# WOLF: A modular estimation framework for robotics based on factor graphs

Joan Solà*, Joan Vallvé*, Joaquim Casals, Jérémie Deray,
Médéric Fourmy, Dinesh Atchuthan, Andreu Corominas-Murtra and Juan Andrade-Cetto

***Abstract*—This paper introduces WOLF, a C++ estimation framework based on factor graphs and targeted at mobile robotics. WOLF can be used beyond SLAM to handle self-calibration, model identification, or the observation of dynamic quantities other than localization. The architecture of WOLF allows for a modular yet tightly-coupled estimator. Modularity is enhanced via reusable plugins that are loaded at runtime depending on application setup. This setup is achieved conveniently through YAML files, allowing users to configure a wide range of applications without the need of writing or compiling code. Most procedures are coded as abstract algorithms in base classes with varying levels of specialization. Overall, all these assets allow for coherent processing and favor code re-usability and scalability. WOLF can be used with ROS, and is made publicly available and open to collaboration.**

***Index Terms*—Sensor Fusion, SLAM**

## I. INTRODUCTION

STATE estimation is key in many complex dynamic systems when it comes to controlling them. In robotics, and especially for robots such as humanoids, legged robots or aerial manipulators, this is of special importance because of their high dynamics, inherent instability, and the necessity of entering in contact with the environment. This estimation includes the robot states at high rate and low lag, important for control, but also and crucially the environment, necessary for motion planning and interaction. Since the formalism of simultaneous localization and mapping (SLAM), which tackles both sides in a unified way, robot state estimation has evolved enormously and now reaches a level of complexity that goes well beyond the SLAM paradigm, tackling aspects such as sensor self-calibration, geo-localization, system identification, or the estimation of dynamic quantities other than localization.

With the increase of real and complex robotic applications of very diverse nature (2D, 3D, wheeled, legged, flying, submarine), the problems of integration, modularity, code re-usability, scalability and the likes are becoming real challenges that demand solutions. The issue then is in finding appropriate software architectures able to handle such variety in a practical way. Building such an estimation system requires the adoption of a number of design decisions affecting both the way the estimator works internally and the way this is used by developers and practitioners.

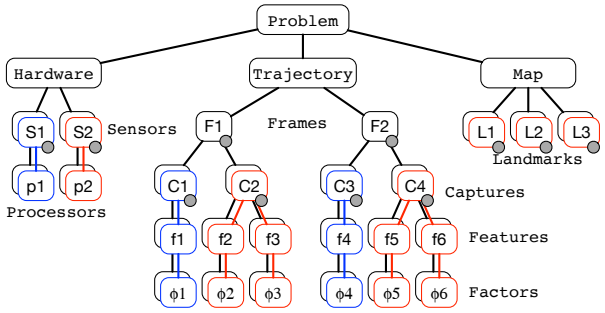



Fig. 1. The WOLF tree, showing the top node `Problem` with branches `Hardware`, `Trajectory` and `Map`. Children nodes shown are: `Sensors` (S) and their associated `Processors` (p), trajectory `Frames` (F), `Captures` (C) of raw data, detected `Features` (f), their corresponding `Factors` ($\phi$), and the `Landmarks` (L) added to the `Map`. Base classes, in black, are contained in WOLF `core`. Classes derived from them, in color, are packaged in separate libraries called WOLF plugins. The displayed tree corresponds to a visual-inertial setup, comprising an IMU plugin (blue) and a vision plugin (red). Each plugin comes with proper sensor, processor, capture, feature, factor, and landmark types. Classes with state-blocks are marked with a gray dot. See the accompanying video.

One of such decisions involves the nature of the estimation engine. Historically, the Bayesian filtering paradigm has been used for sensor fusion and SLAM, with the Extended Kalman Filter or more modern variants. In RT-SLAM [1] we fused vision, IMU and GNSS using EKF. Recent examples are MaRS [2] and PRONTO [3], this second one used for legged robots. With the advent of powerful computers and efficient nonlinear solvers, it is becoming standard to switch to nonlinear iterative maximum a-posteriori (MAP) optimization, particularly in the form of factor graph optimizers solved by least-squares [4]–[6], which achieve better performances in most fronts [7] including speed and accuracy.

A second aspect concerns loosely versus tightly coupled estimators. In loose coupling, a set of independent modules, each accepting the data from one (or a small subset of) sensor, produce state estimates of the system. These are then fused in a second stage to find a unique weighted estimate. Loose coupling as in *e.g.* [8] is practical because it is easy to develop

Manuscript received: October 11, 2021; Revised December 7, 2021; Accepted February 4, 2022.

This paper was recommended for publication by Editor Markus Vincze upon evaluation of the Associate Editor and Reviewers' comments. This work was partially supported by the EU H2020 projects LOGIMATIC (Galileo-2015-1-687534), GAUSS (Galileo-2017-1-776293), TERRINET (INFRAIA-2017-1-730994), and MEMMO (ICT-25-2017-1-780684); and by the Spanish State Research Agency through projects EB-SLAM (DPI2017-89564-P), EBCON (PID2020-119244GB-I00), and the María de Maeztu Seal of Excellence to IRI (MDM-2016-0656).

* These authors contributed equally.

JS, JV, JC, JD, ACM and JAC are/were with the Institut de Robòtica i Informàtica Industrial (IRI), CSIC-UPC, Llorens Artigas 4-6, Barcelona (Corresponding author e-mail: `jsola@iri.upc.edu`). JD was also with PAL Robotics, Pujades 77, Barcelona. MF and DA are/were with LAAS-CNRS, 7 Av. Colonel Roche, Toulouse.

Digital Object Identifier (DOI): see top of this page.

and easily scales up, for example taking advantage of the ROS architecture. However, it does not offer great performances, due to important cross-correlations being ignored, preventing *e.g.* sensor self-calibration. In contrast, tightly coupled systems can enable the observation of otherwise hidden states such as intrinsic and extrinsic sensor parameters, thus improving the overall accuracy. Because of the need for maintaining cross-correlations, sometimes these systems exhibit no modularity, although this can be circumvented as explained below.

Modularity is precisely a third major design criterion. Non-modular estimators such as ORB-SLAM [9] or Cartographer [10] can achieve very high performance, but tend to be specialized to a particular sensor setup. Modifying or scaling them up is difficult, since one needs to be aware of all the intricate inter-relations. Modular systems improve versatility, scalability and re-usability, and can more easily be made to withstand sensor failure, greatly improving robustness. However, modularity can come at the price of compromising performance, since communication between modules is often nonexistent, preventing potential synergies between them.

A very interesting compromise amongst all these dichotomies can be achieved with modern architectures, pioneered by Klein's PTAM [11], that divide the estimator in (a set of) front-ends and one back-end. This philosophy quickly evolved to the graph-SLAM paradigm [12]–[15], where the element of union between front-ends and back-end is a factor graph, which is a probabilistic representation of the full problem that is solved iteratively using off-the-shelf nonlinear solvers. This offers some key advantages. The full representativeness of the graph enables tightly coupled estimation. Modularity is achieved by assigning one front-end to every sensor, each contributing its states and/or factors to the graph. Establishing the right frontiers between front-ends, and the interface with the graph and back-end, allows us to set up many different systems by combining re-usable parts that have been developed separately. Packaging these parts in separate libraries or *plugins* allows independent and decentralized development, thus enabling scalable open-source projects to emerge. This is necessary to ultimately standardize fusion for robotics in practice so that it can be used by non-experts.

*a) Closely related works:* This idea above is the main motivation behind WOLF and is shared by a few other recent projects. They differ in the way the components and algorithms are organized, something that, beyond performance, impacts the way developers and users interact with each system. Perhaps the simplest example is the Fuse stack [12], with sensor and motion front-ends organized in plugins and devoted to contribute factors to the graph directly, and a number of ROS publishers to output the results of the optimization. Unfortunately, Fuse has not been published as a scientific communication and has little documentation, making it difficult to evaluate. Similar proposals are MOLA [13], which has been reportedly tested only with LIDAR and wheel odometry sensors, and RTAB-Map [14] which bases loop closure on processing RGBD images. A slightly better approach in our opinion is Plug-And-Play SLAM [15], a SLAM architecture in C++ that realistically aims at standardizing multi-modal SLAM. The architecture is organized around some SLAM-driven abstract design patterns, which form the core module. New sensor modalities can be added from other modules by editing a configuration file. P&P-SLAM has been demonstrated with wheel odometry, IMU, LIDAR and RGBD vision.

*b) Contributions:* Our proposal WOLF offers some key differences over the aforementioned systems. First, we use an intermediate layer between the front-ends and the graph, called the WOLF tree. This stores all the information in a comprehensive way, mimicking the elements encountered in the real problem. That is, our architecture is mainly constituted by objects and not by procedures like P&P-SLAM. The WOLF tree is completely accessible as all links can be traversed bidirectionally. This allows the front-ends to make use of precious processing information generated by other front-ends, giving them the possibility to benefit each other while keeping the architecture completely modular. The WOLF tree can be printed at any time and becomes a powerful book-keeping tool for tuning and debugging. Second, and in a similar spirit as P&P-SLAM, we provide the most common algorithms in abstract form (motion pre-integration, feature or landmark tracking, and loop closing), making the generation of specific algorithms easier. Third, this abstraction allows us to include sensor self-calibration natively, both for extrinsic and intrinsic parameters. Crucially, we offer generic calibration for sensors requiring pre-integration (odometers, IMU, force/torque). WOLF also handles time-varying sensor parameters, such as IMU biases, natively and generically. Finally, we showcase real applications in both 2D and 3D setups: IMU, wheel odometry, LIDAR, vision (in several ways), force and torque, limb encoders, and GNSS (in several ways), overall comprising a larger variety of sensors than any of the previous proposals.

WOLF is made available freely for research purposes and comes with extensive documentation.[1] We encourage the reader to also watch the accompanying video to complement or clarify the information provided in this paper.

## II. ARCHITECTURE

### A. Overview

One of the key originalities of WOLF is its central data structure, called the WOLF tree (Fig. 1, Sec. II-B). The WOLF tree is a tree of abstract base classes reproducing the elements of the robotic problem (`Sensors`, `Processors`, `Frames`, `Features`, `Factors`, *etc.*). To account for different sensing or processing modalities, these classes are derived to implement specific functionalities. These elements can be added to the system in a modular way, allowing great flexibility to setup different applications. On top of this modularity, scalability is achieved by splitting the project into independent libraries called *plugins*, each containing a set of derived classes corresponding to one of such modalities (*e.g.* laser, vision, imu, and others).

WOLF `Processors`, which are part of the WOLF tree, constitute the estimator's front-ends (Sec. II-C). As the back-end, WOLF interfaces with one graph-based solver (Sec. II-E).

[1]Documentation, access to all repositories and installation instructions can all be found at http://www.iri.upc.edu/wolf.

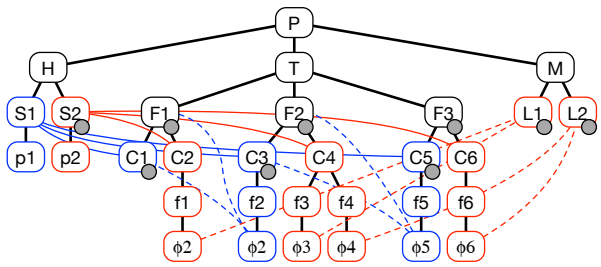



Fig. 2. Pointers across branches allow trees in the likes of Fig. 1 to be converted to a factor graph. `Captures` have always a pointer (solid) to the `Sensor` that created them. `Factors` have pointers (dashed) to nodes of the tree that appear in their measurement model. The tree shows nodes and measurements related to vision (red) and IMU (blue). This setup allows for self-calibration of the camera's extrinsic parameters (state blocks in S2) and the tracking of the IMU biases (in IMU captures C1, C3, C5). See Fig. 3 for the resulting factor graph.

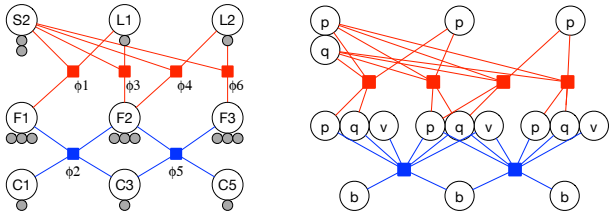



Fig. 3. Two flavors for the factor graph derived from the tree in Fig. 2. *Left*: human-readable graph: nodes are WOLF nodes. State blocks in gray, and factors in color (red: vision, blue: IMU). *Right*: graph as seen by the solver: nodes are state blocks ($p$: position, $q$: orientation, $v$: velocity, $b$: bias).

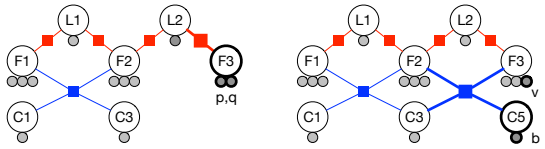



Fig. 4. Dynamic evolution of the `Frame`'s structure of state blocks in a visual-inertial setup. *Left*: The visual processor adds a new `Frame` F3 with state blocks $p, q$ necessary for the new landmark observation factor (thick red). *Right*: Then, the IMU `Processor` joins F3 (see Sec. III-E2) and adds to it a new state block $v$ necessary for the pre-integrated IMU factor (thick blue). It also appends its IMU `Capture` C5 with the bias block $b$.

In brief, incoming raw data is processed by the processors to populate the WOLF tree, which is translated to a factor graph that is repeatedly solved.

To configure a particular application, we designed a fully automated setup mechanism based exclusively on YAML files describing the full robotic setup (Sec. III-A). For convenience, WOLF can be interfaced with ROS (Sec. IV).

### B. *The* WOLF *tree*

WOLF organizes the typical entities appearing in robot state estimation in a tree structure (Fig. 1), with nodes for:

- the robot's `Hardware` consisting of all of its `Sensors`, and their `Processors` of raw data;
- the robot's `Trajectory` over time: `Frames`, `Captures` of raw data, `Features` or metric measurements extracted from that data, and `Factors` relating these measurements to the state blocks of the system;
- the `Map` of `Landmarks` in the environment.

Most of these nodes are basically data holders: `Sensor` (extrinsic and intrinsic parameters), `Frame` (robot state), `Capture` (raw sensory data), `Feature` (metric measurement) and `Landmark` (state, appearance descriptor). Nearly all the processing work resides in three nodes: `Processors` (at the front end), `Factors` (at the back end), and `Problem` (tree management).

In each branch of the tree, WOLF nodes may have any number of children. All links are bidirectional making the tree fully accessible from any point. Crucially, some key connections break the tree structure and provide the basis for the creation of a meaningful network of relations describing the full robotic problem (Fig. 2). These connections are: (a) `Captures` have access to the `Sensor` that created them; and (b) `Factors` have access to each node containing state blocks necessary to compute the factor residual. This richer network is easily interpreted as a factor graph (Fig. 3), which is solved by the solver or back-end.

At startup the WOLF tree is initialized with the full definition of the `Hardware` branch and, optionally, a first `Frame` with the robot's initial condition and/or a pre-defined `Map` of the environment. This information is extracted at run-time from YAML configuration files, see Sec. III-A.

### C. *Front-ends: Processors*

The rest of the WOLF tree is incrementally built by the `Processors` as data is being gathered. `Processors` are responsible for many things: extracting `Features` from raw data in `Captures`; associating `Features` with other nodes in the tree; creating `Frames` (and conditionally to this, creating `Factors` and new `Landmarks`); joining `Frames` created by other `Processors`; integrating motion; and closing loops. See Sec. III-C for further details.

### D. WOLF *tree manager*

Since `Processors` only populate the tree with new nodes, a manager that is able to remove nodes is necessary to keep the problem at a controlled size. The tree manager is a member of the `Problem` node. We provide with `core` two managers implementing sliding windows of `Frames`. Other managers performing *e.g.* graph sparsification [16] should be implemented by deriving from a base class.

### E. *Back-end: graph and solver*

The back-end can be explained by means of a factor graph, with its state blocks and factors, and a graph solver.

*1) State blocks:* Some nodes in the tree (`Sensor`, `Frame`, `Capture` and `Landmark`) contain instances of the state block class. A state block is the minimum partition of the full problem state that carries a particular meaning (position, orientation, sensor parameters, etc.). Crucially, state blocks in `Frames` can be added dynamically (Fig. 4). This allows for `Processors` and `Factors` requiring a different set of state blocks for a given `Frame` to work together seamlessly.

A state block $i$ contains a state vector $\mathbf{x}_i$ and, optionally, a local (or tangent) parametrization $\Delta\mathbf{x}_i$ used to perform estimation on the manifold of $\mathbf{x}_i$. A state block can be fixed by the user, meaning that its values are treated as fixed parameters, and hence not optimized by the solver.

*2) Factors:* `Factors` are responsible for computing a residual $\mathbf{r}(\mathbf{x}_1, \dots, \mathbf{x}_N)$ each time the solver requires it. Each `Factor` contains pointers to all the state blocks $\mathbf{x}_1, \dots, \mathbf{x}_N$ appearing in the observation model. `Factors` can optionally include a loss function allowing for robust estimation in front of outliers. Since the solver requires the Jacobians of $\mathbf{r}$ to perform the optimization, we provide base `Factors` prepared to produce analytical or automatic Jacobians.

*3) Solver:* We designed WOLF to be used with nonlinear graph solvers. These are available as independent libraries such as Google Ceres [17], GTSAM [4], g2o [18], SLAM++ [6], and others. Interfacing WOLF with the solver is relegated to solver wrappers, which are encapsulated in classes deriving from a base class in WOLF `core`. The wrapper essentially makes the WOLF's state-blocks and `Factors` accessible to the solver by bridging the API on both sides. We provide in `core` a wrapper to Google Ceres.

### F. Modular development through independent plugins

As explained already, WOLF consists of both abstract classes and specialized classes. To promote modularity and scalability, WOLF encapsulates all abstract material in WOLF `core` while specializations are packaged in other libraries called WOLF plugins. Each plugin corresponds typically to a certain sensor modality, or to a certain way to process a sensor modality's data. Plugins are meant to be reused across different applications. New plugins can be added in case a certain sensing modality or processing method does not exist amongst the available plugins. These new plugins can be incorporated to the WOLF general repository, or kept elsewhere, preferably available for the community. Currently WOLF includes plugins for `vision`, `apriltag`, `gnss`, `imu`, `laser` and `bodydynamics`, this last one devoted to the estimation of the whole-body dynamics of articulated robots such as humanoids, quadrupeds or aerial manipulators. Plugins can depend on other plugins. For example, the `apriltag` plugin depends on `vision` and contributes a new way of processing visual information.

### G. Multi-threading

There is one dedicated thread for the solver and one for each WOLF ROS publisher (see Sec. IV). At the moment, all processors are executed in one single other thread —see Sec. VI for planned future improvements.

## III. FEATURES

### A. Full auto-configuration from YAML files

WOLF includes a mechanism for providing all configuration parameters to the WOLF problem through YAML files.[2] These files contain: (a) the sensors and their parameters; (b) the processors and their parameters; (c) the solver and its parameters; (d) the tree manager and its parameters; (e) the initial robot state; and optionally (f) the initial map.

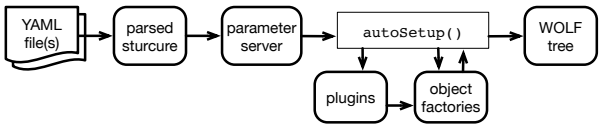



Fig. 5. WOLF auto-configuration pipeline. YAML files are parsed and converted to a parameter server. The method `Problem::autoSetup()` reads from this server and loads the plugins, whose object creators are all registered in the factories in `core`. These are then used in `autoSetup()` to create all objects to initialize the WOLF tree.

Fig. 5 shows the auto-configuration process: starting from a set of YAML files, a parser and a parameter server are used in conjunction with object factories to create the WOLF tree and leave the robot ready to start its mission. WOLF automatically loads the plugins at run-time and registers all derived object creators in dedicated factories in `core`. This means that, having the necessary plugins available, different estimation problems can be set up without the need for writing and compiling code, but just specifying the problem setup at YAML level. See the accompanying video for a demonstration.

### B. Sensor parameters management

Sensor parameters (intrinsic and extrinsic) can be *static* (*i.e.*, constant —not to be confused with *fixed*, see II-E1) or *dynamic* (*i.e.*, time-varying). Sensor parameters declared static are stored in state blocks in the sensor object itself. Dynamic sensor parameters are stored in state blocks in the captures created by the sensor so that they can be associated with the particular timestamp of the capture (and therefore also of its parent frame).

Sensor parameters that are to be estimated require the state blocks to be unfixed (see II-E1). If these are static, we speak of *parameter calibration*. If they are dynamic, we speak of *parameter tracking*. For example, see Figs. 2 and 3, IMU biases are varying with time and need to be continuously tracked. Instead, camera extrinsic parameters are constant and may just require calibration.

### C. Abstract (or generic) processor algorithms

The variability of the processing algorithms may be very large. However, there are three well-identified groups of algorithms in SLAM: motion integration, feature tracking, and loop closing. Their basic functionalities, coded in respective base classes, are explained below.

*1) Motion pre-integrators:* This processor performs generic motion pre-integration with automatic sensor calibration or tracking. It is based on our generalized pre-integration theory [19, Section 4.3] implementing the following pipeline of operations (notation: $\mathbf{u}$: raw motion data; $\mathbf{v}$ calibrated motion data; $\bar{\mathbf{c}}$: calibration parameters' initial guess; $\delta$: current motion delta; $\overline{\Delta}$: pre-integrated motion delta; $\mathbf{J}_x^y \triangleq \partial y/\partial x$: Jacobians computed according to Lie theory [20]; $\mathbf{Q}_x$: covariance of $x$):

- Pre-calibrate motion data $\quad \mathbf{v} = f(\mathbf{u}, \bar{\mathbf{c}}),\ \mathbf{J}^{\mathbf{v}}_{\mathbf{u}},\ \mathbf{J}^{\mathbf{v}}_{\mathbf{c}}$.

[2] See https://gitlab.iri.upc.edu/mobile_robotics/wolf_projects/wolf_ros/demos/wolf_demo_laser2d/-/blob/devel/yaml/demo_laser2d.yaml for an example of YAML file describing a real robotics problem.

- Compute current delta $\delta = g(\mathbf{v}),\ \mathbf{J}^{\delta}_{\mathbf{v}}$.
- Pre-integrate delta $\overline{\Delta} \leftarrow \overline{\Delta} \circ \delta,\ \mathbf{J}^{\overline{\Delta}}_{\overline{\Delta}},\ \mathbf{J}^{\overline{\Delta}}_{\delta}$.
- Integrate covariance $\mathbf{Q}_{\overline{\Delta}} \leftarrow \mathbf{J}^{\overline{\Delta}}_{\overline{\Delta}}\,\mathbf{Q}_{\overline{\Delta}}\,\mathbf{J}^{\overline{\Delta}\top}_{\overline{\Delta}} + \mathbf{J}^{\overline{\Delta}}_{\delta}\mathbf{J}^{\delta}_{\mathbf{v}}\mathbf{J}^{\mathbf{v}}_{\mathbf{u}}\,\mathbf{Q}_{\mathbf{u}}\,(\mathbf{J}^{\overline{\Delta}}_{\delta}\mathbf{J}^{\delta}_{\mathbf{v}}\mathbf{J}^{\mathbf{v}}_{\mathbf{u}})^{\top}$.
- Integrate Jacobian with respect to calibration parameters $\mathbf{J}^{\overline{\Delta}}_{\mathbf{c}} \leftarrow \mathbf{J}^{\overline{\Delta}}_{\overline{\Delta}}\mathbf{J}^{\overline{\Delta}}_{\mathbf{c}} + \mathbf{J}^{\overline{\Delta}}_{\delta}\mathbf{J}^{\delta}_{\mathbf{v}}\mathbf{J}^{\mathbf{v}}_{\mathbf{c}}$.

WOLF can produce high throughput estimates at sensor rates up to the kHz range, which can be used by other processors and for feedback control of highly dynamic robots such as humanoids, quadrupeds or aerial manipulators. These states are computed with $\mathbf{x}_t = \mathbf{x}_i \boxplus \overline{\Delta}_{it}$, where $\mathbf{x}_i$ is the last frame at time $i < t$ and $\overline{\Delta}_{it}$ is the delta pre-integrated from times $i$ to $t$. Classes deriving from `ProcessorMotion` only have to implement the function $g(f(\mathbf{u}, \mathbf{c}))$, the delta composition $\circ$ and the operator $\boxplus$.

*2) Trackers:* They extract features from raw data such as images or laser scans and track them over time. We offer two abstract variants. One associates features in the current capture with features in the capture at the last frame. The second one associates features with landmarks in the map and is able to create new landmarks.

*3) Loop closers:* They search for frames in the past having captures that are, in some robust sense, similar to the present one. Then, based on some geometrical analysis of the data, they establish the appropriate factors between the two sensor frames.

### D. Sensor self-calibration

We distinguish three self-calibration strategies, all logically subject to the observability conditions given by each particular sensor setup. First, the calibration of sensor extrinsic parameters just requires the factors to account for the extra robot-to-sensor transformations, which we provide. Second, intrinsic self-calibration requires the factors' observation model to account for such parameters: they must be specifically coded in each case and cannot be generalized. Finally, the most difficult intrinsic self-calibration for sensors requiring pre-integration is also part of our generalized pre-integration theory [19, Sec. 4.3], see Sec. III-C1. At the factor side, the pre-integrated delta $\overline{\Delta}$ is corrected for values of the calibration parameters $\mathbf{c}$ different from the initial guess $\bar{\mathbf{c}}$ following the linearized formula $\Delta(\mathbf{c}) = \overline{\Delta} \oplus \mathbf{J}^{\overline{\Delta}}_{\mathbf{c}}(\mathbf{c} - \bar{\mathbf{c}})$. The resulting residual $\mathbf{r}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{c}) = \mathbf{Q}^{-\top/2}_{\Delta}(\Delta(\mathbf{c}) \ominus (\mathbf{x}_j \boxminus \mathbf{x}_i))$ between consecutive frames $i, j$ enables the observation of the sensor parameters $\mathbf{c}$. See [21], [22] for seminal works on motion pre-integration for the IMU —notice that $\mathbf{c}$ are biases in the IMU case but can be any sensor parameter, static or dynamic, in the general case. Examples of application of this method are [19, Chap. 3], [23], [24], and also in Sec. V.

### E. Automatic frame synchronization

In multi-sensor setups, efficient frame creation policies can be tricky. We designed a decentralized strategy where each processor may independently decide to create a frame. Once created, the frame is broadcasted for other processors to join (see Fig. 4). This technique requires all sensory data to be timestamped by a common clock, or at least by clocks that are properly synchronized.

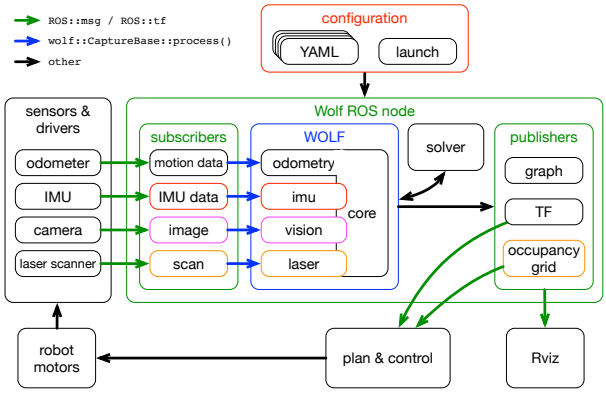



Fig. 6. Using ROS to integrate a robotics estimation problem with WOLF. Each sensor modality uses a WOLF plugin (colored boxes; odometry is part of `core`) and its corresponding ROS subscriber (in the same color). Publishers create ROS messages with useful output data for control and visualization.

*1) Frame creation and broadcast:* Each processor defines its own frame creation policy. Besides trivial policies like time elapsed or distance traveled, frames can be more smartly distributed if the decision is made based on the processed data. For example, a feature tracker may create a frame when the number of feature tracks since the last frame has dropped below a threshold. Frame creation is followed by appending the capture with its features, and creating the appropriate factors (and landmarks, if any). Once created, frames are broadcasted for other processors to join.

*2) Frame join:* Broadcasted frames are received by all other processors. On reception, processors will compare their capture's timestamps with the timestamp of the received frame. A check of the time difference against the processor time tolerances is performed. If passed, the processor adds, if necessary, state blocks to the frame (see II-E1 and Fig. 4), appends the selected capture as a new child of the received frame, and creates the appropriate features and factors (and landmarks, if any).

## IV. ROS INTEGRATION

The WOLF integration in ROS is performed using a node and a set of subscribers and publishers. We provide a unique generic ROS node and several ROS packages with subscribers and publishers following the same structure as the plugins. A basic representation of the information flow in WOLF using ROS is sketched in Fig. 6. We also provide a built-in profiler to help users to optimize the performances in terms of accuracy and real-time constraints.

### A. The WOLF ROS node

In the core ROS package, we provide a node that should serve many different robotics projects without modification. The node has a list of publishers, a list of subscribers, the solver wrapper, and the WOLF tree. Given a set of available plugins, all the user has to do is to write the YAML file(s) (see the accompanying video). These files specify the WOLF tree (see III-A), and the subscribers and publishers. A straightforward ROS launch file specifying the YAML path completes

the application. At startup the node will: (a) parse the YAML configuration files; (b) load the required WOLF plugins and initialize the WOLF tree; (c) configure the solver; and (d) load and initialize all subscribers and publishers.

### B. WOLF *subscribers and publishers*

A WOLF subscriber is an object that has a ROS subscriber and access to the corresponding sensor in the WOLF tree. Derived subscribers have to implement the appropriate callback method, which creates the derived capture object and calls its `process()` method. WOLF takes care of the rest, launching all the sensor's child processors to process the capture's data. Sensor parameters can be recovered from the ROS message.

A WOLF publisher has a ROS publisher, access to the WOLF problem, and a user-defined publishing rate. Derived publishers have to implement the method that fills a ROS message with the desired information from the WOLF tree, and publish it. We provide in `core` publishers for the ROS transforms (tf) and the visualization of the graph.

## V. REAL CASES

We present five distinct and illustrative applications of WOLF. They share significant parts of the code, re-use plugins, and are only determined by the set of YAML files parsed at launch time. All experiments run in real-time on standard modern PC hardware. Please refer to Fig. 7 for the human-readable factor graphs associated with each one of them.

### A. WOLF *demo: 2D LIDAR + odometry (plugins: core and laser)*

This application involves a differential-drive base with a 2D laser scanner (Fig. 7 (a), Fig. 8). It exhibits: a 2D odometry pre-integrator; a laser processor performing laser odometry and mapping based on scan matching, with self-calibration of LIDAR extrinsics; and a simple loop closer also based on scan matching. Frames are produced by the laser odometry processor according to a mixed policy based on distance traveled, angle turned, and the quality of the scan matching. The system can navigate small indoor areas, publishing the graph and the map in two possible formats: a point-cloud that may be used for visualization and debugging, and an occupancy grid useful for planning and control. It constitutes one of the WOLF-ROS demos shipped with the library.

### B. *LOGIMATIC: GNSS + differential drive odometry + LIDAR (plugins: core, gnss and laser)*

This application performs 2D SLAM in a large port area (Fig. 7 (b), Fig. 10) [25]. A big 8-wheeled straddle-carrier is equipped with one GNSS, left and right wheel groups rotary encoders, and 4 long-range LIDARs. A motion processor integrates wheel odometry. One GNSS processor incorporates GNSS fixes as absolute factors at frames. A second GNSS processor computes globally referenced displacements from satellite time-differenced carrier phase (TDCP) accurate to the cm level [26], [27]. Four cooperative LIDAR processors extract and track poly-line features, used for SLAM, and can detect containers and place them in the SLAM map as objects. The system also performs precise geo-localization of the vehicle and mapped area, and self-calibrates the differential drive motion model [23] and the antenna location.

### C. *GAUSS: GNSS + IMU (plugins: core, imu and gnss)*

This 3D implementation involves a UAV with a GNSS receiver, and a low-cost IMU (Fig. 7 (c), Fig. 9) [28]. We use an IMU pre-integrator with bias tracking. GNSS is incorporated in a tightly coupled manner via individual pseudoranges as frame-to-satellite factors. The same GNSS processor also adds individual TDCP factors between pairs of frames at up to 1 min interval. Given a sufficient number of satellites, direct pseudoranges allow for absolute positioning in the 1m range, while TDCP observes absolute displacements in the cm range, much more accurate than individual fixes. The antenna location is self-calibrated.

### D. *MEMMO-vision: 6DoF vision + IMU for the humanoid (plugins: core, imu, vision and apriltag)*

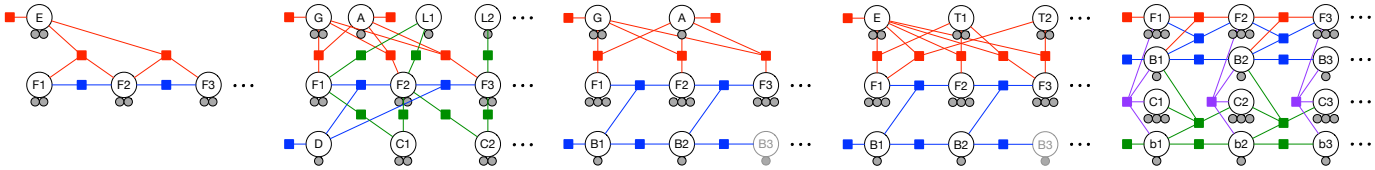
This 3D experiment (Fig. 7 (d), Fig. 11) involves a humanoid robot equipped with a camera and an IMU in its head. The IMU processor is the same as in V-C. The visual processor detects AprilTags [29] in the scene, measuring 6 DoF camera-tag transforms. The tag locations can be either known a-priori (map-based localization), completely unknown (SLAM), or a mixture of them (SLAM with prior visual clues). All these variants are selected at YAML level.

WOLF produces drift-free trajectory estimates accurate to $\leq 4$ cm. It also publishes full robot states referred to a precise gravity direction at the IMU frequency of 200Hz [24].

In a similar approach, we conceived an object-based visual-inertial SLAM [30] (plugins: core, imu, vision and object-slam), where we use the deep learning object detection library CosyPose [31] instead of AprilTags. CosyPose identifies previously learned objects and returns the 6DoF camera-object relative pose. The factor graph is thus the same as in the AprilTag system (Fig. 7 (d)).

### E. *MEMMO-forces: IMU, contact forces and joint angles for the quadruped (plugins: core, imu and bodydynamics)*

The last application of WOLF showcased in this paper performs whole-body estimation [32] of the quadruped Solo12 from the Open Dynamic Robot Initiative (Fig. 7 (e), Fig. 12). Contact forces altering the centroidal robot dynamics (position and velocity of the center of mass (CoM), angular momentum) are pre-integrated using a specialization [32] of our generalized pre-integration method (Sec. III-C1). The IMU data is also pre-integrated to measure the motion of the base. Joint angle measurements are then participating in the estimation in two ways: by providing leg odometry displacements, contributing to the observability of IMU biases; and by relating the base and centroidal states, thus connecting the whole graph and producing a tightly-coupled whole-body estimator. The states of base position, orientation and velocity, IMU biases, CoM position and velocity, angular momentum and CoM sensor




(a) Demo: 2D LIDAR (red), wheel odometry (blue). Fi: frames position $p$ and orientation $q$; E: LIDAR extrinsics $p, q$.

(b) LOGIMATIC : GNSS (red), 2D LIDAR (green), differential drive (blue). G: georeferencing $p, q$; Li: landmarks $p$; D: diff. drive; Ci: Containers $p, q$.

(c) GAUSS : GNSS (red), IMU (blue). G: georeferencing $p, q$; A: antenna $p$; Fi: frames $p, q$ and velocity $v$; Bi: IMU bias.

(d) MEMMO-vision : Visual tags (red), IMU (blue). Ti: visual tags $p, q$; E: camera extrinsics $p, q$.

(e) MEMMO-forces : IMU (blue), contact forces (green), joint angles (red and purple). Ci: centroidal CoM $p, v$, angular momentum $L$; bi: CoM bias.

Fig. 7. Human-readable graphs (see Fig. 3) generated by WOLF for the five real cases exposed. Factors in colors (see sub-captions). WOLF nodes in labeled circles. State blocks in gray dots (see sub-captions). See Sec. V and subsections therein for further details.

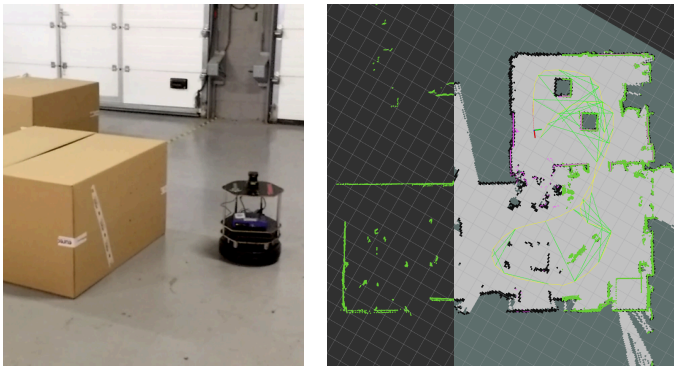


Fig. 8. A turtlebot equipped with a low cost LIDAR and wheel odometry performs indoor localization and mapping. This toy application constitutes a tutorial for getting started with WOLF and ROS, and is shipped with the library. The map can be published as pointcloud (left fraction of the map), as occupancy grid (center), or both (right).

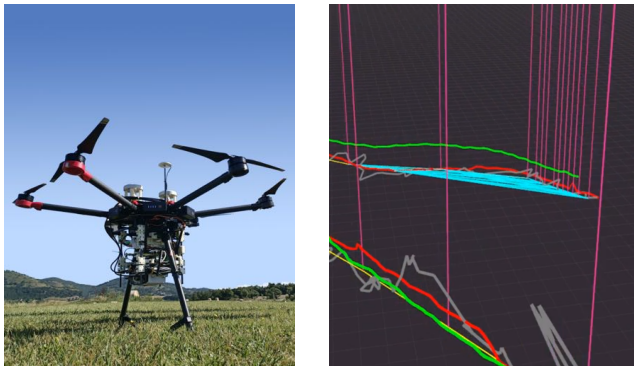


Fig. 9. In the context of the EU H2020 project GAUSS, we equipped a UAV with two GNSS receivers and an IMU. Plain GNSS fixes show numerous multi-path artifacts yielding an erratic trajectory (gray). We combined IMU, GNSS pseudoranges (magenta) and TDCP (cyan) together with robust outlier rejection to achieve smooth and accurate motion estimation (red).

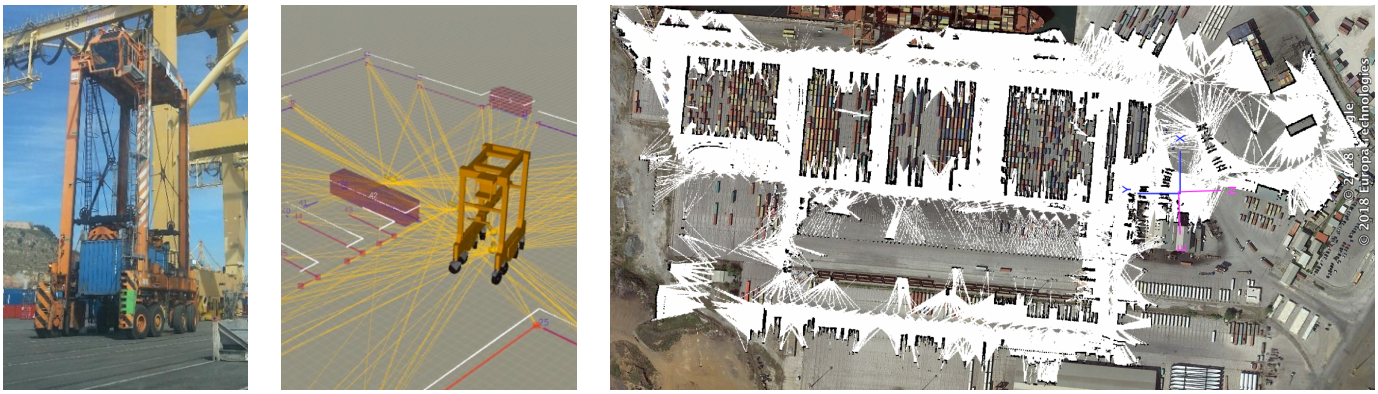



Fig. 10. In the context of the EU H2020 LOGIMATIC project, an autonomous retrofitted 8-wheel straddle carrier equipped with GNSS, LIDAR and odometry (left) mapped an area and detected the containers (center) in the port of Thessaloniki in Greece. The final map produced with WOLF has an area of about 1000m×500m and is shown overlaid on top of an aerial photograph (right). The map reference frame (blue, XY) was precisely geo-referenced by WOLF with respect to the local ENU coordinates (magenta, EN) established by the first GNSS fix.

bias are published at 1 kHz with a lag of less than 1 ms. This provides accurate observation of the centroidal dynamics, which is crucial for balance and gait control.

## VI. CONCLUSIONS AND FUTURE WORK

Over the last six years, WOLF has provided a consistent framework for conceiving and implementing sensor fusion for a vast variety of research projects. We have used it to incorporate abstract algorithms and this has pushed us to truly understand the common connections between many different robotics estimation problems. WOLF has matured during this time and we believe it offers a competitive design, with many interesting features that are desirable in such kind of software packages. We now offer it open to the community with the hope that it can be useful to other teams.

Work needs to be done to improve the real-time performance. In particular, WOLF still lacks the possibility to launch each processor in an individual thread. This is a weak point that is common to many other estimation frameworks in the state of the art, and it seems not straightforward to solve. We

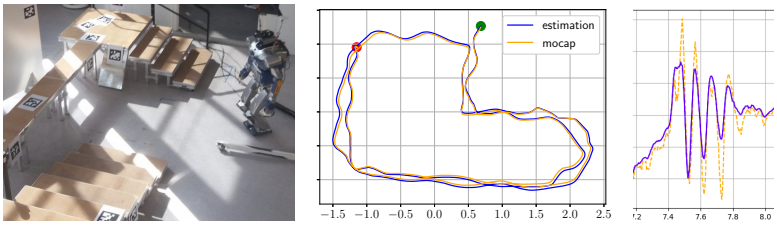



Fig. 11. In the context of the EU H2020 MEMMO project, the humanoid HRP-2 with a visual-inertial setup on its head performs a loop on a circuit with visual tags (left) at LAAS-CNRS. We show a drift-free trajectory reconstruction (center; "mocap" = IR motion capture) and the recovery, at the IMU sampling rate of 200Hz and in the same global reference, of head vibrations (right) resulting from foot impacts while descending the stairs.

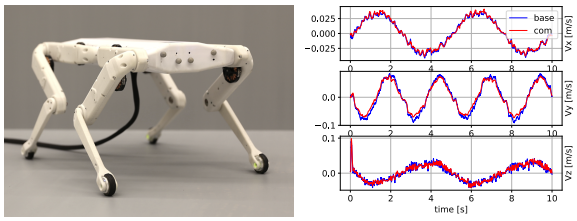



Fig. 12. In the context of the EU H2020 MEMMO project, the quadruped Solo12 (left) with measurements of an IMU, limb encoders and motor currents is used for whole-body base and centroidal estimation. We show precise recovery at 1kHz of the base position, the center of mass and its velocity.

believe that effort must be made in this direction, either in WOLF and/or elsewhere to be able to bring modular, tightly coupled and reconfigurable multi-sensor fusion to reality.

## ACKNOWLEDGMENTS

The authors would like to thank Dylan Bourgeois, César Debeunne, Idril Geer, Peter Guetschel, Josep Martí-Saumell, Sergi Pujol, Àngel Santamaria-Navarro, Jaime Tarraso, Pier Tirindelli and Oriol Vendrell for their valuable contributions to this project.

## REFERENCES

[1] C. Roussillon, A. Gonzalez, J. Solà, J. M. Codol, N. Mansard, S. Lacroix, and M. Devy, "RT-SLAM: A generic and real-time visual SLAM implementation," in *Int. Conf. Computer Vision Systems*, Sophia Antipolis, Sep. 2011, pp. 31–40.

[2] C. Brommer, R. Jung, J. Steinbrener, and S. Weiss, "MaRS: A modular and robust sensor-fusion framework," *IEEE Robotics and Automation Lett.*, vol. 6, no. 2, pp. 359–366, 2021.

[3] M. Camurri, M. Ramezani, S. Nobili, and M. Fallon, "Pronto: A multi-sensor state estimator for legged robots in real-world scenarios," *Frontiers in Robotics and AI*, vol. 7, p. 68, 2020.

[4] F. Dellaert and M. Kaess, "Factor graphs for robot perception," *Foundations and Trends in Robotics*, vol. 6, no. 1-2, pp. 1–139, 2017.

[5] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based SLAM," *IEEE Intelligent Transportation Systems Mag.*, vol. 2, no. 4, pp. 31–43, 2010.

[6] V. Ila, L. Polok, M. Solony, and P. Svoboda, "SLAM++ - A highly efficient and temporally scalable incremental slam framework," *Int. J. of Robotics Research*, vol. 36, no. 2, pp. 210–230, 2017.

[7] H. Strasdat, J. M. M. Montiel, and A. J. Davison, "Real-time monocular SLAM: Why filter?" in *IEEE Int. Conf. Robotics and Automation*, Anchorage, May 2010.

[8] R. Scona, S. Nobili, Y. R. Petillot, and M. Fallon, "Direct visual slam fusing proprioception for a humanoid robot," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2017, pp. 1419–1426.

[9] C. Campos, R. Elvira, J. J. G. Rodriguez, J. M. Montiel, and J. D. Tardos, "ORB-SLAM3: An accurate open-source library for visual, visual–inertial, and multimap SLAM," *IEEE Trans. on Robotics*, vol. 37, no. 6, p. 1874–1890, 2021.

[10] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," in *IEEE Int. Conf. Robotics and Automation*, Stockholm, May 2016, pp. 1271–1278.

[11] G. Klein and D. Murray, "Improving the agility of keyframe-based SLAM," in *Eur. Conf. Computer Vision*, 2008, pp. 802–815.

[12] S. Williams, "Fuse stack," 2018. [Online]. Available: http://github.com/locusrobotics/fuse

[13] J. L. Blanco-Claraco, "A modular optimization framework for localization and mapping," in *Robotics: Science and Systems*, Jun. 2019.

[14] M. Labbé and F. Michaud, "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *J. of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[15] M. Colosi, I. Aloise, T. Guadagnino, D. Schlegel, B. D. Corte, K. O. Arras, and G. Grisetti, "Plug-and-play SLAM: A unified SLAM architecture for modularity and ease of use," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Las Vegas, Oct. 2020, pp. 5051–5057.

[16] J. Vallvé, J. Solà, and J. Andrade-Cetto, "Graph SLAM sparsification with populated topologies using factor descent optimization," *IEEE Robotics and Automation Lett.*, vol. 3, no. 2, pp. 1322–1329, 2018.

[17] S. Agarwal, K. Mierle *et al.*, "Ceres solver," 2019. [Online]. Available: http://ceres-solver.org

[18] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g2o: A general framework for graph optimization," in *IEEE Int. Conf. Robotics and Automation*, Shanghai, May 2011, pp. 3607–3613.

[19] D. Atchuthan, "Towards new sensing capabilities for legged locomotion using real-time state estimation with low-cost IMUs," Thesis, Université Paul Sabatier - Toulouse III, Oct. 2018.

[20] J. Solà, J. Deray, and D. Atchuthan, "A micro Lie theory for state estimation in robotics," *CoRR*, vol. abs/1812.01537, 2018. [Online]. Available: http://arxiv.org/abs/1812.01537

[21] T. Lupton and S. Sukkarieh, "Efficient integration of inertial observations into visual SLAM without initialization," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, St Louis, Oct. 2009, pp. 1547–1552.

[22] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-manifold preintegration for real-time visual-inertial odometry," *IEEE Trans. on Robotics*, vol. 33, no. 1, pp. 1–21, 2017.

[23] J. Deray, J. Andrade-Cetto, and J. Solà, "Joint on-manifold self-calibration of odometry model and sensor extrinsics using pre-integration," in *Eur. Conf. Mobile Robots*, Prague, Sep. 2019.

[24] M. Fourmy, D. Atchuthan, N. Mansard, J. Solà, and T. Flayols, "Absolute humanoid localization and mapping based on IMU Lie group and fiducial markers," in *IEEE-RAS Int. Conf. Humanoid Robots*, Toronto, Oct. 2019, pp. 237–243.

[25] C. Rizzo, J. Andrade-Cetto, J. Solà, J. Vallvé, D. S. López, J. P. G. Villodres, D. Tsitsamis, and H. Rodríguez, "LOGIMATIC Autonomous Navigation," 2019. [Online]. Available: http://hdl.handle.net/10261/195516

[26] F. Van Graas and A. Soloviev, "Precise velocity estimation using a stand-alone GPS receiver," *Navigation*, vol. 51, no. 4, pp. 283–292, 2004.

[27] P. Freda, A. Angrisano, S. Gaglione, and S. Troisi, "Time-differenced carrier phases technique for precise GNSS velocity estimation," *GPS Solutions*, vol. 19, no. 2, pp. 335–341, 2015.

[28] A. Jímenez, J. Andrade-Cetto, I. Tesfai, I. Dontas, C. Capitán, E. Oliveres, H. Jia, and A. Kostaridis, "Galileo and EGNOS as an asset for UTM safety and security," in *25th Ka and Broadband Communications Conf.*, Sorrento, Sep. 2019.

[29] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *IEEE Int. Conf. Robotics and Automation*, Shanghai, May 2011, pp. 3400–3407.

[30] C. Debeunne, M. Fourmy, Y. Labbé, P.-A. Léziart, G. Saurel, J. Solà, and N. Mansard, "CosySLAM: tracking contact features using visual-inertial object-level SLAM for locomotion," submitted to ICRA 2022. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03351438

[31] Y. Labbé, J. Carpentier, M. Aubry, and J. Sivic, "CosyPose: Consistent multi-view multi-object 6D pose estimation," in *Eur. Conf. Computer Vision*, Glasgow, Aug. 2020, pp. 574–591.

[32] M. Fourmy, T. Flayols, N. Mansard, and J. Solà, "Contact forces pre-integration for the whole body estimation of legged robots," in *IEEE Int. Conf. Robotics and Automation*, Xian, May 2021.