r-Portfolio

Print version of r-Portfolio.

r-Portfolio

Contents

Awards

2025

- 愛媛大学工学部工学科コンピュータ科学コース優秀学生 (3 年次)

Publications

国内会議

- 米山瑛人, 杉原壮一郎, 梶原智之, 池田直樹, 谷川千尋. 所見文書と X 線画像を用いた矯正歯科治療の自動診断に向けて. 第 20 回言語処理若手シンポジウム, September 2025.
- 戸田裕子, 前川大輔, 眞鍋光汰, 米山瑛人, 野々村奏, 藤原有希, 梶原智之. **HOTATE：本音と建前の応答対からなる対話コーパスの構築.** 言語処理学会第 32 回年次大会, pp.xxx-xxx, March 2026. (to appear)

国際会議

- Yuko Toda, Daisuke Maekawa, Kota Manabe, Eito Yoneyama, Kanade Nonomura, Yuki Fujiwara, Tomoyuki Kajiwara. **HOTATE: A Japanese Dialogue Corpus Annotated with Responses of Private Thoughts and Public Statements.** In Proceedings of the 15th International Conference on Language Resources and Evaluation (LREC 2026), pp.xxx-xxx, Mallorca, Spain, May 2026. (to appear)

Blog

ポートフォリオリニューアル

タイトルにもある通り, ポートフォリオをリニューアルしました. 以前までは Astro で実装していましたが, 今回は Typst の HTML exporter を活用した構成に移行しました.

> **Typst Documentation — HTML**
> Typst's experimental HTML export captures document structure as semantic, human-readable HTML and exposes the html module for raw and typed HTML elements.
> https://typst.app/docs/reference/html/

ポートフォリオ開設

タイトルにもある通り, ポートフォリオを開設しました. 開設にあたり, Astro Nano を使用しています.

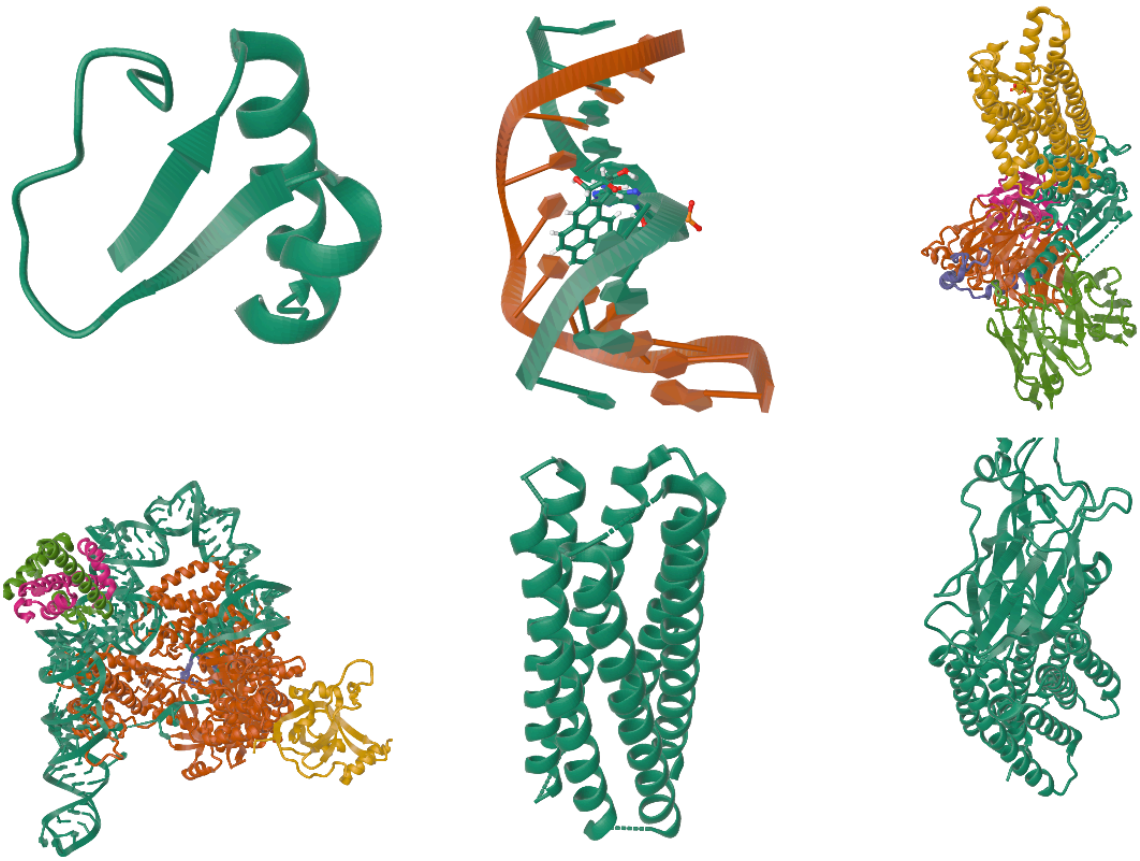
Astro Nano
Astro Nano is a static, minimalist, lightweight, lightning fast portfolio and blog.
https://github.com/markhorn-dev/astro-nano

Projects

molfig

Molfig is a Typst package for rendering molecular structure files in static documents.

It accepts PDB, mmCIF, and BinaryCIF input, converts structures through a CPU-side Mol*-style Model/Structure/Unit layer, exports static OBJ/STL/PLY mesh bytes, and delegates final document rendering to `maquette`.



Quickstart

```
#import "@preview/molfig:0.1.1"
#set page(width: auto, height: auto, margin: 0mm)

// Uses structural data from RCSB PDB / wwPDB.
// PDB ID: 9R1O
// PDB DOI: https://doi.org/10.2210/pdb9R1O/pdb
// Deposition authors: Petrenas, R.; Ozga, K.; Chubb, J.J.; Woolfson, D.N.
// PDB archive data files are available under CC0 1.0.
#let pdb = read("9R1O.pdb", encoding: none)

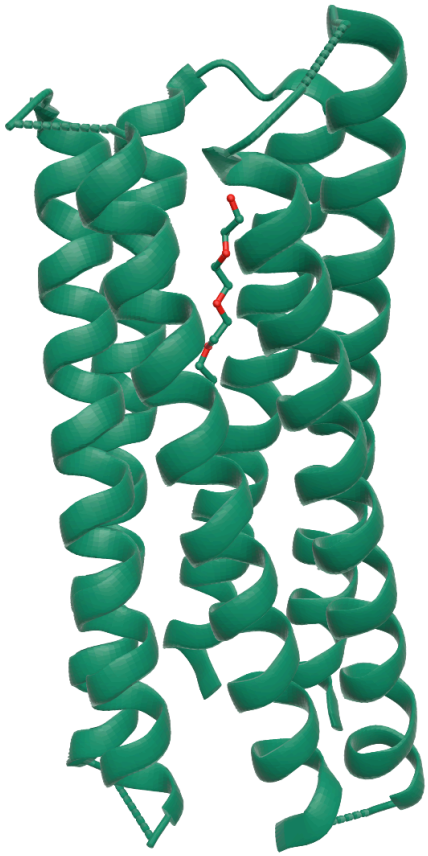
#molfig.render(
  pdb,
  format: "pdb",
  representation: "molstar",
  assembly: "1",
  mesh-format: "obj",
  quality: "high",
  center: true,
  output-format: "svg",
  config: (
    azimuth: 35,
```

```
    elevation: 24,
    background: "",
  ),
)
```

The manual uses PDB entry 9R1O as its complete example. Put `9R10.typ` and `9R10.pdb` in the same directory, then compile the figure PDF:

```
typst compile 9R10.typ
```

Rendered 9R1O Example



Structural data source: RCSB PDB / wwPDB, PDB ID `9R10`, DOI `10.2210/pdb9R10/pdb`. PDB archive data files are distributed under CC0 1.0.

Use `format: "mmcif"` or `format: "bcif"` for text mmCIF and BinaryCIF inputs. For reproducible documents, prefer explicit `format`, `representation`, `assembly`, `alt-loc`, `mesh-format`, and geometry quality options instead of relying on auto-detection.

Examples

The `package/examples` directory contains complete example sources, rendered PDFs, and their accompanying structural data files. The example data files are kept under `package/examples/data`, together with attribution metadata.

Features

- Inputs: PDB, text CIF/mmCIF, and BinaryCIF.
- Structure layer: Mol*-style Model/Structure/Unit concepts, assembly operators, altLoc handling, bond metadata, lookup3d/boundary summaries, secondary structure, coarse IHM spheres/gaussians, and semantic render-object metadata.
- Representations: Mol* default, spacefill, ball-and-stick, cartoon, ribbon, and backbone.
- Assembly support: biological assemblies are represented as source model plus unit operators before static mesh export.
- Alternate locations: select a concrete altLoc, all altLocs, or the highest-occupancy conformer.
- Color themes: `color-theme: "chain-id"` assigns Mol* Chain ID colors and forwards OBJ materials to maquette.
- Outputs: OBJ, companion MTL, binary STL, and ASCII PLY.
- Rendering: `render` passes generated mesh bytes to maquette; `render-object` exposes the mesh, rendered content, and normalized metadata for advanced documents.

Public API

- `render(data, ..., config: (:), width: auto, height: auto)` converts and renders through maquette.
- `render-object(data, ...)` returns generated mesh bytes, rendered content, and metadata.
- `to-obj(data, ...)`, `to-mtl(data, ...)`, `to-stl(data, ...)`, and `to-ply(data, ...)` return export bytes.
- `info(data, ...)` returns molecular and mesh-planning metadata without rendering.
- `mesh-info(data, mesh-format: "obj", config: (:), ...)` delegates to maquette's mesh metadata helpers for the generated mesh.
- `v15-or-later()` returns whether the active Typst compiler supports project-side `path(...)` values.

Common options include `format`, `representation`, `color-theme`, `assembly`, `alt-loc`, `block-index`, `block-header`, `quality`, `sphere-detail`, `linear-segments`, `radial-segments`, `radius-scale`, `atom-radius`, `bond-radius`, `ribbon-radius`, `ribbon-width`, `helix-profile`, `round-cap`, `sheet-arrow-factor`, `tubular-helices`, `infer-bonds`, and `center`.

The `data` argument accepts bytes from `read(..., encoding: none)`, inline string data for small examples, and Typst 0.15+ path values created with `path("...")`.

Choosing A Mesh Format

- Use OBJ for the closest static Mol* exporter parity and readable diffs.
- Use STL when a downstream tool specifically requires binary triangle data.
- Use PLY when package-owned face group metadata is useful in a compact text mesh.

OBJ output can be paired with `to-mtl`. During `render`, OBJ material colors are automatically converted to maquette's `materials` map; entries supplied through `config.materials` override generated colors. OBJ and PLY preserve Molfig group or operator metadata where the format can represent it. Binary STL follows Mol* static exporter behavior and keeps the two-byte facet attribute field at zero.

Documentation

The full Molfig manual is available at `package/docs/documentation.pdf`. It documents:

- installation and import conventions;
- input format handling and BinaryCIF block selection;
- every public command and return shape;
- mesh, representation, assembly, altLoc, and quality options;
- maquette passthrough configuration;
- metadata fields returned by `info` and `render-object`;
- licensing, third-party notices, and example data attribution;

- troubleshooting and development commands;
- the embedded 9R1O rendering and its data source.

The manual source is `package/docs/documentation.typ`, and it reads the package version from `package/typst.toml`.

Notes And Limits

Molfig emits static presentation meshes. It does not implement Mol*'s interactive WebGL surface or volume rendering pipeline. Molecular surface, gaussian surface, gaussian volume, and density/volume visuals are outside the current static export contract.

IHM coarse gaussian rows remain available as coarse model units, but they are not converted into gaussian surface or volume visuals.

License And Notices

Molfig project code is licensed under the MIT License. See `LICENSE`.

Molfig ports or adapts Mol* behavior and includes Mol*-derived reference data in the Rust/WASM implementation. Mol* is licensed under the MIT License, copyright (c) 2017 - now, Mol* contributors.

Bundled example structure files under `package/examples/data` are PDB archive data from RCSB PDB / wwPDB and are available under CC0 1.0. Per-file PDB IDs, DOIs, and recommended attributions are listed in `package/examples/data/README.md`.

See `NOTICE.md` and `THIRD_PARTY_NOTICES.md` for the full distribution notice.

Development

```
cd wasm-plugin
cargo fmt --check
cargo test
cargo build --release --target wasm32-unknown-unknown
cp target/wasm32-unknown-unknown/release/molfig.wasm ../package/molfig.wasm
cd ..
typst compile --root package package/docs/documentation.typ package/docs/documentation.pdf
```

The checked-in `package/molfig.wasm` should be regenerated after Rust changes that affect the Typst plugin. Regenerate `package/docs/documentation.pdf` after public API or documentation changes.

svg2tex-rs

`svg2tex-rs` provides the `svg2tex` command, which converts SVG artwork into PDF literal operators or complete LaTeX source. It is designed for TeX workflows where SVG graphics should stay as vector content whenever possible, while still accepting a broad range of real-world SVG files through hybrid rendering.

Installation

Install from crates.io:

```
cargo install svg2tex-rs
```

Clone and build locally without installing:

```
git clone https://github.com/rice8y/svg2tex-rs.git
cd svg2tex-rs
just build
```

This writes the binary to `target/release/svg2tex`.

Clone and install to `~/.local/bin`:

```
git clone https://github.com/rice8y/svg2tex-rs.git
cd svg2tex-rs
just install
```

The default install target is `~/.local/bin/svg2tex`. Override it with `LOCAL_BIN_DIR=/your/path just install`.

Quick Start

Emit raw PDF literal operators:

```
svg2tex --input drawing.svg --output drawing.literal
```

Emit a standalone LaTeX document:

```
svg2tex --input drawing.svg --tex --output drawing.tex
lualatex drawing.tex
```

Emit an embeddable TeX snippet:

```
svg2tex --input drawing.svg --tex --tex-format snippet --output drawing-snippet.tex
```

Output Modes

- Default mode: raw PDF literal operators only
- `--tex`: complete TeX-oriented output
- `--tex-format standalone`: complete cropped document, best default for direct compilation
- `--tex-format article`: complete article document
- `--tex-format snippet`: macro-only TeX fragment for inclusion in another document

Rendering Model

`svg2tex` tries to preserve SVG content as vector output whenever practical. When a subtree requires raster-domain processing, it uses hybrid rendering instead of silently dropping the feature.

Use these flags to control behavior:

- `--strict`: fail instead of using hybrid rendering
- `--fallback-dpi <N>`: control raster fallback resolution
- `--embed-images`: embed raster images referenced by the SVG

Text and Fonts

Text is flattened into paths during conversion. For reproducible results across machines, prefer explicit font configuration.

Useful flags:

- `--no-system-fonts`
- `--strict-fonts`
- `--report-fonts`
- `--font-file <PATH>`
- `--font-dir <PATH>`
- `--font-family <NAME>`
- `--serif-family <NAME>`
- `--sans-serif-family <NAME>`
- `--monospace-family <NAME>`

Recommended reproducible setup:

```
svg2tex \
  --input drawing.svg \
  --tex \
  --no-system-fonts \
  --strict-fonts \
  --font-file ./fonts/YourFont-Regular.ttf \
  --font-family "Your Font"
```

TeX Engine Selection

Choose the target backend explicitly when needed:

```
svg2tex --input drawing.svg --tex --engine luatex --output drawing.tex
```

Supported engine values:

- `auto`
- `pdftex`
- `luatex`
- `xetex`
- `ptex`
- `uptex`

Limitations

- Text output depends on the fonts you load into the converter
- Some SVG effects are represented through hybrid raster subtrees rather than pure vector PDF
- Exact pixel-perfect matching with every SVG renderer is not guaranteed, especially for complex text and filter interactions

For a fuller user guide, see `docs/documentation.pdf`.

License

typarium

typarium is a Typst package for rendering font specimen cards for system fonts and local fonts.

Fonts specified only by family name are renderable through Typst, but file-level metadata is only available when you provide a local font as raw bytes, pass a Typst 0.15.0+ `path(...)`, or supply metadata manually.

For the full guide and API reference, see documentation.pdf.

Usage

Basic Local Font Specimen

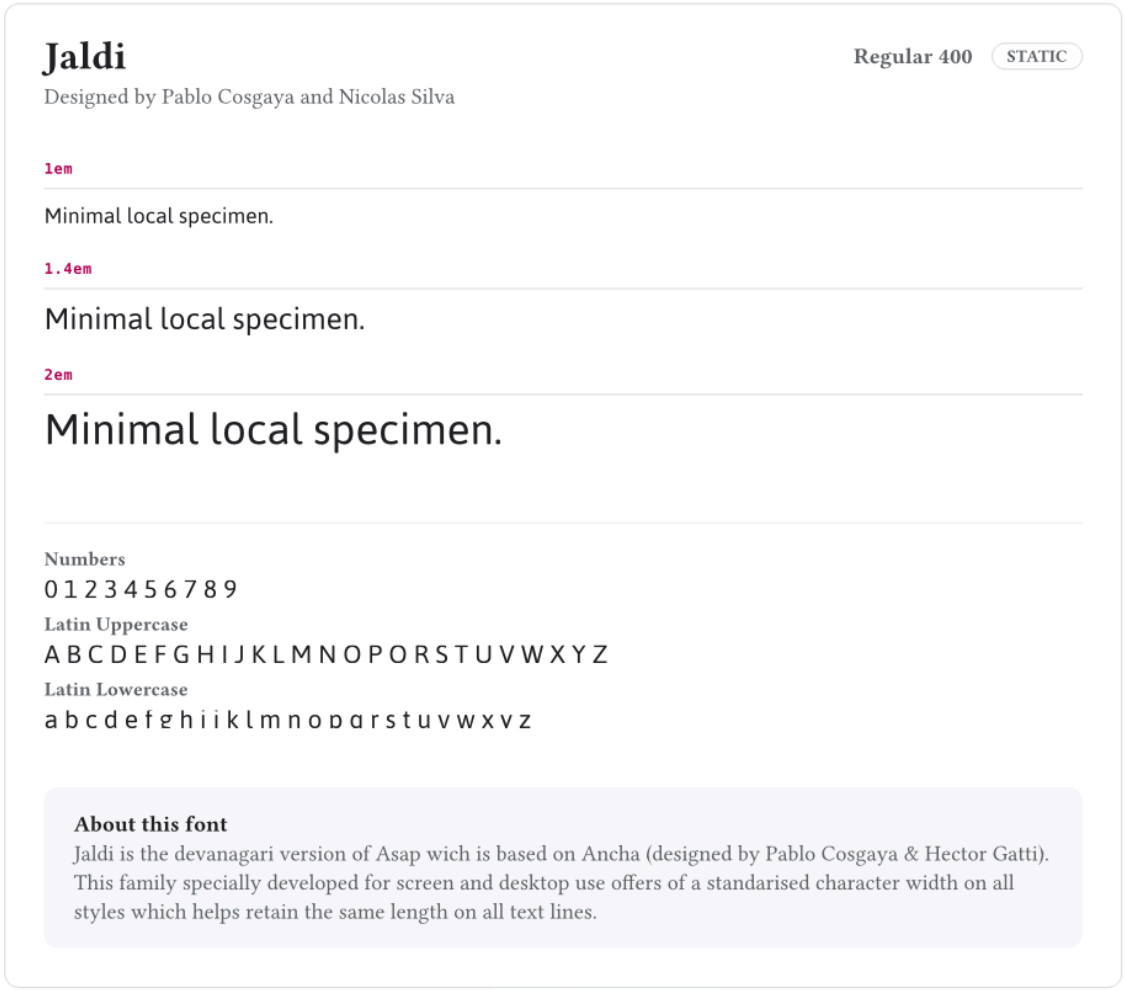
To render a local font file with the default renderer, place the font next to your document and pass it as raw bytes. On Typst 0.15.0 or later, you can pass a resolved `path(...)` value instead and let typarium read it inside the package.

```
#import "@preview/typarium:0.1.1": font-showcase

#font-showcase(
  theme: (
    sample-text: "Minimal local specimen.",
    show-glyphs: true,
    waterfall: (1.0em, 1.4em, 2.0em),
  ),
  fonts: ((path: read("Jaldi-Regular.ttf", encoding: none)),),
)
```

Typst 0.15.0+ path-based equivalent:

```
#font-showcase(
  fonts: ((path: path("Jaldi-Regular.ttf")),),
)
```



Anatomy Renderer

To inspect metrics and technical metadata with the bundled anatomy renderer:

```
#import "@preview/typarium:0.1.1": font-showcase, anatomy-theme, anatomy-render

#font-showcase(
  fonts: (
    (path: read("Jaldi-Regular.ttf", encoding: none)),
    (
      path: read("PlaywriteDKUloopetGuides-Regular.ttf", encoding: none),
      theme: (visual-sample-inset-bottom: 5em),
    ),
  ),
  theme: anatomy-theme + (
    sample-text: "An anatomy-oriented specimen card.",
  ),
  render: anatomy-render,
)
```

Jaldi-Regular

VISUAL RATIO (Unit/Em: 2048)

ASCENT	2350u
CAP-HEIGHT	1243u
X-HEIGHT	930u
DESCENT	-1111u

Technical Specs

Format	Static
Total Glyphs	1222 units
Design UPM	2048
Weight Class	400
Width Class	5
Italic Angle	0°

Legal & Source

License URL ↗

DESIGNER'S SPECIMEN

Whereas recognition of the inherent dignity

Playwrite DK Uloopet Guides

PlaywriteDKUloopetGuides-Regular

Version 1.003

VISUAL RATIO (Unit/Em: 1000)

ASCENT	1275u
CAP-HEIGHT	875u
X-HEIGHT	500u
DESCENT	-375u

Technical Specs

Format	Static
Total Glyphs	1250 units
Design UPM	1000
Weight Class	400
Width Class	5
Italic Angle	0°

Legal & Source

License URL ↗

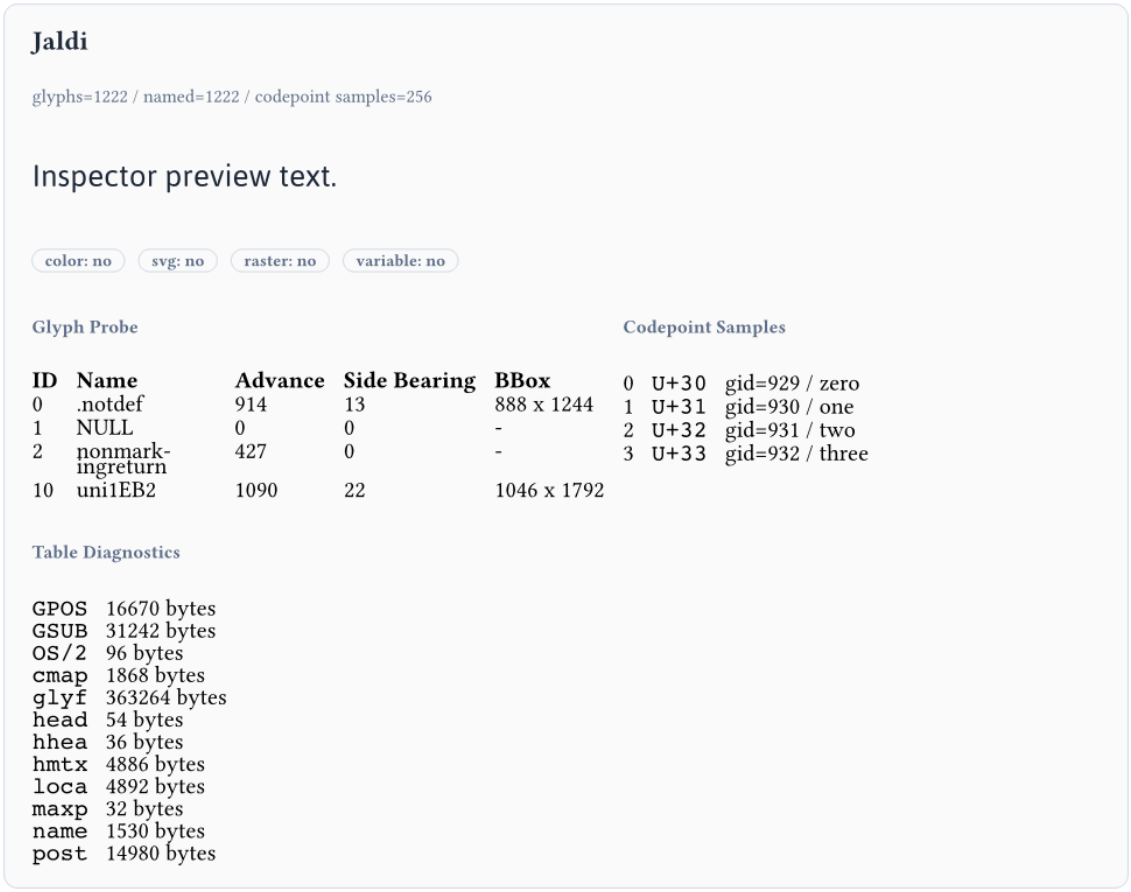
DESIGNER'S SPECIMEN

Inspector Renderer

To inspect parsed glyph metrics, codepoint samples, and table diagnostics with the bundled inspector renderer:

```
#import "@preview/typarium:0.1.1": font-showcase, inspector-theme, inspector-render

#font-showcase(
  theme: inspector-theme,
  render: inspector-render,
  fonts: ((path: read("Jaldi-Regular.ttf", encoding: none)),),
)
```



See also variation-request.typ for a standalone custom renderer that reads `variation-request` and `variation-support` directly.

Custom Theme and Renderer

The recommended pattern is `item -> theme -> metadata -> fallback`.

```
#import "@preview/typarium:0.1.1": font-showcase

#let poster-theme = (
  color-bg: rgb("0f1720"),
  color-primary: rgb("f8fafc"),
  color-muted: rgb("94a3b8"),
  stroke-card: 0.6pt + rgb("334155"),
  card-inset: 1.2em,
  size-title: 1.3em,
  size-meta: 0.78em,
  size-sample: 2.1em,
```

```
  gap-meta: 0.4em,
  gap-sample: 0.9em,
  sample-text: "Theme drives shared renderer options.",
)

#let poster-render = it => {
  let ft = it.font-text
  let opt = (key, default) => {
    let overrides = it.at("item-overrides", default: (:))
    if type(overrides) == dictionary and key in overrides {
      overrides.at(key)
    } else {
      it.theme.at(key, default: it.at(key, default: default))
    }
  }

  block(
    breakable: false,
    width: 100%,
    fill: opt("color-bg", black),
    stroke: opt("stroke-card", none),
    inset: opt("card-inset", 1em),
    radius: 0.6em,
    [
      #text(size: opt("size-title", 1.2em), weight: "bold", fill: opt("color-primary", white))
[#it.name]
      #v(opt("gap-meta", 0.4em))
      #text(size: opt("size-meta", 0.8em), fill: opt("color-muted", white))[
        family=#it.render-name / weight=#it.weight / style=#it.style
      ]
      #v(opt("gap-sample", 0.8em))
      #ft(size: opt("size-sample", 2em), fill: opt("color-primary", white))[
        #opt("sample-text", "Fallback specimen")
      ]
    ],
  )
}

#font-showcase(
  theme: poster-theme,
  render: poster-render,
  fonts: ((path: read("Jaldi-Regular.ttf", encoding: none)),),
)
```

Per-font dictionaries can still override any renderer-facing keys, including `sample-text` and per-card `theme`.

Mixed Input Array

You can combine system fonts, local files, and manual dictionaries in one call:

```
#import "@preview/typarium:0.1.1": font-showcase

#font-showcase(
```

```
  columns: 2,
  theme: (
    sample-text: "Shared default sample text.",
  ),
  fonts: (
    "Libertinus Serif",
    (path: read("Jaldi-Regular.ttf", encoding: none), sample-text: "Parsed from file"),
    (name: "Courier", display-name: "Courier (Manual Label)"),
  ),
)
```

Top-Level Theme Override

To restyle all cards in one showcase:

```
#import "@preview/typarium:0.1.1": font-showcase, default-theme

#font-showcase(
  theme: default-theme + (
    paragraph-text: lorem(20),
    waterfall: (1.4em, 2.4em),
    card-inset: 2em,
    color-primary: rgb("202124"),
  ),
  fonts: ((path: read("Jaldi-Regular.ttf", encoding: none)),),
)
```

Per-Font Theme Override

To restyle only one card inside a multi-font showcase:

```
#import "@preview/typarium:0.1.1": font-showcase, anatomy-theme, anatomy-render

#font-showcase(
  theme: anatomy-theme + (
    sample-text: "Shared anatomy specimen text.",
  ),
  render: anatomy-render,
  fonts: (
    (path: read("Jaldi-Regular.ttf", encoding: none)),
    (
      path: read("PlaywriteDKUloopetGuides-Regular.ttf", encoding: none),
      theme: (visual-sample-inset-bottom: 5em),
    ),
  ),
)
```

Jaldi-Regular

VISUAL RATIO (Unit/Em: 2048)

ASCENT	2350u
CAP-HEIGHT	1243u
X-HEIGHT	930u
DESCENT	-1111u

Technical Specs

Format	Static
Total Glyphs	1222 units
Design UPM	2048
Weight Class	400
Width Class	5
Italic Angle	0°

Legal & Source

License URL ↗

DESIGNER'S SPECIMEN

Shared anatomy specimen text.

Playwrite DK Uloopet Guides

PlaywriteDKUloopetGuides-Regular

Version 1.003

VISUAL RATIO (Unit/Em: 1000)

ASCENT	1275u
CAP-HEIGHT	875u
X-HEIGHT	500u
DESCENT	-375u

Technical Specs

Format	Static
Total Glyphs	1250 units
Design UPM	1000
Weight Class	400
Width Class	5
Italic Angle	0°

Legal & Source

License URL ↗

DESIGNER'S SPECIMEN

API Reference

`font-showcase`

Renders one or more font specimen cards.

```
#font-showcase(
  fonts: auto,
  theme: (:),
  render: auto,
  columns: 1,
)
```

Key Parameters:

- `fonts` (`auto | str | bytes | path | dictionary | array`): Font sources to render. Use raw bytes for Typst 0.14.x compatibility, or `path(...)` on Typst 0.15.0+ for user-local font files.
- `theme` (`dictionary`): Showcase-level renderer configuration and design-token payload passed into the active renderer layer.
- `render` (`function | auto`): Custom card renderer. If `auto`, uses `default-render`.
- `columns` (`int`): Number of cards per grid row.
- Renderer-facing keys such as `sample-text`, `sample-size`, `waterfall`, `show-glyphs`, `show-details`, `leading`, `features`, `lang`, `script`, `title-overflow`, and `title-overflow-badge-position` are supplied through `theme` or per-font dictionaries.

`set-variations`

Normalizes a variable-axis request onto a font item or theme dictionary.

```
#set-variations(target, values)
```

- If `target` is a font-item dictionary, the helper returns that dictionary with `variation-values` inserted.
- If `target` is a font-name string, it returns a dictionary using `name:`. If `target` is raw bytes or a Typst 0.15.0+ path value, it returns a dictionary using `path:`.
- `font-showcase` converts `variation-values` into `variation-request`, which renderers can inspect today even though Typst 0.14.x does not yet apply variable axes in shaping.
- The warning about variable fonts comes from Typst itself, not from typarium. Installing a static instance of the same family can remove the warning, but it does not enable true variable-axis shaping in Typst.

`default-theme`, `terminal-theme`, `anatomy-theme`, `inspector-theme`

Bundled theme dictionaries used by the packaged renderers.

- `default-theme` exposes a broad design-token surface for editorial specimen cards.
- `terminal-theme` provides a compact terminal-style override.
- `anatomy-theme` exposes technical layout tokens for metric-heavy anatomy cards.
- `inspector-theme` exposes inspection-focused tokens such as `probe-glyph-ids` and `sample-codepoint-count`.

`default-render`, `terminal-render`, `anatomy-render`, `inspector-render`

Bundled renderer functions you can pass to `render`.

- `default-render`: editorial specimen layout with badges, waterfall support, descriptions, glyphs, and metadata rows.
- `terminal-render`: command-line styled specimen card.
- `anatomy-render`: metric and metadata focused layout for technical inspection.
- `inspector-render`: glyph probe, codepoint sample, variation request, and table diagnostic inspector.

Input Shapes

`fonts` supports five common forms:

- `auto`: uses the current Typst font context.
- `"Libertinus Serif"`: renders a system font by family name.
- `read("Jaldi-Regular.ttf", encoding: none)`: parses and renders raw font bytes directly.

- `path("Jaldi-Regular.ttf")` on Typst 0.15.0+: resolves a project-local font at the call site and lets typarium parse it.
- `(path: read("Jaldi-Regular.ttf", encoding: none), name: "Custom Label")` or `(path: path("Jaldi-Regular.ttf"), name: "Custom Label")`: parses a file and applies explicit overrides.

Under the Hood

The processing workflow is:

1. `font-showcase` normalizes the input shape and resolves the showcase-level theme.
2. Raw font bytes, including bytes read internally from Typst 0.15.0+ path sources, are passed to `font_parser.wasm`.
3. The Rust parser extracts names, permissions, collection summaries, metrics, variation axes, table diagnostics, codepoint samples, glyph-name indices, and per-glyph metric/image records.
4. Metadata keys are normalized into Typst-friendly hyphenated names.
5. A prepared `font-text` helper is created from the resolved font family and shaping options.
6. Variable-axis requests are normalized into `variation-request` metadata for renderers.
7. The bundled or custom renderer receives the final render payload as `it`.

In practice, if you are designing your own renderer, `theme` and per-font dictionaries are the preferred place for specimen-specific settings.

Custom Renderer Tips

- Put shared renderer options in `theme`.
- Put card-specific exceptions in the font item dictionary.
- Resolve options in the order `item -> theme -> metadata -> literal fallback`.
- Use `set-variations(...)` when you want a host-level variable request contract that custom renderers can read consistently.
- Always render specimen text with `it.font-text` so resolved `features`, `lang`, `script`, and fallback behavior stay intact.
- Set `breakable: false` on the outermost card block when a whole card should stay on one page.

Notes

Local Font Files

For Typst 0.14.x compatibility, pass local fonts as raw bytes via `read("FontFile.ttf", encoding: none)` into the `path` field. On Typst 0.15.0 or later, pass `path("FontFile.ttf")` directly or through the dictionary `path` field; callers do not need to wrap that path value in `read(...)`. Plain string file paths still resolve from the package context, so project-local fonts should use either `read(...)` or `path(...)`.

Family-Name Fonts

If you specify a font only by family name, such as `(name: "Jaldi")`, typarium can render it through Typst's text engine but cannot inspect the underlying font file. In that mode, parsed metadata such as `glyphs`, `glyph-details`, `number-of-glyphs`, name-table records, permissions, and variation axes are not available unless you provide them manually.

Design Model

`default`, `terminal`, `anatomy`, and `inspector` are sample implementations. The real core of the package is the more primitive `font-showcase` host, which resolves font metadata, prepares `font-text`, merges showcase-level and per-font theme overrides, and lets users define their own renderer behavior through `render`.

License

glotter

glotter is a Typst package for detecting the language of text fragments and applying language-aware settings to content.

The Typst plugin is compiled to WebAssembly using `fasttext-pure-rs`, a pure Rust implementation of fastText.

Usage

```
#import "@preview/glotter:0.1.0": *

#let samples = (
  ja: "これは日本語の文章です。",
  en: "This is an English sentence.",
)

#let print-lang(text) = [
  #text $arrow$ #lang(text) \
]

#print-lang(samples.ja)
#print-lang(samples.en)
```

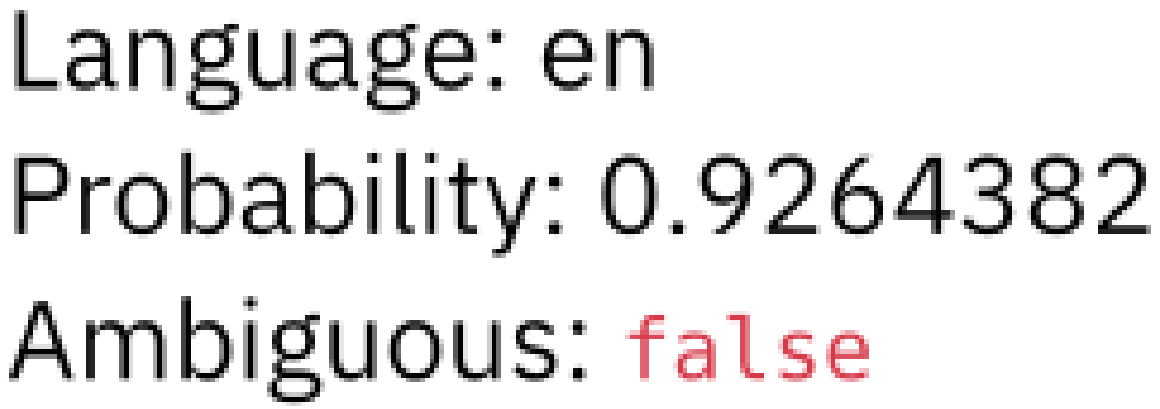
これは日本語の文章です。 → ja
This is an English sentence. → en

To retrieve detailed prediction metadata, use `detect-info`:

```
#let info = detect-info(
  samples.en,
  k: 3,
  min-margin: 0.12,
)

#let print-info(info) = [
  Language: #info.at("lang") \
  Probability: #info.at("probability") \
  Ambiguous: #info.at("ambiguous")
]

#print-info(info)
```



To apply the detected language automatically to content:

Typst can apply language-specific text settings when the language is known, but the language must normally be specified manually. `auto-text` detects the language of a fragment and applies the corresponding `text(lang: ...)` setting automatically.

```
#let sample = "שלום עולם"

*Plain text:*

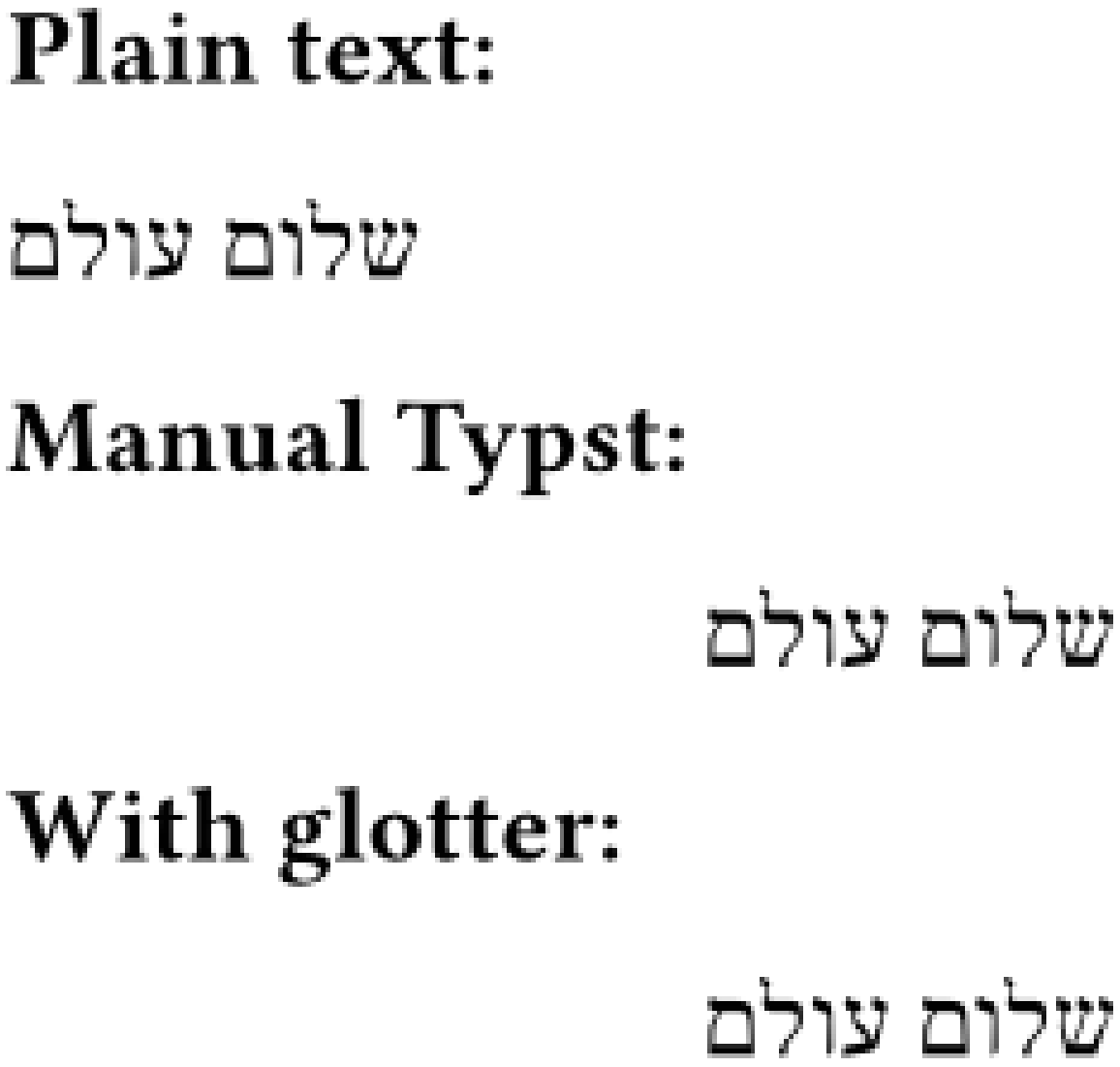
#sample

*Manual Typst:*

#text(lang: "he")[#sample]

*With glotter:*

#auto-text[#sample]
```



For document-wide use, `auto-par` can be installed as a paragraph show rule. Each paragraph is detected independently, so mixed-language documents can be written without wrapping every paragraph in `text(lang: ...)`.

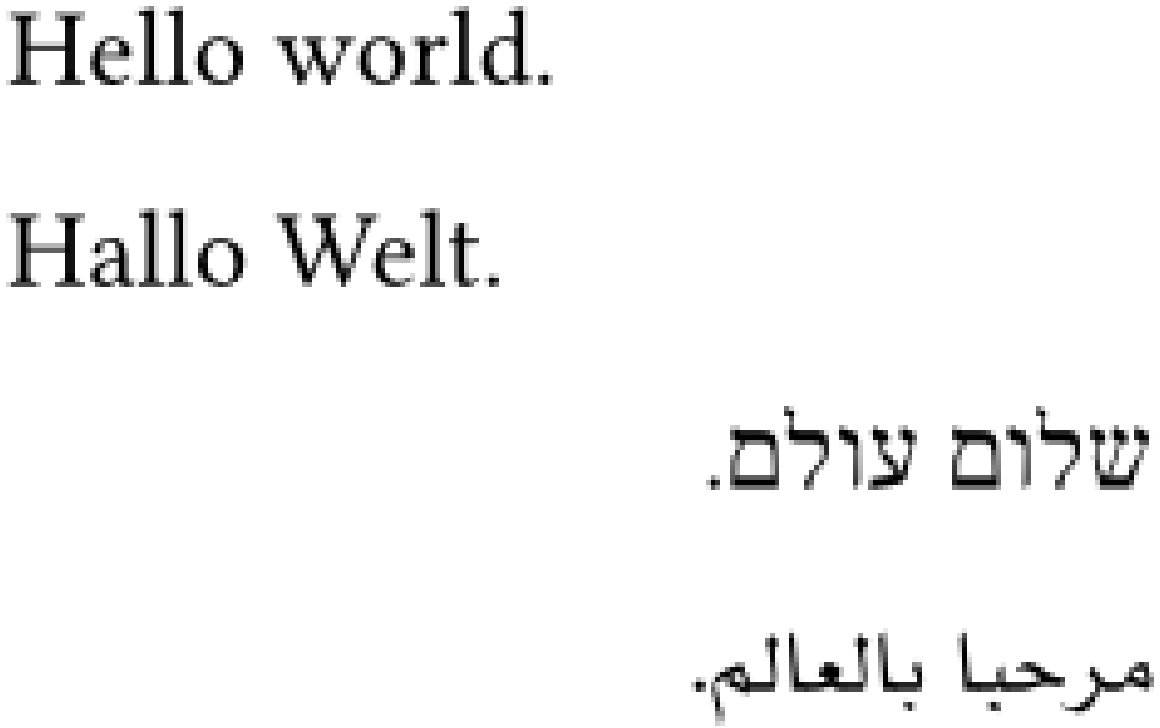
```
#show par: it => auto-par(it, fallback: "en")

Hello world.

Hallo Welt.

שלום עולם.

مرحبا بالعالم.
```



Use Cases

`glotter` can be used directly, but it is also designed as a building block for downstream Typst packages, templates, and document-processing workflows.

Typst already supports language-aware text through `text(lang: ...)`. `glotter` provides the missing step: detecting which language should be applied to a given text fragment.

Multilingual templates

Templates can use `glotter` to accept multilingual content without requiring users to annotate every fragment manually.

```
#let localized-abstract(body) = {
  let info = detect-info(body, fallback: "en")
  let l = info.at("lang")
  let rtl-lang = ("ar", "fa", "he", "ur").contains(l)

  let title = if l == "ja" {
    "概要"
  } else if l == "ar" {
    "ملخص"
  } else if l == "de" {
    "Zusammenfassung"
  } else if l == "fr" {
    "Résumé"
  } else {
    "Abstract"
  }

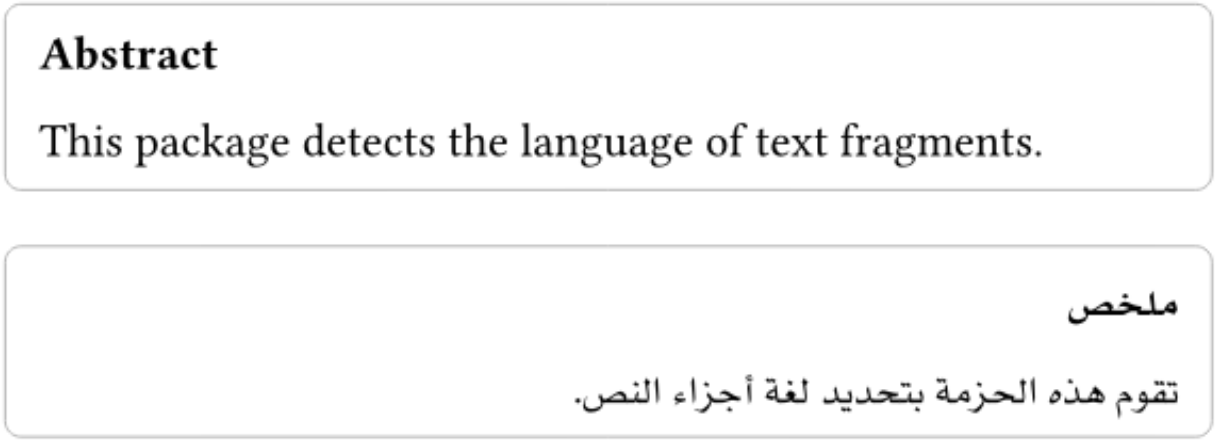
  block(
    width: 10cm,
    inset: 8pt,
    stroke: 0.5pt + luma(180),
    radius: 4pt,
  )[
```

```
    #set text(lang: l, dir: if rtl-lang { rtl } else { ltr })
    #set align(if rtl-lang { right } else { left })

    #strong[#title] \
    #auto-text(fallback: "en")[#body]
  ]
}

#localized-abstract[This package detects the language of text fragments.]

#localized-abstract[تقوم هذه الحزمة بتحديد لغة أجزاء النص.]
```



Language-dependent rendering

Downstream packages can use `detect-info` to branch on the detected language.

```
#import "@preview/note-me:0.6.0": *

#let localized-note(body) = {
  let info = detect-info(body, fallback: "en")
  let l = info.at("lang")

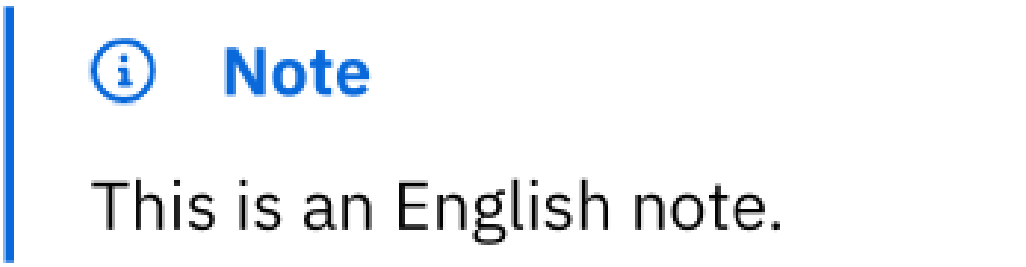
  let title = if l == "ja" {
    "注意"
  } else if l == "de" {
    "Hinweis"
  } else if l == "fr" {
    "Remarque"
  } else {
    "Note"
  }

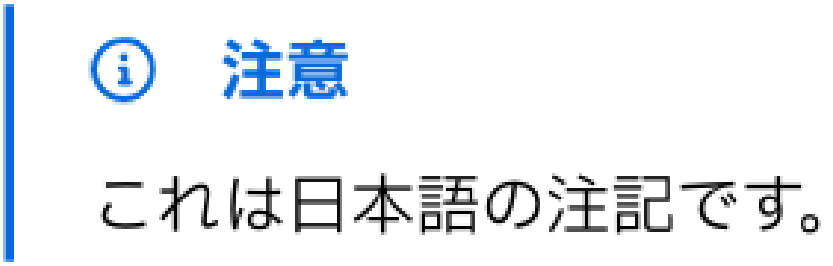
  note(title: title)[
    #auto-text(fallback: "en")[#body]
  ]
}

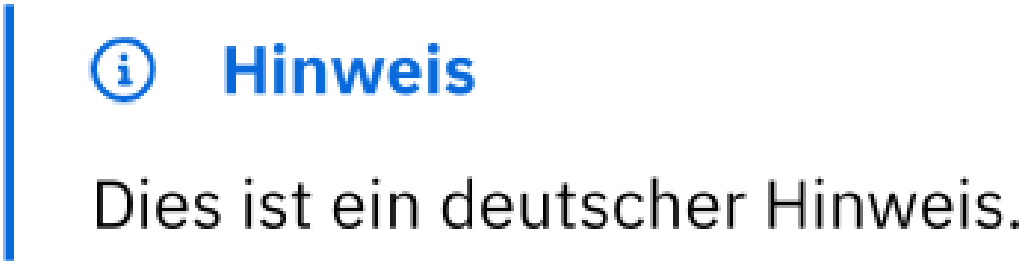
#localized-note[This is an English note.]

#localized-note[これは日本語の注記です。]

#localized-note[Dies ist ein deutscher Hinweis.]
```







API

- `detect(input, k: 1, threshold: 0.0)`: returns raw predictions as dictionaries with `lang`, `label`, and `probability`.
- `detect-info(input, k: 3, threshold: 0.0, min-margin: 0.12, fallback: "und")`: returns a dictionary containing the final language, top prediction, all predictions, probability margin, ambiguity flag, fallback value, and normalized input.
- `lang(input, ...)`: returns a language code.
- `is-lang(input, expected, ...)`: checks whether the detected language matches a given string or any string in an array.
- `is-cjk(input, ...)`, `is-rtl(input, ...)`, `is-latin(input, ...)`: convenience script-family checks.
- `auto-text(body, fallback: "en", debug: false, ...)`: detects the language of `body` and wraps it in `text(lang: detected-language)`; any additional named arguments are forwarded to `text(...)`.
- `auto-par(it, fallback: "en")`: detects a paragraph and applies the detected language; intended for use with `#show par`.
- `debug-info(info)`: renders a compact diagnostic label for a `detect-info` result.
- `supported-languages`: array of the 176 language codes supported by the embedded fastText language identification model.
- `supported-language-count`: number of language codes supported by the embedded fastText language identification model.
- `is-supported-language(language)`: checks whether a language code or fastText label is supported.

The embedded model expects UTF-8 text.

fastText Model Provenance

`package/glotter.wasm` embeds data from the fastText language identification model `lid.176.ftz`, the compressed variant of fastText's 176-language identification model.

Official source: https://fasttext.cc/docs/en/language-identification.html

The official fastText documentation states that the language identification models were trained on data from Wikipedia, Tatoeba, and SETimes, and are distributed under CC-BY-SA-3.0.

License

`glotter` is distributed under `MIT AND CC-BY-SA-3.0`.

This package does not include the standalone fastText model file `lid.176.ftz`. However, this package includes `package/glotter.wasm`, which embeds model data from `lid.176.ftz`. For that reason, the package contains material under both:

- MIT License for the Typst package and plugin code.
- Creative Commons Attribution-ShareAlike 3.0 Unported for the embedded fastText model data.

The package code is MIT licensed. The embedded fastText model data in `glotter.wasm` is licensed under CC-BY-SA-3.0. See `NOTICE.md` for attribution, checksums, modification status, and redistribution notes. License texts are provided in `LICENSE-MIT` and `LICENSE-CC-BY-SA-3.0`.

typshade

Typshade is a Typst package for visualizing multiple sequence alignments in bioinformatics.

It provides a Typst-native interface centered on `shade(...)`, offering a readable and composable way to render alignments, add annotations, and incorporate logos, structure tracks, and graph tracks.

Inspired by TeXshade, Typshade rethinks alignment visualization with a focus on clarity, composability, and a Typst-native user experience.

Why Typshade?

TeXshade is powerful, but its UI reflects TeX: many global commands, implicit state, and order-sensitive setup before the alignment is rendered. Typshade keeps the feature set while making the source read like a figure specification.

TeXshade style:

```
\begin{texshade}{alignment.msf}
  \shadingmode[similar]{identical}
  \shadingcolors{blues}
  \residuesperline{45}
  \setends{1}{80..125}
  \showruler{top}{1}
  \rulersteps{10}
  \showconsensus{bottom}
  \showsequencelogo{top}
  \shaderegion{1}{NPA}{White}{BrickRed}
  \feature{top}{1}{NXX[ST]N}{box[Yellow]}{motif}
\end{texshade}
```

Typshade style:

```
#let alignment = read("alignment.msf", encoding: none)

#shade(
  alignment,
  format: "msf",
  figure: publication(
    similarity: "blues",
    region: "80..125",
    logo: "charge",
    motifs: (
      "NPA": (bg: "BrickRed", text: "active site"),
      "NXX[ST]N": "motif",
    ),
  ),
)
```

The lower-level helpers are still available through `commands:` when you need precise control, but new documents can usually start from the kind of figure you want: publication figure, motif map, structure map, or logo analysis.

Quick Start

```
#import "@preview/typshade:0.1.3": *

#let alignment = read("alignment.msf", encoding: none)
```

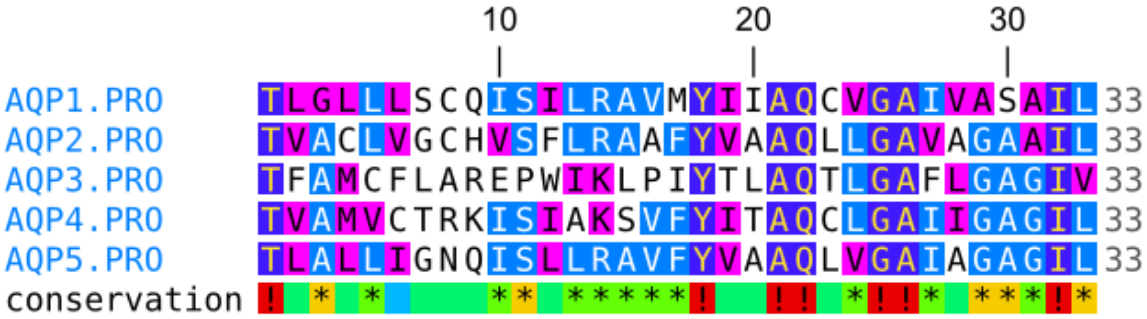
```
#shade(
  alignment,
  format: "msf",
  theme: "screen",
  figure: motif-map(auto),
)
```

`read(..., encoding: none)` remains supported on Typst 0.15 and later. On Typst 0.15 or later, you can additionally pass a resolved project path and let Typshade read the source inside the package:

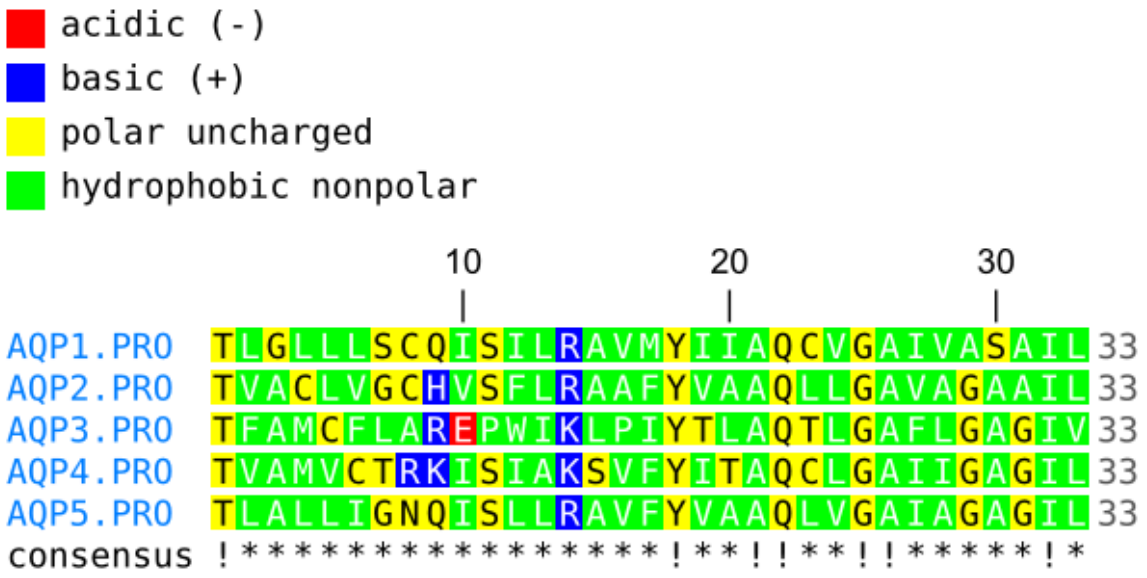
```
#shade(path("alignment.msf"), format: "msf", figure: motif-map(auto))
```

Preview

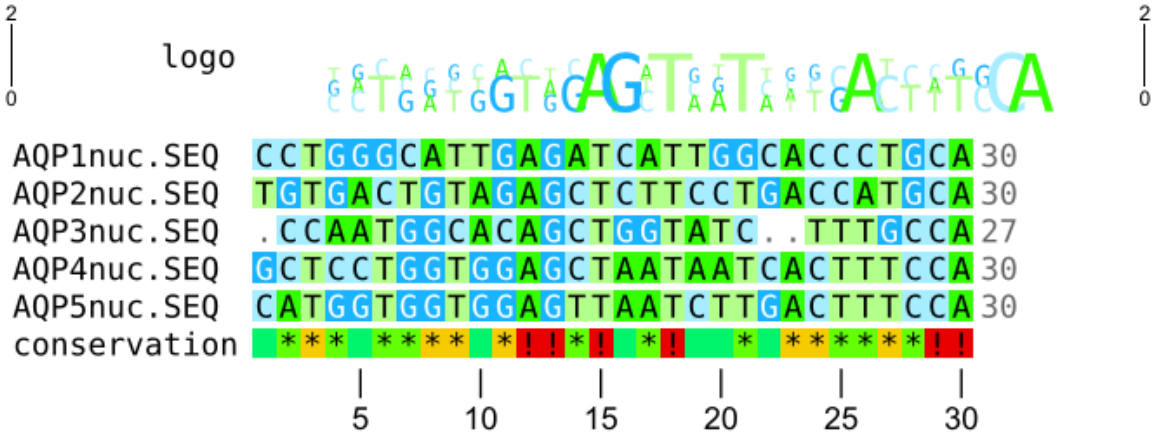
Protein alignment with similarity shading, motif annotations, a ruler, a conservation track, and a legend:



Protein alignment with hydropathy-based functional coloring:



Nucleotide alignment with DNA coloring, a sequence logo, a conservation track, and a ruler:



Typst-Native Helpers

- `shade(...)`: Named-option alignment renderer for new documents.
- `figure:`: Purpose-level recipe slot for complete figure designs.
- `publication`, `motif-map`, `structure-map`, `logo-analysis`, `overview`: High-level recipes.
- `similar`, `identical`, `diverse`, `functional`, `lines`, `window`, `ruler`, `consensus`, `logo`, `legend`: Compact command helpers; `lines(auto)` and `fit: "container"` fit the current Typst container.
- `auto-layout(...)`, `auto-page(...)`, and `fit: "page"`: Typst-aware line length and page-aware block splitting for long figures.
- `color-scheme`, `scoring-mode`, `sequence-window`: Small, readable option helpers.
- `ruler-track`, `consensus-track`, `sequence-logo`, `subfamily-logo`, `legend-track`: Track helpers.
- `structure-tracks(...)`: Adds topology/secondary-structure tracks from sidecar files.
- `shade-preset("publication" | "overview" | "logo" | "functional" | "structure")`: Reusable command bundles.
- `shade-theme("classic" | "print" | "screen" | "warm" | "nature")` and `visual-theme(...)`: Color/style bundles.
- `highlight`, `tint`, `emphasize`, `mark`, `motif`, `graph`: Readable command builders for common annotations; graph metrics include conservation, entropy, gap-fraction, coverage, identity, hydrophobicity, molecular weight, and charge.
- `select-range`, `select-motif`, `select-metric`, `select-and`, `select-or`, `select-not`, and `select-pad`: Composable Selection DSL values for windows, highlights, marks, graphs, and analysis helpers.
- `cell-style(ctx => ...)`: Data-driven per-cell styling with Typst functions.
- `pdb-point`, `pdb-line`, `pdb-plane`: Safer constructors for PDB selections.
- `alignment-position("left" | "center" | "right")`: Overrides the default left-aligned block placement.
- `alignment-summary(...)`, `alignment-debug(...)`, `cell-inspect(...)`, and `selection-preview(...)`: In-document inspection helpers.
- `sequence-list(...)` and `selection-table(...)`: Typst tables for data-aware reports.
- `percent-identity(...)`, `percent-similarity(...)`, and `similarity-table(...)`: Pairwise identity/similarity analysis.
- `alignment-data(...)` and `parse-alignment(...)`: Data access helpers for custom Typst logic.

TeXshade To Typshade

TeXshade idea	Typshade API
texshade environment	`shade(read("alignment.msf", encoding: none), format: "msf", figure: publication(...))`, or `shade(path("alignment.msf"), format: "msf", ...)` on Typst 0.15+

TeXshade idea	Typshade API
`shadingmode`, `shadingcolors`, `threshold`	`similar`, `identical`, `diverse`, `functional`, or `scoring-mode`, `color-scheme`, `threshold`
`residuesperline`, `setends`	`lines`, `window`, `fit`, `auto-layout`, `auto-page`, or Selection DSL values with `sequence-window`
`shownames`, `shownumbering`, `showconsensus`, `showruler`	`names`, `numbers`, `consensus`, `ruler`, or the fine-grained track helpers
`showsequencelogo`, `showsubfamilylogo`, `showlegend`	`logo`, `subfamily-logo`, `legend`
`shaderegion`, `tintregion`, `emphregion`, `feature`	`highlight`, `tint`, `emphasize`, `mark`, `motif`, `graph`
`includeDSSP`, `includeSTRIDE`, `includeHMMTOP`, `includePHD*`	`structure-tracks`, `dssp-track`, `stride-track`, `hmmtop-track`, `phd-topology-track`, `phd-secondary-track`
font and spacing macros	`text-family`, `text-weight`, `text-posture`, `text-size`, `block-gap`, `feature-slot-space`

See docs/documentation.typ for the full guide and a larger correspondence table.

Smart Recipes

Recipes inspect the alignment before rendering. For example, `motif-map(auto)` detects common motifs for the sequence type, focuses the region around them, chooses a readable line length, adds conservation when useful, and enables a logo only when the figure stays readable. Override any option when you need exact control.

Fine-Grained Control

Macro-style command names are intentionally not part of the public API. Use Typst-shaped command helpers when you need detailed control:

- Scoring: `threshold`, `weight-table`, `set-weight`, `gap-penalty`, `residue-style`, `functional-group`.
- Tracks: `names-track`, `numbering-track`, `consensus-name`, `consensus-symbols`, `ruler-name`, `ruler-marker`, `logo-color`.
- Sequence layout: `start-number`, `sequence-length`, `domain`, `hide-sequence`, `sequence-order`, `separation-line`.
- Features and structure: `feature-rule`, `feature-text-label`, `backtranslation-label`, `show-structure-types`, `structure-appearance`, `stride-track`, `dssp-track`, `hmmtop-track`, `phd-topology-track`, `phd-secondary-track`.
- Typography and spacing: `text-family`, `text-weight`, `text-posture`, `text-size`, `character-stretch`, `line-stretch`, `block-gap`, `feature-slot-space`.

Custom Control

Use recipes when you know the purpose of the figure:

```
#let alignment = read("alignment.msf", encoding: none)

#shade(
  alignment,
  format: "msf",
  figure: publication(
    region: "80..125",
    logo: "charge",
    motifs: (
      "NPA": (bg: "BrickRed", text: "active site"),
      "NXX[ST]N": "glycosylation",
    ),
  ),
)
```

Use `commands:` when you want to assemble visible parts yourself:

```
#let alignment = read("alignment.msf", encoding: none)

#shade(
  alignment,
  format: "msf",
  preset: "publication",
  theme: "screen",
  commands: (
    similar(colors: "blues", threshold: 45),
    lines(45),
    window(1, "80..125"),
    ruler("top", sequence: 1, every: 10),
    consensus("bottom", name: "conservation"),
    logo("top", colors: "charge"),
    legend(),
    highlight(1, "NPA", bg: "BrickRed"),
    motif(1, "NXX[ST]N", text: "motif"),
    graph("bottom", 1, "all", "conservation", kind: "color", options: ("ColdHot",)),
  ),
)
```

You can also mix recipe output with explicit, reproducible helper lists:

```
#let alignment = read("alignment.msf", encoding: none)

#shade(
  alignment,
  format: "msf",
  figure: publication(region: "80..125"),
  commands: (
    highlight(1, "NXX[DE][KR]XXQ", fg: "White", bg: "BrickRed"),
    sequence-logo(position: "top"),
  ),
)
```

License

molchemist

molchemist is a Typst package for rendering chemical structures from Molfile / SDF data and from SMILES strings.

It uses a Rust/WASM core to parse molecular graphs and generate `alchemist` ASTs, together with a companion WASM layout plugin for SMILES 2D coordinate generation. Molfile / SDF parsing is powered by `sdfrust`, SMILES parsing is based on `opensmiles`, SMILES 2D coordinate generation uses `CoordgenLibs`, and the final rendering is handled by the declarative drawing engine of `alchemist`.

Third-party license notices and bundled example-data provenance are collected in THIRD_PARTY_NOTICES.md.

Usage

Import `render-mol` for Molfile/SDF inputs, or `render-smiles` for SMILES inputs.

```
#import "@preview/molchemist:0.1.2": render-mol, render-smiles

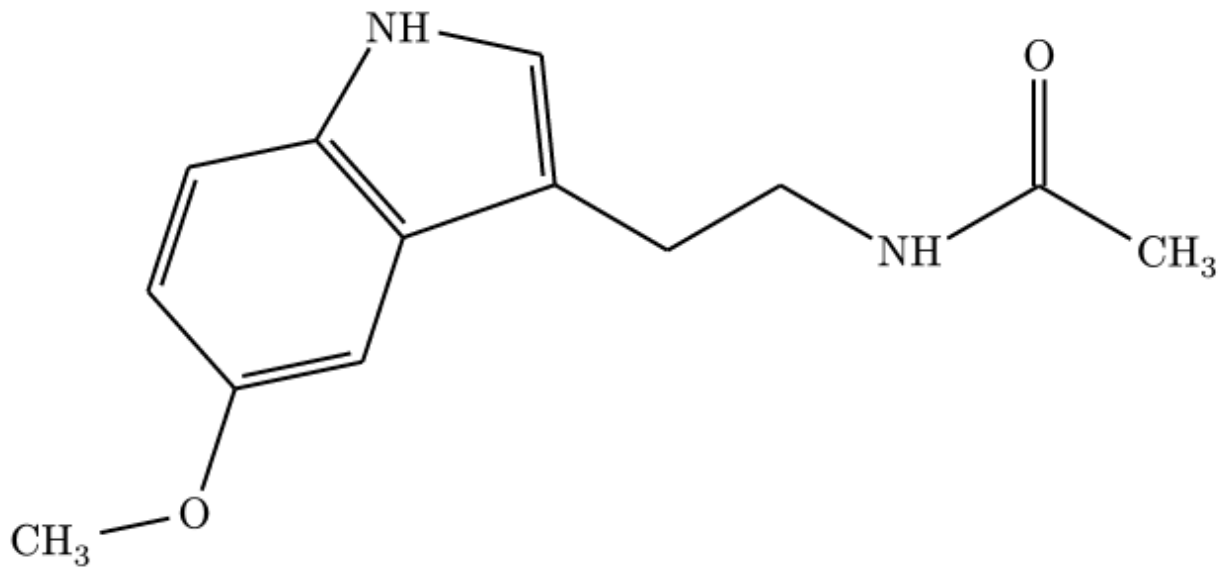
// Read your molecule data
// Example: https://pubchem.ncbi.nlm.nih.gov/compound/93406
#let mol-data = read("Structure2D_COMPOUND_CID_93406.sdf")
```

On Typst 0.15.0 and later, you may also pass `path("Structure2D_COMPOUND_CID_93406.sdf")` directly to `render-mol`; `molchemist` will read the file inside the package. The examples in this README use `read(...)` for compatibility with older Typst versions.

The bundled documentation and README screenshots use small PubChem-derived example structures. See THIRD_PARTY_NOTICES.md for source URLs and NCBI data-usage notes.

For SMILES, `molchemist` generates a 2D layout internally before sending the structure to `alchemist`.

```
// Example: https://pubchem.ncbi.nlm.nih.gov/compound/896
#render-smiles("CC(=O)NCCC1=CNC2=C1C=C(C=C2)OC", abbreviate: true)
```



Adding Annotations

You can overlay arrows and labels on top of a rendered molecule with the `annotations` argument. `molchemist` provides helpers for atom-level, bond-level, and molecule-level annotations without leaving the package API.

```
#import "@preview/molchemist:0.1.2": (
  render-smiles,
  atom-anchor,
  bond-anchor,
  molecule-anchor,
  callout-annotation,
  arrow-annotation,
```

```
)

#render-smiles(
  "OCCc1c(C)[n+](=cs1)Cc2cnc(C)nc(N)2",
  abbreviate: true,
  annotations: (
    callout-annotation(
      atom-anchor(6, anchor: "north"),
      [cationic center],
      side: "north-east",
    ),
    callout-annotation(
      bond-anchor(3, anchor: "50%"),
      [aromatic bond],
      side: "north-west",
    ),
    arrow-annotation(
      molecule-anchor(anchor: "east"),
      (rel: (2.6, 0), to: molecule-anchor(anchor: "east")),
      label: [reaction direction],
      label-offset: (0, -0.45),
      label-anchor: "north",
    ),
  ),
)
```

Use `callout-annotation(...)` for most explanatory labels, `arrow-annotation(...)` for free arrows, and `label-annotation(...)` for low-level labels. `atom-anchor(...)` targets a specific atom, `bond-anchor(...)` targets a specific bond, and `molecule-anchor(anchor: "center")` attaches to the molecule as a whole. Callouts are intentionally restrained for publication figures: unboxed labels, thin monochrome leader lines, no arrowheads by default, and enough clearance from both the label text and the chemical structure. Placement presets such as `side: "north-east"` cover the common cases, while `label-at`, `leader-start`, `leader-end`, `leader-points`, `label-gap`, and `target-gap` are available for small manual corrections when a paper figure needs precise spacing. For final figure-level adjustments, `cetz-annotation((mol) => { ... })` exposes the generated molecule name for direct Cetz drawing. To discover the atom and bond indices for a molecule, enable the debug overlay with `show-indices: true`, `show-indices: "atoms"`, or `show-indices: "bonds"`. In abbreviated or skeletal mode, the overlay only labels elements that are actually rendered.

Publication Figure Guidance

For paper figures, prefer `skeletal: true` for hydrocarbon-heavy structures and `abbreviate: true` when heteroatom hydrogens or terminal groups should remain explicit. Use Full Mode mainly for small molecules or debugging, since explicit hydrogens can make dense structures hard to read.

Keep annotations minimal: use monochrome `callout-annotation(...)` labels, avoid arrowheads unless the line represents a process, and move labels outside the molecular graph. If a leader line visually resembles a chemical bond, increase `target-gap`, move the label with `label-at`, or route the line through `leader-points`.

Rendering Modes

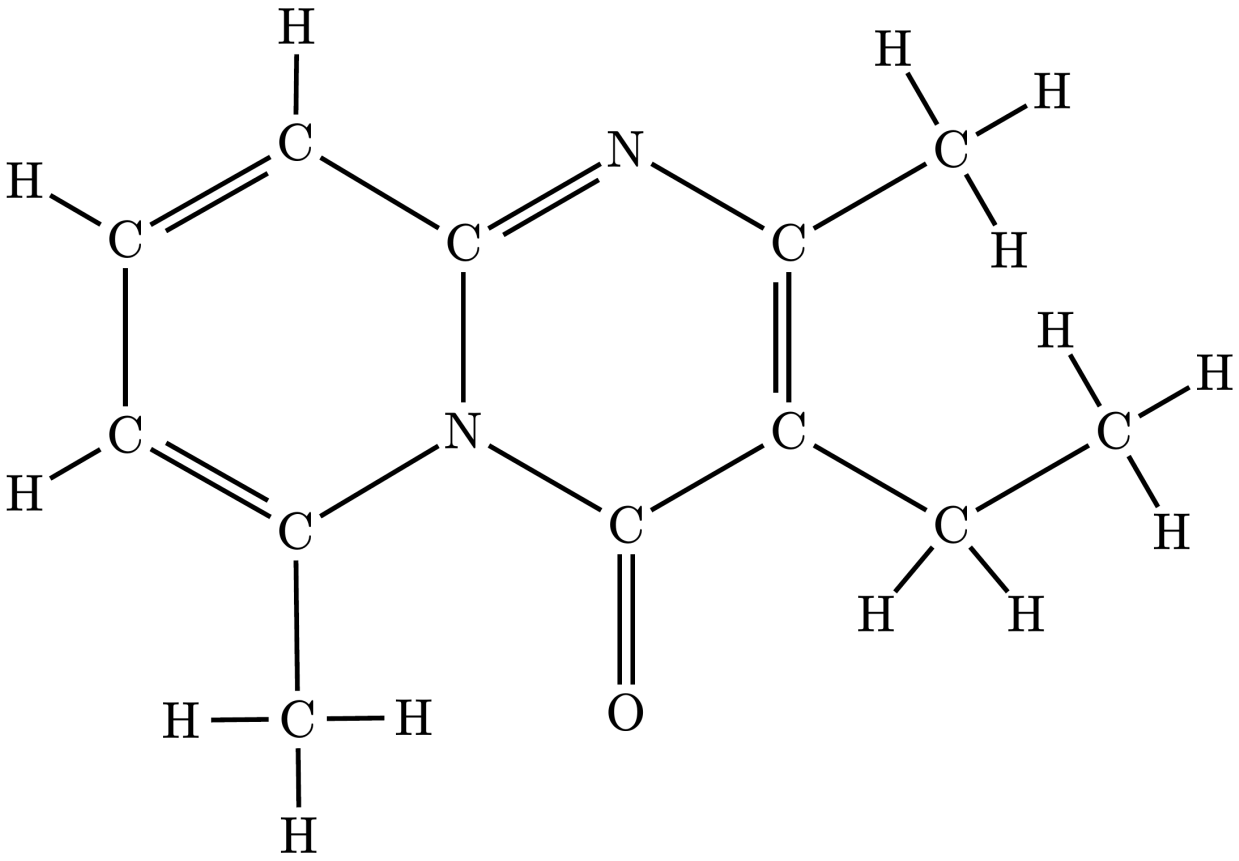
`molchemist` supports three distinct rendering styles to suit your document's needs:

1. Full Mode (Default)

Draws every single atom and bond explicitly exactly as defined in the source file, including all carbons and hydrogens.

Note: For complex molecules, text overlapping may occur. See Known Limitations for workarounds.

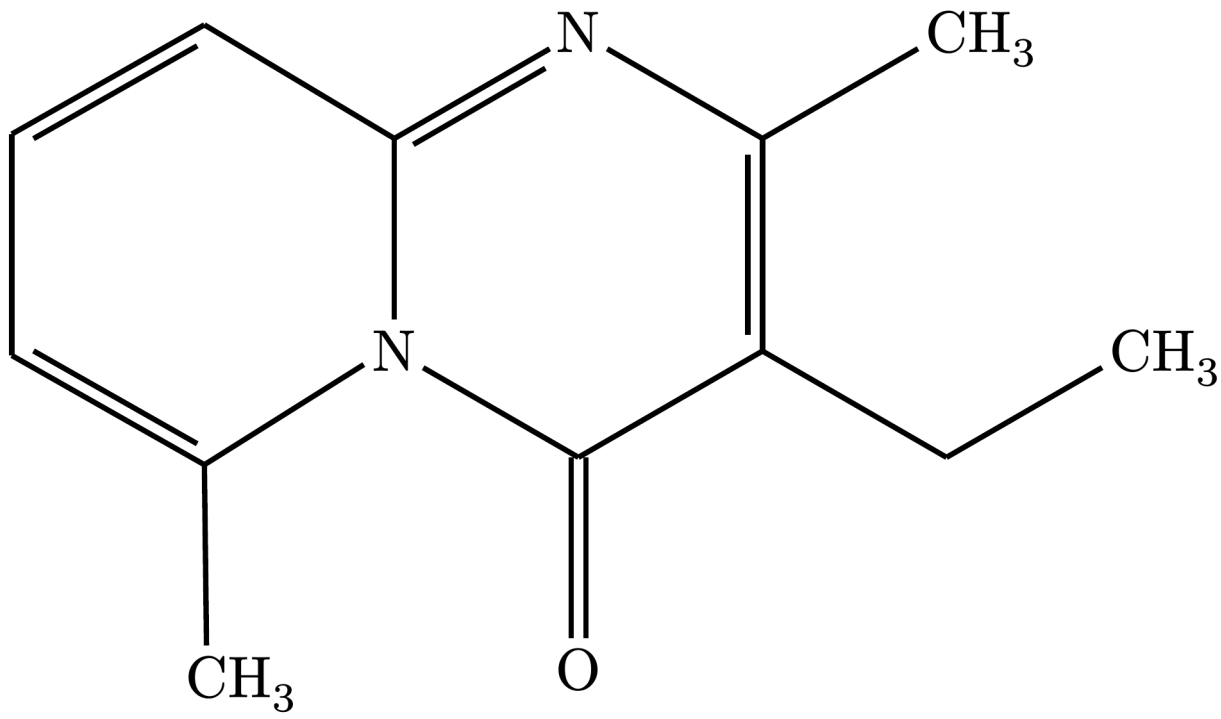
```
#render-mol(mol-data)
```



2. Abbreviated Mode

A standard chemical representation. It hides the carbon backbone, wraps explicit hydrogens into their parent heteroatoms (e.g., `O` + `H` becomes `OH`), and neatly formats terminal carbon groups (e.g., `CH3`).

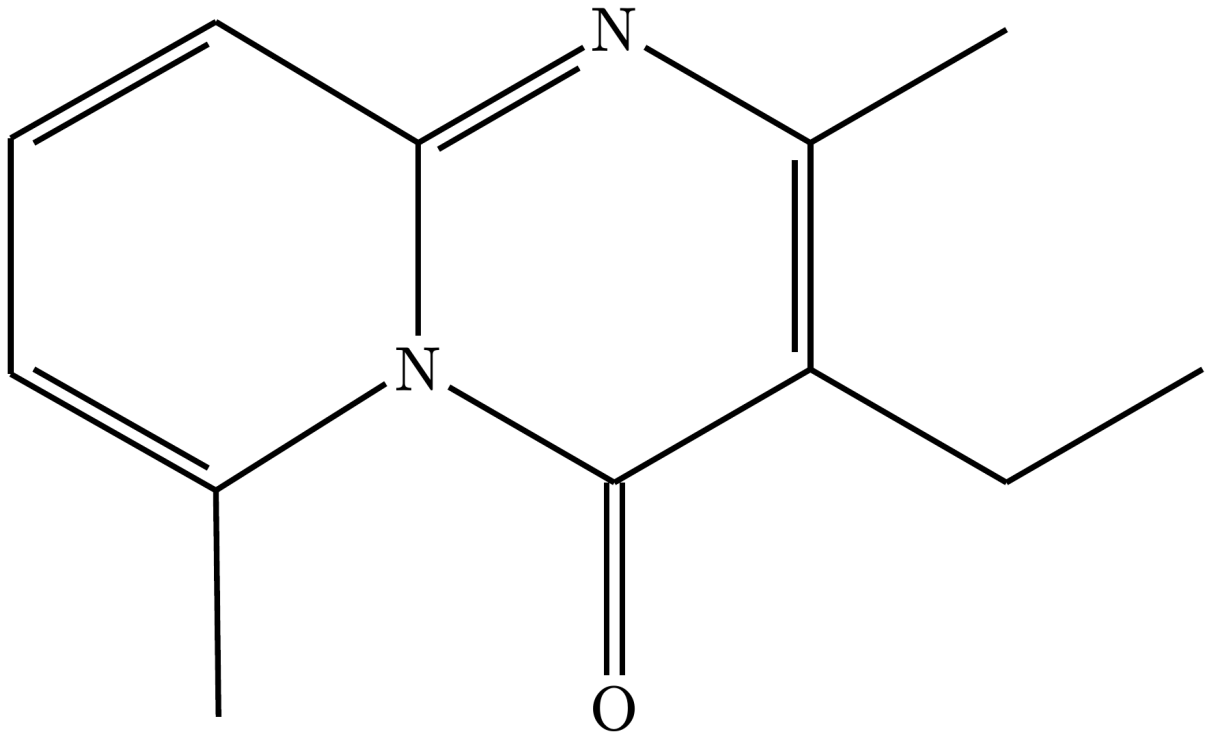
```
#render-mol(mol-data, abbreviate: true)
```



3. Skeletal Mode

A pure skeletal formula. All backbone carbons and their attached hydrogens are completely hidden, leaving only the zigzag lines and heteroatoms.

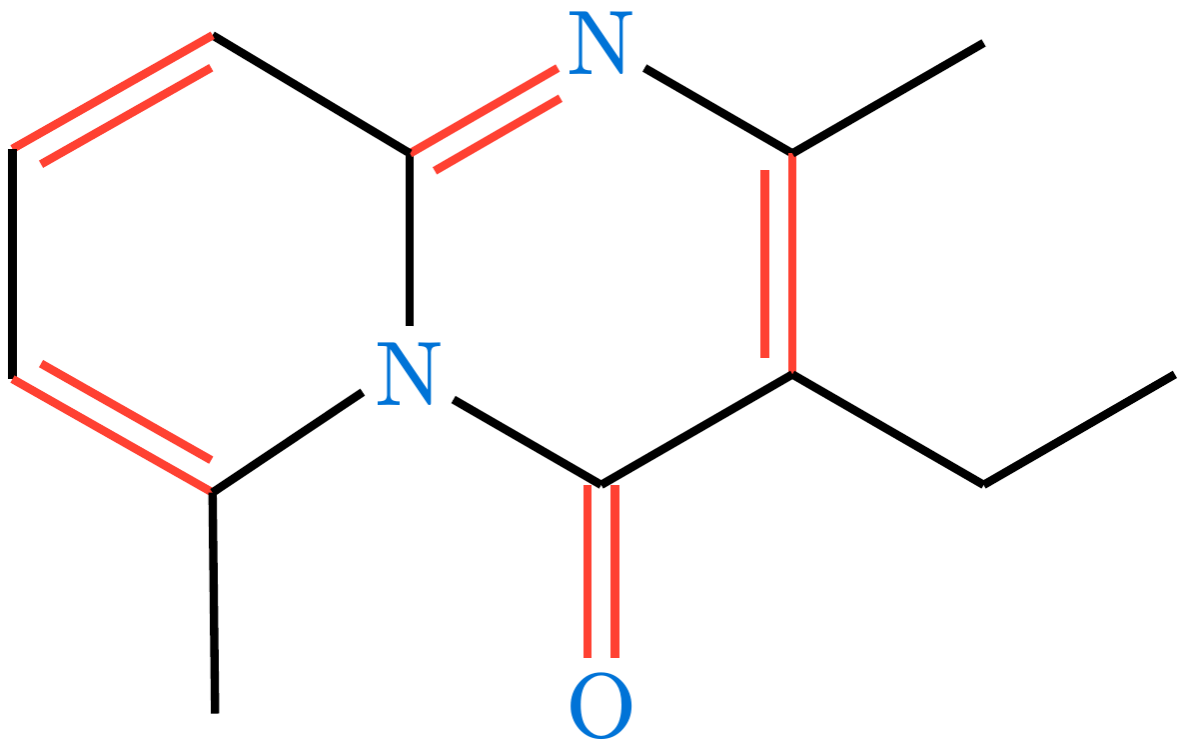
```
#render-mol(mol-data, skeletal: true)
```



Customizing Appearance

Under the hood, `molchemist` parses the graph and generates native `alchemist` elements. You can customize the look of your molecules by passing styling arguments via the `config` dictionary, which are passed directly to `alchemist`'s `skeletize` function.

```
#render-mol(
  mol-data,
  skeletal: true,
  config: (
    atom-sep: 2em,
    fragment-margin: 2pt,
    fragment-color: blue,
    fragment-font: "New Computer Modern",
    single: (stroke: 1pt + black),
    double: (gap: 0.3em, stroke: 1pt + red)
  )
)
```



Important Note on Configuration:

- **Routing overrides:** Because `molchemist` maps the exact 2D absolute coordinates from the source `.sdf`/`.mol` file, `alchemist`'s automatic routing configs (like `angle-increment`, `base-angle`) are bypassed and have no effect.
- **Lewis Structures:** `molchemist` does not automatically infer or generate Lewis structures from SDF files, so `lewis-*` configs are not applicable out of the box.

Advanced: Ejecting to Alchemist Code (Dump Mode)
If you need to manually fine-tune a molecule, add a specific Lewis structure, or integrate the structure into a larger custom `alchemist` drawing, you can use the `dump` parameter.

When `dump: true` is passed, `molchemist` will not render the molecule. Instead, it will output the generated native `alchemist` code block into your document. You can then copy, paste, and modify this code directly.

```
#render-mol(mol-data, dump: true)
```

```
  double(absolute: 30.000727780827354deg, atom-sep: base-sep * 1.2053621002571053, offset: "right")
  fragment("N", name: "a2")
  single(absolute: -29.997863113888936deg, atom-sep: base-sep * 1.2054664907148052)
  fragment("C", name: "a6")
  branch({
    double(absolute: -90deg, atom-sep: base-sep * 1.2053886190984355, offset: "right")
    fragment("C", name: "a3", links: (
      "a4": single(absolute: -150.00213688611106deg, atom-sep: base-sep * 1.2054664907148052),
    ))
    single(absolute: -30.000727780827354deg, atom-sep: base-sep * 1.2053621002571053)
    fragment("C", name: "a7")
    branch({
      single(absolute: 30.00072778082738deg, atom-sep: base-sep * 1.2053621002571044)
      fragment("C", name: "a11")
      branch({
        single(absolute: -60.00296847068038deg, atom-sep: base-sep * 0.7474080148754706)
        fragment("H", name: "a19")
      })
      branch({
        single(absolute: 29.997031529319667deg, atom-sep: base-sep * 0.7474080148754708)
        fragment("H", name: "a20")
      })
      single(absolute: 120.00165197278548deg, atom-sep: base-sep * 0.7473036244664235)
      fragment("H", name: "a21")
    })
    branch({
      single(absolute: -129.9948801549935deg, atom-sep: base-sep * 0.7473674327585829)
      fragment("H", name: "a15")
    })
    single(absolute: -50.00511984500657deg, atom-sep: base-sep * 0.7473674327585822)
    fragment("H", name: "a16")
  })
  single(absolute: 30.000727780827354deg, atom-sep: base-sep * 1.2053621002571053)
  fragment("C", name: "a12")
  branch({
    single(absolute: -60.00296847068038deg, atom-sep: base-sep * 0.7474080148754706)
    fragment("H", name: "a22")
  })
  branch({
    single(absolute: 30.00165197278554deg, atom-sep: base-sep * 0.747303624466423)
    fragment("H", name: "a23")
  })
  single(absolute: 120.00165197278552deg, atom-sep: base-sep * 0.7473036244664242)
  fragment("H", name: "a24")
})
single(absolute: 149.11831959471084deg, atom-sep: base-sep * 1.2554886995491945)
fragment("C", name: "a9")
branch({
  double(absolute: -150.44478717401188deg, atom-sep: base-sep * 1.2555773909515242, offset: "left")
  fragment("C", name: "a13")
  branch({
    single(absolute: -90deg, atom-sep: base-sep * 1.2555327856529301)
    fragment("C", name: "a10")
    branch({
      double(absolute: -29.559997423347433deg, atom-sep: base-sep * 1.255636852575184, offset: "left")
      fragment("C", name: "a8", links: (
        "a1": single(absolute: 30.886401239241575deg, atom-sep: base-sep * 1.2555505724154539),
      ))
      single(absolute: -89.34106169184625deg, atom-sep: base-sep * 1.2053477918944862)
      fragment("C", name: "a14")
      branch({
        single(absolute: -179.3345522447223deg, atom-sep: base-sep * 0.7472708044296151)
        fragment("H", name: "a26")
      })
      branch({
        single(absolute: -89.33465956535612deg, atom-sep: base-sep * 0.7473913351631287)
        fragment("H", name: "a27")
      })
      single(absolute: 0.6561004087622995deg, atom-sep: base-sep * 0.7473899451702973)
      fragment("H", name: "a28")
```

Known Limitations

When rendering highly complex or dense molecules (e.g., polycyclic compounds, dense substituents) in the default **Full Mode**, you may encounter overlapping atoms or intersecting bonds. This occurs because the 2D absolute coordinates provided in the source `.sdf`/`.mol` files might not allocate enough physical space on the canvas to draw every explicit text label without collisions.

Recommended Workarounds:

1. **Use Abbreviated or Skeletal Mode:** For complex organic structures, it is highly recommended to set `abbreviate: true` or `skeletal: true`. This hides redundant atoms, dramatically improving readability and preventing overlaps, which aligns with standard chemical drawing practices.
2. **Increase Bond Length:** If you strictly require Full Mode, you can increase the distance between atoms to create more physical space for the text labels by adjusting the `atom-sep` property in the `config` argument:

```
// The default atom-sep is 3em
#render-mol(mol-data, config: (atom-sep: 4.5em))
```

For SMILES input, the default `render-smiles(...)` mode expands implicit hydrogens into explicit `H` atoms so that `full` mode stays closer to the behavior of `render-mol(...)`. Highly complex or dense molecules can still become visually busy in `full` mode, so `abbreviate: true` or `skeletal: true` will often produce a clearer result. The current implementation also supports tetrahedral `@` / `@@` centers and `/` / `\` double-bond geometry as stereochemical depictions. Extended OpenSMILES chirality classes such as `@AL`, `@SP`, `@TB`, and `@OH` are accepted as well; because the current `alchemist`-based renderer does not have native glyphs for those geometries, they are preserved as stereo annotations below the rendered structure instead of wedge/dash depictions.

Feature Plan

- **Native depiction for extended chirality:** Extended OpenSMILES chirality classes such as allene (`@AL`), square-planar (`@SP`), trigonal-bipyramidal (`@TB`), and octahedral (`@OH`) are currently preserved as textual stereo annotations. A future version may render these classes natively once the expected 2D depiction conventions and the required `alchemist` primitives are clarified.

API Reference

Renderers

```
#render-mol(data, ..options)
#render-smiles(smiles, ..options)
```

Function	Input	Description
`render-mol`	`data: str`, `bytes`, or Typst 0.15+ `path`	Renders Molfile or SDF data. Coordinates are read from the input.
`render-smiles`	`smiles: str`	Parses SMILES, generates a 2D layout, and renders the result.

Both renderers accept the same options:

Option	Type	Default	Description
`abbreviate`	`bool`	`false`	Folds common hydrogens and terminal groups into labels.
`skeletal`	`bool`	`false`	Draws a skeletal formula. Overrides `abbreviate`.
`dump`	`bool`	`false`	Returns generated `alchemist` source instead of rendering.
`config`	`dictionary`	`(:)`	Passes visual settings directly to `alchemist`.
`annotations`	`annotation`, `array`, `none`	`none`	Adds labels, arrows, or custom CeTZ overlays.
`show-indices`	`bool`, `str`	`false`	Shows debug labels for annotation authoring. Use `true`, `"all"`, `"atoms"`, or `"bonds"`.

Anchors

Use anchors to target atoms, bonds, or the whole molecule from annotations.

Function	Returns	Use
`atom-anchor(index, anchor: "mid")`	anchor selector	Target a rendered atom.
`bond-anchor(index, anchor: "50%")`	anchor selector	Target a rendered bond.
`molecule-anchor(anchor: "center")`	anchor selector	Target the rendered molecule group.
`atom-ref(index)`	`str`	Inspect the generated atom anchor name.
`bond-ref(index)`	`str`	Inspect the generated bond anchor name.

Annotations

Pass one annotation or an array of annotations to `annotations`.

Function	Purpose
`callout-annotation(at, label, ..options)`	External publication-style label with a thin leader line.
`arrow-annotation(from, to, ..options)`	Free arrow overlay for process arrows or directional marks.
`label-annotation(at, label, ..options)`	Free text label without a leader line.
`cetz-annotation(body, ..options)`	Low-level CeTZ overlay. The callback receives the generated molecule name.

Common `callout-annotation` controls include `side`, `label-at`, `leader`, `leader-start`, `leader-end`, `leader-points`, `label-gap`, and `target-gap`. Use these when a final figure needs precise spacing.

Use `cetz-annotation` as the escape hatch for advanced figure polishing:

```
#render-smiles(
  "c1ccccc1",
  skeletal: true,
  annotations: cetz-annotation(mol => {
    import cetz.draw: *
    content((to: (name: mol, anchor: "north"), rel: (0, 0.45)))[benzene]
  }),
)
```

License

The `molchemist` package code is distributed under the MIT License. See LICENSE for details.

Redistributed third-party code and bundled example-data provenance, including PubChem-derived SDF/example images, are documented separately in THIRD_PARTY_NOTICES.md.

unidep

A fast, beautiful, and highly customizable Typst package for rendering Universal Dependencies (CoNLL-U) trees, powered by Rust/WASM and CeTZ.

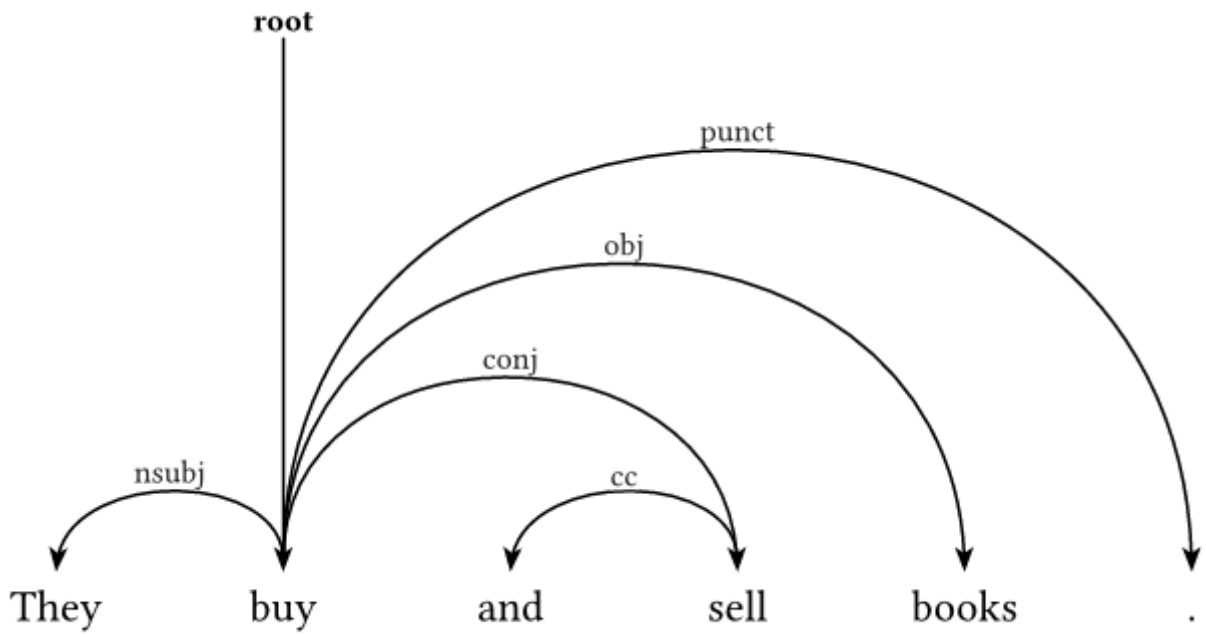
This package provides an elegant way to visualize dependency parsing results, syntax trees, and enhanced dependency graphs directly in Typst without relying on external Python scripts or LaTeX's `tikz-dependency`.

Usage

Import the package and use the `dependency-tree` function. Pass your CoNLL-U text to it.

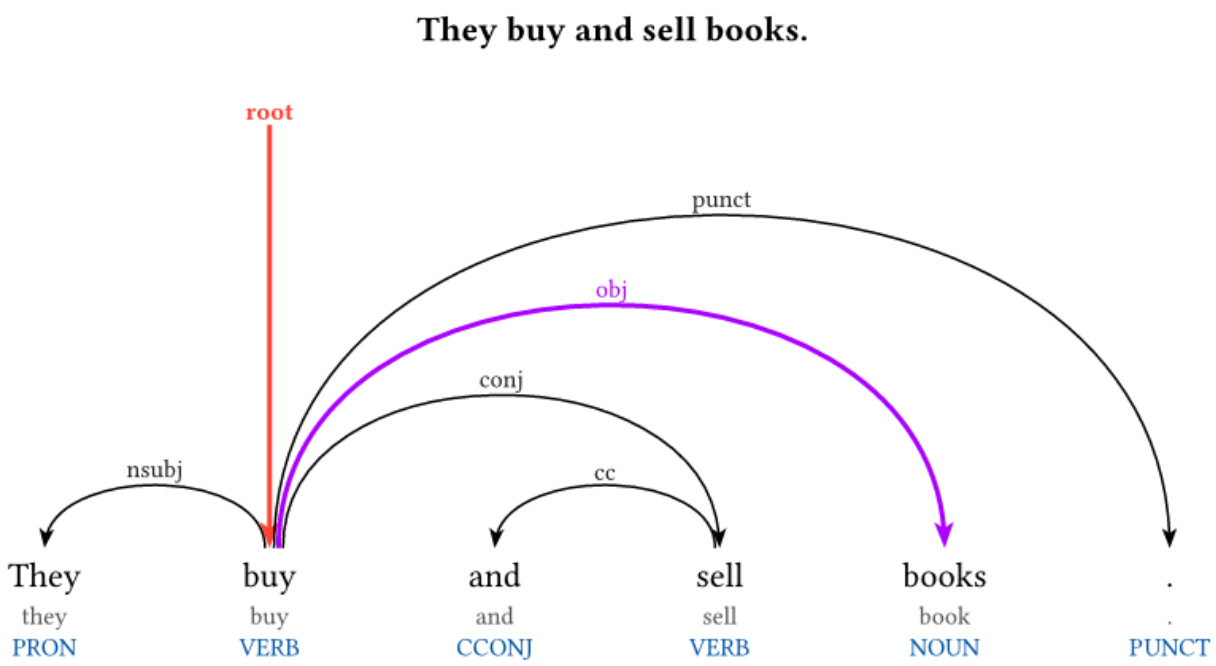
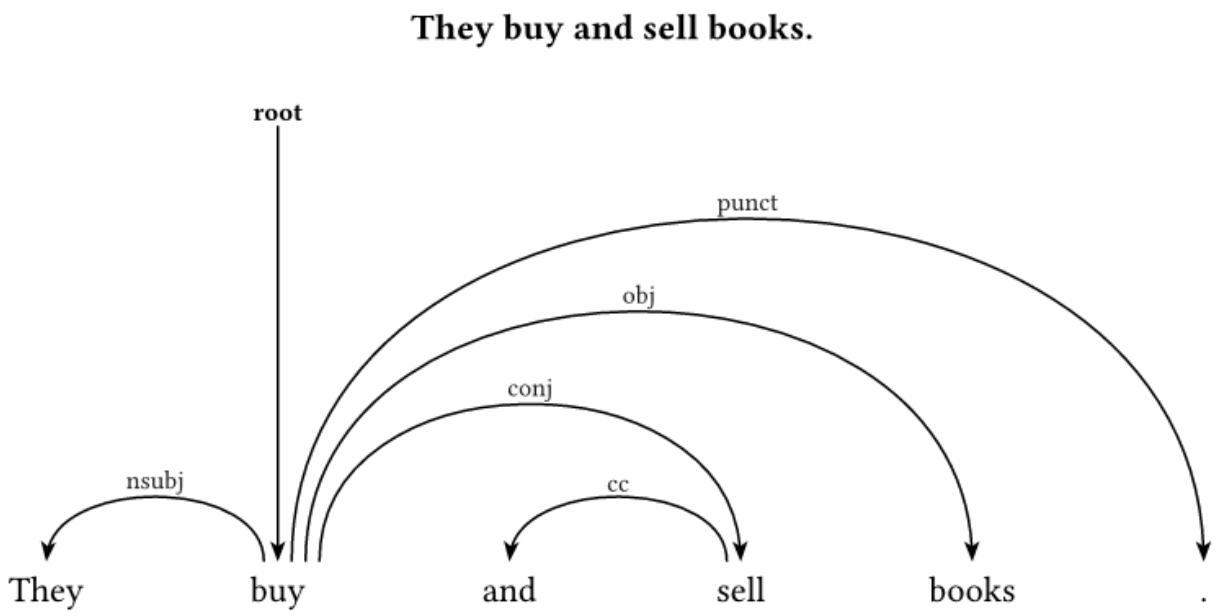
```
#import "@preview/unidep:0.1.4": dependency-tree
#set page(width: auto, height: auto, margin: 3mm)

#let sample-conllu = ```
# sent_id = 1
# text = They buy and sell books.
1 They  they  PRON  PRP _ 2 nsubj _ _
2 buy buy VERB  VBP _ 0 root  _ _
3 and and CCONJ CC  _ 4 cc  _ _
4 sell  sell  VERB  VBP _ 2 conj  _ _
5 books book  NOUN  NNS _ 2 obj _ _
6 . . PUNCT . _ 2 punct _ _
```.text

#dependency-tree(sample-conllu)
```

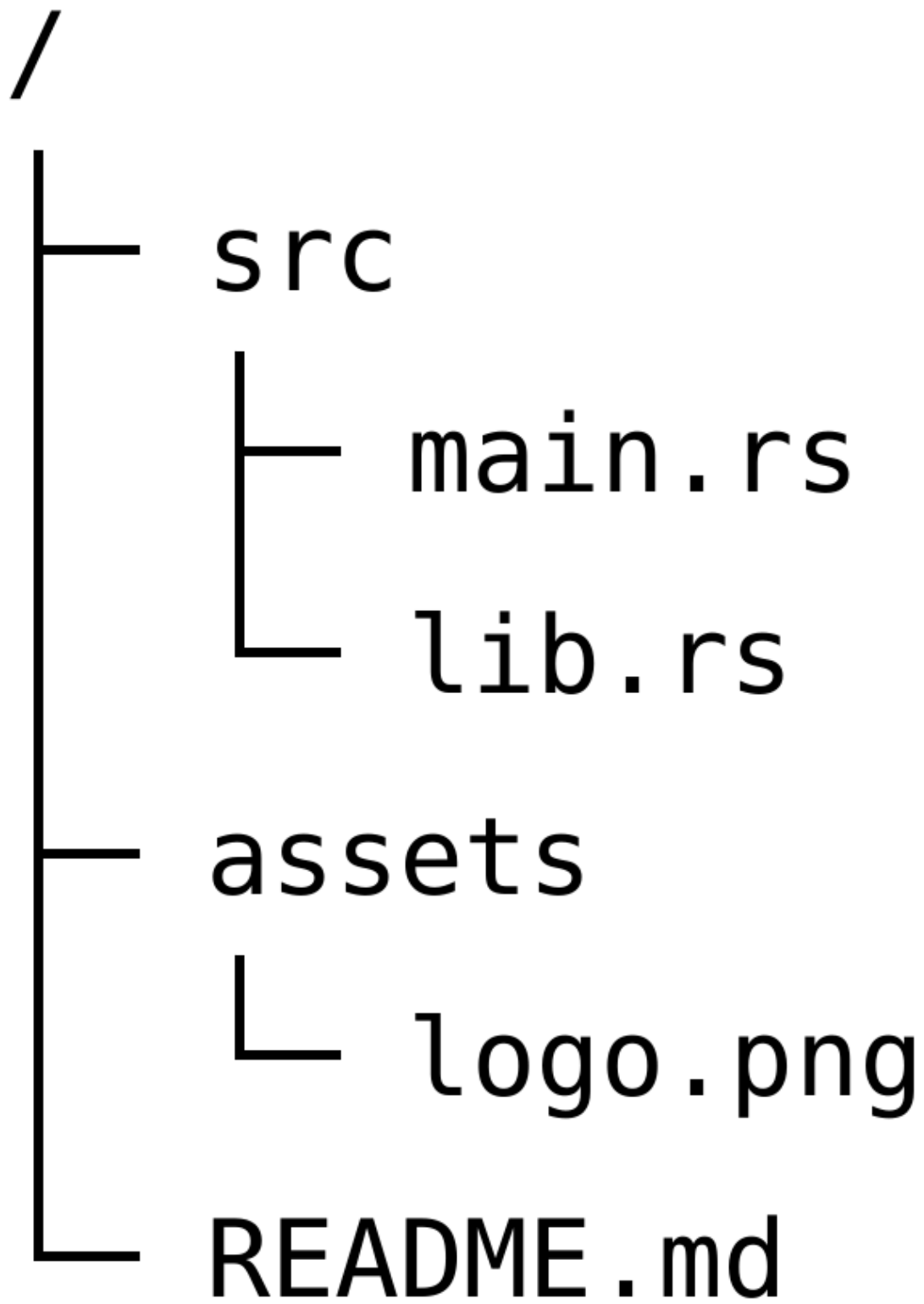



## Advanced Usage & Highlighting

You can display the original sentence text, lemmas, and part-of-speech tags (UPOS, XPOS) directly around the words. Additionally, you can highlight specific dependency arcs by targeting the dependent's ID using the `highlights` dictionary.

```
#dependency-tree(
  sample-conllu,
  show-text: true,
  word-spacing: 2.5,
  text-gap: 0.35em,
```
```

```
  sentence-gap: 0.75em,
  show-upos: true,
  show-lemma: true,
  tail-offset: 0.0,
  tail-spacing: 0.05,
  highlights: (
    "2": red,          // Highlight the ROOT arrow pointing to 'buy'
    "5": rgb("aa00ff") // Highlight the arc pointing to 'books'
  )
)
```



Arc Geometry

When multiple arcs meet around the same token, `tail-spacing` fans out the non-arrowhead side of each arc while keeping the arrowhead anchored to the token.

```
#dependency-tree(
  sample-conllu,
  show-text: true,
  word-spacing: 2.5,
  tail-offset: 0.0,
  tail-spacing: 0.15,
)
```



The default `tail-angle`, `head-angle`, and `tail-offset` already produce upright joins. Adjust them only when you want a softer or steeper look.

API Reference

`dependency-tree(conllu-text, ..args)`

Renders one or more sentences from a CoNLL-U formatted string.

Parameters:

- `conllu-text` (String): The raw text in CoNLL-U format.
- `word-spacing` (Float): Horizontal distance between words. Default: `2.0`.
- `level-height` (Float): Vertical height increment for each arc level. Default: `1.0`.
- `arc-roundness` (Float): Controls the curvature of the bezier arcs. Lower values make arcs steeper (more U-shaped); higher values make them more elliptical. Default: `0.18`.
- `tail-spacing` (Float/None): Horizontal spacing step for non-arrowhead endpoints on the same token side. Lower arcs are placed farther out and higher arcs farther in so clustered starts avoid crossing. Default: `none`, which resolves to `0.0`.
- `head-spacing` (Float/None): Horizontal spacing step for arrowhead endpoints on the same token side. Default: `none`, which resolves to `0.0`.
- `tail-angle` (Angle/Float/None): Connection angle for the non-arrowhead side of each arc. Default: `90deg`.
- `head-angle` (Angle/Float/None): Connection angle for the arrowhead side of each arc. Default: `90deg`.
- `tail-offset` (Float): Vertical offset for the tail (start point) of the dependency arc. Useful for fine-tuning the join near the source token. Default: `0.1`.
- `head-offset` (Float): Vertical offset for the head (end point/arrowhead) of the dependency arc. Useful for fine-tuning the gap between the arrowhead and the token. Default: `0.0`.
- `text-gap` (Relative Length): Vertical gap between the optional sentence text (`# text = ...`) and the dependency tree. Default: `0.5em`.
- `sentence-gap` (Relative Length): Vertical gap inserted after each rendered sentence. Default: `1em`.
- `show-text` (Bool): If `true`, displays the sentence text (extracted from `# text = ...` metadata) above the parsed tree. Default: `false`.
- `show-upos` (Bool): If `true`, displays the Universal POS tag (column 4) below the word. Default: `false`.
- `show-xpos` (Bool): If `true`, displays the language-specific POS tag (column 5) below the word. Default: `false`.
- `show-lemma` (Bool): If `true`, displays the lemma (column 3) below the word. Default: `false`.

- `show-enhanced-as-dashed` (Bool): If `true`, arcs derived from the `DEPS` column (enhanced graph) are drawn as blue dashed lines to distinguish them from the basic tree. Default: `true`.
- `show-root` (Bool): If `true`, renders a vertical root arrow centered on the token. Default: `true`.
- `highlights` (Dictionary): A dictionary of `("ID": color)` pairs to apply custom stroke colors and thicker lines to specific arcs. The key must be the ID of the dependent token (e.g., `"4"`). Default: `(:)`.

License

dtree

dtree is a flexible and highly customizable directory tree visualization package for Typst. It renders directory structures using simple indented text, supports smart icon mapping, automatic styling rules, and vector-based connecting lines.

Usage

1. Basic Usage

The simplest way to use `dtree` is to pass a raw block. By default, it uses the standard monospace font defined in your document.

````
#import "@preview/dtree:0.1.0": dtree

#dtree(```
/
 src
  main.rs
  lib.rs
 assets
  logo.png
 README.md
```)
````
````


/
src
main.rs
lib.rs
assets
logo.png
README.md

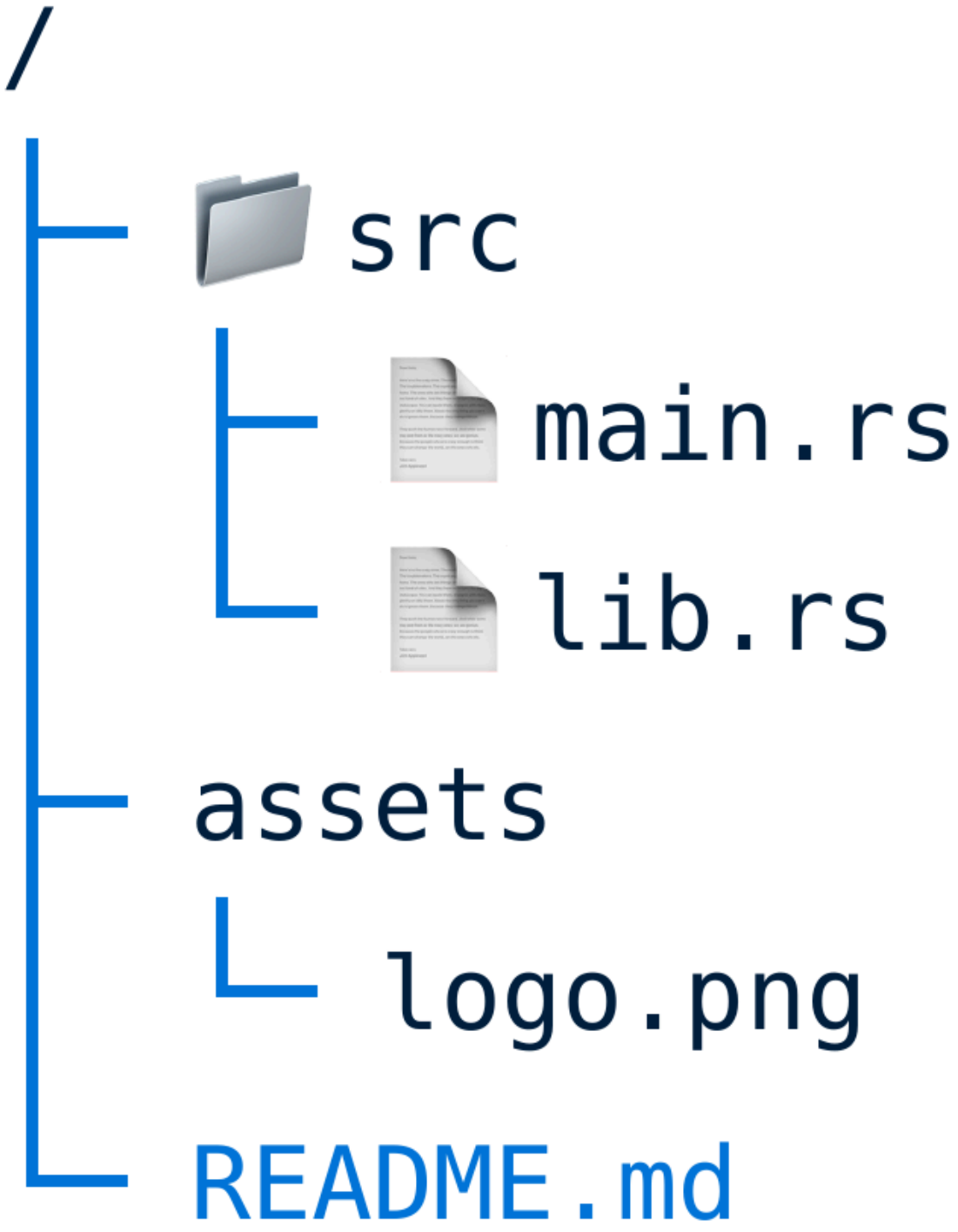
2. Styling and Inline Parameters

You can customize the tree globally or per line. Use the | delimiter to specify an icon or parameters before the filename.

```
#dtree(
  stroke: 1pt + blue,
  fill: navy,
```

```
/
 □ | src
  □ | main.rs
  □ | lib.rs
 assets
  logo.png
 fill=blue | README.md
```

```
)
```



3. Advanced: Icon Rules & Images

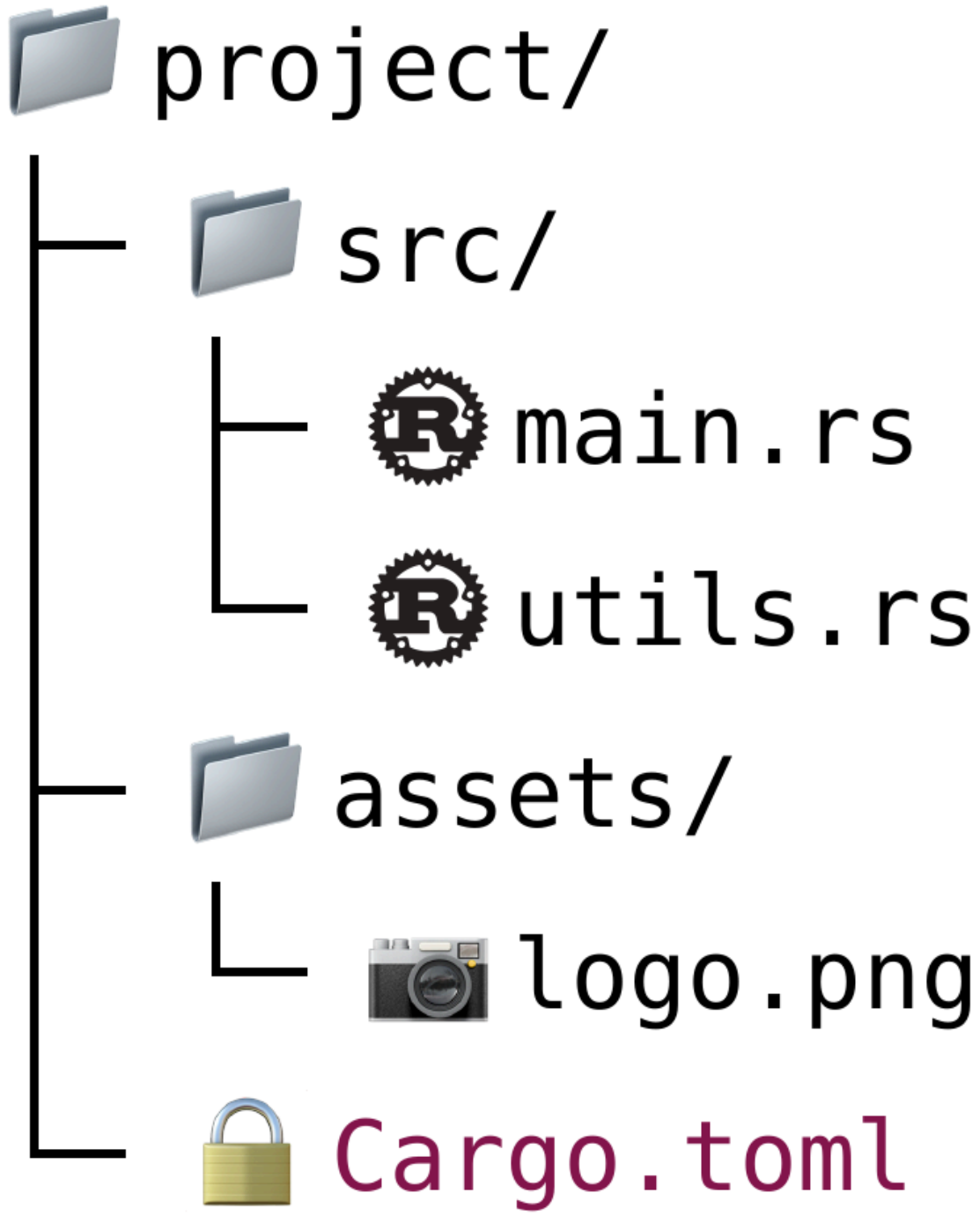
To use images, you must read the file as bytes using `read(..., encoding: none)` and pass it to the `icons` dictionary.

Note: When using `regex` patterns, `icon-rules` must be an **array of pairs** (e.g., `((key, val),)`) instead of a dictionary.

```
#let rust-icon = read("rust-logo.png", encoding: none)
#let config-icon = "□"
```

````
#let image-icon = "□"
#let folder-icon = "□"

#dtree(
  icons: (
    "rs": rust-icon,
    "img": image-icon,
    "dir": folder-icon,
  ),
  icon-rules: (
    (regex("/$"), "dir"),
    ("*.rs", (icon: "rs")),
    ("*.png", (icon: "img")),
    ("Cargo.toml", (icon: config-icon, fill: maroon)),
  ),
```
project/
 src/
  main.rs
  utils.rs
 assets/
  logo.png
 Cargo.toml
```
)
````



Input Syntax

Each line in the raw block is parsed as follows:

`[Icon Name], [Key=Value], ... | [Content]`

1. **Icon Name** (Optional):
   - A key defined in the `icons` dictionary.
   - A raw emoji or text string (e.g., , `Rs`).
   - **Note:** Direct file paths (e.g. `image.png`) are not supported inside the raw block. Please load images via the `icons` dictionary.

2. **Parameters** (Optional): Comma-separated `key=value` pairs.
   - Supported keys: `size`, `dx`, `dy`, `fill` (or `color`), `font`.
3. **Delimiter**: The `|` character separates metadata from the content name.

Examples:

- `  | Documents` (Icon only)
- `fill=red | Important.txt` (Params only)
- `my_icon, size=1.5em, dy=2pt | image.png` (Icon + Params)
- `file.txt` (Content only)

Icon Priority

Icons are resolved in the following order (highest to lowest priority):
1. **Inline specification** — Direct icon specified in the line (e.g., `  | sushi.txt`)
2. **`icon-rules`** — Pattern-based rules that match the content name
3. **`default-icon`** — Fallback icon if no other match is found

API Reference

`dtree`

Argument	Type	Default	Description
body	content or str	Required	The directory structure text. Using a raw block is recommended to preserve indentation.
Layout	indent-width	length	1.5em
The horizontal width of one indentation level.	row-height	length	1.5em
The height of a single row.	spacing	length	0.5em
The space between the tree lines and the content.	indent-marker	str	" "
The character used to calculate indentation levels (e.g., set to "\t" for tabs or " " for 2-space indentation).	Style	stroke	stroke
0.7pt + black	The style of the connecting lines (thickness + color).	font	str or array
none	The font family. If none , uses raw() (inherits code font). If specified, uses text(font: ...) .	size	length
10pt	The font size for filenames.	fill	color
black	The text color for filenames.	Icons	icons
dictionary	(:)	A dictionary mapping names to content or bytes (loaded via read(..., encoding: none)).	icon-rules

Argument	Type	Default	Description
array	()	An array of pairs mapping patterns (Glob *.ext or regex) to an icon name or a style dictionary. Example: ((regex("..."), val),) .	default-icon
content	none	An icon to display if no specific icon or rule matches.	icon-size
length	1em	The default size for icons.	icon-dx
length	-1pt	Horizontal offset for icons.	icon-dy
length	0pt	Vertical offset for icons.	

Parameter Dictionary (`icon-rules` & Inline Params)

When defining `icon-rules` values or using inline syntax, the following keys are available:

- `icon`: (String) The name of the icon to use (only for `icon-rules`).
- `size`: (Length) Size of the icon/image.
- `fill` or `color`: (Color) Text color.
- `font`: (String) Font family.
- `dx`: (Length) Horizontal offset.
- `dy`: (Length) Vertical offset.

License

jlreq-tcf

This package provides commands to arrange footnotes into two columns (left and right) at the bottom of the page, designed to visually blend with the `jlreq` document class.

For full details and visual examples, please refer to the documentation.

Installation

Using l3build (Recommended)

To install this package using `l3build`, run the following command in the package directory:

```
l3build install
```

Manual Installation

Alternatively, you can copy `jlreq-tcf.sty` to a location where LaTeX can find it (e.g., your local `texmf` tree or the same directory as your project `.tex` file).

Usage

Load the package in your preamble:

```
\usepackage{jlreq-tcf}
```

Command Specifications

This package uses its own internal counter (shared between column L and column R) to manage footnote numbers.

Basic Footnotes

These commands insert a footnote mark and the corresponding text immediately.

- **`\footnoteL{<text>}`**
- Typesets a footnote in the **left-hand** column.
- Automatically increments the internal footnote counter.
- **`\footnoteR{<text>}`**
- Typesets a footnote in the **right-hand** column.
- Automatically increments the internal footnote counter.

Separated Mark and Text

You can separate the mark generation from the text definition. This is useful when placing footnotes in titles, tables, or other restricted environments.

- **`\footnotemarkL[<num>]`** / **`\footnotemarkR[<num>]`**
- Prints the footnote mark for the left (L) or right (R) column.
- **With `[<num>]`**: Uses the integer `<num>` as the mark number. Does **not** increment the internal counter.
- **Without `[<num>]`**: Increments the internal counter and uses that value.
- **`\footnotetextL[<num>]{<text>}`** / **`\footnotetextR[<num>]{<text>}`**
- Defines the footnote text for the left (L) or right (R) column without printing a mark in the main text.
- **With `[<num>]`**: Uses `<num>` as the label in the footnote area.
- **Without `[<num>]`**: Uses the current value of the internal counter.

Example

```
\documentclass{jlreq}
\usepackage{jlreq-tcf}
\usepackage{bxjalipsum}
```

```
\begin{document}
% \footnoteL (Left column)
\jalipsum[1]{kusamakura}~\footnoteL{左の脚注1}\par
\jalipsum[2]{kusamakura}~\footnoteL{左の脚注2}\par
\jalipsum[3]{kusamakura}~\footnoteL{左の脚注3}

% \footnoteR (Right column)
\jalipsum[4]{kusamakura}~\footnoteR{右の脚注1}\par
\jalipsum[5]{kusamakura}~\footnoteR{右の脚注2}\par
\jalipsum[6]{kusamakura}~\footnoteR{右の脚注3}

% Separated mark and text
\jalipsum[7]{kusamakura}~\footnotemarkL
\jalipsum[8]{kusamakura}~\footnotemarkR

% Text definitions for the marks above
\footnotetextL[7]{左の脚注4}
\footnotetextR[8]{右の脚注4}
\end{document}
```

Known Issues and Limitations

Please be aware of the following limitations regarding the design and implementation of this package:

1. **Layout Emulation**:

This package attempts to reproduce the footnote layout (rules, spacing, indentation) of the `jlreq` class. However, this is an **emulation** created using `minipage` environments and manual spacing. It does not strictly inherit the internal typesetting logic of `jlreq`. There may be slight visual discrepancies depending on the document settings.

1. **Conflict with Standard `\footnote`**:

Do not use the standard `\footnote` command on the same page as `\footnoteL` or `\footnoteR`. Since this package manages its own output routine for two-column footnotes, using the standard `\footnote` command simultaneously will result in **two separate footnote blocks** appearing at the bottom of the page (one handled by LaTeX's standard mechanism and one by this package). This usage is not supported.

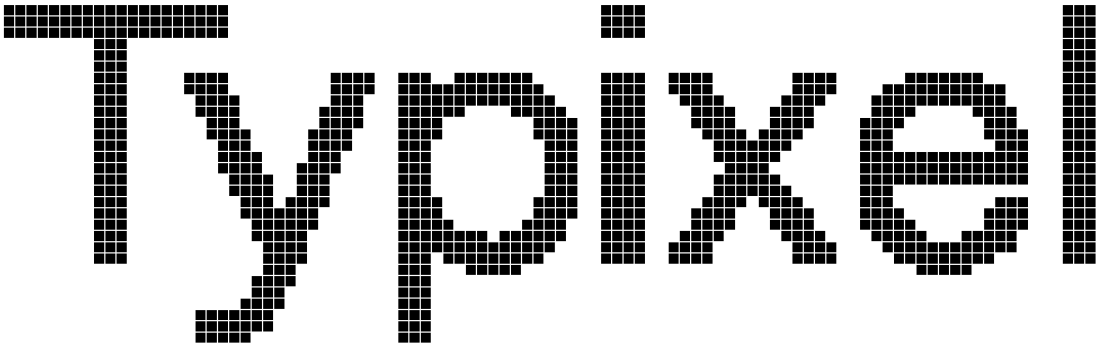
1. **Long Footnotes**:

Extremely long footnotes that exceed the available height of the page may cause the layout to break or push content off the page. The internal mechanism relies on `minipage`, which limits the ability to break long footnote text across pages.

License

This package is distributed under the BSD 2-Clause License. See LICENSE.

Typixel



A Typst package for creating pixel art directly in your documents. Convert images to pixel art or design custom pixel graphics using simple text maps.

Usage

Image to Pixel Art

```
#import "@preview/typixel:0.1.2": *
#set page(width: auto, height: auto, margin: 1cm)

#grid(
  columns: 2,
  gutter: 10pt,
  image("gahag.jpg", width: 5cm),
  pixel-image(
    read("gahag.jpg", encoding: none),
    rows: 32,
    shape: square-shape,
    width: 5cm,
  )
)
```

Text-based Pixel Map

```
#import "@preview/typixel:0.1.2": *
#set page(width: auto, height: auto, margin: 1cm)
```

```
#pixel-map(
  "
  ....XXX....
  ...XXXXX...
  ..XXXXXXX..
  .XXXXXXXXX.
  XXXXXXXXXXX
  ",
  palette: ("X": red, ".": none),
  pixel-size: 10pt
)
```

API Reference

`pixel-image()`

Convert raw image data to pixel art.

Parameters:

- `image-data` (bytes/path): Raw image data from `read("path", encoding: none)`, or a project-side `path("path")` value with Typst 0.15.0 or later
- `columns` (int, default: auto): Number of pixel columns
- `rows` (int, default: auto): Number of pixel rows
- `scale` (float, default: auto): Scale factor for image
- `colors` (int, default: 64): Number of colors in palette
- `transparency-char` (string, default: "."): Character representing transparency
- `pixel-size` (length, default: 5pt): Size of each pixel
- `width` (length, default: auto): Total width (overrides pixel-size)
- `shape` (function, default: square-shape): Shape function for pixels
- `gap` (length, default: 0pt): Gap between pixels

Example:

```
#pixel-image(
  path("logo.png"),
  columns: 48,
  colors: 32,
  pixel-size: 6pt,
  gap: 1pt,
  shape: circle-shape
)
```

For Typst 0.13.x through 0.14.x, pass bytes explicitly:

```
#pixel-image(
  read("logo.png", encoding: none),
  columns: 48
)
```

`pixel-map()`

Create pixel art from a text grid.

Parameters:

- `map` (string): Multi-line string defining the pixel grid
- `palette` (dictionary): Mapping of characters to colors
- `pixel-size` (length, default: 10pt): Size of each pixel
- `width` (length, default: auto): Total width (overrides pixel-size)
- `shape` (function/dictionary, default: square-shape): Shape(s) for pixels
- `gap` (length, default: 0pt): Gap between pixels

Example:

```
#pixel-map(
  "
  .00000.
  0.....0
  0.0.0.0
  0.....0
  .00000.
  ",
  palette: (
    "0": black,
    ".": white
  ),
  pixel-size: 15pt
)
```

`get-pixel-data()`

Obtain the raw pixel art data (grid and palette) from an image without rendering. Useful for custom processing or debugging.

Parameters:

- `image-data` (bytes/path): Raw image data from `read("path", encoding: none)`, or a project-side `path("path")` value with Typst 0.15.0 or later
- `columns` (int, default: auto): Number of pixel columns
- `rows` (int, default: auto): Number of pixel rows
- `scale` (float, default: auto): Scale factor for image
- `colors` (int, default: 64): Number of colors in palette

Returns: A dictionary containing:

- `art` (string): The generated character grid (rows separated by newlines).
- `palette` (dictionary): Mapping of characters to color values (or `none`).

Example:

```
#let data = get-pixel-data(
  path("icon.png")
)

// Access the raw grid string
#let grid-str = data.art
```

Available Shapes

Built-in Shape Functions

- `square-shape` - Standard square pixels
- `circle-shape` - Circular pixels
- `rounded-shape` - Rounded corner squares (customizable radius)
- `diamond-shape` - Diamond/rotated square pixels
- `star-shape` - Five-pointed star pixels
- `cross-shape` - X-shaped pixels (customizable thickness)
- `heart-shape` - Heart-shaped pixels

Using Shapes

Single shape for all pixels:

```
#pixel-map(map, palette: palette, shape: circle-shape)
```

Different shapes per character:

```
#pixel-map(
  map,
  palette: ("X": red, "O": blue),
  shape: (
    "X": star-shape,
    "O": circle-shape
  )
)
```

Custom shape parameters:

```
#pixel-map(
  map,
  palette: palette,
  shape: (w, h, f) => rounded-shape(
    width: w,
    height: h,
    fill: f,
    radius: 30%
  )
)
```

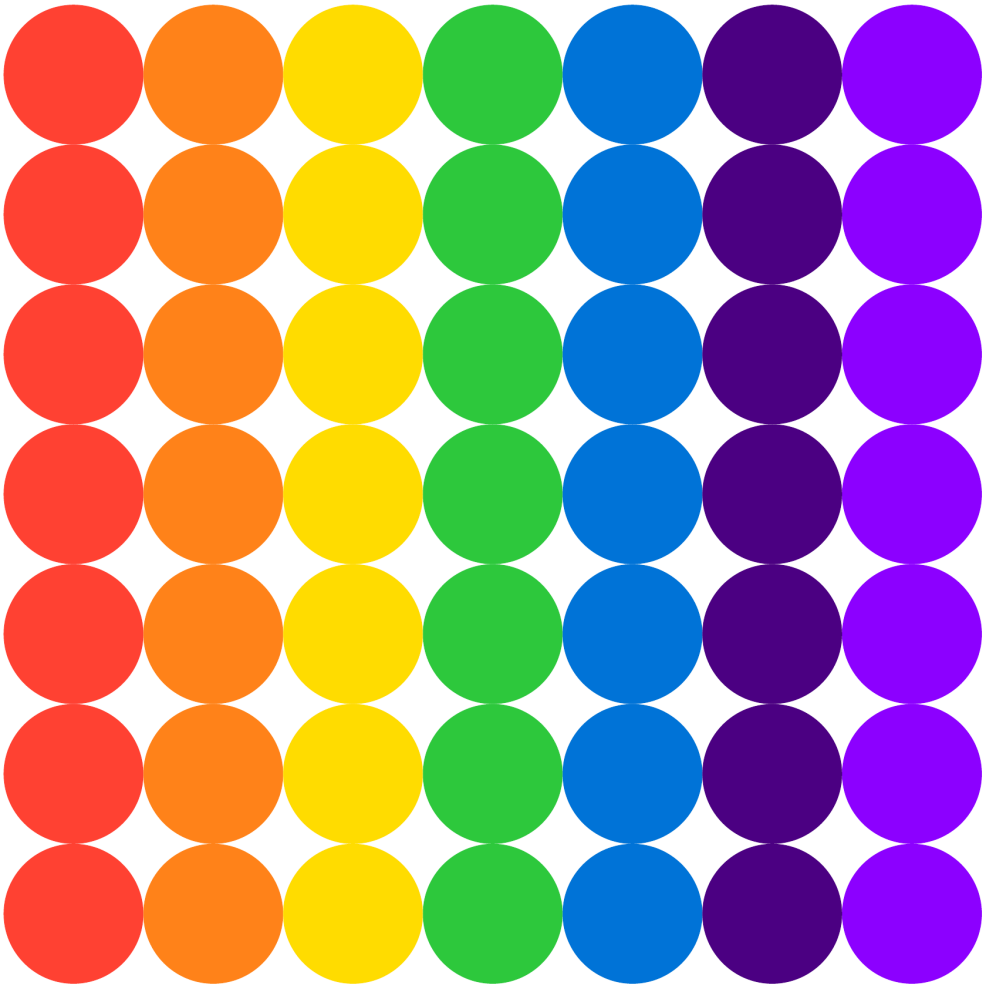
Advanced Examples

Rainbow Gradient

```
#import "@preview/typixel:0.1.2": *
#set page(width: auto, height: auto, margin: 1cm)

#pixel-map(
  "ROYGBIV\n" * 7,
  palette: (
    "R": red,
    "O": orange,
    "Y": yellow,
    "G": green,
    "B": blue,
    "I": rgb("#4B0082"),
    "V": rgb("#8F00FF")
```

```
  ),
  width: 200pt,
  shape: circle-shape
)
```

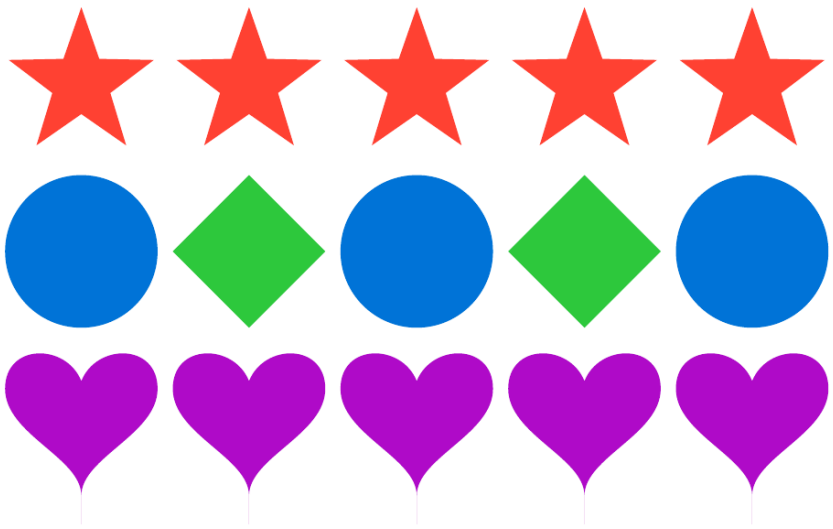


Mixed Shapes

```
#import "@preview/typixel:0.1.2": *
#set page(width: auto, height: auto, margin: 1cm)

#pixel-map(
  "
  SSSSS
  CDCDC
  HHHHH
  ",
  palette: (
    "S": red,
    "C": blue,
    "D": green,
    "H": purple
```

```
  ),
  shape: (
    "S": star-shape,
    "C": circle-shape,
    "D": diamond-shape,
    "H": heart-shape
  ),
  pixel-size: 20pt,
  gap: 2pt
)
```



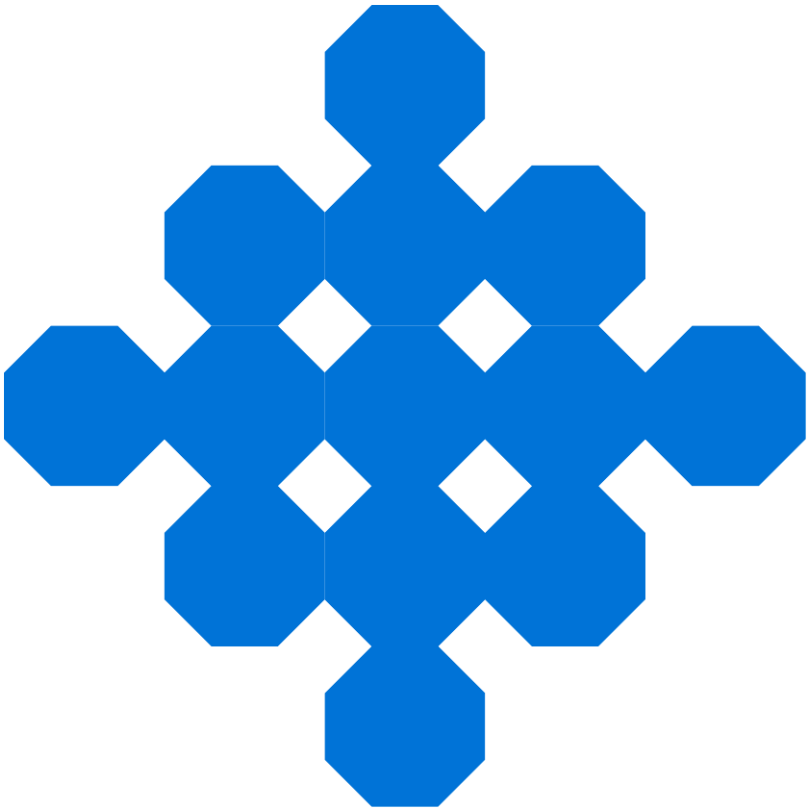
Custom Shape Definition

```
#import "@preview/typixel:0.1.2": *
#set page(width: auto, height: auto, margin: 1cm)

// 1. Define your custom shape
#let octagon-shape(width: 0pt, height: 0pt, fill: black) = {
  let c = 29.29%
  box(width: width, height: height,
    polygon(
      fill: fill,
      (c, 0%), (100% - c, 0%),
      (100%, c), (100%, 100% - c),
      (100% - c, 100%), (c, 100%),
      (0%, 100% - c), (0%, c)
    )
  )
}

// 2. Use it in pixel-map
#pixel-map(
```

```
  "
  ..A..
  .AAA.
  AAAAA
  .AAA.
  ..A..
  ",
  palette: ("A": blue, ".": none),
  pixel-size: 20pt,
  shape: octagon-shape
)
```



Limitations

When converting images using `pixel-image`, the final output height may not **exactly** match the original image's height.

This occurs because:

1. Pixel art requires an **integer number of rows**, so the height must be rounded to the nearest whole pixel.
2. Pixels are forced to be **square** (width = height).

License

This project is distributed under the MIT License. See LICENSE for details.

The example image `gahag.jpg` is based on an image from GAHAG and is licensed under CC0.

auto-jrubby

auto-jrubby is a Typst package that provides automatic Japanese morphological analysis and furigana (ruby) insertion.

It leverages a Rust-based WASM plugin to tokenize text using Lindera (a morphological analysis library) and uses the rubby package to render the furigana.

Features

- **Automatic Furigana Generation:** Automatically determines readings for Kanji based on context and renders them as ruby text.
- **Smart Okurigana Alignment:** Intelligent handling of mixed Kanji/Hiragana words (e.g., 食べる is rendered with ruby た over 食, leaving べる untouched).
- **Morphological Analysis Table:** Visualize the text structure (Part of Speech, Detailed POS, Readings, Base forms) via a formatted data table.
- **Customizable Styling:** Supports custom ruby sizing and positioning via the `rubby` package backend.
- **Flexible Script Output:** Choose between **Hiragana** (default) or **Katakana** for the ruby text.
- **High Performance:** Powered by a Rust WASM plugin using **Lindera** for fast and accurate tokenization.

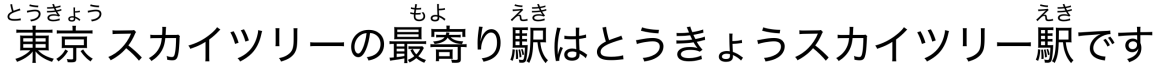
Usage

Basic Furigana

To automatically add readings to Japanese text:

```
#import "@preview/auto-jrubby:0.3.4": *
#set page(width: auto, height: auto, margin: 0.5cm)
#set text(font: "Hiragino Sans", lang: "ja")

#let sample = "東京スカイツリーの最寄り駅はとうきょうスカイツリー駅です"
#show-ruby(sample)
```



Morphological Analysis Table

To debug or display the linguistic structure of the text:

```
#import "@preview/auto-jrubby:0.3.4": *
#set page(width: auto, height: auto, margin: 0.5cm)
#set text(font: "Hiragino Sans", lang: "ja")

#let sample = "東京スカイツリーの最寄り駅はとうきょうスカイツリー駅です"
#show-analysis-table(sample)
```

表層形	品詞	品詞細１	品詞細２	品詞細３	活用形	活用型	原形	読み	発音
東京	名詞	固有名詞	地域	一般	*	*	東京	トウキョウ	トーキョー
スカイ	名詞	一般	*	*	*	*	スカイ	スカイ	スカイ
ツリー	名詞	一般	*	*	*	*	ツリー	ツリー	ツリー
の	助詞	連体化	*	*	*	*	の	ノ	ノ
最寄り駅	名詞	一般	*	*	*	*	最寄り駅	モヨリエキ	モヨリエキ
は	助詞	係助詞	*	*	*	*	は	ハ	ワ
とう	副詞	助詞類接続	*	*	*	*	とう	トウ	トウ
きょう	名詞	副詞可能	*	*	*	*	きょう	キョウ	キョー
スカイ	名詞	一般	*	*	*	*	スカイ	スカイ	スカイ
ツリー	名詞	一般	*	*	*	*	ツリー	ツリー	ツリー
駅	名詞	接尾	地域	*	*	*	駅	エキ	エキ
です	助動詞	*	*	*	特殊・デス	基本形	です	デス	デス

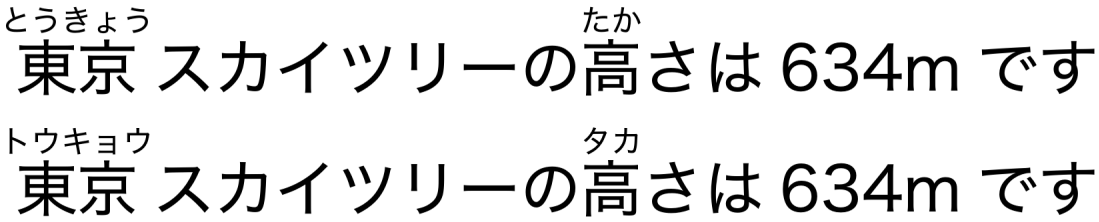
Kana Selection

You can choose to output ruby in Hiragana (default) or Katakana:

```
#import "@preview/auto-jrubby:0.3.4": *
#set page(width: auto, height: auto, margin: 0.5cm)

#let sample = "東京スカイツリーの高さは634mです"

// Default (Hiragana)
#show-ruby(sample)
// Katakana
#show-ruby(sample, kana: "katakana")
```



API Reference

`show-ruby`

Renders the input text with automatic furigana.

```
#show-ruby(
  input-text,
  size: 0.5em,
  leading: 1.5em,
  ruby-func: auto,
  user-dict: none,
  dict: "ipadic",
  kana: "hiragana"
)
```

Parameters:

- `input-text` (string): The Japanese text to analyze and render.

- `size` (length): The font size of the ruby text. Defaults to `0.5em`.
- `leading` (length): The vertical space between lines to accommodate ruby text. Defaults to `1.5em`.
- `ruby-func` (function | auto): A custom ruby function from the `rubby` package.
- If `auto`, it uses the default configuration (`get-ruby(size: size)`).
- If provided, it allows advanced customization of ruby positioning (e.g., specific `pos` or `alignment`).
- `user-dict` (string | array | none): Optional user dictionary for custom tokenization.
- If `string`: A CSV-formatted string with custom dictionary entries.
- If `array`: An array of arrays, where each inner array represents a CSV row.
- If `none`: No user dictionary is used.
- `dict` (string): The dictionary to use for tokenization. Must be one of:
- `"ipadic"` (default): Standard Japanese dictionary
- `"unidic"`: Alternative dictionary with different grammatical analysis
- `kana` (string): The script to use for the ruby text. Must be one of:
- `"hiragana"` (default): Converts dictionary readings to Hiragana.
- `"katakana"`: Uses Katakana readings.

`show-analysis-table`

Renders a table displaying the morphological breakdown of the text.

```
#show-analysis-table(
  input-text,
  user-dict: none,
  dict: "ipadic"
)
```

Parameters:

- `input-text` (string): The text to analyze.
- `user-dict` (string | array | none): Optional user dictionary for custom tokenization.
- `dict` (string): The dictionary to use. Must be one of: `"ipadic"` (default) or `"unidic"`.

Table Columns:

The columns displayed depend on the selected `dict`.

If `dict: "ipadic"` (10 columns):

1. **Surface Form (表層形):** The word as it appears in the text.
2. **Part of Speech (品詞):** Grammatical category (Noun, Verb, etc.).
3. **POS Subcategory 1 (品詞細分類 1)**
4. **POS Subcategory 2 (品詞細分類 2)**
5. **POS Subcategory 3 (品詞細分類 3)**
6. **Conjugation Form (活用形)**
7. **Conjugation Type (活用型)**
8. **Base Form (原形):** The dictionary form of the word.
9. **Reading (読み):** Katakana reading.
10. **Pronunciation (発音)**

If `dict: "unidic"` (18 columns):

1. **Surface Form (表層形)**
2. **POS Major (品詞大分類)**
3. **POS Medium (品詞中分類)**
4. **POS Minor (品詞小分類)**
5. **POS Fine (品詞細分類)**
6. **Conjugation Type (活用型)**
7. **Conjugation Form (活用形)**
8. **Lexeme Reading (語彙素読み)**
9. **Lexeme (語彙素)**
10. **Orthographic Surface (書字形出現形)**

11. **Phonological Surface (発音形出現形)**
12. **Orthographic Base (書字形基本形)**
13. **Phonological Base (発音形基本形)**
14. **Word Type (語種)**
15. **Initial Mutation Type (語頭変化型)**
16. **Initial Mutation Form (語頭変化形)**
17. **Final Mutation Type (語末変化型)**
18. **Final Mutation Form (語末変化形)**

`tokenize`

Low-level function that returns the raw JSON data from the WASM plugin. Useful if you want to process the analysis data manually.

```
#tokenize(
  input-text,
  user-dict: none,
  dict: "ipadic"
)
```

Parameters:

- `input-text` (string): The text to tokenize.
- `user-dict` (string | array | none): Optional user dictionary for custom tokenization.
- `dict` (string): The dictionary to use. Must be one of: `"ipadic"` or `"unidic"`.

Returns: An array of dictionaries containing:

- `surface` (string): The surface form of the token.
- `details` (array of strings): The raw detailed information for the token. The content and length depend on the dictionary used (e.g., POS, conjugation, reading, etc.).
- `ruby_segments` (array of dictionaries): A pre-calculated list of segments for furigana, where each item has `text` and `ruby` fields.

User Dictionary Format

The user dictionary allows you to define custom word segmentation and readings. It uses a simple CSV format with three columns:

```
<surface>,<part_of_speech>,<reading>
```

- `surface`: The word as it appears in text
- `part_of_speech`: Custom part-of-speech label (e.g., "カスタム名詞")
- `reading`: Katakana reading for the word

Usage Examples:

Method 1: Inline string

```
#let sample = "東京タワーの最寄駅は赤羽橋駅です"
#let user-dict-str = "赤羽橋駅,カスタム名詞,アカバネバシエキ"

#show-ruby(sample)
#show-ruby(sample, user-dict: user-dict-str)
```

Method 2: Array of arrays

```
#let sample = "東京タワーの最寄駅は赤羽橋駅です"
#let user-dict-array = (
  ("赤羽橋駅", "カスタム名詞", "アカバネバシエキ")
)

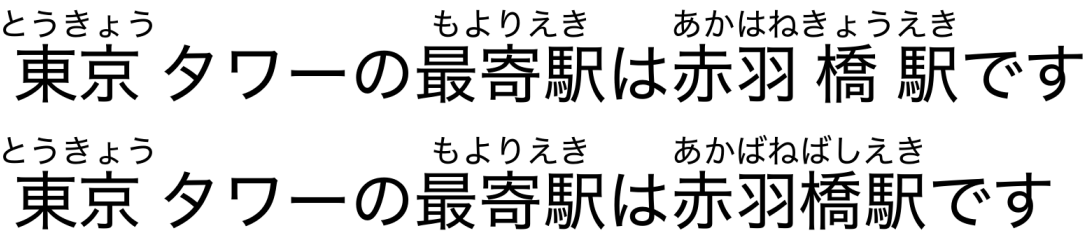
#show-ruby(sample)
#show-ruby(sample, user-dict: user-dict-array)
```

Method 3: Load from CSV file

```
$ cat user_dict.csv
赤羽橋駅,カスタム名詞,アカバネバシエキ
```

```
#let sample = "東京タワーの最寄駅は赤羽橋駅です"
#let user-dict-from-file = csv("user_dict.csv")

#show-ruby(sample)
#show-ruby(sample, user-dict: user-dict-from-file)
```



Under the Hood

This package uses **Lindera** (a Rust port of Kuromoji) with two available dictionary options:

- **IPADIC**: Standard Japanese morphological dictionary
- **UniDic**: Alternative dictionary with different part-of-speech classifications

The processing workflow:

1. The text is passed from Typst to the Rust WASM plugin.
2. Lindera tokenizes the text using the specified dictionary and retrieves readings.
3. A custom algorithm aligns the readings with the surface form to separate okurigana (kana endings of verbs/adjectives) from the kanji stems.
4. The structured data is returned to Typst and rendered using the `rubby` package for furigana display.

Optional: Enabling IPADIC-NEologd

IPADIC-NEologd Support

IPADIC-NEologd (an extended dictionary with contemporary terms and named entities) has been removed from the default distribution due to its large file size. However, you can manually enable it if needed:

1. Navigate to `./wasm-plugins/ipadic-neologd` and build the WASM module:

   ```
   cargo build --target wasm32-unknown-unknown --release
   ```

   1. Copy the built WASM file to the package directory:

   ```
   cp ./target/wasm32-unknown-unknown/release/ipadic_neologd.wasm ../../package/
   ipadic_neologd.wasm
   ```

   1. Navigate to `./package` and update `./package/lib.typ` to allow the `ipadic-neologd` option. Change:

   ```
   if dict not in ("ipadic", "unidic") {
     panic("dict must be one of: ipadic, unidic")
   }
   ```

   to:

   ```
   if dict not in ("ipadic", "ipadic-neologd", "unidic") {
     panic("dict must be one of: ipadic, ipadic-neologd, unidic")
   }
   ```

   1. Install the package locally:

   ```
   just install
   ```

1. Import and use with `@local`:

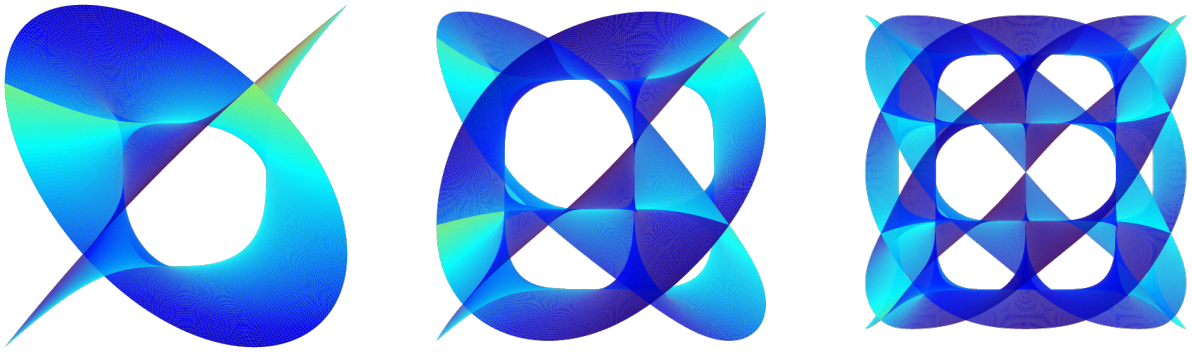
```
#import "@local/auto-jrubby:0.3.3": *
#let sample = "東京スカイツリーの最寄り駅はとうきょうスカイツリー駅です"
#show-ruby(sample, dict: "ipadic-neologd")
```

License

This project is distributed under the AGPL-3.0-or-later License. See LICENSE for details.

CaleTZ

CaleTZ is a Typst package for visualizing **Calabi-Yau manifolds** using **CeTZ** 3D drawing primitives. It generates colorful, smooth 3D surfaces with adjustable parameters, making it easy to explore complex geometries directly in Typst.



Installation

Using `just`

First, install `just` if you don't have it yet:

```
cargo install just
```

Then, install the CaleTZ package:

```
git clone https://github.com/rice8y/caletz.git
cd caletz
just install
```

This installs the package locally to Typst's package directory.

Using `typkg`

First, install `typkg` if you don't have it yet:

```
cargo install --git https://github.com/rice8y/typkg.git
```

Then, install the CaleTZ package:

```
typkg install https://github.com/rice8y/caletz.git
```

This will also install the package locally for Typst usage.

Usage Example

```
#import "@local/caletz:0.2.0": calabi-yau

#set page(width: auto, height: auto, margin: 1cm)
#calabi-yau(
  power: 3,
  angle: 0.4,
  subdivisions: 20
)
```

Parameters

- `power`: Degree of the manifold (e.g., 3). Must be a positive integer.
- `angle`: Adjusts the combination of z-coordinates; smaller values flatten the surface.
- `subdivisions`: Controls mesh density; higher values give smoother surfaces.
- `colormap`: Optional, default `"jet"`. Can be `"viridis"`, `"plasma"`, `"cool"`, or `"hot"`.
- `scale-factor`: Optional, default `3.0`. Scales the entire mesh.
- `rotation`: Optional, default `(30deg, 45deg, 0deg)`. Rotates the 3D view.

> For best results, use a high `subdivisions` value, but note it increases computation.

License

This project is distributed under the MIT License. See LICENSE.

escansel

escancel is a simple and convenient CLI tool to cancel your SLURM jobs — essentially an **easy scancel**. It lists your active jobs and lets you interactively select which ones to cancel, making `scancel` faster and more user-friendly.

Installation

You can install this CLI tool using `uv` in two different ways:

A. Install directly from GitHub (recommended)

```
uv tool install git+https://github.com/rice8y/escancel.git
```

This will fetch and install the latest version directly from the repository.

B. Install from a local clone

1. Clone the repository:

```
git clone https://github.com/rice8y/escancel.git
```

1. Move into the project directory:

```
cd escancel
```

1. Install the package in editable mode using uv tool:

```
uv tool install -e .
```

This is useful if you plan to modify the code locally.

Upgrade

To upgrade escancel to the latest version:

```
uv tool upgrade escancel
```

This will fetch the latest version from the original source and update your installation.

License

CeTZuron

Installation

1. Clone the repository

```
$ git clone https://github.com/rice8y/cetzuron.git
$ cd cetzuron
```

2. Install the package locally via `justfile`, `.sh`, or `.bat`

2-1. Using `justfile`

```
$ just install
```

Example on WSL2 (Ubuntu)

```
$ just install
Package cetzuron version 0.1.0 has been installed to /home/rice8/.local/share/typst/packages/
local/cetzuron/0.1.0
```

2-2. Using `.sh`

```
$ chmod +x install.sh
$ ./install.sh
```

Example on WSL2 (Ubuntu)

```
$ ./install.sh
Package cetzuron version 0.1.0 has been installed to /home/rice8/.local/share/typst/packages/
local/cetzuron/0.1.0
```

2-3. Using `.bat`

```
> install.bat
```

Example on Windows (cmd)

```
> install.bat
C:install.sh
C:justfile
C:README.md
C:typst.toml
C:docs\ae\sample_ae.pdf
C:docs\ae\sample_ae.png
C:docs\ae\sample_ae.typ
C:docs\fcnn\sample_fcnn.pdf
C:docs\fcnn\sample_fcnn.png
C:docs\fcnn\sample_fcnn.typ
C:docs\lstm\sample_lstm.pdf
C:docs\lstm\sample_lstm.png
C:docs\lstm\sample_lstm.typ
C:docs\rnn\sample_rnn.pdf
C:docs\rnn\sample_rnn.png
C:docs\rnn\sample_rnn.typ
C:src\ae.typ
C:src\fcnn.typ
C:src\lib.typ
C:src\lstm.typ
C:src\requirements.typ
C:src\rnn.typ
23 File(s) copied
Package cetzuron version 0.1.0 has been installed to C:
\Users\yoneyama\AppData\Roaming\typst\packages\local\cetzuron\0.1.0
```

Usage

Import the package using `#import`.

```
#import "@local/cetzuron:0.1.0"
```

Fully Connected Neural Network `#fcnn`

Parameters

```
fcnn(
    inputNodes: int,
    middleNodes: int,
    outputNodes: int,
    middleLayers: int,
    label: bool,
) -> content
```

inputNodes: Number of nodes in the input layer **middleNodes:** Number of nodes in hidden layers **outputNodes:** Number of nodes in the output layer **middleLayers:** Number of hidden layers (default: 3) **label:** Whether to show labels (default: true)

Example usage of `#fcnn`

```
#import "@local/cetzuron:0.1.0": *
#set page(width: auto, height: auto)
#set text(lang: "ja", font: "TeX Gyre Termes", size: 10pt)
#show regex("[\p{scx:Han}\p{scx:Hira}\p{scx:Kana}]"): set text(lang: "ja", font: "Harano Aji
Mincho", size: 10pt)

#figure(
  fcnn(3, 4, 3),
  caption: [With Labels]
)
#figure(
  fcnn(5, 4, 3, middleLayers: 1, label: false),
  caption: [Without Labels]
)
```

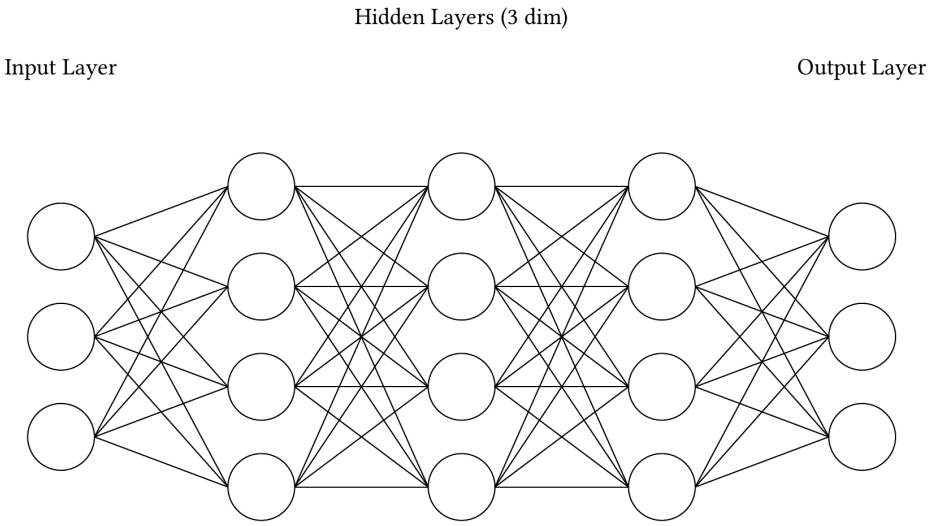


Figure 1: With Labels

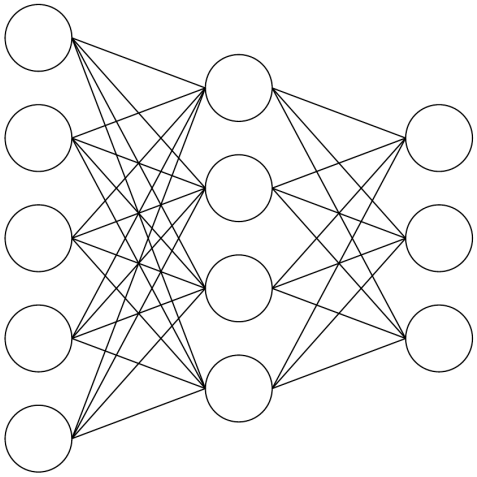


Figure 2: Without Labels

Recurrent Neural Network `#rnn`

Parameters

```
rnn(
    inputNodes: int,
    middleNodes: int,
    outputNodes: int,
    middleLayers: int,
    label: bool,
) -> content
```

inputNodes: Number of nodes in the input layer **middleNodes:** Number of nodes in hidden layers **outputNodes:** Number of nodes in the output layer **middleLayers:** Number of hidden layers (default: 3) **label:** Whether to show labels (default: true)

Example usage of `#rnn`

```
#import "@local/cetzuron:0.1.0": *
#set page(width: auto, height: auto)
#set text(lang: "ja", font: "TeX Gyre Termes", size: 10pt)
#show regex("[\p{scx:Han}\p{scx:Hira}\p{scx:Kana}]"): set text(lang: "ja", font: "Harano Aji
Mincho", size: 10pt)

#figure(
  rnn(3, 4, 3),
  caption: [With Labels]
)
#figure(
  rnn(5, 4, 3, middleLayers: 1, label: false),
  caption: [Without Labels]
)
```

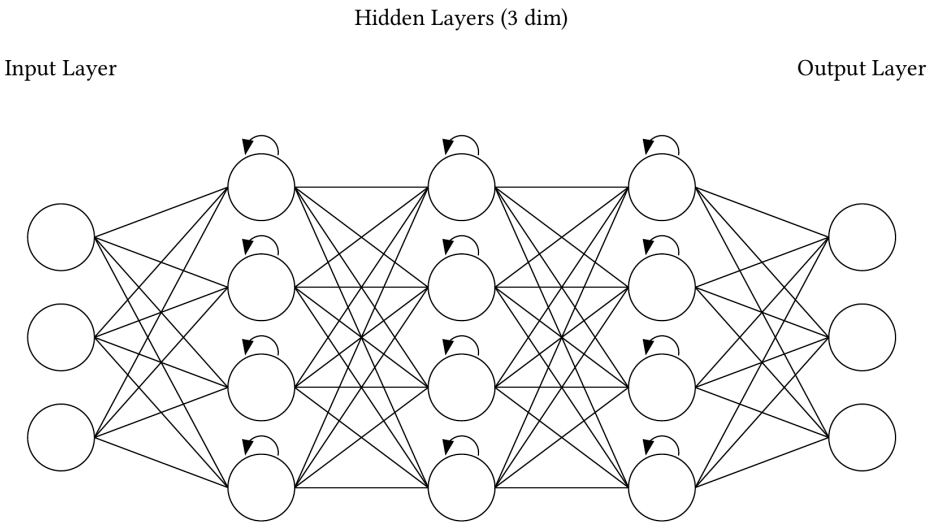


Figure 1: With Labels

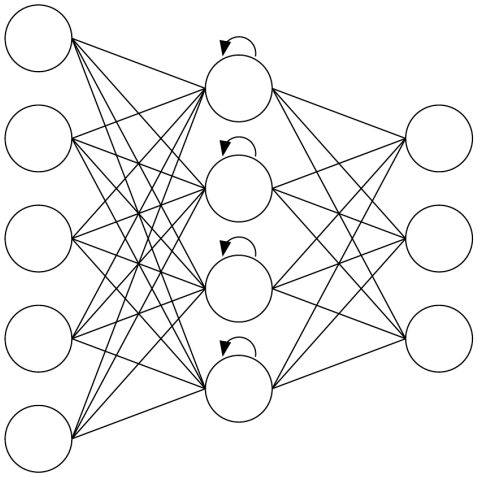


Figure 2: Without Labels

Long Short-Term Memory `#lstm`

Parameters

```
lstm(
    inputNodes: int,
    middleNodes: int,
    outputNodes: int,
    middleLayers: int,
    label: bool,
) -> content
```

inputNodes: Number of nodes in the input layer **middleNodes:** Number of nodes in hidden layers **outputNodes:** Number of nodes in the output layer **middleLayers:** Number of hidden layers (default: 3) **label:** Whether to show labels (default: true)

Example usage of `#lstm`

```
#import "@local/cetzuron:0.1.0": *
#set page(width: auto, height: auto)
#set text(lang: "ja", font: "TeX Gyre Termes", size: 10pt)
#show regex("[\p{scx:Han}\p{scx:Hira}\p{scx:Kana}]"): set text(lang: "ja", font: "Harano Aji
Mincho", size: 10pt)

#figure(
  lstm(3, 4, 3),
  caption: [With Labels]
)
#figure(
  lstm(5, 4, 3, middleLayers: 1, label: false),
  caption: [Without Labels]
)
```

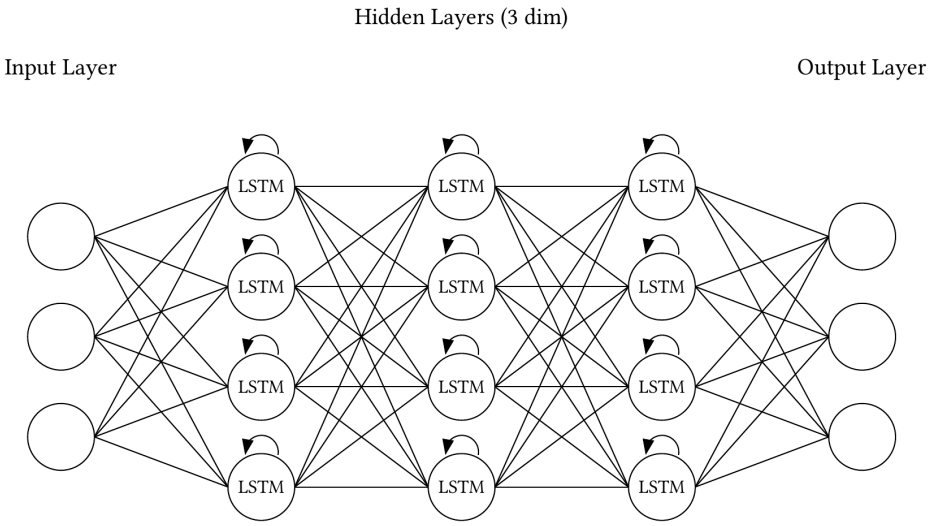


Figure 1: With Labels

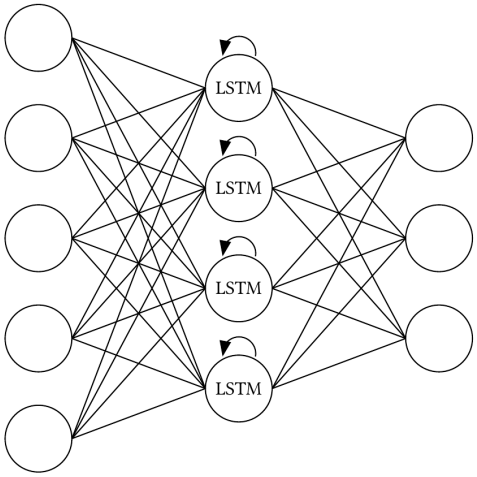


Figure 2: Without Labels

Autoencoder #ae

Parameters

```
ae(
    inputNodes: int,
    middleNodes: int,
    style: string,
    label: bool,
) -> content
```

inputNodes: Number of nodes in input/output layers **middleNodes:** Number of nodes in hidden layer **style:** Shape of hidden layer [“short”, “full”] (default: “short”) **label:** Whether to show labels (default: true)

Example usage of #ae

```
#import "@local/cetzuron:0.1.0": *
#set page(width: auto, height: auto)
#set text(lang: "ja", font: "TeX Gyre Termes", size: 10pt)
#show regex("[\p{scx:Han}\p{scx:Hira}\p{scx:Kana}]"): set text(lang: "ja", font: "Harano Aji
Mincho", size: 10pt)

#figure(
  ae(5, 3),
  caption: [With Labels (short)]
)
#figure(
  ae(5, 3, style: "full"),
  caption: [With Labels (full)]
)
#figure(
  ae(4, 2, style: "full", label: false),
  caption: [Without Labels (full)]
)
```

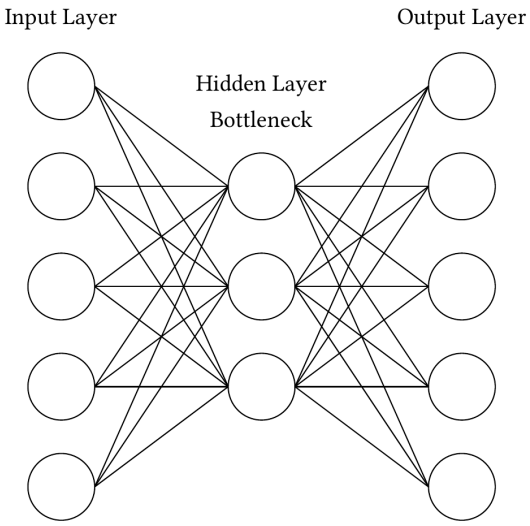


Figure 1: With Labels (short)

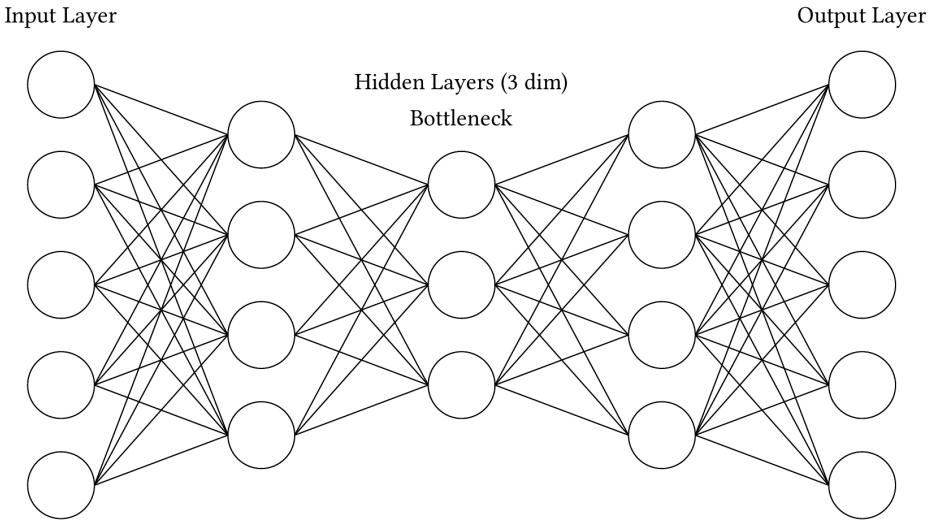


Figure 2: With Labels (full)

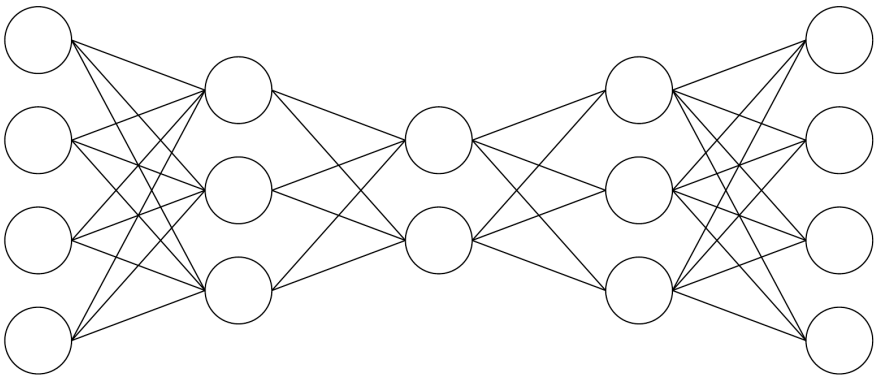
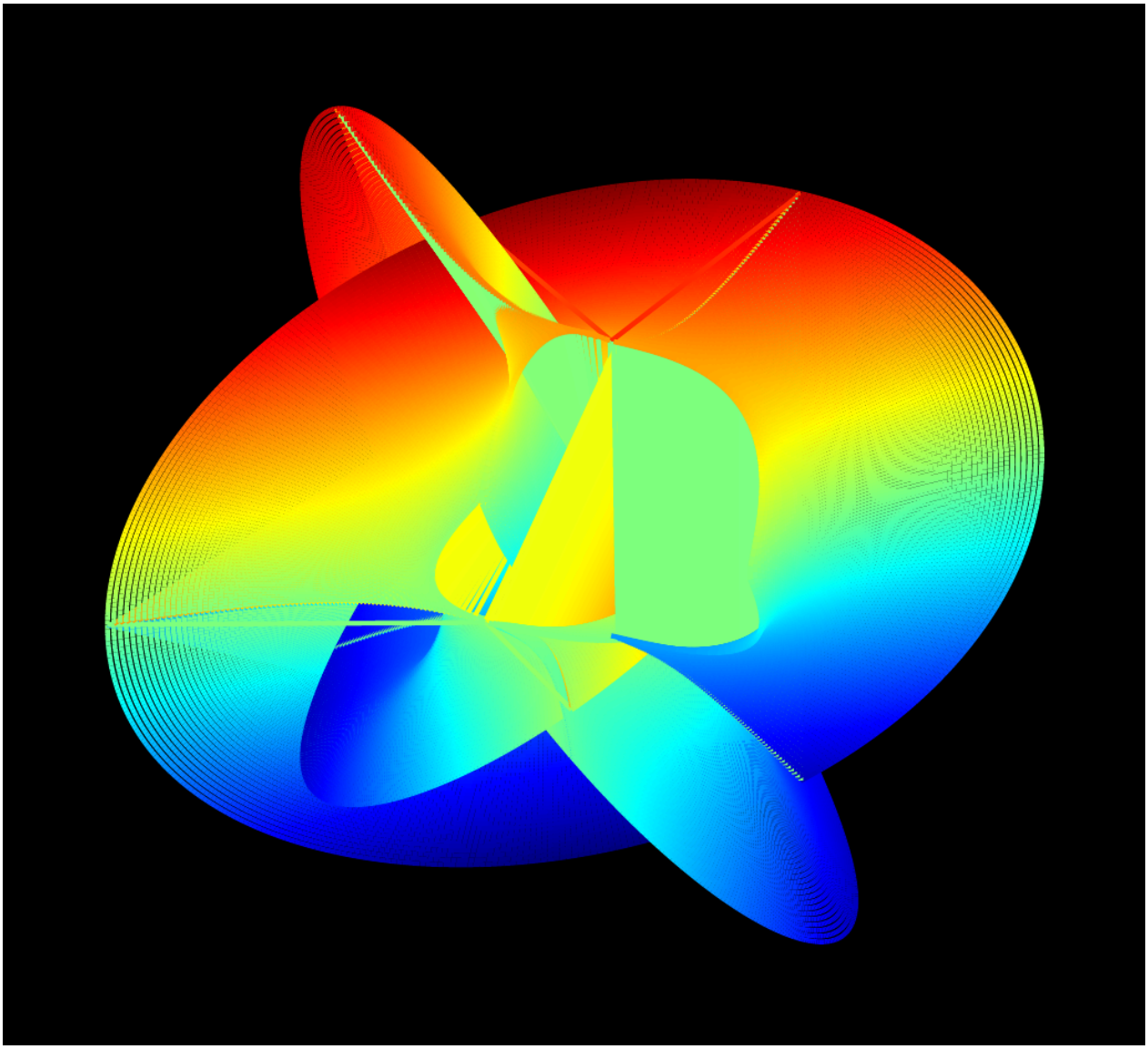


Figure 3: Without Labels (full)

CY3d

This LaTeX package provides a command `\CalabiYau`, which can display Calabi-Yau manifold. This package utilizes PGFplots for drawing Calabi-Yau manifold.



Requirements

This package requires `pgfplots` and `luacode`. Additionally, since Lua is used for coordinate calculations, LuaLaTeX must be used.

Installation

To install this package, you can clone the repository from GitHub:

```
git clone https://github.com/rice8y/cy3d
```

Windows

On Windows, you can simply run the following command to install the package:

```
install
```

Linux/macOS

For Linux and macOS, you can use the provided shell script to install the package:

```
./install.sh
```

Usage

```
\CalabiYau[colormap]{power}{angle}{mesh size}
```

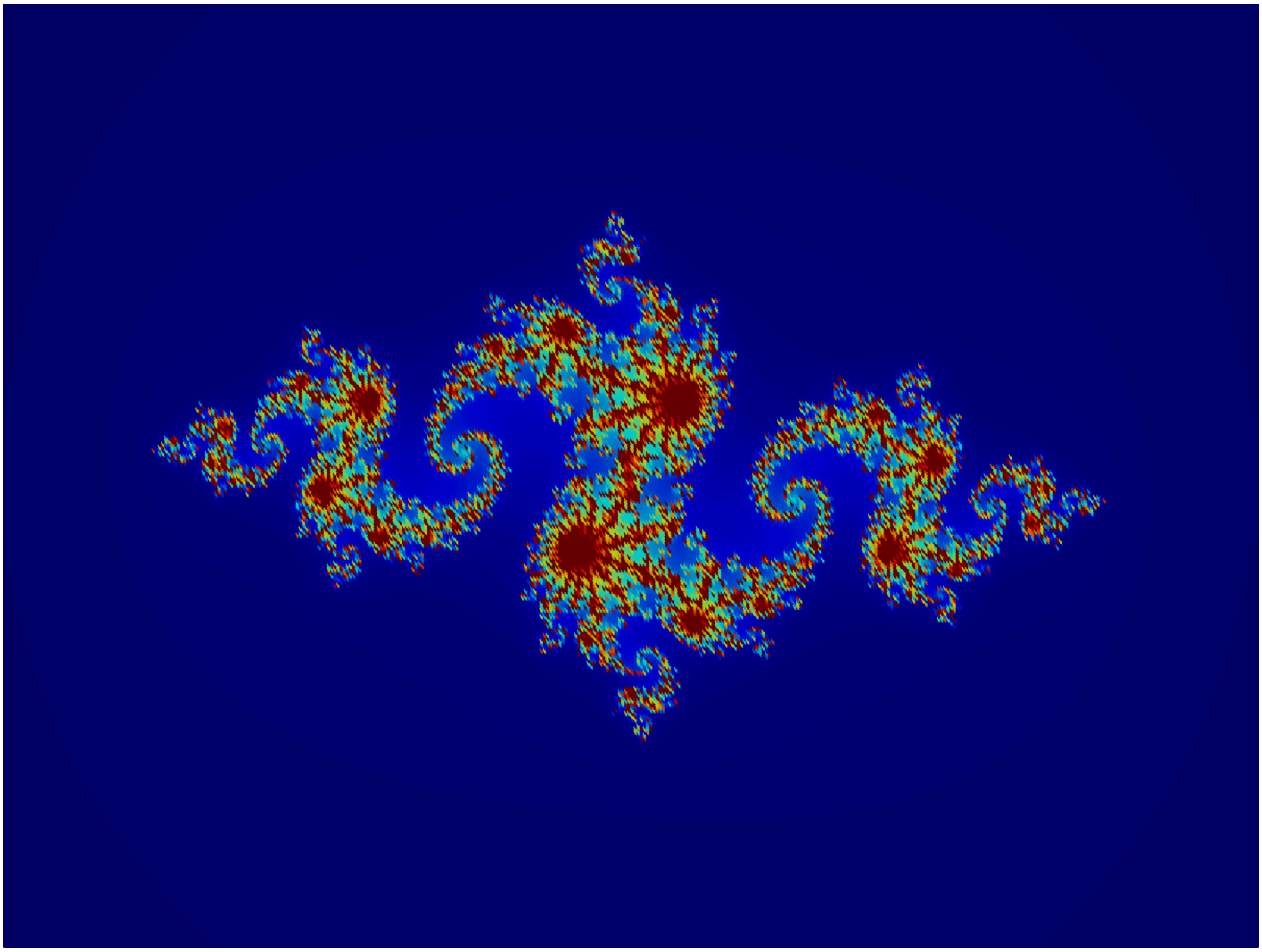
Parameters:

- power: The degree of the Calabi-Yau equation $z_1^n + z_2^n = 1$.
- angle: The angle parameter to adjust the rotation or perspective of the surface.
- mesh size: Defines the resolution of the mesh for plotting the surface.
- colormap (option): Specifies the color scheme used for rendering the surface. This is based on the TikZ colormap. The default colormap is jet.

License

FracTeX

This package provides commands to visualize various types of fractals using PGFPlots and LuaTeX.



Supported Fractals:

- Mandelbrot Set
- Julia Set
- Barnsley Fern (IFS)
- Burning Ship Fractal
- Newton Fractal
- Phoenix Fractal
- Tricorn Fractal
- Buffalo Fractal
- Sierpiński Triangle
- Lyapunov Fractal
- Magnet Fractal
- Multibrot Set
- Gingerbreadman Map

Requirements

This package requires `pgfplots`, `luacode` and `xkeyva`. Additionally, since Lua is used for coordinate calculations, LuaLaTeX must be used.

Installation

To install this package, you can clone the repository from GitHub:

```
git clone https://github.com/rice8y/FracTeX.git
```

Windows

On Windows, you can simply run the following command to install the package:

```
install
```

Linux/macOS

For Linux and macOS, you can use the provided shell script to install the package:

```
./install.sh
```

Usage

Mandelbrot Set

```
\MandelbrotSet
% options:
\MandelbrotSet[xmin=-2,xmax=1,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/1)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Julia Set

```
\JuliaSet
% options:
\JuliaSet[xmin=-2,xmax=2,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,c_re=-0.8,c_im=0.156,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/2)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- c_re/c_im: Complex constant components (default: −0.8/0.156)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Barnsley Fern

```
\BarnsleyFern
% options: \BarnsleyFern[num_points=50000,color=green]
```

Parameters:

- num_points: Number of generated points (default: 50000)
- color: Base color (default: green)

Burning Ship Fractal

```
\BurningShipFractal
% options:
\BurningShipFractal[xmin=-2,xmax=1.5,ymin=-2,ymax=1,dx=0.02,dy=0.02,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/1.5)
- ymin/ymax: Y-axis range (default: −2/1)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Newton Fractal

```
\NewtonFractal
% options:
\NewtonFractal[xmin=-1.5,xmax=1.5,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,max_iter=20,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −1.5/1.5)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 20)
- cmap: Color scheme (default: jet)

Phoenix Fractal

```
\PhoenixFractal
% options:
\PhoenixFractal[xmin=-1.5,xmax=1.5,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,P=0.3,max_iter=50,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −1.5/1.5)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- P: Feedback parameter (default: 0.3)
- max_iter: Iteration limit (default: 50)
- cmap: Color scheme (default: jet)

Tricorn Fractal

```
\TricornFractal
% options:
\TricornFractal[xmin=-2,xmax=1,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/1)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Buffalo Fractal

```
\BuffaloFractal
% options:
\BuffaloFractal[xmin=-2,xmax=1,ymin=-1.5,ymax=1.5,dx=0.02,dy=0.02,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/1)
- ymin/ymax: Y-axis range (default: −1.5/1.5)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Sierpinski Triangle

```
\SierpinskiTriangle
% options: \SierpinskiTriangle[num_points=50000,color=blue]
```

Parameters:

- num_points: Points count (default: 50000)
- color: Base color (default: blue)

Lyapunov Fractal

```
\LyapunovFractal
% options: \LyapunovFractal[amin=2.4,amax=3.6,bmin=2.4,bmax=3.6,da=0.01,db=0.01,
max_iter=100,cmap=hot]
```

Parameters:

- amin/amax: Parameter `a` range (default: 2.4/3.6)
- bmin/bmax: Parameter `b` range (default: 2.4/3.6)
- da/db: Resolution parameters (default: 0.01/0.01)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: hot)

Magnet Fractal

```
\MagnetFractal
% options: \MagnetFractal[xmin=-2,xmax=2,ymin=-2,ymax=2,dx=0.02,dy=0.02,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/2)
- ymin/ymax: Y-axis range (default: −2/2)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Multibrot Set

```
\MultibrotSet
% options: \MultibrotSet[xmin=-2,xmax=2,ymin=-2,ymax=2,dx=0.02,dy=0.02,d=3,max_iter=100,cmap=jet]
```

Parameters:

- xmin/xmax: X-axis range (default: −2/2)
- ymin/ymax: Y-axis range (default: −2/2)
- dx/dy: Resolution parameters (default: 0.02/0.02)
- d: Power parameter (default: 3)
- max_iter: Iteration limit (default: 100)
- cmap: Color scheme (default: jet)

Gingerbreadman Map

```
\GingerbreadmanMap
% options: \GingerbreadmanMap[num_points=50000, color=red]
```

Parameters:

- num_points: Points count (default: 50000)
- color: Base color (default: red)

License

SDetails

SDetails is a Python-based CLI utility designed to improve the visibility of SLURM cluster resources. It provides a user-friendly terminal dashboard showing per-node CPU, memory, and GPU utilization with optional color highlighting, sorting, filtering, and JSON export. Ideal for HPC users and admins who need real-time or static insights into node status.

Installation

You can install this CLI tool using `uv` or `pip` in a few different ways:

A. Install directly from GitHub (recommended)

Using `uv`:

```
uv tool install git+https://github.com/rice8y/sdetails.git
```

Or using `pip`:

```
pip install git+https://github.com/rice8y/sdetails.git
```

This will fetch and install the latest version directly from the repository.

B. Install from a local clone

1. Clone the repository:

```
git clone https://github.com/rice8y/sdetails.git
```

1. Move into the project directory:

```
cd sdetails
```

1. Install the package in editable mode using `uv tool` (or `pip`):

```
uv tool install -e .
# or
pip install -e .
```

This is useful if you plan to modify the code locally.

Usage

SDetails is a CLI tool that enhances SLURM cluster node monitoring with a colorized summary and detailed views.

Example

```
sdetails
```

This command fetches and displays a summary and detailed table of all SLURM nodes.

Options

- `-p`, `--partition <PART>`: Filter by a specific partition
- `-s`, `--sort <FIELD>`: Sort by `nodename`, `partition`, `state`, or `cpu` (default: `nodename`)
- `--no-color`: Disable color output
- `--no-summary`: Skip the cluster summary section
- `--export <FILE>`: Export the current view to a JSON file
- `--watch <SECONDS>`: Refresh the view every N seconds (Ctrl+C to exit)

Example with options

```
sdetails -p gpu --sort cpu --watch 10 --export status.json
```

This will display only the `gpu` partition, sort by CPU availability, auto-refresh every 10 seconds, and export to `status.json`.

License

This project is distributed under the MIT License. See LICENSE.

SSJ

SSJ (Scontrol Show Job) is a Python-based CLI utility that enhances the readability and interactivity of SLURM job inspection. It provides a user-friendly terminal interface to display SLURM job details with optional filtering, formatting, file inspection, and JSON export.

Ideal for HPC users who frequently monitor job metadata, logs, or scripts via `scontrol`.

Installation

You can install this CLI tool using `uv` in two different ways:

A. Install directly from GitHub (recommended)

```
uv tool install git+https://github.com/rice8y/ssj.git
```

This will fetch and install the latest version directly from the repository.

B. Install from a local clone

1. Clone the repository:

```
git clone https://github.com/rice8y/ssj.git
```

1. Move into the project directory:

```
cd ssj
```

1. Install the package in editable mode using uv tool:

```
uv tool install -e .
```

This is useful if you plan to modify the code locally.

Usage

SSJ fetches detailed SLURM job metadata using scontrol show job `<jobid>`, and presents the data in rich tables or JSON format. You can also inspect the job's associated script, stdout/stderr logs, or working directory.

Example

```
ssj 123456
```

This command displays a formatted table with all metadata for job `123456`.

Options

- `-f`, `--fields <KEYS>`: Show only specified fields (partial, case-insensitive match allowed)
- `-g`, `--grep <PATTERN>`: Filter fields using a regex pattern
- `-j`, `--json`: Output raw JSON instead of a table
- `--script`: Show job script contents
- `--stdout`: Show stdout log contents
- `--stderr`: Show stderr log contents
- `--files`: List job-related file paths and their existence
- `--lines <N>`: Show only first N lines of a file (used with `--script`, `--stdout`, `--stderr`)
- `--tail`: Show last N lines instead of first N (requires `--lines`)

Example with options

```
ssj 123456 --fields starttime --grep time
```

Displays job 123456 fields related to timing.

```
ssj 123456 --stdout --lines 20 --tail
```

Shows the last 20 lines of the job's stdout log.

```
ssj 123456 --files
```

Lists script, stdout, stderr, and working directory paths associated with the job, and whether they exist.

License

This project is distributed under the MIT License. See LICENSE.