Dip-Strike Tools

Release 0.2.1

Francesco Pennica, Giuseppe Cosentino

Oct 28, 2025

USER GUIDE

Description: Tools for digitizing, managing, and analyzing plane orientation data (dip and strike) of planar geologic features. Dip-Strike Tools is a QGIS plugin that aims to provide a set of tools for digitizing, managing, and analyzing plane orientation or attitude data (dip and strike) of planar geologic features. It currently provides basic tools to streamline the workflow for geologists working with structural geology datasets, enabling efficient dip and strike data capture and management within QGIS.
Author and contributors: Francesco Pennica, Giuseppe Cosentino
Plugin version: 0.2.1
QGIS minimum version: 3.40
QGIS maximum version: 4.99
Source code: https://github.com/fpennica/dip-strike-tools
Last documentation update: 28 ottobre 2025

Overview

Dip-Strike Tools is a QGIS plugin that aims to provide a set of tools for digitizing, managing, and analyzing plane orientation or attitude data (dip and strike) of planar geologic features. It currently provides basic tools to streamline the workflow for geologists working with structural geology datasets, enabling efficient dip and strike data capture and management within QGIS.

Key Features

- **Interactive Data Collection**: Point-and-click interface for recording dip/strike measurements directly on the map
- **True North Correction**: Automatic adjustment for grid convergence
- **Layer Management**: Automated creation and configuration of specialized geological data layers
- **Field Calculations**: Batch calculation tools for converting between strike and dip azimuths
- **Customizable Geology Types**: Configurable geological type classifications (strata, foliation, faults, etc.)
- **Multi-format Support**: Works with shapefiles, GeoPackages, and other vector formats

CHAPTER

ONE

PLUGIN OVERVIEW

Dip-Strike Tools is a QGIS plugin that aims to provide a set of tools for digitizing, managing, and analyzing plane orientation or attitude data (dip and strike) of planar geologic features. It currently provides basic tools to streamline the workflow for geologists working with structural geology datasets, enabling efficient dip and strike data capture and management within QGIS.

1.1 What are Strike and Dip?

Strike and **dip** are fundamental measurements in structural geology that describe the orientation of planar geological features such as rock layers, fault planes, and fractures. The *dip* is the angle the slope descends, while the *direction* of descent can be represented by either *strike direction* or *dip direction*.

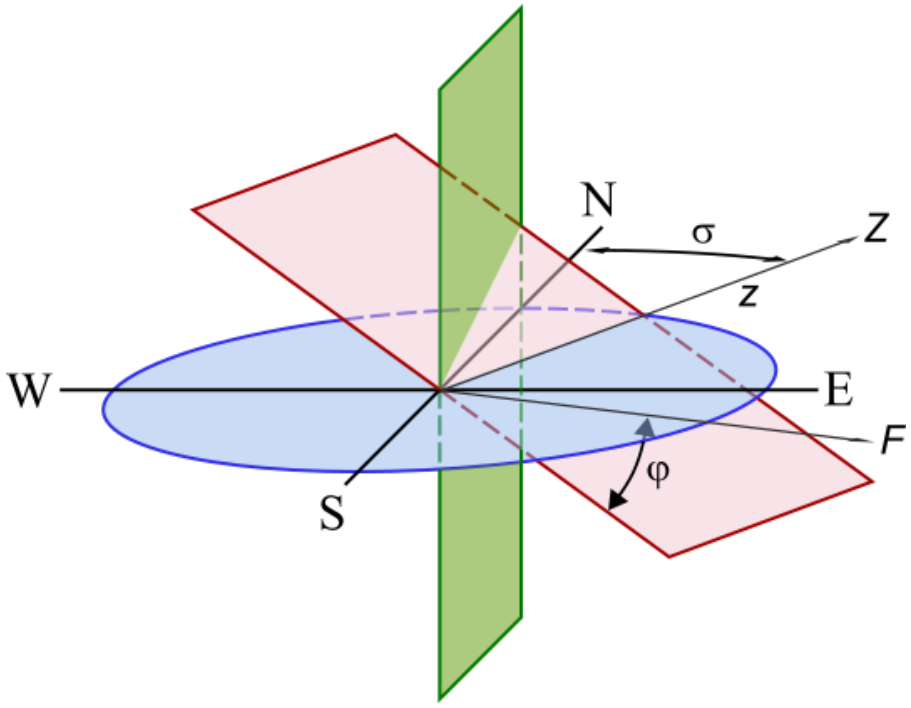


Fig. 1: Schematic depiction of “strike” and “dip” in structural geology. Z: strike line of the red plane, σ: strike angle, F: dip direction, φ: dip angle. (https://en.wikipedia.org/wiki/File:Streichbild.svg)

Note

Key Definitions:

- **Strike** or **Strike line**: a line representing the intersection of a planar feature with a horizontal plane
- **Strike Azimuth** or **Strike direction**: The compass direction of the strike line (0-360°)
- **Dip** or **Dip angle**: The angle of inclination of the planar feature from horizontal (0-90°)
- **Dip Azimuth** or **Dip direction**: The compass direction of the steepest descent down the plane (perpendicular to the strike line)

On geological maps, strike and dip can be represented by a T symbol with a label that gives the dip angle, in degrees, below horizontal. The longer line represents strike, and is in the same orientation as the strike angle. Dip is represented by the shorter line, which is perpendicular to the strike line in the downhill direction. Strike and dip information recorded on a map can be used to reconstruct various structures, determine the orientation of subsurface features, or detect the presence of anticline or syncline folds.

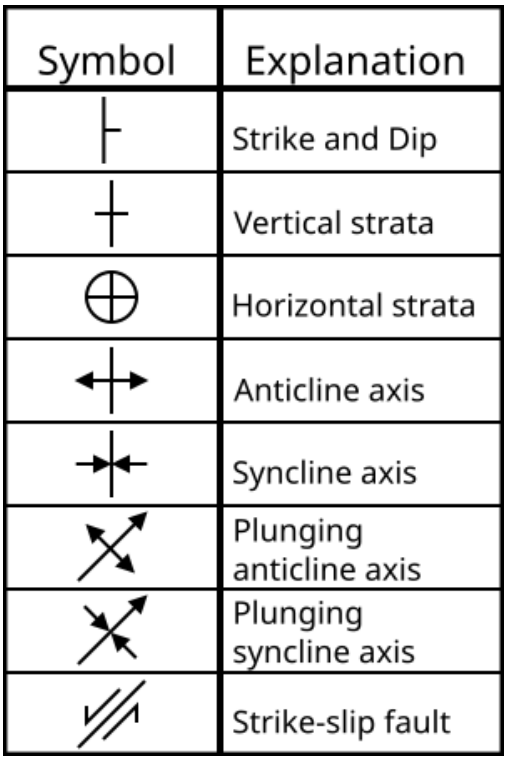


Fig. 2: Common map symbols (https://en.wikipedia.org/wiki/File:Guide_to_common_map_symbols.svg)

1.1.1 Dip and strike data collection and representation

In GIS-based geological mapping, strike and dip are typically represented as point features with associated numerical attributes for strike and dip. These points are then styled using appropriate symbology to visually communicate the orientation of planar features on the map.

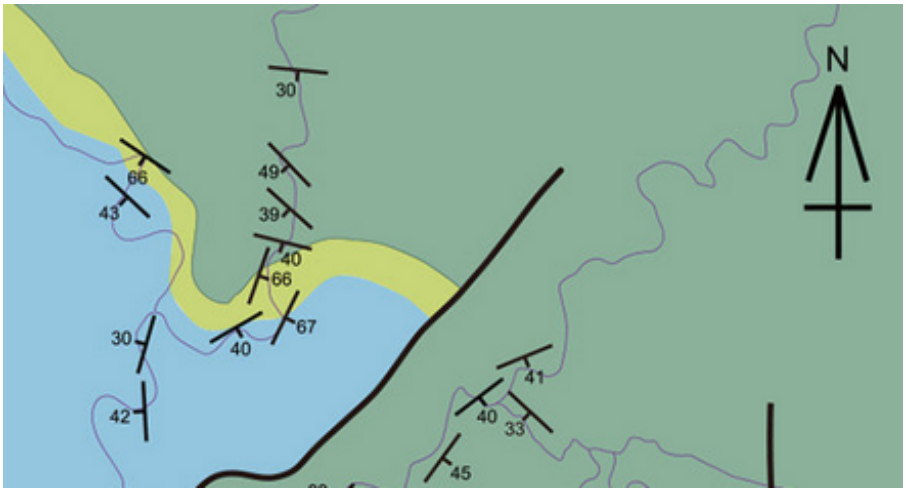


Fig. 3: Example of dip/strike representation on map

When entering data, geologists often need to determine the correct azimuth for strike or dip directions, or convert between strike and dip azimuths. These calculations can be tedious and prone to error if done manually.

A common scenario is the digitization of data from scanned historical (cartaceous) geological maps. In these cases, the dip angle is often provided as a label next to the strike and dip symbol. However, the strike or dip azimuth is usually not explicitly stated and must be determined manually by measuring the orientation (azimuth) of the strike line depicted on the symbol. This manual process can be time-consuming and prone to error, especially when working with large datasets.

The Dip-Strike Tools plugin streamlines these tasks by providing intuitive tools for data entry and conversion. It automates common calculations, reduces the risk of mistakes, and helps ensure that your geological data is both accurate and consistently formatted. The plugin is designed to grow, with future updates planned to add more advanced tools for data management and geological analysis.

1.1.2 "True" north and grid convergence

A bearing (or azimuth) is a clockwise angle measured from North to a direction of interest. However, "North" can refer to different reference directions depending on context:

- **Grid north** is the direction of the map's vertical (south-to-north) grid lines, defined by the map projection.
- **True north** is the direction along the Earth's surface towards the geographic North Pole (the local meridian).
- **Magnetic north** is the direction a compass needle points, towards the Earth's magnetic pole.

> **Note**
>
> *Magnetic declination* is the angle between magnetic north and true north. For most geological mapping, this angle is small and does not significantly affect dip/strike measurements, so the plugin does not account for it.

The **grid convergence** (or *meridian convergence*) is the angle between true north and grid north at a specific location. This angle varies depending on your position on the map and the map projection in use.

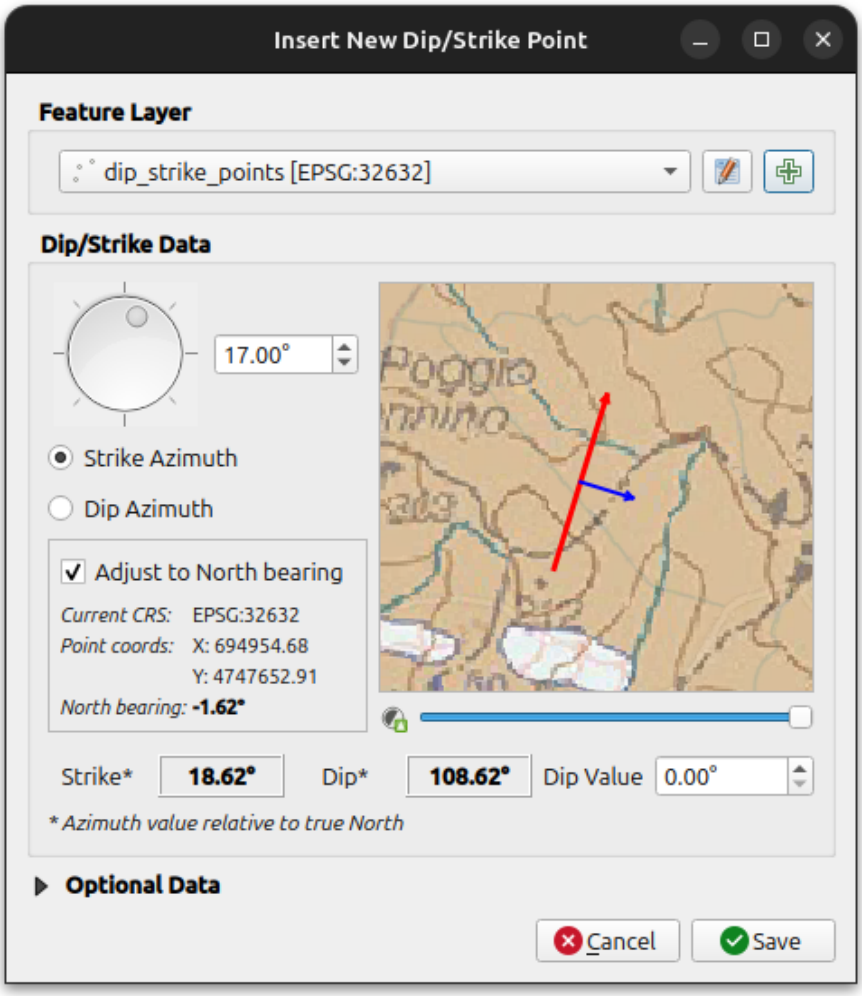


Fig. 4: Inserting a new dip/strike feature

When determining the dip or strike azimuth for a feature in QGIS, the plugin can automatically account for local grid convergence, applying the necessary correction to convert between "grid" azimuths (as measured on the map) and true azimuths (relative to true north). This helps ensure that your orientation data is accurate and consistent, regardless of the coordinate reference system or projection used.

1.2 Available Tools

The plugin currently provides four main tools accessible from the QGIS toolbar:

1.2.1 1. Create New Dip Strike Layer

Creates and configures new vector layers specifically designed for geological data collection. Supports multiple output formats and automatically sets up required fields.

→ *Learn more about Layer Creation*

1.2.2 2. Create Dip Strike Point

Interactive map tool for collecting geological measurements. Click anywhere on the map to record strike/dip data with visual preview and validation.

→ *Learn more about Data Insertion*

1.2.3 3. Calculate Dip/Strike Values

Batch calculation tool for converting between strike and dip azimuths in existing datasets. Includes input validation and rounding options.

→ *Learn more about Field Calculations*

1.2.4 4. Plugin Settings

Configure plugin behavior and customize geological type classifications for your specific workflow.

→ *Learn more about Settings*

CHAPTER

TWO

INSTALLATION

2.1 Stable Version (Recommended)

This plugin is published on the official QGIS plugins repository: https://plugins.qgis.org/plugins/dip_strike_tools/.

2.2 Beta Versions

Enable experimental extensions in the QGIS plugins manager settings panel to access beta releases.

2.3 Development Version

Warning

Early Adopters Only: If you define yourself as early adopter or a tester and can't wait for the release, the plugin is automatically packaged for each commit to main. You can use this address as repository URL in your QGIS extensions manager settings:

```
https://fpennica.github.io/dip-strike-tools/plugins.xml
```

Be careful, this version can be unstable.

For developers interested in contributing or understanding the plugin architecture, see the *development documentation*.

CHAPTER

THREE

LAYER CREATION

The **Create New Dip Strike Layer** tool provides an easy way to set up properly configured vector layers for geological data collection. This tool ensures that your layers have all the required fields and appropriate settings for strike and dip measurements.

> **Note**
>
> Creating a new layer is not strictly required for using the plugin. You can work with any existing point vector layer, and the plugin will automatically check and configure field mappings when you select an existing layer. The layer creation tool is provided as a convenience for users who want optimally configured layers from the start.

3.1 Accessing the Tool

1. Click the **Create New Dip Strike Layer** button in the Dip Strike Tools toolbar
2. Or access it through the plugin menu in QGIS

3.2 Layer Creation Dialog

The layer creation dialog allows you to configure all aspects of your new geological data layer:

3.2.1 Basic Settings

Layer Name

Enter a descriptive name for your layer. The name will appear in the QGIS Layers panel.

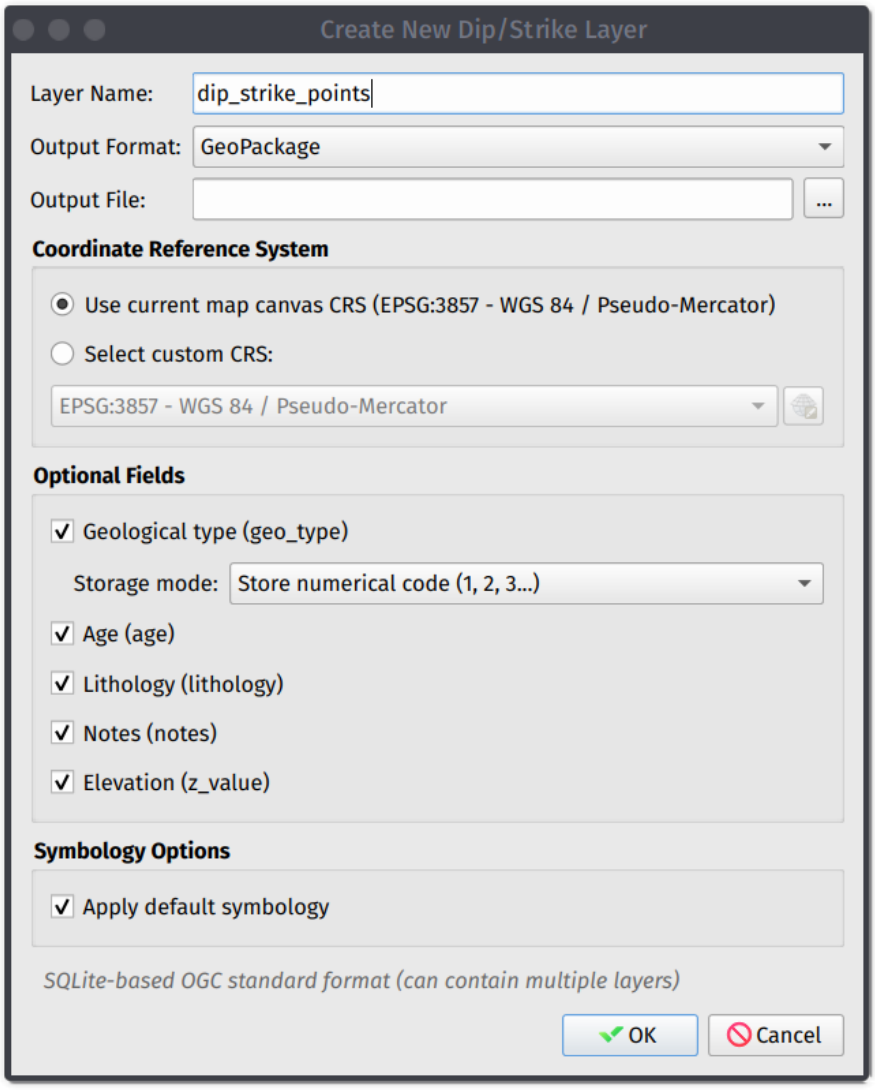


Fig. 1: Layer creation dialog

Layer Type

Choose from three storage options:

Memory Layer (Temporary): Fast creation and editing with data that exists only while QGIS is open. Ideal for quick analysis or temporary data collection, and can be saved later using QGIS "Export" functionality.

Shapefile: Traditional GIS format compatible with most GIS software. Field names are limited to 10 characters and the plugin automatically uses field mapping for compatibility.

GeoPackage: Modern SQLite-based format supporting longer field names and more data types in a single file that contains all data and styling. Recommended for new projects.

3.2.2 Coordinate Reference System (CRS)

Current Project CRS

Uses the same coordinate system as your current QGIS project, ensuring compatibility with existing project data. This is **recommended** for most use cases.

Custom CRS

Select any coordinate reference system when working with specific regional grids. Both geographic (lat/lon) and projected coordinates are supported.

3.2.3 Field Configuration

The plugin automatically creates the required fields for geological measurements:

Required Fields

The plugin creates three essential fields: **Strike Azimuth** for the direction of strike line (0-360°), **Dip Azimuth** for the direction of dip (strike + 90°), and **Dip Value** for the angle of dip from horizontal (0-90°).

Optional Fields

Enable optional fields based on your data collection needs. These include **Geological Type** for classification of the measured feature, **Age** for geological age or formation name, **Lithology** for rock type or lithological description, and **Notes** for additional comments or observations.

3.2.4 Geological Type Configuration

When you enable the **Geological Type** field, you can choose the storage method:

Store Codes

Saves numerical codes (1, 2, 3, etc.) which results in smaller file size but requires reference to decode meanings and is used with lookup tables.

Store Descriptions

Saves full text descriptions (e.g. "Strata", "Foliation", etc.) which creates self-documenting data that is immediately readable but results in larger file size.

> **Tip**
>
> The geological types are defined in the *plugin settings* and can be customized for your specific workflow.

3.2.5 Symbology Options

Apply Default Symbology

Automatically applies geological strike/dip symbols using traditional structural geology conventions with strike lines and dip tick marks.

No Symbology

Creates layer with simple point symbols, allowing custom styling later and is useful when integrating with existing style schemes.

3.3 Layer Creation Process

1. **Configure Settings**: Fill in all required settings in the dialog
2. **Click OK**: The plugin creates and configures the layer
3. **Automatic Setup**: The layer is added to your QGIS project, configured with proper field mappings, marked as a dip/strike layer for the plugin, and styled (if symbology option selected)

3.4 Next Steps

After creating your layer, you can **start data collection** using the *Create Dip Strike Point* tool, **import existing data** by copying features from other layers, **configure styling** to customize the appearance, and **set up your project** by adding base maps and other reference layers.

> **See also**
>
> For advanced field mapping and configuration options, see the *Data Insertion documentation*.

CHAPTER

FOUR

DATA INSERTION AND MANAGEMENT

The plugin's primary data collection tool allows you to click anywhere on the map to insert or edit dip/strike measurements.

4.1 Interactive Point Tool

1. **Access the Tool**: Click the plugin's main toolbar button or use the menu item
2. **Click on Map**: Click at any location where you want to record a measurement
3. **Enter Data**: The data entry dialog opens automatically with the clicked coordinates
4. **Target Layer Selection**: The plugin automatically selects the most appropriate target layer - either the last used layer, or the layer containing an existing point near your click location. If the selected layer lacks proper field configuration, the plugin will prompt you to configure it.

Tip

Feature Highlighting: When using the point tool, the plugin will automatically highlight existing point features that are close to your mouse cursor, even if they are from layers that don't have configured field mappings but are visible in the map. This helps you identify nearby measurements and avoid duplicate entries.

4.2 Data Entry Dialog Features

The interactive data entry dialog provides comprehensive input options:

4.2.1 Embedded Map

The dialog's embedded map offers a preview of the point feature at the clicked location. The feature is represented by a graphical marker with strike and dip lines that automatically update based on the azimuth values you enter.

Note

It is not possible to modify the point location (coordinates) within the dialog. To change the location, close the dialog and use the interactive point tool again, clicking on the new location on the QGIS map canvas. To move an already existing point, use the standard QGIS editing tools.

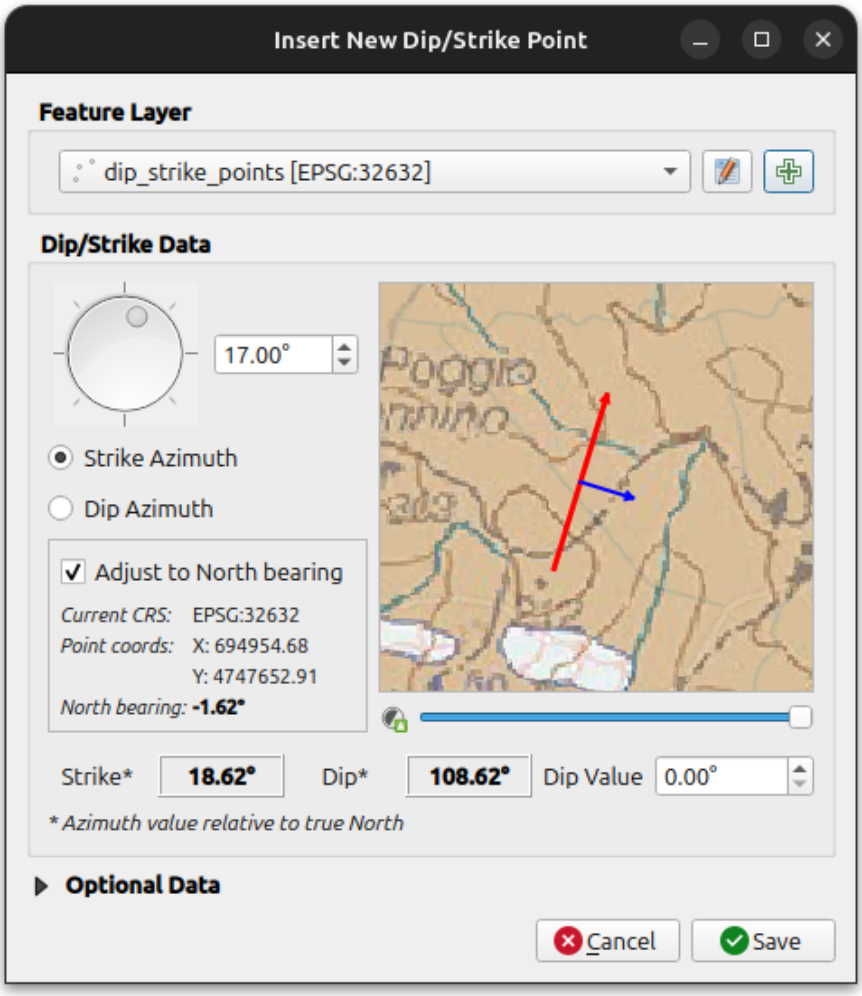


Fig. 1: Inserting a new dip/strike feature

The embedded map displays a portion of the QGIS map around the point location. The layers shown are the same as those displayed on the main QGIS map when using the point tool. To modify the embedded map background, simply adjust the QGIS map by enabling the desired layers, applying symbology, etc., before using the tool.

> **Tip**
>
> You can interact with the map using the mouse scroll wheel to zoom in and out, and by clicking and dragging with the left mouse button to pan.

The slider at the bottom of the map controls the transparency of the embedded map background. This feature helps provide a better view of the point marker against the underlying background.

4.2.2 Required Input

Strike or Dip Azimuth: Enter the azimuth value from 0-360° using either direct numeric input in the spin box or the interactive compass dial (click and drag to set value).

The entered azimuth value can represent either **Strike** or **Dip** direction - select the appropriate mode using the radio buttons. When you change between dip and strike modes, the orientation marker on the embedded map updates automatically to reflect your selection.

True North Correction: Use this checkbox to automatically apply true north correction to your input azimuth values, compensating for grid convergence at your location (see → *Overview* for more).

Dip Value: Enter the dip angle from 0-90° (where 0° is horizontal and 90° is vertical).

Corrected Dip and Strike Azimuth: The final calculated dip and strike values that will be saved to the layer's attribute table are displayed here, with any selected corrections applied.

4.2.3 Optional Data

DTM elevation value extraction

Automatic Elevation Extraction: When a suitable raster DTM (Digital Terrain Model) layer is loaded in your QGIS project, you can select it from the dropdown to automatically extract elevation values at point locations. The extracted elevation will be saved to the configured elevation field in your target layer.

Coordinate System Handling: The tool automatically handles coordinate transformations between the map canvas CRS and the DTM layer CRS, ensuring accurate elevation sampling regardless of coordinate system differences.

Manual Entry Option: If no DTM layer is selected or available, you can manually enter elevation values directly into the elevation field.

Additional Data Fields

Select **Geological Type** from customizable categories. Custom types are configurable in settings.

Additional fields include **Age** for geological age or formation name, **Lithology** for rock type description, and **Notes** for free-text field observations and comments.

4.3 Layer Management

4.3.1 Target Layer Selection

The plugin works with any point vector layer that contains appropriate fields (see → *Layer Creation* for more).

Field Mapping Configuration

When working with existing layers, the plugin automatically checks for compatible fields and suggests mappings. If automatic detection works well, you can start using the layer immediately. If not, use the field mapping tool to manually connect your layer's fields to the plugin's expected data structure.

Automatic Field Detection

The plugin automatically suggests field mappings based on **field names** (recognizes common naming patterns), **data types** (matches numeric fields to azimuth/dip values), and **field order** (considers typical field arrangements).

Manual Configuration

If automatic detection doesn't work, manually configure mappings by opening field configuration (click the gear icon next to layer selection), mapping required fields (set strike azimuth, dip azimuth, and dip value fields), mapping optional fields (configure elevation, geological type, age, lithology, and notes), validating configuration (the dialog shows validation status), and saving settings (mappings are stored with the layer).

CHAPTER

FIVE

CALCULATE DIP/STRIKE VALUES

The Dip-Strike Tools plugin includes a **calculator tool** that can compute dip azimuths from strike azimuths (and vice versa) in existing vector layers or tables.

This tool is useful when you have geological data with either strike or dip azimuth values and need to calculate the corresponding perpendicular direction.

Note

- **Dip Azimuth** = Strike Azimuth + 90° (normalized to 0-360°)
- **Strike Azimuth** = Dip Azimuth - 90° (normalized to 0-360°)

5.1 Using the Tool

Open QGIS and ensure the Dip Strike Tools plugin is loaded, then click the **Calculate Dip/Strike Values** button in the toolbar, or access it through the plugin menu.

5.1.1 Step 1: Select a Layer

Choose any vector layer or table from your current QGIS project. The layer must be valid and loaded in the project, and both geometric layers (points, lines, polygons) and non-spatial tables are supported.

5.1.2 Step 2: Choose Calculation Type

Select the type of calculation you want to perform:

Calculate Dip Azimuth from Strike Azimuth: Converts strike values to dip values

Calculate Strike Azimuth from Dip Azimuth: Converts dip values to strike values

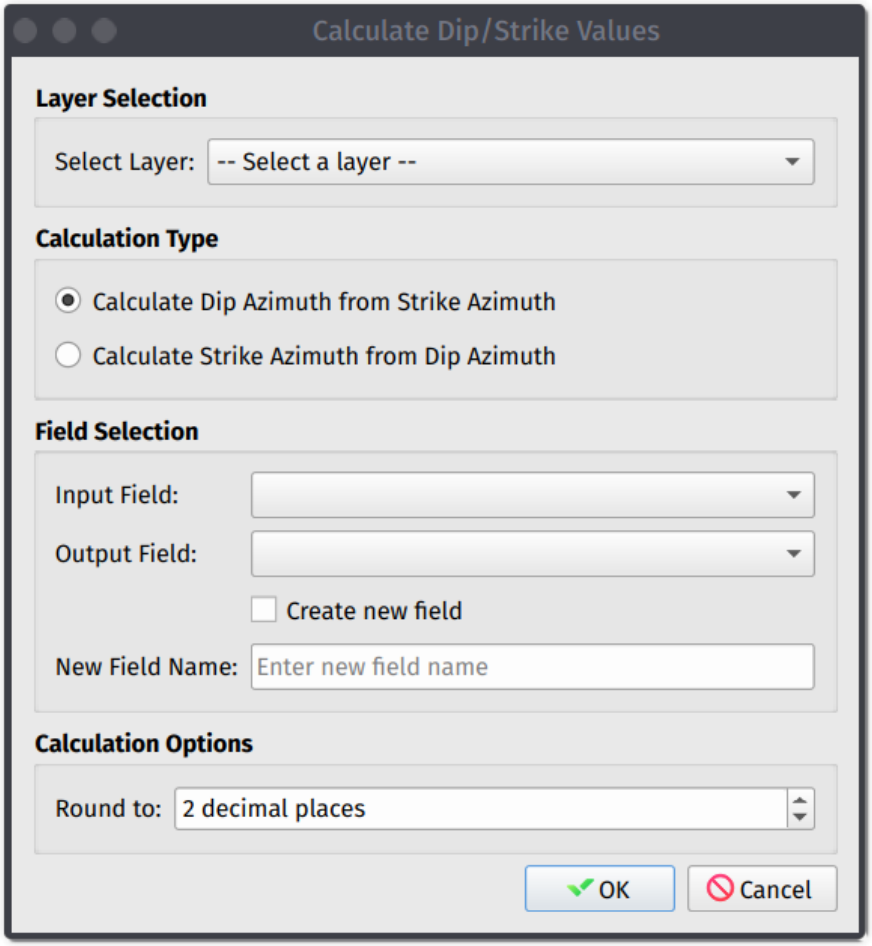


Fig. 1: Field calculator tool

5.1.3 Step 3: Select Input Field

Choose the field containing the source azimuth values. Only numeric fields (integer or double) are available for selection, and values should be in degrees (0-360°).

5.1.4 Step 4: Choose Output Destination

You have two options for storing the calculated values:

- Select an **existing numeric field** from the dropdown. The calculated values will overwrite existing values in this field.
- Check the "Create new field" checkbox, enter a name for the new field, and a new double-precision field will be added to the layer.

5.1.5 Step 5: Set Calculation Options

- **Rounding**: Choose how many decimal places to round the results to Default is 2 decimal places. This affects the precision of the calculated values stored in the output field.

5.1.6 Step 6: Execute Calculation

Click **OK** to start the calculation. The tool will process all features with valid input values, and progress and results are logged in the QGIS message log.

CHAPTER

SIX

PLUGIN SETTINGS AND CUSTOMIZATION

The Dip-Strike Tools plugin provides customization options through QGIS's integrated settings system. This page covers all available settings, customization options, and configuration workflows.

6.1 Accessing Plugin Settings

The plugin settings are integrated into QGIS's main Options dialog. Access plugin settings by going to `Settings` > `Options...` in QGIS menu, navigating to "Dip Strike Tools" in the left sidebar, or using direct access through the plugin's main menu.

6.1.1 Debug Mode

Enable detailed logging for troubleshooting and development purposes.

> **Note**
>
> **When to Enable**: Use debug mode when investigating plugin behavior issues, reporting bugs to developers, or understanding plugin operations in detail.

Effects: Increases log output verbosity, may impact plugin performance slightly, and provides detailed error information.

Usage: Enable temporarily for troubleshooting, disable for normal daily use, and note that it's required for meaningful bug reports.

6.1.2 Geological Types Management

The plugin provides a system for managing geological classification types used in data collection and analysis. The default classes can be customized.

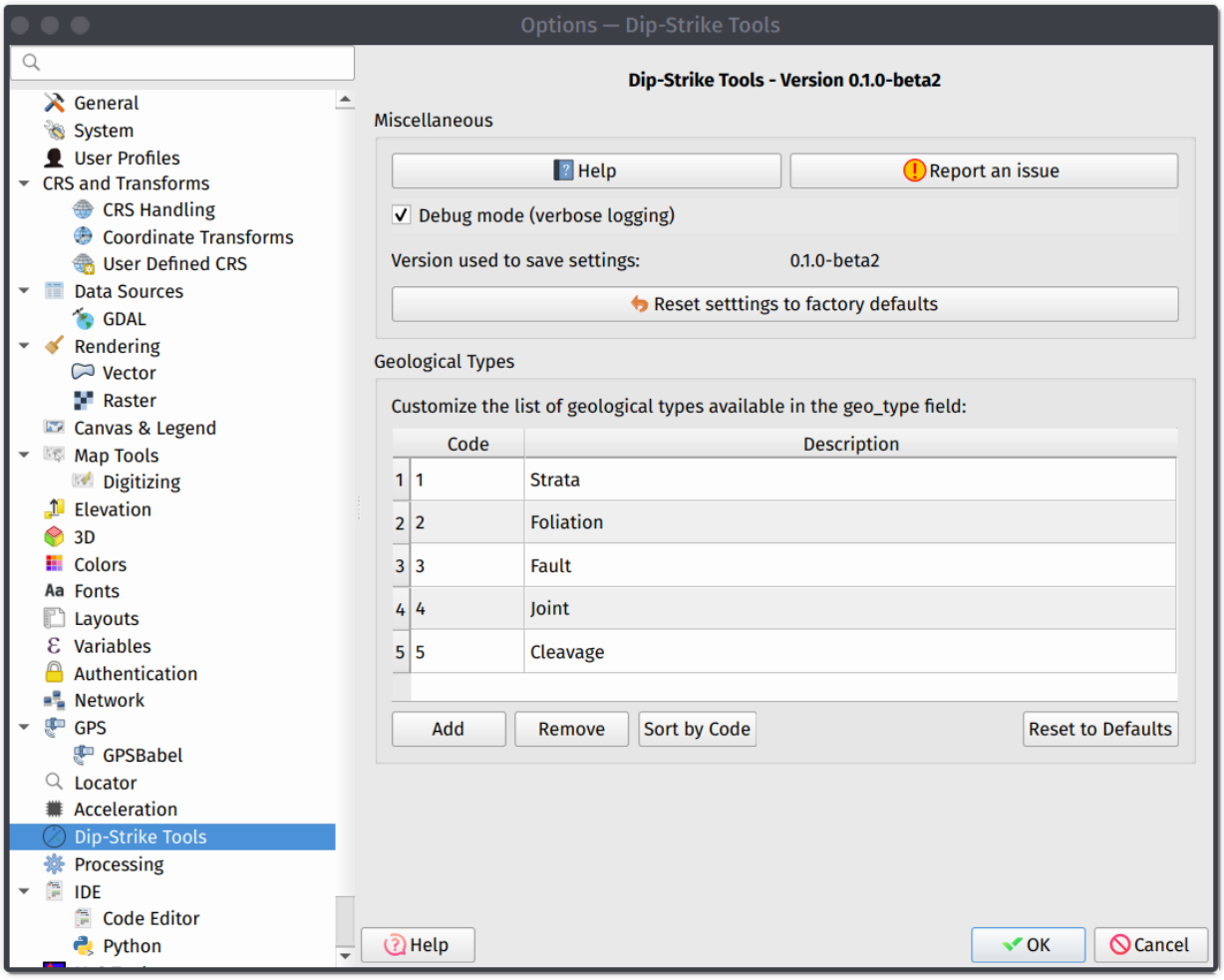


Fig. 1: Field calculator tool

Adding New Types

Access the geological types management table, click the “Add” button to add a new type, provide a short unique code (numbers or letters), add a clear descriptive name, and apply settings to store the new type.

Editing Existing Types

Select the row for the type you want to modify, double-click to edit code or description, ensure codes remain unique, and apply settings to save changes to update the system.

Removing Types

Select the type to remove, click the “Remove” button, confirm the action when prompted, and save changes to apply settings making removal permanent.

> **Warning**
>
> Removing geological types may affect existing data that uses those classifications.

Resetting to Defaults

To start fresh, use the “Reset to Defaults” button, confirm the action by acknowledging the warning dialog, review that default types are restored, and customize again by adding your custom types as needed.

6.1.3 Geological Types in Data Collection

When collecting data, geological types appear in the dropdown lists in data entry dialog.

When you enable the **Geological Type** field in layer configuration, you can choose the storage method:

- **Store codes**: Saves numerical codes (1, 2, 3, etc.) which results in smaller file size but requires reference to decode meanings and is used with lookup tables.
- **Store Descriptions**: Saves full text descriptions (e.g. “Strata”, “Foliation”, etc.) which creates self-documenting data that is immediately readable but results in larger file size.

This choice directly influences the classification that may be used for layer symbology. Plan ahead by designing type systems before starting major projects.

CHAPTER

SEVEN

CONTRIBUTING GUIDELINES

First off, thanks for considering to contribute to this project!

These are mostly guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

7.1 Git hooks

We use git hooks through pre-commit to enforce and automatically check some "rules". Please install them (`pre-commit install`) before to push any commit.

See the relevant configuration file: `.pre-commit-config.yaml`.

7.2 Code Style

Make sure your code *roughly* follows PEP-8 and keeps things consistent with the rest of the code:

- docstrings: sphinx-style is used to write technical documentation.
- linting, formatting, imports sorting: ruff

CHAPTER

EIGHT

DEVELOPMENT

8.1 Environment setup

8.2 Prerequisites

Before setting up the development environment, you need to install some required libraries and the following tools:

1. **uv** - Fast Python package installer and dependency manager
2. **just** - Command runner for project automation

8.2.1 Qt 5 libraries and tools

Fedora 42

```
sudo dnf install qgis qgis-devel python3-devel qt5-designer qt5-linguist
```

8.2.2 Install uv

```
# Using curl (recommended)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Or using pip
pip install uv
```

8.2.3 Install just

```
# On Ubuntu/Debian
sudo apt install just

# Or using cargo
cargo install just

# Or download from releases
curl --proto '=https' --tlsv1.2 -sSf https://just.systems/install.sh | bash -s -- --
↪to ~/.local/bin
```

8.2.4 Suggested IDE: Visual Studio Code

This project includes pre-configured settings for VS Code, available in `.vscode` folder, including linting, formatting and tasks settings.

To get started with VS Code:

```
# Install VS Code (Ubuntu/Debian)
sudo snap install code --classic

# Or download from https://code.visualstudio.com/

# Open the project in VS Code
code .
```

VS Code will automatically suggest installing recommended extensions when you open the project.

8.3 Quick Setup (Recommended)

> **Warning**
>
> At the moment the quick setup is **only** supported on **Linux**. For other OS environments, follow and adapt the manual setup instructions.

The easiest way to set up the development environment is using the automated setup:

```
# Create virtual environment, install dependencies, and set up development links
just bootstrap-dev
```

This command will:

- Create a Python virtual environment with system-site-packages (for PyQGIS)
- Install all development dependencies using uv
- Create symbolic links for development

8.4 Manual Setup

If you prefer to set up the environment step by step:

8.4.1 1. Create virtual environment

Using uv (recommended):

```
# Create virtual environment with system-site-packages for PyQGIS
just create-venv

# Manual override of the system python version if needed
just create-venv-manual 3.13
```

Alternative methods:

Using qgis-venv-creator through pipx:

```
# Install pipx if not already installed
sudo apt install pipx

# Create QGIS-compatible virtual environment
pipx run qgis-venv-creator --venv-name ".venv"
```

Traditional method:

```
# Create virtual environment linking to system packages (for PyQGIS)
python3 -m venv .venv --system-site-packages
source .venv/bin/activate
```

8.4.2 2. Install development dependencies

With uv (recommended):

```
# Install all dependency groups (dev, testing, docs, ci)
uv sync --all-groups

# Or install specific groups
uv sync --group dev --group testing
```

Traditional method:

```
# Activate virtual environment first
source .venv/bin/activate

# Upgrade pip and install dependencies
python -m pip install -U pip
python -m pip install -U pytest  # install dependencies listed in pyproject.toml

# Install git hooks (pre-commit)
pre-commit install
```

8.4.3 3. Development Links and QGIS Profile

Create Development Links

To develop the plugin, you need to create symbolic links so QGIS can find your plugin:

```
# Create symbolic links to your QGIS plugins directory
just dev-link

# Or specify a custom QGIS plugin path
just dev-link /path/to/your/qgis/plugins
```

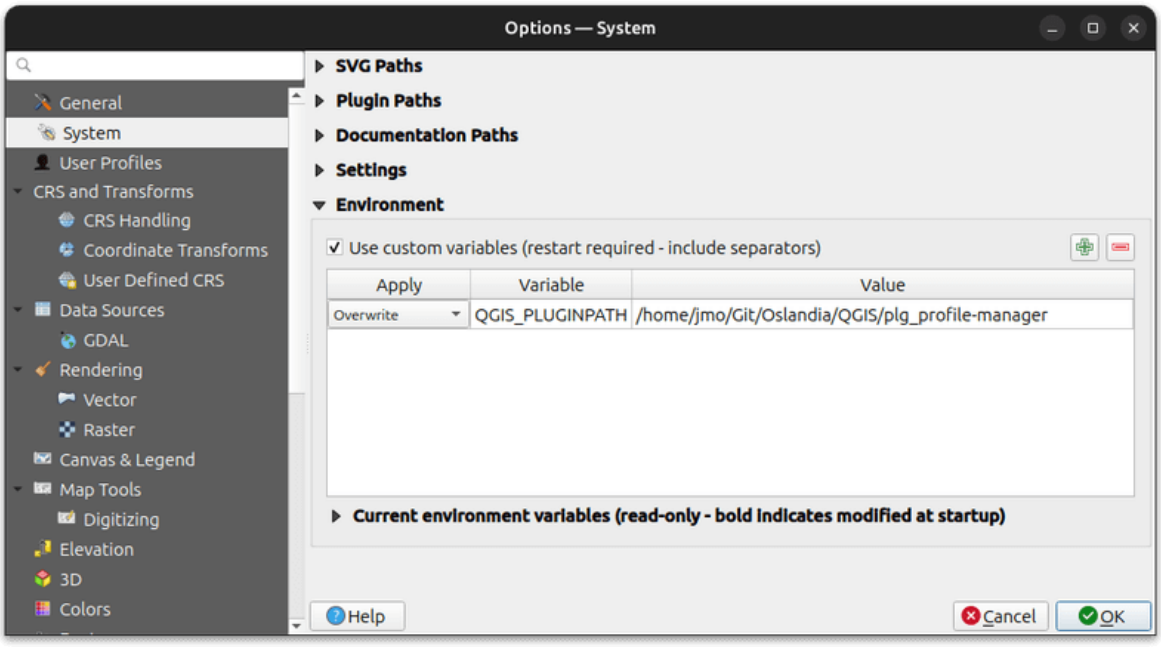
Dedicated QGIS Profile

It's recommended to create a dedicated QGIS profile for the development of the plugin to avoid conflicts with other plugins.

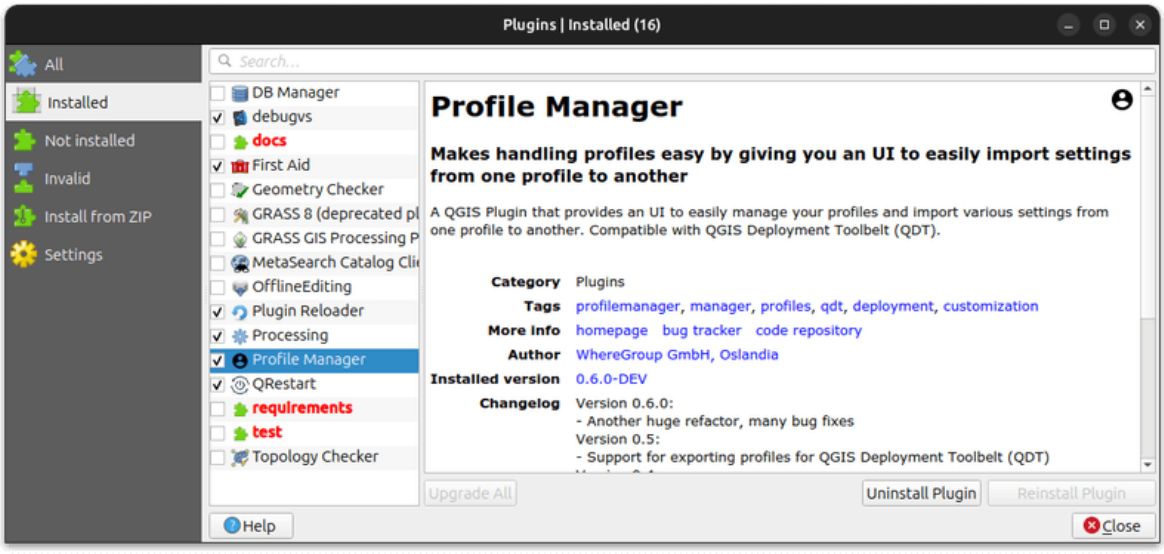
1. From the command-line (a terminal with qgis executable in `PATH` or OSGeo4W Shell):

```
# Linux
qgis --profile plg_dip_strike_tools
# Windows - OSGeo4W Shell
qgis-ltr --profile plg_dip_strike_tools
# Windows - PowerShell opened in the QGIS installation directory
PS C:\Program Files\QGIS 3.40.4\LTR\bin> .\qgis-ltr-bin.exe --profile plg_dip_
↪strike_tools
```

2. Then, set the `QGIS_PLUGINPATH` environment variable to the path of the plugin in profile preferences:



3. Finally, enable the plugin in the plugin manager (ignore invalid folders like documentation, tests, etc.):



Alternative: Using Docker

For a consistent development environment, you can use Docker to run QGIS:

```
# Pull the latest QGIS LTR Docker image
just qgis-ltr-pull

# Start QGIS in Docker (Linux only)
just qgis-docker
```

This automatically mounts your plugin directory and provides a clean QGIS environment.

8.5 Common Development Tasks

The project uses `just` as a command runner to automate common development tasks. Here are the most useful commands:

```
# Show all available tasks
just

# Create virtual environment and set up development links
just bootstrap-dev

# Update dependencies to latest versions
just update-deps

# Run tests with coverage
just test

# Update translation files
just trans-update

# Compile translation files
just trans-compile
```

(continues on next page)

(continued from previous page)

```
# Build documentation with auto-reload (for development)
just docs-autobuild

# Build HTML documentation
just docs-build-html

# Create development symlinks (if not using bootstrap-dev)
just dev-link

# Start QGIS LTR in Docker (Linux only)
just qgis-docker

# Package the plugin for distribution
just package <version>
```

8.5.1 Using VS Code Tasks

If you're using VS Code, the project includes pre-configured tasks that you can run directly from the editor:

- **Ctrl+Shift+P** → "Tasks: Run Task" to see all available tasks
- **Upgrade Dependencies**: Update all development dependencies
- **Run Tests**: Execute the full test suite
- **Translation Update**: Update translation files
- **Translation Compile**: Compile translation files
- **Create Virtual Environment**: Set up the development environment
- **Sync Dependencies**: Install project dependencies

These tasks use the same `just` and `uv` commands but provide a convenient GUI interface within VS Code.

8.5.2 Using uv directly

Use `uv run <command>` to ensure commands run in the correct virtual environment.

```
# Run commands in the virtual environment
uv run pytest
uv run ruff check
uv run pre-commit run --all-files

# Add new dependencies
uv add requests  # Runtime dependency
uv add --group dev black  # Development dependency
uv add --group testing pytest-mock  # Testing dependency

# Install packages
uv sync  # Install all dependencies
uv sync --group dev  # Install dev dependencies only
uv sync --all-groups  # Install all dependency groups

# Update dependencies
```

(continues on next page)

(continued from previous page)

```
uv lock --upgrade  # Update lock file
uv sync  # Apply updates
```

CHAPTER

NINE

PYQT5/PYQT6 COMPATIBILITY IMPLEMENTATION

The plugin features a PyQt5/PyQt6 compatibility implementation trying to make it work across different QGIS installations using either PyQt5 or PyQt6. The compatibility layer should be dropped when QGIS fully transitions to PyQt6.

9.1 Overview

The PyQt5/PyQt6 compatibility layer detects the PyQt version at runtime and provides unified interfaces for PyQt-specific functionality.

9.2 Some Differences Between PyQt5 and PyQt6

1. **QVariant changes**: In PyQt6, `QVariant` is deprecated and Python native types are used instead
2. **QMetaType changes**: API restructuring with nested enum access patterns
3. **Dialog result constants**: Moved from class attributes to enum values (`QDialog.Accepted` → `QDialog.DialogCode.Accepted`)
4. **Cursor shape constants**: Restructured into nested enums (`Qt.ArrowCursor` → `Qt.CursorShape.ArrowCursor`)
5. **Enum structure**: Proper Python enums in PyQt6 vs class attributes in PyQt5

9.3 Compatibility Module Implementation

9.3.1 Core Implementation

The main compatibility logic is implemented in `dip_strike_tools/toolbelt/qt_compat.py`, which provides:

- **Runtime PyQt version detection**: Automatically detects whether PyQt5 or PyQt6 is being used
- **QVariant compatibility**: Handles the deprecation of `QVariant` in PyQt6
- **QMetaType compatibility**: Provides unified access to type constants
- **Dialog result constants**: Unified access to dialog result values
- **Cursor shape constants**: Handles enum structure changes in PyQt6

9.3.2 Key Components

1. Version Detection

```
from dip_strike_tools.toolbelt import IS_PYQT5, IS_PYQT6, get_qt_version_info

# Runtime detection
if IS_PYQT6:
    # PyQt6 specific code
else:
    # PyQt5 specific code

# Get detailed version info
info = get_qt_version_info()
```

2. QVariant Compatibility

```
from dip_strike_tools.toolbelt import QVariant, qvariant_cast

# Use QVariant types consistently
field.setType(QVariant.Double)  # Works in both PyQt5 and PyQt6
field.setType(QVariant.String)  # Works in both PyQt5 and PyQt6

# Cast values safely
value = qvariant_cast(raw_value, QVariant.Double)
```

3. QMetaType Compatibility

```
from dip_strike_tools.toolbelt import QMetaTypeWrapper as QMetaType

# Use QMetaType consistently
field.setType(QMetaType.Double)   # Works in both versions
field.setType(QMetaType.QString)  # Works in both versions
```

4. Dialog Result Constants

```
from dip_strike_tools.toolbelt import DIALOG_ACCEPTED, DIALOG_REJECTED

# Check dialog results consistently
if dialog.result() == DIALOG_ACCEPTED:
    # Handle accepted dialog
elif dialog.result() == DIALOG_REJECTED:
    # Handle rejected dialog
```

5. Enhanced Enum Handling

```
from dip_strike_tools.toolbelt import enum_value, get_dialog_result

# Generic enum value access
value = enum_value(SomeEnum, "ValueName")

# Specific dialog result handling
accepted = get_dialog_result(QDialog, "Accepted")  # Works in both PyQt5/6
```

9.4 Usage Patterns

9.4.1 Field Type Setting

```
from dip_strike_tools.toolbelt import QMetaTypeWrapper as QMetaType

# Create a field with double type
field = QgsField()
field.setName("value_field")
field.setType(QMetaType.Double)  # Works in both PyQt5/6
```

9.4.2 Value Handling

```
from dip_strike_tools.toolbelt import QVariant, qvariant_cast

# Safe value extraction
def extract_value(qvariant_value):
    if qvariant_value is None:
        return None
    elif isinstance(qvariant_value, QVariant):
        if qvariant_value.isNull():
            return None
        else:
            return qvariant_value.value()
    else:
        return qvariant_value
```

9.4.3 Type Checking

```
from dip_strike_tools.toolbelt import QVariant

# Check field types consistently
if field.type() == QVariant.Double:
    # Handle double field
elif field.type() == QVariant.String:
    # Handle string field
```

9.5 Maintenance

When adding new PyQt-dependent code:

1. **Import from toolbelt**: Use compatibility imports instead of direct PyQt imports
2. **Check compatibility module**: Add new compatibility patterns if needed
3. **Test both versions**: Ensure new code works with both PyQt5 and PyQt6
4. **Update documentation**: Document any new compatibility patterns

CHAPTER

TEN

DOCUMENTATION

This project uses Sphinx to generate documentation from docstrings (documentation in-code) and custom pages written in Markdown (through the MyST parser).

10.1 Prerequisites

The project uses:

- **uv** for dependency management
- **just** for task automation

Make sure you have both tools installed and a proper development environment set up (see *environment.md* for details).

10.2 Build documentation website

The project provides several convenient tasks for building documentation:

10.2.1 HTML Documentation

```
# Build HTML documentation
just docs-build-html
```

This will install the required dependencies from the `docs` group in `pyproject.toml` and build the HTML documentation to `docs/_build/html/`. Open `docs/_build/html/index.html` in a web browser to view it.

10.2.2 PDF Documentation

```
# Build PDF documentation (requires LaTeX)
just docs-build-pdf
```

This builds a PDF version of the documentation using LaTeX. The resulting PDF will be in `docs/_build/latex/`.

10.3 Write documentation using live render

For development, you can use the auto-rebuild feature:

```
# Start auto-building server with live reload
just docs-autobuild
```

Open http://localhost:8000 in a web browser to see the HTML render updated automatically when you save a documentation file.

10.4 Manual commands

If you prefer to run the commands manually instead of using `just`:

```
# Install documentation dependencies
uv sync --group docs

# Build HTML manually
uv run sphinx-build -b html -j auto -d docs/_build/cache -q docs docs/_build/html

# Auto-build with live reload manually
uv run sphinx-autobuild -b html docs/ docs/_build --port 8000
```

10.5 MyST Markdown Features

Since this project uses MyST parser, you can leverage advanced Sphinx features in Markdown files, including:

- Cross-references
- Directives and roles
- Code block highlighting
- Admonitions (notes, warnings, etc.)

Refer to the MyST documentation for complete syntax reference.

CHAPTER

ELEVEN

MANAGE TRANSLATIONS

11.1 Requirements

Qt Linguist tools are used to manage translations. Install them using your system package manager:

```
# On Ubuntu/Debian
sudo apt install qttools5-dev-tools

# On Fedora
sudo dnf install qt5-designer qgis-devel

# Or on newer systems with Qt6
sudo apt install qt6-tools-dev
```

11.2 Quick Workflow (Recommended)

The project provides automated commands using `just` for translation management:

```
# Update translation files (extract translatable strings)
just trans-update

# Compile translation files (.ts to .qm)
just trans-compile
```

11.3 Manual Workflow

If you prefer to run the commands manually or need more control:

11.3.1 1. Ensure proper environment

Always use `uv run` to ensure commands run in the correct virtual environment:

```
# Generate the plugin_translation.pro file
uv run python scripts/generate_translation_profile.py
```

11.3.2 2. Update .ts files

Extract translatable strings from source code:

```
# Using uv run (recommended)
uv run pylupdate5 -noobsolete -verbose dip_strike_tools/resources/i18n/plugin_
↪translation.pro

# Or using just (simpler)
just trans-update
```

11.3.3 3. Translate text

Use Qt Linguist to translate your text or edit `.ts` files directly:

```
# Launch Qt Linguist for translation
linguist dip_strike_tools/resources/i18n/*.ts

# Or edit .ts files directly in your text editor
```

11.3.4 4. Compile translations

Convert `.ts` files to binary `.qm` files:

```
# Using uv run (recommended)
uv run lrelease dip_strike_tools/resources/i18n/*.ts

# Or using just (simpler)
just trans-compile
```

11.4 Development Workflow Integration

Translation management is integrated into the development workflow:

```
# During development, after adding new translatable strings
just trans-update

# After translating, before testing
just trans-compile

# Run all development tasks including translations
just bootstrap-dev  # Sets up environment and compiles existing translations
```

11.5 Notes

- **`pylupdate5`**: Extracts translatable strings from source code and updates `.ts` files
  - `-noobsolete`: Avoids removing obsolete translations (maintains translation history)
  - `-verbose`: Provides detailed output for debugging
- **`linguist`**: Launches Qt Linguist GUI for user-friendly translation editing
- **`lrelease`**: Compiles `.ts` files into binary `.qm` files for Qt applications
- **Git tracking**: The resulting `*.qm` files should not be tracked in git history since they're autogenerated binary files. The CI/CD pipeline handles generating and packaging them

CHAPTER

TWELVE

PACKAGING AND DEPLOYMENT

12.1 Overview

This project uses uv for dependency management and just for task automation. The packaging is handled by qgis-plugin-ci tool, which performs a `git archive` operation based on the `CHANGELOG.md`.

12.2 Prerequisites

Ensure you have the required tools installed:

- **uv**: For Python dependency management
- **just**: For task automation
- **Git**: For version control and tagging

12.3 Development Setup

Set up the development environment using the justfile:

```
# Bootstrap complete development environment
just bootstrap-dev

# Or run individual steps:
just create-venv          # Create virtual environment with uv
just dev-link            # Create symbolic links for development
just trans-compile       # Compile translations
```

12.4 Packaging

12.4.1 Install Dependencies

The CI dependencies are managed through uv and defined in `pyproject.toml`:

```
# Sync CI dependencies
uv sync --group ci
```

12.4.2 Create Package

Use the justfile task to package a version:

```
# Package a specific version
just package 1.3.1

# The package task automatically:
# - Syncs CI dependencies
# - Copies required files (LICENSE, CHANGELOG.md, CREDITS.md)
# - Runs qgis-plugin-ci package
# - Restores development links
```

12.4.3 Manual Packaging

If you need to run qgis-plugin-ci directly:

```
# Package latest version from CHANGELOG.md
uv run qgis-plugin-ci package latest

# Package specific version
uv run qgis-plugin-ci package 1.3.1
```

12.5 Release Process

The release process is automated through GitHub Actions and follows a standard git workflow: **1 released version = 1 git tag**.

12.5.1 Release Steps

For a tag `X.y.z` (must be SemVer compliant):

1. **Update `CHANGELOG.md`**: Add the new version entry. You can write it manually or use GitHub's auto-generated release notes:
    1. Go to project's releases and click on `Draft a new release`
    2. In `Choose a tag`, enter the new tag
    3. Click on `Generate release notes`
    4. Copy/paste the generated text from `## What's changed` until the line before `**Full changelog**:...` in the `CHANGELOG.md`, replacing `What's changed` with the tag and publication date.
2. **Update metadata.txt** (optional): Change the version number in `dip_strike_tools/metadata.txt`. It's recommended to use the next version number with `-DEV` suffix (e.g. `1.4.0-DEV` when `X.y.z` is `1.3.0`) to avoid confusion during development.
3. **Create and push the git tag**:

    ```
    # Create annotated tag
    git tag -a X.y.z {git commit hash} -m "Release version X.y.z"
    ```

(continues on next page)

(continued from previous page)

```
# Push tag to main branch
git push origin X.y.z
# or push all tags at once
git push --tags
```

4. **Automated CI/CD**: The GitHub Actions workflow will automatically:
   - Compile translations
   - Package the plugin
   - Create a GitHub release
   - Publish to the official QGIS plugins repository

12.5.2 Testing Release Process

Use the justfile to test the release process without publishing:

```
# Test release process locally
just release-test X.y.z
```

This runs the packaging steps without GitHub token and OSGEO authentication.

12.5.3 Troubleshooting Failed Releases

If the CI/CD pipeline fails or you need to recreate a release:

```
# Delete the problematic tag locally and remotely
git tag -d X.y.z
git push origin :refs/tags/X.y.z

# Fix the issue, create a new tag, and push again
git tag -a X.y.z {corrected commit hash} -m "Release version X.y.z"
git push origin X.y.z
```

12.6 CI/CD Pipeline

The project uses GitHub Actions for continuous integration and deployment:

- **Linter**: Code quality checks with ruff
- **Tester**: Run tests with pytest-qgis
- **Documentation**: Build and deploy documentation
- **Package & Release**: Automated packaging and publishing on tag push

12.6.1 Workflow Files

- `.github/workflows/linter.yml`: Code linting
- `.github/workflows/tester.yml`: Test execution
- `.github/workflows/documentation.yml`: Documentation building
- `.github/workflows/package_and_release.yml`: Release automation

CHAPTER

THIRTEEN

TESTING THE PLUGIN

The plugin uses pytest with pytest-qgis for comprehensive testing. Tests are organized in 2 separate folders with clear separation of concerns:

- `tests/unit`: testing code which is independent of QGIS API (uses mocking)
- `tests/qgis`: testing code which depends on QGIS API (integration tests)

13.1 Test Organization Principles

13.1.1 Unit Tests (`tests/unit/`)

- **Purpose**: Test individual functions and classes in isolation
- **Dependencies**: Use mocking to avoid QGIS dependencies
- **Markers**: `@pytest.mark.unit` or `@pytest.mark.integration` (for complex interactions)
- **Speed**: Fast execution, no external dependencies
- **Example**: Testing mathematical calculations, utility functions

13.1.2 QGIS Integration Tests (`tests/qgis/`)

- **Purpose**: Test functionality that requires actual QGIS environment
- **Dependencies**: Real QGIS classes and interfaces
- **Markers**: `@pytest.mark.qgis`
- **Speed**: Slower execution due to QGIS initialization
- **Example**: Testing map tools, layer operations, GUI components

13.2 Test Markers

Tests are organized using pytest markers that are automatically applied based on location and can be explicitly set:

- `@pytest.mark.unit`: Unit tests that don't require QGIS (auto-applied to `tests/unit/`)
- `@pytest.mark.qgis`: Tests that require QGIS environment (auto-applied to `tests/qgis/`)
- `@pytest.mark.integration`: Integration tests between components (explicit)

13.2.1 Automatic Marker Application

The test configuration (`tests/conftest.py`) automatically applies markers:

- Files in `tests/unit/` get `@pytest.mark.unit` unless explicitly marked
- Files in `tests/qgis/` get `@pytest.mark.qgis` unless explicitly marked
- QGIS tests are automatically skipped if QGIS is not available

13.3 Run tests

13.3.1 Using justfile (recommended)

```
# Run all tests with coverage report
just test

# The justfile command is equivalent to:
uv sync --no-group ci
# the "--no-group ci" option might be required to avoid Qt library conflicts with␣
↪qgis-plugin-ci dependencies
uv sync --group testing
uv run pytest -v --cov=dip_strike_tools --cov-report=term-missing
```

13.3.2 Using pytest directly

```
# Run all tests
uv run pytest -v

# Run with coverage report
uv run pytest -v --cov=dip_strike_tools --cov-report=term-missing

# Run only unit tests (no QGIS required)
uv run pytest -m unit -v

# Run only integration tests
uv run pytest -m integration -v

# Run only QGIS tests
uv run pytest -m qgis -v

# Run unit and integration tests (no QGIS required)
uv run pytest -m "unit or integration" -v
```

(continues on next page)

(continued from previous page)

```
# Run tests excluding QGIS tests (useful when QGIS is not available)
uv run pytest -m "not qgis" -v

# Run specific test file
uv run pytest tests/unit/test_plugin_main.py -v
uv run pytest tests/qgis/test_plugin_main_qgis.py -v

# Run specific test class
uv run pytest tests/unit/test_plugin_main.py::TestDipStrikeToolsPluginBasic -v

# Run specific test method
uv run pytest tests/unit/test_plugin_main.py::TestDipStrikeToolsPluginBasic::test_
↪plugin_import -v
```

13.4 Test Structure Examples

13.4.1 Unit Test Example

Unit tests use mocking to isolate functionality and don't require a QGIS environment:

```
@pytest.mark.unit
class TestMyModule:
    @patch('my_module.QgsProject')
    def test_something(self, mock_project):
        # Test with mocked QGIS dependencies
        pass
```

13.4.2 Integration Test Example

Integration tests test interactions between components. They can be in the same file as unit tests but are marked separately:

```
@pytest.mark.integration
class TestMyModuleIntegration:
    def test_component_interaction(self):
        # Test interactions between mocked components
        pass
```

13.4.3 QGIS Integration Test Example

QGIS tests use pytest-qgis and run in a real QGIS environment:

```
@pytest.mark.qgis
class TestMyModuleQGIS:
    def test_with_qgis(self, qgis_iface):
        # Test with real QGIS interface
        pass
```

13.5 Coverage

The project uses coverage.py to track test coverage:

```
# Generate coverage report
uv run pytest --cov=dip_strike_tools --cov-report=html

# View HTML coverage report
open htmlcov/index.html
```

Coverage configuration is in `pyproject.toml`:

```
[tool.coverage.run]
source = ["dip_strike_tools"]
omit = ["*/tests/*", "*/test_*", "*/__pycache__/*"]
```

13.6 Writing Tests

13.6.1 For new modules

1. **Unit tests** in `tests/unit/test_module_name.py`:
   - Test basic functionality with mocked dependencies
   - Test error handling and edge cases
   - Test without requiring QGIS
2. **Integration tests** in `tests/qgis/test_module_name_qgis.py`:
   - Test with real QGIS environment
   - Test GUI components and user interactions
   - Test QGIS API integration

13.6.2 Example test structure

```
# tests/unit/test_my_module.py
import pytest
from unittest.mock import Mock, patch

@pytest.mark.unit
class TestMyModule:
    def test_basic_functionality(self):
        # Basic unit test
        pass

    @patch('my_module.qgis_dependency')
    def test_with_mocked_qgis(self, mock_qgis):
        # Test with mocked QGIS
        pass

# tests/qgis/test_my_module_qgis.py
import pytest
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.qgis
class TestMyModuleQGIS:
    def test_with_real_qgis(self, qgis_iface):
        # Test with real QGIS
        pass
```

CHAPTER

FOURTEEN

CHANGELOG

14.1 0.2.1 - 2025-10-28

- Code refactoring and cleanup
- More work on PyQt6 compatibility

14.2 0.2.0 - 2025-08-04

- First public release

14.3 0.1.0-beta3 - 2025-07-25 [Unreleased]

- Compatibility with Python 3.9 for QGIS Mac
- Remove requirements files
- Update dependencies and workflows

14.4 0.1.0-beta2 - 2025-07-18 [Unreleased]

- DTM elevation value extraction added

14.5 0.1.0-beta1 - 2025-07-16 [Unreleased]

- First beta