26.4.2018

Stanislav Uschakow

Matlab2Android

A quick tutorial

Inhalt

[1. General Information 2](#_Toc512536133)

[2. Quickstart 2](#_Toc512536134)

[3. Matlab part 3](#_Toc512536135)

[4. Android Part 4](#_Toc512536136)

[5. JNI Conversion 5](#_Toc512536137)

# General Information

This document describes how to convert a matlab script to a working c++ sourcecode to be used in android NDK environment with JNI. It will show all steps necessary using a simple filter as an example. It is assumed the reader has a working matlab version with the Matlab Coder toolbox and Android Studio installed. The key part for android to work with c++ code is the NDK (Native Development Kit) which will generate a native static lib (\*.so) file which than can be used by the JNI. JNI stands for Java Native Interface. It allows the Java to access native functions of the OS such as DLL in windows or shared library (\*.so) when running Linux. This, obviously will break the portability of the Java application unless the libraries are available on all target platforms. Often JNI is used to access code written in C++ which is used here [[JNI wiki](https://de.wikipedia.org/wiki/Java_Native_Interface)]. The NDK uses CMake as its primary environment building tool. [MathWorks webinar on Matlab to iPhone and Android Made Easy is the main source for this document.](https://de.mathworks.com/videos/matlab-to-iphone-and-android-made-easy-107779.html)

# Quickstart

First, I want to briefly describe all steps needed. Later, I will discuss them in detail with code examples.

1. Create a matlab function which should be translated to c++.
2. (Optionally) Create a test environment with sample data for the coder to autogenerate the datatypes.
3. Run matlab coder
   1. Select the entry point (the function defined in 1)
   2. Define datatypes for the output code.
   3. Generate c++ sourcecode
4. Start a new Android project. Don’t forget to check ‘C++’ when generating the project.
5. Copy the generated sourcecode to the Android Studio project location where the autogenerated .cpp file is located.

Note: When compiling on a fresh setup machine Android Studio will ask you to install all tools needed like the NDK and CMake so no manual steps are required for this process.

1. Edit the CMakeLists.txt for NDK to know how to compile the files.
2. Call JNI functions from Java utilizing your shiny matlab generated c++ library.

# Matlab part

Now let’s discuss the steps needs to be done in Matlab in more detail:

1. First, we need a simple example in matlab to be converted to c++. I am using a simple smoothing filter. It is necessary that we have a function instead of a script since later it will serve as the entry point. It is possible to select several functions as well. The code below shows a simple function getting an input vector to be processed by the filter and returning the output of the filter:

core.m:

function output = core (input)

b = 1/3 \* [1 1 1];

output = filter(b, [1], input);

end

1. For the autodetecting of the datatypes we need a testbench using the core function.

coreTestBench.m:  
input = [0 0 1 0 0 0 0 0 0 0 0];

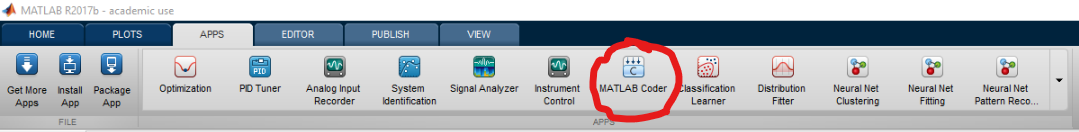
output = core(input);

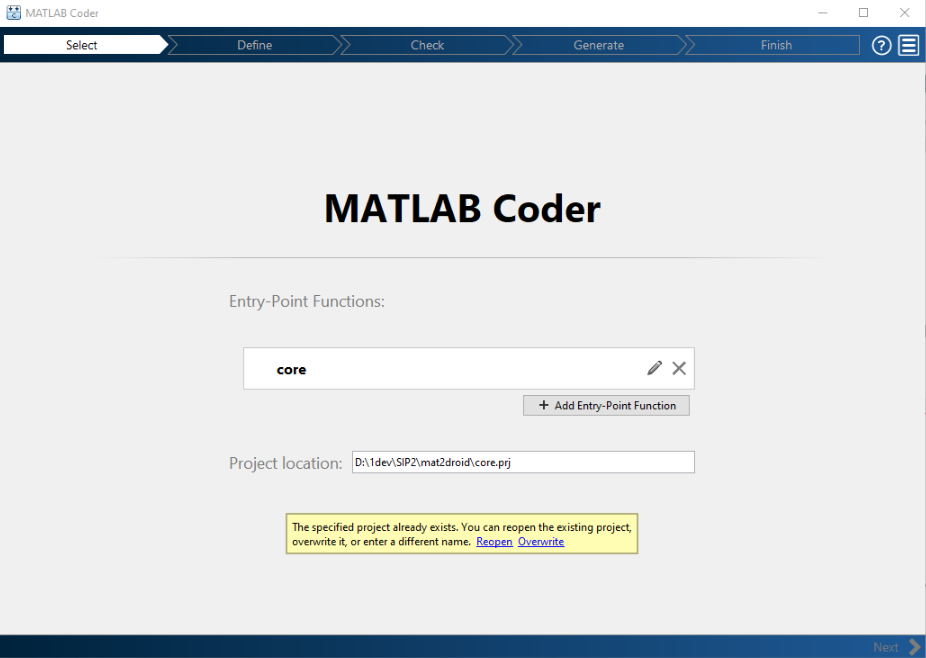
figure(1);

scatter(1:size(input,2),input, 'x'); hold on;

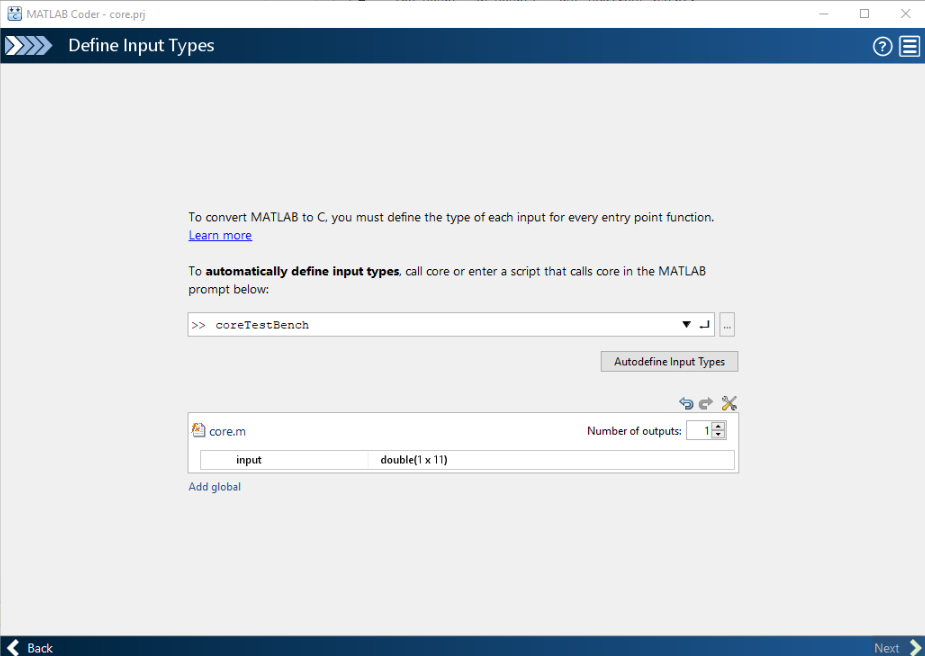
scatter(1:size(output,2),output);

Obviously, we can’t port functions like plot or scatter to android/c++. These functions do not interfere with the autodetection of the datatypes. Matlab coder will suggest a const double[16] input and double[11] output as parameters which is fine.

1. Starting the matlab coder select core.m as the entry point. 



* 1. In the next step select coreTestBench.m as the reference and click the ‘Run Autodetection’ button for the datatypes. The generated definition should look something like:



void core(const double input[11], double output[11]);

* 1. The advanced setup provides an extended configurator for the output code. We’ll stick with C++ and the default configuration.
  2. In the last step matlab will check for runtime errors in the C++ code and generate all necessary files to call the core() method. As far as I know it will not use any syscalls or similar dependencies to be portable.

# Android Part

1. Now, let’s create a basic Android project. Don’t forget to check the checkbox C++ otherwise Android Studio will not be able to utilize NDK for the C++ code. When generating a basic application, Android Studio also will create a sample native-lib.cpp file with a method call returning “hello from c++” as an example. This method we later will be using to interact with our core.
2. Copy all \*.c/\*.h/\*.cpp files from the matlab output folder to the src/cpp/ folder of your Android Studio project. There are plenty of files defining custom datatypes which we all need.
3. In Android Studio under ‘External Build Files’ the CMakeLists.txt file is located. This file describes how the native lib (\*.so) should be compiled. Here we need to add every file to the CMakeLists.txt. [Consult a CMake tutorial for further advice.](https://cmake.org/cmake-tutorial/)

CMakeLists.txt:   
cmake\_minimum\_required(VERSION 3.4.1)

set(SRC

src/main/cpp/core.c

src/main/cpp/core\_initialize.c

src/main/cpp/core\_terminate.c

src/main/cpp/rtGetInf.c

src/main/cpp/rtGetNaN.c

src/main/cpp/rt\_nonfinite.c

src/main/cpp/native-lib.cpp

)

set (INC

src/main/cpp/core.h

src/main/cpp/core\_initialize.h

src/main/cpp/core\_types.h

src/main/cpp/core\_terminate.h

src/main/cpp/rtGetInf.h

src/main/cpp/rt\_nonfinite.h

src/main/cpp/rtGetNaN.h

src/main/cpp/rtwtypes.h

)

add\_library( native-lib SHARED ${INC} ${SRC})

find\_library(log-lib log )

target\_link\_libraries(native-lib ${log-lib} )

1. Now we can to include “core.h” and use the core-method declared.

# JNI Conversion

General information on how to use the native methods inside of Android Java.

Assuming the package name for our application is called com.example.app. Also, this package contains a class Baz. First, we need to load our shiny library inside the class. Now, we can link a method testMethod located in Baz to a native method in the library using the NJI naming convention.

package com.example.app;  
Class Baz {  
 static {  
 System.loadLibrary("native-lib");  
 }  
 public Baz() {};  
 public native void doSomething(); // This is a method located the lib  
}  
  
  
#include <jni.h>

extern "C" JNIEXPORT void JNICALL  
Java\_com\_example\_app\_baz\_doSomething(JNIEnv \*env, jobject instance) {

// Do something useful here.  
}

For passing native data such as a double array you need to convert them to objects Java can work with. Let’s say we want to return a double output[] to our Java class. The c++ method looks like:

extern "C" JNIEXPORT jdoublearray JNICALL  
Java\_com\_example\_app\_baz\_getOutput(JNIEnv \*env, jobject instance) {

const int size = 11;  
double cppDoubleArray[size] = {1.0};  
jdoubleArray out;  
out = env->NewDoubleArray(size);  
env->SetDoubleArrayRegion(cppDoubleArray, 0, size, out);

return out;  
   
}

The Java side would look like this:

public native double[] getOutput();

However, more complex datatypes like structs are not as easily converted since for instance they can have pointers which is not supported in Java. For structs to be passed to Java we would need to write a wrapper Java class which will retrieve data from C++ and convert it to Java datatypes. It’s useful to only provide access/manipulation methods to the native backend instead.