate shift and so is statistically consistent although somewhat alarming. Accordingly a warning is issued. This condition is often experienced at boot time when a large velocity is inferred to explain the apparent shift in position from initial guess to that of the first EKF derived estimate.

2.3 Sensible Configuration and Commissioning

This section is intended as a brief guide to actually using the navigation filters and owning process `pNav` . It is not exhaustive and cannot replace experience and understanding of the underlying techniques.

2.3.1 Defining the Navigation Frame

Different groups of people like to define coordinate origins in different places, in particular the $Z = 0$ plane. In an area with little or no tidal flux the use of an artificial tide observation (tide = 0) can be used to fix the $Z = 0$ plane to the sea surface. All beacon locations will need to have negative Z coordinates[6].

2.3.2 Heuristic Hints

The following is a selection of heuristic statements that are helpful to keep in mind during commissioning and verifying the navigation component of MOOS.

[6]However depth will still be positive as depth = tide-Z.

- If rate states are often being reset, something is wrong. Poor data is being accepted when it should not be.
- The LSQ filter cannot use velocity sensors (it has no idea of history).
- The LSQ filter cannot be used underwater without an LBL net – there are no sensors that measure a quantity proportional to position.
- Increasing vehicle dynamics settings will cause more observations to be accepted but reduce resilience to bad sensor data.
- Decreasing vehicle dynamics will lead to smoother (piece-wise) trajectories but may cause sensor data to be erroneously rejected during swift manoeuvres.
- Increasing estimated sensor precision (in the sensor declaration) will tend to cause more data to be accepted and increase its effect on state estimates. Accepting bad data will increase the chances of the state vector being corrupted to the degree that no data is ever accepted again.
- Try to keep the `EKF_LAG` setting as small as possible. If set too small, LBL data will not be used. If set too large, the final prediction step used to bring the estimate forward to the current time will be inaccurate – bad news for dynamic control.
- The `EKF_LAG` can be very small if no LBL sensing is used.
- Tide estimation can only be accomplished if both depth sensors and LBL data is used (with beacons on the sea bed).
- Tide estimation cannot be used using only altitude and LBL measurements as such a scheme would require a model of the sea bed to explain the altitude data.
- **Always** analyze the navigation logs before performing a long mission with a new sensor configuration or LBL array. Do not launch unless everything seems fine. There are numerous safety features built in but it could be several minutes before problems are detected and the navigator pulls the plug and emits the navigation failure flag (which should be monitored by the `EndMission` task).
- If the EKF is being run by itself (i.e. not booted from the LSQ) make sure that its start coordinates and uncertainties are suitable to allow it to start accepting data when it starts up – i.e. close enough to its true position!
- It is a good plan to set the initial uncertainty in heading to be large. Otherwise orientation data may not be accepted.

Table 2: A summary of common task functionality

Task	**Use**
`TimeOut`	Issues `FinishFlags` after timeout. This is useful for creating a pause between tasks – for example a delay at mission start.
`GoToWayPoint`	Go to a specified XY location. A tolerance can be specified to stop un-ending hunting. Requires X,Y and Yaw data.
`ConstantDepth`	Level Flight. Requires only depth data and pitch data.
`ConstantHeading`	Drives the vehicle at a constant heading. Requires only yaw data.
`XYPattern`	Repeats a pattern of way-points. Requires X,Y and Yaw data. The number of repetitions required can be specified.
`ZPattern`	Performs a series of depth set points. Requires depth and pitch data. For example combining this with a orbit task will execute a helical pattern.
`GoToDepth`	Spiral to a given depth and exit. Requires Pitch and Depth data.
`LimitAltitude`	Fire event flags if too close to sea bed. A Safety task requiring Altitude data. This has saved the vehicle several times.
`LimitDepth`	Fires event flag if the vehicle is too deep. A Safety task that is usually run with timeout=NEVER.
`EndMission`	Abort Mission (highest priority). Set to be the highest priority task (0). Locks out all other tasks. Usually commands all actuation to zero.
`DiveTask`	Dive from surface (e.g reverse dive).
`Orbit`	Orbit a given location in XY plane. The direction, radius and number of orbits can be specified.
`Survey`	Perform a survey (mow lawn) centered on specified position with given rotation and extent.
`TrackLine`	Execute a linear path between two points.
`OverAllTimeOut`	Limit Total length of mission. Always use this task – it is compulsory. Its FinishFlag should be the StartFlag of an EndMission task.
`LimitBox`	Fire event flags if 3D working volume of mission is exceeded. A Safety task useful for trapping navigation failure.