SVO: Java READ ME

Introduction

Over the last year or so I have been attempting to translate SVO from C++ into Java. There have been a number of difficulties, but I believe that the foundations are now in place for the system to work, after more debugging. I assume that no or very few classes will need to be created. You may wish to change some structures and implement a slightly different design. Currently I have tried to reflect the original C++ design as closely as possibly, with the added classes necessary due to the differences in languages (typedefs for example).

For anyone who is new to SVO, I would recommend starting with the paper:

http://rpg.ifi.uzh.ch/docs/ICRA14_Forster.pdf

Also, the original C++ documentation is here, although it doesnt have much clear information:

http://uzh-rpg.github.io/rpg_svo/doc/

I am presuming that while you work on SVO-Java, you will predominantly use an IDE, and as such all the tests in this document have used Eclipse. You may also wish to use an IDE for the original SVO, but I have found that the command line works well enough for this so far. Note that I have used a combination of the directory called ‘Workspace’ and the one called ‘Git’, which are both in the home directory.

The long term aim is to have the code being purely Java. However, currently the OpenCV jni is being used. This is quite a substantial library that was deemed necessary for now. At some point a Java based alternative will be required. Also, the Java SVO is significantly slower than the C++. The time has been ignored for now, as the hope is that it will be able to be accelerated when in use with Tornado anyway, but there will be parts that should be streamlined before this.

Install

Installing the original C++ SVO

You will also need to download the original version, which is available with instructions here:

https://github.com/uzh-rpg/rpg_svo/wiki

Choose the No ROS plain Cmake and follow the installation instructions. Note that this can take a while.

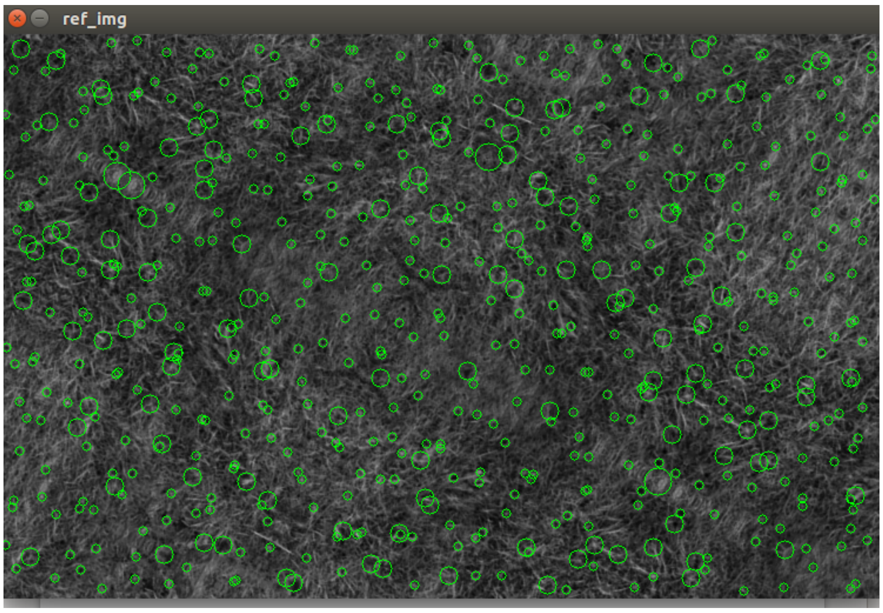
You should end up with a file system such as…

```
~/Documents/SVO_install$ ls
Datasets  fast  opencv-3.4.0  rpg_svo  rpg_vikit  Sophus
```

To run any of the tests, move to rpg_svo/svo/bin and call a test as below:

```
cd /bin && ./test_feature_detection
```

Providing SVO has been installed correctly, you should get an image as follows:

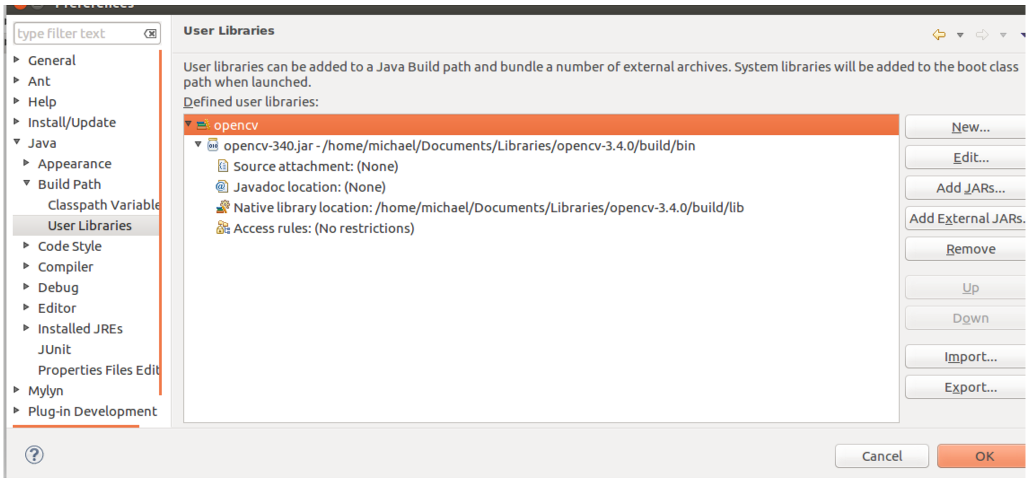


My primary method for editing the original C++ SVO has been to use simple text editor, then recompile and rerun the relevant tests. To do this, follow these steps:

1. Firstly, via the terminal move to the directory in which SVO is saved. This is this location for me: /home/michael/Documents/SVO_install/rpg_svo/svo
2. Use Gedit or another text editor to alter a specific file. E.g: gedit test/test_depth_filter.cpp
3. Recompile with: cd build && cmake .. && make
4. Move back a step and into the test directory to re-run the altered test: cd ../bin && ./test_sparse_img_align

Installing the JAVA SVO

- Available in GitHub repository.
- Requires OpenCv to be installed and set up, then for the library to be added to the project. Follow instructions here: https://docs.opencv.org/2.4/doc/tutorials/introduction/java_eclipse/java_eclipse.html
- Make sure that in eclipse you add the opencv user library. windows>perferences>java>build path> user libraries. Add a new library called open cv, add the external JARs and edit the native library location so it looks as follows:



User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

```
opencv
  opencv-340.jar - /home/michael/Documents/Libraries/opencv-3.4.0/build/bin
    Source attachment: (None)
    Javadoc location: (None)
    Native library location: /home/michael/Documents/Libraries/opencv-3.4.0/build/lib
    Access rules: (No restrictions)
```

- Finally, right click on the project, go to build path>user libraries. Select opencv and finish
- You should now have everything required to run

SVO java progress

The Namespaces are a good place to start understanding the parts of SVO C++, but when it comes to continuing the work in Java, it may be best to focus on the pipeline based tests and their subsequent parts. These come directly from the tests of the pipeline within the C++ SVO, which if you have downloaded the original version should be in “rpg_svo/svo/test”.

On this desktop, they are here:
~/Documents/SVO_install/rpg_svo/svo/test

The java alternatives of these tests are stored in:
/home/michael/git/SVO-J_unfinished/SVO_1/src/testing_package

The SVO original files are all saved in this location on the desktop:
~/Documents/SVO_install/rpg_svo/svo
At this location, you will find the directories for the src, include, build (including make files), the test directory and the bin (for running tests).

Note: there is another package called testing. This is just small tests used during development. It can be ignored.

If you wish to use the already installed SVO-java that I have been editing, use the one saved in
/home/michael/git

There are 6 tests that isolate parts of the pipeline along with a test for the whole system.

1. Test feature alignment.
2. Test feature detection.
3. Test matcher.
4. Test pose optimizer.
5. Test depth filter.
6. Test Sparse Image Align.
7. Test pipeline.

The full pipeline test is yet to be coded, but all the others have been. Below is a table which outlines what degree of completion all of the tests are.
Coded implies that the test has been written in Java but in the fashion of the C++ original. A logic error refers to a test that returns results but the results are not valid in terms of the original. Scope of the code is the amount of features that have been left out from the original version.

Test	Coded	Errors	Scope	Results Validity
Test feature alignment	Yes	Logic	Full	Not Very
Test feature detection	Yes	Logic	Missing features	Not Very

Test matcher	Yes	Logic	Full	Not Very
Test pose optimizer	Yes	RunTime	Full	N/A
Test depth filter	Yes	RunTime	Full	N/A
Test Sparse Image Align	Yes	Logic	Full	Half Perfect, half wrong.
Test pipeline	No	-	-	-

Note: The feature detection is currently not working, although it has been previously. In order to test the validity of other sections of the pipeline agaisnt the original C++ verison, I have used the positions of the features found by the original C++ in place of those in the Java code. For example, in PoseOptimizerTest’s constructor, detect(….) has been commented out in favour of Testing_utilities.read_features(frame_).

Testing

1. Test Feature Alignment

This class and test aims to improve on the current vector estimate (known as cur_px_estimate in the code) for the alignment of reference patches both in one dimension and two. The original documentation states “Subpixel refinement of a reference feature patch with the current image”.

This is a fairly straight forward test in that it contains few classes, and most of those it does contain are extensions of the Jama library. The matrix type classes act as wrappers for the Jama Matrix class, specifying certain vectors. This is to reflect the use of specific sizes of Matricies and Vectors that are represented in the C++ with the use of typedefs. For example, in pose_optimizer.h, the use of “typedef Matrix<double,6,6> Matrix6d;”.

Classes used:

<u>SVO_1310</u>
- Feature_alignment.java

<u>SVO_Vikit</u>
- Timer

<u>Matrix_types</u>
-Matrix2d, Matrix3d
- Vector2d, Vector3d, Vector

<u>Testing package</u>
- test_feature_alignment.

Results

The original version results are as follows

```
michael@cspc022:~/Documents/SVO_install/rpg_svo/svo/bin$ cd ../bin && ./test_feature_align
Loading image '/home/michael/Documents/SVO_install/Datasets/sin2_tex2_h1_v8_d/img/frame_000002_0.png'
1000Xalign 1D took 1.728000ms, error = 0.000033px        (ref i7-W520: 1.982000ms, 0.000033px)
1000Xalign 2D took 1.190000ms, error = 0.015102px        (ref i7-W520: 2.306000ms, 0.015102px)
1000Xalign 2D SSE2 0.737000ms, error = 0.021881px        (ref i7-W520: 0.460000ms, 0.021881px)
```

While the SVO-Java ones are:

```
Loading image frame_000002_0.png
1000Xalign 1D took 7992.0ms, error = 8.430118761434994px        (ref i7-W520: 1.982000ms, 0.000033px)
1000Xalign 2D took 7492.0ms, error = 0.999999999999994px        (ref i7-W520: 2.306000ms, 0.015102px)
```

Note: There is no 3rd row in the Java results as SSE2 is not being used.
The results for Java are significantly slower, and even more significantly different in terms of the error. The second error value is also so close to 1 that it seems likely that it is 1 but with a slight floating point precision issue.

Next Steps

Descover the cause of the logistic errors that are causing such drastic differences between the C++ and the Java. The easiest way I have found in establishing were values diverge is simply to use a lot of print outs of the variable values in both the C++ and Java version. Remember that this will require you to compile the original SVO again.

```
cd ../build && cmake .. && make
cd /bin && ./test_feature_align
```

As a starting point, the values for px_est appear to differ from the C++ values in the for loop around line 65 of test_feature_alignment.java. This is a likely place to find some logic errors.

2. Test Feature Detection

The feature detection class aims to highlight the key features within the images, in order to track their movement in future frames. This class differs slightly from the original, and from what I had originally planned. My initial plan was to create an Edward Rosten style fast detector, or adapt a prexisting java version. However, since OpenCV has a feature detector, this has been employed instead. Also, it is worth noting that the image pyramid was not working. As this is not necessary it has for now been removed. Subsequently, only the first layer is taken into account.

Classes used:

<u>SVO_1310</u>
- Frame
- Feature
- Frame_Utils
- AbstractDetector

- FastDetector
- Config
- ImgPyr
- Corner
- FeatureType

<u>Vikit</u>
- math_utils
- Vision
- AbstractCamera
- ATANCamera
- Timer

<u>Matrix_types</u>
- Vector3d, Vector2d, Vector
- Matrix6d

<u>Testing package</u>
- ImShow
- test_feature_detection

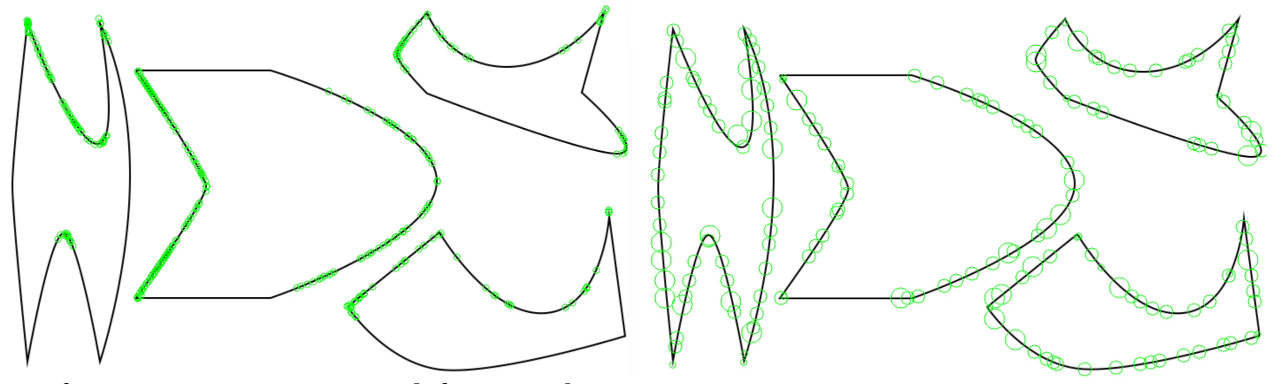
Results

The grass dataset images provide a difficult example to check by eye. Instead, there is a simple black and white image with some random shapes saved within the dataset directory called shapes.png. By using this shape with the OpenCV detector, it is evident that it does find the significant features. However, when applied to the grass image, all the features are located within the left hand quarter of the image.

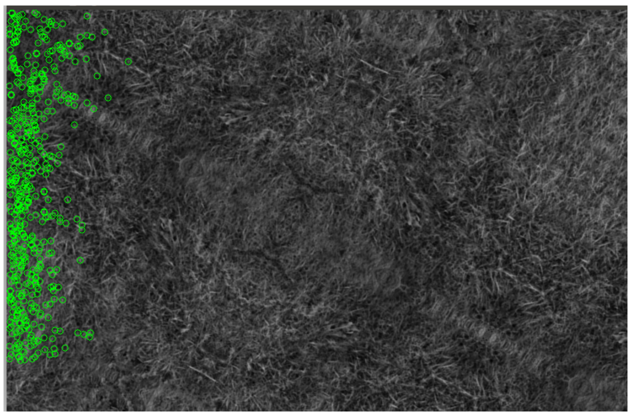
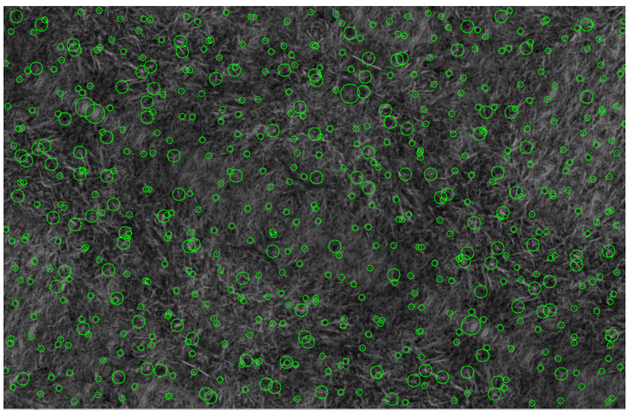
As you can see in the below image the c++ version offers much greater coverage, with features being identified on different pyramid levels (represented by the size of the circle). The java version from OpenCV does find feature points. It is likely that with some tweaking you would be able to get a more uniform spread of features.

Shapes is available here:

Using shapes.png. Java on the left and C++ on the right.



Using frame_000002_0.png. Java left C++ right.

Next Steps
The next steps for the feature detection will be to establish why all the features are located on the left and then to introduce the further pyramid layers.

If you alter the field *desired_number_of_corners_per_img* in Config.java from 500 (a somewhat arbitrary value taken from the original SVO) to different amounts, you can see that the image is finding corners/features in a left to right manner, travelling down the image. Also, you can see that the maximum value is 570, at which point the features found go the whole way down the image, but still stay on the left. I suspect the issues do not arise from the width parameters, as the detector fills the image when the shapes.png image is used, or in displaying the features, as they are again displayed correctly across the page for shapes.png.

As a side note, perhaps the code stops searching through a line if a feature is found. If that is the case, most of the features would be on the left hand side if the parameters for a feature to be a point were lower than that of the C++ code as is found in the image before.

3. Test Matcher

The Matcher class attempts to find a feature in more than one image. So, if a point in the centre of the image of frame 1 is now in the top left of frame 2, then the camera must have moved in the direction of the bottom right. This class matches the feature found in those frames. Test_matcher assesses how many feature points converge over two images, frame_00002 and frame_00006.

Note: The java version currently only checks the top 6 lines (after the 4 line border). This is purely because of the time constraints with running the entire thing. If the whole image is searched then it takes an inpractical amount of time (upwards of an hour). To alter the number of lines in the image that are assessed, change this line:

```
            for(int y = 4; y<10; y++)//for(int y = 4; y<depthMap.rows()-4; y++)
```

Classes Used

SVO_1310

-Feature_Alignment
-Frame
-Matcher
-Feature
-Frame_Utils
-Config
-Warp
-ImgPyr
-Matcher_Options
-Depth
-Feature_type

Vikit
-Pinhole_Camera
-math_utils
-Vision
-ZMSSD
-AbstractCamera
-BlenderUtils

Matrix_Types
-JamaUtils
-Matrix2d, Matrix3d, Matrix6d
-Vector, Vector2d, Vector3d

Testing Package
-Test_Matcher

Sophus
-Quaternion
-So2, So3

Results

The test_matcher class shows that drastically less converged points occur in the Java, even taking into account that in the following tests results, only about 1/5th of the image has been checked. The reduction in the amount of image being matched is due to the time it takes to complete. For only 100 lines, as below, it takes over 20 minutes to execute. To run the whole test, change this line:

```
for(int y = 4; y<100; y++)    //for(int y = 4; y<depthMap.rows()-4; y++)
```

C++

```
Loading image '/home/michael/Documents/SVO_install/Datasets/sin2_tex2_h1_v8_d/img/frame_000002_0.png'
Loading image '/home/michael/Documents/SVO_install/Datasets/sin2_tex2_h1_v8_d/img/frame_000006_0.png'
n converged:      61957 mm (ref: 216114)
mean error:       0.652882 mm (ref: 0.410084)
50-percentile:    0.116889 mm (ref: 0.083203)
80-percentile:    0.218163 mm (ref: 0.161824)
95-percentile:    0.337842 mm (ref: 0.263539)
```

Java (for 100 lines rather than the full 480)

```
Loading image img/frame_000002_0.png
Loading image img/frame_000006_0.png
n converged:      761 mm (ref: 216114)
mean error:       83.00908448639937 mm (ref: 0.410084)
50-percentile:    82.96496272087097 mm (ref:0.083203
80-percentile:    83.20156931877136 mm (ref: 0.161824)
95-percentile:    83.35007429122925 mm (ref: 0.263539)
```

It is evident that the Java is both less accurate, in terms of the percentile values for errors, and less successful in finding points that converge. Originally, the errors were around 300 in the Java, but have been improved and should be able to be more so.

Next Steps

The excessive amount of time taken for each run seems to arise from each loop in the x=4; x<depthMap.cols-4 taking about 30ms. (Note that within the code there is a commented out Timer that I have used for testing). As this equates to 30ms * 472 * 744 /1000 = 10,535.04 seconds. This test obviously contains some logic errors and finding those so that the number of converged points is more similar to the C++, as well as decreasing the erros is paramount.

After this, this test may be a good place to start with any work on Tornado. Its exceptionaly long run time, but ease with which to shorten it may provide a good place to start any accelleration.

4. Test Sparse Image Align

Sparse Image Align aims to optimize the estimated posisiton of the frame by reducing the photometric error of the reference patches. This test has fairly good results for the |t| value but still has some issues in producing translation errors accurately, with respect to the original version. I believe this is the part of the pipeline that will have exactly the same results as the C++ with the least amount of debugging.

Classes Used

<u>SVO_1310</u>
- Frame
- Point
- FastDetector
- Feature
- SparseImgAlign
- AbstractDetector
- Frame_Utils
- Config
- ImgPyr
- FeatureType
- PointType

<u>SVO_Vikit</u>
- Pinhole_Camera
- math_utils
- Vision

- NLLSSolver
- AbstractCamera
- ImageNameAndPose
- Timer
- Blender_Utils
- FileReader
- Method

<u>Testing_Package</u>
- Testing_utilities
- test_sparse_img_align
- SparseImgAlignTest

<u>Matrix_Types</u>
- JamaUtils
- Vector, Vector2d, Vector3d, Vector6d
- Matrix3d, Matrix6d

<u>Sophus</u>
- Quaternion
- So3, Se3

Results

The primary point to make with the Sparse Image Align tests results is that the value of |t| is the same between both the Java and C++ versions (The C++ displays only to 6 decimal places). Also worth noting is that the translation errors are different between the two. The translation errors in the Java is currently the same value as |t|, so there is an error here. The time is significantly longer, with each iteration taking about 27 seconds.

Java Results

```
RUN EXPERIMENT: read 186 dataset entries.
Added 570 3d pts to the reference frame.
i=0 loop finished

[ 1 ]   time = 27365.0ms        |t| = 0.16001828020573147    translation error = 0.1600182802057315
[ 2 ]   time = 27066.0ms        |t| = 0.32003681038280607    translation error = 0.32003681038280607
[ 3 ]   time = 26791.0ms        |t| = 0.4800559654873587     translation error = 0.48005596548735874
[ 4 ]   time = 27158.0ms        |t| = 0.6400770578610051     translation error = 0.6400770578610052
[ 5 ]   time = 27117.0ms        |t| = 0.8000318618655138     translation error = 0.8000318618655139
[ 6 ]   time = 28559.0ms        |t| = 0.9600643988816591     translation error = 0.9600643988816592
[ 7 ]   time = 29317.0ms        |t| = 1.1199672941653258     translation error = 1.119967294165326
[ 8 ]   time = 28300.0ms        |t| = 1.2799595384229934     translation error = 1.2799595384229936
[ 9 ]   time = 27808.0ms        |t| = 1.4399798401366608     translation error = 1.439979840136661
[ 10 ]  time = 29206.0ms        |t| = 1.5998258811508217     translation error = 1.5998258811508221
[ 11 ]  time = 27548.0ms        |t| = 1.7597240749617546     translation error = 1.759724074961755
[ 12 ]  time = 27154.0ms        |t| = 1.9195528255299472     translation error = 1.9195528255299477
[ 13 ]  time = 27396.0ms        |t| = 2.079328506994507      translation error = 2.079328506994508
[ 14 ]  time = 27351.0ms        |t| = 2.2390100848366012     translation error = 2.2390100848366017
[ 15 ]  time = 27432.0ms        |t| = 2.3984848404774213     translation error = 2.3984848404774217
[ 16 ]  time = 27113.0ms        |t| = 2.557859280726759      translation error = 2.5578592807267593
[ 17 ]  time = 26922.0ms        |t| = 2.7171044183100523     translation error = 2.717104418310053
[ 18 ]  time = 27501.0ms        |t| = 2.87599006430829       translation error = 2.8759900643082905
[ 19 ]  time = 27282.0ms        |t| = 3.0347091936460755     translation error = 3.034709193646076
[ 20 ]  time = 27299.0ms        |t| = 3.1930472373580723     translation error = 3.193047237358073
[ 21 ]  time = 27291.0ms        |t| = 3.351005945980999      translation error = 3.3510059459809995
[ 22 ]  time = 27610.0ms        |t| = 3.508454994723463      translation error = 3.508454994723464
[ 23 ]  time = 27616.0ms        |t| = 3.6652738628921067     translation error = 3.6652738628921075
[ 24 ]  time = 27243.0ms        |t| = 3.8214931793737397     translation error = 3.8214931793737406
[ 25 ]  time = 27406.0ms        |t| = 3.976951920755392      translation error = 3.976951920755393
[ 26 ]  time = 27327.0ms        |t| = 4.131486295511584      translation error = 4.131486295511584
[ 27 ]  time = 27210.0ms        |t| = 4.284943067299731      translation error = 4.284943067299731
[ 28 ]  time = 26730.0ms        |t| = 4.437201372486946      translation error = 4.437201372486946
[ 29 ]  time = 26856.0ms        |t| = 4.588171837453344      translation error = 4.588171837453346
End of test.

C++ Results
michael@cspc022:~/Documents/SVO_install/rpg_svo/svo/build$ cd ../bin &&
./test_sparse_img_align
RUN EXPERIMENT: read 187 dataset entries.
Added 570 3d pts to the reference frame.
[  1] time = 4.778000 ms      |t| = 0.160018    translation error = 0.000082
[  2] time = 5.037000 ms      |t| = 0.320037    translation error = 0.000381
[  3] time = 5.341000 ms      |t| = 0.480056    translation error = 0.000124
[  4] time = 5.637000 ms      |t| = 0.640077    translation error = 0.000161
[  5] time = 5.499000 ms      |t| = 0.800032    translation error = 0.000324
[  6] time = 5.140000 ms      |t| = 0.960064    translation error = 0.000321
[  7] time = 4.283000 ms      |t| = 1.119967    translation error = 0.000267
[  8] time = 4.740000 ms      |t| = 1.279960    translation error = 0.001103
[  9] time = 4.249000 ms      |t| = 1.439980    translation error = 0.000854
[ 10] time = 3.329000 ms      |t| = 1.599826    translation error = 0.001692
[ 11] time = 5.634000 ms      |t| = 1.759724    translation error = 0.001379
[ 12] time = 5.732000 ms      |t| = 1.919553    translation error = 0.001529
[ 13] time = 2.845000 ms      |t| = 2.079329    translation error = 0.064925
[ 14] time = 3.926000 ms      |t| = 2.239010    translation error = 0.195165
[ 15] time = 4.374000 ms      |t| = 2.398485    translation error = 0.311108
[ 16] time = 2.297000 ms      |t| = 2.557859    translation error = 0.202392
[ 17] time = 3.406000 ms      |t| = 2.717104    translation error = 0.172640
[ 18] time = 6.926000 ms      |t| = 2.875990    translation error = 0.253802
[ 19] time = 2.383000 ms      |t| = 3.034709    translation error = 0.405134
[ 20] time = 2.812000 ms      |t| = 3.193047    translation error = 0.552380
[ 21] time = 1.840000 ms      |t| = 3.351006    translation error = 0.638793
[ 22] time = 12.342000 ms     |t| = 3.508455    translation error = 2.498009
[ 23] time = 5.288000 ms      |t| = 3.665274    translation error = 3.042834
[ 24] time = 4.508000 ms      |t| = 3.821493    translation error = 3.027737
[ 25] time = 3.226000 ms      |t| = 3.976952    translation error = 2.937300
[ 26] time = 5.966000 ms      |t| = 4.131486    translation error = 2.967488
[ 27] time = 10.547000 ms     |t| = 4.284943    translation error = 3.065960
[ 28] time = 4.471000 ms      |t| = 4.437201    translation error = 3.245093
[ 29] time = 6.382000 ms      |t| = 4.588172    translation error = 3.403362
```

Next Steps

The time for each iteration of the for loop at line 91 in SparseImgAlignTest is significantly longer than the c++, at just under 30 seconds rather than a few milliseconds.

5. Test Pose Optimizer

According to the documentation, the pose optimizer is intended to do “Motion-only bundle adjustment, minimizing the reprojection error of a single frame”.

Classes Used

SV0_1310
-Frame
-Point
-PoseOptimizer
-Feature
-Frame_Utils
-Config
-ImgPyr
-FeatureType
-PointType

Vikit
-Pinhole_Camera
-math_Utils
-Vision
-AbstractCamera
-Blender_Utils
-TukeyWeightFunction
-MADScaleEstimator

testing_Package
-Testing_Utilities
-PoseOptimizerTest
-test_pose_optimizer

Matrix_Types
-JamaUtils
-Vector, Vector2d, Vector3d, Vector6d
- Matrix3d, Matrix6d

Sophus
- Quaternion
- So3, Se3

Results

This test currently has run-time errors. However, when I was working on this a few months previously it did run to completion, so it is likely that the runtime errors are minimal and fairly easy to fix, before moving on to dealing with the validity of the results. When this test was working previously the norm(dT) values were very similar to that of the original SVO, whilst the new_chi2 values varied a lot.

SVO C++ results:

```
Loading image '/home/michael/Documents/SVO_install/Datasets/sin2_tex2_h1_v8_d/img/frame_000002_0.png'
Added 570 features to frame.
Add 1.000000 px noise to each observation
it 0     Success          new_chi2 = 15.3143       norm(dT) = 0.246568
it 1     Success          new_chi2 = 0.165028      norm(dT) = 0.0259165
it 2     Success     new_chi2 = 0.0095597     norm(dT) = 0.000272372
it 3     Success     new_chi2 = 0.00953925    norm(dT) = 1.07694e-07
it 4     FAILURE     new_chi2 = 0.00953925
n deleted obs = 59   scale = 63.8296   error init = 43.1281    error end = 1.00252
```

Having quickly inspected the error, it seems to arise from PoseOptimizer ~line 114:

```
                    Vector6d dT = new
Vector6d(A.chol().solve(b.transpose()).transpose().getArray());
```

This causes a ‘Matrix is not symmetric positive definite’ to be thrown by the Library Jama when attempting to solve the cholesky decomposition with chol().solve(…). This error is being thrown due to Matrix6d A, Vector6d b & Vector2d e having null or NaN values.

ToDo

Before working on the logic errors that lead to incorrect values, the runtime errors mentioned above will need to be solved.

6. Test Depth Filter

A Bayesian depth filter is applied to feature points to ensure fewer outliers are inserted into the map. A point has to ‘converge’ before it is added to the map.

There is an issue with both the C++ SVO and the Java SVO. The original SVO depth filter test is not running at the moment, although I believe it was until recently. It throws an error which is explained here:

http://eigen.tuxfamily.org/dox-devel/group__TopicUnalignedArrayAssert.html

Classes Used

SVO_1310
Feature_alignment
Frame
Matcher
DepthFilter
Feature
Frame_Utils
AbstractDetector
FastDetector
Config
Seed
Warp
ImgPyr
NormalDistribution
Matcher_Options
Corner
Depth

DepthFilter_Options
FeatureType

<u>Vikit</u>
Pinhole_Camera
math_Utils
ZMSSD
ImageNameAndPose
Timer
Vision
Blender_Utils
AbstractCamera
FileReader

<u>testing_package</u>
DepthFilterTest
test_Depth_Filter

<u>matrix_types</u>
JamaUtils
Matrix3d, Matrix6d, Matrix2d
Vector, Vector2d, Vector3d

<u>Sophus</u>
Quaternion
So3, Se3

Results

The Java has some strange runtime errors that need addressing. Each execution of the Depth Filter test results in a different outcome. The number of images that are read varies. Usually the program finds a Singular Matrix around frame 4-7, but occasionally this happens much later. I am currently unaware as to why the results are not consistent here.