PyDelphin Scripting Tutorial

Inhaltsverzeichnis

1. Überblick . . . **2**
2. Aktueller Stand des Frameworks . . . **2**
3. Empfohlene Arbeitsweise . . . **3**
4. Vorbereitung . . . **3**
5. Grundlegende API . . . **4**
6. Beispiel 1: einfache Variantenstudie . . . **4**
6.1. Varianten definieren . . . 4
6.2. Projekt laden und Init-Parameter ändern . . . 5
6.3. Anfangsbedingung ändern . . . 5
6.4. Varianten schreiben . . . 5
6.5. Manifest schreiben . . . 6
7. Solver-Jobs erzeugen und starten . . . **6**
8. Beispiel 2: Klima, Material und Schichtdicke . . . **7**
8.1. Empfohlene Struktur . . . 7
8.2. Projektteil mit PyDelphin . . . 7
8.3. Materialzuweisungen austauschen . . . 7
8.4. Materialdatei als Uebergangslösung . . . 8
8.5. Schichtdicke und Diskretisierung . . . 8
9. Assignments inspizieren . . . **9**
10. Ergebnisse auswerten . . . **9**
10.1. Bewertungsmodelle auswerten . . . 10
10.1.1. Feuchteakkumulation . . . 10
10.1.2. Frostschädigung . . . 11
10.1.3. Schimmelbewertung für 1D-Projekte . . . 11
10.1.4. Weitere Bewertungsfunktionen . . . 12
10.2. Solver-Logs für Performancevergleiche . . . 12
10.3. Überwachter Solverlauf . . . 13
10.4. Material-Ersatzstudie . . . 13
10.5. Allgemeine kombinierte Variantenstudien . . . 14
10.6. Material-/Klima-Bewertungsstudie . . . 14
10.7. Vollständiges Evaluator-Beispiel . . . 15
10.8. Konfigurierbare Studien über JSON . . . 15
11. Vollständiger Testlauf . . . **17**
12. Nächste Schritte . . . **17**

1. Überblick

`pydelphin` ist eine kleine Python-Bibliothek welche DELPHIN Projekte lesen und schreiben sowie Parameter verändern kann.

Dieses Tutorial beschreibt:

- Lesen, Inspizieren und Schreiben von DELPHIN-Projekten
- Aendern von `Init`-Parametern wie Dauer, Startjahr und Klimadatei
- Aendern von `Conditions`, z.B. Anfangsbedingungen und Randbedingungen
- Auswerten von `Assignments`, z.B. Material-, Rand- und Ausgabezuweisungen
- Erzeugen von Projektvarianten
- Optionales Starten des DELPHIN-Solvers
- Austauschen und Hinzufügen von Materialreferenzen und Materialzuweisungen
- Einlesen von ASCII-Ergebnisdateien und Solver-Logs
- Ermitteln grundlegender Kennwerte für Variantenvergleiche
- Auswerten der Feuchteakkumulation sowie der Frost- und Schimmelbewertung
- Umgang mit Diskretisierung und `CmdDiscretise`
- Grenzen des aktuellen Standes und die nächsten Adapter

Die zugehörige Testanwendung liegt unter:

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

2. Aktueller Stand des Frameworks

Der aktuelle Stand von `pydelphin` deckt den Kern ab, der für das Erzeugen von Variantenprojekten notwendig ist:

- `read_project(path)` liest `.d6p`-Dateien.
- `write_project(project, path)` schreibt `.d6p`-Dateien wieder.
- `project.init` liefert Zugriff auf `Init` und `SimulationParameter`.
- `project.conditions` liefert Zugriff auf `Interfaces`, `InitialConditions`, `FieldConditions` und weitere Bedingungsgruppen.
- `project.assignments` liefert Zugriff auf Zuweisungen wie Material, Interface, Output und InitialCondition.
- Die Original-XML-Struktur bleibt erhalten. Nicht typisierte Abschnitte gehen beim Lesen/Schreiben nicht verloren.
- `project.material_references` ermöglicht das Hinzufügen und Austauschen von Materialreferenzen und Materialzuweisungen.
- `read_d6o()` liest ASCII-Ergebnisdateien mit Metadaten und Datenzeilen.
- `read_tsv()` liest tabellarische ASCII-Dateien wie die Solverstatistiken.
- `read_solver_logs()` liest alle TSV-Logs eines Solverlaufs gemeinsam.

- `read_screenlog()` liest strukturierte `SolverFeedback`-Zeilen als Fallback.
- `column_statistics()` berechnet grundlegende Kennwerte und optional ein Zeitintegral.
- `ParallelSolverRunner`, `StudyResult` und `StudyEvaluation` bündeln parallele Solverläufe und Auswertungen.
- `assess_moisture_accumulation()`, `assess_frost_risk()` und `assess_mould_risk()` werten DELPHIN-interne Bewertungsgrößen aus.
- Die standardisierten Evaluatoren `PerformanceEvaluator`, `MoistureAccumulationEvaluator`, `FrostRiskEvaluator` und `MouldRiskEvaluator` verwenden eine gemeinsame Schnittstelle.

3. Empfohlene Arbeitsweise

Man sollte folgendermaßen vorgehen:

1. Basisprojekt in DELPHIN erstellen und testen.
2. Variantenprojekte mit Python erzeugen.
3. Bei 1D-Varianten die Schichtdicken und das variable Gitter mit `pydelphin.rediscretize_1d()` anpassen; für 2D/3D-Projekte optional `CmdDiscretise` verwenden.
4. DELPHIN-Solver für alle Varianten starten.
5. Ergebnisse einsammeln und auswerten.



Die Schritte sollten getrennt bleiben. Wenn viele Simulationen laufen, ist es praktisch, die erzeugten `.d6p`-Dateien und ein Manifest zuerst zu prüfen, bevor der Solver gestartet wird.

4. Vorbereitung

Das Tutorial wird aus dem DELPHIN-Repository heraus ausgeführt. Für eine lokale Entwicklung ohne Installation reicht `PYTHONPATH`:

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

Die Anwendung nutzt standardmässig dieses Basisprojekt:

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

Die erzeugten Varianten werden standardmässig hier abgelegt:

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

Das Verzeichnis `generated/` ist bewusst als Ziel für automatisch erzeugte Daten gedacht. Es sollte keine manuell gepflegten Eingabedaten enthalten.

5. Grundlegende API

Ein Projekt wird mit `read_project()` geladen:

```
from pathlib import Path

from pydelphin import read_project, write_project

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

Häufig benötigte Informationen sind direkt am Projektobjekt verfügbar:

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

Die wichtigsten Projektabschnitte sind als Adapter erreichbar:

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

Ein Projekt wird mit `write_project()` geschrieben:

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

Der Writer erhält den DELPHIN-kompatiblen XML-Stil der Eingabedatei. Wenn ein Projekt keine explizite `xmlns:IBK` -Deklaration hatte, wird diese beim Schreiben standardmässig auch nicht neu eingefügt.

6. Beispiel 1: einfache Variantenstudie

Eine Anfangsbedingung wird in mehreren Stufen variiert, daraus werden mehrere DELPHIN-Projekte erzeugt. Dazu wird die benannte `InitialCondition` im Projekt gesucht und ihr `IBK:Parameter` angepasst.

6.1. Varianten definieren

Die Testanwendung variiert zwei Dinge:

- Klimadatei
- Anfangsfeuchte der Initialbedingung `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"
```

Daraus entstehen sechs Varianten. Die Namen werden aus Klimastandort und Feuchtewert gebildet, z.B. `potsdam_m220.d6p`.

6.2. Projekt laden und Init-Parameter ändern

Die Simulationsdauer und die Klimadatei liegen im `Init`-Abschnitt. Der Adapter stellt dafür direkte Methoden bereit:

```
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)
```

Intern wird dabei der passende XML-Knoten geändert. Das Projektobjekt bleibt weiterhin die Quelle für den anschliessenden Writer.

6.3. Anfangsbedingung ändern

Die Anfangsbedingung liegt im `Conditions`-Abschnitt. Der Adapter kann sie nach Namen suchen:

```
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")
```

Der Vorteil gegenüber Text-Templates ist, dass das Skript mit der fachlichen Struktur arbeitet: `InitialCondition` mit Name `Ziegel nass`, Parameter `Value`, Einheit `kg/m3`.

6.4. Varianten schreiben

Die eigentliche Erzeugung der Varianten liest das Basisprojekt für jede Variante neu ein, ändert die Werte und schreibt eine neue `.d6p`-Datei:

```
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
```

Das erneute Lesen des Basisprojekts pro Variante verhindert, dass Aenderungen von einer Variante versehentlich in die nächste Variante übernommen werden.

6.5. Manifest schreiben

Zusätzlich schreibt die Testanwendung ein `variants.tsv`-Manifest. Das ist für spätere Auswertung wichtig, weil die gewählten Parameter dann nicht aus dem Dateinamen rekonstruiert werden müssen.

Beispiel für `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. Solver-Jobs erzeugen und starten

Wie im alten Tutorial wird der DELPHIN-Solver über die Kommandozeile gestartet. Die Testanwendung startet den Solver aber nur, wenn `--run-solver` angegeben wird. Dadurch kann man die Projektdateien zuerst erzeugen und prüfen.

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

Die neue API startet die Varianten über `VariantStudy.run()`. Mit `max_workers` wird die Anzahl gleichzeitig laufender Solver begrenzt:

```
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` steuert die internen Solverprozesse einer Variante, `max_workers` dagegen die Anzahl paralleler Varianten. Beide Werte müssen im Hinblick auf Speicherverbrauch und Prozessoranzahl gemeinsam gewählt werden.



Bei rechenintensiven DELPHIN-Modellen ist es oft sinnvoller, weniger parallele Solverprozesse zu starten und jedem Prozess mehr Threads mit `-p=<N>` zu geben. Die optimale Einstellung hängt stark von Modellgrösse, Speicherbandbreite und Ausgabemenge ab.

8. Beispiel 2: Klima, Material und Schichtdicke

Das zweite Beispiel im alten Tutorial variiert drei Dinge:

- Klimadatei
- Wärmeleitfähigkeit eines Dämmstoffs in einer `.m6`-Materialdatei
- Schichtdicke und anschliessende Diskretisierung

Mit dem aktuellen Stand von `pydelphin` kann der Projektteil bereits sauber über Adapter abgebildet werden. Materialdateien werden noch nicht strukturiert gelesen und geschrieben. Für eine übergangslösung kann man Materialdateien weiterhin als Textdateien kopieren und nur die `.d6p`-Projektdatei über `pydelphin` bearbeiten.

8.1. Empfohlene Struktur

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

Das Basisprojekt wird in DELPHIN vorbereitet und getestet. Danach wird die Variante in ein separates Zielverzeichnis geschrieben. Originaldaten bleiben unverändert.

8.2. Projektteil mit PyDelphin

Die Klimadatei wird weiterhin über `project.init` gesetzt:

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

Materialreferenzen können direkt im Projekt ergänzt werden:

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

8.3. Materialzuweisungen austauschen

Soll ein vorhandenes Material durch ein anderes bereits im Projekt registriertes Material ersetzt werden, kann dies für alle Materialzuweisungen erfolgen:

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

Zwei Materialzuweisungen können auch getauscht werden:

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

Die Änderung betrifft nur die Assignment-Referenzen. Die Materialdefinitionen und deren Dateien werden nicht überschrieben.

8.4. Materialdatei als Uebergangslösung

Eine einfache Materialvariation kann weiterhin über eine Template-Datei erfolgen:

```
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")
```

Das ist bewusst nur eine Zwischenlösung. Der spätere Materialadapter sollte Parameter wie `LAMBDA`, `RHO`, `CE`, `MEW` usw. strukturiert lesen, prüfen und schreiben.

8.5. Schichtdicke und Diskretisierung

Für eindimensionale Projekte kann `pydelphin` die Materialschichten aus den Materialzuweisungen rekonstruieren und ein variables X-Gitter erzeugen. Dabei werden die Material-, Rand- und Elementzuweisungen an die neuen Zellindizes angepasst. Das Projekt wird direkt verändert und kann anschließend mit `write_project()` gespeichert werden:

```
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` folgt der Reihenfolge von `layers_1d()`. Wird es nicht angegeben, bleiben die vorhandenen Schichtdicken erhalten und nur das Gitter wird neu aufgebaut. `min_width` und `max_width` begrenzen die Zellbreiten; `stretch` steuert die symmetrische Verfeinerung an den Schichtgrenzen. Die Funktion ist derzeit bewusst auf 1D-Projekte beschränkt. Für komplexe 2D- und 3D-Geometrien bleibt `CmdDiscretise` das geeignete Werkzeug.

9. Assignments inspizieren

`Assignments` verbinden Material-, Rand-, Anfangs- und Ausgabeobjekte mit Elementbereichen oder Randseiten. Der Adapter liefert alle Zuweisungen als Liste:

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

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

Eine einzelne Zuweisung kann nach Typ und Referenz gesucht werden:

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

Die `Range`-Werte bleiben als Integer-Liste erhalten. In 2D-Projekten sind das vier Werte, in 3D-Projekten üblicherweise sechs Werte:

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

10. Ergebnisse auswerten

ASCII-Ergebnisdateien werden mit `read_d6o()` gelesen. Der Header bleibt als Metadaten erhalten; die Datenzeilen enthalten die Zeit in der ersten Spalte und anschließend die Werte der im Header aufgeführten Indizes:

```
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- und Integratorstatistiken werden als TSV-Tabellen eingelesen:

```
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])
```

Grundlegende Kennwerte können mit `column_statistics()` bestimmt werden:

```
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)
```

Für eine Variantenstudie ist die empfohlene Struktur:

1. Variantenmanifest lesen.
2. Für jede Variante die erwarteten Ergebnisdateien bestimmen.
3. `.d6o`-Dateien lesen.
4. Relevante Zeitpunkte oder Kennwerte extrahieren.
5. Ergebnisse zusammen mit den Variantenparametern als TSV/CSV speichern.

Die erste Auswertungsstufe unterstützt bewusst nur ASCII-Dateien. Binäre Ausgabedateien sind nicht Bestandteil dieser API.

10.1. Bewertungsmodelle auswerten

Die Bewertungsfunktionen verwenden die von DELPHIN geschriebenen internen ASCII-Ergebnisdateien. Die benötigten Assessments müssen bereits beim Erzeugen des Projekts aktiviert sein. Für die Schimmelbewertung wird beispielsweise folgender Assessment-Typ eingetragen:

```
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. Feuchteakkumulation

Die Feuchteakkumulation wird standardmäßig aus `Internal_MoistMassIntegral.d6o` gelesen. Falls diese Datei nicht vorhanden oder leer ist, wird die stündliche Variante `Internal_MoistMassIntegral_1h.d6o` verwendet. Die Funktion liefert die Feuchtemasse an den Jahresenden, die relative Änderung zwischen den Jahren und eine Einschätzung, ob die Änderung unter dem Grenzwert liegt:

```
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)
```

Mit `from_total=False` können stattdessen die Dateien `Internal_MoistMassIntegral_1h_<Index>.d6o` der ausgewählten Schichten ausgewertet werden. Die übergebenen `layers` sind dabei die numerischen Schichtindizes. Die Funktion setzt eine ausreichende Anzahl vollständiger Simulationsjahre voraus.

10.1.2. Frostschädigung

Für das Frostkriterium muss im Projekt `IceCriteriaDIN4108` aktiviert sein. `assess_frost_risk()` liest anschließend `Internal_IceCheckIntegral.d6o` und die darin über `GEO_FILE` referenzierte G6A-Geometrie. Positive Werte werden zu zusammenhängenden Risikobereichen gruppiert. Die Auswertung unterstützt 1D- und 2D-Geometrien:

```
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)
```

Jeder Bereich enthält die betroffenen Elementindizes, die beteiligten Materialien, seine Lage und seine Fläche. Die Funktion wertet den letzten verfügbaren Zeitpunkt der Frost-Integralausgabe aus. Die Assessment-Typen können gemeinsam aktiviert werden:

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

10.1.3. Schimmelbewertung für 1D-Projekte

`assess_mould_risk()` wertet ausschließlich eindimensionale Projekte aus, da nur dort die zu prüfende innere Oberfläche eindeutig feststeht. Die Funktion liest:

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

Ausgewertet wird standardmäßig das letzte Simulationsjahr. Die Funktion verwendet dabei dieselben Modellgrundlagen wie der DELPHIN-Ausgabereport:

```
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)
```

Die Zeitangaben werden in Sekunden zurückgegeben. Die Ergebnisse enthalten folgende Kriterien:

- `rh80_longest_continuous_s`: längste zusammenhängende Zeitspanne mit mehr als 80 % relativer Feuchte.
- `isopleth_limit_duration_s`: gesamte Zeit über der roten `Limit for germination`-Kurve des Isoplethenmodells.
- `isopleth_limit_longest_continuous_s`: längste zusammenhängende Zeit über dieser Kurve.
- `vtt_duration_index_1_s` und `vtt_duration_index_2_s`: gesamte Zeit mit einem VTT-Schimmelindex von mindestens 1 bzw. 2.
- `vtt_longest_index_1_s` und `vtt_longest_index_2_s`: jeweils längste zusammenhängende Überschreitung.
- `vtt_max_index`: maximal erreichter VTT-Index.

Die VTT-Klassenauswahl entspricht den Einstellungen des DELPHIN-Reports. Sie kann über `material_class`, `surface_class` und `decline_class` angepasst werden. Die Werte 1 bis 4 entsprechen jeweils den vier Klassen im VTT-Modell. Ein mehrdimensionales Ergebnis oder eine Ergebnisdatei mit mehreren Oberflächenwerten wird mit `AssessmentDataError` abgewiesen.

10.1.4. Weitere Bewertungsfunktionen

`assess_moisture_accumulation()` wertet die jährlichen Feuchtemassen aus. `assess_frost_risk()` verwendet `Internal_IceCheckIntegral.d6o` und fasst zusammenhängende Frost-Risikobereiche zusammen. Beide Funktionen sind für die jeweiligen von DELPHIN unterstützten Projektgeometrien vorgesehen; die Schimmelbewertung bleibt bewusst auf 1D beschränkt.

10.2. Solver-Logs für Performancevergleiche

Jeder Solverlauf sollte sein eigenes `log`-Verzeichnis erhalten. `progress.tsv` liefert Simulationszeit, reale Rechenzeit und Fortschritt. Je nach Integrator und Lösungsverfahren kommen Dateien wie `integrator_cvode_stats.tsv` oder `LES_iterative_stats.tsv` hinzu.

Für Variantenvergleiche sollten mindestens folgende Werte übernommen werden:

- Variantenname aus dem Manifest
- Simulationszeit und reale Rechenzeit
- letzter Fortschrittswert
- Anzahl der Integrationsschritte
- rechte Seiten- und lineare Löserauswertungen, sofern vorhanden
- Konvergenz- und Fehlerzähler, sofern vorhanden

Die TSV-Tabellen werden absichtlich generisch eingelesen, da die Spalten je nach Integrator und Lösungsverfahren variieren können.

10.3. Überwachter Solverlauf

Für langsame oder möglicherweise nicht konvergierende Varianten sollte der Solver nicht direkt mit `subprocess.run()` gestartet werden. `run_solver()` überwacht die Laufzeit und den Fortschritt 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)
```

Mögliche Statuswerte sind `success`, `solver_error`, `wall_time_limit`, `simulation_progress_timeout` und `missing_progress`.

10.4. Material-Ersatzstudie

Eine One-at-a-Time-Studie ersetzt nacheinander jedes verwendete Material durch ein robustes Referenzmaterial. Für jede Variante wird ein eigenes Projektverzeichnis erzeugt; das Basisprojekt bleibt unverändert:

```
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()` liefert ein `StudyResult`. Die Datei `material_study.tsv` enthält pro Variante den Materialnamen, den Status, die Echtzeit, die erreichte Simulationszeit, die Fortschrittsquelle und den Vollständigkeitsstatus. Ein Timeout ist dabei ein verwertbares Studienergebnis und wird nicht als Python-Ausnahme verschluckt.

Die Studie kann direkt mit dem Performance-Evaluator ausgewertet werden:

```
from pydelphin import PerformanceEvaluator

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

10.5. Allgemeine kombinierte Variantenstudien

Für Studien mit mehreren voneinander unabhängigen Parametern steht `VariantStudy` zur Verfügung. Die API erzeugt automatisch das kartesische Produkt aller benannten Optionen. Fachliche Änderungen am Projekt werden über eine Konfigurationsfunktion vorgenommen:

```
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):
    # Material- und Klimadateien kopieren und Projekt hier anpassen.
    # values["material"] und values["climate"] enthalten die gewählten Werte.
    pass

variants = study.generate(configure)
```

Im Beispiel entstehen vier Varianten. Die Parameterkombinationen werden in `variants.tsv` zusammen mit dem jeweiligen Projektpfad gespeichert. Die Konfigurationsfunktion erhält neben dem Projekt und den Parameterwerten auch das Zielverzeichnis, damit variantenspezifische Material- oder Klimadateien kopiert werden können.

Solverlauf und Auswertung können direkt verbunden werden:

```
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()
```

Wenn die Varianten zunächst nur erzeugt wurden, können sie später mit `reuse_existing=True` erneut geladen oder direkt über die gespeicherten Projektpfade gerechnet werden.

10.6. Material-/Klima-Bewertungsstudie

Das Beispiel `examples/material_climate_assessment_study.py` kombiniert mehrere Materialvarianten mit mehreren Klimastandorten. Als Grundlage dient `TestProjekte/TestWand_1.d6p`. Der vorhandene Ziegel wird dabei durch fünf aktuelle Varianten der Gruppe `Altbauziegel Dresden Z⋯` ersetzt. Zusätzlich wird für jeden Klimastandort eine unveränderte Referenzvariante erzeugt.

Ohne Solverlauf werden zunächst 30 Projekte erzeugt:

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

Die verwendeten TRY-2011-Standorte sind Essen, Potsdam, Rostock-Warnemünde, Hamburg-Fuhlsbüttel und Stötten. Für jede Variante werden die Assessment-Typen `MoistureAccumulation`, `IceCriteriaDIN4108` und `MoldGrowth` aktiviert.

Mit einem überwachten Solverlauf werden die Projekte berechnet und anschließend die drei Bewertungsmodelle ausgewertet:

```
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
```

Die Datei `material_climate_assessment.tsv` enthält pro Variante den Solverstatus und die Bewertungswerte. Dazu gehören die Feuchteakkumulation, die Anzahl und Fläche der Frost-Risikobereiche sowie die Schimmelzeiten über 80 % RH, der Isoplethen-Grenze und den VTT-Indizes 1 und 2.

10.7. Vollständiges Evaluator-Beispiel

Das Beispiel `examples/study_with_evaluators.py` zeigt den vollständigen Ablauf mit der neuen API. Es erzeugt Varianten für zwei Klimastandorte, startet sie parallel und verwendet alle vier standardisierten Evaluatoren:

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

Die gemeinsame Auswertung wird als `evaluation.tsv` geschrieben. Auch bei abgebrochenen Läufen bleibt die Performanceauswertung erhalten. Fehlen für eine fachliche Bewertung Ergebnisdateien, wird nur für diesen Evaluator ein Fehler in der Auswertungszeile eingetragen.

10.8. Konfigurierbare Studien über JSON

Für wiederkehrende Variantenstudien muss nicht jedes Mal ein eigenes Python- Script angepasst werden. Das Script `examples/run_configured_study.py` liest eine JSON-Datei ein und führt die darin beschriebene Studie aus. Die Konfiguration ist damit von der Ausführung getrennt und kann beispielsweise von einer späteren grafischen Oberfläche erzeugt werden.

Aktuell wird der Studientyp `material_replacement` unterstützt. Dabei wird zunächst eine Referenzvariante erzeugt und anschließend jedes im Projekt verwendete Material einzeln durch das angegebene Ersatzmaterial ersetzt. Das Ersatzmaterial selbst wird nicht nochmals gegen sich selbst ausgetauscht.

Ein vollständiges Beispiel befindet sich 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
}
```

Die Pfade werden relativ zum Verzeichnis der JSON-Datei interpretiert. Dadurch kann eine Konfigurationsdatei zusammen mit den zugehörigen Studienunterlagen verschoben werden. Absolute Pfade sind ebenfalls möglich.

Die Studie wird aus dem Verzeichnis `scripts/PyDelphin` gestartet:

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

Die wichtigsten Konfigurationsfelder sind:

- `project`: Basisprojekt, aus dem die Varianten erzeugt werden.
- `output_dir`: Zielverzeichnis für Varianten und Auswertungen.
- `solver`: Pfad zur ausführbaren DELPHIN-Solverdatei.
- `replacement_material`: Name des Ersatzmaterials im Projekt.
- `replacement_file`: Materialdatei des Ersatzmaterials.
- `max_workers`: maximale Anzahl gleichzeitig laufender Solverprozesse.
- `solver_run`: Laufzeit-, Fortschritts- und Prozessparameter des Solvers.
- `evaluators`: Liste der standardisierten Auswerter.
- `reuse_existing`: bereits erzeugte Varianten wiederverwenden oder neu erzeugen.

Verfügbare Evaluatoren sind derzeit:

- `performance` für Solverstatus, Echtzeit, Simulationszeit und Fortschrittsquelle,
- `moisture` für die Feuchteakkumulation,
- `frost` für die Frostschadensbewertung,
- `mould` für die Schimmelbewertung eindimensionaler Projekte.

Während der Ausführung wird für jede Variante unmittelbar beim Start eine Meldung ausgegeben:

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

Nach Abschluss wird zusätzlich der Solverstatus und die benötigte Echtzeit angezeigt. Dadurch ist auch bei langen Berechnungen oder Varianten ohne frühe Ergebnisdateien erkennbar, dass der Prozess noch läuft. Das bestehende Script `material_study_3d_bsd.py` bleibt für speziell angepasste Studien weiterhin verwendbar.

11. Vollständiger Testlauf

Die aktuelle Testanwendung kann ohne Solver gestartet werden:

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

Erwartete Ausgabe:

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

Die Unit-Tests prüfen zusätzlich, dass die erzeugten Projekte wieder gelesen werden können:

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

12. Nächste Schritte

Die wichtigsten nächsten Ausbaustufen sind:

- Eine optionale Anbindung von `CmdDiscretise` für 2D- und 3D-Projekte. Für 1D-Projekte ist die variable Diskretisierung mit `rediscretize_1d()` bereits vorhanden.
- Eine robuste Wiederaufnahme bereits abgeschlossener Varianten.
- Zusätzliche Plausibilitätsprüfungen für Ergebnisdateien, insbesondere eine eindeutige Zuordnung von Projekt, Geometrie und Assessment-Ausgaben.

Die jetzige Basis ist dafür geeignet, weil Projektdateien bereits verlustarm gelesen, fachlich geändert, geschrieben und wieder eingelesen werden können.