MoeLia Suite

Multi-Objective Evolutionary Ju**Li*A_lgorithms*** project emerged with the goal of offering a suite for MOEAs (Multi-Objective Evolutionary Algorithms) where it is possible to utilize not just existing codes and implementations, but develop custom code to integrate it into the MoeLia Suite, following a straightforward and well-documented modular approach.

Getting Started

Actually the project is available to be used only via Julia Pkg or Git.

github repository

Import as dependency in a project

Once started the Julia shell, activate your project using `Pkg.activate` then use `Pkg.add` followed by the github repository link.
Julia will automatically clone the package with all the necessary dependencies and treat that as a simple package to use in the code.
After the cloning you can use the package thanks to `using MoeLia`

Local Forking for Suite Development

After forking and cloning locally the repository, the Julia's Pkg manager can be used through `using Pkg` for then running `Pkg.develop(PackageSpec(path="relative/path/to/MoeLia/"))` to allow the usage of the suite's Package as well as having the advantage of enhancing the MoeLia code.

Business Logic

This section give a brief description of the implementation logic of the suite in order to make that readible and comprehensible. The Suite is composed by four core components:

- Functions
- Implementations
- APIs exposed by moelia_*
- Runners

Functions

Core component exposing reusable functions for general purpose pipelines.
Currently composed by:

- Auxiliaries : generated in order to contains functions for specific algorithms like nsga-II
- Crossovers
- Mutators
- Populators

Implementations

Component exposing the fully implemented MATs (MoeLia Algorithm Types) via a Dictionary implemented in algos.jl .
It actually contains the following modules:

- Algos : implements MATs and exposes those through a Dictionary
- Implementations : exposes APIs to access the Dictionary

APIs

Set of core modules that exposes all the necessary data structures and their relative APIs.
By convention the files exposing the APIs are named with Pascal case.

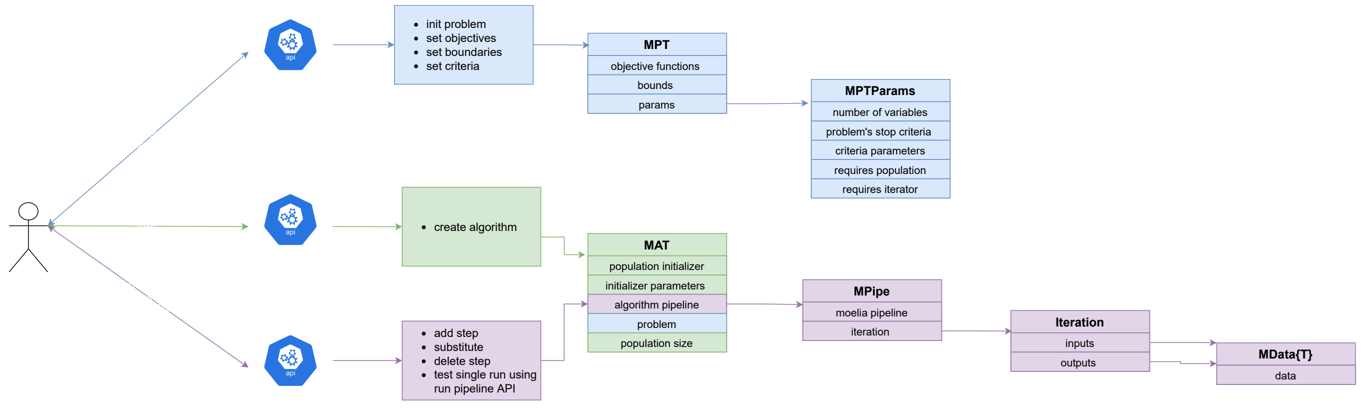
- MoeLia problem : logical component exposing the MPT (MoeLia Problem Type) that allows to define several parameters related to the problem itself and some mutable parameters about secondary problem attributes.
  - MPTParams (mutable struct)
    - criteria served as Function
    - number of variable
    - number of population
    - ...
  - MPT
    - objective functions : served as a vector of Function Julia type
    - bounds : consisting of a vector of Tuples where the first element represents the lower bound and the second the upper one
    - params : for optional parameters
- MoeLia algorithms : logical component exposing the MAT (MoeLia Algorithm Type) that allows to define several parameters related to itself.
  - population initializer
  - initializer parameters
  - algorithm pipeline
  - problem : **of type MPT**
  - population size
- MoeLia pipeline : logical component containing the structs for the Core MPipe (Moelia Pipeline) Object.
  - MData keeps track of the single instance of a data and its type
    - data
  - Iteration keeps track of the overall steps of the pipeline corresponding to a single run of it
    - inputs
    - outputs
  - MPipe core objects representing the pipeline
    - mpipe : consisting of a Vector of tuples describing the steps

  - iter : representing the history of the pipeline, each iteration keeps track of both input and output data relative to its run

Runners

Exposes several versions of the running loop for the Multi-Objective Evolutionary Algorithms, to be specific a basic version and a verbose version allowing to see details about input/output data.

Development Flow



How to develop using MoeLia

The MoeLia suite aims to be modular and extensible in order to encourage contributions. Here are the basic steps to develop using MoeLia:

- After including the MoeLia package as indicated in previous chapter, use the MoeliaProblem APIs to create a new problem instance specifying the objective functions, variable bounds etc. Thanks to its modularity it is very easy to pass self-implemented functions or re-use existing ones.
- Once an instance of MPT has been created, use Moelia Algorithm APIs to define an empty algorithm where the populator and other algorithm-relative components can be plugged in.
- Finally interact with the created MAT by using Moelia Pipeline APIs to alter the behavior of the core loop or extend it with custom steps. The pipeline keeps track of iterations allowing to access inputs, outputs and history in a modular way. The pipeline has been developed to be fully flexible allowing the usage of self-made components as well as re-using existing ones from the Functions module. It also allows to clone, and set different versions of an algorithm so that various experiments can be conducted with ease.

The package is included with some code examples that can be accessed by cloning the "examples" folder. Those demonstrate how to create a problem, define an algorithm, run a pipeline and access results in a very detailed way including modalities where anonymous functions are set in the correct way to access other step's inputs without altering the core runners behavior:

- pipeline_test.jl => Example of how to define and run a basic pipeline using existing implementations as well as researcher-defined ones.
- from_scratch.jl => Full example of problem creation, algorithm definition and running a pipeline from scratch mixing together existing components with researcher-defined ones.

- implementations_test.jl => Example of how to access existing implementations directly and run them. It also shows how to correctly clone and modify a fully implemented algorithm's pipeline to allow testing variants of suite's algorithms.

How to enhance the suite

As mentioned, the package is meant to be easily extensible and modifiable, for this reason conventions have been put in place to allow contributions in a smooth way:

- Create a new module under the "Functions" component to expose reusable functions for general purpose pipelines. This allows others to import and reuse utility functions. Modules must be named descriptively (e.g. "Crossovers" inside "crossovers" folder) and only exported by Functions.jl

- Enhance the "Implementations" component by adding a new MAT implementation module under "Algos" that defines a new algorithm type and registers it with the dictionary.

- APIs can be enhanced by adding new logical components or extending existing ones like MPTParams to support additional problem attributes; define new APIs in files using PascalCase convention (e.g. "MoeliaPipeline.jl" implements APIs for structs defined in "mtypes.jl")

- MoeLia.jl must work as entry point, hence it must include and export all the library's modules intended to be accessed externally.

- Document every implementation to keep code maintainability.

Dependencies

MoeLia actually requires Random and Dates as dependencies. If more are to be added use `Pkg.add` after the package activation to also include them in the `.toml` files of the suite in order to be used with the `using` keyword in external files.

Future Implementations

Disable Run History

Implement a functionality in the MoeliaPipeline and Runner modalities to disable input/output history entirely or partially, tracking only specific iterations. This can be achieved by creating a version of `run_pipeline` with a `disable_history` parameter, preventing data from being pushed. Subsequently, a runner version with the same `disable_history` parameter should be implemented. This runner will execute the original `run_pipeline` at the beginning and end, while using the `disable_history` version for the remaining steps.

Plotting Functionality

As in the actual state, the library only allows for plotting by introducing a step in the pipeline or a version of a runner function that introduces the last population's plotting. A possible future implementation for the Library could be to incorporate a built-in way to plot results by exploiting external packages like Plots that could be integrated with other packages for handling better data visualization and 3D shapes like:

- CalculusWithJulia
- Contour which is integrated with the latest versions of Julia Plots.