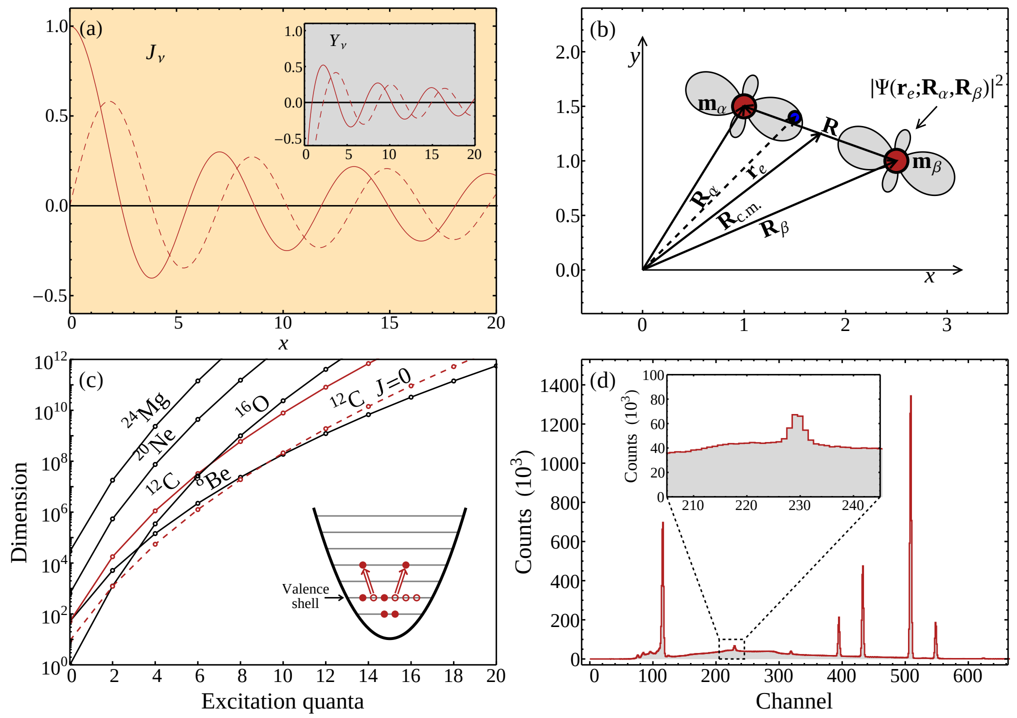
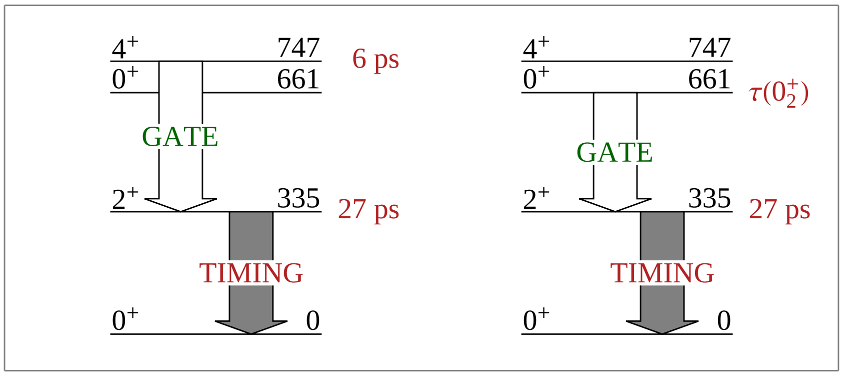
SciDraw

Publication-quality scientific figures with Mathematica

A user's guide and reference manual

Mark A. Caprio, *Department of Physics, University of Notre Dame*





Version 0.0.7 (March 28, 2015)

Contents

I User's guide 1

1 Introduction 2

1.1 What is SciDraw? . . . 2
1.2 Design philosophy of SciDraw . . . 2
1.3 Using this user's guide and reference manual . . . 3
1.4 Notation and conventions . . . 3
1.5 Further information and updates . . . 4
1.6 Acknowledgement of use . . . 4

2 Installation 5

3 Basic concepts by tutorial 9

3.1 **Tutorial 1**: Getting started with figures and panels . . . 9
3.1.1 Problem statement . . . 9
3.1.2 Setting up the "canvas" . . . 10
3.1.3 Setting up the main panel frame . . . 12
3.1.4 **Interlude**: Panels and coordinates . . . 15
3.1.5 **Interlude**: Typesetting in Mathematica . . . 16
3.1.6 Including a plot from Mathematica . . . 18
3.1.7 Adding some annotations: Labels and rules . . . 19
3.1.8 Setting up the inset panel . . . 22
3.1.9 The final figure . . . 23
3.1.10 **Supplement**: Inserting Mathematica graphics (including three-dimensional plots) . 25
3.2 **Tutorial 2**: Drawing a diagram [STUB] . . . 29
3.3 **Tutorial 3**: Getting started with data plots and legends [STUB] . . . 30

4 Topical discussions 31

4.1 Drawing objects . . . 31
4.1.1 Basic drawing shapes . . . 31
4.1.2 Arrows . . . 36
4.1.3 Annotation objects [STUB] . . . 37
4.2 Standard options for objects . . . 38
4.3 Formatting text . . . 42
4.4 Level schemes . . . 47
4.4.1 Note for LevelScheme users . . . 47
4.4.2 Levels, extensions, and connectors . . . 47
4.4.3 Transition arrows . . . 50
4.5 Generating EPS/PDF output for publication [STUB] . . . 57

II Reference manual 58

5 Setting up the canvas with `Figure` 59

6 Objects 61

7 Coordinates, anchors, and regions 62
7.1 Points and anchors 62
7.1.1 Points 62
7.1.2 Anchors 63
7.1.3 Retrieving information about anchors 64
7.1.4 Generating anchors from objects 65
7.1.5 Displacing and rotating anchors 65
7.2 Curves 66
7.2.1 Points on curves 67
7.2.2 Curves from graphics 68
7.3 Relative positions within a rectangle (*i.e.*, text offsets) 69
7.4 Rectangular regions 70

8 Options for figure objects 72
8.1 `FigObject`: Common default options 72
8.1.1 Overall appearance 72
8.1.2 Outline appearance 73
8.1.3 Fill appearance 74
8.1.4 Point appearance 75
8.1.5 Text appearance 75
8.1.6 Text font characteristics 76
8.1.7 Text background and frame 76
8.1.8 Text positioning 77
8.1.9 Layering 79
8.2 Attached label options 79

9 Styles and advanced option control 81
9.1 Defining styles 81
9.2 Using styles 82
9.3 Overriding options by object name 83
9.4 Scoping changes to default options 84

10 Panels and axes 85
10.1 `FigurePanel` basics 85
10.2 Multipanel arrays 94
10.3 Iterated generation of panels 100
10.4 `FigAxis` 104
10.5 `WithOrigin` 105
10.6 `FigureGroup` 105

11 Basic drawing shapes 107
11.1 `FigLine` 107
11.2 `FigPolygon` 109
11.3 `FigRectangle` and `FigCircle` 110
11.4 `FigPoint` 116
11.5 Splines 116

12 Arrows 118

13 Annotations **121**
13.1 `FigLabel` 121
13.2 `FigBracket` 122
13.3 `FigRule` 124

14 Graphics inclusion **126**
14.1 `FigGraphics` 126
14.2 `FigInset` 126

15 Level schemes **128**
15.1 `Lev` 128
15.2 `ExtensionLine` 130
15.3 `Connector` 130
15.4 `BandLabel` 131
15.5 `Trans` 131
15.6 Decay scheme generation 133

16 Data plotting **134**
16.1 `DataPlot` 134
16.1.1 Data sets 134
16.1.2 Data plot appearance 137
16.1.3 Defining new axis scales, symbol shapes, and curve shapes 141
16.2 `DataLegend` 142
16.3 Data manipulation utilities 144

III Appendices **146**

A Known issues **147**

B Revision notes **148**

C Licenses **149**

Part I

User’s guide

1 Introduction

1.1 What is SciDraw?

SciDraw is a system for preparing publication-quality scientific figures with Mathematica. SciDraw provides both a *framework for assembling figures* and *tools for generating their content*. In general, SciDraw helps with generating figures involving mathematical plots, data plots, and diagrams.

The structural framework includes:

- Generation of panels for multi-panel and inset figures,
- Customizable tick marks,
- Style definitions, for uniformly controlling formatting and appearance across multiple figures,
- Graphical objects for annotating figures with text labels, axes, *etc.*

Any graphics (plots, images, *etc.*) which you can produce in Mathematica (or import into Mathematica) can, with occasional restrictions, easily be included in a SciDraw figure.

Beyond these structural elements, SciDraw then provides an object oriented drawing system which makes many hard-to-draw scientific diagrams comparatively easy to generate. The object oriented approach, plus the use of styles, allows extensive manual fine tuning of the appearance of text and graphics, while also helping ensure uniformity across figures. It also greatly simplifies the arrangement of objects (and text labels) in relation to each other — especially when it comes to attaching text labels to objects (shapes, data curves, arrows, *etc.*) in the figure, as well as connecting these shapes to each other.

SciDraw also provides data plotting and legend generation capabilities complementary to those built into Mathematica. The scope of these is relatively focused — on making standard two-dimensional data plots, but making them well.

SciDraw's origins lay in the preparation of high-quality level schemes, or level energy diagrams, as used in nuclear, atomic, molecular, and hadronic physics — SciDraw is the successor to LevelScheme [Comput. Phys. Commun. **171**, 107 (2005)] and retains the capabilities of this package. SciDraw automates many of the tedious aspects of preparing a level scheme, such as positioning transition arrows between levels or placing text labels alongside the objects they label. It also includes specialized features for creating certain common types of decay schemes encountered in nuclear physics.

1.2 Design philosophy of SciDraw

A few basic principles have guided the design of SciDraw. One is to have a system whereby even major formatting changes to a figure can be made relatively quickly. Objects in a figure (such as curves, arrows, text labels, or drawing shapes) are attached to each other, so that if one object is moved the rest follow automatically. For instance, in a level energy diagram, transition arrows are attached to levels, labels attached to levels and to transitions, *etc.* Another principle is for objects to have reasonable default properties, so that a figure can initially be drawn with minimal attention to formatting features. But the user must then have near-complete flexibility in fine tuning formatting details to accomodate whatever special cases might arise. This is accomplished by making the more sophisticated formatting features accessible through various optional arguments or *options*, for which the user can specify values. The user can specify the values of options for individual objects, or the user can set new *default* values of options for the whole figure to control the formatting of many objects at once. Especially powerful formatting control is provided through the use of *styles*, which allow common formatting choices to be made (and adjusted) across many figures, or for specific sets of objects within a figure, all at once. Finally, attention has been paid to providing a uniform user interface for all drawing objects, based upon a consistent notation for the specification of properties for the outline, fill, and text labels of objects.

1.3 Using this user's guide and reference manual

The *user's guide* is still under construction. There should be enough to to get you started, especially if you are brave of heart. The *reference manual* is complete and can, together with the *example notebooks*, fill in most of the missing details.

The *user's guide* is meant to get you started quickly, to familiarize you with the basic tools at your disposal, and to help you absorb the basic principles at play in SciDraw. You will want to start learning SciDraw by working through the *tutorials* (Sec. 3). The tutorials introduce the essential concepts of SciDraw, without attempting to encyclopedically cover all the details. These are fully-worked examples which lead you, step-by-step, through the thought process which goes into drawing a figure with SciDraw. After the tutorials, you will find more focused *topical discussions* delving into matters such as multipanel plotting and level schemes (Sec. 4).

The *reference manual* provides a comprehensive reference to the SciDraw interface, organized by topic. You will be well-served to familiarize yourself with the reference manual at the same time as you go through the user's guide. You will find convenient reference tables and further details there.

Just as important are the *example notebooks* which come along with SciDraw. These include code for the examples in the user's guide, and many additional examples as well. Users often find these the best way of learning how to work with SciDraw.

The documentation for the *CustomTicks package* is found in a separate file `CustomTicksGuide.pdf`, which also comes with SciDraw.

The old *LevelScheme user's guide* (Version 3.53) is also included with this distribution, in `LevelSchemeGuide.pdf`. While the present guide is still under construction, there are a couple of special topics where you will still be referred to the LevelScheme guide.

Prerequisites. It is assumed that you have some basic experience starting Mathematica, evaluating cells, and opening and saving notebook files. You should be comfortable with using the Mathematica `Plot` or `ListPlot` functions to generate some basic graphics, and you should have a working knowledge of the more common options for two-dimensional graphics in Mathematica, such as `PlotRange` and `FrameLabel`.

Links to help. In general, this guide is *not* an introduction to Mathematica, but and effort is made to give pointers to relevant Mathematica documentation along the way. These are given in **sans serif** type. For instance, if this guide tells you to see **tutorial/VisualizationAndGraphicsOverview** for more information, you can open the Mathematica help browser and enter this link into the search bar. In fact, you will probably want to read this — it is the Mathematica *Virtual Book* chapter on "Visualization and Graphics" — to get an introduction to graphics in Mathematica, if you have not done so already.

1.4 Notation and conventions

Many dimensions (such as line thicknesses or text position adjustments) will be specified in "printer's points", where $1\,\mathrm{pt} = 1/72\,\mathrm{inch}$ or $0.35\,\mathrm{mm}$. These are convenient and customary units to use for controlling text and graphics. A thin line is about 1 pt thick, and characters of normal text are $\sim 10\,\mathrm{pt}$ high.

The Mathematica option symbol ("$\to$") which appears in example input in this guide is entered from the keyboard as a hyphen followed by a greater-than sign ("->"). The double bracket characters ('⟦⋯⟧') which appear in example input in this guide are entered from the keyboard as Esc `-[-[-` Esc and Esc `-]-]-` Esc , respectively.

1.5 Further information and updates

Further information and updates to SciDraw may be obtained through the SciDraw home page:

`http://scidraw.nd.edu`

1.6 Acknowledgement of use

If you use SciDraw to prepare the figures for your publication, an acknowledgement is always welcome. For example, you might include a statement such as the following in the "Acknowledgements" section:

> The figures for this article have been created using the SciDraw scientific figure preparation system [M. A. Caprio, Comput. Phys. Commun. 171, **107** (2005), `http://scidraw.nd.edu`].

Feel free to modify this statement as appropriate, *e.g.*, changing "the figures for this article" to "Figure 5".

However, acknowledging SciDraw in individual figure captions is not recommended. A full acknowledgement is cumbersome in a caption, while a simple bibliographic reference would be mistaken to mean that the figure *data* were generated with SciDraw or taken from the Computer Physics Communications paper.

Note: The Computer Physics Communications paper indicated here [M. A. Caprio, Comput. Phys. Commun. 171, **107** (2005)] is the old paper on LevelScheme, the predecessor software to SciDraw. Hopefully an updated reference will be available someday soon, so please check back.

2 Installation

DON'T PANIC! Installation is actually reasonably straightforward. These instructions are only as long as they are since they err on the side of completeness.

> ***Requirements: This version of SciDraw requires Mathematica 8 or higher. It has been tested under Mathematica 10.***

Distribution contents. The SciDraw package is distributed as a ZIP archive. To start with, you need to extract the files from this ZIP archive.[1] You will find that the extracted files are in two directories (*i.e.*, folders):

The directory `packages` contains the Mathematica packages which make up SciDraw. These are in several subdirectories, named `SciDraw`, `CustomTicks`, `BlockOptions`, *etc.* *In order to be able to load SciDraw, you will need to move these package subdirectories to a location where Mathematica can find them, as discussed in detail below.*

The directory `doc` contains the documentation (including this guide and a separate guide for the `CustomTicks` package) and several notebooks containing example SciDraw figures. *You may move the documentation to any convenient location, so you can easily find and refer to it later.*

Background on packages. If you are not yet familiar with the idea of "packages" in Mathematica, and how to load them, now would be a good time to learn the basics from **tutorial/MathematicaPackages**. You might also find it helpful to be familiar with the information in **tutorial/NamingAndFindingFiles**.

Installing the package files. You need to decide upon a suitable place in your directory structure where you would like to keep package files — including SciDraw, and perhaps other packages as well. For example, you might create a directory named `mathematica` in your home directory, to contain all your Mathematica packages, documentation, *etc.* For instance, on a Microsoft Windows 7 system, this would have a name like

```
C:\Users\mcaprio\mathematica
```

on an Apple Macintosh OS X system

```
/Users/mcaprio/mathematica
```

or, on a Linux system,

```
/home/mcaprio/mathematica
```

Then, it is important to realize that, as far as Mathematica is concerned, SciDraw is actually a *collection* of packages — contained in the various subdirectories named `SciDraw`, `CustomTicks`, `BlockOptions`, *etc.* which we mentioned above. You need to move those subdirectories into your new Mathematica package directory,[2] *e.g.*, for the Windows system, the subdirectories would now be

```
C:\Users\mcaprio\mathematica\SciDraw
C:\Users\mcaprio\mathematica\CustomTicks
```

[1] The way to decompress a ZIP file depends on your operating system, *e.g.*, modern versions of Windows can open ZIP files automatically, or Unix/Linux systems should have an `unzip` utility available from the command line.

[2] **Common error:** That is, all those subdirectories will have to be moved directly into the top level of `mcaprio/mathematica` for Mathematica to find them, not buried deeper in some subdirectory. So, for instance, "moving" `packages` as a whole into `mcaprio/mathematica` will not work. Pay careful attention to the example names given below. There is no "`packages`" in them.

etc., on an Apple Macintosh OS X system

```
/Users/mcaprio/mathematica/SciDraw
/Users/mcaprio/mathematica/CustomTicks
```

etc., or, on a Linux system,

```
/home/mcaprio/mathematica/SciDraw
/home/mcaprio/mathematica/CustomTicks
```

etc.,

Setting the search path. Mathematica must still be told that this new directory is a place where it should look in order to find package files. More specifically, Mathematica only searches for package files in the directories listed in Mathematica's variable `$Path`.[3] You must append the name of your your own package directory to this list, using `AppendTo`. For instance, given the example directory names we chose above, for Windows 7 we would have[4]

```
AppendTo[$Path, "C:\\Users\\mcaprio\\mathematica"];
```

for the Macintosh

```
AppendTo[$Path, "/Users/mcaprio/mathematica"];
```

or for Linux

```
AppendTo[$Path, "/home/mcaprio/mathematica"];
```

You may also read the example in **ref/$Path** for a more elegant approach involving the use of the Mathematica `$HomeDirectory` environment variable.

Note that, every time Mathematica is restarted, the `$Path` variable goes back to its "factory default" value. Therefore, your package directory needs to be added to `$Path` *each time you restart Mathematica*.

That starts to sound like a nuisance, doesn't it? Thus, you will probably want to adopt one of two simple solutions, to avoid retyping this modification to the path each time:

1) The most obvious — but still not entirely satisfactory — solution is to include the `AppendTo` command in each notebook just before `Get["SciDraw`"]`. But even this becomes tedious after a while. And, if you share the notebook between different computer systems with different directory names — or share it with collaborators who use different directory names — you will frequently have to edit the path name given here.

2) The most satisfactory and permanent solution — though it takes a little bit more setup work work up front, right now — is to include the `AppendTo` command in your personal `init.m` startup file.[5] To find where this file is located, first evaluate the Mathematica variable `$UserBaseDirectory`. For instance, on a Windows 7 system, you might find

```
C:\Users\mcaprio\AppData\Roaming\Mathematica
```

on a Macintosh,

```
/Users/mcaprio/Library/Mathematica
```

[3]See **ref/$Path** for a more detailed explanation of the Mathematica search path.

[4]Note the need to use *double* backslashes inside the input string. See **tutorial/InputSyntax** to understand why. If you have ever programmed with, *e.g.*, C/C++ or Python, you will be familiar with such "backslash escapes". Actually, I prefer to use a single forward slash as the path separator even under Windows, as in the Mac and Linux examples — using the forward slash in place of the backslash works just fine under modern versions of Windows.

[5]For more information on the initialization file, see **tutorial/ConfigurationFiles**.

or, on a Linux system,

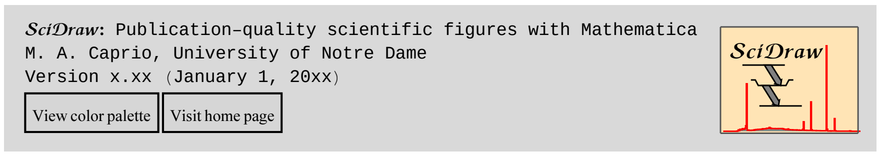
```
/home/mcaprio/.Mathematica
```

Then, this "user base directory" should have a subdirectory named `Kernel`, which in turn should contain the initialization file `init.m`. Open `init.m` — either with Mathematica or any text editor — and insert the `AppendTo` command described above, anywhere in the file.

Loading the package. Once your `$Path` is set correctly, you can just load SciDraw as usual for a Mathematica package[6] with[7]

```
Get["SciDraw`"]
```

If you prefer, you can use the equivalent but the shorter form `<<SciDraw``. You should see the SciDraw startup message[8]



```
SciDraw: Publication-quality scientific figures with Mathematica
M. A. Caprio, University of Notre Dame
Version x.xx (January 1, 20xx)
```

CAUTION: Load the package first! You must be sure to always load the package *before* you first try to use any of the SciDraw commands. *Not doing so is a very common source of trouble!* If you ever accidentally try to use any of the symbols defined in a Mathematica package, before loading the package, when you do attempt to load the package you will see "shadowing" error messages such as

```
Figure::shdw : Symbol Figure appears in multiple contexts
    {SciDraw`, Global`}; definitions in context SciDraw` may shadow or be shadowed by other definitions. ≫

FigurePanel::shdw : Symbol FigurePanel appears in multiple contexts
    {SciDraw`, Global`}; definitions in context SciDraw` may shadow or be shadowed by other definitions. ≫
```

Then the package will not be able to run properly for the rest of your Mathematica session.[9] You should exit and restart Mathematica, and try loading the package again.[10]

Alternative installation procedure — Mathematica `Applications` directory. Although the following procedure is not one I choose to follow myself, for reasons which will be noted below, it is preferred by some users and is therefore described here for completeness. Mathematica actually is installed with a certain directory designated as a location for add-on "application" packages, and already included in the default `$Path`.[11] This location is a perfectly acceptable location for SciDraw, and you could simply

[6] Again, see **tutorial/MathematicaPackages** for more on the basics of packages.

[7] If you are more advanced in using packages, you are probably asking "isn't it more efficient to use `Needs["SciDraw`"]` rather than `Get["SciDraw`"]`?". True, `Needs` would insure that SciDraw is not reloaded if it has already been loaded in the current session, *e.g.*, from another notebook. But then you wouldn't get the "splash" cell displayed above – which has convenient buttons on it for accessing, most notably, a named color palette.

[8] *Common mistake:* You need to end the package name with a *backward* single quote "`", not a forward single quote "′".

[9] There is a fundamental reason relating to how the Mathematica language handles *contexts* (*i.e.*, namespaces) for symbols. See **tutorial/MathematicaPackages** for an explanation.

[10] Actually, you do not really need to exit Mathematica. All you need to do is quit the Mathematica kernel (Evaluation>Quit Kernel>Local from the menus).

[11] See **tutorial/MathematicaFileOrganization**.

move the files there. However, using Mathematica's designated directory for add-on packages is not necessarily the most convenient choice — it is usually harder for you to keep track of this location (since it is system-dependent), the packages included there might not automatically be included in backups of your home directory, it might not be convenient to share this location between multiple systems on a network running different operating systems, *etc.* Briefly, to find the directory which has been designated for add-ons, evaluate the Mathematica variable `$UserBaseDirectory`, as described above, to find your "user base directory". This directory will have a subdirectory named `Applications`, which is the one we are looking for. The `Applications` directory should already be included in the default Mathematica `$Path` — you can evaluate `$Path` to check this. This is where you would place all the SciDraw package subdirectories. For instance, under Windows 7, typical names would be

```
C:\Users\mcaprio\AppData\Roaming\Mathematica\Applications\SciDraw
C:\Users\mcaprio\AppData\Roaming\Mathematica\Applications\CustomTicks
```

etc.

Note for LevelScheme users. LevelScheme and SciDraw can both be *installed* at the same time, but they cannot both be *loaded* in the same Mathematica session. You must quit the kernel (or quit Mathematica) between using one and the other. There are many symbol names (such as `Figure`) which are common to both packages, and which would therefore conflict with or "shadow" each other.

It is important to realize that LevelScheme and SciDraw share several subpackages (*e.g.*, `CustomTicks` and `InheritOptions`). The older versions "left over" from your LevelScheme distribution might not be up-to-date enough to work properly with SciDraw. So you want to make sure the old versions are not lingering anywhere in your search `$Path`, where they might accidentally be loaded instead of the newer version. If you follow the recommended installation instructions above, for both LevelScheme Version 3.53 and SciDraw, you will be fine, so long as are sure to replace the old versions of the subpackages (from the LevelScheme distribution) with the new ones (from the SciDraw distribution). That is, delete any old `/home/mcaprio/mathematica/CustomTicks`, taken from LevelScheme, and replace it with the new version from SciDraw, and similarly for the other subpackages.

3 Basic concepts by tutorial

We will start with some "tutorials", or worked examples, to get you started with SciDraw. Once you follow through these tutorials, you should be well on the path towards using SciDraw comfortably and effectively.

The front cover of this guide, which was generated with SciDraw, illustrates some of the many types of plots and diagrams one might wish to include in a scientific figure. The tutorials are loosely built around the idea of walking you through the ideas you would need to generate each of the different plots on the cover. Not only will you learn how to draw the panels on the cover, but you will learn a lot more on the way.

Tutorial 1. The first tutorial focuses on "setting up" a figure. On the front cover of this guide, in panel (a), we use SciDraw to set up the framework of the figure — specifically, the inner and outer frames and labels — while relying on the Mathematica `Plot` command to generate the actual graphics of the plots. The first tutorial walks you through creating this figure, while introducing some basic ideas you need in order to use SciDraw effectively.

Tutorial 2. This tutorial focuses on "drawing" with SciDraw. That is, is walks you through using SciDraw's drawing objects (rectangles, circles, arrows, brackets, axes, *etc.*) to make a scientific diagram. Along the way, you will learn the principles of how diagrams are labeled (neatly and easily!) with text using SciDraw. Although we will start with a more streamlined example, the principles are the same as used to generate the schematic diagram of a molecule on the front cover of this guide, in panel (b).

Tutorial 3. This tutorial focuses on data plots. Notice that panels (c) and (d) of the front cover illustrate plots of numerical data — assembled together with insets and annotations. These are generated with SciDraw's capabilities for styling and annotating data plots.

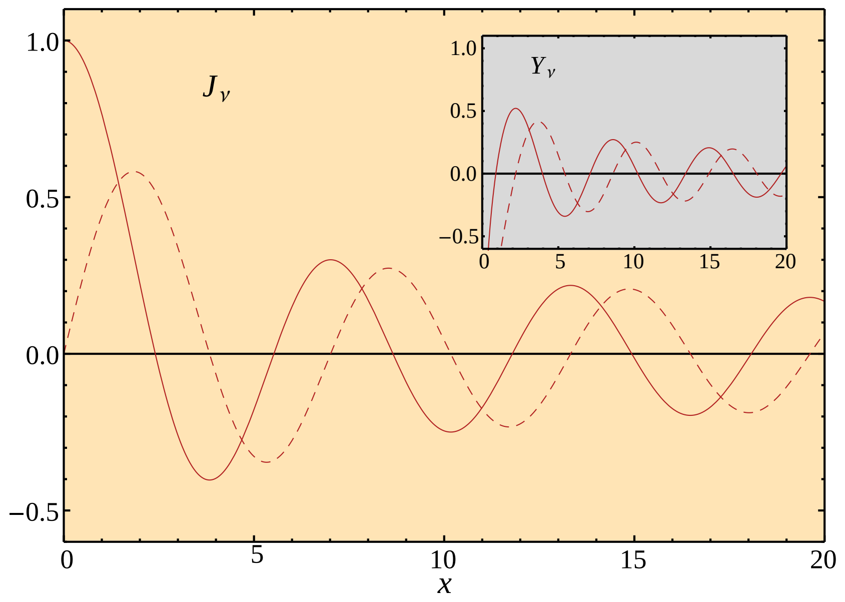
You may be wondering now about the level scheme diagram at the bottom of the front cover. Level schemes are a more specialized topic. They are covered separately in Sec. 4.4.

You can find all the example code in the notebook `Examples-Guide.nb`, which is included with SciDraw. You will want to follow along, running the code as you read the tutorial. You will also want to try out some simple modifications to the code, for instance, playing with the values of formatting options. We will *intentionally* digress a bit in the tutorials to cover some key ideas, so you can try these ideas out, as well.

3.1 Tutorial 1: Getting started with figures and panels

3.1.1 Problem statement

Here we will generate the figure seen in panel (a) — although now we will draw it as a figure on its own, not as part of a four-panel plot:



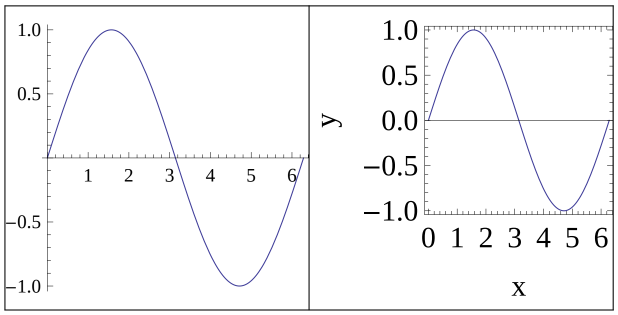
That is, we would like to plot the Bessel functions J_0 and J_1, in the main panel, as well as the Bessel functions Y_0 and Y_1, but smaller, in an inset panel. We also would like to place text labels "J_ν" and "Y_ν" in the figure, serving as titles to the main and inset panels, respectively.

3.1.2 Setting up the "canvas"

The first step in setting up this figure is in fact common to all figures in SciDraw. Much as an artist might, we must first set up a "canvas" on which to draw the figure (or perhaps a scrap of paper if we are less ambitious).

To understand the meaning and reason for this first step, we must first understand the problem. The normal Mathematica plotting commands always squeeze their *entire* output — the plot itself, plus frame and tick labels and axis labels, *etc.*, into an area of some given size — either the default size chosen by Mathematica or else as selected with the `ImageSize` option. So, if the axis labels or tick labels grow, the actual plot itself shrinks. Compare, for instance, these Mathematica two plots, both of which are supposedly "2 inches by 2 inches" (recall 1 inch is 72 printer's points) — in fact, the plot region itself is smaller in both cases, and *much* smaller in plot at right:

```
(* first plot *)
Plot[
  Sin[x], {x, 0, 2 * Pi},
  ImageSize → 72 * {2, 2}, AspectRatio → 1
 ];

(* second plot *)
Plot[
  Sin[x], {x, 0, 2 * Pi},
  Frame → True,
  FrameLabel → {"x", "y"}, FrameStyle → Larger,
  ImageSize → 72 * {2, 2}, AspectRatio → 1
 ];

(* now let's show them together *)
GraphicsGrid[{{%%, %}}, Frame -> All, Spacings -> 0]
```

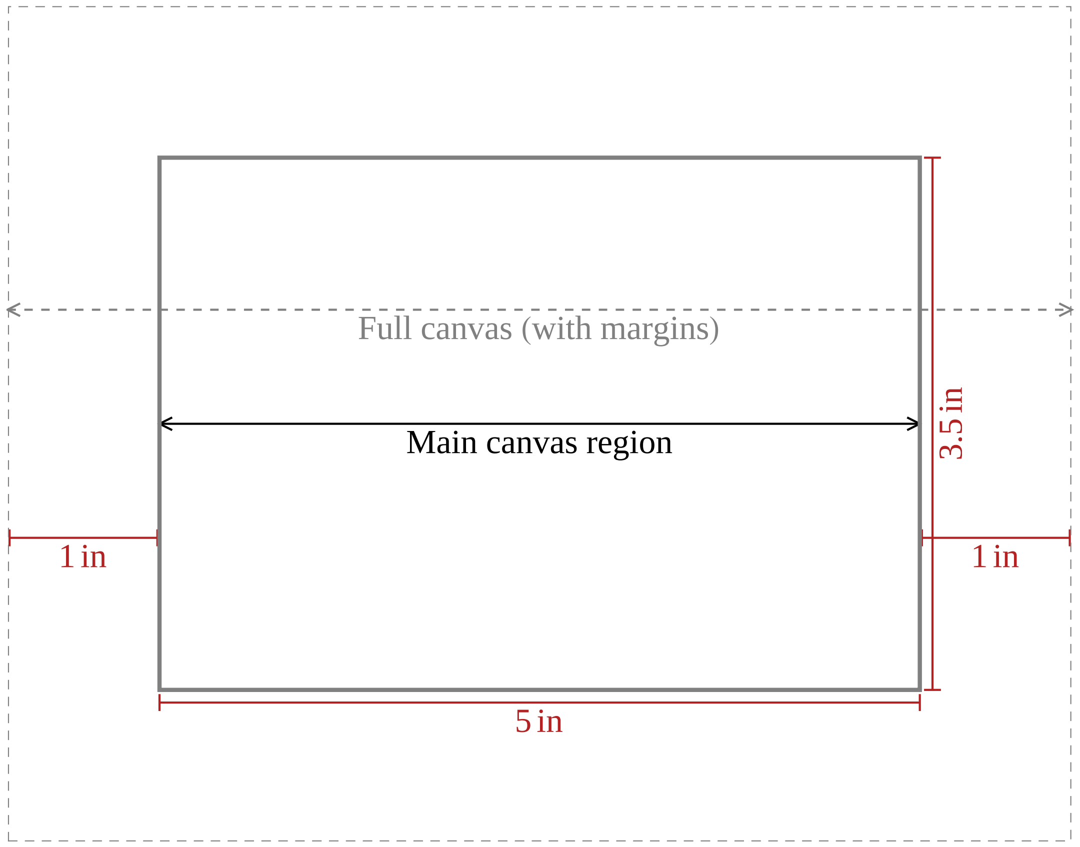


This automatic shrinking may not matter much for a simple plot which is meant to be displayed by itself. But it starts to become a nuisance if you are making several plots which should be the same size. Then it becomes a *real* nuisance in a complicated figure, where you have carefully and delicately placed text labels around the curves and diagrams — which are then thrown off when the figure shrinks, since font sizes do *not* shrink along with the curves.

Now, for our Bessel function figure, let's say we want the plot region to be 5 inches by 3.5 inches. We ask for a canvas which is this size by setting up a `Figure`, with option `CanvasSize->{5,3.5}`:

```
Figure[
 (* the actual body of the figure will go here *),
 CanvasSize → {5, 3.5}
]
```

The actual output from this is not much to look at — just a big blank rectangle (not shown here)! But the important thing to realize is that SciDraw will actually give us a canvas which is larger than the requested 5 inches by 3.5 inches, by a 1 inch margin on each side. This means there is plenty of room for tick labels and frame labels to fit out in the margin, and to shrink or grow as they will, without squeezing the plot. If you would ever like to explicitly see the boundaries of the "main canvas region" (the part you draw the plot itself in) and the "full canvas with margins" (where the frame labels will end up) outlined for you as you are drawing a figure, you can add the option `CanvasFrame->True` to `Figure`:



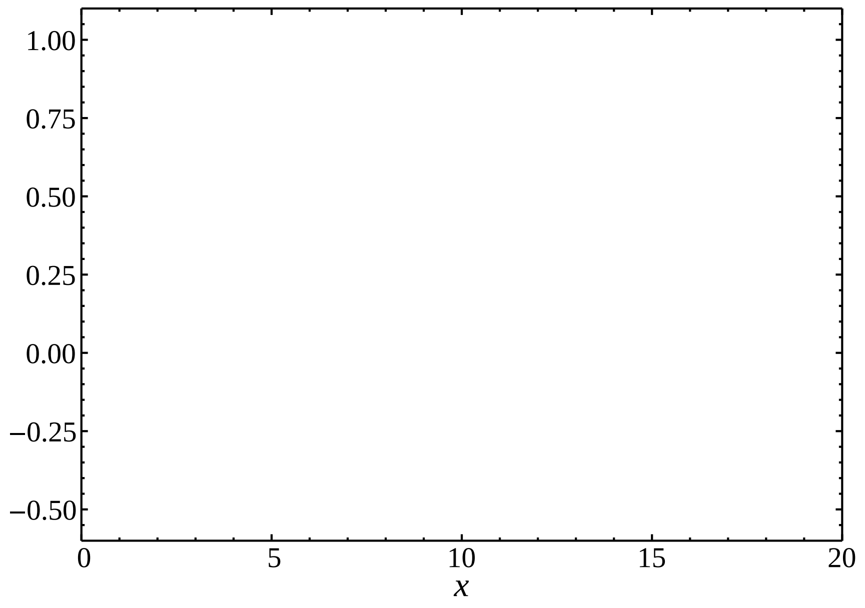
In fact, there are a few more options for `Figure`, summarized in Sec. 5 — you might wish to change the size of the margin (say, `CanvasMargin->0`, if you really don't need the extra space) or select an alternative unit to inches (say, `CanvasUnits->Furlong`, or maybe even something exotic like `CanvasUnits->Centimeter`).

3.1.3 Setting up the main panel frame

But, returning to our figure, while generating a blank rectangle is not a bad start, we still have a few more steps to go! The next step is also common to all SciDraw figures — setting up the panel in which the plot is to be drawn. (There may, in fact, be more than one panel, as on the cover, but that discussion can wait for later.) The panel is generated with `FigurePanel`:[1]

[1]Notice how `Figure` is a wrapper command for everything, while the `FigurePanel` "nests" inside the `Figure`. Similarly, the inset panel will nest within this main panel. If you are a LaTeX user, it might help to think of the `Figure` as the "document" and the `FigurePanel` as analogous to an "environment". This analogy will continue to be useful as we expand our familiarity with panels and related SciDraw constructs.

```
Figure[
 FigurePanel[
  {
   (* the actual body of the main panel will go here *)
  },
  XPlotRange -> {0, 20}, XFrameLabel -> textit["x"],
  YPlotRange -> {-0.6, 1.1}
 ],
 CanvasSize → {5, 3.5}
]
```



The first two things we think of when we set up a panel are the *ranges* for the coordinate axes and the *frame labels* for the coordinate axes. The ranges are set with the `XPlotRange` and `YPlotRange` options to `FigurePanel`. The labels — and these are optional — are set using the `XFrameLabel` and `YFrameLabel` options. (We will come back to the `textit["x"]` later, in Sec. 3.1.5, but you can probably figure out from the context that it gives an italic *x*, especially if you have used LaTeX.)

The other thing that you might notice is that we specify options for each of the axes — *x* and *y* — separately. This is different from the Mathematica plot functions, where you have to specify the properties for both the axes together at once, for instance:

```
PlotRange → {{0, 20}, {-0.6, 1.1}}, FrameLabel → {textit["x"], None}
```

SciDraw actually does accept the classic Mathematica style for the plot options as well, and you are free to use it. But, in practice, with SciDraw, you will probably find it easier to separate out the information for the two axes, as we have done here. This way, you can first think about the *x* axis, and enter all the information for it, then move on to the *y* axis. Then, you can also tweak the options for one axis without worrying about the options for the other. For instance, in our example we set the *x*-axis label without having to say anything about the *y*-axis, which didn't need a label.[2]

In fact, there are some more panel options we would like to adjust right now.

[2]The real importance of separating out the *x*-axis options and *y*-axis options comes later, when we graduate to multipanel plots. Then you will usually have several panels sharing the same *x*-axis properties (all the panels in the same column) and several sharing the same *y*-axis properties (all the panels in the same row), and it will be imperative

The general principle is that SciDraw provides reasonable (or at least that's the goal) defaults for properties, but then provides systematic ways of overriding these defaults through options.

The full set of options for `FigurePanel` is summarized in Sec. 10.1. First, though, there are many styling properties which can be specified not just for panels and panel labels, as we are discussing now, but also any text or graphics we might include in a figure — say, the color or font family or line thickness.[3] The options which control these common properties are summarized in Sec. 8.

For instance, by default, SciDraw draws text in the `"Times"` font at 16 point size. After staring at this figure, we might decide we would rather like a slightly smaller font for the frame labels, so we tell `FigurePanel` we want `FontSize->15`.[4]

Also, the tick marks on the y-axis, as we just drew them above, are ridiculously close together. By default, SciDraw follows Mathematica's choice for the tick marks, which is usually reasonable but of course will not always be ideal for any given figure. In fact, since here the major tick spacing is in steps of 0.25, the tick labels end up having *two* digits after the decimal, which gives a very "busy"appearance. SciDraw provides much finer control over tick marks, through the `LinTicks` and `LogTicks` functions.[5] In this example, the ticks need to run from -1 to 1. Steps of 0.5 between major ticks seem about right, and would save us a digit after the decimal place. Then having maybe 5 minor ticks, *i.e.*, in steps of 0.1, would be plenty. So the full option is `YTicks->LinTicks[-1,1,0.5,5]`.

And, in the figure we are trying to draw, notice that we also have a background color for the panel — this is accomplished with `Background->Moccasin`. Actually, there are many ways of naming colors in Mathematica (see **guide/Colors**). You can use any of these with SciDraw. For drawing purposes, it is very convenient to refer to the large set of named colors which were provided in early Mathematica versions. For instance, at least I personally find `Moccasin` and `Firebrick` to be more descriptive and easier to remember than `RGBColor[1.,0.894101,0.709799]` and `RGBColor[0.698004,0.133305,0.133305]`. The named colors were phased out with Mathematica 6,[6] but SciDraw makes these names available for easy use. You can view a convenient palette of these colors by clicking the "View color palette" button on the startup message SciDraw displays, or by entering `NamedColorPalette[]` at any time.

to be able to provide these separately.

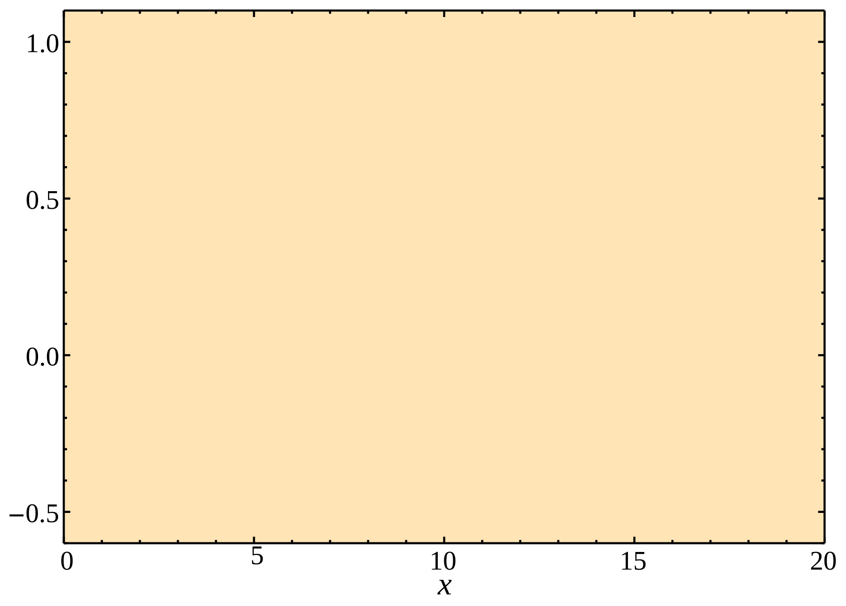
[3]The basic properties are the same ones that in Mathematica you would conventionally control with font options to `Style` (see **guide/FontOptions**) or graphics directives (see **guide/GraphicsDirectives**), so it would help for you to read up on those topics if you haven't already.

[4]In fact, in order for a figure to look right, the font for the tick labels should typically be $\sim 20\%$ smaller than that for the frame label. SciDraw takes care of this automatically. But, as you might guess, this choice, too, you can control by options!

[5]The tick mark control is provided by the CustomTicks package. This package is included as part of SciDraw, but it is a stand-alone package which can also be loaded on its own, and used with any Mathematica graphics function which accepts the `Ticks` option (see **ref/Ticks**). It allows you to contruct sets of linear, logarithmic, or even general nonlinear tick marks for use with the `Ticks` option. For the full story, see the separate CustomTicks guide (`CustomTicksGuide.pdf`) included with SciDraw.

[6]See **Compatibility/tutorial/Graphics/Colors** for the full story.

```
Figure[
 FigurePanel[
  {
   (* the actual contents of the main panel will go here *)
  },
  XPlotRange -> {0, 20}, XFrameLabel -> textit["x"],
  YPlotRange -> {-0.6, 1.1},
  YTicks -> LinTicks[-1, 1, 0.5, 5],
  FontSize → 15,
  Background → Moccasin
 ],
 CanvasSize → {5, 3.5}
]
```



3.1.4 Interlude: Panels and coordinates

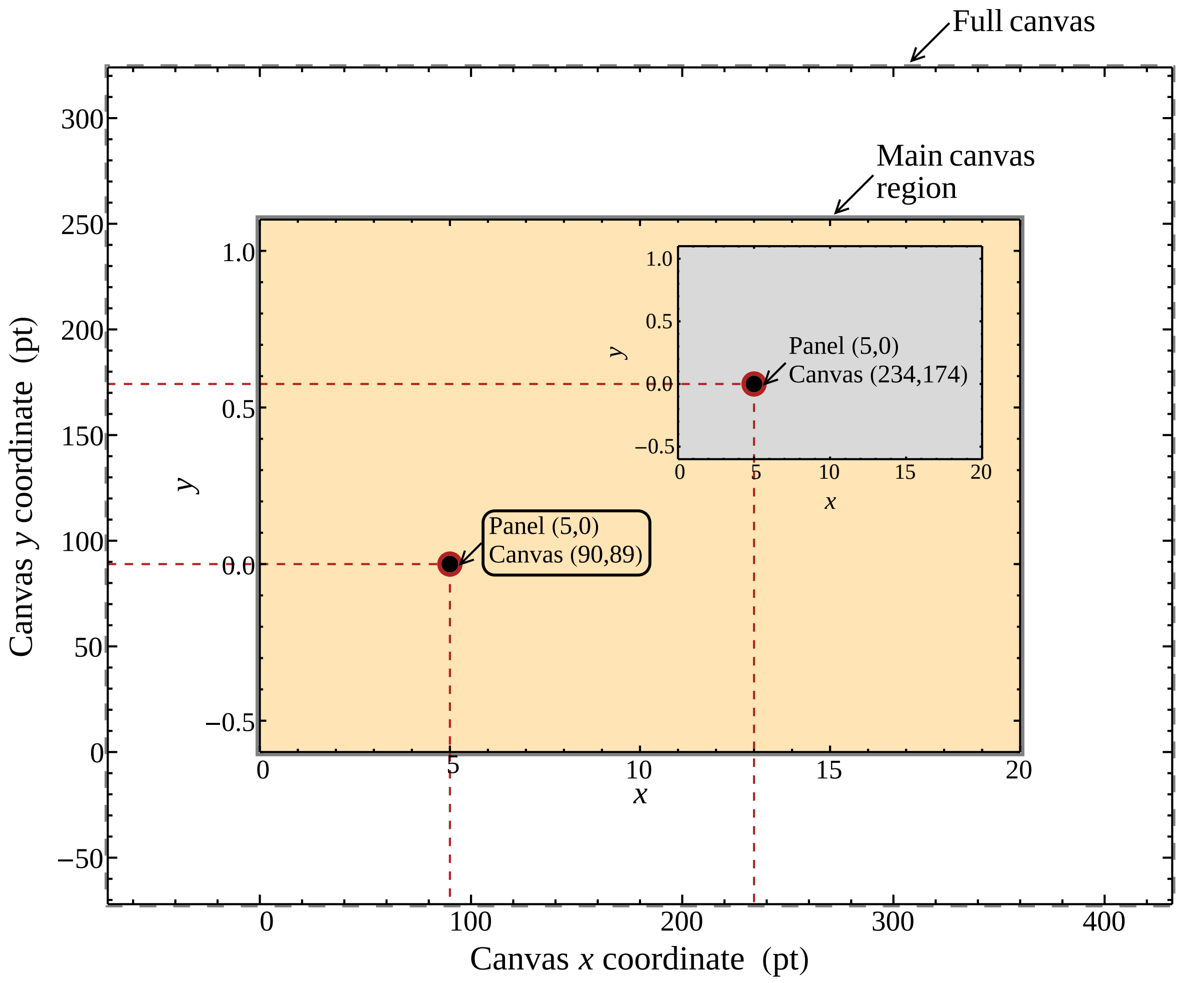
It is worth stepping back now to see what we really accomplish by setting up the panel. There is the concrete, *visible* aspect of drawing the frame (the frame edges themselves, tick marks, tick labels, and frame labels), background, and perhaps a panel letter, as well. But there is an equally important *invisible* aspect. By defining the plot ranges for the axis, we are defining the mathematical coordinate system for everything which is plotted or drawn within the panel. The `FigurePanel` gives us a "window" onto this mathematical world — it sets up that the mathematical coordinates $x \in [0,20]$ and $y \in [-0.6,1.1]$ should map onto this rectangular region of the canvas. In a little while, when we set up the panel for the *inset* at the upper right of the figure, we will again set up a window into a different world, where the coordinates also run over these same ranges. But the point $(5,0)$, say, in the main panel ends up at a very different point on the canvas than the point $(5,0)$ in the inset panel. It is up to SciDraw to map points in these various panel coordinate systems onto points on the big canvas — which, in the end, is all that Mathematica knows about or understands when it displays the `Figure` graphics. It will be helpful to keep this in mind later when we get into the nitty gritty of telling SciDraw where and how to place things in a figure.

Although you won't practically use this quite yet, it may be helpful for you to keep in the back of your mind that, at any given moment, we really have two ways of describing a point:

(1) the point's *canvas coordinates*, where it is physically on the canvas, *i.e.*, if you just took a ruler to the page and measured from the lower left hand corner, and

(2) the point's *panel coordinates*, *i.e.*, how we would describe it mathematically if we read off a position from the x and y axes marked on the panel's edges.

These different descriptions of the same point are illustrated in the following figure:



To be more precise about what we mean by canvas coordinates... As you can see from the outermost frame in this illustration, the *canvas coordinates* are measured in *printer's points*, and $(0,0)$ is the bottom left of the *main* canvas region.

If you wanted to connect the two circled points with a line, saying that the line goes from $(5,0)$ to $(5,0)$ wouldn't be very helpful! Having an underlying canvas, on which this line actually goes from $(90,89)$ to $(234,174)$, is crucial.

3.1.5 Interlude: Typesetting in Mathematica

When we set the x-axis label just now, we promised we would come back to the question of what we meant by `textit["x"]`... If you have used LaTeX, you probably recognize that `\textit` is the LaTeX command for italics. SciDraw provides several functions to help with formatting text labels in figures. These do not constitute anywhere as near an exhaustive framework as, say, LaTeX itself provides — but you might find some of them useful, and you will see them throughout the examples. A summary is given in Sec. 4.3.

More broadly, before we set out to label our figures, it would be helpful for us to take a moment to review how text formatting and mathematical typesetting work in Mathematica. In principle, Mathematica allows you to typeset virtually any mathematical expression you could imagine building. Therefore, you can also typeset just about any expression you could imagine as a label for a SciDraw figure. However, Mathematica does not give us anywhere near as fluent way a to do this formatting and typesetting as you might be used to in LaTeX. On the one hand, if you are a whiz with Mathematica's point-and-click palettes and keyboard shortcuts for typesetting,[7] then you are pretty well set. On the other hand, if you are like me and have very little patience for WYSIWYG editing — especially once expressions get a little more complicated, and you want to cut and paste and move parts of them around — you will instead probably find yourself using the approach which Mathematica uses interally to handle typesetting. This is more cumbersome but also more robust.

In Mathematica, typeset expressions are built out of boxes.[8] For example, an expression which you *see* on screen as, say, x^2 — an italic x with a superscript 2 — is represented *internally* as a Mathematica symbolic expression like any other

```
Superscript[Style["x", Italic], 2]
```

It is only when this expression is actually displayed that the notebook front end draws it as an italic x with a superscript 2. The functions `Superscript` and `Style` are called *box generator functions*. They don't actually *do* anythings themselves, but, when they appear in a symbolic expression like this one, they tell the notebook front end (or the `Export` function for graphics, *etc.*) how to format the expression.

Let us compare the two approaches for the simple case of our label for the x-axis — an italic "x". We could have simply entered the "x" in italics, from the keyboard, when we typed the string for the `XFrameLabel` option

```
XFrameLabel → "x" (* notice the italic x, not roman x *)
```

That is, we could have used Ctrl-I then x then Ctrl-I, or selected Format>Face>Italic from the menus.

This seems easy enough... Except, in old versions of Mathematica, if you did that in an input expression, like here, about the third or fourth time you opened the notebook, the string would be spontaneously corrupted and you would have to retype the whole thing from scratch (Mathematica bugs!). This has still happened to me as recently as Mathematica 9. For instance, I had `XFrameLabel->`*N* spontaneously become

```
XFrameLabel -> "\!\(\* StyleBox[\"N\",\nFontSlant->\"Italic\"]\)"
```

So I just don't even try it any more.

What's more, if you can reliably tell `"x"` from `"x"` on screen, you have a better I than eye (?). Otherwise, you are likely to slip up pretty often (assuming you care about such details, which you probably do, if you are bothering to learn SciDraw in the first place).

On the other hand, the way to do this by the Mathematica styling function `Style`

```
XFrameLabel → Style["x", Italic] (* or Style["x",FontSlant->Italic] *)
```

is a bit of a mouthful. Hence SciDraw's LaTeX-like shorthands

```
XFrameLabel → textit["x"]
```

[7]See **howto/EnterMathematicalTypesetting** or **guide/MathematicalTypesetting**.

[8]See **tutorial/RepresentingTextualFormsByBoxes** and **tutorial/FormattedOutput** for the concepts, then, in particular, see **ref/Row**, **ref/Subscript**, and **ref/Style** for a practical quick start.

To more fully illustrate the point of WYSIWYG *vs.* box input, let's look at how we might typeset an expression like $J_\nu(x)$ — although, in the end, we will actually choose a slightly simpler label for this figure. Either the WYSIWYG

```
YFrameLabel → "Jν(x)"   (*WYSIWYG *)
```

or box formatting

```
YFrameLabel → Row[{Subscript[textit["J"], "ν"], "(", textit["x"], ")"}]
  (* box formatting *)
```

approach works. You can take your pick. But you can expect the latter form in the examples which come with SciDraw.

Actually, aside from personal preference, you will find that typesetting by box generator functions turns out to be very powerful if you are programming your labels. This will be illustrated in the tutorial in Sec. **??**, and it really needs to wait until we have a little more experience drawing labels in figures. But, for a rough idea, say you had 20 different curves to label J_0, J_1, J_2, ..., J_{20}. (I guess that actually makes 21, doesn't it?) And you draw these curves inside a loop over `n` from 0 to 20. It will be a whole lot easier to type `Subscript[textit["J"],n]` once and for all, than to go back and label these 21 curves manually. Here's a simpler Mathematica example even before we learn how to draw and label curves[9]

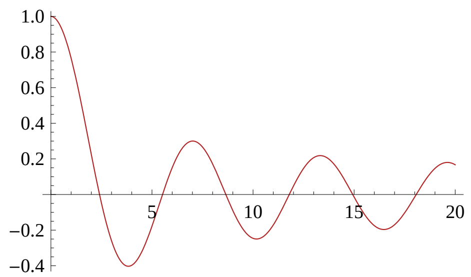
```
Table[
 Style[Row[{Subscript[textit["J"], n], "(", textit["x"], ")"}],
  FontFamily → "Times New Roman"],
 {n, 0, 20}
]
```

{$J_0(x)$, $J_1(x)$, $J_2(x)$, $J_3(x)$, $J_4(x)$, $J_5(x)$, $J_6(x)$, $J_7(x)$, $J_8(x)$, $J_9(x)$, $J_{10}(x)$,
 $J_{11}(x)$, $J_{12}(x)$, $J_{13}(x)$, $J_{14}(x)$, $J_{15}(x)$, $J_{16}(x)$, $J_{17}(x)$, $J_{18}(x)$, $J_{19}(x)$, $J_{20}(x)$}

3.1.6 Including a plot from Mathematica

Now that have a pretty good understanding of what we are doing as we set up the panel, we can get back to the business of drawing the figure. Plotting the Bessel functions in Mathematica is straightforward (it is assumed that you are familiar with `Plot`).

```
Plot[BesselJ[0, x], {x, 0, 20}, PlotStyle → Firebrick]
```

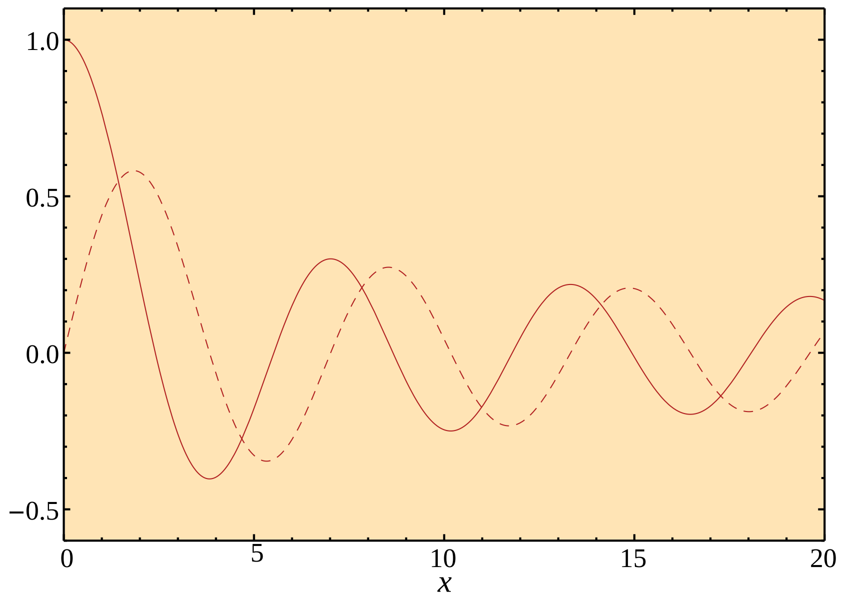


Then, incorporating this plot into the SciDraw figure is trivial. In fact, we can include any Mathematica graphics. We just "wrap" it with the SciDraw function `FigGraphics` and include it where we marked

[9]You aren't familiar with `Table`? This is some of the basic Mathematica you will want to read up on *now* (see **tutorial/RepetitiveOperations** and **tutorial/MakingTablesOfValues**), and it will pay off quickly.

before that `(* the actual body of the main panel will go here *)`.

```
Figure[
 FigurePanel[
  {

   (* plots *)
   FigGraphics[Plot[BesselJ[0, x], {x, 0, 20}, PlotStyle → Firebrick]];
   FigGraphics[Plot[BesselJ[1, x], {x, 0, 20}, PlotStyle → {Firebrick, Dashed}]];

  },
  XPlotRange -> {0, 20}, XFrameLabel -> textit["x"],
  YPlotRange -> {-0.6, 1.1},
  YTicks -> LinTicks[-1, 1, 0.5, 5],
  FontSize → 15,
  Background → Moccasin
 ],
 CanvasSize → {5, 3.5}
]
```



3.1.7 Adding some annotations: Labels and rules

Looking back at the figure we are trying to draw, we see that there are still some missing ingredients in the main panel. Now is where SciDraw's drawing and annotation tools come into play. For one thing, we want to insert a text label J_ν near the top left, to serve as a title for the panel. We already saw how to typeset this text, but not how to insert it into a figure. For another, we also need to draw a horizontal rule (*i.e.*, straight line segment) running across the panel where the x-axis would be (*i.e.*, at $y = 0$). These tasks allow us to meet our first two members of the zoo of SciDraw *drawing objects*.

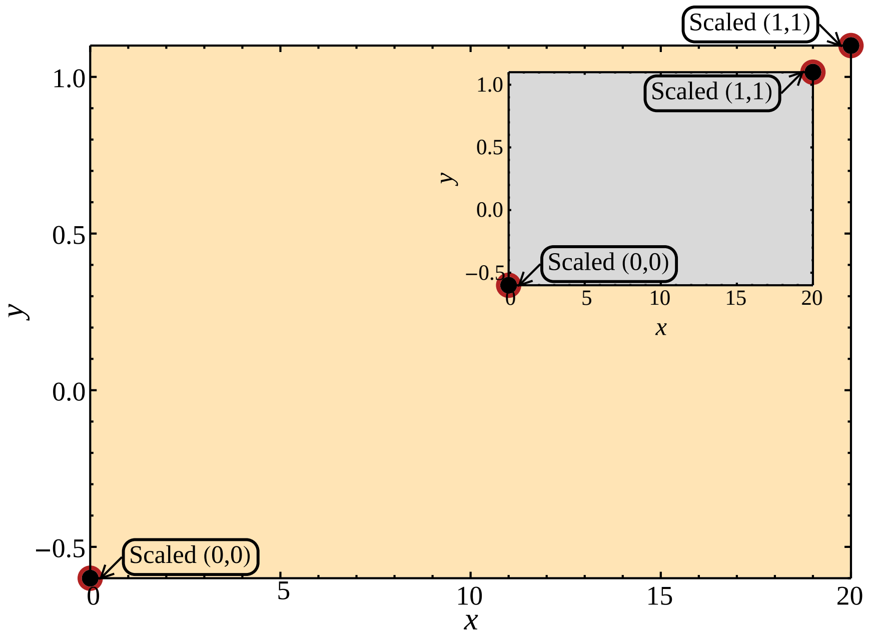
First, the *label*. You will insert *lots* of labels in your lifetime. There are two essential pieces of information which you must specify — *where* and *what*. The basic syntax is `FigLabel[`*p*`,`*text*`]`, where *p* is the point where we want the label to go.[10]

By eye, it looks like we want to put the label $\sim 20\%$ of the away *across* the figure and $\sim 85\%$ of the way up the figure.[11] We *could* try to convert this position into mathematical coordinates within the panel. Reading off the axes, we would decide that the coordinates should be $(4, 0.845)$ or thereabouts.

```
FigLabel[{4, 0.845}, Subscript[textit["J"], "ν"], FontSize → 15]
```

This will work. But it is not ideal. First, determining the panel coordinates (reading off the axes) is a tedious extra step. Worse, in the process of preparing any given figure, you are likely to have to adjust the coordinate axis ranges many times. It would be quite inconvenient to have the panel title label move around every time you do so! Even worse, we often want to put labels in corresponding positions, for consistency, across several panels or even several figures. This is well nigh impossible if the coordinate ranges for the panel axes may be entirely different.

The solution is that we have yet another way to describe the coordinates of a point — by where it is fractionally within the panel, from left to right and from bottom to top. These fractions are known as the *scaled* coordinates. You might already be familiar with Mathematica's `Scaled` coordinate notation.[12] These coordinates run from 0 to 1 across a Mathematica plot. SciDraw generalizes this idea so that scaled coordinates run from 0 to 1 across each individual panel of a figure. For instance, the bottom left is `Scaled[{0,0}]`, the center is `Scaled[{0.5,0.5}]`, and the top right is `Scaled[{1,1}]`. Let us repeat part of our earlier diagram on the topic of coordinates, to illustrate this:



Thus, to put our label $\sim 20\%$ of the way *across* the figure and $\sim 85\%$ of the way *up*, we use

[10]Whenever you are ready, you can refer to Sec. 13 for the not-so-basic syntax. Actually, the most important place to look first is in Sec. 8, where you will find various options for framing and positioning text. These options can be applied to *any* text in a figure, not just a `FigLabel`. We will have more to say about labels in Sec. **??** of the following tutorial.

[11]When some users try visualize the "fraction" of the way across the panel where something should go, apparently they find that it helps to let their eyes blur out a bit, so they can view the panel as a big empty rectangle, without being distracted by the curves and such. I don't actually do this literally, but maybe it will work for you. In any case, the mindset is useful.

[12]See **ref/Scaled**.

```
FigLabel[Scaled[{0.2, 0.85}], Subscript[textit["J"], "ν"], FontSize → 15]
```

Then, for drawing the horizontal *rule*. There are actually many ways we could accomplish this. SciDraw lets us draw any line (or curve connecting a series of points) as a `FigLine` object, `FigLine[`$\{p_1, p_2, \ldots, p_n\}$`]` (see Sec. 11.1).[13] So, we could use

```
FigLine[{{0, 0}, {20, 0}}]
```

Or, being clever, and realizing this curve should run from the far left to the far right, we could use scaled coordinates along the x-axis, as

```
FigLine[{{Scaled[0], 0}, {Scaled[1], 0}}]
```

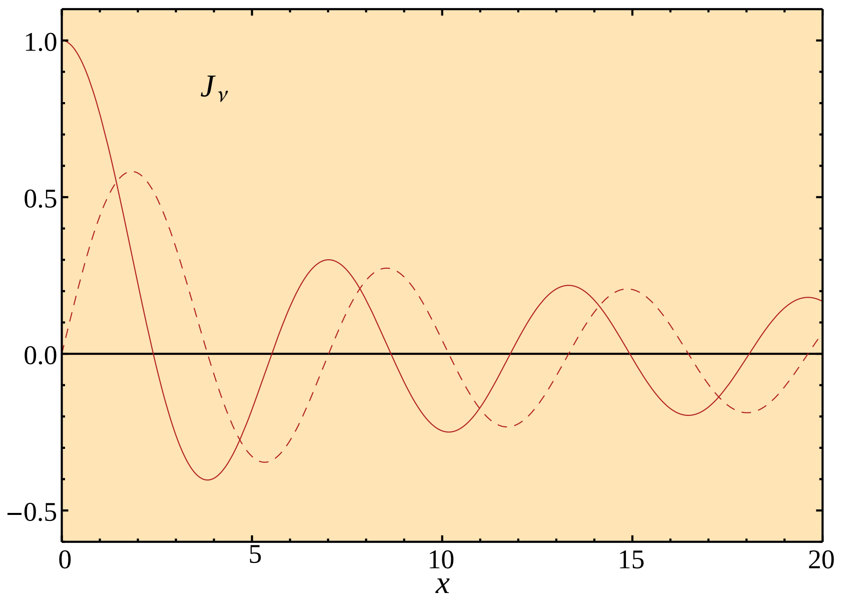
But horizontal and vertical rules show up *so* often that I finally got tired of thinking this through and just added a new drawing object `FigRule`. This has the form `FigRule[Horizontal,`$y, \{x_1, x_2\}$`]` or `FigRule[Vertical,`$x, \{y_1, y_2\}$`]` (see Sec. 13.3). Even more simply, we can alternatively specify that the range should be `All`, to obtain a rule which extends the full panel width or height. Thus, for the rule at $y = 0$, we use

```
FigRule[Horizontal, 0, All]
```

Collecting everything so far, we have:

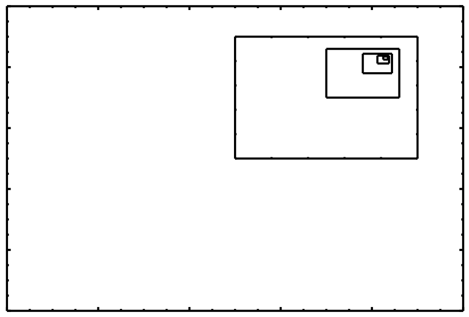
```
Figure[
 FigurePanel[
  {

   (* panel label *)
   FigLabel[Scaled[{0.2, 0.85}],
    Subscript[textit["J"], "ν"], FontSize → 15];

   (* horizontal rule *)
   FigRule[Horizontal, 0, All];

   (* plots *)
   FigGraphics[Plot[BesselJ[0, x], {x, 0, 20}, PlotStyle → Firebrick]];
   FigGraphics[
    Plot[BesselJ[1, x], {x, 0, 20}, PlotStyle → {Firebrick, Dashed}]];

  },
  XPlotRange -> {0, 20}, XFrameLabel -> textit["x"],
  YPlotRange -> {-0.6, 1.1},
  YTicks -> LinTicks[-1, 1, 0.5, 5],
  FontSize → 15,
  Background → Moccasin
 ],
 CanvasSize → {5, 3.5}
]
```

[13] The idea is the same as for the Mathematica graphics primitive `Line` (see **ref/Line**).



3.1.8 Setting up the inset panel

Finally, we are ready to add the *inset panel*. There is not much to this, now. We already know how to make a panel with `FigurePanel`. Panels can be “nested”, one inside another. That is, you can include an inset panel as part of the contents of an outer panel, just as we included graphics, labels, and rules above. In fact, you can do so indefinitely, in an infinite regress, if you are really in need of entertainment.



The only new piece of information which we must specify is the *region* which the panel should cover. This is given as the `PanelRegion` option. In SciDraw, rectangular regions are described just as in the usual Mathematica `PlotRange` option, as $\{\{x_1,x_2\},\{y_1,y_2\}\}$. We would like the inset panel in the figure we are working on to fill an area running from $\sim 55\%$ to $\sim 95\%$ of the way along the x and y directions of the panel. Once again, the tedious and ultimately unreliable way would be to to try to read off the positions on the axes and specify this range in terms of the panel coordinates — say `{{10,20},{0.5,1.0}}`. However, more naturally, SciDraw also allows the region to be specified in scaled coordinates.[14] So, we can simply use

[14]For more on specifying regions, see Sec. 7.4.

```
FigurePanel[
  {
   (* the actual body of the inset panel will go here *)
  },
  PanelRegion -> Scaled[{{0.55, 0.95}, {0.55, 0.95}}]
 ];
```

Then let's not forget the usual panel options, for the axes. We will choose a smaller font size for the labels on this smaller inset panel (`FontSize->12`) and a new background color as well. Then the contents of the panel really introduce nothing new. We draw them as before, but now using the functions Y_ν in place of J_ν.

3.1.9 The final figure

Putting it all together, the code for the figure is something like:

```
Figure[
 FigurePanel[
  {

   (* panel label *)
   FigLabel[Scaled[{0.2, 0.85}], Subscript[textit["J"], "ν"], FontSize → 15];

   (* horizontal rule *)
   FigRule[Horizontal, 0, All];

   (* plots *)
   FigGraphics[Plot[BesselJ[0, x], {x, 0, 20}, PlotStyle → Firebrick]];
   FigGraphics[Plot[BesselJ[1, x], {x, 0, 20}, PlotStyle → {Firebrick, Dashed}]];

   (* inset panel *)
   FigurePanel[
    {
     (* panel label *)
     FigLabel[Scaled[{0.2, 0.85}], Subscript[textit["Y"], "ν"], FontSize → 12];

     (* horizontal rule *)
     FigRule[Horizontal, 0, All];

     (* plots *)
     FigGraphics[Plot[BesselY[0, x], {x, 0, 20}, PlotStyle → Firebrick]];
     FigGraphics[Plot[BesselY[1, x], {x, 0, 20}, PlotStyle → {Firebrick, Dashed}]];


    },
    PanelRegion -> Scaled[{{0.55, 0.95}, {0.55, 0.95}}],
    XPlotRange -> {0, 20},
    YPlotRange -> {-0.6, 1.1},
    YTicks -> LinTicks[-1, 1, 0.5, 5],
    FontSize → 12,
    Background → LightGray
   ];

  },
  XPlotRange -> {0, 20}, XFrameLabel -> textit["x"],
  YPlotRange -> {-0.6, 1.1},
  YTicks -> LinTicks[-1, 1, 0.5, 5],
  FontSize → 15,
  Background → Moccasin
 ],
 CanvasSize → {5, 3.5}
]
```

3.1.10 Supplement: Inserting Mathematica graphics (including three-dimensional plots)

Before we abandon this example, this would be a good place to illustrate an important point regarding the inclusion of Mathematica graphics within a figure, which it will help you to be aware of: There are really *two very different ways* in which you might want to include graphics in a figure: one which naturally involves mathematical coordinates, and one which doesn't.

Sometimes, as with the plot of a function, the graphics intrinsically lies in a *mathematical* coordinate system — which we need to make sure gets aligned with the axes on our panel. For instance, here, that first peak of the Bessel function had better lie at the panel coordinates $(1,0)$, or that first node at $(2.40,0)$.

In fact, it is informative to look at what the output of `Plot` really is, as a Mathematica `Graphics` expression. Let us inspect it by displaying it in `InputForm`, so that the front end actually shows us the expression instead of rendering it as an image:[15]

```
Plot[BesselJ[0, x], {x, 0, 20}, PlotStyle → Firebrick] // InputForm

Graphics[{{{
  Hue[0.67, 0.6, 0.6],
  Line[{
    {0., 1.}, {0.006134358411192534, 0.9999905924338464},
    ⟨⟨⟨ ... lots of points omitted here ... ⟩⟩⟩
    {19.993135319588603, 0.16747959167242185},
        {19.999999591836733, 0.16702469161939587}
  }]
  }}},
  {
    AspectRatio -> GoldenRatio^(-1),
    Axes -> True, AxesLabel -> {None, None},
    ⟨⟨⟨ ... some plot display options omitted here ... ⟩⟩⟩
    PlotRange -> {{0, 20}, {-0.40275898943117655, 1.}}}
  }
 ]
```

We see that the main result of `Plot` is really just a list of mathematical coordinates of two-dimensional points describing the curve — that is, contained in a "`Line` primitive", with some cosmetic wrapping put around it describing how this curve should be displayed.

In a case such as this, where mathematical coordinates are involved, `FigGraphics` is the right SciDraw object to use to incorporate the graphics into the SciDraw figure. `FigGraphics` takes care of transforming the curve to the right position on the canvas, so that its mathematical coordinates align with the panel's coordinate axes.

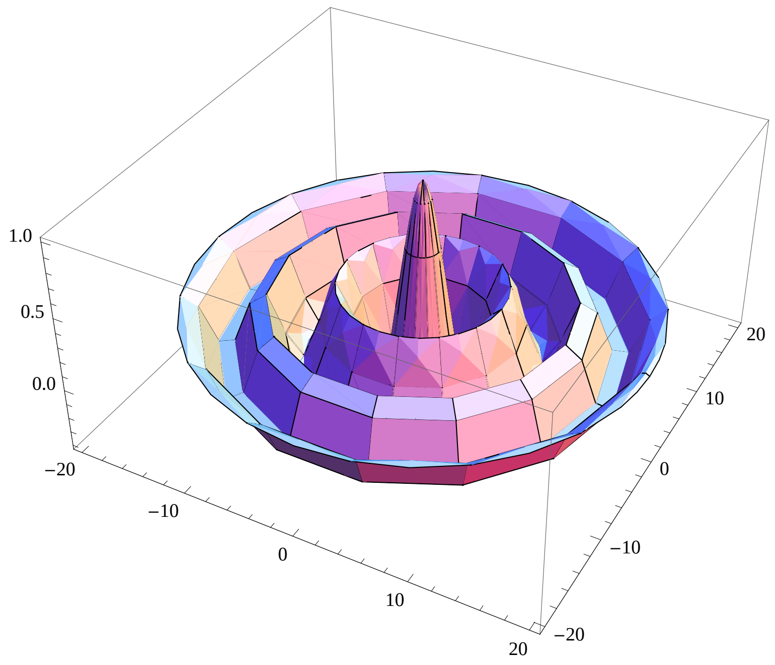
However, there are other types of graphics which we might want to include in a figure — such as a rasterized photographic image, or a three-dimensional plot which we are embedding in a two-dimensional figure — which don't have any natural association with mathematical coordinates at all. In this case, there is another SciDraw object, `FigInset`, which is the one we want to use — `FigInset` just "shoves" or

[15]Graphics in Mathematica are represented internally as a `Graphics` expression, which contains a list of graphics *primitives* such as lines, points, and text. Only when this expression is passed back to the front end, to be displayed, is it actually rendered as an image. It would actually be very good background for you to read **guide/SymbolicGraphicsLanguage** to learn what all the parts of this expression really mean. The more you understand about Mathematica graphics, the more you can successfully work with it.

insets the included graphics into whatever rectangular region on the figure we choose.[16] The syntax is `FigInset[`*graphics*`,`*region*`]`.[17]

It is worth taking a moment to illustrate the use of `FigInset`, and also to see how easy it is to insert three-dimensional plots into a figure. Suppose we wanted to include a surface plot of $J_0(r)$ in the figure (we will do a polar three-dimensional surface plot using `ParametricPlot3D`, but the details are not important right now):[18]

```
ParametricPlot3D[
 {r * Cos[phi], r * Sin[phi], BesselJ[0, r]},
 {r, 0, 20}, {phi, 0, 2 * Pi},
 BoxRatios → {20, 20, 10}
]
```



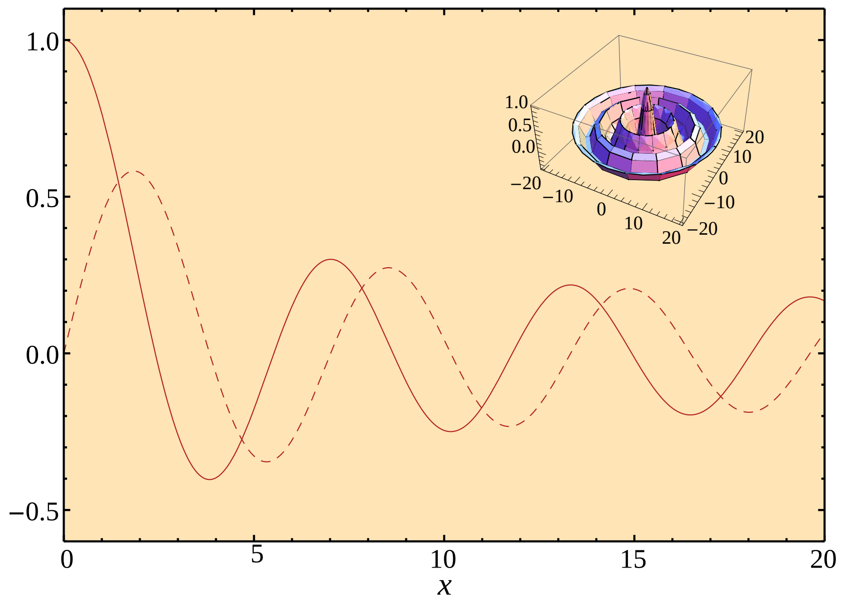
It would be nice to insert this figure so that it covers the same region as the inset panel did in our example — from the $\sim 55\%$ point to $\sim 95\%$ point horizontally and vertically on the figure, so we try

[16]The name `FigInset` comes from the Mathematica `Inset` primitive (see **ref/Inset**), which provides this functionality. Insetting in this sense is not to be confused with the idea of an inset panel, which, as we saw, is simply created using `FigurePanel`.

[17]See also Sec. 14 of the reference manual for more on `FigGraphics` and `FigInset`.

[18]Actually, you should be warned that three-dimensional surface graphics in Mathematica, ever since Mathematica 6, by default uses newer graphics features (smooth shading and adaptive sampling) which are not well-supported by the PostScript and PDF formats. The features allow the surfaces to obtain a very "smooth" apearance on screen (*not* shown in this guide). However, the result is that, if you are planning on obtaining EPS or PDF output for publication (as you very likely are), you will end up with tremendous file sizes. You can disable the problematic features, and trade off some of this smoothness for a much smaller file size, by giving `Plot3D` or `ParametricPlot3D` the options `MaxRecursion->2` (you can play with this number to obtain a suitable compromise between size and quality), `Mesh->Full`, and `NormalsFunction->None`. That is what we have actually done to generate the output in this tutorial, as you can inspect in `Examples-Guide.nb`.

```
FigInset[
  ParametricPlot3D[
   {r * Cos[phi], r * Sin[phi], BesselJ[0, r]},
   {r, 0, 20}, {phi, 0, 2 * Pi},
   BoxRatios → {20, 20, 10}
  ],
  Scaled[{{0.55, 0.95}, {0.55, 0.95}}]
 ];
```



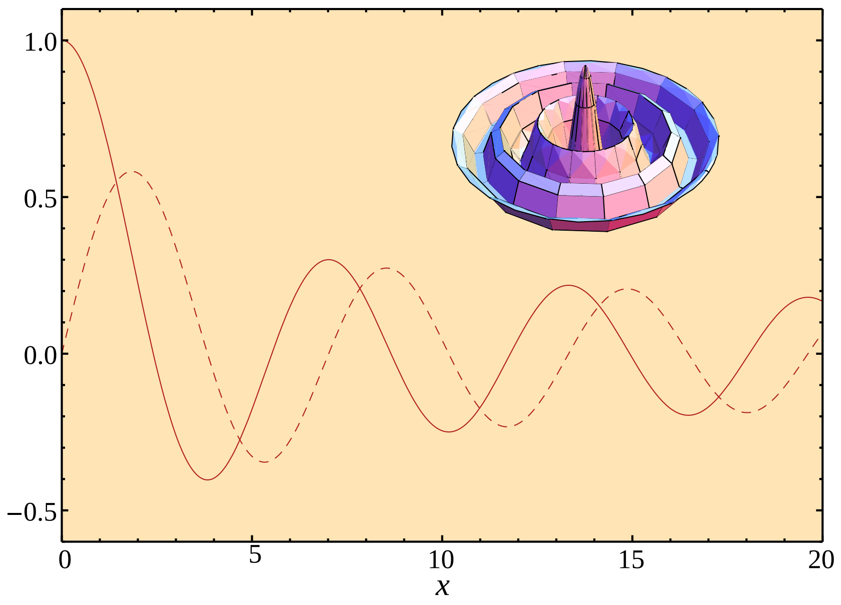
Oops... Maybe that is not quite what we were trying for! `FigInset` shows *everything* which Mathematica would display for the graphics. This includes the three-dimensional axes and frame box — and a whole lot of white space, since it turns out that Mathematica inscribes the plot in a three-dimensional box which is in turn inscribed in a two-dimensional rectangle! We can turn off the box with `Axes->None` and `Boxed->False` when we generate the parametric plot.[19] We can also compensate for the whitespace by extending the region covered by this inset. For this, we could just enter a larger region by hand, but `FigInset` also gives us `RegionExtension` and `RegionDisplacement` options, which save us these calculations. For instance, `RegionExtension->Scaled[0.3]` expands the inset region by 30%.[20] Here we use

```
FigInset[
  ParametricPlot3D[
   {r * Cos[phi], r * Sin[phi], BesselJ[0, r]},
   {r, 0, 20}, {phi, 0, 2 * Pi},
   BoxRatios → {20, 20, 10}, Axes → None, Boxed → False
  ],
  Scaled[{{0.55, 0.95}, {0.55, 0.95}}],
  RegionExtension → Scaled[0.3], RegionDisplacement → Scaled[{-0.075, 0}]
 ];
```

[19]See **guide/GraphicsOptionsAndStyling**.

[20]See Sec. 14 for options to `FigInset`, and Sec. 7.4 for the underlying principles of adjusting regions.

Putting this all together, we have our new figure (you can find the full source code in `Examples-Guide.nb`):



3.2 Tutorial 2: Drawing a diagram [STUB]

This tutorial is under construction. Please check back later.

3.3 Tutorial 3: Getting started with data plots and legends [STUB]

This tutorial is under construction. Please check back later.

For now, there is a mini-tutorial at the beginning of the file `Examples-Plots.nb`, to get you started with data plots and legends — see the example entitled "Mini-tutorial: An introduction to data plotting". This notebook also contains several further examples, including code for panels (c) and (d) from the cover of this guide.

Then, please read Sec. 16 of the reference manual. This provides a thorough discussion of data plotting (not just the syntax, but also the concepts and some practical examples).

4 Topical discussions

4.1 Drawing objects

This section provides an overview of the drawing objects which SciDraw makes available for general use in figures. The SciDraw drawing objects uniformly have names beginning with `Fig`, for "figure". We can roughly (*very* roughly) divide these objects into basic drawing *shapes* and more specialized objects meant for use as *annotations* in a figure. The following subsections provide some simple commentary on using the basic drawing shapes (Sec. 4.1.1), arrows (Sec. 4.1.2), and annotations (Sec. 4.1.3). Other, more specialized drawing objects for use in level schemes are discussed separately in Sec. 4.4.

You can find the source code for all figures from this section in `Examples-Guide.nb`.

4.1.1 Basic drawing shapes

We first survey the basic drawing shapes. The basic shapes include:

– `FigLine` for *lines* (Sec. 11.1). These may be built from many points p_1, p_2, ..., p_n and therefore actually more generally represent *curves*. (In computer graphics terminology, these are *polylines*.)

– `FigPolygon` for *polygons* (Sec. 11.2). These are closed lines with fills.

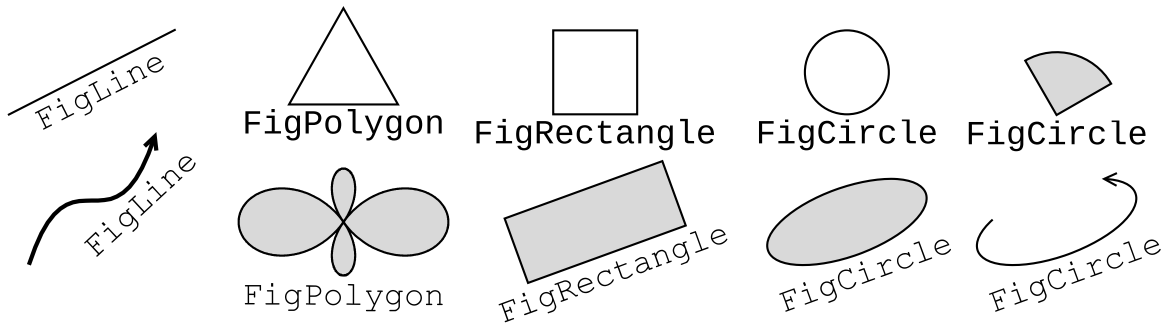
– `FigRectangle` for *squares* or *rectangles* (Sec. 11.3). True, any rectangle can be drawn as a polygon, by listing all four corner points. But there are more convenient ways to describe a rectangle, such as giving its center point and dimensions, and more convenient labeling approaches which can be defined for a rectangle (in term of left/right/top/bottom sides).

– `FigCircle` for *circles* or *ellipses* (Sec. 11.3). More generally, this category encompasses circular or elliptical arcs and pie wedges.

These are essentially enhanced versions of the Mathematica shape drawing primitives, but with outline, fill, and labels combined in one object. In particular, if you are familiar with the Mathematica symbolic graphics language, you will realize that these shapes correspond roughly to Mathematica's `Line`, `Polygon`, `Rectangle`, and `Circle`/`Disk` primitives, respectively.[1] There is a `FigPoint` object for drawing simple *points* or *dots* (Sec. 11.4). In terms of what can be drawn with it, `FigPoint` is basically redundant to a filled-in `FigCircle`. However, it is provided for completeness, as the SciDraw analog to the Mathematica `Point` primitive. *Spline curves* are also supported, through `FigBSpline` and `FigBezier` objects (Sec. 11.5).

Examples of `FigLine`, `FigPolygon`, `FigRectangle`, and `FigCircle` objects are shown in the following figure:

[1]See **guide/SymbolicGraphicsLanguage** or **guide/GraphicsObjects** for an introduction to Mathematica's graphics primitives.



The ease of use which these shape objects provide, with the machinery set up for controling their appearance through options, makes them useful for many diagramming, drawing, and plotting tasks. They support SciDraw's more powerful drawing capabilities — for instance, positions can be specified in any of SciDraw's coordinate systems or by "anchoring" to other objects.

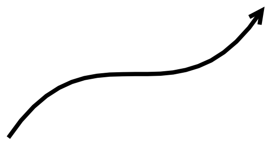
A fuller discussion of all these "extras" is given in the Reference Manual. In fact, you will probably wish to follow along in (Sec. 11) of the Reference Manual for the complete picture as you read the present section. In any case, you will find it helpful to refer to Table 11.1 for an overview of the syntax of these objects.

`FigLine.` `FigLine[`*curve*`]` produces an arbitrary open curve. A `FigLine` just has an outline but no fill.

Here *curve* is (more or less) just a list of points $\{p_1, p_2, \ldots, p_n\}$. However, the points within this curve specification are not limited just to your grandmother's $\{x, y\}$ coordinate pairs. It is worth checking out Secs. 7.1–7.2 for some more powerful possibilities.

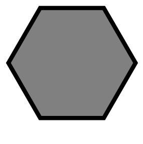
Arrow heads may be drawn on either end of the line, by specifying `ShowTail->True` or `ShowHead->True`, and the properties of these arrowheads (length and width) can be adjusted through options.

```
PointList=Table[{x,x^3},{x,-1,1,0.1}];
FigLine[PointList,LineThickness->2,ShowHead->True];
```



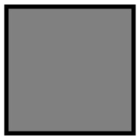
`FigPolygon.` `FigPolygon[`*curve*`]`, in contrast, may be used to draw a closed curve or, equivalently, a polygon. A `FigPolygon` has both an outline and a fill.

```
Sides = 6;
PointList = Table[
  {Cos[2*Pi*n/Sides], Sin[2*Pi*n/Sides]},
  {n, 0, Sides - 1}
  ];
FigPolygon[PointList, LineThickness -> 2, FillColor -> Gray];
```



`FigRectangle`. `FigRectangle` produces a square or rectangle. A `FigRectangle` has both an outline and a fill. Note that there are several different ways of specifying the rectangle's coordinates. It is usually most convenient to specify the *x* and *y* coordinate region covered by the rectangle as `FigRectangle[{{`x_1`,`x_2`},{`y_1`,`y_2`}}]`.

```
(* FigRectangle with region syntax *)
FigRectangle[{{-1,1},{-1,1}},LineThickness->2,FillColor->Gray];
```



This syntax is inspired by that of the Mathematica `PlotRange` option. It is actually a special case of a more general syntax `FigRectangle[`*region*`]`, where the *region* may be specified, *e.g.*, in terms of scaled coordinates, as described in Sec. 7.4.

An alternative syntax `FigRectangle[`p_1`,`p_2`]`, where p_1 and p_2 are diametrically opposed corner points is also provided for consistency with the syntax of the Mathematica `Rectangle` primitive. However, I have never found this syntax to be particularly intuitive.

```
(* FigRectangle with corner point syntax *)
FigRectangle[{-1,-1},{+1,+1},LineThickness->2,FillColor->Gray];
```

More useful is the alternative syntax borrowed from the traditional way of describing a circle. We can use `FigRectangle[`*p*`,Radius->`*r*`]` for a square or, more generally, `FigRectangle[`*p*`,Radius->{`r_x`,`r_y`}]` for a rectangle. Here *p* is the center point, and *r* is the half-width (*i.e.*, the perpendicular distance from the center out to any side), or $\{r_x, r_y\}$ are the horizontal and vertical half-widths, respectively.

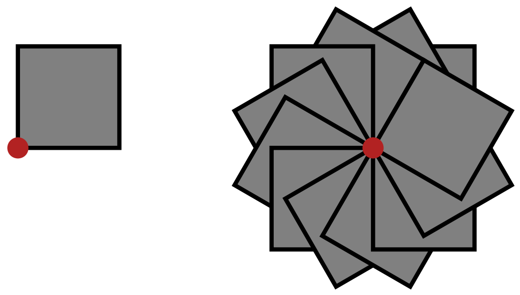
```
(* FigRectangle with center-radius syntax *)
FigRectangle[{0,0},Radius->1,LineThickness->2,FillColor->Gray];
```

Although the basic syntax of `FigRectangle` describes a rectangle which is happily aligned with the coordinate axes, it is also possible to subsequently rotate this rectangle, and you can choose the pivot point about which to rotate it. First, let us draw a single rectangle (okay, a square) with its *lower-left corner* at the origin (at left in following figure). We use the option `AnchorPoint->{-1,-1}` to indicate that the given point *p* should be interpreted as the lower-left corner rather than the center. This point is indicated by the red dot.

```
FigRectangle[
  {0, 0},
  Radius -> 1,
  AnchorOffset -> -1, -1,
  LineThickness -> 2, FillColor -> Gray
  ];
```

Then, let us illustrate rotating this square by several different angles, again around its lower-left corner (at right in the following figure).

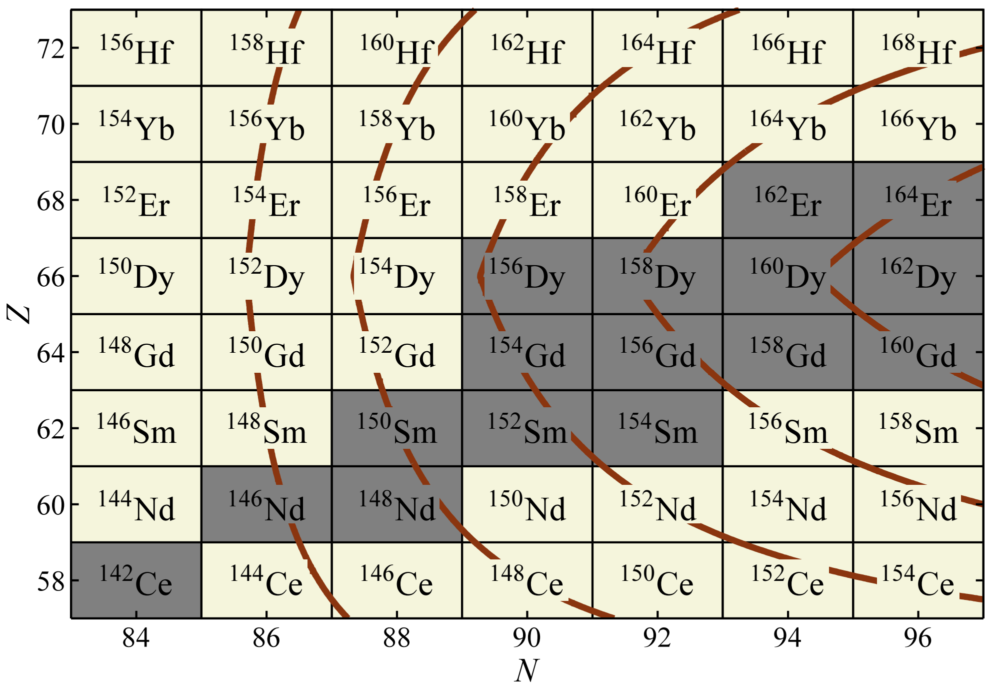
```
Angles = 12;
Do[
 FigRectangle[
  0, 0,
  Radius -> 1,
  AnchorOffset -> -1, -1,
  PivotOffset -> -1, -1, Rotate -> (2*Pi*n/Angles),
  LineThickness -> 2, FillColor -> Gray
  ],
 n, 0, Angles - 1
 ];
```



A rectangle can be used for various purposes within a figure. Naturally, it might represent a rectangular object in a diagram, as in Tutorial 2. However, it may also be used to frame or highlight a part of the diagram (say, a level in a level scheme), depending whether the outline, fill, or both are shown. It may conveniently be used for the construction of many kinds of block diagrams, tables, grids, and bar charts, since its ready-made outline, fill, and sundry labels cover most common features needed in table cells. In this regard, it is likely to be used in conjunction with some form of iteration, typically with `Do`, so as to automate the construction of a grid of boxes.

For instance, a table of nuclides can be created with the help of data provided by the Mathematica on-line `IsotopeData`, which provides the chemical symbol for each element and information on whether or not it is stable (among other trivia). Thus, the labeling of each square and shading of the stable isotopes can be automated, as in the following example. The full code for this example may be found in `Examples-Guide.nb`.

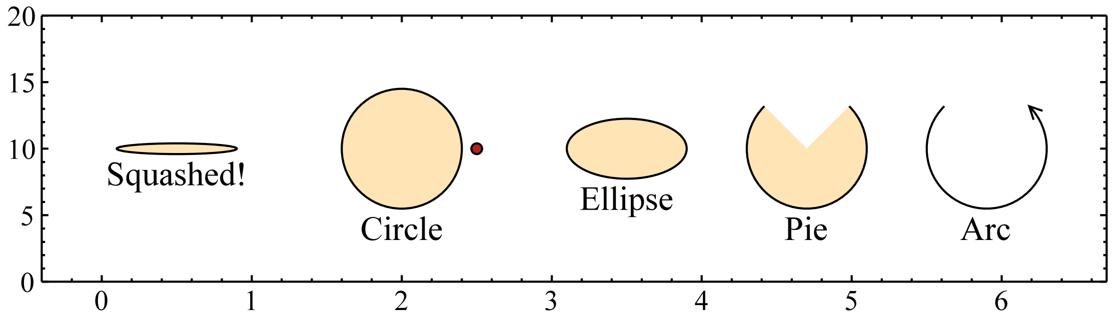
```
Do[
  IsotopeSymbol = IsotopeData[{ZZ, NN + ZZ}, "Symbol"];
  IsotopeStable = IsotopeData[{ZZ, NN + ZZ}, "Stable"];
  FigRectangle[
   {NN, ZZ},
   Radius -> 1,
   CenterLabel -> IsotopeSymbol,
   Style -> If[IsotopeStable, "StableIsotope", "UnstableIsotope"]
   ],
  {ZZ, ZMin, ZMax, 2}, {NN, NMin, NMax, 2}
  ];
```



`FigCircle`. The `FigCircle` object may be used to draw a circle or an ellipse. The syntax is `FigCircle[`p`,Radius->`r`]` for a circle, or, more generally, `FigCircle[`p`,Radius->{`r_x`,`r_y`}]` for an ellipse. A `FigCircle` may alternatively be used to draw an arc or sector, covering a specified range of angles, in which case arrowheads can also be drawn at either end of the arc.

Shapes which might technically be "circles" in terms of mathematical coordinates generally suffer from being distorted into ellipses when draw on the page (recall the example in (Sec. **??**) of Tutorial 2). For instance, a "circle" of radius 1 will extend out from its center by 1 unit with respect to the x axis, horizontally, and by 1 unit with respect to the y axis, vertically. But this will not look like a circle at all on the canvas unless the scales (*i.e.*, one unit equals how many inches?) for the horizontal and vertical axes happen to be identical! `FigCircle` allows the units in which the horizontal and vertical radii are given to be specified explicitly: for instance, both radii scaled according to the horizontal axis or both radii scaled according to the vertical axis. This facilitates the drawing of true circles and can be convenient for specifying ellipses as well. Thus, possible forms of the radii argument include `r`, `{r1,r2}`, `Horizontal[r]`, `Horizontal[{r1,r2}]`, `Vertical[r]`, and `Vertical[{r1,r2}]`.

```
FigCircle[{0.5, 10}, Radius -> 0.4,
 BottomLabel -> "Squashed!"];  (* not a circle at all! *)
FigCircle[{2.0, 10}, Radius -> Horizontal[0.4],
 BottomLabel -> "Circle"];
FigCircle[{2.5, 10}, Radius -> Vertical[0.4], FillColor -> Firebrick];
FigCircle[{3.5, 10}, Radius -> Horizontal[{0.4, 0.2}],
  BottomLabel -> "Ellipse"];
FigCircle[{4.5 + 0.2, 10}, Radius -> Horizontal[0.4],
  AngleRange -> {3*Pi/4, Pi/4 + 2*Pi},
  BottomLabel -> "Pie"];
FigCircle[{5.5 + 0.4, 10}, Radius -> Horizontal[0.4],
  AngleRange -> {3*Pi/4, Pi/4 + 2*Pi},
  ShowFill -> False, ShowHead -> True, BottomLabel -> "Arc"];
```



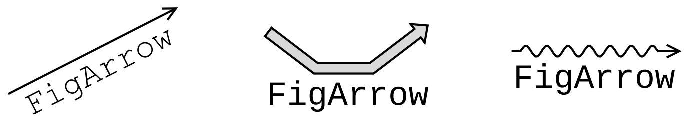
However, you might not need to bother with the mathematical axes at atll. If you simply want the radius to be a certain size on the page, in printer's points, without regard to any mathematical coordinates, you can specify the radius as `Canvas[`r`]` for a circle or `Canvas[`$\{r_x, r_y\}$`]` for an ellipse.

4.1.2 Arrows

Sometimes arrows might serve as basic elements of a diagram, but in other contexts they might serve as annotatations on the figure (*e.g.*, labeling parts of a diagram). Therefore, their place in the rough classification we have given for objects, as "shapes" or "annotations", a bit ambiguous. Although a simple arrow can be drawn just by putting an arrowhead at the end of a line, as with `FigLine`, the SciDraw also provides a more sophisticated `FigArrow` object for drawing more general types of arrows.

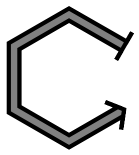
`FigArrow`. The basic syntax `FigArrow[`*curve*`]` matches that of `FigLine`. Depending on the arrow type, a `FigArrow` can consist of both an outline and a fill.

In its simplest form, `FigArrow` produces a curve with an arrowhead at the end — this is identical to what can already be drawn with `FigLine`. However, SciDraw also provides "block" arrows, double-shafted arrows, and "squiggle" arrows. The following figure shows some examples of arrows drawn with `FigArrow`:



It will actually be more convenient to discuss the options for generating these arrow types in the context of transition arrows for level schemes. Please refer to the discussion of the `Trans` object — which is essentially just a special type of `FigArrow` — in Sec. 4.4.3. An exhaustive discussion of the options for `FigArrow` is given in Sec. 12 of the Reference Manual. For now, though, here is one example which exercises several of the options for arrows.

```
FigArrow[PointList,
 ArrowType -> "DoubleLine", LineThickness -> 2, FillColor -> Gray,
 HeadLip -> 5,
 ShowTail -> True, TailLip -> 5, TailLength -> 0
 ];
```

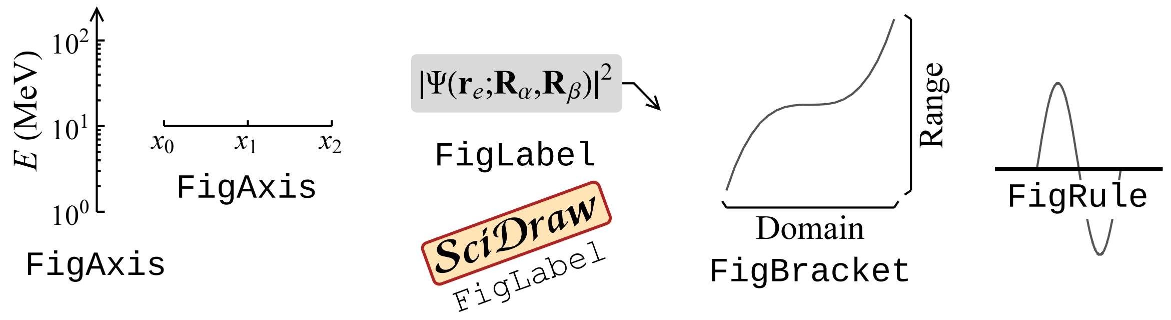


4.1.3 Annotation objects [STUB]

Finally, we survey the objects which serve primarily as "annotations". These objects include:

- `FigAxis` for *axes* (Sec. 10.4).
- `FigLabel` for *labels* (Sec. 13.1).
- `FigBracket` for *brackets* (Sec. 13.2).
- `FigRule` for horizontal or vertical *rules* (Sec. 13.3).

These objects are illustrated in the following figure:

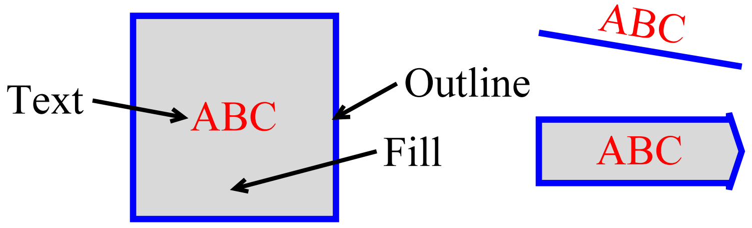


`FigAxis` is documented in the Reference Manual in the context of panels, in Sec. 10. The rest of these objects are documented together in Sec. 13. You will find it helpful to refer to Tables 10.18 and 13 for an overview of the syntax of these objects.

Coming soon... A brief discussion.

4.2 Standard options for objects

Every "object" in a figure is built from up to three distinct parts: an *outline*, a filled area or *fill*, and attached *text* labels, as illustrated in the following color-coded diagram.[2] Not all objects have all these parts — for instance, a `FigLine` has no filled area.



The appearance of an object is controlled by setting options for these parts. A couple of options affect the entire object, while the others affect only the outline, fill, or text. An exhaustive discussion, including a complete listing of available options, with a full discussion of allowed values of all these options, may be found in the Reference Manual. However, it may be helpful to give a brief introduction to the more essential options. These are listed in Table 4.1, grouped according to their general function:

- The options in the first group (`Show` and `Color`) affect the appearance of all parts of an object.
- The options in the second group (`ShowLine`, `LineColor`, ...) affect the appearance of just the outline.
- The options in the third group (`ShowFill`, `FillColor`, ...) affect the appearance of just the fill.
- The options in the fourth group (`ShowText`, `TextColor`, `FontFamily`, ...) affect the appearance of the text, including its font characteristics.
- The options in the fifth group (`TextBackground`, `TextFrame`, ...) instead control the appearance of a possible background fill behind the text, or frame around the text.
- The options in the sixth group (`TextOffset`, `TextOrientation`, ...) control the detailed positioning of text labels.
- The remaining options (`Style` and `Layer`) are more general in scope, applying to the object as a whole. The option `Style` is related to the powerful approach of using styles to uniformly control the appearance of similar objects (as introduced in Sec. **??**), while the option `Layer` allows fine tuning of which objects appear "in front of" or "behind" other objects (Sec. 8.1.9).

The color of the entire object (outline, fill, and text) can be set all at once with the option `Color`. Or the color can be controlled independently for the individual components by setting the `LineColor`, `FillColor`, or `FontColor` options.

How does this system for controlling colors — either all at once or separately — work? The options `LineColor`, `FillColor`, or `FontColor` have as their default the special value `Default`, which has a special meaning to SciDraw — that the color of this part of the object (the line, fill, or text, respectively) should be taken from the value for the `Color` option given to the object as a whole. Thus, *e.g.*, if we invoke

```
FigCircle[...,Color->Red]
```

[2]The ideas of *outline* and *fill* roughly correspond to those of *edge* and *face*, respectively, for a Mathematica graphics primitive (see, *e.g.*, **ref/EdgeForm** and **ref/FaceForm**).

Table 4.1 Some of the most commonly-needed options affecting the appearance of an object. See Sec. 8 for a complete listing.

Option	*Default*	
`Show`	`True`	Whether or not the object should be visible.
`Color`	`Black`	Default color to be used for all parts of object, unless overridden by `LineColor`, `FillColor`, `PointColor`, `TextColor`, *etc.*
`ShowLine`	`Default`	Whether or not the outline should be visible.
`LineColor`	`Default`	Color to be used for the outline.
`LineThickness`	`1`	Line thickness to be used for the outline.
`LineDashing`	`None`	Dashing to be used for the outline.
`ShowFill`	`Default`	Whether or not the fill should be visible.
`FillColor`	`Default`	Color to be used for the fill.
`ShowText`	`Default`	Whether or not the text should be visible.
`TextColor`	`Default`	Color to be used for the text.
`FontFamily`	`"Times"`	The font family.
`FontSize`	`16`	The font size, in printer's points.
`FontWeight`	`Plain`	The font weight (or boldness).
`FontSlant`	`Plain`	The font slant (or italicization).
`TextBackground`	`None`	The text background color.
`TextFrame`	`False`	Whether or not to show a frame line around the text.
`TextFrameColor`	`Default`	Color to be used for the text frame line.
`TextMargin`	`None`	The margin, in printer's points, between the text and the edge of its background coloring and frame line.
`TextOffset`	`Automatic`	The offset of the anchor within the text rectangle.
`TextOrientation`	`Automatic`	The text orientation, or rotation about the anchor.
`TextRectify`	`True`	Whether or not to automatically rectify text which would, otherwise, appear inverted.
`TextNudge`	`None`	An additional arbitrary displacment of the text, in printer's points.
`TextBuffer`	`None`	An additional displacement of the text, in printer's points, in the direction from the anchor point to the text center.
`Style`	`None`	The style (or list of styles) from which all option defaults should be taken.
`Layer`	`Automatic`	The drawing layer of the object.

this does not *directly* make the circle red. Rather, it sets `Red` as the default color for the outline, fill, or text. And each of these components can either respect that color (if left as `Default`) or take on a different color (if individually overridden, *e.g.*, by `FillColor->Moccasin`).

Colors may be specified using either the standard Mathematica color names (`Red`, `Blue`, *etc.*) or directives (`GrayLevel[...]`, `RGBColor[...]`, *etc.*).[3] SciDraw also makes available the much larger "legacy" set of color names, such as `Moccasin` and `Firebrick`, which were defined in early versions of Mathematica. When SciDraw is loaded, it displays a button labeled "View color palette" in the notebook.

[3]See **guide/Colors**.

You can view a chart of the named colors (and paste them into your notebook) at any time by clicking on this button.

Similarly, it is possible to specify that the outline, fill, or text of the object not actually be drawn, by setting `ShowLine->False`, `ShowFill->False`, or `ShowText->False`, or even that the object as a whole not be drawn, by setting `Show->False`. For instance, hiding the fill with `ShowFill->False` makes the object transparent, so objects behind can show through. This is not the same as simply making the fill the same color as the background, since then it would still block any objects behind it from view. An identical effect may be achieved by setting the color to `None`, but sometimes a simple boolean switch like `Show` is more convenient to work with.

The thickness and dashing of the outline are controlled by the options `LineThickness` and `LineDashing`, respectively. Thus thickness is given in printer's points, *e.g.*, `LineThickness->2` sets the thickness to 2 printer's points. The dashing is likewise given in printer's points, *e.g.*, `LineDashing->6` for (relatively long) dashes of length 6 printer's points. More complicated patterns are also possible, *e.g.*, `LineDashing->{6,2,2,2}` for a dash-dotted line where the "unit cell" consists of a dash of length 6, gap of length 2, dot of length 2, and another gap of length 2. Further possibilities are discussed in Sec. 8.1.2.

The font style options map directly onto to the standard options for text formatting in Mathematica, discussed in more detail in the Mathematica documentation.[4] The default font used by SciDraw is `"Times"`, but usually any font installed on your system can be used. The typical settings for the remaining two options are `FontWeight->Bold` for boldface or `FontSlant->Italic` for italic. (See also Sec. 4.3 for LaTeX-like control of text attributes.)

The remaining options will be discussed further in future additions to the user's guide. But, for now, see (Sec. 8) of the Reference Manual.

Table 4.2 Options for controlling attached labels (extracted from Tables 8.8 and 8.10).

Option	*Default*	
X`Label`	`None`	The label contents.
X`LabelPosition`	`Automatic`	Additional positioning argument to use when generating the label anchor.
X`ShowText`, *X*`TextColor`, *X*`FontFamily`,...,	*As shown above in Table 4.1, but now applying to one label individually.*	
X`TextBackground`,...		*As shown above in Table 4.1, but now applying to one label individually.*
X`TextOffset`, *X*`TextOrientation`, *X*`TextRectify`, *X*`TextNudge`, *X*`TextBuffer`		*As shown above in Table 4.1, but now applying to one label individually.*

Any SciDraw object can have one or more *attached labels*, depending upon the type of object. The basic principles were illustrated in Tutorial 2(Sec. **??**). Here we provide a summary reference.

[4]See **tutorial/TextAndFontOptions** for an overview of font options.

Each attached label has a name indicating its position relative to the object. Typical names would be `Left` for the left, `Right` for the right, `Top` for the top, `Bottom` for the bottom, `Center` for the center, or `Head` and `Tail` for the ends of an arrow. These attached label names are actually the names for anchor locations recognized by `GetAnchor`. Some anchors defined for an object also serve as likely places for attaching a label, and therefore attached labels have been defined for those locations. Other anchors do not denote likely spots for a label and therefore do not have a corresponding attached label defined. All attached label names (and all anchor names), for all objects, are detailed in the Reference Manual.

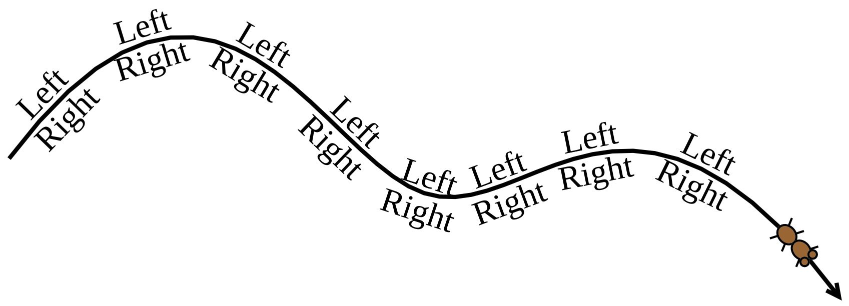
Most of the time, the predefined "attached" labels suffice. But, more generally, SciDraw provides a framework for positioning as many labels as desired, along an object, by first retrieving an anchor with `GetAnchor`, then labeling with `FigLabel` — also as illustrated in Tutorial 2 (Sec. **??**).

The contents of the attached label are given through the corresponding named option, designated *X*`Label` in Table 4.2, for instance, `RightLabel->"This way up!"`. The value `None` means that no label should be drawn.

The positioning can also be adjusted, via the option *X*`LabelPosition`. There is generally a default position for the label. For instance, the `Right`, `Center`, or `Left` label on an arrow shaft would, by default, be midway along the arrow shaft. But, this can be refined with the *X*`LabelPosition` option.[5] For instance, `RightLabelPosition->0.7` would place the label 70% of the way along the shaft, measured from tail to head. On the other hand, for some labels, such as a `Center` label on a `FigCircle`, there may be nothing to adjust. The center is the center is the center. Period.

Each attached label has its own "copy" of the various text options in Table 4.2. Thus, *e.g.*, while `TextColor->Red` would make *all* attached labels red, `RightTextColor->Red` makes only the `Right` label red. Or, similarly, `RightTextNudge->{2,3}`, say, would nudge the `Right` label to the right by 2 printer's points and up by 3 printer's points.

In SciDraw, several object types involve curves: most notably, `FigLine`, `FigArrow`, `Trans`, and `DataPlot`. The "left" and "right" sides of a line or curve refer to left and right sides from the *curve's perspective*, not necessarily the left or right side from *your* perspective, looking down at the page! If you have ever talked about something being "on the left side of the road" or "on the right side of the road", you should be fine with this! You didn't necessarily mean the west side when you said "left" or the east side when you said "right", even though those would seem to be the left or right sides for someone looking at a map. Think from the viewpoint of a car driving along the curve or, more entomologically speaking, from the viewpoint of an insect crawling along the curve:



Incidentally, you can find the source code for this illustration in `Examples-Guide.nb`.

[5]More precisely, this option provides any additional arguments which `GetAnchor` accepts, in order to retrieve the anchor at which we will position the label.

4.3 Formatting text

Mathematica offers advanced capabilities for typesetting text and formulas. These capabilities provide great flexibility in typesetting complex text for figure labels. They are described in the Mathematica documentation and are *not* the topic of the present section. Rather, SciDraw provides some *extra* functions to help typeset labels for scientific figures. These are summarized in the present section. The basic idea was introducted in tutorial Sec. 3.1.5, which you will want to read first.

The functions described in this section may be used independently of the rest of SciDraw. That is, they may be used anywhere where text is being typeset in Mathematica, not necessarily within a SciDraw figure. The result may need to be displayed in `DisplayForm` to appear as expected.

Table 4.3 Formatting via LaTeX-like text attributes.

`textmd[`*text*`]`	Produces ordinary-weight text.
`textbf[`*text*`]`	**Produces boldface text.**
`textup[`*text*`]`	Produces ordinary upright text.
`textsl[`*text*`]`	*Produces slanted text.*
`textit[`*text*`]`	*Produces italic text.*
`textrm[`*text*`]`	Produces roman text (Times).
`textsf[`*text*`]`	Produces sans-serif text (Helvetica).
`texttt[`*text*`]`	`Produces typewriter text (Courier).`
`textcolor[`*color*`,`*text*`]`	Produces text of arbitrary color.
`textsize[`*size*`,`*text*`]`	Produces text of arbitrary size.

While options such as `FontSize` or `FontWeight` allow you to adjust the font attributes of a label *as a whole* in a SciDraw object, you will often need to have finer formatting control over the font attributes of parts of label. For these purposes, SciDraw provides several functions for changing font attributes, summarized in Table 4.3.

LaTeX-like text attributes. SciDraw provides several functions for changing font attributes (weight, slant, family) analogous to the standard LaTeX font commands `\textbf`, `\textit`, *etc.* The following example illustrates their use inside a Mathematica `Row` object, and together with some other Mathematica typesetting constructs (as well as `SuperPrime` from Table 4.4 below).

```
Row[{SuperPrime[textbf["r"], 2], "(", textit["t"], ")=3",
  Superscript[textit["t"], 2], Subscript[OverHat[textbf["e"]], textit["x"]]}]
```

$\mathbf{r}''(t)\text{=}3t^2\hat{\mathbf{e}}_x$

These LaTeX-like functions are all are simply typing shortcuts for much longer Mathematica `Style` directives.

`textcolor`. SciDraw also provides a function `textcolor` for changing font color, analogous to the LaTeX `color` package's `\textcolor` command. The *color* can be any color specification (Sec. 8.1.1), *e.g.*, `textcolor[FireBrick,"Text"]`.

`textsize`. In the same spirit, SciDraw provides a function `textsize` to change font size. Here *size* is in printer's points.

Table 4.4 Mathematical typesetting.

`MultipletLabel[{`*value*$_1$`,`*value*$_2$`,...}]`	Groups a multiplet of values inside parentheses.

`SuperPrime[`*expr*`]`
`SuperPrime[`*expr*`,`*n*`]`
	Attaches one or more superscript primes to *text*.

`SolidusFractionBox[`*a*`,`*b*`]`	Typesets fraction a/b in slashed form.
`SolidusFractionize[`*x*`]`	Extracts the numerator and denominator of a rational number and typesets in slashed form.
`DiagonalFractionBox[`*a*`,`*b*`]`	Typesets fraction a/b in diagonal form.
`DiagonalFractionize[`*x*`]`	Extracts the numerator and denominator of a rational number and typesets in diagonal form.
`TextFractionBox[`*a*`,`*b*`]`	Typesets fraction a/b in reduced-size text form.
`TextFractionize[`*x*`]`	Extracts the numerator and denominator of a rational number and typesets in reduced-size text form.
`UnitsLabel[`*unit*`,...,{`*unit*`,`*power*`},...]`	Formats a string of units, with optional exponents.

SciDraw offers sundry additional functions for general mathematical typesetting, summarized in Table 4.4.

`MultipletLabel`. `MultipletLabel[{`*value*$_1$`,`*value*$_2$`,...}]` typesets a multiplet of values, by default separated by commas and grouped with parentheses. These may be changed via options `EntrySeparator` and `Delimiter` (defaulting to `EntrySeparator->","` and `Delimiter->{"(", ")"}`).

```
MultipletLabel[{"+", "-", "+"}]
MultipletLabel[{"+", "-", "+"}, EntrySeparator → "", Delimiter → {"⟨", "⟩"}]
```

(+,−,+)
⟨+−+⟩

`SuperPrime`. `SuperPrime[`*expr*`]` attaches a superscript prime to the given expression (as in x'), while `SuperPrime[`*expr*`,`*n*`]` attaches *n* primes (as in x''').

Fraction typesetting. A *slashed fraction* (or *solidus fraction*) is typeset on one line, with numerator and denominator separated by a slash (or *solidus*), as in a/b. A *diagonal fraction* is typeset compactly, with the numerator raised and the denominator lowered (so that they line on a diagonal), as in ${}^{a}/_{b}$. These forms are typically neater and more readable in labels than the Mathematica default *vertical fraction* (or *built-up fraction*) $\frac{a}{b}$. The reduced-size *text fraction* is still in vertical form but with reduced font size, as in $\tfrac{a}{b}$, in the spirit of the AMSLaTeX `\tfrac` command.

```
SolidusFractionize[1 / 2]
DiagonalFractionize[1 / 2]
1 / 2
```

1/2
${}^{1}/_{2}$
$\frac{1}{2}$

While the solidus fraction functions can be used with confidence, the diagonal fraction functions produce highly variable results, due to inconsistency in how the different elements (numerator, slash, and denominator) are spaced relative to each other when actually rendered by Mathematica, which vary significantly for different font choices. Spacings which look fine on screen in the Mathematica notebook may look horrible in an exported EPS/PDF file, and *vice versa*. Manual control can be obtained through options `Spacings` (for the spacing between numerator, slash, and denominator) and `Baseline` (for heights of these elements, respectively, above the baseline). The default spacings have been chosen to provide reasonable results for the Times font, when exported to EPS, under Mathematica 10 (these default values are `Spacings->{-0.1,-0.2}` and `Baseline->{0.5,0,0}`). There is an additional option `KernForSuperscript` (with default `KernForSuperscript->-0.15`), which may be used to adjust spacing between the fraction and an adjacent superscript. Tedious and inelegant trial by error may be necessary.

`UnitsLabel`. `UnitsLabel[`*unit*`, ..., {`*unit*`, `*power*`}, ...]` formats a compound unit, built up as a string of unit factors (with optional exponents) separated by thin spaces.

```
UnitsLabel["kg", {"m", 2}, "/", {"s", 2}]
UnitsLabel["kg", {"m", 2}, {"s", -2}]
UnitsLabel[{textit["e"], 2}, {"fm", 4}]
```

$\mathrm{kg}\,\mathrm{m}^2/\mathrm{s}^2$
$\mathrm{kg}\,\mathrm{m}^2\,\mathrm{s}^{-2}$
$e^2\,\mathrm{fm}^4$

Table 4.5 Text formatting for isotopes.

`Isotope[`*element*`]]`
`Isotope[`*A*`,`*element*`]`
`Isotope[`*A*`,`*Z*`,`*N*`,`*element*`]`
`Isotope[`*A*`,`*Z*`,`*N*`,`*super*`,`*element*`]`
	Typesets given element, with optional subscripts and superscripts.

`Isotope[{`*Z*`,`*N*`}]`
`Isotope[`*super*`,{`*Z*`,`*N*`}]`
	Similarly, deducing element symbol and atomic mass from *Z* and *N*.

Table 4.6 Chemical element and isotope data.

`ElementAbbreviation[`*Z*`]`	Returns element abbreviation for given atomic number.
`ElementName[`*Z*`]`	Returns element name for given atomic number.
`StableIsotopes[`*Z*`]`	Returns a list of stable isotopes, by atomic mass *A*, for given atomic number.
`IsotopeIsStable[{`*Z*`,`*N*`}]`	Returns whether or not isotope is stable.

A function providing isotope labels is summarized in Table 4.5, and associated functions providing access to chemical element data are summarized in Table 4.6.

`Isotope`. `Isotope` provides isotope labels with optional subscripts and superscripts (the syntax is inspired by the LaTeX `isotope` package). In the argument lists shown in Table 4.5, *Z* represents the proton number (atomic number), *N* represents the neutron number, *A* represents the mass number, *element* represents the element symbol, and *super* represents a right superscript (used, *e.g.*, to represent an excited state or ionization). Putting them all together gives

$${}^{A}_{Z}\mathrm{El}^{\mathrm{sup}}_{N}$$

The element symbol argument *element* may be given either as a string (which is simply used directly as the element symbol) or as a nonnegative integer indicating the atomic number (in which case the element symbol is automatically substituted).

The following examples illustrate the various syntactic forms:

```
Isotope["C"]
Isotope[12, "C"]
Isotope[12, 6, "C"]
Isotope[12, 6, 6, "C"]
Isotope[12, 6, 6, "*", "C"],
Isotope[12, None, 6, "C"]
Isotope[12, None, None, "*", "C"]
```

C ${}^{12}\mathrm{C}$ ${}^{12}_{6}\mathrm{C}$ ${}^{12}_{6}\mathrm{C}_{6}$ ${}^{12}_{6}\mathrm{C}^{*}_{6}$ ${}^{12}\mathrm{C}_{6}$ ${}^{12}\mathrm{C}^{*}$

```
Isotope[12, 6, 6, "*", 6]
```

${}^{12}_{6}\mathrm{C}^{*}_{6}$

```
Isotope[{3, 3}]
Isotope["*", {3, 3}]
```

${}^{6}\mathrm{Li}$ ${}^{6}\mathrm{Li}^{*}$

Chemical element and isotope data. The functions `ElementAbbreviation`, `ElementName`, `StableIsotopes`, and `IsotopeIsStable` provide chemical element symbols, chemical element names, and information on the stability of isotopes. Again, *Z* represents proton number (atomic number) and *N* represents neutron number.

```
ElementAbbreviation[6]
ElementName[6]
StableIsotopes[6]
IsotopeIsStable[{6, 7}]
```

```
C
Carbon
{12, 13}
True
```

Note that these functions rely purely on offline data included with SciDraw (based on data from the legacy Mathematica `ChemicalElements` package, circa Mathematica 5). Unlike the Mathematica `ElementData` or `IsotopeData` databases, which require internet access, the SciDraw functions can be used without regard to internet access. (Sort of convenient, right?)

See the section "Bonus: Chart of nuclides with contour plot" in `Examples-Guide.nb` for an example of the use of data from these functions.

Table 4.7 Spectroscopic labels.

`LabelJP[`*spin*`,`*parity*`]]`	Produces a spin-parity label.
`LabelJiP[`*spin*`,`*i*`,`*parity*`]]`	Produces a spin-parity label with an index subscript.

Spectroscopic labels. The functions `LabelJP` and `LabelJiP` (Table 4.7) provide level spin-parity labels, for use in level schemes. The parity argument is optional (default "+") and may be `+1`, `-1`, or `None`. By default, fractional spins are formatted as slashed fractions, but this may be controled through the option `Rational`, which may take the values `SolidusFractionize` (default), `DiagonalFractionize`, or `Fractionize`.

```
LabelJiP[0, 2]
LabelJiP[7 / 2, 2, -1]
LabelJiP[7 / 2, 2, -1, Rational → DiagonalFractionize]
LabelJiP[textit["J"], textit["i"], +1]
```

0^+_2 $7/2^-_2$ ${}^7/{}_2{}^-_2$ J^+_i

Further spectroscopic labeling functions are in various stages of development and will hopefully be documented with future releases of SciDraw. As an example, `ShellLabel[`$\{n, l, j\}$`]` provides labels for nuclear shell model orbitals:

```
ShellLabel[{0, 0, 1 / 2}]
MultipletLabel[ShellLabel /@
  {{0, 0, 1 / 2}, {0, 1, 3 / 2}, {0, 1, 1 / 2}, {0, 2, 5 / 2}, {0, 2, 3 / 2}, {1, 0, 1 / 2}}]
```

$0s_{1/2}$
$(0s_{1/2},0p_{3/2},0p_{1/2},0d_{5/2},0d_{3/2},1s_{1/2})$

Note for users of LevelScheme (on `Row` and `Column`). LevelScheme provided a function `TightRowBox[{`*entry*$_1$`,`*entry*$_2$`, ...}]` to combine several text elements side-by-side, which suppressed the excessive spacing obtained when using Mathematica's `RowBox` for this purpose. `TightRowBox` is no longer necessary, since the same effect can now be obtained with Mathematica's `Row` object, introduced in Mathematica 6.

LevelScheme likewise provided a function `StackText[`*alignment*`,`*spacing*`,{`*line*$_1$`,`*line*$_2$`, ...}]` for generating multiline labels using the given horizontal alignment (*e.g.*, `Left` or `Center`) and line spacing. `StackText` is no longer necessary, since the same effect can now be obtained with the Mathematica `Column` object, introduced in Mathematica 6. The syntax is `Column[{`*line*$_1$`,`*line*$_2$`, ...}]`, `Column[{`*line*$_1$`,`*line*$_2$`, ...},`*alignment*`]`, or `Column[{`*line*$_1$`,`*line*$_2$`, ...},`*alignment*`,`*spacing*`]`.

Note that the *spacing* in `Column` must be explicitly specified as `0` to get tight vertical spacing, as one might have been accustomed to from using `StackText` with a *spacing* of `0`. The default vertical spacing with `Column` is somewhat looser. Moreover, the meaning of the *spacing* parameter differs slightly between `StackText` and `Column`. It was interpreted as being in x-heights by `StackText` (see **ref/GridBox**) but is interpreted as being in units of the current font height by `Column`, thus yielding somewhat larger spacings.

4.4 Level schemes

This section provides a practical overview of the tools at your disposal for drawing level schemes, or level energy diagrams. The source code for all the examples in this section, as well some as more sophisticated examples of level schemes, may be found in `Examples-Guide.nb`. Please see also Sec. 15 of the reference manual for a complete description of the syntax and options.

4.4.1 Note for LevelScheme users

If you have previously drawn level schemes with LevelScheme, you will find that the object names and syntax carry over more or less directly to SciDraw. Therefore, a level scheme you have drawn with LevelScheme should be useable with SciDraw after straightforward modifications. The main differences are:

(1) First off, you will need to put your diagram contents in a `FigurePanel`. You will need to set `Frame->False` for the `FigurePanel`, if you do not want a frame, and convert any `ImageSize` option for `Figure` (in printer's points) to a `CanvasSize` option (in inches, so just divide by 72!).

(2) Level names, which in LevelScheme were given as the first argument to `Lev[`*name*`,...]` now follow the standard object naming scheme, discussed in Sec. 6 of the reference manual. The level name should therefore be given as `Lev⟦`*name*`⟧[...]`.

(3) Option names for labels, colors, *etc.*, must be changed to the new standardized (and less cryptically abbreviated) option names in SciDraw (Sec. 8). For instance, LevelScheme's `LabL` option becomes SciDraw's `LeftLabel` option. You will need to take care as to what is meant by the "left" and "right" side of a transition arrow. This is generally the opposite sense from what it was in LevelScheme, for the reasons discussed below — the basic idea was presented in Sec. **??**.

(4) If you included labels, shapes, or other annotation in your figure, the LevelScheme objects must be replaced by the new SciDraw objects. For instance, LevelScheme's `ManualLabel` and `ScaledLabel` are subsumed by SciDraw's `FigLabel`, or `SchemeCircle` becomes `FigCircle`, with some possible differences in syntax.

4.4.2 Levels, extensions, and connectors

Levels are drawn with the `Lev` object. There are also auxiliary objects `ExtensionLine` and `Connector` for drawing extension lines and connectors, and a special type of label `BandLabel` which can be convenient for labeling rotational bands. These objects are shown here in Table 4.9.[6]

Table 4.8 Levels, extension lines, and connectors (extracted from Table 15.1).

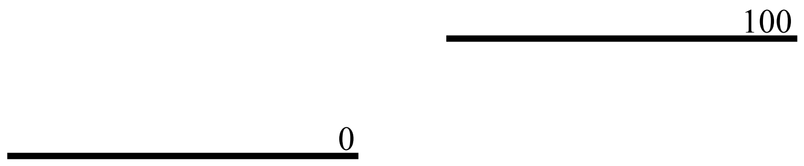
`Lev[`x_1`,`x_2`,`E`]`	Generates a level.
`ExtensionLine[`*level*`,`*side*`,`*dx*`]`	Generates an extension line to a level.
`Connector[`$level_1$`,`$level_2$`]`	Generates a connector line between levels.
`BandLabel[`*level*`,`*text*`]`	Generates a "band" label beneath the given level.

Each specification of a level with `Lev` includes left and right endpoint coordinates (*i.e.*, giving the horizontal span) and an energy coordinate (*i.e.*, giving the vertical position). The level can also be given an object name. This is usually just a string, but it can also be a more general Mathematica expression — see Sec. 6 of the reference manual for guidelines. The name does not affect the appearance of the level itself.

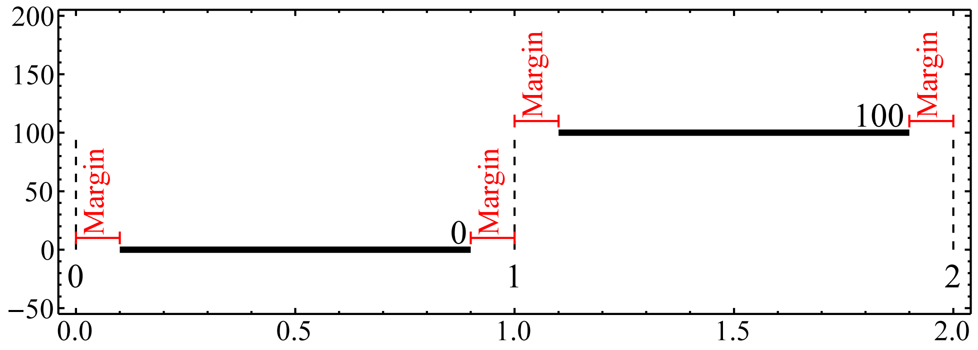
[6]See also sections Secs. 15.1–15.4 of the reference manual.

Rather, it is used later to refer back to the level, when we want to draw extension lines, transitions, *etc.*, which connect to the level.[7]

```
Figure[
 FigurePanel[
  {
   SetOptions[Lev, LineThickness → 3];
   Lev⟦"lev0"⟧[0, 1, 0, RightLabel → 0];
   Lev⟦"lev100"⟧[1, 2, 100, RightLabel → 100];
  },
  PlotRange → {{0, 2}, {-50, 200}}, ExtendRange → Automatic, Frame → False
 ],
 CanvasSize → {6, 2}
]
```



The actual left and right end points of the level are indented from the nominal left and right end coordinates x_1 and x_2, by an amount controlled by the option `Margin`. By default, `Margin->0.1`. This allows end points to be specified in round numbers, *e.g.*, levels can be specified as extending from `1` to `2` and from `2` to `3`, while the margin ensures that the ends of the levels do not actually bump into each other. It may be easier to see how this works if we make our coordinate system more readily apparent, for the moment, by turning on the frame and tick marks.



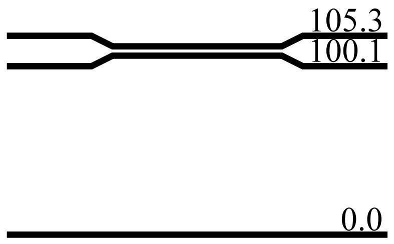
Levels can have left, center, and right labels. Specifying the special option value X`Label->Automatic` causes the level energy to be used as the text of that label. Thus, energy labels can be created on all levels simply by invoking `SetOptions[Lev,`X`Label->Automatic]` and can later be removed as easily. When Mathematica displays real numbers, it removes all trailing zeros after the decimal point, regardless of how the number was originally entered. Thus, for instance, a level energy entered as `0.00` would be truncated to "`0.`" in the energy label, which is undesirable. To circumvent this, we give the energy argument to `Lev` as a *string*, surrounded by quotation marks. `Lev` will extract the *numerical* value for use as the vertical coordinate of the level but will still use the *string* verbatim as the text for energy labels.

[7]In fact, I find it a good idea to habitually give levels names right when I first define them, so I do not have to go back and add names later if and when I decide to draw annotations.

In level schemes with closely-spaced levels, it is sometimes necessary to raise or lower the end segments of levels to make room for text labels. This gives levels which appear to have "gull wings". Gull wings can be created by specifying a nonzero value for the option `WingHeight`, postive for elevated wings and negative for lowered wings.[8] The dimensions of the gull wings can be customized using the options `WingSlopeWidth`, `WingTipWidth`, and `MakeWing` (see Table 15.2 of the reference manual for details).

Here is an illustration of automatic energy labels and gull wings:

```
SetOptions[Lev, RightLabel → Automatic, RightTextOffset → BottomRight];
Lev⟦"lev0"⟧[0, 1, "0.0"];
Lev⟦"lev100"⟧[0, 1, "100.1", WingHeight → -5];
Lev⟦"lev105"⟧[0, 1, "105.3", WingHeight → +5];
```

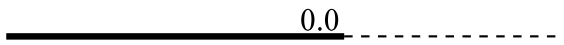


Alternatively, a fixed number of decimal digits may be obtained by setting the option `DecimalDigits`.[9]

```
SetOptions[Lev, RightLabel → Automatic,
  DecimalDigits → 1, RightTextOffset → BottomRight];
Lev⟦"lev0"⟧[0, 1, 0];
Lev⟦"lev100"⟧[0, 1, 100.1, WingHeight → -5];
Lev⟦"lev105"⟧[0, 1, 105.3, WingHeight → +5];
```

Extension lines are attached to an existing level using the `ExtensionLine` object. They extend the level by a specified horizontal length to the left or right.

```
SetOptions[ExtensionLine, LineDashing → 4];
ExtensionLine["lev0", Right, 0.5];
```

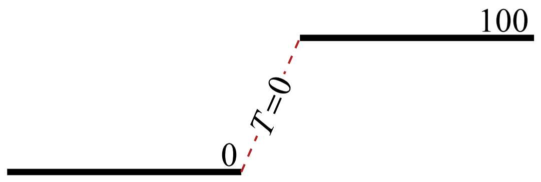


Connector lines between levels are drawn with the `Connector` object. Just like other objects, `Connector` objects can have attached labels.

```
SetOptions[Connector, LineDashing → 4,
  LineColor → Firebrick, TextBackground → Automatic];
Connector["lev0", "lev100", CenterLabel → Row[{textit["T"], "=0"}]];
```

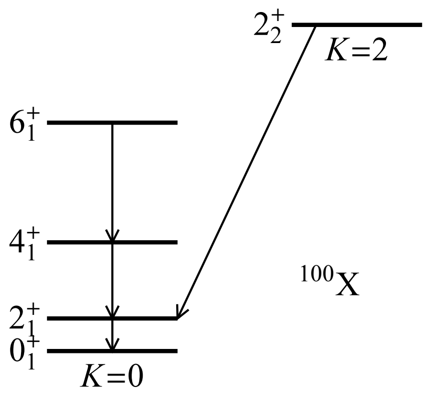
[8]Incidentally, we are borrowing the "gull wing" terminology from aeronautics. Technically, if `WingHeight` is negative, the level has "*inverted* gull wings", the Corsair providing the canonical example...

[9]Actually, it is even possible to specify a custom labeling function, as described in Sec. 15.1 of the reference manual.



Finally, we illustrate the use of `BandLabel[`*level*`,`*text*`]`. This example also provides an illustration of using `Do` loops to automate the drawing a level scheme. Notice the shorthand function `LevelLabel[{`*J*`,`*i*`,`*P*`}]` for generating a spin-parity label J_i^P. Here we just show the code for one band — see `Examples-Guide.nb` for the full code.

```
(* ground state K=0 band, with energies from rotational formula *)
Do[
  Lev[[{"lev", J, 1}]][0, 1, J * (J + 1) / 6 * 100, LeftLabel → LevelLabel[{J, 1, +1}]],
  {J, 0, 6, 2}
 ];
BandLabel[{"lev", 0, 1}, Row[{textit["K"], "=0"}]];
```



4.4.3 Transition arrows

Transition arrows within a level scheme are drawn using the `Trans` object, which is modeled on `FigArrow` but is specially tailored for drawing *transition* arrows. All the basic principles of using `FigArrow` carry over to `Trans`.[10] The difference lies in how the endpoints are specified, as shown here in Table 4.9.

Table 4.9 Transition arrows (extracted from Table 15.1).

`Trans[`*level*$_1$`,`*level*$_2$`]`	Generates a transition arrow between levels.
`Trans[`*level*$_1$`,`*pos*$_1$`,`*level*$_2$`,`*pos*$_2$`]`	

Let us also summarize the principal options for controlling the appearance of a `Trans` arrow (the style, width, arrowhead properties, and geometry), as shown here in Table 4.10. Except for the last two, these options are taken from `FigArrow`.

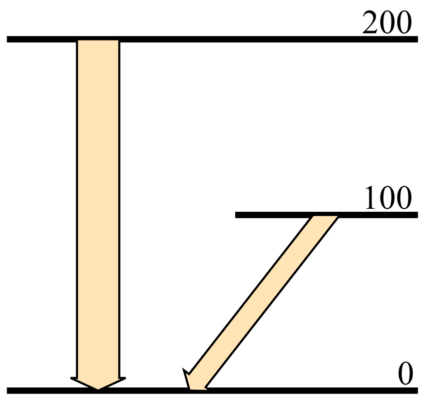
[10]Therefore, you will want to see the reference manual sections for both `FigArrow` (Sec. 12) and `Trans` (Sec. 15.5) for further information — and these will refer you back to `FigLine` (Sec. 11.1) for some of the basic properties.

Table 4.10 The principal options for arrow geometry (extracted from Tables **??** and 15.9).

Option	*Default*	
`ArrowType`	`"Line"`	Arrow type (outline and fill geometry).
`Width`	`5`	Arrow shaft width, in printer's points.
`ShowHead`	`True`	Whether or not to draw arrowhead at head.
`HeadLength`	`6`	Length of arrowhead at head, in printer's points.
`HeadLip`	`3`	Extension of arrowhead beyond shaft, on each side, at head, in printer's points.
`ShowTail, ...`		*Similarly for the tail.*
`TailFlush`	`True`	Whether or not shaft tail should be flush to the level (default is `False` for `FigArrow`).
`EndPositions`	`0.5`	Horizontal positions of the arrow endpoints, if not given as arguments (`Trans` only).
`IntermediatePoints`	`None`	Points through which the arrow should pass along the way (`Trans` only).

The syntax `Trans[`$level_1$`,`pos_1`,`$level_2$`,`pos_2`]` draws a transition arrow starting a horizontal distance pos_1 from the left end of $level_1$ and ending a horizontal distance pos_2 from the left end of $level_2$. The distance is calculated from the *nominal* left end of the level, ignoring the margins, rather than from the visible end point. This simplifies the mental arithmetic required for positioning. For instance, an arrow starting from the middle of a level which nominally extends from 0 to 1 can be obtained simply by specifying a position `0.5`.

```
Trans["lev200", 0.5, "lev0", 0.5, Width → 20];
Trans["lev100", 0.5, "lev0", 0.9, Width → 10];
```



If either pos_1 or pos_2 is specified as `Automatic`, the arrow is made vertical, and its horizontal position determined by whichever coordinate is not specified as `Automatic`. This is especially useful when it is desired that the arrow should remain vertical even though one or both of the levels might need to be moved horizontally as the level scheme is edited.[11]

```
Trans["lev200", 0.5, "lev0", Automatic, Width → 20]; (* note use of Automatic *)
Trans["lev100", 0.5, "lev0", 0.9, Width → 10];
```

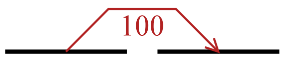
[11]Without the `Automatic` value, a new value for pos_1 or pos_2 would have to be entered manually each time the left end of either level moved.

The abbreviated form `Trans[`*level*$_1$`,`*level*$_2$`]` takes its starting and ending positions from the option `EndPositions->{`*pos*$_1$`,`*pos*$_2$`}`, or simply `EndPositions->`*pos* if both positions are the same. This is useful if many transition arrows are to be drawn with the same horizontal start and end positions, as is often the case for the transitions within a band or between two bands. Then `EndPositions` can simply be specified once using `SetOptions`, or via a style, and it will apply to all the transitions.

```
Trans["lev200", "lev0", Width → 20]; (* note default EndPositions→0.5 *)
Trans["lev100", "lev0", EndPositions → {0.5, 0.9}, Width → 10];
```

Often it is necessary to introduce one or more "kinks" into a transition arrow, `i.e.`, to draw the arrow as a multi-segment curve. The first and last points of the arrow are specified as usual by giving the level names, while the intermediate points are specified as a list, via the option `IntermediatePoints->{`p_2`,...,`p_{n-1}`}`. Most simply, each point may be specified as an ordinary coordinate pair $\{x, y\}$. However, more commonly, you will find it convenient to specify points *relative* to the head or tail of the arrow, either in ordinary coordinates or canvas coordinates (printer's points) — namely, `DisplaceHead[{`x`,`y`}]`, `DisplaceHead[Canvas[{`x_c`,`y_c`}]]`, `DisplaceTail[{`x`,`y`}]`, or `DisplaceTail[Canvas[{`x_c`,`y_c`}]]`.[12] For example:

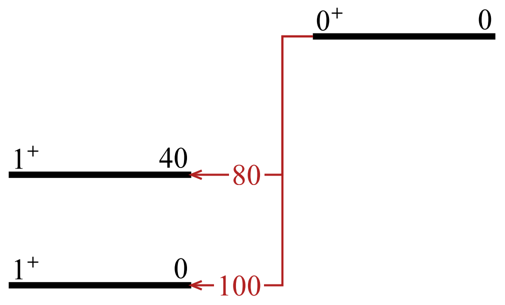
```
Trans[
  "leva", "levb",
  IntermediatePoints → {FromTail[Canvas[{20, 20}]], FromHead[Canvas[{-20, 20}]]},
  RightLabel → 100
 ];
```



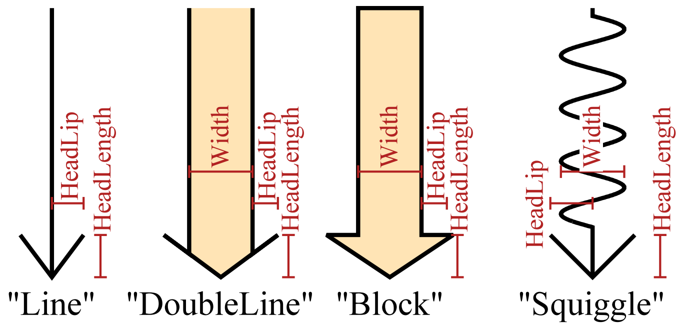
As another example, intermediate points are needed in beta decay diagrams.

```
SetOptions[Lev, Margin → 0.2];
SetOptions[Trans,
  EndPositions → {0.2, 0.8},
  IntermediatePoints → {FromTail[{-0.10, 0}], FromHead[{+0.30, 0}]},
  HeadLength → 5, HeadLip → 2,
  Color → Firebrick, TextBackground → Automatic, TextOrientation → Horizontal,
  CenterTextOffset → Right, (* for right alignment of labels *)
  CenterLabelPosition → {-1, 0.2}
  (* for label at fraction 0.2 into last segment -- see below *)
 ];

(* ... levels defined here... *)

Trans[{"parent", 0}, {"daughter", 0}, CenterLabel → 100];
Trans[{"parent", 0}, {"daughter", 40}, CenterLabel → 80];
```

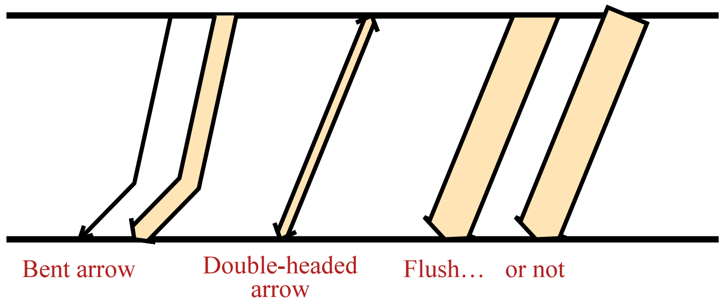
[12]These intermediate points fall into the category of "points along a curve" — see the discussion in Sec. 7.2.1 of the reference manual.



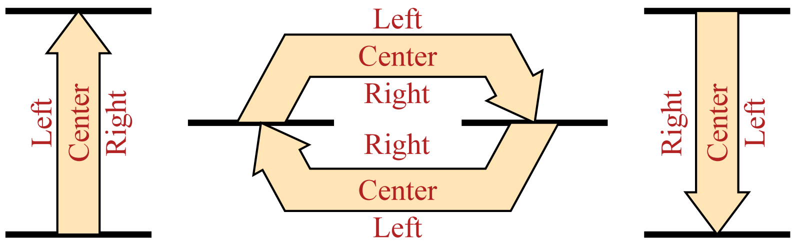
Arrows can be drawn in several different styles, selected by the option `ArrowType`. An arrow of type `"Line"` has a single shaft, and an arrowhead constructed from line segments, the lengths and angles of which are customizable. An arrow of type `"DoubleLine"` is similar but has two lines in its shaft. The area between the lines can be shaded as well. Note that the default color for the fill is the same as for the line, which would leave the lines indistinguishable from the fill, defeating the point of having multiple lines. Thus, in practice, `"DoubleLine"` is almost always used with either a separate `FillColor` option or with `ShowFill->False`. An arrow of type `"Block"` is drawn as a polygon with both an outline and fill. An arrow of type `"Squiggle"` has a sinusoidal squiggle for its shaft. Note that the quotation marks in the names of the arrow types are important — these option values are strings, not symbols. These styles — and the arrow geometry parameters for each — are illustrated in the following.



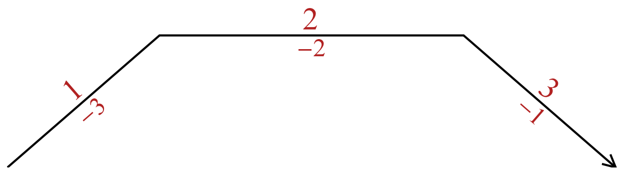
Some of the possible variations on transition arrows are shown below. As already noted, an arrow can have "kinks", or multiple segments, specified by `IntermediatePoints` — we illustrate what that looks like for different arrow types here. An arrow can be "double headed" or even have an arrowhead only on its tail, as controlled by the `ShowHead` and `ShowTail` options. The tail of a double-line or block arrow is normally drawn flush against the starting level, with the default `TailFlush->True`, but this may be overridden by setting `TailFlush->False`.



Arrows can have labels attached to their left, center, right, head, or tail. The principle is the same as for attaching labels to any curve. Recall, from Sec. **??**, that the "left" and "right" sides are defined relative to the *curve*, not from your viewpoint looking at the page. If an arrow happens to be pointing upward (as for *excitation* transitions in level schemes), then these labels will, happily, also be on the left and right sides of the page, respectively. But, if an arrow is pointing downward (as for *decay* transitions), do note that the `Left` label will be to the right side of the page, and *vice versa*. If you are drawing a decay scheme, where all the arrows point more or less downward, this may take a little getting used to.[13] It is clearest if we just look at a couple of examples.



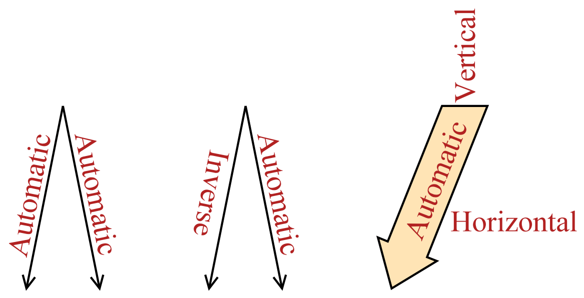
For the `Left`, `Center`, and `Right` labels, the position of the label along the arrow shaft is controlled with the option *X*`LabelPosition`. If a simple numerical value is given for the option, this specifies the position as a fraction of the distance from the tail to the head. The labels are by default at 0.5, the midpoint of the arrow. More sophisticated positioning specifications, *e.g.*, in terms of distances in printer's points from the tail or head, are available as well (see the summary of anchors for `FigLine` in Table 11.3). Here, we just note that, for arrows with more than one segment, the labels may appear on any of the segments, as specified by a position given as {*segment*,*pos*}, for instance, {`2,0.5`} for the middle of the second segment. Segments are numbered 1, 2, ... starting from the tail of the arrow (or, alternatively, −1, −2, ... backwards from the head of the arrow), as illustrated below.



If the *X*`TextOrientation` option for a label is specified as `Automatic` (the default), the label will be aligned flush along the arrow shaft, giving a very neat appearance. Normally, the label will be flipped so that it is right side up. If a label "upside down" relative to this angling is prefered, as it occasionally might be for near-vertical arrows, the option can be overriden with the value `Inverse`. Ordinary horizontal or vertical labels can be specified, as usual, with the `Horizontal` and `Vertical` option values. You will likely also have to adjust the *X*`TextOffset` option at the same time to get a satisfactory result, as for the rightmost arrow shown below.

[13]Users of LevelScheme will notice that this is a change in convention. The notation used in LevelScheme, for historical reasons, was narrowly focused on decay schemes. SciDraw, on the other hand, needs to maintain a useable notation for labeling arrows and curves in general. Sorry, gamma ray spectroscopists. You should be able to adapt to this quickly. Or you can always switch to Coulomb excitation.

```
(* rightmost transition arrow *)
Trans[
  "lev100", "lev0", EndPositions → {0.7, 0.3},
  CenterLabel -> "Automatic", CenterLabelPosition → 0.4,
  TailLabel → "Vertical", TailTextOrientation → Vertical, TailTextOffset → Left,
  (* that's the left end of the text *before* rotation! *)
  LeftLabel → "Horizontal",
  LeftTextOrientation → Horizontal, LeftTextOffset → TopLeft
 ];
```

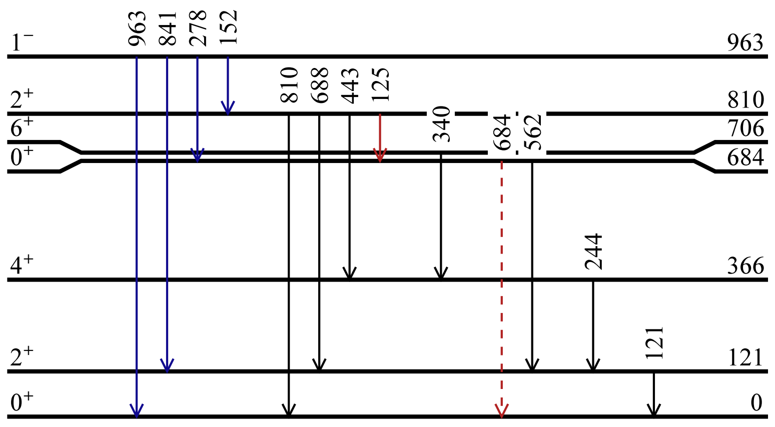


Some special definitions are provided to facilitate the drawing of decay schemes in the classic style for such schemes. Such schemes consist of a stacked series of levels, connected by an array of vertical arrows which are equally spaced horizontally and grouped by starting level.[14] The command `AutoLevelInit[`x'`,`dx`,`Dx`]` initializes autospacing, specifying horizontal coordinate x' for the first transition, spacing dx between transitions from the same level, and spacing Dx between groups of transitions from different levels. Then `AutoLevel[`$level_1$`]` selects a new starting level for transitions, and `AutoTrans[`$level_2$`]` draws a transition to the designated ending level.

The following example illustrates the use of these decay scheme generation tools. Note that negative spacings Dx are specified in `AutoLevelInit` to draw the transitions successively from right to left. The transition labels are specified with the usual label options for `Trans`, *e.g.*, `TailLabel->`*label*. It is usually desirable to set the option `TextBackground->Automatic` for `Trans`, to create a white-out box behind each label, blocking out any higher-lying levels behind the label. You will usually want to expand this box to extend a little beyond the text itself, with the `TextMargin` option. To prevent this box from cutting into the level line of the level from which the transition originates, the label can be nudged upwards, with the `TextNudge` option.

[14]See Sec. 15.6 of the reference manual.

```
(* autospaced transitions *)
SetOptions[Trans, TextBackground → Automatic, TextMargin → 1, TextNudge → 2];
AutoLevelInit[0.85, -0.04, -0.08];
AutoLevel["lev121"];
AutoTrans["lev0", TailLabel → "121"];
AutoLevel["lev366"];
AutoTrans["lev121", TailLabel → "244"];
AutoLevel["lev684"];
AutoTrans["lev121", TailLabel → "562"];
AutoTrans["lev0", TailLabel → "684", LineColor → Firebrick, LineDashing → 4];
AutoLevel["lev706"];
AutoTrans["lev366", TailLabel → "340"];
AutoLevel["lev810"];
AutoTrans["lev684", TailLabel → "125", LineColor → Firebrick];
AutoTrans["lev366", TailLabel → "443"];
AutoTrans["lev121", TailLabel → "688"];
AutoTrans["lev0", TailLabel → "810"];
AutoLevel["lev963"];
AutoTrans["lev810", TailLabel → "152", LineColor → Ultramarine];
AutoTrans["lev684", TailLabel → "278", LineColor → Ultramarine];
AutoTrans["lev121", TailLabel → "841", LineColor → Ultramarine];
AutoTrans["lev0", TailLabel → "963", LineColor → Ultramarine];
```



4.5 Generating EPS/PDF output for publication [STUB]

Coming soon… For now, please refer to Section VIII on page 47 of the old *LevelScheme user’s guide* (Version 3.53) — please see `LevelSchemeGuide.pdf`, which has been included with this SciDraw distribution.

Part II

Reference manual

5 Setting up the canvas with `Figure`

Table 5.1 The basic figure display command.

`Figure[`*body*`]`	Constructs and displays a figure, from the commands in *body*.

Description. The `Figure` command sets up the canvas on which a figure is drawn, as introduced in Sec. 3.1.2 of the user's guide.

Arguments. `Figure` takes just a single argument — the figure *body*, *i.e.*, the Mathematica commands which generate the figure contents — as indicated in Table 5.1. For a more complete technical discussion, see the *Note on panel body syntax* in Sec. 10.1.

Table 5.2 Options for `Figure`.

Option	*Default*	
`CanvasSize`	`{6,4.25}`	Dimensions of the *main* canvas area.
`CanvasMargin`	`1`	Additional *margin* around the main canvas area.
`CanvasUnits`	`Inch`	Length unit for the `CanvasSize` and `CanvasMargin` options.
`Style`	`None`	Style or list of styles applied to `Figure` and to all objects within it.
`Background`	`None`	Background color for the entire canvas.
`CanvasFrame`	`False`	Whether or not to draw canvas border lines to aid the drawing process.

Options. The options for `Figure` are summarized in Table 5.3.

`CanvasSize`. The `CanvasSize` option specifies the $\{width, height\}$ dimensions of the *main* canvas area, as defined in Sec. 3.1.2, in the units given by `CanvasUnits`.

`CanvasMargin`. The `CanvasMargin` option specifies the size of the *margin* around the main canvas area, as defined in Sec. **??**, in the units given by `CanvasUnits`. The margin may simply be given as a single number d (same on all sides), $\{d_x, d_y\}$ (different for horizontal and vertical directions), or $\{\{d_L, d_R\}, \{d_B, d_T\}\}$ (different for left, right, bottom, and top).

`CanvasUnits`. The `CanvasUnits` option defines the unit for the `CanvasSize` and `CanvasMargin` options. This may be any length unit defined in the Mathematica `Units` package (see **Units/tutorial/Units**). The units `Inch` (default), `Centimeter`, or `Point` are likely to be most useful for figure preparation.

`Style`. The `Style` option specifies a style (or list of styles) which should be applied to the figure. See Sec. 9 for further discussion of styles.

`Background`. The `Background` option sets a background color for the entire canvas. However, note that this is not typically the effect you would be after. You will more likely instead prefer just to set a background for the panel or panels within the figure, through the `Background` option to `FigurePanel` (Sec. 10.1).

`CanvasFrame`. With `CanvasFrame->True`, a dashed line is drawn around the main canvas area (excluding margins) and a solid line around the full canvas (including margins). This can help you to judge the available drawing area as you arrange the elements of your figure. This is simply a visual aid during the design process and is not intended for use in the final production version of the figure.

Table 5.3 Additional options for `Figure`, for automatic exporting.

Option	*Default*	
`Export`	`False`	Whether or not to generate exported file.
`ExportDirectory`	`Automatic`	Directory in which to write exported file, or `Automatic` for current working directory.
`ExportFileName`	`"figure-*.eps"`	File name or file name pattern for exported file.
`ExportFormat`	`"EPS"`	Export format.
`ExportOptions`	`{}`	Additional options for `Export`.
`ExportStamp`	*See text.*	Expression giving unique file stamp.

Automatic export options. `Figure` can automatically export each figure as it draws it. This is convenient especially if one is automatically generating a large number of figures, e.g., for an animation. The option value `Export->True` enables automatic exporting. The option `ExportFileName` specifies a string which serves as a template for the filename. A `*` in the string indicates where a unique identifier should be inserted, *e.g.*, `"figure-*.eps"`. The option `ExportDirectory` may be used to indicates the directory into which exporting should occur (*e.g.*, `ExportDirectory->"c:/work/manuscript"`), if this is not already specified in `ExportFileName`. The `ExportFormat` and `ExportOptions` options specify the export format argument and any additional options for the `Export` command (see **ref/Export**). By default the unique identifier provided by `Figure` is a timestamp, but the `ExportStamp` argument provides a hook for replacing this default timestamp. This expression must be wrapped in `Hold`, for technical reasons. The default timestamp is generated with `Hold[DateString["YearShort", "Month", "Day", "-", "Hour24", "Minute", "Second", "-", "Millisecond"]]` (see **ref/DateString**).

6 Objects

Syntax. The name of an object is given as an optional first argument, which may be delimited either by doubled brackets, as `[[`*name*`]]`, or by the “double bracket” special characters, as ⟦*name*⟧.[1] These latter double bracket characters are entered as Esc - [- [- Esc and Esc -] -] - Esc, respectively. If no optional argument is given, the object has no name (or, rather, a dummy name is provided internally by SciDraw for internal use). The generic syntax for SciDraw objects is thus

object[*arguments*]

or

object⟦*name*⟧[*arguments*]

Names may be reused within a single figure. However, stored information for the latter instance of a given name overwrites any information stored for the earlier instance.

Naming convention. It is strongly suggested that object names be chosen consistently as strings rather than as Mathematica symbols or other types of Mathematica expressions, *e.g.*, `"Node"` rather than `Node`.[2] However, often you may find it useful to include data in the object name, especially if you are using Mathematica programming constructs such as `Table` to automate your work. In this case, the recommended form for the name is as a *list*, starting with a descriptive string as the first entry, followed by the data, *e.g.*, `{"Node",1}`, `{"Node",2}`, ...

[1] See **ref/character/LeftDoubleBracket**. The double bracket syntax should be familiar from Mathematica's syntax for indexing elements of lists (see **ref/Part**). In fact, the present syntax for object names was more or less dictated by the limited notational possibilities available within Mathematica's syntax — here we have actually coopted Mathematica's double bracket notation for `Part`, for an entirely different purpose.

[2] If symbols are used, and values are accidentally assigned to those symbols elsewhere in your work, this confuses the naming scheme. Alternatively, if values are not assigned, all the names will be highlighted in blue by the front end, as undefined names, defeating the purpose of this highlighting as a warning sign to you against typographical or other naming errors.

7 Coordinates, anchors, and regions

7.1 Points and anchors

Many (in fact, almost all) of the commands to draw figure objects in SciDraw require that you describe the location and shape of the object in terms of points — for instance, the points along a curve (for `FigLine`), the starting and ending points of an arrow (for `FigArrow`), or the position of a label (for `FigLabel`). SciDraw expands upon the concept of a *point* by introducing the idea of an *anchor*.[1] A point consists simply of (x, y) position information. An anchor contains not only an (x, y) position but also some further information needed to control the positioning and orientation of a text label — an offset and an orientation.

Interchangeability of points and anchors. Although for some purposes only the coordinates of a point are needed (*e.g.*, for points along a curve given to `FigLine`), and in others the full anchor information is needed (*e.g.*, for positioning and orienting the text of a label), an anchor may always be given where a point is expected as an argument, and *vice versa*. That is, if a point is expected for the argument, an anchor will be accepted instead, and the offset and orientation information will just be thrown away as irrelevant. Conversely, if an anchor is expected for the argument, a point will be accepted instead, and SciDraw will just understand you to mean the default offset `{0,0}` (text centered on the point) and orientation `0` (horizontal text) for the remaining information.

7.1.1 Points

Table 7.1 Ways of specifying coordinates for points.

$\{x, y\}$	A point specified by its *xy* coordinates, in the current panel's coordinate system.
`Scaled[`$\{x_s, y_s\}$`]`	A point specified by its scaled coordinates, *i.e.*, its position as fraction of the way across the panel.
`Canvas[`$\{x_c, y_c\}$`]`	A point specified by its coordinates in printer's points on the canvas.
$\{x,$ `Scaled[`y_s`]`$\}, \ldots$	Hybrid forms are also accepted (see text).

The possible ways of specifying the coordinates[2] for a point as an argument to a figure object are summarized in Table. 7.1.

Ordinary *xy* coordinates. Typically, you will just give the point as an (x, y) pair, written as $\{x, y\}$. This (x, y) pair will be interpreted in the context of the current coordinate system.

Scaled coordinates. Mathematica defines so-called *scaled* coordinates (see **ref/Scaled**) which run from `{0,0}` at the lower left corner of a plot to `{1,1}` at the upper right corner. A point in scaled coordinates is given as `Scaled[`$\{x_s, y_s\}$`]`. SciDraw interprets scaled coordinates with respect to the *current panel*, rather than the figure as a whole. Scaled coordinates are useful for specifying a position within the panel's frame, regardless of what `PlotRange` has been defined for the panel. For instance, if you want to put a label at the top center of a panel, it would be nuisance (at best) for you to have to calculate the (x, y) coordinates of this point and insert them as the coordinate argument to `FigLabel` — and then to have to go back and edit

[1]The concept of anchors in SciDraw is introduced in Sec. **??** of the user's guide.

[2]The concepts of scaled and canvas coordinates in SciDraw are introduced in Sec. 3.1.4 of the user's guide.

these (x,y) coordinates if you ever change the panel's `PlotRange`. Instead, you can easily and reliably refer to the top center as `Scaled[{0.5,1}]`.

Canvas coordinates. Internally, SciDraw makes extensive use of canvas coordinates to determine positions. On the rare occasion you might want to refer a position directly on the figure canvas, this can be done by specifying a point as `Canvas[{`x_c`,`y_c`}]`. However, from the user's perspective, the primary application is to specify *displacements* measured in printer's points (for instance, see the discussion of `DisplacePoint` in Sec. 7.1.5) or the *dimensions* of objects (for instance, see the discussion of `Radius` in Sec. 11.3), where the notation `Canvas[{`dx`,`dy`}]` indicates that these measurements are in printer's points.

Hybrid coordinates. It is also possible to specify the coordinates `{`x`,`y`}` as a hybrid of these coordinate systems. For instance, a coordinate specification of the form `{`x`,Scaled[`y_s`]}` is useful if you wish to position a label or marker, say, at a given x value, but at the top or bottom of the figure, or a given fraction of the way up the figure. For example, the point at horizontal coordinate value 0.1 and at the top of the figure would be specified as `{0.1,Scaled[1]}`.

Individual coordinates (x or y). Sometime just a single coordinate — the horizontal position x or vertical position y — is required by a figure object. For instance, a vertical rule is specified by `FigRule[Vertical,`x`,{`y_1`,`y_2`}]`. In such cases, the coordinate may be specified according to a similar scheme: the x coordinate itself, `Scaled[`x_s`]` for a scaled position across the panel, or `Canvas[`x_c`]` for a canvas position. Moreover, instead of just the single coordinate, a point can be given, and the corresponding x (or y) value will be extracted — this is particularly powerful in that it allows an *anchor* (Sec. 7.1.2) to be given in place of an x or y position. Thus, various possibilities would be:

```
FigRule[Vertical,0.,{...}];  (* rule at x=0 *)
FigRule[Vertical,Scaled[0.5],{...}];  (* rule at middle of panel *)
FigRectangle⟦"R1"⟧[...];  (* here is rectangle "R1" *)
FigRule[Vertical,GetAnchor["R1",Left],{...}];  (* rule aligned
    with left edge of rectangle "R1" *)
```

7.1.2 Anchors

Table 7.2 Explicitly constructing an anchor.

`Anchor[`p`,{`x_o`,`y_o`},`θ`]`	An anchor constructed from a point (or anchor) p, an offset (optional), and an angle (optional).

`Anchor`. Typically, you will *not* be generating an anchor directly with `Anchor` but will rather use automatically generated anchors obtained from objects via `GetAnchor` (Sec. 7.1.4).[3] Nonetheless, the possible ways of generating an anchor to use as an argument to a figure object are summarized in Table. 7.2. An anchor can be generated by specifying all three components of the anchor — the point (x,y), offset (x_o,y_o), and orientation θ — explicitly, as `Anchor[`*point*`,{`x_o`,`y_o`},`θ`]`. Here the *point* can be given in any of the forms described in Table 7.1. The offset and orientation arguments are both optional, and will be taken as `{0,0}` and `0`, respectively, if omitted.

[3]`Anchor` is heavily used for constructing anchors *internally* in SciDraw, when it generates the anchors associated with objects. But these uses are more technical, and you are only likely to encounter them if you become involved with programming to add on new types of drawing objects for SciDraw.

For completeness, we note that you can also generate a new anchor from an existing anchor *a*, as `Anchor[`a`,{`x_o`,`y_o`},`θ`]`, This can be useful if you wish to override the values of the offset or orientation parameters given in *a*. If either the offset or orientation argument is omitted, the value from *a* will be used.

Comment: From the user's perspective, one possible use for explicitly constructing an anchor is to name a point for future use in positioning other objects, for instance,

```
a=Anchor[{0.352,0.246}];
```

Why might you want to give points a name, instead of refering to them directly by their coordinates? Or perhaps saving the coordinate specification in a variable, *e.g.*, `p={0.352,0.246}`, and then using that variable *p*? A saved anchor stays fixed on the canvas even if the meanings of *coordinates* change — say, when you enter a new panel or apply `WithOrigin` (Sec. 10.5). So, for instance, if you wish to draw connectors or arrows between different panels of the same figure, you can use saved anchors as the endpoints.

7.1.3 Retrieving information about anchors

Table 7.3 Functions for extracting information from existing anchors.

`AnchorCoordinates[`*p*`]`	Returns the coordinates of the given anchor *p*, reexpressed in the current coordinate system.
`AnchorAngle[`*p*`]`	Returns the orientation angle of a given anchor *p*.
`AnchorOffset[`*p*`]`	Returns the offset information associated with a given anchor *p*.
`CanvasRayAngle[{`p_1`,`p_2`}]`	Returns the orientation angle of the ray between points (or anchors) p_1 and p_2.

Extracting quantitative information from an anchor. The functions `AnchorCoordinates`, `AnchorAngle`, and `AnchorOffset` may be used to extract information from an existing anchor, as summarized in Table 7.3. (As usual, the argument *p* may also be a simple coordinate argument, in which case it is upgraded to an anchor as described above. The function `CanvasRayAngle[{`p_1`,`p_2`}]` returns the angle between two points *on the canvas*, which is in general not what would be obtained by naively feeding their mathematical coordinates into `ArcTan`, unless the scaling of the *x* and *y* axes on the canvas is identical.

Table 7.4 Visualizing anchors.

`FigAnchorMarker[`*anchor*`]`	Visually displays the point, orientation, and text offset (optional) of the given *anchor*.

`FigAnchorMarker.` `FigAnchorMarker[`*p*`]` displays the point (as a dot) and orientation (as a line) of the given anchor *p*. A framed text box illustrating the text offset may also *optionally* be drawn by giving an option `Text` (*e.g.*, `Text->"    "`).

Drawing a `FigAnchorMarker` (Table 7.4) can be useful, as you are preparing a figure, for understanding what an anchor really represents — especially if the results in the figure are not agreeing with what you expect! Just to be clear, `FigAnchorMarker` is meant purely as a visual aid or debugging tool in preparing the figure. You would not normally include a `FigAnchorMarker` in your final figure.

The line length, in printer's points, is controlled by the option `Length`. The appearance of the `FigAnchorMarker` is controlled by all the usual options (`PointSize`, `LineColor`, *etc.*) summarized in Sec. 8, but by default overridden to `PointSize->5`, `LineColor->Red`, `TextFrame->True`, `TextBackground->LightGray`, and `Layer->4`.

`ShowAnchor`.

7.1.4 Generating anchors from objects

Table 7.5 Function for obtaining an anchor from an object.

`GetAnchor[`*object*`,`*name*`]`
`GetAnchor[`*object*`,`*name*`,`*argument*`]`
	Returns an anchor at position *name* on a previously-drawn figure object *object*, with an optional *argument*.

`GetAnchor`. Most often, you will generate an anchor automatically from an existing figure object, as `GetAnchor[`*object*`,`*name*`,`*argument*`]` (Table 7.5). Here *name*, represents a named position. The possibilities vary depending on the type of figure object and are documented for each different object as part of its description, *e.g.*, Table 11.8 for `FigRectangle`. In general, *name* will be a descriptive name for the anchor's location on the object, such as `Left`, `Right`, `Tail`, or `Head`,[4] *e.g.*,

```
    GetAnchor[obj,Left]
```

An additional argument might also be accepted or required. For instance, for a `FigRectangle`, we may also give a number indicating *where* on the `Left` side, *e.g.*,

```
    GetAnchor[obj,Left,0.25]
```

7.1.5 Displacing and rotating anchors

Table 7.6 Functions which generate new points or anchors from existing points or anchors.

`DisplacePoint[`p`,`d_1`,`d_2`,...]`	Returns a point obtained by starting from point (or anchor) p and moving by successive displacement vectors d_1, d_2, ...
`ProjectPoint[`p`,`*direction*`,`*coord*`]`	Returns a point obtained by starting from point (or anchor) p and projecting to *coord* along the given *direction*.
`DisplaceAlongAnchor[`a`,`*dist*`]`	Returns a point obtained by starting from anchor a and moving a distance *dist* in printer's points in the direction given by the anchor's orientation angle θ.
`CentroidPoint[{`$p1$`,`$p2$`,...}]`	Returns a point at the centroid of points p_1, p_2, ...
`RotateAnchor[`a`,`θ`]`	Returns an anchor obtained by starting from anchor a and rotating the angle θ.

[4] All of the anchor names which you are likely to use in practice (like the examples `Left`, `Right`, `Tail`, and `Head` just given) are Mathematica symbols. However, in the documentation you will see a few more obscure examples, meant primarily for SciDraw's internal use, which are strings (like `"PanelLetter"`). These have been left as strings to avoid unnecessarily adding new symbols (and thus the possibility of future name clashes) to the Mathematica namespace.

Some functions for calculating new points (or anchors) from existing points (or anchors) are summarized in Table 7.6.

`DisplacePoint`. The `DisplacePoint` function generates a new point by moving a given vector displacement "relative to" a given point. (You can actually add several displacements d_1, d_2, …at once.) A displacement may be given as $\{dx, dy\}$, `Scaled[`$\{dx, dy\}$`]`, or `Canvas[`$\{dx, dy\}$`]`, with meanings analogous to those described for coordinate points in Table 7.1, or as `None`, which is equivalent to `{0,0}`. Thus, for instance,

```
DisplacePoint[{3.4,5.6},Canvas[{0,10}]]
```

starts from the coordinate point $(3.4, 5.6)$ in the current panel's coordinate system and moves upward by 10 printer's points. As another example, if `"circle"` is the name of an object,

```
DisplacePoint[GetAnchor["circle",Right],Canvas[{10,0}]]
```

lies 10 printer's points off its right hand side.

Note that if `DisplacePoint` is given an anchor *a* as its argument, the text offset and orientation information is preserved. `DisplacePoint[`a`,`d_1`,...]` may therefore be thought of as "parallel transport" of an anchor.

`ProjectPoint`. The `ProjectPoint` function generates a new point by projecting to a given horizontal position (*x*-coordinate) or vertical positin (*y*-coordinate). The argument *direction* may be `Horizontal` or `Vertical`. The new coordinate value *coord* may be given in any of the various ways described for "individual coordinates" in Sec. 7.1.1, *e.g.*, as `Scaled[`x_s`]`, or by giving an anchor from which the horizontal position should be taken. Note that if `ProjectPoint` is given an anchor *a* as its argument, the text offset and orientation information is preserved, much as for `DisplacePoint`.

`DisplaceAlongAnchor`. `DisplaceAlongAnchor[`*a*`,`*dist*`]` is similar to `DisplacePoint`, but it takes a *distance* to move, rather than a vector *displacement*, as an argument, and it makes use of the anchor orientation angle θ to determine the *direction* in which to move relative to the anchor. The distance *dist* is in printer's points and may be positive or negative (or zero). For example, if `"arrow"` is the name of an arrow, the following code would draw a circle 10 printer's points in front of the head of the arrow:

```
FigCircle[DisplaceAlongAnchor[GetAnchor["arrow",Head],10]];
```

`CentroidPoint`. The `CentroidPoint[`$\{p_1, p_2, \ldots\}$`]` function may be used with any set of one or more points or anchors. Although this function accepts an arbitrary number of points, the most useful case in practice is the *midpoint* of two points p_1 and p_2.

`RotateAnchor`. `RotateAnchor[`a, θ`]` returns an anchor obtained by taking the anchor (or point) *a* and rotating the orientation by the angle θ. The angle may be a number, representing an angle in radians in the counterclockwise sense, or `None`.

7.2 Curves

Several types of figure object take not just a *single* point as their argument but a *list* of two or more points, representing a curve. Examples include `FigLine[`$\{p_1, p_2, \ldots, p_n\}$`]`, `FigPolygon[`$\{p_1, p_2, \ldots, p_n\}$`]`, and `FigArrow[`$\{p_1, p_2, \ldots, p_n\}$`]` (described in detail in Sec. 11).

7.2.1 Points on curves

The discussion in Sec. 7, on how to specify points as arguments, still applies to the points in a curve specification $\{p_1,p_2,\ldots,p_n\}$. Thus, each of these points p_1, p_2, ..., p_n may be given simply as $\{x, y\}$ or by any of the more other ways of specifying a point in Table 7.1. They may also be given as anchors, in any of the ways shown in Table **??**. For example, if `"r1"` and `"r2"` are the names of rectangles, then a line from the right side of the first to the left side of the second may be drawn as

```
FigLine[{GetAnchor["r1",Right],GetAnchor["r2",Left]}];
```

Alternatively, to illustrate the use of anchor *names* to specify points for a curve, we note that the same result is obtained by

```
GetAnchor⟦"r1right"⟧["r1",Right];
GetAnchor⟦"r2left"⟧["r2",Left];
FigLine[{"r1right","r2left"}];
```

Table 7.7 Curve points specified relative to the head or tail of the curve.

`DisplaceTail[`d_1`,`d_2`,...]`	Returns a point obtained by starting from the *tail* point of the curve and moving by one or more successive displacement vectors d_1, d_2, ...
`ProjectTail[`*direction*`,`*coord*`]`	Returns a point obtained by starting from the *tail* point of the curve and projecting in the given *direction* to the given *coord*.
`AlongTail[`*dist*`]`	Returns a point obtained by starting from the *tail* point of the curve and, if this point has been given as an anchor, moving a distance *dist* in printer's points along the direction given by the orientation angle θ of this anchor.
`DisplaceHead[`d_1`,`d_2`,...]`	Returns a point obtained by starting from the *head* point of the curve and moving by one or more successive displacement vectors d_1, d_2, ...
`ProjectHead[`*direction*`,`*coord*`]`	Returns a point obtained by starting from the *head* point of the curve and projecting in the given *direction* to the given *coord*.
`AlongHead[`*dist*`]`	Returns a point obtained by starting from the *head* point of the curve and, if this point has been given as an anchor, moving a distance *dist* in printer's points along the direction given by the orientation angle θ of this anchor.

For curves, the positions of points along a curve $\{p_1, \ldots, p_n\}$ can also be specified relative to the *tail* of the curve (the first point p_1) or the *head* of the curve (the last point p_n), as summarized in Table 7.7.[5] This is really just a convenient way of accessing the functions `DisplacePoint`, `ProjectPoint`, or `DisplaceAlongAnchor` (Table 7.6), but with the starting point argument understood as the head or tail of the curve. Thus, for instance `DisplaceTail[`d_1`,`d_2`,...]` is really just a convenient shorthand

[5]Our convention of referring to p_1 as the *tail* and p_n as the *head* originates from the case, commonly occuring in figures, in which the curve is for an arrow (Sec. 12).

for `DisplacePoint[`p_1`,`d_1`,`d_2`,`...`]`, and `ProjectTail[`*direction*`,`*coord*`]` is really just a convenient shorthand for `ProjectPoint[`p_1`,`*direction*`,`*coord*`]`.

Table 7.8 Options affecting the positions of the first and last points, for figure objects which take a curve as an argument.

Option	*Default*	
`TailRecess`	`None`	Distance, in printer's points, by which the actual tail point should be recessed from the nominal tail point p_1.
`HeadRecess`	`None`	Distance, in printer's points, by which the actual head point should be recessed from the nominal tail point p_n.

It is sometimes desirable to have the curve stop a bit short of its *nominal* end point, *i.e.*, the end point you give in the argument — for instance, if the "curve" represents an arrow pointing *towards* an object but which you do not want to actually touch the object. Any figure object which takes a curve as its argument therefore also takes the options `TailRecess` and `HeadRecess`, summarized in Table 7.8. These control how far back the tail or head points should be *recessed* (*i.e.*, backed off), in printer's points, from the nominal coordinates given. For instance, in the rectangle example earlier in this section

```
FigLine[
{GetAnchor["r1",Right],GetAnchor["r2",Left]},
TailRecess->10,HeadRecess->10
];
```

provides a separation of 10 printer's points between the ends of the line and sides of the rectangles.

7.2.2 Curves from graphics

SciDraw can also extract curves from Mathematica graphics, for instance, the output of Mathematica's plotting functions. Whenever a figure object requires a curve as its argument, you can give a Mathematica `Graphics` or `ContourGraphics` expression in place of the argument.

Table 7.9 Option for extracting curves from graphics.

Option	*Default*	
`Line`	`1`	Which curve to extract from a graphics object, if it contains several. With `Line->Join`, the curves are all joined end to end.

`Line`. Simple graphics — for intance, a `Plot` of a single function — will often contain only a single curve, in which case the result is unambiguous. However, more complicated graphical output will in general contain several curves. For instance, a `ContourPlot` contains a separate curve for each contour line. Which one will SciDraw extract and use as the curve argument for the figure object? This is controlled by the remaining option `Line` shown in Table 7.9. You can either choose to extract the *n*th curve found in the graphics, with `Line->`*n* (by default, the first curve is used), or you can choose to concatenate all these curves into one long curve with `Line->Join`.

Table 7.10 Extracting curves from graphics.

`GrabPoints[`*graphics*`]`	Extracts one or more curves from Mathematica `Graphics` or `ContourGraphics` expressions.

`GrabCurves.` SciDraw provides a utility function `GrabCurves[`*graphics*`]`, summarized in Table 7.10, which extracts a curve from the given *graphics*, in the fashion just described, and simply returns a list of the points. `GrabCurves` takes the same option `Line->`*n* or `Line->Join` just described above. It also accepts the option value `Line->All`, in which case it produces a list of *all* the curves extracted from *graphics*. This function is useful for debugging, *e.g.*, figuring out *which* curve from *graphics* is the one you want to use in your figure. It is also useful if you wish to use Mathematica to manipulate or transform the list of curve points in some fashion before giving them as an argument to a figure object.

7.3 Relative positions within a rectangle (*i.e.*, text offsets)

In various contexts, it is necessary to specify a "corner" or "side" of an object, or more generally to specify a point within an object by its relative position across an object. The most notable context in which such relative positions occur is in the positioning of text, when we specify the text anchor point's position relative to the label, termed the "offset",[6] via the `TextOffset` option for SciDraw labels. We therefore follow the same convention as used in the Mathematica `Text` primitive: the relative position is given as $\{x_r, y_r\}$, where the coordinates run from -1 to $+1$ across the face of the object, or from `{-1,-1}` at the lower left corner to `{+1,+1}` at the upper right corner. The coordinates x_r and y_r need not be be constrained to the range -1 to $+1$, that is, the relative position can lie beyond the edges of the object. Relative positions are also occasionally used in other contexts, *e.g.*, in designating anchor points for positioning and rotating `FigRectangle` and `FigCircle` objects (Sec. 11.3) .

Table 7.11 Ways of specifying relative positions across an object.

$\{x_r, y_r\}$	Relative position, with coordinates running from -1 to $+1$ across the face of the object.
`Center`	{0,0}
`Left`	{-1,0}
`Right`	{+1,0}
`Bottom`	{0,-1}
`Top`	{0,+1}
`BottomLeft`	{-1,-1}
`BottomRight`	{+1,-1}
`TopLeft`	{-1,+1}
`TopRight`	{+1,+1}

SciDraw also defines more descriptive aliases for the relative relative position values representing each of the sides and corners, and for the center. These are summarized in Table 7.11.

[6]The Mathematica documentation actually uses the term "offset" with two entirely distinct meanings. For the `Text` primitive, the fractional position argument is denoted by *offset* (see **ref/Text**). This is not to be confused with Mathematica's `Offset` notation for indicating points by their offset in printer's points relative to some starting point (x,y) (see **ref/Offset**). The `Offset` notation is not needed by SciDraw since displacements in printer's points can be handled more systematically through reference to canvas coordinates with the `Canvas` notation.

7.4 Rectangular regions

Table 7.12 Ways of specifying rectangular regions.

`{{`x_1`,`x_2`},{`y_1`,`y_2`}}`	A region specified by its left/right/bottom/top *xy* coordinates, in the current panel's coordinate system.
`Scaled[{{`x_{1s}`,`x_{2s}`},{`y_{1s}`,`y_{2s}`}}]`	A region specified in terms of fractional distance across the current panel, from `{0,0}` at the lower left to `{1,1}` at the upper right.
`Canvas[{{`x_{1c}`,`x_{2c}`},{`y_{1c}`,`y_{2c}`}}]`	A region specified by its coordinates in printer's points on the canvas.
`All`	The entire region covered by the current panel, equivalent to `Scaled[{{0,1},{0,1}}]`. If a panel has not yet been defined, this refers to the main area of the canvas.

Some figure objects take a *rectangular region* as an argument. For instance, `FigurePanel` draws a panel covering a region (Sec. 10.1), and `FigRectangle` (Sec. 11.3) draws a rectangle covering a region. Regions are specified in the form $\{\{x_1,x_2\},\{y_1,y_2\}\}$, following the usual form of the Mathematica `PlotRange` option. The region may be given in ordinary *xy* coordinates (the coordinate system of the current panel), in scaled coordinates (as a fraction of the current panel), or in canvas coordinates, as summarized in Table 7.12. These three forms are directly analogous to the three ways of specifying a point in Table 7.1. The region `All` refers to the entire region covered by the current panel.

Table 7.13 Functions which operate on regions or calculate new regions.

`BoundingRegion[{`p_1`,`p_2`,`$\ldots$`,`obj_1`,`obj_2`,`$\ldots$`}]`	Returns a region specification which circumscribes the given points (or anchors) p_1, p_2, …, and objects obj_1, obj_2, …
`AdjustRegion[`*region*`,`*options*`]`	Returns a new region obtained by displacing and/or extending the given *region*.
`RegionPoint[`*region*`,`*relative*`]`	Returns a point at a relative position across the given region.

Furthermore, SciDraw provides a few functions for working with and modifying region specifications, summarized in Table 7.13.

`BoundingRegion`. The `BoundingRegion` function returns a region specification which may be used as the argument for figure objects. The region returned by `BoundingRegion` circumscribes the given set of objects and/or points (or anchors). This is useful for drawing a rectangular *box* around them with `FigRectangle` (Sec. 11.3) or a *bracket* which spans their width or height with `FigBracket` (Sec. 13.2).

`AdjustRegion`. Given a region specification, we may use `AdjustRegion` to obtain from it a displaced region, or a region which has been extended outward (or contracted inward) on one or more of its sides. `AdjustRegion[`*region*`,RegionDisplacement->`*displacement*`]` takes the given *region* and obtains from it a new region specification, translated by the given *displacement*. This *displacement* is specified as described for the displacement arguments to `DisplacePoint` in Sec. 7.1.5. Similarly, `AdjustRegion[`*region*`,RegionExtension->`*extension*`]` takes the given *region* and obtains from it a new region specification, extended by the given *extension*. The *extension* may simply be given as an amount to add in *xy* coordinates, in the form *dd* (same on all sides), $\{dx, dy\}$ (different for horizontal and vertical directions), or $\{\{dx_1, dx_2\}, \{dy_1, dy_2\}\}$ (different for left, right, bottom, and top). Alternatively, a fractional change in the region is specified similarly as `Scaled[...]`, or a distance to be added in printer's points is specified similarly as `Canvas[...]`. Either the *displacement* or the *extension* may be specified as `None`.

`RegionPoint`. The `RegionPoint` function returns a point specification which may be used as the argument for figure objects. The point returned by `RegionPoint[`*region*`,` *relative*`]` is at the given relative position within the given region. The argument *relative* may be given in any of the forms described in Sec. 7.3, that is, as $\{x_r, y_r\}$, running from `{-1,-1}` at the lower left corner to `{+1,+1}` at the upper right corner, or using any of the alternate names described in Table 7.11.

8 Options for figure objects

8.1 `FigObject`: Common default options

A basic set of options are shared by *all* figure objects. These are the options defined for `FigObject`. Default values may be set through `SetOptions[FigObject,...]`. These are the values which will be inherited by all other figure objects if their own value for the option is left as `Inherited`.

8.1.1 Overall appearance

Table 8.1 Options affecting the appearance of all parts of an object.

Option	*Default*	
`Show`	`True`	Whether or not the object should be visible.
`Color`	`Black`	Default color to be used for all parts of object, unless overridden by `LineColor`, `FillColor`, `PointColor`, `TextColor`, *etc.*
`Opacity`	`None`	Default opacity to be used for all parts of object, unless overridden by `LineOpacity`, `FillOpacity`, `PointOpacity`, `TextOpacity`, *etc.*
`Directives`	{}	Additional graphics style primitives to apply to all parts of the object.
`Style`	`None`	The style (or list of styles) from which all option defaults should be taken.

As introduced in Sec. **??** of the user's guide, some options affect the entire object, while others affect only the outline, fill, or text — that is, if the object in question actually *has* an outline, fill, or text. The options which affect the appearance of all parts of the object are summarized in Table 8.1.

`Show`. With `Show->False`, no graphical output is generated. For instance, this is regularly used with `FigurePanel`. Even though the object is invisible, the geometric information (object location and dimensions) is still stored. Thus, `Show->False` may be useful for creating a "phantom" object, which can provide an anchor as you are positioning other objects, or to hide objects in a figure which you might later wish to show again in another version of the figure. Note that, even if `Show->False` is set for the object as a whole, individual parts might actually be shown if this setting is overridden with the `ShowLine`, `ShowFill`, *etc.*, options described below.

`Color`. The color of the entire object (outline, fill, and text) may be set all at once with the option `Color`. Colors are specified using either the standard Mathematica color names (`Red`, `Blue`, *etc.*) or directives (`GrayLevel[...]`, `RGBColor[...]`, *etc.*) (see **guide/Colors**).

SciDraw also defines the much larger "legacy" set of color names, such as `Moccasin` and `Firebrick`, which were available in earlier versions of Mathematica, through version 5 (see **Compatibility/tutorial/Graphics/Colors**). When SciDraw is loaded, it displays a button labeled "View color palette" in the notebook. You can view a chart of the named colors at any time by clicking on this button.

Furthermore, SciDraw accepts `None` as a color. Any part of an object which has color `None` is simply not drawn, just as if `Show` were set to `False`. Note that, even if `Color->None` is set for the object as a whole, individual parts might actually be shown if the `LineColor`, `FillColor`, *etc.*, are specified as described below.

`Opacity`. The opacity (or transparency, if you think in terms of glasses half empty) of the entire object may be set with the option `Opacity`. This should be a number from 0 to 1, as described in **ref/Opacity**. The default value, `Opacity->None`, perhaps slightly counterintuitively, means "opaque", *i.e.*, no opacity *setting* for the object![1]

Warning: Although it is tempting to use object transparency in your drawings, the `Opacity` option should be used with extreme caution, especially in figures intended for publication. Transparency is *not* supported in PostScript or PDF output. If you attempt to export a figure which has been drawn using object transparency, Mathematica will likely fall back on rasterizing the entire figure (producing excessively large, low-quality output, not suitable for rescaling in a paper or in presentation slides), or else Mathematica will attempt to simulate transparency by breaking the graphics into small polygonal regions with different shadings (again producing excessively large output), or else Mathematica might simply freeze up or crash. So use transparency at your own peril…

`Directives`. The `Directives` option need not normally be used. It is provided so that, if new versions of Mathematica introduce new graphics style directives, they can immediately be used with SciDraw.

`Style`. The `Style` option specifies a style (or list of styles) which should be applied to the object. See Sec. 9 for further discussion of styles.

8.1.2 Outline appearance

Table 8.2 Options affecting the object outline.

Option	*Default*	
`ShowLine`	`Default`	Whether or not the outline should be visible.
`LineColor`	`Default`	Color to be used for the outline.
`LineOpacity`	`Default`	Opacity to be used for the outline.
`LineThickness`	`1`	Line thickness to be used for the outline.
`LineDashing`	`None`	Dashing to be used for the outline.
`LineCapForm`	`None`	Cap form (whether or not line ends are rounded) to be used for the outline.
`LineJoinForm`	`{"Miter",3.25}`	Line join style (mitering or beveling) to be used for the outline.
`LineDirectives`	`{}`	Additional graphics style primitives to apply to the outline.

Options which affect the outline of an object are summarized in Table 8.2.

`ShowLine`, `LineColor`, `LineOpacity`. The color (and opacity) for the outline can take the default values for the object as a whole, or they can alternatively be specified separately from the rest of the object. The value `Default` for any of these options (`ShowLine`, `LineColor`, or `LineOpacity`) means that the setting given by `Show`, `Color`, or `Opacity` (Table 8.1) for the object as a whole should be used. Any other value overrides the default settings given in `Show`, `Color`, and `Opacity`. With either `ShowLine->False` or `LineColor->None`, display of the object's outline is suppressed.

[1]If the color directive specified in the `Color` option contains opacity information, that will take precedence over the `Opacity` setting.

`LineThickness`. The `LineThickness` should be a positive number, which gives the thickness in printer's points. Alternatively, any Mathematica thickness directive may be used, as described in **guide/GraphicsDirectives**: `Thin`, `Thick`, `Thickness[...]`, or `AbsoluteThickness[...]`, this last of which is equivalent to just giving a simple number as the option value. The SciDraw default thickness of 1 pt is (intentionally) bolder than Mathematica's default thickness of 0.5 pt.

`LineDashing`. The `LineDashing` should be a positive number, which gives the dash length in printer's points. More complicated dashing patterns may be specified as a list $\{d_1, g_1, \ldots\}$ of alternating dash and gap lengths. Alternatively, any Mathematica dashing directive may be used, as described in **guide/GraphicsDirectives**: `Dotted`, `DotDashed`, `Dashed`, `Dashing[...]`, or `AbsoluteDashing[...]`, this last of which is equivalent to just giving a simple number or list as the option value. SciDraw also accepts `Dashing->None`, which means no dashing.

`LineCapForm`, `LineJoinForm`. These options are provided for completeness, to allow control over the cap form (whether or not line ends are rounded) and join style (mitering or beveling) for the outline. However, there is rarely reason to change these aspects of the line styling away from the defaults. The values for these options should be given in the same form as the arguments to the Mathematica `CapForm[...]` and `LineForm[...]` directives, described in **guide/GraphicsDirectives**.

`LineDirectives`. The `LineDirectives` option need not normally be used. It is provided so that, if new versions of Mathematica introduce new graphics style directives, they can immediately be used with SciDraw.

8.1.3 Fill appearance

Table 8.3 Options affecting the object fill.

Option	*Default*	
`ShowFill`	`Default`	Whether or not the fill should be visible.
`FillColor`	`Default`	Color to be used for the fill.
`FillOpacity`	`Default`	Opacity to be used for the fill.
`FillDirectives`	{}	Additional graphics style primitives to apply to the fill.

Options which affect the outline of an object are summarized in Table 8.3.

`ShowFill`, `FillColor`, `FillOpacity`. The color (and opacity) for the fill can take the default values for the object as a whole, or they can alternatively be specified separately from the rest of the object. The value `Default` for any of these options (`ShowFill`, `FillColor`, or `FillOpacity`) means that the setting given by `Show`, `Color`, or `Opacity` (Table 8.1) for the object as a whole should be used. Any other value overrides the default settings given in `Show`, `Color`, and `Opacity`. With either `ShowFill->False` or `FillColor->None`, display of the object's fill is suppressed.

`FillDirectives`. The `FillDirectives` option need not normally be used. It is provided so that, if new versions of Mathematica introduce new graphics style directives, they can immediately be used with SciDraw.

8.1.4 Point appearance

Table 8.4 Options affecting geometric points.

Option	*Default*	
`ShowPoint`	`Default`	Whether or not the point should be visible.
`PointColor`	`Default`	Color to be used for the point.
`PointOpacity`	`Default`	Opacity to be used for the point.
`PointSize`	`3`	The size (diameter) to be used for the point.
`PointDirectives`	`{}`	Additional graphics style primitives to apply to the point.

Objects may also contain graphical "points", that is, Mathematica `Point` directives (see **ref/Point**). The main example in SciDraw is the `FigPoint` object (Sec. 11.4). Options which affect the appearance of points are summarized in Table 8.4.

`ShowPoint`, `PointColor`, `PointOpacity`. The color (and opacity) for the point can take the default values for the object as a whole, or they can alternatively be specified separately from the rest of the object. The value `Default` for any of these options (`ShowPoint`, `PointColor`, or `PointOpacity`) means that the setting given by `Show`, `Color`, or `Opacity` (Table 8.1) for the object as a whole should be used. Any other value overrides the default settings given in `Show`, `Color`, and `Opacity`. With either `ShowPoint->False` or `PointColor->None`, display of the point is suppressed.

`PointSize`. The `PointSize` should be a positive number, which gives the *diameter* (not radius!) in printer's points. For consistency with Mathematica, any of the point size directives described in **guide/GraphicsDirectives** may also be used: `Tiny`, `Small`, `Medium`, `Large`, `PointSize[...]`, or `AbsolutePointSize[...]`, this last of which is equivalent to just giving a simple number as the option value. However, using any of these directives causes Mathematica to size the point without SciDraw's knowledge, preventing SciDraw from accurately calculating anchor locations relative to the edges of this point. The SciDraw default diameter 3 pt is the same as Mathematica's default.

`PointDirectives`. The `PointDirectives` option need not normally be used. It is provided so that, if new versions of Mathematica introduce new graphics style directives, they can immediately be used with SciDraw.

8.1.5 Text appearance

Table 8.5 Options affecting the graphical appearance of text.

Option	*Default*	
`ShowText`	`Default`	Whether or not the text should be visible.
`TextColor`	`Default`	Color to be used for the text.
`TextOpacity`	`Default`	Opacity to be used for the text.
`TextStyleOptions`	`{}`	Additional style options or directives to apply to the text.

Options which affect the appearance of the textual elements of an object are summarized in Table 8.5.

`ShowText`, `TextColor`, `TextOpacity`. The color (and opacity) for the text can take the default values for the object as a whole, or they can alternatively be specified separately from the rest of the object. The value `Default` for any of these options (`ShowText`, `TextColor`, or `TextOpacity`) means that the setting given by `Show`, `Color`, or `Opacity` (Table 8.1) for the object as a whole should be used. Any other value overrides the default settings given in `Show`, `Color`, and `Opacity`.

`TextStyleOptions`. The `TextStyleOptions` option need not normally be used. It may contain a list of additional style *options* or *directives* in the form accepted as arguments to `Style` (**ref/Style**).

8.1.6 Text font characteristics

Table 8.6 Options affecting the font characteristics (typeface) used for text.

Option	*Default*	
`FontFamily`	`"Times"`	The font family.
`FontSize`	`16`	The font size, in printer's points.
`FontWeight`	`Plain`	The font weight (or boldness).
`FontSlant`	`Plain`	The font slant (or italicization).
`FontTracking`	`Plain`	The horizontal spacing between letters.
`FontVariations`	{}	Options specifying addtional font variations, such as underlining.

Options which affect the font characteristics (or typeface) of the textual elements of an object are summarized in Table 8.6. Common examples would be `FontFamily->"Helvetica"` for the Helvetica font, `FontSize->12` for 12 pt type, `FontWeight->"Bold"` for boldface type, or `FontSlant->"Italic"` for italic type. Detailed information on these options may be found in the Mathematica documentation under **ref/FontFamily**, **ref/FontSize**, *etc.*

8.1.7 Text background and frame

Table 8.7 Options affecting the background and frame for text.

Option	*Default*	
`TextBackground`	`None`	The text background color.
`TextFrame`	`False`	Whether or not to show a frame line around the text.
`TextFrameColor`	`Default`	Color to be used for the text frame line.
`TextRoundingRadius`	`0`	The rounding radius, in printer's points, for the corners of the text background coloring and frame line.
`TextMargin`	`None`	The margin, in printer's points, between the text and the edge of its background coloring and frame line.
`TextPadding`	`False`	Whether or not the text should be padded to a full line height before applying the margin given by `TextMargin`.

The text background can serve to "blank out" drawing elements behind the text which would otherwise be distracting or prevent the text from being readable. A text background and/or frame can also be used to emphasize or set apart (or perhaps simply gratuitously decorate) text. Options which affect the background and frame of the textual elements of an object are summarized in Table 8.7.

`TextBackground`. The `TextBackground` may be any color specification. Alternatively, if the background is intended solely to blank out drawing elements behind the text, `TextBackground->Automatic` should be used. This indicates that the background color should be taken from the background color of the present panel. The default value `TextBackground->None` means that no background is drawn.

`TextFrame`. A frame is only drawn if `TextFrame->True`.

`TextFrameColor`. The color for the frame is by default the same as the color of the text, but it can be changed with the option `TextFrameColor`. The thickness, dashing, or opacity of text frames *cannot* be changed in Mathematica (at least as of version 8).

`TextRoundingRadius`. The `TextRoundingRadius` option controls the rounding of the frame corners. The radius is given in printer's points and may either be a single number r or a pair $\{r_x, r_y\}$. Further discussion of rounding may be found in **ref/RoundingRadius**.

`TextMargin`. The `TextMargin` option controls the separation of the frame and/or the edge of the background coloring from the text. The value is given in printer's points. Different separations may be given for different edges: the option may be a single number for all edges d, different values for the horizontal and vertical edges $\{d_x, d_y\}$, or different values for each edge $\{\{d_L, d_R\}, \{d_B, d_T\}\}$. Further information may be found in the discussion of the Mathematica `FrameMargins` frame option in **ref/FrameMargins**.

`TextPadding`. This option is provided for completeness and consistency with Mathematica. It ensures that the frame is at least as high as a line of text, even if the characters in the label do not take up that full height. Further information may be found in the discussion of the Mathematica `ContentPadding` frame option in **ref/ContentPadding**.

8.1.8 Text positioning

Table 8.8 Options affecting positioning of text.

Option	*Default*	
`TextOffset`	`Automatic`	The offset of the anchor within the text rectangle.
`TextOrientation`	`Automatic`	The text orientation, or rotation about the anchor.
`TextRectify`	`True`	Whether or not to automatically rectify text which would, otherwise, appear inverted.
`TextNudge`	`None`	An additional arbitrary displacment of the text, in printer's points.
`TextBuffer`	`None`	An additional displacement of the text, in printer's points, in the direction from the anchor point to the text center.
`TextBaseBuffer`	`Automatic`	Base displacement for all text, to ensure a minimal gap between text and graphics. *Not normally adjusted by user.*

SciDraw provides extensive possibilities for fine control over text positioning. Options which affect the positioning of the textual elements of an object are summarized in Table 8.8. These all modify the positioning of the text relative to that defined by an *anchor*.

`TextOffset`. The `TextOffset` describes where the anchor point lies within the text bounding box, from `{-1,-1}` at the lower left corner to `{+1,+1}` at the upper right corner. A value of `Automatic` means to use the default offset value specified by the anchor.

`TextOrientation`. The `TextOrientation` describes the orientation of the text baseline. The orientation is specified as an angle θ counterclockwise from the x-axis, as usual for plane polar coordinates. The angle must be given in radians, but recall that the Mathematica constant `Degree` may be used for convenient conversion, *e.g.*, `TextOrientation->45*Degree`. A value of `Automatic` means to use the default orientation angle specified by the anchor object. The value `Horizontal` is equivalent to `0*Degree`, and `Vertical` is equivalent to `90*Degree`. Occasionally it is useful to specify the value `Inverse`, which means inverted (*rotated* by 180°) relative to the default orientation angle specified by the anchor object.

`TextRectify`. What if a curve bends around so that text running tangent to it would be upside down, by which I mean more than 90° away from the usual horizontal direction? What if an arrow is pointing down and to the left, so that text running along it would be upside down? By default, with `TextRectify->True`, SciDraw will *automatically* rectify the text, that is, rotate it by 180° so that it is “right side up”. (Here I mean within 90° of the usual horizontal direction — roughly speaking, so you do not have to stand on your head to read it!) However, this feature may be disabled with `TextRectify->False`.

`TextNudge`, `TextBuffer`. The options `TextNudge` and `TextBuffer` both allow you to shift the position of the text. The distances for both are in printer’s points.

If you simply wish to *nudge* the text by a specific *xy* displacement — regardless of whether or not it is attached to an object and what direction that object happens to be in — you would specify `TextNudge->`$\{d_x, d_y\}$. More often than not, you will want to nudge text vertically, *i.e.*, upward or downward, so SciDraw allows you to specify a vertical nudge in shorthand form, as a single number `TextNudge->`d_y.

On the other hand, `TextBuffer` is specifically relevant to text which is attached to an object, such as text running along a line or an arrow. Often you will simply want adjust the *buffer* space between text and the object it is attached to — move it away from or closer to the object — without worrying about exactly what direction (in terms of x and y components) that happens to be. This is accomplished with `TextBuffer`. For instance, if you are using a heavy line thickness for your arrows, and the text attached to all your arrows is therefore “bumping up” against the lines, you might want to move the text *away* from the arrows, say, by an extra 1 pt, which would be accomplished with `SetOption[FigArrow,TextBuffer->1]`. (A positive value moves the text “away from” the object, a negative value “towards” the object.)

`TextBaseBuffer`. The `TextBaseBuffer` option need not normally be used. It is provided so that, if you encounter special circumstances, you can uniformly override SciDraw’s default spacing between a text label and the graphics to which it is attached. For instance, this might be necessary if you encounter positioning problems with an unusual font or exporting to an unusual format, or if you are systematically using very thick lines for your plots. To help insure at least a minimal hairline gap between text and the graphics it is attached to, SciDraw normally imposes some minimal default “buffer”, in the sense described above under `TextBuffer`, to which any value specified by the user via the `TextBuffer` option is actually then *added*. This base value is given by the option `TextBaseBuffer`. Normally, you would just leave `TextBaseBuffer` at its default value `Automatic`, which SciDraw presently interprets as a buffer of 1 printer’s point.

8.1.9 Layering

Table 8.9 Layer option.

Option	*Default*	
`Layer`	`Automatic`	The drawing layer of the object.

`Layer.` The option `Layer` determines how each graphical element ranks relative to others in terms of being in the “background” or “foreground”, through a *layer* number. With `Layer->Automatic`, the layer number is 1 for drawing elements (lines, fills, and points), 2 for text background coloring (which should normally hide drawing elements but not obscure nearby overlapping text), and 3 for text. If, instead, a numerical value is specified for the `Layer` option for a given figure object, all graphical elements of the object wil be drawn in the given layer. This lets you bring an object to the foreground or push it to the background. Layers are only relevant for the ordering (background to foreground) of elements within the same panel (Sec. 10.1). If panels are nested (inset) within each other, once all objects within the inner panel are generated, the entire panel (panel frame and labels, plus contents) is “flattened”, and elements appear together at the same layer in the outer panel. This layer is normally 1 but can be controlled by the `Layer` option given to the inner `FigurePanel`.

8.2 Attached label options

Table 8.10 Attached label options.

Option	*Default*	
X`Label`	`None`	The label contents.
X`LabelPosition`	`Automatic`	Additional positioning argument to use when generating the label anchor.
X`ShowText`, *X*`TextColor`,..., *X*`FontFamily`,..., *X*`TextBackground`,..., *X*`TextOffset`, *X*`TextNudge`,...	`Default`	*As defined in Tables 8.5–8.8.*

Attached labels, as introduced in Sec. **??** of the user’s guide, are labels which can be drawn along with an object, simply by specifying the label text as an option to the object. These labels are attached to a few possible predefined positions on an object, consisting of some or all of the positions defined by the named anchors described in Sec. 7.1.2. Using attached labels is usually the most convenient alternative for simple labeling tasks. However, it is not as completely flexible as creating the label separately, as a `FigLabel` — for instance, only one attached label can be given for each named anchor point. The options used to generate these labels are summarized in Table 8.10.

***X*`Label`.** Each attached label is named according to the anchor to which it is attached. If the name of the label is *X*, then the text for this label is given as the option *X*`Label`. For example, for a `FigRectangle` (Sec. 11.3), the possible attached label names *X* are `Left`, `Right`, `Bottom`, `Top`, and `Center`. Thus, for `FigRectangle`, the text is given with the options `LeftLabel->`*text*, `RightLabel->`*text*, `BottomLabel->`*text*, `TopLabel->`*text*, or `CenterLabel->`*text*.

***X*LabelPosition.** As just noted, the attached labels are attached at the positions of named anchors. The locations of many of these anchors can modified by the inclusion of an additional *argument*, as indicated in Table **??** of Sec. 7.1.2. The available anchor names and possible additional argument are defined for each type of figure object in the reference section for the given object, *e.g.*, for `FigRectangle` these may be found in Sec. 11.3. The possible values of the option *X*`LabelPosition` are `Automatic` (if there is no extra argument to be used when determining the anchor) or else the value to be used for that argument.

Standard text options. For each of the standard options affecting the graphical appearance of text (Table 8.5), the typeface of text (Table 8.6), or the text background and frame (Table 8.7), there is also a corresponding option which controls that attribute specifically for label *X*. If the value of any of these options is given as `Default` for a specific label, then the label will just use the value of the option specified for the object as a whole. Thus, *e.g.*, corresponding to the option `FontSize`, there is also, for the `Left` label, an option `LeftFontSize`. If the `LeftFontSize` option is left unspecified, as `Default`, then the `Left` label will inherit the value of `FontSize` given for the object as a whole.

9 Styles and advanced option control

9.1 Defining styles

Table 9.1 Defining and inspecting styles.

`DefineStyle[`*style*`, {` *symbol*`->{`*options*`},`...`,` *parent*`,`...`} ]`	Generates a style named *style*, by applying the given options for the given symbols and/or inheriting from the given parent styles.
`StyleOptions[`*style*`]`	Returns the options associated with *style*, for all symbols.
`StyleOptions[`*style*`,`*symbol*`]`	Returns the options associated with *style*, for the given *symbol*.

Description. A style defines default values for options for one or more Mathematica symbols — we are most interested here in defining default option values for figure objects.[1] These default option values later be retrieved and applied to an object, just as if you had set them with `SetOptions`, by giving the option `Style->`*style* to the object. (The default values in the style can still be overridden by giving the object an explicit value for the option.) Or the option values defined in a style can variously be applied to all objects in a figure, or in a given panel of a figure, or for selected sets of objects within a figure, as discussed below in Sec. 9.2. In this fashion, a style can actually provide default options, not just for figure objects, but for any Mathematica function which accepts options, such as the Mathematica `Plot` function or the CustomTicks function `LinTicks`.[2]

`DefineStyle`. The command `DefineStyle[`*style*`,{`...`}]`, summarized in Table 9.1, defines a named *style*. Options may be given for specific symbols, as in `DefineStyle[`*style*`, {`...`,` *symbol*`->{`*option*`->`*value*`,`...`},`...`} ]`. Alternatively, all options specified in a previously-defined "parent" style my be incorporated by reference to the name of that style, as Options may be given for specific symbols, as in `DefineStyle[`*style*`,{`...`,`*parent*`,`...`}]`.

Naming convention. The same naming convention which was recommended for objects, in Sec. 6, applies to style names as well. A style name should typically be given as a string, *e.g.*, `"SpectrumPlot"`. Alternatively, a style name may be given as a list beginning with a string and followed by some other descriptive information, *e.g.*, `{"SpectrumPlot","grayscale"}` or `{"SpectrumPlot","color"}`. This list syntax is especially useful for defining style names which accept variable arguments, as described below.

Styles with arguments. The *style* name given in `DefineStyle[`*style*`,{`...`}]` may involve named patterns, much as in the left hand side of a Mathematica function definition.[3] These named patterns serve as arguments in the style name, which may be used anywhere in the list {...} which defines the style. For example:

```
DefineStyle[
  {"MyCircleStyle",r_},
```

[1]The basic concepts of styles are introduced in Sec. ?? of the user's guide.

[2]Styles are implemented in a standalone package, named `StyleOptions`, which may be loaded and used independently of SciDraw.

[3]See **tutorial/DefiningFunctions** for an introduction to defining functions.

```
    {
    FigCircle->{FillColor->Firebrick,Radius->Canvas[r]}
    }
  ];
```

Combining styles or generating *ad hoc* styles with `MakeStyle`. In place of a style name, one may specify

`MakeStyle[{`*style1*`,`*style2*`,`...`}]`

which is the union of the given styles. This is useful, *e.g.*, if different styles are defined to control different aspects of a figure's appearance (*e.g.*, in a data plot, one style controlling the data symbols and one controlling the data lines). More generally, the argument to `MakeStyle` follows exactly the syntax for the second argument to `DefineStyle`. Thus, one may intersperse explicit option specifications as well, as

`MakeStyle[{`*symbol*`->{`*options*`},`...`,`*parent*`,`...`}]`

Precedence of options. The general rule is that the first (leftmost) specification of an option takes precedence. Thus, if multiple definitions for the default value of a given option arise within a *single option list* for a given symbol, the definition which appears first in the option list takes precedence. Similarly, if *multiple option lists* are provided for a given symbol — either explicitly in the style definition or via references to a parent style — then the default values for options specified in the leftmost option list take precedence over any default values for the same options specified later. This scheme follows Mathematica's own treatment of options and, more generally, replacement rules, where the first applicable rule is the one which is applied. As a consequence, if styles are combined with `MakeStyle[{`*style1*`,`*style2*`,`...`}]`, and if *style1* and *style1* both provide values for the same option, the first style *style1* overrides the second style *style2*.

Retrieving style options with `StyleOptions`. If you are ever in doubt as to the option value definitions which may are implied by a given style, these may be revealed in all their gory detail with `StyleOptions[`*style*`]`. To prune these down to the options for a given *symbol*, use `StyleOptions[`*style*`,`*symbol*`]`. Using `StyleOptions` to inspect the options generated by a style is particularly useful if the definition of a style is complex, *e.g.*, inheriting from parent styles, and perhaps is not not behaving as you expect.

9.2 Using styles

Applying styles to a `Figure`. To apply a style to an entire figure, one uses the `Style` option (Table 8.1), as

`Figure[`...`,Style->`*style*`];`

or

`Figure[`...`,Style->{`*style*`,`...`}];`

for multiple styles. If a style contains options for `Figure`, *e.g.*, `CanvasSide` or `CanvasMargin` (Table 5.3), these will be used by `Figure`. However, applying a style to a `Figure` does much more than simply control the properties of the `Figure` itself. All option values defined in *style* (for figure objects, functions, *etc.*) are set as the default values while the figure *body* is evaluated.

Applying styles to a `FigurePanel`. Similarly, to apply a style to single panel of a figure, one gives the `Style` option to `FigurePanel`. If a style contains options for `FigurePanel`, *e.g.*, `PanelRegion` or `XFrameLabel`, these will be used by `FigurePanel`. Moreover, all options values defined in *style* (for figure objects, functions, *etc.*) are set as the default values while the panel *body* is evaluated. Similar comments apply to `Multipanel`.

Applying styles to an individual figure object. A style may be applied to a given figure object by giving it as the `Style` option (Table 8.1). For example,

```
FigLine[...,Style->style];
```

All option values stored in *style* are temporarily set as the defaults while the object is being constructed. Thus, for instance, *style* can affect the appearance of a `FigLine` if it contains option definitions for `FigLine` itself or, through inheritance, for `FigObject`.[4]

Styles for data plots. Data plots are described in Sec. 16. Here we note that, if a style contains options for `DataCurve`, `DataSymbol`, or `DataFill`, all these options may be applied to a given data plot (along with any options for `DataPlot` itself) by giving the `Style` option to `DataPlot`, as

```
DataPlot[...,Style->style];
```

Table 9.2 Applying styles within a limited scope, *e.g.*, to a group of objects.

`WithStyle[`*style*`,`*body*`]`	Evaluates *body* with options set according to *style*.

`WithStyle`. `WithStyle[`*style*`,`*body*`]` evaluates *body*, temporarily setting all options values defined in *style* (for figure objects, functions, *etc.*) as the default values while *body* is evaluated.

9.3 Overriding options by object name

Table 9.3 Environment in which options can be specified for given figure objects, by pattern matching on the object identifiers.

`SetOptionOverrides[` *patt*`->{`$rule_1$`,`$rule_2$`,...}]` `SetOptionOverrides[` `{`$patt_1$`->{`$rule_1$`,`$rule_2$`,...}, ...}]`	Evaluates *body*, defining special option rules to be used when creating certain objects.

`SetOptionOverrides`. `SetOptionOverrides[`*patt*`->{`$rule_1$`,`$rule_2$`,...}]` or `SetOptionOverrides[{`$patt_1$`->{`$rule_1$`,`$rule_2$`,...}, ...}]` defines special option rules to be used when creating objects with specific names or names matching the given patterns (see **guide/Patterns**).

[4]In principle, a style can be applied to all subsequently-drawn objects of a given type by setting the default `Style` option, for example, `SetOptions[FigLine,Style->`*style*`]`. However, if you are thinking of doing this, it is worth considering first whether or not you might apply the style to the `Figure` or `FigurePanel` as a whole, or via the `WithStyle` environment (described below) instead. I have come to the conclusion that using `SetOptions` to set a style is in general confusing and probably never a good idea. It can be rather difficult to keep track of styles imposed in this way. (Which objects do they actually apply to?) And any `Style` option given subseqently would completely supplant this style, rather than leaving open the option of adding to it cumulatively. The `WithStyle` environment in general provides a more robust approach to imposing a style for several objects at once.

This is extremely useful when a large number of objects are generated automatically using Mathematica's programming tools such as `Do`, but the options for some specific objects require manual "tweaking". It is possible that more than one pattern might match the same object name, in which case the options given by each of these of these rules will be applied, in order (therefore, if the same option is specified more than once, the first appearance of the option will be the one which takes precedence). `SetOptionOverrides` may be applied repeatedly, adding cumulatively to the existing override definitions. These definitions are limited in scope to the current frame or panel.

9.4 Scoping changes to default options

Table 9.4 Environment in which the default option values for figure objects can be locally redefined.

`ScopeOptions[`*body*`]`	Evaluates *body*, localizing any changes to options for figure objects.

`ScopeOptions.` `ScopeOptions[`*body*`]` evaluates *body*, localizing any changes to options for figure objects,[5] so that they do not affect the defaults outside of *body*. The options for the `CustomTicks` functions `LinTicks` and `LogTicks` are localized as well. (However, `ScopeOptions` does not affect the options for other, non-SciDraw Mathematica symbols.) `ScopeOptions` is already automatically applied to the *body* of a `Figure` or `FigurePanel`, so changes to default option values do not "bleed" over to other figures or panels.

[5] The effect is similar to that of a "group" in LaTeX, consisting of the text inside braces. Changes to font properties, configuration parameters, *etc.*, made within a LaTeX group are localized to that group.

10 Panels and axes

Table 10.1 Panel generation with `FigurePanel` and `Multipanel`.

`FigurePanel[{`*body*`}]`	Generates a standalone panel (Sec. 10.1).
`Multipanel[`*body*`]`	Generates a multipanel array, containing the panels in *body* (Sec. 10.2).
`FigurePanel[{`*body*`},{`*row*`,`*col*`}]`	Generates one panel in a multipanel array (Sec. 10.2).

`FigurePanel[{`*body*`},All]`
`FigurePanel[{`*body*`},{`*rowpatt*`,`*colpatt*`}]`
	Generates multiple panels in a multipanel array iteratively (Sec. 10.3).

10.1 `FigurePanel` basics

Description. `FigurePanel` sets a specified rectangular region as the current plotting region and defines a coordinate system for plotting within that region. It also draws various ancillary elements (frame line, frame labels, tick marks and labels, panel letter, and colored background) which serve to demarcate and annotate this region. A `FigurePanel` thus functions both as an *object* unto itself and also as an *environment* in which other objects are drawn.

Arguments. The syntax for `FigurePanel` is summarized in Table 10.1. The basic syntax `FigurePanel[{`*body*`}]` for a standalone panel is pretty — all the fun is in the options which may be given to `FigurePanel`. Additional arguments are defined when `FigurePanel` is used within a multipanel array, as discussed in Sec. 10.2 and Sec. **??**.

Use of braces around the panel body. The panel contains the objects created in the process of evaluating the expression {*body*}. `FigurePanel` requires that the entire body be wrapped in braces, as {*body*}. This is required by SciDraw to enforce readability (in particular, taking advantage of the way Mathematica's front end automatically indents code within braces) and to preempt the frustration arising from what would otherwise be some of the most common typographical errors.

To explain, let us compare the situation if one *does* use braces, *e.g.*,

```
FigurePanel[
  {
    FigLine[...];
    FigLine[...];
    FigLine[...];
  },
  XPlotRange->{-1,1},YPlotRange->{-1,1}
];
```

to the hypothetical situation where we omitted the braces, *e.g.*,

```
FigurePanel[
  FigLine[...];
  FigLine[...];
  FigLine[...],  (* notice comma not semicolon *)
```

```
    XPlotRange->{-1,1},YPlotRange->{-1,1}
  ];  (* WRONG -- NOT ALLOWED *)
```

Notice how the extent of the body is clear when we use braces, while the body runs in confusingly with the options if we omit the braces. Moreover, the braces remove a common source of error as you edit the contents — in the hypothetical braceless form, to add a new, fourth object, you would need to remember to change the comma (“`,`”) after the last `FigLine` to a semicolon (“`;`”), while, in the braced form, every expression can be terminated uniformly with a semicolon.

If you have previously used LevelScheme, you might be in the habit of separating the objects in the body by commas, to make a list of objects, *e.g.*,

```
  FigurePanel[
    {
      FigLine[...],
      FigLine[...],
      FigLine[...]
    },
    XPlotRange->{-1,1},YPlotRange->{-1,1}
  ];
```

This would be perfectly acceptable, and produces identical output to the example code above, but it is not particularly recommended.
[1]

Table 10.2 `PanelRegion` option for `FigurePanel`.

Option	*Default*	
`PanelRegion`	`All`	The region to be covered by a panel.

`PanelRegion`. By default, the panel covers the entire current drawing region. Alternatively, the region to be covered by the panel may be specified by the option `PanelRegion->`*region* (Table 10.2), in which the *region* may be specified in any of the forms described in Sec. 7.4. (Most commonly, if the main canvas region of a `Figure` is to be entirely covered by a single panel, no `PanelRegion` option is needed.) The given *region* in `PanelRegion->`*region* is the area to be covered just by the frame (or plotting region) of the panel — any frame labels or tick labels may be expected to extend beyond this region. Such an interpretation of *region* ensures that the panel frame size remains predictably constant even as the tick or

[1]If you are wondering what the differences might be, here is a technical note…In LevelScheme the goal was to make a *list* of objects. That is no longer relevant in SciDraw. In SciDraw, the expression which arises as the end result of evaluating {*body*} simply does not matter — it is thrown away! Rather, what actually shows up in the figure depends only upon the figure objects created (as a “side effect”) during the evaluation of {*body*}. If you are an object-oriented programmer, you can think of the figure object function, such as `FigLine[]`, as the *constructor* for an object, which registers that object in a list of objects to later be drawn. Alternatively, if you are a Mathematica programmer, you can think of the figure object function as “sowing” (see **tutorial/CollectingExpressionsDuringEvaluation**) the object into a list of objects to be drawn. Syntactically, in Mathematica, semicolons tie several expressions together to form a compound expression (see **ref/CompoundExpression**), while commas separate arguments to a function or entries of a list. So, if you are wondering what the syntax means to Mathematica, in the recommended form of the {*body*} argument shown above, since the contents of the braces terminate with a semicolon, the returned value of the compound expression is actually `Null`, so the full expression in braces evaluates to a list, with this as its one entry, {Null} — again, irrelevant, since the result is thrown away.

frame labels change.[2]

General appearance options. The appearance of the panel elements is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1), outline options (Table 8.2), and text options (Tables 8.5–8.7). The outline options affect the panel frame, and the text options affect the various panel labels.

Table 10.3 Options for `FigurePanel`, controlling the background.

Option	*Default*	
`Background`	`None`	The panel background fill color.
`BackgroundOpacity`	`None`	The panel background opacity.
`BackgroundDirectives`	`{}`	Additional graphics style primitives to apply to the background.

Background options. A panel background may be drawn, consisting of a solid rectangle covering the entire panel *region*. This will hide any graphics drawn "behind" the panel. The background color is specified through the option `Background`. The remaining options in Table 10.3, `BackgroundOpacity` and `BackgroundDirectives`, are analogous to the corresponding fill directives described in Table 8.2.

Table 10.4 Options for `FigurePanel`, controlling the plot range.

Option	*Default*	
XY`PlotRange`*	`{0,1}`	Coordinate range for plotting or drawing within the panel.
XY`ExtendRange`*	`None`	Fractional amount by which the plot range should be extended.

* This option controls properties which can have separate values for the *horizontal* and *vertical* directions. A table entry *XYoption* here actually represents three different options: options `X`*option* and `Y`*option*, and alternatively just plain *option* (default value `Default`), as described in the text.

***XY*`PlotRange`.** The option `XPlotRange->`$\{x_1,x_2\}$ sets the horizontal plotting range for the panel, and `YPlotRange->`$\{y_1,y_2\}$ sets the vertical plotting range. As discussed in Sec. 3.1.3, it is often organizationally easier (at the cost of slightly more typing) to define the x axis properties through separate options from the y axis properties. This is true even for a standalone panel and becomes imperative in multipanel arrays. However, for consistency with the Mathematica plotting commands, the plot range may also be specified for both axes simultaneously with the option `PlotRange->`$\{\{x_1,x_2\},\{y_1,y_2\}\}$ (see `ref/PlotRange`). The default value is `PlotRange->Default`, which simply indicates that the range is to be taken from `XPlotRange` and `YPlotRange` instead.

***XY*`ExtendRange`.** The *XY*`ExtendRange` options control the extension of the coordinate range covered by the panel, relative to the specified *XY*`PlotRange`, in the spirit of the `PlotRangePadding` option for the usual Mathematica `Plot` commands (see **ref/PlotRangePadding**). The option `XExtendRange->`d gives a fractional amount by which the plot range should be extended in each direction along the x axis, or `XExtendRange->`$\{dx_1,dx_2\}$ gives different fractional amounts for the left and right edges. The option `YExtendRange` is defined similarly for the y axis. Alternatively, the fractional extension for both axes may be given through the option `ExtendRange`, in the form dd (same

[2]This behavior of `FigurePanel` is in intentional contrast to the behavior of Mathematica plotting output, where the frame will shrink or grow as the frame or axis labels change, as illustrated in Sec. 3.1.2 of the user's guide.

on all sides), $\{dx, dy\}$ (different for horizontal and vertical directions), or $\{\{dx_1, dx_2\}, \{dy_1, dy_2\}\}$ (different for left, right, bottom, and top). The value `ExtendRange->Automatic` is a shorthand for a 0.02 (2%) extension on all sides, which is the default extension used by the usual Mathematica plotting functions, such as `Plot`. Alternatively, `ExtendRange->None` gives no extension. (The default value, `ExtendRange->Default`, indicates that the range extensions are to be taken from `XExtendRange` and `YExtendRange` instead.)

Table 10.5 Options for `FigurePanel`, controlling whether of not to draw the frame edges.

Option	*Default*	
XY`Frame`*	`True`	Whether or not to display the frame. The four individual edges may be controlled separately.

* This option controls properties which can have separate values for *each edge* of the panel. A table entry *XYoption* here represents five options: options `X`*option* and `Y`*option* for the bottom and left edges, options `XX`*option* and `YY`*option* (default value `Default`) to override these for the top and right edges, and alternatively *option* (default value `Default`) for consistency with Mathematica's form $\{\{L,R\},\{B,T\}\}$ for frame options.

`Frame` (or *XY*`Frame`). With `Frame->False`, no frame, frame labels, or tick marks will be drawn for the panel. The effect is much as when `Frame->False` is given to the usual Mathematica plotting commands (see **ref/Frame**). However, for completeness, it should be noted that this option follows the general syntax for edge options (discussed in the *General note on edge options* immediately beneath Table 10.6 below). Thus, each of the four edges of the panel can be enabled or disabled independently.

Table 10.6 Options for `FigurePanel`, controlling the frame labels.

Option	*Default*	
XY`FrameLabel`*	`None`	Label to display on each edge.
XY`ShowFrameLabel`*	`Exterior`	Whether or not to show the label on each edge.
XY`FrameLabelPosition`*	`Automatic`	Position of the label on each edge.
XY`TickLabelAllowance`*	`Automatic`	Allowance for tick labels, between panel edge and frame label.
XY`FrameTextColor`*,..., *XY*`FrameFontFamily`*,..., *XY*`FrameTextBackground`*,...	`Default`	*As defined in Tables 8.5–8.7.*

* This option controls properties which can have separate values for *each edge* of the panel. A table entry *XYoption* here represents five options: options `X`*option* and `Y`*option* for the bottom and left edges, options `XX`*option* and `YY`*option* (default value `Default`) to override these for the top and right edges, and alternatively *option* (default value `Default`) for consistency with Mathematica's form $\{\{L,R\},\{B,T\}\}$ for frame options.

General note on edge options. The bottom and left edges of a panel — which may be thought of as the *primary edges* — typically require different treatment from the top and right edges — which may be thought of as the *secondary edges*. In particular, *frame labels* and *tick labels* are typically only displayed on the primary edges, although *tick marks* (without their labels) are also typically displayed on the secondary edges. These distinctions affect the natural default choice of option values for the secondary edges. For some properties (tick mark properties), the secondary edges should normally "mirror" the primary edges, while, for others properties (tick label and frame label properties), the secondary edge should not normally simply mirror the primary edge. We also need the flexibility to break these rules. For instance, the left (primary) edge might be used to label the vertical axis in one set of units, while the right (secondary) edge might be used to label the axis in a different set of units. In this case, the secondary edge requires a different frame label and set of tick marks (and tick labels) from the primary edge.

A robust approach has been developed in SciDraw which takes the "sensible" defaults for the secondary edges, while also allowing these to be overridden, and doing so in a way which systematically works both for standalone panels and for multipanel arrays. For a given *option*, the value for the bottom edge is specified through `X`*option* and the value for the left edge through `Y`*option*. The secondary edges are controlled through the options `XX`*option* for the top edge and `YY`*option* for the right edge. If these are left as `XX`*option*`->Default` and `YY`*option*`->Default`, the aforementioned sensible defaults (spelled out in detail for each option below) are used.

The four options `X`*option*, `Y`*option*, `XX`*option*, and `YY`*option* provide complete control over all four edges of the panel. They are usually the recommended means of controlling the edges, and the only viable route for multipanel arrays. However, as an alternative, in acknowledgement of the syntax of the options for the usual Mathematica plotting commands, just plain *option* is also accepted. The plain form *option* is indeed convenient and recommended when you need to specify a single value for all edges at once, *e.g.*, giving `LineColor->Gray` is simpler and more natural than giving `XLineColor->Gray` and `YLineColor->Gray` separately. In general, the plain form *option* may be used to specify a single value for all sides as *value*,[3] different values for the horizontal and vertical primary edges as {*horizontal*, *vertical*}, or different values for each edge as {{*left*, *right*}, {*bottom*, *top*}}.

In multipanel arrays, an important distinction arises between "exterior" edges at the outer edge of the array (these should have frame labels and tick labels drawn alongside them) and "interior" edges which abut a neighboring panel (and therefore often should not have frame labels and tick labels drawn alongside them). The distinction between interior and exterior edges is handled through the special option value `Exterior`. This value is allowed in place of `True` or `False` for certain options (`ShowFrameLabel` and `ShowTickLabels`) described below. It is interpreted as `True` for exterior edges but `False` for interior edges.

*XY***`FrameLabel`.** The text for the labels on the frame edges ("axis labels") is given through *XY*`FrameLabel`. *Secondary edges:* When left as `Default`, the frame label *text* for the secondary edge mirrors that on the primary edge, but it should be noted that this text will not actually be *displayed* unless the label is enabled with `ShowFrameLabel`, described below.

*XY***`ShowFrameLabel`.** This option handles the eccentricities of frame labeling on exterior *vs.* interior edges and primary *vs.* secondary edges. The default value `Exterior` insures that frame labels appear only on exterior edges. However, in a multipanel array with large gaps between panels, you might wish frame labels to appear even on interior edges. This result may be obtained with `ShowFrameLabel->True` (or `XShowFrameLabel->True` and/or `YShowFrameLabel->True` for separate control of both directions). *Secondary edges:* When left as `Default`, the option value for the secondary edge defaults to `False`, or no frame labels.

*XY***`FrameLabelPosition`.** The position of the frame label along the edge may be specified as *XY*`FrameLabelPosition->`*u*, where `0` represents the lower/left end of the edge and `1` represents the upper/right end of the edge. The default *XY*`FrameLabelPosition->Automatic` is equivalent to `0.5`, or the middle of the edge. Note that the distance of the label *outward* from the edge is calculated

[3]A single value for all edges is only permitted where it is commonly meaningful for all edges to share the same value, *e.g.*, for line style options, but not where the horizontal and vertical axes are generally expected to require distinct values, *e.g.*, for frame labels. An exception is that for many options the single value `None` is permitted, even if a single value is not otherwise permitted.

automatically from the dimensions of the tick mark labels, to place the frame label just beyond these. However, adjustments may be made using *XY*`FrameLabelTextBuffer`, *XY*`FrameLabelTextNudge`, or *XY*`FrameLabelTextMargin`.

***XY*`TickLabelAllowance`.** The trickiest part of positioning a frame label, without automated help, would be to position it just far enough from the frame edge to allow room for the tick labels.[4] With *XY*`TickLabelAllowance->Automatic`, `FigurePanel` calls upon Mathematica to calculate the actual dimensions of the text for all tick labels, and allows just enough room for the largest tick label. However, this automatically calculated separation between the edge and the frame label may be overridden with *XY*`TickLabelAllowance->`*d*, where the distance *d* is in printer's points. This may be useful, for instance, if you wish to align labels for different panels all at the same distance from their respective panel edges.

Frame label appearance options. The remaining options in Table 10.6 control the text formatting for the frame labels, following the usual conventions of Tables 8.5–8.7.

Table 10.7 Options for `FigurePanel`, controlling the frame tick marks and tick labels.

Option	*Default*	
XY`Ticks`*	`Automatic`	Tick specification for each edge.
XY`ShowTicks`*	`True`	Whether or not to show ticks on each edge.
XY`TickLengthReference`*	`Automatic`	A base length (typically comparable to the width or height of the panel), in printer's points, used in the calculation of the lengths of the tick marks.
XY`TickLengthScale`*	`1`	Scale factor by which to adjust tick lengths, relative to default length.
XY`ShowTickLabels`*	`Exterior`	Whether or not to show the tick *labels* accompanying the tick marks on each edge.
XY`TickLabelRange`*	`Automatic`	Coordinate range in which to permit tick labels
XY`TickFontSizeFactor`*	`0.85`	Tick font size, as a fraction of `FontSize`, if *XY*`TickFontSize` is given as `Automatic`.
XY`TickTextColor`*,…, *XY*`TickFontFamily`*,…, *XY*`TickTextBackground`*,…	`Default`	*As defined in Tables 8.5–8.7, except XY*`TickFontSize` *has default value* `Automatic`*.*

* This option controls properties which can have separate values for *each edge* of the panel. A table entry *XYoption* here represents five options: options `X`*option* and `Y`*option* for the bottom and left edges, options `XX`*option* and `YY`*option* (default value `Default`) to override these for the top and right edges, and alternatively *option* (default value `Default`) for consistency with Mathematica's form $\{\{L,R\},\{B,T\}\}$ for frame options.

***XY*`Ticks`.** The specification for `Ticks` may be given in the form of a standard Mathematica tick specificication (see **ref/Ticks**), that is, as a list of tick marks (with information on their lengths, labels, *etc.*) or else as a function which can automatically generate such a list of tick marks (given, as its arguments, the range to be covered). You may use `LinTicks` (or `LogTicks`) from the `CustomTicks` package to generate a list of tick marks to give for this option (see the CustomTicks guide included with SciDraw). With *XY*`Ticks->None`, no tick marks are drawn. With *XY*`Ticks->Automatic`, the `LinTicks` function is

[4] LevelScheme users will fondly (?) remember having to manually adjust the `BufferB` and `BufferL` options to `FigurePanel`, by trial and error, for this purpose.

used. Therefore, any option values set for `LinTicks` — with `SetOptions` or via styles (Sec. 9) — will affect the generation of ticks for the panel. *Secondary edges:* When left as `Default`, the ticks on the secondary edge mirror those on on the primary edge, although it should be noted that the *labels* on these ticks may not actually be displayed for the secondary edge, as controlled through `ShowTickLabels`, described below.

*XY***`ShowTicks`.** By default, tick marks are shown on all edges, but this can be disabled for either the horizontal or vertical axes with *XY*`ShowTicks->False`. *Secondary edges:* When left as `Default`, the secondary edge mirrors the primary edge.

*XY***`TickLengthReference`.** This option can normally be left at its default value `Automatic`, but its functioning should be described, since this relates to an important aspect of SciDraw's treatment of tick marks. Mathematica tick mark specifications — including those automatically generated by `LinTicks` for the `FigurePanel` edges — contain a *length* (see **ref/Ticks**). For the usual Mathematica plotting commands, this *length* indicates how long the tick mark should be "as a fraction of the distance across the whole plot". However, this *length* must be reinterpreted in the context of panels and multipanel figures. In fact, even for a single panel, the "distance across the whole plot" can be quite different in the horizontal and vertical directions, leading to a jarring disparity in tick mark lengths. It is visually preferable to have the same tick mark length on the bottom/top edges as on the left/right edges, and therefore to use a common reference length for all these sets of tick marks. In a figure with many side-by-side panels, tick mark lengths calculated with reference to the distance across the entire figure will be overly long in proportion to a single panel. The same problem arises for a small inset panel, which normally requires shorter tick marks than those for the larger panel. Therefore, `FigurePanel` draws tick marks for the various edges as a fraction of a reference length more suited to the panel being drawn. With *XY*`TickLengthReference->Automatic`, the default, the reference length for drawing tick marks on all edges is not the "distance across the whole plot" but rather is the average of the width and the height of the panel being drawn. This length may alternatively be given via this *XY*`TickLengthReference` option, specified in printer's points. *Secondary edges:* When left as `Default`, the secondary edge mirrors the primary edge.

*XY***`TickLengthScale`.** This option allows for all tick marks to be uniformly scaled.[5] For example, `TickLengthScale->2` would double the length of all tick marks on a panel. The length on which *XY*`TickLengthScale` acts is the tick length specified in the tick mark list given to `Ticks` (or, if that was `Automatic`, the length of the default ticks generated by SciDraw using `LinTicks`), after it has already been scaled according to the `TickLengthReference` as described above. *Secondary edges:* When left as `Default`, the secondary edge mirrors the primary edge.

*XY***`ShowTickLabels`.** This option handles the eccentricities of tick mark labeling on exterior *vs.* interior edges and primary *vs.* secondary edges. The default value `Exterior` insures that tick labels appear only on exterior edges. However, in a multipanel array with large gaps between panels, you might wish tick labels to appear even on interior edges, which may be obtained with `ShowTickLabels->True` (or `XShowTickLabels->True` and/or `YShowTickLabels->True` for separate control of both directions). *Secondary edges:* When left as `Default`, the secondary edge defaults to `False`, or no tick mark

[5]It is worth noting that the mode of controling tick lengths provided by *XY*`TickLengthScale`, as an option for the *axis*, diverges from the standard approach in Mathematica, which is that the lengths of tick marks can only be controlled on a tick-by-tick basis, through the list of tick marks given as the `Ticks` option (see **ref/Ticks**). There is some redundancy here, in that one *could* implicitly adjust the default lengths by for tick marks on SciDraw panels by resetting the default value of the eponymous `TickLengthScale` option to `LinTicks`. However, it is usually more convenient to be able to do the adjustment to the tick lengths as an option to `FigurePanel`. Indeed, tick length adjustments would be virtually impossible to achieve on an axis-by-axis basis through options to `LinTicks`, except by explicitly giving options *XY*`Ticks->LinTicks[...,TickLengthScale->...]` for each axis.

labels.

***XY*TickLabelRange.** Special handling of tick labels is required to prevent unsightly labels at the extremes of the coordinate range, dangling beyond the edges of the panel — and perhaps overlapping with neighboring panels. The solution is imperfect (human intervention might always be necessary in unusual cases) but provides a pretty good fix for most situations. If the *XY*`TickLabelRange` option is left at its default value, `Automatic`, then tick labels are limited to the given *XY*`PlotRange` — whereas, the actual coordinate range covered by the plot can (and usually should) be made a bit larger using the *XY*`ExtendRange` option (most simply by specifying `ExtendRange->Automatic`, as described above). With *XY*`TickLabelRange->All`, all tick labels are shown, to the edge of the frame, regardless of *XY*`PlotRange`. A range may also be given explicitly, as *XY*`TickLabelRange->`$\{x_1, x_2\}$.

Tick label appearance (including *XY*TickFontSizeFactor). Typically, for a readable and natural appearance, tick labels should be $\sim 15\%$ smaller than the panel frame labels (that would be about 1 pt to 2 pt smaller, for typical font sizes). The option *XY*`TickFontSize` by default has the special value `Automatic` — which is not normally otherwise defined for `FontSize` options in SciDraw.[6] In this case, the font size for the tick labels is derived from the value of the `FontSize` option for the whole panel, by multiplying this font size by the value of the option *XY*`TickFontSizeFactor` (default `0.85`). The remaining options in Table 10.7 for the appearance of the tick label text follow the usual conventions of Tables 8.5–8.7.

Table 10.8 Options for `FigurePanel`, controlling the panel letter. See also Table 10.14 for automatic panel letter generation in multipanel arrays.

Option	*Default*	
`PanelLetter`	`Automatic`	The panel letter text, which may be automatically generated.
`PanelLetterPosition`	`Automatic`	The corner in which the panel letter appears and how far it is inset from that corner.
`PanelLetterTextColor`,..., `PanelLetterFontFamily`,..., `PanelLetterTextBackground`,...	`Default`	*As defined in Tables 8.5–8.7.*

PanelLetter. The text for a panel letter label is specified as `PanelLetter->`*text*. The default `PanelLetter->Automatic` produces no label for a standalone panel and an automatically generated label, described further in Sec. 10.2, for panels in a multipanel array. Setting `PanelLetter->None` disables the panel letter even in a multipanel array.

PanelLetterPosition. The panel letter position is specified through `PanelLetterPosition` as $\{$*corner*,*indent*$\}$. Here *corner* specifies a starting corner for positioning the label (more generally, a "relative position" within the panel, specified as summarized in Table 7.11) and *indent* is the amount in printer's points by which the label should be indented, both horizontally and vertically, from that corner. The position used when `PanelLetterPosition->Automatic` is $\{$`TopLeft,15`$\}$.

Panel letter appearance options. The remaining options in Table 10.8 control the text formatting for the panel letter, following the usual conventions of Tables 8.5–8.7.

[6]The value `Automatic` here is not to be confused with the usual value `Default`, which here would make the tick font size simply *equal* to the `FontSize` option for the panel.

Table 10.9 Options for `FigurePanel`, controlling the rendering of the objects within the panel.

Option	*Default*	
`Clip`	`True`	Whether or not contents should be clipped to the rectangular region covered by the panel.
`Rasterize`	`False`	Whether or not contents should be rasterized.
`ImageResolution`	`300`	Resolution for rasterization, in dots per inch.
`Layer`	`Automatic`	*As defined in Table 8.9, but see text.*

`Clip`. By default, all graphics drawn within the panel will be clipped to the boundaries of rectangular region covered by the panel. However, with `Clip->False`, graphics will be allowed to extend out to the edge of the canvas (or, if this panel is nested within another, at least out to the edge of the enclosing panel). Finer control of which drawing elements are clipped is obtained by using `FigureGroup`, described in Sec. 10.6. [**UNDER CONSTRUCTION:** The clipping process is under development and subject to change.]

`Rasterize` & `ImageResolution`. To reduce the output size of complicated figures (*e.g.*, with thousands of data points) for inclusion in a journal article, it is sometimes necessary to rasterize the figure. This is a compromise, since rasterization generally yields an inferior, fuzzy appearance compared to vector graphics. If the entire figure is rasterized, *e.g.*, using external drawing or image-processing software, the result is thus inferior output for the entire figure — not just the offending data, but also the panel edges, tick marks, and labels — which is particularly harmful to readability. SciDraw provides the alternative of rasterizing just the panel contents, but leaving the panel frame, ticks, and labels as high-quality vector graphics and text. Rasterization is requested with `Rasterize->True`, and the resolution can be specified through the `ImageResolution` option. Finer control of which drawing elements are rasterized is obtained by using `FigureGroup`, described in Sec. 10.6.[7] [**UNDER CONSTRUCTION:** The rasterization process is under development and subject to change.]

`Layer`. This option for `FigurePanel` controls the layer in which the entire panel (not just the panel frame and labels, but also the panel contents given in {*body*}) appears relative to any objects outside the panel. See Sec. 8.1.9 for further discussion.

Table 10.10 Special option for providing a name for a `FigurePanel` object.

Option	*Default*	
`ObjectName`	`None`	Object name for the panel.

`ObjectName`. There are instances in which it is useful to name a panel, as when you would later wish to attach annotations (labels, lines, *etc.*) to the edges of the panel. However, for technical reasons relating to the Mathematica expression evaluation process, the object name for a `FigurePanel` cannot be specified as "`FigurePanel`⟦*name*⟧`[...]`", according to the usual syntax for figure objects. Thus, an alternative route to "naming" a panel is provided, which is by giving it the option `ObjectName->`*name*.

`Anchors`. The anchors which may be generated from a `FigurePanel` object include all the anchors — `Left`, `Right`, `Bottom`, `Top`, and `Center` — defined for `FigRectangle` (Sec. 11.3), as summarized in Table 11.8. The anchors are defined with respect to the rectangular frame of the panel,

[7] For technical reasons, rasterization is not presently supported for panels containing graphics included through `FigGraphics` and `FigInset` (Sec. 14).

that is, the plotting region. For completeness, we note that a `"PanelLetter"` anchor is defined. This anchor is meant primarily for internal use in positioning the panel letter label, which is constructed as the corresponding attached label.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Left`, `Right`, `Bottom`, `Top`, and `Center` anchor positions. The positioning parameters, as summarized in Table 11.8, may be specified through the *X*`LabelPosition` options. Note that these labels would *not* normally be used for axis labels, which are instead specified with *XY*`FrameLabel`*, as described above. They are mainly provided for completeness and consistency with the `FigRectangle` labels, but they may also be used for an external panel label, analogous to the Mathematica `PlotLabel` option for the usual Mathematica plotting commands (see **ref/PlotLabel**).

10.2 Multipanel arrays

Description. `Multipanel` is used to define the geometry and settings for a rectangular array of panels with shared axes.

Arguments. The syntax for `Multipanel` and `FigurePanel`, for use in generating a standalone panel, is summarized in Table 10.1. Within the *body* given to `Multipanel`, each individual panel is generated with `FigurePanel[`{*body*}`,`{*row*`,`*col*}`]`, as discussed further below. Beyond this basic syntax, there is powerful support for generating several panels in an "iterated" fashion with a single `FigurePanel` command, as discussed below in Sec. 10.3 — for these purposes, a pattern may be given in place of {*row*,*col*}, or the form `FigurePanel[`{*body*}`,All]` may be used.

Table 10.11 Basic geometry options for `Multipanel`.

Option	*Default*	
`Dimensions`	`{1,1}`	The number of rows and columns within a multipanel array.
`PanelRegion`	`All`	The region to be covered by a panel.

Basic multipanel geometry. The basic geometry options for `Multipanel` are summarized in Table 10.11. The option `Dimensions->`{*rows*`,`*cols*} is used to specify that the multipanel array should have the given numbers of rows and columns {*rows*`,`*cols*}. The region to be covered by the multipanel array may also be specified with the option `PanelRegion->`*region*, much as described for `FigurePanel` in Sec. 10.1. The given *region* is the area to be covered just by the frames (or plotting regions) of the panels, and any frame labels or tick labels may be expected to extend beyond this region. If no *region* is specified, the value `All` is assumed. Thus, if the entirety of the main canvas region of a `Figure` is to be covered by a multipanel array, as is by far the most commonly the case, no `PanelRegion` option is needed.

Body contents. The multipanel array contains the `FigurePanel` objects given in the argument {*body*}. These are constructed as `FigurePanel[`{*body*′}`,`{*row*`,`*col*}`]`, where {*body*′} is now the body of the panel itself (Table 10.1). The indexing follows the usual ordering used for matrices in mathematics, *i.e.*, row index increasing *downward*:

$$\begin{array}{cccc} (1,1) & (1,2) & \cdots & (1,cols) \\ (2,1) & (2,2) & \cdots & (2,cols) \\ \cdots & \cdots & \cdots & \cdots \\ (rows,1) & (rows,2) & \cdots & (rows,cols) \end{array}$$

This ordering is also consistent with the ordering used by the Mathematica `Grid` (see **ref/Grid**) and `GraphicsGrid` (see **ref/GraphicsGrid**) constructs. For the reasons discussed in the *Note on panel body syntax* in Sec. 10.1, it may be more readable to enclose the {*body*} in braces, *e.g.*,

```
Multipanel[
  {
    FigurePanel[...,{1,1}];
    FigurePanel[...,{1,2}];
    ...
  },
  Dimensions->{4,2},
  XPlotRange->{-1,1},...
];
```

However, the braces are not strictly required for `Multipanel`, and, indeed, we will see that they are not particularly necessary for readability when we use automatic "iterated" generation of panels in Sec. 10.3, *e.g.*,

```
Multipanel[
  FigurePanel[...,All],
  Dimensions->{4,2},
  XPlotRange->{-1,1},...
];
```

Table 10.12 Layout adjustment options for `Multipanel`.

Option	*Default*	
`XPanelSizes`	`1`	List of column widths on relative scale, or single width shared by all columns.
`YPanelSizes`	`1`	*Similarly for row heights.*
`XPanelGaps`	`0`	List of intercolumn gap widths on a relative scale, or single width shared by all gaps.
`YPanelGaps`	`0`	*Similarly for interrow gap heights.*
`XPanelGapsExterior`	`False`	List characterizing the intercolumn gaps as interior or exterior, or a single value shared by all gaps.
`YPanelGapsExterior`	`False`	List characterizing the interrow gaps as interior or exterior, or a single value shared by all gaps.
XY`EdgeExterior`	`Automatic`	Whether or not edge is considered exterior, for finer panel-by-panel control than above options.

Multipanel sizing and spacing. By default, all columns of panels are of equal width, all rows of panels are of equal height, and there are no gaps between. However, arbitrary proportions for the columns, rows, and gaps between them can be specified using `XPanelSizes`, `YPanelSizes`, `XPanelGaps`, and `YPanelGaps`. The columns and intercolumn gaps fill the available horizontal space, keeping the proportions given in these options, so only the proportions matter. For instance, doubling the values of both `XPanelSizes` and `XPanelGaps` has no effect.[8] For `XPanelSizes`, either a single value may be given

[8]More precisely, the only effect is on the definition of the "column width" unit discussed below under *More on coordinates*.

(but typically this would be left as `1`) or a list of values may be given for each column. For `XPanelGaps`, either a single value may be given (say, `0.05` for a 5% gap) or a list of values for the different intercolumn gaps. A similar discussion applies to the rows (`YPanelSizes`) and interrow gaps (`YPanelGaps`).

Multipanel exterior edge control. The options `XPanelGapsExterior` and `YPanelGapsExterior` determine whether or not the panel edges bordering each of the intercolumn gaps should be taken as exterior panel edges. If a gap is marked as exterior (`True`), then the edges bordering it are considered to be exterior.

Finer panel-by panel control is provided by the *XY*`EdgeExterior` family of options. Recall from above that, values may be specified as, *e.g.*, panel-by-panel lists of values, or lists of rules for the value to be determined according to the panel indices. When this option is left as `Automatic`, for any given edge, the classification of that edge as interior or exterior is determined by `XPanelGapsExterior` and `YPanelGapsExterior`. *Secondary edges:* When left as `Default`, the secondary edge mirrors the primary edge.

There are a few typical situations in which one would use *XY*`EdgeExterior`: (1) In the very common situation in which *every* panel in a multipanel array is to be treated as a loner, we may simply give the option `EdgeExterior->True` to `Multipanel`. (2) If some panels in a multipanel array are not actually drawn, neighboring edges on other panels, which would normally be "interior" edges, are now effectively on the "exterior" of multipanel array, and you might want frame labels and tick labels to be drawn on them accordingly. (3) If a single panel is drawn spanning multiple rows or columns (using `RegionExtension`), it might actually reach the edge of the multipanel array even though its nominal row and column indices would not indicate this, so again there is a need to override the default assumption as to which edges in a multipanel array are to be treated as exterior.

The distinction of which edges are designated interior and which are designated exterior is only relevant for options which have been given the value `Exterior`.

Panel options within a multipanel array. The remaining options for `Multipanel` are the same as for `FigurePanel`. These are as described in Sec. 10.1, with a few additions discussed below.[9] Essentially, the options given to `Multipanel` are "passed on" to any panel constructed as `FigurePanel[{`*body*`},{`*row*`,`*col*`}]`. For instance,

```
Multipanel[
  {
    FigurePanel[...,{1,1}];
    FigurePanel[...,{1,2}];
    ...
  },
  Dimensions->{4,2},
  XPlotRange->{-1,1},YPlotRange->{-1,1},
  XFrameLabel->textit["x"],YFrameLabel->textit["y"],
  Background->Moccasin
];
```

would cause each panel to have *x* and *y* plot ranges `{-1,1}`, with the *x* and *y* frame labels as given, and a background color `Moccasin`. As for any panel options not explicitly given to `Multipanel`, the default values in effect at the time `Multipanel` is invoked are "frozen in" and used for any panel con-

[9]In fact, `FigurePanel` is the "parent" object of `Multipanel`. That is, the default values for the options for `Multipanel` are inherited from the defaults for `FigurePanel`. Therefore, if a change to panel styling is made by setting the default options for `FigurePanel`, this change will also affect panels in multipanel arrays, even though the options for such panels are, as explained in a moment, determined by `Multipanel`.

structed as `FigurePanel[{`*body*`},{`*row*`,`*col*`}]`, regardless of any changes which might be made to the `FigurePanel` default options — with `SetOptions` or styles — within the `Multipanel` body (these changes would still affect, *e.g.*, inset panels).

Different values for the panel options may be given for different rows or columns of the multipanel array, or even more specifically on a panel-by-panel basis. For the horizontal plot range options or horizontal edge options, with names of the form `X`*option*, the form already defined for `FigurePanel` was

    Xoption->val

However, now, with `Multipanel`, we may also have column-by-column specifications

    Xoption->{val1,val2,...,valcols}

or even a full two-dimensional array of panel-by-panel specifications

```
Xoption->{
  {val1,1,val1,2,...,val1,cols},
  {val2,1,val2,2,...,val2,cols},
  ...
}
```

Similarly, for the vertical plot range options or vertical edge options, with names of the form `Y`*option*, we may have row-by-row specifications

    Yoption->{val1,val2,...,valrows}

or a full array of panel-by-panel specifications as above. The considerations discussed above also apply to the secondary edge options `XX`*option* and `YY`*option*.

However, note that whole-panel options with plain names of the form *option* can only be given a single value, to apply to all panels in the array, rather than panel-by-panel specifications. This constraint, though not entirely desirable, is imposed since, otherwise, syntax ambiguities arise.[10]

Even more flexibility is provided by the rule-based (and pattern-based) specification

    Xoption->{patt$_1$->val$_1$,...}

or similarly for `Y`*option*. For instance,

```
XFrameLabel->{
  {1,_} -> textit["x"],
  {_,_} -> textit["t"]
}
```

chooses the horizontal axis label "*x*" for all panels in the first *row* (any column) and "*t*" otherwise (as the fall-through case). Even when you could specify a list of values row-by-row or column-by-column, as above, the rule-based syntax may be more natural to use, and easier to update if you decide to increase or decrease the number of rows. Note that `RuleDelayed` (*i.e.*, `:>`) may be used (see **ref/RuleDelayed**), to define the option values as a functions of the row or column indices.

Options for individual panels within a multipanel array. Although we have just discussed many ways of controlling the options for panels through `Multipanel`, we should not lose track of the possibility

[10]For instance, for a 2×2 array of panels, would `ExtendRange->{{0.1,0.2},{0.3,0.4}}` be interpreted as single value of the option `ExtendRange`, namely `{{0.1,0.2},{0.3,0.4}}` (*i.e.*, to be applied identically to all panels in the array, and for each panel interpreted as a set of numbers for the four different edges), or four different values of the option `ExtendRange`, namely `0.1`, `0.2`, `0.3`, and `0.4` (*i.e.*, to be applied separately to the four different panels)?

that the value for any option to `FigurePanel` may still also be given directly to `FigurePanel` when the panel is created. We now discuss a few options which are especially meant to be used in this way.

Table 10.13 Options for `FigurePanel`, controlling the geometry of the panel, primarily intended for panels in a multipanel array.

Option	*Default*	
`RegionExtension`	`None`	Amount by which to extend region covered by panel.
`RegionDisplacement`	`None`	Displacement by which to shift region covered by panel.
`LockPanelScale`	`False`	Whether or not to lock axis scales against region adjustment.

`RegionExtension` and `RegionDisplacement`. Although the options `RegionExtension` and `RegionDisplacement` of Table 10.13 are actually valid options for all panels, their main use is in multipanel arrays. Hence, we have deferred their discussion until here. These options would normally be applied on a panel-by-panel basis to one or more individual panels within the multipanel array, rather than to the full set of panels. These options are used to extend or adjust the region covered by the panel. They have the same form and meaning as discussed in the context of `AdjustRegion` (Sec. 7.4). In the context of a multipanel array, the typical form for `RegionExtension` would be $\{\{dx_1,dx_2\},\{dy_1,dy_2\}\}$, with these numbers given in the natural (x,y) coordinates of the multipanel array — "column widths" and "row heights" — as described below under *More on body coordinates*. The normal purpose would be to expand one panel to span multiple rows or columns. Alternatively, `RegionDisplacement` may be used to shift a panel out of alignment with the rest of the grid, for instance, if it would more logically straddle two different rows.

`LockPanelScale`. When the region is adjusted, there are two possible intentions for what should be taken as the plot range: (1) The plot range should be taken as given, *i.e.*, covering the entirety of the resized panel. (2) The axes on the extended region should align with what they *would* have been before extension, so that tick marks line up with those in neighboring panels, regardless of what this leads to in terms of plot range. For instance, a panel might be made taller specifically to allow a larger y plot range than on neighboring panels, but with the intention that the y axis should still align with those of the neighboring panels.

The option `LockPanelScale` provides for these two scenarios. If `LockPanelScale` is `False`, the default, then the plot ranges are taken as given, with no special treatment due to the region adjustment. Setting `LockPanelScale` to `True` "locks" the axes (their scale, *i.e.*, coordinate difference per canvas difference, and their origin) to what they would have been before the region was adjusted. This therefore ostensibly makes them align with the axes of the neighboring panels. The plot ranges are adjusted accordingly.

Note that tick labels will still, by default, be clipped to the originally-specified plot range, for the reasons described in the discussion of the option *XY*`TickLabelRange` (Sec. 10.1), regardless of `LockPanelScale`. Therefore, a manual adjustment of the form *XY*`TickLabelRange->All` or *XY*`TickLabelRange->`$\{x_1,x_2\}$ will normally be necessary to ensure that all ticks are shown, if the range has been increased. An adjustment may also be neccessary to ensure that no unsightly tick labels dangle at the edge of the range, if the range has been decreased.

Table 10.14 Options for `FigurePanel`, controlling automatic panel letter generation.

Option	*Default*	
`PanelLetterBase`	`"a"`	The starting character for the panel lettering sequence.
`PanelLetterDirection`	`Horizontal`	Whether the lettering sequence proceeds across or down.
`PanelLetterOrigin`	`{1,1}`	The multipanel array entry in which the lettering sequence starts.
`PanelLetterDimensions`	`Automatic`	The number of rows and columns in which the lettering sequence is calculated, starting at the given origin as upper left.
`PanelLetterCorrection`	`0`	Number by which to advance the panel lettering sequence.
`PanelLetterDelimiters`	`{"(",")"}`	Text in which to sandwich the panel letter.

`Style` option for panels in a multipanel array. It is worth keeping in mind that a `Style` option given to a panel inside a multipanel array does not have as strong an effect as might naively be expected. A style only sets the *default values* for options, as if one had set these with `SetOptions`. Thus, even if the style given to a panel within a multipanel array specifies options for `FigurePanel`, these will be ignored, in favor of the option values which were already established by the enclosing `Multipanel`. However, the `Style` option can still be of some use for a panel in a multipanel array, if the style is meant to influence the objects drawn *within* the panel, *i.e.*, setting options which will affect the lines (`FigLine`), shapes, plots, *etc.*, within the panel.

Automatic panel lettering. Each panel in a multipanel array can be annotated with a panel letter which is automatically generated from the panel's (*row*,*col*) position in the multipanel array. As noted in Sec. 10.1 (see Table 10.8), this labeling is obtained with `PanelLetter->Automatic` (the default). The standard lettering starts from lowercase "a", beginning at the top left panel, and proceeds in "English reading order" (across then down), with the panel letters sandwiched in parentheses. However, this configuration may be changed using the options in Table 10.14. These options may be used as options to `Multipanel`, to change the lettering sequence for the multipanel array as a whole, or they may be given as options to `FigurePanel` for individual panels within the array, if the lettering for individual panels requires special treatment. The starting letter can be changed, say, with `PanelLetterBase->"A"` for uppercase letters, or `PanelLetterBase->"e"` if the present array is meant to continue a previously-drawn sequence of panels labeled "(a)" through "(d)". This same shift might be accomplished more naturally (*i.e.*, without the need for you, the user, to count through the alphabet, to get to the letter "e") with `PanelLetterCorrection->4`. The option `PanelLetterCorrection` is even more useful if one or more panels within an array is left empty, so that the lettering sequence for subsequent panels must be adjusted by one step, *e.g.*, with `PanelLetterCorrection->-1`, or similarly if more panels are left empty. The origin for lettering can be shifted away away from the top left corner using `PanelLetterOrigin`, say, `PanelLetterOrigin->{1,2}` if the leftmost column of panels will be left unlettered (for instance, perhaps they contain "decorative" diagrams rather than data). Similarly, panels at the right or bottom of the array can be neglected in the lettering sequence using `PanelLetterDimensions` to give the dimensions of the "lettered" region of the array, say, `PanelLetterDimensions->{3,2}`, if the rightmost column of panels in a 3×3 array should be ignored in calculating the panel letter. This lettered region starts from the given `PanelLetterOrigin` as its upper left. The ordering can be changed from "English reading order" (across then down) to "Chinese reading order" (down then across) with

`PanelLetterDirection->Vertical`. And, finally, the delimiters can be changed to any valid Mathematica expressions. For instance, we would use `PanelLetterDelimiters->{"[","]"}` for bracketed letters as “[a]”, or `PanelLetterDelimiters->{"(",Superscript[")","′"]}` (see **ref/character/Prime**) for primed parentheses as “(a)′”.

`Multipanel` body contents beyond just panels. In comparing the syntax of `FigurePanel` with that of `Multipanel` (Table 10.1), the similarities are more than just superficial. It is helpful to think of the `Multipanel` as one big “panel”, which just happens not to have any sort of visible “frame”. Then all the panels of the multipanel array, *i.e.*, the `FigurePanel` objects given as {*body*}, are drawn as insets to this big “panel”. In fact, just like the {*body*} of a `FigurePanel`, the {*body*} of a `Multipanel` may contain other “loose” objects as well, not confined within a `FigurePanel`. These might include arrows (`FigArrow`) to be drawn between panels, or extra labels (`FigLabel`), or perhaps brackets (`FigBracket`) grouping several panels together. These objects are free to extend beyond the *region* of the `Multipanel` (*i.e.*, they will not be clipped to *region*).

Coordinates within a `Multipanel`. Furthermore, much as `FigurePanel` defines a coordinate system for use in its {*body*}, so does `Multipanel`. If, as just described, you include objects (such as arrows or labels) within a `Multipanel`’s *body*, but outside of individual panels, you need to be aware of the coordinates with respect to which these are drawn. You can specify coordinates for these objects in all the usual ways you would expect in SciDraw (Sec. 7.1.1). The meaning of `Scaled` coordinates is clear, as a fractional distance across the whole *region*, *e.g.*, `Scaled[{0.5,1}]` is the top center. But what does a point {*x*,*y*} mean? The *xy* coordinates are measured in “panel widths” and “panel heights”, starting from `{0,0}` at the lower left corner. The most important use arises if you wish to break away from a simple rectangular arrangement of panels, in which case you will specify adjustments to the dimensions and positioning of individual panels using these “panel width” and “panel height” coordinates, through the options of Table 10.13. For an array of equally sized panels, the meaning of one “panel width” and one “panel height” is unambiguous. For an array of unequally sized panels, the meaning is defined by the lists `XPanelSizes` or `YPanelSizes`, respectively. One “panel width” or “panel height” is defined as the width or height of a hypothetical panel of relative size “1” (though the number `1` need not actually appear as an entry anywhere in either of these lists). Thus, *e.g.*, for a multipanel array with three columns and

```
XPanelSizes->{1,2,2},
XPanelGaps->0.1
```

the *x* coordinate will run from 0 to 5.2 (*i.e.*, $1+0.1+2+0.1+2$).

`Anchors`. The anchors which may be generated from a `MultiPanel` object are the same as for a `FigRectangle`, as described in Sec. 11.3. These anchors are summarized in Table 11.8. The anchors are defined with respect to the rectangular boundary of the multipanel array.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Left`, `Right`, `Bottom`, `Top`, and `Center` anchor positions. The positioning parameters, as summarized in Table 11.8, may be specified through the *X*`LabelPosition` options. Since many of the other options which can be given to `Multipanel` are simply “passed through” to the `FigurePanel` objects within the multipanel array, it is worth noting that this is *not* the case for the label options. For instance, a `TopLabel` option will produce a *single* label at the top of the array, not a label at the top of each panel within the array.

10.3 Iterated generation of panels

Motivation. Very often, some or all of the different panels within a multipanel array are essentially identical — in terms of the commands used to generate them — differing only in the data set which is to be plotted in them (and maybe some of the labels that go along with the data set). Certainly, one could generate the *n* panels by cutting and pasting the code for generating one panel, *n* times, and editing the code in the body of each panel. For example, if we were trying to make a 5×5 plot, where the data to be plotted in panel (i, j) is given in `MeasuredResults[`i`,`j`]`, we might try

```
Multipanel[
  {
    FigurePanel[
      {
        (* code for 1st panel's data plot *)
        DataPlot[MeasuredResults[1,1]]
      },
      {1,1}
    ];
    FigurePanel[
      {
        (* code for 2nd panel's data plot *)
        DataPlot[MeasuredResults[1,2]]
      },
      {1,2}
    ];
    ...
    FigurePanel[
      {
        (* code for 25th panel's data plot *)
        DataPlot[MeasuredResults[5,5]]
      },
      {5,5}
    ];
  },
  Dimensions->{5,5},...
];
```

Of course, you are clever, so you would realize that cutting and pasting something 25 times generally means you are overlooking a simpler approach. Isn't that what loops were invented for? You would actually write something like

```
Multipanel[
  Do[
    FigurePanel[
      {
        (* code for {i,j} panel's data plot *)
        DataPlot[MeasuredResults[i,j]]
      },
      {i,j}
    ],
    {i,1,5}, {j,1,5}
  ],
  Dimensions->{5,5},...
  ];
```

After discovering that virtually every multipanel figure I was drawing had this same structure, I realized it could be convenient if SciDraw provided a shortcut for such "iterated" generation of panels. This is what we now summarize.

Iterated syntax. If `FigurePanel` is called as `FigurePanel[{`*body*`},All]`, `FigurePanel` iterates over all possible (row,col) positions in the multipanel array, and evaluates $\{body\}$ once for each position, and draws the corresponding panel.[11] Thus, this single call to `FigurePanel` may be thought of as equivalent to several repeated calls to `FigurePanel[{`*body*`},{`*row*`,`*col*`}]`, once for each position (row,col). The resulting code has the form

```
  Multipanel[
    FigurePanel[
      {
        DataPlot[MeasuredResults[PanelRowIndex,PanelColumnIndex]]
      },
      All
    ],
    Dimensions->{5,5},...
  ];
```

A more selective subset of panels may be generated automatically by using any pattern in place of the `All`, for instance, `{1,_}` to generate just the first row of panels, or `Except[{2,1}]` to leave out the panel in position $(2,1)$.[12]

Table 10.15 Variables defined for use within the *body* of a `FigurePanel` within a multipanel array.

`PanelRowIndex`	Row index *row* (1-based).
`PanelColumnIndex`	Column index *col* (1-based).
`PanelIndices`	Row/column index pair {*row*, *col*}.
`PanelSequenceNumber`	Sequential index for panel, from 1 at the top left, proceeding across then down (English reading order).
`PanelRows`	Total rows in array, *i.e.*, *rows* from `Dimensions` option to `Multipanel`.
`PanelColumns`	Total columns in array, *i.e.*, *cols* from `Dimensions` option to `Multipanel`.

Positional variables. How does the code in the *body* know which panel to generate the contents of, if we don't explicitly write a loop with iteration variables like `row` and `col` in the `Do` loop example above? `FigurePanel` predefines several variables — `PanelRowIndex`, `PanelColumnIndex`, *etc.* — which

[11] FORTRAN programmers might be reminded of the "implied `DO` loop" format specifier in a `WRITE` statement.

[12] Actually, the implementation is really "the other way around", from what would be implied by the presentation in this more natural introduction. `FigurePanel[{`*body*`},{`*rowpatt*`,`*colpatt*`}]` always iterates over all possible values of {*row*,*col*}. Each {*row*,*col*} is checked against the pattern {*rowpatt*,*colpatt*}, and, if it is found to match, a panel is drawn. `FigurePanel[{`*body*`},All]` is just a shorthand for `FigurePanel[{`*body*`},{_,_}]`. And the basic "single panel" syntax `FigurePanel[{`*body*`},{`*row*`,`*col*`}]`, *e.g.*, `FigurePanel[{`*body*`},{1,1}]`, is actually just the special case in which the "pattern" reduces to a literal pattern, here, `{1,1}`.

indicate the panel position, as summarized in Table 10.15. You may use these variables, as you wish in *body* — they are particularly useful in conjunction with `If` and `Switch` statements, to systematically control the contents or appearance of panels, according to their row and column. For example,

```
FigurePanel[
  {
    (* code for (row,col) panel's data plot *)
    ...
    (* only display legend in {1,1} panel *)
    If[
      PanelIndices=={1,1},
      DataLegend[...]
    ]
  },
  All
];
```

Table 10.16 Positional assignment functions defined for use within the *body* of a `FigurePanel` within a multipanel array.

`SetByPanelRow[`*var*`,{`val_1`,`val_2`,`...`}]`	Sets *var* by row index.
`SetByPanelColumn[`*var*`,{`val_1`,`val_2`,`...`}]`	Sets *var* by column index.
`SetByPanelIndices[`*var*`,{` `{`$val_{1,1}$`,`$val_{1,2}$`,`...`},`...`}]`	Sets *var* by {*row*, *col*} indices, or by pattern matching against these indices.
`SetByPanelIndices[`*var*`,{` $patt_1$`->`val_1`,`...`}]`	
`SetByPanelSequence[`*var*`,{`val_1`,`val_2`,`...`}]`	Sets *var* by sequential panel index.

Table 10.17 Positional value functions defined for use within the *body* of a `FigurePanel` within a multipanel array.

`ByPanelRow[{`val_1`,`val_2`,`...`}]`	Returns value determined by row index.
`ByPanelColumn[{`val_1`,`val_2`,`...`}]`	Returns value determined by column index.
`ByPanelIndices[{{`$val_{1,1}$`,`$val_{1,2}$`,`...`},`...`}]`	Returns value determined by {*row*, *col*} indices, or by pattern matching against these indices.
`ByPanelIndices[{`$patt_1$`->`val_1`,`...`}]`	
`ByPanelSequence[{`val_1`,`val_2`,`...`}]`	Returns value determined by sequential panel index.

Obtaining values based on panel position. For convenience, functions are defined which assign values to a variable, depending on position (row, column, or specific panel) in a multipanel array, as summarized in Table 10.16. For instance, `SetByPanelRow[`*var*`,`*values*`]` is short for *var*=*values*`[[PanelRowIndex]]`, where *values* is a list of values. Or, more simply, functions are defined which simply return a value, depending on position (row, column, or specific panel) in a multipanel array, as summarized in Table 10.17. For instance, `ByPanelRow[`*values*`]` is short for *values*`[[PanelRowIndex]]`.

10.4 `FigAxis`

Table 10.18 Standalone axis.

`FigAxis[`*side*`,`*coord*`,`*range*`]`	Generates a freestanding coordinate axis.

Description. A `FigAxis` is used to draw a standalone axis, anywhere within a figure — including tick marks that are linked to the current coordinate system. The options for a `FigAxis` are nearly identical to those for the edges of a `FigurePanel`, so it makes sense to discuss `FigAxis` now, in the context of panels.[13]

Arguments. `FigAxis[`*side*`,`*coord*`,`*range*`]` (Table 10.4) draws an axis positioned according to the arguments. The *side* — `Bottom`, `Left`, `Top`, or `Right` — determines the overall orientation (horizontal or vertical) and configuration (where the labels and tick marks should be drawn relative to the axis). For instance, a `Bottom` axis is like the bottom edge of a panel's frame — horizontal, with labels below and ticks extending above by default. The *coord* specifies the coordinate at which the axis should be placed (this is an *x* coordinate value for a vertical axis, or a *y* coordinate value for a horizontal axis). Note that this coordinate may be given in any of the various ways described for "individual coordinates" in Sec. 7.1.1, *e.g.*, as `Scaled[`x_s`]` for positioning by scaled position across the current panel or by giving an anchor from which the horizontal position should be taken. The *range* specifies the coordinate range the axis should cover (this is a *y* coordinate range for a vertical axis, or an *x* coordinate range for a horizontal axis).

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Table 10.19 Options for `FigureAxis`, controlling the axis label.

Option	*Default*	
`AxisLabel`	`None`	Label to display along axis.
`ShowAxisLabel`	`Exterior`	Whether or not to show the label.
`AxisLabelPosition`	`Automatic`	Position of the label along the axis.
`TickLabelAllowance`	`Automatic`	Allowance for tick labels, between axis and axis label.
`AxisTextColor`,..., `AxisFontFamily`,..., `AxisTextBackground`,...	`Default`	*As defined in Tables 8.5–8.7.*

Axis label options. The axis label options are almost identical to those described for panel edges in the discussion of `FigurePanel` (Sec. 10.1), as summarized in Table 10.6, except that the prefix *XY*`Frame` is replaced by `Axis`. Thus, the label is specified via the `AxisLabel` option, its positioning is controlled via the `AxisPosition` option, *etc.*

Tick options. The tick options are almost identical to those described for panel edges in the discussion of `FigurePanel` (Sec. 10.1), as summarized in Table 10.7, just not preceded by an *XY*. Thus, the ticks

[13]The similarity between `FigAxis` and the edges of a `FigurePanel` is more than just one of syntax. The four edges of a panel are actually drawn as four separate axes, each one generated by the same underlying code as `FigAxis`.

are specified via the `Ticks` option, and their formatting is controlled by options `ShowTickLabels`, `TickFontSizeFactor`, `TickTextColor`, *etc.*

Table 10.20 Arrowhead extension options for `FigAxis`.

Option	*Default*	
`TailExtension`	`None`	Distance by which to extend axis at tail.
`HeadExtension`	`10`	Distance by which to extend axis at head.

Axis-specific options. A standalone axis may (and by default does) have an arrowhead on it, at the positive end (head), and can have one at the negative end (tail) as well. The options controlling the arrowheads are the same as for a `FigLine`, discussed in Sec. 11.1 and summarized in Table 11.2. By default, `ShowHead->True` for a `FigAxis`. Also, by default, a `FigAxis` is extended a bit beyond its given *range*, to allow room for the arrowhead. The options `HeadExtension` and `TailExtension`, summarized in Table 10.20, control this extension.

Anchors. The anchors which may be generated from a `FigAxis` object include all the anchors — `Left`, `Right`, `Head`, `Tail`, *etc.* — defined for `FigLine` (Sec. 11.1), as summarized in Table 11.3. For completeness, we note that an `"Axis"` anchor is also defined. This anchor is meant primarily for internal use in positioning the axis label, which is constructed as the corresponding attached label.[14]

Attached labels. The only attached label for a `FigAxis` is the axis label, described above.

10.5 `WithOrigin`

Table 10.21 Command affecting the current coordinate origin.

`WithOrigin[`*p*`,`*body*`]`	Temporarily shifts the coordinate system used while evaluating *body*, to use the given origin.

`WithOrigin`. `WithOrigin[`*p*`,`*body*`]` shifts the origin of the current panel's coordinate system to the point *p*, while evaluating *body*. Thus, in particular, `WithOrigin[{`*x*`,`*y*`},`*body*`]` effectively shifts everything drawn in *body* by a displacement {*x*,*y*}. For convenience, a shorthand form is defined for a simple horizontal shift — `WithOrigin[`*x*`,`*body*`]` is equivalent to {*x*`,0`}. Note that `WithOrigin` may be nested, *i.e.*, used recursively. Recall that the *body* argument to `FigurePanel` is *required* to by enclosed in braces as {*body*} (see discussion in Sec. 10.1). While this is not strictly required for the *body* argument to `WithOrigin`, doing so is usually recommended for readability.

10.6 `FigureGroup`

Table 10.22 Environment to group figure objects and control their rendering.

`FigureGroup[{`*body*`}]`	Evaluates {*body*}, flattening and optionally clipping or rasterizing the contents.

[14]The anchor name `"Axis"` is chosen so that, under the naming conventions we have adopted for attached labels (Sec. 8.2), the corresponding option name is `AxisLabel`.

`FigureGroup`. `FigureGroup[{`*body*`}]` provides selective control over rendering — specifically, clipping and rasterization — of groups of objects within a panel. All the objects generated in {*body*} are sorted into their appropriate order (from background to foreground) according to their `Layer` option. `FigureGroup` then clips and/or rasterizes the objects according the the options `Clip`, `Rasterize`, and `ImageResolution`, with the same meanings and defaults as shown for `FigurePanel` in Table 10.9. The result appears flattened into a single layer in the current panel, which by default is layer 1 but which may be controlled through the `Layer` option to `FigureGroup`. Like `ScopeOptions` (Sec. 9.4), `FigureGroup` also localizes any changes to options for figure objects, so that they do not affect the defaults outside of {*body*}. [**UNDER CONSTRUCTION:** `FigureGroup` is still experimental and is subject to change.]

11 Basic drawing shapes

Table 11.1 Figure objects for basic drawing shapes.

`FigLine[`*curve*`]`	Generates a "line" or, more generally, curve.
`FigPolygon[`*curve*`]`	Generates a closed curve or polygon.
`FigRectangle[`*p*`]`	Generates a square or rectangle.
`FigRectangle[`p_1`,`p_2`]`	
`FigRectangle[`*region*`]`	
`FigCircle[`*p*`]`	Generates a circle/ellipse or arc thereof.
`FigCircle[`*region*`]`	
`FigPoint[`*p*`]`	Generates a geometric point.
`FigBSpline[`*curve*`]`	Generates a spline curve from the given control points.

In this section, we consider figure objects which may be used to draw basic geometric shapes, summarized in Table 11.1. These objects are modeled upon the Mathematica graphics "primitives" (lines, polygons, rectangles, circles, splines, and points). However, these are the SciDraw object-oriented, option-aware, style-aware, easily-labelable, anchorable-to versions, capable of taking their position arguments as either coordinates or as anchors (or perhaps even grabbing a sequence of curve points from plotting output) as described in Sec. 7 and of leaping tall buildings in a single bound. There are many further capabilities thrown in for practical convenience in generating scientific diagrams as well.

11.1 `FigLine`

Description. A `FigLine` is used to draw a line or curve connecting a series of points $\{p_1, p_2, \ldots, p_n\}$.[1] It is therefore a generalization of the Mathematica `Line` primitive (see **ref/Line**). The object consists of an outline only. Since arrowheads can be drawn at either end of the curve, as described below, `FigLine` also suffices for drawing simple arrows.

Arguments. In `FigLine[`*curve*`]` (Table 11.1), the points $\{p_1, p_2, \ldots, p_n\}$ are determined by the argument *curve*, which may be specified in any of the ways described in Sec. 7.2. Since one possibility is to take the *curve* directly from the output of a Mathematica `Plot` command, `FigLine` provides a convenient means for restyling (and labeling) Mathematica plots.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2). The interpretation of the *curve* is controlled by the options summarized in Table 7.8.

[1] A curve built as a series of line segments is more properly called a *polyline* in computer graphics terminology.

Table 11.2 Options for arrowheads on `FigLine` and other line-like objects.

Option	*Default*	
`ShowTail`	`False`	Whether or not to draw arrowhead at tail.
`TailLength`	`6`	Length of arrowhead at tail, in printer's points.
`TailLip`	`3`	Half-width of arrowhead at tail, in printer's points.
`ShowHead`	`False`	Whether or not to draw arrowhead at head.
`HeadLength`	`6`	Length of arrowhead at head, in printer's points.
`HeadLip`	`3`	Half-width of arrowhead at head, in printer's points.

Arrowhead options. Simple arrowheads can be drawn at either end of the curve, controlled through the options summarized in Table 11.2. The first three options in Table 11.2 control the arrowhead at the "tail" (starting point) of the curve, and the last three control the arrowhead at the "head" (ending point) of the curve. Let us consider the latter for illustration. Setting `ShowHead->True` causes the arrowhead to be drawn. The `HeadLength` gives the arrowhead's length in printer's points, measured along the direction of the curve (*i.e.*, the "shaft" of the arrow). A negative length gives a "reversed" arrowhead, extending beyond the end of the curve. The `HeadLip` gives the half-width, or distance of each side of the arrowhead out from the curve, in printer's points. It may be given as a pair of numbers $\{l_L, l_R\}$ for the left and right sides. If one or the other of these values is given as `0`, the result is a one-sided or "barbed" arrow head.

Table 11.3 Named anchors for `FigLine` objects.

Name	*Argument*	
`Center`	*u*	*Position:* A fractional distance *u* along the total length of the curve (from `0` at the tail to `1` at the head). *Text offset:* Centered on curve — {`0,0`}. *Orientation:* Tangent to curve.
	{`Displace–AlongTail,` *dist*} {`Displace–AlongHead,` *dist*}	Similarly, but a distance *dist*, in printer's points, from the tail or head of the curve, along the tangent to the curve (as in Table 7.7).
	{`Horizontal,`*x*}, {`Vertical,`*y*}	Similarly, but at a position obtained by seeking a given *x* or *y* coordinate along curve by numerical rootfinding. (For complicated curves, this may be unreliable.)
	{*s*,*u*}, *etc.*	Any of the argument forms listed above, applied to the *s*th segment of the curve
`Left`	(similarly)	Similarly, but with text offset {`0,-1`}, *i.e.*, text along the "left" side of the curve.
`Right`	(similarly)	Similarly, but with text offset {`0,+1`}, *i.e.*, text along the "right" side of the curve.
`Tail`	—	*Position:* At the tail point of the curve. *Text offset:* Past the end of the curve — {`+1,0`}. *Orientation:* Tangent to curve.
`Head`	—	*Position:* At the head point of the curve. *Text offset:* Past the end of the curve — {`-1,0`}. *Orientation:* Tangent to curve.
`Point`	*n*	*Position:* At the *n*th point. *Text offset:* Centered on the point — {`0,0`}. *Orientation:* Horizontal.

Anchors. The anchors which can be generated from a `FigLine` object (using `GetAnchor` as described in Sec. 7.1.4) are summarized in Table 11.3. The concept of "left" and "right" sides of a curve is introduced in Sec. **??** of the user's guide. Any of the argument forms can instead be applied to just the *s*th segment of the curve, by giving it as {*s*`,`...}, *e.g.*, {`1,0.5`} for half-way along the first segment. Negative values of *s* count back from the last segment.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Left`, `Center`, `Right`, `Tail`, and `Head` anchor positions. The positioning parameters, as summarized in Table 11.3, may be specified through the *X*`LabelPosition` options.

11.2 `FigPolygon`

Description. A `FigPolygon` may be used to draw a closed curve or, equivalently, a polygon. It is therefore a generalization of the Mathematica `Polygon` primitive (see **ref/Polygon**). The object consists of an outline and fill (either or both of which may be shown or hidden, as selected using the usual options of Sec. 8). Hence, the polygon may be filled or open.

Arguments. The points are determined by the argument *curve* (see Table 11.1), which may be specified in any of the ways described in Sec. 7.2.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1), outline options (Table 8.2), and fill options (Table 8.3). The interpretation of the *curve* is controlled by the options described in Sec. 7.2 (Table 7.8). `FigPolygon` also accepts `VertexColors`, `FillTexture`, and `VertexTextureCoordinates` options, which define color gradient fills and textured fills, with exactly the same meanings as they have for the Mathematica `Polygon` primitive (see **ref/VertexColors**, *etc.*).

Anchors. The anchors which can be generated from a `FigPolygon` object are the same as for `FigLine`, as described in as summarized in Table 11.3. (For purposes of generating anchors, the curve is completed as a closed curve, *i.e.*, if the argument *curve* is given as the points $\{p_1, p_2, \ldots, p_n\}$, then the full closed curve is $\{p_1, p_2, \ldots, p_n, p_1\}$.)

Attached labels. Attached labels are available for the `Left`, `Center`, and `Right` anchor positions. These represent labels on the perimeter curve of the polygon, as defined for `FigLine` in Table 11.3. The only difference between the left, center, and right labels is the text offset (whether the text will fall inside, on, or outside the perimeter). They should not be mistaken for labels at the left, center, and right of the polygon *per se*, considered as a planar area, which would in fact be hard to define in general. The positioning parameters, as summarized in Table 11.3, may be specified through the *X*`LabelPosition` options.

11.3 `FigRectangle` and `FigCircle`

Description. The `FigRectangle` and `FigCircle` objects have many features in common, and are therefore discussed together in this section.[2] A `FigRectangle` may be used to draw a square or, more generally, a rectangle. Similarly, a `FigCircle` may be used to draw a circle or, more generally, an ellipse. A `FigCircle` may alternatively be used to draw a circular arc or sector, covering a specified range of angles, in which case arrowheads can also be drawn at either end of the arc.

Note that `FigRectangle` is a generalization of the Mathematica `Rectangle` primitive (see **ref/Rectangle**), and `FigCircle` is a generalization of the Mathematica `Circle` and `Disk` primitives (see **ref/Circle** and **ref/Disk**).

Either of these objects consists of an outline and fill (either or both of which may be shown or hidden, as selected using the usual options of Sec. 8). Hence, the shape may be filled or open. Also, either of these shapes may be rotated.

Arguments. The geometry for a `FigRectangle` may be specified in a few different ways (see Table 11.1).

The most basic way of specifying the geometry of either a `FigRectangle` or a `FigCircle` is the same — give its center point and its size. The form `FigRectangle[`*p*`]` or `FigCircle[`*p*`]` is used to give the center point *p*. (The point *p* may be specified in any of the ways described in Sec. 7.1.) Then the `Radius` option is used to give the dimensions. Actually, this mechanism is very flexible (*e.g.*, the point *p* can be at any relative position over the face of the shape, not just the center), as discussed below under *Rectangle or circle geometry*.

If the `FigRectangle` is being used to draw a box around a region $\{\{x_1, x_2\}, \{y_1, y_2\}\}$, then the form `FigRectangle[`*region*`]` is most appropriate. (The *region* may, more generally, be specified in any of the

[2]After describing rectangles in a section on circles, we will then attempt the feat of fitting a square peg into a round hole. Or perhaps squaring the circle.

ways described in Sec. 7.4.) Analogously, `FigCircle[`*region*`]` draws the circle or ellipse inscribed in this region.

For consistency with the usual Mathematica `Rectangle` primitive, a rectangle may be specified by giving two diametrically opposite corner points, as `FigRectangle[`p_1`,`p_2`]`. (Each of these points may be specified in any of the ways described in Sec. 7.1.) However, giving the corner points is rarely the most convenient approach from the user's point of view.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1), outline options (Table 8.2), and fill options (Table 8.3).

Table 11.4 Options for controlling the geometry of `FigRectangle` and `FigCircle`.

Option	*Default*	
`Radius`	`1`	Half-width (and half-height) of a square/rectangle, or radius (or semi-axis lengths) of a circle/ellipse, specified as detailed in Table 11.5
`AnchorOffset`	`Center`	Relative position of anchor point on shape.
`PivotOffset`	`Automatic`	Relative position of pivot point on shape.
`Rotate`	`None`	Rotation angle around pivot point.

Table 11.5 Ways of specifying the "radii" for a `FigRectangle` and `FigCircle`.

r or $\{r_x, r_y\}$	Radius (or horizontal and vertical radii) in the current panel's coordinate system.
`Scaled[`r`]` or `Scaled[`$\{r_x, r_y\}$`]`	Radius (or horizontal and vertical radii) as a fraction of the size of the panel.
`Canvas[`r`]` or `Canvas[`$\{r_x, r_y\}$`]`	Radius (or horizontal and vertical radii) in printer's points, *i.e.*, as measured on the canvas.
`Horizontal[`r`]` or `Horizontal[`$\{r_x, r_y\}$`]`	Radius (or horizontal and vertical radii) given in the scale set by the x axis.
`Vertical[`r`]` or `Vertical[`$\{r_x, r_y\}$`]`	Radius (or horizontal and vertical radii) given in the scale set by the y axis.

Rectangle or circle/ellipse geometry. The options for controlling the geometry of a `FigRectangle` or `FigCircle` are summarized in Table 11.4.

If the form `FigRectangle[`p`]` or `FigCircle[`p`]` is used, p is by default taken to be the center of the shape. However, p may represent *any* point on the shape, as determined by the option `AnchorOffset`. The value of the option `AnchorOffset` should be the relative position of the point on the face of the shape and may be specified in any of the ways described in Sec. 7.3, for instance, `Bottom` for the bottom edge of the rectangle or circle. For an ellipse, the relative coordinate is with respect to a rectangle which circumscribes the ellipse — for instance, `BottomLeft` would be the bottom left corner of this rectangle.

The width and height of the shape are then specified through the `Radius` option, which can take on values of the various forms summarized in Table 11.5. To understand the basic idea, let us focus on the basic forms r and $\{r_x, r_y\}$. The name `Radius`, of course, comes from the case of a circle, but the basic idea is equally applicable to either a rectangle or an ellipse.[3] For a `FigCircle`, a radius `Radius->1`, for instance, gives a circle of radius 1, while `Radius->{2,1}` gives an ellipse of semi-major axis 2 (horizontally) and semi-minor axis 1 (vertically). Analogously, for a `FigRectangle`, a "radius" `Radius->1`

[3]The syntax follows that of the radius argument to the Mathematica `Circle` or `Disk` primitive.

gives a square of half-width 1 (*i.e.*, 2×2), while `Radius->{2,1}` gives a rectange of half-width 2 and half-height 1 (*i.e.*, 4×2).

More generally, the first three forms for the `Radius` option in Table 11.5 are analogous to the three ways of specifying a point (Table 7.1). The two "new" forms, `Horizontal[...]` and `Vertical[...]`, address the aspect ratio issues illustrated in Sec. **??**.

For *any* of the forms of `FigRectangle` or `FigCircle` given in Table 11.1 — let us take a rectangle for illustration, but the same comments apply to an ellipse — the rectangle's position and dimensions are initially given assuming the rectangle is aligned with the coordinate axes. From this starting point, the rectangle may then be rotated. The rotation angle is specified through the option `Rotate`. This angle is in radians, measured counterclockwise from horizontal, as usual for plane polar coordinates. The default value is `None`, equivalent to `0`. If the particular form `FigRectangle[`p`]` is used, and if p is actually an anchor containing orientation information, rather than merely a point, the special value `Rotate->Automatic` means that the rotation angle should be taken from the orientation angle of the anchor p. This possibility is very convenient if the rectangle's position is "anchored" to another object and if it is also desired that the rectangle should also be rotated to align with that object. The rotation need not be around the center of the rectangle but rather may be around any arbitrarily-chosen pivot point, say, one of the corners of the rectangle, as determined by the option `PivotOffset`. The value of the option `PivotOffset` should be the relative position of the pivot point on the face of the rectangle and may be specified as described in Sec. 7.3. The default value `PivotOffset->Automatic` means that the value is taken from the option `AnchorOffset`, *i.e.*, rotation is about the same point p as was used to position the rectangle.

Table 11.6 Option for controlling the rendering of the corners of a `FigRectangle`.

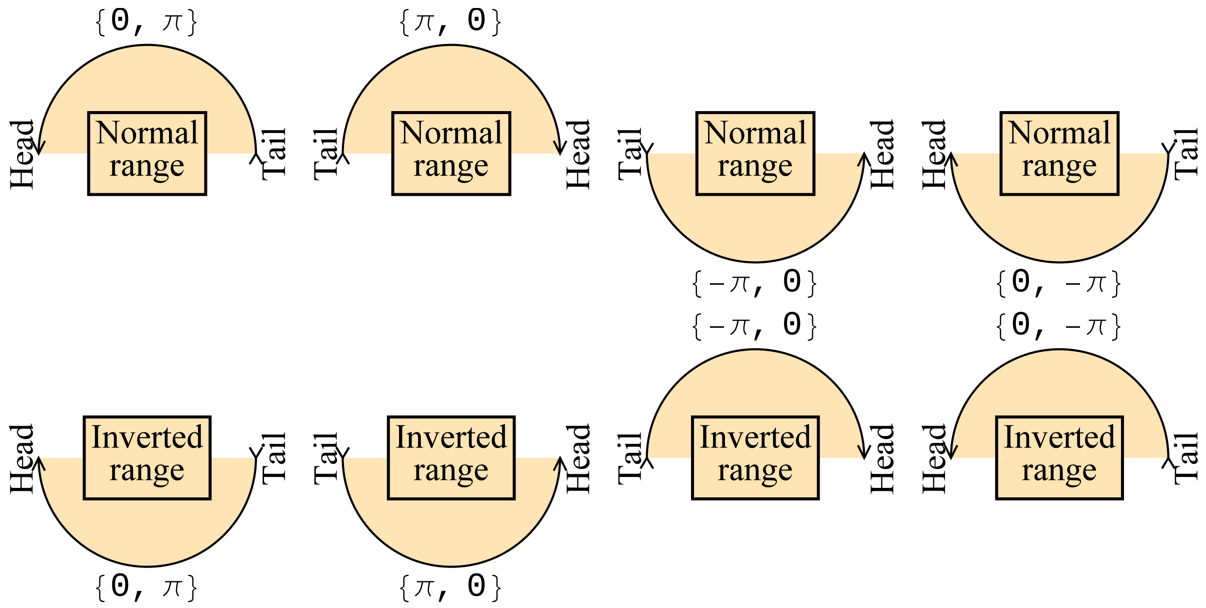
Option	*Default*	
`RoundingRadius`	`None`	Radius of the circle to use in rendering rounded corners.

`RoundingRadius` (`FigRectangle` only). The corners of a `FigRectangle` may be rounded, as specified with the `RoundingRadius` option (Table 11.6). For the general concept of a rounding radius, see **ref/RoundingRadius**. The radius may either be a single number r, or the horizontal and vertical directions may be treated asymmetrically as $\{r_x, r_y\}$. SciDraw allows the option `RoundingRadius` to be specified in any of the ways described above for the option `Radius` of Table 11.4. The default value `None` means no rounding, equivalent to a radius of `0`.

Table 11.7 Options for controlling the rendering of a segment or arc (`FigCircle` only).

Option	*Default*	
`AngleRange`	`None`	Angle range $\{\theta_1, \theta_2\}$ for segment or arc, in radians, or $\{p_1, p_2\}$ to subtend rays to given points, or `None` for full circle/ellipse
`InvertAngleRange`	`False`	Choose complementary subcircle
`CurveClosed`	`False`	Whether the outline of a partial circle $\{\theta_1,\theta_2\}$ should be drawn open (as an arc) or closed (as a sector).

Angle range selection (`FigCircle` only). Options controlling the drawing of a segment or arc are summarized in Table 11.7. A circular arc subtending the angles $[\theta_1, \theta_2]$ is selected with `AngleRange->`$\{\theta_1, \theta_2\}$. Sometimes it is more convenient to specify instead that the arc should subtend the rays $\overrightarrow{pp_1}$ and $\overrightarrow{pp_2}$, with `AngleRange->`$\{p_1, p_2\}$, especially if the arc is being used to label the angle between those rays. Here, the points p_1 and p_2 may be specified in any of the ways described in Sec. 7.1 — in particular, they may be anchors.



The meaning of the arc angles θ_1 and θ_2 is a bit tricky in Mathematica (and not fully documented[4]). The arc connecting two angles is initially ambiguous — you can imagine going around the circle from θ_1 to θ_2 the "counterclockwise" way or the "clockwise" way (or, from a different perspective, the "short" way or the "long" way). Mathematica apparently resolves this ambiguity (and SciDraw therefore follows suit) by first sorting the two angles in increasing numerical order and then going *counterclockwise* from the numerically-smaller angle to the numerically-large angle. Thus, *e.g.*, either `{0,Pi}` or `{Pi,0}` generates the upper half circle, while either `{0,-Pi}` or `{-Pi,0}` generates the lower half circle. The *order* of the angles as arguments does not affect the arc drawn. On the other hand, the order of the angles as arguments *does* have meaning to SciDraw, when it comes to drawing arrowheads or choosing a tangent direction (*e.g.*, when anchoring text labels to the arc) — SciDraw thinks of the arc as a *directed* curve *from* θ_1 (the tail) *to* θ_2 (the head).

Notice that the choice of subcircle for the arc — in the above example, whether the arc was drawn around the lower half circle or the upper half circle — is generally inverted by adding 2π to the numerically smaller angle. As a matter of convenience, `FigCircle` will do this for you, if you specify `InvertAngleRange->True`.

`CurveClosed` (`FigCircle` only). In the case of an sector or arc, the question arises as to how the outline should be drawn. With `CurveClosed->False` (the default), the outline is drawn as an arc, running along the circumference of the circle but *not* closing along the radii.[5] With `CurveClosed->True`, the outline is drawn around the whole a sector (or pie slice), running along the circumference of the circle and closing along the radii.[6]

Arrowhead options (`FigCircle` only). Simple arrowheads can be drawn at either end of a circular/elliptical arc, controlled through the usual arrowhead options summarized in Table 11.2.

[4]See **ref/Circle** and **ref/Disk**.

[5]This is the behavior of the Mathematica `Circle` primitive, when it is given angle arguments $\{\theta_1, \theta_2\}$.

[6]This is the behavior of the Mathematica `Disk` primitive, when it is given angle arguments $\{\theta_1, \theta_2\}$.

Table 11.8 Named anchors for `FigRectangle` objects.

Name	*Argument*	
`Center`	—	*Position:* At the center of the shape. *Text offset:* Centered — {`0,0`}. *Orientation:* Horizontal.
`Left`	y_r (optional)	*Position:* On the left edge of the rectangle, at relative position y_r, from `-1` to `+1`, or at the midpoint if y_r is omitted. *Text offset:* Outside the rectangle — {`+1,0`}. *Orientation:* Horizontal.
`Right`	y_r (optional)	Similarly, on the right edge of the rectangle.
`Bottom`	x_r (optional)	Similarly, on the bottom edge of the rectangle.
`Top`	x_r (optional)	Similarly, on the top edge of the rectangle.
`Offset`	$\{x_r,y_r\}$	*Position:* At a relative position $\{x_r, y_r\}$ on the face of the rectangle, which may more generally be specified as described in Sec. 7.3. *Text offset:* Centered — {`0,0`}. *Orientation:* Horizontal.

Table 11.9 Named anchors for `FigCircle` objects.

Name	*Argument*	
`Center`	—	*Position:* At the center of the circle/ellipse. *Text offset:* Centered — {`0,0`}. *Orientation:* Horizontal.
`Left`	—	*Position:* At the left side of the circle/ellipse. *Text offset:* Outside the circle — {`+1,0`}. *Orientation:* Horizontal.
`Right`	—	Similarly, at the right side of the circle/ellipse.
`Bottom`	—	Similarly, at the bottom of the circle/ellipse.
`Top`	—	Similarly, at the top of the circle/ellipse.
`Tail`	—	*Position:* At the tail point θ_1 of the circular arc. *Text offset:* Past the end of the circular arc — {`+1,0`}. *Orientation:* Tangent to circular arc (in the $-\theta$ sense).
`Head`	—	*Position:* At the head point θ_2 of the circular arc. *Text offset:* Past the end of the circular arc — {`-1,0`}. *Orientation:* Tangent to circular arc (in the $-\theta$ sense).
`Offset`	$\{x_r,y_r\}$	*Position:* At a relative position $\{x_r,y_r\}$ on the face of the circle/ellipse, which may more generally be specified as described in Sec. 7.3. *Text offset:* Centered — {`0,0`}. *Orientation:* Horizontal.
`Tangent`	*u* or {`"Angle"`,θ} (optional)	*Position:* A fractional distance *u* (from `0` to `1`) along the circumference or circular arc, or at polar angle *theta*, or at the midpoint if not specified. *Text offset:* Outside the circle — {`0,-1`} for a clockwise arc, or {`0,+1`} for a clockwise arc. *Orientation:* Tangent to circumference or circular arc (in the $-\theta$ sense).
`Normal`	(similarly)	Similarly, but with orientation along the outward normal of the circle or ellipse (this is equivalent to the $+r$ direction for a true circle, but it is more generally defined as the true normal to the curve on the canvas, orthogonal to the tangent direction). The text is still outside the circle, but this is now obtained with offset {`-1,0`} for a clockwise arc, or {`+1,0`} for a clockwise arc..
`HeadRadius` `TailRadius`	*u* (optional)	*Position:* On the bounding radial line segments delimiting the sector/arc, at fractional distance *u*. *Text offset:* Chosen to put the labels outside the sector, if the angles are given such that the arc runs in the counterclockwise sense from tail to head — {`-1,0`} for `HeadRadius` or {`+1,0`} for `TailRadius`. *Orientation:* Outward along the radius.

Anchors. The anchors which can be generated from a `FigRectangle` are summarized in Table 11.8. Those which can be generated for a `FigCircle` are summarized in Table 11.9. [**UNDER CONSTRUC-**

TION: The `Tangent` and `Normal` anchors are subject to change and may be replaced by a common "arc" anchor, with different text orientation options.]

Attached labels. For a `FigRectangle`, attached labels are available for the `Left`, `Right`, `Bottom`, `Top`, and `Center` anchor positions.

For a `FigCircle`, attached labels are furthermore available for the `Tangent`, `Normal`, `Head`, `Tail`, `HeadRadius` and `TailRadius` anchor positions. For a `FigCircle`, it is worth keeping in mind that: (1) the `Left`, `Right`, `Bottom`, and `Top` are based on the circumscribed rectangle, for the *full* circle or ellipse, even if only an arc or sector is actually drawn, so they may not fall where you might naively expect them to in this case, (2) the `HeadRadius` and `TailRadius` labels label the bounding radii (sides) of a sector, and (3) the `Tangent` and `Normal` labels are both for positioning labels on the arc, but differ in the default orientation (and offset) for the text.

The positioning parameters, as summarized in Table 11.8, may be specified through the *X*`LabelPosition` options.

11.4 `FigPoint`

Description. A `FigPoint` may be used to draw a simple geometric point. It is therefore a generalization of the Mathematica `Point` primitive (see **ref/Point**). `FigPoint` is essentially redundant to `FigCircle`, since a point looks identical to a filled circle of the same radius. However, it is provided for consistency with Mathematica's full set of graphics primitives.

Arguments. The geometry is specified by giving the center and *diameter* of the point (see Table 11.1). In `FigPoint[`p`]`, p determines the location of the center, and the diameter is given through the option `PointSize` (Table 8.4). (The point p may be specified in any of the ways described in Sec. 7.1.)

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and point options (Table 8.4).

Anchors. The anchors which can be generated from a `FigPoint` are the same as those for a `FigCircle`, as summarized in Table 11.9.

Attached labels. Attached labels are available for the `Left`, `Right`, `Bottom`, `Top`, and `Center` anchor positions. The positioning parameters, as summarized in Table 11.9, may be specified through the *X*`LabelPosition` options.

11.5 Splines

Description. A `FigBSpline` is used to draw a spline curve with control points $\{p_1, p_2, \ldots, p_n\}$. It is therefore a generalization of the Mathematica `BSplineCurve` primitive (see **ref/BSplineCurve**).[7] The object consists of an outline only. Arrowheads can be drawn at either end of the curve. The syntax of `FigBSpline` is identical to that of `FigLine` (Sec. 11.1). Only the rendering of the curve is different, namely, as a spline with given control points, instead of a simple set of line segments connecting those points.

[7]SciDraw also defines a `FigBezier` object, as a generalization of the Mathematica `BezierCurve` primitive (see **ref/BezierCurve**). However, its functionality is presently limited to simple (*i.e.*, noncompound) Bezier curves of cubic order. Therefore, it is not listed in Table 11.1. This is due to an undocumented limitation of the Mathematica `BezierFunction` function (as of Mathematica 8-10, issue [TS 22944]), which might be rectified in future releases of Mathematica.

Arguments. The points $\{p_1, p_2, \ldots, p_n\}$ are determined by the argument *curve* (see Table 11.1), which may be specified in any of the ways described in Sec. 7.2.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2). The interpretation of the *curve* is controlled by the options summarized in Table 7.8.

Arrowhead options. Simple arrowheads can be drawn at either end of the curve, controlled through the options summarized in Table 11.2.

Spline control options. The spline characteristics are determined by the options `SplineClosed`, `SplineDegree`, `SplineKnots`, and `SplineWeights`, with exactly the same meanings as they have for the Mathematica `BSplineCurve` primitive (see **ref/SplineClosed**, *etc.*, as well as the general discussions in **ref/BSplineCurve** and **ref/BSplineFunction**).

Anchors. The anchors which can be generated from a `FigBSpline` are the same as those for a `FigLine`, as summarized in Table 11.3. However, the parameter u is no longer guaranteed to be strictly proportional to the arc length along the curve.

Attached labels. Attached labels are available for the `Left`, `Center`, `Right`, `Tail`, and `Head` anchor positions. The positioning parameters, as summarized in Table 11.3, may be specified through the *X*`LabelPosition` options.

12 Arrows

Table 12.1 Figure object for arrows.

`FigArrow[`*curve*`]`	Generates an arrow.

Description. A `FigArrow` is used to draw any of a variety of arrow types. Depending on the arrow type, a `FigArrow` can consist of both an outline and a fill. In its simplest form, `FigArrow` produces a curve with an arrowhead at the end — this is identical to what can already be drawn with `FigLine`. However, SciDraw also provides "block" arrows, double-shafted arrows, and "squiggle" arrows, as described below. Furthermore, `FigArrow` is designed to be *extensible* by more advanced users (*i.e.*, with a some programming effort), in the sense that more exotic arrow shapes and fills can be defined through customization functions.

Arguments. `FigArrow[`*curve*`]` (Table 12.1) takes a curve as its argument. This curve may be specified in any of the ways described in Sec. 7.2.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1), outline options (Table 8.2), and fill options (Table 8.3). The interpretation of the *curve* is controlled by the options summarized in Table 7.8.

Table 12.2 Options for arrow geometry.

Option	*Default*	
`ArrowType`	`"Line"`	Arrow type (outline and fill geometry).
`Width`	`5`	Arrow shaft width, in printer's points.
`ArrowJoinForm`	`"Miter"`	Join style (mitering or beveling) to be used for block (or other wide) arrows with multiple segments.
`ShowTail`	`False`	Whether or not to draw arrowhead at tail.
`TailLength`	`6`	Length of arrowhead at tail, in printer's points.
`TailLip`	`3`	Extension of arrowhead beyond shaft, on each side, at tail, in printer's points.
`ShowHead`	`True`	Whether or not to draw arrowhead at head.
`HeadLength`	`6`	Length of arrowhead at head, in printer's points.
`HeadLip`	`3`	Extension of arrowhead beyond shaft, on each side, at head, in printer's points.
`TailFlush`	`False`	When no arrowhead is drawn at tail, whether or not shaft end should be flush to the endpoint anchor orientation.
`HeadFlush`	`False`	When no arrowhead is drawn at head, whether or not shaft end should be flush to the endpoint anchor orientation.

`ArrowType`. The type, or shape, of the arrow is specified by the option `ArrowType`. This option chooses the general nature of the outline and fill geometry. The possible values are `"Line"` for a simple line shaft and arrowhead, `"Block"` for a solid block shape, `"DoubleLine"` for a double shafted arrow (optionally with fill between the shafts), or `"Squiggle"` for a sinusoidal arrow as traditionally used to schematically represent a photon. (Other arrow types may be defined, with some programming effort, as described under `DefineArrowType` below.)

`Width`. The `Width` option defines the width of the arrow shaft, in printer's points. This option is irrelevant for the `Line` arrow type. (The width refers to the distance between the lines on the left and right sides of the arrow shaft and is not to be confused with the thickness of the line, which is still controlled by the option `LineThickness`.)

`ArrowJoinForm`. Since the arrow shaft may now have finite width (for the `"Block"` or `"DoubleLine"` arrow types), and arrows may also consist of more than one segment (if three or more points are given for *curve*), the choice arises as to whether to miter or bevel the joints between segments. This choice affects the "exterior" angle at each joint. With the default `ArrowJoinForm->"Miter"`, a sharp point is drawn. This is usually the natural choice but may lead to problems for hairpin joints (reflex angles near 360°), where the tip on the joint will have to extend outward by many times the usual shaft width. Instead, with `ArrowJoinForm->"Bevel"`, the joint will be truncated to the width of the arrow shaft.

Arrowhead options. The next several options in Table 12.2 (`ShowTail`, *etc.*) simply generalize the `FigLine` arrowhead options of Table 11.2, with slightly modified meanings for wide arrows. Note that the default for `ShowHead` is now `True`, naturally enough for an object which is meant to represent an arrow. Also, now that the arrow shaft can have a nonzero width, as for the `Block` arrow type, `HeadLip` (or `TailLip`) really does represent a "lip" or extension past the width of the shaft, rather than simply the half-width of the arrowhead.

`TailFlush` & `HeadFlush`. When an arrow is drawn from one object to another, it may be desirable for the tail of the arrow to be drawn flush against the object (*i.e.*, tangent to the object's boundary). This is requested with `TailFlush->True`. For instance, it is commonly expected that the tail of a "transition" arrow in a level energy diagram will be flush against the starting level (*i.e.*, horizontal) even when the arrow shaft itself does not run vertically (see also the discussion of these options for `Trans` in Sec. 15.5). Information on the tangent direction is available to `FigArrow` if the starting point p_1 given to `FigArrow` is an *anchor*, rather than simply a point. Some (but not all) of the predefined anchors which can be generated from objects store the polar angle of the tangent to the object as the anchor orientation angle. If necessary `Anchor` (Sec. 7.1.2) may be used to modify the orientation angle in the anchor given to `FigArrow`. The option `TailFlush` only applies if `ShowTail->False`, that is, if the tail is drawn as a bare shaft end, with no arrowhead. A similar option `HeadFlush` is provided for the head of the arrow and only applies if `ShowHead->False` (this might be relevant if the `FigArrow` is used simply to draw a connecting shaft between two objects, with no actual arrowhead on either end).

Table 12.3 Options controlling the appearance of squiggle arrows.

Option	*Default*	
`SquiggleWavelength`	`10`	Wavelength of sinusoid, in printer's points.
`SquiggleSide`	`Right`	Side of arrow on which first crest of sinusoid should occur.
`SquiggleBuffer`	`2`	Minimum length of straight segment of shaft (in addition to the arrowhead, if any) before sinusoid begins, in printer's points.
`PlotPoints`	`32`	Number of plotting points, per wavelength, to use in rendering sinusoid.

Squiggle arrow options. Further options for fine-tuning the appearance of sinusoidal `"Squiggle"` arrows are listed in Table 12.3. The wavelength of a `"Squiggle"` arrow is controlled with the option `SquiggleWavelength`. The sinusoidal part of a squiggle arrow always contains an integer number of "humps" or half wavelengths. A short length of straight arrow shaft appears at either end of the sinusoid, making up the extra length needed for the arrow, before any arrowhead. The minimum length of these segments is controlled by the option `SquiggleBuffer`.

Anchors. The anchors for `FigArrow` are as defined for `FigLine` in Table 11.3. However, now the `Left`, `Center`, and `Right` anchors really are distinct. Not only are the text offsets different for these anchors, as in Table 11.3, but, now that the shaft may have finite width, the `Left` anchor lies on the left edge of the shaft, the `Center` anchor on the centerline of the shaft, and the `Right` anchor on the right edge of the shaft.

Attached labels. Attached labels are available for the `Left`, `Center`, `Right`, `Tail`, and `Head` anchor positions.

Table 12.4 Function for defining new arrow types.

`DefineArrowType[`*name*`,`*function*`]`	Defines a new arrow type.

`DefineArrowType`. `DefineArrowType[`*name*`,`*function*`]` defines *name* to represent a new arrow shape, for use with the option `ArrowType`. The name should typically either be a string or else a brace-delimited list consisting of a string plus one or more parameter patterns (the concept is illustrated for `DefineDataSymbolShape` in Sec. 16.1.3). The *function* should be a pure function which accepts several arguments, including a list of canvas points for the curve, head and tail anchors, and the arrow width, and is responsible for generating the graphics for the arrow. This framework for defining new arrow types is primarily meant for relatively experienced Mathematica programmers and requires some work with the technical "internals" of SciDraw (in contrast to the data plot customization framework discussed in Sec. 16.1.3, which requires relatively little programming experience or effort). Examples may be found in the SciDraw source code file `FigArrow.nb`.

13 Annotations

Table 13.1 Additional figure objects, meant primarily for annotation of a figure.

`FigLabel[`*p*`,`*text*`]`
`FigLabel[`*object*`,`*name*`,`*text*`]`
`FigLabel[`*object*`,`*name*`,`*arg*`,`*text*`]`
`FigLabel[`*text*`]`
	Generates a label.

`FigBracket[`*side*`,`*coord*`,`*range*`]`
`FigBracket[`*sense*`,`$\{p_1, p_2\}$`]`
	Generates a bracket.

`FigRule[`*direction*`,`*coord*`,`*range*`]`
`FigRule[`*direction*`,`*coord*`,All]`
	Generates a horizontal or vertical rule line.

13.1 `FigLabel`

Description. A `FigLabel` is used to place a label at a specified point or anchor location. It is therefore a generalization of the Mathematica `Text` primitive (see **ref/Text**). The object consists of text (possibly with a background and frame). Optionally, a callout line may also be part of the label.

Arguments. In the usual form `FigLabel[`*p*`,`*text*`]` (Table 13.1), the given point — or anchor — *p* "anchors" the text, and the position of the label relative to this anchor is given through the usual options `TextOffset`, *etc.*, of Table 8.8. (The point *p* may be specified in any of the ways described in Sec. 7.1.) The form `FigLabel[`*object*`,`*name*`,`*text*`]` or `FigLabel[`*object*`,`*name*`,`*arg*`,`*text*`]` attaches the label to a previously-drawn object, and is essentially a shorthand for `FigLabel[GetAnchor[`*object*`,`*name*`],`*text*`]` or `FigLabel[GetAnchor[`*object*`,`*name*`,`*arg*`],`*text*`]`. The optional argument to the anchor may alternatively be given through the option `Position` (see below). The shortest form, `FigLabel[`*text*`]`, instead allows *p* to be given through the option `Point`.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the text and background/frame options (Tables 8.5–8.8). The appearance of the callout line, if present, is controlled by the outline options (Table 8.2).

Table 13.2 Options for controlling the position a `FigLabel`.

Option	*Default*	
`Point`	`{0,0}`	Point or anchor specification.
`Position`	`Automatic`	Additional positioning argument to use when generating the label anchor.
`Displacement`	`None`	Shift of the label relative to its given position.
`DisplaceedPoint`	`None`	Shift of the label to an entirely new position.

`Point`. If the form `FigLabel[`*text*`]` is used, the position of the label must be given through the option `Point`, shown in Table 13.2. This is particularly useful in that is allows the label position to be determined by the settings in a style (Sec. 9). For instance, suppose the label meant to serve as a plot title, and its style is selected with `Style->`*style*. Then setting the option `Point->Scaled[0.5,1]` for `FigLabel` in this *style* would position the plot label at the center top of the panel. This choice could later be changed uniformly — in one fell swoop for all plots making use of *style* — simply by changing the value of the option `Point` in *style*.

`Position`. In the form `FigLabel[`*object*`,`*name*`,`*text*`]`, the option `Position->`*arg* may be used as an alternative way to specify an argument for the anchor, in the sense of Sec. 7.1.2, and yields the same result as `FigLabel[`*object*`,`*name*`,`*arg*`,`*text*`]`. The option `Position` therefore serves the same purpose as *X*`LabelPosition` does for attached labels (Table 8.10) and is provided for consistency with that option. `Position->Automatic` (the default) gives no argument for the anchor.

Displaced location for text.. The option `Displacement` (Table 13.2) specifies a displacement by which the label should be shifted, relative to its given position. The displacement may be given in any of the forms described for the arguments to `DisplacePoint` in Sec. 7.1.5. Alternatively, the option `DisplacedPoint` specifies an entirely new point at which to put the text of the label. (If both `Displacement` and `DisplacedPoint` are specified, the point determined by `DisplacedPoint` takes precedence.) These options are essential to the drawing of callout lines, which extend from the displaced point to the originally-specified point.

Callout options. If the label is given a `Displacement` or a `DisplacedPoint`, then a connector or *callout* line can be drawn, connecting the displaced point to the originally-specified point.

Table 13.3 Options for controlling display of the callout line for `FigLabel`.

Option	*Default*	
`ShowLine`	`False`	Whether or not to display the callout line. *As defined in Table 8.2.*
`IntermediatePoints`	`None`	Points through which the callout line should pass along the way.

Just as a curve determining a `FigLine` or `FigArrow` may consist of more than two points, the callout line for a `FigLabel` may have "kinks", *i.e.*, it may run through additional intermediate points not on the line from the starting point to the ending point. A list of intermediate points is provided through the option `IntermediatePoints`. These may be specified in any of the forms allowed for points along a curve, as described in Sec. 7.2.1. In practice, it may be particularly useful to use `DisplaceTail`, `DisplaceHead`, and related specifications (Table 7.7).

Simple arrowheads can be drawn at either end of the callout line, controlled through the usual arrowhead options summarized in Table 11.2. By default, the head but not the tail is shown.

Anchors. The anchors which can be generated from a `FigLabel` object are identical to those for a `FigRectangle`, as summarized in Table 11.8. Furthermore, `Head` and `Tail` anchors are defined for the callout line, as defined in Table 11.3.

Attached labels. Not applicable.

13.2 `FigBracket`

Description. A `FigBracket` is used to draw a horizontal or vertical bracket.

Arguments. In `FigBracket[`*side*`,`*coord*`,`*range*`]` (Table 13.1), the *side* — `Bottom`, `Left`, `Top`, or `Right` — determines the overall orientation (horizontal or vertical) and configuration (where the labels are drawn relative to the bracket and which way the end caps face).

The *coord* specifies the coordinate at which the bracket should be placed (this as an x coordinate for a vertical rule, or a y coordinate value for a horizontal rule). Note that this coordinate may be given in any of the various ways described for "individual coordinates" in Sec. 7.1.1, *e.g.*, as `Scaled[`x_s`]`, or by giving an anchor from which the horizontal position should be taken.

The *range* specifies the coordinate range the bracket should cover (this is a y coordinate range for a vertical rule, or an x coordinate range for a horizontal rule). Instead of a *range*, a rectangular region may be given, in any of the various ways described in Sec. 7.4 — this is mainly meant for use in conjunction with `BoundingRegion`, the idea here being that you can draw a bracket which runs the width or height of an object (or group of objects) by using the bounding region as an arguement to `FigBracket`.

Alternatively, in `FigBracket[`*sense*`,{`p_1`,`p_2`}]` (Table 13.1), for general oblique orientations of the bracket, the *sense* — `Above` or `Below` — determines the configuration (where the labels are drawn relative to the bracket and which way the end caps face).

Table 13.4 Options for controlling the position of a `FigBracket`.

Option	*Default*	
`Shift`	`0`	Shift to apply to bracket coordinate position.

Shift. If the form `FigBracket[`*side*`,`*coord*`,`*range*`]` is used, then the bracket can be shifted from the given coordinate value *coord* by a distance given through the option `Shift`, shown in Table 13.4. A positive value corresponds to an "outward" displacement of the bracket, *e.g.*, leftward for a `Left` bracket, or rightward for a `Right` bracket. The shift amount may be given in any of the forms described under "Individual coordinates" in Sec. 7.1.1, *e.g.*, `Shift->Canvas[20]` to move the bracket outward by 20 printer's points. Note the use of the name `Shift`, to refer to the displacement of a single coordinate, to avoid ambiguity with `Displacement`, which we use elsewhere for two-dimensional displacements (*e.g.*, Table 13.2).

Table 13.5 Options for endcaps on `FigBracket`.

Option	*Default*	
`ShowEnd`	`True`	Whether or not to draw end caps.
`EndLength`	`0`	Length of endcap, in printer's points, *i.e.*, how far it protrudes past the end (this is the negative of the usual sense for `HeadLength`).
`EndLip`	`{3,0}`	Width of endcap, in printer's points. Lengths `{`*in*`,`*out*`}` are in the inward and outward directions, respectively.
`ShowTail`, `TailLength`, ..., `ShowHead`, `HeadLength`, ...	`Automatic`	*As in Table 11.2, but take values from "endcap" options if left as* `Automatic`.

Endcap options. The options controlling the endcaps on a `FigBracket`, summarized in Table 13.5, are similar in spirit to the arrowhead options for `FigLine` (Table 11.2).

Table 13.6 Options for `FigureBracket`, controlling the bracket label.

Option	*Default*	
`BracketLabel`	`None`	Label to display along bracket.
`ShowBracketLabel`	`Exterior`	Whether or not to show the label.
`BracketLabelPosition`	`Automatic`	Position of the label along the bracket.
`BracketTextColor`,..., `BracketFontFamily`,..., `BracketTextBackground`,...	`Default`	*As defined in Tables 8.5–8.7.*

Bracket label options. The bracket label options, summarized in Table 13.6, have the form of the usual options for an attached label (Table 8.10).

Anchors. The anchors which may be generated from a `FigBracket` object include all the anchors — `Left`, `Right`, `Head`, `Tail`, *etc.* — defined for `FigLine` (Sec. 11.1), as summarized in Table 11.3. For completeness, we note that a `"Bracket"` anchor is also defined. This anchor is meant primarily for internal use in positioning the bracket label, which is constructed as the corresponding attached label.

Attached labels. The principal attached label meant for use with a `FigBracket` is the bracket label, described above. However, attached labels are also available for the `Left`, `Center`, `Right`, `Tail`, and `Head` anchor positions, for consistency with `FigLine`.

13.3 `FigRule`

Description. A `FigRule` is used to draw a horizontal or vertical rule.

Arguments. In `FigRule[`*direction*`,`*coord*`,`*range*`]` (Table 13.1), the *direction* — `Horizontal` or `Vertical` — determines the overall orientation of the rule.

The *coord* specifies the coordinate at which the rule should be placed (this as an x coordinate for a vertical rule, or a y coordinate value for a horizontal rule). Note that this coordinate may be given in any of the various ways described for “individual coordinates” in Sec. 7.1.1, *e.g.*, as `Scaled[`x_s`]`, or by giving an anchor from which the horizontal position should be taken.

The *range* specifies the coordinate range the rule should cover (this is a y coordinate range for a vertical rule, or an x coordinate range for a horizontal rule). For the common special case of drawing a rule which extends across the entire width or height of the panel, the *range* may be given as `All`. Instead of a *range*, a rectangular region may be given, in any of the various ways described in Sec. 7.4 — this is mainly meant for use in conjunction with `BoundingRegion`, the idea here being that you can draw a rule which runs the width or height of an object (or group of objects) by using the bounding region as an arguement to `FigRule`.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Arrowhead options. Simple arrowheads can be drawn at either end of the curve, controlled through the options summarized in Table 11.2.

Anchors. The anchors which can be generated from a `FigRule` are the same as for a `FigLine`, as summarized in Table 11.3. For determining the tail and head, if the entries in *range* have been given in increasing order, then a horizontal bracket runs from left to right, and a vertical bracket from bottom to top.

Attached labels. Attached labels are available for the `Left`, `Center`, `Right`, `Tail`, and `Head` anchor positions, as for a `FigLine`.

14 Graphics inclusion

Table 14.1 Inclusion of externally-generated graphics in a SciDraw figure.

`FigGraphics[`*graphics*`]`	Incorporates Mathematica graphics into a figure, aligned with the current panel's coordinate system.

`FigInset[`*graphics*`]`
`FigInset[`*graphics*`,` *region*`]`
	Renders Mathematica graphics much as is would be displayed by `Show`, scaled so that it covers the given *region*.

14.1 `FigGraphics`

Description. `FigGraphics` incorporates two-dimensional Mathematica graphics into the current panel, so that points in the graphics lie at their correct positions in the current panel's coordinate system. `FigGraphics` is only appropriate for certain graphics, such as plotting output, which are defined in terms of (x,y) coordinates, so that they can meaningfully be aligned with the x and y scales of the panel.

Arguments. In `FigGraphics[`*graphics*`]`, the argument *graphics* may be of type `Graphics`, `ContourGraphics`, or `DensityGraphics`.

Options. `FigGraphics` accepts the options `Show` and `Layer`, with their usual meanings from Sec. 8.1.

14.2 `FigInset`

Description. `FigInset` renders Mathematica graphics (either two-dimensional or three-dimensional) and raster images[1] as they would normally be displayed by Mathematica — either directly as output from a plotting function or as displayed with `Show` (see **ref/Show**) — and scales the result to fill the current panel, or a given *region* of the panel (hence the name "inset"). For instance, the displayed graphics in the figure will even include Mathematica-generated axes (in two or three dimensions) or frames if the original output would have included these. `FigInset` is generally *not* appropriate if you are expecting mathematical coordinates in a plot to align with the coordinate axes on the panel or coordinates of other objects drawn in the panel (that is what `FigGraphics` is for). `FigInset` *is* appropriate if the graphics are simply being considered as a "picture" which needs to fill a given area. Thus, `FigInset` is meant primarily for use with images, three-dimensional plots, and certain specialized Mathematica two-dimensional graphics which contain geometric shapes but do not naturally align with coordinate axes.

Arguments. In `FigInset[`*graphics*`]` or `FigInset[`*graphics*`,` *region*`]`, the argument *graphics* may be of type `Graphics`, `ContourGraphics`, `DensityGraphics`, `Image`, or `Graphics3D`. The graphics will appear as they would be displayed by `Show` (see **ref/Show**). You may wish to apply `Show` directly to the argument to change its appearance, before passing the argument to `FigInset`. For instance, you can remove three-dimensional axes from a three-dimensional plot:

```
    Example3D=ParametricPlot3D[...];
```

[1]Mathematica graphics fall into three main categories: two-dimensional graphics built from primitives (see **ref/Graphics**), raster images (see **ref/Image**), and three-dimensional graphics built from primitives (see **ref/Graphics3D**). All of these are accomodated by `FigInset`.

```
...
FigInset[
 Show[Example3D,Axes->None,Boxed->False]
];
```

The `region` is specified as described in Sec. 7.4. If no *region* is given, this argument is taken as `All`, *i.e.*, the full region covered by the current panel.

Insetting non-graphical expressions (including Mathematica-generated legends). The *graphics* argument to `FigInset` must be genuine Mathematica graphics, *e.g.*, a `Graphics`, `DensityGraphics`, `Image`, or `Graphics3D` object. However, you might sometimes wish to inset another Mathematica formatted expression, such as a Mathematica-generated legend (not to be confused with SciDraw's own `DataLegend`!), which is not technically graphics. This can be accomplished by explicitly wrapping the Mathematica expression as graphics, using Mathematica `Inset` and `Graphics` expressions, for example,

```
Graphics@Inset@PointLegend[Red,Green,Blue,"red","green","blue"]
```

Options. `FigInset` accepts the options `Show` and `Layer`, with their usual meanings from Sec. 8.1. The region covered by `FigInset` may also be adjusted using `RegionExtension` and `RegionDisplacement` options, which have form and meaning as discussed in the context of `AdjustRegion` (Sec. 7.4).

Anchors. The anchors which may be generated from a `FigInset` object are the same as for a `FigRectangle`, summarized in Table 11.8. The anchors are defined with respect to the rectangular region (given as the *region* argument) in which the graphics are inscribed.

15 Level schemes

Table 15.1 Figure objects for level schemes.

`Lev[`x_1`,`x_2`,`E`]`	Generates a level.
`ExtensionLine[`*level*`,`*side*`,`*dx*`]`	Generates an extension line to a level.
`Connector[`$level_1$`,`$level_2$`]`	Generates a connector line between levels.
`BandLabel[`*level*`,`*text*`]`	Generates a “band” label beneath the given level.

`Trans[`$level_1$`,`$level_2$`]`
`Trans[`$level_1$`,`pos_1`,`$level_2$`,`pos_2`]`
	Generates a transition arrow between levels.

15.1 `Lev`

Description. A `Lev` is used to draw a horizontal line representing an energy level.[1] The object consists of an outline only.

Arguments. In `Lev[`x_1`,`x_2`,`E`]` (see Table 15.1), the arguments x_1 and x_2 give the *nominal* horizontal endpoints of the level, and the energy coordinate E gives the *nominal* vertical coordinate. The *actual* horizontal endpoints are indented by an amount determined by the option `Margin`, and the *actual* vertical position can be shifted by an amount determined by the option `VerticalShift`, *e.g.*, to prevent the lines for two closely-spaced levels from overlapping each other (both options are discussed below). The argument E may be given as a *string*, provided that string represents an expression which, when evaluated by Mathematica, yields a number. In this case, the string will be used verbatim for any energy labels attached to the level (see *Automatic energy labels* below).

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Table 15.2 Options for controlling the geometry of `Lev`.

Option	*Default*	
`Margin`	`0.1`	Horizontal indent for each side of level.
`VerticalShift`	`None`	Vertical shift to apply to level, in printer’s points.
`WingHeight`	`0`	Height of wing, for each side of level, in printer’s points.
`WingTipWidth`	`40`	Width of wing tip, for each side of level, in printer’s points.
`WingSlopeWidth`	`10`	Width of wing sloped segment, for each side of level, in printer’s points.
`MakeWing`	`True`	Whether or not to make wing, on each side of level.

`Margin`. The `Margin` option gives the horizontal indent for the sides of a level relative to the nominal endpoints x_1 and x_2. Thus, for `Margin->`d, the true endpoints are $x_1 + d$ and $x_2 - d$. Different values may be specified for the left and right sides, by giving these values as a list, as `Margin->`$\{d_1, d_2\}$.

[1]An overview of the concepts involved in drawing a level scheme is presented in Sec. 4.4 of the user’s guide.

VerticalShift. The `VerticalShift` option gives a vertical shift to apply to level, in printer's points.

WingHeight, WingTipWidth, WingSlopeWidth & MakeWing. These options control the wing dimensions. A wing is only drawn if `WingHeight` is nonzero *and* `MakeWing` is `True`. For all these options, different values may be specified for the left and right sides, by giving these values as a list, *e.g.*, `MakeWing->{False,True}` indicates that a wing is only to be drawn on the right hand side of the level.

Table 15.3 Named anchors for `Lev` objects.

Name	*Argument*	
`Left`	—	*Position:* At the left endpoint of the level. *Text offset:* To left of line — `{+1,0}`. *Orientation:* Horizontal — *likewise for all anchors below*.
`Right`	—	*Position:* At the right endpoint of the level. *Text offset:* To right of line — `{-1,0}`.
`Bottom`	—	*Position:* At the midpoint of the level. *Text offset:* Beneath line — `{0,+1}`.
`Top`	—	*Position:* At the midpoint of the level. *Text offset:* Above line — `{0,-1}`.
`Center`	—	*Position:* At the midpoint of the level. *Text offset:* Centered on line — `{0,0}`.
`Level`	x'	*Position:* At height of the central part of level, and horizontal position x' relative to the nominal left endpoint x_1. *Text offset:* Centered on line — `{0,0}`.
	`{Left,`x'`}` or `{Right,`x'`}`	Similarly, but at the height of the left or right endpoint of level, which may be different from the height of central part of level, if there are wings.

Anchors. The anchors which can be generated from a `Lev` object are summarized in Table 15.3.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Left`, `Right`, `Bottom`, `Top`, and `Center` anchor positions.

Table 15.4 Options for automatic energy label generation for `Lev`.

Option	*Default*	
`DecimalDigits`	`Automatic`	Decimal digits for fixed-point formatting of energy labels.
`EnergyLabelFunction`	`Automatic`	Custom function to generate energy label.

Automatic energy labels. If the value of any of the attached labels is set to `Automatic`, the text of that label will be a formatted representation of the level energy. Formatting is controlled by the options summarized in Table 15.4 and is accomplished as follows: (1) If E was given as a string, *e.g.*, `"100.0"`, that string is used. (2) If `EnergyLabelFunction` is a function (any value other than `Automatic` is presumed to represent a function name or lambda function), this function will be applied to the numerical value of the energy. (3) For `EnergyLabelFunction->Automatic`, the function `FixedPointForm` (see the `CustomTicks` documentation) will be used to format the number according to the number of decimal digits given by `DecimalDigits`. (4) However, with `DecimalDigits->Automatic`, the default value, the number will be formatted as it would normally be displayed by Mathematica (which, for floating point numbers, may cause trailing zeros to be omitted after the decimal point).

Table 15.5 Functions related to level objects.

`LastLevel[]`	Returns a reference to the most recent `Lev` object.
`LevelEnergyLabel[`*level*`]`	Returns the formatted energy label generated for use with a given `Lev` object.

Level object functions. Two functions which return information on levels are summarized in Table 15.5. `LastLevel[]` returns a reference to the most recently constructed level. It may be used anywhere a level name is accepted as an argument. `LevelEnergyLabel[`*level*`]` returns the formatted energy label text which was constructed for a given level (regardless of whether or not it was actually used in an attached label at the time the level was constructed). For instance, this expression may be used in a `FigLabel` or to label an arrow to the level.

15.2 `ExtensionLine`

Description. An `ExtensionLine` is used to draw a horizontal extension to an energy level drawn with `Lev`. The object consists of an outline only.

Arguments. `ExtensionLine[`*level*`,`*side*`,`*dx*`]` (Table 15.1) draws an extension line of length *dx* on side *side* (`Left` or `Right`) of the level *level*.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Table 15.6 Option for controlling the geometry of `ExtensionLine`.

Option	*Default*	
`ToWing`	`True`	Whether or not to draw the extension at the height of the wing, if any.

`ToWing`. With `ToWing->True`, which is the default, the extension line will attach to the endpoint of the *wing*, if there is one on the given side (Table 15.6). However, with `ToWing->False`, the extension line will be drawn at the same height as the central part of the level.

Anchors. The anchors which can be generated from an `ExtensionLine` object are `Left` and `Right` anchors, similar to those noted for `Lev` in Table 15.3.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Left` and `Right` anchor positions.

15.3 `Connector`

Description. A `Connector` is used to draw a connecting line (from left to right) between two levels drawn with `Lev`. The object consists of an outline only.

Arguments. In `Connector[`$level_1$`,`$level_2$`]` (see Table 15.1), the arguments are the names of the levels to be connected. The line will extend from the right endpoint $level_1$ to the left endpoing of $level_2$.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Table 15.7 Named anchors for `Connector` objects.

Name	*Argument*	
`Bottom`	*u* (optional)	*Position:* A fraction *u*, from `0` to `1`, along the line, or at the midpoint if *u* is omitted. *Text offset:* Beneath the line — {`0,+1`}. *Orientation:* Sloped along the line.
`Top`	*u* (optional)	Similarly, but with text offset {`0,-1`}, *i.e.*, above the line.
`Center`	*u* (optional)	Similarly, but with text offset {`0,0`}, *i.e.*, centered on the line.

Anchors. The anchors which can be generated from a `Connector` are summarized in Table 15.7.

Attached labels. Attached labels (as described in Sec. 8.2) are available for the `Bottom`, `Top`, and `Center` anchor positions.

15.4 `BandLabel`

Description. A `BandLabel` is a special label meant to label a "band" of levels, in the spectroscopic sense. Note that `BandLabel` is essentially a shorthand for a special form of `FigLabel`, in that `BandLabel[`*level*`,`*text*`,VerticalShift->`*v*`]` is a more concise way to obtain the same result as one could also obtain with `FigLabel[`*level*`,Bottom,`*text*`,Displacement->Canvas[{0,`*v*`}]]`.

Arguments. The label `BandLabel[`*level*`,`*text*`]` (Table 15.1) is drawn at the `Bottom` anchor of the given level, shifted by a vertical distance given by the option `VerticalShift`.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the text and background/frame options (Tables 8.5–8.8).

Table 15.8 Option for controlling the position a `BandLabel`.

Option	*Default*	
`VerticalShift`	`-3`	Vertical displacement of label relative to level, in printer's points.

`VerticalShift`. Typically a band label would be drawn with a vertical space between it and the level it is attached to, rather than directly flush with the bottom of the level. This distance is specified through the option `VerticalShift` (Table 15.8).

Anchors. The anchors which can be generated from a `FigLabel` object are identical to those for a `FigRectangle`, as summarized in Table 11.8.

Attached labels. Not applicable.

15.5 `Trans`

Description. A `Trans` is used to draw an arrow (representing a "transition") from one level to another. Its properties and use are essentially identical to those of a `FigArrow` (Sec. **??**), except that the starting and ending points for a `Trans` are determined from the positions of `Lev` objects rather than from a generic *curve* argument.

Arguments. In the form `Trans[`$level_1$`,`pos_1`,`$level_2$`,`pos_2`]` (see Table 15.1), the arguments are the names of the starting and ending levels, respectively, and the horizontal positions on these levels to be used for the arrow endpoints. The positions are measured with respect to the *nominal* left endpoints of the levels, ignoring any margins, *i.e.*, pos_1 is measured horizontally relative to the x_1 argument of $level_1$, and pos_2 is measured horizontally relative to the x_1 argument of $level_2$. If one of these arguments pos_1 or pos_2 is given as `Automatic`, the arrow will be drawn vertically, at the position determined by the other, explicitly specified point. In the form `Trans[`$level_1$`,`$level_2$`]`, the position arguments are instead taken from the option `EndPositions` (described below).

Parent object. `Trans` inherits its option values from `FigArrow`, for which the options are described in detail in Sec. 12.[2] However, for `Trans`, the default values of `TailFlush` and `HeadFlush` are changed to `True`.

General options. The appearance is controlled through the usual options of Sec. 8, in particular, the general options (Table 8.1) and outline options (Table 8.2).

Table 15.9 Options for controlling the endpoints and intermediate points of a `Trans`.

Option	*Default*	
`EndPositions`	`0.5`	Horizontal positions of the arrow endpoints, relative to the nominal left endpoints of the starting or ending levels.
`IntermediatePoints`	`None`	Points through which the arrow should pass along the way.

Transition arrow geometry. The options for controlling the endpoints and intermediate points of a `Trans` are summarized in Table 15.9.

If the form `Trans[`$level_1$`,`$level_2$`]` is used, without the position arguments pos_1 and pos_2, then their values are taken from the option `EndPositions`, which may either be given as a single value `EndPositions->`*pos* or a list of two different positions `EndPositions->`$\{pos_1, pos_2\}$ for the tail and head. Either of the position arguments may be a number or `Automatic`, as described earlier.

Just as the curve determining a `FigArrow` may consist of more than two points, a `Trans` may have "kinks", *i.e.*, it may run through additional intermediate points not on the line from the starting point to the ending point. A list of intermediate points is provided through the option `IntermediatePoints`. These may be specified in any of the forms allowed for points along a curve, as described in Sec. 7.2.1. In practice, it may be particularly useful to use `DisplaceTail`, `DisplaceHead`, and related specifications (Table 7.7).

Anchors. The anchors which can be generated from a `Trans` are identical to those for a `FigLine` or `FigArrow`, summarized in Table 11.3. See also the discussion of anchors for `FigArrow` (Sec. 12).

Attached labels. Attached labels (as described in Sec. 8.2) are available for same anchor positions as for `FigArrow`, namely, `Tail`, `Head`, `Left`, `Center`, and `Right`.

[2]That is, the default values for the options are initially defined as `Inherited`, and, if they are left as `Inherited`, their values will be taken from the default values defined for `FigArrow`. For the general options of Sec. 8, these values may in turn also be `Inherited`, in which case they are taken from the default values defined for `FigObject`.

15.6 Decay scheme generation

Table 15.10 Automatic decay scheme generation commands.

`AutoLevelInit[`x'`,`dx`,`Dx`]`	Initializes the positioning parameters for decay scheme generation.
`AutoLevel[`$level_1$`]`	Selects a new starting level for transitions.
`AutoTrans[`$level_2$`]`	Draws a transition to the designated ending level.

These commands automate aspects of the generation of beta decay schemes. `AutoLevelInit[`x'`,`dx`,`Dx`]` initializes the parameters, for the first transition to be drawn at horizontal coordinate x' relative to the nominal left endpoint of the level (the *pos* argument of `Trans`), with step dx between transitions from the same level and Dx between the last transition from one level and the first transition from the next level. `AutoLevel[`$level_1$`]` selects the starting level for subsequent transitions drawn with `AutoTrans`. `AutoTrans[`$level_2$`]` draws a transition to the designated ending level. Their use is illustrated in Sec. 4.4 of the user's guide.

16 Data plotting

Table 16.1 Figure objects for data plotting.

`DataPlot[`*data*`]`	Generates a plot of the given data set *data*.
`DataLegend[`*p*`,{{`*style*$_1$`,`*text*$_1$`},{`*style*$_2$`,`*text*$_2$`},...}]`	Generates a data legend, located at *p*, for the given plot styles.

16.1 `DataPlot`

Description. `DataPlot` generates an (x,y) plot from a data set.[1] We first consider the numerical *data set* — the appropriate format for the data set and how its interpretation is controlled by options to `DataPlot` — in Sec. 16.1.1. Then we consider the *appearance* of the plot — controlled through the options defined for different plot elements `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop` and through styles — in Sec. 16.1.2

Comment on object name. As usual for figure objects, a data plot may be given a name when it is created, as `DataPlot`⟦*name*⟧`[`*data*`]`. This name may be used to attach labels to the plot, connect arrows to it, *etc.*, exactly as for a `FigLine` object.

16.1.1 Data sets

Data set as an array. A `DataPlot` must be given a *data set*, by which we mean simply an *array* of numbers, *i.e.*, a list of lists. Each row contains the information for one data point, much as in a standard spreadsheet. An array (or, equivalently, a matrix) is represented in Mathematica as a list of lists. An introduction is provided in **tutorial/VectorsAndMatrices**. Here we simply note that Mathematica provides many functions for manipulating such arrays, for instance, selecting subsets of the rows or columns (see **guide/HandlingArraysOfData**). You can display the array in a more readable, tabular form using `TableForm` (see **ref/TableForm**). In the simplest case, each row of a data set will contain only (x,y) information, and the set will therefore be represented as a list of lists of the form `{{`x_1`,`y_1`}, {`x_2`,`y_2`}, ..., {`x_n`,`y_n`}}`.

Generating the data set. The data in a data set can come from various sources. For small data sets, you might simply enter the data manually into the Mathematica notebook, *e.g.*, either directly as a list of lists

```
DataSet1={{1,4.3},{2,5.7},...};
```

This can be input equivalently but in a more spreadsheet-like format using the table input capability of Mathematica's notebook interface (see **tutorial/EnteringTablesAndMatrices**).

More powerful is the possibility of calculating arrays of data *within* Mathematica and then plotting them. The `Table` command is particularly useful in this context (see **tutorial/MakingTablesOfValues**). As a simple example, if `f` is some function you have defined, and you want to plot its values on the interval $[0,10]$ in steps of 0.25, you might define

```
DataSet2=Table[{x,f[x]},{x,0,10,0.25}];
```

Alternatively, you can input tabular data from external data files using `Import` (see **tutorial/ImportingAndExportingData**). For example,

[1] An example-based introduction to data plotting is given in Sec. **??** of the user's guide.

```
DataSet3=Import["c:/data/expt5/run001.dat"];
```

In more complex situations, you could write your own code to read in data using Mathematica's lower-level input functions, such as `Read`, `ReadList`, or `BinaryRead`.

SciDraw provides some tools to help manipulate data once they are in tabular form (Sec. 16.3).

Table 16.2 Options for `DataPlot`, for selecting data columns.

Option	*Default*	
`DataColumns`	`{1,2}`	The data set columns $\{c_x, c_y\}$ which provide x and y coordinate values of the data points.
`XErrorColumn`	`None`	The data set column c_{σ_x} for x uncertainties, or columns $\{c_{\sigma_x^-}, c_{\sigma_x^+}\}$ for downward and upward x uncertainties.
`YErrorColumn`	`None`	The data set column c_{σ_y} for y uncertainties, or columns $\{c_{\sigma_y^-}, c_{\sigma_y^+}\}$ for downward and upward y uncertainties.
`SymbolOptionColumns`	`None`	Data set columns providing control of the data symbol appearance on a point-by-point basis.
`ColumnNames`	`None`	List $\{name_1, \ldots\}$ of column names.

Selecting columns in the data set. In general, the array which stores a data set can have more than just two columns. The x and y data need not be in columns 1 and 2, respectively. Indeed, different columns of the same array can be used as x and y values in different data plots. Uncertainties ("error bars") for the x and y values can also be stored as columns in the array. `DataPlot` interprets these columns according to the options listed in Table 16.2.

The default option values `DataColumns->{1,2}`, `XErrorColumn->None`, and `YErrorColumn->None` suppose a data set with the simple data structure

$x\ y$

described above.

For an example of alternative column assignments, suppose we have nuclear mass M and quadrupole moment Q data, with uncertainties σ_M and σ_Q, tabulated as functions of nuclear neutron number N and proton number Z, with the columns arranged as

$N\ Z\ M\ \sigma_M\ Q\ \sigma_Q$

Then a plot of Q *vs.* Z, with error bars shown on Q, would be obtained by using the options `DataColumns->{2,5}`, `XErrorColumn->None` (default), and `YErrorColumn->6`. A plot of M *vs.* N, with error bars shown on M, would be obtained from the same table by using the options `DataColumns->{1,3}`, `XErrorColumn->None` (default), and `YErrorColumn->4`.

Data may have distinct lower and upper uncertainties, most generally as $(x^{+\sigma_x^+}_{-\sigma_x^-}, y^{+\sigma_y^+}_{-\sigma_y^-})$. The different quantities in this expression may be stored in different columns, for instance, as

$x\ \sigma_x^-\ \sigma_x^+\ y\ \sigma_y^-\ \sigma_y^+$

Then, a plot including error bars would be obtained by using the options `DataColumns->{1,4}`, `XErrorColumn->{2,3}`, and `YErrorColumn->{5,6}`. Note that we adopt the convention that the lower uncertainties should be tabulated as *positive* values.

Missing data and missing uncertainties. Missing x or y values in a data set can be indicated by giving them value `Missing[]`. This follows the convention of many of Mathematica's database and plotting functions (see **ref/Missing**). If either the x or y value is missing for a data point, the data point will be ignored. Likewise, the *x* or *y* uncertainties may also be given as `Missing[]`, in which case the data point will be plotted but the error bar will be omitted.

Controlling symbol appearance point-by-point (`SymbolOptionColumns`). Sometimes, it is necessary to override the default appearance properties (shape, size, color, *etc.*) of the symbols representing one or more individual data points in a data set. Sometimes the appearance needs to be modified for just some subset of the data points, *e.g.*, to highlight these points, while the rest of the data points retain their default appearance. Alternatively, it may be necessary to specify some aspect of the appearance individually for *each* data point, *e.g.*, if the size of each data point is meant to convey some quantitative information. The appearance of the data symbols is controlled through options, described in detail below in Sec. 16.1.2. Here we simply note that the data symbol options `SymbolShape`, `SymbolSize`, `SymbolSizeScale`, and all general/line/fill appearance options (such as `Color`, `FillColor`, or `LineColor`) can be overridden on a point-by-point basis. Values for the option are given in one of the columns of the data set, selected by specifying `SymbolOptionColumns->{`*option*`->`*column*`}` as an option to `DataPlot`. For instance, with `SymbolOptionColumns->{SymbolSize->3,FillColor->4}`, the third column of the data set will control the symbol sizes, and the fourth column will control the symbol colors. If the value for any of the appearance properties is given as `Default` (or `Missing[]`) for some data point, the default value of the option will be used for that data point.

Giving names to columns (`ColumnNames`). Rather than spacifying a column by its number, you may also specify the column by a name. Column names are defined through the option `ColumnNames`, which is a list of names for the columns. Column names *must* follow the same naming convention as was recommended for objects, in Sec. 6. A column name should typically be given as a string, *e.g.*, `"x"`. Alternatively, a column name may be given as a list beginning with a string and followed by some other descriptive information, *e.g.*, `{"y",1}` or `{"y",2}`. Thus, for instance, in the above example, you could alternatively specify

```
DataPlot[...,
  ColumnNames->{"N","Z","M","sigmaM","Q","sigmaQ"},
  DataColumns->{"Z","Q"},
  YErrorColumn->"sigmaQ"
]
```

You can freely intermix column numbers and names.

Table 16.3 Options for `DataPlot`, for selecting axis scale transformations.

Option	*Default*	
`XAxisScale`	`None`	The axis scale transformation to apply to the *x* axis data.
`YAxisScale`	`None`	The axis scale transformation to apply to the *y* axis data.

`XAxisScale` & `YAxisScale`. The `XAxisScale` and `YAxisScale` options control the axis scales — linear (possibly rescaled), logarithmic, or otherwise. The possible values are `None` for ordinary linear axes (default), `{Scaled,`*m*`}` for rescaling so that the values are plotted in units of the given multiplier *m* ($x \to x/m$), `{Linear,`*a*`,`*b*`}` for an arbitrary linear rescaling $x \to ax+b$, `Log` for logarithmic scaling to base 10, or more generally `{Log,`*base*`}` for logarithmic scaling to a given *base*. Alternatively, other scalings can be defined through customization functions as described in Sec. 16.1.3.

16.1.2 Data plot appearance

Table 16.4 Options for `DataPlot` affecting the appearance of the data plots.

Option	*Default*	
`Style`	`None`	*This is the usual `Style->`style option, as defined in Table 8.1. However, we repeat it explicitly here, since this option takes on special importance for `DataPlot`, since styles are used to control the plot appearance, by storing sets of options for `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop`.*
`DataLine`	{}	List of additional options to be applied to the data curve `DataLine`, overriding those in the defaults or set by *style*.
`DataSymbol`	{}	List of additional options to be applied to the data symbol and error bars `DataSymbol`, overriding those in the defaults or set by *style*.
`DataFill`	{}	List of additional options to be applied to the data fill `DataFill`, overriding those in the defaults or set by *style*.
`DataDrop`	{}	List of additional options to be applied to the data drop lines `DataDrop`, overriding those in the defaults or set by *style*.

Plot elements. The appearance of the data plot is controlled by three different sets of options, for the several different posible elements of a data plot — the curve (`DataLine`), the symbol (`DataSymbol`), which may also include error bars, and the fill and/or drop lines beneath or above the curve (`DataFill`/`DataDrop`). By default, the data plots generated by `DataPlot` include a curve and symbols but no fill, but this choice is controlled through the options for these plot elements.

Controlling the plot style. Styles (Sec. 9) play an especially important role in plotting data with `DataPlot` and subsequently generating corresponding legends with `DataLegend`. styles are used to control the plot appearance, by storing sets of options for `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop`.

To start with, we should observe that the data plot appearance can be set at three different levels of generality, that is, affecting (1) all plots, (2) all plots of a given *style*, or (3) just an individual plot. To elaborate:

(1) If `DataPlot` is simply invoked as `DataPlot[`*data*`]`, with no options, then the appearance of the data plot is controlled by the default values of the options for `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop`, considered in detail below (Tables 16.5–16.7). These may be set, as usual, with `SetOptions`.

(2) Different plot styles may be defined and named using `DefineStyle` (see Sec. 9.1). If a style is to be used to control the style of a plot, it should define options for `DataLine`, `DataSymbol`, `DataFill`, and/or `DataDrop`. This style is then invoked for a specific plot through the usual `Style` option. The general scheme is thus

```
DefineStyle[plotstyle,{DataLine->{...},DataSymbol->{...},...}];
...
```

```
DataPlot[data,Style->plotstyle];
```

The same style that is used to control the appearance of the plot can then be used to generate a corresponding entry in the legend generated with `DataLegend`.

(3) Options for `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop` may be specified for just a single data plot by giving lists of these options `DataPlot` — through options named, originally enough, `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop` — as summarized in Table 16.4. Options for `DataLine`, `DataSymbol`, `DataFill`, and `DataDrop` given directly in this fashion take precedence over the default option values or, if a `Style` option has been given, any option values obtained from that style. `DataPlot` in this case is invoked as

```
DataPlot[data,DataLine->{option->value,...},
DataSymbol->{option->value,...},...,Style->plotstyle];
```

This is a quick-and-dirty approach meant for adjusting the appearance of one-off plots, not for systematic use with plots which should have similar styling across multiple figures or plots which are to be labeled in legends. One should therefore develop the habit of using a style instead.

Table 16.5 Options for `DataLine`.

Option	*Default*	
`CurveShape`	`"Straight"`	Shape of the curve between data points.
`Show`, `Color`, `LineThickness`, `LineDashing`, ...	`Inherited`	*As in defined in Tables 8.1 and 8.2.*

Curve appearance options. The appearance of the data *curve* is controlled by the options for `DataLine`, summarized in Table 16.5. The graphical appearance of the line itself (color, thickness, *etc.*) is controlled, as usual, through the basic figure object options described in Sec. 8. The relevant options are the general options in Table 8.1 and outline options in Table 8.2. Display of the curve is disabled by setting `Show->False`.

`CurveShape`. The shape of the curve is specified by the option `CurveShape`. The possible values are `"Straight"` for straight line segments between data points (default), `"Step"` for rectangular steps, or `"Histogram"` for rectangular steps separated by drop lines to the *x*-axis. (Alternatively, `"SideStep"` and `"SideHistogram"` can be used for “sideways” plots, where *x* is the dependent variable. Then the steps are vertical and the drop lines go horizontally to the *y*-axis.) Other curve shapes may be defined as needed by the user, as described in Sec. 16.1.3.

Table 16.6 Options for `DataSymbol`.

Option	*Default*	
`SymbolShape`	`"Circle"`	Shape to be used for the data points.
`SymbolSize`	`2.5`	Size (diameter) to be used for the data points.
`SymbolSizeScale`	`1.0`	Adjustment factor to symbol size (diameter), not affecting error bar size.
`Show`, `Color`, `LineColor`, `LineThickness`, `LineDashing`, `ShowFill`, `FillColor`, ...	`Inherited` ...,	*As in defined in Tables 8.1, 8.2, and 8.3.*

`SymbolShape`. The `SymbolShape` option determines the shape of the data symbols. Several symbol shapes are predefined. The possible closed (polygonal) shapes are: `"Circle"`, `"Square"`, `"Diamond"`, `"UpTriangle"`, `"DownTriangle"`, `"LeftTriangle"`, `"RightTriangle"`, {`"Polygon"`,*n*} (for a regular *n*-gon). There are several predefined shapes which also just contain line segments: `"Plus"`, `"Cross"`, `"Horizontal"`, `"Vertical"`, and `"Asterisk"`. A value of `None` indicates that no *shape* should be drawn for the symbol, but note that this still allows the *error bars* to appear (in contrast, `Show->False` or `Color->None` would hide the entire symbol, including error bars). Others symbol shapes may be defined as needed by the user, as described in Sec. 16.1.3.

Symbol appearance options. The appearance of the data *symbol* and *error bars* is controlled by the options for `DataSymbol`, summarized in Table 16.6. Each symbol has both an outline (which includes the error bars) and a fill. The graphical appearance of these (line color, line thickness, fill color, *etc.*) is controlled, as usual, through the basic figure object options described in Sec. 8. The relevant options are the general options in Table 8.1, the outline options in Table 8.2, and the fill options in Table 8.3. An "open" symbol is obtained by setting either `ShowFill->False` or `FillColor->None`. Display of the symbol and error bars is disabled entirely by setting `Show->False`. Please refer to the discussion of `SymbolOptionColumns` in Sec. 16.1.1 to see how the appearance of each symbol can be controlled separately.

`SymbolSize`. The `SymbolSize` option determines the size of the data symbols, in printer's points. This quantity is, roughly speaking, the full distance across the symbol — thus, *e.g.*, for the circle, the *diameter*, not the radius. This option also sets the widths of the caps for the error bars.

`SymbolSizeScale`. The `SymbolSizeScale` option provides an adjustment factor to the size of the data symbol specified by `SymbolSize`. However, this adjustment specifically does *not* affect the width of the caps for any error bars. There are two principal uses for this option. This option may be used to provide some aesthetic "compensation" to the natural sizes defined for symbols. For instance, you might decide you want to make the symbols in a plot which uses crosses a little bigger than in a plot which uses squares, say, `SymbolSizeScale->1.2`, even though the same `SymbolSize` is specified for both plots — perhaps to ensure that a cross will still be visible even if it is hidden behind a square. This option is also the recommended means to convey z data through symbol sizes. The base size for all symbols is set by the `SymbolSize` option (perhaps in a style), then the `SymbolSizeScale` values are taken from the z data column, specified via `SymbolOptionColumns`.

Table 16.7 Options for `DataFill` or `DataDrop`.

Option	*Default*	
`Filling`	`None`	Type of fill to add under (or around) curve.
`Direction`	`Vertical`	Fill direction.
`Show`, `Color`,...	`Inherited`	*As in defined in Tables 8.1, 8.2, and 8.3.*

Fill appearance options. The appearance of the data *fill* is controlled by the options for `DataFill`, summarized in Table 16.7. The "fill" is actually a polygon, which can in general have both an outline and a fill. The graphical appearance of these (line color, line thickness, fill color, *etc.*) is controlled, as usual, through the basic figure object options described in Sec. 8. The relevant options are the general options in Table 8.1, the outline options in Table 8.2, and the fill options in Table 8.3.

Two of the basic appearance options for `DataFill` do not have the usual default values indicated in Sec. 8. Most commonly, it is not desirable for the fill beneath a data curve to have an outline, as this would visually overlap with the data curve itself. Thus, the default value of the option `ShowLine` is `False`, differing from the usual default value shown in Table 8.2. Also, it is usually not desirable for the fill to partially or completely obstruct the axis or other previously-drawn data plots. Therefore, the default value of the option `Layer` is `0`, also differing from the usual default value shown in Table 8.9. This serves to "push back" the fill, so that it is behind the data plots and any other objects drawn in the current panel.

`Filling`. The option `Filling` may have the values `None` for no fill (default), `Axis` for filling from the curve to the axis, `-Infinity` or `Infinity` for filling below or above the curve, respectively, or some other numerical value for filling from the curve to that value. Alternatively, the name of another `DataPlot` object may given as the option value, in which case a fill is drawn in the region between two plots. In particular, a *data band* can be drawn by drawing a fill between data plots of the lower and upper boundary curves (which could themselves be made invisible, leaving just the band, if desired, by setting `Show->False` for `DataLine` and for `DataSymbol`).

`Direction`. The `Direction` option may have the values `Vertical` for filling from the data curve *up or down* to a given *y* value (default) or `Horizontal` for filling from the data curve *sideways* to a given *x* value. This option is irrelevant for fills *between* two data plots.

Drop line appearance options. The appearance of the data *drop lines* is controlled by the options for `DataDrop`, summarized in Table 16.7. The graphical appearance of the lines themselves (color, thickness, *etc.*) are controlled, as usual, through the basic figure object options described in Sec. 8. The relevant options are the general options in Table 8.1 and outline options in Table 8.2.

`Filling`. The option `Filling` may have the values `None` for no drop lines (default), `Axis` for drop lines from the curve to the axis, `-Infinity` or `Infinity` for drop lines below or above the curve, respectively, or some other numerical value for drop lines from the curve to that value.

`Direction`. The `Direction` option may have the values `Vertical` for drop lines from the data curve *up or down* to a given *y* value (default) or `Horizontal` for drop lines from the data curve *sideways* to a given *x* value.

Anchors for `DataPlot`. The anchors which can be generated from a `DataPlot` object are the same as for a `FigLine`, which are summarized in Table 11.3 of Sec. 11.1. The anchor positions are defined with reference to the "curve" which would pass through the data points, regardless of whether the curve (`DataLine`) or just the symbols are visible in the plot. Note that the `Left` and `Right` anchor names refer to the "left" and "right" sides of the *curve*, as introduced in Sec. **??**, not necessarily the left and right sides of the *page*. Indeed, for the common case in which we are plotting relatively "flat" data (slope approximately zero) and specify the points in the usual order (increasing *x* coordinate) the "left" side of the curve is actually the "top" of the plot.

Table 16.8 Special default values for options affecting the attached labels for `DataPlot`.

Option	*Default*	
`HeadTextOrientation`	`Horizontal`	*Note special default values, relative to Table 8.8.*
`TailTextOrientation`	`Horizontal`	

Attached labels for `DataPlot`. Attached labels (as described in Sec. 8.2) are available for the `Left`, `Center`, `Right`, `Tail`, `Head`, and `Point` anchor positions. The `Head` and `Tail` labels for `DataPlot` are most commonly used to label the left or right endpoints, respectively, of a relatively horizontal curve — in recognition of this situation, the default values for their *X*`TextOrientation` options have been changed from the usual `Automatic` (which would align the labels with the tangent to the curve) to `Horizontal`, as noted in Table 16.8.

16.1.3 Defining new axis scales, symbol shapes, and curve shapes

Table 16.9 Functions for defining new axis scales, symbol shapes, and curve shapes for data plots.

`DefineAxisScale[`*name*`,`*function*`]`	Defines a new data axis scale function.
`DefineDataSymbolShape[`*name*`,` *points*`]`	Defines a new data symbol shape.
`DefineDataLineShape[`*name*`,`*function*`]`	Defines a new data curve shape.

Further possible values for some of the data plotting options may be defined using the customization functions summarized in Table 16.9.

`DefineAxisScale.` `DefineAxisScale[`*name*`,`*function*`]` defines *name* to represent a new axis scaling type, for use with the options `XAxisScale` and `YAxisScale`. The name should typically either be a string or else a brace-delimited list consisting of a string plus one or more parameter patterns. (In place of a string, it may also be appropriate to use the symbolic name of an existing Mathematica function, as in `Log`.) The *function* should be a function of a single variable,[2] which accepts the coordinate value as its argument and returns the scaled value, or else `-Infinity` or `+Infinity` for out-of-range values. For instance, the `Log` and {`Log,`*base*} axis scales described above in Sec. 16.1.1 are defined within the SciDraw code by

```
LimitedLog[Base_,x_?Positive] = Log[Base,x];
LimitedLog[Base_,_?NonPositive] = -Infinity;
DefineAxisScale[Log, LimitedLog[10,#]&];
DefineAxisScale[{Log,Base_}, LimitedLog[Base,#]&];
```

[2]The *function* could either be the name of a function or be the function itself, as a "pure function" — basically, anything which Mathematica can successfully apply to a single numerical argument and get back a return value. See **ref/Function** for more on defining a pure function.

DefineDataSymbolShape. `DefineDataSymbolShape[`*name*`,`*points*`]` defines *name* to represent a new data symbol shape, for use with the option `SymbolShape`. The name should typically either be a string or else a brace-delimited list consisting of a string plus one or more parameter patterns. The argument *points* should be an expression which evaluates to a list of points. These would describe the symbol centered on the origin and contained in a box approximately covering the coordinate intervals $[-1,+1]$ — in which case the `SymbolSize` option will then reasonably represent a size in printer's points. For instance, the `"Diamond"` and `{"Polygon",`*n*`}` symbol shapes described above in Sec. 16.1.2 are defined within the SciDraw code by

```
DefineDataSymbolShape[
  "Diamond",
  1.2*{{0.,1.},{-1.,0.},{0.,-1.},{1.,0.}}
];
DefineDataSymbolShape[
  {"Polygon",n_},
  N[Table[
    {Cos[2*Pi*i/n+Pi/2],Sin[2*Pi*i/n+Pi/2]},
    {i,1,n}
  ]]
];
```

DefineDataLineShape. `DefineDataLineShape[`*name*`,`*function*`]` defines *name* to represent a new data curve shape, for use with the option `CurveShape`. The name should typically either be a string or else a brace-delimited list consisting of a string plus one or more parameter patterns, as illustrated above for `DefineDataSymbolShape`. The *function* should be a pure function which accepts a list of (x,y) data points (these will be in canvas coordinates) as its argument and returns a list of points for the curve. Examples may be found in the SciDraw source code file `FigData.nb`.

16.2 DataLegend

Description. `DataLegend` generates a legend to identify one or more data plots.

Arguments. A `DataLegend` may simply be viewed as a special type of `FigLabel` — one in which the label *text* is replaced by an array of legend entries. In `DataLegend[`*p*`,{{`$style_1$`,`$text_1$`},{`$style_2$`,`$text_2$`},...}]` (Table 16.1), the point *p* determines where the legend should be positioned. By default, it gives the position at which the top left corner of the legend should be anchored, but this can be changed with the `TextOffset` option, as discussed further below.

The list `{{`$style_1$`,`$text_1$`}, {`$style_2$`,`$text_2$`},...}` consists of a list of pairs of plot styles together with the text which should accompany each one. The style is used to draw a sample line segment and data symbol. The special value `None` may be given in place of $style_i$, in which case no line segment or symbol is drawn, but the text is still included (this is useful for including, *e.g.*, comment entries in the legend, or headers for different groups of data sets). The more general form `{`*style*`,`*text*`,`*symbol*`}` allows the style for the sample line segment in a legend entry to be taken from the options defined for a symbol other than a data plot (`DataLine`). Typical examples would be a legend entry based on the appearance of `FigLine` or `FigRule`. No sample symbol is shown in this case, just the sample line segment. (For consistency, the standard behavior is recovered with `{`*style*`,`*text*`,DataPlot}`, which is equivalent to `{`*style*`,`*text*`}`.)

Table 16.10 Options for `DataLegend`.

Option	*Default*	
`TextOffset`	`TopLeft`	*Note different default value relative to Table 8.8.*
`Width`	`20`	The width, in printer's points, of each line sample entry.
`InternalSeparation`	`2`	The separation, in printer's points, between the line sample and text label within an entry.
`EntrySpacing`	`{10,2}`	The horizontal and vertical separation, in printer's points, between entries.
`RowLimit`	`None`	The maximum number of rows of entries.

Appearance and positioning options (including `TextOffset`). The appearance and positioning options for `DataLegend` are exactly as for a `FigLabel` (Sec. 13.1). Positioning is controlled by giving a point *p* for the position, and indicating through the option `TextOffset` option (Table 8.8) where exactly this point should lie on the face of the rectangular legend. It is often more suitable to specify the position of a legend by a corner, than by the center, if one edge is to be kept near and a fixed distance away from an edge of the surrounding panel. (The center is not an appropriate anchoring point, since all sides of the legend move outward unpredictably from the center as new entries are added or old entries are edited.) The default value of the option `TextOffset` for `DataLegend` is therefore `TopLeft` (or {`-1,+1`}), differing from the usual default value shown in Table 8.8. Note also that it is generally most convenient to use `Scaled` coordinates for the *point*, so that the legend falls at a definite fractional position within the panel, and is not affected by changes in the choice of axis scales.

`DataLegend` background and frame. The `DataLegend` can also be given a background and frame, controlled through the options in Table 8.7.

`Width`. The `Width` option determines the horizontal length, in printer's points, of the line sample. The same width is reserved as blank space even if no line is actually visible and only a symbol is present.

`InternalSeparation`. The `InternalSeparation` option determines the separation, in printer's points, between the line sample part of each legend entry (left part) and its accompanying text label (right part).

`EntrySpacing`. The `EntrySpacing` option determines the {*horizontal*`,`*vertical*} separation, in printer's points, between successive columns and rows, respectively, of entries in the legend.

`RowLimit`. The `RowLimit` option determines the maximum number of rows of entries in the legend. If this number is exceeded, entries wrap to a new column.

16.3 Data manipulation utilities

Table 16.11 Data manipulation utility functions.

`MakeDataSet[`*expr*`,`*data*`]`
`MakeDataSet[`*expr*`,{`$data_1$`,`$data_2$`,...}]`
	Creates a new data set through row-by-row operations.

`AttachIndex[`*data*`]`
`AttachIndex[`*start*`,`*data*`]`
`AttachIndex[`*start*`,`*step*`,`*data*`]`
	Prepends a column, containing a running index, to *data*.

`SelectByColumn[`*data*`,`*c*`,`*patt*`]`	Selects those rows of *data* for which the value in column *c* matches the pattern (or value) *patt*.

`MakeDataSet` The `MakeDataSet` function, summarized in Table 16.11, returns a data set in which each row is obtained from corresponding row of either a single data set, in the form `MakeDataSet[`*expr*`,`*data*`]`, or multiple data sets, in the form `MakeDataSet[`*expr*`,{`$data_1$`,`$data_2$`,...}]`, by evaluating the given expression *expr*. All the data sets given as arguments to `MakeDataSet` in the latter form must have the same number of rows as each other. This expression should evaluate to a list, which becomes the rows of the new data set.

This expression may invoke various special expressions `DataEntry[`*col*`]`, `DataEntry[{`*row*`,`*col*`}]`, `DataEntry[`*set*`,`*col*`]`, or `DataEntry[`*set*`,{`*row*`,`*col*`}]` to refer to entries in the various data sets (the present row if *row* is omitted, and the first set if *set* is omitted). This expression may also invoke the symbol `Row`, which represents the row number, starting with 1. Thus, *e.g.*, the expression `DataEntry[`*set*`,`*col*`]` represents the value in column *col* of the current row of the *set*th data set in the argument list. If just a single data set is given to `MakeDataSet`, it makes more sense to use the short form `DataEntry[`*col*`]`, which represents the value in column *col* of the first (and, in this case, only) data set in the argument list.

The column number *col* may be given as in the usual form for an argument to `Part` (see **ref/Part**). A single positive integer simply represents the column number starting from 1 for the leftmost column. A single negative integer represents a column number counting backwards from -1 for the rightmost column. If the column is given as `All`, all the entries in the given row of the data set will be inserted at this point, in place of the `DataEntry`. A subrange of entries from the row may be specified using the Mathematica "span" notation for ranges of indices (see **ref/Span**), such as `2;;4` to insert the entries from the second through fourth columns.

A range of rows may be selected, for use in constructing the new data set, by giving `MakeDataSet` the option `Range`, which follows the same syntax as a range specification for the Mathematica function `Take` (see **ref/Take**), *e.g.*, `Range->{2,-1}` for all rows starting with the second.

Example: Suppose data set `DataSet1` contains rows

$x\ y_1\ y_2$

but for plotting we also wish to append a column containing the values y_2^2. That is, we need a data set with rows

$x\ y_1\ y_2\ y_2^2$

To construct such a data set, we would build each row by taking all the entries from the given row of `DataSet1` and appending the square of the last entry. This is accomplished with

```
NewDataSet=MakeDataSet[
  {DataEntry[All],DataEntry[3]^2},
  DataSet1
];
```

Example: Another typical example would be to take the "differences" of two data sets. For instance, one data set might contain the "theoretical" value and the other the "experimental" value, and we might wish to plot the difference between the two. Suppose data set `DataSet1` contains rows

$x\ y_1$

and `DataSet2` contains rows

$x\ y_2$

We are assuming that both data sets contain exactly the same number of rows and contain data for exactly the same x values, in exactly the same order. To construct a data set `DataSet3` with rows

$x\ y_1 - y_2$

we would build each row by copying the first entry from the given row of `DataSet1` (or, equivalently, `DataSet2`) and appending the difference of the second entries of the two data sets. This accomplished with

```
NewDataSet=DataOperation[
  {DataEntry[1,1],DataEntry[1,2]-DataEntry[2,2]},
  {DataSet1,DataSet2}
];
```

Example: For an example involving ranges of columns, suppose data set `DataSet1` contains rows

$x\ y_1\ z_1$

and `DataSet2` contains rows

$x\ y_2\ z_2$

To construct a data set `DataSet3` with rows

$x\ y_1\ z_1\ y_2\ z_2$

we would build each row by taking all the entries from the given row of `DataSet1` and appending all the entries from each row of `DataSet2` *except* the first. This accomplished with

```
NewDataSet=DataOperation[
  {DataEntry[1,All],DataEntry[2,2;;-1]},
  {DataSet1,DataSet2}
];
```

Or, instead of `DataEntry[2,2;;-1]`, for the "second through last" columns, we could equivalently use `DataEntry[2,2;;3]`, for the "second through third" columns, in this example.

`AttachIndex` The `AttachIndex` function, summarized in Table 16.11, takes data — which may either be a proper data set (matrix) or simply a list of values (vector) — and prepends a column, containing a running index. By default, for `AttachIndex[`*data*`]`, the index starts from 1 and increases in steps of 1, but alternate starts and steps may be specified as `AttachIndex[`*start*`,`*data*`]` or `AttachIndex[`*start*`,`*step*`,`*data*`]`.

`SelectByColumn` The `SelectByColumn` function, summarized in Table 16.11, is defined in the same spirit as the Mathematica `Cases` function. `SelectByColumn[`*data*`,`*c*`,`*patt*`]` selects those rows of *data* for which the value in column *c* matches the pattern (or value) *patt*.

Part III

Appendices

A Known issues

Here are some problems which have been noted by or reported to the author:

Self-test error. When using SciDraw, you may occasionally see a self-test error message pop up in the Mathematica "Messages" window:

```
INTERNAL SELF-TEST ERROR: CAPopup—c—1519
Click here to find out if this problem is known, and to help improve
Mathematica by reporting it to Wolfram Research.
```

The problem is apparently harmless and can be ignored. *(Reported under: Mathematica 9.0.1)*

These are the only ones of which the news has come to Notre Dame. There may be many others but they haven't been discovered.

B Revision notes

Version 0.0.0 (November 24, 2013): First public beta release.

Version 0.0.1 (December 20, 2013): New option `TickLengthScale`. New option `TextBaseBuffer`.

Version 0.0.2 (January 8, 2014): Documentation update.

Version 0.0.3 (February 20, 2014): Composite styles with `MakeStyle`.

Version 0.0.4 (October 6, 2014): CustomTicks updated (version 1.9). Introduction of callout line for text and options `TextCallout` and `TextCallout`*XXXX*. Extended functionality of `SetByPanelIndices`. New functions `ByPanelRow`, *etc.* Extended functionality of `SymbolOptionColumns`. New options `ColumnNames` and `SymbolSizeScale`. New predefined symbol shapes `"Horizontal"`, `"Vertical"`, and `"Asterisk"`. Extended `DataLegend` descriptor syntax {*style*, *text*, *symbol*}. `FigurePanel` accepts general pattern for panel indices in iterated generation of panels. `FigurePanel` option `ExteriorEdgeMask` deprecated in favor of *XY*`EdgeExterior`.

Version 0.0.5 (October 11, 2014): `FigBracket` supports oblique orientations. Fix issue with `DataPlot`'s error bars (introduced in 0.0.4).

Version 0.0.6 (December 21, 2014): `DataPlot` supports drop lines via `DataDrop` options. `FigLabel` supports callout line and new option `DisplacedPoint`. Earlier realization of text callouts via `TextCallout` option (Version 0.0.4) deprecated. Fix issue with `FigArrow`'s `TailLength` and `TailLip` options. New option `LockPanelScale` for `FigurePanel`. New option `Shift` for `FigBracket`. `FigCircle` supports syntax with region argument. Updates to anchor generation/retrieval (new functions `Anchor` and `GetAnchor`, direct access to `FigAnchor` deprecated), updates to point/anchor displacement functions (`RelativeTo` replaced by `DisplacePoint`, `AlongAnchor` replaced by `DisplaceAlongAnchor`, new function `ProjectPoint`), updates to curve point specifications (`FromHead`/`FromTail` replaced by `DisplaceHead`/`DisplaceTail` and `DisplaceAlongHead`/`DisplaceAlongTail`, new specifications `ProjectHead`/`ProjectTail`), and new function `RotateAnchor`. Documented text formatting constructs (`textit`, `MultipletLabel`, *etc.*).

Version 0.0.7 (March 28, 2015): Internal optimizations for performance improvements. Add curve anchor position specified by `Horizontal`/`Vertical`.

C Licenses

Different parts of the SciDraw package are distributed under different licenses:

> The *code* for SciDraw is distributed under the GNU Public License, Version 2. Here, the *code* refers to the contents of the directory `packages` in the SciDraw distribution.
>
> The *documentation* for SciDraw is distributed under the GNU Free Documentation License, Version 1.2. Here, the *documentation* refers to the contents of the directory `doc` in the SciDraw distribution.

The GNU Public License, Version 2

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and

disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.
11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

GNU Free Documentation License, Version 1.2, November 2002

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

Verbatim Copying

Copying in Quantity

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

Combining Documents

Collection of Documents

Aggregating with independent Works