APÊNDICE A – Preparação biblioteca OpenCV

O primeiro passo diz respeito a construção das bibliotecas estáticas do OpenCV com o CMake, Ninja e Emscripten. Para isso foram realizados as seguintes ações:

1) Fazer o download e descompactar as bibliotecas OpenCV 3.3.0 e módulos extra OpenCV Contrib:

```
francisco@netuno ~/Documentos/TCC $ wget https://github.com/opencv/opencv/archive/3.3.0.zip
```

```
francisco@netuno ~/Documentos/TCC $ unzip 3.3.0.zip
```

```
francisco@netuno ~/Documentos/TCC $ wget https://github.com/opencv/opencv/archive/3.3.0.zip
```

```
francisco@netuno ~/Documentos/TCC $ unzip 3.3.0.zip
```

2) Instalação CMake:

```
francisco@netuno ~/Documentos/TCC $ apt-get install cmake
```

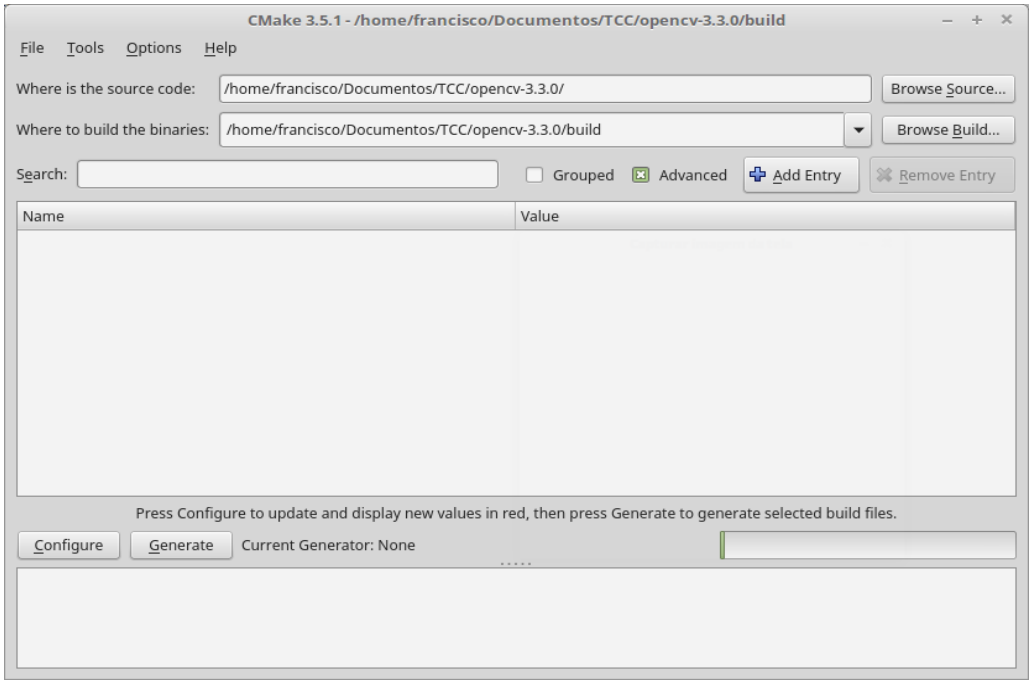
3) Instalação Ninja:

```
francisco@netuno ~/Documentos/TCC $ apt-get install ninja
```

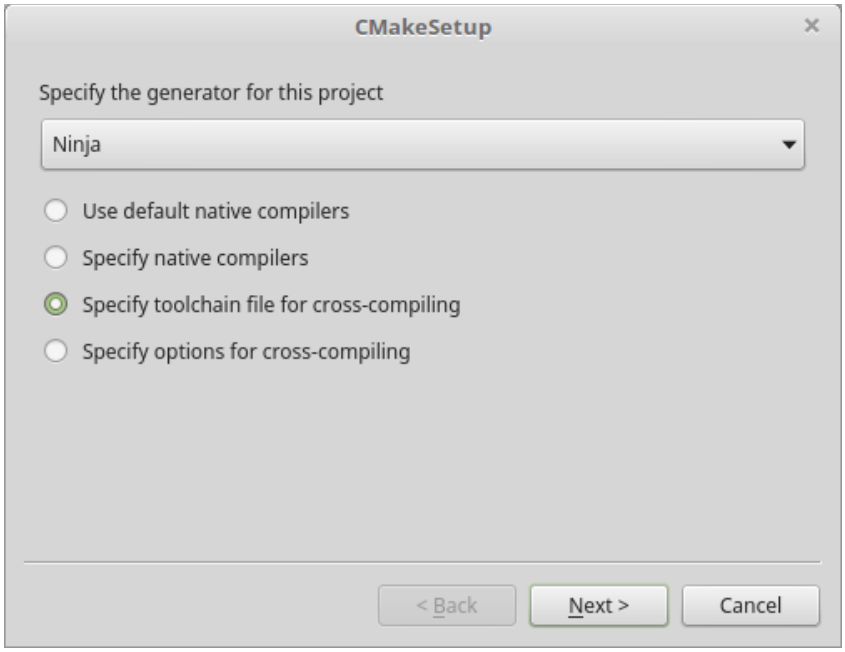
4) Instalação Emscripten: O Emscripten possui uma versão *portable, as instruções para instalação estão em: https://kripken.github.io/emscripten-site/docs/getting_started/downloads.html*

5) Configurar sistema de *build*: Abrir o CMake, configurar o projeto OpenCV utilizando o ninja e selecionando o *toolchain* Emscripten.

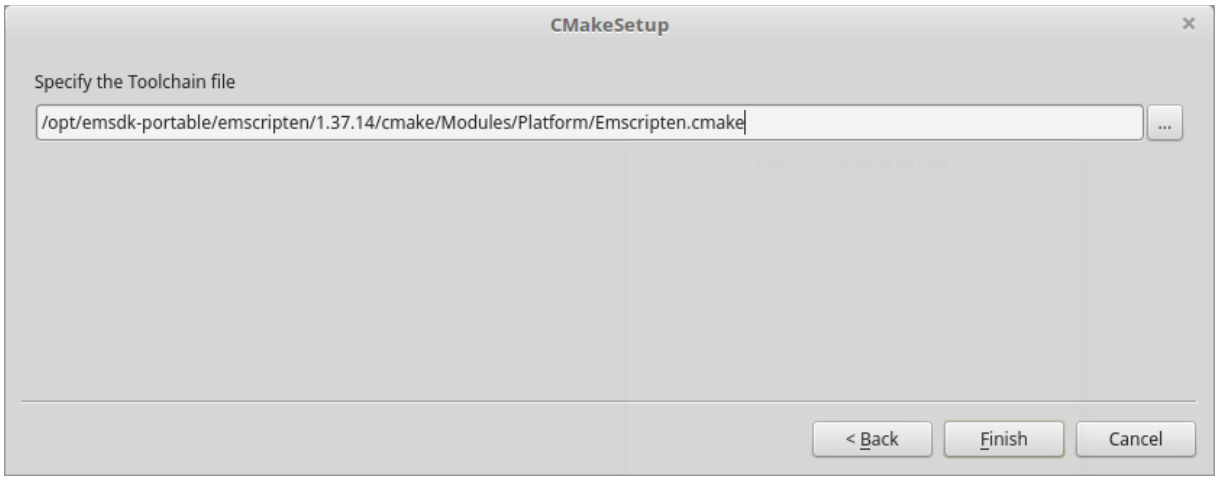
- Neste passo são necessários informar o local dos arquivos fonte da biblioteca OpenCV e o diretório onde serão gerados a saída compilada. Após isso deve ser feita a configuração do projeto, clicando no botão “Configure”.



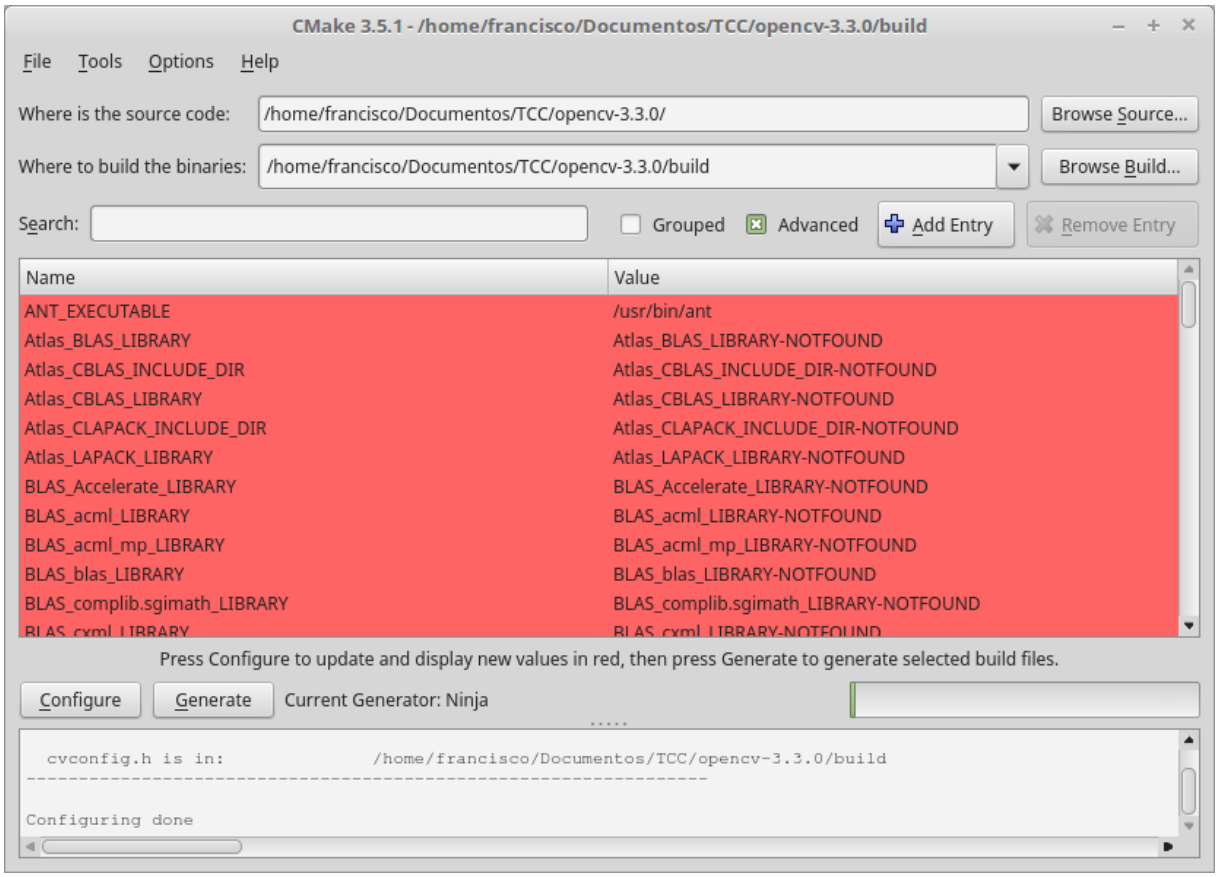
- Na tela de configuração, selecionar o o Ninja como gerador para este projeto. Marcar também a opção para especificar um *toolchain* de compilação.



- Especificar o caminho para o Emscripten *toolchain.* Ele é fornecido juntamente com a instalação do Emscripten Portable.



- Após isso o CMake fará alguns testes. Se não ocorrerem erros o projeto pode ser configurado.



- Neste experimento foram feitas as seguintes alterações nas diretivas de compilação:

- Foram habilitados:

BUILD_opencv_*
BUILD_JPEG
BUILD_PNG

- Desabilitados

BUILD_DOCS
BUILD_opencv_freetype
BUILD_opencv_biospired
BUILD_EXAMPLES
BUILD_FAT_JAVA_LIB
BUILD_IPP_IW
BUILD_PACKAGE
BUILD_PERF_TESTS
BUILD_SHARED_LIBS
BUILD_TESTS
BUILD_WITH_DEBUG_INFO
CV_ENABLE_INTRINSICS
WITH_PTHREADS_PF

- Para as diretivas CPU_BASELINE e CPU_DISPATCH devem ser selecionas as opções em branco.

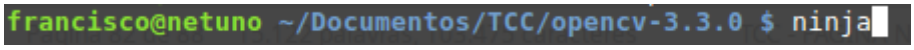
- Mudar a diretiva CMAKE_BUILD_TYPE para “Release”

- Configurar as opções do compilador Emscripten CMAKE_CXX_FLAGS and CMAKE_C_FLAGS para:

--llvm-lto 1 --bind -s WASM=1 -s ALLOW_MEMORY_GROWTH=1 -s DISABLE_EXCEPTION_CATCHING=0 -s ASSERTIONS=2 --memory-init-file 0 -O3

- Setar a variável OPENCV_EXTRA_MODULES_PATH com o diretório onde estão os módulos extras

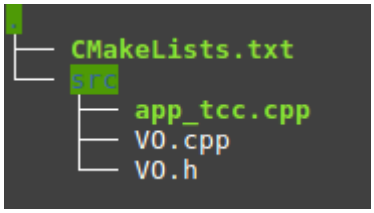
Após as alterações o projeto deve ser reconfigurado, clicando no botão “Configure” novamente. Quando o processo estiver pronto basta clicar no botão “Generate”. Para finalizar basta rodar o comando ninja dentro da pasta OPENCV_DIR/build. Com isso as bibliotecas OpenCV estarão compiladas para uso.



APÊNDICE B – Projeto C++ utilizando o OpenCV

Este passo a passo tem o objetivo de descrever o projeto C++ e a geração do ".wasm" que é carregado pelo navegador. Serão apresentadas o processo de configuração e compilação do projeto C++ , até a integração e uso do WebAssembly em uma página *web*. Da mesma forma que o processo de compilação apresentado no APÊNDICE A, este roteiro necessita a instalação do CMake, Ninja e Emscripten.

1) Criação do projeto, com a seguinte estrutura:

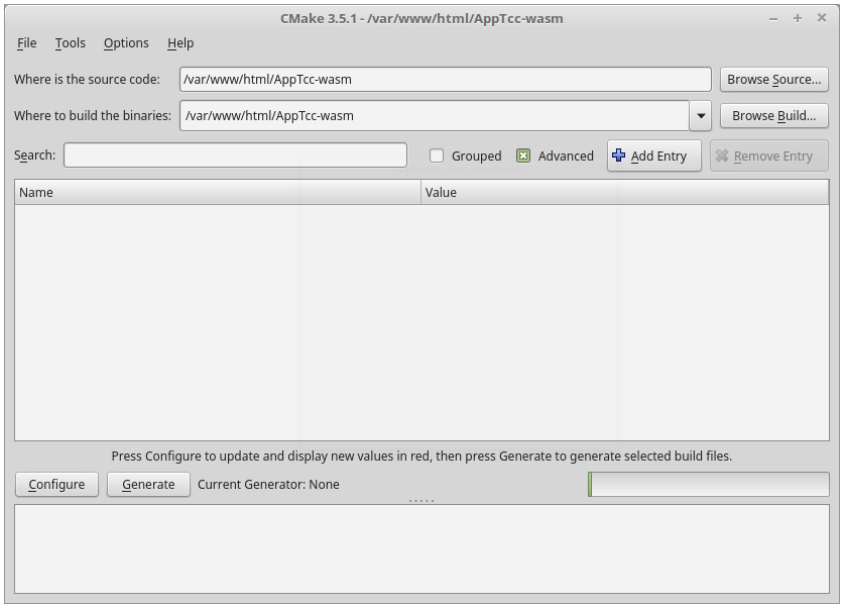


- **CmakeLists.txt:** contém as regras para construção do projeto
- **app_tcc.cpp:** arquivo principal que contém a função que será chamado via JavaScript, responsável por fazer o processamento dos *frames* do vídeo
- **VO.cpp:** contém a classe com funções relacionadas a analise e rastreio de pontos nas imagens
- **VO.h:** *headers* da classe contida no arquivo VO.cpp

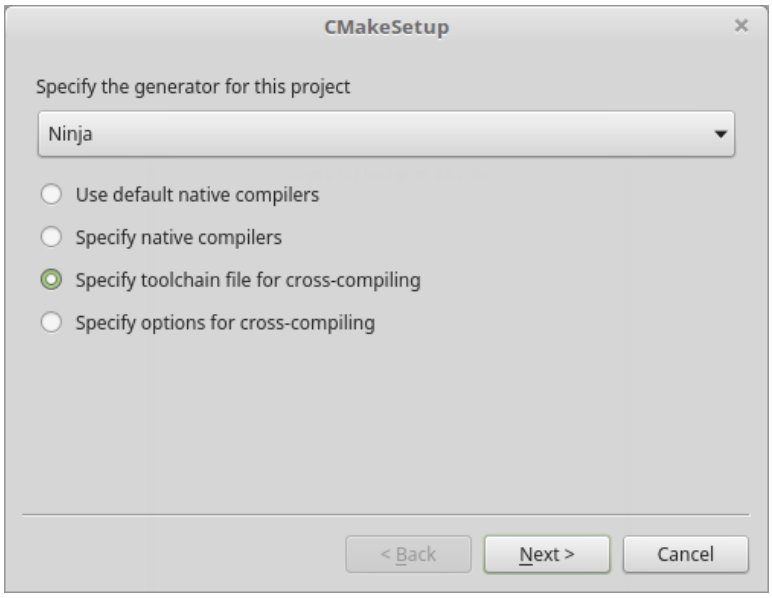
2) Configuração do sistema de *build:* O arquivo CmakeLists.txt contém as regras para geração do projeto utilizando o CMake, onde são definidas dependências, bibliotecas e o executável gerado.

```
project (AppTccWasm) # nome do projeto
cmake_minimum_required(VERSION 3.0)

set(OpenCV_STATIC ON)
find_package(OpenCV REQUIRED) # biblioteca OpenCV é requerida
include_directories(${OpenCV_INCLUDE_DIRS} .)

add_library(app_tcc_lib src/VO.cpp src/VO.h) # adiciona bibliotecas do projeto

if (EMSCRIPTEN)

    add_executable (app_tcc_wasm src/app_tcc.cpp) # arquivo executável

    target_link_libraries(app_tcc_wasm ${OpenCV_LIBS} app_tcc_lib) # inclui bibliotecas OpenCV no projeto

    # após a compilação, copia o arquivo app_tcc_wasm.js para a pasta do projeto web
    add_custom_command(TARGET app_tcc_wasm POST_BUILD
                       COMMAND ${CMAKE_COMMAND} -E copy_if_different
                       ${CMAKE_CURRENT_BINARY_DIR}/app_tcc_wasm.js
                       /var/www/html/AppTcc/www/testes/teste-wasm/app_tcc_wasm.js)

    # após a compilação, copia o arquivo app_tcc_wasm.js para a pasta do projeto web
    add_custom_command(TARGET app_tcc_wasm POST_BUILD
                       COMMAND ${CMAKE_COMMAND} -E copy_if_different
                       ${CMAKE_CURRENT_BINARY_DIR}/app_tcc_wasm.wasm
                       /var/www/html/AppTcc/www/testes/teste-wasm/app_tcc_wasm.wasm)

endif()

if(UNIX)

    # flags compilação Emscripten
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++1z --llvm-lto 1 --bind -s WASM=1 ")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -s ALLOW_MEMORY_GROWTH=1 -s DISABLE_EXCEPTION_CATCHING=0 ")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -s ASSERTIONS=2 ")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -s DISABLE_EXCEPTION_CATCHING=0 -s NO_FILESYSTEM=1 ")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -s NO_EXIT_RUNTIME=1 ")
    SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} --memory-init-file 0 -O3")

    if(DEFINED CMAKE_BUILD_TYPE)
        SET(CMAKE_BUILD_TYPE ${CMAKE_BUILD_TYPE})
    else()
        SET(CMAKE_BUILD_TYPE Release)
    endif()

endif()
```

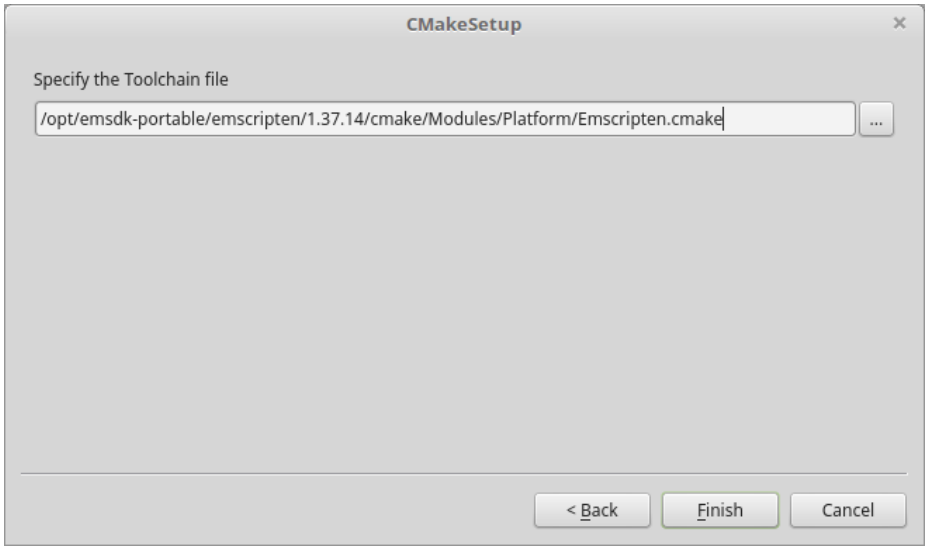
- A configuração e geração dos arquivos de configuração com CMake precisa ser feito apenas na inicialização do projeto. Inicialmente, com o CMake aberto, basta informar o diretório fonte e o diretório onde será construído o projeto. Neste exemplo foi definido o diretório “/var/www/html/AppTcc-wasm”.



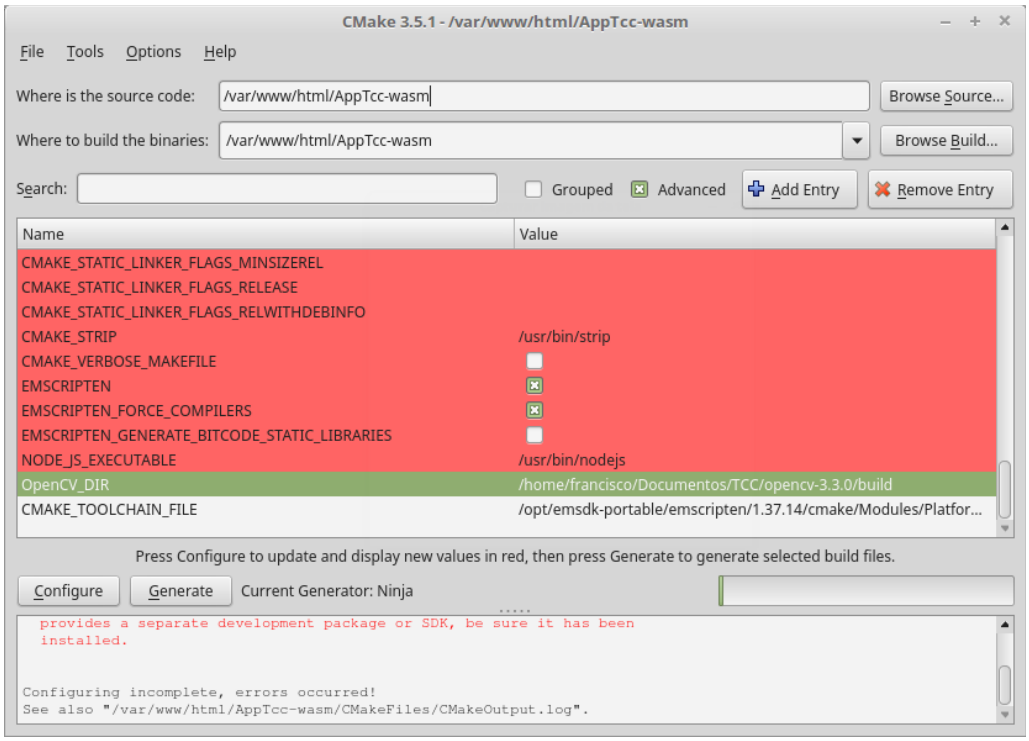
- Em seguida, ao clicar em “configure”, é preciso especificar o ”Ninja” como alvo para geração do projeto e marcar a opção que permite especificar um *toolchain* para compilação.



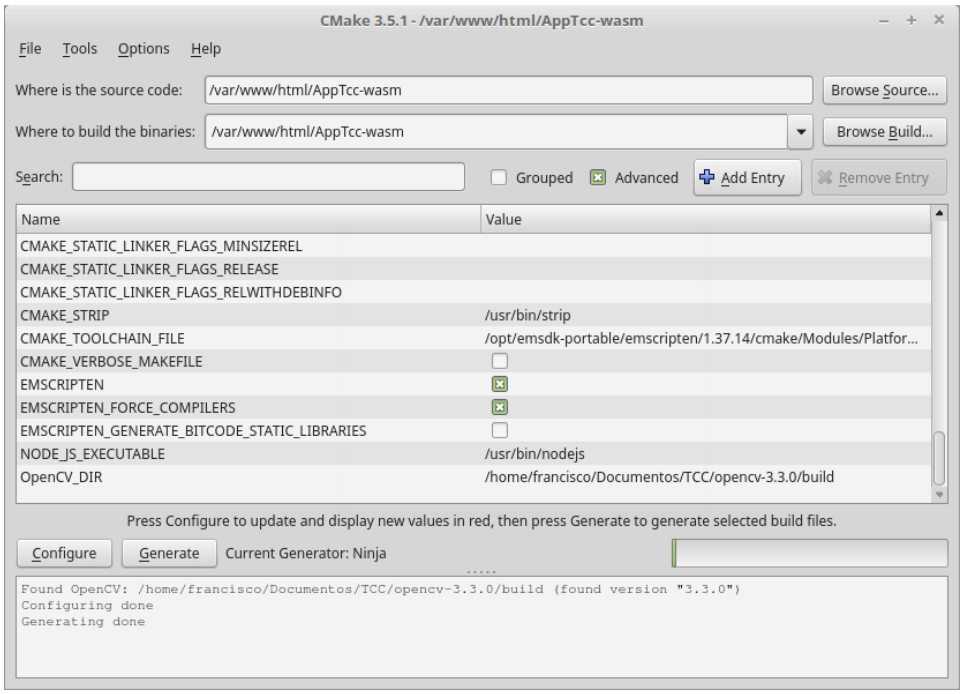
- Aqui é preciso especificar o Emscripten como *toolchain* para compilação, informando o caminho do arquivo “Emscripten.cmake”. Este arquivo encontra-se no diretório do Emscripten Portable.



- O CMake fará algumas checagens do ambiente de desenvolvimento. É preciso definir a diretiva “OpenCV_DIR” com o caminho da pasta onde foi gerado o projeto OpenCV (ver APÊNDICE A). Neste exemplo o foi utilizado o diretório “/home/francisco/Documentos/TCC/OpenCV-3.0.0/build”



- Ao final da configuração basta gerar os arquivos que definem a construção do projeto clicando no botão “Generate”



3) Compilar projeto: Após a gerar os arquivos de configuração é o momento de fazer a compilação, executando o comando ”ninja” de dentro da pasta raiz do projeto.

```
francisco@netuno /var/www/html/AppTcc-wasm $ ninja
[0/1] Re-running CMake...
-- Configuring done
-- Generating done
-- Build files have been written to: /var/www/html/AppTcc-wasm
[4/4] Linking CXX executable app_tcc_wasm.js
emcc: warning: cannot represent a NaN literal '0x2ad8a80' with c
emcc: warning: cannot represent a NaN literal '0x2ad8a80' with c
francisco@netuno /var/www/html/AppTcc-wasm $
```

Serão gerados dois novos arquivos como resultado da compilação, o “app_tcc_wasm.js” e o “app_tcc_wasm.wasm”. O “.js” é uma espécie de cola, que contém todo o código responsável por instanciar o módulo WebAssembly a partir do “.wasm”, configurar a memória e tornar disponível as funções escritas em C++.

4) Integração HTML: O exemplo abaixo mostra a integração do .wasm em uma página HTML e um trecho de código JavaScript chamando a função WebAssembly.

```
<!doctype html>
<html>
<body>
<script>

var Module = {
    wasmBinaryFile: 'app_tcc_wasm.wasm',  // define o arquivo .wasm que deve ser carregado

    _main: function() { // função chamada quando módulo WebAssembly esta pronto
        console.log("WASM Pronto");
    },
    _exit: function(x) {
        console.log("fim do programa");
    }
};
</script>

<script type="text/javascript">

    ...

    let homography_matrix;
    let homography_matrix_size = 9; // matrix 3x3
    try {

        // chama a função WebAssembly criada para calcular a homografia
        homography_matrix = Module._vo_homography(img_data.width,      // largura da imagem
                                        img_data.height,    // altura da imagem
                                        fp.frame_bytes.byteOffset, // ponteiro referenciando a imagem
                                        frameIndex, // número do frame
                                        homography_matrix_size); // tamanho do vetor de resultados

    } catch (e) {
        throw e
    }
    // popula o array com o resultado da homografia obtido a partir da função WebAssembly
    var homography = [];
    for (let v = 0; v < homography_matrix_size; v++) {
        homography.push(Module.HEAPF64[homography_matrix / Float64Array.BYTES_PER_ELEMENT + v])
    }

    ...

</script>
<script type="text/javascript" src='app_tcc_wasm.js'></script>
</body>
</html>
```