

---

# Matplotlib for C++

Mar 23, 2020



---

## Content

---

<b>1</b>	<b>How to use this documentation</b>	<b>3</b>
1.1	Function definitions . . . . .	3
1.2	Help for compiling . . . . .	3
1.3	Style of a line . . . . .	3
1.4	Frequent problems . . . . .	3
	<b>Index</b>	<b>25</b>



This is the documentation to *Matplotlib for C++*, a C++ wrapper for Python's *matplotlib* (MPL) plotting library. The code is organised in [this](#) GitHub repository, which is a fork of [that](#) repository.

---

**Note:** *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++.

---



---

## How to use this documentation

---

### 1.1 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.

### 1.2 Help for compiling

The section [Compiling a program](#) explains the compilations of a program using the `matplotlibcpp.h` header.

### 1.3 Style of a line

Refer [The style of a line](#) to tune the appearance of the lines you plot.

### 1.4 Frequent problems

[Frequent problems](#) lists typical problems with Matplotlib for C++ and (hopefully) how to resolve them.

---

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

---

### 1.4.1 First examples

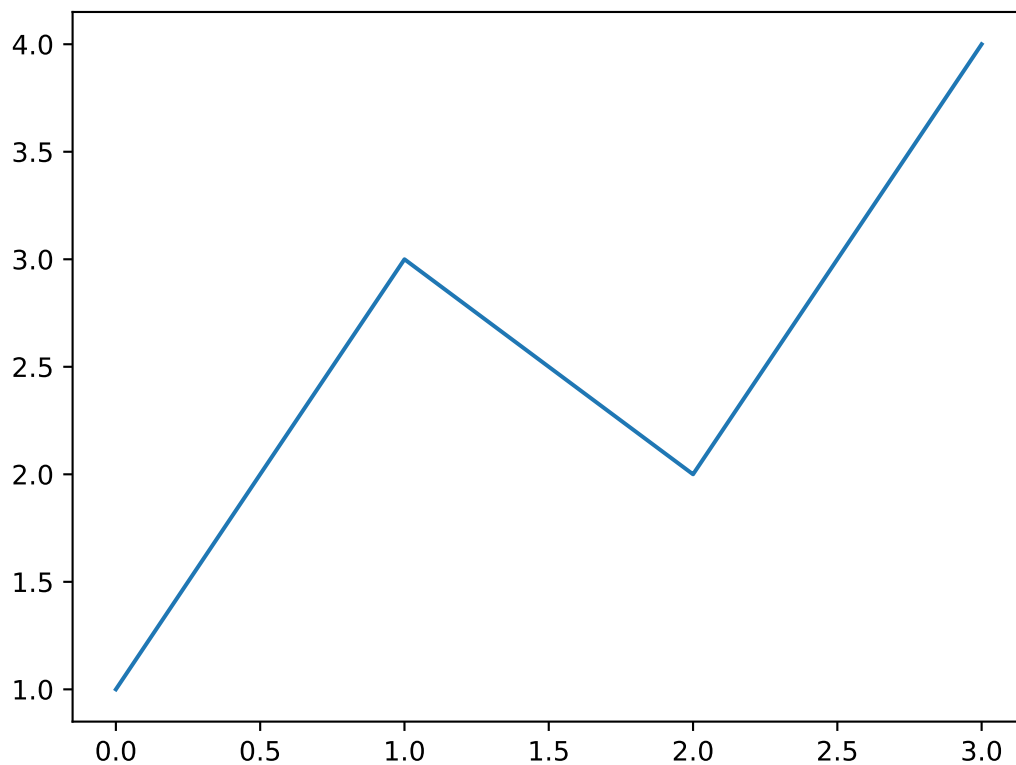
#### Minimal working example

```
#include "matplotlibcpp.h"
#include <vector>

namespace plt = matplotlibcpp;

int main() {
    std::vector<double> y = {1, 3, 2, 4};
    plt::plot(y);
    plt::savefig("minimal.pdf");
}
```

produces the output





## Standard usage

```
#define _USE_MATH_DEFINES // for sin/log
#include "../matplotlibcpp.h"
#include <cmath>
#include <iostream>

namespace plt = matplotlibcpp;

int main() {
    int n = 5000; // 5000 data points
    std::vector<double> x(n), y(n), z(n), w(n, 2);
    for (int i = 0; i < n; ++i) {
        x.at(i) = i * i;
        y.at(i) = sin(2 * M_PI * i / 360.0);
        z.at(i) = log(i);
    }

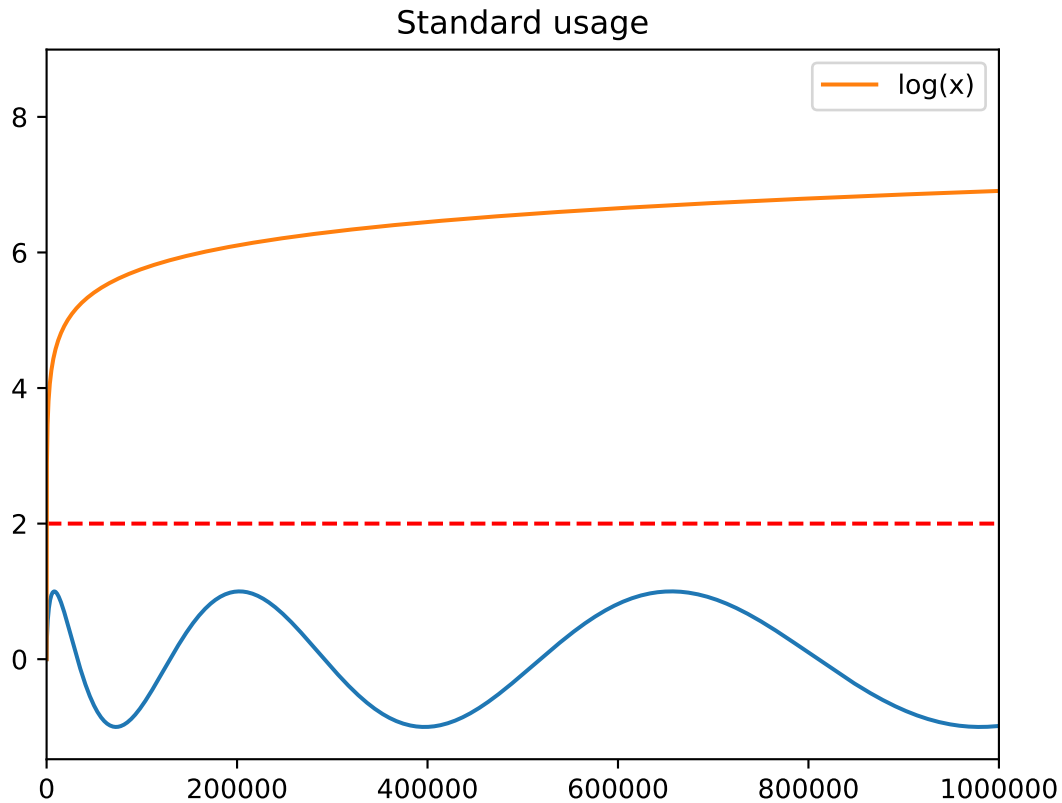
    plt::figure(); // declare a new figure (optional if only one is used)

    plt::plot(x, y); // automatic coloring: tab:blue
    plt::plot(x, w, "r--"); // red dashed line
    plt::plot(x, z, {"label", "log(x)"}); // legend label "log(x) "

    plt::xlim(0, 1000 * 1000); // x-axis interval: [0, 1e6]
    plt::title("Standard usage"); // set a title
    plt::legend(); // enable the legend

    plt::savefig("standard.pdf"); // save the figure
}
```

produces the output



## Eigen vectors

```
#include "../matplotlibcpp.h"
#include <Eigen/Dense>
#include <iostream>

namespace plt = matplotlibcpp;

int main() {

    Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(200, 0, 6);
    Eigen::VectorXd y, z;

    // y = exp(sin(x)), z = exp(cos(z))
    y = x.array().sin().exp().matrix();
    z = x.array().cos().exp().matrix();

    plt::figure();

    plt::semilogx(x, y, "tab:red");
    plt::semilogx(x, z, "tab:blue", {"linestyle", "--"});

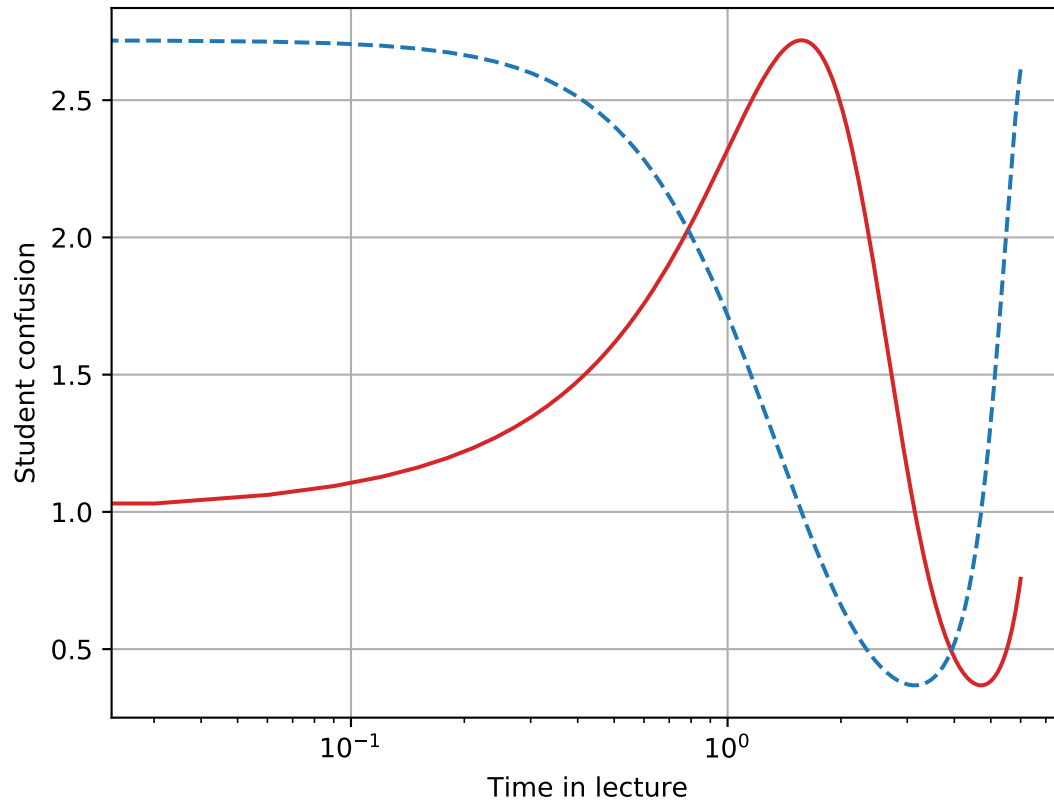
    plt::xlabel("Time in lecture");
    plt::ylabel("Student confusion");
```

(continues on next page)

(continued from previous page)

```
plt::grid();  
plt::show(); // show the figure instead of saving it  
}
```

produces the output



## 1.4.2 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. If 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
```

### 1.4.3 The Docs

#### *matplotlibcpp* namespace

All functions are organised in the namespace *matplotlibcpp*. For convenience (and in spirit of the Python norm) we usually define the abbreviation *plt*:

```
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;
```

The function can then be accessed via:

```
matplotlibcpp::plot(x, y);
plt::loglog(x, y); // if we defined namespace plt = matplotlibcpp
```

#### 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 occurred, 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  $0, \dots, n - 1$ . 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 occurred, 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() {
    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> &key-words = {})



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() {

    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*)

Call *plt::figure()* and set the figure size to *w* x *h* pixels.

**Parameters**

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

template<typename **Vector** = std::vector<double>>

**inline** void **legend** (const std::string &*loc* = "best", const *Vector* &*bbox\_to\_anchor* = *Vector*())



Enable 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 right”, “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)

### Example

```
// 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 supitle(const std::string &supitlestr, const std::map<std::string, std::string> &key-
    words = {})
```



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

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



Save the current figure.

Supported file types depend on the user backend, but usually contain *pdf*, *eps* and *png*. To find all supported formats try

```
$ python3
>>> import matplotlib.pyplot as plt
>>> plt.gcf().canvas.get_supported_filetypes_grouped()
```

**Parameters**

- **filename** – Save the figure to *filename* (must contain file format)
- **keywords** – Additional keywords, see *Other Parameters* [here](#) for a complete list

**Examples**

```
plt::plot(x, y);
plt::savefig("plot.pdf");
```

Always the current state of the figure is stored.

```
plt::plot(time, apple_sales);
plt::savefig("sales.pdf"); // contains only apple_sales
plt::plot(time, kiwi_sales);
plt::savefig("sales.pdf"); // contains apple and kiwi sales
```

Calling *plt::show()* clears the plot!

```
plt::plot(x, y);
plt::show();
plt::savefig("is_this_empty.pdf"); // yes, this will be empty

plt::plot(x, y);
plt::savefig("this_isnt_empty.pdf"); // always call savefig *before* show
plt::show();
```

Optimally use the available canvas space with `{{"bbox_inches", "tight"}}`. This can be useful if e.g. the axis labels are too far outside and get cut off.

```
plt::savefig("fig.pdf", {{"bbox_inches", "tight"}});
```

**inline** void **show**(const bool *block* = true)



Display the figure.

**Parameters** **block** – If true, the execution of the code is stopped until the displayed figure is closed. Otherwise the code is not stopped. Depending on the backend, figures might not get displayed at all.

### 1.4.4 The style of a line

The style of a line in a plot has three characteristics; the marker, the color and the line. They can be specified using the keywords map or the formatting string.

The keywords map is a possibility to specify additional parameters for the plot commands. To set the line style it can be used as follows.

```
// for a red dashed line with circle markers
plt::plot(x, y, {{"color", "red"}, {"marker": "o"}, {"linestyle": "--"}})

// shorthand notation for color and linestyle is usually supported
plt::plot(x, y, {{"c", "red"}, {"marker": "o"}, {"ls": "--"}})
```

See sections [Marker](#), [Color](#) and [Line](#) for supported values.

The formatting string is a convenient notation to set the style of a line. Almost all plot commands support the formatting string as first argument after the *x* and *y* data.

A formatting string's structure is

```
s = "<marker><color><line>"
```

The rules are

1. The valid parameters (<>) are listed in the following sections.
2. Different ordering of the parameters is supported. However, to avoid ambiguity this order should be used throughout.
3. Parameters might be dropped.
4. If only the colour parameters is specified, more color parameters are supported, see section [Color](#).

#### Examples

```
"b"      // blue line
"og"     // green circles, no connecting line
"og-"    // green circles with connecting line
"r--"    // dashed red line
"seagreen" // a solid line in the colour seagreen
"#008000" // a solid line coloured in the HEX code
```

The following sections list the supported markers, colours and linestyles.

### Marker

character	marker
o	circle
s	square
D	diamond
d	thin diamond
p	pentagon
H	hexagon type 1
h	hexagon type 2
v	triangle down
^	triangle up
<	triangle left
>	triangle right
*	star
+	plus
x	x
.	point
,	pixel
	vertical lines
_	horizontal lines
1	<i>tri_down</i> marker
2	<i>tri_up</i> marker
3	<i>tri_left</i> marker
4	<i>tri_right</i> marker

### Line

character	line
-	solid
--	dashed
:	dotted
-.	dash-dotted

### Color

The color can be specified via a character code, a tableau color, a full name or a HEX code.

**Note:** If a formatting string contains more parameters than just the color, only the character code is supported.

### Character code

character	color
b	blue
r	red
g	green
c	cyan
m	magenta
y	yellow
k	black
w	white

The resulting colors are

#### Base Colors



### Tableau color

The tableau colors are the default colors of the MPL color cycle. If no color is specified, the color of the lines will iterate over these colors.

The tableau colors are

#### Tableau Palette



Instead of `tab:<colorname>` it is equivalent to use `C<color number>`. The colors are numbered according to above image and C0 is `tab:blue` and C9 is `tab:cyan`.

### Examples

```
plt::plot(x, y, "tab:green"); // equivalent to "C2"
plt::plot(x, y, "C3");      // equivalent to "tab:red"
plt::plot(x, y);            // automatic coloring: "tab:blue"
plt::plot(x, y);            // automatic coloring: "tab:orange"
plt::plot(x, y);            // automatic coloring: "tab:green"
```

## Full name

To make way for your creativity all colors in the following table are supported. It's a perfect way to waste a lot of time.

### CSS Colors

	black		bisque		forestgreen		slategrey
	dimgray		darkorange		limegreen		lightsteelblue
	dimgrey		burlywood		darkgreen		cornflowerblue
	gray		antiquewhite		green		royalblue
	grey		tan		lime		ghostwhite
	darkgray		navajowhite		seagreen		lavender
	darkgrey		blanchedalmond		mediumseagreen		midnightblue
	silver		papayawhip		springgreen		navy
	lightgray		moccasin		mintcream		darkblue
	lightgrey		orange		mediumspringgreen		mediumblue
	gainsboro		wheat		mediumaquamarine		blue
	whitesmoke		oldlace		aquamarine		slateblue
	white		floralwhite		turquoise		darkslateblue
	snow		darkgoldenrod		lightseagreen		mediumslateblue
	rosybrown		goldenrod		mediumturquoise		mediumpurple
	lightcoral		cornsilk		azure		rebeccapurple
	indianred		gold		lightcyan		blueviolet
	brown		lemonchiffon		paleturquoise		indigo
	firebrick		khaki		darkslategray		darkorchid
	maroon		palegoldenrod		darkslategrey		darkviolet
	darkred		darkkhaki		teal		mediumorchid
	red		ivory		darkcyan		thistle
	mistyrose		beige		aqua		plum
	salmon		lightyellow		cyan		violet
	tomato		lightgoldenrodyellow		darkturquoise		purple
	darksalmon		olive		cadetblue		darkmagenta
	coral		yellow		powderblue		fuchsia
	orangered		olivedrab		lightblue		magenta
	lightsalmon		yellowgreen		deepskyblue		orchid
	sienna		darkolivegreen		skyblue		mediumvioletred
	seashell		greenyellow		lightskyblue		deeppink
	chocolate		chartreuse		steelblue		hotpink
	saddlebrown		lawngreen		aliceblue		lavenderblush
	sandybrown		honeydew		dodgerblue		palevioletred
	peachpuff		darkseagreen		lightslategray		crimson
	peru		palegreen		lightslategrey		pink
	linen		lightgreen		slategray		lightpink

## Examples

```
plt::plot(x, y, "seagreen");
plt::plot(x, y, "firebrick");
```

## HEX code

HEX codes are supported as #xxxxxx. To find the color of your desire a variety of online tools are available, as [this one](#) for instance.



## 1.4.5 Frequent problems

### Figure layout

#### Cutoff labels

If the axis ticks are too long, the axis labels might be outside of the figure. When saving the figure, the labels can be cutoff.

For instance the following code produces a cutoff y-axis label.

```
#include "../matplotlibcpp.h"
#include <Eigen/Dense>
#include <iostream>

namespace plt = matplotlibcpp;

int main() {
    Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(200, 0, 6);
    Eigen::VectorXd y, z;

    // y = exp(sin(x)), z = exp(cos(z))
    y = x.array().sin().exp().matrix();
    z = x.array().cos().exp().matrix();

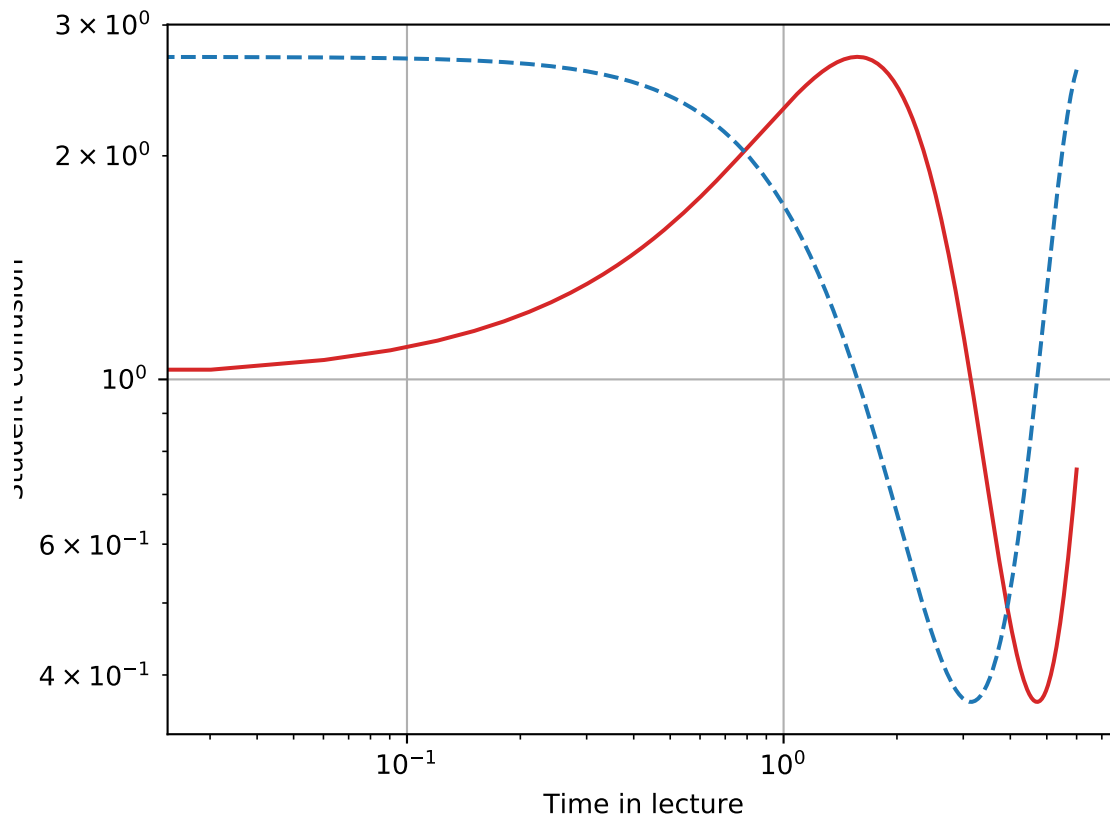
    plt::figure();

    plt::loglog(x, y);
    plt::loglog(x, z);

    plt::xlabel("Time in lecture");
    plt::ylabel("Student confusion");

    plt::grid();
    plt::savefig("loglog.pdf"); // !
}
```

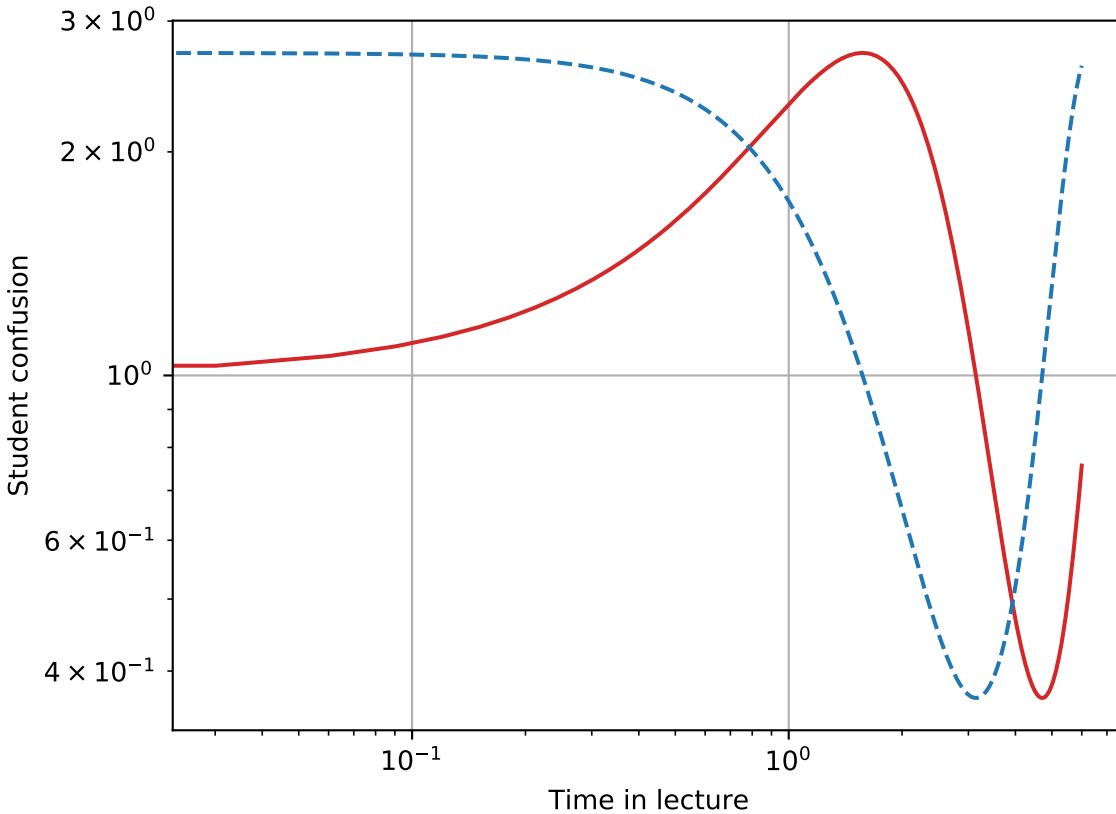
The output is



To fix this problem we can tell MPL to use the available space on the empty sides of the figure, which can be done either by a call to `tight_layout` or specifying `bbox_inches="tight"` in `savefig`.

Thus the issue is fixed by

```
plt::savefig("loglog.pdf", {"bbox_inches", "tight"}); // !
```



### 1.4.6 License

The MIT License (MIT)

Copyright (c) 2014 Benno Evers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 1.4.7 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.

### 1.4.8 To do

#### Generalise to Vector

a lot of functions

#### Layout

- *figure\_size* should be deprecated, instead use figure with keywords
- errorbars

## M

`matplotlibcpp::axis` (*C++ function*), 16  
`matplotlibcpp::fignum_exists` (*C++ function*), 14  
`matplotlibcpp::figure` (*C++ function*), 14  
`matplotlibcpp::figure_size` (*C++ function*), 14  
`matplotlibcpp::legend` (*C++ function*), 14  
`matplotlibcpp::loglog` (*C++ function*), 11, 12  
`matplotlibcpp::plot` (*C++ function*), 10, 11  
`matplotlibcpp::savefig` (*C++ function*), 16  
`matplotlibcpp::semilogx` (*C++ function*), 12, 13  
`matplotlibcpp::semilogy` (*C++ function*), 13  
`matplotlibcpp::show` (*C++ function*), 17  
`matplotlibcpp::suptitle` (*C++ function*), 15  
`matplotlibcpp::text` (*C++ function*), 13  
`matplotlibcpp::title` (*C++ function*), 15  
`matplotlibcpp::xlim` (*C++ function*), 15  
`matplotlibcpp::ylim` (*C++ function*), 15

## V

`Vector` (*C++ type*), 9  
`Vector::at` (*C++ function*), 9  
`Vector::begin` (*C++ function*), 10  
`Vector::data` (*C++ function*), 9  
`Vector::end` (*C++ function*), 10  
`Vector::operator[]` (*C++ function*), 9  
`Vector::size` (*C++ function*), 9  
`Vector::value_type` (*C++ type*), 9