The Journal of Open Source Software

A parallel global multiobjective framework for optimization: pagmo

Francesco Biscani[1] **and Dario Izzo**[2]

1 Max Planck Institute for Astronomy (Heidelberg, D) **2** Advanced Concepts Team, European Space Research and Technology Center (Noordwijk, NL)

DOI: 10.21105/joss.02338

Software

- Review
- Repository
- Archive

Editor: Eloisa Bentivegna

Reviewers:

- @dgoldri25
- @jangmys

Submitted: 08 June 2020
Published: 15 June 2020

Summary

Mathematical optimization is pervasive in all quantitative sciences. The ability to find good parameters values in a generic numerical experiment while meeting complex constraints is of great importance and, as such, has always been an active research topic of mathematics, numerics and, more recently, artificial intelligence.

Given the vast amount and diversity of optimization problems, as well as of solution approaches, and considering the need to be able to exploit modern computational architectures, the development of a tool able to help in such a pervasive task is not trivial.

In this paper we introduce `pagmo`, a C++ scientific library for massively parallel optimization. `pagmo` is built around the idea of providing a unified interface to optimization algorithms and problems, and to make their deployment in massively parallel environments easy.

Efficient implementantions of bio-inspired and evolutionary algorithms are sided to state-of-the-art optimization algorithms (Simplex Methods, SQP methods, interior points methods, etc.) and can be used cuncurrently (also together with algorithms coded by the user) to build an optimization pipeline exploiting algorithmic cooperation via the asynchronous, generalized island model (Izzo, Ruciński, & Biscani, 2012)

`pagmo` can be used to solve constrained, unconstrained, single objective, multiple objectives, continuous and integer optimization problems, stochastic and deterministic problems, as well as to perform research on novel algorithms and paradigms and easily compare them to state-of-the-art implementations of established ones.

For users that are more comfortable with the Python language, the package `pygmo` following as closely as possible the `pagmo` API is also available.

The optimization problem

In `pagmo` optimization problems are considered to be in the form:

$$\begin{aligned} \text{find:} \quad & \mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \\ \text{to minimize:} \quad & \mathbf{f}(\mathbf{x}, s) \in \mathbb{R}^{n_{obj}} \\ \text{subject to:} \quad & \mathbf{c}_e(\mathbf{x}, s) = 0 \\ & \mathbf{c}_i(\mathbf{x}, s) \leq 0 \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^{n_{cx}} \times \mathbb{Z}^{n_{ix}}$ is called *decision vector* or *chromosome*, and is made of n_{cx} real numbers and n_{ix} integers (all represented as doubles). The total problem dimension is then indicated with $n_x = n_{cx} + n_{ix}$. $\mathbf{lb}, \mathbf{ub} \in \mathbb{R}^{n_{cx}} \times \mathbb{Z}^{n_{ix}}$ are the *box-bounds*, $\mathbf{f} : \mathbb{R}^{n_{cx}} \times \mathbb{Z}^{n_{ix}} \rightarrow$

$\mathbb{R}^{n_{obj}}$ define the *objectives*, $\mathbf{c}_e : \mathbb{R}^{n_{cx}} \times \mathbb{Z}^{n_{ix}} \to \mathbb{R}^{n_{ec}}$ are non linear *equality constraints*, and $\mathbf{c}_i : \mathbb{R}^{n_{cx}} \times \mathbb{Z}^{n_{ix}} \to \mathbb{R}^{n_{ic}}$ are non linear *inequality constraints*. Note that the objectives and constraints also depend from an added value s representing some stochastic variable. Both equality and inequality constraints are considered as satisfied whenever their definition is met within a tolerance $\mathbf{c}_{tol}$.

Note that there is no special treatment of a possible linear part of the *objectives* or *constraints*, and as such solvers in pagmo have no additional help to approach linear programming tasks.

Given the generic form used to represent a problem, pagmo is suitable to solve a broad range of optimization problems, ranging from single and multiobjective problems to box-bounded and non linearly constrained problems to stochastic problems to continuous, integer and mixed integer problems.

The pagmo jargon

The discussion on the relation between artificial evolution and mathematical optimization is an interesting one (Smith, 1978). In pagmo optimization, of all types, is regarded as a form of evolution. Solving an optimization problem is, in pagmo, described as *evolving* a *population*. Each *decision vector* is thus referred to also as *chromosome* and its *fitness* is defined as a vector containing *objectives*, *equality constraints* and *inequality constraints* in this order. Regardless on whether the user is using, as a solver, a sequential quadratic programming approach, an interior point optimizer an evolutionary strategy or some meta-heuristic, in pagmo he will always have to call a method called *evolve* to improve over the initial solutions stored in a *population*. A *population* may or may not live in an *island*. When it does, its *evolution* is delegated to a different computational unit (a process, thread or remote CPU). Stretching this jargon even further, in pagmo a set of *islands* concurring to the same optimization is called an *archipelago*. When solutions are also exchanged among *populations* living on the same *archipelago*, the quality of the overall optimization is often improved (Izzo et al., 2012). This exchange of information among different solvers is referred to as *migrations* and the allowed migration routes (affecting the overall process significantly (Ruciński, Izzo, & Biscani, 2010)) as the *topology* of the *archipelago*.

Exploiting parallelism

Parallelizing optimization tasks in a generic fashion is one of the leading software design principles of pagmo. According to the type of optimization task, and in particular to the computational weight of computing the problem *fitness* function, a different granularity of the parallelization option may be ideal.

Island Model

As a coarse-grained parallelism, pagmo offers an implementation of the so-called generalized island model (Izzo et al., 2012). Early ideas on distributing genetic algorithms over multiple CPUs were developed in the early 90s by Reiko Tanese, one of John Holland's students (Tanese, 1989). The idea that migrations could improve the quality of the solutions obtained for some optimization task as well as offer a quasi-linear speedup was, though, confined mainly to genetic algorithms and called island model. In pagmo any solver, inspired by the darwinian evolution paradigm, by swarm intelligence, by any meta-heuristics or based on mathematical optimality conditions is allowed to exchange information during an *evolution* with other solvers connected to it via defined *migration* paths.

Cuncurrent fitness evaluations

In some situations it is preferable to parallelize at a finer grain the *evolution* pipeline (e.g., if the objective function evaluation is extremely costly). For this purpose, pagmo provides a *batch fitness evaluation* framework which can be used by selected algorithms to perform in parallel the objective function evaluation of multiple independent decision vectors. The parallel evaluation can be performed by multiple threads, processes, nodes in an HPC cluster or even by GPU devices (via, e.g., OpenCL or CUDA).

Code Design

C++

pagmo is written in standard-compliant C++17, and it extensively employs modern programming techniques. *Type erasure* is used pervasively throughout the codebase to provide a form of runtime polymorphism which is safer and more ergonomic than traditional object-oriented programming. Template meta-programming techniques are used for compile-time introspection, and, with the help of sensible defaults, they allow to minimise the amount of boilerplate needed to define new optimisation problems. pagmo is designed for extensive customisation: any element of the framework (including solvers, islands, batch fitness evaluators, archipelago topologies, migration policies, etc.) can easily be replaced with custom implementations tailored for specific needs.

Python

In order to provide an interactive mode of usage (and in order to participate in the ecosystem of what is arguably the most popular language for scientific computing today), pagmo provides a complete set of Python bindings called pygmo, implemented via `pybind11` (Jakob, Rhinelander, & Moldovan, 2017). pygmo exposes all pagmo features, including the ability to implement new problems, solvers, batch evaluators, topologies etc. in pure Python, using an API which closely matches the C++ pagmo API. Additionally, pygmo offers Python-specific features, such as the ability to use `ipyparallel` (Ragan-Kelley, 2020) for cluster-level parallelisation, and wrappers to use optimisation algorithms from `Scipy` (Virtanen et al., 2020) as pygmo algorithms.

Testing and documentation

The pagmo development team places a strong emphasis on automated testing. The code is fully covered by unit tests, and the continuous integration pipeline checks that the code compiles and runs correctly on a variety of operating systems (Linux, OSX, Windows) using different compilers (GCC, Clang, MSVC). Both the C++ and Python APIs are fully documented, and as a policy we require that every PR to pagmo or pygmo must not decrease testing or documentation coverage.

Some API examples

In this section, we will show how pagmo and pygmo can be used to solve a very simple optimisation problem using the Differential Evolution (DE) algorithm (Storn & Price, 1997). The problem that we will solve is the minimisation of the unidimensional sphere function,

$$f(x) = x^2,$$

subject to the box bounds $x \in [0, 1]$. This is, of course, a trivial problem with solution $x = 0$, and it is used here only for didactic purposes.

C++

```
#include <iostream>
#include <utility>

#include <pagmo/algorithm.hpp>
#include <pagmo/algorithms/de.hpp>
#include <pagmo/population.hpp>
#include <pagmo/problem.hpp>
#include <pagmo/types.hpp>

using namespace pagmo;

// Definition of the optimisation problem.
struct sphere_1d
{
  // Definition of the box bounds.
  std::pair<vector_double, vector_double> get_bounds() const
  {
    return {{0.}, {1.}};
  }
  // Definition of the objective function.
  vector_double fitness(const vector_double &dv) const
  {
    return {dv[0] * dv[0]};
  }
};

int main()
{
  // Create a random population of 20 initial
  // guesses for the sphere_1d problem.
  population pop{sphere_1d{}, 20};

  // Create the optimisation algorithm.
  // We will use 500 generations.
  algorithm algo{de{500}};

  // Run the optimisation, which will
  // produce a new "evolved" population.
  auto new_pop = algo.evolve(pop);

  // Print to screen the fitness of the
  // best solution in the new population.
  std::cout << "Fitness of the best solution: "
            << new_pop.champion_f()[0] << '\n';
}
```

In pagmo, decision vectors and problem bounds are represented via the `pagmo::vector_double` type, which is currently just an alias for `std::vector<double>`. The fitness function also returns a `vector_double`, because, generally-speaking, the fitness vector must accommodate

multiple scalar values to represent multiple objectives and constraints. Here, however, the `sphere_1d` problem is single-objective and unconstrained, and thus the only element in the fitness vector will be the value of the objective function.

In this example, 20 initial conditions for the optimisation are randomly chosen within the problem bounds when creating the `pop` object. It is of course possible to set explicitly the initial conditions, if so desired. The Differential Evolution algorithm object is then created, specifying 500 generations as a stopping criterion.

The initial population `pop` is then evolved, and the result is a new population of optimised decision vectors, `new_pop`. The fitness of the best decision vector (the "champion") is then printed to screen.

`sphere_1d`, as an unconstrained, single-objective, continuous optimisation problem, is the simplest optimisation problem type that can be defined in `pagmo`. More complex problems can be defined by adding new member functions to the problem class. For instance:

- by implementing the `get_nec()` and `get_nic()` member functions, the user can specify the number of, respectively, equality and inequality constraints in the problem. If, like in the case of `sphere_1d`, these functions are not implemented, `pagmo` assumes that the problem is unconstrained;
- by implementing the `get_nobj()` member function, the user can specify the number of objectives in the optimisation problem. If this function is not implemented, `pagmo` assumes that the problem is single-objective.

Python

```
from pygmo import problem, algorithm, population, de

# Definition of the optimisation problem.
class sphere_1d:
  # Definition of the box bounds.
  def get_bounds(self):
    return ([0], [1])
  # Definition of the objective function.
  def fitness(self, dv):
    return [dv[0]**2]

# Create a random population of 20 initial
# guesses for the sphere_1d problem.
pop = population(sphere_1d(), 20)

# Create the optimisation algorithm.
algo = algorithm(de(500))

# Run the optimisation, which will
# produce a new "evolved" population.
new_pop = algo.evolve(pop)

# Print to screen the fitness of the
# best solution in the new population.
print(new_pop.champion_f)
```

As shown in this example, the `pygmo` Python API very closely follows the `pagmo` C++ API.

`pygmo` seamlessly integrates with the wider scientific Python ecosystem. For instance:

- in addition to generic Python iterables (list, tuples, etc.), NumPy arrays (Walt, Colbert, & Varoquaux, 2011) can be used as data types to represent decision vectors, constraints, gradients, Hessians, etc.;
- various optimisation analysis tools based on Matplotlib (Hunter, 2007) are provided;
- archipelago topologies can be exported, imported and studied as NetworkX graph objects (Hagberg, Swart, & S Chult, 2008).

Availability

Both `pagmo` and `pygmo` are available in the `conda` package manager through the `conda-forge` community-driven channel. Additionally, the core team also maintains `pip` packages for Linux.

The wider `pagmo` user community provides also additional packages for Arch Linux, OSX (via Homebrew) and FreeBSD.

Acknowledgments

We acknowledge the support of the Google Summer of Code initiative, the European Space Agency Summer of Code in Space and Dow Corporation during different phases of the development. Many of our colleagues and friends have, in the years, supported the project contributing to evolve its code base and API to what we have today. We would like to mention, in particular, Luís Felismino Simões, Marek Ruciński, Marcus Märtens, Krzysztof Nowak, Giacomo Acciarini and Moritz v. Looz.

References

Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using networkx*. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55

Izzo, D., Ruciński, M., & Biscani, F. (2012). The generalized island model. In *Parallel architectures and bioinspired algorithms* (pp. 151–169). Springer.

Jakob, W., Rhinelander, J., & Moldovan, D. (2017). Pybind11 – seamless operability between c++11 and python.

Ragan-Kelley, M. (2020). Ipyparallel: Interactive parallel computing in python. *GitHub repository*. GitHub. Retrieved from https://github.com/ipython/ipyparallel

Ruciński, M., Izzo, D., & Biscani, F. (2010). On the impact of the migration topology on the island model. *Parallel Computing*, *36*(10-11), 555–571. doi:10.1016/j.parco.2010.04.002

Smith, J. M. (1978). Optimization theory in evolution. *Annual Review of Ecology and Systematics*, *9*(1), 31–56.

Storn, R., & Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, *11*(4), 341–359.

Tanese, R. (1989). Distributed genetic algorithms for function optimization.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi:10.1038/s41592-019-0686-2

Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30. doi:10.1109/mcse.2011.37