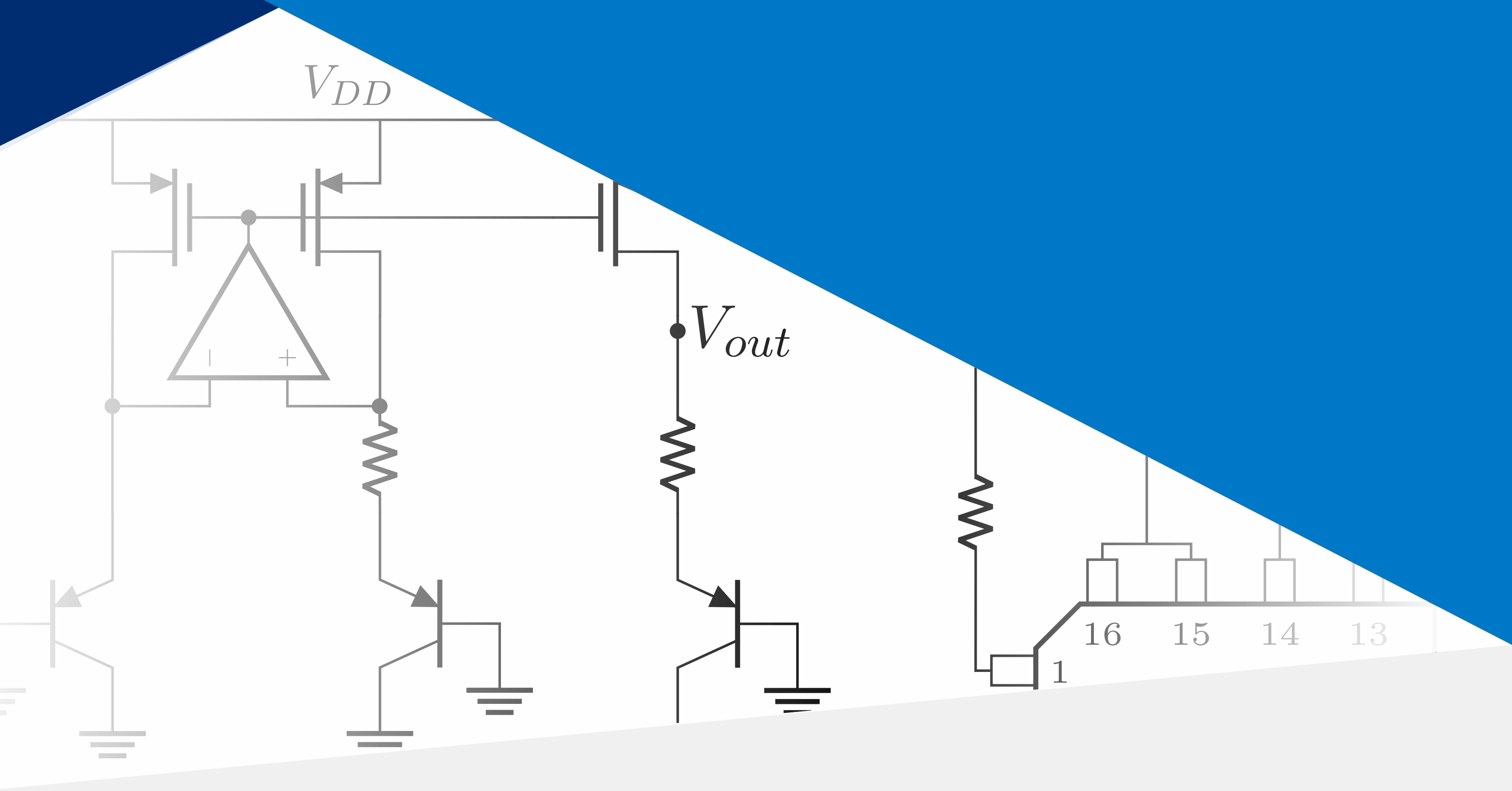


A Practical Guide to TikZ and CircuitikZ

Graphs, circuits, and timing diagrams in LaTeX, with additionals on PGFs and Python plotting

Giovanni Altieri

Politecnico di Milano

Preface

This personal manual was born as a byproduct of a thesis project, during which TikZ, `circuitikz`, and Python became essential tools for creating circuit schematics, plots, and timing diagrams.

Over time, the document evolved into a curated collection of concepts and reusable code snippets, along with LaTeX settings and configurations useful for reproducibility.

This document can be used as a quick introduction to these tools.and aims to:

- demonstrate TikZ and `circuitikz` from the practical perspective of electronics engineers,
- provide reusable examples,
- students save time and avoid unnecessary effort.

It is not intended to replace the official documentation. Rather, it offers an introductory collection of principles and practical tips for using TikZ and `circuitikz`, along with LaTeXuseful code snippets, and custom configurations.

For a deeper exploration of these subjects, readers are encouraged to consult the official references:

PGF/TikZ documentation

CircuitikZ documentation

Python Matplotlib documentation

Contents

Chapter 1

Getting Started

1.1 Why Use TikZ and CircuitikZ?

In electrical and electronic engineering, schematic diagrams are fundamental for analysis, documentation, and communication. Such diagrams must be precise, clear, and unambiguous. Although graphical editors are useful for quick sketches, they often lack consistency, reproducibility, and integration with technical documents.

1.1.1 Why TikZ?

TikZ is a powerful graphics package for LaTeX that enables figures to be created **programmatically**. This ensures a high level of precision, since **coordinates, shapes, and connections are explicitly specified**. It also enables consistency in projects. In fact, styles and formatting rules can be defined once and reused throughout the document. Because the diagrams are code-based, they are easily reproducible and can be version-controlled alongside the text.

TikZ is a general-purpose tool capable of producing a wide variety of technical illustrations, including block diagrams, control system representations, and mathematical visualizations.

1.1.2 Why CircuitikZ?

CircuitikZ is a specialized extension of TikZ tailored to electrical and electronic schematics. It provides predefined symbols for standard components such as resistors, capacitors, inductors, voltage and current sources, transistors,logic gates, and operational amplifiers. Such components are customizable and available in american, european and IEEE standard format.

Since CircuitikZ is built on top of TikZ, it remains fully compatible with its drawing capabilities, allowing circuit elements to be combined seamlessly with custom graphics, annotations, and additional technical elements. Together, TikZ and CircuitikZ offer a structured, precise, and reproducible approach to schematic design within LaTeX.

Integration with LaTeX

Because schematics are written as text, they fit naturally inside a LaTeX document. The style automatically matches the rest of your document (same fonts, spacing, and overall look) . You can edit or regenerate diagrams anytime without losing quality, which makes updating notes, reports, or papers much easier. Being native to LATEX, TikZ and CircuitikZ integrate naturally with equations, figures, references, and captions. Circuit parameters can be expressed using mathematical notation, shared macros, or document-wide variables.

Or just export it as screenshot

If you prefer other text editors, with TikZ and CircuitikZ you can simply generate the picture in blank document (graph, plot or schematic) and export it as a screenshot. This is quick and convenient, especially for slides or informal documents.

Integration of External Data in Plots

With TeX, you can easily data data coming from measurements or simulations using .CSV files, or even load .PGF files generated by other software. There are plenty of ways to customize it, giving you a lot of flexibility to control the final appearance. Later on, there is a dedicated chapter explaining how to export data processed in Python into a .PGF file, so it can be directly used inside your LaTeX documents.

1.2 Required Packages

This section lists all the packages I use to compile. They are grouped by purpose for clarity.

Images and Graphics

`graphicx` Include images.

`eso-pic` Place background images on title pages.

`subcaption` Subfigures support.

`xcolor` Custom colors.

Mathematics

`amsmath, amssymb, amsfonts, amsthm` Standard math environments and symbols.

`bm` Bold math symbols.

`empheq` Braced-style systems of equations.

TikZ and Circuitikz

`tikz` For high-quality figures.

`tikz libraries` arrows.meta, positioning, decorations, shapes, calc, fit, intersections, through.

`circuitikz` Drawing circuits.

`fadings` Fading effects in TikZ.

Other Useful Packages

`pgfplots` Plots and graphs, with `groupplots` library.

`lipsum` Dummy text.

`pdfpages` Include PDFs.

`afterpage` Execute commands after a page break.

`fancyhdr` Custom headers and footers.

`mathabx` Additional math symbols.

1.3 What TikZ and `circuitikz` can do together

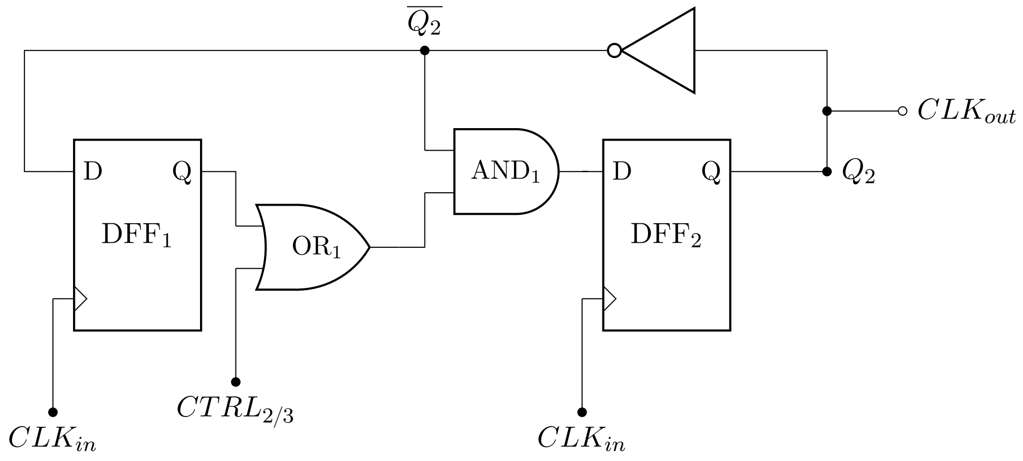


Figure 1.1: A ÷2/÷3 frequency divider circuit using logic gates.

The circuit in Figure 1.1 represents a common building block in electronics: the *frequency divider*.

By the end of this manual, you will learn:

- How to represent basic components such as resistors, transistors, logic gates, amplifiers in `circuitikz`.
- How to correctly label nodes and voltages for clarity.
- How to structure a figure environment for professional-quality illustrations.

- The workflow for including external TikZ diagrams in your document.

Its timing diagram is reported in Figure 1.2.

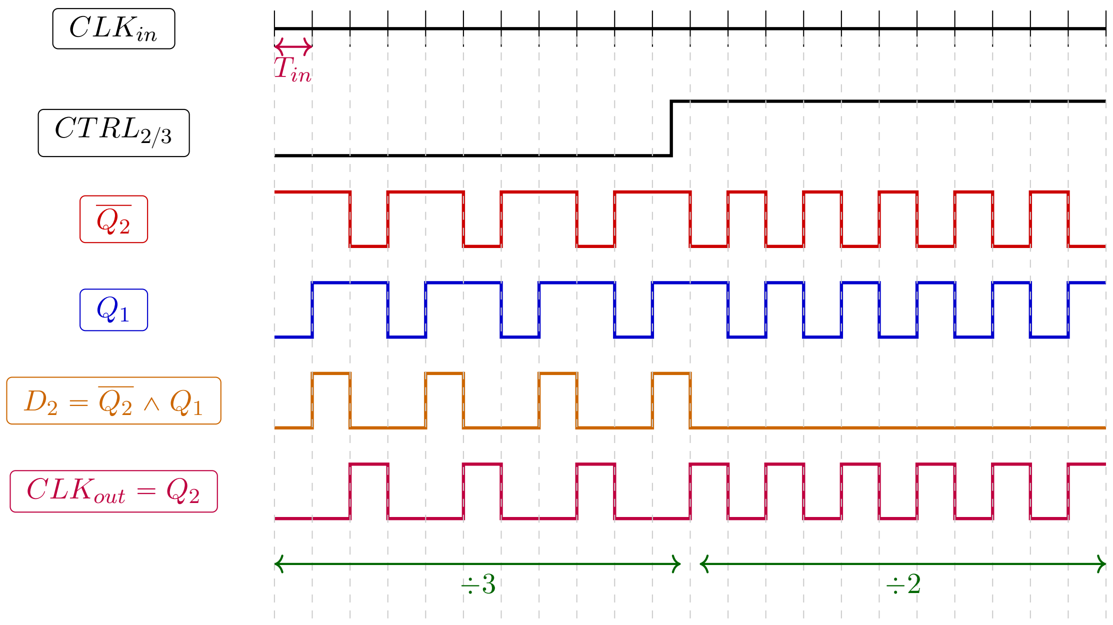


Figure 1.2: Example timing diagram illustrating signal transitions and frequency divisions.

By mastering TIKZ and `circuitikz` for plots, schematics, timing diagrams, and so on, you can gain a toolset that can describe even complex temporal behaviors easy to illustrate and communicate, a skill essential for reports, theses, or publication-quality schematics.

Equivalently, an analog circuit can be illustrated. For instance, in Figure 1.3, a bandgap circuit is represented.

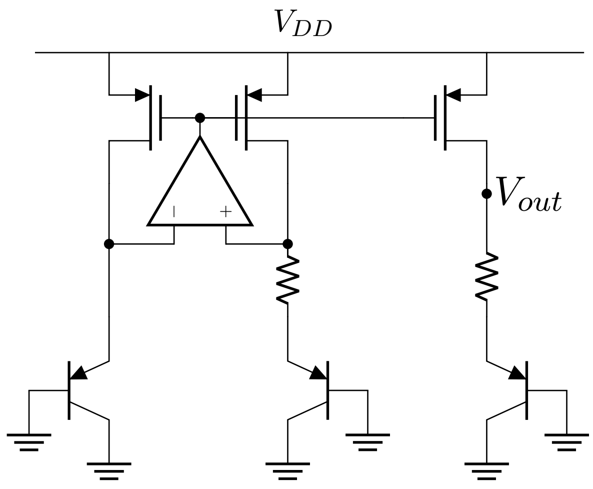


Figure 1.3: Bandgap circuit

These examples highlight several key features of TIKZ and `circuitikz`. They demonstrate how multiple components can be arranged in a clean and readable layout, how nodes, voltages, and signals can be labeled with precision, and how analog and digital elements can be combined within the same schematic. They also show the scalability of the approach, which applies equally well to simple circuits such as voltage dividers and to more complex, multi-stage designs. Finally, they illustrate the flexibility offered by style, color, and line customization to improve both clarity and visual quality.

With these foundations in place, the manual now turns to practical examples and reusable patterns that can be directly adapted to real-world projects.

Chapter 2

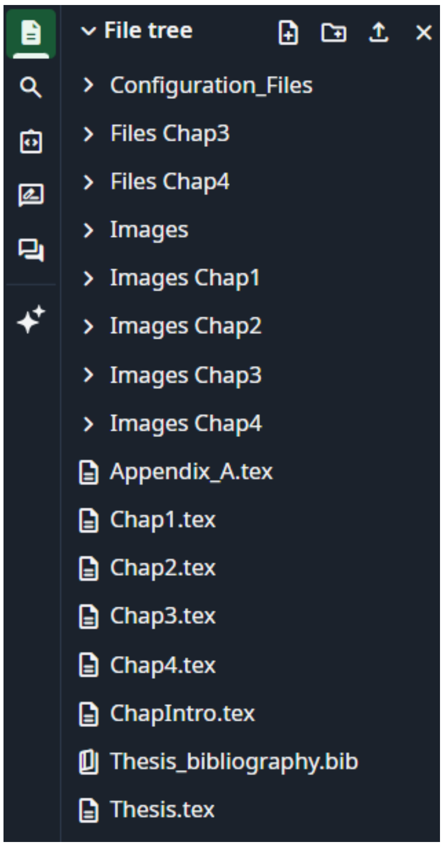
Organizing your file

2.1 Create a modular directory

To keep the project modular, readable, and easy to maintain, organize the files using a clear chapter-based hierarchy. At the root of the project, place the main document and all global resources:

- `main.tex`: main entry point of the document
- `bibliography.bib`: global bibliography file

These files should remain clean and only include or reference other components.



```
Configuration_Files
Files Chap3
Files Chap4
Images
Images Chap1
Images Chap2
Images Chap3
Images Chap4
Appendix_A.tex
Chap1.tex
Chap2.tex
Chap3.tex
Chap4.tex
ChapIntro.tex
Thesis_bibliography.bib
Thesis.tex
```

I suggest each chapter to be stored in its own `.tex` file at the top level. This allows chapters to be edited, compiled, and reused independently. For each chapter, symmetrically, create a dedicated folder to store auxiliary files, images and files.

Use these folders for:

- Images
- TikZ/CircuitikZ diagrams
- Tables
- Code listings
- Custom inputs included with `\input{}` or `\include{}`

Finally, store all global settings and reusable definitions in `Configuration_Files/`. This folder may contain:

- Style files
- Custom commands
- CircuitikZ or TikZ styles
- Package configurations

Use the function `\input` in your `main.tex` to incorporate code in other files.

```
% ------------  Introduction --------------
\input{ChapIntro}
% -------------- CHAPTERS -----------------
\input{Chap1}
\input{Chap2}
\input{Chap3}
\input{Chap4}
```

When defining packages and custom settings, it is good practice to place configuration code in a separate file.

```
% Fading Options
\usetikzlibrary{fadings}
\input{fading_config}

% Listing Definitions
\usepackage{listings}
\input{listing_config}
```

For each TikZ or CircuitikZ picture, save the full code in a separate `.tex` file, in a proper directory.
To insert it in a `figure` Environment, avoid `\includegraphics` and use instead the `\input` command

```
\begin{figure}[ht]
    \centering
    \input{Images Chap1/example_timing}
    \caption{Example timing diagram.}
    \label{fig:example_timing}
\end{figure}
```

Tip: Start a Figure Environment using `\bfi + ENTER`, or copy-paste an image from folder or screenshot directly into plain text.

Chapter 3

TikZ: draws and nodes

In this chapter, I want to share my personal understanding about TikZ . It's not about memorizing commands or following strict rules; it's about developing a sense of how to place components, connect nodes, and make everything look clean and understandable.

3.1 The Environments

The `\begin{tikzpicture} ... \end{tikzpicture}` Environment is the core drawing environment provided by TikZ. Inside it, you create graphics using coordinate-based commands. It is general-purpose and can be used to draw geometric shapes, graphs, state machines, plots, and custom diagrams.

The `\begin{circuitikz} ... \end{circuitikz}` environment comes from the CircuitikZ package and is realized as an extension for electrical and electronic schematics.

3.2 The fundamental objects

In TikZ everything is fundamentally based on **coordinates** (x, y), given by default in cm units. Every line, every component, every connection is placed using a pair of numbers that specify its position on the page.

There are fundamental commands that are of paramount importance in almost everything you'll do. I suggest to focus on this five:

- **`draw`** – This is the command that actually draws lines, shapes, or components.
- **`node`** – Nodes are the "anchors" in TikZ. They are points that can hold text, labels, or even shapes.
- **`path`** – The invisible version of a draw. It's a series of coordinates or points that define a route.
- **`coordinate`** – The invisible version of a node. Used mostly for point referrals storage.
- **`fill`** – Used when filling shapes with colors is needed.

3.3 Draws and nodes

3.3.1 Basic commands

A `draw` is a command used to create visible lines or shapes in your diagram.
Start a drawing using `\draw`, then put the starting and arriving coordinates of the line, connected by "`--`" or `to`:

```
\draw (x1, y1)
    to[OPTIONS] (x2, y2);
```

You can chain multiple points to create connected draws (use `cycle` to close back to the start):

```
\draw (0,0)
    -- (2,0) -- (2,2) -- (0,2) -- cycle;
```

A `node` is used to mark points or attach labels to positions on a draw or path.
The basic syntax is:

```
\draw (x,y)
    node[OPTIONS] (REFERENCE) {LABEL};
```

where the features are put in OPTIONS. If you intend to refer the node in future, give a name in (REFERENCE). The LABEL is optional, **however the {} are mandatory**.

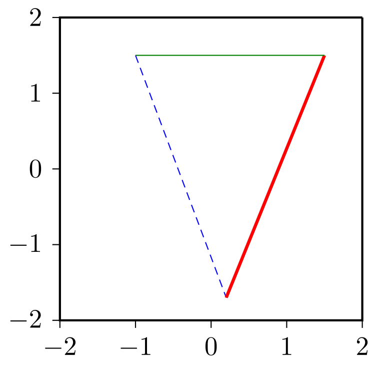
`\node` can be used as stand-alone coomand:

```
\node[OPTIONS] (REFERENCE) at (x, y) {LABEL};
```

which is equivalent to the previous command.

Also `draw` can be customized using [OPTIONS], in color, thickness, and style.

Example 3.3.1. Define a triangle using **customized draws**.



```
\begin{tikzpicture}
    \draw[blue, densely dashed] (-1,1.5) -- (0.2,-1.7);
    \draw[red, very thick] (0.2,-1.7) -- (1.5,1.5);
    \draw[green!50!black] (1.5,1.5) -- (-1,1.5);
\end{tikzpicture}
```

`draw` (-1,1.5)–(0.2,-1.7) in blue, dashed,
`draw` (0.2,-1.7)–(1.5,1.5) in red, thick,
`draw` (1.5,1.5)–(-1,1.5) in half-green half-black;

Tip: `path` and `coordinate` are the invisible equivalent of `draw` and `node`, therefore do not use [OPTIONS] with them. Instead, use them as the backbone of your design.

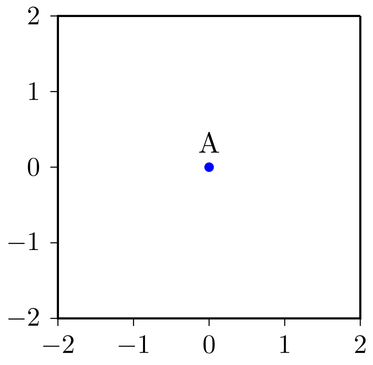
Remember: Always put {} after every node, even if unused. This is odd, **but it is necessary**.

3.3.2 Some drawing examples

Start a drawing using the `\draw` or `\path`, describe a collection of points to shape your route, and terminate with `";"`.

Also `\node` and `\coordinate` can start a command, but they do not expect points in route.

Example 3.3.2. Define a **circle node** at the origin.

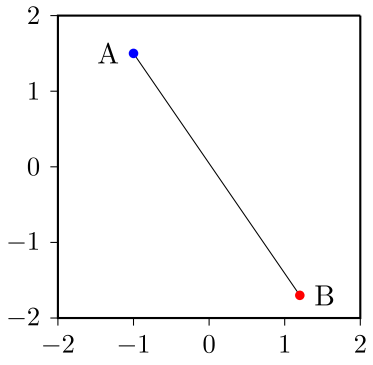


```
\begin{tikzpicture}
    \draw (0,0)
        node[circ, blue, label=above:A] (A) {};
\end{tikzpicture}
```

Start `draw` (0,0),
place `node` (with blue, circle, and label "A" above)

A draw or a path?

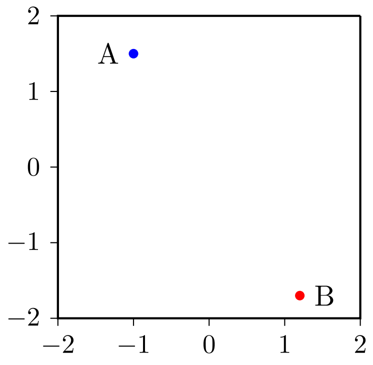
A path is the invisible version of a draw.
Example 3.3.3. Define a **draw** and the two extreme **circle nodes**.



```
\begin{tikzpicture}
    \draw (-1,1.5)
        node[circ, blue, label=above:A] (A){}
        -- (1.2,-1.7)
        node[circ,red, label=right:B] (B){};
\end{tikzpicture}
```

Start `draw` from (-1,1.5),
place `node` (blue, circle, and label "A" left),
`draw` to (1.2,-1.7),
place `node` (red, circle, and label "B" right);

Example 3.3.4. Define a **path** and the two extreme **circle nodes**.



```
\begin{tikzpicture}
    \path (-1,1.5)
        node[circ, blue, label=above:A] (A){}
        -- (1.2,-1.7)
        node[circ,red, label=right:B] (B){};
\end{tikzpicture}
```

Start `path` from (-1,1.5),
place `node` (blue, circle, and label "A" left),
`path` to (1.2,-1.7),
place `node` (red, circle, and label "B" right);

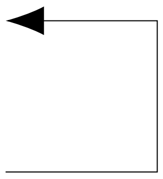
Tip: Indentation has no effect. Use to highlight the command and the starting coordinate, then write instructions line by line.

3.3.3 Creating coordinates along a path

When starting a new `draw` or `path`, TikZ expects **coordinate points in sequence**. Along the sequence, **`nodes` can be created**, or **`coordinates` can be stored**.

Displecement coordinates

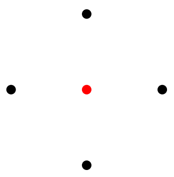
A new coordinate can be given in absolute terms, or **as displacement from the previous position**, using `--++(`$\delta x, \delta y$`)` or `--+(`$\delta x, \delta y$`)`.



```
\begin{tikzpicture}
    \draw[-{Latex[scale=3]}]
        (0,0) --++(2,0) --++(0,2) --++(-2,0);
\end{tikzpicture}
```

Exercise 3.3.1. Consider the next example.

Can you understand the difference between `--++(`$\delta x, \delta y$`)` and `--+(`$\delta x, \delta y$`)` ?



```
\begin{tikzpicture}
    \path (0,0) node[circ,red]{} --+(1,0) node[circ]{}
        --+ (0,1) node[circ]{} --+(-1,0) node[circ]{}
        --+(0,-1) node[circ]{};
\end{tikzpicture}
```

Solution 3.3.1. `-++(`$\delta x, \delta y$`)` draw to a new point **and** updates the reference point.

In contrast, `-+(`$\delta x, \delta y$`)` moves the drawing point without changing the reference point.

Orthogonal interesections

Coordinates can be given as **orthogonal intersection** of points using `|-` or `-|`. Thick of it as square lines that go "up-and-right" or "right-and-up". Refer to these as first and second orthogonal drawings.

`(A) |- (B)` draws the first orthogonal drawing between (A) and (B).
`(A |- B)` constructs a point at the first orthogonal intersection, using A_x and B_y.

Example 3.3.5. Find the first orthogonal intersection of (0,0) and (3,2).

```
\begin{tikzpicture}
    \draw (0,0) |- (3,2)
        (0,0 |- 3,2) node[circ, red]{};
\end{tikzpicture}
```

Exercise 3.3.2. Try and do the analogous using `-|`. What happens?

Solution 3.3.2. The line now start horizontally (-), then finishes vertically (|).

Therefore, the second orthogonal intersection is found (right-and-up).

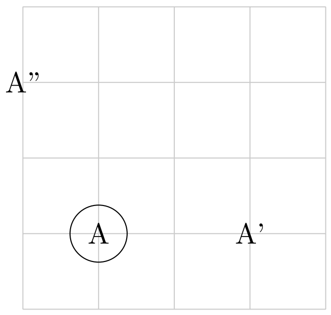
Accessing a modified node

An existing node can be shifted, rotated and scaled around before insertion in the path. It will suffice to add an OPTION before its name. The syntax is therefore `([OPTIONS] A)`.

Point OPTIONS include:

- **Shifts**: `xshift=1cm` — `yshift=5mm` — `shift={(1cm,2mm)}`
- **Scaling**: `scale=2` — `xscale=1.5` — `yscale=0.5` (relative to origin)
- **Rotation**: `rotate=30` (around origin) — `rotate around={45:(A)}` (around A)
- **Slanting / Shearing**: `xslant=0.5` — `yslant=0.3`

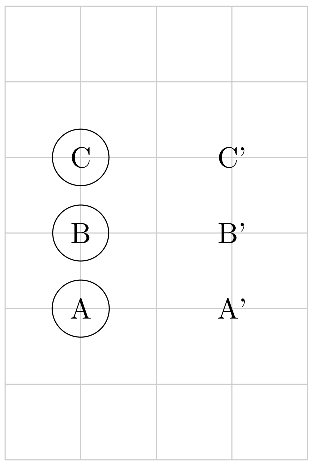
Example 3.3.6. Shifting an existing node.



```
\begin{tikzpicture}
    \draw[opacity=0.2] (-1,-1) grid (3,3);

    \draw (0,0) node[draw, circle] (A){A}
        ( [xshift=2cm] A) node{A’}
        ( [shift={(-1,2)}] A) node{A"};

\end{tikzpicture}
```

When the same operations is done multiple times, they can be **collected into a simpler keyword beforehand**, in the `tikzset` Enviroment, that will be discussed later.

Example 3.3.7. Shifting multiple nodes.



```
\begin{tikzpicture}[
    r2/.style={xshift=2cm}
    c/.style={draw, circle}
]
    \draw[opacity=0.2] (-1,-2) grid (3,4);
    \draw (0,0) node[c](nA){A}
        (0,1) node[c](nB){B}
        (0,2) node[c](nC){C}
        ([r2]nA) node{A’}
        ([r2]nB) node{B’}
        ([r2]nC) node{C’};

\end{tikzpicture}
```

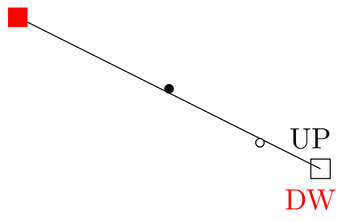
3.3.4 The `foreach` loop

TikZ provides a powerful `foreach` loop to repeat drawing commands with varying coordinates. Syntax is straightforward:

```
\foreach \x in {0,1,2,3,4} {
    \draw (\x,0) node[circ]{};
}
```

3.3.5 Creating nodes along a path

Right after a point, multiple nodes (or coordinates) can be created, as:



```
\begin{tikzpicture}
    \draw (0,0) -- (4,-2) node[draw]{}
        node[midway, above, circ]{}
        node[pos=0.8, below, ocirc]{}
        node[label=above:UP,
            label={[red]below:DW}]{}
        node[pos=0, fill, red]{};
\end{tikzpicture}
```

The ordered OPTIONS for node creation are:

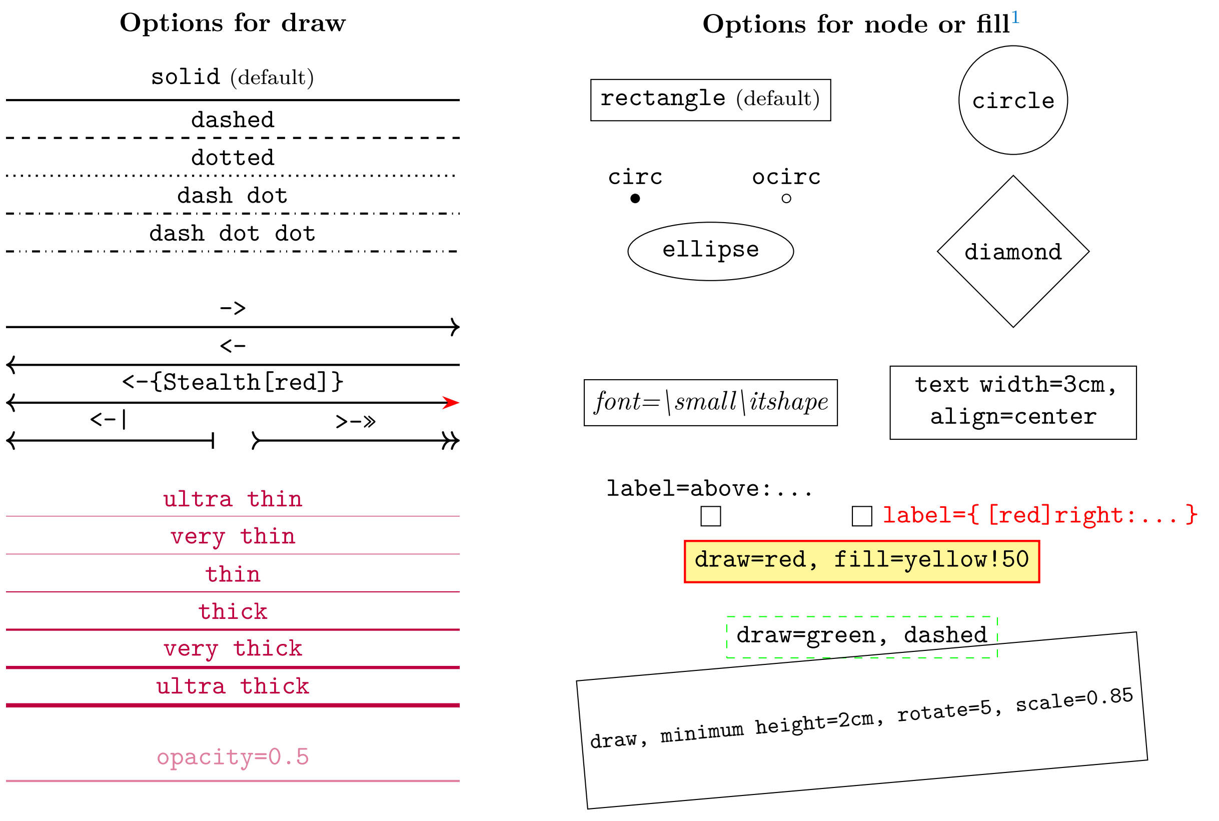
1. **Position along the last segment**:
   `pos=1`(default at the end), `pos=0.5`(midway), `pos=` any decimal.
2. **Additional shift**:
   `above, below, left, ...`, or `[xshift=, yshift=]`
3. **Shape**:
   `circ, ocirc` (adimentional).
4. **`draw=color` or `fill=color`**:
   default color for draw black, use colors with fill
5. **`draw/fill` additionals**:
   `rectangle(default), circle, ellipse...  , color= , rotate= ,`
   `xscale= , yscale= , minimum height= , minimum width= .`
6. Label:
   `label={[OPTIONS]POSITION: TEXT}`.

The priority order in node creation is not binding. However, **I suggest to mentally consider this order** when adding nodes along a draw/path.

3.4 Options for draws, nodes and labels

3.4.1 Options for `draw` and `node`

A `draw` command and `nodes` can accept several [OPTIONS]:



Important: For a shape to be visible in a node, the **`draw/fill` OPTION must be expressed**.

3.4.2 Labels

Labels are particular options for nodes, and can be placed in two ways:

- `node[..., label={[COLOR]POSITION:text}]`
  This is a node **OPTION** and allows advanced control over **multiple labels** in color and position.
  POSITION can be expressed as words {`above, right, left, below`, or mixed}
  or as degree angle {`0, 45, 90`, any angle} around the node center.

- `node[..., text=COLOR, font=STYLE] {text}`
  This is the standard **label** and places **only one label at the center** of the node.

These two methods can also be used simultaneously.
The label OPTION requires curly brackets: `label={[OPTS]:text}`.
They are basically not needed by the shifts operators: `label=above:text`

[1]Obviously, if a line around the label is present, it requires the `draw` OPTION in the node.

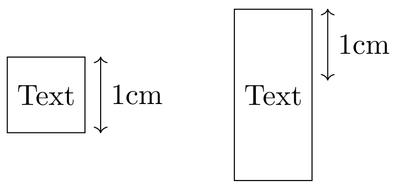
3.4.3 Node sizing

Nodes are specifically programmed to occupy the less space possible. For example, if a node contains only text, unless specified, its shape will adjust precisely to fit that text.

To **modify the sizing of a node** two alternatives are possible: defining its **absolute dimensions**, or specifying **additional padding** to the minimum content.

- `minimum width` / `minimum height`: force minimum absolute sizes.
- `inner sep`: adds uniform padding in x and y.
- `inner xsep`/`inner ysep`: add specific paddings in x and y.

Example 3.4.1. Difference between **`minimum height`** and **`inner yset`**.



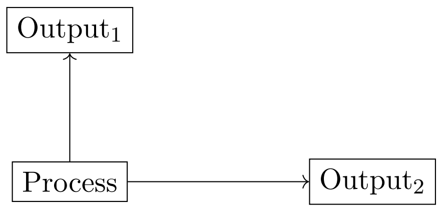
```
\begin{tikzpicture}[
    r/.style={xshift=0.2cm},
    r3/.style={xshift=3cm}
]
    \node[draw, minimum height=1cm](R1){Text};
    \draw[<->]
        ([r]R1.south east) -- ([r]R1.north east)
        node[midway, right]{1cm};

    \node[draw, inner ysep=1cm, r3](R2){Text};
    \draw[<->]
        ([r]R2.20) -- ([r]R2.north east)
        node[midway, right]{1cm};

\end{tikzpicture}
```

3.5 Block diagrams

3.5.1 Basic draw shapes

The make block diagrams, first define nodes at required position using `\node[draw, rectangle]` and connect them using `\draw[->]`.

Example 3.5.1. Define a 3 blocks diagram.

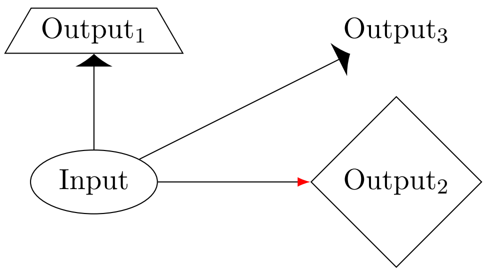


```
\begin{tikzpicture}
    \node[draw, ellipse] (A) {Process};
    \node[draw, ellipse] (B) at (4,0)
        {Output$_2$};
    \node[draw, ellipse] (C) at (0,2)
        {Output$_1$};

    \draw[->] (A) -- (B);
    \draw[->] (A) -- (C);
\end{tikzpicture}
```

Alternatives to `rectangle` are: `ellipse, diamond, circle, trapezium, rounded rectangle`. To customize their dimensions use OPTIONS akin to `[minimum width=3cm, minimum height=1.5cm]`.

Alternatives to `->` are: `-Latex, -Stealth, -Triangle`, etc.
Each of them is customizable using `[ -{Latex[width=4mm,length=6mm, red]} ]`.

Example 3.5.2. Define a 4 blocks diagram, with various draw shapes.



```
\begin{tikzpicture}
    \node[draw, ellipse] (A) {Input};
    \node[draw, diamond] (B) at (4,0)
        {Output$_2$};
    \node[draw, trapezium] (C) at (0,2)
        {Output$_1$};
    \node[trapezium] (D) at (4,2)
        {Output$_3$};

    \draw[-{Latex[red]}] (A) -- (B);
    \draw[-{Latex[width=5mm]}] (A) -- (C);
    \draw[-{Latex[width=5mm]}] (A) -- (D);
\end{tikzpicture}
```

Tip: Define all the nodes beforehand, then express the connections.

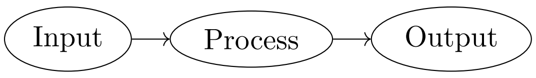
Question: What happens to a node without `draw` ?

If `draw` is not specified, `\node` has no ink to print lines, therefore the result would be an **invisible rectangle**, like *Output*$_3$ in the last example.

3.5.2 Positioning shapes

```
\usetikzlibrary{positioning}
```

An alternative to absolute coordinates is using positioning referred to another node.
The syntax is `\node[right=DISTANCE of A, ...] (B) {...};`
Example 3.5.3. Define a 3 blocks diagram using positioning.



```
\begin{tikzpicture}[node distance=1.5cm]
    \node[draw, ellipse] (A) {Input};
    \node[draw, ellipse, right=0.5cm of A]
        (B) {Process};
    \node[draw, ellipse, right=0.5cm of B]
        (C) {Output};

    \draw[->] (A) -- (B);
    \draw[->] (B) -- (C);
\end{tikzpicture}
```

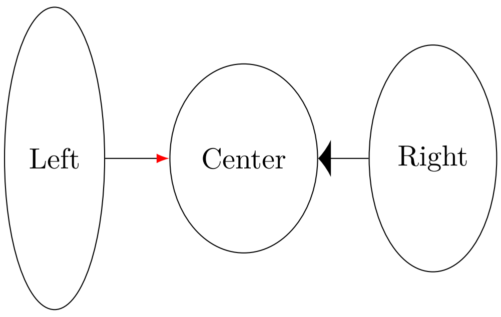
The distance can be defined at instance level, using `tikzpicture` OPTIONS,
as `[node distance=1.5cm]`. Later in the node, `DISTANCE` can be left black, or else overridden.

3.5.3 Calculating positions

```
\usetikzlibrary{calc}
```

The `calc` library allows you to perform coordinate calculations inside TikZ, which is extremely useful for positioning nodes, drawing shapes, or aligning things.
The syntax allowes is `($ CALCULATE $)`, in which all operand nodes must present the (..) coordinate brackets.

Example 3.5.4. Define a 3 blocks customized block, with predefined distance.



```
\begin{tikzpicture}
    \node[draw, ellipse, minimum height=4cm]
    (A) {Left};
    \node[draw, ellipse, minimum height=3cm]
    (B) at ($(A)+(5,0)$){Right};
    \node[draw,ellipse, minimum height=2.5cm]
     (C) at ($(A)!0.5!(B)$) {Center};

    \draw[-{Latex[red]}] (A) -- (C);
    \draw[-{Latex[width=5mm]}] (B) -- (C);
\end{tikzpicture}
```

- `($(A)+(5,0)$)` finds a new coordinate displacing (A).
- `$(A)!0.5!(B)$` find the middle point between (A) and (B).
- Operation combinations are also allows

3.5.4 Anchors to the node

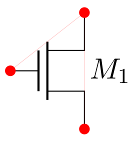
In TikZ, **anchors** are specific points on a node that you can refer to when positioning other elements, drawing lines, or placing decorations. By default, a node anchor can be accessed using **cardinal points** {`.center, .north, .south, .east, .west`}, and their combinations, or using **any degree angle around the node** {`.50, .180, .270`}.

In Circuitikz, component are defined along their custom anchors, like in transistors.

Anchors can be used as coordinates, and their syntax is:

```
    (NODE_NAME.anchor)
```

Example 3.5.5. Highlight source, gate and drain in a NMOS transistor.



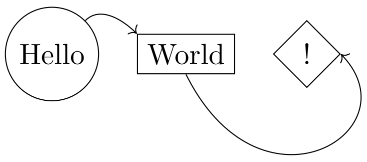
```
\begin{tikzpicture}[radius=2pt]
    \draw (0,0) node[nmos](M1){$M_1$};
    \fill[red] (M1.S) circle -- (M1.D) circle
        -- (M1.G) circle;
\end{tikzpicture}
```

3.5.5 Bending arrows

Sometimes, a straight arrow between nodes can overlap other elements or look too rigid. TikZ allows you to curve arrows for better visual clarity. There are two main OPTIONS to do this:

- Using `bend left=ANGLE` or `bend right=ANGLE`
- Using `out=ANGLE`$_1$`, in=ANGLE`$_2$
- Using control points (Bezier curves), as offset points from start and from end: `..controls +(OFF`$_1$`) and +(OFF`$_2$`)..`

Example 3.5.6. Define a 3 blocks diagram using bending arrows.



```
\begin{tikzpicture}
    \node[draw, circle] (A) {Hello};
    \node[draw, rectangle, right=0.5cm of A]
        (B) {World};
    \node[draw, diamond, right=0.5cm of B]
        (C) {!};

    \draw[->] (A.north east)
        to[out=45, in=135] (B.north west);
    \draw[->] (B.south) ..
        controls +(1,-2) and +(1,-1) ..
        (C.east);
\end{tikzpicture}
```

3.5.6 Intersections and Passing-throughs

```
\usetikzlibrary{intersections,through}
```

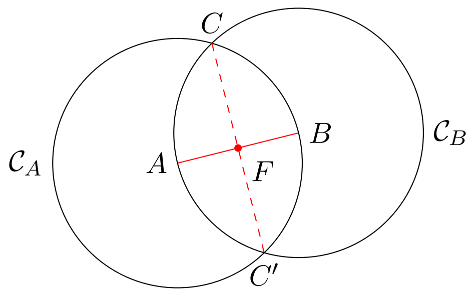
In geometric constructions, **draw/path traces** can be saved under a reference by means of OPTION **"name path="**. In particular, TikZ allows one to compute new points as intersections of previously named paths, such as lines and circles.

Given two paths `path1` and `path2`, their **intersection points** can be automatically computed using the "**name intersections=**" OPTION and stored under user-defined names for later reuse.

```
name intersections={
    of=path1 and path2,
    by={inter1, inter2}
}
```

Circles may be defined implicitly by a point (T) through which the circle passes, using the OPTION `circle through=(T)`, instead of OPTION `circle`.

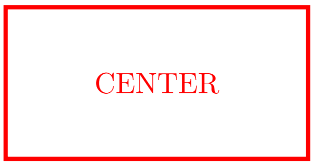
Example 3.5.7. Construct two circles passing through given points and compute their intersection using named paths.



```
\begin{tikzpicture}

\coordinate[label=left:$A$] (A) at (0,0);
\coordinate[label=right:$B$] (B) at (2,0.5);
\draw[name path=AB, red] (A) -- (B);

\node[name path=circleA,
      draw, circle through=(B),
      label=left:$\mathcal{C}_A$] at (A) {};
\node[name path=circleB,
      draw, circle through=(A),
      label=right:$\mathcal{C}_B$] at (B) {};

\path[name intersections={
    of=circleA and circleB,
    by={[label=above:$C$]C, [label=below:$C'$]C'}
    }];

\draw[name path=CCp, dashed, red] (C) -- (C');

\path[name intersections={of=AB and CCp, by=F}];
\node[fill, red, circle, inner sep=1pt,
    label=below right:$F$] at (F) {};

\end{tikzpicture}
```

3.5.7 Rectangle operation

Since rectangles are extremely common, TikZ provides a special rectangle syntax:
`\draw (x1,y1) rectangle (x2,y2);`
This syntax defines a rectangle using two extreme points.



```
\begin{tikzpicture}

    \draw[red, ultra thick] (0,0) rectangle (2,1)
        node[midway]{CENTER};

\end{tikzpicture}
```

Use rectangles to sketch on your images.
Just input an image into a node label using `\includegraphics[]`.

```
\begin{tikzpicture}[ultra thick, font=\bfseries]
    \node {
        \includegraphics[width=0.97\linewidth]{Images Chap2/freq_div.png}
    };
    \draw[red] (-4.1,1.65) rectangle (1.6,-1.75)
        node[midway, yshift=2cm]{Counter$_1$};
    \draw[red] (1.7,1.65) rectangle (7.3,-1.75)
        node[midway, yshift=2cm]{Counter$_2$};
    \draw[orange] (-4.2,1.6) rectangle (-6.3,-0.4)
        node[midway, yshift=1.5cm]{Dual Modulus};
    \draw[green!80!black] (-4.2,-1.1) rectangle (-7,-0.35)
        node[midway, yshift=-0.9cm]{Control Logic};

\end{tikzpicture}
```

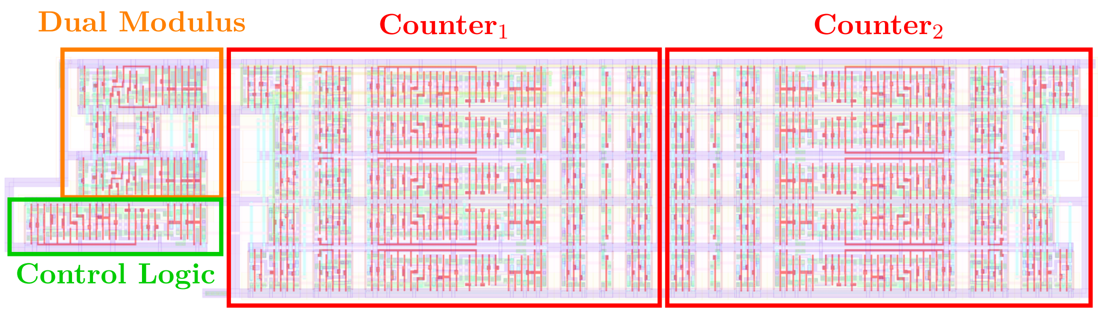


Figure 3.1: Integer-N frequency divider layout

3.5.8 Fit operation

```
\usetikzlibrary{fit}
```

To highlight nodes in a drawing or schematic, `fit` OPTION can be used in a node. It creates a shape that covers all the required nodes automatically.

In particular, the syntax is `fit= (node1) (node2) ... (nodeN)`

```
\begin{circuitikz}
    \input{ImagesChap3/ripple_counter}                 % Load the source picture
    \node[                                             % Add fitting shape
        fit=(FF0) (FF1) (FF2) (FF3) (FF4),
        draw=purple, ultra thick, fill=purple!50, fill opacity=0.1,
        rounded corners, inner xsep=20pt, inner ysep=10pt,
        label={[rotate=90, anchor=south, purple, font=\bfseries]left:Flip-flops}
    ]{} ;
\end{circuitikz}
```

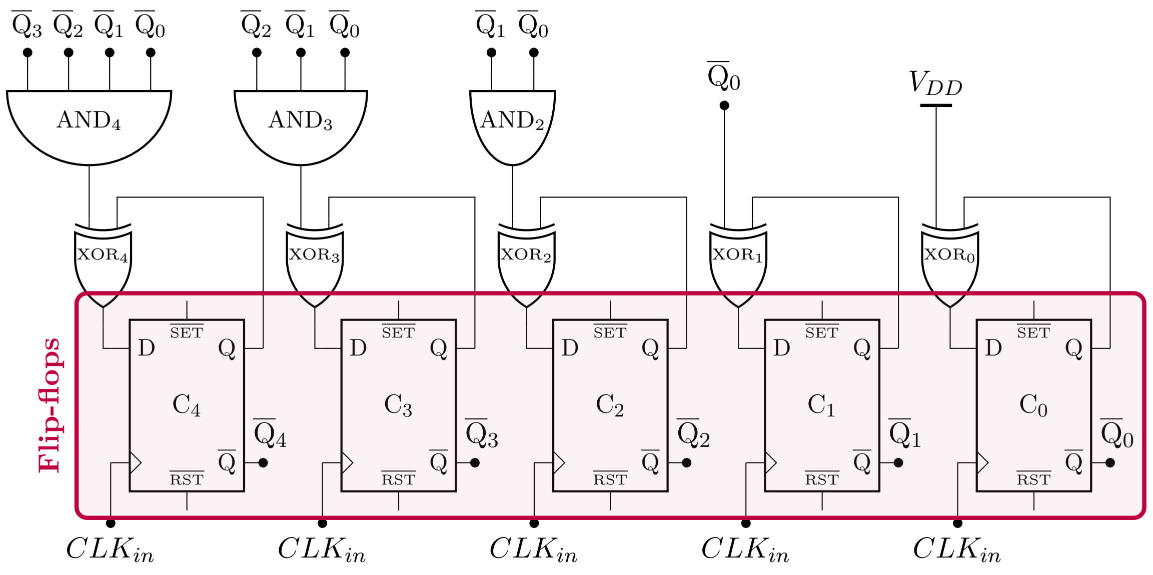
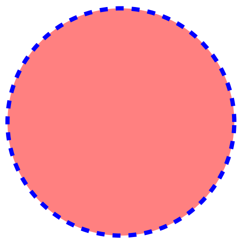


Figure 3.2: Ripple down synchronous counter architecture.

3.5.9 Circle operation

In TikZ, the circle operation allows you to draw a circle centered at a specified coordinate with a given radius (in cm units). Syntax is `circle (RADIUS)`.



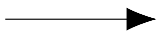
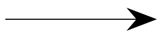
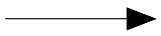
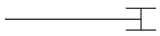
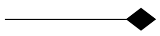
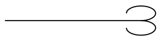
```
\begin{tikzpicture}

    \draw[blue, ultra thick, dashed, fill=red!50]
        (0,0) circle (1.5);

\end{tikzpicture}
```

3.6 Block and lines customizations

3.6.1 Arrow heads

`\usetikzlibrary {arrows.meta}`

Arrows can be defined in customized style for their heads. Standard styles are:

-  `>=Latex`
-  `>=Stealth`
-  `>=Triangle`
-  `>=Bar`
-  `>=Diamond`
-  `>=Hooks[arc=270]`

To **customize a arrowtip**, curly brackets are mandatory: `{Latex[OPTIONS]}`

3.6.2 `\tikzset` OPTIONS setting

TikZ and CircuitikZ allow you to define global OPTIONS using `\tikzset` and `\ctikzset`. These OPTIONS can be reused across multiple diagrams, making your circuits consistent and easier to maintain.

```
\tikzset{     % or \ctikzset{
    >={Stealth[length=3mm, width=2mm]},

    block/.style={
        draw, rectangle,
        ultra thick,
        minimum width=1.2cm,
        minimum height=1cm,
        align=center
    },

    line/.style={
        ->, thick
    }
}
```

3.6.3 `.style` operator

./style is effective in globally define a new symbol, which is a set of existing options for draws or nodes.
In this example, a custom rectangular node is defined under the name-style **`block`** , while a thick arrow is put under the name-style **`line`**.

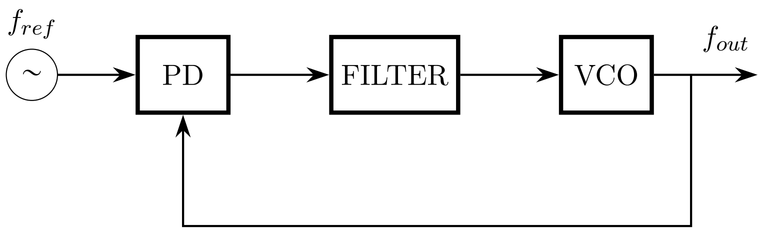
Equivalently, the `\tikzset` and `\ctikzset` Environments can be accessed using instance OPTIONS:

```
% Same of \tikzset{OPTIONS}
\begin{tikzpicture}[
    OPTIONS
]
...
\end{tikzpicture}
```

```
% Same as \ctikzset{OPTIONS}
\begin{circuitikz}[
    OPTIONS
]
...
\end{circuitikz}
```

Example 3.6.1. Draw the block diagram for a Type-I Phase-Locked Loop.

```
\begin{circuitikz}
    \node[circle, draw, label=above:$f_{ref}$] (source) at (-2.5,0) {$\sim$};
    \node[block] (pfd) at (0,0) {PD};
    \node[block] (filt) at (3.5,0) {FILTER};
    \node[block,label=15:$f_{out}$] (vco) at (7,0) {VCO};

    \draw[line] (source.east) -- (pfd.west);
    \draw[line] (pfd.east) -- (filt.west);
    \draw[line] (filt.east) -- (vco.west);
    \draw[line] (vco.east) -- (9.5,0);
    \draw[line] (8.4,0) -- (8.4,-2) -| (pfd.south);
\end{circuitikz}
```



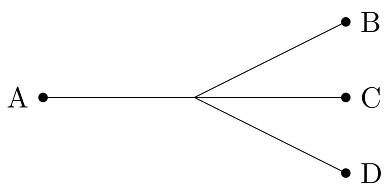
3.7 Advanced branching

It is common in circuit schematics to encounter branching nodes. Using the standard approach, one typically needs to save the current position into a coordinate, draw the first branch, return to the saved coordinate, and then draw the second branch. This process is acceptable for long branches, but for small path it may be simplified.

A practical workaround is to use the `current point is local` OPTION. I suggest defining a shorter keyword for this option, such as `c`, to simplify the syntax and improve readability.

To branch use `{[c] BRANCH_1} +(0,0) BRANCH_2 ;`

Example 3.7.1. A draw started from (A) needs to branch to (B), (C), and (D).



```
\begin{tikzpicture}[
    c/.style = { /tikz/current point is local }
]
    \draw (A)--++(2,0)
        {[c] --(B)} +(0,0)
        {[c] --(C)} +(0,0)
        --(D);

\end{tikzpicture}
```

Note: The `+(0,0)` is needed as a *reset to branch* operation.

3.8 Exercises

Exercise 3.8.1. Create a keyword in TikZ that shifts any node by (1,1).

Exercise 3.8.2. Draw a symmetric figure (e.g., a diamond or star) using only displacement movements.

Exercise 3.8.3. Explore the `automata` TIkZpackage in the official documentation. How does it differ from the conventional drawing style?

Exercise 3.8.4. Based on the concepts in this chapter, propose a method to apply custom transparency to a PNG image.

Exercise 3.8.5. Based on the concepts in this chapter, try to replicate the cover of this manual, neglecting the circuits.

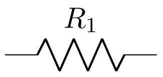
Chapter 4

CircuitikZ: components and scheamatics

4.1 Path-Style and Node-Style components

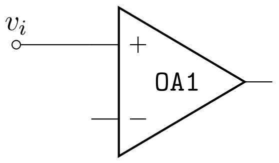
CircuiTikZ commands are just TikZ commands, with additional nodes and draws coming from the electrical world.

Component can be **path-style**, when definable using `to[···]`, as in:



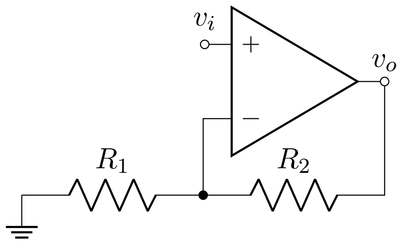
```
\begin{tikzpicture}[american]
    \draw (0,0) to[R=$R_1$] (2,0);
\end{tikzpicture}
```

or can be **node-style**, if definable only using `node[···]`, as in:



```
\begin{circuitikz}[american]
    \draw (0,0)
        node[above]{$v_i$} to[short,o-]++(1,0)
        node[op amp, noinv input up, anchor=+]
        (OA){\texttt{OA1}};
\end{circuitikz}
```

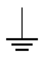
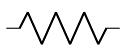
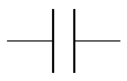
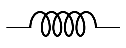
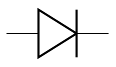
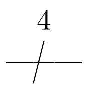
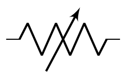
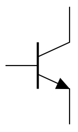
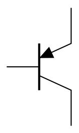
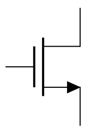
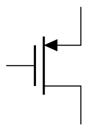
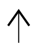
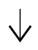
Example 4.1.1. Design a non-inverting configuration.



```
\begin{circuitikz}[american,
    c/.style = { /tikz/current point is local }
]
    \draw node[op amp, noinv input up](OA) {}
        (OA.+) node[ocirc, label=above:$v_i$]{}
        (OA.out) to[short,o-]++(0,-1.5)
        to[R,l_=$R_2$,-*]++(-2.4,0) {[c]|-(OA.-)
    }+(0,0) to[R,l_=$R_1$]++(-2.4,0) node[ground]{}
        (OA.out) node[above]{$v_o$};
\end{circuitikz}
```

4.2 Most used components

For the complete list of components, please refer to the complete documentation on CircuitikZ. Here is provided a quick reference list for the most used components.

`node[ground]`		`node[tlground]`		`node[tground]`	
`to[R,american]`		`to[C]`		`to[cute inductor]`	
`to[Do]`		`to[multiwire=4]`		`to[vR, american]`	
`node[pnp]`		`node[pnp]`		`node[nmos]`	
`node[pmos]`		`node[vcc]`		`node[vee]`	

4.2.1 Suggested transistor customization

For obtaining the convention of transistors I show in the list, I apply pre-customization as follows:

```
\ctikzset{
    tripoles/mos style=arrows,       % Arrow style for MOSFETs
    transistors/arrow pos=end,       % Arrow at the end
    tripoles/pmos style/nocircle,    % No circle on the PMOS gate
}
```

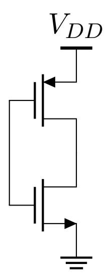
When needed, each STYLE-OPTION can be removed for further customization.

4.2.2 Supply voltage suggestion

For supply voltage, instead of the `vcc` suggest `tground` with label above:

```
    ... node[tground, label=above:$V_{DD}$]{};
```

Example 4.2.1. Design an inverter.



```
\begin{circuitikz}
\draw (0,0) node[tlground]{}
    node[nmos, anchor=S](nmos){}
    (nmos.D) node[pmos, anchor=D](pmos){}
    (pmos.S) node[tground, label=above:$V_{DD}$]{}
    (nmos.G) -- (pmos.G);
\end{circuitikz}
```

4.3 Main component customizations

Component customizations can be applied using `\ctikzset{}` Environment. You can also apply the customization locally as OPTION in the `to[...]` ofr path-style components or in the `node[...]` for node-style.

When describing geometrically a path-style, refer to an horizontal direction of drawing.

4.3.1 Resistors

Key	**Default**	**What It Controls**
`resistors/scale`	1.0	Overall size (use it to set Height)
`resistors/width`	0.8	Width (horizontal size)
`resistors/zigs`	3	Number of zig-zag segments (American style only)
`resistors/thickness`	2.0	Line thickness

Table 4.1: Customization keys for resistive components in Circuitikz

4.3.2 Capacitors

Key	**Default**	**What It Controls**
`capacitors/scale`	1.0	Overall size
`capacitors/width`	0.8	Width (in-plates distance)
`capacitors/height`	0.6	Height (plate sizes)
`capacitors/thickness`	2.0	Line thickness

Table 4.2: Customization keys for capacitors in Circuitikz

Note that capacitors allow for setting of both the width and the height.

4.3.3 Inductors

Key	**Default**	**What It Controls**
`inductors/scale`	1.0	Overall size
`inductors/width`	0.6 (cute), 0.8 (else)	Width (horizontal size)
`inductors/coils`	5 (cute), 4 (else)	Number of coils
`inductors/thickness`	2.0	Line thickness

Table 4.3: Customization keys for inductors in Circuitikz

4.4 Transistors

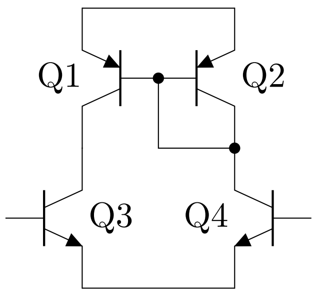
Transistors are fundamental active components used in amplification, switching, and digital logic. In Circuitikz, both BJTs (Bipolar Junction Transistors) and FETs (Field-Effect Transistors) can be represented with standardized schematic symbols

4.4.1 Transistor anchors

For NMOS, PMOS, NFET, and PFET transistors one has base, gate, source and drain anchors (B, G, S and D). For NPN and PNP transistors, the anchors are base, emitter and collector anchors (B, E and C).

Instantiate a transistor using nodes and other transistors' anchors.

Example 4.4.1. Design a diffential stage.

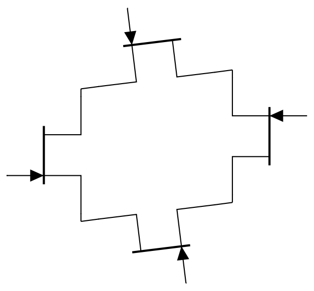


```
\begin{circuitikz}
    \draw (0,0) node[pnp] (pnp2) {Q2}
        (pnp2.B) node[pnp, xscale=-1, anchor=B]
            (pnp1) {\ctikzflipx{Q1}}
        (pnp1.C) node[npn, anchor=C] (npn1) {Q3}
        (pnp2.C) node[npn, xscale=-1, anchor=C]
            (npn2) {\ctikzflipx{Q4}}
        (pnp1.E) -- (pnp2.E) (npn1.E) -- (npn2.E)
        (pnp1.B) node[circ] {} |- (pnp2.C) node[circ] {};
\end{circuitikz}
```

4.4.2 Transistor paths

When convenient, standard transistors can be placed directly along a path using the usual **bipole syntax**. To specify a transistor, simply prefix its node name with a "T" (for transistor).
To refer the transistor as a node use the name OPTION, as in `to[..., n=NAME]`.

Example 4.4.2. Define a closed transistor path.



```
\begin{circuitikz}
    \draw (0,0) to[Tnjfet] (0,2) to[Tnjfet] (2,2)
        to[Tnjfet] (2,0) to[Tnjfet] (0,0);
\end{circuitikz}
```

4.4.3 Transistor customization

The complete graphics configuration are available accesing the circuitikz.sty online. I gathered the main style options for transistors

`\ctikzset{tripoles/nmos/... } \ctikzset{tripoles/pmos/... }`

`Key`	**Default**	**Description**
`width`	0.7	The total width of the transistor body.
`gate height`	0.35	Height of the gate line within the transistor body.
`base height`	0.5	Vertical distance from the bottom of the transistor to the base of the gate.
`conn height`	0	Vertical offset of the connection points (source/drain).
`height`	1.1	Overall height of the transistor.
`base width`	0.5	Width of the base of the transistor body.
`gate width`	0.62	Width of the gate line.
`arrow pos`	0.6 (NMOS) / 0.4 (PMOS)	Relative position of the arrow on the source/drain to indicate current direction.
`bodydiode scale`	0.3	Scale factor for the body diode symbol inside the transistor.
`bodydiode distance`	0.3	Distance of the body diode from the transistor body.
`bodydiode conn`	0.6	Relative position along the connection line for the body diode attachment.
`curr direction`	1 (NMOS) / -1 (PMOS)	Direction of the current arrow; 1 for NMOS, -1 for PMOS.

```
\ctikzset{tripoles/nmos/... }
\ctikzset{tripoles/pmos/... }
```

`Key`	**Default**	**Description**
`width`	0.6	Total width of the transistor body.
`base height`	0.4	Vertical distance from the bottom to the base pin.
`base height 2`	0.15	Secondary base height used for special positioning within the transistor.
`conn height`	0	Vertical offset of the connection points (collector/emitter).
`height`	1.1	Overall height of the transistor.
`base width`	0.5	Width of the base of the transistor body.
`arrow pos`	0.5	Relative position of the arrow on the emitter/-collector to indicate current flow.
`bodydiode scale`	0.3	Scale factor for the body diode symbol inside the transistor.
`bodydiode distance`	0.3	Distance of the body diode from the transistor body.
`bodydiode conn`	0.6	Relative position along the connection line for the body diode attachment.
`curr direction`	1 (NPN) / -1 (PNP)	Direction of the current arrow; 1 for NPN, -1 for PNP.

4.5 Logic ports

A standard selection for logic gates I use IEEE, using `\ctikzset{logic port=ieee}`. In alternative, you can also choose between `american` or `european`.

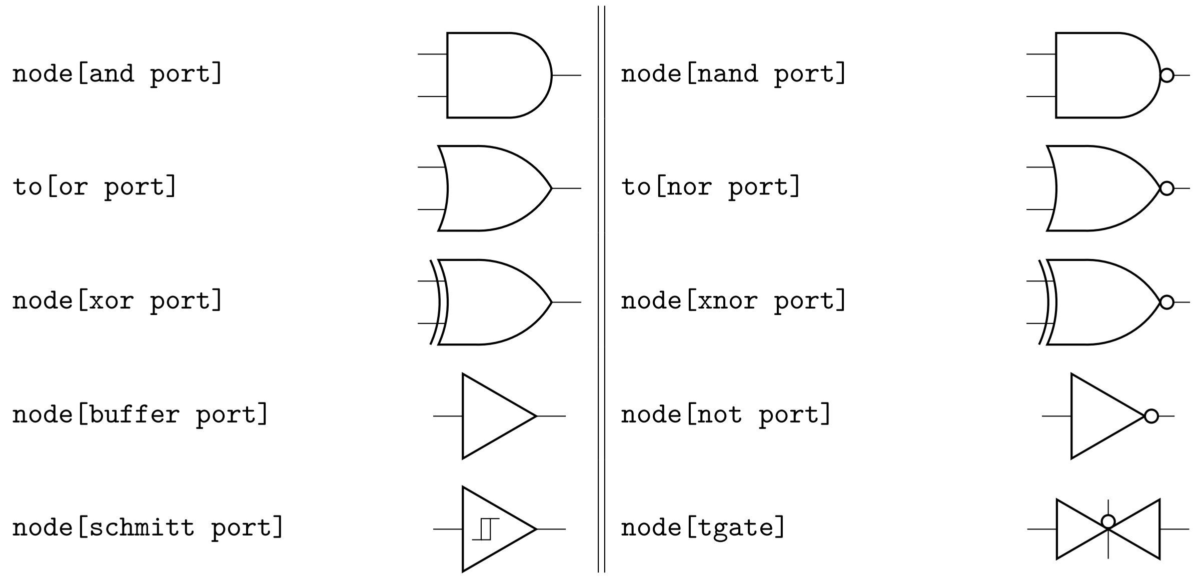


Figure 4.1: IEEE standard family of logic gates.

4.5.1 Logic ports anchors

All gates allow for several anchors:

- Cardinal points, as all nodes.
- `right, left`, directly on the gate body.
- `in 1, in 2, ..., out`, pins that extend out from the gate.
- `bin 1, bin 2, „„ bout`, body-pins, located directly on the gate body.
- `ibin1, ibin2,...`, inner pins of XOR gates.

4.5.2 Logic ports customization

To move the gate around use node OPTIONS. I suggest:

- `scale=, xscale=, yscale=` for adjusting sizes.
- `rotate=` for defining a degree angle.
- `xscale=-1, yscale=-1` to flip along vertical and horizontal axis.

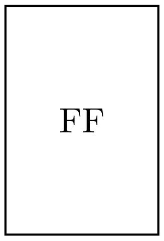
Key	Default	Description
`baselen`	0.4	The basic length for every dimension, as a fraction of the (scaled) resistor length.
`height`	2	The height of the port, in terms of `baselen`. Pin distance is given by this parameter divided by the inner pins.
`pin length`	0.7	Length of the external pin leads that are drawn with the port. This length is always calculated starting from the inner body of the shape.
`not radius`	0.154	Radius of the “not circle” added to the negated-output ports. The default value is the IEEE recommended one.
`xor bar distance`	0.192	Distance of the detached input shape in xor and xnor ports. The default value is the IEEE recommended one.
`xor leads in`	1	If set to 0, there will be no leads drawn between the detached input line and the body in the xor and xnor ports. IEEE recommends 1 here.
`schmitt symbol size`	0.3	Size of the small Schmitt symbol to use near input leads.

Figure 4.2: Customizations taken from CircuitikZ

For IEEE standard gates, the following customizations are available:

4.6 Flip-flops

The default flip-flop is empty: it appears as a simple rectangular box, like a blank DIP chip with 8 customizable pins: 6 lateral, 1 north, 1 south:



```
\begin{circuitikz}
    \draw (0,0) node[flipflop]{FF};
\end{circuitikz}
```

As you can see, in a void flip-flop no external pins are drawn: you need to define the meaning of each pin to make them visible.

4.6.1 Flip-Flops library

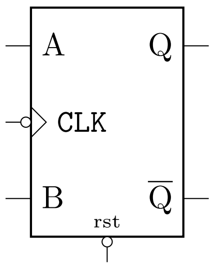
Several standard flip-flops are selectable off-the-shelf: `latch`, `flipflop SR`, `flipflop D`, `flipflop T`, `flipflop JK`.
Default visualization uses the $\overline{\mathrm{Q}}$ convention. To subvert it, TikZ allows for a **`dot on notQ`** OPTION, and a circle on the pin will appear instead.
Other OPTIONS are **`add asynch SR`**, to add vertical asynchronous $\overline{\mathrm{SET}}$ and $\overline{\mathrm{RST}}$, and **`external pins width=0`** to set pin widhts.

4.6.2 Custom Flip-Flops

To define a specific flip-flop, you must set a series of keys under the `\ctikzset` directory `multipoles/flipflop/`, corresponding to pins 1–6, and the up (`u`) and down (`d`) signals:

- **Text labels:** `t0, t1, ..., t6, tu, td` — sets a label on the corresponding pin.
- **Clock wedge flags:** `c0, ..., c6, cu, cd` — with values 0 or 1, draws a triangle shape on the border of the corresponding pin.
- **Negation flags:** `n0, ..., n6, nu, nd` — with values 0 or 1, draws an `ocirc` shape on the outer border of the corresponding pin.

To simplify the setup of these keys, an auxiliary style `flipflop def` is provided. For example:

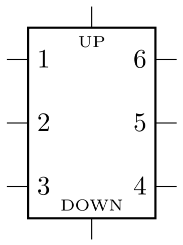


```
\tikzset{flipflop AB/.style={
    flipflop,
    flipflop def={
        t1=A, t3=B, t6=Q, t4={\ctikztextnot{Q}},
        td=rst, nd=1, c2=1, n2=1, t2={\texttt{CLK}}
    }
}}
\begin{circuitikz}
    \draw (0,0) node[flipflop AB]{};
\end{circuitikz}
```

4.6.3 Flip-Flops anchors

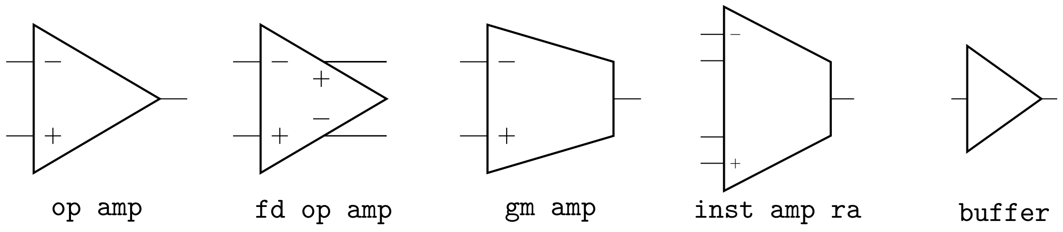
Flip-flop use anchors similar to logic gates:

- Cardinal points, as all nodes.
- `pin 1, pin 2, ..., pin 6, up, down`, pins that extend out from the gate(anticlockwise).
- `bpin 1, bpin 2, ..., bpin 6, bup, bdown`, body-pins located on the gate body (anticlockwise).



4.7 Amplifiers

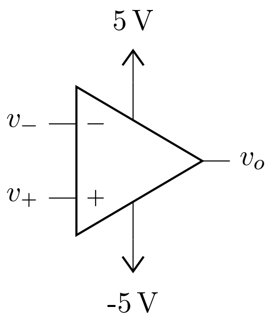
Circuitikz provides a library of amplifier types and customization options.



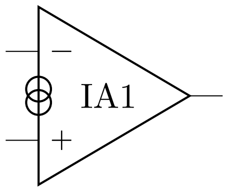
4.7.1 Amplifier anchors

Amplifiers have anchors and body-anchors,

- Cardinal points, as all nodes
- Anchors: `+`, `-`, `out`, `up`, `down`
- Body anchors: just add "b" at any anchor (`b+`, `b-`, `bout`, ...)
- `leftedge`: used for components between input pins.



Example 4.7.1. Draw a Norton Amplifier.



```
\begin{circuitikz}
    \draw (0,2.2) node[op amp](OA){IA1};
    \node[oosourceshape, rotate=90, scale=0.5]
    at (OA.leftedge) {};
\end{circuitikz}
```

4.8 Custom Symbols creation

A new symbol can be defined from existing one, when multiple version are needed.
Use the `/.style={OPTIONS}` to create a custom component usable later.
This is akin to what done for custom block definitions in Section 3.6.2.

```
\ctikzset{
        myR/.style={
        american resistor,
        resistors/.cd,
        scale=0.8, width=1.5, zigs=5, thickness=2.1
    },
    myL/.style={
        cute inductor,
        inductors/.cd,
        scale=0.9, width=1.2, coils=6
    },
    myFF/.style={flipflop D,
        flipflop def={t1=J, t2=K, t6=Q, t4={\ctikztextnot{Q}}, c3=1},
        scale=1.2,
    },
    ieeestd ports/.cd,
    not radius=0.1, xor bar distance=0.3, xor leads in=0
}
```

then use `myR` and `myL` in your path definitions, `myFF` as your flip-flops. Global definition or custom block appear as set in the `\tikzset` Environment, unless further specified. This means that global specifications can be overridden during block instanciation.

4.8.1 Change directory (.cd)

Note the use of `/.cd`. This is the **Change Directory operator**.
Suppose you wanted to set a new resistor with custom scale, width, zigs, and thickness. That would mean to express the `resistor/...` directory four times, which may become cumbersome. Instead, TikZ enables writing `resistor/.cd` as change directory command, and then just refer to the style entries. After this, a new /.cd would be required to change other OPTIONS.

```
    ieeestd ports/.cd, not radius=0.1, xor bar distance=0.3
```

4.9 Exercises

Exercise 4.9.1. Create a resistor with: 6 zigs, increased thickness, reduced height.

Exercise 4.9.2. Place an NPN transistor and add: a collector resistor to VCC, an emitter resistor to ground, a base bias resistor. Use only transistor anchors (B, C, E).

Exercise 4.9.3. Recreate a differential pair using two mirrored MOSFET. Use xscale=-1, `\ctikzflipx{}`. Avoid absolute coordinate guessing.

Exercise 4.9.4. Using IEEE logic gates: create two AND gates, feed their outputs into an OR gate, add a NOT gate at the final output.

Exercise 4.9.5. Modify the IEEE gate appearance: reduce not radius, remove XOR leads, change base length. Use .cd properly to avoid repetition.

Exercise 4.9.6. Draw a D flip-flop with: clock wedge, asynchronous reset, negated Q output. Then: connect D to a logic gate and connect Q to an output node.

Exercise 4.9.7. Define a custom flip-flop using flipflop def: two inputs A and B, clock input, Q and not-Q outputs. Scale it and instantiate it twice.

Exercise 4.9.8. Draw a complete inverting amplifier including power rails. Try using a feedback coordinate and compare to the advanced branching method.

Exercise 4.9.9. Instantiate an instrumentation amplifier and connect a gain resistor.

Exercise 4.9.10. Create a reusable macro: `\MyInverter{name}{position}`, that automatically generates: PMOS, NMOS, VDD node, ground, output node. Instantiate it three times aligned horizontally.
Hint: use the `\newcommand` syntax

Chapter 5

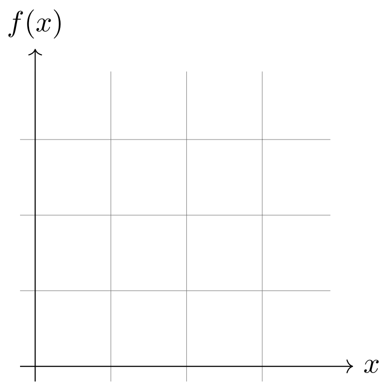
Plotting functions and data

5.1 Plotting using `\draw`

5.1.1 Coordinate system

Constructing a **simple coordinate system** in TikZ is straightforward. In the first quadrant, we only need two axes using `\draw` ,and optionally a grid to guide drawing a `grid` between two opposite anchors.

Example 5.1.1. Construct coordinate axis for the first quadrant of the plane.



```
\begin{tikzpicture}

  \draw[very thin, gray] (-0.2,-0.2) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4.2) node[above] {$f(x)$};

\end{tikzpicture}
```

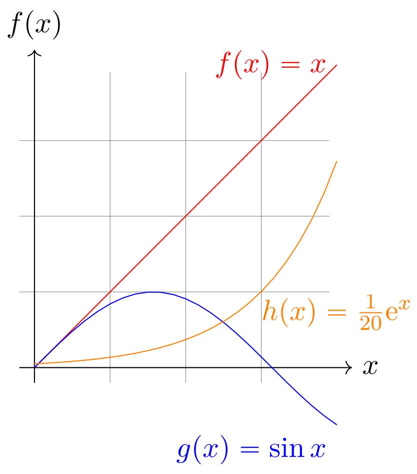
5.1.2 `\draw` plotting

When **plotting a function**, TikZ can automatically compute coordinates from a mathematical expression using its built-in math engine. Use the `plot` command, and then provide a coordinate expression that can use the special macro \x (or another variable via the variable option). The domain needs to be specified, globally or locally.

Plotting can be done using conventional `(\x, f(\x))` notation, or by defining a `variable=\t` in the OPTIONS, and using parametrized notation `( g(\t), h(\t) )`.

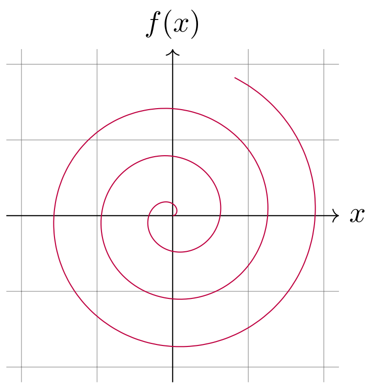
Note: Trigonometric function use the form `\sin(\x r)`, where the `r` represent values given in **radiants**.

Example 5.1.2. Plot multiple function on the coordinate system.



```
\begin{tikzpicture}[domain=0:4]

    \draw[very thin, gray] (-0.2,-0.2) grid (3.9,3.9);

    \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
    \draw[->] (0,-0.2) -- (0,4.2) node[above] {$f(x)$};

    \draw[red]    plot (\x,\x)
        node[left] {$f(x)=x$};
    \draw[=blue]   plot (\x, {sin(\x r)})
        node[below left] {$g(x)=\sin x$};
    \draw[orange] plot (\x, {0.05*exp(\x)})
        node[yshift=-2cm]
        {$h(x)=\frac{1}{20}\mathrm e^x$};

\end{tikzpicture}
```

Example 5.1.3. Plot the Archimedean spiral.



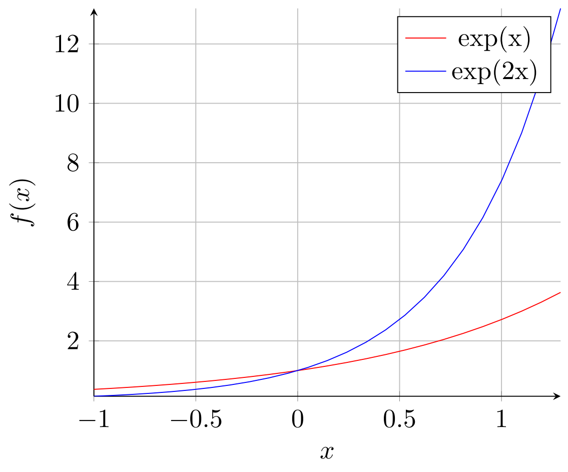
```
\begin{tikzpicture}

    \draw[very thin, gray] (-2.2,-2.2) grid (2.2,2.2);

    \draw[->] (-2.2,0) -- (2.2,0) node[right] {$x$};
    \draw[->] (0,-2.2) -- (0,2.2) node[above] {$f(x)$};

    \draw[purple, scale=0.5,domain=0:20,
        smooth, variable=\t, samples=200]
        plot ({0.2*\t*cos(\t r)},{0.2*\t*sin(\t r)});

\end{tikzpicture}
```

5.2 Plotting using the pgfplots

```
\usepackage{pgfplots}
\pgfplotsset{width=10cm,compat=1.9}
```

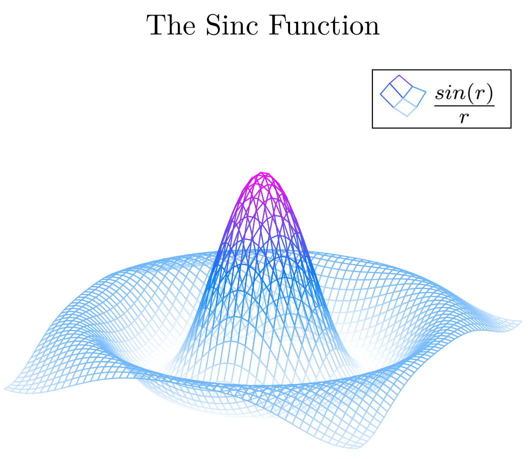
An easy alternative to manual drawing of axis and plots is the PGF package. Axis creation is automated.

Example 5.2.1. Plot the exponential function.



```
\begin{tikzpicture}
\begin{axis}[
    axis lines = left,
    xlabel = \(x\),
    ylabel = {\(f(x)\)},
    domain=-1:1.29 , grid=major
    ]

    \addplot[red]{exp(x)};
    \addlegendentry{exp(x)}
    \addplot[blue]{exp(2*x)};
    \addlegendentry{exp(2x)}

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

Example 5.2.2. Plot the sinc function in 3D.



```
\begin{tikzpicture}
\begin{axis}[
    title=The Sinc Function,
    hide axis,
    colormap/cool,
    ]
    \addplot3[
        mesh,
        samples=50,
        domain=-8:8,
    ]
    {sin(deg(sqrt(x^2+y^2)))/sqrt(x^2+
    y^2)};
    \addlegendentry{\(\frac{sin(r)}{r
    }\)}
\end{axis}
\end{tikzpicture}
```

5.2.1 The axis Environment for functions

All plots created with the `pgfplots` package are contained inside the `axis` environment. This environment automatically generates coordinate axes, scales, ticks, and labels, eliminating the need for manual drawing. The appearance and behavior of the axes are controlled through key-value OPTIONS.

2-D function axis plotting

Multiple functions can be plotted within the same `axis` environment. Each `\addplot` command adds a new curve while preserving the same coordinate system. Express the function **using the variable `x`**. Legends are created using `\addlegendentry`.

3-D function axis plotting

When using `\addplot3`, the `axis` environment automatically switches to a three-dimensional coordinate system. Express the function **using the variable `x` and `y`**

- The `mesh` option creates a wireframe surface representation.
- `samples=50` controls the resolution of the plotted surface.
- `domain=-8:8` defines the range for both x and y.

In the next section is a complete table for all OPTIONS appliable to any `addplot`.

5.2.2 Plotting .PGF files

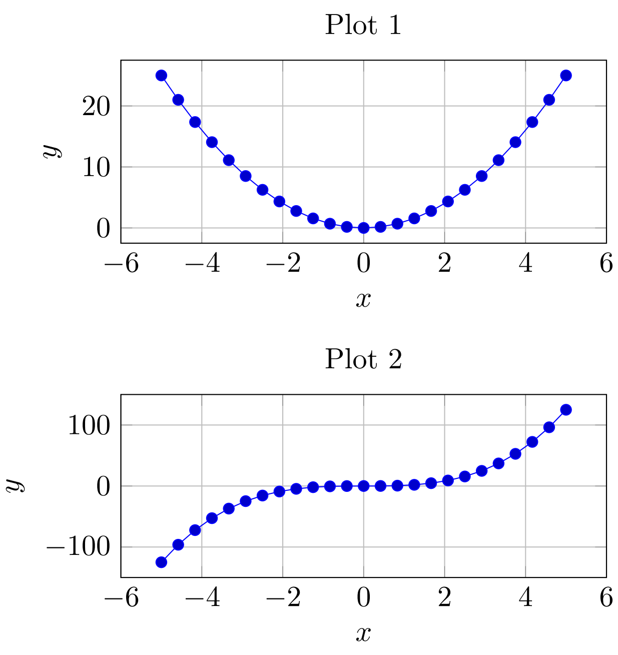
The package also allows the direct inclusion of .PGF figures within the document, ensuring full compatibility with the document fonts and layout. With this approach, the axes do not need to be defined in the LaTeX source, since they can be generated externally and imported directly (e.g., from Python or MATLAB).

```
\begin{figure}
    \centering
    \input{Files Chap4/RO_frequencies/midFreq.pgf}
    \caption{Medium frequency range with different slopes}
    \label{fig:med_freq}
\end{figure}
```

Refer to Chapter 6 for details on generating PGF files with Python.

5.2.3 The `groupplot` Environment for multiple plots

Example 5.2.3. Display two plots.



```
\begin{tikzpicture}
    \begin{groupplot}[
        group style={
        group size=1 by 2,
        vertical sep=2cm},
        width=8cm, height=4cm,
        grid=both, xlabel={$x$},
        ylabel={$y$}
    ]

    \nextgroupplot[title={Plot 1}]
    \addplot {x^2};

    \nextgroupplot[title={Plot 2}]
    \addplot {x^3};

    \end{groupplot}
\end{tikzpicture}
```

Note that these are line and circles are the default visualization OPTIONS for `\addplot`.

Add the OPTION common to all plots in the `groupplot` OPTIONS, while the use the `nextgroupplot` OPTIONS to customize plot by plot.

I suggest to always use the following scheme:

```
\begin{groupplot}[
    group style={
        group size=X by Y,
        vertical sep=...,
        horizontal sep=...},
        OPTIONS_ALL
]

    \nextgroupplot[OPTIONS_1]
    \addplot {x^2};

    \nextgroupplot[OPTIONS_2]
    \addplot {x^3};
\end{groupplot}
```

5.2.4 The axis Environment for external .CSV data

In addition to plotting analytical functions, `pgfplots` can plot numerical data stored in external files. This is achieved using the `table` plotting method, instead of the `{f(x)}` expression.

The following command plots data read from a .CSV file:

`\addplot[OPTIONS] table[FILE_SPECS] {filename.csv};`

For example,

```
\addplot[line width=1.0pt, blue]
    table[
    col sep=comma,
    x=x,
    y=y,
    each nth point=20
    ] {Files_Chap3/nmos_1dev.csv};
```

```
% nmos_1dev.csv FILE
x       ,y
0       ,0.4098898850268283
0.05    ,0.3605040012037113
0.1     ,0.3107999121069504
0.15    ,0.2643973013984135
0.2     ,0.2232355820165562
...
```

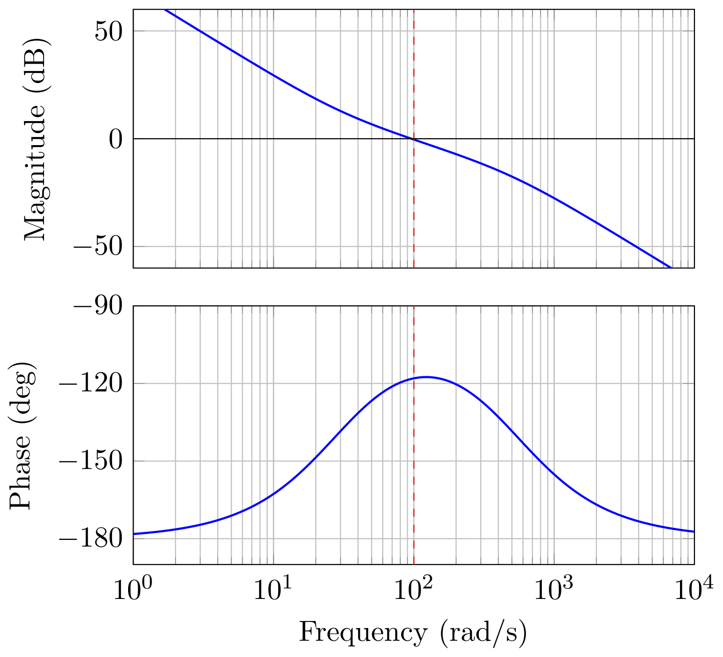
The external file must be a text-based table, such as a **CSV file**, containing column headers. Each column represents a variable that can be mapped to an axis.
File Specifications [FILE_SPECS] are needed in order to read the numbers from the file and assign them correctly in the plot. They are usually:

- **`col sep=comma`** specifies that the columns in the file are separated by commas.
- **`x=x`** selects the column named `x` for the horizontal axis.
- **`y=y`** selects the column named `y` for the vertical axis.
- **`each nth point=20`** downsamples the data.

Its good use to collect al CSV files in ordered folders, divided by chapters or sections, for better organization of the file and future readability.

5.2.5 The axis Environment for Bode Plots

Example 5.2.4. Display the Bode plots of magnitude and phase of a Type-II transfer function.



```
\begin{tikzpicture}[scale=1]
\begin{groupplot}[
    group style={
        group size=1 by 2,  vertical sep=0.5cm,
        x descriptions at=edge bottom,
    },
    xmode=log, width=7cm, height=4.5cm,
    grid=both, xmin=1, xmax=10000,
    xlabel={Frequency (rad/s)},
]

    \nextgroupplot[ ylabel={Magnitude (dB)}, ymin=-60, ymax=60 ]
        \addplot[blue, thick, domain=1:10000, samples=200]
            {20*log10(2800) + 10*log10((x/30)^2 + 1)
             - 40*log10(x) - 10*log10((x/500)^2 + 1)};
        \draw[black, thin] (axis cs:1,0) -- (axis cs:10000,0);
        \addplot[red, dashed] coordinates {(100, -60) (100, 60)};

    \nextgroupplot[ ylabel={Phase (deg)}, ymin=-190, ymax=-90,
        ytick={-180, -150, -120, -90} ]
        \addplot[blue, thick, domain=1:10000, samples=200]
            {-180 + atan(x/30) - atan(x/500)};
        \addplot[red, dashed] coordinates {(100, -190) (100, -90)};

\end{groupplot}
\end{tikzpicture}
```

To construct a Bode diagram using the `axis` environment, combine two logarithmic plots—one for the magnitude and one for the phase—stacked vertically and sharing the same frequency axis.

The `groupplot` environment allows multiple axes to be arranged in a grid. For Bode plots, a single column with two rows is typically used, where the upper plot represents the magnitude response and the lower plot represents the phase response.

The `groupplot` OPTIONS needed in for Bode plots are:

- `group size=1 by 2`: creates a vertical arrangement with two stacked plots.
- `vertical sep`: controls the spacing between them.
- `xmode=log`: enables logarithmic frequency x-axis.

The magnitude plot is expressed in decibels and is computed using the dB definition If

$$L(s) = \frac{2800}{s^2} \cdot \frac{\frac{s}{30} - 1}{\frac{s}{500} - 1}$$

then:

```
    {20*log10(2800) + 10*log10((x/30)^2 + 1)
    - 40*log10(x) - 10*log10((x/500)^2 + 1)};
```

The phase plot is displayed in degrees and is obtained by summing the phase contributions of each factor of the transfer function.

```
    {-180 + atan(x/30) - atan(x/500)};
```

5.2.6 Complete `axis` OPTIONS summary

Figure 5.1 reports all the main OPTIONS applicable to

- **`groupplot`**: valid for all pictures in a matrix group.
- **`nextgroupplot`**: valid for single pictures in a matrix group.
- **`axis`**: valid for a picture of trances
- **`addplot`**: valid for a single trace.
- **`addplot3`**: valid for a single trace in 3D.

Option	**Description**
width=, height=	Define dimentions of the plot
xmode=log, ymode=log	Changes the x-axis or y-axis (default linear)
x expr=ln(\x), y expr=ln(\x)	Changes the x-axis or y-axis to custom function.
xlabel= / ylabel=	Define labels for the horizontal and vertical axes.
hide axis	Removes axis lines and labels for a cleaner 3D visualization.
colormap/cool	Applies a predefined color gradient to the surface.
axis lines = left	Displays only the left and bottom axes. By default, a box axis system is drawn.
domain=-1:1.29	Specifies the range of the independent variable used by all plots unless overridden.
title=	Places a title above the plot.
xmin, xmax, ymin, ymax	Puts constraints on the visualization intervals for each axis.
xtick / ytick	Redefines the axis ticks, e.g., `xtick={0,20,40,60,80,100}`.
legend pos	Relocates the legend, e.g., `legend pos=north west`.
grid	Adds grid lines: `grid=major/minor/both` .
grid style	Sets style for all grids, e.g., `grid style=dashed`.
ymajorgrids / xmajorgrids	Turns on major grids and allows customization.
yminorgrids / xminorgrids	Turns on minor grids and allows customization.

major grid style={gray!70}
minor grid style={gray!70}
	Customize each grid

`xminorgrid` style={gray!70}	Customize only `xminorgrid`

Table 5.1: Common pgfplots axis options and their descriptions.

5.3 Plotting timing diagrams

Timing diagrams can be constructed by representing digital signals and the transitions between their logic levels over time. A digital signal evolves in discrete steps and, at each cycle or time interval, it can perform only one of three fundamental actions: **remain** at the same level, transition from **low to high**, or transition from **high to low**.

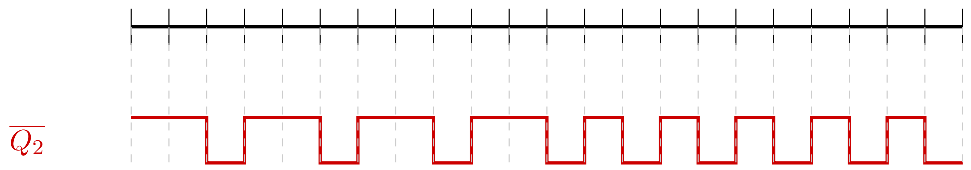
Therefore, a minimal set of drawing commands is sufficient to describe any timing diagram. My commands defined as follows:

```
\newcommand{\1}{ --++(0,0.5)  }  % Low - high transition
\newcommand{\0}{ --++(0,-0.5) }  % High - low transition
\newcommand{\5}{ --++(2,0)    }  % Signal remains constant
```

To create uniform ticks on an axis exploit the foreach loop, and use a dashed vertical lines to align all diagrams.

Example 5.3.1. Plot a timing diagram.

```
\begin{tikzpicture}[x=0.25cm,y=1.2cm]
\def\yA{1}
\def\yB{0}
    \draw[very thick] (0,\yA) -- (44,\yA);                  % Thicked axis
    \foreach \x in {0,2,...,44} {
        \draw (\x,\yA-0.2) -- (\x,\yA+0.2);
    }

    \draw[very thick, red!80!black] (0,\yB)                 % Red signal
    \5\5\0\5\1\5\5\0\5\1\5\5\0\5\1\5\5
    \0\5\1\5\0\5\1\5\0\5\1\5\0\5\1\5\0\5\1\5\0\5;
    \node[red!80!black] at (-5.5,-0.25) {$\overline{Q_2}$};

    \foreach \x in {0,2,...,44} {                           % Vertical dashes
        \draw[dashed, gray!40] (\x,1) -- (\x,-0.5);
    }
\end{tikzpicture}
```



Note the use of the `\def\y` command to define global y-level to place and align timing diagrams.

Chapter 6

Creating PGF in Python

6.1 The `matplotlib` library

The matplotlib library is one of the most widely used tools in Python for data visualization and scientific plotting. A particularly important feature of matplotlib is its ability to export figures using the PGF backend. PGF is a graphics format designed to integrate seamlessly with LaTeX documents, allowing plots to be rendered using the same fonts, mathematical notation, and styling as the surrounding text. This ensures visual consistency and typographical quality that is difficult to achieve with standard image formats such as PNG or JPEG.

A typical matplotlib workflow begins with the creation of a figure and one or more axes, followed by the addition of graphical elements such as lines, markers, text, and annotations. Each of these elements can be extensively customized in terms of color, style, size, and positioning. When the PGF backend is enabled, these properties are translated into corresponding LaTeX and PGF instructions, allowing LaTeX to handle font rendering and mathematical typesetting directly.

6.2 Pyplot module for plotting PGF

```
import matplotlib.pyplot as plt
```

pyplot is a module in Matplotlib designed to provide a **simple interface for creating plots**. It offers a collection of functions that allow you to generate figures, add plots, configure axes, labels, titles, legends, and customize visual styles. Each pyplot function operates on the **current figure and axes**.

When you call `plt.subplots()`, it returns a figure object (fig) and one or more axes objects (ax). The figure acts as a container for all plot elements, while the axes define the coordinate system and space where the actual plotting occurs.

My suggestion is to **create many axis** using, for example:

```
fig, (ax1, ax2, ...) = plt.subplots( ROWS, COLUMNS, figsize= (X,Y) )
```

and **then populate each axis using functions**, where each axis is passed as a parameter.

Example 6.2.1. Plot the functions x^2 and x^3 in Python and export in PGF.

```python
import numpy as np
import matplotlib.pyplot as plt

# Selection: PGF{1} or PLOT{0}
PGF = 0

# Functions to generate y-values
def draw_x2(ax):
    x = np.linspace(-10, 10, 400)
    ax.plot(x, x**2, label="y = x^2", color="blue")
def draw_x3(ax):
    x = np.linspace(-10, 10, 400)
    ax.plot(x, x**3, label="y = x^3", color="red")

# Common styling for all axes
def style_axis(ax):
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.grid(True)
    ax.legend()

# Create figure and axes
fig1, (ax1, ax2) = plt.subplots(2, 1, figsize=(11, 6))
draw_x2(ax1)
draw_x3(ax2)

# Apply common styling once for both
for ax in (ax1, ax2):
    style_axis(ax)
plt.tight_layout()

# Save as PRINT PGF or PLOTTING IN WINDOW
fig1.savefig("plots.pgf") if PGF else plt.show()
```

In each function, the axis is passed as a parameter and is manipulated using pyplot functions.

The last line allows **either printing a .PGF file, or plotting on window**, depending on a selector.

Therefore, Python allows you not only to conduct complex computations and obtain results efficiently but also to visualize them through high-quality plots, supporting multiple formats, including images, vector graphics, and even LaTeX-compatible PGF files.

6.3 Customizing exporting folder

My suggestion is to **save the .PGF file in the same directory as the Python script** to keep the workflow organized and self-contained. This ensures that the generated figure is always alongside the code, simplifying version tracking, sharing, and inclusion in LaTeX documents.

```python
import os

# Set the directory and name to create path
script_dir = os.path.dirname(os.path.abspath(__file__))
output_file = "figure.pgf"
output_path = os.path.join(script_dir, output_file)

# Plotting
.....

# Save PGF file and show preview
plt.tight_layout()
plt.savefig(output_path, bbox_inches="tight")
plt.show()
```

6.4 Import PGF to LaTeX

Similarly to what done fior .tex drawings, also for .PGF use the Figure Environment with input command.

```latex
\begin{figure}
    \centering
    \input{Chap2/PGF/voltages.pgf}
\end{figure}
```

6.5 PGF plot customization

When designing your plot, many options are customizable for plotting.

```python
import matplotlib.pyplot as plt

plt.rcParams.update({
    # --- LaTeX integration ---
    "pgf.texsystem": "pdflatex",
    "text.usetex": True,
    "pgf.rcfonts": False,
    "font.family": "serif",
    "font.size": 10,              # default LaTeX article font
    "axes.labelsize": 11,
    "axes.titlesize": 12,
    "xtick.labelsize": 10,        # similar to \footnotesize
    "ytick.labelsize": 10,
    "legend.fontsize": 8

    "figure.figsize": (6, 4),     # width, height in inches
    "figure.dpi": 300
})

# Plotting options
...


# Save PGF file and show preview
plt.tight_layout()
plt.savefig(output_path, bbox_inches="tight")
plt.show()
```

6.5.1 The importance of `plt.show()`

Calling `plt.show()` opens a foating pop-up window displaying the plot at its actual size. You can position it over a LaTeX page zoomed to 100% to visually compare dimensions. In fact, when the .PGF file is included in the document, the plot will match the size of this floating window.

Use this method to adjust figure dimensions and font sizes until the desired proportions are achieved.

6.6 Complete Matplotlib OPTIONS summary

Figure 6.1 shows all OPTIONS applicable on pyplot axis.

Option	**Description**
figsize=(w,h)	Defines the dimensions of the figure in inches.
ax.set_xscale('log'), ax.set_yscale('log')	Changes the x-axis or y-axis scale (default is linear).
ax.set_xlabel(), ax.set_ylabel()	Defines labels for the horizontal and vertical axes.
ax.axis('off')	Removes axis lines, ticks, and labels for a cleaner visualization.
cmap='cool'	Applies a colormap to plotted data.
ax.spines['top'].set_visible(False), ax.spines['right'].set_visible(False)	Displays only selected axes instead of a full box.
ax.set_xlim(), ax.set_ylim()	Specifies the visualization interval for each axis.
ax.set_title()	Places a title above the plot.
ax.set_xticks(), ax.set_yticks()	Redefines axis ticks, e.g., ax.set_xticks([0,20,40]).
ax.legend(loc=...)	Relocates the legend, e.g., loc='upper left'.
ax.grid(True)	Enables grid lines on the plot.
ax.grid(which='major/minor/both')	Selects which grid lines to display.
ax.grid(linestyle=..., linewidth=..., color=...)	Sets style properties for all grid lines.
ax.minorticks_on()	Enables minor ticks for both axes.
ax.grid(which='minor')	Turns on minor grid lines.
ax.grid(which='major', color='gray', alpha=0.7)	Customizes major grid appearance.
ax.grid(which='minor', axis='x')	Customizes minor grid lines on the x-axis only.

Table 6.1: Common Matplotlib axis options and their descriptions.

6.7 Esercises

Exercise 6.7.1. Create two separate plots using Matplotlib.

- Plot y = sin(x) for x in [0, 2pi].
- Plot y = cos(x) for x in [0, 2pi].

Use ax.plot() for both and customize line color, line style, and labels for x and y axes.

Exercise 6.7.2. Take the plots from Exercise 6.1 and save them as .PGF files using `fig.savefig("plot.pgf")`. Then include the .PGF file in a LaTeX document using `\input{plot.pgf}` inside a figure Environment.

Exercise 6.7.3. Create a figure with 2 rows and 2 columns of subplots. Pass each axis as a parameter to a separate plotting function and apply a common styling function for labels, grid, and legend.

Exercise 6.7.4. Create a plot of y = tan(x) for x in [-pi/2 + 0.1, pi/2 - 0.1]. Enable major and minor grids, set major grid to gray with alpha 0.5, set minor grid to dashed blue lines. Include a legend and axis labels.

Chapter 7

Restyling and Fadings

7.1 Custom colors

```
\usapakage{xcolor}
```

7.1.1 Defining colors

This package allows to define custom colors under a new keyword, and later be used in texts or OPTIONS.

```
\definecolor{myblue}{RGB}{0,102,204}
\textcolor{myblue}{Custom colored text}
```

All colors classes are enabled: RGB ,CMYK ,HTML (hex values), Gray , Named colors.

7.1.2 Mixing colors

The xcolor package also allows you to create new colors by mixing existing ones.

```
draw[ blue!50!red ] ...
node[ fill= blue!50 ] ...
```

The syntax is explained as follows:

- `COLOR1!PERCENT!COLOR2`: specifies a blend where percentage determines how much of COLOR1 contributes to the final color, and the remainder comes from COLOR2.
- `COLOR1!PERCENT`: as the normal syntax, but taking `white` as COLOR2.

7.2 Custom Listings

```
\usepackage{listings}
```

7.2.1 Global listing style definition

Keep your listing definitions in separate file from the main.tex.

```
\usepackage{listings}
\usepackage{xcolor}
\input{listing_colors}
```

then in the file define colors and custom style, for example:

```
% Define colors for background and syntax highlighting
\definecolor{pybg}{HTML}{FFF9E6}
\definecolor{pykeyword}{RGB}{0,0,128}      % dark blue
\definecolor{pystring}{RGB}{163,21,21}     % dark red
\definecolor{pycomment}{RGB}{0,128,0}      % green
\definecolor{pyidentifier}{RGB}{0,0,0}     % black

\lstdefinestyle{pythonstyle}{           % Globall
    language=Python,                           % Set the programming language
    backgroundcolor=\color{pybg},              % Set the background color
    basicstyle=\ttfamily\small,                % Use monospaced font with small size
    keywordstyle=\color{pykeyword}\bfseries,   % Style Python keywords
    stringstyle=\color{pystring},              % Style string literals
    commentstyle=\color{pycomment}\itshape,    % Style comments
    identifierstyle=\color{pyidentifier},      % Style variable and function names
    frame=single,               % Draw a box/frame around the code block
    breaklines=true,            % Automatically break long lines
    columns=fullflexible,       % Improve alignment of characters
    keepspaces=true,            % Preserve spaces (important for Python indentation)
    showstringspaces=false      % Do not visually mark spaces inside strings
}
```

7.2.2 Custom global listings

Producing listings for TeX in Tex can be tricky, since custom commands recognize themselves as real commands and not as normal characters. That is kind of meta. The approach I adopt is to redefine plain words and control sequences with tow separate commands

Use `morekeywords=[k]` and `keywordstyle=[k]` for plain words. Use `literate=` for single character, control sequences and commands (in particular with backslash or $)..

```
\definecolor{codebg}{RGB}{245,245,245}      % Background
\definecolor{keyword}{RGB}{192,48,96}       % Draw commands
\definecolor{option}{RGB}{161,0,0}          % Options inside [...]
\definecolor{comment}{RGB}{128,128,128}     % Comments
\definecolor{nodes}{RGB}{0,92,161}          % Node commands
\colorlet{label}{green!70!black}


\lstdefinestyle{circuitikzstyle}{               % Globally
    ...
    % Draw-related keywords (blue)
    keywordstyle=\color{keyword}\bfseries,
    morekeywords={draw,path,to, controls, plot},
    % Node-related keywords (purple)
    keywordstyle=[2]\color{nodes}\bfseries,
    morekeywords=[2]{node,coordinate, fill},
    % Metachar commands
    literate=
        {\\draw}{{\color{keyword}\bfseries\textbackslash draw}}1
        {\\fill}{{\color{nodes}\bfseries\textbackslash fill}}1
        {--}{{{\color{keyword}\text{-}\text{-}}}}2
        {name\ path}{{\color{black}\bfseries name path}}9
        {\$f\_\{ref\}\$}{{\color{label}\$f\_\{ref\}\$}}1
        {\$\\overline\{Q\_2\}\$}{{\color{label}\$\textbackslash overline\{Q\_2\}\$}}1
        \{x\}\$}{{\color{label}\$h(x)=\textbackslash frac\{1\}\{20\}\textbackslash
    mathrm e\textasciicircum x\$}}1
}
```

The `literate` option is used to manually redefine how specific character sequences are printed inside a listings Environment. This is necessary when working with TeX-based code, because many elements—such as control sequences (\draw, \fill), special operators (-, _, $), or inline math expressions—are not treated as ordinary text. Instead of relying on automatic language rules, literate explicitly tells listings: *whenever this exact sequence appears, replace it with this formatted version.*

7.2.3 Additional local custom listing definitions

Suppose in a single listing of your code a word must be highlighted as keyword. You do not want to define it globally, because would affect all the listings. For this, local customization OPTIONS are available, where custom words can be add as keywords only for the current listing instance.

Use `\emph={[k]}` and `\emphstyle={[k]}`. Use `\alsoletter={}` to allow characters in the `emph` list.

```
\begin{lstlisting}[style=pythonstyle,                % Locally
    alsoletter={!},
    emph={We, will, be, blue, !},
    emphstyle=\color{blue},
    emph={[2]Color, us, of, red, instead},
    emphstyle={[2]\color{red}}
]

....
```

Diffence between keywordstyle and emphstyle

Although they achieve the same objective, mind use the first globally, while the second only locally. Although any command can be used at any scope without causing errors, I sugest the following:

- **`morekeywords`**, **`keywordstyle`** and **`literate`** prefer for global.
- **`emph`**, **`emphstyle`** and **`alsoletter`** prefer local additionals.

This rule is not imposed by TeX; it is simply my recommendation based on practical judgment.

The problem is that once a set of `morekeywords` is defined globally, it can be overridden locally, hence resetting the global list altogether in the instance. The same is true for , or any other command. For this reason, I suggest to keep glabal separate from local additionals.

7.3 Fading Package

```
\usetikzlibrary{fadings}
\input{fading_config}
```

The fadings library in TikZ provides a powerful mechanism for creating transparency gradients within graphical elements. if you need to smoothly blend your shapes into the background or create soft edges transition effects in diagrams, this can be useful. In circuit schematics and technical illustrations, fading can be used to emphasize active components, and de-emphasize secondary structures. Because fading operates at the transparency level, it integrates naturally with overlays, clipping paths, and layered drawings.

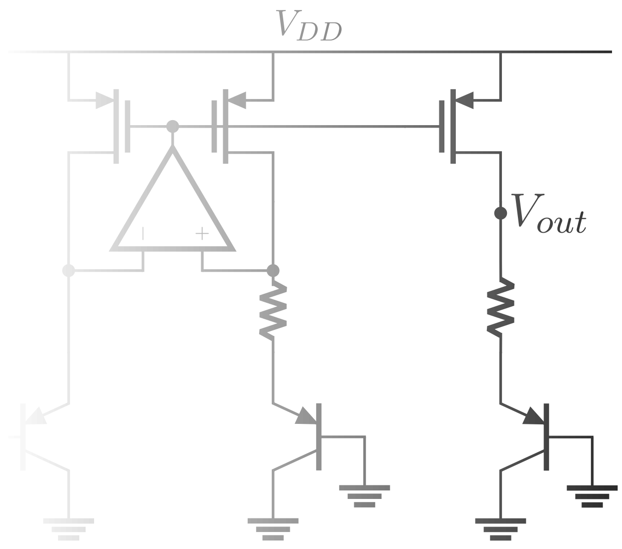


Figure 7.1: Fading bandgap

In the `fading_config` use the `\tikzfading` command to define fading colors. For fading direction, transparencies must be defined from start to finish. The fading package introduces a pseudo-color known as `transparent` which must be used along with opacity levels.

```
% Define fading for each side
\tikzfading[name=fade l, left color=transparent!100, right color=transparent!0]
\tikzfading[name=fade r, right color=transparent!100, left color=transparent!0]
\tikzfading[name=fade d, bottom color=transparent!100, top color=transparent!0]
\tikzfading[name=fade u, top color=transparent!100, bottom color=transparent!0]
```

`transparent!0` means fully transparent, `transparent!100` means fully opaque.

For example, `fade r` represent a fading which has the full color on the right and no color on the left, like in Figure 7.1.

When applying fading, define a filled rectangle, choose a color (usually white) and apply a fading OPTION using `path fading=` .

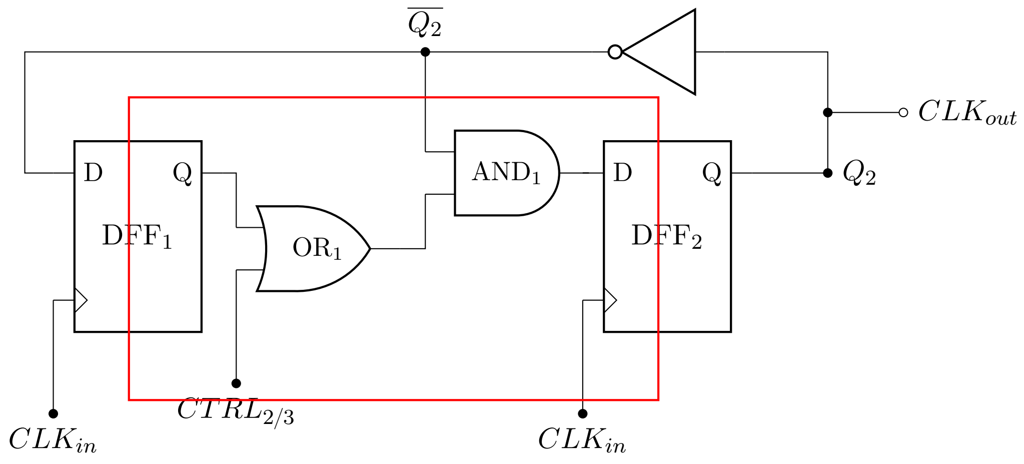
```
\begin{tikzpicture}
    ...
    \fill[white,path fading=fade r] (2,2) rectangle (-2,-2);
\end{tikzpicture}
```

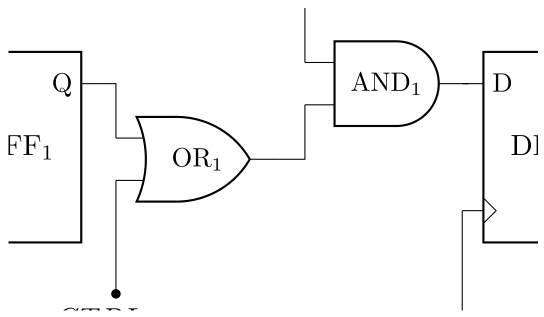
7.4 Clipping the picture

Clipping means restricting drawing to a specific region. Anything drawn after a `\clip` command will only appear inside the clipping area.
The clipping area can be a rectangle, but also any shape or closed path.

```
\begin{circuitikz}
    \clip (1,1) rectangle (8,-3);

    ..ORIGINAL CODE..
\end{circuitikz}
```

Example 7.4.1. Clipping a picture.





7.5 Itemize Settings

In LaTeX, the vertical spacing of list environments can be adjusted by modifying a few length parameters:

- `\itemsep`: controls the space between individual items.
- `\topsep`: sets the space added above and below the entire list.
- `\parsep`: defines the space between paragraphs within the same item.
- `\partopsep`: adds extra space when the list begins a new paragraph.
- `\leftmargin`: determines the indentation of the list relative to the surrounding text (better not to change).

These parameters can be locally adjusted inside a specific list environment using `\setlength{...}{...}`. Careful tuning allows for compact layouts or more open spacing, depending on the document style requirements.

```
\begin{itemize}
    \setlength{\itemsep}{0pt}
    \setlength{\topsep}{0pt}
    \setlength{\parsep}{0pt}
    \setlength{\partopsep}{0pt}

    \item First
    \item Second
\end{itemize}
```

7.5.1 Custom itemize

```
    \usepackage{enumitem}
```

The `enumitem` package provides enhanced control over LaTeX lists (‘itemize‘, ‘enumerate‘, ‘description‘) compared to the default behavior. Use it to apply OPTIONS directly in the Environment.

- `label=$\rightarrow$`: to replace bullets with symbols, letters, or text.
- `label=Step 1`: to create custom ordering text.
- `label=`: to eliminate labels.
- `label=`: to eliminate labels.
- `itemsep=` : to apply all length parameters directly as OPTIONS.

Example 7.5.1. Snippet for the resistors itemize.

```
\begin{itemize}[
    itemsep=3pt, leftmargin=2.5cm, label={%
        \tikz[baseline=-0.6ex]{\draw[->>] (0,0) to [R] ++(2cm,0);}
    }
]

    \item \texttt{label=\$\textbackslash rightarrow\$}: to replace bullets with
        symbols, letters, or text.
    \item \texttt{label=Step \arabic*}: to create custom ordering text.
    \item \texttt{label={}}: to eliminate labels.
    \item \texttt{label={}}: to eliminate labels.
    \item \texttt{itemsep=\ }: to apply all length parameters directly as OPTIONS.
\end{itemize}
```

Chapter 8

Useful Stuff

8.1 Overleaf and coding shortcuts

My personal snippets for faster Overleaf coding.

`ALT + SHIFT + Arrow UP :`	`Duplicate line (or multiple) above`
`ALT + SHIFT + Arrow DOWN:`	`Duplicate line (or multiple) below`
`ALT + Arrow UP:`	`Move line (or multiple) above`
`ALT + Arrow DOWN:`	`Move line (or multiple) below`
`CTRL + B :`	`Make \textbf`
`CTRL + I :`	`Make \textit`
`CTRL + ù :`	`Comment/Uncomment line (or multiple)`
`WIN + SHIFT + S:`	`Crop and screenshot window`
`CTRL + V (after screen or img copy):`	`Fast upload and \begin{figure}`
`\bfi + ENTER:`	`Fast \begin{figure}`
`WIN + SPACE:`	`Switch keyboard language` `(hate when it happens)`

Acknowledgments

Sincere thanks to Chßrkistian and LuckyJack, who believed in this project from the start.

Happy drawing.