Melon - a Task Scheduling Package for Todo List Applications

using Markov Chain Monte-Carlo Methods

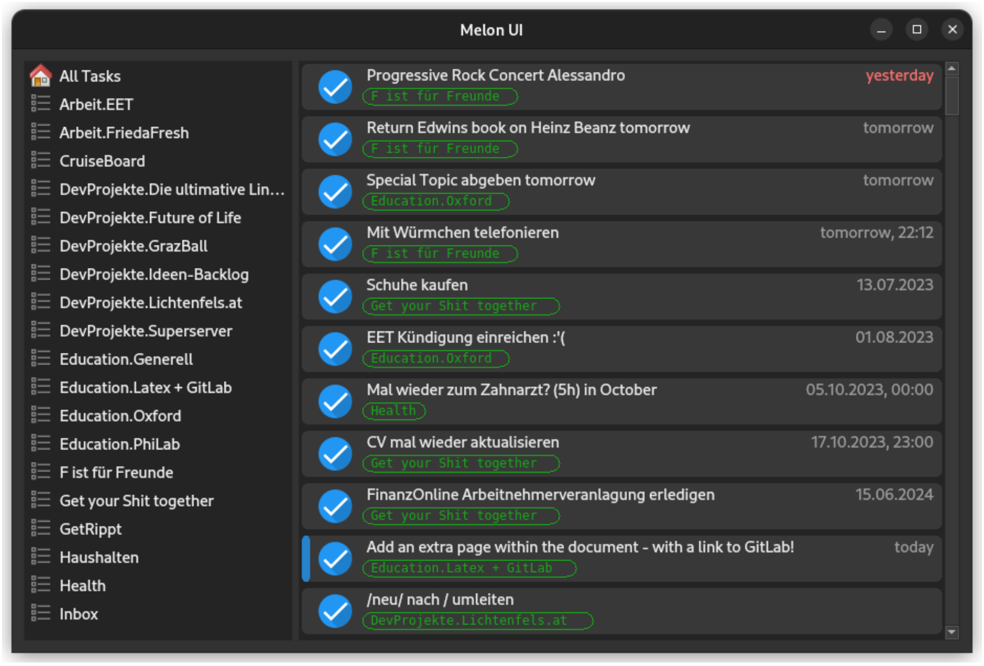
An MMSC Special Topic on Python in Scientific Computing

Candidate Number: 1072462

Abstract

In this project report we will implement a task scheduling method on the basis of a Markov chain Monte-Carlo (MCMC) method with Simulated Annealing. The project is available as a software package **melon-scheduler** on PyPi.

The algorithm is implemented four times, twice in Python, once in Rust and also in C++. Python module bindings to these low-level language implementations are provided using `rust-cpython` and `pybind11`, respectively.



Progressive Rock Concert Alessandro	F ist für Freunde	yesterday
Return Edwins book on Heinz Beanz tomorrow	F ist für Freunde	tomorrow
Special Topic abgeben tomorrow	Education.Oxford	tomorrow
Mit Würmchen telefonieren	F ist für Freunde	tomorrow, 22:12
Schuhe kaufen	Get your Shit together	13.07.2023
EET Kündigung einreichen :'(	Education.Oxford	01.08.2023
Mal wieder zum Zahnarzt? (5h) in October	Health	05.10.2023, 00:00
CV mal wieder aktualisieren	Get your Shit together	17.10.2023, 23:00
FinanzOnline Arbeitnehmerveranlagung erledigen	Get your Shit together	15.06.2024
Add an extra page within the document - with a link to GitLab!	Education.Latex + GitLab	today
/neu/ nach / umleiten	DevProjekte.Lichtenfels.at	

Figure 1: The Graphical User Interface (GUI) accompanying the scheduler. Double clicking tasks allows the user to edit them. Clicking the blue check icon marks them as completed. The grey text on the side represents the relevant due date. Selecting a calendar (todo-list) from the list on the left-hand side will filter the task list to only that category.

1 Problem Introduction

This report is concerned with finding a good scheduling approach for a given set of tasks (todos) with duration, priority, location and due date. The software attached with this report, going by the name of *Melon*, consists of two parts: the `melon` task scheduling package itself and the Graphical User Interface written using the Qt6 framework, contained in `melongui`. Both of these are published as a package **melon-scheduler**, available on PyPi. It may be installed using

- `$ pip install melon-scheduler` for just the scheduler, without the GUI,
- `$ pip install melon-scheduler[gui]` with the GUI or optionally,
- `$ pip install melon-scheduler[gui,plots,numba]` with all extras.

The package is capable of downloading and synchronising tasks from a calendar server supporting the industry-standard CalDAV protocol, displaying and editing them in the GUI and finally scheduling them into a calendar (cf. Figure 2). The scheduling mechanism we implemented is an MCMC method.

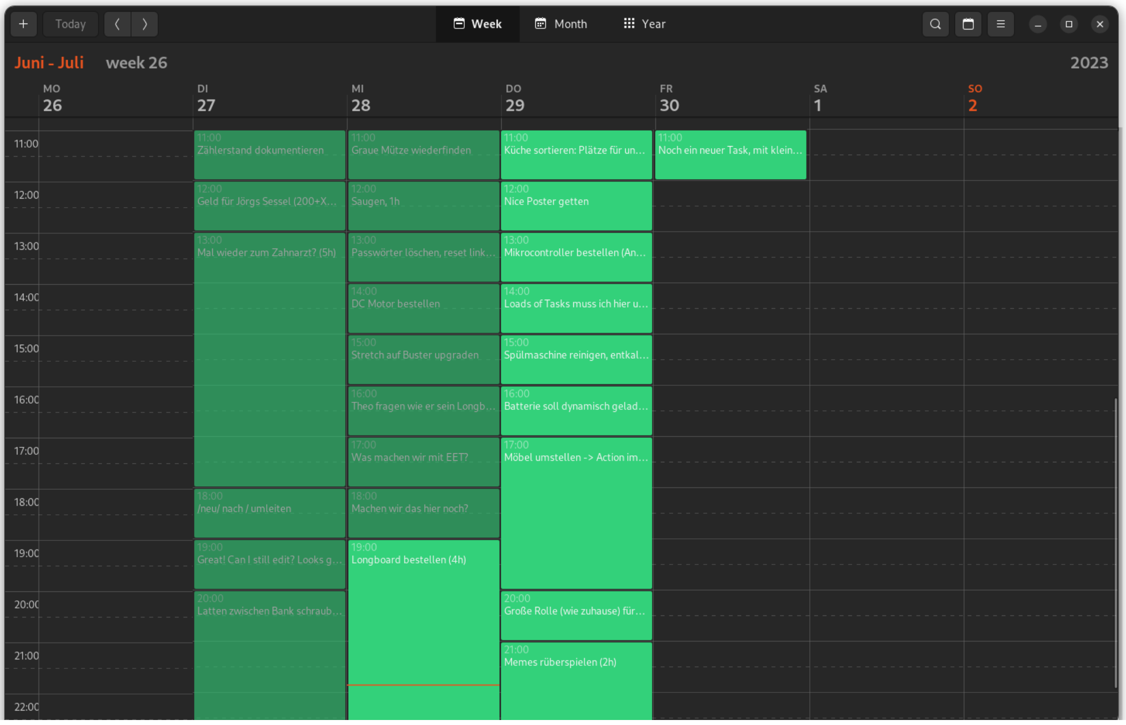

+
Today
Week
Month
Year
Juni - Juli
week 26
2023
MO 26
DI 27
MI 28
DO 29
FR 30
SA 1
SO 2
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00
11:00 Zählerstand dokumentieren
12:00 Geld für Jörgs Sessel (200+X…
13:00 Mal wieder zum Zahnarzt? (5h)
18:00 /neu/ nach / umleiten
19:00 Great! Can I still edit? Looks g…
20:00 Latten zwischen Bank schraub…
11:00 Graue Mütze wiederfinden
12:00 Saugen, 1h
13:00 Passwörter löschen, reset link…
14:00 DC Motor bestellen
15:00 Stretch auf Buster upgraden
16:00 Theo fragen wie er sein Longb…
17:00 Was machen wir mit EET?
18:00 Machen wir das hier noch?
19:00 Longboard bestellen (4h)
11:00 Küche sortieren: Plätze für un…
12:00 Nice Poster getten
13:00 Mikrocontroller bestellen (An…
14:00 Loads of Tasks muss ich hier u…
15:00 Spülmaschine reinigen, entkal…
16:00 Batterie soll dynamisch gelad…
17:00 Möbel umstellen -> Action im…
20:00 Große Rolle (wie zuhause) für…
21:00 Memes rüberspielen (2h)
11:00 Noch ein neuer Task, mit klein…

Figure 2: The scheduled tasks, as displayed in *Gnome Calendar* (the default duration for each task is one hour). Potentially existent events could be taken into account for task scheduling as well, just as well as breaks.

1.1 Idea of the Algorithm

We work with the following assumption on the state: the entire scheduling output, given the set of tasks, is solely determined by the order in which the tasks are scheduled. That is, for a given order of tasks, the full schedule can be created using the supplied input data. With this assumption, finding the absolute optimum is tedious, especially for a large number of tasks $N \gg 1$, as there are $N!$ possible ways to order the tasks.

The idea of the Monte-Carlo method implemented here is to minimise a penalty function (borrowing the term *energy* from physics) over the discrete state space of size $\mathcal{O}(N!)$ using a stochastic approach, as sketched in Section 2. The four key properties we aim to optimise for are:

- spending a minimal amount of time to complete all tasks,
- scheduling high priority tasks first,
- a low number of commutes between locations and
- having all tasks completed on time.

Due to this choice of state representation, the problem broadly mimics a Traveling Salesman Problem.

2 Primer on the Underlying Theory

Many complex problems cannot be solved using analytical methods due to, for instance, their discrete nature. MCMC methods with transition probabilities allow us to explore a huge state space regardless and minimise a function (energy E) therein. There are many use cases in physics, such as for the simulation of the Ising model, where the term *energy* originates from.

The idea of the iterative procedure below, due to Metropolis et al. 1953 and Hastings 1970 is to start from an initial state $\boldsymbol{x}^0$[1], permute it slightly and then accept that new proposal with a probability proportional to the exponential of their energy difference. If the proposal's energy is better (lower) than the current state's energy, the acceptance probability is 1 and therefore automatically $\boldsymbol{x}^{n+1} = \boldsymbol{x}^*$. This allows us to explore state space, but not get stuck in local minima as there is always a non-zero chance for the iteration to escape the local minimum.

[1] In our case, this could be a pre-ordering of tasks obtained by traditional sorting mechanisms, on a simpler metric such as the priority or due date.

The Metropolis-Hastings sweep() sub-routine

```
for $N^2$ many times, repeat
  sample a candidate $\boldsymbol{x}^*$.
  set $\boldsymbol{x}^{n+1} = \boldsymbol{x}^*$ with acceptance probability
    $p_{\text{accept}} = \min\left(1, \mathrm{e}^{-\beta(E^{n+1}-E^n)}\right)$, with $\beta \in \mathbb{R}^+$ a transition factor.
  Otherwise, let $\boldsymbol{x}^{n+1} = \boldsymbol{x}^n$.
```

This algorithm is then employed as a subroutine to an outer iteration, a stochastic global optimisation technique called Simulated Annealing. In our specific case, we will minimise the function $E(\boldsymbol{x})$ given in Equation (1) based on the four properties stated above, using the combination of Metropolis-Hastings and Simulated Annealing.

Simulated Annealing

```
let k = 1
until convergence, repeat
  set the temperature $T = T_0 k^q$ and therefore $\beta = \frac{1}{T}$.
  perform a sweep()
  evaluate $\langle E \rangle$ and $\left\langle \Delta E^2 \right\rangle$ over the sweep.
  set k = k + 1
```

The key idea is to lower the temperature T, starting from an initial value T_0, over the course of the simulation to reduce the transition probability. For each temperature T we evaluate the average of the energy

$$\langle E \rangle \simeq \frac{1}{n} \sum_{\boldsymbol{x}} E(\boldsymbol{x}), \quad \text{and} \quad \left\langle E^2 \right\rangle \simeq \frac{1}{n} \sum_{\boldsymbol{x}} E^2(\boldsymbol{x})$$

with n the number of iterations for this temperature, hence the variance is given by

$$\left\langle \Delta E^2 \right\rangle := \left\langle E^2 \right\rangle - \langle E \rangle^2 .$$

When the variance subceeds a certain threshold, this usually represents an indicator for us stop the iteration. In the present implementation however, we keep the number of sweeps constant at 15, across the four different implementations in order to make the runtimes comparable.

3 Package Design and Architecture

The CalDAV format, short for the Calendaring Extensions to Web Distributed Authoring and Versioning (WebDAV) as introduced in Daboo, Dusseault and Desruisseaux 2007 defines three types of entities: VEVENTs, VTODOs and VJOURNALs. These entities are organised into calendars, for our purposes these could be thought of as different todo lists. *Melon* interacts with CalDAV servers and objects through Python's `caldav` package. A decent amount of the code in `melon` and `melongui` is concerned with the interaction from the package to these objects. Within the scope of this report, we will focus on a smaller version of these VTODO objects, created for a swift interface to the scheduler algorithm implementations.

This small object version, containing data relevant to the scheduling mechanism, looks like this:

```python
import dataclasses
from datetime import datetime

@dataclasses.dataclass
class Task:
    uid: str  # unique identifier of the task
    duration: float  # estimated, in hours
    priority: int  # between 1 and 9
    location: int  # number indicating the location, 0 is "hybrid"
    due: datetime | None  # when the task is due
```

So each task has an associated UID, duration, priority, location and due date. UIDs are useful because they make value collisions very unlikely. This is not to say that these should not be checked, but if two separate calendar clients that each generated a set of UIDs, connected to a server, it is very unlikely to have to resolve potential conflicts.

As mentioned above, the energy we minimise over the state $\boldsymbol{x}$ (to schedule the tasks) is a combination of four properties. To obtain our energy function, we propose numerical expressions for each of them (again, the lower, the better the state) and then perform a weighted sum over all four. Roughly stated, the function we minimise is given by

$$\begin{aligned} E(\boldsymbol{x}) = \text{slot end}_N - \text{slot start}_1 + \sum_{j=1}^{N} \mathbb{1}_{\text{slot end}_j > \text{due}_j} \cdot 100 \\ + \sum_{j=2}^{n} (1 - \mathbb{1}_{\text{location}_{j-1}, \text{location}_j}) \cdot 30 + \sum_{j=1}^{N} j \cdot \text{priority}_j \, , \end{aligned} \tag{1}$$

where the state variable $\boldsymbol{x}$ can be computed to an ordered sequence of

$$(\text{slot start}_j, \text{slot end}_j, \text{priority}_j, \text{location}_j, \text{due}_j)_{j \in \{1,\ldots,N\}}$$

ranging from $j = 1$ the task scheduled into the first slot to $j = N$ the last one. Results of the simulation may be found in Section 4.

3.1 Four Different Implementations

In order to compare runtimes, the same algorithm was implemented four times. Once in pure Python, once using the `numba` library and once in Rust and in C++. Numba uses Just-In-Time compilation to speed up subsequent calls of a subroutine. Rust and C++ are good choices for iterative procedures as they allow for low-level access to the implementation. Bindings are provided using `rust-cpython` and `pybind11`.

- MCMCScheduler: Pure Python implementation of the scheduling algorithm
- NumbaMCMCScheduler: Python with Numba Extension
- RustyMCMCScheduler: Rust implementation, with bindings via `rust-cpython`
- CppMCMCScheduler: C++ implementation, providing bindings with `pybind11`

4 Results

This section will present convergence and runtime results of the scheduling algorithm.

4.1 Energy Convergence

For the pure Python implementation of the MCMC scheduler, we recorded average energy and variance in between sweeps, which are documented in Figure 3 and Figure 4 for different lengths of day. Both figures compare different sweep exponents q from -1 to -3, which governs how the temperature varies over the course of the simulation, and how that affects the convergence speed.

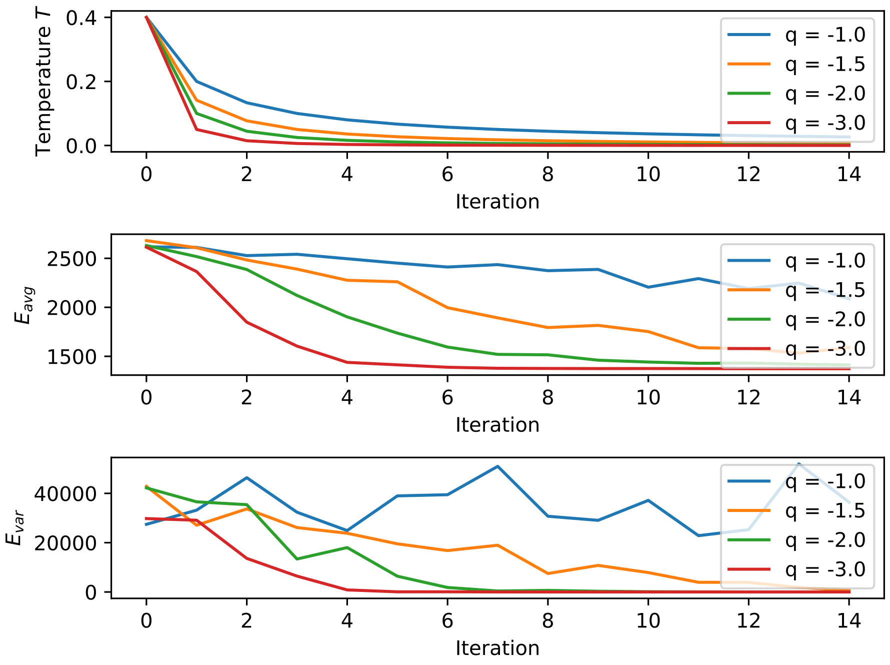


Figure 3: Temperature, average energy $E_{avg} = \langle E \rangle$ and energy variance $E_{var} = \langle \Delta E^2 \rangle$ for a 14-hour work day. Low variance can be used as a stopping criterion (cf. Section 2).

As we can see, the average energy $\langle E \rangle$ generally decreases, and much faster so for higher values of $|q|$. The variance at the bottom stays high for the slowly-progressing run, indicating that convergence has not yet been achieved, but decays quicker for stronger temperature decreases (in red).

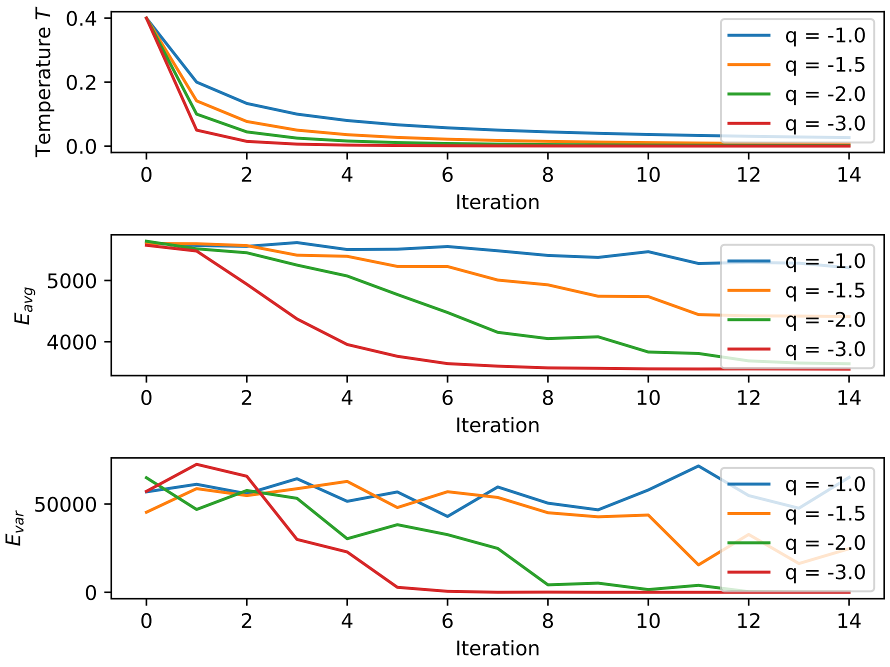


Figure 4: Temperature, average energy $E_{avg} = \langle E \rangle$ and energy variance $E_{var} = \langle \Delta E^2 \rangle$ for an 8-hour work day. The average energy here is higher than in Figure 3 because days are shorter and therefore long-duration task orderings matter a lot more, negatively impacting the optimality.

4.2 Runtime Performance

The following benchmarks were all accumulated on an x86_64 Intel® i7-5600U CPU running at 2.6 GHz verified through 3 individual runs, keeping parameters consistent along them.

Table 2 contains profiling data of the pure Python implementation. Considering the high number of calls but low own time per function call, this suggests that there is little progress to be made using only Python, motivating the use of Numba and the low-level implementations in C++ and Rust to compare.

Table 1: Runtime Comparison of the different implementations run on the same scenarios with $N = 80$ tasks. Each runtime is given as the average over three runs.

Implementation	**Language**	**Runtime / seconds**
MCMCScheduler	Python	31.3887
NumbaMCMCScheduler	Python	1.9335
RustyMCMCScheduler	Rust	0.4034
CppMCMCScheduler	C++	0.4062

And indeed, Table 1 shows us a 77-times speed up of the Rust and C++ implementations as compared to the pure Python implementation. Figure 5 reinforces this data for different input sizes, also revealing the complexity proportionality. As expected, Rust and C++, both compiled languages, perform very similarly in terms of their runtime. Memory usage analysis was not carried out as part of this project, but we suspect Rust and C++ to perform equivalently once again.

What is perhaps more surprising is that Numba is almost five times slower than the Rust and C++ implementations, which is rooted in the limitations of the just-in-time compilation of Python code. Nevertheless, Numba achieves an impressive speed-up, being 15 times faster than the pure Python implementation. One should also note that the Numba compilation time was approximately 3.2 seconds, taking place once for each Python process.

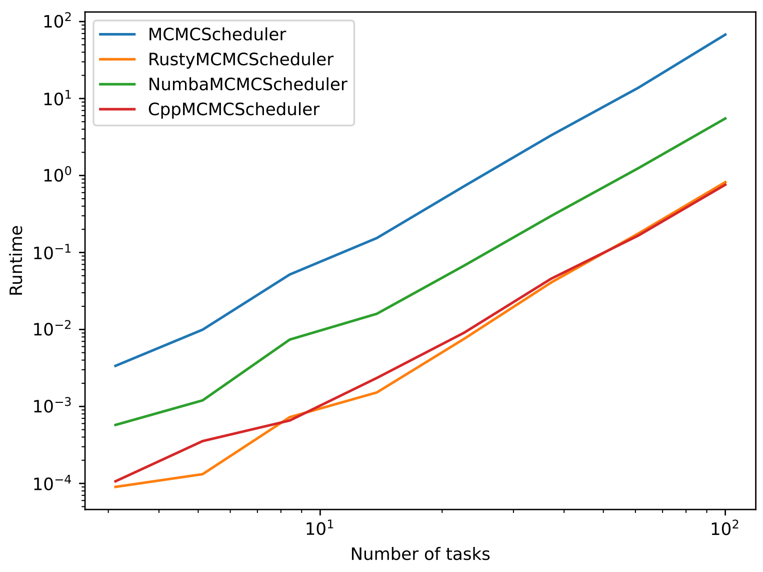


Figure 5: The runtimes of each implementation of the scheduler algorithm (Python, Rust, Python Numba, C++) for an increasing number of tasks.

Fitting a first-order polynomial on the logarithm of the runtime vs the logarithm of the number of tasks, so through the individual lines in Figure 5, we obtain the following slopes:

- MCMCScheduler: 2.874
- RustyMCMCScheduler: 2.728
- NumbaMCMCScheduler: 2.678
- CppMCMCScheduler: 2.570

So the empirical runtime complexity of the algorithm is between $\mathcal{O}(2.5)$ and $\mathcal{O}(2.9)$.

Table 2: Profile obtained by running `$ inv profile-scheduler | grep purepython.py`.

Ncalls	Total	/ call	Cum.	/ call	Filename:line	Function
1	0.000	0.000	14.236	14.236	purepython:142	schedule
10	0.209	0.021	14.235	1.424	purepython:123	mcmcSweep
36010	1.144	0.000	13.784	0.000	purepython:97	computeEnergy
2196671	5.391	0.000	11.353	0.000	purepython:41	spreadTasks
1006332	1.389	0.000	1.842	0.000	purepython:27	generateNextSlot
2196610	0.798	0.000	0.972	0.000	purepython:108	<genexpr>
2196610	0.384	0.000	0.384	0.000	purepython:106	<genexpr>
36000	0.082	0.000	0.217	0.000	purepython:83	permuteState
36011	0.046	0.000	0.124	0.000	purepython:19	startingSlot

5 Installation and Usage

This project uses one of the latest versions of Python, 3.11.4.

5.1 Package Usage

After running `$ pip install melon-scheduler[gui]` and starting a Python console, the following code snippet should start the GUI:

```python
from melongui.main import main
main()  # to start the GUI
```

which launches a User Interface such as the one depicted in Figure 1.

Creation of and interaction with `Todos` in the calendar can be simple:

```python
from melon.melon import Melon

melon = Melon()  # loads the config and initialises
melon.autoInit()  # initiates a network connection to the server
matches = list(melon.findTask("Submit report"))
matches[0].complete()  # marks the todo as complete and syncs

calendar = melon.calendars["My Calendar"]
calendar.sync()  # fetches updates from the server
todo = calendar.createTodo("New Todo")
todo.dueDate = datetime.date.today()
todo.save()  # saves the todo to the server
```

To load todos from a remote calendar, as specified in the configuration file, and schedule them, use the following code snippet:

```python
from melon.melon import Melon
from melon.scheduler.rust import RustyMCMCScheduler

melon = Melon()
melon.autoInit()
melon.scheduleAllAndExport("task-schedule.ics",
 ↪  Scheduler=RustyMCMCScheduler)
```

This will create an iCalendar file `task-schedule.ics` containing an event for each time slot allocated to the completion of a task in the todo list. These can be displayed in a regular calendar application and the output might look like according to Figure 2.

In order to run the scheduler on demonstration data, please run

```python
from melon.scheduler.rust import RustyMCMCScheduler

tasks = generateManyDemoTasks(N=80)
scheduler = RustyMCMCScheduler(tasks)
result = scheduler.schedule()
```

There are many more usage examples available in the `tests`.

If not specified in the initialiser, Melon loads a configuration file located in the user's home configuration directory, so on Linux ~`/.config/melon/config.toml`. The file uses Tom's Obvious, Minimal Language (TOML) format and has the following contents:

```toml
[client]
url = "https://my-caldav-server.org:2023/dav/user/calendars/"
username = "user"
password = "password"
```

5.2 Full Project Usage

The ZIP file contains a number of configuration files at the top level, the two main code folders `melon` and `melongui`, `tests`, `docs` and the `report`. To install dependencies from the `pyproject.toml` file, please run

```
$ poetry install
```

which will automatically create a virtual environment.

There are two main entrypoints to running the code: `main.py` to run the GUI, as well as `tasks.py` which contains miscellanous development and analysis scripts. *Melon* uses `invoke` to these common development tasks which are all callable by running

```
$ inv (name-of-the-task) (arguments) (--keyword-arguments)
```

Table 3: Running `$ inv -l` yields a selection of available `invoke` tasks.

`build-docs`	Builds documentation using Sphinx.
`compare-runtime`	Compares runtime of the different scheduling implementations.
`compile`	Assuming a full setup, compiles the low-level implementations of the scheduler algorithm in C++ and Rust.
`ipython-shell`	Starts an IPython shell with Melon initialised.
`plot-convergence`	Plots scheduler convergence to a file.
`plot-runtime-complexity`	Simulates with a varying number of tasks and plots runtime complexity.
`profile-scheduler`	Profile the pure Python MCMC Scheduler.
`schedule-and-export`	Run the MCMC scheduler and export the resulting events as an ICS file.
`start-mock-server`	Starts a Xandikos (CalDAV) server on port 8000.

Low-Level Language Setup In order to compile the C++ implementation of the scheduler algorithm, starting from the root folder of the project (containing `CMakeLists.txt` and `conanfile.txt`), please run

```
$ conan install . --output-folder=build --build=missing
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make -j4
```

To compile the Rust implementation, simply

```
$ cargo build -release
```

again making sure that the current working directory is the root folder of the project (containing `Cargo.toml`).

This should have created two `.so` files in the respective folders. The import paths are already adjusted to be able to import these in `cpp.py` and `rust.py`, but they will also be copied to the correct `melon/scheduler` folder using `$ inv compile`.

We recommend usage with `xandikos`, a version-controlled DAV server, capable of syncing calendars (events, todos and journals) and contacts. Following the standard protocol, *Melon* is also compatible with commercial services such Google Calendar or Microsoft Office, as long as these offer an API endpoint with suitable authentication.

The code should mostly be platform-independent, for example due to the usage of `pathlib.Path`. Compiling the low-level language implementations might be more cumbersome however and is untested on platforms other than Linux.

6 Code Quality Measures

Writing good code is an art, but there are a few concepts, principles and tools to approach the problem using a standardised approach. Some of these are:

Formatting the *Melon* code is done by the `black` software package. Configuration thereof, as well as that for most other tools, can be found in the `pyproject.toml` file. To format all Python code, run `$ black .` which will recursively explore the entire folder. The C++ code is formatted using `clang-format` while the Rust code is formatted using `rustfmt`.

Docstrings help us document the code, *within* the code. Every class, method and function in the project, including `melon`, `melongui`, the `tasks` and the `tests`, has a docstring. These follow a specific format in order for Sphinx to be able to pick up arguments, exceptions raised and return values as well as their associated types. This coverage can be verified using `$ interrogate -vv`, cf. Table 4.

Documentation is important to make the purpose and usage of the code package clear. This project uses `sphinx` to generate documentation in PDF format which one may find at the end of this report. To generate it, run `$ inv build-docs`.

Dependency Management in this project is done using `poetry`, which not only manages install packages and manages virtual environments, but also keeps track of dependency groups. To install all direct dependencies, run `$ poetry install`.

Type Checking is done with `pyright` instead of `mypy` as it is much faster and analyses the entire project at once. This tool detects when, for instance, attempting to call a non-existent method on an object, or passing the wrong type to a function call, etc. The *Melon* code therefore contains numerous type hints. To verify all type hints, run `$ pyright .` in the root folder of the project.

Using Appropriate Language Features Tools such as `autoflake` and `pyupgrade` automatically correct unused imports or deprecated code usage. `ruff` is a highly performant linter written in Rust, that not only warns the programmer on common mistakes, but can also perform small fixes to the structure of the code such as import reordering. `nitpick` is a tool to synchronise linter configuration across projects.

Table 4: Output table of `$ interrogate -vv`: **Passed** (minimum: 80.0%, actual: 100.0%). Each file has a number of classes, functions and methods as displayed in **Total** and `interrogate` yields the proportion of those having a docstring.

Name	**Total**	**Miss**	**Cover**	**Cover**
main.py	2	0	2	100 %
tasks.py	8	0	8	100 %
docs/conf.py	1	0	1	100 %
melon/__init__.py	1	0	1	100 %
melon/calendar.py	11	0	11	100 %
melon/config.py	1	0	1	100 %
melon/melon.py	19	0	19	100 %
melon/todo.py	20	0	20	100 %
melon/visualise.py	3	0	3	100 %
melon/scheduler/__init__.py	1	0	1	100 %
melon/scheduler/base.py	10	0	10	100 %
melon/scheduler/cpp.py	3	0	3	100 %
melon/scheduler/numba.py	8	0	8	100 %
melon/scheduler/purepython.py	12	0	12	100 %
melon/scheduler/rust.py	3	0	3	100 %
melongui/__init__.py	1	0	1	100 %
melongui/calendarlist.py	6	0	6	100 %
melongui/mainwindow.py	14	0	14	100 %
melongui/taskitemdelegate.py	12	0	12	100 %
melongui/tasklist.py	14	0	14	100 %
melongui/taskwidgets.py	8	0	8	100 %
tests/__init__.py	1	0	1	100 %
tests/test_melon.py	7	0	7	100 %
tests/test_scheduler.py	9	0	9	100 %
TOTAL	175	0	175	100 %

6.1 Tests and Coverage

Software testing is a vital part of any programming endeavour to ensure high levels of overally code quality. This submission only contains tests for the `melon` package of the code, not for the GUI, which will be subject to future efforts. There are 34 tests provided along with the code.

In order to simulate the interaction with a Calendaring Extensions to WebDAV (CalDAV) server, we provide a tool to start a mock server using Docker, a containerisation engine that abstracts code execution to individual entities called containers. To start a the `xandikos` mock server, please run

```
$ inv start-mock-server
```

Once the server is running, the tests may be run simply by:

```
$ pytest
```

As we can see using `$ pytest --durations=0`,

```
 ======================== slowest durations ========================
3.53s call TestScheduler::test_length[NumbaMCMCScheduler]
0.19s call TestMelon::test_init_store_and_load
```

the slowest test is the first routine involving the Numba scheduler which takes some time to pre-compile the functions. So even when the runtime of the Numba scheduler itself is low (cf. Section 4.2), the test will always take some extra time.

The four different algorithm implementations are tested against each other and on different parameters in order to ensure they work correctly.

6.1.1 Code Coverage

Maintaining Code Quality `pre-commit` is a tool that can install a git hook to the code repository, which automatically runs a set of checks before every commit, hence the name. For this project, various checks listed below are employed, being run before each and every commit to keep code quality high throughout the entire development process.

A similar, more team-friendly option is to use GitHub Actions Continuous Integration (CI) / Continuous Delivery (CD), or simply CI/CD.

All the tools described above will be installed automatically using

```
$ poetry install --with=dev.
```

Table 5: Test coverage of the `melon` package: platform linux, python 3.11.4-final-0. Each file is analysed by the number of statements (lines) in the file executed as part of the tests. This table may be reproduced using `$ pytest --cov=melon`.

Name	Statements	Miss	Cover
melon/__init__.py	0	0	100 %
melon/calendar.py	57	5	91 %
melon/config.py	12	0	100 %
melon/melon.py	121	8	93 %
melon/scheduler/__init__.py	0	0	100 %
melon/scheduler/base.py	40	3	92 %
melon/scheduler/cpp.py	18	5	72 %
melon/scheduler/purepython.py	83	0	100 %
melon/scheduler/rust.py	18	5	72 %
melon/todo.py	101	17	83 %
melon/visualise.py	42	2	95 %
TOTAL	**492**	**45**	**91 %**

Table 6: Pre-Commit hooks run on all files of the repository using `$ pre-commit run --all-files`. Each hook can either pass, fail or modify existing code.

```
prettier...........................................................Passed
fix end of files...................................................Passed
trim trailing whitespace...........................................Passed
black..............................................................Passed
ruff...............................................................Passed
check blanket noqa.................................................Passed
check for eval()...................................................Passed
interrogate........................................................Passed
autoflake..........................................................Passed
pyupgrade..........................................................Passed
pyright............................................................Passed
pytest-check.......................................................Passed
clang-format.......................................................Passed
latex-format-all...................................................Passed
```

Publishing to PyPi As highlighted above, the project can be installed from this PyPi repository. In order to build and publish the project, one can simply run

```
$ poetry publish --build
```

Although it would be possible to compile the C++ and Rust implementations on a CI service using a "platform matrix", the published package only contains compilation targets for the x86_64 platform and Python 3.11.

7 Conclusion

In this Special Topic, we implemented a task scheduling application, tested it thoroughly (code coverage over 90 %) and used it for the development of a Todo List Application. The optimisation is done using a Markov chain Monte-Carlo method with Simulated Annealing. This algorithm was implemented four times, in Python, Rust and C++. The low-level language implementations in Rust and C++ outperformed the high-level language implementations in Python by nearly two orders of magnitude.

7.1 Acknowledgements

The task check icon is the logo of the *Tasks.org* Free and Open Source Android App, the artwork may be found here.

References

Daboo, Cyrus, Lisa M. Dusseault and Bernard Desruisseaux (Mar. 2007). *Calendaring Extensions to WebDAV (CalDAV)*. RFC 4791. DOI: 10.17487/RFC4791. URL: https://www.rfc-editor.org/info/rfc4791.

Hastings, W. K. (1970). 'Monte Carlo Sampling Methods Using Markov Chains and Their Applications'. In: *Biometrika* 57, pp. 97–109. DOI: 10.1093/biomet/57.1.97.

Metropolis, N., Arianna W. Rosenbluth, Marshall N. Rosenbluth, A. H. Teller and Edward Teller (1953). 'Equation of state calculations by fast computing machines'. In: *Journal of Chemical Physics* 21, pp. 1087–1092. DOI: 10.1063/1.1699114.

Acronyms

CalDAV	Calendaring Extensions to WebDAV	5, 17
CD	Continuous Delivery	17
CI	Continuous Integration	17
GUI	Graphical User Interface	1
MCMC	Markov chain Monte-Carlo	1
TOML	Tom's Obvious, Minimal Language	12
WebDAV	Web Distributed Authoring and Versioning	5

A Accessing VTODO properties

A profiler may be used to identify parts of the code that are slow. In the case of the GUI, the Item Delegate's paint() method must be performant in order to provide a smooth user experience. This can be achieved when looking at different means of accessing the UID of a task, which as per Table 8 is a highly frequent action. Here is a comparison of different approaches:

```
In [1]: %timeit str(t.icalendar_component["uid"])
  122 µs ± 1.06 µs per loop (7 runs, 10,000 loops each)
In [2]: %timeit t.vtodo.contents["uid"][0].value
  355 ns ± 7.14 ns per loop (7 runs, 1,000,000 loops each)
In [3]: %timeit
↪  t.vobject_instance.contents["vtodo"][0].contents["uid"][0].value
  296 ns ± 7.06 ns per loop (7 runs, 1,000,000 loops each)
```

```
In [4]: %timeit t._vobject_instance.contents["vtodo"][0].contents["uid"][0].value
  208 ns ± 23.7 ns per loop (7 runs, 10,000,000 loops each)
```

As we can see, the last option is the fastest which has therefore been implemented in `todo.py`.

Table 8: Profile obtained by running `$ ./main.py --profile | grep todo.py`.

ncalls	tottime	percall	cumtime	percall	filename:lineno	function
16958	0.008	0.000	0.939	0.000	todo.py:36	vtodo
32475	0.047	0.000	0.705	0.000	todo.py:96	uid
856	0.003	0.000	0.579	0.001	todo.py:26	upgrade
117	0.000	0.000	0.489	0.004	todo.py:111	priority
417	0.001	0.000	0.461	0.001	todo.py:121	isIncomplete
5512	0.003	0.000	0.278	0.000	todo.py:45	summary
856	0.002	0.000	0.112	0.000	todo.py:21	__init__
1363	0.006	0.000	0.024	0.000	todo.py:164	__lt__
7844	0.004	0.000	0.009	0.000	todo.py:61	dueDate
2605	0.001	0.000	0.003	0.000	todo.py:85	dueTime

B Sphinx Documentation

As mentioned above, this report also contains the documentation generated from the docstrings. It starts on the next page.

CHAPTER

ONE

MELON

Index file of the *melon* library, a task scheduling software package.

Modules

melon.calendar	This module contains the Calendar class.
melon.config	This submodule only does one thing: loading configuration from the right place.
melon.melon	This file is the main entry point of the melon package, containing the Melon class, the main point of contact for users of this package.
melon.scheduler	Melon.Scheduler library housing different implementations of an MCMC task scheduler.
melon.todo	This module contains the Todo class.
melon.visualise	A collection of (quality measure) visualisation helpers.

1.1 melon.calendar

This module contains the Calendar class.

Classes

Calendar(calendar)	Class representing a calendar (or todo list, if you want to call it that, this name is given by CalDAV).
Syncable(calendar, objects, sync_token)	The synchronisable collection of CalDAV objects, handling efficient syncs between server and client.

1.1.1 melon.calendar.Calendar

class melon.calendar.**Calendar**(*calendar: Calendar*)

Class representing a calendar (or todo list, if you want to call it that, this name is given by CalDAV). A calendar is a collection of objects that can be synced to a CalDAV server. In this implementation, the objects are stored within the *syncable* subclass.

__init__(*calendar: Calendar*) → None

A copy constructor

Args:
calendar (caldav.Calendar): Argument

Methods

__init__(calendar)	A copy constructor
add_event([ical, no_overwrite, no_create])	Add a new event to the calendar, with the given ical.
add_journal([ical, no_overwrite, no_create])	Add a new journal entry to the calendar, with the given ical.
add_todo([ical, no_overwrite, no_create])	Add a new task to the calendar, with the given ical.
build_date_search_query(start[, end, ...])	WARNING: DEPRECATED
build_search_xml_query([comp_class, todo, ...])	This method will produce a caldav search query as an etree object.
calendar_multiget(event_urls)	get multiple events' data @author mtorange@gmail.com @type events list of Event
children([type])	List children, using a propfind (resourcetype) on the parent object, at depth = 1.
createTodo([summary])	Args:
date_search(start[, end, compfilter, ...])	Deprecated.
delete()	Delete the object.
event(uid)	
event_by_uid(uid)	
event_by_url(href[, data])	Returns the event with the given URL
events()	List all events from the calendar.
freebusy_request(start, end)	Search the calendar, but return only the free/busy information.
get_display_name()	Get calendar display name
get_properties([props, depth, ...])	Get properties (PROPFIND) for this object.
get_property(prop[, use_cached])	
get_supported_components()	returns a list of component types supported by the calendar, in string format (typically ['VJOURNAL', 'VTODO', 'VEVENT'])
journal_by_uid(uid)	
journals()	List all journals from the calendar.
loadFromFile(client, principal, name, ...)	Args:
object_by_uid(uid[, comp_filter, comp_class])	Get one event from the calendar.
objects([sync_token, load_objects])	objects_by_sync_token aka objects

continues on next page

Table 1 – continued from previous page

`objects_by_sync_token`([sync_token, load_objects])	objects_by_sync_token aka objects
`save`()	The save method for a calendar is only used to create it, for now.
`save_event`([ical, no_overwrite, no_create])	Add a new event to the calendar, with the given ical.
`save_journal`([ical, no_overwrite, no_create])	Add a new journal entry to the calendar, with the given ical.
`save_todo`([ical, no_overwrite, no_create])	Add a new task to the calendar, with the given ical.
`save_with_invites`(ical, attendees, ...)	sends a schedule request to the server.
`search`([xml, comp_class, todo, ...])	Creates an XML query, does a REPORT request towards the server and returns objects found, eventually sorting them before delivery.
`set_properties`([props])	Set properties (PROPPATCH) for this object.
`storageObject`()	Returns:
`storeToFile`()	Save the calendar objects to a local file on disk, in iCal format.
`sync`()	Synchronise me
`todo_by_uid`(uid)	
`todos`([sort_keys, include_completed, sort_key])	fetches a list of todo events (refactored to a wrapper around search)

Attributes

`canonical_url`
`client`
`id`
`name`
`parent`
`url`

`createTodo`(*summary: str = 'An exciting new task!'*)

> **Args:**
> summary (str): Argument

`static` **`loadFromFile`**(*client: DAVClient*, *principal: Principal*, *name: str*, *sync_token: str*, *url: str*)

> **Args:**
> client (caldav.DAVClient): Argument principal (caldav.Principal): Argument name (str): Argument sync_token (str): Argument url (str): Argument

`storageObject`() → dict

> **Returns:**
> (dict):

storeToFile()

Save the calendar objects to a local file on disk, in iCal format.

sync()

Synchronise me

1.1.2 melon.calendar.Syncable

class melon.calendar.**Syncable**(*calendar*, *objects*, *sync_token*)

The synchronisable collection of CalDAV objects, handling efficient syncs between server and client.

__init__(*calendar*, *objects*, *sync_token*)

Methods

__init__(calendar, objects, sync_token)	
objects_by_url()	returns a dict of the contents of the Synchronizable-CalendarObjectCollection, URLs -> objects.
sync()	This method will contact the caldav server, request all changes from it, and sync up the collection
upgrade(synchronisable, calendarName)	Upgrades the third-party caldav.SynchronizableCalendarObjectCollection to a Syncable
upgradeObjects(calendarName)	Converts all objects in self.objects to Todos.

Attributes

calendar
objects
sync_token

static **upgrade**(*synchronisable: SynchronizableCalendarObjectCollection*, *calendarName: str*) → *Syncable*

Upgrades the third-party caldav.SynchronizableCalendarObjectCollection to a Syncable

Args:
synchronisable (caldav.SynchronizableCalendarObjectCollection): the original instance

Returns:
(Syncable): the syncable

upgradeObjects(*calendarName: str*)

Converts all objects in self.objects to Todos.

1.2 melon.config

This submodule only does one thing: loading configuration from the right place.

Functions

`load_config`()	Loads, or re-loads, the configuration file.

1.2.1 melon.config.load_config

`melon.config.`**`load_config`**`()`

Loads, or re-loads, the configuration file.

1.3 melon.melon

This file is the main entry point of the melon package, containing the Melon class, the main point of contact for users of this package. It can be initialised like this:

melon = Melon() melon.autoInit()

Classes

`Melon`([url, username, password, maxCalendars])	The Melon class, wrapping a caldav client and principal, loading specifics from the config.

1.3.1 melon.melon.Melon

`class` `melon.melon.`**`Melon`**(*url='http://localhost:8000/dav/user/calendars/'*, *username=None*, *password=None*, *maxCalendars: int | None = None*)

The Melon class, wrapping a caldav client and principal, loading specifics from the config. Through me, users have access to calendars and todos. I also handle load, sync and store functionality.

`__init__`(*url='http://localhost:8000/dav/user/calendars/'*, *username=None*, *password=None*, *maxCalendars: int | None = None*) → None

Initialises the Melon client

Args:

url (str, optional): URL to the CalDAV server. Defaults to CONFIG["client"]["url"]. username (str, optional): Username. Defaults to CONFIG["client"]["username"]. password (str, optional): Password. Defaults to CONFIG["client"]["password"]. maxCalendars (int, optional): the highest number of calendars to load. Useful for testing.

Methods

`__init__`([url, username, password, maxCalendars])	Initialises the Melon client
`addOrUpdateTask`(todo)	Args:
`allIncompleteTasks`()	Returns all incomplete todos
`allTasks`()	Returns an iterable of all tasks in all calendars as a single list
`autoInit`()	Args:
`connect`()	Args:
`exportScheduleAsCalendar`(scheduling)	A read-only ICS calendar containing scheduled tasks.
`fetch`()	Args:
`findTask`(string)	Finds a task given a search query
`getTask`(uid)	Returns task with given UID
`load`()	Args:
`scheduleAllAndExport`(file[, Scheduler])	Runs the scheduler on all tasks and exports as an ICS file.
`store`()	Args:
`syncAll`()	Args:
`syncCalendar`(calendar)	Args:
`tasksToSchedule`()	Returns all incomplete tasks as scheduler.Task objects

Attributes

`HIDDEN_CALENDARS`

`addOrUpdateTask`(*todo:* Todo)

Args:
todo (Todo): Argument

`allIncompleteTasks`() → Iterable[*Todo*]

Returns all incomplete todos

Yields:
Iterator[Iterable[Todo]]: incomplete todos

`allTasks`() → Iterable[*Todo*]

Returns an iterable of all tasks in all calendars as a single list

Yields:
Iterator[Iterable[Todo]]: iterator of all tasks

`autoInit`()

Args:

`connect`()

Args:

`exportScheduleAsCalendar`(*scheduling: Mapping[str,* TimeSlot*]*) → Calendar

A read-only ICS calendar containing scheduled tasks. Can be stored to disk using schedule.to_ical().

Args:
scheduling (Mapping[str, TimeSlot]): Mapping of task UID to TimeSlot

Returns:
icalendar.Calendar: the calendar containing events (time slots) proposed for the completion of tasks

`fetch`()

Args:

`findTask`(*string: str*) → Iterable[*Todo*]

Finds a task given a search query

Args:
string (str): the search query.

Yields:
Iterator[Iterable[Todo]]: the generated search results.

`getTask`(*uid: str*) → *Todo*

Returns task with given UID

Args:
uid (str): the Unique Identifier

Raises:
ValueError: when the task could not be found

Returns:
Todo: the Todo with given uid

`load`()

Args:

`scheduleAllAndExport`(*file: str, Scheduler: type[melon.scheduler.base.AbstractScheduler] = <class 'melon.scheduler.purepython.MCMCScheduler'>*)

Runs the scheduler on all tasks and exports as an ICS file.

Args:
file (str): filesystem path that the ics file should be exported to

`store`()

Args:

`syncAll`()

Args:

`syncCalendar`(*calendar:* Calendar)

Args:
calendar: Argument

`tasksToSchedule`() → list[*melon.scheduler.base.Task*]

Returns all incomplete tasks as scheduler.Task objects

Returns:
list[Task]: _description_

1.4 melon.scheduler

Melon.Scheduler library housing different implementations of an MCMC task scheduler.

Modules

melon.scheduler.base	The scheduler algorithm
melon.scheduler.cpp	The scheduler algorithm
melon.scheduler.libcppscheduler	Schedule tasks
melon.scheduler.libscheduler	This module is implemented in Rust.
melon.scheduler.numba	Numba implementation of the scheduler algorithm.
melon.scheduler.purepython	The scheduler algorithm
melon.scheduler.rust	The scheduler algorithm

1.4.1 melon.scheduler.base

The scheduler algorithm

Functions

generateDemoTasks()	Generates a fixed set of demo tasks.
generateManyDemoTasks(N[, proportionOfDueDates])	Generates a larger set of randomly generated demo tasks.

melon.scheduler.base.generateDemoTasks

melon.scheduler.base.**generateDemoTasks**() → list[*melon.scheduler.base.Task*]

Generates a fixed set of demo tasks.

Returns:
list[Task]: the generated list of tasks

melon.scheduler.base.generateManyDemoTasks

melon.scheduler.base.**generateManyDemoTasks**(*N: int*, *proportionOfDueDates: float = 0.5*) → list[*melon.scheduler.base.Task*]

Generates a larger set of randomly generated demo tasks.

Args:
N (int): Number of tasks to be generated proportionOfDueDates (float, optional): what percentage (from 0 to 1) of tasks should have a due date.

Defaults to 0.5.

Returns:
list[Task]: the list of tasks

Classes

AbstractScheduler(tasks)	Abstract Base Class (ABC) for schedulers.
Task(uid, duration, priority, location, due)	Slim struct representing a task
TimeSlot(timestamp, duration)	Slim struct representing a time slot, so an event consisting of a start and end date.

melon.scheduler.base.AbstractScheduler

`class melon.scheduler.base.`**`AbstractScheduler`**(*tasks: list[*melon.scheduler.base.Task*]*)

Abstract Base Class (ABC) for schedulers.

`__init__`(*tasks: list[*melon.scheduler.base.Task*]*) → None

Initialises the scheduler, working on a set of pre-defined tasks.

Args:
tasks (list[Task]): the tasks to be scheduled

Methods

__init__(tasks)	Initialises the scheduler, working on a set of pre-defined tasks.
schedule()	Schedules the tasks using an MCMC procedure.
uidTaskMap()	Generates a dictionary for task lookup by UID.

`schedule`() → Mapping[str, *TimeSlot*]

Schedules the tasks using an MCMC procedure.

Returns:
Mapping[str, TimeSlot]: the resulting map of Tasks to TimeSlots

`uidTaskMap`() → Mapping[str, *Task*]

Generates a dictionary for task lookup by UID.

Returns:
Mapping[str, Task]: the dictionary keyed by UID of each task.

melon.scheduler.base.Task

`class melon.scheduler.base.`**`Task`**(*uid: str*, *duration: float*, *priority: int*, *location: int*, *due: datetime | None*)

Slim struct representing a task

`__init__`(*uid: str*, *duration: float*, *priority: int*, *location: int*, *due: datetime | None*) → None

Methods

__init__(uid, duration, priority, location, due)	
asTuple(start)	Returns a low-level representation of this instance.

Attributes

uid
duration
priority
location
due

asTuple(*start: datetime*) → tuple[str, float, int, int, float]

Returns a low-level representation of this instance.

Args:
start (datetime): Start time reference for the due date

Returns:

tuple[str, float, int, int, float]: low-level representation (uid, duration, priority, location, due).
due is 0 if there is no due date.

melon.scheduler.base.TimeSlot

class melon.scheduler.base.**TimeSlot**(*timestamp: datetime*, *duration: float*)

Slim struct representing a time slot, so an event consisting of a start and end date.

__init__(*timestamp: datetime*, *duration: float*) → None

Methods

__init__(timestamp, duration)

Attributes

end	Returns:
timedelta	Returns:
timestamp	
duration	

property end: datetime

Returns:
datetime: the end timestamp of this time slot

property timedelta: timedelta

Returns:
timedelta: the duration as a datetime.timedelta instance

1.4.2 melon.scheduler.cpp

The scheduler algorithm

Classes

CppMCMCScheduler(tasks)	Markov Chain Monte-Carlo Task Scheduler, implemented in Rust.

melon.scheduler.cpp.CppMCMCScheduler

class melon.scheduler.cpp.**CppMCMCScheduler**(*tasks: list[melon.scheduler.base.Task]*)

Markov Chain Monte-Carlo Task Scheduler, implemented in Rust.

__init__(*tasks: list[melon.scheduler.base.Task]*) → None

Initialises the scheduler, working on a set of pre-defined tasks.

Args:
tasks (list[Task]): the tasks to be scheduled

Methods

__init__(tasks)	Initialises the scheduler, working on a set of pre-defined tasks.
schedule()	Runs the Rust implementation of the scheduler.
uidTaskMap()	Generates a dictionary for task lookup by UID.

schedule() → Mapping[str, *TimeSlot*]

Runs the Rust implementation of the scheduler.

Returns:

Mapping[str, TimeSlot]: the resulting schedule

1.4.3 melon.scheduler.libcppscheduler

Schedule tasks

Functions

schedule(arg0)	Schedule tasks

melon.scheduler.libcppscheduler.schedule

melon.scheduler.libcppscheduler.**schedule**(*arg0: list*) → list

Schedule tasks

1.4.4 melon.scheduler.libscheduler

This module is implemented in Rust.

1.4.5 melon.scheduler.numba

Numba implementation of the scheduler algorithm.

Functions

computeEnergy(tasks, state)	For the given state, compute an MCMC energy (the lower, the better)
mcmcSweep(tasks, initialState, temperature)	Performs a full MCMC sweep
permuteState(state)	Proposes a new state to use instead of the old state.
schedule(tasks)	Schedules the given tasks in low-level representation into calendar.
spreadTasks(tasks)	Spreads the given list of tasks across the available slots in the calendar, in order.

melon.scheduler.numba.computeEnergy

`melon.scheduler.numba.computeEnergy`(*tasks: Sequence[tuple[str, float, int, int, float]]*, *state: list[int]*) → float

For the given state, compute an MCMC energy (the lower, the better)

Args:
tasks (Sequence[tuple[str, float, int, int, float]]): list of tasks (uid, duration, priority, location, due) state (State): state of the MCMC algorithm

Returns:
float: the energy / penalty for this state

melon.scheduler.numba.mcmcSweep

`melon.scheduler.numba.mcmcSweep`(*tasks: Sequence[tuple[str, float, int, int, float]]*, *initialState: list[int]*, *temperature: float*) → list[int]

Performs a full MCMC sweep

Args:
tasks (Sequence[tuple[str, float, int, int, float]]): list of tasks initialState (State): initial ordering temperature (float): temperature for Simulated Annealing

Returns:
State: new state

melon.scheduler.numba.permuteState

`melon.scheduler.numba.permuteState`(*state: list[int]*) → list[int]

Proposes a new state to use instead of the old state.

Returns:
State: the new state, a list of indices within tasks representing traversal order

melon.scheduler.numba.schedule

`melon.scheduler.numba.schedule`(*tasks: Sequence[tuple[str, float, int, int, float]]*) → Sequence[tuple[str, float, float]]

Schedules the given tasks in low-level representation into calendar.

Args:
tasks (Sequence[tuple[str, float, int, int, float]]): vector of tasks (uid, duration, priority, location, due)

Returns:
Sequence[tuple[str, float, float]]: vector of allocated timeslots (uid, timestamp, duration)

melon.scheduler.numba.spreadTasks

`melon.scheduler.numba.`**`spreadTasks`**`(`*tasks: Sequence[tuple[str, float, int, int, float]]*`)` → Sequence[tuple[str, float, float]]

Spreads the given list of tasks across the available slots in the calendar, in order.

Args:
tasks (Sequence[Task]): list of tasks to schedule

Yields:
Iterator[tuple[str, TimeSlot]]: pairs of (UID, TimeSlot), returned in chronological order

Classes

NumbaMCMCScheduler(tasks)	Markov Chain Monte-Carlo Task Scheduler, implemented in Python with numba speed-up.

melon.scheduler.numba.NumbaMCMCScheduler

`class` `melon.scheduler.numba.`**`NumbaMCMCScheduler`**`(`*tasks: list[*melon.scheduler.base.Task*]*`)`

Markov Chain Monte-Carlo Task Scheduler, implemented in Python with numba speed-up.

`__init__``(`*tasks: list[*melon.scheduler.base.Task*]*`)` → None

Initialises the scheduler, working on a set of pre-defined tasks.

Args:
tasks (list[Task]): the tasks to be scheduled

Methods

__init__(tasks)	Initialises the scheduler, working on a set of pre-defined tasks.
schedule()	Runs the Rust implementation of the scheduler.
uidTaskMap()	Generates a dictionary for task lookup by UID.

`schedule``()` → Mapping[str, *TimeSlot*]

Runs the Rust implementation of the scheduler.

Returns:
Mapping[str, TimeSlot]: the resulting schedule

1.4.6 melon.scheduler.purepython

The scheduler algorithm

Classes

AvailabilityManager()	This class manages the user's availability in a calendar.
MCMCScheduler(tasks)	MCMC class to schedule tasks to events in a calendar.

melon.scheduler.purepython.AvailabilityManager

`class melon.scheduler.purepython.`**`AvailabilityManager`**

This class manages the user's availability in a calendar.

`__init__``()` → None

Initialises the availability manager according to defaults.

Methods

__init__()	Initialises the availability manager according to defaults.
generateNextSlot(previous)	Following a daily schedule, returns the next possible working slot
spreadTasks(tasks)	Spreads the given list of tasks across the available slots in the calendar, in order.
startingSlot()	Starting slot, starting at 10am today

`generateNextSlot``(`*previous:* TimeSlot`)` → *TimeSlot*

Following a daily schedule, returns the next possible working slot

Args:
previous (TimeSlot): the previous working slot

Returns:
TimeSlot: the next working slot

`spreadTasks``(`*tasks: Iterable[*Task*]*`)` → Iterable[tuple[str, *melon.scheduler.base.TimeSlot*]]

Spreads the given list of tasks across the available slots in the calendar, in order.

Args:
tasks (Iterable[Task]): list of tasks to schedule

Yields:
Iterator[tuple[str, TimeSlot]]: pairs of (UID, TimeSlot), returned in chronological order

`startingSlot``()` → *TimeSlot*

Starting slot, starting at 10am today

Returns:
TimeSlot: the first working slot

melon.scheduler.purepython.MCMCScheduler

class melon.scheduler.purepython.**MCMCScheduler**(*tasks: list[melon.scheduler.base.Task]*)

MCMC class to schedule tasks to events in a calendar.

__init__(*tasks: list[melon.scheduler.base.Task]*) → None

Initialises the MCMC scheduler, working on a set of pre-defined tasks.

Args:
tasks (list[Task]): the tasks to be scheduled

Methods

__init__(tasks)	Initialises the MCMC scheduler, working on a set of pre-defined tasks.
computeEnergy(state)	For the given state, compute an MCMC energy (the lower, the better)
mcmcSweep()	Performs a full MCMC sweep
permuteState()	Proposes a new state to use instead of the old state.
schedule()	Schedules the tasks using an MCMC procedure.
uidTaskMap()	Generates a dictionary for task lookup by UID.

Attributes

State	alias of tuple[int, ...]

State

alias of tuple[int, ...]

computeEnergy(*state: tuple[int, ...]*) → float

For the given state, compute an MCMC energy (the lower, the better)

Args:
state (State): state of the MCMC algorithm

Returns:
float: the energy / penalty for this state

mcmcSweep()

Performs a full MCMC sweep

permuteState() → tuple[int, ...]

Proposes a new state to use instead of the old state.

Returns:
State: the new state, a list of indices within self.tasks representing traversal order

schedule() → Mapping[str, *TimeSlot*]

Schedules the tasks using an MCMC procedure.

Returns:
Mapping[str, TimeSlot]: the resulting map of Tasks to TimeSlots

1.4.7 melon.scheduler.rust

The scheduler algorithm

Classes

RustyMCMCScheduler(tasks)	Markov Chain Monte-Carlo Task Scheduler, implemented in Rust.

melon.scheduler.rust.RustyMCMCScheduler

`class melon.scheduler.rust.`**`RustyMCMCScheduler`**(*tasks: list[melon.scheduler.base.Task]*)

Markov Chain Monte-Carlo Task Scheduler, implemented in Rust.

`__init__`(*tasks: list[melon.scheduler.base.Task]*) → None

Initialises the scheduler, working on a set of pre-defined tasks.

Args:

tasks (list[Task]): the tasks to be scheduled

Methods

`__init__`(tasks)	Initialises the scheduler, working on a set of pre-defined tasks.
`schedule`()	Runs the Rust implementation of the scheduler.
`uidTaskMap`()	Generates a dictionary for task lookup by UID.

`schedule`() → Mapping[str, *TimeSlot*]

Runs the Rust implementation of the scheduler.

Returns:

Mapping[str, TimeSlot]: the resulting schedule

1.5 melon.todo

This module contains the Todo class.

Classes

Todo(*args[, calendarName])	A class representing todos (= tasks), subclassing the caldav.Todo object which in turn stores VTODO data.

1.5.1 melon.todo.Todo

class melon.todo.**Todo**(**args*, *calendarName: str | None = None*, ***kwargs*)

A class representing todos (= tasks), subclassing the caldav.Todo object which in turn stores VTODO data.

__init__(**args*, *calendarName: str | None = None*, ***kwargs*)

Initialises the base class

Methods

__init__(*args[, calendarName])	Initialises the base class
accept_invite([calendar])	
add_attendee(attendee[, no_default_parameters])	For the current (event/todo/journal), add an attendee.
add_organizer()	goes via self.client, finds the principal, figures out the right attendee-format and adds an organizer line to the event
change_attendee_status([attendee])	
children([type])	List children, using a propfind (resourcetype) on the parent object, at depth = 1.
complete([completion_timestamp, ...])	Args:
copy([keep_uid, new_parent])	Events, todos etc can be copied within the same calendar, to another calendar or even to another caldav server
decline_invite([calendar])	
delete()	Delete the object.
expand_rrule(start, end)	This method will transform the calendar content of the event and expand the calendar data from a "master copy" with RRULE set and into a "recurrence set" with RECURRENCE-ID set and no RRULE set.
generate_url()	
get_display_name()	Get calendar display name
get_due()	A VTODO may have due or duration set.
get_duration()	According to the RFC, either DURATION or DUE should be set for a task, but never both - implicitly meaning that DURATION is the difference between DTSTART and DUE (personally I believe that's stupid.
get_properties([props, depth, ...])	Get properties (PROPFIND) for this object.
get_property(prop[, use_cached])	
isComplete()	Returns:
isIncomplete()	Returns:
isTodo()	Returns:
is_invite_request()	
is_loaded()	

continues on next page

Table 2 – continued from previous page

`load`([only_if_unloaded])	(Re)load the object from the caldav server.
`save`([no_overwrite, no_create, obj_type, ...])	Save the object, can be used for creation and update.
`set_due`(due[, move_dtstart, check_dependent])	The RFC specifies that a VTODO cannot have both due and duration, so when setting due, the duration field must be evicted
`set_duration`(duration[, movable_attr])	If DTSTART and DUE/DTEND is already set, one of them should be moved.
`set_properties`([props])	Set properties (PROPPATCH) for this object.
`set_relation`(other[, reltype, set_reverse])	Sets a relation between this object and another object (given by uid or object).
`split_expanded`()	
`tentatively_accept_invite`([calendar])	
`toTask`()	Converts this Todo into the scheduler-compatible Task struct.
`uncomplete`()	Undo completion - marks a completed task as not completed
`upgrade`(todo, calendarName)	A copy constructor constructing a melon.Todo from a caldav.Todo

Attributes

`canonical_url`	
`client`	
`data`	vCal representation of the object as normal string
`dueDate`	Returns:
`dueDateTime`	Returns:
`dueTime`	Returns:
`icalendar_component`	icalendar component - should not be used with recurrence sets
`icalendar_instance`	icalendar instance of the object
`id`	
`instance`	vobject instance of the object
`name`	
`parent`	
`priority`	Returns:
`summary`	Returns:
`uid`	This method has to be fast, as it is accessed very frequently according to profiler output.
`url`	
`vobject_instance`	vobject instance of the object
`vtodo`	Returns the VTODO object stored within me.
`wire_data`	vCal representation of the object in wire format (UTF-8, CRLN)

`complete`(*completion_timestamp: datetime* | *None = None*, *handle_rrule: bool = True*, *rrule_mode: Literal['safe', 'this_and_future'] = 'safe'*) → None

Args:

completion_timestamp (Union[datetime.datetime, None], optional): Argument
(default is None)

handle_rrule (bool, optional): Argument
(default is True)

rrule_mode (Literal['safe', 'this_and_future'], optional): Argument
(default is 'safe')

`property dueDate: date | None`

Returns:
(datetime.date | None):

`property dueDateTime: datetime | None`

Returns:
(datetime.datetime | None):

property dueTime: time | None

Returns:
(datetime.time | None):

isComplete() → bool

Returns:
(bool):

isIncomplete() → bool

Returns:
(bool):

isTodo() → bool

Returns:
bool: whether this object is a VTODO or not (i.e. an event or journal).

property priority: int

Returns:

int: the priority of the task, an integer between 1 and 9,
where 1 corresponds to the highest and 9 to the lowest priority

property summary: str

Returns:
(str):

toTask() → *Task*

Converts this Todo into the scheduler-compatible Task struct.

Returns:
Task: a melon.scheduler.Task

property uid: str | None

This method has to be fast, as it is accessed very frequently according to profiler output. Therefore we use do not use self.vtodo.

Returns:
(Union[str, None]):

static upgrade(*todo: Todo*, *calendarName: str*) → *Todo*

A copy constructor constructing a melon.Todo from a caldav.Todo

Args:
todo (caldav.Todo): Argument calendarName (str): Argument

property vtodo: Component

Returns the VTODO object stored within me. This is faster than accessing the icalendar_component.

Returns:
(vobject.base.Component):

1.6 melon.visualise

A collection of (quality measure) visualisation helpers.

Functions

plotConvergence(data, labels[, filename])	Plots convergence data to a file
radarChart(values, title[, filename])	Plots a helpful priority chart.

1.6.1 melon.visualise.plotConvergence

`melon.visualise.`**`plotConvergence`**(*data: ndarray*, *labels: Sequence*, *filename: str | None = None*)

Plots convergence data to a file

Args:
data (np.array): data of temp, E_avg, E_var filename (str): path to file

1.6.2 melon.visualise.radarChart

`melon.visualise.`**`radarChart`**(*values: tuple[float, float, float]*, *title: str*, *filename: str | None = None*)

Plots a helpful priority chart.

Adapted from https://gist.github.com/sausheong/3997c7ba8f42278866d2d15f9e63f7ad.

Args:
data (tuple[float, float, float]): data title (str): titles of plots

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search