Melon - a Task Scheduling Package for Personal Todo Lists

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 review the central concepts utilised in the group work conducted to make progress in the Partial Differential Equation (PDE) problem associated with the electrochemical model of a battery cell and present numerical results.

Our Goal: Numerically obtain the solution $\{a(x,T), b(x,T)\}$.

The Finite Difference schemes are implemented in Julia and Python, whereas the Spectral Method is implemented in C++.

Figure 1: The Graphical User Interface (GUI) of the Spectral Solver.

1 Problem Introduction

UIDs are useful because they make collisions very unlikely, which is not to say that these should not be checked, but if two clients are connected that each generated a set of UIDs it is very unlikely to have to do conflict resolving.

We recommend usage with `xandikos`, a version-controlled DAV server, capable of syncing calendars (events, todos and journals) and contacts.

Published on PyPi.

1.1 Usage

Use `invoke -l` to list all available tasks.

Is platform-independent, for example due to the usage of `pathlib.Path`.

2 Underlying Theory

The Metropolis-Hastings algorithm (Metropolis et al. 1953; Hastings 1970)

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

3 Code Quality

3.1 Formatting

3.2 Docstrings

3.3 Documentation

3.4 Tests

Server can be started using Docker.

3.4.1 Coverage

3.5 Type Checking

Using `pyright` instead of `mypy` as it is much faster.

3.6 Using Appropriate Language Features

Using `autoflake` and `pyupgrade`. Used `logging`.

3.7 Maintaining Code Quality

Using `pre-commit` and GitHub Actions CI/CD. Uses `invoke` to manage common development tasks.

```
1 import numpy
2 x = 5
3 print(x ** 2)
```

interrogate -v einfügen

Table 1: RESULT: PASSED (minimum: 80.0%, actual: 100.0%)

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	100.0%

Screenshot von gnome-calendar

Screenshot GUI

3.8 Autocorrelation Analysis

3.9 Coole Pie-Plots mit Verteilungen

4 Runtime Performance

```
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)
```

Table 2: 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

Table 3: Profile obtained by running `inv profile-scheduler & grep purepython.py`.

ncalls	tottime	percall	cumtime	percall	filename:lineno	function
1	0.000	0.000	14.236	14.236	purepython.py:142	schedule
10	0.209	0.021	14.235	1.424	purepython.py:123	mcmcSweep
36010	1.144	0.000	13.784	0.000	purepython.py:97	computeEnergy
2196671	5.391	0.000	11.353	0.000	purepython.py:41	spreadTasks
1006332	1.389	0.000	1.842	0.000	purepython.py:27	generateNextSlot
2196610	0.798	0.000	0.972	0.000	purepython.py:108	<genexpr>
2196610	0.384	0.000	0.384	0.000	purepython.py:106	<genexpr>
36000	0.082	0.000	0.217	0.000	purepython.py:83	permuteState
36011	0.046	0.000	0.124	0.000	purepython.py:19	startingSlot

5 Results

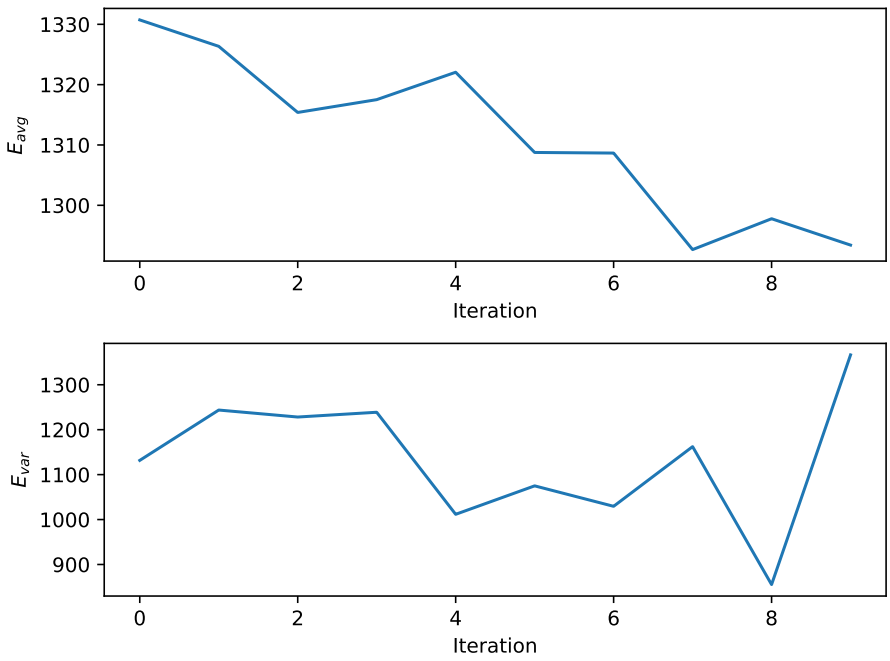

Figure 2: Convergence

Numba Compilation time: 2.5625428539933637

Table 4: Runtime Comparison of the different implementations run on the same scenarios. Each runtime is given as the average over three runs. The finite difference schemes (for the one-dimensional case) were run with $N_x = N_t = 4000$ up to $T = 40$. The spectral method was run using a series expansion of order 15, also up to $T = 40$. The remaining parameters (α, κ_0, E_0, etc.) were all identical.

Implementation	**Language**	**Runtime / seconds**
MCMCScheduler	Python	16.6180
NumbaMCMCScheduler	Python	1.3117
RustyMCMCScheduler	Rust	0.3660
CppMCMCScheduler	C++	0.1513

6 Acknowledgements

The visualisation code (`visualise.py`) is adapted from *Optimising todo lists with Monte Carlo simulations.ipynb* 2023.

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

References

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.

Optimising todo lists with Monte Carlo simulations.ipynb (July 2023). [Online; accessed 1. Jul. 2023]. URL: https://gist.github.com/sausheong/3997c7ba8f42278866d2d15f9e63f7a

Acronyms

GUI	Graphical User Interface	1
PDE	Partial Differential Equation	1

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	

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 SynchronizableCalendarObjectCollection, 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.

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])	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='https://waldcloud.hopto.org:2023/dav/peter/calendars/'*, *username='leser'*, *password='p1140'*)

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='https://waldcloud.hopto.org:2023/dav/peter/calendars/'*, *username='leser'*, *password='p1140'*) → 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”].

Methods

__init__([url, username, password])	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.numba	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)	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.

melon.scheduler.base.generateManyDemoTasks

melon.scheduler.base.**generateManyDemoTasks**(*N: int*) → list[*melon.scheduler.base.Task*]

Generates a larger set of randomly generated demo tasks.

Classes

AbstractScheduler(tasks)	Abstract Base Class (ABC) for schedulers.
Task(uid, duration, priority, location)	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.

schedule() → Mapping[str, *TimeSlot*]

Schedules the tasks using an MCMC procedure.

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

melon.scheduler.base.Task

class melon.scheduler.base.**Task**(*uid: str*, *duration: float*, *priority: int*, *location: int*)

Slim struct representing a task

__init__(*uid: str*, *duration: float*, *priority: int*, *location: int*) → None

Methods

__init__(uid, duration, priority, location)	

Attributes

uid
duration
priority
location

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.

schedule() → Mapping[str, *TimeSlot*]

Runs the Rust implementation of the scheduler.

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

1.4.3 melon.scheduler.numba

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]]*, *state: list[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

melon.scheduler.numba.mcmcSweep

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

Performs a full MCMC sweep

Args:
tasks (Sequence[tuple[str, float, int, int]]): 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]]*) → Sequence[tuple[str, float, float]]

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

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

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]]*) → 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.

schedule() → Mapping[str, *TimeSlot*]

Runs the Rust implementation of the scheduler.

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

1.4.4 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.

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

1.4.5 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.

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:
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.datetime | datetime.date | None):

`property dueTime: time | None`

Returns:
(datetime.datetime | datetime.date | 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

Functions

plotConvergence(data, filename)	Plots convergence data to a file
priorityChart(data, title)	Plots a helpful priority chart

1.6.1 melon.visualise.plotConvergence

melon.visualise.**plotConvergence**(*data: ndarray*, *filename: str*)

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.priorityChart

melon.visualise.**priorityChart**(*data*, *title*)

Plots a helpful priority chart

Args:
data (): data title (str): titles of plots

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m