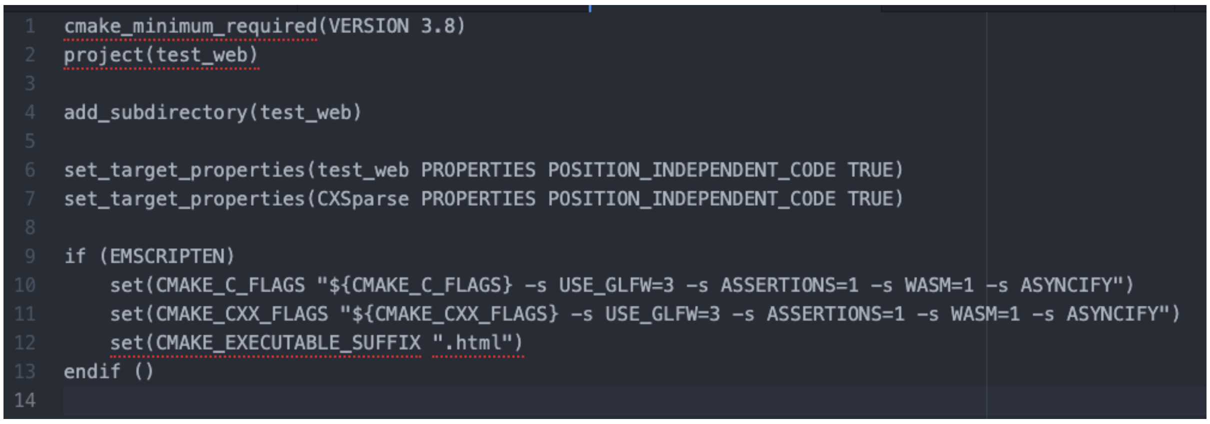
Running C++ on a web browser

Documentation by Heikki Rasilo, 2021

1 Emmake to make library files from C++ code

1. First create an empty folder for your project, in my case “Interactive_tests” (the *root directory* for your project)
2. Copy a folder including all the C code files into this folder, in my case the folder is called “test_web”.
3. Now you need to link all the files needed in your C project together in a consistent library. This can be done using *Make* (more specifically we are using *Emmake* to use the emscripten builder for web-assembly). As a side-note, you can also build a library usable in Python with *make*.
4. In order to use *make,* you need a *Makefile* based on which the library is created. Makefiles can be created with *CMake.* You can check the basics of using CMake for example here: https://thatonegamedev.com/cpp/how-to-setup-cmake-for-c-windows/
5. For Cmake to know what your project consists of, and which files should be linked, you need to make CMakeLists.txt -files for your project. Go to the project’s root directory and create a file named “CMakeLists.txt”. In my project the file looks like this:



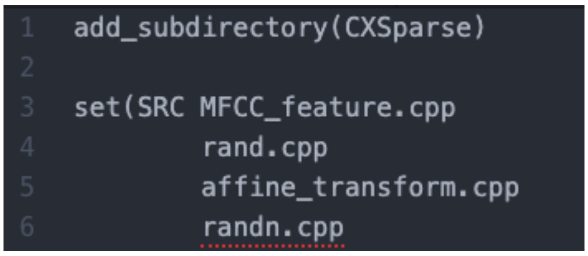
```
1   cmake_minimum_required(VERSION 3.8)
2   project(test_web)
3
4   add_subdirectory(test_web)
5
6   set_target_properties(test_web PROPERTIES POSITION_INDEPENDENT_CODE TRUE)
7   set_target_properties(CXSparse PROPERTIES POSITION_INDEPENDENT_CODE TRUE)
8
9   if (EMSCRIPTEN)
10      set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -s USE_GLFW=3 -s ASSERTIONS=1 -s WASM=1 -s ASYNCIFY")
11      set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -s USE_GLFW=3 -s ASSERTIONS=1 -s WASM=1 -s ASYNCIFY")
12      set(CMAKE_EXECUTABLE_SUFFIX ".html")
13  endif ()
14
```

- The first two lines request the use of a high enough version of cmake, and give the project a name “test_web”.
- On line 4, a subdirectory “test_web” (where my C-code files are located) is added to the project. **Here the processing of the code shifts directly to the CMakeLists.txt file in the added “test_web” -folder (see point 6 below), before continuing to the further lines of this file!**
- Line 6 configures cmake to make *position independent code*, i.e. the code is not tied to a fixed address in memory. More info on this here (https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/)
- Line 7 does the same for the CXSparse folder that is located in my case inside the test_web folder. This is a folder created by Matlab-coder that is used to create C-code from my Matlab project, storing some parts of the required code.

- Lines 9-13 are configuring for Emscripten to create WASM (web-assembly) code. I followed the advice given here:

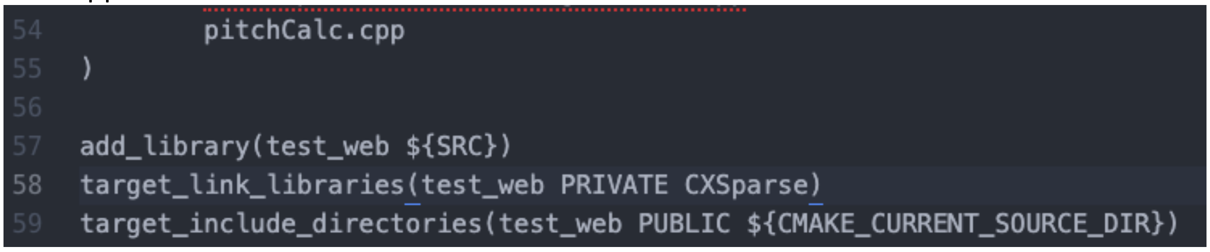
  (https://thatonegamedev.com/cpp/programming-a-c-game-for-the-web-emscripten/) for selecting the settings.

6. This far I have not yet listed any C-files to be included in the library. Since the C-files are located in the *test-web* -folder, I need to make a new CMakeLists.txt file inside that folder:



```
1   add_subdirectory(CXSparse)
2
3   set(SRC MFCC_feature.cpp
4           rand.cpp
5           affine_transform.cpp
6           randn.cpp
```

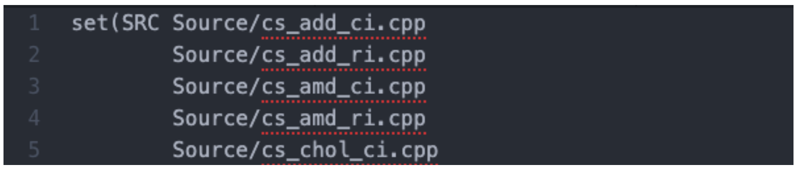
… All cpp-files in that folder …



```
54          pitchCalc.cpp
55  )
56
57  add_library(test_web ${SRC})
58  target_link_libraries(test_web PRIVATE CXSparse)
59  target_include_directories(test_web PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

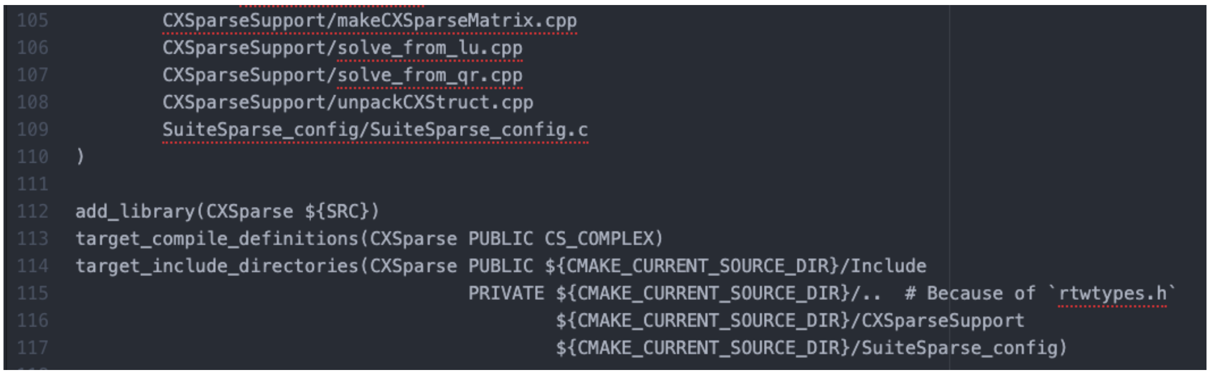
   - The first line again shifts the processing to the CMakeLists.txt file in the “CXSparse”-folder, before processing this file further!
   - Line 57 adds all the files mentioned after line 3 in the library.
   - Line 58 links the CXSparse library contents to the test_web library. This is linked as PRIVATE, meaning that the CXSparse functions will be used internally by test_web, but they cannot be called from outside of this context.
   - Line 59 Includes the directory of the current CMakeLists.txt –file (i.e. (…/Interactive_tests/test_web) in the *test_web* -library. This is used to search for header files “*.h” in the given directories.

7. CMakeLists.txt in the CXSparse subfolder:



```
set(SRC Source/cs_add_ci.cpp
        Source/cs_add_ri.cpp
        Source/cs_amd_ci.cpp
        Source/cs_amd_ri.cpp
        Source/cs_chol_ci.cpp
```

…



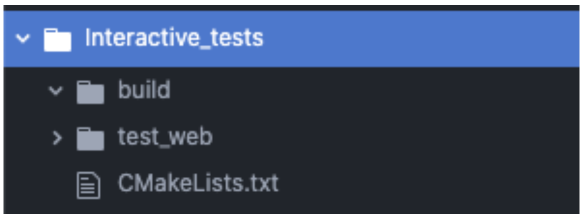
```
105          CXSparseSupport/makeCXSparseMatrix.cpp
106          CXSparseSupport/solve_from_lu.cpp
107          CXSparseSupport/solve_from_qr.cpp
108          CXSparseSupport/unpackCXStruct.cpp
109          SuiteSparse_config/SuiteSparse_config.c
110  )
111
112  add_library(CXSparse ${SRC})
113  target_compile_definitions(CXSparse PUBLIC CS_COMPLEX)
114  target_include_directories(CXSparse PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/Include
115                                      PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/..  # Because of `rtwtypes.h`
116                                              ${CMAKE_CURRENT_SOURCE_DIR}/CXSparseSupport
117                                              ${CMAKE_CURRENT_SOURCE_DIR}/SuiteSparse_config)
```

- Lines 1-109 Add cpp-files again, including from its subfolders
- Line 112, add these files to the CXSparse library
- Line 113, there's some complex numbers used in the CXSparse files, so this line compiles by definition by with the CS_COMPLEX tag. This is then undefined in the beginning of the cpp-files that don't need it.
- Line 114-> Include again all the necessary directories to this target library. All directories including necessary header files are added.

8. To use *Cmake*, go to the root directory and make a folder called "build" by calling:

```
mkdir build
```

At this point the folder structure looks like:



9. Go to the build folder and make the make-file using *emcmake.*

```
cd build
../emsdk/upstream/emscripten/emcmake cmake ..
```

Note, it is important to use the emcmake here, it configures cmake to use the emscripten configurations. Make sure to use your installation folder of emscripten here.

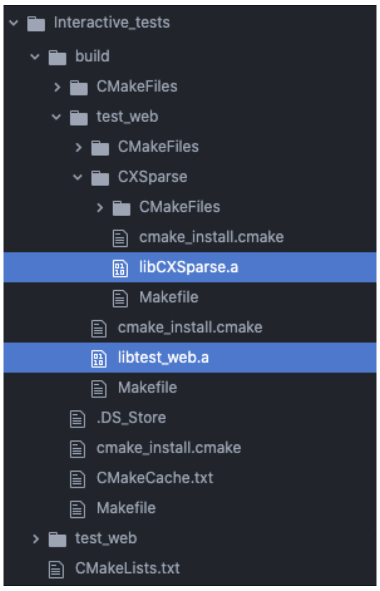
10. At this point the makefiles are done, and you can build the libraries by calling

```
../emsdk/upstream/emscripten/emmake make ..
```

If there's bugs in the c++ code. They should be caught at this time.

11. Now the library files are created, in my case they appear as *libtest_web.a* in the test_web -folder (inside the build folder), and *libCXSparse.a* under the CXSparse

folder.



```
Interactive_tests
  build
    CMakeFiles
    test_web
      CMakeFiles
      CXSparse
        CMakeFiles
        cmake_install.cmake
        libCXSparse.a
        Makefile
      cmake_install.cmake
      libtest_web.a
      Makefile
    .DS_Store
    cmake_install.cmake
    CMakeCache.txt
    Makefile
  test_web
  CMakeLists.txt
```

12. Copy the library files in the root folder:

```
cd ..
cp build/test_web/libtest_web.a .
cp build/test_web/CXSparse/libCXSparse.a .
```

2 Compiling libraries in Web Assembly code

Now it’s time to compile the libraries in the web-assembly (WASM) code. This is done using *emcc*. There are several options to give to the compiler, but here my command looks like:

```
../emsdk/upstream/emscripten/emcc libCXSparse.a libtest_web.a
   -o hello3.html
 -O1
 -s WASM=1
 --shell-file shell_minimal.html
 -s NO_EXIT_RUNTIME=1
 -s "EXPORTED_RUNTIME_METHODS=['ccall']"
 -s EXPORT_ALL=1
 -s LINKABLE=1
 -s EXPORTED_FUNCTIONS="['_malloc']"
 -s TOTAL_STACK=20mb
 -s INITIAL_MEMORY=25mb
 -s ALLOW_MEMORY_GROWTH=1
 -s WASM_BIGINT --profiling
```

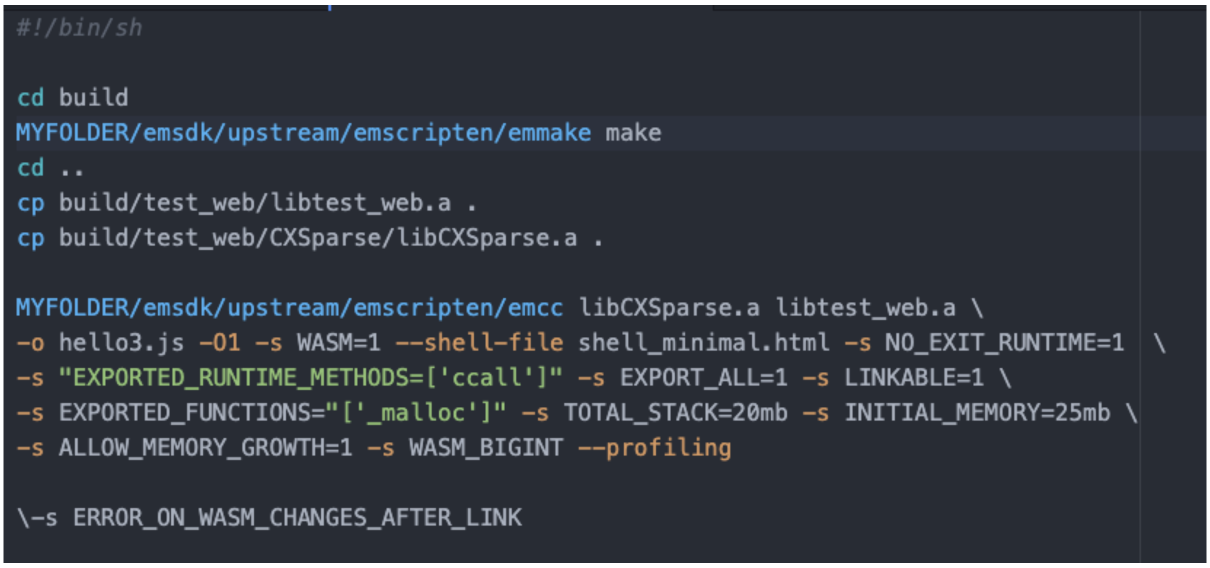
To create an example html-page that you can start adjusting to your needs, on the **FIRST CALL** to emcc, use the *-o filename.html* option. This will create the html-file based on the cell file given also in the options. I’m using the shell_minimal.html (https://github.com/emscripten-core/emscripten/blob/main/src/shell_minimal.html), that has to be copied in the root directory as well.

Note that it is **very important** that when you keep on working on the html-file and adding functionality, keep in mind that calling emcc with this flag again will overwrite the html-file. If you need to adjust the c-code and recompile, use a flag

```
-o filename.js
```

instead (**i.e. REMOVE the *filename.html* from the flags)**. This ensures that only the .js file (and the WASM code) is updated and you don't lose the changes made in the html-file.

For convenience, after the first-time creation of the html file, I created a bash-script *make_and_link.sh* that makes linking and compiling very fast, whenever a change in the c-code is needed:



```
#!/bin/sh

cd build
MYFOLDER/emsdk/upstream/emscripten/emmake make
cd ..
cp build/test_web/libtest_web.a .
cp build/test_web/CXSparse/libCXSparse.a .

MYFOLDER/emsdk/upstream/emscripten/emcc libCXSparse.a libtest_web.a \
-o hello3.js -O1 -s WASM=1 --shell-file shell_minimal.html -s NO_EXIT_RUNTIME=1  \
-s "EXPORTED_RUNTIME_METHODS=['ccall']" -s EXPORT_ALL=1 -s LINKABLE=1 \
-s EXPORTED_FUNCTIONS="['_malloc']" -s TOTAL_STACK=20mb -s INITIAL_MEMORY=25mb \
-s ALLOW_MEMORY_GROWTH=1 -s WASM_BIGINT --profiling

\-s ERROR_ON_WASM_CHANGES_AFTER_LINK
```

Codebox 1

You can just run this in the root folder using

```
bash make_and_link
```

after any change in the C-code. Your html-file remains unchanged but the underlying functionality is updated.

NOTE: Make always a hard refresh on your browser after changes in the code! It is possible that changes do not take place in your html-page without hard refresh due to some caching issue. In Chrome in Mac for example, this is done using cmd+shift+R.

3 Calling Web Assembly functions using JavaScript

The automatically created HTML-file now consists of formatting, an output text box etc. You can clean up this file and remove all unnecessary components. I recommend leaving the text box here, since (using the default module) your "printf" commands from the C-code will be printed here, that can be very handy for debugging.

You can quickly start using and developing the code, by starting a server using Python. Go to the directory of your html page and use

```
python3 -m http.server
```

Out comes something like:

```
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

Now you can copy this http://0.0.0.0:8000/ in your browser and select the html file.

The *Module* is used to communicate between the Emscripten generated code and JavaScript. For example, calling a function written in c++ will be done using *Module.ccall.* In the default generated html, the Module is initialized, but it can be modified if needed.

3.1 Example of a function call

Module.ccall function can be used to call functions in the c++ code. Using this function requires the flag:

```
-s "EXPORTED_RUNTIME_METHODS=['ccall']"
```

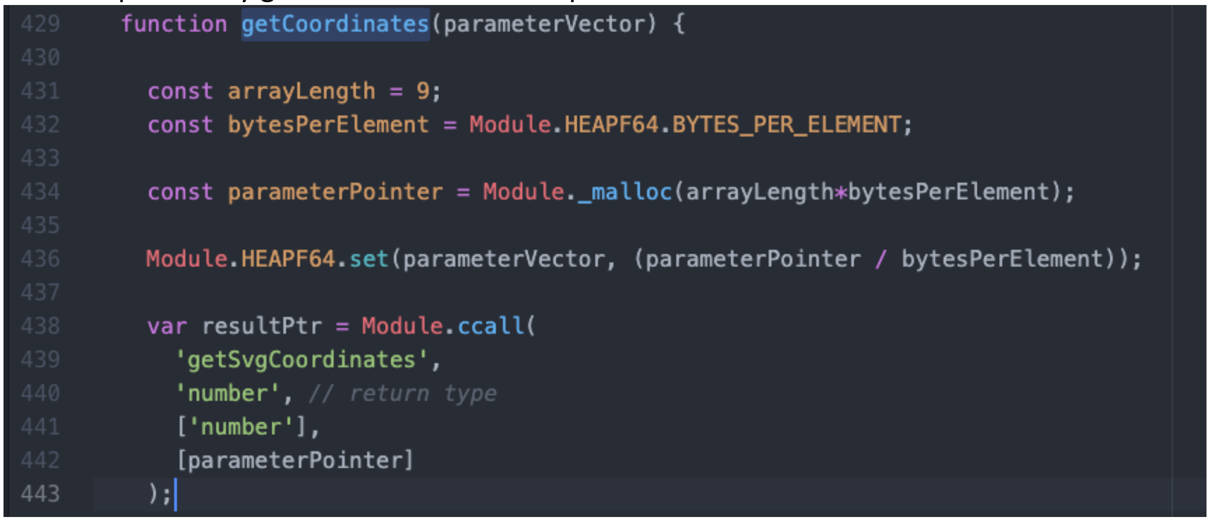
To be used in the compilation phase (see Codebox 1).

In my example, in the end of the html.file, I create the initial SVG of the vocal tract model, as soon as the Module has been loaded. Without waiting the module to be loaded, the program probably crashes, since the Module cannot be used.

```
  Module['onRuntimeInitialized'] = function() {
      makeInitialSVG()
  }
```

"makeInitialSVG()" is a call to a JavaScript function written in the same html file. This function calls a JS function getCoordinates, that returns coordinates of the lines of the vocal tract that are to be drawn on the website. In order for the c-code to calculate the positions of the lines, it needs to get values of the nine vocal tract parameters as input. The *Module.ccall* function cannot take an array as an input parameter
(check: https://emscripten.org/docs/api_reference/preamble.js.html#id3),
so I have to allocate memory where I write this parameter vector, and then give a pointer to that memory as an input to *ccall*, since this pointer is just one number and 'number'-style parameter is allowed.

The first part of my getCoordinates JavaScript-function looks like:



```
function getCoordinates(parameterVector) {

  const arrayLength = 9;
  const bytesPerElement = Module.HEAPF64.BYTES_PER_ELEMENT;

  const parameterPointer = Module._malloc(arrayLength*bytesPerElement);

  Module.HEAPF64.set(parameterVector, (parameterPointer / bytesPerElement));

  var resultPtr = Module.ccall(
    'getSvgCoordinates',
    'number', // return type
    ['number'],
    [parameterPointer]
  );
```

Codebox 2

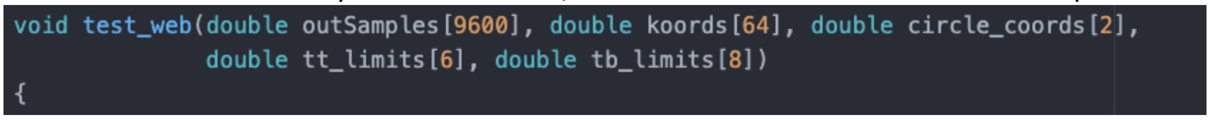
First, I define the length of the array to be written in the memory. The array is of double-precision float format, so each array element takes *Module.HEAPF64.BYTES_PER_ELEMENT* amount of memory. *Module._malloc* allocates the necessary amount of memory, and parameterPointer points to this memory. To use malloc, you need to add the

```
-s EXPORTED_FUNCTIONS="['_malloc']"
```

tag in the compiling phase (see Codebox 1). Next you write the parameterVector in the allocated memory, pointed by the pointer. Now you can call the C-function getSvgCoordinates with ccall, using the parameterPointer as input.

3.2 The C++ side, to make things callable

In c++, you have to make a few changes to make function calls work properly. My example is the test_web.cpp file, that was automatically coded to C++ from Matlab, using Matlab's C-coder application. Originally this file had just one function to test the functionality of the vocal tract model. It didn't take any input parameters, but it set the vocal tract parameters to some values and returned the synthesized sound, and information about the vocal tract position:



```
void test_web(double outSamples[9600], double koords[64], double circle_coords[2],
              double tt_limits[6], double tb_limits[8])
{
```

However, this file provided a good starting point to adjust things to be more suitable for the web-interface.

Following guidelines from
(https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm),
I added a main function to test the C++ file. This function will be called by default in the loading of the module (called in the automatically generated .js-file in the callMain-function).

```
#include <stdio.h>
#include <emscripten/emscripten.h>

int main() {
    printf("Hello World, from test_web file\n");
}
```

This text should be written in the textbox of your html page when the page is loaded.

Under this main function, the following lines must be added.

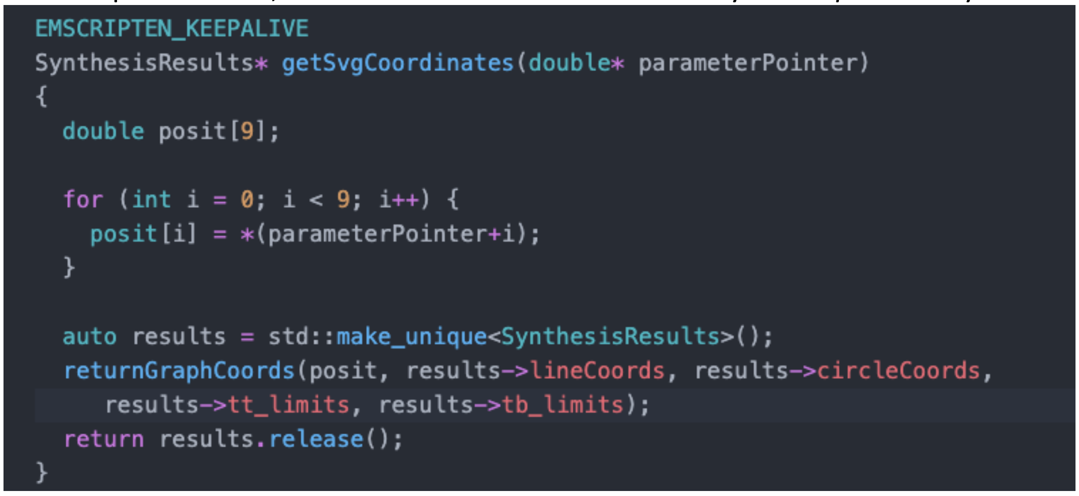
```
#ifdef __cplusplus
extern "C" {
#endif
```

This prevents the compiler from modifying the names of functions. Apparently, for c++ code the function names are changed in the compilation stage to include some information of the function variables. This wrapping prevents that from happening, and it will be closed in the very end of the same file:

```
#ifdef __cplusplus
}
#endif
```

By default, other function definitions in the file are eliminated. You have to keep them alive by using EMSCRIPTEN_KEEPALIVE, that requires the library *emscripten/emscripten.h* to be loaded (see the code box above). Now I wrote a new function inside the test_web.cpp file, that will be called from JavaScript, using *ccall* as seen before.

This function is defined to take a pointer as an input parameter (*parameterPointer)*, exactly what we are calling it with using the *ccall*. I define a new array of size 9, called *posit*, that will be storing the contents of the memory, where the *parameterPointer* is pointing to. Next, element per element, I write the contents of the memory to the *posit*-array.

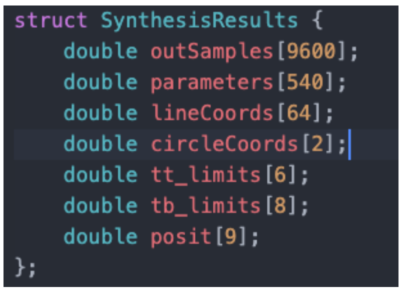


```
EMSCRIPTEN_KEEPALIVE
SynthesisResults* getSvgCoordinates(double* parameterPointer)
{
  double posit[9];

  for (int i = 0; i < 9; i++) {
    posit[i] = *(parameterPointer+i);
  }

  auto results = std::make_unique<SynthesisResults>();
  returnGraphCoords(posit, results->lineCoords, results->circleCoords,
     results->tt_limits, results->tb_limits);
  return results.release();
}
```

Codebox 3

The C++ function, returnGraphCoords gives the required coordinates for drawing the vocal tract image, given the parameter values as an input. These coordinates have to be somehow transferred back to our JavaScript side. For this, in the beginning of this file (test_web.cpp), I define a struct *SynthesisResults*, that will be used to store all the parameters that we may need at the JS side.

Note that the *SynthesisResults*-struct is created outside of any function and it has a global scope. This is important so that the memory for this struct is not automatically deallocated during the execution of the program. If you would define this struct inside the getSvgCoordinates for example, and return a pointer to this struct, the memory would be deallocated in the end of the function and may be corrupted before it is used on the JavaScript side.

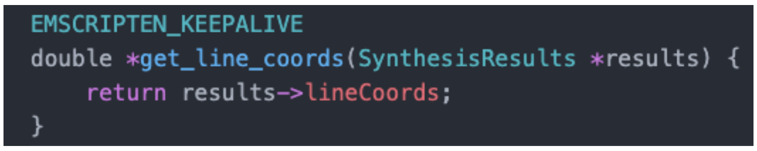


```
struct SynthesisResults {
    double outSamples[9600];
    double parameters[540];
    double lineCoords[64];
    double circleCoords[2];
    double tt_limits[6];
    double tb_limits[8];
    double posit[9];
};
```

A unique pointer to this struct will be created, and the outputs from *returnGraphCoords* will be stored in this struct. The unique pointer is needed, so that unique access to that memory can be returned to the JavaScript side, and the memory can be deallocated if needed later by destroying the pointer. The pointer is passed as an output argument, that can be caught in the *ccall*-function on the JS side:

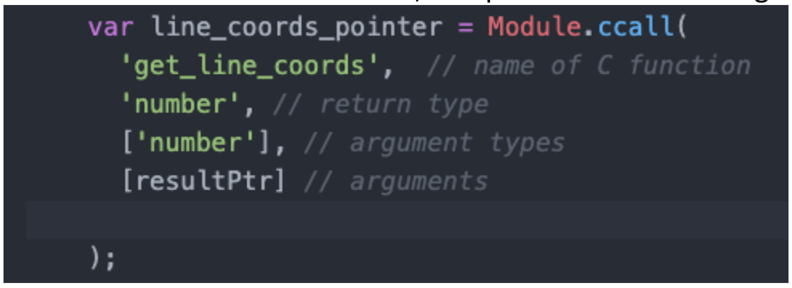
```
return results.release();
```

This pointer is caught in the resultPtr variable in JS, as seen in Codebox 2. In order to easily return the arrays stored in the different fields of the SynthesisResults struct, I have written dedicated functions on top of the test_web.cpp-file. An example function to return pointer to the *lineCoords*-field:



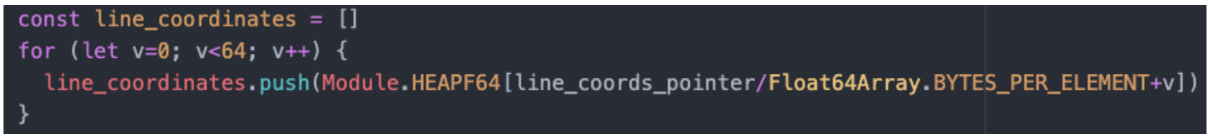
```
EMSCRIPTEN_KEEPALIVE
double *get_line_coords(SynthesisResults *results) {
    return results->lineCoords;
}
```

This function takes the resultPtr as an input parameter, and returns a pointer to the *results->lineCoords* field. On the JS side, this pointer is obtained again by ccall:



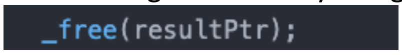
```
    var line_coords_pointer = Module.ccall(
      'get_line_coords',  // name of C function
      'number', // return type
      ['number'], // argument types
      [resultPtr] // arguments

    );
```

Now we have only obtained a pointer to the memory, where the line coordinates are stored. We have to read that memory into a JavaScript array in order to be able to use that in our program. This can be done by declaring an empty array in JS, and reading the memory from the pointer directly into the array:



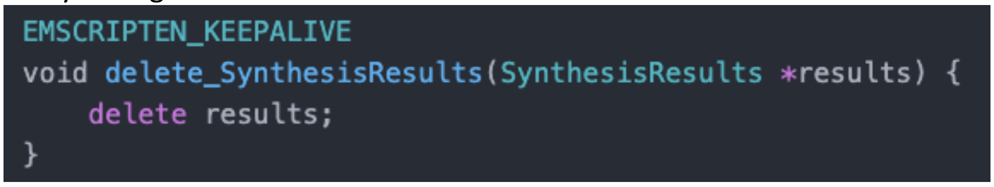
```
const line_coordinates = []
for (let v=0; v<64; v++) {
  line_coordinates.push(Module.HEAPF64[line_coords_pointer/Float64Array.BYTES_PER_ELEMENT+v])
}
```

As in c++, the arrays were of *double* datatype, which is equal to Float64 datatype, they are stored in the HEAPF64 memory. In order to point at the right locations in that memory, the correct number of bytes required by an array element has to be used, defined by the datatype (float64). When all the data has been read from memory, free the pointer, deallocating the memory using:



```
  _free(resultPtr);
```

Or by calling the destructor function:

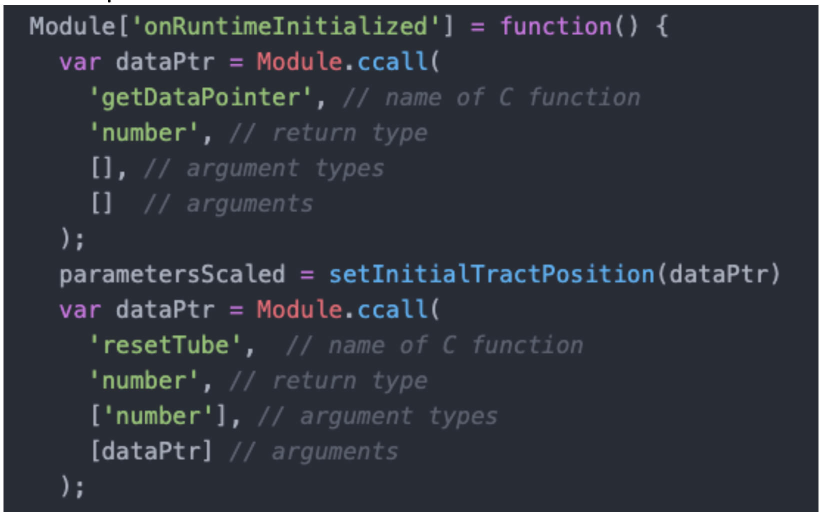


```
EMSCRIPTEN_KEEPALIVE
void delete_SynthesisResults(SynthesisResults *results) {
    delete results;
}
```

NOTE: Remember to deallocate the memory, after all the data has been read to JavaScript variables! Otherwise your program will continuously grow in memory usage, because memory for a new struct is allocated every time the function is called.

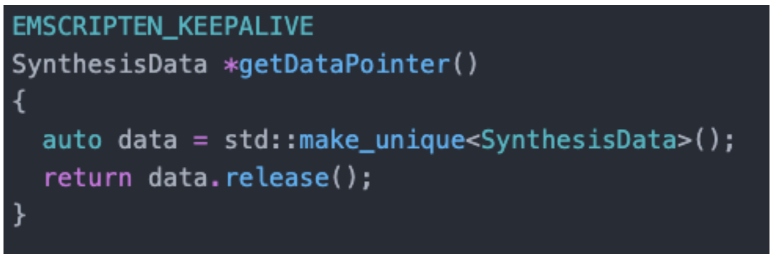
Optionally, instead of freeing the memory, and creating a new SynthesisResults struct in c++ every time a function is called, you could initialize the struct in the start-up of the HTML-page. A pointer to that struct could be then sent back and forwards between C++ and JavaScript. This helps to maintain fields of the struct unchanged in between several calls to the C++ function. This is useful for example when a status of the vocal tract synthesizer has to be saved, to continue the synthesis from the previous state on the next call to the function. This is done in the following example, where as soon as the JS module has been loaded, I create a struct in the C++ and return a pointer to it. This pointer is then sent back to the “resetTube” c++ function, and again back to JavaScript:

JavaScript calls:

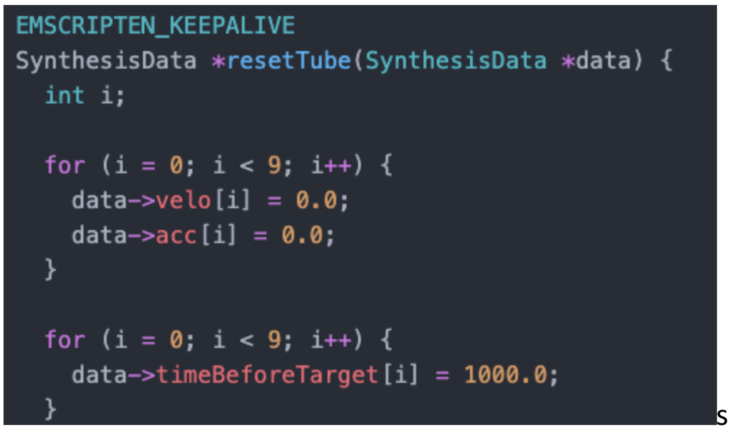


```
Module['onRuntimeInitialized'] = function() {
  var dataPtr = Module.ccall(
    'getDataPointer', // name of C function
    'number', // return type
    [], // argument types
    []  // arguments
  );
  parametersScaled = setInitialTractPosition(dataPtr)
  var dataPtr = Module.ccall(
    'resetTube',  // name of C function
    'number', // return type
    ['number'], // argument types
    [dataPtr] // arguments
  );
```

C++ code:



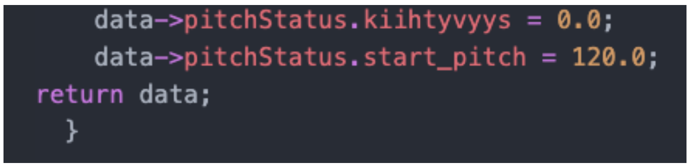
```
EMSCRIPTEN_KEEPALIVE
SynthesisData *getDataPointer()
{
  auto data = std::make_unique<SynthesisData>();
  return data.release();
}
```



```
EMSCRIPTEN_KEEPALIVE
SynthesisData *resetTube(SynthesisData *data) {
  int i;

  for (i = 0; i < 9; i++) {
    data->velo[i] = 0.0;
    data->acc[i] = 0.0;
  }

  for (i = 0; i < 9; i++) {
    data->timeBeforeTarget[i] = 1000.0;
  }
```

s

…



```
      data->pitchStatus.kiihtyvyys = 0.0;
      data->pitchStatus.start_pitch = 120.0;
  return data;
    }
```