The effect of luminiferous aether on ICP registration in a constrained tabletop environment

David Rix
University of Cape Town

ABSTRACT

We develop a wrapper for *libpointmatcher*, a C++ registration library, and use this wrapper to expose registration functionality to software developed in C# in the Unity game engine. Users of a tabletop registration system developed with this module and using a Kinect for Xbox One (K4X1) depth sensor report registration errors occurring in the late afternoon. We attempt to replicate this issue in the morning twilight period, but fail to find any connection between error and ambient visible light, or ambient infrared light. We find that ambient infrared light, as perceived by the depth sensor, remains constant even while visible light increases. We recommend a review of the literature on infrared light, and suggest that additional sensor(s) may improve reliability for a tabletop system.

CCS Concepts

• **Computing methodologies ~ Computer vision** • *Computing methodologies ~ Computer vision problems*

Keywords

null result; registration; iterative closest point (ICP); libpointmatcher; Kinect for Xbox One (K4X1); visible light; infrared light

1. INTRODUCTION

One application of computer vision systems is as supporting component in a tangible interfaces [Shaer and Hornecker 2010]. We were tasked with the implementation of such a module, in the context of a tangible system intended for workplace use. In this system, users would place a plastic figurine on a desk, and expect to see its virtual counterpart appear in a 3D scene displayed on a monitor. Multiple figures would be placed on the tabletop, and they should appear in the correct relative positions and orientations: an accurate representation of the real-world tabletop scene.

One way to take an image of the tabletop scene would be to use an time-of-flight sensor such as the Kinect for Xbox One (K4X1). The sensor using a time-of-flight mechanism interrogates the scene with infrared light, recording the depth rather than color of each pixel in view [Butkiewicz 2014]. The resultant image, known as a depthmap, contains a partial 3D surface of all objects in view, known as a 2.5D image. Given a full 3D representation of an model object, known to be in the scene, we can align that representation with the partial scene surface, effectively finding the actual position and orientation of the object in the scene.

This process is known as *model-surface registration* [Berger 2013]. One popular algorithm for this task is *iterative closest point* (ICP) [Chen and Medioni 1991; Besl and McKay 1992; Salvi 2007], which requires an initial estimate to start the alignment process. Result accuracy is unfortunately sensitive to this initial estimate [Salvi 2007]. In the context of a prototype system, we can account for this limitation by placing constraints on the tabletop system. For example, requiring that all figurines are upright, so that our initial estimate does not need to consider rotations on all three axes.

However, this addresses only algorithmic limitations on the registration process. The proposed setting for the system is a workplace desk, and consideration must be made for the physical properties of the system components, including the desk and objects being sensed [Berger et al. 2013], the sensor [Butkiewicz 2014] and ambient light [Butkiewicz 2014; Michelson and Morely 1887].

2. PREVIOUS WORK

libpointmatcher[1] [Pomerleau et al. 2011, 2013] is a C++ ICP implementation used primarily for research in robotic vision systems. (Robotic systems use ICP results to stitch multiple 2.5D surfaces together, thereby navigating and mapping unknown locations in a process known an *simultaneous localization and mapping* (SLAM) [Aulinas et al. 2008].) This library generalizes the body of ICP research by abstracting the ICP process in a sequence of data filters (filters applied to the 3D model data, 2.5D surface data and result data) and processing filters (parameterized algorithms that process and/or augment the data).

Previous works on ICP can be expressed as libpointmatcher filters, compared to one another, and selected based on suitability to a system's particular vision environment [Pomerleau et al. 2011, 2013]. For example, Chen and Medioni [1991] proposed a point-to-plane alignment algorithm, which performs well in artificial environments [Pomerleau et al. 2011], while Besl and McKay [1992] proposed point-to-point. Both are available as processing filters in libpointmatcher.

3. IMPLEMENTATION

3.1 Depthmap

The K4X1 depthmap is non-uniformly scaled, the depth dimension does not match the scaling of the XY plane. A visual assessment was used to scale the Z dimension down by a factor of 4 in order to match the XY scaling.

David Rix. 2015. *The effect of luminiferous aether on ICP registration in a constrained tabletop environment.* Honours report, University of Cape Town. 8 pages. November 2015. http://people.cs.uct.ac.za/~previz2015/registration/

1 https://github.com/ethz-asl/libpointmatcher

The original intention for the K4X1 is as a game controller, and a game presents a player with a representation of themselves, the representation is presented as a mirror image. In other words, when the player raises their right hand, their "reflection" raises its left hand. Because this is the universal use case for the K4X1, the depthmap and other image sources have an inverted X axis. In our case we needed a real-world representation, and the X axis needed to be counter-inverted.

Finally, before conceptualising a depthmap as 2.5D data, we must consider that it represents a perspective projection from the real world onto the camera frame. Thus, the XY coordinates must undergo an inverse perspective projection (sometimes called a camera-to-world projection) so that our final XYZ point is placed in the correct location.

A common formula used for perspective projection is:

$$Bx = Ax * (Bz / Az)$$

where Bx is the result for the screen x-coordinate, Ax the actual x-coordinate, Bz the known distance from the camera to screen and Az the actual distance from the camera. Substitute y for x for a similar formula for y-axis.

Given a depthmap made up of Bx, By, Az values, we want to reconstruct Ax and Ay. For the x-axis this would be:

$$Ax = Bx * (Az / Bz)$$

However, we do not know Bz, the distance from the K4X1 camera to the virtual screen. To determine this value, we placed a number of objects of equal height in a diagonal line in front of the sensor. The line stretched back along the Z dimension, so that the objects were distorted via perspective projection such that they did not appear to have an equal height in a depth map represention. We incrementally adjusted Bz via Newton's method until our inverse perspective projection resulted in objects of equal height.

We calculated Bz to be 400 for our accepted depthmap scale.

Visualising the data in this early in development was challenging, since the effects of scale where not known beforehand. The libpointmatcher team recommend ParaView[2] for 3D data visualization. However, ParaView does not allow on-the-fly manipulation of the data, and does not have an orthogonal projection view. Although we see and arrange objects in perspective, we typically reason with orthogonal logic, for example, knowing that a sequence of objects are the same height, even though they recede into the distance (see above). Therefore we implemented a primitive visualisation tool in our target platform (*Unity 5*[3]) to bootstrap this process. Once we had achieved a world coordinate system, ParaView proved immensely useful for exploring datasets.

3.2 Wrapping for C#

Our target platform was the Unity 5 game engine and the K4X1 sensor. Thus we required a software interface between Unity and the libpointmatcher, a C++ library. This interface was generated via a wrapper template (*Appendix A*) written for the Simplified Wrapper and Interface Generator (SWIG). This template wraps only the high-level API provided by libpointmatcher. Challenges included coordinating the compatibility of libpointmatcher compilation for Microsoft Visual C++ (required for our Windows implementation) and SWIG's internal compiler, while maintaining compatbility with GCC (SWIG's primary compilation target). Not all API methods were eventually exposed some object members remain inaccessible, but the primary registration methods were successfully exposed. Having achieved this we were able to build LibPmSharp, a C# wrapper DLL for libpointmatcher, and develop *_registration_module*, a mesh maintenance and registration module for Unity. This module was successfully used in our original visualization tool, a tangible tabletop system and a sensor-based experiment (below).

3.3 Mesh coordination

The module coordinated registration between meshes that originated in 3D animation (leaving artefacts like invisble rigging, articulated eyeballs and mouth pockets) and then used to 3D print the models to be registered. MeshMixer's "Make solid" tool[4] was used to reduce the number of points in the meshes, and return a mesh with an even distribution of points (eyeballs intially had a very large number of points).

In order to for registration results to be meaningful we reversed the intended meaning of *reference* and *reading* in libpointmatcher: we registered the 2.5D surface against the 3D model in order to answer the question "what transformation would place the model in the correct position and rotation within the surface?" This transformation was used to place a virtual object in the appropriate position in the virtual scene.

Models were reoriented so that their origin point fell at the midpoint of their base, matching the anchor point of their Unity counterparts.

3.4 Tabletop filters

Initial work revealed that wooden tabletops created scoop-shaped reflections of the models in their varnish, which confused registration attempts. We eliminated these by defining a bounding volume that models had to be placed in.

Models also presented a comet-like artefact trailing from their surfaces into the Z-dimension. These outliers can be ignored using the *TrimmedDistOutlierFilter*, which drops points that fall beyond a user-determined outlier factor [Chetverikov 2002].

Augmenting the surface datapoints with invariant values called features [Sharp 2002] meant that the registration process could match pointclouds using these as well as XYZ coordinates. We used normals oriented in the direction of the sensor. (The filters used were *SurfaceNormalDataPointsFilter*, *Observation-DirectionDataPointsFilter* and *OrientNormalsDataPointsFilter* [Pomerleau et al. 2011]). A full example of a data filter, as used for character models on a tabletop, is given in *Appendix B*.

4. AIMS

In a related usability study that makes use of *_registration_module* [Rix 2015], users reported that accuracy was negatively impacted under low light conditions in the late afternoon. This is a surprising suggestion when considering the previously discussed basic mechanism of time-of-flight sensors (see *Introduction*). However, this does not rule out some interaction between time of day and ambient infrared light. We investigate this report by recreating an environment with changing sunlight levels.

2 http://www.paraview.org

3 http://unity3d.com

4 http://www.meshmixer.com

Hypothesis 1. Registration error (difference between reported and expected y-rotation angle) will decrease with increasing ambient visible light.

Hypothesis 2. Registration error will decrease with increasing ambient infrared light.

5. METHODOLOGY

A stage area measuring 14 x 57 cm is marked out on a wooden tabletop. The stage is backed by two panels of matte gray chipboard 21cm high. A K4X1 depth sensor is placed on a separate tabletop, 70.5cm away from the stage area. The sensor is directed towards the stage, facing away from an east-facing window.

Two pixel blocks are marked on the RGB image of the stage, such that each block covered the top half of a chipboard panel (A in Figure X). Ambient intensity of visible light was calculated as the average pixel intensity both blocks.

Two pixel blocks were similarly derived from the infrared image of the stage. Ambient intensity of infrared light was calculated as the average pixel intensity of both blocks.

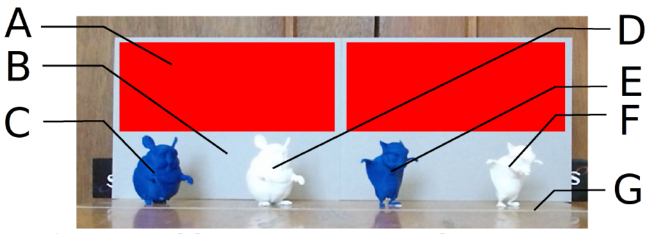

Figure 1. Models on stage area, as seen by RGB camera. A: Pixel block. B: Chipboard. C: Blue concave model. D: White concave model. E: Blue convex model. F: White convex model. G: Stage marker.

Four models were placed upright along the middistance of the stage, facing 45º left (where 0º would be west-facing). The models were 3D prints of character assets from *Big Buck Bunny* [Goedegebure et al. 2008], an animated short. They were a convex-appearing character[5], printed once in blue plastic (C in Figure X) and once in white (D), and a concave-appearing character[6] printed once in blue (E) and once in white (F).

Non-overlapping 3D bounding boxes were derived from the depth image of the stage, such that each model was uniquely associated with a bounding box.

Registration of a mesh was made against the live data found in the bounding box. The actual y-rotation of the registration result was compared to the expected value of 45º, and an error value derived. The error was normalised to lie between 0º and 180º inclusive (in other words, an error of -10º, +10º and +350º would be considered equivalent).

Registration attempts were made on all 4 models between 4:30 AM and 8:45 AM (local time). This time period covered the twilight period that accompanies sunrise, when perceived visibility increases with ambient sunlight.

6. RESULTS

6.1 Effect of light

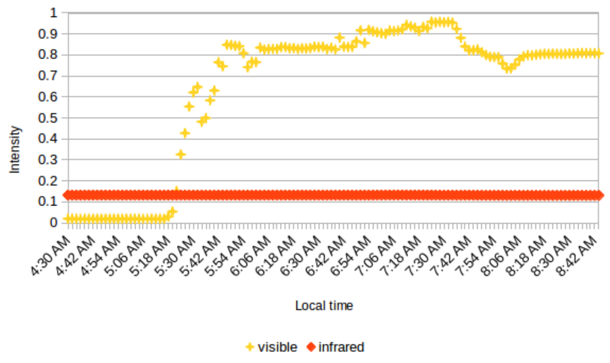

Figure 2. Intensity of visible and infrared light over twilight period.

In figure X we observe that the intensity of visible light (calculated as the average of RGB pixel intensity) increases over the morning twilight period, specifically between 5:18 AM and 6:18 AM, as expected. The increase in non-linear. On the other hand, the intensity of infrared light (calculated as the average of infrared pixel intensity) is linear and near-constant.

In figure X, we note that registration error over the same period is erratic, and does not appear to decrease over the morning twilight period for any model.

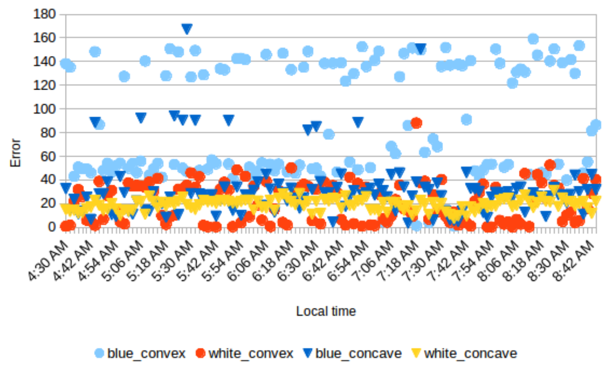

Figure 3. Registration error over twilight period.

These erratic results are seen again when plotting registration error directly against visible light intensity in figure X.

5 Gamera the chinchilla. She appears convex from most orientations.

6 Frankie the flying squirrel. His hunched pose is somewhat concave from this orientation (he would appear convex if his back was to the sensor).

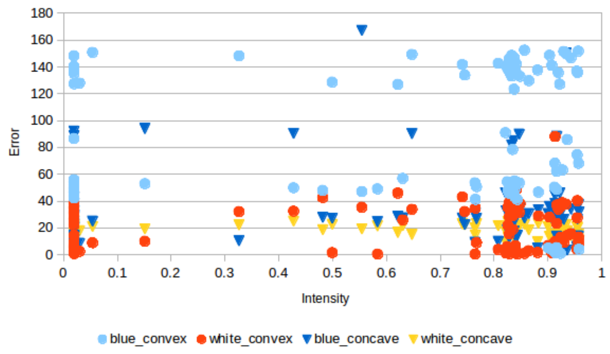

Figure 4. Registration error against visible light intensity.

The implied low correlation is confirmed by calculating Pearson's correlation coefficient between model error and intensity. These results are summarized in Table X. In all cases, no correlation was found.

Table 1. Correlation coefficient between registration error and intensity

Model	Visible	Infrared
blue convex	0.148	-0.118
white convex	0.004	0.045
blue concave	0.034	0.142
white concave	0.150	-0.172

Potential interaction between shape and color

A two-way ANOVA assumes that variance between samples is homogenous. This is confirmed with Levene's test for homogeneity of variance. Brown-Forsythe's variation (centered at the median rather than the mean) is robust to unusual distributions, and was used in this case.

For this sample, Df = 3, F = 90.022 and W = 508. W > F ($p < 0.001$), rejecting the hypothesis that variance is homogenous, and a two-way ANOVA cannot be performed.

7. DISCUSSION

We were not able to confirm a correlation between error and ambient visible light (measured as the average intensity of RGB pixels) or ambient infrared light (measured as the average intensity of grayscale pixels from an infrared camera).

Possible explanations include differences between the setting of this diagnostic investigation and the original setting where the problem occurred. These include:

- **Angle of sunlight.** In the original setting, sunlight shone from behind the models towards the Kinect sensor, whereas in our setting sunlight shone from behind the sensor onto the models. Sunlight from behind models may silhouette them, or create significant local shadows (removing datapoints required for registration). Sunlight shining in the direction of the sensor may cause glare or other artefacts.
- **Light transition pattern.** It could be that the light transition pattern of morning twilight does not match the early-to-late afternoon light transition experienced in the original setting. For example, mid-afternoon light may be significantly brighter and reduce error beyond the limits seen in our study.
- **Distance from sensor.** In our settings models were close to the front limit of the stage, whereas in the original setting users had a deeper stage area, and reported that bringing models closer to the Kinect alleviated reported error somewhat. Increased distance necessarily reduces the 2D area a given model takes up on a depthmap, and therefore reduces the number of datapoints. It could be that this effect is exaggerated in lower light levels in relation to distance from the sensor. Similarly, size of the model may be a factor.

We were surprised to note that ambient infrared light, as measured off a chipboard panel, is near-constant and independent of ambient visible light. This result suggests there may be no correlation between visible and infrared light levels. In this case, no correlations with error were found because there was no change in the relevant light frequency, as recorded by the depth sensor. More technical detail on the infrared mechanism of the K4X1 and the infrared component of sunlight would be required to explore this phenomenon further.

If infrared levels are generally constant, it may be that light levels at any frequency are not the cause of error, nor the erratic registration results in general. It could be that at for this particular combination of model size and distance, the depthmap data is too ambiguous for reliable registration, and the afternoon observation was a coincidence.

Changes to data filters improved registration results early in the development process. A more thorough study may reveal further refinements or alternatives that return more reliable results.

Synthesizing datapoints from one or more sensors (for example, a sensor at the same height but an orthogonal angle) may significantly reduce the ambiguity of pointclouds and return reliable registration results.

8. CONCLUSIONS

In a morning twilight study measuring registration error, we could not conclude that light level was the cause of the error, despite original reports from users that registration error increased in the late afternoon. We did observe that infrared light levels, as perceived by the K4X1 sensor, remained constant for the duration of the study.

A technical review of the infrared component of sunlight and the K4X1's infrared mechanism would be an essential first step in addressing the issue. Disambiguation data from additional sensor(s) would likely result in much more reliable results, even in limiting conditions.

9. FUTURE WORK

Future work should investigate the potential relationship between model shape, size, color, distance from sensor, any interactions and their relationship with registration error. Increased size necessarily increases the number of datapoints captured by the depth sensor. Of particular pertinence to applied computer vision is the question of whether, and to what extent, added information improves registration accuracy.

On a related note, the use of additional sensors to provide multiple views on the same scene could help to resolve otherwise ambiguous cases.

libpointmatcher provides a highly configurable chain of ICP filters. If we consider these to be the genomes of registration implementations, we might be able to take an evolutionary approach to exploring potential filtersets and their fitness for particular computer vision applications. A virtual world might be a convenient testbed for such an exploration, although one would have to consider how closely, for example, results from a virtual sensor match sensor noise and other real-world attributes.

10. ACKNOWLEDGMENTS

I would like to acknowledge François Pomerleau, Stéphane Magnenat and all contributors to the excellent libpointmatcher library. I like to thank François in particular for answering my many many questions.

I would also like to acknowledge Katherine Rix, my partner, for a huge amount of assistance in determining suitable filters to use in a tabletop environment, and for her implementation and optimization of the Haussdorf measure of the fit of two surfaces. This work would not have reached a point ready for research within the available time without her help.

Finally, I would like to acknowledge my supervisor, Dr James Gain who offered invaluable guidance on the mathy bits. I would also like to thank Dr Patrick Marais for offering me the benefit of his experience in more than one conversation.

11. REFERENCES

[1] Josep Aulinas, Yvan R. Petillot, Joaquim Salvi, and Xavier Lladó. 2008. The SLAM problem: a survey. *CCIA*, 363-371.

[2] Kai Berger, Stephan Meister, Rahul Nair, and Daniel Kondermann. 2013. A state of the art report on Kinect sensor setups in computer vision. *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications* (2013), 257-272.

[3] Paul J. Besl and Neil D. McKay. 1992. Method for registration of 3-D shapes. In *IEEE Trans. Pattern Analysis Mach. Intell*. 14, 2 (February 1992), 239-256.

[4] Thomas Butkiewicz. 2014. Low-cost coastal mapping using Kinect v2 time-of-flight cameras. *Oceans* – St. John's (14-19 September 2014), 1-9.

[5] Yung Chen and Gérard Medioni. 1991. Object modeling by registration of multiple range images. *Proceedings of the IEEE International Conference on Robotics and Automation* 3, 9-11 (April 1991), 2724-2729.

[6] Dmitry Chetverikov, Dmitry Svirko, Dmitry Stepanov, and Pavel Krsek. 2002. The trimmed iterative closest point algorithm. *16th International Conference on Pattern Recognition* (2002), 3, 545-548.

[7] Sacha Goedegebure, A. Goralczyk, E. Valenza, N. Vegdahl, W. Reynish, B. V. Lommel, C. Barton, J. Morgenstern, and T. Roosendaal. *Big Buck Bunny* (2008) Retrieved November 1 2015 from https://peach.blender.org

[8] Albert A. Michelson and Edward W. Morley. 1887. On the Relative Motion of the Earth and of the Luminiferous Ether. *Sidereal Messenger*, 6 (1887), 306-310.

[9] François Pomerleau, Stéphane Magnenat, Francis Colas, Ming Liu, and Roland Siegwart. 2011. Tracking a depth camera: Parameter exploration for fast ICP. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, 3824-3829.

[10] François Pomerleau, Francis Colas, Roland Siegwart, and Stéphane Magnenat. Comparing ICP variants on real-world data sets. *Autonomous Robots* 34, 3 (2013), 133-148.

[11] Katherine Rix. 2015. *Viability of a tangible tabletop for industry storyboarding.* Honours report, University of Cape Town. 13 pages. November 2015. Retrieved November 1 2015 from http://people.cs.uct.ac.za/~previz2015/tangible/

[12] Joaquim Salvi, Carles Matabosch, David Fofi, and Josep Forest. 2007. A review of recent range image registration methods with accuracy evaluation. *Image and Vision computing* 25, 5 (2007), 578-596.

[13] Orit Shaer and Eva Hornecker. 2010. Tangible user interfaces: past, present, and future directions. *Foundations and Trends in Human-Computer Interaction* 3, 1-2 (2010), 1-137.

[14] Gregory C. Sharp, Sang W. Lee, and David K. Wehe. ICP registration using invariant features. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 1 (2002), 90-102.

Appendix A: libpm a SWIG template for libpointmatcher

```
%module libpm
%{
/* Includes the headers in the wrapper code (note: order matters) */
#include "pointmatcher\IO.h"
#include "pointmatcher\Timer.h"
#include "pointmatcher\PointMatcherPrivate.h"
#include "pointmatcher\PointMatcher.h"
#include "pointmatcher\Parametrizable.h"
#include "pointmatcher\Registrar.h"
%}

/* Gracefully handle all exceptions */
// from http://swig.org/Doc3.0/SWIGDocumentation.html#Library_stl_exceptions

%include "exception.i"

%exception {
  try {
    $action
  } catch (const std::exception& e) {
    SWIG_exception(SWIG_RuntimeError, e.what());
  }
}

/* Standard wrappers for other exceptions, strings and arrays */

%include "std_except.i"
%include "std_string.i"

%include "carrays.i"
%array_class(float, floatArray);

/* C# wrappers for array parameters */

%include "arrays_csharp.i"
%apply float INPUT[]  {float* array_in}
%apply float OUTPUT[] {float* array_out}
// wraps arrays used in helper functions mapArrayToMatrix and mapMatrixToArray
// (note that array_out seems unreliable and iteration over a floatArray is preferred;
// array_in works as expected)
// see also http://stackoverflow.com/questions/5822529/swig-returning-an-array-of-doubles

/* Prerequisite headers - SWIG definitions */

%rename(process) operator ();
%rename(shiftLeft) operator <<;
%rename(shiftRight) operator >>;
```

```
%rename(isEqual) operator ==;
// name C++-only operators

%ignore loggerMutex;
// SWIG .cxx compilation fails for this object
// (occurs when using underlying Boost library)

%ignore getNameParamsFromYAML;
// MSVC compilation of .cxx fails for this method
// (because the .cxx uses Parametrizable::Parameters instead of
// PointMatcherSupport::Parametrizable::Parameters)

/* Prerequisite headers - include header files */

#define NABO_VERSION "1.0.6"
#define NABO_VERSION_INT 10006
// last tested version (token definition required in headers below)

%include "../pointmatcher/Registrar.h"
%include "../pointmatcher/Parametrizable.h"
// parse the prerequisite header files

/* Primary API - SWIG definitions */

%ignore getFeatureViewByName;
%ignore getFeatureRowViewByName;
%ignore getDescriptorViewByName;
%ignore getDescriptorRowViewByName;
// SWIG .cxx compilation fails for these methods
// (occurs when instantiating Eigen:Block with PointMatcher<float>)

%ignore getLimitNames;
%ignore getConditionVariableNames;
// SWIG .cxx compilation fails for these methods
// (occurs when instantiating PointMatcher<float>::TransformationChecker::
// StringVector)

/* Primary API - header file */

#define WRAPPER_VERSION "0.3.0"
#define WRAPPER_VERSION_INT 00300
// version number for this interface file

%include "../pointmatcher/PointMatcher.h"
// parse the primary API

%template(PM) PointMatcher<float>;
// create a concrete class from the PointMatcher<T> template
```

Appendix B: Tabletop filter

```
readingDataPointsFilters:
  - IdentityDataPointsFilter

referenceDataPointsFilters:
  - SurfaceNormalDataPointsFilter
  - ObservationDirectionDataPointsFilter:
     x: 0
     y: 0
     z: 0
  - OrientNormalsDataPointsFilter:
     towardCenter: 1

matcher:
  KDTreeMatcher:
    knn: 1
    epsilon: 0
    searchType: 1

outlierFilters:
  - TrimmedDistOutlierFilter:
      ratio: 0.9

errorMinimizer:
  PointToPlaneErrorMinimizer

transformationCheckers:
  - CounterTransformationChecker:
      maxIterationCount: 40
  - DifferentialTransformationChecker:
      minDiffRotErr: 0.001
      minDiffTransErr: 0.01
      smoothLength: 3

inspector:
  NullInspector

logger:
  NullLogger
```