11
Using LaTeX in Science and Technology

While the prior chapter focused on mathematics, we'll now explore various scientific fields such as chemistry, physics, computer science, technology, and electronics. Given the significant reliance on mathematics in these disciplines, make sure you also explore *Chapter 10*, *Writing Advanced Mathematics*. This chapter will be an overview, showing specific recipes for how LaTeX can be used across diverse scientific domains.

We'll cover the following main topics:

- Typesetting an algorithm
- Printing a code listing
- Programming with Lua
- Creating graphs
- Writing quantities with units
- Drawing Feynman diagrams
- Writing chemical formulas
- Drawing molecules
- Representing atoms
- Drawing molecular orbital diagrams and atomic orbitals
- Printing a customized periodic table of elements
- Drawing electrical circuits

This chapter aims to showcase various packages through practical examples while providing insights into their utilization. For more intricate details, the manuals of these packages serve as a reference.

Typesetting an algorithm

An **algorithm** constitutes a fundamental concept within computer science. It represents a systematic set of step-by-step operations executed to accomplish specific tasks, such as calculations or data processing, e.g., sorting.

Algorithms can be visualized using a flow chart, which we made in *Chapter 6*, *Creating Graphics*. In this recipe, we will print an algorithm using **pseudocode** with syntax highlighting. Our example will show the calculations that display the **Mandelbrot** set, a visually stunning classic fractal generated by computations involving complex numbers.

How to do it...

We will utilize the `algorithmicx` package written by Szász János. We will break down the process into several small steps for more transparent comprehension. As usual, the complete code is available for download from `https://latex-cookbook.net`, eliminating the need for manual typing. At the end, you will see an image with the output. Consider switching between the output image and the quite comprehensive instructional steps to observe the incremental construction of the algorithm layout. Here it goes:

1. As usual, start with a document class. Load additional packages you intend to use; in this case, we need the `dsfont` and `mathtools` packages:

   ```
   \documentclass{article}
   \usepackage{dsfont}
   \usepackage{mathtools}
   ```

2. Load these three algorithm packages:

   ```
   \usepackage{algorithm}
   \usepackage{algorithmicx}
   \usepackage{algpseudocode}
   ```

3. You can define your own commands—in our case, a statement for local variables:

   ```
   \algnewcommand{\Local}{\State\textbf{local
     variables: }}
   ```

4. We define any other macros we need. We'll also create a shortcut `\Let` command for recurring variable assignments using the `\State` command. To ensure proper left-hand side alignment, we'll use the `\mathmakebox` command to put an argument in a box with a minimum width of `1em`:

   ```
   \newcommand{\Let}[2]{\State
     $\mathmakebox[1em]{#1} \gets #2$}
   ```

5. Start the document:

```
\begin{document}
```

6. Open an `algorithm` environment:

```
\begin{algorithm}
```

7. Provide a caption and a label for cross-referencing:

```
  \caption{Mandelbrot set}
  \label{alg:mandelbrot}
```

8. Start an `algorithmic` environment with an *n* option for numbering every *n*th line. We choose `1` as this option, numbering each single line:

```
  \begin{algorithmic}[1]
```

9. You can state requirements if any exist:

```
    \Require{$c_x, c_y, \Sigma_{\max} \in \mathds{R},
      \quad i \in \mathds{N}, \quad i_{\max} > 0,
      \quad \Sigma_{\max} > 0$}
```

10. We write down the function name with arguments:

```
    \Function{mandelbrot}{$c_x, c_y, i_{\max},
              \Sigma_{\max}$}
```

11. Now we use our own `\Local` macro for declaring local variables:

```
      \Local{$x, y, x_1, y_1, i, \Sigma$}
```

12. We use a statement to initialize local variables:

```
      \State $x, y, i, \Sigma \gets 0$}
```

13. We can add a comment to the line:

```
      \Comment{initial zero value for variables}
```

14. Now, write down a `while` loop that contains assignments:

```
      \While{$\Sigma \leq \Sigma_{\max}$
             and $i < i_{\max}$}
        \Let{x_1}{x^2 - y^2 + c_x}
        \Let{y_1}{2xy + c_y}
        \Let{x}{x_1}
        \Let{y}{y_1}
```

```
          \Let{\Sigma}{x^2 + y^2}
        \EndWhile
```

15. Add an `if` … `then` conditional statement:

```
        \If{$i < i_{\max}$}
          \State \Return{$i$}
        \EndIf
```

16. We specify a return value and end the function:

```
        \State \Return{0}
      \EndFunction
```

17. End all open environments and the document:

```
    \end{algorithmic}
  \end{algorithm}
  \end{document}
```

18. Compile and take a look at the outcome:

```
Algorithm 1 Mandelbrot set
Require: $c_x, c_y, \Sigma_{\max} \in \mathbb{R}, \quad i \in \mathbb{N}, \quad i_{\max} > 0, \quad \Sigma_{\max} > 0$
 1: function MANDELBROT($c_x, c_y, i_{\max}, \Sigma_{\max}$)
 2:     local variables: $x, y, x_1, y_1, i, \Sigma$
 3:     $x, y, i, \Sigma \leftarrow 0$                    ▷ initial zero value for variables
 4:     while $\Sigma \leq \Sigma_{\max}$ and $i < i_{\max}$ do
 5:         $x_1 \leftarrow x^2 - y^2 + c_x$
 6:         $y_1 \leftarrow 2xy + c_y$
 7:         $x \leftarrow x_1$
 8:         $y \leftarrow y_1$
 9:         $\Sigma \leftarrow x^2 + y^2$
10:     end while
11:     if $i < i_{\max}$ then
12:         return $i$
13:     end if
14:     return 0
15: end function
```

Figure 11.1 – An algorithm with pseudocode

How it works...

The `algorithm` environment is a wrapper that allows the algorithm to float to a good position, just like figures and tables. So, page breaks within algorithms are avoided and pages can be well filled. Furthermore, it supports captions and labels for cross-referencing and adds the `\listofalgorithms` command, which generates a list of algorithms similar to a list of figures.

The inner `algorithmic` environment does the specific typesetting. It supports commands that are commonly used in algorithm descriptions. These are the commands we used:

- The `\Require` command is for a short list of requirements for the algorithm. The output starts with the **Require:** keyword in bold.
- The `\Function` command prints the **function** keyword in bold, followed by the function name in small caps and parameters in parentheses. The `\EndFunction` command prints **end function** in bold.
- The `\While` and `\EndWhile` commands generate a loop in the manner **while** ... **do** ... **end while**.
- The `\If` and `\EndIf` commands generate a conditional statement in the manner **if** ... **then** ... **end if**.
- The `\State` command starts a new algorithm line with a suitable indentation.

The complete set of commands is described in the package manual, accessed by inputting `texdoc algorithmicx` or going to `https://texdoc.org/pkg/algorithmicx`.

There's more...

There's more than the pseudocode style. You can use the `algpascal` layout, which supports **Pascal** language syntax and performs the block indentation automatically. To achieve this, replace the command `\usepackage{algpseudocode}` with the command `\usepackage{algpascal}`. In the same way, you can use the `algc` layout instead, which is the equivalent of the **C** language.

Experienced users may define their own command sets. This and existing layout features are described in the package manual.

Printing a code listing

Documentation often includes code snippets, as well as computer science theses. While the first recipe of this chapter handed pseudocode for algorithms and the subsequent recipe did actual programming, our focus now shifts to typesetting the code. To keep it concise, we'll use a simple "hello world" program as an example.

How to do it...

We'll utilize the `listings` package initially written by Carsten Heinz and designed explicitly for this task. Follow these steps:

1. Start with any `document` class:

   ```
   \documentclass{article}
   ```

2. Load the `listings` package:

```
\usepackage{listings}
```

3. Begin the document:

```
\begin{document}
```

4. Begin a `lstlisting` environment with an option for the language:

```
\begin{lstlisting}[language = C++]
```

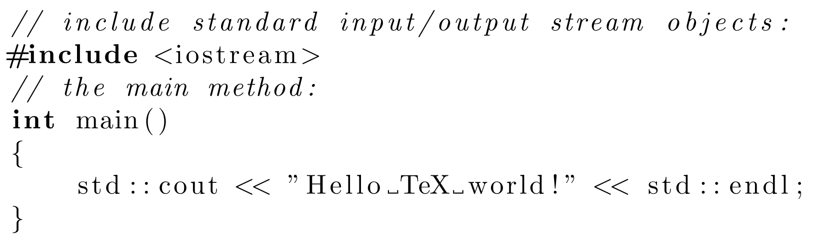
5. Continue with the code you would like to print:

```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main()
{
    std::cout << "Hello TeX world!" << std::endl;
}
```

6. End the `lstlisting` environment and the document:

```
\end{lstlisting}
\end{document}
```

7. Compile and have a look at the output:



```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main()
{
    std::cout << "Hello␣TeX␣world!" << std::endl;
}
```

Figure 11.2 – A C++ listing

How it works...

The fundamental steps are straightforward:

1. Load the `listings` package.
2. Enclose each code listing within a `lstlisting` environment, optionally specifying the language as seen previously.

The manual provides a comprehensive list of supported languages, continually expanding for over 25 years. You can define your own language style or find one for your favorite language online.

Commands and environments within the listings package use the `\lst` prefix to avoid naming conflicts with other packages.

You can tailor the appearance of all your listings with a single command:

```
\lstset{key1 = value1, key2 = value2}
```

This command offers an extensive `key=value` interface with numerous keys. Let's look at how to use it, focusing on particularly useful keys.

Modify the preceding example in this way:

1. Add the `xcolor` package to your document preamble:

   ```
   \usepackage{xcolor}
   ```

2. Load the `inconsolata` package to utilize an excellent typewriter font:

   ```
   \usepackage{inconsolata}
   ```

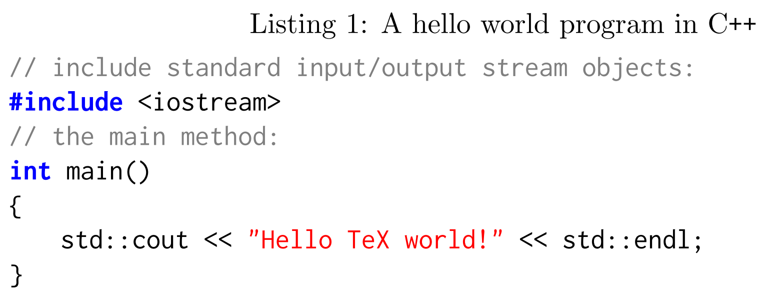
3. Define macros, such as the programming language logo, to maintain a consistent appearance:

   ```
   \newcommand{\Cpp}{C\texttt{++}}
   ```

4. After `\usepackage{listings}`, insert settings via `key=value`:

   ```
   \lstset{
     language         = C++,
     basicstyle       = \ttfamily,
     keywordstyle     = \color{blue}\textbf,
     commentstyle     = \color{gray},
     stringstyle      = \color{green!70!black},
     stringstyle      = \color{red},
     columns          = fullflexible,
     numbers          = left,
     numberstyle      = \scriptsize\sffamily\color{gray},
     caption          = A hello world program in \Cpp,
     xleftmargin      = 0.16\textwidth,
     xrightmargin     = 0.16\textwidth,
     showstringspaces = false,
     float,
   }
   ```

5. With these settings, you can now utilize the `\begin{lstlisting}` command without additional arguments. Compile your adjusted example and observe the changes:



Listing 1: A hello world program in C++

```
// include standard input/output stream objects:
#include <iostream>
// the main method:
int main()
{
    std::cout << "Hello TeX world!" << std::endl;
}
```

Figure 11.3 – A customized listing

There's more...

Like the standard LaTeX `verbatim` environment and the `\verb` command, `lstlisting` provides a companion for embedding small code snippets inline—the command `\lstinline` does it. Write it as follows:

```
Use \lstinline!#include <iostream>! for
including i/o streams.
```

You can use any character as a delimiter instead of the exclamation mark as long as it doesn't appear in the code snippet.

For longer listings, you can save them in external files. Instead of the standard `\input` command, use the following command:

```
\lstinputlisting[options]{filename}
```

The same options available for the `lstlisting` environment can be applied here. For instance, the following command includes only lines `4` to `10`:

```
\lstinputlisting[firstline=4, lastline=10]{filename}
```

This allows the breakdown of lengthier listings along with explanatory text.

Similar to the regular LaTeX `\listoffigures` command, you can generate a list of listings with their captions using the `\lstlistoflistings` command.

Programming with Lua

While LuaTeX comes with numerous more advancements regarding, for example, font support and **MetaPost** graphics support, we will focus on pure Lua programming in this section to carve out benefits to program and use algorithms.

TeX, primarily a text-processing language, has limited programming capabilities and needs advanced data-handling functionalities. That makes general-purpose programming a challenge. To address this, TeX developers sought a scripting language to add modern programming capabilities. Their strategic choice was **Lua**, a versatile, lightweight, and highly portable scripting language designed to be embedded in other applications. This decision led to the development of **LuaTeX**, a new TeX engine that, combined with the LaTeX format, is called **LuaLaTeX**.

While LuaTeX offers various advancements, including enhanced font and MetaPost support, this recipe uses pure Lua programming to run algorithms directly within our LaTeX document.

Note

Use the LuaLaTeX compiler option in your LaTeX editor for the examples in this chapter.

How to do it...

Let's implement an iterative algorithm, **Heron's method** (also called the **Babylonian method**), to calculate the square root of a number. This method is detailed at `https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Heron's_method`. In essence, it works as follows:

1. Start with an estimate, *x*, which could approximate the square root of *n*.
2. If *x* is smaller than the actual square root, then *n*/*x* is larger than the root since *x***x* = n is our objective. Conversely, if *x* is greater than the root, *n*/*x* would be smaller. To refine our approximation, we select the average of *x* and *n*/*x* as the new value for *x*.
3. Go back to *step 2* and repeat this process multiple times.

Let's see how to do this calculation in LaTeX! Follow these steps to get the square root of 2:

1. Start with any document class:

```
\documentclass{article}
```

2. Load the `luacode` package for extended Lua support:

```
\usepackage{luacode}
```

3. Begin the document with some text:

```
\begin{document}
The value of $\sqrt{2}$ is \approx
```

4. Open a `luacode` environment:

```
\begin{luacode}
```

5. Now use Lua—declare your variable `x` with an initial value of `1`:

```
  local x = 1
  for i=1,10 do
    x = (x + 2/x)/2
  end
```

6. Print the result to the LaTeX document:

```
  tex.print(x)
```

7. End the `luacode` environment and the document:

```
\end{luacode}
\end{document}
```

8. Compile using LuaLaTeX and have a look at the output:

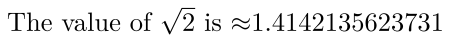

The value of $\sqrt{2}$ is $\approx$1.4142135623731

Figure 11.4 – Output of text and calculated result

How it works...

We utilized a `luacode` environment from the package by Manuel Pégourié-Gonnard to embed a Lua program in our LaTeX document. We defined a variable and employed a for loop to compute the final value through ten iterative repetitions. Using the `tex.print` command, we displayed the value of the Lua variable within our document.

The Lua language is comprehensively documented at `https://www.lua.org/docs.html`.

While we went through this example to understand how to embed Lua code, there's an alternate calculation method. Lua features a mathematical library that provides various mathematical functions. For instance, to print the value of the square root of 2, you could execute this command:

```
tex.print(math.sqrt(2))
```

You don't need the `luacode` environment and package for small Lua code snippets such as this. You can execute any single-line Lua code using the `\directlua` command. You can modify the previous code to have this as the LaTeX document body:

```
The value of $\sqrt{2}$ is
\approx\directlua{tex.print(math.sqrt(2))}.
```

The output will be the same as in the previous figure. The mathematical library is documented in *Section 6.7* of the current Lua reference manual, with version 5.4 available at `https://www.lua.org/manual/5.4/manual.html#6.7`, and you can find examples for all functions at `http://lua-users.org/wiki/MathLibraryTutorial`.

There's more...

Let's explore a more comprehensive example that shows the excellent integration of Lua with LaTeX. Utilizing the `pgfplots` package developed by Christian Feuersänger alongside Lua, we'll generate an image illustrating the Mandelbrot set through the algorithm outlined at the beginning of this chapter. Follow these steps:

1. Start with any document class. I opted again for the `standalone` class with some white margin:

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `pgfplots` package and initialize it with options for the plot width and the version for compatibility:

   ```
   \usepackage{pgfplots}
   \pgfplotsset{width=7cm, compat=1.18}
   ```

3. Load the `luacode` package and open a `luacode` environment:

   ```
   \usepackage{luacode}
   \begin{luacode}
   ```

4. Enter the following Lua code for a function declaration following the algorithm detailed at the beginning of this chapter. Define and initialize local variables with an initial value of zero, perform calculations within a while loop, and transfer the result to TeX:

   ```
     function mandelbrot(cx, cy, imax, smax)
       local x, y, x1, y1, i, s
       x, y, i, s = 0, 0, 0, 0
       while (s <= smax) and (i < imax)  do
         x1 = x * x - y * y + cx
         y1 = 2 * x * y + cy
         x = x1
         y = y1
   ```

```
          i = i + 1
          s = x * x + y * y
        end
        if (i < imax) then
          tex.print(i)
        else
          tex.print(0)
        end
      end
```

5. End the `luacode` environment, begin the document body, and open a `tikzpicture` environment:

```
\end{luacode}
\begin{document}
\begin{tikzpicture}
```

6. Like in the previous chapter, open a `pgfplots` `axis` environment with the following options in square brackets:

```
\begin{axis}[
  colorbar,
  point meta max = 30,
  tick label style = {font=\tiny},
  view={0}{90}]
```

7. Use the `\addplot3` command to generate a 3D plot. The Z-values of the plot command are calculated using the `\directlua` command with the `mandelbrot` function:

```
  \addplot3 [surf, domain = -1.5:0.5, shader = interp,
             domain y = -1:1, samples = 200]
      { \directlua{mandelbrot(\pgfmathfloatvalueof\x,
          \pgfmathfloatvalueof\y,10000,4)} };
```

8. Close the `axis` and `tikzpicture` environment and finish the document:

```
  \end{axis}
\end{tikzpicture}
\end{document}
```

9. Compile the document using LuaLaTeX. The complex calculation may take some time. This is the generated plot:

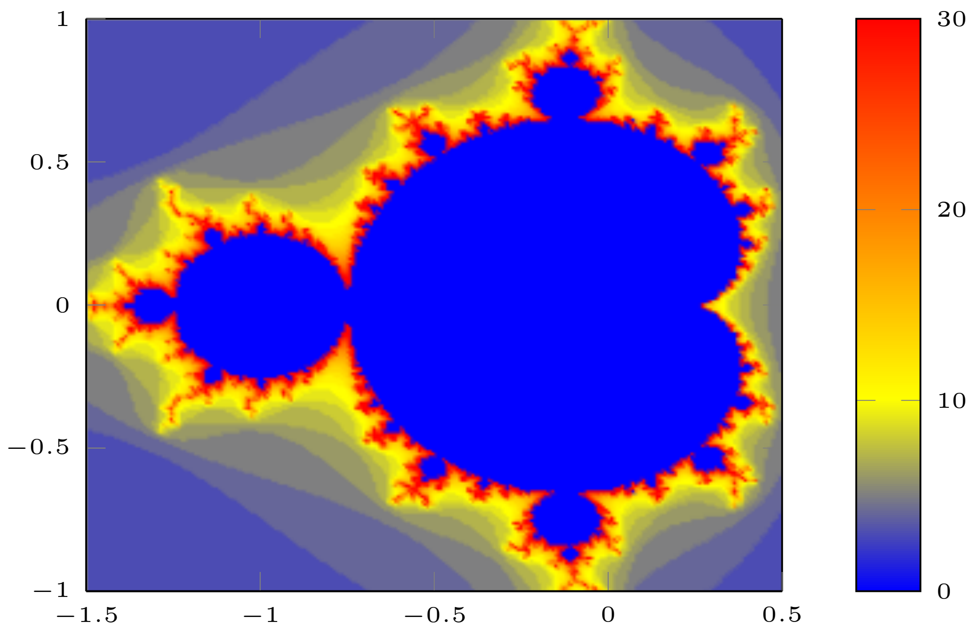


Figure 11.5 – The Mandelbrot set

How it works...

We combined a `luacode` environment, where we defined a Lua function, with the `\directlua` command in the document.

We utilized the `pgfplots` package to iterate through (*x*,*y*) values. The result of the Lua `mandelbrot` function is a color. While we aimed to generate a two-dimensional image, the result is used as the Z-value in a 3D plot. That Z-value is colored in proportion to its value. The picture looks two-dimensional because we chose a viewing angle directly above the xy-plane.

Creating graphs

Graph theory, commonly employed in fields such as operations research and computer science, typically involves models and drawings primarily composed of repeated vertices, edges, and labels. There are LaTeX packages that help efficiently generate consistent graphs.

How to do it...

The `tkz-graph` package developed by Alain Matthes provides a user-friendly interface, various preconfigured styles, and extensive customization options. Let's start with a minimal example:

1. Begin with any document class. In this case, I've opted for the standalone class to generate a compact PDF containing the desired image. Additionally, I've included an option for a border value to create a slight margin around the graph.

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `tkz-graph` package:

   ```
   \usepackage{tkz-graph}
   ```

3. Define the distance between two vertices in cm:

   ```
   \SetGraphUnit{3}
   ```

4. Begin the document body:

   ```
   \begin{document}
   ```

5. Open a `tikzpicture` environment. Here, you may optionally rotate the graph, giving a value in degrees:

   ```
   \begin{tikzpicture}[rotate=18]
   ```

6. Define a set of vertices. Optionally, choose a shape for their positioning:

   ```
     \Vertices{circle}{A,B,C,D,E}
   ```

7. Decide which vertices shall be connected by edges in which order:

   ```
     \Edges(A,B,C,D,E,A,D,B,E,C,A)
   ```

8. Close the `tikzpicture` environment and end the document:

   ```
   \end{tikzpicture}
   \end{document}
   ```

9. Compile and have a look at the picture:

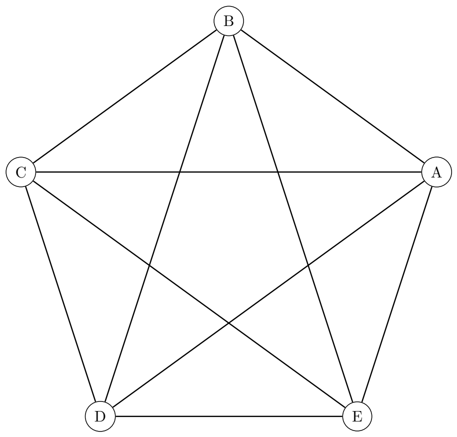


Figure 11.6 – A basic graph

How it works...

After loading the `tkz-graph` package, we used the `\SetGraphUnit` command to choose a value in centimeters for the distance between the vertices because the default value of 1 cm is pretty small. We did this in the preamble, so it's applied to all graphs consistently. We can also use the `\SetGraphUnit` command in the document within the `tikzpicture` environment. In that case, it applies only to the current TikZ picture.

We used the `\Vertices` command to define a set of vertices. We can name them using capital letters, small letters, numbers, or even mathematical expressions such as `$x_1$`. The initial argument defines the geometric structure of the graph, providing various options:

- `line`: This option places the vertices along a line.
- `circle`: This option places all vertices on a circle.
- `square`: With this option, the vertices are positioned as corners of a square. This should be used only with exactly four vertices.
- `tr1`, `tr2`, `tr3`, `tr4`: The vertices are placed in four different types of rectangular triangle formations. Use it with precisely three vertices.

While that doesn't look like many choices, you can use several `\Vertices` commands to build a complex graph. There's a node option to help with positioning. First, define a node or, better, a coordinate as follows:

```
\coordinate (a) at (4,2);
```

Then, you can use the `Node` option to place the vertices starting at that node or coordinate position like so:

```
\Vertices[Node]{square}{a,b,c,d}
```

That helps in assembling larger graphs by combining multiple smaller graphs.

Finally, we used the `\Edges` command that generates a sequence of edges by connecting a list of vertices in their given order.

We can easily modify the appearance of the graph using a single command. Insert the following command into your document preamble after you loaded the `tkz-graph` package:

```
\GraphInit[vstyle=Shade]
```

Compile and see how the graph has changed:

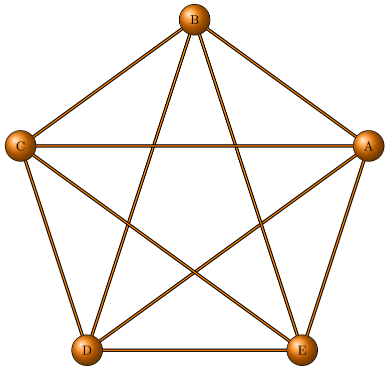


Figure 11.7 – A graph with a shading style

The `vstyle` option provides various graph styles, defining how vertices are displayed, and some styles produce non-regular edges:

- `Empty`: This option gives simple vertices without a circle or any border.
- `Classic`: Using this option, vertices are displayed as filled circles, and the vertex name is positioned outside of the circle.
- `Normal`: This option gives circular vertices with the vertex name inside.
- `Simple`: This option generates black-filled circular vertices without printing the vertex names.
- `Art`: This option turns vertices to shaded balls without printing the vertex name in orange by default. The edges are regular lines but colored orange, too.
- `Shade Art`: This works like the `Art` option but with thicker orange lines and black borders for edges.

- `Shade`: This option looks like `Shade Art` but has vertex names inside the balls, just as you saw in *Figure 11.7*.
- `Hasse`: This style produces circular, non-filled vertices without printing names.
- `Dijkstra`: This style prints circular vertices with the name inside
- `Welsh`: This style produces circular vertices with the name outside the vertex node.

The edges are regular black lines except with the `Art`, `Shade`, and `Shade Art` options.

There's more...

You can create your own graph style or customize the existing styles. Let's explore the additional features with a more complex example. Follow these steps:

1. Like in the previous example, start with the document class, load the `tkz-graph` package, choose a basic style, and set a distance between the nodes in centimeters:

```
\documentclass{standalone}
\usepackage{tkz-graph}
\GraphInit[vstyle = Shade]
\SetGraphUnit{5}
```

2. Modify the styles called `VertexStyle`, `EdgeStyle`, and `LabelStyle`. Use the `.append style` syntax to add new settings to the pre-defined style without replacing them. You can use regular TikZ options as follows:

```
\tikzset{
  VertexStyle/.append style =
    { inner sep = 5pt, font = \Large\bfseries},
  EdgeStyle/.append style   = {->, bend left},
  LabelStyle/.append style  =
    { rectangle, rounded corners, draw,
      minimum width = 2em, fill = yellow!50,
      text = red, font = \bfseries}
}
```

3. You can also use the `\renewcommand` macro to modify style elements like this for a different vertex ball color:

```
\renewcommand{\VertexBallColor}{blue!30}
```

4. Begin the document and open a `tikzpicture` environment:

```
\begin{document}
\begin{tikzpicture}
```

5. Declare a first vertex B:

```
    \Vertex{B}
```

6. Set a vertex A to the west (WE) and C to the east (EA):

```
    \WE(B){A}
    \EA(B){C}
```

7. Draw edges between the vertices:

```
    \Edge[label = 1](A)(B)
    \Edge[label = 2](B)(C)
    \Edge[label = 3](C)(B)
    \Edge[label = 4](B)(A)
```

8. Add loops, which are edges from a vertex to itself:

```
    \Loop[dist = 4cm, dir = NO, label = 5](A.west)
    \Loop[dist = 4cm, dir = SO, label = 6](C.east)
```

9. Adjust the bend angle of the edges for the final two wider edges:

```
    \tikzset{EdgeStyle/.append style = {bend left = 50}}
    \Edge[label = 7](A)(C)
    \Edge[label = 8](C)(A)
```

10. End the picture and the document:

```
  \end{tikzpicture}
  \end{document}
```

11. Compile and have a look at the result:

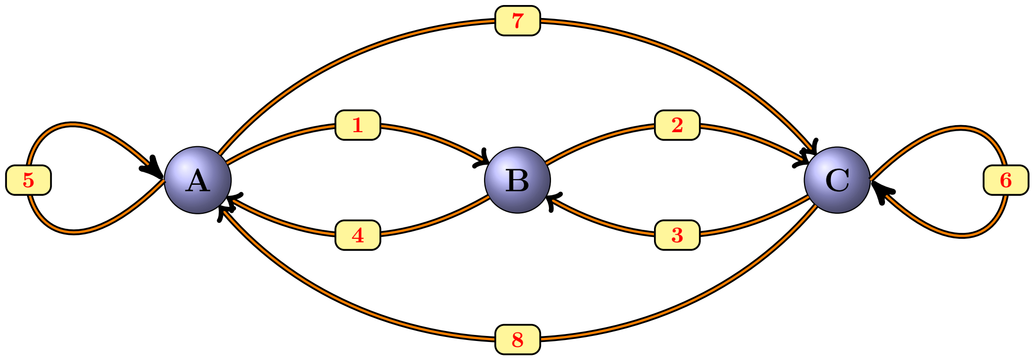


Figure 11.8 – A customized graph

How it works...

Similar to other recipes in this book, the basic procedure is as follows:

1. Define styles.
2. Position vertices.
3. Add edges.
4. Repeat if needed.

For positioning vertices, there's a simple syntax:

```
<direction>(B){A}
```

`<direction>` can be as follows:

- `\EA` for placing B to the east of `A`
- `\WE` for positioning it to the west
- `\NO` for positioning it to the north
- `\SO` for positioning it to the south
- `\NOEA`, `\NOWE`, `\SOEA`, and `\SOWE` work as combinations of the preceding directional commands

The entire `\Edge` syntax is as follows:

```
\Edge[options](vertex1)(vertex2)
```

Options can be line width, labels, styles, and colors. For such detailed options, please refer to the package manual available by running `texdoc tkz-graph` via the command line or online at `https://texdoc.org/serve/tkz-graph/0`.

Writing quantities with units

Unlike pure mathematics, we often encounter units alongside quantities in natural sciences such as chemistry, physics, and engineering. It's essential to distinguish units from variables. Consider this example: let's create a formula that multiplies the speed `s` of one meter per second by the factor `m`. At first glance, it might seem straightforward like this:

```
\( m \cdot s = m \cdot 1 m s^{-1} \)
```

The LaTeX standard output would be as follows:

$$m \cdot s = m \cdot 1ms^{-1}$$

Figure 11.9 – A bad example of printing variables and units

What do you think about this? Units and variables seem identical. Imagine multiplying both sides of the equation by `s` or dividing by `m`... it becomes pretty perplexing. Furthermore, our space between `1` and `m` has been lost.

To adhere to common standards in writing, we often require the following:

- Upright presentation of units to differentiate them from italicized math variables
- A small space between a quantity and its accompanying unit
- Customizable appearance without changing the formula code, especially when a journal requests a different style
- Semantic writing—replacing abbreviations such as "m" and "s" with complete terms such as "meters" and "seconds"—enhances clarity
- Intelligent parsing of numbers within quantities
- Incorporating features such as striking out or highlighting to explain a calculation effectively

Is it possible to achieve all of these requirements? Definitely!

How to do it...

The `siunitx` package by Joseph Wright offers methods to align with international standards for unit systems while allowing customization to suit various typographic styles.

Now, let's rectify the formula mentioned earlier by following these steps:

1. Start with any document class:

   ```
   \documentclass{article}
   ```

2. Load the `siunitx` package:

   ```
   \usepackage{siunitx}
   ```

3. Begin the document:

   ```
   \begin{document}
   ```

4. Write the preceding formula but this time use the command `\SI{quantity}{units}`:

```
\( m \cdot s = m \cdot \qty{1}{\m\per\s} \)
```

5. End the document for now:

```
\end{document}
```

6. Compile and take a look:

$$m \cdot s = m \cdot 1\,\mathrm{m\,s^{-1}}$$

Figure 11.10 – Improved display of variables, values, and units

7. You can also opt for longer, more natural unit names to achieve the same result as mentioned earlier:

```
\( m \cdot s = m \cdot \qty{1}{\meter\per\second} \)
```

8. Let's adjust the reciprocal units. After loading the `siunitx` package, add the following line to your preamble:

```
\sisetup{per-mode = symbol}
```

9. Compile to see the difference:

$$m \cdot s = m \cdot 1\,\mathrm{m/s}$$

Figure 11.11 – Alternative display of units

10. If you want to emphasize changes, you can use the `cancel` and `color` packages. Add them to your preamble:

```
\usepackage{cancel}
\usepackage{color}
```

11. Let's test this together with scientific, exponential notation. So, modify your formula line as follows:

```
\( m \cdot s = m \cdot
  \qty{1e-3}{\cancel\m\highlight{red}\km\per\s} \)
```

12. Compile to see the latest result:

$$m \cdot s = m \cdot 1 \times 10^{-3}\,\cancel{\mathrm{m}}\,\mathrm{km}/\mathrm{s}$$

Figure 11.12 – Emphasizing in a formula

How it works...

The command `\qty{quantity}{units}` accomplishes two tasks:

- It interprets the quantity in its initial argument, effectively formatting numbers and comprehending complex numbers and exponential notations. The output formatting eliminates unnecessary spaces and groups large numbers into blocks of three with a thin space.
- It processes the units provided, ensuring proper typesetting with a thin space between the quantity and unit.

In essence, `\qty` combines two commands, which you also can use directly:

- `\num{numbers}` parses numbers in the argument and formats them properly.
- `\unit{units}` typesets the units. For example, `\unit{\kilo\gram\meter\per \square\second}`, or the shorter `\unit{\kg\m\per\square\s}`, gives the following:

$$\mathrm{kg\,m/s^2}$$

Figure 11.13 – Combined units

The package implements a basic set of SI standardized units via macros, including derived units. You can utilize `\meter`, `\metre`, `\gram`, and so on, as well as derived units such as `\newton`, `\watt`, `\hertz`, among many others. Even non-SI units are supported, such as `\hour` or `\hectare`. The package also supports common prefixes such as `\kilo`, `\mega`, and `\micro`. For a comprehensive list of features, refer to the detailed manual accessible via the `texdoc siunitx` command from the command line or by visiting `https://texdoc.org/pkg/siunitx`.

Drawing Feynman diagrams

A Feynman diagram is a mathematical visualization of the behavior of subatomic particles. There are several ways to generate them using LaTeX.

How to do it...

We will use the `tikz-feynman` package. The author documented it in *J. Ellis, 'TikZ-Feynman: Feynman diagrams with TikZ', (2016), arXiv:1601.05437 [hep-ph]*, and you can access the documentation executing `texdoc tikz-feynman` via the command line or at `https://texdoc.org/pkg/tikz-feynman`.

The positions of the vertices are calculated using Lua, so we must compile with **LuaLaTeX**. Follow these steps:

1. Start with any document class:

```
\documentclass[border=10pt]{standalone}
```

2. Load the `tikz-feynman` package:

```
\usepackage{tikz-feynman}
```

3. Load additional useful TikZ libraries and begin the document:

```
\usetikzlibrary{positioning,quotes}
\begin{document}
```

4. Utilize the `\feynmandiagram` command as follows:

```
\feynmandiagram [horizontal=a to b] {
  i1 [particle=$e^-$] -- [fermion] a
    -- [fermion] f1 [particle=$e^-$],
  a -- [photon, "$\gamma$", red, thick,
      momentum' = {[arrow style=red]$k$}] b,
  i2 [particle=$\mu^-$] -- [anti fermion] b
    -- [anti fermion] f2 [particle=$\mu^-$],
};
```

5. End the document:

```
\end{document}
```

6. Compile and look at the result:

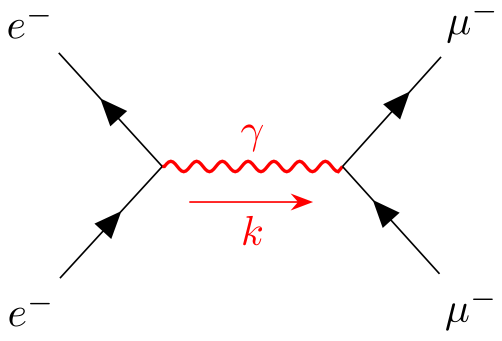


Figure 11.14 – A Feynman diagram

How it works...

We used `i1` and `f1` as initial and final nodes for one part and `i2` and `f2` for the other part. `a` and `b` are the nodes in the middle.

`fermion`, `anti fermion`, and `photon` are predefined line styles. You can also add TikZ styles.

The `particle` option is used to set labels. The `momentum` option adds further annotations.

There's more...

You can consider the alternative packages `feynmf` and `feynmp`. Visit `https://feynm.net` to explore a gallery of Feynman diagrams generated by various packages. Visit `https://wiki.physik.uzh.ch/cms/latex:feynman` to see a vast amount of examples.

Writing chemical formulas

The presentation of chemical formulas and equations differs from mathematical ones in several ways:

- Atomic symbols are represented by upright letters, distinct from italicized mathematical variables
- Numbers are often employed as subscripts, signifying the count of atoms.
- The alignment of numerous subscripts and superscripts is essential for a good formula layout
- Left subscripts and superscripts are also required in some cases
- Special symbols for bonds and arrows are necessary for chemical equations

However, accomplishing such requirements is challenging with basic LaTeX. Let's find a more effective solution.

How to do it...

We'll utilize the `chemformula` package that Clemens Niederberger wrote to practice chemical notation in LaTeX. Let's start:

1. Choose a `document` class, such as `scrartcl` of the **KOMA-Script** bundle, and the `chemformula` package and begin with the document:

```
\documentclass{scrartcl}
\usepackage{chemformula}
\begin{document}
```

2. Start with an unnumbered section to verify that formulas work in headings. Use the `\ch` command for writing formulas. Give atoms and numbers as arguments straight away, without the _ and ^ syntax used when writing mathematics:

   ```
   \section*{About \ch{Na2SO4}}
   \ch{Na2SO4} is sodium sulfate.
   ```

3. Electric charges of ions are written directly without using _ and ^:

   ```
   It contains \ch{Na+} and \ch{SO4^2-}.
   ```

4. Adducts can be denoted with a star or a dot, with numbers automatically identified as stoichiometric factors. Leave a blank space as a separator as here:

   ```
   \ch{Na2SO4 * 10 H2O} is a decahydrate.
   ```

5. Chemical formulas can also be used in math mode. For instance, create a centered equation with a forward arrow, also called a reaction arrow, indicated by `->`:

   ```
   \[
     \ch{Na2SO4 + 2 C -> Na2S + 2 CO2}
   \]
   ```

6. We can have it numbered, too, like math equations. This time, we use an equilibrium arrow, `<=>`:

   ```
   \begin{equation}
     \ch{Na2SO4 + H2SO4 <=> 2 NaHSO4}
   \end{equation}
   ```

7. If a number is left of an atom, it acts as a left subscript. But we can clearly indicate the meaning using _ and ^ before an atom, such as for isotopes:

   ```
   \section*{Isotopes}
   \ch{^{232}_{92}U140} is uranium-232.
   ```

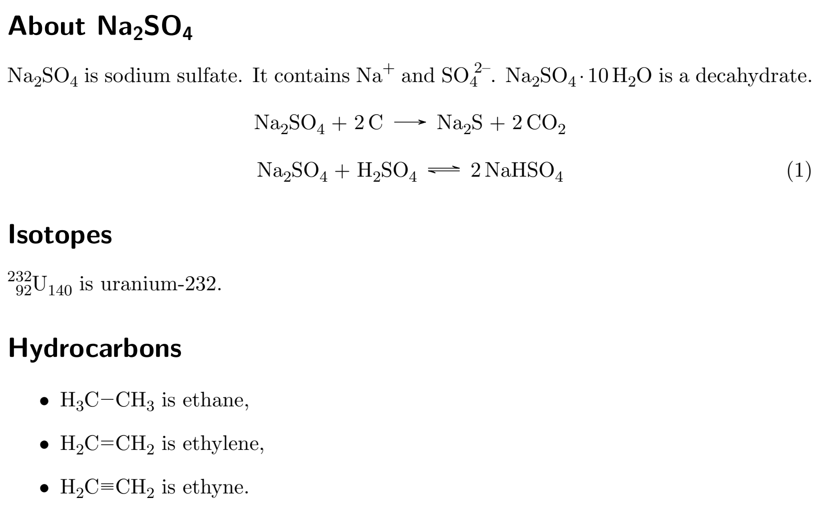
8. Different bond types (single, double, triple) are represented by `-`, `=`, or `+`, respectively. We can see this in a list of hydrocarbons:

   ```
   \begin{itemize}
     \item \ch{H3C-CH3} is ethane,
     \item \ch{H2C=CH2} is ethylene,
     \item \ch{H2C+CH2} is ethyne.
   \end{itemize}
   ```

9. That's enough for now, let's finish the document:

   ```
   \end{document}
   ```

10. Compile and see what you have done:



About Na_2SO_4

Na_2SO_4 is sodium sulfate. It contains Na^+ and SO_4^{2-}. $Na_2SO_4 \cdot 10\,H_2O$ is a decahydrate.

$$Na_2SO_4 + 2\,C \longrightarrow Na_2S + 2\,CO_2$$

$$Na_2SO_4 + H_2SO_4 \rightleftharpoons 2\,NaHSO_4 \qquad (1)$$

Isotopes

${}^{232}_{92}U_{140}$ is uranium-232.

Hydrocarbons

- $H_3C{-}CH_3$ is ethane,
- $H_2C{=}CH_2$ is ethylene,
- $H_2C{\equiv}CH_2$ is ethyne.

Figure 11.15 – Chemical formulas

How it works...

The input syntax is designed to be natural and straightforward:

- Atoms are represented by letters
- Numbers are automatically formatted as subscripts, signifying the number of atoms in the formula
- Stoichiometric numbers, representing molecule quantities, precede the molecule with a space in between

This simplicity not only aids in typing but also allows effortless copy-pasting from PDFs, Word documents, or the internet.

The most common bonds are written as follows:

- `-` represents a single bond
- `=` indicates a double bond
- `+` signifies a triple bond

The following syntax defines reaction arrows:

- `->`, `<-`: These draw regular arrows pointing to the right or the left
- `-/>`, `</-`: These draw broken arrows pointing to the right or the left (do not react)
- `<->`: This draws a resonance arrow (arrows with tips at the left and the right)

- `<>`: This draws a right-facing arrow at the top and a left-facing arrow under it
- `<=>`: This draws an equilibrium arrow (half of an arrow tip at each side)
- `<=>>`: This draws an equilibrium arrow with a tendency to the right, so the top arrow to the right is larger
- `<<=>`: This draws an equilibrium arrow with a tendency to the left, so the lower arrow to the left is larger

You can incorporate mathematical equations, chemical expressions, or text above or below arrows by using this syntax:

```
<=>[\text{above}] [\text{below}]
```

The package manual elaborates on more arrow types and additional features. Access it using the command `texdoc chemformula` using the command line or open it at `https://texdoc.org/pkg/chemformula`.

There's more...

The `mhchem` package operates similarly but varies in certain aspects, as outlined in the `chemformula` manual. The newer `chemformula` package was designed for enhancements and is part of the `chemmacros` bundle, which brings even more features for chemical notation.

A comprehensive collection of TeX chemistry packages, along with descriptions, is accessible at `https://www.cnltx.de/known-packages`.

There's another package list on CTAN: `https://ctan.org/topic/chemistry`.

We'll explore another exceptional package for drawing molecules in our upcoming recipe.

Drawing molecules

In the previous example, we practiced writing molecular formulas. Now, let's delve into visualizing them. We'll create a visual representation of a cluster of atoms interconnected by various types of lines.

How to do it...

This seemingly complex task becomes much simpler with the `chemfig` package developed by Christian Tellechea. It offers a concise syntax for rendering molecular structures. Let's create a few:

1. Start with any document class and load the `chemfig` package:

   ```
   \documentclass{article}
   \usepackage{chemfig}
   ```

2. Let's organize molecules in a table. To do this, widen the rows slightly and initiate a `tabular` environment with a column aligned to the right and another to the left:

```
\renewcommand{\arraystretch}{1.5}
\begin{tabular}{rl}
```

3. For molecules, use the `\chemfig` command. Represent atoms as letters and depict a single bond using a dash:

```
    Hydrogen: & \chemfig{H-H} \\
```

4. Depict a double bond using an equal sign:

```
    Oxygen:   & \chemfig{O=O} \\
```

5. Use a tilde for a triple bond:

```
    Ethyne:   & \chemfig{H-C~C-H}
```

6. End the table and add some space:

```
\end{tabular}
\qquad
```

7. Enclose branches within parentheses. Incorporate options using square brackets separated by commas. The first option indicates an angle. As we'll see later, you can specify multiples of 45 degrees or arbitrary angles. The second option signifies a factor for interatomic distance. We'll set it to `0.8` for a more compact drawing. Use this for the methane structure:

```
Methane: \chemfig{[,0.8]C(-[2]H)(-[4]H)(-[6]H)-H}
```

8. Finish the document:

```
\end{document}
```

9. Compile and take a look at the drawings:

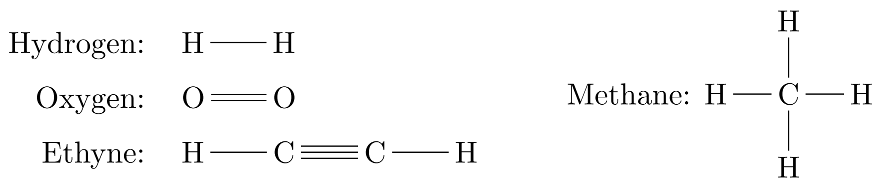


Figure 11.16 – Visual representations of molecules

How it works...

The `chemfig` employs TikZ for its drawing functions, handling the bounding box automatically to prevent overlap with other text. Experienced users have the flexibility to embed TikZ code if needed.

The primary command is `\chemfig`, which requires an argument consisting of the following arguments:

- Letters for atoms
- Symbols for bonds, such as `-`, `=`, and `~` for simple, double and triple bonds, respectively
- Options for bonds in square brackets, separated by commas
- Branches of atoms and bonds within parentheses

The most crucial option for bonds is the angle. It can be specified as follows:

- An integer number representing a multiple of 45 degrees, such as `[2]` for 90 degrees
- An absolute angle in degrees, indicated by a double colon, such as `[:60]` for 60 degrees
- A relative angle in degrees, marked by two double colons, such as `[::30]` for 30 degrees in relation to the previous bond

Positive and negative numbers are allowed.

A branch enclosed in parentheses allows you to open a path using an opening parenthesis, structure it as shown previously, and conclude it using a closing parenthesis. This returns you to the same position from where the branch started.

Tip

In complex molecules, find the longest chain and draw it first. Then, add the branches. Use relative angles for easy rotation of the entire molecule.

There's more...

There are further features we should take a look at.

Drawing rings

Molecular rings are commonly represented as regular polygons. They can be drawn using this syntax:

```
atom*n*(code)
```

Here, `n` indicates the number of sides of the polygon, and the `chemfig` code within parentheses depicts the structural arrangement within the ring.

For instance, the famous Benzene ring with all its atoms can be drawn this way:

```
\chemfig{C*6((-H)-C(-H)=C(-H)-C(-H)=C(-H)-C(-H)=)}
```

This line gives us the following picture:

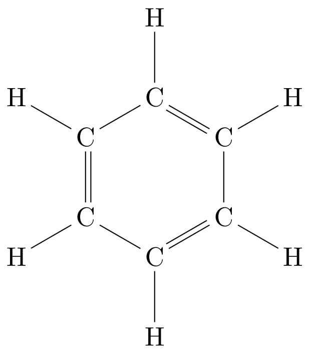


Figure 11.17 – The Benzene ring with all atoms

Naming molecules

Underneath a molecule, its name can be written using this syntax:

```
\chemname[distance]{\chemfig code}{name}
```

The optional `distance` value defines the distance to the baseline of the molecule, defaulting to 1.5 ex. For instance, to place the name `Benzene` below the carbon skeleton of a Benzene ring, use the following:

```
\chemname{\chemfig{*6(=-=-=-)}}{Benzene}
```

This will result in the following drawing:

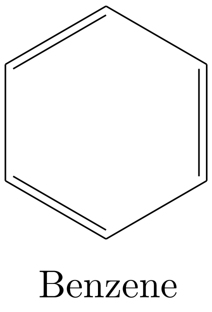


Figure 11.18 – The simplified Benzene ring with a label

Using building blocks

In LaTeX, you can create new macros using the `\newcommand` syntax. The `chemfig` package provides a similar feature—you can create your own macros for recurring use as follows:

```
\definesubmol{name}{code}
```

Now, we can use this macro in formulas by writing `!name` as a shortcut. For instance, this defines a molecular section with a carbon atom and two hydrogen atoms:

```
\definesubmol{C}{-C(-[2]H)(-[6]H)}
```

We can use the `!C` shortcut to draw the Pentane molecule:

```
\chemfig{H!C!C!C!C!C-H}
```

Remarkably, this concise code generates a considerably large molecule representation:

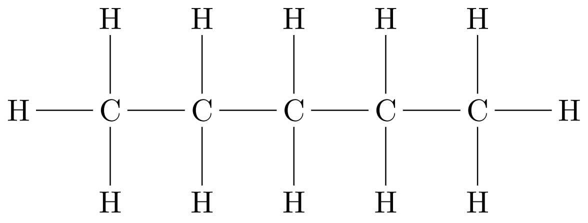


Figure 11.19 – The Pentane molecule

Applying style options

We can apply various style options to molecule drawings. The `\chemfig` command takes one optional argument in square brackets, which is a list of `key=value` options, and a mandatory argument for the molecule code in curly braces. It looks like this:

```
\chemfig[key1=value1, key2=value2, ...]{code}
```

Here are two commonly used styles:

- `chemfig style`: This is a list of options that apply to the entire `tikzpicture` environment of the molecule, grouped in braces
- `atom style`: This is a list of options for the atom nodes, again grouped in braces

For instance, applying these options would scale the entire picture and set the nodes to appear blue:

```
\chemfig[chemfig style = {scale=1.5, transform shape},
  atom style = {color=blue}]{H-C~C-H}
```

These options would result in thicker lines and a 15-degree rotation of the nodes:

```
\chemfig[chemfig style = {thick},
  atom style = {rotate=15}]{C(-[2]H)(-[4]H)(-[6]H)-H}
```

Here is the combined output from both lines:

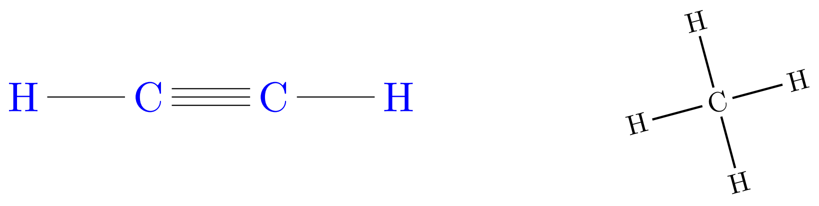


Figure 11.20 – Customized molecule drawings

Note

You may come across an outdated syntax on the internet: in a previous version of the package, the `\chemfig` command had two optional arguments, each enclosed in square brackets, as follows:

```
\chemfig[options for tikzpicture][options for nodes]{code}
```

The first argument's options modified the entire `tikzpicture` environment of the molecule, and the second argument's options adjusted the style of each node. Rewrite it using `chemfig` and `atom` styles.

For more options and features, please read the package manual by inputting `texdoc chemfig` using the command line or online at `https://texdoc.org/pkg/chemfig`.

Using ready-drawn carbohydrates

Though `chemfig` simplifies drawing, creating complex molecules can still be time-consuming, especially when dealing with numerous structures when you write lecture notes or a thesis covering carbohydrates. Fortunately, we don't have to start from scratch every time.

The `carbohydrates` package provides a lot of `chemfig`-drawn carbohydrates for you to use. It includes trioses, tetroses, pentoses, and hexoses in various models: the **Fischer** (full and skeleton), **Haworth**, and **chain** models. You can draw them as ring isomers and as chain isomers.

Let's have a look at how easy it becomes, for example, with glucose:

```
\glucose[model=fischer, chain]\quad
\glucose[model={fischer=skeleton}, chain]
```

This draws the Fischer models; the skeleton version doesn't show the H and C atoms:

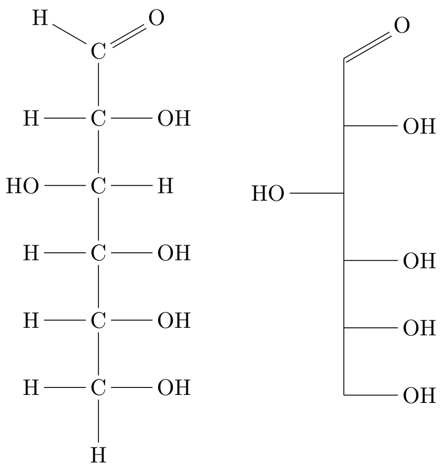


Figure 11.21 – Glucose molecules displayed using the Fischer models

Now let's draw glucose but with other models:

```
\glucose[model=haworth, chain]\hfill
\glucose[model=haworth, ring]\hfill
\glucose[model=chain, ring]
```

We get the following drawings:

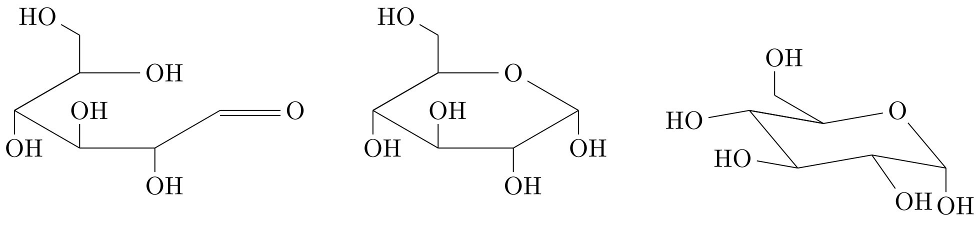


Figure 11.22 – Glucose molecules displayed using the Haworth and chain models

Already implemented are the following molecules:

- `\glycerinaldehyde` (triose)
- `\erythrose`, `\threose` (tetroses)
- `\ribose`, `\arabinose`, `\xylose`, `\lyxose` (pentoses)
- `\allose`, `\altrose`, `\glucose`, `\mannose`, `\gulose`, `\idose`, `\galactose`, `\talose` (hexoses)

The package manual tells you all the details about options and usage. You can open it using the command line by running the command `texdoc chemformula` or online at `https://texdoc.org/pkg/carbohydrates`.

Representing atoms

Now that we've mastered drawing molecules, shall we explore further? Can we draw atoms? Absolutely!

How to do it...

We'll utilize a package named after the renowned physicist Niels Bohr and written by Clemens Niederberger—the `bohr` package. Follow these steps:

1. Start with a document class, load the `bohr` package, and begin with the document:

   ```
   \documentclass{article}
   \usepackage{bohr}
   \begin{document}
   ```

2. Use the command `\bohr{number of electrons}{element name}`, to draw the Fluorine atom:

   ```
   \bohr{10}{F}
   ```

3. For the next drawing, adjust the nucleus radius as follows:

   ```
   \setbohr{nucleus-radius=1.5em}
   ```

4. With this adjustment, there's more space at the center for an ion symbol. In this instance, employ the `\bohr` command with an optional argument specifying the number of electron shells within square brackets. This will illustrate a sodium ion:

   ```
   \bohr[3]{10}{$\mathrm{Na^+}$}
   ```

5. That's all for now! Conclude the document:

   ```
   \end{document}
   ```

6. Compile to see the result:

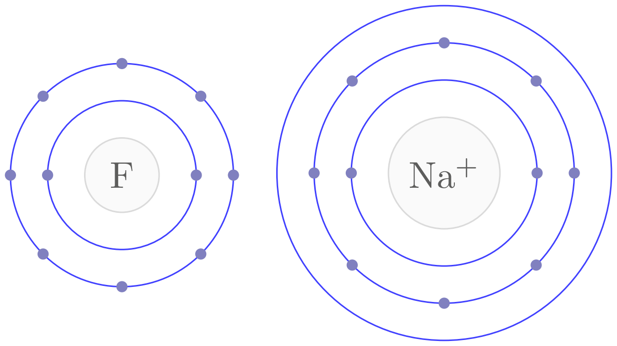


Figure 11.23 – Atoms and electrons

How it works...

It was pretty straightforward. However, I wanted to demonstrate how one can write about science effortlessly today.

After loading the package, all we required was this single command:

```
\bohr[number of shells]{number of electrons}{element name}
```

The `\setbohr` command provides a `key=value` interface for further fine-tuning. We'll skip over the extensive list of optional parameters to avoid overwhelming those who aren't working with physics or chemistry. You can read all customization details in the manual, which you can open by running `texdoc bohr` via the command line or online at `https://texdoc.org/pkg/bohr`.

Drawing molecular orbital diagrams and atomic orbitals

A **molecular orbital** (**MO**) diagram describes chemical bonding in molecules and displays energy levels. First, we will create such an MO diagram, and then we will draw atomic orbitals with a more visual approach.

How to do it...

We will use the `tikzorbital` package written by Germain Salvato-Vallverdu. These are the steps:

1. Start with any `document` class; we choose the `standalone` class here. Then load the `tikzorbital` package that implicitly loads TikZ:

   ```
   \documentclass[border=10pt]{standalone}
   \usepackage{tikzorbital}
   ```

2. Load the `positioning` and `quotes` TikZ libraries and begin the document:

```
\usetikzlibrary{positioning,quotes}
\begin{document}
```

3. Open a `tikzpicture` environment, and define a custom ^ style to get small, center-aligned sans-serif text where we want it:

```
\begin{tikzpicture}[note/.style =
  {align = center, font = \sffamily\scriptsize}]
```

4. Use the `\drawLevel` command to draw an energy level line, that we call `1s1`, with an electron visualized in the upward direction:

```
  \drawLevel[elec = up]{1s1}
```

5. Continue using `drawLevel` commands, now with a positioning coordinate and a width option:

```
  \drawLevel[elec = up, pos = {(5,0)}]{1s2}
  \drawLevel[elec = pair, pos = {(2,-2)},
    width = 2]{sigma}
  \drawLevel[pos = {(2,2)}, width = 2]{sigmastar}
```

6. Draw a dashed line between the various right and left anchors of the energy level lines:

```
  \draw[dashed]
    (right 1s1) -- (left sigma)
    (right 1s1) -- (left  sigmastar)
    (left  1s2) -- (right sigmastar)
    (left  1s2) -- (right sigma);
```

7. Draw labels for the energy levels:

```
  \node[left]  at (left 1s1) {{$1s_1$}};
  \node[right] at (right 1s2) {{$1s_2$}};
  \node[right] at (right sigma) {$\sigma$};
  \node[right] at (right sigmastar) {$\sigma^*$};
```

8. Print some text nodes for explanation using our `note` style:

```
  \node[below = 0.4cm of middle 1s1, note]
    {Atomic\\Orbital};
  \node[below = 0.4cm of middle 1s2, note]
    {Atomic\\Orbital};
  \node[below = 0.4cm of middle sigma, note]
    {Molecular Orbital};
```

9. Finish the drawing with an arrow indicating the energy level:

```
    \draw[very thick, -stealth] (-1.5,-2.5)
      to["Energy", note, sloped] (-1.5,2.5);
```

10. End the tikzpicture environment and the document:

```
  \end{tikzpicture}
  \end{document}
```

11. Compile and look at the outcome:

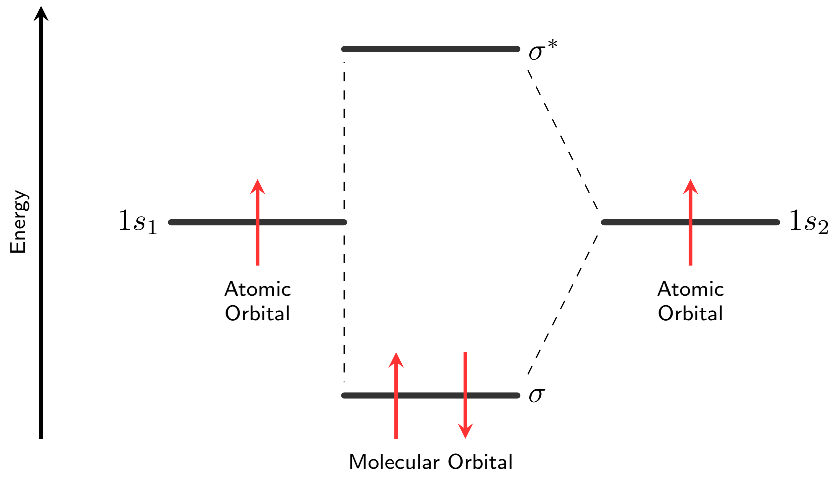


Figure 11.24 – A molecular orbital diagram

How it works...

The `\drawLevel` command is the most relevant here, as it draws a thick line with arrows representing the spin of the electrons at that level. It understands the following options:

- `elec`: This defines the number of electrons with their direction. The value can be `up`, `down`, `updown`, or `pair`, with the last two both having the same effect of displaying two electrons in the up and down directions, as seen in *Figure 11.24*.
- `pos`: This is the position of the left side of the energy level as (*x*,*y*) coordinate, enclosed in curly braces to ensure the correct parsing. If you omit it, (0,0) will be used.
- `width`: This is the width of the energy level, which is `1` by default.

Some style options allow customizing color, thickness, arrows, and line style, as listed in the manual. You can open the manual by running `texdoc tikzorbital` at the command prompt or visiting `https://texdoc.org/pkg/tikzorbital`.

The `\drawLevel` command generates anchors to the `left`, `right`, and `middle` of it that we can use for drawing.

Apart from the `\drawLevel` command, we used TikZ commands for drawing lines and nodes; you can read more about the TikZ commands in my book, *LaTeX Graphics with TikZ*, or start at `https://tikz.org`.

A good starting point to learn more about MO diagrams is `https://en.wikipedia.org/wiki/Molecular_orbital_diagram`. I wrote this example in LaTeX to represent one of the figures at `https://www.ch.ic.ac.uk/vchemlib/course/mo_theory/main.html`, where you can also find more MO diagrams.

There's more...

The `tikzorbital` package provides the `\orbital` command to visualize atomic orbitals. Here's a quick example:

```
\begin{tikzpicture}
  \orbital{dyz}
  \orbital[pos = {(2.4,0)}]{dx2y2}
  \orbital[pos = {(4.5,0)}]{dz2}
\end{tikzpicture}
```

This gives us the following picture:

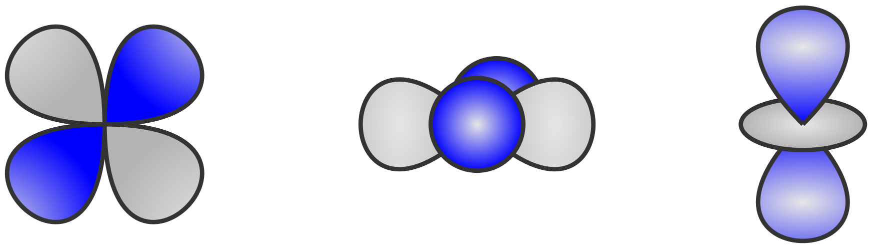


Figure 11.25 – Atomic orbitals

Furthermore, the package provides an `\atom` command that can even be used to build molecule drawings. Here's an example that displays the diatomic molecule hydrogen fluoride (HF):

```
\begin{tikzpicture}
  \atom[name=F, color=red]{
    blue/270/south/2, blue/180/west/2,
    blue/90/north/2,  blue/0/east/1}
  \atom[name=H, color=gray, pos={(1.5,0)},
      scale=0.7]{gray/180/west/1}
\end{tikzpicture}
```

The output of this code is the following:

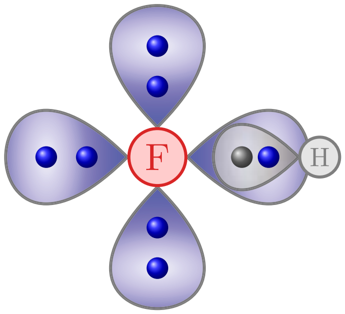


Figure 11.26 – The hydrogen fluoride molecule

This shall quickly demonstrate what we can achieve using a few commands of the `tikzorbital` package. If you are interested, you can take a deep dive into the package manual, which also provides other examples.

Printing a customized periodic table of elements

In the previous recipes, we read a lot about atoms and elements. Do you remember that huge poster of the periodic table of elements in the chemistry room at your school? Let's make it ourselves!

How to do it...

We will use the `pgf-PeriodicTable` package written by Hugo Gomes. Take the following steps:

1. Start with any `document` class:

   ```
   \documentclass[border=10pt]{standalone}
   ```

2. Load the `pgf-PeriodicTable` package:

   ```
   \usepackage{pgf-PeriodicTable}
   ```

3. Begin the document:

   ```
   \begin{document}
   ```

4. Use the `\pgfPT` command:

   ```
   \pgfPT
   ```

5. End the document:

   ```
   \end{document}
   ```

6. Compile and look at the result:

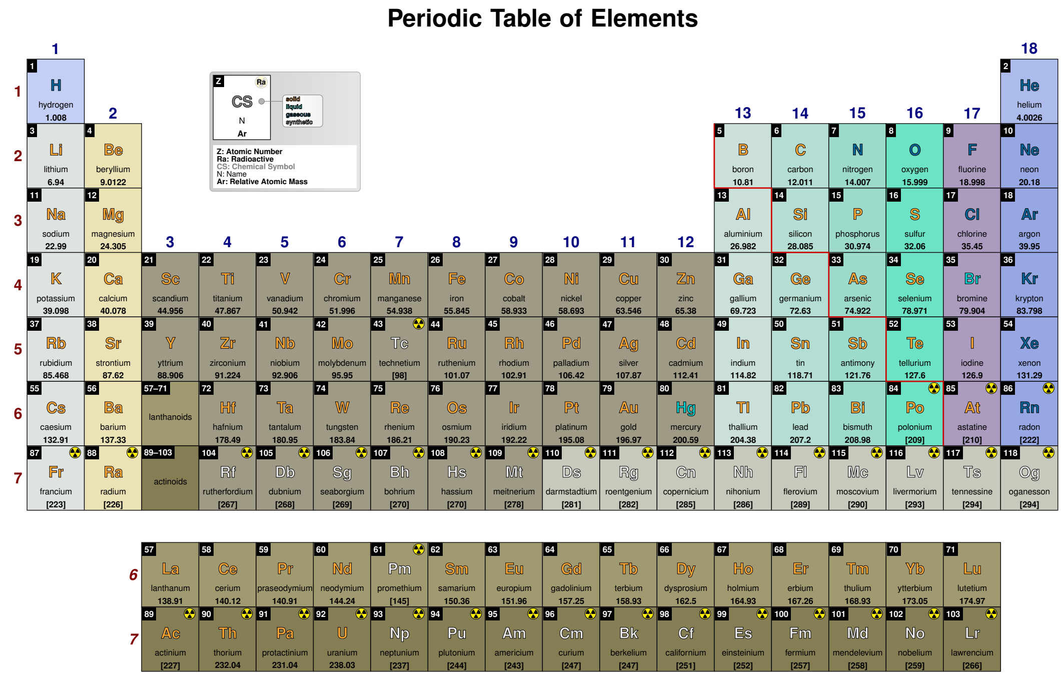


Figure 11.27 – The periodic table of elements

How it works...

That was too easy. The `\pgfPT` command understands many options, and you can customize a lot, including the colors. Let's leave a detailed reference to the package manual that you can find at `https://texdoc.org/pkg/pgf-PeriodicTable`.

Let's use that command to print, for example, the **IUPAC** groups 1 and 2 (also known as the **lithium group** and the **beryllium group**), periods 2 and 3, as follows:

```
\documentclass[border=10pt]{standalone}
\usepackage{pgf-PeriodicTable}
\usepgfPTlibrary{colorschemes}
\pgfPTGroupColors{example}{G1=red!90!black, G2=orange}
\begin{document}
\pgfPT[show title = false, back color scheme = example,
  legend box = {draw=blue!50, fill=blue!20},
  show extra legend,
  Z list = {1,3,4,11,12}]
```

```
\end{document}
```

This gives us the following output:

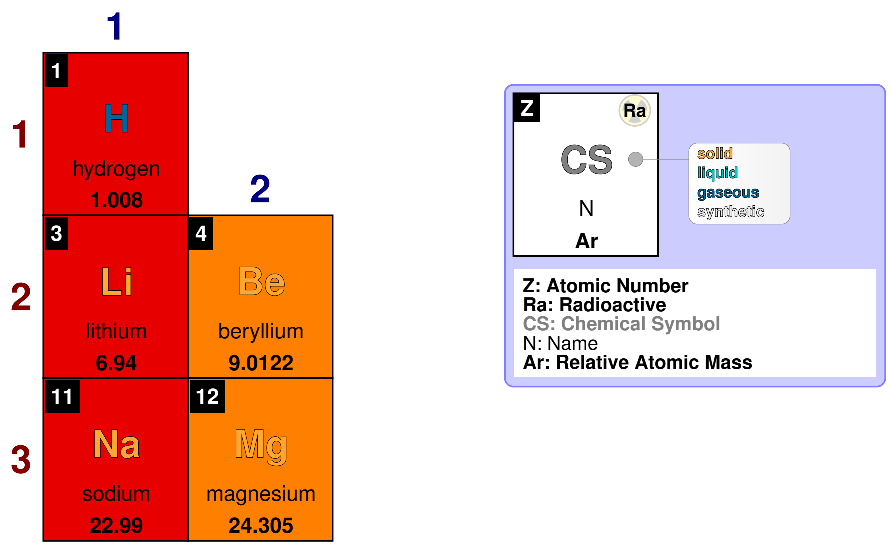


Figure 11.28 – A customized part of the periodic table of elements

Drawing electrical circuits

Technical documents in the domain of electrical engineering often comprise numerous formulas and many drawings. LaTeX excels in math typesetting, making it a top choice for authors. As we explored in a prior section of this chapter, the `siunitx` package makes representing electrical units in compliance with standards easy.

Drawing electrical circuits directly in LaTeX has various benefits. Unlike importing external images, drawings done within LaTeX can have annotations that precisely match the text regarding fonts and styles for perfect consistency.

Therefore, this section focuses on generating circuit diagrams. We aim to design a circuit featuring typical electrical components such as resistors, diodes, capacitors, bulbs, and more.

Note

The drawing in this recipe serves as a sample, and attempting to replicate it with actual components at home is not advised.

How to do it...

The TikZ graphics package provides several libraries for drawing electrical and logical circuits. We'll select one that adheres to the IEC standard. The code is a bit long, so it's recommended to download it with the code bundle from the publisher's website or from `https://latex-cookbook.net/chapter-11`. Here's a step-by-step guide:

1. Start with a `document` class. For this illustration, choose the `standalone` class, which generates a PDF file matching the size of our drawing. Then, load the `tikz` package:

```
\documentclass[border=10pt]{standalone}
\usepackage{tikz}
```

2. Load the `circuits.ee.IEC` TikZ library, symbols complying with the IEC norm:

```
\usetikzlibrary{circuits.ee.IEC}
```

3. Begin the document:

```
\begin{document}
```

4. Open a `tikzpicture` environment and define the following options:

   - The desired style
   - The `x` and `y` unit dimensions
   - An annotation style for a smaller font size
   - Graphic symbol settings, if desired
   - A switch contact style

   Here's the command with selected sample values:

```
\begin{tikzpicture}[
    circuit ee IEC,
    x = 3cm, y = 2cm,
    every info/.style = {font = \scriptsize},
    set diode graphic = var diode IEC graphic,
    set make contact graphic =
      var make contact IEC graphic,
  ]
```

5. Start by drawing six contact points in two rows, three per row. Utilize a `\foreach` loop for convenience:

```
  \foreach \i in {1,...,3} {
    \node [contact] (lower contact \i) at (\i,0) {};
```

```
      \node [contact] (upper contact \i) at (\i,1) {};
    }
```

6. As we defined the contacts' names, given in parentheses, we can refer to them using `upper contact 1`, `lower contact 3`, and similar. So, we will connect the upper-left contact and the lower-left contact by a line with a diode in the middle:

```
    \draw (upper contact 1) to [diode]
      (lower contact 1);
```

7. We saw that we stated the component name as an option for the path. We can do the same for a capacitor:

```
    \draw (lower contact 2) to [capacitor]
      (upper contact 2);
```

8. The component keys can have options. So, we draw a line with a resistor, which has an electrical resistance of 6 ohm, with that value as annotation:

```
    \draw (upper contact 1) to [resistor = {ohm = 6}]
          (upper contact 2);
```

9. Annotations can be different. Here, we use a symbol for an adjustable resistor:

```
    \draw (lower contact 2) to [resistor = {adjustable}]
          (lower contact 3);
```

10. We can have even more options. Useful options are near start and near end for positioning two components at a line:

```
    \draw (lower contact 1) to [
        voltage source = {near start,
        direction info = {volt = 12}},
              inductor = {near end}]
      (lower contact 2);
```

11. Do it similarly for an open contact and a battery with some text as annotation:

```
    \draw (upper contact 2) to
        [ make contact = {near start},
               battery = {near end,
                  info = {loaded}}]
      (upper contact 3);
```

12. Let's finish with a bulb. We will make it a bit bigger than the default:

```
    \draw (lower contact 3) to
        [bulb = {minimum height = 0.6cm}]
      (upper contact 3);
```

13. End the `tikzpicture` environment and the document:

```
  \end{tikzpicture}
  \end{document}
```

14. Compile and have a look at the circuit:

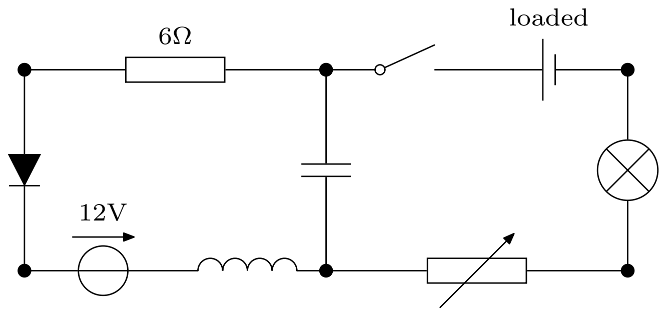


Figure 11.29 – A fictional electronic circuit

How it works...

The TikZ manual references the circuit libraries, showcasing symbols and their associated options. We can only outline some selected details of the breadth of content here. But here's a stepwise summary of our approach:

1. Load the necessary library and defining styles, either as an option to the `tikzpicture` environment or globally through the `\tikzset` command.
2. Position contacts and other nodes, which can be done using pure coordinates combined with a `\foreach` loop or with the aid of the `positioning` TikZ library. Another option would be utilizing a TikZ `matrix of nodes`.
3. Draw lines between the nodes using `to` paths, which take components as options.

Components can have further options, such as for additional information (`info above`), positioning (`near start` or `near end`), or color and size.

In our example, we opted for verbose naming and ample spacing to enhance code readability, a practice particularly beneficial in complex drawings.

Access the TikZ manual by entering `texdoc tikz` via the command line or read it online by visiting `https://texdoc.org/pkg/tikz`. The relevant sections are in a *Circuit Libraries* section in the *Libraries* part. You can also go directly to `https://tikz.dev/library-circuits` to read that section online.

See also

While scientific and technical writing is often based on mathematical writing, there are distinct field-specific notations, conventions, and requisites.

Developers and power users in the LaTeX community created numerous LaTeX packages and classes dedicated to a certain scientific field. The CTAN catalog is a good place to explore what's out there. Visit the CTAN topic categories; here are a few examples:

- `https://ctan.org/topic/physics`
- `https://ctan.org/topic/biology`
- `https://ctan.org/topic/chemistry`
- `https://ctan.org/topic/astronomy`
- `https://ctan.org/topic/electronic`

You can also visit `https://ctan.org/topics/cloud` to find the field of science you are looking for.

LaTeX's capability to generate scientific illustrations is fascinating. There's an abundance of examples available, and I curate a TikZ gallery featuring diverse drawings alongside different scientific disciplines. The gallery is conveniently organized by scientific field, allowing you to explore various graphics created using LaTeX. For example:

- `https://texample.net/tikz/examples/area/physics/` showcases approximately 50 examples, encompassing 3D atom clusters, energy level diagrams, optics, mechanics, astronomy, and more
- `https://texample.net/tikz/examples/area/chemistry/` presents 15 illustrations for chemistry, including a periodic table of elements
- `https://texample.net/tikz/examples/area/computer-science/` contains around 40 drawings covering networks, database topics, protocols, algorithms, and related topics

Even more drawings with their code can be found at `https://tikz.net`. Over 500 TikZ examples are available to explore by topic, spanning physics, engineering, computing, mathematics, and beyond.

These curated resources offer a rich repository of LaTeX-generated graphics across various scientific domains. Take their source code as a basis for your own drawings.