User guide for `libmrpt-srba`: A generic C++ framework for Relative Bundle Adjustment (RBA)

Jose-Luis Blanco-Claraco
joseluisblancoc@gmail.com
`http://www.mrpt.org/`

MRPT version: 1.2.2
Document build: September 12, 2014

Revision history:

- May 2013: Second version. Added Fig. 8 and the section on spanning trees.
- March 2013: First version. Released along MRPT 1.0.0.

In case you want to cite SRBA in your academic publications, here is a BibTeX entry:

```
@INPROCEEDINGS{blanco2013srba,
  author = {Blanco, J.L. and Gonzalez-Jimenez, J. and Fernandez-Madrigal, J.A.},
  month = {{may}},
  title = {{Sparser Relative Bundle Adjustment (SRBA): constant-time
    maintenance  and local optimization of arbitrarily large maps}},
  booktitle = {IEEE International Conference on Robotics and Automation (ICRA)},
  year = {2013}
}
```

and here the one for this guide:

```
@MISC{libmrpt-srba-guide,
  author = {Jose-Luis Blanco-Claraco},
  title = {{User guide for \texttt{libmrpt-srba}: A generic
      C++ framework for Relative Bundle Adjustment (RBA)}},
  howpublished = {http://www.mrpt.org/srba},
  year = {2013}
}
```

Contents

1 Introduction

Bundle adjustment (BA) is the name given to one solution to visual Simultaneous Localization and Mapping (SLAM) (or Structure From Motion, SFM) based on maximum-likelihood estimation (MLE) over the space of map features and camera poses. However, it is by no way limited to visual maps, since the same optimization techniques employed in BA are also applicable to many other kind of feature maps, not necessarily involving visual information, or even to maps of pose constraints (*Graph-SLAM*).

For readers without a solid background in mobile robotics or computer vision, it is *strongly* recommended to start reading seminar works on SLAM [2,5,11], BA [12] and graph-SLAM [6] before putting hands on programming with `libmrpt-srba`.

The idea of *Relative Bundle Adjustment (RBA)* and *Relative SLAM* was introduced in a series of works by Gabe Sibley and colleagues in [8–10].

Sparser RBA (SRBA) is the name of the generic and extensible framework for RBA, implemented in the C++ library `libmrpt-srba`. It features the introduction of a *constant-time algorithm* for maintaining problem graphs with arbitrary topologies [4], as well as a generic design which allows turning RBA into *relative Graph-SLAM* (i.e. networks of relative pose constraints whose solutions are also relative poses). The *sparser* in its name refers to the proposal of creating graph edges in a way that increases the sparseness of the involved matrices [4].

2 Library installation

`libmrpt-srba` is one of the libraries of the Mobile Robot Programming Toolkit (MRPT). It is header-only and makes *intensive* use of templates and design patterns for the sake of customization, flexibility and extensibility.

Note however that it depends on other non-header-only libraries[1], so in practice before using `libmrpt-srba` in your program you need both (i) access to headers (`.h` files) and (ii) binary libraries to link against.

In Ubuntu, installing the package `libmrpt-dev` (version 1.0.0 or newer) is the easiest way to have everything ready to start coding your own programs. You can also install `mrpt-apps` for the application `srba-slam` and a set of sample datasets (see §8).

If your official repository has an older version of the package, use this PPA repository instead:

```
sudo add-apt-repository ppa:joseluisblancoc/mrpt
sudo apt-get update
sudo apt-get install libmrpt-dev mrpt-apps
```

Binary packages for Windows are also available online. If you prefer to build MRPT from sources, please visit the official web[2] for detailed instructions.

If you prefer to build MRPT from sources and do not need all the functionality of the rest of libraries, uncheck all the CMake configuration variables `BUILD_mrpt-*` for the unneeded modules. To make sure everything is working properly it is recommended to run the set of *unit tests* associated to `libmrpt-srba` with:

```
make run_tests_mrpt_srba  # To run the unit tests of mrpt-srba only
make test                 # To run all unit tests in MRPT
```

The tests include solving a few predefined datasets, verifying the expected results from Schür-complement functions, etc. At present there are 13 unit tests just for `libmrpt-srba` and more than 160 for the entire MRPT. It is almost impossible to guarantee that a library is bug-free with 100% certainty, but at least those tests assure that no regressions will be introduced along future features of bug fixes.

[1] The link-time dependencies are: `mrpt-base` for geometry, math auxiliary classes, serialization,... and `mrpt-opengl` for generating 3D representations of the RBA problems. Despite its name, the latter library can be built for platforms without any functional `OpenGL` implementation, though it is recommended to always visualize the results for getting a better insight of what is going on in your programs. The header-only library `Eigen` [7] is also a mandatory dependency, but an embedded version is shipped with `mrpt-base`.

[2] http://www.mrpt.org/

3 RBA primer

This manual will not explain the mathematical details of how RBA is modeled and solved – please, refer to cited papers. Though, it is mandatory to clearly establish which **entities** define an RBA problem before discussing the library API.

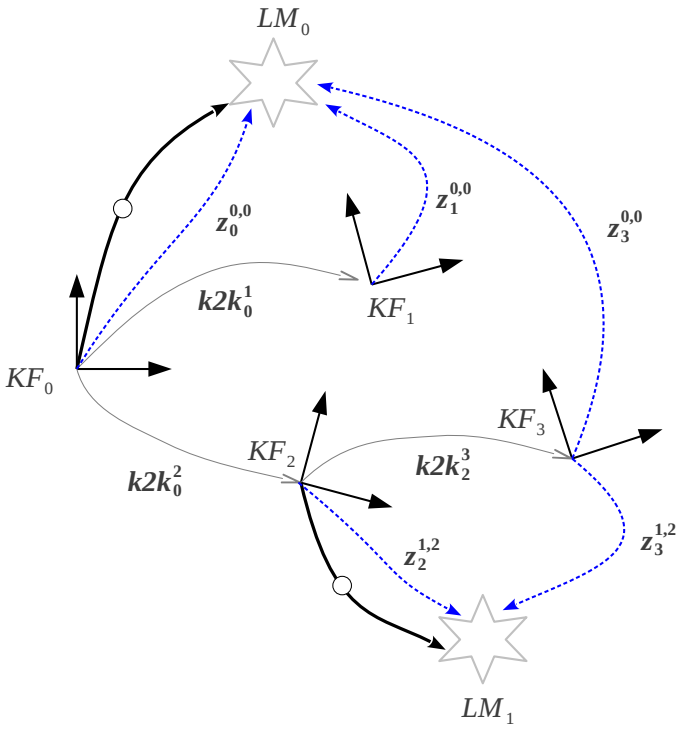


Figure 1: A toy RBA problem.

An example such that the one in Fig. 1 will help introducing the different elements. The illustration depicts many elements, some of which are *known* data, others are the problem *unknowns*. The goal of RBA is to recover a *maximum-likelihood estimation (MLE)* of those unknowns. Optionally, the covariances and cross-covariances between estimated variables can be also evaluated.

We find the following entities in `libmrpt-srba` (refer to Fig. 1):

- **Keyframes**: A keyframe (KF), as each KF_i in the figure, represents the *pose* of the robot (or the camera or whatever) at one particular instant of time. In RBA we will never work with the absolute coordinates of any of these KFs. This is completely different than "common" Bundle Adjustment, where these poses are the unknowns to estimate.
- **Keyframe-to-keyframe (k2k) edges**: An edge $k2k_i^j$ represents the relative pose of KF_i with respect to KF_j. Notice that "inverse poses"

(i.e. in the inverse order than one would expect from the edge direction) are stored for efficiency[3]. These edges are always treated as *unknowns* to be estimated from observation data. They can be parameterized in different ways (see §5.1), but the two choices in practice are either SE(2) or SE(3) poses.

- **Landmarks**: A landmark (LM) is any entity which can be observed from different locations. Typically a 2D or 3D point in space, but could be a line, segment, plane or any user-defined entity. They are represented in the Fig. 1 as stars denoted as LM_i. The concept of absolute coordinates of a LM does not exist in RBA.

- **Relative position and "base keyframe" of a LM**: Each LM is associated to exactly one KF, its base KF, with respect to which the LM has a *relative position*. This relative position can be either "known" ("*fixed*" in the C++ API) or "unknown", in which case it is also estimated during the problem optimization. There exist several possible ways of parameterizing relative positions, as discussed in §5.2. These relative positions are represented in the figure as thick edges with circle marks in the middle.

- **Observations**: An observation $z_k^{i,j}$ stands for any piece of sensory data which *is related*, somehow, with the position of the $i-th$ LM, whose base KF is j, as seen from the observing KF k. They are depicted as dashed lines in the figure above. Rank-deficient observations, like those from monocular cameras, are acceptable but two or more observations are then required before being able to estimate the relative position of the observed LM.

[3]In RBA we may need to chain sequences of relative poses and the direction will often be from newer KFs towards older KFs, hence if we store inverse poses we save the computational burden of inverting them over and over again.

4 Programmer's first steps

The central class in `libmrpt-srba` is the template `RbaEngine<>`, which adopts a "policy-based design" [1]:

```
template <
  class KF2KF_POSE_TYPE,
  class LM_TYPE,
  class OBS_TYPE,
  class RBA_OPTIONS = RBA_OPTIONS_DEFAULT>
class RbaEngine;
```

Thus, by setting each of the *template arguments* we literally control the process of code generation to address one particular instance of a RBA problem. Since all this happens at *compile time*, the compiler produces optimized code for the problem at hand (e.g. SSE2 code for multiplying matrices of a particular size), avoids code bifurcations, etc.

Due to the combinatorial nature of all the possibilities, detailed in §5, one template class can generate specialized code for dozens of concrete problems, including new ones defined by the user without modifying the library at all. The only price to pay is the longer compiling time associated to any complex C++ program that exploits metaprogramming with templates.

4.1 The simplest program

The following code illustrates the declaration of an RBA problem for 3D point landmarks, with SE(3) relative poses for keyframes and 3D range-bearing observations. Only two keyframes are defined, which means that after introducing the second one there will be only one k2k edge (an unknown), which will be estimated by the least-squares optimizer along the relative positions of all landmarks.

```
#include <mrpt/srba.h>

using namespace mrpt::srba;
using namespace std;

typedef RbaEngine<
  kf2kf_poses::SE3,         // Parameterization  KF-to-KF poses
  landmarks::Euclidean3D,   // Parameterization of landmark positions
  observations::RangeBearing_3D // Type of observations
  >
  my_srba_t;

int main(int argc, char**argv)
{
  my_srba_t rba;  //  Create an empty RBA problem

  // Define observations of KF #0:
  // ------------------------------------------------
  my_srba_t::new_kf_observations_t  list_obs;
  my_srba_t::new_kf_observation_t   obs_field;
  obs_field.is_fixed = false; // Landmarks have unknown relative
                              // positions (i.e. are unknowns )
```

```
  obs_field.is_unknown_with_init_val = false; // We don't have
       // any guess on the initial LM position (will invoke the
       // inverse sensor model)

  // For each observation:
  for (...) {
    obs_field.obs.feat_id = ...; // The landmark ID
    obs_field.obs.obs_data.range = ...;
    obs_field.obs.obs_data.yaw   = ...;
    obs_field.obs.obs_data.pitch = ...;
    list_obs.push_back( obs_field );
  }

  //  This is the main API entry point: Define KF #0
  my_srba_t::TNewKeyFrameInfo new_kf_info; // Placeholder of out info.
  rba.define_new_keyframe(
    list_obs,       // Input observations for the new KF
    new_kf_info,    // Output info
    true            // Run optimization of the local area
    );

  // Define observations of KF #1:
  // ------------------------------------------
  list_obs.clear();
  // For each observation:
  for (...) {
    obs_field.obs.feat_id = ...; // The landmark ID
    obs_field.obs.obs_data.range = ...;
    obs_field.obs.obs_data.yaw   = ...;
    obs_field.obs.obs_data.pitch = ...;
    list_obs.push_back( obs_field );
  }

  //  This is the main API entry point: Define KF #1
  rba.define_new_keyframe(
    list_obs,       // Input observations for the new KF
    new_kf_info,    // Output info
    true            // Run optimization of the local area
    );

  cout << "Created KF #" << new_kf_info.kf_id
   << " | # kf-to-kf edges created:" <<
   new_kf_info.created_edge_ids.size() << endl <<
   "Optimization error: " <<
   new_kf_info.optimize_results.total_sqr_error_init <<
   " -> " <<
   new_kf_info.optimize_results.total_sqr_error_final << endl;

  // Save RBA graph as Graphviz file:
  rba.save_graph_as_dot("graph.dot", true /* LMs=save */);

  return 0;
}
```

4.2 API entry points

It must be highlighted that, at present, the only API for inserting new KFs into the RBA problem is the method `define_new_keyframe()`:

```
template <...> class RbaEngine {
    ...
  void define_new_keyframe(
    const typename traits_t::new_kf_observations_t  & obs,
    TNewKeyFrameInfo  & out_new_kf_info,
    const bool        run_local_optimization = true
    );
    ...
};
```

which accepts as its main input the list of all the observations gathered at this new KF. This method *always creates a new KF*, without analyzing whether it was too close or too far from other KFs. Thus, it is (for now) the user's responsibility not to call the method too often, which would create an unnecessarily large amount of KFs with the subsequent degradation of the performance. *The constant-time complexity* of the overall method assumes that there exists a maximum number of KFs per area –which is perfectly reasonable[4].

As output, this method fills a `RbaEngine<>::TNewKeyFrameInfo` structure with:

- The ID of the newly-created KF.
- The list of all the created KF-to-KF edges. At least one, or more in the case of loop-closures. The way in which edges are created depends on the *edge-creation policy* (see §6.2). An exception will be raised if no suitable edge is found to connect the new KF to the existing graph.
- The numerical results from the local optimization process: the exact list of unknowns that undergone optimization, the initial and final root mean-squared error (RMSE), etc.

Please, refer to the pseudocode description of this method in §11.1 for a better insight of what happens inside.

[4] TO DO: Future versions of the software will probably incorporate an automatic mechanism to help deciding whether to insert KFs.

4.3 Tutorials

More complete versions of the program above, including sample datasets and rendering of the optimization result as OpenGL scenes are shipped with the source code[5]. Screenshots are shown in Figs. 2–3.

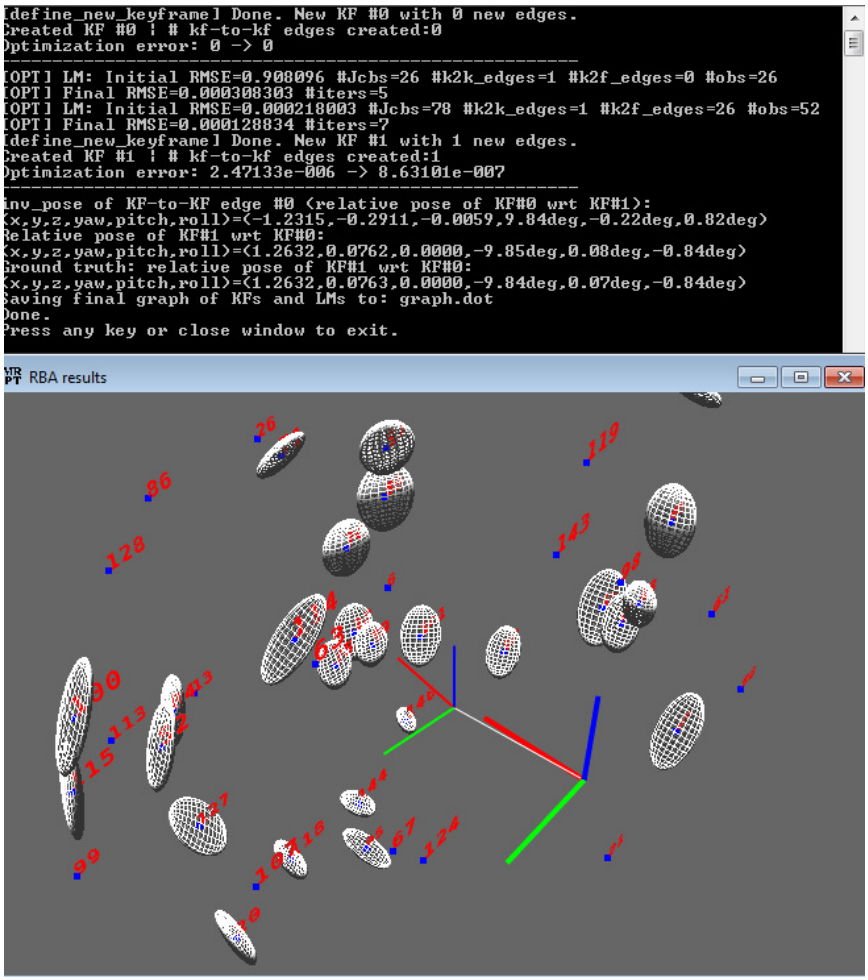


Figure 2: Screenshot for `tutorial-srba-range-bearing-se3.cpp`. Only two KFs are defined in this example.

[5]See the directory `[MRPT]/samples/srba-examples/srba-tutorials/`, or browse online.

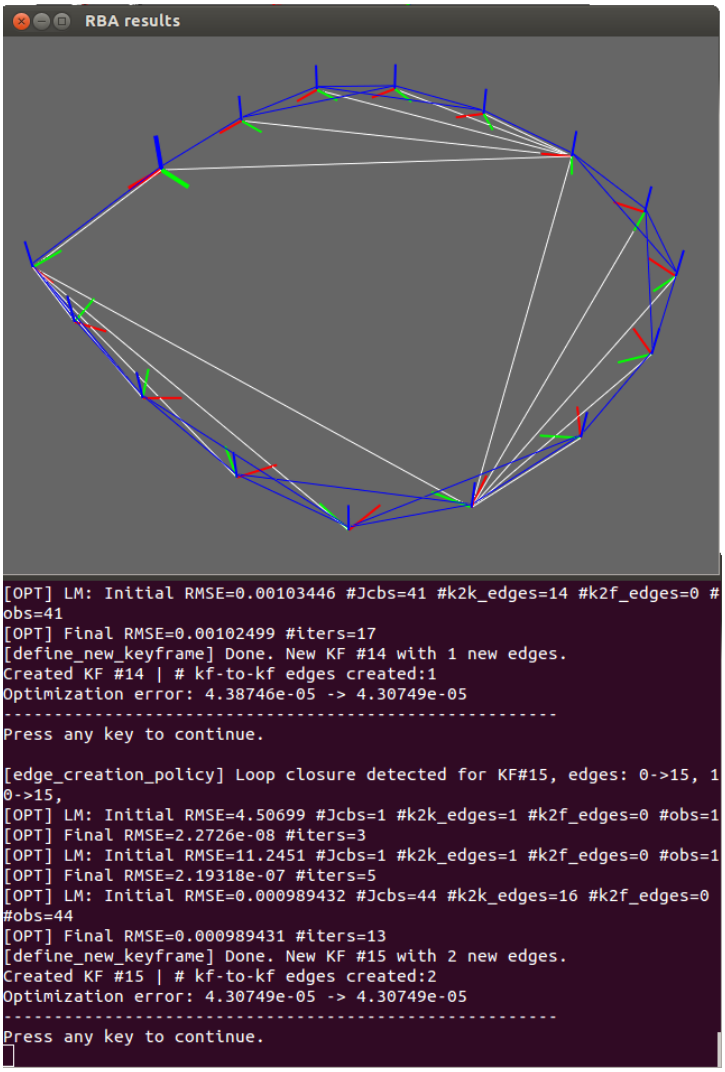


Figure 3: Screenshot for `tutorial-srba-relative-graph-slam.cpp`. White lines represent KF-to-KF edges (unknowns), blue lines are observations (known data).

5 Configuring `RbaEngine<>`: template arguments

In the following, all class names assume the existence of a previous:

```
using namespace mrpt::srba;
```

5.1 `KF2KF_POSE_TYPE`: KF-to-KF relative poses

This template argument selects the model for the relative poses between keyframes. The two natural possibilities are SE(2) and SE(3) poses, which you should employ depending on whether your problem can be considered *planar* or not:

- `kf2kf_poses::SE2` : For 2D relative poses, i.e. they consists of a (x, y) displacement plus a heading ϕ. Poses are mapped to `mrpt::poses::CPose2D` classes.
- `kf2kf_poses::SE3` : For 3D relative poses. Given that the least-squares optimization runs on the linearized neighborhood of the manifold [3] around the instantaneous solutions, the choice between different parameterizations (i.e. quaternions, Euler angles,etc.) becomes irrelevant. In this case poses become `mrpt::poses::CPose3D` classes, which internally hold the 3×3 rotation matrices for SO(3) rotations and (x, y, z) vectors for the translational parts, with optional conversion to/from quaternions and yaw/pitch/roll angles.

Notice that the usage of 2D relative poses limits the robot trajectory to one single plane, but does not restrict landmarks to also be planar. It is perfectly legal to use planar poses and 3D landmarks.

5.2 `LM_TYPE`: Relative landmark parameterizations

- `landmarks::Euclidean2D` : Point landmarks, parameterized with 2D Euclidean coordinates with respect to the base KF.
- `landmarks::Euclidean3D` : Point landmarks, parameterized with 3D Euclidean coordinates with respect to the base KF.
- `landmarks::RelativePoses2D` : A kind of "fake landmark" used in relative graph-SLAM. Represents the pose of the base KF, which can be observed from another KF.

The implementation of these models can be seen in:
`#include <mrpt/srba/models/landmarks.h>`.

5.3 OBS_TYPE: Observation types

The following observations are implemented and ready to use in the library:

- `observations::MonocularCamera` : A pair of pixel coordinates (x, y) for the observed landmark.
- `observations::StereoCamera` : Pixel coordinates for one left and one right camera.
- `observations::Cartesian_2D` : The (x, y) coordinates of the observed landmark, relative to the sensor.
- `observations::Cartesian_3D` : The (x, y, z) coordinates of the observed landmark, relative to the sensor.
- `observations::RangeBearing_2D` : The distance and the angle (polar coordinates) of the observed landmark, relative to the sensor.
- `observations::RangeBearing_3D` : The distance and two angles (yaw and pitch) of the observed landmark, relative to the sensor.
- `observations::RelativePoses_2D` : The (x, y, ϕ) pose of the observed KF, relative to the observer KF.

The implementation of these models can be seen in:
`#include <mrpt/srba/models/observations.h>`.

5.4 Sensor models

In order to use together each combination of landmark parameterization (`LM_TYPE`) and observation (`OBS_TYPE`) a correct *sensor model* must be implemented which takes care of providing a set of Jacobians, an inverse sensor model, etc.

These are the models already implemented with this library:

- `landmarks::Euclidean3D + observations::MonocularCamera`: 3D landmarks in Euclidean coordinates, observed with a monocular camera (without distortion).
- `landmarks::Euclidean3D + observations::StereoCamera`: 3D landmarks in Euclidean coordinates, observed with a stereo camera (without distortion).
- `landmarks::Euclidean2D + observations::Cartesian_2D`: 2D landmarks in Euclidean coordinates, whose Euclidean coordinates are directly observed.

- `landmarks::Euclidean3D + observations::Cartesian_3D`: 3D landmarks in Euclidean coordinates, whose Euclidean coordinates are directly observed.
- `landmarks::Euclidean2D + observations::RangeBearing_2D`: 2D landmarks in Euclidean coordinates, observed via range and bearing.
- `landmarks::Euclidean3D + observations::RangeBearing_3D`: 3D landmarks in Euclidean coordinates, observed via range and bearing.
- `landmarks::RelativePoses2D + observations::RelativePoses_2D`: The sensor model for 2D relative graph-SLAM.

The implementation of these models can be seen in:
`#include <mrpt/srba/models/sensors.h>`.

5.5 `RBA_OPTIONS`: Other options

This template argument must be a user-defined structure with a set of `typedef`s that configure specific aspects of the RBA problem, explored below:

```
struct my_rba_options
{
  typedef <TYPE_1>  sensor_pose_on_robot_t;
  typedef <TYPE_2>  obs_noise_matrix_t;
  typedef <TYPE_3>  solver_t;
};
```

5.5.1 Choices for `sensor_pose_on_robot_t`

- `options::sensor_pose_on_robot_none`: The pose of a KF corresponds exactly to the pose of the sensor. That is, there is no distinction between the pose of the "robot" and that of the "sensor".
- `options::sensor_pose_on_robot_se3`: The sensor is located at an arbitrary pose with respect to the frame of reference of each KF (the "robot"). This is the most common case of sensors placed on a mobile robot, and should be used if we are interested in obtaining the pose of the robot instead of that of the sensor itself. In the case of cameras, this option is employed within the program `srba-slam` to establish a change of coordinates between the Z-points-up standard for robot poses and the Z-points-forward of typical camera models. Using this option has a small computational overhead in comparison to the one above.

The list of possible types can be found in:
`#include <mrpt/srba/srba_options_sensor_pose.h>`.

5.5.2 Choices for `obs_noise_matrix_t`

This option controls the way in which the information matrix $\mathbf{\Lambda}_k$ for the $k-th$ observation (inverse of the sensor error covariance matrix) is incorporated into the weighted least-squares optimizer. Depending on the chosen model, different data fields to set the model parameters will be available under `srba.parameters.obs_noise`.

- `options::observation_noise_identity`: This is the most computationally efficient method, since it is assumed that:

$$\mathbf{\Lambda}_k = \Lambda \mathbf{I} \tag{1}$$

  with $\mathbf{I}$ an identity matrix of the correct size. The error is exactly the same for all observations.

- `options::observation_noise_constant_matrix`: In this case $\mathbf{\Lambda}_k$ can be any arbitrary matrix[6]. The same matrix will be used for all observations.

The list of possible types can be found in:
`#include <mrpt/srba/srba_options_sensor_noise.h>`.

5.5.3 Choices for `solver_t`

At present all solvers are different versions of the Levenberg-Marquardt (LM) algorithm:

- `options::solver_LM_schur_dense_cholesky`: A LM solver, using the Schür complement to build a reduced system of equations for relative poses only, which is then solved using a dense Cholesky factorization.

- `options::solver_LM_schur_sparse_cholesky`: Like above, but using a sparse Cholesky factorization (with the `CSparse` library) for the reduced system.

- `options::solver_LM_no_schur_sparse_cholesky`: A LM solver, directly using a sparse Cholesky factorization (with the `CSparse` library) on the entire system of equations (both poses and landmarks).

The list of possible types can be found in:
`#include <mrpt/srba/srba_options_solver.h>`.

[6] As long as it is a valid information matrix: symmetric, positive definite.

6 Configuring `RbaEngine<>`: dynamic parameters

In contrast to the template arguments, which determine the type of problem at compile time, there exist another set of parameters, suitable for change at run-time.

They are all found in the public field `parameters`, which in turn is built up of other `structs` whose contents are determined at compile time depending on the template arguments:

```
template <...> class RbaEngine {
  ...
  struct TAllParameters
  {
    /** Different parameters for the SRBA methods */
    TSRBAParameters  srba;

    /** Sensor-specific parameters (sensor calibration, etc.) */
    typename OBS_TYPE::TObservationParams   sensor;

    /** Parameters related to the relative pose of sensors wrt the
        robot (if applicable) */
    typename RBA_OPTIONS::sensor_pose_on_robot_t::parameters_t
        sensor_pose;

    /** Parameters related to the sensor noise covariance matrix */
    typename RBA_OPTIONS::obs_noise_matrix_t::parameters_t  obs_noise;
  };

  TAllParameters parameters;
  ...
};
```

6.1 Depth of spanning trees

Two of the most important parameters are:

```
template <...> class RbaEngine {
  ...
  struct TSRBAParameters
  {
  ...
  /** Maximum depth for maintained spanning trees. */
  topo_dist_t          max_tree_depth;
  /** The maximum topological distance of keyframes
     to be optimized around the most recent keyframe. */
  topo_dist_t          max_optimize_depth;
  ...
  };
  ...
};
```

In general, `parameters.srba.max_optimize_depth` should not be larger than `parameters.srba.max_tree_depth`, since optimization needs using the prebuilt spanning trees. Normally, both values will be the same and very small (e.g. 3 or 4).

6.2 Edge-creation policy

Another critical parameter is:

```
template <...> class RbaEngine {
  ...
  struct TSRBAParameters
  {
  ...
    TEdgeCreationPolicy edge_creation_policy;
  ...
  };
  ...
};
```

Here, `parameters.srba.edge_creation_policy` controls the policy to decide, given a set of observations for a new KF, how many KF-to-KF edges must be created and how should they be connected.

Check the doxygen documentation for `TEdgeCreationPolicy` for a description of the possibilities.

Note that the user can implement new custom policies by overriding the virtual method:

```
template <...> class RbaEngine {
  ...
  virtual void edge_creation_policy(
    const TKeyFrameID                  new_kf_id,
    const typename traits_t::new_kf_observations_t    & obs,
    std::vector<TNewEdgeInfo> &new_k2k_edge_ids );
  ...
};
```

in which case the value of `parameters.srba.edge_creation_policy` is ignored.

6.3 Levenberg-Marquardt solver parameters

All these can be found under `parameters.srba.*`:

6.3.1 Fluid relinearization

`double min_error_reduction_ratio_to_relinearize`**:** If the error-reduction ratio between two consecutive solver iterations is below this threshold, Jacobians will not be re-evaluated at the new solution. This a kind of "fluid relinearization", which reduces the computational burden.

6.3.2 Covariance recovery

`TCovarianceRecoveryPolicy cov_recovery`**:** Controls how or whether covariances are to be recovered from the final Hessian matrix used by the solver. Naive exact recovery implies inverting a sparse matrix, leading to a dense (fully-correlated) covariance matrix, which is normally expensive. That is why, by default, only an approximation of the landmark covariances are evaluated. See the Doxygen documentation for `TCovarianceRecoveryPolicy` for further details.

Note that the Hessian itself (i.e. the sparse inverse of the covariance of all unknowns) is also always provided in the member `extra_results`:

```
template <...> class RbaEngine {
  ...
  struct TOptimizeExtraOutputInfo
  {
    ...
    typename RBA_OPTIONS::solver_t::extra_results_t    extra_results;
    ...
  };
  ...
};
```

after each successful optimization. The specific format in which this Hessian is provided (i.e. sparse vs. dense) depends on the selected solver (see §5.5.3). The user can apply any advanced algorithm for recovering only the part of the covariances really required for the application at hand.

6.3.3 Others

These other parameters are also worth mentioning:

- `bool optimize_new_edges_alone`: Runs an optimization with the new KF-to-KF edges, one by one, as if they were the unique unknowns before running the local area optimization. This assures that there are no variables with such a bad initialization that could ruin the overall estimation. Normally should be left to `true`, disable for a speed up if you are sure this step is not needed in your problem.
- `bool use_robust_kernel`: Employs a robustifying kernel (pseudo-Huber) to reduce the impact of outliers. Enabled by default. Should also tune the `kernel_param` parameter.
- `size_t max_iters`: Maximum number of iterations in the least-squares solver.

7 Accessing `RbaEngine<>`: the RBA problem graph

This section is meant to be complemented by checking the complete Doxygen documentation online.

7.1 Programmatic access to edges and nodes

The first step for programmatic access to the data structures that represent KFs, observations, etc. is getting a (`const`, i.e. unmodifiable) reference to the internal object that holds the entire RBA state:

```
template <...> class RbaEngine {
    ...
  typedef TRBA_Problem_state<...> rba_problem_state_t;
  ...
  const rba_problem_state_t & get_rba_state() const;
    ...
};
```

which is an instance of the class `TRBA_Problem_state<...>`. It is recommended to check its Doxygen documentation for a better description of all the existing data fields, as well as the description in §10.2. The most relevant public data fields are:

- `keyframe_vector_t keyframes`: A vector with information about each existing KF, indexed by their ID.
- `k2k_edges_deque_t k2k_edges`: All KF-to-KF edges, indexed by their ID.
- `TRelativeLandmarkPosMap unknown_lms`: The positioning of each (non-fixed) landmark.

7.2 Exporting as OpenGL objects

The following method is provided to convert (part of) an RBA problem into a graphical representation suitable for rendering:

```
template <...> class RbaEngine {
    ...
  void build_opengl_representation(
    const srba::TKeyFrameID root_keyframe,
    const TOpenGLRepresentationOptions &options,
    mrpt::opengl::CSetOfObjectsPtr out_scene,
    mrpt::opengl::CSetOfObjectsPtr out_root_tree = mrpt::opengl::
        CSetOfObjectsPtr()
    ) const;
    ...
};
```

It is recommended to check the Doxygen documentation of `TOpenGLRepresentationOptions` to see all available rendering options, but just a few important remarks:

- In RBA there is no "global map", obviously. So each graphical representation must explicitly choose the origin of coordinates. This is done by selecting the `root_keyframe` (a KF's ID) which becomes the root of a spanning tree of all nearby KFs up to some maximum topological distance (settable in `options`).
- For obtaining relative coordinates with respect to the selected root, this method is capable of building deeper spanning trees than those maintained for online optimization. However, this clearly has an unbounded computational cost that grows with the desired topological distance, so keep this distance as reduced as possible if the intention is to render the state of the RBA problem in real time. In particular, it is recommended to set:

```
typename my_srba_t::TOpenGLRepresentationOptions opengl_params;
opengl_params.span_tree_max_depth = rba.parameters.srba.
    max_tree_depth; // Render these past keyframes at most.
```

7.3 Exporting as Graphviz graphs

The following method allows exporting the RBA problem, in its current state, to a plain text file in the standard Graphviz format:

```
template <...> class RbaEngine {
    ...
  bool save_graph_as_dot(
    const std::string &targetFileName,
    const bool all_landmarks = false
    ) const;
    ...
};
```

Note that exporting all landmarks will lead to excessively dense graphs in any mid to large-size problem. An example is shown in Fig. 4.

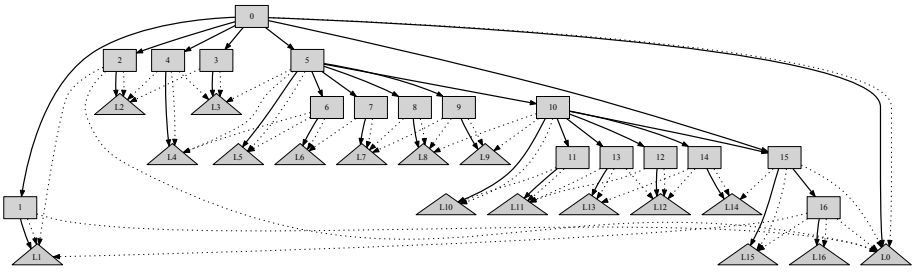


Figure 4: Example of an RBA graph rendered with Graphviz's `dot`: rectangles are keyframes, triangles are landmarks, dotted lines are observations and solid lines are either kf-to-kf edges or landmark relative positions (with respect to their *base* keyframes).

7.4 Generic graph visitor

There exists a fully-customizable breadth-first search (BFS) *visitor*, which allows any user-supplied class to process every node and edge in the RBA graph, starting at a given root KF and expanding in a classic breadth-first fashion:

```
template <...> class RbaEngine {
    ...
  template <
    class KF_VISITOR,
    class FEAT_VISITOR,
    class K2K_EDGE_VISITOR,
    class K2F_EDGE_VISITOR
    >
  void bfs_visitor(
    const TKeyFrameID  root_id,
    const topo_dist_t  max_distance,
    const bool rely_on_prebuilt_spanning_trees,
    KF_VISITOR       & kf_visitor,
    FEAT_VISITOR     & feat_visitor,
    K2K_EDGE_VISITOR & k2k_edge_visitor,
    K2F_EDGE_VISITOR & k2f_edge_visitor ) const;
    ...
};
```

The option `rely_on_prebuilt_spanning_trees` is the only one requiring a few words. It selects between two different ways of performing the BFS:

- Relying on prebuilt spanning trees: such that the search is limited to the maximum depth of those trees, and
- Not relying on them: then a complete BFS algorithm is run, suitable for exploration of the entire RBA problem graph.

The four template argument, the visitor classes, must be defined by the user. They may be four different classes or just one. Next follows a sketch of the expected minimum interface for each class:

```
/* Implementation of FEAT_VISITOR */
struct MY_FEAT_VISITOR
{
  bool visit_filter_feat(
    const TLandmarkID lm_ID,
    const topo_dist_t cur_dist)
  {
    // Return "true" if it's desired to visit this landmark node.
  }
  void visit_feat(
    const TLandmarkID lm_ID,
    const topo_dist_t cur_dist)
  {
    // Process this landmark node.
  }
};
```

```
/* Implementation of KF_VISITOR */
struct MY_KF_VISITOR
{
  bool visit_filter_kf(
    const TKeyFrameID kf_ID,
    const topo_dist_t cur_dist)
  {
    // Return "true" if it's desired to visit this keyframe node.
  }
  void visit_kf(
    const TKeyFrameID kf_ID,
    const topo_dist_t cur_dist)
  {
    // Process this keyframe node.
  }
};
```

```
/* Implementation of K2K_EDGE_VISITOR */
struct MY_K2K_EDGE_VISITOR
{
  bool visit_filter_k2k(
    const TKeyFrameID current_kf,
    const TKeyFrameID next_kf,
    const k2k_edge_t* edge,
    const topo_dist_t cur_dist)
  {
    // Return "true" if it's desired to visit this kf-to-kf edge.
  }
  void visit_k2k(
    const TKeyFrameID current_kf,
    const TKeyFrameID next_kf,
    const k2k_edge_t* edge,
    const topo_dist_t cur_dist)
  {
    // Process this kf-to-kf edge.
  }
};
```

```
/* Implementation of K2F_EDGE_VISITOR */
struct MY_K2F_EDGE_VISITOR
{
  bool visit_filter_k2f(
    const TKeyFrameID current_kf,
    const k2f_edge_t* edge,
    const topo_dist_t cur_dist)
  {
    // Return "true" if it's desired to visit this kf-to-feat edge.
  }
  void visit_k2f(
    const TKeyFrameID current_kf,
    const k2f_edge_t* edge,
    const topo_dist_t cur_dist)
  {
    // Process this kf-to-feat edge.
  }
};
```

8 The `srba-slam` application

This program is ready to use in binary installations of MRPT, so it could be a good starting point to test the possibilities of SRBA without having to write a single line of code.

8.1 Interface

`srba-slam` is a command-line program that loads a dataset from plain text files and processes it with a given instance of the generic `RbaEngine`. A typical call must specify the desired models for relative poses, landmarks and observations, as well as the data set file:

```
srba-slam {--se2|--se3} {--lm-2d|--lm-3d} --obs [StereoCamera|...]
          -d DATASET.txt [--sensor-params-cfg-file SENSOR_CONFIG.cfg]
          [--noise NOISE_SIGMA] [--verbose {0|1|2|3}] [--step-by-step]
```

The sensor configuration file (`--sensor-params-cfg-file`) is only required for certain types of observations (e.g. camera calibration files for vision sensors). Noise parameters can be provided via `--noise` (and `--noise-ang` for angular components) to modify the weighting of least squares optimization and, optionally (if `--add-noise` is set) actual random noise will be also generated and added to the dataset.

Naturally, not all possible RBA problems are precompiled in this application. To see the list of available problems, run:

```
srba-slam --list-problems
```

To explore all the existing parameters, please execute:

```
srba-slam --help
```

or see the program man page:

```
man srba-slam
```

8.2 Running with sample datasets

A small collection of datasets is shipped with MRPT and can be found under:

```
[MRPT_ROOT]/share/mrpt/datasets/srba-demos/
```

where `[MRPT_ROOT]` is:

- `C:/Program Files/MRPT/` if a Windows binary package was installed,
- `/usr/` if a GNU/Linux binary package was installed, or
- the root directory of source packages.

In all cases, shipped files are not the datasets themselves, but the *sources* for the RWT dataset generator[7]. Follow instructions in the `README.txt` to generate the datasets.

Then, see one of the `*.sh` files in the same directory for example calls to `srba-slam` for each dataset. As an example, this is how to run the 2D relative graph-SLAM dataset:

```
srba-slam --se2 --graph-slam -d dataset_30k_rel_graph_slam_SENSOR.txt
          --submap-size 10
          --max-spanning-tree-depth 3 --max-optimize-depth 3
          --verbose 1 --noise 0.001 --noise-ang 0.2 --add-noise
          --gt-map dataset_30k_rel_graph_slam_GT_MAP.txt
          --gt-path dataset_30k_rel_graph_slam_GT_PATH.txt
          # --step-by-step
```

[7]See: http://code.google.com/p/recursive-world-toolkit/

9 Spanning trees

The need to hold spanning trees for every KF in the problem is motivated by the realization that *topological paths* appear in the observation model of landmarks (or in that of relative poses in relative Graph-SLAM). In particular, we need the shortest path between an observing KF and the base KF of the observed landmark, as highlighted in Figure 5.

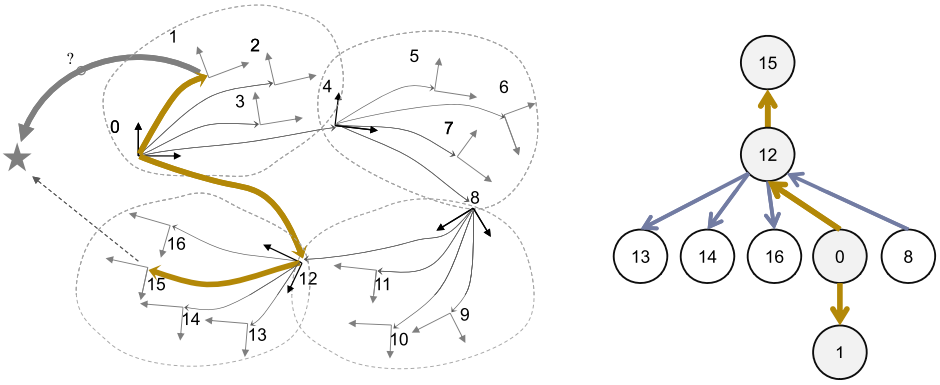
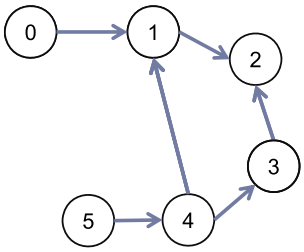


Figure 5: (left) Paths between keyframes involved in observation models. (right) The corresponding spanning tree.

Internally, each spanning tree is stored as a pair of sparse tables:

- $ST.D[i][j]$: A symmetric table with the topological distance between keyframes $i \leftrightarrow j$.
- $ST.N[i][j]$: This table holds the next edge to follow if we are at i and want to reach j.

Our paper [4] describes an algorithm for incrementally update those tables as new keyframes and edges are added to the problem.



ST.D (minimum distances)

From \ To	0	1	2	3	4	5
0	*					
1	1	*		(symmetric)		
2	2	1	*			
3	3	2	1	*		
4	2	1	2	1	*	
5	3	2	3	2	1	*

ST.N (next)

From \ To	0	1	2	3	4	5
0	*	0␣1	0␣1	0␣1	0␣1	0␣1
1	0␣1	*	1␣2	1␣2	4␣1	4␣1
2	1␣2	1␣2	*	3␣2	3␣2	1␣2
3	3␣2	3␣2	3␣2	*	4␣3	4␣3
4	4␣1	4␣1	4␣1	4␣3	*	5␣4
5	5␣4	5␣4	5␣4	5␣4	5␣4	*

Figure 6: (top) An example RBA problem. (left-bottom) The $ST.D$ table. (right-bottom) The $ST.N$ table.

10 Library inner structure

10.1 Directory layout

As said above, `libmrpt-srba` is a header-only library. Thus the headers include both: "public" declarations and template implementations.

The user normally includes just one public header,

```
#include <mrpt/srba.h>
```

which in turn includes all the other required headers. However, thinking of those programmers that intend modifying or extending the library, the directory layout of all headers is explained below.

- ▷ **mrpt**
  - ▷ `srba.h`: The main file that users should include.
  - ▷ **srba**
    - ▷ RbaEngine.h
    - ▷ (other"public" headers)
    - ▷ **models**
      - ▷ `kf2kf_poses.h`
      - ▷ `landmarks.h`
      - ▷ `observations.h`
      - ▷ `sensors.h`
    - ▷ **impl**
      - ▷ Implementations of template methods.

10.2 Data structures

All the data types and containers employed in this library have been carefully selected taking into account that the goal is achieving a constant (bounded) computational cost: no matter how many keyframes (N) or landmarks (M) already exist in the problem, introducing a new keyframe with a bounded number of observations should *never* lead to operations that scale with N or M, not even logarithmically.

This need has materialized into the requirement of a network of crossed *references* (in the C++ sense) between the different containers, which may seem quite complex at first glance but is necessary to avoid look-up operations and assure the efficiency of all the SRBA sub-algorithms.

A detailed view of these relationships is provided in Fig. 7 for the reference of those interested in the low-level implementation details.

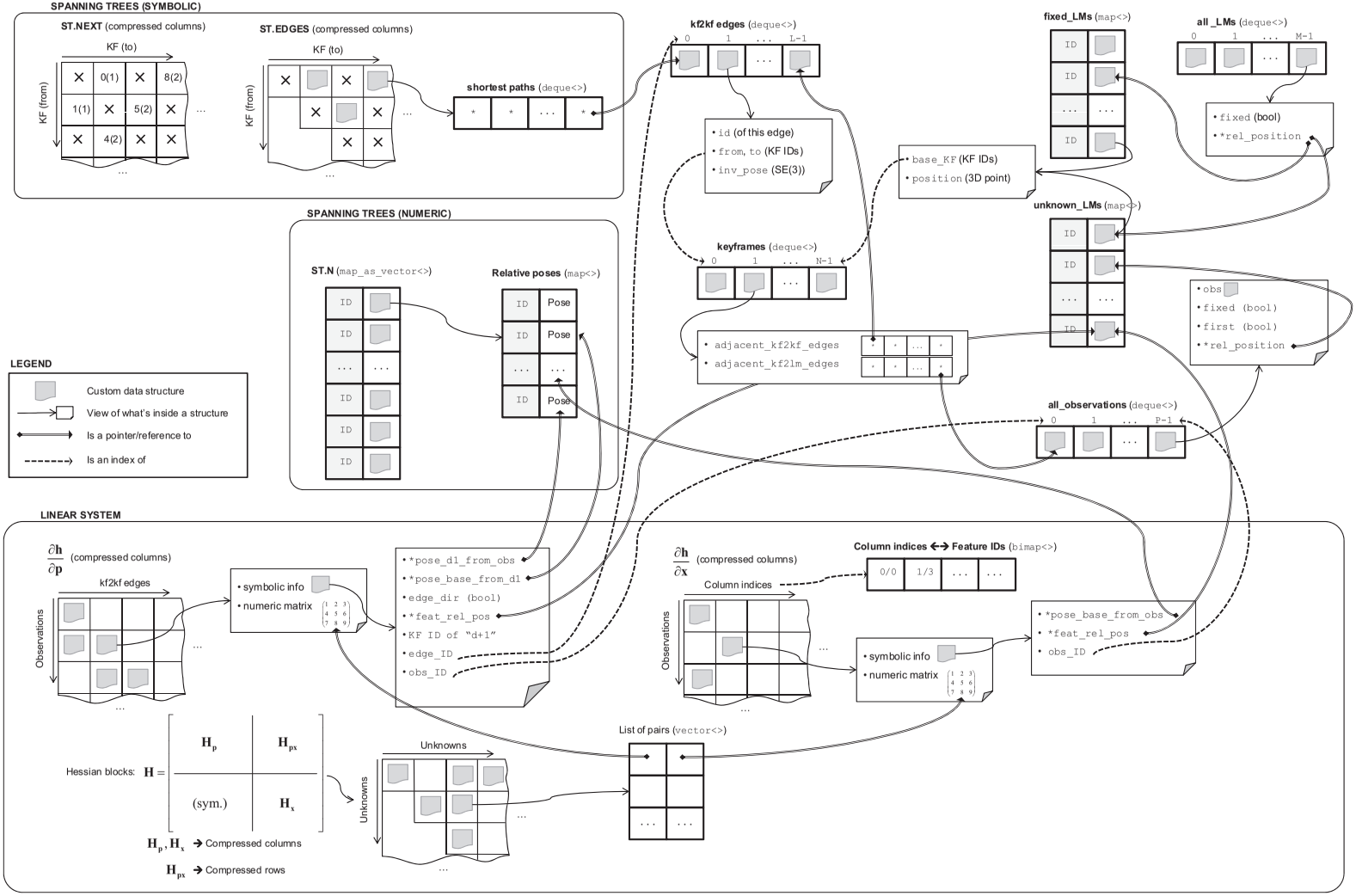


Figure 7: Detailed data structures. Refer to the legend for the format of structures and pointers/references.

11 Pseudo-code

These symbols are employed in the following:

- D_{max}: Maximum depth for which spanning trees are maintained. It is also the maximum topological distance from the given KF to look for unknowns to optimize during a "local area" optimization. Obviously, all KFs within that distance are also in the spanning tree.
- N_o: Number of observed landmarks from the new KF being inserted in the problem.
- N_R: The maximum number of reachable KFs for a fixed D_{max}. It is reasonable to consider that this number is bounded for any map, as long as redundant KFs are not continuously added for the same area. Figure 8 illustrates the meaning of N_R, in a particular example where the local optimization is performed centered at the keyframe #5.

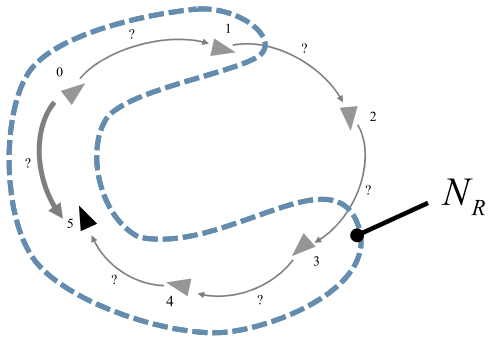


Figure 8: Example regarding the definition of N_R. Here, $N_R = 5$ including the origin #5.

- γ: A constant coefficient that models the "sparseness" of the graph being solved by least-squares minimization.
- $(\mathbf{z}_n^i, \alpha_n^i)$: The $i-th$ individual observation of the $n-th$ time step, and its corresponding data association (i.e. its correspondence, the ID of an existing landmark or a new ID if this is a new LM).

11.1 `define_new_keyframe()`

Algorithm 1 `srba_define_new_keyframe`
Worst case: $O((\gamma N_R)^3 + N_o(D_{max} + \log N_R))$

C++: `RBA_Problem::define_new_keyframe()`

Input: $(\mathbf{z}_n, \alpha_n) = \left\{ \left(\mathbf{z}_n^1, \alpha_n^1\right), ..., \left(\mathbf{z}_n^{N_o}, \alpha_n^{N_o}\right) \right\}$ ▷ Set of N_o new observations $\mathbf{z}_n^i$ and their data association α_n^i
Input: *run_local_optimization* ▷ Whether to also run optimizations
Output: The updated, locally consistent map

// Update keyframes (KFs) data structures
$n \leftarrow$number of KFs in the map ▷ Assign a free ID to the new KF – $O(1)$
$KF[n] \leftarrow$empty KF data structure ▷ Insert at the end of `std::map` – $O(1)$

// Apply edge-creation policy to decide how to handle loop closures, etc.
while $\varnothing \neq [(i_k \leftrightarrow n) =$ `decide_edge_to_create`$()]$ **do** ▷ $O(N_o \log N_R)$
  `add_kf2kf_edge`$(i_k \leftrightarrow n)$ ▷ Update KF-to-KF edge structures – $O(1)$
  `update_sym_spanning_trees`$(i_k \leftrightarrow n)$ ▷ $O(N_R^2 \log N_R)$
end while ▷ Typ. iterations: $O(\gamma)$

// Update symbolic Jacobian structures ▷ $O\left(N_o(D_{max} + \log N_R)\right)$
for each $\left(\mathbf{z}_n^i, \alpha_n^i\right) \in (\mathbf{z}_n, \alpha_n)$ **do** ▷ For each of the N_o new observations
  `add_observation`$\Big($ $\underbrace{\mathbf{z}_n^i}_{\text{obs. data}}$, $\underbrace{n}_{\text{observing KF}}$, $\underbrace{\alpha_n^i}_{\text{landmark ID}}$ $\Big)$ ▷ $O(D_{max} + \log N_R)$
end for

if *run_local_optimization* **then**
  if *optimize_new_edges_alone* **then**
    // Initialize new edges
    for each $(i_k \leftrightarrow n)$ **do** ▷ For each new kf2kf edge created above
      `non_linear_optimizer`$(i_k \leftrightarrow n)$ ▷ $O(N_o)$
    end for
  end if

  // Update SLAM estimation
  edges_to_optimize $\leftarrow$ all within a D_{max} distance from n ▷ $O(N_R)$
  `non_linear_optimizer`(**edges_to_optimize**) ▷ $O((\gamma N_R)^3)$
end if

11.2 `update_sym_spanning_trees()`

This algorithm is explained in [4].

Algorithm 2 `update_sym_spanning_trees` Worst case: $O(N_R^2 \log N_R)$

Input:
$(i_k \leftrightarrow n)$ ▷ A new edge
D_{max} ▷ The maximum desired depth of span. trees

1: $ST_{D_{max}-1}(i_k) \leftarrow \{\forall v / d(v, i_k) \leq D_{max} - 1\}$ ▷ $O(N_R)$
2: $ST_{D_{max}}(n) \leftarrow \{\forall v / d(v, n) \leq D_{max}\}$ ▷ $O(N_R)$

3: **for each** $r \in ST_{D_{max}}(n)$ **do** ▷ $O(N_R)$ iterations
4: **for each** $s \in ST_{D_{max}-1}(i_k)$ **do** ▷ $O(N_R)$ iterations
5: // New tentative distance between r and s
6: $d \leftarrow \mathcal{ST}.\mathcal{D}[n][r] + \mathcal{ST}.\mathcal{D}[i_k][s] + 1$ ▷ $O(\log N_R)$
7: **if** $(s \in$ `spanning_tree`(r) **and** $d < \mathcal{ST}.\mathcal{D}[r][s])$ **or** $(s \notin$ `spanning_tree`(r) **and** $d \leq D_{max})$ **then** ▷ $O(\log N_R)$
8: // Shorter or new path found. Update trees:
9: $\mathcal{ST}.\mathcal{D}[r][s] \leftarrow d$
10: $\mathcal{ST}.\mathcal{N}[r][s] \leftarrow \begin{cases} i_k & r = n \\ \mathcal{ST}.\mathcal{N}[r][n] & r \neq n \end{cases}$
11: $\mathcal{ST}.\mathcal{D}[s][r] \leftarrow d$ ▷ $O(\log N_R)$
12: $\mathcal{ST}.\mathcal{N}[s][r] \leftarrow \begin{cases} n & s = i_k \\ \mathcal{ST}.\mathcal{N}[s][i_k] & s \neq i_k \end{cases}$
13: **end if**
14: **end for**
15: **end for**

References

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Professional, 2001.

[2] T. Bailey and H. Durrant-Whyte. Simultaneous localisation and mapping (SLAM): Part II-State of the art. *Robotics and Automation Magazine*, 13:108–117, 2006.

[3] J.L. Blanco. A tutorial on se(3) transformation parameterizations and on-manifold optimization, 2010.

[4] José-Luis Blanco, Javier González-Jiménez, and Juan-Antonio Fernández-Madrigal. Sparser relative bundle adjustment (srba): constant-time maintenance and local optimization of arbitrarily large maps. In *IEEE International Conference on Robotics and Automation (ICRA)*, may 2013.

[5] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics and Automation Magazine*, 13(2):99–110, 2006.

[6] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[7] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[8] Christopher Mei, Gabe Sibley, Mark Cummins, Paul Newman, and Ian Reid. Rslam: A system for large-scale mapping in constant-time using stereo. *International journal of computer vision*, 94(2):198–214, 2011.

[9] G. Sibley. Relative bundle adjustment. Technical report, Department of Engineering Science, Oxford University, Tech. Rep, 2009.

[10] G. Sibley, C. Mei, I. Reid, and P. Newman. Adaptive relative bundle adjustment. In *Robotics Science and Systems Conference*, pages 1–8, 2009.

[11] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. The MIT Press, September 2005.

[12] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle adjustment—a modern synthesis. *Vision algorithms: theory and practice*, pages 153–177, 2000.