LevelScheme user’s guide

M. A. Caprio, Department of Physics, University of Notre Dame

Version 3.53 (January 10, 2013)

I. Introduction

■ LevelScheme: A scientific figure preparation system

LevelScheme is a scientific figure preparation system for *Mathematica*. LevelScheme provides a general infrastructure for the preparation of publication-quality figures, combining technical drawings or diagrams, mathematical plots, data plots, and annotations. It features extensive support for multipanel and inset plotting, customizable tick mark generation, diagram construction, and labeling.

LevelScheme originated as a tool for drawing level schemes, or level energy diagrams, as used in nuclear, atomic, molecular, and hadronic physics. LevelScheme includes a full suite of drawing tools for the construction of such diagrams. LevelScheme 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. LevelScheme allows extensive manual fine tuning of the drawing appearance, text formatting, and object positioning. It also includes specialized features for creating several common types of level schemes encountered in nuclear physics. The full power of Mathematica's programming language may be used in constructing the figure contents, so, for instance, level energies and transition properties shown in the diagram can be directly computed from models or input from data files.

■ Preliminary comments

A few basic principles have guided the design of the LevelScheme package. One is to have a system whereby even major formatting changes to a figure can be made relatively quickly. Objects in a level scheme are attached to each other (transitions attached to levels, labels attached to levels and to transitions, *etc.*), so that if one object is moved the rest follow automatically. Another principle is for objects to have reasonable default properties, so that an unsophisticated level scheme can 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 ("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. 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.

It is assumed that the reader of this guide has some basic experience starting *Mathematica*, evaluating cells, and opening and saving notebook files. The reader would also benefit from having used the *Mathematica* `Plot` command to generate some basic graphics. For instance, you should read about plotting and graphics in *The Mathematica Book* (which you can find under the "Help" menu listed as "Virtual Book").

■ Notation and conventions

Many dimensions (such as line thicknesses or text position adjustments) will be specified in "printer's points", where 1 pt = 1/72 inch ≈ 0.35 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 ~10 pt high.

The *Mathematica* option symbol ("→") which appears throughout this guide is entered from the keyboard as a hyphen followed by a greater-than sign ("`->`").

■ Further information

Updates to LevelScheme, and further documentation, may be obtained through the LevelScheme home page:

`http://scidraw.nd.edu/levelscheme`

The LevelScheme package has been published in Computer Physics Communications [M. A. Caprio, Comput. Phys. Commun. **171**, 107 (2005)]. This article is also available as the arXiv electronic preprint `arXiv:-physics/0505065`.

■ Acknowledgement of use

If you use LevelScheme 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 LevelScheme scientific figure preparation system [M. A. Caprio, Comput. Phys. Commun. **171**, 107 (2005), http://scidraw.nd.edu/levelscheme].

Feel free to modify this statement as appropriate, *e.g.*, changing "the figures for this article" to "Figure 5". (**Note:** Acknowledging LevelScheme in *individual figure captions* is *not* recommended, since a full acknowledgement is cumbersome there, and simply referencing the Computer Physics Communications paper in a caption could be mistaken to mean that the figure data is taken from that paper.)

II. Installation

This version of LevelScheme is for use with Mathemaica 6 and higher (and has been tested under Mathematica 8 and 9). If you are still using Mathematica 5 or below, please visit the LevelScheme home page to download the appropriate legacy version of LevelScheme.

■ Installing the files

The package is distributed as a ZIP file. First, you need to decompress this ZIP file. This will produce a directory named `LevelScheme`, which will contains several subdirectories (`LevelScheme/doc`, `LevelScheme/CustomTicks`, `LevelScheme/LevelScheme`, *etc.*).

Then, you need to decide upon a suitable place in your directory structure where you would like to keep these files. For example, on a Windows machine, you might have already created a directory named

```
C:\Research\MathematicaStuff
```

to contain all your *Mathematica* packages, documentation, *etc.* In this case, you would move the `LevelScheme` directory (containing all those subdirectories) here, and it would be called

```
C:\Research\MathematicaStuff\LevelScheme
```

(This directory will then contain contain subdirectories `C:\Research\MathematicaStuff\LevelScheme\doc`, `C:\Research\MathematicaStuff\LevelScheme\CustomTicks`, `C:\Research\MathematicaStuff\LevelScheme\LevelScheme`, *etc.*)

However, *Mathematica* must still be told that this directory is a place where it should look in order to find package files. *Mathematica* only searches for package files in the directories listed in Mathematica's variable `$Path`. Therefore, you need to add the LevelScheme directory to this list, by using the `AppendTo` command. For instance, using the example directory names from above, you would need to evaluate the command

```
AppendTo[$Path, "C:\\Research\\MathematicaStuff\\LevelScheme"];
```

Please note that the LevelScheme directory needs to be added to `$Path` *before* you try to load LevelScheme with `Get[“LevelScheme`”]` (see below for more on loading the package). Moreover, each time *Mathematica* is restarted, the `$Path` variable goes back to its “factory default” value. Therefore, the LevelScheme directory needs to be added to `$Path` *each time* you restart *Mathematica*. There are two simple solutions, and you may choose either one:

1) You can include this `AppendTo` command in each notebook just before `Get[“LevelScheme`”].`

2)You can include the `AppendTo` command in your personal `init.m` startup file. You can edit your startup file by going to the directory given by `$UserBaseDirectory`, looking in the subdirectory named `Kernel`, and opening the file named `init.m`. See the *Mathematica* help on “init.m” for more explanation.

Caution: On Windows systems, beware of the treatment of the backslash character “\”. Directory names are separated by “\” under Windows. However, “\” has a special meaning in *Mathematica* input (this will be familiar to C language programmers as well) and so must be entered as a double backslash “\\”. Alternatively, you may use a forward slash in the input, as “/”, and this will be perfectly acceptable to Windows. Under Unix/Linux operating systems, a forward slash should always be used.

Note: You can learn more about file names and file organization in *Mathematica* by searching for “tutorial/MathematicaFileOrganization” and “tutorial/NamingAndFindingFiles” in the help browser. If you would like to learn more about the package search path, search for “$Path”.

■ Loading the package

Each time you restart *Mathematica*, if you want to use the LevelScheme package's plotting commands, you must tell *Mathematica* to load the LevelScheme package, by entering the following command:

```
Get["LevelScheme`"]
```

You should see something like the following as output:

```
LevelScheme scientific figure preparation system
M. A. Caprio, Department of Physics, University of Notre Dame
Comput. Phys. Commun. 171, 107 (2005)
Version 3. xx
```

View color palette | Visit home page

Caution: You must be sure to always load the package *before* you first try to use any of the LevelScheme commands. If you ever accidentally try to use any of the LevelScheme commands before loading the package, the package will not be able to run properly for the rest of your *Mathematica* session. Instead, you will see error messages such as

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

This is an inconvenience common to all *Mathematica* packages. If you encounter this situation, just exit and restart *Mathematica* (or quit the kernel), and try again.

Note: If you would like to learn more about packages, see the *Mathematica* help. You will want to search for "tutorial/MathematicaPackages".

Note: If you prefer, you can equivalently enter `<<"LevelScheme`"`. This is equivalent to `Get["LevelScheme`"]`.

Documentation and examples

The directory `LevelScheme/doc` contains several documentation and example files:

1. This user’s guide (`LevelSchemeGuide.pdf).`
2. A separate, smaller guide with details on customizing tick marks and tick labels for your axes (`CustomTicksGuide.pdf`).
3. Several notebooks containing example figures, with the code used to generate them (`Examples-Schemes.nb`, `Examples-Diagrams.nb`, `Examples-FunctionPlots.nb`, and `Examples-DataPlots.nb`).
4. A preprint of the Computer Physics Communications paper describing LevelScheme (`physics_0505065v2.pdf`).

III. Basics

Creating a figure

A figure is drawn by giving a list of "objects" (such as energy levels, arrows, shapes, text labels, or data plots), as the argument to the command `Figure`. The actual syntax for specifying these objects is the subject of most of the rest of this guide.

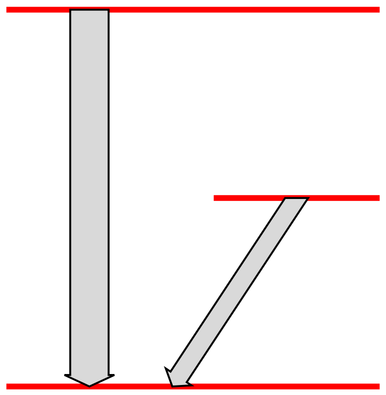
`Figure[{` *object1*, *object2*, … `}]`	Constructs and displays a figure

The basic figure display command.

The positions of objects in the figure are specified in terms of an *x*-*y* coordinate system. For a level scheme, the meaning of the *x* coordinate is usually somewhat arbitrary (it is just used to control left-right positioning), while the *y* coordinate usually represents energy. The `Figure` command must be told the horizontal and vertical range of coordinate space to be displayed, using an option `PlotRange→{{`*xmin*`,`*xmax*`},{`*ymin*`,`*ymax*`}}`. The desired size of the drawing on the page, in printer's points, can be specfied with the `ImageSize` option.

An example of a simple figure follows.

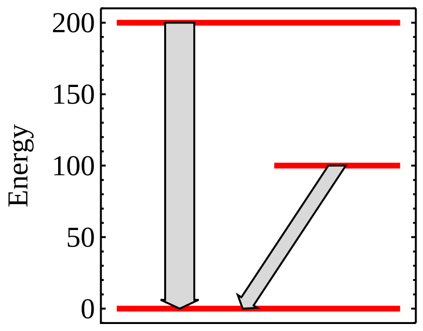
```
Figure[
    {
      SetOptions[Lev,Thickness→3,Color→Red],
      SetOptions[Trans,ArrowType→ShapeArrow,FillColor→LightGray],
      Lev[lev1,0,2,0],
      Lev[lev2,1,2,100],
      Lev[lev3,0,2,200],
      Trans[lev2,0.5,lev1,0.9,Width→10],
      Trans[lev3,0.5,lev1,0.5,Width→20]
      },
    PlotRange→{{0,2},{-10,210}}, ImageSize→72*{3,3}
    ]
```



Caution: You might be tempted to try to cut and paste this example (or any of the following examples) from the PDF file version of the user guide directly into a *Mathematica* notebook. However, this will probably not work. Since some of the special characters (the arrows) do not copy properly. Instead, you can retype the example from scratch. Or, you can find this and other examples in the *example notebooks provided with the package*.

The `Figure` command accepts many additional options controlling the appearance of the drawing. Most of these are identical to options accepted by the usual *Mathematica* plotting and display functions such as `Plot` and `Show`. A complete summary will be given in a later section. Here we just illustrate how a "frame" can be drawn about the figure using some of these options.

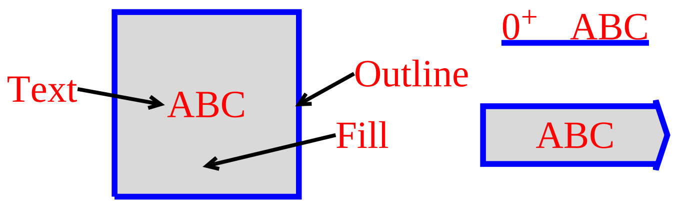
```
Frame→True,FrameTicks→{None,Automatic},LabL→"Energy",FontSize→15
```



The same figure as above but with a frame.

■ General drawing principles

Each "object" in the figure is built from up to three distinct parts: an outline, a filled area, and attached text labels, as illustrated in the following color-coded diagram. Not all objects have all these parts; for instance, a level has no filled area.



Decomposition of each object into outline, fill, and text.

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. Following is a summary of all the drawing options.

option name	default value	
`Color`	`Black`	Default color used for all parts of object, unless overridden by `LineColor`, `FillColor`, or `FontColor`
`Show`	`True`	Whether or not to draw the object

Options affecting appearance of all parts of an object.

option name	default value	
`ShowLine`	`True`	Whether or not to draw the outline
`LineColor`	`Automatic`	Color for outline; if `Automatic`, value specified by `Color` is used instead
`Thickness`	`1`	Thickness of line in printer's points
`Dashing`	`None`	Line dashing style; may be `None` for no dashing, `Automatic` for default dash lengths, a numerical length *length* in printer's points, a series of dash lengths {*length1*, *length2*, …}, or a *Mathematica* `AbsoluteDashing` directive

Options affecting appearance of just the outline.

option name	default value	
`ShowFill`	`True`	Whether or not to draw the fill
`FillColor`	`Automatic`	Color for fill; if `Automatic`, value specified by `Color` is used instead

Options affecting appearance of just the fill.

option name	default value	
`FontFamily`	`“Times New Roman”`	Font family, which may be any font installed on the system
`FontSize`	`12`	Character height in printer' s points
`FontColor`	`Automatic`	Color for text; if `Automatic`, value specified by `Color` is used instead
`FontWeight`	`“Plain”`	Font weight; `“Bold”` gives boldface
`FontSlant`	`“Plain”`	Font slant; `“Italic”` gives italic
`FontTracking`	`“Plain”`	Horizontal spacing between letters
`BackgroundFontSi-zeFactor`	`1.0`	Controls extent of optional whited-out background region when `Background` *X* option is set (see below)

Options affecting font characteristics for text.

option name	default value	
`Layer`	`Automatic`	Controls whether object is drawn in front of or behind others (see discussion in later section)
`ClipToRectangle`	`True`	Controls clipping of graphics to the current plotting region (see discussion in later section)

Advanced drawing control options.

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. Colors are specified using either the color names from the *Mathematica* Colors package or any of the *Mathematica* color directives (see the *Mathematica* documentation for `GrayLevel` or `RGBColor`). When LevelScheme 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.

The line thickness and dashing are controlled by the options `Thickness` and `Dashing`. To turn on dashing with default dash lengths, simply set `Dashing→Automatic`. Otherwise, specify dash lengths as described in the table above. For instance, `Dashing→{6,2}` produces a dash-dot pattern. For more advanced comments on fine-tuning the appearance of dashing, see Appendix B.

It is possible to specify that the outline or fill of the object not actually be drawn. 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.

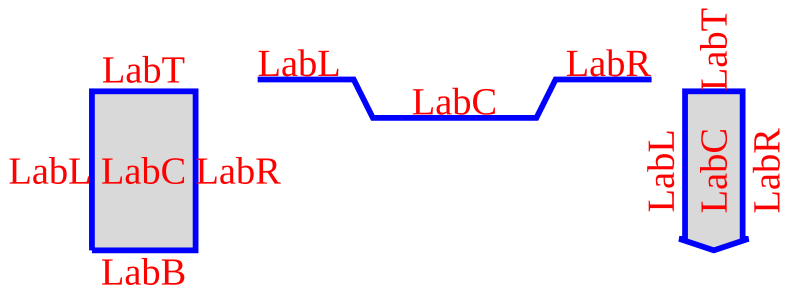
The font style options are the standard options for text formatting in *Mathematica*, discussed in more detail in the *Mathematica* documentation. The default font used by the LevelScheme package is Times, but usually any font installed on the system can be used.

The advanced options `Layer` and `ClipToRectangle` are discussed in later sections.

option name	default value	
`Lab` *X*	`None`	Contents of label *X*
`ShowLab` *X*	`True`	Whether or not to draw label *X*
`Orientation` *X*	`Automatic`	Orientation of label *X*; may be set to `Horizontal`, `Vertical`, `Automatic` to align text along object (which is the same as `Horizontal` except for labels on arrows or axes), `Inverted` to obtain text 180 ° rotated from `Automatic`, or an arbitrary *angle* (see discussion)
`Offset` *X*	`Automatic`	Position of label *X* with respect to its anchor point (see discussion); specified as { *xoffset*, *yoffset* }; `Automatic` yields a reasonable default position
`Nudge` *X*	`0`	Horizontal and vertical fine-tuning adjustment of label position; specified in printer' s points as *ynudge* for just a vertical adjustment or as { *xnudge*, *ynudge*}
`Background` *X*	`None`	Specifies existence and color of whited-out area behind label; `Automatic` uses the background color of the drawing

Options for individual labels. Substitute the appropriate label position letter for *X*.

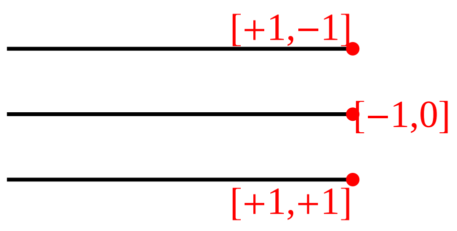
A single object can have one or more labels attached, depending upon the type of object. Each of these has a "name" indicating its position: "L" on the left, "R" on the right, "T" at the top (or tail, in the case of an arrow), "B" at the bottom, or "C" in the center. The contents of these label are set using the options `LabL`, `LabR`, *etc.*, according to this naming scheme. The label names are illustrated for a few types of object in the following diagram. There are several options which control the contents and positioning for each label individually.



Names of labels, indicating their positions.

The most important of these options is `Offset`*X*, which controls where the label actually lies relative to its nominal position. For instance, the "right-hand" label for a level, which is nominally positioned at the right-most point of the level line, actually is typically drawn either *above* the right end of the level, *outside* the right end of the level, or *below* the right end of the level. The right endpoint of the level serves as an "anchor" point (the red dot in the following diagram) for the label. Then the option `OffsetR`→`{`*xoff*`,` *yoff* `}` specifies where this "anchor" point lies within an imaginary box circumscribing the text, with {-1,-1} as the lower left corner of the text and {+1,+1} as the upper right corner. This is based upon the "offset" notation used in the basic *Mathematica*

`Text` command. The following illustrates where the text lies for some example offsets.



The anchor point offset relative to a text label.

The position of the text can further be fine-tuned with the `Nudge`*X* option, which nudges the label horizontally and vertically by the specified number of printer's points.

The orientation can be set to any arbitrary angle with the `Orientation`*X* option. The angle is measured counterclockwise from horizontal in radians; to specify it in degrees, as is usually more convenient, multiply by the conversion factor `Degree`, *e.g.*, `45*Degree`. Preset orientations, including `Horizontal` and `Vertical` (see option description above), are also available.

The option `Background`*X* is used to create a whited-out area behind the label, hiding anything else in the drawing behind the label.

■ Setting default values for options

Sometimes an option's value only needs to be set for a single object. But very often it is desirable to to change the default value of an option for all objects of a given type in the entire figure. This is accomplished using the *Mathematica* `SetOptions` command, which takes the form `SetOptions[`*objecttype*`,`*option*→*value*`,`…`]`, where *objecttype* is one of the types of drawing object used in the figure. Some examples would be

```
SetOptions[Lev,Thickness→3,Color→Red,FontSize→20]
SetOptions[Trans,Thickness→2,Dashing→Automatic]
```

For several options which control the basic drawing style, described at the beginning of preceding section, you can also specify a global default value. This allows you to make a uniform style change to your whole drawing. To set the global defaults, specify the options for `SchemeObject`. For example,

```
SetOptions[SchemeObject,Thickness→2,FontFamily->"Helvetica",FontSize→15]
```

These global default values are used by all types of objects, unless overridden by a `SetOptions` for that particular type of object.

If you use the `SetOptions` command *outside* a `Figure` definition, the new values for the options will apply to all figures for the rest of the *Mathematica* session; whereas, if you use the `SetOptions` command *inside* a `Figure` definition, the new values apply only to that one figure. It is usually preferable to adopt a practice of setting option values *inside* each individual figure, to avoid unwanted collateral effects on other figures.

`SetOptions` affects only those objects defined *following* it in the figure, so you can change style midway through a figure. The default value of an option set with `SetOptions` can always be overridden if needed for individual objects. The following provides an example of the use of `SetOptions` several times within one figure, in this case to choose a different color for the labels on each band of levels, and an example of overriding the default options for a single object, here the level with the black label.

```
SetOptions[Lev,LabR→"ABC",FontSize→20],
SetOptions[Lev,FontColor→Red],
Lev[Dummy,0,1,0],Lev[Dummy,0,1,50],Lev[Dummy,0,1,100],Lev[Dummy,0,1,150],

SetOptions[Lev,FontColor→Green],
Lev[Dummy,1,2,0],Lev[Dummy,1,2,50,FontColor→Black],Lev[Dummy,1,2,100],Lev[Dummy,1,2,150],

SetOptions[Lev,FontColor→Blue],
Lev[Dummy,2,3,0],Lev[Dummy,2,3,50],Lev[Dummy,2,3,100],Lev[Dummy,2,3,150]
```

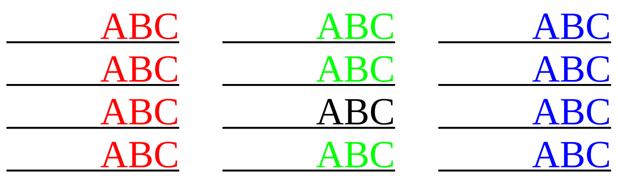


Illustration of the use of SetOptions several times within one figure.

IV. Drawing objects

■ Levels, extensions, and connectors

Levels are drawn with the command `Lev`. There are also auxiliary commands for drawing extension lines and connectors, discussed later in this section.

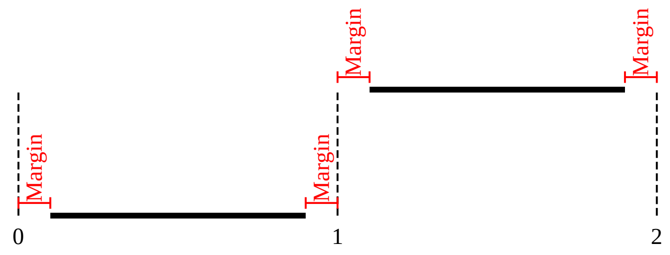
`Lev[` *name*, *x1*, *x2*, *energy*`]`	Energy level

Level drawing command.

option name	default value	
`Margin`	`0.1`	Horizontal inset of each end of the level from the left and right coordinates specified, in abscissa coordinate units
`WingHeight`	`0`	Elevation of gull wings relative to central segment of level, in printer' s points
`WingRiseWidth`	`10`	Width of sloped segment of gull wings, in printer' s points; can be specified as {*lwidth*, *rwidth*} for asymmetric left and right wings
`WingTipWidth`	`30`	Width of flat segment of gull wings, in printer' s points; can be specified as {*lwidth*, *rwidth*} for asymmetric left and right wings
`MakeWingL`	`True`	Whether or not to make gull wing on left side
`MakeWingR`	`True`	Whether or not to make gull wing on right side

Options for level drawing command.

Each specification of a level with `Lev` includes a name (or ID), left and right coordinate, and energy coordinate. The name can be any symbol of the user's choosing. The name does not affect the drawing of the level itself. Rather, it is used later to refer back to the level, when drawing extension lines, transitions, *etc.*, which connect to the level. The actual left and right end points of the level are indented from the nominal left and right end coordinates specified, by an amount controlled by the option `Margin`. 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.



The left and right end points of a level are indented by an adjustable margin.

Levels can have left, center, and right labels. Specifying the special option value `Lab`*X*`→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,Lab`*X*`→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, 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 use the string verbatim in automatic energy labels.

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, giving levels which appear to have "gull wings". These can be created by specifying a nonzero value for the option `WingHeight`, postive for elevated wings and negative for lowered wings. Automatic energy labels and gull wings are both illustrated in the following example. The dimensions of the gull wings can be customized using the options `WingRiseWidth`, `WingTipWidth`, `MakeWingL`, and `MakeWingR`.

```
SetOptions[Lev,Thickness→3,LabR→Automatic],
Lev[lev0,0,1,"0.0"],
Lev[lev100,0,1,"100.1",WingHeight→-5],
Lev[lev105,0,1,"105.3",WingHeight→+5]
```

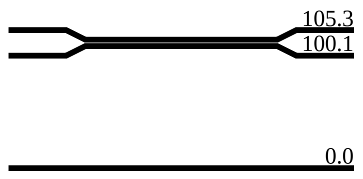


Illustration of automatic energy labels and gull wings.

`ExtensionLine[` *level*, *side*, *length*`]`	Extension line to left or right of level; *side* may be `Left` or `Right`
`ExtensionLine[` *level*, *posn1*, *posn2*`]`	Extension line with arbitrary starting and ending positions relative to left end coordinate of level
`Connector[` *level1*, *level2*`]`	Connector line from right end of *level1* to left end of *level2*

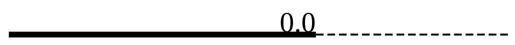
Level extension line and connector drawing commands.

option name	default value	
`ToWing`	`True`	Controls whether an extension line appears at the same vertical coordinate as the gull wing (if present) or as the main part of the level

Option for extension line command.

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

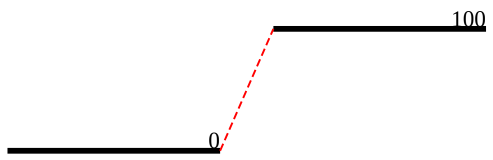
```
SetOptions[Lev,Thickness→3,LabR→Automatic,FontSize→15],
SetOptions[ExtensionLine,Thickness→1,Dashing->Automatic],
Lev[lev0,0,1,"0.0"],
ExtensionLine[lev0,Right,0.5,Dashing→Automatic]
```



Creation of an extension line.

Connector lines between levels are drawn with the command `Connector`. A simple example follows.

```
SetOptions[Connector,Dash→True,Color->Red],
Connector[lev0,lev100]
```



Creation of a connector line.

■ Arrows and transition arrows

Arrows are drawn either with the command `SchemeArrow` or `Trans`. `SchemeArrow` is meant for general-purpose use, *e.g.*, to draw arrows in technical diagrams or to annotate a figure. `Trans` is meant for drawing transition arrows in level schemes.

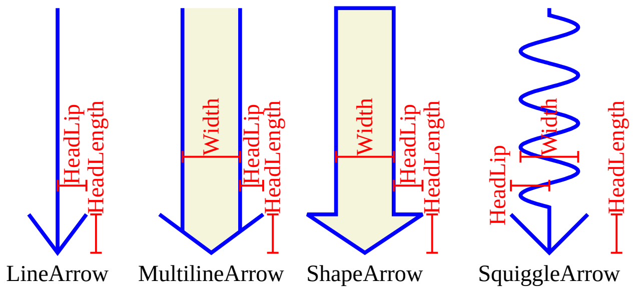
`SchemeArrow[` *point1*`,` *point2*`]`	Arrow from *point1* to *point2*
`SchemeArrow[{` *point1*`,` …`,` *pointn* `}]`	Polygonal arrow with several segments, connecting the points indicated

Basic arrow drawing command.

option name	default value	
`ArrowType`	`LineArrow`	Arrow shape type; value can be `LineArrow`, `MultilineArrow`, `ShapeArrow`, or `SquiggleArrow`
`HeadLength`	`9`	Length of arrowhead, in printer' s points
`HeadLip`	`3`	Half-width or extension of arrowhead outward from arrow shaft, in printer' s points
`Width`	`5`	Width of arrow shaft, in printer' s points (for `MultilineArrow`, `ShapeArrow`, or `SquiggleArrow` only)
`ShaftLines`	`2`	Number of parallel lines in shaft (for `MultilineArrow` only)
`ShowTail`	`False`	Controls whether or not an arrowhead appears at the tail of the arrow (for `LineArrow` or `SquiggleArrow` only)
`ShowHead`	`True`	Controls whether or not an arrowhead appears at the head of the arrow (for `LineArrow` or `SquiggleArrow` only)
`TailBevel`	`False`	For a non-vertical arrow, controls whether or not the tail of the arrow is sliced off to make it horizontal (for `MultilineArrow` or `ShapeArrow` only)

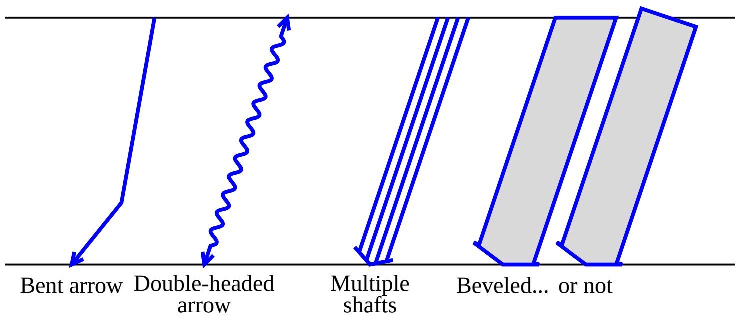
Options controlling the arrow shape.

Arrows can be drawn in four different styles, selected by the option `ArrowType`. An arrow of type `LineArrow` has an arrowhead constructed from two line segments, the lengths and angles of which are customizable. An arrow of type `MultilineArrow` is similar but has two or more 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, `MultilineArrow` is almost always used with either a separate `FillColor` option or with `ShowFill→False`. An arrow of type `ShapeArrow` is drawn as a polygon with both an outline and fill. An arrow of type `SquiggleArrow` has a sinusoidal squiggle for its shaft. These styles and the options controlling the arrow dimension parameters for each are illustrated in the following.



The three available arrow styles and their dimension parameters.

Some of the possible variations are shown below. A `LineArrow` can have multiple segments, specified by giving a list of points `SchemeArrow[{`*point1*`, …, `*pointn*`}]`. A `LineArrow` or `SquiggleArrow` can be "double headed" or even have an arrowhead only on its tail, as controlled by the `ShowHead` and `ShowTail` options. The number of tail shafts for a `MultilineArrow` can be controlled with the `ShaftLines` option. The tail of a `MultilineArrow` or `ShapeArrow` can be "beveled" so that it is horizontal, by setting `TailBevel→True` (this is more commonly used with transition arrows, described below). These possibilities are illustrated in the following.

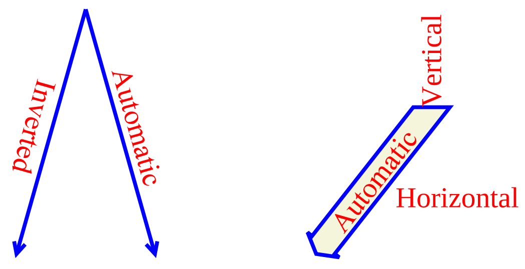


Variant arrow shapes.

option name	default value	
`Posn` *X*	`Automatic`	Position of label *X* along arrow shaft; value is specified as *fraction* of distance from tail to head, or as distance `FromTail[` *dist* `]`, `FromHead[` *dist* `]`, `FromTailVertical[` *dist* `]`, `FromHeadVertical[` *dist* `]`, `FromTailHorizontal[` *dist* `]`, or `FromHeadHorizontal[` *dist* `]` in printer' s points (for left, center, and right labels only)
`Buffer` *X*	`Automatic`	Buffer spacing between label *X* and arrow shaft, as multiple of current font height (for left and right labels only)
`Segment` *X*	`Automatic`	For an arrow with multiple segments, specifies which segment label *X* is attached to (for left, center, and right labels only); may be positive integer `1` through *n* where *n* is the number of segments, to specify segment counting from tail, or negative integer to count back from head

Options controlling label placement for transition arrows. These complement the usual LevelScheme label positioning options.

Arrows can have left, center, right, and tail labels. (The nominal "left" and "right" labels are only actually properly named if the arrow is pointing downward, as usual in level schemes.) If the `Orientation`*X* 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. If a label "upside down" relative to this angling is prefered, as occasionally might be for near-vertical arrows, the option can be overriden with the value `Inverted`. Ordinary horizontal or vertical labels can be specified, as usual, with the `Horizontal` and `Vertical` option values.

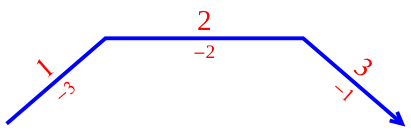


Comparison of angles for arrow labels.

The position of each label along the arrow shaft is controlled with the option `Posn`*X* . 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 tail label is by default at a position of 0, and the other labels are by default at 0.5, the midpoint of the arrow. More sophisticated positioning specifications, in terms of distances in printer's points from the tail or head, are available as well. These are summarized in the option table above. For instance, when several different transition arrows originate from the same level, it may be desirable to have their labels all aligned at the same height as each other. If these transition arrows are of different lengths and have different orientations, it would not be easy to specify this alignment simply in terms of the fractional position along the arrow shaft. Instead, the option value `Posn`*X*`→FromTailVertical[`*dist*`]` can be used. The labels are then positioned the specified distance *vertically* down from the tail, regardless of how far horizontally and thus how for along the shaft this means they must go.

Developing a consistent system for setting the exact label positions, and for providing reasonable default positions, proved to be challenging, since the arrows can have arbitrary orientation and the labels themselves can have various orientations. For vertical arrows with purely horizontal or vertical labels, the usual `Offset`*X* system works well, since, *e.g.*, the arrow shaft should in this case always be at the far left end of the right-hand label or at the center bottom of the tail label. But the offset system fails miserably for angled arrows with angled text. Instead, it is much easier to note that, for text aligned *along* the arrow, the *center* of the text should always be a distance of about half a character height perpendicularly outward from the nearest side of the arrow shaft. The `Offset`*X* option for the left and labels has consequently been supplemented with a `Buffer`*X* option which controls the perpendicular distance out to the label's anchor point as a multiple of *half* the current font height, as determined by `FontSize`. If `Offset`*X* and `Buffer`*X* are set to `Automatic`, the actual values are chosen "intelligently" based upon the label orientation. This hybrid system of positioning is in practice not very complicated to use, since the default values usually produce decent results, and adjustments can be carried out with a little experimentation.

For arrows with more than one segment, the left, center, and right labels may appear on any of the segments, as specified by the option `Segment`*X*. 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.



`Trans[` *level1*, *posn1*, *level2*, *posn2* `]`	Transition arrow from *level1* to *level2*, with horizontal starting and ending positions explicitly specified
`Trans[` *level1*, *level2* `]`	Transition arrow from *level1* to *level2*, abbreviated form

Transition arrow drawing command. Points may also be specified explicitly, as for `SchemeArrow`.

Transition arrows in level schemes are drawn using `Trans`. Rather than starting and ending *points*, starting and ending *levels* must be specified.

The arrow drawn by `Trans` is identical to the arrow drawn by `SchemeArrow`, except that it uses the option values defined for `Trans`. This is useful if some arrows in a level scheme represent transitions while others are annotations, since it allows the stylistic options for one type to be set without interfering with those for the other type. The default options for `Trans` are initially the same as for `SchemeArrow`, except that `TailBevel` is by default turned on.

option name	default value	
`EndPositions`	`{0.5, 0.5}`	Arrow endpoint horizontal positions used by abbreviated form `Trans[` *level1*`,` *level2*`]`
`FromWing`	`False`	Controls whether tail of arrow is at height of gull wing or main part of initial level
`ToWing`	`False`	Controls whether head of arrow is at height of gull wing or main part of final level

Options controlling the positions of transition arrow endpoints.

The command `Trans[`*level1*`,`*posn1*`,`*level2*`,`*posn2*`]` draws a transition arrow starting a horizontal distance *posn1* from the left end of *level1* and ending a horizontal distance *posn2* from the left end of *level2*. 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`. If either *posn1* or *posn2* is specified as `Automatic`, the arrow is made vertical, 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. Without the `Automatic` value, a new value for *posn1* or *posn2* would have to be entered manually each time the left end of either level moved.)

The abbreviated form `Trans[`*level1*`,`*level2*`]` takes its starting and ending positions from the option `EndPositions`. 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`, and it will apply to all the transitions.

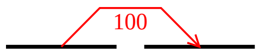
The alternate forms `Trans[`*point1*`,`*point2*`]` or `Trans[{`*point1*`,`…`,`*pointn*`}]` allow an arrow to be drawn freehand between the specified points, where no levels exist, exactly as for `SchemeArrow`. Note, however, that to draw an arrow with one end on a level and the other end "dangling", it is often more convenient to define an invisible "phantom" level, by calling `Lev` with the option setting `Show→False`, and then to use the command `Trans` to draw an arrow between the visible level and the phantom level.

option name	default value	
`Kink`	`None`	Coordinates of kink points for bent arrow (for `LineArrow` only); value may be a single point or a list `{`*point1*`,` …`,` *pointn*`}`; value `None` indicates no kink

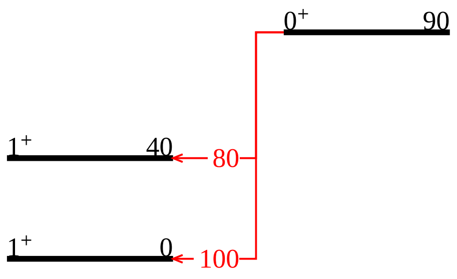
Option for specification of intermadiate points in transition arrow.

Often it is necessary to introduce one or more "kinks" into a transition arrow, *i.e.*, make a multi-segment arrow. The first and last points of the arrow are specified as usual by giving the level IDs, while the intermediate points are specified with the option `Kink`. The value given for `Kink` my be either a single point or a list of several points. Each point may be specified simply as a coordinate pair `{`*x*`,`*y*`}` in ordinary user coordinates. Alternatively, each may be specified as a position *relative* to the head or tail of the arrow, in user coordinates or printer's points, as `FromHead[{`*x*`,`*y*`}]`, `FromTail[{`*x*`,`*y*`}]`, `FromHead[Point[{`*x*`,`*y*`}]]`, or `FromTail[Point[{`*x*`,`*y*`}]]`. Note that the `FromHead` or `FromTail` notations may also be used for the intermediate points in `SchemeArrow[{`*point1*`,`…`,`*pointn*`}]` or `Trans[{`*point1*`,`…`,`*pointn*`}]`. Some examples of kinked transition arrows follow.

```
Trans[
  lev1,lev2,
  Kink→{FromTail[Point[{20,20}]],FromHead[Point[{-20,20}]]},SegmentL→2,
  LabL→100
  ],
```



```
SetOptions[Lev,Margin→0.2],
SetOptions[Trans,
  EndPositions→{0.2,0.8},Kink→{FromTail[{-0.10,0}],FromHead[{+0.30,0}]},
  HeadLength→5,HeadLip→2,
  BackgroundC→Automatic,OffsetC→{+1,0},PosnC→0.2,
  Color→Red
  ],
...
Trans[p90,d0,LabC→100],
Trans[p90,d40,LabC→80],
```



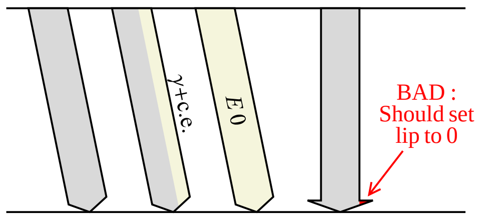
option name	default value	
`ConversionColor`	`White`	Color for conversion electron part of arrow.
`ConversionSide`	`Right`	Side on which conversion shading appear; value can be `Left` or `Right`
`ConversionCoeff`	`None`	Conversion coefficient, which is the ratio of the width of the conversion electron shaded part of the arrow to the width of the non–shaded part; value may be `None`, or a nonnegative real number, or `Infinity` for a fully–converted transition

Options controlling split shading for combined gamma ray and conversion electron transition arrows. These are used for `ShapeArrow` only.

Gamma-ray transitions with a conversion electron component are traditionally indicated by an arrow shaded with two different colors. Such an arrow may be drawn as a `ShapeArrow` with the option `Conversion-Coeff`. The appearance of the conversion electron shading can be controlled with the options `Conversion-Color` and `ConversionSide`. It is strongly recommended to set `HeadLip→0` for these arrows, since otherwise even a transition with zero or small conversion coefficient will have a big shaded "corner" in its arrowhead.

```
Figure[{

  SetOptions[Trans, ArrowType → ShapeArrow, HeadLip → 0,
   Width → 20, FillColor → LightGray, ConversionColor → LightBeige],
  Lev[lev1, 0, 1.3, 0],
  Lev[lev2, 0, 1.3, 1],
  Trans[lev2, 0.2, lev1, 0.3, ConversionCoeff → 0, HeadLip → 0],
  Trans[lev2, 0.4, lev1, 0.5, ConversionCoeff → 0.5, HeadLip → 0, LabR -> "γ+c.e."],
  Trans[lev2, 0.6, lev1, 0.7, ConversionCoeff → Infinity,
   HeadLip → 0, LabC → RowBox[{textit["E"], "0"}]],
  Trans[lev2, 0.9, lev1, 0.9, ConversionCoeff → 0, HeadLip → 7, ConversionColor → Red],
  SchemeArrow[{1.05, 0.3}, {0.9, 0} + LevelScheme`Coord`DPCOfPA[{10 + 5, 9}], Color → Red,
   OrientationT → Horizontal, OffsetT → {-0.5, -1},
   LabT →
    StackText[Center, 0, {RowBox[{"BAD", hspace[-0.5], ":"}], "Should set", "lip to 0"}]]

 },
 PlotRange → {{0, 1.3}, {0, 1}}, ImageSize → 72 * {4, 1.5}, ExtendRange -> 0.01
]
```

```
Trans::conversionlip :
  For conversion–electron arrows, it is recommended that HeadLip be set to 0, yielding a cornerless arrowhead.  Otherwise, a
     misleading plot is obtained, since the corner of the arrowhead is still shaded even for vanishing conversion coefficient.
```



option name	default value	
`SquiggleWavelength`	`10`	Wavelength of sinusoid in printer's points
`SquiggleBuffer`	`2`	Minimum length of straight arrow shaft at either end of sinusoid, before arrowhead or end of arrow
`SquiggleSide`	`Right`	Side on which sinusoid starts
`PlotPoints`	`32`	Number of plotting points along curve per wavelength

Options controlling squiggle properties.

The wavelength of a `SquiggleArrow` 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`.

■ General drawing shapes

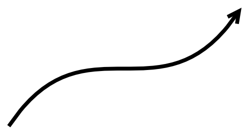
The remaining drawing commands produce general-purpose shapes, not special to level schemes. They are essentially enhanced versions of the *Mathematica* shape drawing primatives, but with outline, fill, and labels combined in one object. Their ease of use, with the machinery set up for controling their appearance through options, makes them useful for many diagramming, drawing, and plotting tasks.

`SchemeLine[{`*point1*`,` *point2*`, … }]`	Line
`SchemePolygon[{`*point1*`,` *point2*`, … }]`	Polygon, with outline and fill
`SchemeBox[{{`*x1*`,` *x2*`}, {`*y1*`,` *y2*`}}]`, `SchemeBox[{`*x1*`,` *y1*`}, {`*x2*`,` *y2*`}]`, `SchemeSquare[{`*x*`,` *y*`},` *radii*`]`	Rectangular box with outline and fill
`SchemeCircle[{`*x*`,` *y*`},` *radii*`]`	Circle or ellipse, as specified by *radii*, with outline and fill
`SchemeCircle[{`*x*`,` *y*`},` *radii*`, {`*theta1*`,` *theta2*`}]`	Circular or elliptical arc, as specified by *radii*, with outline and fill

Shape drawing commands.

`SchemeLine` produces an arbitrary *open* curve. It is thus simply an alternative to the *Mathematica* `Line` primative, but one which respects LevelScheme outline style options. Arrow heads may be drawn on either end of the line, by specifying `ShowTail→True` or `ShowHead→True`, and the properties of these arrowheads can be specified exactly as described above for line arrows.

```
PointList=Table[{x,x^3},{x,-1,1,0.1}],
SchemeLine[PointList,Thickness->2,ShowHead→True]
```

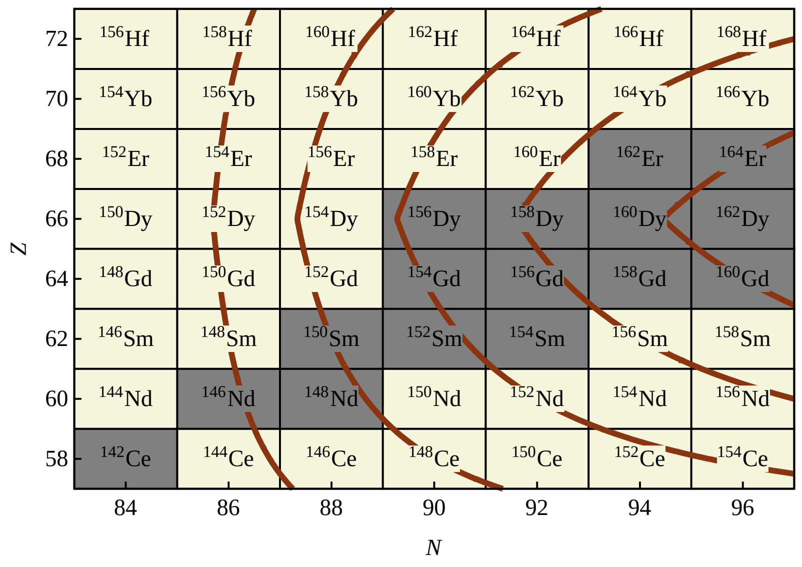


`SchemePolygon` produces an arbitrary *closed* curve, with both an outline and fill. It is thus an enhanced version of the *Mathematica* `Polygon` primative.

`SchemeBox` produces a rectangle with outline, fill, and top, bottom, left, right, and center labels. It is thus an enhanced version of a *Mathematica* `Rectangle` graphics primative. Note that there are several different ways of specifying the rectangle's coordinates. It is usually most convenient to specify the *x* and *y* coordinates as a pair of ranges $\{\{x_1, x_2\}, \{y_1, y_2\}\}$. Alternately, you can specify the corner points $\{x_1, y_1\}$ and $\{x_2, y_2\}$. The first syntax is provided for consistenty with `PlotRange`, the second for consistency with `Rectangle`. If you wish instead to specify the center point of the rectangle and *x* and *y* half-widths, this is accomplished using the otherwise-identical `SchemeSquare` drawing object. The syntax for specifying the half-widths is the same as for the "radii" given to `SchemeCircle`, discussed below.

`SchemeBox` can be used for various purposes within a level scheme, such as to highlight a level, provide a boxed title for the scheme, or create a gray band representing a resonance. However, `SchemeBox` is also a general-purpose drawing element ideal 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. It is especially useful in conjunction with the *Mathematica* `Table` list-construction function, which can be used to automate the construction of large arrays of boxes. For instance, a table of nuclides can be created with the help of data provided by the *Mathematica* `Miscellaneous`ChemicalElements`` package, which provides the chemical symbol for each element and a list of stable isotopes. 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-Plots.nb`.

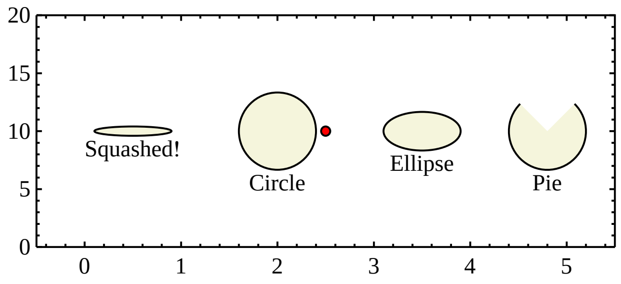
```
Table[
 SchemeBox[
  {{NValue - 1, NValue + 1}, {ZValue - 1, ZValue + 1}},
  LabC -> ChemicalSymbol[NValue, ZValue],
  FillColor → If[IsStable[NValue, ZValue], Gray, LightBeige],
  BackgroundC → If[IsStable[NValue, ZValue], Gray, LightBeige]
  ],
 {ZValue, ZMin, ZMax, 2}, {NValue, NMin, NMax, 2}
 ]
```



`SchemeCircle` produces a circle or ellipse with outline, fill, and top, bottom, left, right, and center labels. Beginning and ending angles can be specified for drawing an arc. `SchemeCircle` is thus an enhanced version of the *Mathematica* `Circle` and `Disk` primatives. Arrow heads may be drawn on either end of the arc, by specifying `ShowTail→True` or `ShowHead→True`, as described above for line arrows. Note that an extra label `LabX` is also defined, drawn a fraction `PosnX` of the way along the arc.

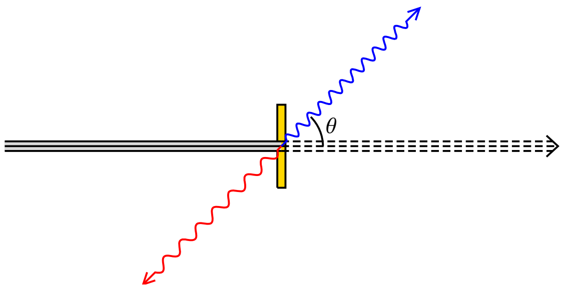
Circles in *Mathematica* generally suffer from being distorted into ellipses. A "circle" of radius 1 in *Mathematica* is drawn as an ellipse 1 horizontal plotting unit wide and 1 vertical plotting unit high — But this is not a circle at all unless the horizontal and vertical plotting scales happen to be identical! `SchemeCircle` allows the units in which the horizontal and vertical radii are given to be specified explicitly: both radii in horizontal plotting units, both radii in vertical plotting units, or both radii in printer's points. This facilitates the drawing of true circles and can be convenient for specifying ellipses as well. The possible forms of the *radii* argument are *r*, `{`*r1*`,`*r2*`}`, `Horizontal[`*r*`]`, `Horizontal[{`*r1*`,`*r2*`}]`, `Vertical[`*r*`]`, `Vertical[{`*r1*`,`*r2*`}]`, `Point[`*r*`]`, and `Point[{`*r1*`,`*r2*`}]`.

```
SchemeCircle[{.5,10},0.4,LabB→"Squashed!"],
SchemeCircle[{2.0,10},Horizontal[0.4],LabB→"Circle"],
SchemeCircle[{2.5,10},Vertical[0.4],FillColor→Red],
SchemeCircle[{3.5,10},Horizontal[{0.4,0.2}],LabB→"Ellipse"],
SchemeCircle[{4.8,10},Horizontal[0.4],{3*Pi/4,Pi/4+2*Pi},LabB→"Pie"],
```



The following simple diagram makes use of several of the drawing objects just described.

```
Figure[
    {

      (* target *)
      SchemeSquare[{0,0},{0.03,0.3},FillColor→Gold],

      (* beam *)
      SetOptions[SchemeArrow,ArrowType→MultilineArrow,
      ShaftLines→3,FillColor→LightGray],
      SchemeArrow[{-2,0},{0,0},ShowHead→False,HeadLength→0],
      SchemeArrow[{0,0},{2,0},Dashing→Automatic,ShowFill→False],

      (* gamma rays *)
      SetOptions[SchemeArrow,ArrowType→SquiggleArrow],
      SchemeArrow[{0,0},{1,1},Color→Blue,SquiggleWavelength→8],
      SchemeArrow[{0,0},{-1,-1},Color→Red,SquiggleWavelength→12],
      SchemeCircle[
        {0,0},0.3,{0,Pi/4},
        ShowFill→False,
        LabX→"θ",PosnX→0.5,BufferX→1,OrientationX→Horizontal
      ]

      },
    PlotRange→{{-2,2},{-1,1}},
    ImageSize→72*{4,2}
    ]
```



■ Labels

Several commands are provided for drawing stand-alone labels which are not part of any other object. Some of these labels are positioned manually, and others are positioned automatically with respect to a named level. The label positioning options are similar to those discussed above, but with no letter appended to their names: `ShowText` (not `ShowLab`), `Orientation`, `Offset`, `Nudge`, and `Background`.

`ManualLabel[` *point*, *contents*`]`	Writes a label at coordinates *point*
`ScaledLabel[` *scaledpoint*, *contents*`]`	Writes a label at scaled coordinates *scaledpoint*, where {0, 0} is the lower left corner of the current plot region and {1, 1} is the upper right corner

General label drawing commands.

`ManualLabel` is used to place a label at a specific position according to coordinates.

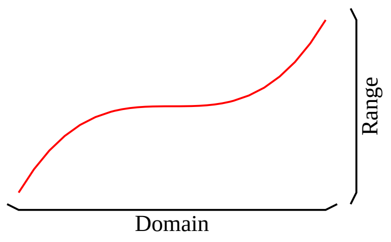
`ScaledLabel` is used to place a label at a specific fraction of the way across and up the display region, without reference to the coordinate system. (See the section on coordinate systems below for a description of "scaled" coordinates.) This is useful for plot titles, which should not have their position affected by the choice of coordinate range.

`SchemeBracket[` `Top/Bottom`, { *x1*, *x2* }, *y*`]`, `SchemeBracket[` `Left/Right`, *x*, { *y1*, *y2* }`]`	Bracket with label

Bracket drawing commands.

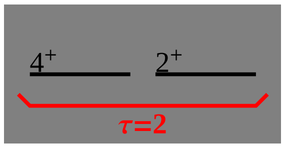
`SchemeBracket` is used to produce bracket-like labels. The bracket consists of a main line segment with an angled line segment at each end. It is therefore very similar in construction to a line arrow with an arrowhead at each end. In fact, `SchemeBracket` respects the same arrowhead control options (`ShowHead`, `HeadLength`, `HeadLip`). The bracket label is specified with `LabB`, `LabL`, `LabT`, or `LabR`, as appropriate, and its positioning along the length of the bracket is controlled with the option `PosnB`, `PosnL`, `PosnT`, or `PosnR`, much like for an arrow label. The following provides a simple example.

```
SchemeBracket[Bottom,{-1,1},-1.2,LabB->"Domain"],
SchemeBracket[Right,+1.2,{-1,1},LabR->"Range"]
```



When `SchemeBracket` is used to annotate levels in level schemes, the *x* range or *y* range may be replaced by a pair of level IDs, as in the following example.

```
SetOptions[SchemeBracket,HeadLength→6,HeadLip→6,Color→Red,FontSize→15,FontWeight-
>"Bold"],
SchemeBracket[Bottom,{lev1,lev2},-0.5,LabB->"τ=2"]
```



`LevelLabel[` *level*`,` *side*`,` *contents*`]`	Writes a label adjacent to the left or right endpoint of the level *level* (with an optional call–out line) or above or below the level, depending whether *side* is `Left`, `Right`, `Top`, or `Bottom`
`BandLabel[` *level*`,` *contents*`]`, `BandLabel[` *level*`,` *posn*`,` *contents*`]`	Writes a label centered below the level *level* or at horizontal position *posn* relative to its left end

Special label drawing commands for level schemes.

option name	default value	
`CallOutVector`	`None`	Specification {*xdist*, *ydist*} of call-out line between level and label; value `None` indicates no call–out line
`Gap`	`0`	Horizontal gap between level end point and label, in printer' s points; if a call–out line is present, can be specified as {*inner*, *outer*} to separately control the distance between the level and the line and the distance between the line and the label

Special label positioning options for `LevelLabel`.

`LevelLabel` essentially provides an extra left, right, top, or bottom label for a level. For left or right labels, the annotation can be connected to the end point of the level with a "call-out" line, and the horizontal gap between the annotation and the level can also be controlled, as specified by the options `CallOutVector` and `Gap`.

`BandLabel` positions its label by default immediately below the center of the specified level, or the horizontal position can be specified with an optional extra parameter. This is useful for putting a label beneath a band of levels. `BandLabel` produces essentially the same result as a `LevelLabel` attached to the `Bottom` of a level, but the separate command `BandLabel` is provided both for backward compatibility and so that label font options can be set separately for band and level labels.

The following figure illustrates the use of several label types.

```
SetOptions[SchemeObject,FontSize→15,FontColor→Red],
ScaledLabel[{0.05,0.95},"Expt",Offset→{-1,+1}],
ManualLabel[{0.5,50},"⋮",FontWeight->"Bold"],
LevelLabel[lev0,Right,"100 y",CallOutVector→{15,-5},Dashing→Automatic],
LevelLabel[lev100,Right,"50 ps"],
BandLabel[lev0,"g.s. band"]
```

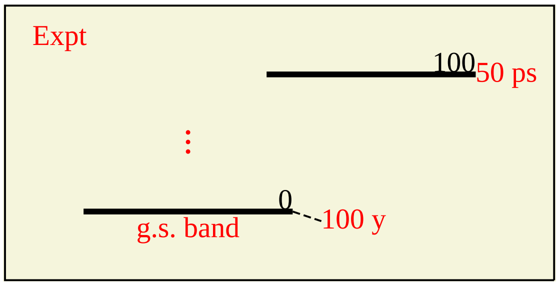


Illustration of the use of labels.

V. Including *Mathematica* plots and other graphics

▪ Two-dimensional graphics

`RawGraphics[` *graphics* `, ...]`	Includes one or more *Mathematica* graphics objects in the drawing

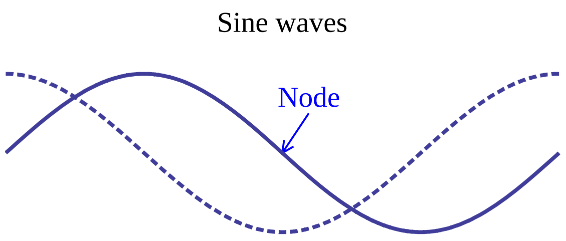
Commands for including graphics.

Command for including *Mathematica* graphics in a LevelScheme drawing.

Any two-dimensional graphics drawn in *Mathematica*, such as function plots, data plots, and geometric figures, can be included in a LevelScheme drawing using the `RawGraphics` command. The usual LevelScheme drawing style options can be applied to the graphics (in this case, they are `Show`, `Color`, `Thickness`, and `Dashing`). However, these only control the *default* drawing style. Some *Mathematica* graphics functions (*e.g.*, `Plot` when the `PlotStyle` option is specified) might override these. The drawing `Layer` (discussed in a later section) can be specified as well.

A very simple example combining *Mathematica* function plots with LevelScheme's convenient labeling and annotation features follows.

```
ScaledLabel[{0.5,0.9},"Sine waves",FontSize→15],
SchemeArrow[{Pi+0.3,0.5},{Pi,0},LabT->"Node",
  OrientationT→Horizontal,FontSize→15,Color→Blue],
RawGraphics[Plot[Sin[x],{x,0,2*Pi}],Thickness→2],
RawGraphics[Plot[Cos[x],{x,0,2*Pi}],Thickness→2,Dashing→Automatic]
```



Users who have previously used `Show` or `GraphicsArray` to manipulate and combine *Mathematica* plots should note that there is no need to set `DisplayFunction→Identity` while generating plots (see the *Mathematica* documentation for an explanation if this is unfamiliar). This is since the normal *Mathematica* plot display function is temporarily disabled inside `Figure`. Also, `RawGraphics` is capable of processing graphics involving any combination of conventional coordinates and coordinates specified in *Mathematica* `Offset` or `Scaled` notation.

■ Three-dimensional graphics

`ViewPort3D[{`*graphics*`, …}, {{`*x1*`, `*x2*`}, {`*y1*`, `*y2*`}}]`	Displays three–dimensional graphics within a LevelScheme figure, inset within the coordinate rectangle {{*x1*, *x2*}, {*y1*, *y2*}}
`ScaledViewPort3D[{`*graphics*`, …}, {{`*x1s*`, `*x2s*`}, {`*y1s*`, `*y2s*`}}]`	Displays three–dimensional graphics within a LevelScheme figure, inset within the scaled coordinate rectangle {{*x1s*, *x2s*}, {*y1s*, *y2s*}}

Command for including 3D graphics.

Any three-dimensional *Mathematica* graphics may be included in a two-dimensional figure using the commands `ViewPort` or `ScaledViewPort`. The commands `ViewPort` or `ScaledViewPort` accept all the usual options for `Graphics3D`, such as options controlling the display range and perspective (`PlotRange`, `BoxRatios`, `ViewPoint`, *etc.*) and options controlling the coloring and lighting effects (`Lighting`, *etc.*). All these options are described in detail in the *Mathematica* help.

`SchemeLine3D[{`*point1*`, `*point2*`, …}]`	Line
`ManualLabel3D[`*point*`, `*contents*`]`	Label at coordinates *point*
`SchemeArrow3D[`*P1*`, `*P2*`]`, `SchemeArrow3D[`*P1*`, `*P2*`, `*headangle*`]`	Arrow from *point1* to arbitrary *point2*, with optional *headangle* controlling head orientation

3D graphics objects.

At present, LevelScheme includes only a very limited set of tools for drawing and annotating three-dimensional diagrams. Three-dimensional analogues of `SchemeLine`, `ManualLabel`, and `SchemeArrow` are provided. See the *Mathematica* documentation for information on displaying three-dimensional graphics. Some of these were used to draw the diagram in panel (b) of the example figure above.

The three-dimensional analogue of `SchemeLine` is `SchemeLine3D`. The usual LevelScheme line style options can all be specified: `Color`, `LineColor`, `Dashing`, and `ShowLine`.

The three-dimensional analogue of `ManualLabel` is `ManualLabel3D`. Most, but not all, of the usual LevelScheme label control options can be specified: `Color`, `FontFamily`, `FontSize`, `FontWeight`, `FontSlant`, `FontTracking`, `FontColor`, `BackgroundFontSizeFactor`, `Offset`, `Orientation`, `ShowText`, `Background`.

The three-dimensional version of the LevelScheme-style arrow is `SchemeArrow3D`. It is a simple line arrow, with optional head and tail, and one label (`LabC`) attached. The arrowhead will only appear with the proper proportions if the 3D display option `BoxRatios` is set in proportion to the *x*, *y*, and *z* plot ranges.

VI. Figure construction

■ Coordinate systems

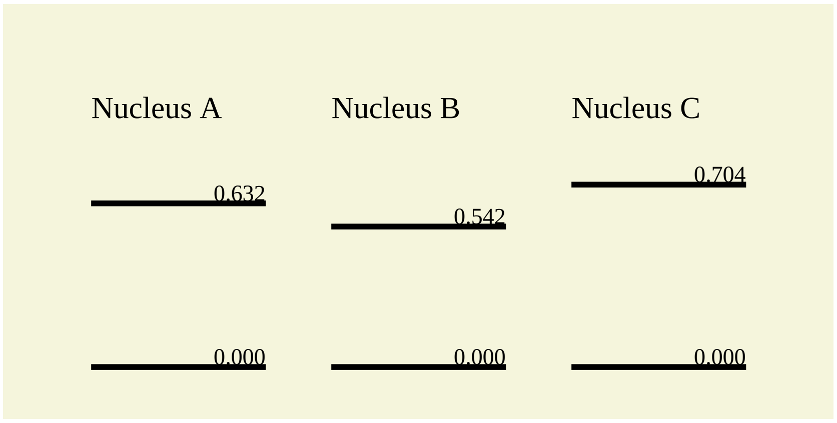
Preparing a figure with multiple parts becomes much easier if it is possible to draw each part in a separate "coordinate system" and then, separately, decide how these parts should be arranged with respect to each other. In this section, we first consider a simple tool for shifting parts of a diagram with respect to each other and for rescaling axes, then we address the full machinery needed to make inset plots or to make multipanel plots.

`SetOrigin[` *x0* `]`, `SetOrigin[{` *x0*, *y0* `}]`, `SetOrigin[]`	Sets the user origin to { *x0* , 0} or to { *x0*, *y0* }, or with no argument resets offset to zero
`SetScale[` *yscale* `]`, `SetScale[{` *xscale*, *yscale* `}]`, `SetScale[]`	Sets the user coordinate scale factors to {1, *yscale* } or to { *xscale*, *yscale*}, or with no argument resets scale to unity
`SetRegion[{{` *x3c*, *x4c* `}, {` *y3c*, *y4c* `}}, {{` *x3r*, *x4r* `}, {` *y3r*, *y4r* `}}]`, `SetRegion[]`	Sets the current plotting region coordinate system as specified, or with no arguments resets it to the full canvas

Commands for coordinate system control.

Frequently it is necessary to draw side-by-side level schemes or diagrams with multiple parts. Side-by-side level schemes can easily be drawn by preceding the code for each scheme with `SetOrigin[`*x0*`]` to control its horizontal position. All the coordinates for the objects in each scheme are specified relative to the "zero" for that scheme, so the user does not need to manually add an offset to the horizontal coordinates of each level. This allows easy adjustment of the inter-scheme spacing simply by redefining the *x0* values.

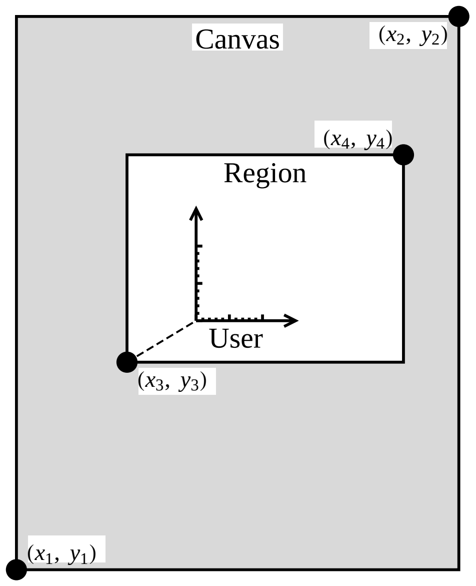
```
SetOrigin[0],
ManualLabel[{0.1,1.0},"Nucleus A"],
Lev[lev0,0,1,"0.000"],
Lev[lev1,0,1,"0.632"],

SetOrigin[1.1],
ManualLabel[{0.1,1.0},"Nucleus B"],
Lev[lev0,0,1,"0.000"],
Lev[lev1,0,1,"0.542"],

SetOrigin[2.2],
ManualLabel[{0.1,1.0},"Nucleus C"],
Lev[lev0,0,1,"0.000"],
Lev[lev1,0,1,"0.704"]
```



Side-by-side level schemes created with `SetOrigin`.

Even within a single level scheme, if there are multiple families of levels (*e.g.*, bands or group representations) it can be convenient to enter the horizontal coordinates within each family of levels as if they started from zero and then use `SetOrigin` to move the families around to their final positions. This practice makes later adjustments to the layout (spreading out the spacing, inserting new families of levels) much simpler. `SetOrigin` can be used to produce a vertical offset as well, and `SetScale` allows multiplicative factors to be applied to all horizontal or vertical coordinates. Thus, for instance, `SetScale` is useful for *ad hoc* adjustments to the energy scale of a level scheme. In the following section, tools will be described allowing each part of a diagram to be accompanied by axes which respect the current scale factors.

To construct more sophisticated multipart figures, it is very useful to be able to define a smaller rectangular plotting "region" within the full figure (or "canvas", to use artistic imagery) and to arbitrarily choose a new range for the horizontal and vertical plotting scales within this region. Two sets of information are needed to define the plotting region: (1) where the region lies on the canvas and (2) the coordinate range used within the region. These are specified using `SetRegion`. The arguments are the "canvas" coordinates and "region" coordinates for the lower and upper limits of the rectangle, which are, equivalently, the coordinates of the corner points denoted 3 and 4 in the diagram below.



LevelScheme canvas, region, and user coordinates.

The effects of `SetOrigin` or `SetScale` can be combined with those of `SetRegion`: for instance, after `SetRegion` is used to define a plotting region, `SetOrigin` can be used to move around parts of a diagram within this region. In practice, most users will rarely if ever use `SetRegion` directly. Instead, the `Panel` command of the following section will be used to set the plotting region and draw axes, labels, *etc.*, all in one step.

The following is a brief summary of the different coordinate systems used by LevelScheme. The *Mathematica* graphics system recognizes only one set of coordinates, the *Mathematica* plotting coordinates, which span the range defined by the option `PlotRange` given to `Figure`. However, the LevelScheme package defines four other coordinate systems, superposed upon these basic coordinates. The five, in total, coordinate systems are
Canvas coordinates: The usual *Mathematica* plotting coordinates.
Absolute coordinates: The physical distance in printer's points from the lower left corner of the plot.
Region coordinates: The redefined coordinates on an inset rectangle within the canvas.
Scaled coordinates: The fractional distance, from 0 to 1, across the inset rectangle.
User coordinates: Coordinates which may differ from the region coordinates by having an additional user-defined offset and scale. These are the coordinates in which the user specifies all object positions.

Normally all graphics are clipped to the current plotting region. This is true both for LevelScheme drawing objects and for *Mathematica* graphics included via `RawGraphics`. (Clipping may be disabled by specifying `ClipToRectangle→False`.) Lines, polygons, and rectangles are truncated exactly to the edge of the region. Circles and points are simply included or excluded depending upon whether or not their center point lies within the region. Text is similary included or excluded depending upon the coordinates of the reference point.

■ Panels and multipanel plots

A major capability needed for generating publication-quality figures is the ability to create plotting panels in arbitrary positions, in order to make inset plots, multipanel plots, or other arrangements of plots. This capability is not built in to *Mathematica*, but the LevelScheme package provides a flexible system for producing plots with subpanels, based on the coordinate arithmetic infrastructure of the preceding section.

`FigurePanel[{{`*x3c*`,` *x4c*`}, {`*y3c*`,` *y4c*`}}]`	Draws a panel, optionally including a frame, frame ticks, frame labels, a background color, and a panel letter label; sets the plotting region to this panel
`ScaledFigurePanel[{{`*x3s*`,` *x4s*`}, {`*y3s*`,` *y4s*`}}]`	Draws a panel, as above, but covering the specified range of scaled coordinates, *i.e.*, fraction of the current plotting region

Panel generation commands.

option name	default value	
`XPlotRange`	`Automatic`	Horizontal coordinate range covered by plotting region coordinates; if `Automatic`, same as canvas coordinates
`YPlotRange`	`Automatic`	Vertical coordinate range covered by plotting region coordinates; if `Automatic`, same as canvas coordinates
`PlotRange`	`Automatic`	Alternative means of specifying coordinate range covered by plotting region coordinates; if `Automatic`, plot range is controlled by `XPlotRange` and `YPlotRange` options instead
`ExtendRange`	`0`	Fractional amount by which plot range should be extended on each side, to allow extra visual space or room for labels; separate values `{`*horizontal*`,` *vertical*`}` or `{{`*left*`,` *right*`}, {`*bottom*`,` *top*`}}` may be specified

Panel plot range options.

option name	default value	
`Background`	`None`	Background color for panel
`Frame`	`True`	Whether or not to draw frame line
`ShowEdge`	`{True, True, True, True}`	If frame line is drawn, which individual edges to draw
`Color`	`Black`	Color for frame, ticks, and labels (but not background)
`LineColor`, `Thickness`, `Dashing`, `DashingGap`, `DashingCorrection`, `ShowLine`	*same defaults as usual*	Line style options for frame
`FontColor`, `FontSize`, `FontFamily`, `FontWeight`, `FontSlant`, `FontTracking`, `BackgroundFontSizeFactor`	*same defaults as usual*	Font style options for frame labels, also serving as default for ticks and panel letter if not overriden by other options below

Basic panel style options.

option name	default value	
`LabB`, `LabL`, `LabT`, `LabR`	`None`	Bottom, left, top, and right frame labels
... and the usual label positioning options, plus ...		
`Posn` *X*	`Automatic`	Position of label along frame edge *X*; value is specified as *fraction* of distance between ends of axis; *see similar option for transition arrow labels*
`Buffer` *X*	`Automatic`	Buffer spacing between label *X* and arrow shaft, as multiple of half current font height; *see similar option for transition arrow labels*
`FrameLabel`	`Automatic`	Alternative specification of frame labels, provided for consistency with `Plot`, *etc.*; value of `Automatic` specifies that the usual LevelScheme labels `LabB`, `LabL`, `LabT`, and `LabR` should be used

Panel frame label options.

option name	default value	
`FrameTicks`	`None`	Tick mark definition list for frame edges; value may be `None`, `Automatic` , or a list of up to four individual axis specifications
`TickNudge`	{`0, 0, 0, 0`}	Horizontal and vertical fine-tuning adjustment to be applied to all the tick labels on each of the four panel edges
`ShowTicks`	{`True, True, True, True`}	Whether or not to allow display of tick marks (if any specified by `FrameTicks`) on each of the four panel edges
`ShowTickLabels`	{`True, True, True, True`}	Whether or not to allow display of labels on tick marks (if any specified by `FrameTicks`) on each of the four panel edges
`TickLineColor`, `TickThickness`, `TickShowLine`	`Automatic`	Line style options for ticks; if `Automatic` , same as for frame line; can be overridden for individual ticks by *Mathematica* tick style directives (see CustomTicks documentation)
`TickFontColor`, `TickFontSize`, `TickFontFamily`, `TickFontWeight`, `TickFontSlant`, `TickFontTracking`	`Automatic`	Font specifications for tick mark labels; if `Automatic`, values are obtained from options `FontColor` , …

Panel tick mark options.

option name	default value	
`PanelLetter`	`None`	Contents of panel letter label
`PanelLetterCorner`	`{-1, +1}`	Corner in which panel letter should appear; `{-1, +1}` for left–top, `{+1, +1}` for right–top, *etc.*
`PanelLetterInset`	`{15, 15}`	Horizontal and vertical inset of panel letter label from specified corner, in printer's points
`PanelLetterFontColor`, `PanelLetterFontSize`, `PanelLetterFontFamily`, `PanelLetterFontWeight`, `PanelLetterFontSlant`, `PanelLetterFontTracking`	`Automatic`	Font specifications for panel letter label; if `Automatic`, values are obtained from options `FontColor`, …
`ShowPanelLetter`, `PanelLetterOffset`, `PanelLetterNudge`, `PanelLetterOrientation`, `PanelLetterBackground`	*same defaults as for usual label positioning options*	Positioning options for panel letter label

Panel letter formatting options.

The command `FigurePanel` sets the specified rectangle as the current plotting region and draws the various ancillary items, such as a frame, tick marks, and labels, around it. The `PlotRange` option determines the coordinate ranges plotted within this box ("region coordinates"). After the contents of the panel have been drawn, the plotting region can be restored to the whole canvas by calling `SetRegion[]` with no arguments.

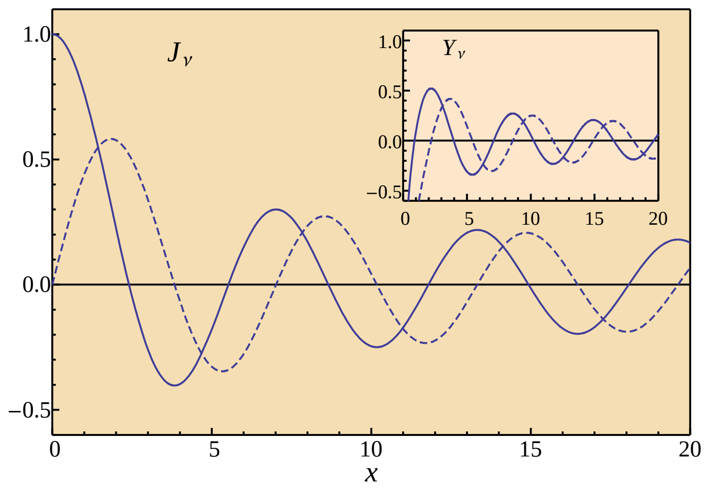
There are *many* formatting options, listed above, which can be used to control the details of a panel. These control five main parts of the panel: (1) a solid colored background rectangle, (2) a frame line, (3) ticks on each frame edge, (4) an axis label on each frame edge, and (5) a panel letter label. Several options require a list of four values, one for each edge. These are specified in the same ordering convention used by *Mathematica* plotting functions: bottom, left, top, right.

As already noted, the `PlotRange` option determines the coordinate ranges plotted within the panel. But, if `PlotRange` is left as `Automatic`, the horizontal and vertical plot ranges can instead be set separately, with the options `XPlotRange` and `YPlotRange`. This is often convenient in more complicated multipanel plots, as described below. If `XPlotRange` and `YPlotRange` in turn are left as `Automatic`, the canvas coordinates covered by the panel are simply used.

The tick mark intervals and properties can be chosen manually using the function `LinTicks` (see the separate user guide for the LevelScheme CustomTicks package). Or, they can be specified as `Automatic`, in which case they are constructed automatically by the CustomTicks package, using whatever current style options are in effect for `LinTicks`.

Following is an illustration making use of many of these options. It is worth noting some details of the example. The canvas plot range (`{{-0.2,1.1},{-0.2,1.1}}`) is chosen to allow a margin around the outermost panel (`{{0,1},{0,1}}`) so that the tick and axis labels around it can still fit within the plot. A panel axis label is by default positioned flush against the panel frame. So in this example the bottom label (`LabB`) must be manually moved down (`BufferB→2.5`) to allow room for the tick marks. (LevelScheme does not do this automatically, since *Mathematica* does not provide any mechanism whereby a *Mathematica* program can calculate how much room to allow for text.) For aesthetic reasons, the tick mark labels for the main panel in this example are drawn in a smaller font size (`TickFontSize`) than the main frame labels, and the tick marks for the inset panel are smaller yet.

```
Figure[
  {

    (* main panel *)
    FigurePanel[
      {{0, 1}, {0, 1}},
      PlotRange -> {{0, 20}, {-0.6, 1.1}},
      FrameTicks -> {LinTicks[0, 20], LinTicks[-1, 1, 0.5, 5]},
      FontSize -> 15, LabB -> textit["x"], BufferB -> 2.5,
      TickFontSize -> 12,
      Background -> Wheat
      ],
    SchemeLine[{{0, 0}, {20, 0}}],
    ScaledLabel[{0.2, 0.9}, SubscriptBox[textit["J"], "ν"], FontSize -> 15],
    RawGraphics[Plot[BesselJ[0, x], {x, 0, 20}]],
    RawGraphics[Plot[BesselJ[1, x], {x, 0, 20}], Dashing -> Automatic],

    (* inset panel *)
    ScaledFigurePanel[
      {{0.55, 0.95}, {0.55, 0.95}},
      PlotRange -> {{0, 20}, {-0.6, 1.1}},
      FrameTicks -> {LinTicks[0, 20], LinTicks[-1, 1, 0.5, 5]},
      TickFontSize -> 10,
      Background -> Eggshell
      ],
    SchemeLine[{{0, 0}, {20, 0}}],
    ScaledLabel[{0.2, 0.9}, SubscriptBox[textit["Y"], "ν"], FontSize -> 12],
    RawGraphics[Plot[BesselY[0, x], {x, 0, 20}]],
    RawGraphics[Plot[BesselY[1, x], {x, 0, 20}], Dashing -> Automatic]

    },
  PlotRange -> {{-0.2, 1.1}, {-0.2, 1.1}}, ImageSize -> 72*{6, 4}
  ]
```



`Multipanel[{{` *x3c*`,` *x4c*`},{` *y3c*`,` *y4c*`}}], {` *rows*`,` *columns*`}]`	Defines settings for a multipanel array
`Panel[{` *row*`,` *column*`}]`	Draws a panel as part of a multipanel array

Multipanel array generation commands.

option name	default value	
`XPanelSizes`	`1`	List of column widths on relative scale, or single width shared by all columns; `Automatic` adjusts the widths to provide equal horizontal coordinate scales on all panels
`YPanelSizes`	`1`	List of row heights on relative scale, or single height shared by all rows; `Automatic` adjusts the heights to provide equal vertical coordinate scales on all panels
`XGapSizes`	`0`	List of intercolumn gap widths on relative scale, or single width shared by all intercolumn gaps
`YGapSizes`	`0`	List of interrow gap heights on relative scale, or single height shared by all interrow gaps
`Margin`	`0`	Margin in printer' s points by which the multipanel plot as a whole should be indented relative to the given coordinates, typically to allow room for frame labels; separate values `{`*horizontal*`,` *vertical*`}` or `{{` *left*`,` *right*`},{` *bottom*`,` *top*`}}` may be specified

Multipanel layout options.

option name	default value	
`XPlotRanges`	`Automatic`	List of horizontal plot ranges to be used for the columns of a multipanel array, or single range to be repeated for all columns, or array giving values for all panels individually
`YPlotRanges`	`Automatic`	List of vertical plot ranges to be used for the rows of a multipanel array, or single range to be repeated for all rows, or array giving values for all panels individually
`XFrameLabels`	`None`	List of horizontal axis labels to be used for the columns of a multipanel array, or single value to be repeated for all columns, or array giving values for all panels individually
`YFrameLabels`	`None`	List of vertical axis labels to be used for the rows of a multipanel array, or single value to be repeated for all rows, or array giving values for all panels individually
`XFrameTicks`	`Automatic`	List of horizontal axis tick specifications to be used for the columns of a multipanel array, or single specification to be repeated for all columns, or array giving values for all panels individually
`YFrameTicks`	`Automatic`	List of vertical axis tick specifications to be used for the rows of a multipanel array, or single specification to be repeated for all rows, or array giving values for all panels individually

Multipanel axis specification options.

option name	default value	
`ShowFrameLabelsExterior`	`{True, True, False, False}`	Whether or not to allow display of frame labels on exterior panel edges in a multipanel array
`ShowFrameLabelsInterior`	`{False, False, False, False}`	Whether or not to allow display of frame labels on interior panel edges in a multipanel array
`ShowTickLabelsExterior`	`{True, True, False, False}`	Whether or not to allow display of tick labels on exterior panel edges in a multipanel array
`ShowTickLabelsInterior`	`{False, False, False, False}`	Whether or not to allow display of tick labels on interior panel edges in a multipanel array

Multipanel setup options controlling internal and external axis labeling.

option name	default value	
`First`	`“a”`	Character from which panel letter sequence starts
`Format`	`{“(”, “)”}`	Strings prepended and appended to panel letter
`Order`	`Horizontal`	Controls whether panel lettering proceeds across or down

Automatic panel letter generation options, for use in a multipanel plot.

LevelScheme provides tools to automate the layout of the most common form of multipanel plot, consisting of a rectangular array of panels with shared axes. The command `Multipanel` is used to define the settings for a rectangular array of panels. At minimum, `Multipanel` must be told the total rectangular region of the canvas to be used and the number of rows and columns of panels in the plot. Then `FigurePanel[{`*row*`,`*column*`}]` is used to create each individual panel. (Rows are numbered from top to bottom and columns from left to right, starting from 1, following the usual mathematical convention for indexing matrix entries.)

`Multipanel` can be given several options, which either affect the formatting of individual panels or control the layout of the array as a whole. Almost any of the formatting options for `FigurePanel` listed at the beginning of this section can be given to `Multipanel`, and the values will be saved to be used as the defaults for the panels in the multipanel plot. (When `Multipanel` it invoked, it stores a complete set of formatting option values for the panels, so that their style is "frozen" at this point. After this point, changes can safely be made to the default options for `SchemeObject` or even `FigurePanel`, and this will have no effect on the panels in the multipanel plot.) The options which *cannot* be specified for `Multipanel` are `XPlotRange`, `YPlotRange`, `PlotRange`, `LabB`, `LabL`, `LabT`, `LabR`, `FrameLabel`, `FrameTicks`, and `PanelLetter`, since these must be determined separately for each panel in the plot.

The plot ranges used within each panel are determined from the options `XPlotRanges` and `YPlotRanges`. The frame labels for each panel are determined from `XFrameLabels` and `YFrameLabels`. Frame labels only appear on the bottom and left outside edges of the array of panels. The frame ticks for each panel are determined from `XFrameTicks` and `YFrameTicks`. Usually it is desirable to suppress the major tick labels everywhere except the extreme outside edges of the array of panels. Major tick labels on exterior edges are controlled with the option `ShowTickLabelsExterior`, and those on interior edges are controlled with the option `ShowTickLabelsInterior`. So, for instance, the default values

```
ShowTickLabelsExterior→{True,True,False,False},
ShowTickLabelsInterior→{False,False,False,False}
```

produce tick labels only on the far bottom and left edges of the figure, while the values

```
ShowTickLabelsExterior→{True,True,False,False},
ShowTickLabelsInterior→{True,True,False,False}
```

produce tick labels on the bottom and left edges of each panel individually.

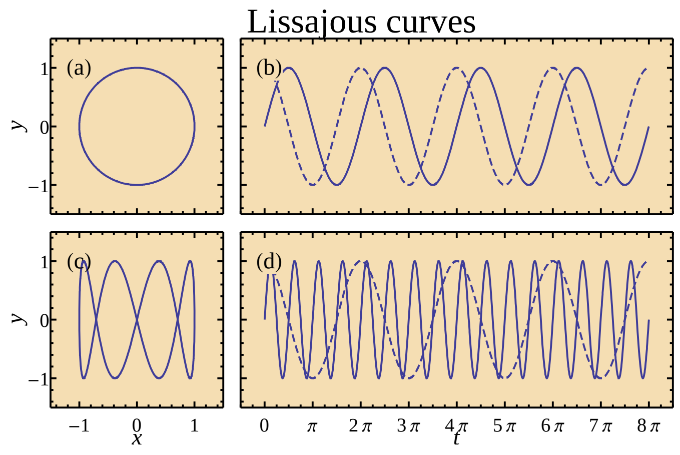
Any of the formatting options specified for `Multipanel` can be explicitly overriden for a single panel, if desired, by giving them as options to `FigurePanel` as usual. For instance, it is often convenient to override `XPlotRange` or `YPlotRange` for an individual panel, to set the plot range independently from those of the other panels in the same row or column.

Panel letters are calculated automatically from the row and column indices. A starting letter other than `"a"` can be specified with the option `First`, for instance, `"A"` for capital letters or some other letter if the earlier panels of the figure are to be drawn separately. Panel letters can be turned off with `ShowPanelLetter→False`.

By default, all columns of panels are of equal width, all rows 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`, `XGapSizes`, and `YGapSizes`. The columns and intercolumn gaps fill the available horizontal space, keeping the proportions given in these options; only the proportions matter, so multiplying both `XPanelSizes` and `XGapSizes` by the same factor has no effect. The rows and interrow gaps fill the vertical space in their specified proportions similarly. If `XPanelSizes` is set to `Automatic`, all the panel widths are made proportional to the *x* plot ranges, so all panels share the same *x* axis scale, and similary for the panel heights if `YPanelSizes` is set to `Automatic`.

The following is an example of a multipanel plot definition with gaps and unequal column sizes. The full code may be found in `Examples-Plots.nb`.

```
  Multipanel[
   {{0, 1}, {0, 1}},
   {2, 2},
   XPlotRanges -> {{-1.5, 1.5}, {-Pi/2, 8*Pi + Pi/2}},
   YPlotRanges -> {-1.5, 1.5},
   XFrameLabels -> {textit["x"], textit["t"]}, BufferB -> 2.5,
   YFrameLabels -> textit["y"], BufferL -> 3,
   TickFontSize -> 10,
   XFrameTicks -> {LinTicks[-2, 2, 1, 5], LinTicks[-Pi, 9*Pi, Pi, 4, TickLabelFunction ->
(Rationalize[#/Pi]*Pi &)]},
   YFrameTicks -> LinTicks[-2, 2, 1, 5],
   XPanelSizes -> {1, 2.5}, XGapSizes -> {0.1},
   YPanelSizes -> {1, 1}, YGapSizes -> {0.1},
   Background -> Wheat,
   PanelLetterBackground -> Wheat
   ],

  FigurePanel[{1, 1}],
  RawGraphics[ParametricPlot[{Cos[1*t], Cos[1*t - Pi/2]}, {t, 0, 2*Pi}]],

  ...
```

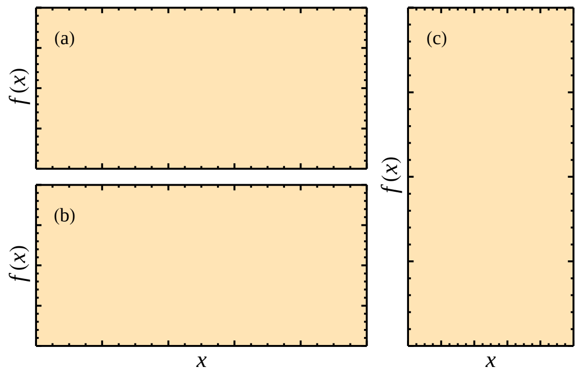


option name	default value	
`PanelAdjustments`	`None`	Adjustments to positions of panel edges {{ *left*, *right* }, { *bottom*, *top* }} outward from their default positions in a multipanel plot, or negative for inward adjustments, in the relative units defined by `XPanelSizes` and `YPanelSizes`; `None` gives no adjustment
`PanelShift`	`None`	Adjustment of panel position {*Δx*, *Δy*} relative to its regular grid position in a multipanel plot, in the relative units defined by `XPanelSizes` and `YPanelSizes`; `None` gives no adjustment

Geometry adjustment options for `Panel`, applicable when `Panel` is used as part of a multipanel plot.

Less conventional panel layouts can be achieved by overriding the dimensions of individual panels. In this case, the automatic left-to-right then top-to-bottom lettering scheme may also need to be overridden. An example follows, in which the top right panel is extended vertically to span two rows.

```
Figure[
 {

  SetOptions[Multipanel, ShowTickLabels -> {False, False, False, False},
   PanelLetterFontSize -> 10,
   Margin -> {{40, 40}, {40, 0}}, Background -> Moccasin],

  Multipanel[
   {{0, 1}, {0, 1}},
   {2, 2},
   XPlotRanges -> {0, 1}, YPlotRanges -> {-1, 1},
   XFrameLabels -> textit["x"], YFrameLabels -> textit["f(x)"], BufferL -> 1.5,
   XPanelSizes -> {2, 1}, XGapSizes -> 0.25, YGapSizes -> 0.1,
   Order -> Vertical
   ],
  FigurePanel[{1, 1}],
  FigurePanel[{2, 1}],
  FigurePanel[{1, 2},
   PanelAdjustments -> {{0, 0}, {+1.1, 0}},
   ShowFrameLabels -> {True, True, False, False}
   ]

  },
 PlotRange -> {{0, 1}, {0, 1}},
 ImageSize -> 72*{5, 3}
 ]
```



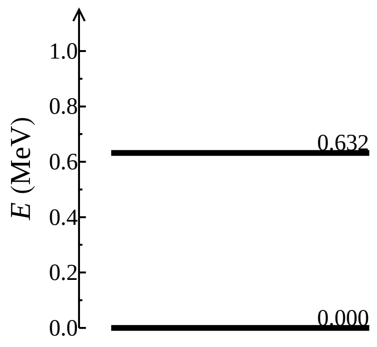
■ Axes

`SchemeAxis[Top/Bottom, {`*x1*`,` *x2*`},` *y*`]`, `SchemeAxis[Left/Right,` *x*`, {`*y1*`,` *y2*`}]`	Axis line, with optional tick marks, arrowhead, and axis label

Stand–alone axes.

`SchemeAxis` is used to produce free-standing axes within a figure. Any number of axes may be drawn, as needed, and the axes respect the plot regions or user scaled coordinates currently in effect. They are thus very useful in providing scales for multipart figures. The axis line has the appearance of a `LineArrow` and respects the same arrowhead control options (`ShowHead`, `HeadLength`, `HeadLip`). Ticks are specified with the option `Ticks` and must be given explicitly (*e.g.*, with `LinTicks`), not as `Automatic`. All the tick formatting options listed above for `Panel` (`ShowTicks`, `TickNudge`, `TickFontSize`, *etc.*) can be used, except that only a single value should be given instead of a list of four values. A single axis label can be specified with `LabB`, `LabL`, `LabT`, or `LabR`, as appropriate, and its positioning controlled as shown for `Panel` above. The following provides a simple example.

```
SchemeAxis[
  Left,0,{0,1.15},
  Ticks→LinTicks[0,1,0.2,2],TickFontSize→12,
  LabL->Row[{textit["E"]," (MeV)"}],FontSize→15,BufferL→4
  ]
```



Creating a free-standing axis.

■ Further description of the `Figure` command

The `Figure` command, introduced briefly earlier, accepts a list of LevelScheme objects as its argument. The list can contain any of the objects created by the commands described above, and arbitrary *Mathematica* `Graphics` objects can be included so long as they are enclosed within a `RawGraphics` object. The `Figure` command is very forgiving about the format of the object list. The objects can be contained in arbitrarily nested sublists (for instance, if the `Table` list creation function is used to automatically produce grids of boxes or bands of levels). Any non-graphical entry in the list is quietly ignored, so arithmetic scratchwork, variable value assignments, and careless extra commas can be included in the list without causing problems.

option name	default value	
`PlotRange`	*mandatory*	Coordinate range covered by plotting region; specified in the form `{{` *x1*`,` *x2* `}, {` *y1*`,` *y2* `}}`; *this option is not actually optional, since the values are needed for LevelScheme's internal calculations*
`ImageSize`	`Automatic`	Specifies the absolute size `{`*x*`,` *y*`}` of the displayed scheme in printer's points; if *Automatic* a 4 inch width is used with the golden ratio aspect ratio
`Axes`	`False`	Controls whether or not *Mathematica*'s plot axes are displayed
`Ticks`	`None`	Tick marks for axes

Formatting options for `Figure` differing from or beyond those for `Panel`.

The options for `Figure` are essentially a subset of those encountered above for `Panel`, with the few additions listed above. However, the behavior obtained with some of these options is slightly different, since `Figure` relies on the *Mathematica* `Show` function to draw the frame and tick marks, while `Panel` takes care of all drawing itself.

Of the basic panel properties, `PlotRange`, `Background`, `Frame`, `Color`, most of the line style options (`LineColor`, `Thickness`, and `Dashing`), and most of the font style options (`FontFamily`, `FontSize`, `FontWeight`, `FontSlant`, `FontTracking`, `FontColor`) apply to `Figure` as well. For `Figure`, the line style options affect only the frame or axis lines, not the tick marks. Frame labels can be specified either with `LabB`, `LabL`, `LabT`, and `LabR` or with `FrameLabel`, as for `Panel`, but none of the LevelScheme label positioning options apply. `FrameTicks` is used to specify the frame ticks, and the tick font style options can be given as for `Panel`.

A major "quirk" arising from `Figure`'s reliance on *Mathematica*'s `Show` for drawing the frame is that `Figure` *shrinks* the main body of the plot to something smaller than the size specified with `ImageSize` to make room for any tick mark or frame labels on the outside. But font sizes are not scaled down with the rest of the plot, so the proportion of labels to the graphics around them is affected. The shrinking is also very undesirable if predictability and consistency of the size of the plot frame is desired (for instance, if the plot is later to be displayed alongside others of the same size). To avoid these problems, `Figure` can be used with `Frame→False`, and any frame needed can be drawn manually *inside* the plot canvas with `Panel`, as in the example given earlier.

■ Parallel versions of the same figure (conditional inclusion)

`SchemeFlags`	`{}`	Flags set for conditional inclusion of parts of scheme

Conditional inclusion option for the `Figure` command.

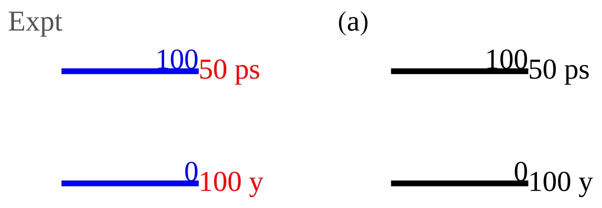
`SchemeIfDef[`*flag*`,` *object1*`, …]`	Includes objects in scheme only if *flag* is specified in the `SchemeFlags` option
`SchemeIfNDef[`*flag*`,` *object1*`, …]`	Includes objects in scheme only if *flag* is *not* specified in the `SchemeFlags` option

Conditional figure construction commands.

The `SchemeFlags` option and the `SchemeIfDef` and `SchemeIfNDef` commands are for use in maintaining multiple parallel versions of the same figure, *e.g.*, one in color for conference presentation and one in black and white with figure letters for publication, without the need for multiple separate copies of the code for the figure. It is tedious to maintain two copies of the code, since this leads to double work whenever changes need to be made. Rather, a single copy can be used, and any segment of code which is applicable only to one version should be placed within a `SchemeIfDef` command. It will then only be evaluated as part of the scheme when the designated flag is set with `SchemeFlags`. Here is an example: the level scheme on the left was generated with `SchemeFlags→{"Title","Color"}` and that on the right with `SchemeFlags→{"FigureA"}`.

```
SchemeIfDef["Title",
  ScaledLabel[{0.05,0.95},"Expt",Color→DimGray]
  ],
SchemeIfDef["FigureA",
  ScaledLabel[{0.05,0.95},"(a)"]
  ],
SchemeIfDef["Color",
  SetOptions[Lev,Color→Blue],
  SetOptions[LevelLabel,Color→Red]
  ]
```

```
GraphicsArray::obs : GraphicsArray is obsolete. Switching to GraphicsGrid. >>
```



Using conditional code inclusion to maintain parallel versions of a figure.

■ Layers

LevelScheme organizes graphics into "layers". Objects assigned to lower-numbered layers (background) are drawn before, and thus might be hidden by, objects assigned to higher-numbered layers (foreground). Objects within the same layer are rendered in the order they appear in the list given to `Figure`. By default, outlines and fills are in layer 1, white-out boxes are in layer 2, and text is in layer 3. All *Mathematica* graphics included using `RawGraphics` appear in layer 1, and the colored background generated by `Panel` is drawn in layer 0. Layered drawing prevents text labels from being hidden by other drawing elements in dense level schemes or technical diagrams. Most importantly, with this layering system, white-out boxes hide any lines or fills behind them, but they do not block neighboring text. This keeps nearby labels with white-out boxes from obstructing each other. The layer of an object may be modified with the `Layer` option, *e.g.*, to push it to the background or pull it to the foreground.

■ Custom tick marks

The default tick marks produced by *Mathematica*'s plotting functions are typically not suitable for publication. Most notably, *Mathematica* drops trailing zeros after the decimal point in its default tick marks, leading to a series of ticks of "ragged" lengths (*e.g.*, "0.", "0.5", "1.", …). The CustomTicks package, a component of the LevelScheme system, provides extensive customization of tick mark placement and formatting. It may be used to generate the tick mark specifications to be given as options to `SchemeAxis`, `Panel`, or `Figure`, and it may also be used with *Mathematica* plotting functions in general. Linear, logarithmic, and general nonlinear axes are supported. The flexibility achieved matches or exceeds that available with most commercial scientific plotting software. (Beyond the considerable built-in customization options, the user can supply arbitary label formatting or axis transformations functions using the *Mathematica* programming language.) Documentation for this package may be found in the file `CustomTicksGuide.pdf`.

■ Text formatting

Mathematica offers advanced capabilities for typesetting text and formulas. This provides great flexibility in typesetting complex text for figure labels. These are described in the *Mathematica* documentation and are *not* the topic of the present section.

Rather, LevelScheme provides some *extra* commands to help typeset labels for scientific figures. These are summarized here.

`StackText[` *alignment*, *linespacing*, { *line1*, … }]	Produces multiline label; alignment can be `Left`, `Center`, or `Right`

Commands for constructing composite labels.

The commands `TightRowBox` and `StackText` are provided to facilitate laying out composite labels. `TightRowBox` functions similarly to *Mathematica*'s `RowBox`, combining several text elements side-by-side, but it eliminates undesirable horizontal spacing which usually appears between elements in a `RowBox`. `StackText` allows the construction of multiline labels with various forms of centering and adjustable line spacing. Examples are provided at the end of this section.

`textup[` *text*]	Produces ordinary upright text
`textsl[` *text*]	*Produces slanted text*
`textit[` *text*]	*Produces italic text*
`textmd[` *text*]	Produces ordinary-weight text
`textbf[` *text*]	**Produces boldface text**
`textrm[` *text*]	Produces Roman text (Times)
`texttt[` *text*]	`Produces typewriter text (Courier)`
`textsf[` *text*]	Produces sans-serif text (Helvetica)
`textcolor[` *color*, *text*]	Produces text of arbitrary color
`textsize[` *size*, *text*]	Produces text of arbitrary size
`hspace[` *dist*]	Produces a horizontal displacement; *dist* is specified in ems (a unit equal to the width of the letter M, chosen here for technical reasons)

Some LaTeX-like text formatting commands.

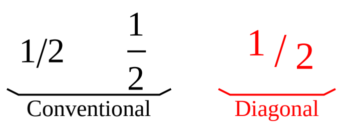
LevelScheme provides several LATEX-like commands for changing typeface. (These are simply typing shortcuts for much longer *Mathematica* StyleForm directives.)

`DiagonalFractionBox[` *a*, *b*`]`	Typesets fraction a/b in compact, diagonal format
`DiagonalFractionize[` *x*`]`	Extracts the numerator and denominator of a rational number *x* and typesets as above

Diagonal fraction formatting.

Mathematica only displays fractions in horizontal or "slash" form (*e.g.*, "1/2") or in vertical form (numerator above denominator). LevelScheme provides commands `DiagonalFractionBox` and `DiagonalFractionize` for typesetting fractions in the more compact diagonal format, *i.e.*, with the numerator in the "northwest" and the denominator in the "southeast", which is often more readable in figure labels.

```
ManualLabel[{1,0},DiagonalFractionize[1/2]]
```



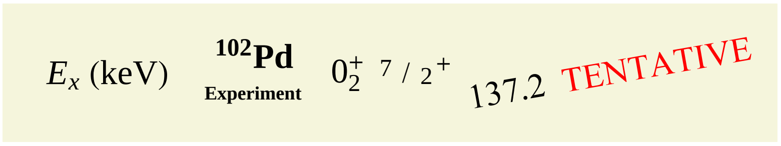
`LabelJP[` *spin*, *parity*`]`	Produces a level spin–parity label; parity argument is optional (default +) and may be +`1`, −`1`, or `None`
`LabelJiP[` *spin*, *i*, *parity*`]`	Produces a level spin–parity label with subscript *i*; parity argument is optional (default +) and may be +`1`, −`1`, or `None`

Spin-parity labels.

There are also specialized spin-parity labels `LabelJP` and `LabelJiP` for use in level schemes.

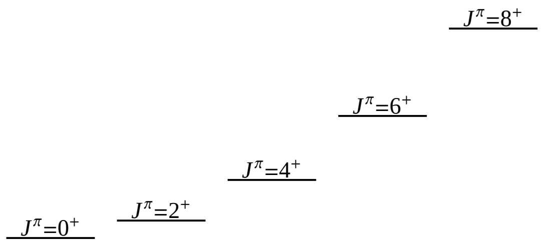
Following are some examples of labels formatted using these commands.

```
ManualLabel[{0,0},Row[{textit[SubscriptBox["E","x"]]," (keV)"}]],
ManualLabel[{0.7,0},textbf[StackText[Left,0,{Row[{SuperscriptBox["","102"],"Pd"}],textsize
[10,"Experiment"]}]]],
ManualLabel[{1.15,0},LabelJiP[0,2]],
ManualLabel[{1.45,0},LabelJP[DiagonalFractionize[7/2]]],
ManualLabel[{2.4,0},Row[{"137.2","  ",textcolor[Red,"TENTATIVE"]}],Orientation→10*Degree]
```



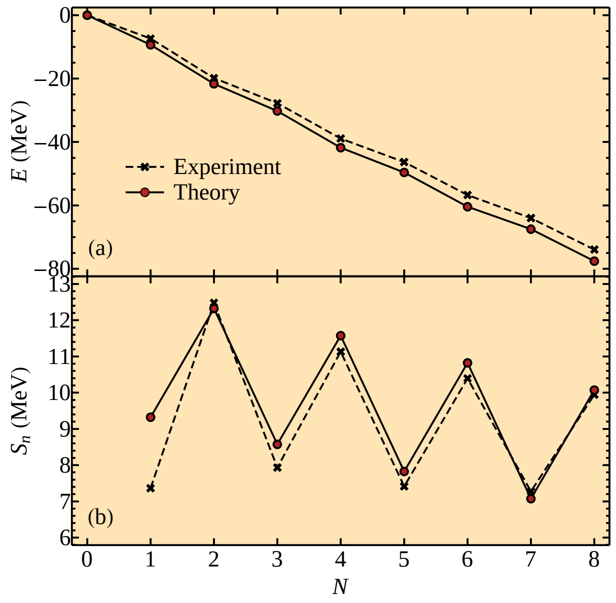
Although the "box" commands for creating labels may seem a little cumbersome, they are also very powerful, since the expressions you use to create text are expressions like any other *Mathematica* expression. For instance, they can include references to variables, like the variable *J* in the example below.

```
Table[
  Lev[
    J/2,J/2+1,J*(J+1),
    LabC->RowBox[{SuperscriptBox[textit["J"],"π"],"=",LabelJP[J]}]
    ],
  {J,0,8,2}
  ]
```



VII. Data plotting (under development)

Extensive data plotting capabilities are under development in LevelScheme. These will be fully realized (and documented!) in a future release. However, the examples given in the file `Examples-DataPlots.nb` will allow you to get started if your are particularly adventurous. Here let us just take an example which illustrates the main components of generating a data plot. Here is the plot we are working towards:



First we need to assemble our data in a table (a table or matrix is represented in *Mathematica* as a list of lists, as you can read more about in the *Mathematica* help). We might enter data manually, for instance:

```
ExptData={{0, 0., None}, {1, -7.363, 7.363}, {2, -19.843, 12.480}, {3, -27.776, 7.933},
{4, -38.906, 11.13}, {5, -46.321, 7.415}, {6, -56.715, 10.394}, {7, -63.991, 7.276}, {8,
-73.936, 9.945}};
```

This results in the data table

```
0   0.        None
1   −7.363    7.363
2   −19.843   12.48
3   −27.776   7.933
4   −38.906   11.13
5   −46.321   7.415
6   −56.715   10.394
7   −63.991   7.276
8   −73.936   9.945
```

Here we will use the first column for the "*x*" values and the second and third columns as "*y*" values. We might instead reading it in from a data file, using `Import[]`. We can even calculate the numerical values from a formula or using some more sophisticated function written in *Mathematica*:. For instance, if we have defined a function named SeniorityData to calculate the predictions of the "seniority model" in nuclear physics

```
TheoryData=SeniorityData[-9.323,0.75];
```

Now we are ready to prepare the figure. You can find the complete code for this example in `Examples-DataPlots.nb`. The following notes explain what you will find in the example code. We need to specify the appearance of the data *line* and data *symbols* for each data set. We can define defaults for all data sets

```
SetOptions[DataSymbol, SymbolSize -> 4],
```

and we can also define specific named styles, based on these defaults but overriding some of the options, for later use

```
DefineDataStyle["expt", DataLine -> {Dashing -> True}, DataSymbol -> {SymbolShape -> 
"Cross", Thickness -> 1.5}],
DefineDataStyle["theory", DataSymbol -> {FillColor -> Firebrick}],
```

Then it is easy to plot the data itself, either just using the first two columns of the data table as (XY) pairs

```
FigurePanel[{1,1}],
DataPlot[ExptData,DataStyle→"expt"],
DataPlot[TheoryData,DataStyle→"theory"],
```

or choosing specific X and Y data columns

```
FigurePanel[{2,1}],
DataPlot[DataSet[ExptData,DataColumns→{1,3}],DataStyle→"expt"],
DataPlot[DataSet[TheoryData,DataColumns→{1,3}],DataStyle→"theory"],
```

Then we can also draw a legend based on the line and symbol styles, defined either in a named style (as in this example) or any data set which was plotted

```
DataLegend[{0.1,0.45},
{
{"expt","Experiment"},
{"theory","Theory"}
}
],
```

The first argument `{0.1,0.45}` here gives the position for the upper left-hand corner of the legend, as scaled coordinates (fraction of the way across and up) within the panel.

VIII. Producing Encapsulated PostScript (EPS) or Portable Document Format (PDF) output

After creating a level scheme in *Mathematica*, you will usually want to export the graphics for use in a document or presentation. The most robust approach to exporting *Mathematica* graphics is to produce an Encapsulated PostScript (EPS) file or Portable Document Format (PDF) file.

■ Method #1: Select image in notebook and save as EPS/PDF

In the Mathematica's notebook interface, you can simply select the figure graphics (for instance, use the mouse to click on the image) and select "File > Save selection as..." from the menus. Chose EPS or PDF as the format, as you prefer. Under some operating systems (*e.g.*, Microsoft Windows), you can get to this menu more directly by right-clicking on the figure.

■ Method #2: Export the figure

You can store the graphics produced by `Figure` in a variable, and then give this as an argument to `Export`. For example:

```
P=Figure[...];
Export["c:\\work\\fig3.eps",P,"EPS"];
```

The advantage here is that you already have the file name typed into the notebook in case you want to export a revised version later (or want to export several different files with similar names).

■ Method #3: "Printing" to a PostScript driver (for EPS) or PDF distiller (for PDF)

Mathematica's export functions are now much more robust and reliable than in earlier versions, where they *usually* failed to produce reasonable output (if text was involved). However, they still *occasionally* fail to produce accurate output. There is fallback approach, which is generally the most robust of these three approaches, but it requires a little setup in advance.

For EPS output: To produce a satisfactory EPS file, you can "print" the graphics to a PostScript printer driver and save the results to a file. It is first necessary to install the printer driver software for a color PostScript printer, such as the HP Color LaserJet. (There is no need to have any printer actually physically attached to the computer.) The instructions vary depending upon your operating system.

Example — EPS printer driver installation for Windows XP: From the task bar, select Start»"Control Panel". Open the "Printers and Faxes" window. Select "Add Printer". Following the prompts, select "Local printer" and deselect the check box for "Automatically detect and install". Choose "FILE:" as the port to print to. For the model of printer, select "Hewlett Packard HP Color LaserJet PS". For the printer name, enter some descriptive name, such as "EPS file". Select "No" when asked whether or not to make this the default printer or to print a test page. Select "Finish". Now, back in the "Printers and Faxes" window, find this new printer in the list of printers. Select (highlight) it and, under the "File" menu, choose "Properties..." to modify its properties. Under the "Device Settings" tab, set both "Minimum font size to download as outline" and "Maximum font size to download as bitmap" to 0. Select "OK". Open the "Properties..." window for this printer again. Under the "General" tab, click "Printing Preferences...". Under the "Layout" tab, click "Advanced..." and, in the list of options, under "Document Options", find "PostScript Options". You may need to click the "+" sign to the left to expand this list of options. Set "PostScript Output Option" to "Encapsulated PostScript (EPS)", set "TrueType Font Download Option" to "Outline", and set "PostScript Language Level" to "1". (To avoid problems with washed-out colors, you may also wish to find the "Graphic" option "ICM Method" and set it to "ICM Disabled".) Select "OK".

Now, whenever you are ready to create an EPS file of one of your level schemes in *Mathematica*, first turn off *Mathematica*'s printing of page headers. (Under File»"Printing settings"»"Headers and footers", check the box by

"No header on first page" and uncheck the box by "Include line".) You only need to do this once for the notebook, and these settings will be saved with the notebook. Select the cell containing the level scheme you wish to print by clicking with the mouse on the blue bracket to its right. Choose File»"Print selection", select the EPS "printer" you just installed, and press "OK". A window should pop up prompting you for the name you would like for the EPS file.

For PDF output: To produce a PDF file, you may follow the instructions for EPS output above, to generate an EPS file, and then run a PDF converter (ps2pdf, Acrobat Distiller, *etc.*) on this EPS file. Or, you can directly install a PDF printer driver program (such as an Acrobat Distiller printer).

■ Adjusting the EPS bounding box

Before you include this EPS file in another document (*e.g.*, a LATEX document), you will probably want to adjust the "bounding box" for the file. Various programs can be used to do this. For instance, you may wish to use Ghostgum's GSView (`http://www.ghostgum.com.au`, and read the help topic “PS to EPS”) or the widely-available command-line utilities ps2epsi and epstool. Let us briefly consider what this means and why it is desirable. The graphics in an EPS file are specified in terms of coordinates on an imaginary piece of standard-sized paper. These graphics usually only fill a portion of the coordinate space, leaving the rest as white space. If you wish to include the EPS file as a figure in a word processor or LATEX document, it is necessary to specify to the word processor what portion of the page contains the actual graphics, so that the software can crop tightly around the graphics. This is accomplished by adding a "bounding box" definition to the file. The *Mathematica* `Export` command sets the bounding box to encompass the full figure area you defined with `PlotRange`. But this generally leaves some white space around the actual drawing elements, which you can trim off by defining a tighter bounding box.

■ Note on cutting and pasting (use at your own peril)

Under some operating systems, such as Microsoft Windows, it is also possible to simply "cut and paste" graphics from *Mathematica* into other applications via the windowing system clipboard. However, beware that any text in the figure may fail to display properly if the resulting file is ever used on a computer which does not have the *Mathematica* fonts installed -- for instance, undoubtedly, the conference computer on which you will by trying to display your PowerPoint presentation.

Appendices

■ Appendix A: Transition autospacing for decay schemes

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.

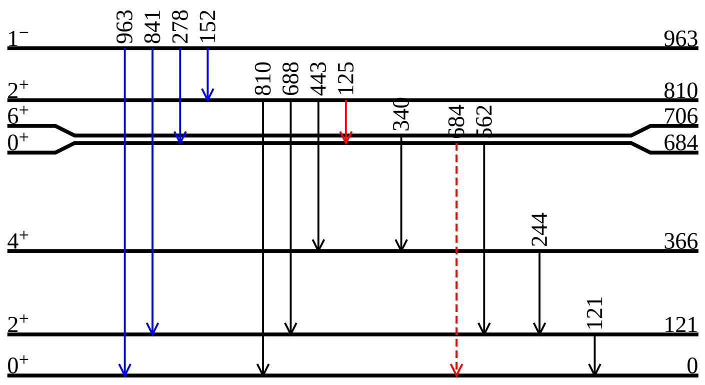
`AutoLevelInit[`*x0*`,` *dintra*, *dinter*`]`	Initializes autospacing, specifying horizontal coordinate *x0* for the first transition, spacing *dintra* between transitions from the same level, and spacing *dinter* between groups of transitions from different levels
`AutoLevel[`*level1*`]`	Specifies that the following transitions originate from level *level1*
`AutoTrans[`*level2*`]`	Draws a transition to *level2*; any options are passed on to `Trans`

Transition autospacing commands.

The following example illustrates the use of the `AutoLevelInit`, `AutoLevel`, and `AutoTrans` commands. Negative spacings are specified in `AutoInit` to draw the transitions successively from right to left.

It is usually desirable to set the option `BackgroundT→Automatic` for `Trans`, to create a white-out box behind each label, blocking out any higher-lying levels behind the label. If this box also cuts into the level line of the level from which the transition originates, the label can be nudged upwards with `NudgeT`.

```
SetOptions[Lev,Thickness→2,LabR→Automatic,WingTipWidth→25,
  Margin→0],
Lev[lev0,0,1,"0",LabL→LabelJP[0,+1]],
Lev[lev121,0,1,"121",LabL→LabelJP[2,+1]],
Lev[lev366,0,1,"366",LabL→LabelJP[4,+1]],
Lev[lev684,0,1,"684",LabL→LabelJP[0,+1],WingHeight→-5],
Lev[lev706,0,1,"706",LabL→LabelJP[6,+1],WingHeight→+5],
Lev[lev810,0,1,"810",LabL→LabelJP[2,+1]],
Lev[lev963,0,1,"963",LabL→LabelJP[1,-1]],

SetOptions[Trans,BackgroundT→Automatic,NudgeT→2],
AutoLevelInit[0.85,-0.04,-0.08],
AutoLevel[lev121],
AutoTrans[lev0,LabT→"121"],
AutoLevel[lev366],
AutoTrans[lev121,LabT→"244"],
AutoLevel[lev684],
AutoTrans[lev121,LabT→"562"],
AutoTrans[lev0,LabT→"684",LineColor→Red,Dashing→Automatic],
AutoLevel[lev706],
AutoTrans[lev366,LabT→"340"],
AutoLevel[lev810],
AutoTrans[lev684,LabT→"125",LineColor→Red],
AutoTrans[lev366,LabT→"443"],
AutoTrans[lev121,LabT→"688"],
AutoTrans[lev0,LabT→"810"],
AutoLevel[lev963],
AutoTrans[lev810,LabT→"152",LineColor→Blue],
AutoTrans[lev684,LabT→"278",LineColor→Blue],
AutoTrans[lev121,LabT→"841",LineColor→Blue],
AutoTrans[lev0,LabT→"963",LineColor→Blue]
```



Example of transition autospacing.

Appendix B: Notes for advanced users

Following are a few older and very cursory notes on some undocumented features. However, a massive updating of the documentation is in the works.

Line dashing: The option Dashing allows any combination of *dash* lengths to be specified, but the *gap* between them is automatically calculated. The following options allow control over the gap length. If different gap lengths are desired between successive dashes (rarely needed), this may also be accomplished, by giving the option `Dashing→AbsoluteDashing[…]` (see the *Mathematica* help for `AbsoluteDashing`).

option name	default value	
`DashingGap`	`Automatic`	The length of the gap between dashes, in printer' s points; if `Automatic` the length is determined from the average gap length specified with `Dashing`
`DashingCorrection`	`True`	Whether or not the dash lengths and gap lengths should be corrected for the finite "pen width" in PostScript graphics (recommended to prevent the gaps from filling in for thick lines)

Advanced dashing options.

Coordinate conversion: The function `ConvertCoordinate` converts coordinates between the five LevelScheme coordinate systems. `ConvertCoordinate[`*system*`,`*newsystem*`,`*type*`,{`*x*`,`*y*`}]` converts a point, while `ConvertCoordinate[`*system*`,`*newsystem*`,`*type*`,`*x*`,`*axis*`]` converts a single coordinate, either *x* or *y* depending upon whether *axis* is 1 or 2. The coordinate systems are specified as `AbsoluteCoords`, `CanvasCoords`, `RegionCoords`, `ScaledCoords`, or `UserCoords`. The conversion *type* (`C` or `D`) indicates whether the conversion is of a coordinate (scale and offset) or of a displacement (scale only). Coordinate conversion can be of use in carrying out positioning tasks involving adjustments in printer's points. In multipanel plots, it can be convenient to convert the coordinates of a point from user coordinates to canvas coordinates and save the result for later use in drawing annotations, such as arrows, spanning multiple panels.

Functions for converting coordinate ranges or regions, `ConvertRange[`*system*`,`*newsystem*`,`*type*`,{`*x1*`,`*x2*`},`*axis*`]` and `ConvertRegion[`*system*`,`*newsystem*`,`*type*`,{{`*x1*`,`*x2*`},{`*y1*`,`*y2*`}}]`, are also available.

Point saving and retrieval: `SavePoint[`*ID*`,{`*x*`,`*y*`}]` or `SavePoint[`*ID*`,{`*x*`,`*y*`},`*system*`]` saves the location of a point for later retrieval with `GetPoint[`*ID*`]` or `GetPoint[`*ID*`,`*system*`]`. The coordinates are automatically converted at the time of retrieval to whatever coordinate system is currently active. This is especially useful for drawing annotations (*e.g.*, arrows) which span different panels of a plot with different coordinate system definitions.

Levels: The ID argument is optional (however, it is good practice to always habitually define IDs for levels, to avoid having to go back and haphazardly add IDs when they are needed to specify transitions, connectors, *etc.*). In addition to left, center, and right labels, top and bottom labels can be specified as well (these are only rarely useful, as they are usually redundant to the center label).

The symbol `LastLevel` evaluates to the ID of the most recently defined level, and the energy of a level can be retrieved with `LevelEnergy[`*level*`]`.

The energy of a level may be adjusted upwards or downwards with the option `EnergyNudge`. The default energy label formatting may be overridden by providing a function as the value for the option `EnergyLabelFunction`.

Panels: `PanelLetter[]` returns a string giving the panel letter of the current panel. It accepts the panel letter formatting options listed earlier for `Multipanel`.

In a multipanel plot, additional options `X/YMarginSizes` may be used to specify the widths of gaps to the left of the leftmost panel, to the right of the rightmost panel, above the topmost panel, or below the bottommost panel, on the same relative scale as used in the `X/YPanelSizes` and `X/YGapSizes` options. This is occasionally useful for layout purposes.

Graphics utilities: If a *Mathematica* `Graphics` object contains just a single curve, as is often the case for the output of `Plot` and other *Mathematica* plotting and geometry routines, the `GrabPoints` function may be used to extract a list of the points from which this curve is constructed. (The `ExtractLines` function can be used to extract a list of curves from more complicated plots.) These points can be used as the argument to `SchemeLine` or `SchemePolygon`. This allows greater control of the appearance of the plot than is available with `RawGraphics`, since it allows the curve to be used as the boundary of a filled region or to be maniplated in various ways.

`GrabPoints[` *graphics* `]`	Extracts a list of points from a simple enough graphics object (typically the output of a plotting routine)
`ExtractLines[` *graphics* `]`	Extracts a list of lines from a graphics object

Command for manipulating graphics.

Appendix C: For users of earlier versions

Mathematica 6 introduced major changes to how graphics are created and also many new function names (some of which conflicted with those defined by LevelScheme). Therefore, LevelScheme had to change along with *Mathematica*. If you were using an older version of LevelScheme, under *Mathematica* 5 or earlier, you might wish to be aware of the following:

1. The LevelScheme command `Figure` (or `Scheme`), like all plotting functions in *Mathematica* now, should no longer be followed by a semicolon. A semicolon in *Mathematica* suppresses output. If the figure command is followed by a semicolon, the graphics it produces will not be displayed.

2. The LevelScheme command `Panel` has been renamed `FigurePanel`, to avoid conflicting with the new *Mathematica* symbol `Panel`. (Similarly, `ScaledPanel` has been renamed `ScaledFigurePanel`.)

3. The LevelScheme command `ViewPort` has been renamed `ViewPort3D`, to avoid conflicting with the new *Mathematica* symbol `ViewPort`. (Similarly, `ScaledViewPort` has been renamed `ScaledViewPort3D`.)

4. Object transparency is now possible in *Mathematica* graphics. LevelScheme supports transparency through new drawing options `Opacity`, `LineOpacity`, `FillOpacity`, `FontOpacity`, *etc.*, similar to the existing color options. *Caution:*Use of transparency is not compatible with PostScript output and may cause *Mathematica* to hang if you try to print or export.

5. Only a new, more-limited set of colors (`LightRed`, *etc.*) defined are defined by *Mathematica* now. However, LevelScheme defines the pre-*Mathematica* 6 named colors (`AliceBlue`, *etc.*) so that they are available for your use. Both sets of colors are displayed together in the LevelScheme color palette.