Deliverable D 3.1
Report and Release of first cooperative world modeling software framework including 2D mapping and basic versions of 3D mapping

Contract number: FP7- 231378 Co3 AUVs
Cooperative Cognitive Control for Autonomous Underwater Vehicles

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number 231378.
## Identification sheet

<table>
<thead>
<tr>
<th>Project ref. no.</th>
<th>FP7-231378</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>Co3AUVs</td>
</tr>
<tr>
<td>Status &amp; version</td>
<td>[Final] &quot;1.0&quot;</td>
</tr>
<tr>
<td>Contractual date of delivery</td>
<td>month 18</td>
</tr>
<tr>
<td>Actual date of delivery</td>
<td>March 2011</td>
</tr>
<tr>
<td>Deliverable number</td>
<td>D 7.2</td>
</tr>
<tr>
<td>Deliverable title</td>
<td>Report and Release of first cooperative world modeling software framework including 2D mapping and basic versions of 3D mapping</td>
</tr>
<tr>
<td>Nature</td>
<td>Report</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU</td>
</tr>
<tr>
<td>WP contributing to the deliverable</td>
<td>WP7</td>
</tr>
<tr>
<td>WP / Task responsible</td>
<td>Jacobs</td>
</tr>
<tr>
<td>Editor</td>
<td>Andreas Birk</td>
</tr>
<tr>
<td>Editor address</td>
<td>Jacobs University Robotics, School of Engineering and Science Campus Ring 1, 28759 Bremen, Germany</td>
</tr>
<tr>
<td>EC Project Officer</td>
<td>Franco Mastroddi</td>
</tr>
<tr>
<td>Keywords</td>
<td>marine robotics, Simultaneous Localization and Mapping (SLAM), 2D registration, 3D registration, cooperative mapping</td>
</tr>
</tbody>
</table>

**Abstract (for dissemination):**

The first version of the mapping software from the Co3-AUVs project includes contributions to 2D and 3D mapping as well as cooperative mapping. Concretely, following achievements were made. A 2D spectral registration method was developed that is well suited for online registration of camera images for mosaicking as well as for sonar scans. The registration is embedded in pose-graph SLAM, which supports cooperative mapping very well. This includes the option of online cooperative mapping under the severe constraints of underwater communications. Furthermore, two methods for 3D registration were developed with one building upon previous own work that was carried over to the underwater domain and one being a completely new development using spectral registration. The different methods were tested in different settings and led to several related publications.

---

**List of annexes (if any)**

- TBD
Co$^3$-AUVs World Modeling
software overview, version 1.0

Max Pfingsthorn, Kaustubh Pathak, Sören Schwertfeger,
Ravi Rathnam, Heiko Bülow, Narunas Vaskevicius, and Andreas Birk

December 2010

Contents

1 Introduction 2
2 Pose Graph 2
3 Pose Graph Incremental Mapping 4
  3.1 Loop Detection ........................................ 5
  3.2 Pose Graph Optimization ............................. 5
4 2D Spectral Registration 6
5 Visualization, Interaction, and Storage 7
  5.1 ImageSLAM implementation overview .................. 7
6 Cooperative Mapping 10
7 3D Registration 12
  7.1 Planar-Patches based Registration .................... 12
     7.1.1 Planar Patches Extraction ......................... 12
     7.1.2 Scene Registration Using Matching of Planar-Patches 14
  7.2 3D Spectral Registration .............................. 15
1 Introduction

Figure 1 shows all software components of the 2D and 3D world modeling framework. The following report dedicates a section to each major component shown.

The central component is the pose graph, a state-of-the-art data structure for Simultaneous Localization and Mapping (section 2). Of further general importance is the construction of the graph using motion estimates and loop detection, as well as its iterative improvement through optimization (section 3).

Section 5 describes the interaction with the pose graph, the visualization of spatial (sensor or processed) data present in the pose graph, as well as its generic storage.

Subsequently, the cooperative construction of a pose graph with multiple robots is described in section 6.

Sections 4 and 7 describe a number of 2D and 3D registration methods that give rise to the metric information stored in the pose graph.

2 Pose Graph

Being the center of the framework, the pose graph data structure has been designed to be very generic and extensible. It focuses on representing the
Figure 2: Graph structure of observations and poses explained. Two nodes are connected by an edge iff their associated observations overlap.

graph structure of poses where sensor observations are made, as shown in figure 2.

A UML Class Diagram of the relevant implementation is shown in figure 3. The main PoseGraph contains a list of PoseGraphNode and PoseGraphEdge, along with various accessors and mutators. Each PoseGraphNode has a list of adjacent edges for efficient traversal of the graph. The PoseGraphEdge has pointers to the two nodes it connects. Additionally, the edge retains the specific transformation estimate between sensor data stored in the nodes produced by a registration. In order to ensure a good optimization result, the registration result should also report uncertainty information in the form of a covariance matrix, which is stored in the transformation object.

Any serializable data type (in terms of the serialization described in section 5) can be attached to a PoseGraphNode using a LocalizedAttachment. This wrapper class adds important meta information about the data, such as a type descriptor for easy retrieval and a transformation relative to the node pose which defines the coordinate frame of the data. For example, sonar sensors are rarely mounted in the center of the robot, and the LocalizedAttachment keeps track of exactly this offset of the sensor coordinate frame relative to the robot coordinate frame.

In addition, many different kinds of data can be stored in this PoseGraph.
so to query or retrieve the exact instance in a node, a type and instance specific ID is used to differentiate the attachments. For example, the aforementioned sonar data may always be stored with ID 1, while GPS readings may have ID 2. This ID is application specific, no general global set exists as of date.

A few utility classes are also provided: PoseGraphVisitor allows for generic traversal of the pose graph in arbitrary order, for example in order to visualize the structure or to generate secondary information, such as a minimum spanning tree. Currently, breadth first traversal and Prin’s minimum spanning tree traversal is implemented.

PoseGraphListener allows another class to receive change information from the main PoseGraph. This feature is mainly used for the synchronization of pose graphs between cooperating robots. Another implementation of this interface allows the spatial indexing of PoseGraphNode in an octtree for fast neighborhood searches.

3 Pose Graph Incremental Mapping

Without loops, the pose graph would only contain sequential registration results, shown as single arrows in figure 2. Only the new, seemingly redundant, information contained in the loop edges (shown as double arrows in the same figure) give rise to inconsistencies in the graph, mostly due to accumulated error in the sequential registrations. These inconsistencies can be removed by recomputing the transformations for each edge using numerical optimization methods.

Thus, the mapping process can be broken down into several sub-processes (figure 4):

- Sensor data acquisition
• Sensor data registration (section 4 and 7)
• Loop detection
• Optimization

![Diagram of the mapping process](image)

Figure 4: The mapping process

### 3.1 Loop Detection

In order to add more information to the graph and improve the overall accuracy, loop detection finds node pairs that are not already connected by an edge and do contain potentially overlapping sensor data.

Currently, loops are detected by searching for nodes in a neighborhood of newly added nodes. Additionally, after optimization and potentially changed global node poses, such a search may be run for all existing nodes.

To this end, an implementation of the PoseGraphListener is used, namely the PoseGraphIndexer. This class is notified of changes to the PoseGraph, such as new and changed nodes. Neighborhood searches are accelerated using an octree.

Another PoseGraphListener implementation listens to new nodes and automatically runs a search for potential loops within a given distance. Once potentially overlapping pairs are found, the associated sensor data is registered with a given registration method. If this registration process succeeds, an edge is created between the two nodes in question.

### 3.2 Pose Graph Optimization

Optimization methods for pose graphs are abstracted with the PoseGraphRelaxer (from the numerical optimization term relaxation) interface. This interface contains the main method:

```cpp
bool PoseGraphRelaxer::relax ( PoseGraph* )
```

Implementations will apply their optimization method directly on the given PoseGraph instance and return whether convergence was achieved or not.

PoseGraphRelaxer defines a few more methods that are optional for the implementations, namely to facilitate online optimization. This is useful for interactive display of the optimization process, and, given that the method supports it, for continued optimization of the graph while it is built.

The currently available optimization methods are:
• **TranslationOnlyRelaxer**: Our method to only optimize the translational part of the transformations in the graph in closed form. This is especially useful for situations where the rotational registration is usually very precise, as in the case with plane-based registration (section 7.1) [1].

• **TreeParamOlsonRelaxer**: Based on the well-known TORO library\(^1\) [2]. This method optimizes each edge individually, ordered by level in a minimum spanning tree. It has very good global convergence characteristics.

• **SLOMRelaxer**: Based on the SLOM library\(^2\) [3]. This method uses well-known weighted least-squares methods, such as Levenberg-Marquardt, to optimize the whole graph at once each iteration.

If rotational error is low, the **TranslationOnlyRelaxer** is the best choice. Otherwise, the use of **TreeParamOlsonRelaxer** is recommended.

### 4 2D Spectral Registration

There exist several versions of the 2D Spectral Registration. They are implemented either in MatLab or C++. They can be applied to (underwater) images, supporting scale, rotation and translation, or for scan matching, for example of sonar data. The scan matching does not support scale since this is already known. This makes the results of the scan matcher more robust against noise and other errors.

The 2D registration, its applications, and different evaluations are described in depth in several publications [4, 5, 6, 8, 9, 10].

In the following some implementation details of the C++ version of the registration algorithm used in the 2D image SLAM program described in section 5 are presented.

The **FMIRegistration** for the **ImageSLAMmapper** stores the data of the images to be registered as **FMIImage**. This class not only holds the intensity data but also intermediate results of the registration process can be stored. Through the re-use of those results significant amount of computation time can be saved. Nevertheless, in order to save precious memory, those intermediate matrices can also be freed. This is transparent to the actual registration algorithm. The data cached in this way is the phase matrix and the polar logarithmic phase. It should also be mentioned that this data is saved taking the symmetries of phase information into account, using only roughly half the memory compared to storing the full matrix.

\(^1\)http://openslam.org/toro.html
\(^2\)http://openslam.org/slom.html
The `ImageSLAMmapper` thus actually saves the `FMIImage` for the current frame such that it can be re-used in the next iteration. After that iteration the (now useless) `FMIImage` is freed. When searching for loops though, the `FMIImage`s are stored in the graph to speed up the computation.

Another neat feature of the implementation is the `FFTMath` class which, depending on a compile switch, transparently compiles, links and wraps around either the GNU Scientific Library (gsl) or fftw [11] for the Fast Fourier Transform functions.

Whenever useful, precomputed lookup-tables are used to speed up the algorithm. Those are used for the 1D Gaussian window, the 2D spectral windowing and the polar-log-interpolation-table. The latter uses the symmetry of this matrix to take just one forth of the full-sized memory, which can also speed up calculation by avoiding cache-misses.

The FMI algorithm has the disadvantage to only be able to register rotations of up to 90° in each direction. If an application expects bigger rotations (e.g. while doing loop-detection) an additional step is performed that correctly detects rotations over the full 360° range at the price of additional computation.

The result of a registration can be analyzed in different ways. The uncertainty can be stored in an uncertainty matrix, giving such values for the scale, rotation and translation. Other methods to assess the validity of a match are to compute the signal to noise ratio or to compare the energy of the selected peak with the energy of the next powerful peaks.

### 5 Visualization, Interaction, and Storage

A graphical user interface to the 2D image SLAM solution is implemented using the Qt framework [12]. Figure 5 shows an overview of the data-flow in the application.

#### 5.1 ImageSLAM implementation overview

The applications main class is `ImageSLAMGui`, which is instantiating all the widgets, the `VideoSource` selected and `ImageSLAMmapper`. This class is mainly the glue that puts together the different parts of the actual mapping algorithm like `PoseGraphMapper`, `NetworkImageUndistorter` (Network- because this is a format which can be immediately and efficiently send over the Network - this feature is not used in this application), the `FMIimageMatcher` (the name of the ImageRegistration class), the `TreeParamOlsonRelaxer2D` and the `TiledQImage`.

There are two threads in this application. The main `QtThread` and the thread of the `VideoSource` selected. The `ImageSLAMmapper` and the `VideoPlayerWidget` register with the `VideoSource` with a callback function to
get informed of newly arriving frames. While the VideoPlayerWidget just gets a reference to the frame and sends a signal to the Qt thread, the ImageSLAMmapper is doing the following computations still in the (called-back) thread of the VideoSource.

First the frame gets undistorted in the NetworkImageUndistorter. This undistorted, square frame is then saved in the newly created current PoseGraphNode and additionally used (together with the previous frame) in the FMIimageMatcher. The matcher then saves the computed transform (as the class Transform) between the former and current PoseGraphNode in the PoseGraph. The global pose of the current PoseGraphNode is calculated in the PoseGraph and then used to paint the current frame on the according tiles of the TiledQImage, taking the scale, rotation and selected alpha-blending algorithm into account. Then the Qt thread is signaled that the content of the TiledQImageWidget potentially has to be updated.

In our serialization framework classes that are to be stored (or send over the network) implement the Storable interface. In this application this is done for NetworkImage, Transform, all parts of the PoseGraph and other (in this context) minor classes that can be attached to the PoseGraph. The ImageSLAMmapper has the option to store the undistorted frames (NetworkImage) received from the VideoSource in one file (.raw). Those can be replayed using VideoStorage. Additionally this class can also replay a consecutively numbered series of image files from a folder. For both options parameters in the config file define the replay-speed and if or how many frames are to be skipped.

Figure 6 shows a screen-shot of the GUI. On the top right the VideoPlayerWidget with the current frame is visible. The zoom-able map, namely the TiledQImageWidget is in the lower half. Above it are the sliders of the RangeSelectorWidget that can be used to display just a part of the map (only useful if the mapping has stopped). Above this are the controls for the LoopDetectorWidget. The sliders of the RangeSelectorWidget can be used to manually find two overlapping frames. A click on “Loop found: Add Edge” will then try to add an additional edge between the two selected frames. For this it will use the FMIimageMatcher which in this case will then run in the Qt thread. An automatic loop-detection method and the relaxation process can be triggered with the other two buttons. Both of these algorithms are also executed in the Qt thread.

On the top left the other options for the GUI can be found. The “Go”/“Stop” button starts and stops the mapping, “Clear Map” deletes the contents of the TiledQImage while “Save Map...” saves the TiledQImage as one image. The PoseGraph can be saved and loaded with the according buttons. Other options are the activation/ stopping of the saving of the live
Figure 5: ImageSLAM data-flow chart
video-feed as raw data and to replay a video (both described above). In certain circumstances it might be advisable to constrain the algorithm to a certain height which can be done using the “Constant Scale” button. The “Update Options” allow to turn off the live display of the map (basically pause the update of the RangeSelectorWidget - in case a high live frame rate is needed) or to disable the saving of the frames in the PoseGraphNode (to save memory - of course use of the RangeSelectorWidget will not be possible).

Figure 6: ImageSLAM GUI

6 Cooperative Mapping

The pose graph data structure is ideally suited for cooperative mapping [13, 14]. A single PoseGraph structure can contain many disconnected subgraphs, potentially one per robot. What is usually referred to as map-merging between multiple robots can be thought of as loop closing across
Figure 7: A multi-robot pose graph structure. Loop edges are drawn as thick lines. Blue edges connect the previously separate robot maps.
robot maps.

Figure 7 shows a team of two robots and the corresponding pose graph. First, before the robots cross path, the two maps are two disconnected components in the graph. Loop edges that connect two nodes from the same robot do not change this structure. Once overlap is identified between the two robot maps, loop edges (shown in blue) between the two disconnected components give rise to a combined graph and map.

The only constraint is that updates to the local graph have to be broadcast to all robots in the team. However, this may be done in bulk at a time when a connection exists. From a slightly more general perspective, this process can be viewed as a synchronization of two PoseGraph instances. This is implemented in the PoseGraphSynchronizer.

Specifically, each PoseGraphNode and PoseGraphEdge have a UUID attribute which is a tuple consisting of three parts:

1. The platform ID specifies the robot that produced this UUID.
2. The generator ID identifies the specific generator on the robot that produced this UUID.
3. The last part is unique to this UUID object given the previous two IDs and is produced by a counter in the specific generator.

Using this UUID, it is possible to identify all nodes and edges in the graph uniquely across robots and allows for elaborate synchronization methods.

Additionally, updating a pose graph over a network is much more efficient than other map representations, especially the commonly used occupancy grid [13, 14]. In an underwater setting and reasonable bandwidth assumptions, a pose graph can be shared among many robots even during the acquisition of a high-resolution image mosaic. The insight here is that only new edges have to be broadcast in order to maintain a usable pose graph. The attached sensor data is only needed when edges between robot subgraphs are generated [14].

7 3D Registration

7.1 Planar-Patches based Registration

The planar-patches based 3D registration module consists of two major parts, namely, planar-patches extraction and matching. The UML class diagram of the module is shown in figure 8, where the block arrows indicate the data-flow at run-time of the module.
Figure 8: Plane Registration UML diagram
7.1.1 Planar Patches Extraction

As it can be seen from the diagram (figure 8), a 3D-sensor sample is represented by a `ImagePointCloud` object, which contains a list of 3D points in the sensor frame. The point-cloud is further processed by the planar-patches extraction module. `PlaneFitterFactory`, based on the module configuration, creates an instance of a class implementing `PlaneFitterInterface`, which is responsible for segmenting a point-cloud into planar regions. `PlaneFitterInterface` has been implemented by the `RegionGrowing` class, which is based on the algorithm presented in [15]. This algorithm returns a list of `ImagePointCloudRegion` objects. Each of these objects describes a planar region in the point-cloud and contains parameters of the corresponding plane along with their uncertainty information in `UncertainPlane` object.

The segmentation step does not provide boundary information for the extracted regions. This is done by the objects implementing `OutlinerInterface`. There are several different concrete implementations of this interface, e.g. the `AlphaShapesOutliner` using the CGAL\(^3\) library. Some of these implementations have been described in [16, 17, 15]. The polygonalization step uses the segmentation result to create a `PlanarPatchesCloud` object by finding boundaries for each of the supplied `ImagePointCloudRegion` objects.

The final output from the planar-patches extraction module is a list of `Outline` objects which consist of polygons with corresponding plane-parameters and their covariances.

7.1.2 Scene Registration Using Matching of Planar-Patches

After extraction of planar-patches from the point-clouds of a 3D-sensor sample, we have a compact representation of the scene’s essential geometry. The next step is to do scene-registration, i.e. estimate the vehicle’s 3D-motion (6 degrees of freedom) between two 3D-sensor samples by matching the planar-patches to determine their most geometrically consistent correspondences. This work was begun in a DFG-funded project for land-robots operating in uncertain environments and has been explained in [18, 1] and further developed within the current project [19]. The challenge now was to extract and match the planar-patches from the very noisy data typically obtained from a 3D-SONAR.

As shown in the UML diagram, the plane-matching module works on a pair of `PlanarPatchesCloud` objects, each of which contains a list of planar-patches extracted from one 3D-sensor sample. The general `PlaneCloudMatcherInterface` has been implemented by the concrete MUMC (Minimally Uncertain Maximal-Consensus) algorithm [18]. The outputs of the algorithm are:

\(^3\)www.cgal.org
1. **Correspondences** This is a list of pairs of planar-patches: the first element of the pair belongs to the first \textit{PlanarPatchesCloud} object and the second element to the second \textit{PlanarPatchesCloud} object respectively. A correspondence pair signifies that these two patches observed in the two sensor samples actually correspond to the same physical patch. These correspondences are found by the \textit{MUMC} algorithm by minimizing an uncertainty metric over a pruned set of feasible correspondences.

2. **UncertainTransform** Based on the fixed patch-correspondences, a 3D-transform, consisting of a 3D translation and a 3D-rotation (represented by a unit-quaternion) is computed by closed-form least-squares. An attractive feature of our approach is that the covariance matrix of this transform can also be simultaneously computed, which is indispensable for the pose-graph optimization step.

3. **PlaneCloudMatchingStats** These consist of the observed matching-parameters which can be used to tune thresholds, and also observed run-times for statistical comparison.

### 7.2 3D Spectral Registration

The theoretical background of the spectral registration of 3D sensor data is described in detail in [20]. The method is based on decoupling translational from rotational registration by processing the magnitude of 3D spectral data using special projections to derive descriptor functions which allow a subsequent registration of yaw, roll/pitch and translation, as shown in figure 9.

After scan recording depending on the sensor type the data needs to be converted to a discrete grid. The grid size and the ratio of the used metric must be equal for all dimensions. A further preprocessing is the use of a 3D spectral window function before the Fourier transformation.

The initial step is the 3D Fourier transformation of the grid data. In a first registration step the yaw angle is determined generating a descriptor by a projection in spherical coordinates. After the spectral data is aligned corresponding to the yaw angle, the second step is the registration of the roll/pitch angle using a rectangular projection to generate a second descriptor. Here the angles are determined as simple x,y translation within this descriptor. Finally, the scan is re-rotated according to roll, pitch and yaw. The remaining step is the registration of the x, y and z translation by 3D phase matching.
Scan recording

Scan pre-processing
- convert to discrete grid
- 3D spectral windowing

Yaw registration
- Spherical resampling of spectrum
- Rotational registration of resampled descriptor

Roll/Pitch registration
- Align magnitude of spectrum according to determined yaw
- Rectangular resampling of spectrum
- 2D registration of resampled descriptor

Translation
- Re-rotate Scan according to determined roll, pitch and yaw
- Determine x, y and z by phase correlation

Output of 3D translation (x, y and z)
and
3D Rotation (roll, pitch and yaw)

Figure 9: 3D Registration
References


