Matplotlib for C++

Aug 01, 2019

Content

This is the documentation to *Matplotlib for C++*, a C++ wrapper for Python's *matplotlib* (MPL) plotting library.

CHAPTER 1

A short overview

1.1 The GitHub repo

The code is organised in this GitHub repository, which is a fork of that repository.

1.2 Purpose

This is: A lightweight, easy-to-use interface to create stylish and clean plots in C++ using basic MPL commands.

This is **not**: A translation of MPL to C++.

1.2.1 Compiling a program

Requirements

Matplotlib for C++ requires a working Python installation as well as Matplotlib. Python2.7 and Python3 (>= 3.6) have been tested, but other versions should work as well. In the linking process the exact version of Python to use can be specified by linking the according library.

On Unix it is recommended to install Python via the package manager to assert that all dependencies are installed properly.

```
<package-manager> install python3 python3-dev  # or -devel depending on the platform
```

If Python is installed from source problems in the linking may occur. How to resolve these is explained in the next section, or in *this* code-block.

Install matplotlib via pip

```
pip3 install matplotlib  # or pip for Python 2
```

Includes and Linking

The header `matplotlibcpp.h` depends on the Python header, `Python.h`, the corresponding Python library `libpython`, and on `numpy/arrayobject.h`. If not in the standard include paths, the paths to the header files, the path to the library, and the library itself have to be specified for the compiler using the options `-I`, `-L` and `-l` respectively. Note, that all Python constituents should be of the same Python version. Matplotlib for C++ supports both, Python 2.7 and Python 3 versions.

In detail:

- The Python header `Python.h`

  The Python header comes with the Python installation. It it cannot be found on your system try installing the Python development packages. The location of this header has to be specified using the option `-I`.

  Typical locations:

  - Linux: */usr/local/include/python3.7*
  - Mac: if installed with Homebrew */usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/include/python*

- The Python library `libpython*.so`

  The program has to be linked against the compiled Python library. Depending on the Python version the name of the library differs, for Python 3.7 it is `libpython3.7.so` (or `libpython3.7m.so`). Then link the library by specifying `-lpython3.7` (or `-lpython3.7m`).

  Additionally to the linking the location of the library must be specified if not installed in the usual directory. For Linux systems this is usually not necessary, for Mac however it mostly is. The location of the library has to be specified using the option `-L`.

  If Python has not been installed using the package manager (but e.g. from source) twofold problems with linking the library can occur. The first are missing dependencies of the Python library, these can be added via `-lpthread -lutil -ldl`. The second is that dynamic libraries have to be exported which is resolved by adding `-Xlinker -export-dynamic`.

  Typical locations:

  - Linux: Path usually already included
  - Mac: */usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib*

- Numpy array `numpy/arrayobject.h`

  By default Matplotlib for C++ uses Numpy arrays. This requires the above header file. However it is possible to avoid this header by defining `-DWITHOUT_NUMPY`.

  - Linux: */usr/local/lib/python3.7/site-packages/numpy/core/include*
  - Mac: If installed via Homebrew, same as for Linux.

Examples

On Linux with the GNU compiler `g++` and C++11.

```
# using Python 2.7
g++ main.cpp -std=c++11 -I/usr/local/include/python2.7 \
  -I/usr/local/lib/python2.7/site-packages/numpy/core/include -lpython2.7
```

```
# using Python3.7 and no Numpy
g++ main.cpp -std=c++11 -DWITHOUT_NUMPY -I/usr/local/include/python2.7 -lpython2.7
```

On Mac with the GNU compiler `g++` and C++14.

```
g++ main.cpp -std=c++14 \
 -I /usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/include/
↪python3.7m \
 -I /usr/local/lib/python3.7/site-packages/numpy/core/include \
 -L /usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib \
 -lpython3.7
```

With exporting dynamic libraries and linking to all dependencies of the Python library on a Linux system:

```
g++ main.cpp -std=c++11 -I/usr/local/include/python3.7m \
  -I/usr/local/lib/python3.7/site-packages/numpy/core/include \
  -lpython3.7m \
  -lpthread -lutil -ldl \ # library dependencies
  -Xlinker -export-dynamic \ # export dynamic libraries
```

1.2.2 The Docs

The formatting string

Bla

The *Vector* type

type Vector

Functions in the Matplotlib-C++ library are designed to work with a generic vector type where possible. All template types named *Vector** must support the following operations. See the STL vector documentation for more detail on the implementation.

Note: Check the declarations with the STL doc

typedef double **value_type**

Definition of the underlying type, *double* may be replaced with another suitable type.

std::size_t **size** ()

Return the size of the vector.

value_type **operator[]** (**const** std::size_t *i*)

value_type **at** (**const** std::size_t *i*)

Return the *i* th element of the vector.

value_type ***data** ()

Return a pointer to the first element of the data in the vector. The data must furthermore be stored in a consecutive manner.

value_type ***begin** ()

Return a pointer to the first element of the data in the vector.

value_type ***end** ()

Return a pointer directly behind the last element of the data in the vector.

Plot commands

template<typename **VectorX**, typename **VectorY**>
bool **plot** (**const** *VectorX* &*x*, **const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = {})

Plot *y* versus *x*.

The two vectors x and y must have the same length. The formatting string *s* can specify the colour, markers and style of the line. The map *keywords* may contain additional named arguments for the plot.

Template Parameters

- **VectorX** – vector-like type, see `Vector`
- **VectorY** – vector-like type, see `Vector`

Parameters

- **x** – x data for the plot
- **y** – y data for the plot
- **s** – (optional) formatting string, see *here*
- **keywords** – (optional) map specifying additional keywords, see here

Returns true if no error has occured, false otherwise

Minimal working example

```
#include <vector>
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;

int main() {
  std::vector<double> x = {1, 2, 3, 4};
  std::vector<double> y = {1, 4, 9, 16};

  plt::plot(x, y);
  plt::show();

  return 0;
}
```

Example with formatting strings

```
plt::plot(x, y, "r*");  // Red stars as markers, no line
```

```
plt::plot(x, y, "bo-");  // Blue dots + blue line
```

Example with keywords

```
plt::plot(x, y, "bo-", {{"label", "f(x)"}});  // add the label f(x)
plt::legend();                                // remember to activate the legend
```

```
plt::plot(x, y, {{"label", "$y = x^2$"}});  // latex is supported
plt::legend();
```

template<typename **VectorY**>

bool **plot** (**const** *VectorY* &*y*, **const** std::string &*format* = "", **const** std::map<std::string, std::string> &*keywords* = {})

Plot *y*.

For a vector y of size n, the x data is set to $0, ..., n-1$. The formatting string *s* can specify the colour, markers and style of the line. The map *keywords* may contain additional named arguments for the plot.

Template Parameters **VectorY** – vector-like type, see `Vector`

Parameters

- **y** – y data for the plot
- **s** – (optional) formatting string, see *here*
- **keywords** – (optional) map specifying additional keywords, see here

Returns true if no error has occured, false otherwise

Examples

```
#include <vector>
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;

int main() {

  std::vector<int> y = {1, 2, 3};
  plt::plot(y, "bo-");
  plt::show();

  return 0;
}
```

```
Eigen::VectorXd y = {1, 2, 3};
plt::plot(y, {{"label", "1 to 3"}});
plt::show();
```

template<typename **VectorX**, typename **VectorY**>
bool **loglog** (**const** *VectorX* &*x*, **const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = {})

Plot *y* versus *x* in double logarithmic scale.

See `plot()` for explanation of the parameters.

Note: All following plots will be in double logarithmic scale, also calls to *plot*.

Example

```
#include <Eigen/Dense>
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;

int main() {
```

(continues on next page)

(continued from previous page)

```
  int n = 5000;
  Eigen::VectorXd x(n), y(n), z(n), w = Eigen::VectorXd::Ones(n);
  for (int i = 0; i < n; ++i) {
    double value = (1.0 + i) / n;
    x(i) = value;
    y(i) = value * value;
    z(i) = value * value * value;
  }

  plt::loglog(x, y);          // f(x) = x^2
  plt::loglog(x, w, "r--");  // f(x) = 1, red dashed line
  plt::loglog(x, z, "g:", {{"label", "$x^3$"}}); // f(x) = x^3, green dots + label

  plt::title("Some functions of $x$"); // add a title
  plt::show();
}
```

template<typename **VectorY**>
bool **loglog** (**const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = { })

Plot *y* in double logarithmic scale.

See *plot()* for explanation of the parameters.

Note: All following plots will be in double logarithmic scale, also calls to *plot*.

Examples

Assuming `vector` and `matplotlibcpp` import and the namespace definition `plt = matplotlibcpp`.

```
std::vector<int> y = {1, 10, 100, 1000};
plt::loglog(y);
```

```
std::vector<double> y1 = {1, 2, 4},
                    y2 = {1, 3, 9};
plt::loglog(y, "bo-", {{"label", "powers of 2"}});
plt::plot(y, "ro-", {{"label", "powers of 3"}});  // also in loglog scale
```

template<typename **VectorX**, typename **VectorY**>
bool **semilogx** (**const** *VectorX* &*x*, **const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = { })

Plot *y* versus *x* in logarithmic *x* and linear *y* scale.

See *plot()* for explanation of the parameters.

Note: All following plots will inherit the logarithmic *x* scale, also calls to *plot*.

template<typename **VectorY**>

bool **semilogx** (**const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = { })

Plot *y* in logarithmic *x* and linear *y* scale.

See `plot()` for explanation of the parameters.

Note: All following plots will inherit the logarithmic *x* scale, also calls to *plot*.

template<typename **VectorX**, typename **VectorY**>
bool **semilogy** (**const** *VectorX* &*x*, **const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = { })

Plot *y* versus *x* in linear *x* and logarithmic *y* scale.

See `plot()` for explanation of the parameters.

Note: All following plots will inherit the logarithmic *y* scale, also calls to *plot*.

template<typename **VectorY**>
bool **semilogy** (**const** *VectorY* &*y*, **const** std::string &*s* = "", **const** std::map<std::string, std::string> &*keywords* = { })

Plot *y* in linear *x* and logarithmic *y* scale.

See `plot()` for explanation of the parameters.

Note: All following plots will inherit the logarithmic *y* scale, also calls to *plot*.

template<typename **Numeric**>
void **text** (*Numeric* *x*, *Numeric* *y*, **const** std::string &*s* = "")

Place text at location (x, y).

Template Parameters **Numeric** – A scalar-like type

Parameters

- **x** – The x location of the text
- **y** – The y location of the text
- **s** – The text to be placed in the plot

Example

```
#include <vector>
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;

int main() {
```

(continues on next page)

(continued from previous page)

```
    std::vector<double> x = {0.1, 0.2, 0.5};
    plt::plot(x, "s");
    plt::text(1.0, 0.1, "Text under a square");
    plt::show();

    return 0;
}
```

Figure commands

inline long **figure** (long *number* = -1)

Initialise a new figure with the ID *number*.

Parameters **number** – The number of the figure. If set to *-1* default numbering (increasing from *0* on) is used.

Returns The number of the figure.

inline bool **fignum_exists** (long *number*)

Check if a figure of given number exists.

Parameters **number** – The number of the figure

Returns true, if a figure with given number exists, false otherwise

inline void **figure_size** (size_t *w*, size_t *h*)

Set the figure size to *w* x *h* inches.

Parameters

- **w** – The width of the figure in inches
- **h** – The height of the figure in inches

template<typename **Vector** = std::vector<double>>
inline void **legend** (**const** std::string &*loc* = "best", **const** *Vector* &*bbox_to_anchor* = *Vector*())

Set the figure legend.

Template Parameters **Vector** – vector-like type, see `Vector`, defaults to *std::vector<double>*

Parameters

- **loc** – The location of the legend. May be any of: "best", "upper left", "upper center", "upper left", "center left", "center", "center right" (== "right"), "lower left", "lower center", "lower right"
- **bbox_to_anchor** –

  If set to a vector of length 2 or 4 it specifies the location (and size) of the legend's bounding box. Format is (*x*, *y*) or (*x*, *y*, *width*, *height*). The coordinates are interpreted in the same units as the plot axes (thus no normalised coordinates)

```
// Put the legend in the center of the bottom right quadrant.
// First argument: loc, second: bbox_to_anchor
plt::legend("center", {0.5, 0, 0.5, 0.5});
```

template<typename **Numeric**>
void **xlim** (*Numeric left*, *Numeric right*)

Set the *x* axis limits.

Template Parameters **Numeric** – A scalar-like type

Parameters

- **left** – The left axis limit
- **right** – The right axis limit

template<typename **Numeric**>
void **ylim** (*Numeric bottom*, *Numeric top*)

Set the *y* axis limits.

Template Parameters **Numeric** – A scalar-like type

Parameters

- **bottom** – The bottom axis limit
- **top** – The top axis limit

inline double ***xlim** ()
Get the *x* axis limits.

Returns A pointer to an array of length 2 containing *[left, right]*

inline double ***ylim** ()
Get the *y* axis limits.

Returns A pointer to an array of length 2 containing *[bottom, top]*

inline void **title** (**const** std::string &*titlestr*, **const** std::map<std::string, std::string> &*keywords* = {})

Set the title of the plot.

Parameters

- **titlestr** – Title of the plot
- **keywords** – Additional keywords, see here for a list

inline void **suptitle** (**const** std::string &*suptitlestr*, **const** std::map<std::string, std::string> &*keywords* = {})

Add a centered title to the figure.

Parameters

- **suptitlestr** – Title of the figure
- **keywords** – Additional keywords, see here for a list

inline void **axis** (**const** std::string &*option*)

Set some axis properties.

Parameters **`option`** – The option to activate

option	Result
on	Turn on axis lines and labels
off	Turn off axis lines and labels
equal	Set equal scaling (i.e., make circles circular) by changing axis limits.
scaled	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box.
tight	Set limits just large enough to show all data.
auto	Automatic scaling (fill plot box with data).
image	*scaled* with axis limits equal to data limits.
square	Square plot; similar to *scaled*, but initially forcing same x- and y-axis length.

1.2.3 License

1.2.4 Questions

See here https://github.com/Cryoris/matplotlib-cpp for the fork this documentation is based on, or https://github.com/lava/matplotlib-cpp for the parent repository.

1.2.5 To do

Generalise to Vector

a lot of functions

CHAPTER 2

How to use this documentation

2.1 Function definitions

This is the core of the documentation, located at *The Docs*. To find the definition and explanations for a special command use the search field on the top left, since this page can get a bit lengthy.

Bear in mind, that *matplotlibcpp* is a C++ wrapper to the Python library MPL. Thus, to learn more about the functions that are eventually called the matplotlib documentation might be useful. Most functions have a link to the MPL function they call, marked with the MPL logo:

However, the function signatures might differ and Matplotlib for C++ does *not* support the full functionality of MPL. The purpose is providing an easy-to-use wrapper to MPL in C++, not to fully translate the library.

2.2 Help for compiling

The section *Compiling a program* explains the compilations of a program using the `matplotlibcpp.h` header.

Tip: Criticism (preferably constructive), ideas and contributions are welcome! For contact, see *Questions*.

Index