PyDelphin Scripting Tutorial

Table of Contents

1. Overview . . . **2**
2. Current Status of the Framework . . . **2**
3. Recommended Workflow . . . **3**
4. Setup . . . **3**
5. Basic API . . . **3**
6. Example 1: Simple Variant Study . . . **4**
6.1. Defining Variants . . . 4
6.2. Loading a Project and Modifying Init Parameters . . . 5
6.3. Modifying an Initial Condition . . . 5
6.4. Writing Variants . . . 5
6.5. Writing a Manifest . . . 6
7. Creating and Running Solver Jobs . . . **6**
8. Example 2: Climate, Material, and Layer Thickness . . . **6**
8.1. Recommended Directory Structure . . . 7
8.2. Project Operations with PyDelphin . . . 7
8.3. Replacing Material Assignments . . . 7
8.4. Material Files as an Interim Solution . . . 8
8.5. Layer Thickness and Discretization . . . 8
9. Inspecting Assignments . . . **9**
10. Evaluating Results . . . **9**
10.1. Evaluating Assessment Models . . . 10
10.1.1. Moisture Accumulation . . . 10
10.1.2. Frost Damage . . . 11
10.1.3. Mould Risk Assessment for 1D Projects . . . 11
10.1.4. Additional Assessment Functions . . . 12
10.2. Solver Logs for Performance Comparisons . . . 12
10.3. Monitored Solver Runs . . . 12
10.4. Material Replacement Study . . . 13
10.5. General Combined Variant Studies . . . 13
10.6. Material/Climate Assessment Study . . . 14
10.7. Complete Evaluator Example . . . 15
10.8. Configurable Studies Using JSON . . . 15
11. Complete Test Run . . . **16**
12. Next Steps . . . **17**

1. Overview

`pydelphin` is a small Python library for reading and writing DELPHIN projects and modifying their parameters.

This tutorial covers:

- Reading, inspecting, and writing DELPHIN projects
- Modifying `Init` parameters such as duration, starting year, and climate file
- Modifying `Conditions`, such as initial and boundary conditions
- Inspecting `Assignments`, such as material, boundary, and output assignments
- Generating project variants
- Optionally running the DELPHIN solver
- Replacing and adding material references and material assignments
- Reading ASCII result files and solver logs
- Calculating basic metrics for comparing variants
- Evaluating moisture accumulation as well as frost and mould risk
- Working with discretization and `CmdDiscretise`
- Limitations of the current implementation and planned adapters

The accompanying example application is located at:

```
scripts/PyDelphin/examples/tutorial_variant_study.py
```

2. Current Status of the Framework

The current version of `pydelphin` provides the core functionality needed to generate project variants:

- `read_project(path)` reads `.d6p` files.
- `write_project(project, path)` writes `.d6p` files.
- `project.init` provides access to `Init` and `SimulationParameter`.
- `project.conditions` provides access to `Interfaces`, `InitialConditions`, `FieldConditions`, and other condition groups.
- `project.assignments` provides access to assignments such as Material, Interface, Output, and InitialCondition.
- The original XML structure is preserved. Untyped sections are not lost during reading or writing.
- `project.material_references` allows material references and assignments to be added and replaced.
- `read_d6o()` reads ASCII result files, including metadata and data rows.
- `read_tsv()` reads tabular ASCII files such as solver statistics.
- `read_solver_logs()` reads all TSV logs of a solver run together.
- `read_screenlog()` reads structured `SolverFeedback` lines as a fallback.
- `column_statistics()` calculates basic statistics and, optionally, a time integral.

- `ParallelSolverRunner`, `StudyResult`, and `StudyEvaluation` provide functionality for parallel solver runs and evaluations.
- `assess_moisture_accumulation()`, `assess_frost_risk()`, and `assess_mould_risk()` evaluate DELPHIN's internal assessment quantities.
- The standardized evaluators `PerformanceEvaluator`, `MoistureAccumulationEvaluator`, `FrostRiskEvaluator`, and `MouldRiskEvaluator` use a common interface.

3. Recommended Workflow

The recommended procedure is:

1. Create and test the base project in DELPHIN.
2. Generate project variants using Python.
3. For 1D variants, adjust layer thicknesses and the variable grid using `pydelphin.rediscretize_1d()`; for 2D/3D projects, optionally use `CmdDiscretise`.
4. Run the DELPHIN solver for all variants.
5. Collect and evaluate the results.



These steps should be kept separate. When running many simulations, it is useful to inspect the generated `.d6p` files and the manifest before starting the solver.

4. Setup

This tutorial is run from within the DELPHIN repository. For local development without installation, setting `PYTHONPATH` is sufficient:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/tutorial_variant_study.py
```

By default, the application uses the following base project:

```
scripts/PyDelphin/TestProjekte/Wand-2D_Schaumglas_1.d6p
```

By default, the generated variants are stored here:

```
scripts/PyDelphin/generated/tutorial_variants
```

The `generated/` directory is deliberately intended for automatically generated data. It should not contain manually maintained input data.

5. Basic API

Load a project using `read_project()`:

```
from pathlib import Path

from pydelphin import read_project, write_project

project = read_project(Path("TestProjekte/Wand-2D_Schaumglas_1.d6p"))
```

Frequently needed information is available directly from the project object:

```
print(project.file_version)
print(project.project_info.project_name)
print(project.discretization.dimension)
print(len(project.materials))
```

The main project sections are accessible through adapters:

```
init = project.init
conditions = project.conditions
assignments = project.assignments
```

Write a project using `write_project()`:

```
write_project(project, Path("generated/example_copy.d6p"))
```

The writer preserves the DELPHIN-compatible XML style of the input file. If a project did not have an explicit `xmlns:IBK` declaration, the writer does not add one by default.

6. Example 1: Simple Variant Study

An initial condition is varied across several values, producing multiple DELPHIN projects. To do this, the named `InitialCondition` is located in the project and its `IBK:Parameter` is modified.

6.1. Defining Variants

The example application varies two parameters:

- Climate file
- Initial moisture content of the initial condition `Ziegel nass`

```
CLIMATE_VARIANTS = [
    ("potsdam", "${Project Directory}/climate/DE-04-TRY-2010__Potsdam__Jahr_00000K0_00081m.c6b"),
    ("bremerhaven", "${Project Directory}/climate/DE-01-TRY-2010__Bremerhaven__Jahr_00000K0_00007m.c6b"),
]
MOISTURE_VALUES = [220.0, 280.0, 320.0]
SIMULATION_DURATION_A = 1.0
INITIAL_CONDITION_NAME = "Ziegel nass"
```

This produces six variants. Their names are formed from the climate location and the moisture value, e.g. `potsdam_m220.d6p`.

6.2. Loading a Project and Modifying Init Parameters

The simulation duration and climate file are stored in the `Init` section. The adapter provides direct methods for these parameters:

```
def set_simulation_duration(project, duration_a: float) -> None:
    if project.init is None:
        raise ValueError("Missing Init section.")
    project.init.set_duration(duration_a, "a")


def set_climate_file(project, climate_path: str) -> None:
    if project.init is None:
        raise ValueError("Missing Init section.")
    project.init.set_climate_data_file_path(climate_path)
```

Internally, the corresponding XML node is modified. The project object remains the source for the subsequent write operation.

6.3. Modifying an Initial Condition

The initial condition is located in the `Conditions` section. The adapter can look it up by name:

```
def set_initial_moisture(project, condition_name: str, value_kg_m3: float) -> None:
    if project.conditions is None:
        raise ValueError("Missing Conditions section.")
    condition = project.conditions.initial_condition(condition_name)
    if condition is None:
        raise ValueError(f"Missing InitialCondition '{condition_name}'.")
    condition.set_parameter("Value", value_kg_m3, "kg/m3")
```

Compared with text templates, this has the advantage of operating on the semantic structure: an `InitialCondition` named `Ziegel nass`, a parameter named `Value`, and the unit `kg/m3`.

6.4. Writing Variants

To generate the variants, the base project is read again for each variant, its values are modified, and a new `.d6p` file is written:

```
def generate_variants(base_project, output_dir, variants):
    output_dir.mkdir(parents=True, exist_ok=True)
    project_paths = []

    for variant in variants:
        project = read_project(base_project)
        set_climate_file(project, variant.climate_path)
        set_simulation_duration(project, variant.duration_a)
        set_initial_moisture(project, "Ziegel nass", variant.initial_moisture_kg_m3)

        project_path = output_dir / f"{variant.name}.d6p"
        write_project(project, project_path)
        project_paths.append(project_path)
```

```
    return project_paths
```

Reloading the base project for each variant prevents changes from one variant from inadvertently carrying over into the next.

6.5. Writing a Manifest

The example application also writes a `variants.tsv` manifest. This is useful for subsequent evaluations because the selected parameters do not need to be reconstructed from file names.

Example of `variants.tsv`

```
variant project_file    climate duration_a  initial_moisture_kg_m3
potsdam_m220    potsdam_m220.d6p    potsdam 1   220
potsdam_m280    potsdam_m280.d6p    potsdam 1   280
potsdam_m320    potsdam_m320.d6p    potsdam 1   320
bremerhaven_m220    bremerhaven_m220.d6p    bremerhaven 1   220
```

7. Creating and Running Solver Jobs

As in the previous tutorial, the DELPHIN solver is started from the command line. However, the example application only starts the solver if `--run-solver` is specified. This allows the project files to be generated and inspected first.

```
PYTHONPATH=src python3 examples/tutorial_variant_study.py   --solver /home/fechner/Delphin6/DelphinSolver   --run
-solver
```

The new API runs variants via `VariantStudy.run()`. The `max_workers` parameter limits the number of solver instances running simultaneously:

```
result = study.run(
    "/path/to/DelphinSolver",
    SolverRunConfig(wall_time_limit_s=900, progress_timeout_s=300, processes=1),
    configure=configure,
    max_workers=2,
    verbose=True,
)
```

`processes` controls the internal solver processes for a single variant, whereas `max_workers` controls the number of variants running in parallel. Both settings should be chosen together, taking memory consumption and the number of available processors into account.



For computationally intensive DELPHIN models, it is often better to run fewer solver instances in parallel and give each instance more threads using `-p=<N>`. The optimal configuration strongly depends on model size, memory bandwidth, and the amount of output data.

8. Example 2: Climate, Material, and Layer Thickness

The second example in the previous tutorial varies three parameters:

- Climate file
- Thermal conductivity of an insulation material in an `.m6` material file
- Layer thickness and subsequent discretization

With the current version of `pydelphin`, the project-related operations can already be handled cleanly through adapters. Material files are not yet read and written in a structured manner. As an interim solution, material files can still be copied as text files while the `.d6p` project file is processed using `pydelphin`.

8.1. Recommended Directory Structure

```
BaseProject/
  template.d6p
  climate/
    *.c6b
  materials/
    template.m6
    *.m6
variations/
  materials/
  *.d6p
  variants.tsv
```

The base project is prepared and tested in DELPHIN. Each variant is then written to a separate output directory. The original data remains unchanged.

8.2. Project Operations with PyDelphin

The climate file continues to be set via `project.init`:

```
project = read_project(base_project)
project.init.set_climate_data_file_path("${Project Directory}/climate/Potsdam.c6b")
```

Material references can be added directly to the project:

```
project.material_references.add_reference(
    "Schaumglas Variante",
    "${Project Directory}/materials/Schaumglas_Variante.m6",
    color="#ff865900",
    hatch_code="13",
)
```

8.3. Replacing Material Assignments

To replace an existing material with another material already registered in the project, all corresponding material assignments can be updated:

```
project.material_references.replace_assignments(
    "Schaumglas [70]",
    "Schaumglas Variante",
```

```
)
```

Two material assignments can also be swapped:

```
project.material_references.swap_assignments("Ziegel Außen", "Dämmung Innen")
```

This operation only changes the assignment references. The material definitions and their files are not overwritten.

8.4. Material Files as an Interim Solution

A simple material variation can still be implemented using a template file:

```
material_template = (base_dir / "materials" / "template.m6").read_text(encoding="utf-8")
material_text = material_template.replace("${LAMBDA}", "0.045")
(target_dir / "materials" / "casi_0.045.m6").write_text(material_text, encoding="utf-8")
```

This is deliberately only a temporary solution. A future material adapter should read, validate, and write parameters such as `LAMBDA`, `RHO`, `CE`, `MEW`, etc. in a structured manner.

8.5. Layer Thickness and Discretization

For one-dimensional projects, `pydelphin` can reconstruct material layers from the material assignments and generate a variable grid in the X direction. Material, boundary, and element assignments are adjusted to the new cell indices. The project is modified directly and can subsequently be saved using `write_project()`:

```
from pydelphin import (
    VariableDiscretizationConfig,
    layers_1d,
    read_project,
    rediscretize_1d,
    write_project,
)

project = read_project("project.d6p")
layers = layers_1d(project)
print([(layer.material, layer.thickness) for layer in layers])

rediscretize_1d(
    project,
    VariableDiscretizationConfig(
        min_width=0.002,
        max_width=0.02,
        stretch=1.2,
    ),
    layer_thicknesses=[0.02, 0.16, 0.02],
)
write_project(project, "project_rediscretized.d6p")
```

`layer_thicknesses` follows the order returned by `layers_1d()`. If it is omitted, the existing layer thicknesses are retained and only the grid is rebuilt. `min_width` and `max_width` limit the cell widths; `stretch` controls symmetric grid refinement at layer boundaries. The function is currently intentionally limited to 1D projects. For complex 2D and 3D geometries, `CmdDiscretise` remains the appropriate tool.

9. Inspecting Assignments

`Assignments` link material, boundary, initial-condition, and output objects to element ranges or boundary faces. The adapter returns all assignments as a list:

```
project = read_project("TestProjekte/TestWand_1.d6p")

for assignment in project.assignments.by_type("Material"):
    print(assignment.reference, assignment.location, assignment.range_values)
```

An individual assignment can be found by type and reference:

```
inside = project.assignments.first("Interface", "Innen")
print(inside.location)
```

The `Range` values are preserved as a list of integers. In 2D projects, there are four values; in 3D projects, there are usually six:

```
print(inside.dimension)
print(inside.range_values)
```

10. Evaluating Results

ASCII result files are read using `read_d6o()`. The header is retained as metadata. The data rows contain time in the first column, followed by the values for the indices listed in the header:

```
from pydelphin import read_d6o

result = read_d6o("results/Temperaturprofil.d6o")
print(result.metadata["QUANTITY"])
print(result.metadata["VALUE_UNIT"])
print(result.indices)
print(result.rows[0])
```

Solver and integrator statistics are read as TSV tables:

```
from pydelphin import read_solver_logs

logs = read_solver_logs("log")
progress = logs.progress
if progress is not None and progress.rows:
    last = progress.rows[-1]
    print("Simulationszeit:", last[0], "s")
    print("Rechenzeit:", last[1], "s")
    print("Fortschritt:", last[2], "%")

integrator = logs.tables.get("integrator_cvode_stats.tsv")
if integrator is not None:
    print(integrator.headers)
    print(integrator.rows[-1])
```

Basic metrics can be calculated using `column_statistics()`:

```
from pydelphin.results.analysis import column_statistics

values = [row[1] for row in result.rows]
times = [row[0] for row in result.rows]
stats = column_statistics(values, times)
print(stats.minimum, stats.maximum, stats.mean, stats.integral)
```

For a variant study, the recommended procedure is:

1. Read the variant manifest.
2. Determine the expected result files for each variant.
3. Read the `.d6o` files.
4. Extract relevant time points or metrics.
5. Save the results together with the variant parameters as TSV/CSV.

The initial evaluation functionality deliberately supports only ASCII files. Binary output files are not part of this API.

10.1. Evaluating Assessment Models

The assessment functions use the internal ASCII result files written by DELPHIN. The required assessments must already be enabled when generating the project. For example, the following assessment type is enabled for mould risk:

```
from pydelphin import enable_assessments, read_project, write_project

project = read_project("project.d6p")
enable_assessments(project, mould_growth=True)
write_project(project, "project_with_assessments.d6p")
```

10.1.1. Moisture Accumulation

By default, moisture accumulation is read from `Internal_MoistMassIntegral.d6o`. If this file is missing or empty, the hourly version `Internal_MoistMassIntegral_1h.d6o` is used. The function returns the moisture mass at the end of each year, the relative change between years, and an assessment of whether the change is below the specified threshold:

```
from pydelphin import assess_moisture_accumulation

moisture = assess_moisture_accumulation(
    "results",
    threshold_percentage=1.0,
)
print("Jahresmassen:", moisture.annual_masses)
print("Relative Änderungen:", moisture.relative_changes)
print("Stabiles Jahr:", moisture.stable_year)
print("Längere Rechnung nötig:", moisture.needs_longer_simulation)
```

With `from_total=False`, the files `Internal_MoistMassIntegral_1h_<Index>.d6o` for selected layers can be evaluated instead. The supplied `layers` are the numerical layer indices. The function requires a sufficient number of complete

simulation years.

10.1.2. Frost Damage

For the frost criterion, `IceCriteriaDIN4108` must be enabled in the project. `assess_frost_risk()` then reads `Internal_IceCheckIntegral.d6o` and the G6A geometry referenced through `GEO_FILE`. Positive values are grouped into contiguous risk regions. The evaluation supports 1D and 2D geometries:

```
from pydelphin import assess_frost_risk

frost = assess_frost_risk("results")
print("Frostschädigung vorhanden:", frost.has_risk)
for area in frost.areas:
    print(area.materials, area.area, area.indices)
```

Each region contains the affected element indices, the materials involved, its location, and its area. The function evaluates the last available time point in the frost integral output. Assessment types can be enabled together:

```
enable_assessments(
    project,
    moisture_accumulation=True,
    ice_criteria=True,
    mould_growth=True,
)
```

10.1.3. Mould Risk Assessment for 1D Projects

`assess_mould_risk()` evaluates only one-dimensional projects, because only in this case is the interior surface to be assessed unambiguously defined. The function reads:

- `Internal_MouldCheckTemperature.d6o`
- `Internal_MouldCheckRH.d6o`

By default, the last simulation year is evaluated. The function uses the same underlying models as the DELPHIN output report:

```
from pydelphin import assess_mould_risk

mould = assess_mould_risk(
    "results",
    material_class=1,
    surface_class=1,
    decline_class=4,
)

print("Längste Zeit über 80 % RH:", mould.rh80_longest_continuous_s)
print("Zeit über Isoplethen-Limit:", mould.isopleth_limit_duration_s)
print("Zeit mit VTT-Index >= 1:", mould.vtt_duration_index_1_s)
print("Zeit mit VTT-Index >= 2:", mould.vtt_duration_index_2_s)
print("Maximaler VTT-Index:", mould.vtt_max_index)
```

Durations are returned in seconds. The results include the following criteria:

- `rh80_longest_continuous_s`: longest uninterrupted period with relative humidity above 80%.
- `isopleth_limit_duration_s`: total time above the red `Limit for germination` curve of the isopleth model.
- `isopleth_limit_longest_continuous_s`: longest uninterrupted period above this curve.
- `vtt_duration_index_1_s` and `vtt_duration_index_2_s`: total time with a VTT mould index of at least 1 or 2, respectively.
- `vtt_longest_index_1_s` and `vtt_longest_index_2_s`: longest uninterrupted period exceeding the respective thresholds.
- `vtt_max_index`: maximum VTT index reached.

The choice of VTT classes corresponds to the settings in the DELPHIN report. It can be adjusted using `material_class`, `surface_class`, and `decline_class`. Values from 1 to 4 correspond to the four classes in the VTT model. Multidimensional results or result files with multiple surface values are rejected with `AssessmentDataError`.

10.1.4. Additional Assessment Functions

`assess_moisture_accumulation()` evaluates annual moisture masses. `assess_frost_risk()` uses `Internal_IceCheckIntegral.d6o` and combines contiguous frost-risk regions. Both functions are intended for their respective project geometries supported by DELPHIN; mould assessment remains intentionally limited to 1D.

10.2. Solver Logs for Performance Comparisons

Each solver run should have its own `log` directory. `progress.tsv` provides simulation time, actual computation time, and progress. Depending on the integrator and solution method, additional files such as `integrator_cvode_stats.tsv` or `LES_iterative_stats.tsv` are generated.

Variant comparisons should include at least the following values:

- Variant name from the manifest
- Simulation time and actual computation time
- Last recorded progress value
- Number of integration steps
- Right-hand-side evaluations and linear solver evaluations, if available
- Convergence and error counters, if available

The TSV tables are deliberately read generically because their columns may vary with the integrator and solution method.

10.3. Monitored Solver Runs

For slow or potentially non-converging variants, the solver should not be started directly using `subprocess.run()`. Instead, `run_solver()` monitors runtime and progress in `progress.tsv`:

```
from pydelphin import SolverRunConfig, run_solver

config = SolverRunConfig(
    wall_time_limit_s=1800,
    progress_timeout_s=300,
    processes=2,
```

```
)
result = run_solver("variant/project.d6p", "/path/to/DelphinSolver", config)
print(result.status.value, result.wall_time_s, result.termination_reason)
```

Possible status values are `success`, `solver_error`, `wall_time_limit`, `simulation_progress_timeout`, and `missing_progress`.

10.4. Material Replacement Study

A one-at-a-time study successively replaces each material used in the project with a robust reference material. A separate project directory is created for each variant; the base project remains unchanged:

```
from pydelphin.studies import MaterialReplacementStudy
from pydelphin import SolverRunConfig

study = MaterialReplacementStudy(
    base_project="project/project.d6p",
    replacement_name="Robustes Referenzmaterial",
    replacement_path="${Project Directory}/materials/reference.m6",
    output_dir="material_study",
)
results = study.run(
    "/path/to/DelphinSolver",
    SolverRunConfig(wall_time_limit_s=1800, progress_timeout_s=300),
)
```

`run()` returns a `StudyResult`. For each variant, the `material_study.tsv` file records the material name, status, elapsed wall-clock time, reached simulation time, progress source, and completion status. A timeout is a useful study result and is not silently swallowed as a Python exception.

The study can be evaluated directly using the performance evaluator:

```
from pydelphin import PerformanceEvaluator

evaluation = results.evaluate({"performance": PerformanceEvaluator()})
rows = evaluation.rows()
```

10.5. General Combined Variant Studies

For studies involving several independent parameters, `VariantStudy` is available. The API automatically generates the Cartesian product of all named options. Project-specific modifications are performed in a configuration function:

```
from pydelphin.studies import VariantOption, VariantStudy

study = VariantStudy(
    base_project="TestProjekte/TestWand_1.d6p",
    output_dir="generated/combined_study",
)
study.add_parameter("material", [
    VariantOption("ZA", "AltbauziegelDresdenZA_1833.m6"),
    VariantOption("ZC", "AltbauziegelDresdenZC_1834.m6"),
])
study.add_parameter("climate", [
    ("Essen", "Essen.c6b"),
    ("Stoetten", "Stoetten.c6b"),
```

```
])

def configure(project, values, variant_directory):
    # Copy material and climate files and modify the project here.
    # values["material"] and values["climate"] contain the selected values.
    pass

variants = study.generate(configure)
```

This example produces four variants. The parameter combinations are saved in `variants.tsv` together with the respective project paths. In addition to the project and parameter values, the configuration function receives the target directory so that variant-specific material or climate files can be copied.

Solver execution and evaluation can be combined directly:

```
from pydelphin import FrostRiskEvaluator, SolverRunConfig

result = study.run(
    "/path/to/DelphinSolver",
    SolverRunConfig(wall_time_limit_s=900),
    configure=configure,
    verbose=True,
)
evaluation = result.evaluate({"frost": FrostRiskEvaluator()})
rows = evaluation.rows()
```

If the variants were initially only generated, they can later be loaded again using `reuse_existing=True` or simulated directly using the saved project paths.

10.6. Material/Climate Assessment Study

The example `examples/material_climate_assessment_study.py` combines several material variants with several climate locations. It is based on `TestProjekte/TestWand_1.d6p`. The existing brick material is replaced with five current variants from the `Altbauziegel Dresden Z⋯` group. An unchanged reference variant is also generated for each climate location.

Initially, 30 projects are generated without running the solver:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/material_climate_assessment_study.py
```

The TRY-2011 locations used are Essen, Potsdam, Rostock-Warnemünde, Hamburg-Fuhlsbüttel, and Stötten. For each variant, the assessment types `MoistureAccumulation`, `IceCriteriaDIN4108`, and `MoldGrowth` are enabled.

With monitored solver execution, the projects are simulated and then evaluated using the three assessment models:

```
PYTHONPATH=src python3 examples/material_climate_assessment_study.py \
    --run-solver \
    --solver /path/to/DelphinSolver \
    --wall-time-limit 1800 \
    --progress-timeout 300 \
    --max-workers 2
```

For each variant, `material_climate_assessment.tsv` contains the solver status and assessment values. These include moisture accumulation, the number and area of frost-risk regions, and durations above 80% RH, the isopleth threshold, and VTT indices 1 and 2.

10.7. Complete Evaluator Example

The example `examples/study_with_evaluators.py` demonstrates the complete workflow using the new API. It generates variants for two climate locations, runs them in parallel, and uses all four standardized evaluators:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/study_with_evaluators.py \
    --solver /path/to/DelphinSolver \
    --max-workers 2
```

The combined evaluation is written to `evaluation.tsv`. Performance evaluation results are preserved even for interrupted solver runs. If result files needed for an engineering assessment are missing, an error is recorded in the evaluation row for that evaluator only.

10.8. Configurable Studies Using JSON

Recurring variant studies do not require a separate Python script to be modified every time. The script `examples/run_configured_study.py` reads a JSON file and executes the study described in it. Configuration is therefore separated from execution and could, for example, be generated by a future graphical user interface.

Currently, the `material_replacement` study type is supported. First, a reference variant is created; then each material used in the project is individually replaced by the specified replacement material. The replacement material itself is not replaced with itself again.

A complete example is available in `examples/material_replacement_study.json`:

```
{
  "study_type": "material_replacement",
  "project": "../TestProjekte/TestWand_1.d6p",
  "output_dir": "../generated/material-replacement-study",
  "solver": "/path/to/DelphinSolver",
  "replacement_material": "Altbauziegel Dresden ZG [1836]",
  "replacement_file": "../../../data/DB_materials/AltbauziegelDresdenZG_1836.m6",
  "max_workers": 1,
  "solver_run": {
    "wall_time_limit_s": 900,
    "progress_timeout_s": 600,
    "poll_interval_s": 1,
    "processes": 1
  },
  "evaluators": ["performance", "moisture", "frost", "mould"],
  "evaluation_file": "study_evaluation.tsv",
  "reuse_existing": true
}
```

Paths are interpreted relative to the directory containing the JSON file. This allows the configuration file to be moved together with the associated study files. Absolute paths are also supported.

The study is started from the `scripts/PyDelphin` directory:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/run_configured_study.py \
    examples/material_replacement_study.json
```

The main configuration fields are:

- `project`: base project from which variants are generated.
- `output_dir`: target directory for variants and evaluations.
- `solver`: path to the DELPHIN solver executable.
- `replacement_material`: name of the replacement material in the project.
- `replacement_file`: material file for the replacement material.
- `max_workers`: maximum number of solver instances running simultaneously.
- `solver_run`: solver runtime, progress, and process settings.
- `evaluators`: list of standardized evaluators.
- `reuse_existing`: reuse previously generated variants or generate new ones.

The currently available evaluators are:

- `performance` for solver status, elapsed wall-clock time, simulation time, and progress source,
- `moisture` for moisture accumulation,
- `frost` for frost damage assessment,
- `mould` for mould risk assessment in one-dimensional projects.

During execution, a message is printed immediately when each variant starts:

```
[1/13] Starte Variante: baseline
```

After completion, the solver status and elapsed wall-clock time are also displayed. This makes it possible to see that the process is still running, even during long simulations or for variants that do not produce result files early in the run. The existing `material_study_3d_bsd.py` script remains available for specially customized studies.

11. Complete Test Run

The current example application can be started without running the solver:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/tutorial_variant_study.py
```

Expected output:

```
Wrote 6 variants to .../generated/tutorial_variants
Wrote manifest to .../generated/tutorial_variants/variants.tsv
```

The unit tests additionally verify that the generated projects can be read again:

```
cd /home/fechner/Delphin6
PYTHONDONTWRITEBYTECODE=1 PYTHONPATH=scripts/PyDelphin/src   python3 -m unittest discover scripts/PyDelphin/tests
```

12. Next Steps

The most important planned extensions are:

- Optional integration with `CmdDiscretise` for 2D and 3D projects. For 1D projects, variable discretization is already available through `rediscretize_1d()`.
- Robust resumption of previously completed variants.
- Additional plausibility checks for result files, especially an unambiguous mapping between the project, geometry, and assessment outputs.

The current foundation supports these extensions because project files can already be read with minimal loss of information, modified at the semantic level, written, and read again.