Compiling PCL on Windows

When we wanted to setup PCL for Windows together with other libraries, we noticed that this is not a trivial task. There is a list of official precompiled binaries, limited to Visual Studio 2010 and PCL 1.6.0 at http://pointclouds.org/downloads/windows.html. We wanted to port a software from Linux to Windows requiring a version, newer than PCL 1.6.0. Moreover the ported software has dependencies to other frameworks. A compiler/compiler version needed to be found, which can generate compatible binaries for all dependencies. We found, that MSVC 12.0 was capable to do this for our specific case. For a PCL-only-setup, newer versions of MSVC should work fine.  
Prior to Visual Studio we tried to use gcc with mingw and msys which led to major problems (see ‘what not to do’ section).

Prequesities:

This tutorial was tested on Windows 10 Pro x64.

* Git Bash from https://git-for-windows.github.io/ —git 2.8.1
* CMake with CMake-gui from https://cmake.org/download/ —cmake 3.5.2

Overview/Hints

If you want to understand what exactly happens during installation, this section is for you. It may also help if you run into trouble which are not covered elsewhere in this tutorial. However, if you wich to only quickly install the PCL, you may skip this section to the step-by-step explanation of how to create a working library.

* We used Visual Studio 2013, 64bit (and NMake) for each dependency (MSVC 12.0 is equal to Visual Studio 2013).
  + Choose VSMC 12 when CMake asks about this for each dependency.
  + Choose 64bit inside Visual Studio for each dependency.
* Linking against system library (libc.lib or msvc.lib)
  + Static vs. Dynamic: This point is about linking against system libraries (“C/C++ Runtime Library”), not about the type of library we generate/download (Not .lib or .dll)! **Link against system library dynamically.** The reason for this is, that libraries with this configuration are commonly used and broadly available to download. Moreover, libraries statically linked against the system library may conflict with other libraries linking statically against other versions of the system library.
  + For boost the correct libs are named **without** “-s”. Note, that there is no standard naming convention.
  + Flann use the “\_s” postfix to indicate their correct .lib-file.
  + The correct compileroption for msvc is “/MD” (dynamic liniking, dll) and not “/MT”.
  + Multi-threading: Link against multi-threaded system library. There is a pitfall for boost libaries: While “/MT” as a compileroption means “static-linking”, for boost “-mt” in the name of the library means “multi-threaded”. **We need “-mt”-libraries from boost.**
* Debug vs. Release: For Windows this must be correct. CMake lets you define both for each library, a debug and release version file path. If this is done wrong, it leads to undefines symbols errors. For boost Debug-libs have a “-gd” in their name. Most other libraries append an additional “d” at the end of the libs name. E.g. libyaml-cppmd.lib is the release version, libyaml-cppmdd.lib is debug. **Note the second d**.
  + For flann you generate release and debug by hitting compile in Visual Studio twice and changing the configuration in the dropdwon menu.
  + For other dependecies the CMake may not be compatible to generate multi-configuration project files. In this case CMake must be executed twice and CMAKE\_BUILD\_TYPE must be set to “Release” or “Debug”.
* Some libraries require C++11 features to be compatible with upns4d+ (this is PCL, which will generate an excpetion in boost, if PCL is not c++11 and is used in a c++11 project). Enabling C++11 for libraries is done with CMAKE\_CXX\_FLAGS and CMAKE\_CXX\_FLAGS\_DEBUG, having added **-std=c++11**.
* Usually we can generate dynamically linked libraries for most dependencies. Sometimes (e.g. Protobuf) “\_\_dllexport” would be needed in the code, to make functions public available. Since we do not want to alter the code, we instead generate a static library. Unaffected by that, it must still **link dynamically against system libraries**.

With this is mind, we can start the installation step-by-step.

Getting the Sourcecode

Get pcl source code with

git clone https://github.com/PointCloudLibrary/pcl.git

Then change directory

cd pcl

and checkout the version you want to use, e.g.

git checkout pcl-1.7.2

Statisfing Dependencies

**Boost**

* Download Boost 1.56 Windows binaries from https://sourceforge.net/projects/boost/files/boost-binaries/1.56.0/boost\_1\_56\_0-msvc-12.0-64.exe

We chose exactly 1.56 for our specific project and PCL version (which, as explained above, has to embed into an environment of other libraries).  
For older boost versions “boost-serialization” is not compatible with windows.  
Newer boost versions on the other hand are not compatible with Visual Studio 2010. This is not an issue, if all other dependencies of your project are compatible with a newer version of VS.

**Flann**

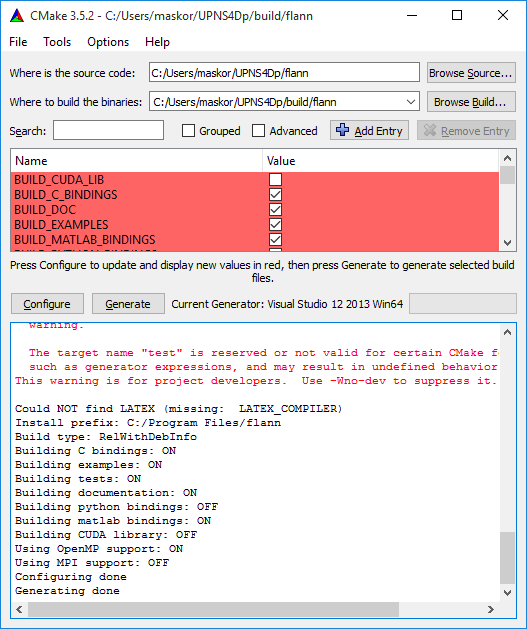
Download flann source code

git clone https://github.com/mariusmuja/flann.git

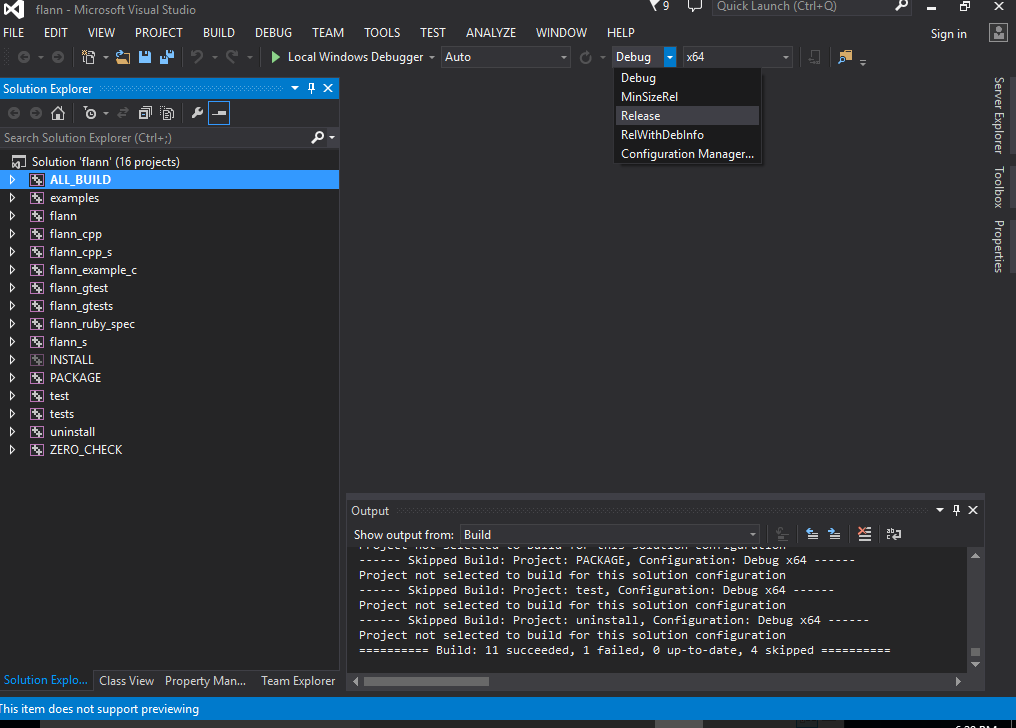
flann must be compiled manually.  
Creating libs for flann manually is relatively straight forward using CMake-gui, choosing Visual Studio 2013 64 Bit.

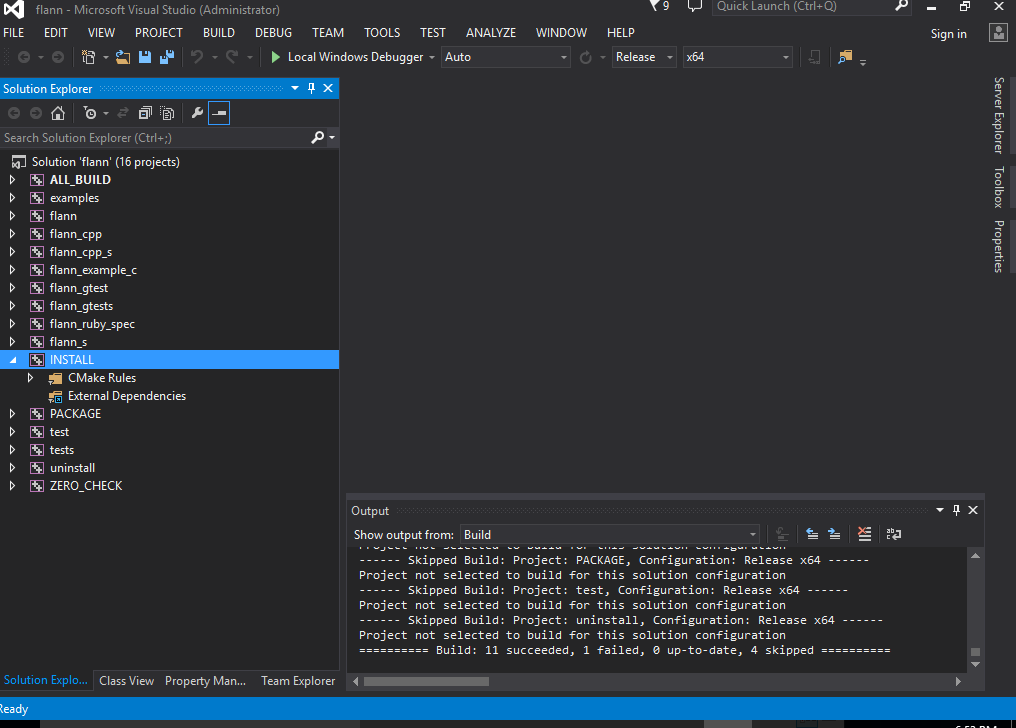
In CMake-gui:

1. Select source code location of flann (root of cloned git repo)
2. change build directory from “C:\Program Files\…” to a directory where you have full access rights e.g userdir
3. Click on configure
4. Select your compiler and System Architecture \*Cmake will start to configure the project. Ignore the warnings about missing hdf5 libs and Cmake policy. If Cmake prints ‘Configuring done’ the configuration was successful.
5. In order to create solution files for flann click generate.
6. Now start Visual studio and open the solution file from your specified Flann build folder.



In Visual Studio:

-Select BUILD\_ALL project  
build Release x64 and Debug x64 Libraries. This can take some time, grab a Coffee.   


Restart Visual studio with Administrator Privileges and build the INSTALL project  


If this does not work directly (which was the case in MSVC 11) you can add add compileroption “/bigobj” to all projects. In Visual Studio this is done by clicking every project flann, flann\_cpp, flann\_s, and open the property pages. Note that you can multiselect projects. Note that the configuration is dropdown on the top of the property dialog is stet to “All Configurations”.

It is also important to recognize the options for “static” or “dynamic” linking of the “C/C++ Runtime Libray”. This is **not** about creating a dynamically linked (dll) or statically linked (lib). **All Libraries must coincide** for this option. Otherwise hard to find linking errors will occur. Usually prebuilt libraries found on the internet use dynamic linking against the runtime. On the other handy side, users will have to install the Visual Studio Runtime when they want to use the final software.

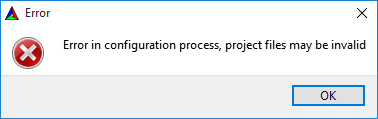
This option depends on all other libraries you use and will very likely be dynamic linking against C/C++ Runtime Library.

**Eigen**

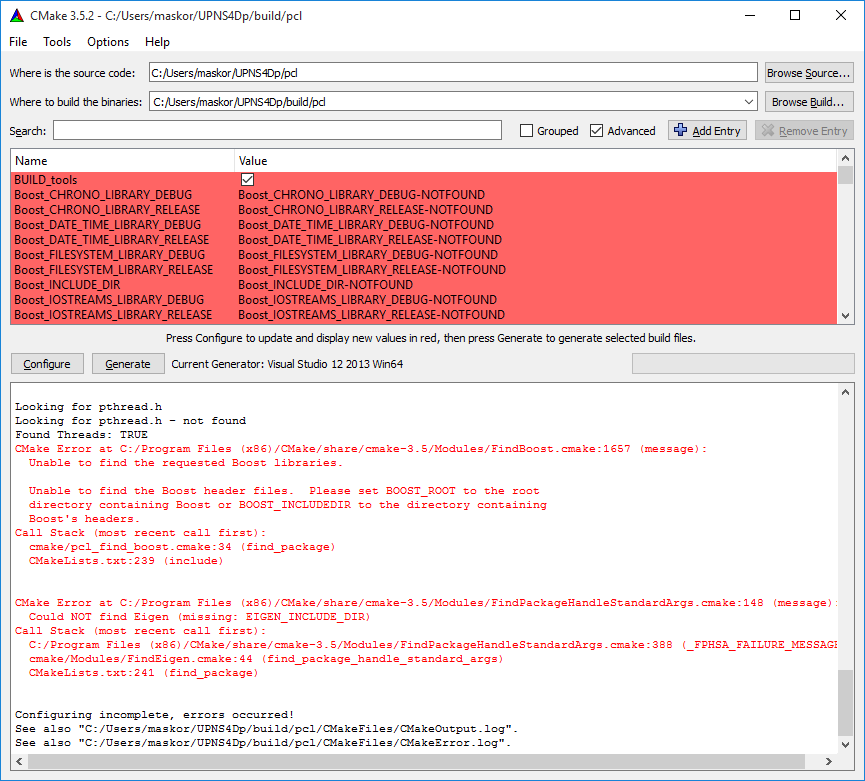
Download eigen from http://eigen.tuxfamily.org/ —Tested with eigen 3.2.8  
Eigen is a header only library and does not need to be compiled/configured at all.

Configure PCL

* Open CMake-gui and delete cache. File->Delete Cache
* select Advanced
* As Source: select root folder of pcl
* Specify build folder for pcl
* click configure and select Visual Studio 12 2013 Win64



The Configuration will probably fail. Not all dependencies are met.



define path to missing Libs.

**BOOST:**

Choose all lib64-msvc-12.0/boost\_[X]-vc12-mt[-gd]-1\_56.lib libraries. e.g

**BOOST\_CHRONO\_LIBRARY\_DEBUG**: \lib64-msvc-12.0\boost\_chrono-vc120-mt-gd-1\_56.lib  
**BOOST\_CHRONO\_LIBRARY\_RELEASE**: \lib64-msvc-12.0\boost\_chrono-vc120-mt-1\_56.lib

Be aware that you use the right libs for DEBUG (mt-gd) and RELEASE (mt).

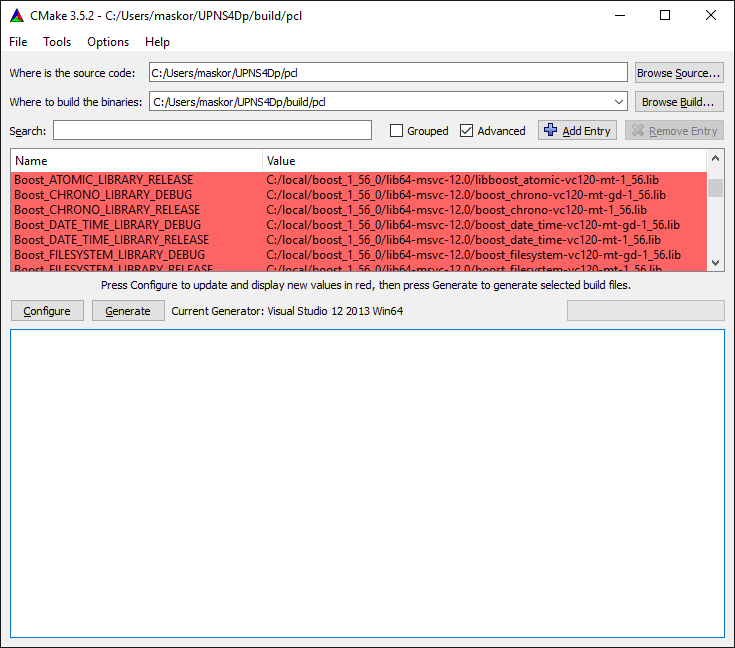
BOOST\_MPI\_LIBRARY is not needed.

**EIGEN:**

**EIGEN\_INCLUDE\_DIR**: eigen-eigen-b30b87236a1b (hash may differ for newer versions)

**FLANN:**

**FLAN\_INCLUDE\_DIR**: \src\cpp  
**FLANN\_LIBRARY\_DEBUG**: \lib\Debug\flann\_cpp\_s.lib  
**FLANN\_LIBRARY\_RELEASE**: \lib\Release\flann\_cpp\_s.lib



Click Configure and Generate.

Notes:  
If you want to use PCL 1.7.2 in a Project with c++11 enabled, PCL must also be compiled with c++11. (This might be not needed in newer versions of PCL 1.8)

* Show Advanced variables in cmake-gui and set CMAKE\_CXX\_FLAGS\_DEBUG to **-std=c++11**

**COMPILE PCL**

Open Visual Studio and the newly generated PCL.sln Solution.  
Build **“ALL\_BUILD”** for debug and release. COFFEE TIME. After that restart Visual Studio as Administrator and Build **“INSTALL”**.

Copy Boost dll’s from your boost binaries to C:\Program Files\PCL\bin

**Finally done.**

What not to do

* Do not use mingw or msys. While gcc is a great compiler, its windows ports are not advanced enough to compile PCL yet and will generate out of memory exceptions (even on 16GB RAM machines, also with —large-address-aware and -O0). This is an gcc issue: “cc1plus.exe: out of memory allocating 4567004 bytes” http://www.pcl-users.org/compile-pcl-on-windows-error-td4036955.html
* Do not use an older boost library: boost-serialization will not work
* You might use newer boost versions if you intent to be compatible with VS 2010.
* When using older version of MSVC, c++11 won’t work. There are even more problems which are hard to overcome:
  + flann: After cmake-gui.exe the generated MSVC 11 project wont work directly. Go to “Project Properties” of all projects flann, flann\_cpp, flann\_s, … add compileroption “/bigobj” under “C/C++ -> Command Line”.
* We had trouble compiling boost and using this custom version.
* Do not compile mingw packages or qt creator on your own.
* A seg fault during runtime in “boost::math::lanczos::lanczos\_initializer” indicates a problem with PCL not built with c++11 flag, while the project uses c++11.