



Scientific Plotting Made Simple

A ROOT-based Plot Organizer

The ROOT plotting challenge

- ROOT is powerful, but everyday plotting often requires substantial boilerplate code.
- Similar I/O and styling tasks are repeatedly implemented in different macros.
- Often plotting logic becomes intertwined with analysis code, making iteration slow.
- Producing figures requires handling files, data types, styling, and other ROOT details.

The SciRooPlot approach

- Introduces an abstraction layer on top of ROOT.
- Provides a common plotting infrastructure and organization.
- Lets users describe the desired figure rather than its implementation.
- Preserves the familiar ROOT ecosystem and styling conventions.
- Enables interactive, project-based plotting workflows.

ROOT

- ✓ Histograms
- ✓ Graphs
- ✓ Functions
- ✓ Files



SciRooPlot

- ✓ Abstract plot specifications
- ✓ Plot grouping & organization
 - ✓ Interactive plotting
- ✓ Project-based workflows

SciRooPlot Workflow

- Store analysis results in ROOT or CSV files.
- Define plot specifications using C++ or Python interface and save them as a SciRooPlot project.
- Generate and explore plots interactively via the `plot` command-line tool.
- Create and manage projects with the `srp` command-line tool.

Installation

```
$ git clone https://github.com/SciRooPlot/SciRooPlot.git  
$ cd SciRooPlot  
$ ./scripts/install.sh
```

- The installer prints a command that enables the SciRooPlot environment.
- Add it to the shell startup script (e.g. `~/.bashrc` or `~/.zshrc`) to load SciRooPlot automatically in new terminals.
- After setup, the `srp` and `plot` command-line tools are available system-wide.
- Future updates can be installed with: `$ srp update`

Your First Project

- Initialize a new SciRooPlot project:

```
$ srp init-cpp <project> [<dir>]
```

- Creates a project directory (./<project> or ./<dir>).
- Contains DefinePlots.cpp, where plot specifications are defined.

```
myProject/  
├── DefinePlots.cpp  
└── output/
```

- Project automatically registered with SciRooPlot.
- Includes working examples using dummy data.
- Generate your first plot immediately:

```
$ plot examples ptSpec
```

- Plots always stay in sync with DefinePlots.cpp.

Project Management

Select another project

```
$ srp select <project>
```

List all projects

```
$ srp projects
```

Change output directory (default is ./output)

```
$ srp set <project> outdir <path>
```

Additional commands:

\$ srp help	List available options.
\$ srp show <project>	Show project settings.
\$ srp remove <project>	Unregister project.
\$ srp print <file>	List root file contents.
\$ srp open <file>	Open root file.

Using the App

- Plots specified in `DefinePlots.cpp` have a unique **name** within a **group**.
- These identifiers are used to select plots from the command line.

```
$ plot <group> <name> [<mode>]
```

- Tab completion simplifies browsing through available names.
- Both `<group>` and `<name>` support regular expressions.
- Multiple plots can be generated with a single command.

Mode	Description
show	Open plots interactively (default mode).
list, print	List plots matching the request or print their settings.
pdf, eps, svg, png	Export plots as graphics files.
macro	Generate ROOT .c macros that reproduce the plots.
gif	Combine matching plots into an animated GIF (gif+N sets the frame delay in units of 10 ms).
file	Save all requested plots into a single ROOT file.
data	Save input data for requested plots into a ROOT file.

Regular Expressions

Select plot names matching pattern

```
$ plot paperPlots 'moneyPlot[1,2]'
```

NB.: Quotes are only required if regex contains special shell characters like `[]`, `()` and `|`.

Select all plots in a figure group

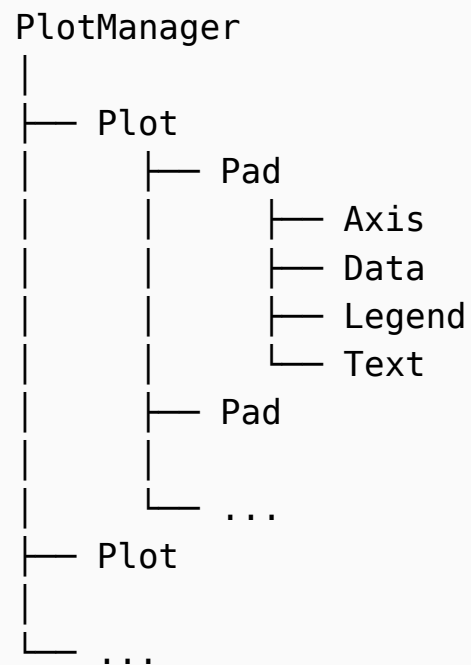
```
$ plot paperPlots .+
```

Select plots in groups starting with pattern

```
$ plot pp_.+ myPlot
```

General Code Structure

- Plots are handled and exported by PlotManager.
- Input files are registered under user-defined aliases.
- Data objects within input files are identified by name.
- Each Plot has a unique name within a figure group
- It consists of one or more Pads (plot[1], plot[2], ..)
- Pads contain settings of Axis, Data, Text and Legend.



DefinePlots.cpp

```
int main(int argc, char* argv[])
{
    PlotManager pm("myProject");
    pm.AddDataSource("dataSource", "/path/to/file.root");
    { // -----
        Plot plot("plotName", "groupName");
        plot[1].AddData("dataObjectName", "dataSource", "my label");
        plot[1].AddLegend();
        plot[1].AddText("some text");
        plot[1]['X'].SetTitle("x axis title");
        pm.AddPlot(plot);
    } // -----
    pm.SaveProject();
    return 0;
}
```

Language Note

This documentation uses the C++ interface.
Find the [Python version](#) here.

Adding Data Sources

- A data source is a collection of input files with a unique identifier that is later used for the plot definitions.
- Entire ROOT files or individual subdirectories/lists therein can be registered as input sources.
- Multiple inputs can be added either via successive `AddDataSource()` calls or by passing them as a list.
- Adding a directory to a data source registers all ROOT files it contains, including the ones in subdirectories.
- File paths are always absolute, but the `SRC_DIR` helper enables paths relative to `DefinePlots.cpp`.
- Shell environment variables (including user-defined ones) are supported and expanded automatically.
- Registered files are searched in alphabetical order. Within each file, the directory hierarchy is traversed until the first matching data object is found.
- Local ROOT objects can also be directly added.

```
pm.AddDataSource("sourceA", "/path/to/file1.root");

pm.AddDataSource("sourceF", "/path/to/file6.root:dir/or/list");

pm.AddDataSource("sourceD", {"path/to/file4.root",
                             "path/to/file5.root"});

pm.AddDataSource("sourceE", "/path/to/directory/");

pm.AddDataSource("sourceB", SRC_DIR + "../path/file2.root");

pm.AddDataSource("sourceC", "${HOME}/path/to/file/file3.root");
```

```
auto myHist = new TH1D("myHist", "", 100, -5, 5);
pm.AddDataSource("sourceG", myHist);

auto myGraph = new TGraph();
myGraph->SetName("myGraph"); // graph needs a name!
auto myFunc = new TF1("myFunc", "gaus", -5, 5);
pm.AddDataSource("sourceG", {myGraph, myFunc});
```

Plot Appearance

- Defining base plots avoids repeating common layout settings across many plots.
- Base plots are ordinary `Plot` objects registered with the `PlotManager` via `AddBasePlot()`.
- Default settings applied to `plot[0]` automatically affect all pads unless overridden.
- Several ready-to-use base plots (e.g. 1d, 2d, 1d_ratio) are provided with `SciRooPlot`.
- Existing base plots can be used directly or modified to create custom layouts.
- Axis ranges, titles, scales, and other axis properties are configured intuitively through `plot[pad]['X']`, `['Y']`, and `['Z']`.
- List of accessors of `Plot`, `Pad` and `Axis` in appendix.

```
Plot plot("myBasePlot");
plot.SetDimensions(710, 710);
plot.SetTransparent();

plot[0].SetMargins(0.07, 0.14, 0.12, 0.07);
plot[0].SetDefaultMarkerSize(1.2).SetDefaultLineWidth(2.);
plot[0].SetDefaultTextFont(43).SetDefaultTextSize(24);
plot[0].SetDefaultColors({kBlack, kBlue, kRed});
plot[0].SetDefaultMarkerStyles({kFullCircle});
plot[0].SetDefaultLineStyles({kSolid, kDashed});

plot[0]['X'].SetTitleSize(28).SetTitleOffset(1.1);
plot[0]['Y'].SetTitleSize(28).SetTitleOffset(1.5);

plot[1].SetPosition(0., 0., 1., 1.);

pm.AddBasePlot(plot);
```

```
pm.AddBasePlot(PlotManager::MakeBasePlot("1d"));
// create new plot on basis of "1d" base plot:
Plot plot("myPlot", "myGroup", "1d");
plot[1]['X'].SetRange(0., 10.).SetTitle("x title");
```


Adding Data

- The data-type agnostic `AddData()` function accepts virtually all ROOT histograms, graphs, profiles, functions and trees.
- Data is identified by name and automatically searched for in the corresponding data source across all associated files.
- Directory or list paths within ROOT files can be prepended to object names to disambiguate identical names.
- Data from multiple input aliases can be combined seamlessly within the same plot.
- By default, the first added object defines the plot frame. This can be overridden via `SetDefinesFrame()`.
- Ratios are added analogously via `AddRatio()` and support divisions between different data types. If the abscissae or binning differ, the denominator is interpolated.

Supported types

- Histograms (TH1, TH2, TH3, THn, THnSparse)
- Profiles (TProfile, TProfile2D)
- Graphs (TGraph, TGraph2D)
- Functions (TF1, TF2, TF3)
- Efficiencies (TEfficiency)
- Trees (TTree)
- Tables (.csv, .dat, .txt, .tsv, .tab)

```
plot[1].AddData("myGraph", "sourceA");
plot[1].AddData("myFunc", "sourceA");
plot[1].AddData("path/to/myHist", "sourceB");

plot[1].AddData("largerHisto", "sourceB")
        .SetDefinesFrame();

plot[1].AddRatio("numDataName", "sourceA",
                "denomDataName", "sourceB");
```

Data Appearance

- Data appearance is configured by chaining setters directly to `AddData()` or afterwards by accessing previously added data via `plot[1](n)` (equivalent to `plot[1].GetData(n)`).
- Drawing styles are configured via `SetOptions()`, accepting both native ROOT drawing option strings and predefined aliases (e.g. `points`, `line`, `curve`, `band`, `boxes`, `hist`, `colz`).
- Complete Data layouts can be defined once and reused across many plots, ensuring a consistent appearance.
- See the appendix for the complete list of available Data and Ratio setters.

```
plot[1].AddData("h", "sourceA").SetOptions("HIST");
plot[1].AddData("h", "sourceA").SetOptions(points);

// modify data appearance later
plot[1](1).SetColor(kRed);
plot[1](2).SetMarkerStyle(kOpenCircle);

Data layout = Data()
    .SetDataSource("sourceA")
    .SetOptions(curve)
    .SetLineStyle(kDashDotted)
    .SetLineWidth(4.)
    .SetLineColor(kOrange);

// reuse appearance (requires data source to be defined)
plot[1].AddData("graph1", layout);
plot[1].AddData("graph2", layout).SetColor(kBlue);

plot[1].AddRatio("h1", layout, "h2", "sourceB");
```

Default Styles and Color Palettes

- Default colors, marker styles, line styles and other properties can be defined once per pad.
- Style lists are applied cyclically to newly added data.
- Continuous color gradients can be generated from arbitrary color endpoints.
- The same gradients can be used both for automatic styling of many data objects and as palettes for 2D histograms.

```
// cyclic default styling
vector<int> colors = {kBlue, kRed, kGreen+2};
vector<int> styles = {kOpenCircle, kOpenCross};
plot[1].SetDefaultMarkerColors(colors);
plot[1].SetDefaultMarkerStyles(styles);

// continuous color gradient
vector<tuple<float, float, float, float>> rainbowColors =
{ {0., 0., 1., 0.}, // blue
  {0., 1., 1., 0.25}, // cyan
  {0., 1., 0., 0.50}, // green
  {1., 1., 0., 0.75}, // yellow
  {1., 0., 0., 1.} // red
}; // (r,g,b,pos)
plot[1].SetDefaultColors(rainbowColors);

// same gradient as 2D palette
plot[1].SetPalette(rainbowColors);

// as alternative to standard root palettes
plot[1].SetPalette(kBird);
```

Legends and Text

- Legend entries are generated automatically and inherit the appearance of the corresponding data by default.
- Labels support placeholders such as <mean>, <integral>, <entries>, <maximum>, which are expanded automatically.
- Multiple legends can coexist and data can be assigned to specific legends via SetLegend(n).
- Legend entries can be customized individually.
- custom legends and legend entries can be made
- Text can be added also with multiple lines

```
plot[1].AddData("h1", "input", "Data");
plot[1].AddData("f1", "input", "Fit");
plot[1].AddData("g1", "input", "Graph").SetLegend(2);

// generates a legend with entries "Data", "Fit"
plot[1].AddLegend(0.7, 0.8);
// generates a second legend with entry "Graph"
plot[1].AddLegend(0.2, 0.2);

plot[1].GetLegend(2)
    .GetEntry(1)
    .SetLabel("Best fit");

plot[1].AddLegend(0.9, 0.1); // rel. (x, y) pos. in pad

plot[1].AddText(0.18, 0.88, "Work in progress");
```

Projecting and Profiling Multidimensional Data

- Existing multidimensional histograms can be projected directly when adding them to a plot.
- Standard projections (ProjectX(), ProjectY(), ...) are available for 2D and 3D histograms.
- Projections can be restricted to selected ranges of the remaining axes using either bin numbers or user coordinates.
- The generic Project() function supports arbitrary-dimensional histograms and arbitrary projection axes.
- One- and two-dimensional profiles can be created analogously from multidimensional histograms.
- The same functionality is available for numerator and denominator of ratios.

```
// Standard projection of a 2D histogram
plot[1].AddData("my2DHist", "input")
        .ProjectX();

// Restrict Y before projecting
plot[1].AddData("my2DHist", "input")
        .ProjectX(20, 80);

// Generic projection of an N-dimensional histogram
plot[1].AddData("myNDHist", "input")
        .Project({2, 0}, {{1, 20, 80}, {3, 10, 40}});

// Profile of a 2D histogram
plot[1].AddData("my2DHist", "input")
        .ProfileX();

plot[1].AddData("myNDHist", "input")
        .Profile({2}, {{0, 10, 50}});
```

Processing Tree and Table Data

- Tree leaves and table columns of csv files are accessed through the same interface.
- Create histogram projections, profiles and scatter plots directly from tabular data.
- Binning can be inferred automatically or specified explicitly with uniform or custom bin edges.
- Data can be filtered, derived quantities defined and arbitrary C++ expressions used for projections and selections.
- Entry ranges can be selected to process only subsets of the input.
- The same functionality is available for both ROOT trees and CSV tables.

```
// Histogram projection
plot[1].AddData("myTree", "input")
        .Project1D({"pt", 100})
        .Filter("eta > 0");

// Profile
plot[1].AddData("myTree", "input")
        .Profile1D("eta", "pt");      // <pt> vs eta

// Scatter plot using arbitrary expressions
plot[1].AddData("myTree", "input")
        .Define("r", "sqrt(x*x+y*y)")
        .Scatter("r", "z")
        .Filter("abs(z) < 5");

// CSV tables are handled identically
plot[1].AddData("myData", "input")
        .Scatter("a", "b");
```

Work in Progress...

