CS100
Introduction to Programming

Lecture 21. Cmake and interfacing with C++

Today's learning objectives

- Build systems tour
- Meeting CMake
- Basic CMake usage
- Calling C++ from Matlab

Why build systems?

- We write an application (source code) and need to:
  - Compile the source-code
  - Link to other libraries
  - Distribute your application as source and/or binary
- We would also like to be able to:
  - Run tests on your software
  - Run test of the redistributable package
  - See the results of that

Compiling

- Manually?

```
g++ -DMYDEFINES -c myapp.o myapp.cpp
```

- Unfeasible when:
  - we have many files
  - some files should be compiled only in a particular platform
  - different defines depending on debug/release, platform, compiler, etc.
- We really want to automate these steps

Linking

- Manually?

```
ld -o myapp file1.o file2.o file3.o -lc -lmylib
```

- Unfeasible if we have many files, or if dependencies depend on the platform we are working on, etc.
- We also want to automate this step

Distribute your software

- Traditional way of doing things:
  - Developers develop code
  - Once the software is finished, other people package it
  - There are many packaging formats depending on operating system version, platform, Linux distribution, etc.
- We'd like to automate this but, is it possible to bring packagers into the development process?

Testing

- We like to use unit tests when developping software
- When and how to run unit tests? Usually a three step process:
  - manually invoke the build process (e.g. make)
  - when finished, manually run a test suite
  - when finished, look at the results and search for errors and/or warnings
  - can we test the packaging? Do we need to invoke the individual tests or the unit test manually?

Outline

- Build systems tour
- Meeting CMake
- Basic CMake usage
- Calling C++ from Matlab

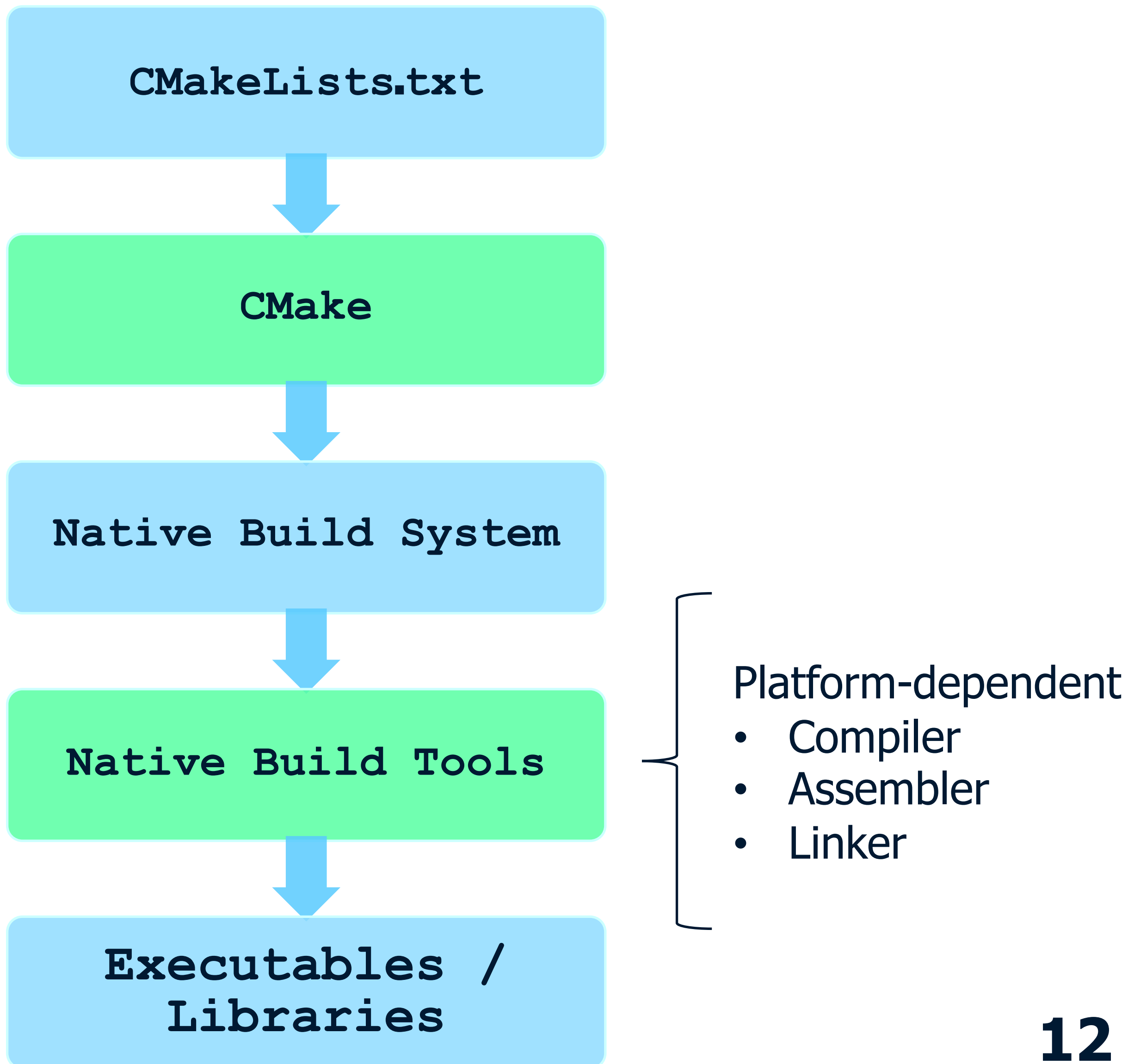
What is Cmake?

- CMake:
  - Generates native build environments
  - Supports multiple platforms
    - UNIX/Linux->Makefiles
    - Windows->VSProjects/Workspaces
    - Apple->Xcode
  - Open-Source
  - **Cross-Platform**

CMake features

- Manage complex, large build environments (KDE4)
  - Very Flexible & Extensible
  - Support for Macros
  - Modules for finding/configuring software (bunch of modules already available)
  - Extend CMake for new platforms and languages
  - Create custom targets/commands
  - Run external programs
- Very simple, intuitive syntax
- Support for regular expressions (*nix style), support for “In-Source” and “Out-of-Source” builds, and cross compilation
- Integrated Testing & Packaging (Ctest, CPack)

Build-system generator



CMake basic concepts

- `CMakeLists.txt`
  - Input text files that contain the project parameters and describe the flow control of the build process in a simple language (CMake language)
- CMake Modules
  - Special cmake files written for the purpose of finding a certain piece of software and to set it's libraries, include files and definitions into appropriate variables so that they can be used in the build process of another project. (e.g. `FindJava.cmake`, `FindZLIB.cmake`, `FindQt4.cmake`)

CMake basic concepts

- The **Source Tree** contains:
  - CMake input files (**`CMakeLists.txt`**)
  - Program source files (**`hello.cpp`**)
  - Program header files (**`hello.hpp`**)
- The **Binary Tree** contains:
  - Native build system files (**`Makefiles`**)
  - Output from build process:
    - Libraries
    - Executables
    - Any other build generated file
- Source and binary trees may be:
  - In the same directory (**in-source** build)
  - In different directories (**out-of-source** build)

CMake basic concepts

- **`CMAKE_MODULE_PATH`**
  - Path to where the CMake modules are located
- **`CMAKE_INSTALL_PREFIX`**
  - Where to put files when calling **`'make install'`**
- **`CMAKE_BUILD_TYPE`**
  - Type of build (**`Debug, Release, ...`**)
- **`BUILD_SHARED_LIBS`**
  - Switch between shared and static libraries

CMake basic concepts

- Variables can be changed directly in the build files (**`CMakeLists.txt`**) or through the command line by prefixing a variable's name with **`'-D'`**:
  - **`cmake -DBUILD_SHARED_LIBS=OFF`**
- A GUI is also available: ccmake

The CMake workflow

- Create a build directory ("out-of-source-build" concept)
  - **`mkdir build ; cd build`**
- Configure the package for your system:
  - **`cmake [options] <source_tree>`**
- Build the package:
  - **`make`**
- Install it
  - **`make install`**
- The last 2 steps can be merged into one (just "**`make install`**")

Simple executable

- **`PROJECT( helloworld )`**
- **`SET( hello_SRCS hello.cpp )`**
- **`ADD_EXECUTABLE( hello ${hello_SRCS} )`**

- **`PROJECT`** is not mandatory but should be used
- **`ADD_EXECUTABLE`** creates an executable from the listed sources
- Typically: add sources to a list (**`hello_SRCS`**), do not list them in **`ADD_EXECUTABLE`**.

Simple library

- **`PROJECT( mylibrary )`**
- **`SET( mylib_SRCS library.cpp )`**
- **`ADD_LIBRARY( my SHARED ${mylib_SRCS} )`**

- **`ADD_LIBRARY`** creates an static library from the listed sources
- Add **`SHARED`** to generate shared libraries (Unix) or dynamic libraries (Windows)

Shared vs static libs

- Static libraries: upon linking, adds the used code to your executable
- Shared/Dynamic libraries: upon linking, tell the executable where to find some code it needs
- If you build shared libs in C++, you should also use so-versioning to state binary compatibility
  (too long to be discussed here)

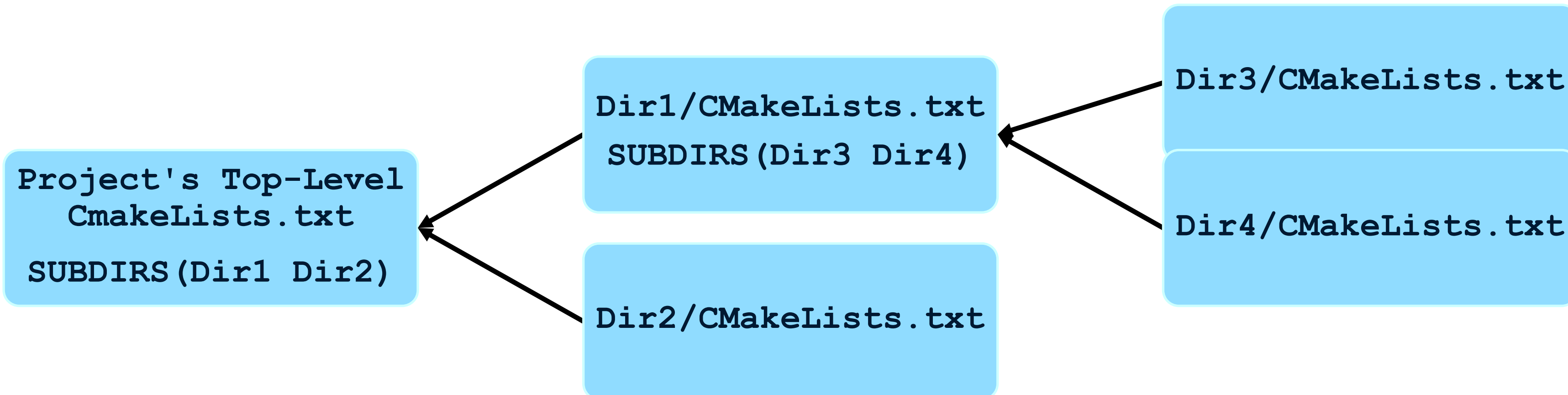
Showing verbose info

- To see the command line CMake produces:
  - `SET(CMAKE_VERBOSE_MAKEFILE on)`
- Or:
  - `$make VERBOSE=1`
- Or:
  - `$export VERBOSE=1`
  - `$make`
- **Tip:** only use it if your build is failing and you need to find out why

The CMake cache

- Created in the build tree (`CMakeCache.txt`)
- Contains Entries `VAR:TYPE=VALUE`
- Populated/Updated during configuration phase
- Speeds up build process
- Can be re-initialized with `cmake -C <file>`
- GUI can be used to change values
- There should be no need to edit it manually

Source tree structure



- Subdirectories added with **`SUBDIRS/ADD_SUBDIRECTORY`**
- Child inherits from parent (feature that is lacking in traditional **`Makefiles`**)
- Order of processing: **`Dir1;Dir3;Dir4;Dir2`** (When CMake finds a **`SUBDIR`** command it stops processing the current file immediately and goes down the tree branch)

Outline

- Build systems tour
- Meeting CMake
- Basic CMake usage
- Calling C++ from Matlab

Adding other sources

- clockapp
  - `build`
  - `trunk`
    - `doc`
    - `img`
    - `libwakeup`
      - `wakeup.cpp`
      - `wakeup.hpp`
    - `clock`
      - `clock.cpp`
      - `clock.hpp`

```
PROJECT(clockapp)
ADD_SUBDIRECTORY(libwakeup)
ADD_SUBDIRECTORY(clock)
```

```
SET(wakeup_SRCS wakeup.cpp)
ADD_LIBRARY(wakeup SHARED
${wakeup_SRCS})
```

```
SET(clock_SRCS clock.cpp)
ADD_EXECUTABLE(clock $
{clock_SRCS})
```

Variables

- No need to declare them
- Usually, no need to specify type
- `SET` creates and modifies variables
- `SET` can do everything but `LIST` makes some operations easier
- Use `SEPARATE_ARGUMENTS` to store space separated arguments (i.e. a string) into a list (semicolon-separated)

Changing build parameters

- CMake uses common, sensible defaults for the preprocessor, compiler and linker
- Modify preprocessor settings with **`ADD_DEFINITIONS`** and **`REMOVE_DEFINITIONS`**
- Compiler settings: **`CMAKE_C_FLAGS`** and **`CMAKE_CXX_FLAGS`** variables
- **Tip:** some internal variables **`(CMAKE_*)`** are read-only and must be changed executing a command

Debug and release builds

- `SET(CMAKE_BUILD_TYPE Debug)`
- As any other variable, it can be set from the command line:
  - `cmake -DCMAKE_BUILD_TYPE=Release ../trunk`
- Specify debug and release targets and 3rdparty libs:
  - `TARGET_LINK_LIBRARIES(wakeup RELEASE ${wakeup_SRCS})`
  - `TARGET_LINK_LIBRARIES(wakeupd DEBUG ${wakeup_SRCS})`

Find installed software

- **`FIND_PACKAGE( xxx REQUIRED )`**
- CMake includes finders **`(FindXXXX.cmake)`** for around 130 software packages, many more are available on the internet
- If using a non-CMake **`FindXXXX.cmake`**, tell CMake where to find it by setting the **`CMAKE_MODULE_PATH`** variable
- Think of **`FIND_PACKAGE`** as an **`#include`**

Outline

- Build systems tour
- Meeting CMake
- Basic CMake usage
- Calling C++ from Matlab

Matlab

- MATLAB (by Mathworks) is a good development platform for prototyping
- It is heavily optimized for vector operations
  - **<u>Good</u>** for fast calculations on vectors and matrices
  - **<u>Bad</u>** if you can not state your problem as a vector problem
    - Slow implementations of sequential programs!
    - Especially, for-loops are slow

What are MEX files?

- MEX stands for "MATLAB Executable"
- MEX-files are a way to call your custom C or FORTRAN routines directly from MATLAB as if they were MATLAB built-in functions
- MEX-files can be called exactly like M-functions in MATLAB
- MEX-files are basically just special C or C++ files that are compiled from within Matlab, thus making them callable

Reasons for MEX-files

- The ability to call large existing C or FORTRAN routines directly from MATLAB without having to rewrite them as M-files
- Speed
  - You can rewrite bottleneck computations (like for-loops) as a MEX-file for efficiency!

Components of a MEX-file

- A gateway routine, `mexFunction`, that interfaces C and MATLAB data
  - Analoguous to the `main`-function for regular programs started from the console
- A computational routine, called from the gateway routine, that performs the computations that the MEX-file should implement
- **Preprocessor macros, for building platform independent code**

The gateway routine

- The name of the gateway routine must be `mexFunction`
- Parameters:
    - `prhs`: An array of right-hand input arguments
    - `plhs`: An array of left-hand output arguments
    - `nrhs`: The number of right-hand arguments, or the size of the `prhs` array
    - `nlhs`: The number of left-hand arguments, or the size of the `plhs` array

The gateway routine

```
void mexFunction(
     int nlhs, mxArray *plhs[],
     int nrhs, const mxArray *prhs[]) {
  /* more C code ... */
}
```

The computational routine

- The computational routine is called from the gateway routine
- It is a good idea to place the computational routine in a separate subroutine although it can be included in the gateway routine as well

Some important points

- The parameters `prhs`, `plhs`, `nrhs` and `nlhs` are required
- The header file, `mex.h`, that declares the entry point and interface routines is also required
- The name of the file with the gateway routine will be the command name in MATLAB
- The file extension of the MEX-file is platform dependent
    - The `mexext` function returns the extension for the current machine

An extra important point

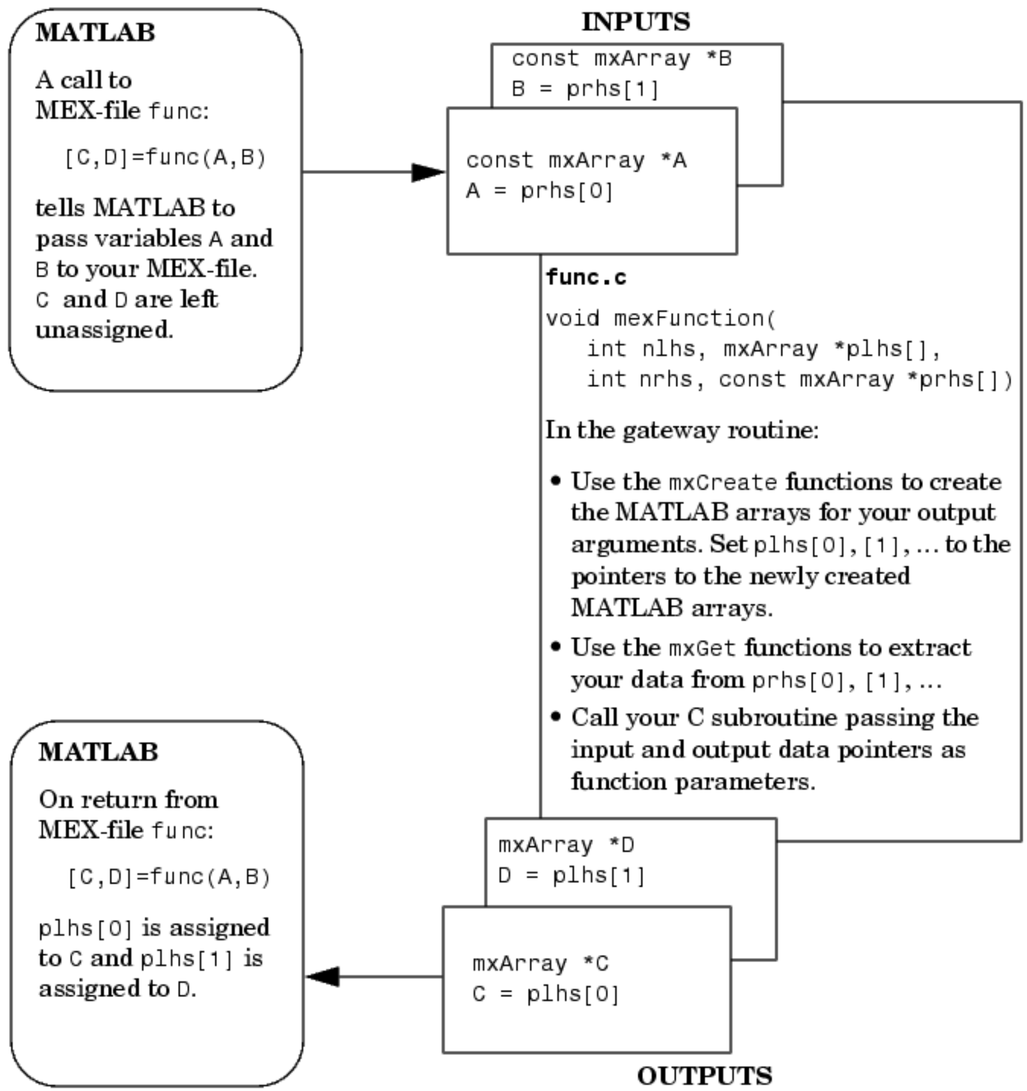
- MATLAB indices are 1-based, and C-indices are 0-based!

Input and output, I/O

- `[C,D] = func(A,B)`
- Get pointers to A and B
  - `mxGetPr(prhs[0])`
  - `mxGetPr(prhs[1])`
- Allocate memory for C and D
  - `mxCreate*` (NumericArray, DoubleMatrix, ...)
- Get pointers to C and D
  - `mxGetPr(plhs[0])`
  - `mxGetPr(plhs[1])`

I/O

- Overview of the communication between MEX and MATLAB



I/O Example

- Program that:
  - Takes a matrix as input
  - Performs some operations on it
  - Returns a matrix of same size
  - Types are uint16

I/O Example

```
unsigned short *pind_l,*pind_r;
int number_of_dims, nelem;
const int *dim_array;
...

number_of_dims = mxGetNumberOfDimensions(prhs[0]);
dim_array = mxGetDimensions(prhs[0]);
nelem = mxGetNumberOfElements(prhs[0]);

plhs[0] = mxCreateNumericArray(
    number_of_dims, dim_array, mxUINT16_CLASS, mxREAL);

pind_l = (unsigned short *) mxGetPr(plhs[0]);
pind_r = (unsigned short *) mxGetPr(prhs[0]);

/* call computational routine */
myfunction(pind_l,pind_r, ... );
```

mxArray

- All scalars, vectors, matrices etc. in MATLAB are represented in mxArrays
  - **mxCreateCellArray**
  - **mxCreateCellMatrix**
  - **mxCreateCharArray**
  - **mxCreateCharMatrixFromStrings**
  - **mxCreateDoubleMatrix**
  - **mxCreateDoubleScalar**
  - **mxCreateLogicalArray**
  - **mxCreateLogicalMatrix**
  - **mxCreateLogicalScalar**
  - **mxCreateNumericArray**
  - **mxCreateNumericMatrix**
  - **mxCreateSparse**
  - **mxCreateSparseLogicalMatrix**
  - **mxCreateString**
  - **mxCreateStructArray**
  - **mxCreateStructMatrix**

mxArray suitable for images

- 2D
  - `mxCreateNumericArray`
  - `mxCreateDoubleMatrix`
- 3D
  - `mxCreateNumericArray`
- Indexing
  - Column wise, as in MATLAB (column major)

0	3	6
1	4	7
2	5	8

mx and mex

- Routines in the API that are prefixed with `mx` allow you to create, access, manipulate, and destroy mxArrays
- Routines prefixed with `mex` perform operations back in the MATLAB environment
  - Note: mex routines are only available in MEX-functions

Checking of input types

- Input arguments needs to be checked
  - `mxIsDouble, mxIsUint16, ...`

```
/* Check data type of input argument. */
if (!(mxIsUint16(prhs[0]))) {
  mexErrMsgTxt("Input array must be of type uint16.");
}
```

- `mexErrMsgTxt` prints error in Matlab

2D

```c
#include "mex.h"

void xtimesy(double x, double *y, double *z, int m, int n)
{ ... }

/* the gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[]) {
  double *y,*z;
  double x;
  int status,mrows,ncols;

  /* check for proper number of arguments */
  if(nrhs!=2)
    mexErrMsgTxt("Two inputs required.");
  if(nlhs!=1)
    mexErrMsgTxt("One output required.");

  /* check to make sure first input argument is scalar */
  if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||
      mxGetN(prhs[0])*mxGetM(prhs[0])!=1 ) {
    mexErrMsgTxt("Input x must be a scalar.");
  }
```

2D

```
  ...

  /* get the scalar input x */
  x = mxGetScalar(prhs[0]);

  /* create a pointer to the input matrix y */
  y = mxGetPr(prhs[1]);

  /* get the dimensions of the matrix input y */
  mrows = mxGetM(prhs[1]);
  ncols = mxGetN(prhs[1]);

  /* set the output pointer to the output matrix */
  plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);

  /* create a C pointer to a copy of the output matrix */
  z = mxGetPr(plhs[0]);

  /* call the C subroutine */
  xtimesy(x,y,z,mrows,ncols);
}
```

3D

```c
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]) {
  unsigned int *out_reg, *in_reg;
  int number_of_dims, nelem, zMax;
  const int *dim_array;

  if (!(mxIsUint32(prhs[0]))) {
    mexErrMsgTxt("First input array must be of type uint32.");
  }

  number_of_dims = mxGetNumberOfDimensions(prhs[0]);
  dim_array = mxGetDimensions(prhs[0]);

  zMax = dim_array[2];

  plhs[0] = mxCreateNumericArray(number_of_dims,dim_array,
                                 mxUINT32_CLASS, mxREAL);

  out_reg = (unsigned int *) mxGetPr(plhs[0]);

  in_reg = (unsigned int *) mxGetPr(prhs[0]);
  nelem = mxGetNumberOfElements(prhs[0]);

  remove_function(out_reg,in_reg,
                  nelem,dim_array[1],dim_array[0],zMax);
}
```

Compiling MEX-files

- Compile your mex function on the MATLAB command line using the mex command
  - `mex myfunc.c`
- Easy compilation of required files by adding them on the command line
  - `mex myfunc.c special.cpp`
- Compile outside MATLAB in your favourite development environment

Output files

- .dll
- .mexa64
- .mexw32
- .mexglx

Setting up the environment

- mex –setup
  - Choose compiler (see example)
  - Default is a C compiler. Add others for C++.
- Compatible compilers are listed on Mathworks webpage

mex -setup

```
>> mex -setup
Please choose your compiler for building external interface (MEX)
    files:
Would you like mex to locate installed compilers [y]/n?

Select a compiler:
[1] Lcc C version 2.4.1 in C:\PROGRAM FILES\MATLAB\R2006A\sys\lcc
[2] Microsoft Visual C/C++ version 8.0 in C:\Program
     Files\Microsoft Visual Studio 8
[3] Microsoft Visual C/C++ version 7.1 in C:\Program
     Files\Microsoft Visual Studio .NET 2003
[0] None
```

Calling MEX-functions

- You can call MEX-files exactly as you would call any other M-function
- If you call a MATLAB function the current working directory and then the MATLAB path is checked for the M- or MEX-function
- MEX-files take precedence over M-files when like-named files exist in the same directory
- Help text documentation is read from the .m file with same name as the MEX-file. Add your usage tips in the .m file

Linking of MEX-functions

- Easy to link against other C/C++ libraries
  - → Use MEX-function to as a wrapper to the library!

- Syntax:
  - `mex -I../include -I../AnotherIncludePath -L../SomePathToLibrary/lib -lsomelib myMexFunction.cpp`

Examples: OpenGV wrapper

- OpenGV-wrapper:
  - See https://github.com/laurentkneip/opengv/blob/master/matlab/opengv.cpp

Treating an mxArray as an Eigen matrix!

- Use:

```
Map<Matrix<typename Scalar,int RowsACT,int ColsACT> >
```

- What it does:
  - Wraps around existing data so we can treat it like an Eigen matrix
  - Does not require copying the data (only a header)
  - Takes
    - A pointer to the data in memory
    - Possibly size parameters (if dynamically sized)

Treating an mxArray as an Eigen matrix!

- Example:

```
//prhs[0] is mxArray of doubles with n rows and m columns

//get pointer to the data
double * inputPtr1 = (double *) mxGetPr(prhs[0]);

//get dimensions of data
const int * dim_array1 = mxGetDimensions(prhs[0]);

//…

//init Eigen Map to treat column i as an Eigen::VectorXd
Eigen::Map<Eigen::VectorXd> eigenMatrix(
    &(inputPtr1[dim_array1[0]*i]),dim_array1[0]);
```

- Both mxArray and Eigen are column major!!

Pros and cons

- Pros:
  - Fast calculations
  - Easy to learn and use
- Cons:
  - Slow implementation compared to M-files
  - Platform dependent implementation
    (need to recompile MEX-files for every platform!)