

Towards “DESDEO 2.0”

Recent developments in the restructuring of DESDEO

DEMO seminar, 12.3.2024

Giovanni Misitano

In this talk...

- Modelling of multiobjective optimization problems
- Parsing and evaluating problems
- Scalarization of problems
- Solving scalarized problems
- Implementing interactive scalarization-based methods
- The new structure of the project
- The new structure of the documentation

NOT in this talk...

- Evolutionary methods
 - The web application programming interface (web-API)
 - Databases
 - User interfaces
 - Visualization components
 - Simulation or surrogate-based problems/methods
-
- **...but in a future talk instead!**

Modelling problems

- Problems are now at the heart of DESDEO.
- The problem model is implemented using pydantic models.
 - Readily serializable, and they come with many other utilities, such as validation.
 - Easy to implement in Python, easy to parse from and to JSON.
 - JSON is an important format when implementing the API, database(s), and user interfaces, and will support many aspects found in interactive multiobjective optimization.
- All mathematical expressions are stored utilizing the MathJSON format.

The problem model

- There is only one problem model in DESDEO now.
- The problem has a name and description, and a bunch of *fields*, which contain other models.

Problem	
name	str
description	str
constants	UnionType[list[Constant], NoneType]
variables	list[Variable]
objectives	list[Objective]
constraints	UnionType[list[Constraint], NoneType]
extra_funcs	UnionType[list[ExtraFunction], NoneType]
scalarizations_funcs	UnionType[list[ScalarizationFunction], NoneType]
discrete_representation	UnionType[DiscreteRepresentation, NoneType]

Constant

- Defines a constant utilized in the definition of a multiobjective optimization problem.
- A constant has a name, a **symbol**, and a value. These are all basic Python variables.
- Having constants is optional.

Constant	
name	str
symbol	str
value	UnionType[float, int, bool]


Variable

- Defines a variable utilized in the definition of a multiobjective optimization problem.
- A variable has a name, **symbol**, variable type, lower and upper bounds, and an initial value.

Variable	
name	str
symbol	str
variable_type	VariableTypeEnum(str, Enum)
lowerbound	UnionType[float, int, bool, NoneType]
upperbound	UnionType[float, int, bool, NoneType]
initial_value	UnionType[float, int, bool, NoneType]

Objective

- Defines an objective function utilized in the definition of a problem.
- An Objective has a name, **symbol**, its function representation, sense (min/max), ideal and nadir values, and the objective's type (analytical or discrete).



Objective	
name	str
symbol	str
func	UnionType[list, NoneType]
maximize	bool
ideal	UnionType[float, NoneType]
nadir	UnionType[float, NoneType]
objective_type	ObjectiveTypeEnum(str, Enum)

Constraint

- Defines a constraint function utilized in the definition of a problem.
- A constraint has a name, a **symbol**, type of constraint (EQ, LTE), its function definition (standard form $g(x) \leq 0$), and whether the constraint is linear or not.

Constraint	
name	str
symbol	str
cons_type	ConstraintTypeEnum(str, Enum)
func	list
linear	bool

Extra function

- Defines any functions that are utilized when defining other parts of a problem.
- Has a name, a **symbol**, and its function representation.

⌞ ExtraFunction	
name	str
symbol	str
func	list

Scalarization function

- Defines any scalarizations of a problem.
- Has a name, a **symbol**, and its function representation.

#	ScalarizationFunction	
	name	str
	symbol	UnionType[str, NoneType]
	func	list

Discrete representation

- Defines any discrete representations of a problem.
- Has variable values and their corresponding objective function values, and whether the representation is non-dominated or not.

DiscreteRepresentation	
variable_values	dict[str, list[UnionType[float, int, bool]]]
objective_values	dict[str, list[float]]
non_dominated	bool

The problem model once more

Problem	
name	str
description	str
constants	UnionType[list[Constant], NoneType]
variables	list[Variable]
objectives	list[Objective]
constraints	UnionType[list[Constraint], NoneType]
extra_funcs	UnionType[list[ExtraFunction], NoneType]
scalarizations_funcs	UnionType[list[ScalarizationFunction], NoneType]
discrete_representation	UnionType[DiscreteRepresentation, NoneType]

Constant	
name	str
symbol	str
value	UnionType[float, int, bool]

Variable	
name	str
symbol	str
variable_type	VariableTypeEnum(str, Enum)
lowerbound	UnionType[float, int, bool, NoneType]
upperbound	UnionType[float, int, bool, NoneType]
initial_value	UnionType[float, int, bool, NoneType]

Objective	
name	str
symbol	str
func	UnionType[list, NoneType]
maximize	bool
ideal	UnionType[float, NoneType]
nadir	UnionType[float, NoneType]
objective_type	ObjectiveTypeEnum(str, Enum)

Constraint	
name	str
symbol	str
cons_type	ConstraintTypeEnum(str, Enum)
func	list
linear	bool

ExtraFunction	
name	str
symbol	str
func	list

ScalarizationFunction	
name	str
symbol	UnionType[str, NoneType]
func	list

DiscreteRepresentation	
variable_values	dict[str, list[UnionType[float, int, bool]]]
objective_values	dict[str, list[float]]
non_dominated	bool

What is so nice about this?

- All the information available and related to the problem is now stored in just one place, i.e., the problem model.
- This model can be readily expanded without breaking any existing functionalities.
- One of the core principles is that everything that is needed to solve the problem, is available in this model. E.g., scalarizations.
- Another important aspect of the model is its immutability. This helps avoid a ton of bugs and keeps the user supplied definition of the problem “pure”. This last point is especially important when we want to solve the same problem multiple times and with different methods.

How are problems defined in practice

Variables

```
from desdeo.problem.schema import Variable

variable_1 = Variable(
    name="BOD", symbol="x_1", variable_type="real", lowerbound=0.3, upperbound=1.0, initial_value=0.65
)
variable_2 = Variable(
    name="DO", symbol="x_2", variable_type="real", lowerbound=0.3, upperbound=1.0, initial_value=0.65
)
```

Function expressions

```
f_1 = "-4.07 - 2.27 * x_1"
f_2 = "-2.60 - 0.03 * x_1 - 0.02 * x_2 - 0.01 / (1.39 - x_1**2) - 0.30 / (1.39 - x_2**2)"
f_3 = "-8.21 + 0.71 / (1.09 - x_1**2)"
f_4 = "-0.96 + 0.96 / (1.09 - x_2**2)"
f_5 = "Max(Abs(x_1 - 0.65), Abs(x_2 - 0.65))"
```

Objective functions

```
from desdeo.problem import Objective

objective_1 = Objective(name="DO city", symbol="f_1", func=f_1, maximize=False)
objective_2 = Objective(name="DO municipality", symbol="f_2", func=f_2, maximize=False)
objective_3 = Objective(name="(negated) ROI fishery", symbol="f_3", func=f_3, maximize=False)
objective_4 = Objective(name="(negated) ROI city", symbol="f_4", func=f_4, maximize=False)
objective_5 = Objective(name="BOD deviation", symbol="f_5", func=f_5, maximize=False)
```

The problem

```
river_problem = Problem(
    name="The river pollution problem",
    description="The river pollution problem to maximize return of investments and minimize pollution.",
    variables=[variable_1, variable_2],
    objectives=[objective_1, objective_2, objective_3, objective_4, objective_5],
)
```


The JSON format

```
{
  "name": "The JSON notation problem",
  "description": "The JSON notation problem is a notation for representing and exchanging
  data.",
  "version": "1.0.0",
  "author": "John Doe",
  "email": "john.doe@example.com",
  "url": "http://example.com",
  "license": "MIT",
  "dependencies": {
    "lodash": "^4.17.11",
    "moment": "^2.29.1",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "^3.4.1",
    "typescript": "^3.9.7"
  },
  "devDependencies": {
    "@types/lodash": "^4.14.168",
    "@types/moment": "^2.29.1",
    "@types/react": "^16.9.55",
    "@types/react-dom": "^16.9.8",
    "@types/react-scripts": "^1.0.0",
    "@types/typescript": "^3.9.7",
    "typescript": "^3.9.7"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "keywords": [
    "JSON",
    "notation",
    "problem"
  ],
  "repository": {
    "type": "git",
    "url": "https://github.com/example/json-notation-problem"
  },
  "bugs": {
    "url": "https://github.com/example/json-notation-problem/issues"
  },
  "homepage": "https://github.com/example/json-notation-problem"
}
```



A closer look

```
{
  "name": "The river pollution problem",
  "description": "The river pollution problem to maximize return of investments and minimize pollution.",
  "constants": null,
  "variables": [
    {
      "name": "BOD",
      "symbol": "x_1",
      "variable_type": "real",
      "lowerbound": 0.3,
      "upperbound": 1.0,
      "initial_value": 0.65
    },
    {
      "name": "DO",
      "symbol": "x_2",
      "variable_type": "real",
      "lowerbound": 0.3,
      "upperbound": 1.0,
      "initial_value": 0.65
    }
  ],
  "objectives": [
    {
      "name": "DO city",
      "symbol": "f_1",
      "func": [
        "Add"
```

Symbols

- As we saw, the function expressions can be supplied in a very understandable format.
- They could also be supplied in a MathJSON format:
- Whichever we choose, the **symbols** of the variables and functions are very important as they are utilized to reference one other.

```
[  
  "Add",  
  ["Divide", "y", "z"],  
  ["Max", "x", "y", "z"],  
  [  
    "Negate",  
    ["Power", ["Sin", ["Add", ["Power", "z", "x"], 4.2]], 3]  
  ]  
]
```

Parsing and evaluating problems

- We have two kinds of parser thus far in DESDEO.
 - From the DESDEO problem model to other formats/models.
 - From other formats/models to the DESDEO problem model.*
- Currently, we can parse the DESDEO problem format to a polars-based format, and to a pyomo-based format.
- We already saw an example of the other type of parser when we defined a problem in the previous example. That is, we can parse function expression from an infix notation to the MathJSON format.

```
f_1 = "-4.07 - 2.27 * x_1"  
f_2 = "-2.60 - 0.03 * x_1 - 0.02 * x_2 - 0.01 / (1.39 - x_1**2) - 0.30 / (1.39 - x_2**2)"  
f_3 = "-8.21 + 0.71 / (1.09 - x_1**2)"  
f_4 = "-0.96 + 0.96 / (1.09 - x_2**2)"  
f_5 = "Max(Abs(x_1 - 0.65), Abs(x_2 - 0.65))"
```

Evaluation

- Parsing and evaluation go together, but evaluators take the problem model a step further and implement general interfaces for different solver (i.e., optimizers) to be able to understand the problem format.
- We have three types of evaluators right now:
 1. a polars-based evaluator,
 2. a naïve discrete evaluator (proximal evaluator, a variant of the polars evaluator),
 3. and a pyomo-based evaluator.

Common aspects among evaluators

- As said, the job of an evaluator is to accommodate various solvers.
- This means that when a solver tries to solve a problem and provides variables, the evaluator will take care that the problem is evaluated with the variables and the results of the evaluation are provided in a compatible format back to the solver.
- In some other cases, an evaluator needs to only provide a correct model for the solver, which is the case with the pyomo evaluator, for instance.

Common aspects among evaluators

- The order of evaluation is, however, universal and should be followed by all evaluators, current and future ones. This is:
 1. Constant are evaluated first. E.g., their symbols are replaced by their corresponding value in all the function expressions found in a problem.
 2. Then extra functions are evaluated with any provided decision variables, and their symbols are replaced in the rest of the problem formulation.
 3. Then objective functions are evaluated and similarly replaced.
 4. Then constraint functions are evaluated and replaced.
 5. Lastly, any scalarization functions are evaluated.

Evaluation order

- In other words, definition wise, scalarization functions can depend on all the other symbols present in the problem.
- Likewise, constraints can depend on all symbols, but those of scalarization functions.
- Objective functions can depend on all symbols, except those of constraints and scalarization functions.
- Etc...

Solvers (optimizers)

- Parsers and evaluators are mostly means to an end, and that end is to utilize solvers.
- DESDEO, however, does not currently implement any optimization routine *per se*, rather, it implements interfaces to existing solvers.
- These interfaces, as one might guess, heavily rely on the evaluators, which on the other hand rely on the parsers.

Solving a problem

- On the surface, once a problem has been defined, solving it is readily achieved:

```
from desdeo.tools import create_scipy_minimize_solver
problem: Problem # a defined problem
solver = create_scipy_minimize_solver(problem)
results: SolverResults = solver("f_1")
```

SolverResult:

```
{
  "optimal_variables": {
    "x_1": 1.0,
    "x_2": 4.0
  },
  "optimal_objectives": {
    "cost": 10.75,
    "time": 8.22
  },
  "constraint_values": {
    "budget": -101.0,
    "deadline": -0.1
  },
  "success": true,
  "message": "Optimization completed successfully."
}
```

N.B., evaluators always provide
a "f_1_min" option!

Solvers

- Python-based solvers
 - Scipy
 - Nevergrad (WIP)
- Other solvers
 - Coin-or-bonmin (mixed-integer, differentiable)
 - Any solver available to pyomo (WIP, e.g., Gurobi, coin-or-branch, almost any AMPL-based solver)
- AMPL = A Mathematical Programming Language

Scalarization

- Solving a multiobjective optimization problem by optimizing one of its objectives is not very exciting.
- We can instead scalarize problems, which is very straightforward:

```
from desdeo.tools import add_asf_nondiff

# a Problem with the objectives 'f_1', 'f_2', 'f_3'
problem: Problem
# a reference point with an aspiration level for each objective function
reference_point = {"f_1": 5.9, "f_2": 4.2, "f_3": -1.6}

problem_w_asf, target = add_asf_nondiff(
    problem,
    symbol="target",
    reference_point=reference_point
)

# to solve
results = solver(target)
```

Scalarization

- Functions that add scalarizations to a problem will return at least a copy of the problem with the added scalarization and the symbol of the added scalarization.
- However, these functions can do much more, such as adding additional constraints to the problem, which is necessary in, e.g., the epsilon-constraint scalarization, or for example when defining the differentiable variant of the achievement scalarizing function.

Scalarization, another example

```
from desdeo.tools import add_epsilon_constraints

# a Problem with the objectives 'f_1', 'f_2', 'f_3'
problem: Problem
# a reference point with an aspiration level for each objective function
epsilons = {"f_1": 5.9, "f_2": 4.2, "f_3": -1.6}
epsilon_symbols = {"f_1": "eps_1", "f_2": "eps_2", "f_3": "eps_3"}

problem_w_eps, target, constraint_symbols = add_epsilon_constraints(
    problem,
    symbol="target",
    constraint_symbols=epsilon_symbols,
    objective_symbol="f_2",
    epsilons=epsilons
)

# to solve
results = solver(target)
```

Scalarization-based methods

- We have all the tools to implement a plethora of the different interactive scalarization-based methods. I.e., “MCDM”, not population-based methods.
- We implement a method by implementing different functions that achieve the different steps required by a method.
- These functions are then combined into the method.

Example: NAUTILUS Navigator

Calculate the iteration/navigation point:

```
def calculate_navigation_point(
    problem: Problem,
    previous_navigation_point: dict[str, float],
    reachable_objective_vector: dict[str, float],
    number_of_steps_remaining: int,
) -> dict[str, float]:
```

Solve for the reachable bounds:

```
def solve_reachable_bounds(
    problem: Problem,
    navigation_point: dict[str, float],
    bounds: dict[str, float] | None = None,
    create_solver: CreateSolverType | None = None,
) -> tuple[dict[str, float], dict[str, float]]:
```

Solve for the reachable solution:

```
def solve_reachable_solution(
    problem: Problem,
    reference_point: dict[str, float],
    previous_nav_point: dict[str, float],
    create_solver: CreateSolverType | None = None,
    bounds: dict[str, float] | None = None,
) -> SolverResults:
```

Calculate the distance to the front:

```
def calculate_distance_to_front(
    problem: Problem,
    navigation_point: dict[str, float],
    reachable_objective_vector: dict[str, float]
) -> float:
```


Example: NAUTILUS Navigator

- The four core functions can be used to implement NAUTILUS Navigator.
- Utilizing them, any other needed functions, such as initialization, and stepping (back and forward) are implemented.
- Functions pre-fixed with **solve** do some kind of optimization, while function pre-fixed with **calculate** do not optimize anything.
- Interaction is implemented in the web-API and frontend, where these functions, and their combinations, are leveraged to enable building interfaces that implement the method.
- Methods can still be implemented using the command line and notebooks as well, but in terms of interaction, assumptions are only made regarding the I/O of the functions implementing a method.

A more functional approach

- In contrast to the previous version of DESDEO, we have taken a much more functional approach when designing the framework, relying less on classes.
- As we saw, adding a scalarization function is implemented as a function that takes all gets all the information it needs to perform the scalarization as its arguments.
- Likewise, scalarization does not modify the original problem, instead it returns a modified copy of it with the added scalarization and other necessary elements.

A more functional approach

- In cases where storing an intermediate state makes sense, we have implemented classes, such is the case with the evaluators and parsers.
- With solvers, we have taken a factory-approach, where we implement a function that returns another specialized function, which has been setup to be able to solve the provided problem.

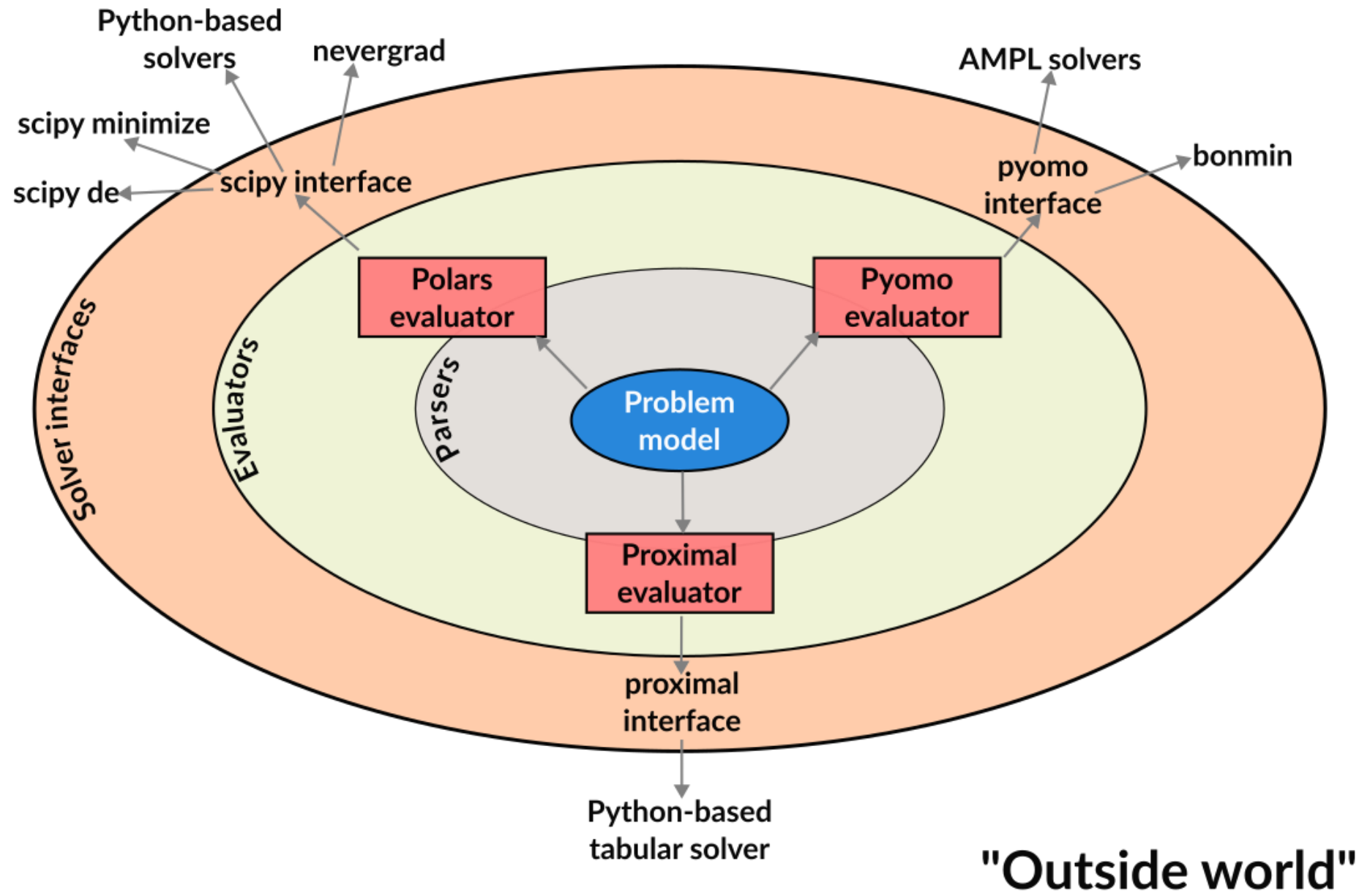
A more functional approach

- Otherwise, we try to be functional where we can.
- This comes with many advantages:
 - less side-effects,
 - purity (always same output with the same arguments),
 - memoization (cache input-output pairs to functions),
 - less bugs, and
 - easier to test.
- Disadvantages:
 - having to pass more arguments to functions,
 - possible redundancies

A more functional approach

- But most importantly, a functional approach allows us to be truly modular.
- It does not take much imagination how different interactive methods can now be combined, switched from and to, and hybridized, when they are implanted as shown in the example.
- Many of the design decisions in the restructuring of DESDEO have been driven by the needs of the web-API (and interfaces), and modularity.
- This has led to a much simpler framework, that is yet much more capable than what we previously had.

So far in a nutshell



The structure of the project

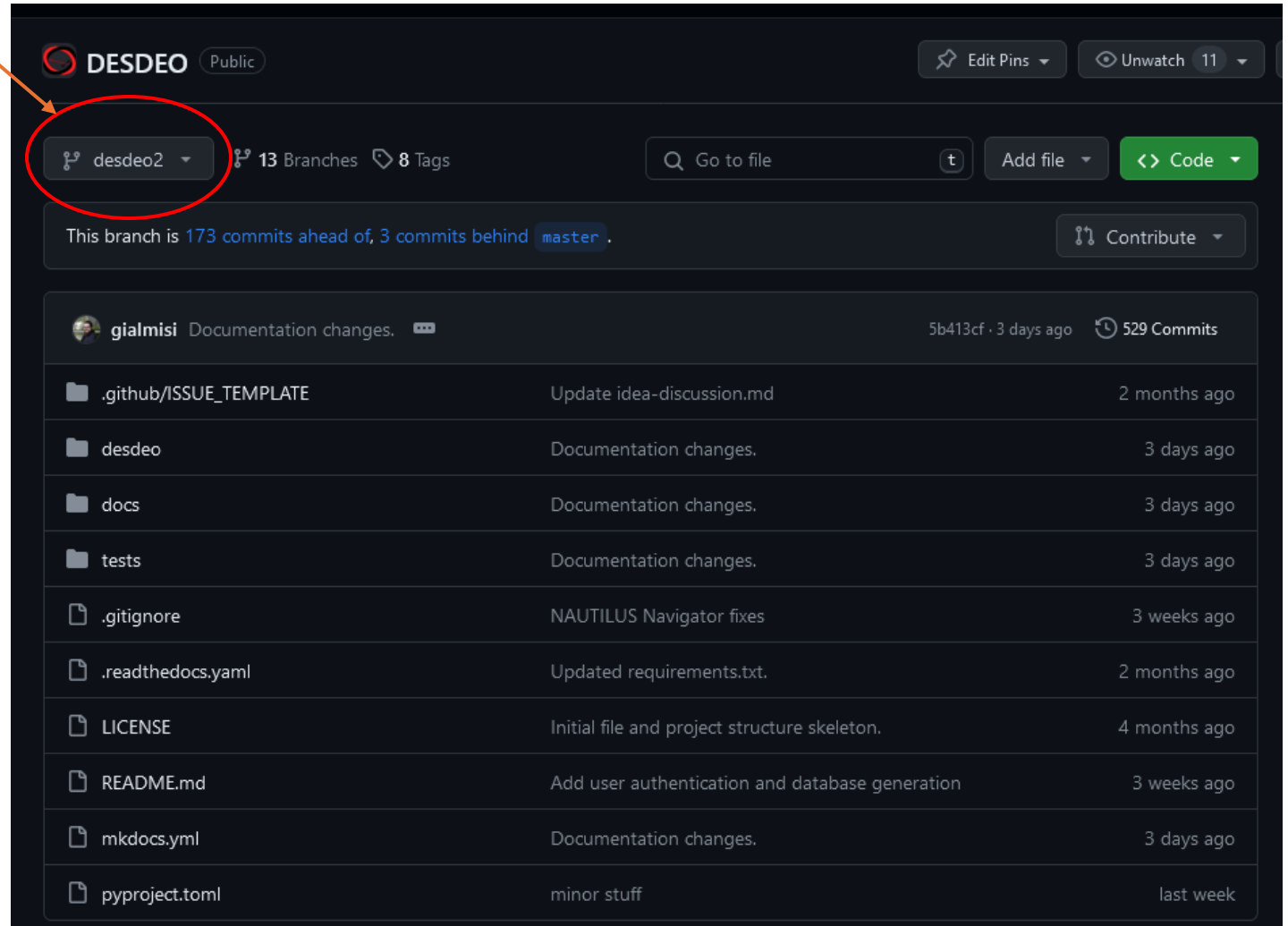
- Previously, we had the *core packages* of DESDEO (desdeo-problem, desdeo-tools, desdeo-mcdm, desdeo-emo) in their own packages and repositories.
- This made working on DESDEO a living nightmare. The original decision was driven by good intentions, but these intentions were far from reality.
- This approach also split the documentation, another living nightmare.

The structure of the project

- Still, having certain functionalities divided into different parts of the framework makes sense. And making this division based on the idea behind the core packages is still sensible.
- But instead as packages, we have desdeo-problem, -tools, -mcdm, and -emo, as modules now. (and desdeo-webapi!)
- There is only a single DESDEO package, and consequently a single documentation as well.

Root of the
project

Obs!



DESDEO Public

desdeo2 13 Branches 8 Tags








Go to file t Add file <> Code

This branch is 173 commits ahead of, 3 commits behind master. Contribute

gialmisi Documentation changes. 5b413cf · 3 days ago 529 Commits

.github/ISSUE_TEMPLATE	Update idea-discussion.md	2 months ago
desdeo	Documentation changes.	3 days ago
docs	Documentation changes.	3 days ago
tests	Documentation changes.	3 days ago
.gitignore	NAUTILUS Navigator fixes	3 weeks ago
.readthedocs.yaml	Updated requirements.txt.	2 months ago
LICENSE	Initial file and project structure skeleton.	4 months ago
README.md	Add user authentication and database generation	3 weeks ago
mkdocs.yml	Documentation changes.	3 days ago
pyproject.toml	minor stuff	last week

DESDEO

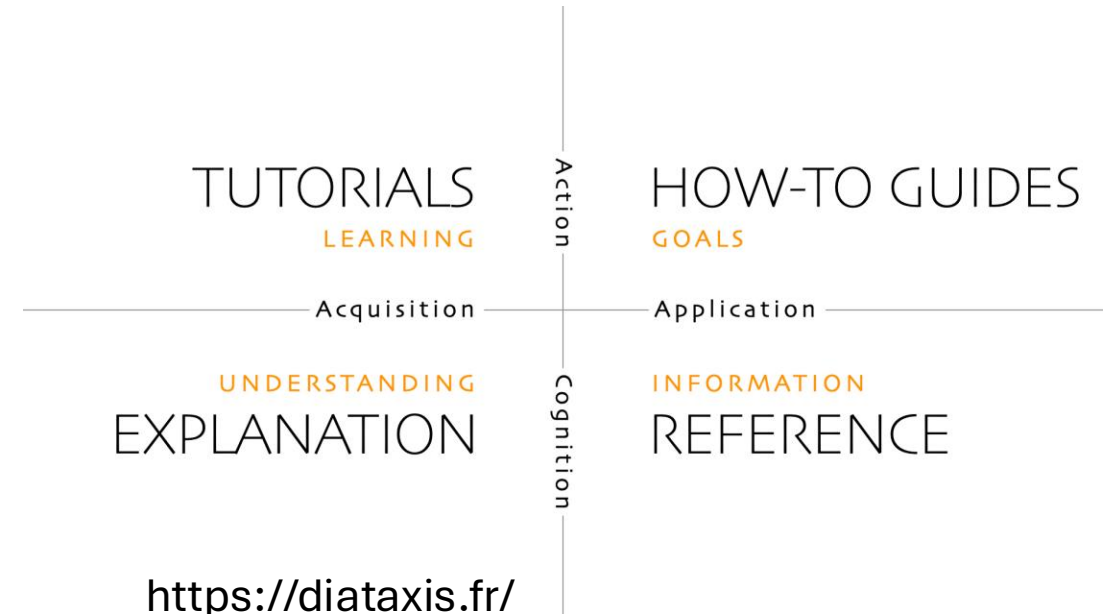
Name	
	..
	api
	emo
	mcdm
	problem
	tools
	__init__.py

desdeo-
problem

Name
..
__init__.py
evaluator.py
infix_parser.py
json_parser.py
pyomo_evaluator.py
schema.py
testproblems.py
utils.py

The documentation

- Has been given a much higher priority.
- Is much more learning- and understanding-oriented.
- Much easier to write than previously.
- Follows (roughly) the Diataxis philosophy



Best to show rather than tell...

- Almost everything, and more, we have seen during this talk, is also present in the documentation.
- Utilized materials for MkDocs.

The minimum viable product

- Capabilities to deal with mixed-integer problems.
 - Is there. Will be improved further.
- Functional interactive plots and UI.
 - Making constant progress.
- Implementations of interactive methods.
 - Not many methods yet, but it is much, much easier to implement methods now.
- Getting archive/database to be functional independent of the method used
 - Web-API stuff, being worked on.

The minimum viable product

- Documentation
 - Is in a better shape than ever. Much easier to work on.
- Support for 2 types of users: registered users and guests
 - Is worked on. We support even more types now.
- Scenarios.
 - ?
- Old codes by previous members.
 - ?

Before the release of 2.0

- Nothing implemented in the EMO side yet.
- Surrogate and simulation-based problems are not supported yet.
- The web-API and the database needs some work as well.
- More methods should be implemented.
- More functionalities, especially scalarization functions, should be implemented.
- More solver interfaces should be implanted.

Contributions are welcome!

Links

- DESDEO 2.0: <https://github.com/industrial-optimization-group/DESDEO/tree/desdeo2>
- The new 2.0 documentation: <https://desdeo.readthedocs.io/en/desdeo2/>
- MathJSON: <https://cortexjs.io/math-json/>
- Pydantic: <https://docs.pydantic.dev/latest/>
- Polars: <https://pola.rs/>
- Pyomo: <http://www.pyomo.org/>
- COIN-OR: <https://www.coin-or.org/>
- Materials for MkDocs: <https://squidfunk.github.io/mkdocs-material/>