ARTICLE IN PRESS

Applied Soft Computing Journal xxx (xxxx) xxx

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

Liger: A cross-platform open-source integrated optimization and decision-making environment

João A. Duro [a,*], Yiming Yan [a], Ioannis Giagkiozis [a], Stefanos Giagkiozis [a], Shaul Salomon [a,b], Daniel C. Oara [a], Ambuj K. Sriwastava [a], Jacqui Morison [c], Claire M. Freeman [c], Robert J. Lygoe [d], Robin C. Purshouse [a], Peter J. Fleming [a]

[a] Department of Automatic Control and Systems Engineering, The University of Sheffield, UK
[b] Department of Mechanical Engineering, Ort Braude College of Engineering, Karmiel, Israel
[c] Jaguar Land Rover Limited, UK
[d] Product Development Europe, Dunton Technical Centre, Ford Motor Co. Ltd, UK

ARTICLE INFO

Article history:
Received 6 January 2020
Received in revised form 11 September 2020
Accepted 22 October 2020
Available online xxxx

Keywords:
Software engineering
Multi-objective optimization
Multi-criteria decision-making
Evolutionary algorithms
Metaheuristics

ABSTRACT

Real-world optimization problems involving multiple conflicting objectives are commonly best solved using multi-objective optimization as this provides decision-makers with a family of trade-off solutions. However, the complexity of using multi-objective optimization algorithms often impedes the optimization process. Knowing which optimization algorithm is the most suitable for the given problem, or even which setup parameters to pick, requires someone to be an optimization specialist. The lack of supporting software that is readily available, easy to use and transparent can lead to increased design times and increased cost. To address these challenges, Liger is presented. Liger has been designed for ease of use in industry by non-specialists in optimization. The user interacts with Liger via a visual programming language to create an optimization workflow, enabling the user to solve an optimization problem. Liger contains a novel optimization library known as Tigon. The library utilizes the concept of design patterns to enable the composition of optimization algorithms by making use of simple reusable operator nodes. The library offers a varied range of multi-objective evolutionary algorithms which cover different paradigms in evolutionary computation; and supports a wide variety of problem types, including support for using more than one programming language at a time to implement the optimization model. Additionally, Liger functionality can be easily extended by plugins that provide access to state-of-the-art visualization tools and are responsible for managing the graphical user interface. Lastly, new user-driven interactive capabilities are shown to facilitate the decision-making process and are demonstrated on a control engineering optimization problem.

1. Introduction

For a company to remain at the competitive edge of the industry in which it is involved, its products and services need to account for as many performance objectives as possible, which may include reduced cost, improved safety, higher quality, amongst others. When two or more conflicting objectives are involved in an optimization problem the task is to find multiple optimal trade-off solutions, where no solution is said to be better with respect to all objectives. Given that classical optimization methods are only capable of finding one single optimized solution in a single optimization run, the alternative is to rely on evolutionary algorithms since their population-based approach is capable of finding multiple optimal solutions in its final population [1]. Multi-objective Evolutionary Algorithms (MOEAs) have advanced with great pace over the last 30 years in many respects, and are immensely valuable for real-world applications [2].

However, a general problem facing any non-expert in optimization could well be which optimization algorithm is best suited for dealing with the optimization problem at hand. To make matters worse, most optimizers require their parameters to be tweaked, and may also require some other form of setup such as which constraint handling approach to use [3]. It is therefore not surprising to find practitioners making poor decisions and using inappropriate methods for dealing with their problems, which

* Corresponding author.
E-mail addresses: j.a.duro@sheffield.ac.uk (J.A. Duro), yiming.yan@outlook.com (Y. Yan), i.giagkiozis@protonmail.com (I. Giagkiozis), stevegiagkiozis@gmail.com (S. Giagkiozis), shaulsal@braude.ac.il (S. Salomon), dcoara1@sheffield.ac.uk (D.C. Oara), a.k.sriwastava@sheffield.ac.uk (A.K. Sriwastava), jmoriso1@jaguarlandrover.com (J. Morison), cfreem48@jaguarlandrover.com (C.M. Freeman), blygoe@ford.com (R.J. Lygoe), r.purshouse@sheffield.ac.uk (R.C. Purshouse), p.fleming@sheffield.ac.uk (P.J. Fleming).

https://doi.org/10.1016/j.asoc.2020.106851

Please cite this article as: J.A. Duro, Y. Yan, I. Giagkiozis et al., Liger: A cross-platform open-source integrated optimization and decision-making environment, Applied Soft Computing Journal (2020) 106851, https://doi.org/10.1016/j.asoc.2020.106851.

could be because they are not familiar with anything better. Another aspect concerns those MOEAs that elicit the Decision-Maker's (DM's) preferences, a-priori or progressively, in order to guide the search towards the most preferred solution by the DM [4–6]. These algorithms are often known in the literature as Multiple Criteria Decision-Making (MCDM) based MOEAs [7,8]. One of the major advantages of these algorithms is that it may be only required to find a crowded set of solutions near the most preferred solution by the DM as opposed to the complete Pareto-optimal Front (POF); however, it is also the case that their applicability may be impaired owning to the subjectivity and cognitive limitation of the DM [7]. To counter these issues, our argument is that there is a need for an optimization software with an intuitive Graphical User Interface (GUI) that:

1. offers an interface that is simple and straightforward to use by the non-expert in optimization;
2. accommodates the needs of researchers by providing an open, flexible, reusable and sustainable optimization environment, and;
3. that incorporates decision-making support tools which could enable the DMs to articulate their preferences more rationally, and to assist in the process of identifying the most preferred solution.

To address the above, the Liger software has been designed and developed. Liger is a cross-platform open-source optimization and decision-making support software written in C++. The intent behind the development of Liger is to create an easy-to-use optimization software while at the same time providing a versatile optimization framework. The aim is for an extensible piece of software to be used "out of the box". It is built upon a visual programming language that is used to create what we call an "optimization workflow". A workflow contains a set of operator nodes connected in a way that resembles the optimization process. There are different operators to choose from and a workflow can be created via a simple drag-and-drop functionality that is provided by the GUI. Some of the tasks performed by the operators are for instance: to load the optimization problem, initialize a population of solutions, run an optimizer, and display the obtained solutions in a graphical plot. For this, Liger provides access to a diversified list of MOEAs, offers a rich set of (visualization) tools that provide an interactive decision-making support experience, and its functionality can be easily extended via a plugin system. Liger was first introduced to the community at GECCO 2013 [9] and its key points are a design that is extensible, easy to use by the non-expert in industry, and an application built around a visual programming language on which optimization workflows can be created. This version of Liger has been demonstrated on an industry–led case study involving the calibration of a diesel engine [10], where it was shown that Liger is able to support the analyst and DM throughout the process of obtaining an optimized engine calibration that complies with performance and regulatory requirements. Following our involvement on other projects that required using Liger to solve other real-world optimization problems, many of those suggested by our industrial partners (e.g. Ford and Jaguar Land Rover), several enhancements and new advancements have been made, and in that, a new version has been released[1] and its distinctive contribution when compared with the old version relates to:

(I) A new optimization library, namely Tigon. The version in [9] relied on the jMetal library as the underlying optimization library. However, the fact that jMetal is implemented in Java makes it less suitable for intensive computing tasks in general due to lower run-time performance when compared with other libraries that can run directly on the computer's processor (e.g. C and C++). The new library provides a modular and flexible approach towards the design of optimization algorithms.

(II) Simplified workflow interface. Previously in [9] the order of execution within an optimization workflow bears some resemblance to the Simulink engine,[2] implying that the nodes in the workflow had to be connected in a feedback loop. However, this can be confusing for those users that are not familiar with the concept of feedback loop, or that have no experience with the Simulink engine. In the new version the nodes are all connected in series without the need for a feedback loop. The new interface is therefore more intuitive and simpler to use.

(III) Enhanced interactive decision-making. In [9] decision-making could only take place at the end of the optimization run, where a DM could select the most desirable solution from amongst the available options by making use of the Liger visualization tools (e.g. interaction with a parallel coordinates plot). In the new version:

   (a) The user is able to inspect the obtained solutions at any point during the optimization run by pausing the optimization engine before a termination criterion is satisfied (e.g. before the number of iterations is exhausted).
   (b) There is support for progressive articulation of the DM's preferences during the optimization run when using Pareto-based MOEAs. For this the DM can specify a goal vector in objective space to steer the search towards the most desirable solution. This relies on the preferability relation concept defined in [4] that is used to induce an order between the solutions based on the provided goals. The goal vector can be defined before the optimization run starts, and can be modified after the user pauses the optimization run and inspects the existing solutions.
   (c) Real-time visualization of the obtained solutions during the optimization run in the Liger visualization tools. The user is able to specify the update rate, either with respect to iterations or time in seconds.

The rest of this paper is organized as follows. Following the above short introduction to Liger, we discuss the pros and cons of other optimization libraries in Section 2. The general concept behind Liger and its architecture is described in Section 3, and the Tigon optimization library is described in Section 4. Section 5 explains how to compose an optimization algorithm by using the Tigon operators. Section 6 describes the GUI, provides a short tutorial on how to create, setup and run a simple optimization workflow, and also demonstrates the interactive decision-making capabilities of Liger on a control engineering optimization problem. The paper concludes with a summary in Section 7.

2. Related work

A large number of MOEAs are found in the literature [11] but several factors prevent researchers and other optimization practitioners from using them. For instance, the authors of such optimizers often do not provide access to the source code, and even if they do, several steps need to be taken before a user is able to start the optimization process. To facilitate this, several software libraries have been proposed over the past 20 years.

[1] Liger is released under the LGPL licence and its source code is hosted in GitHub, https://github.com/ligerdev/liger.

[2] Simulink is a graphical block diagramming programming environment developed by MathWorks.

In the following, we will focus on libraries with an open-source licence.

One example is known as PISA [12], a modular framework that contains a library of MOEAs, optimization problems, and also performance assessment tools. The framework relies on a text-based interface implemented in C that splits an optimization process into two modules. The first module contains parts specific to the optimization problem, and the second module deals with other parts that are independent from the optimization problem (mainly related to the selection process of the MOEA). However, the communication between modules relies on text files, implying that the time required to complete the optimization process is mostly spent waiting for the completion of hard drive input–output operations. This makes PISA not so suitable for intensive computing tasks when compared with other libraries that do not rely on the hard drive and take advantage of faster memories, such as the Random Access Memory (RAM). Another example is ParadisEO-MOEO [13], an object-oriented framework implemented in C++ specifically dedicated to the design of MOEAs, which is an extended version of the Evolving Objects (EO) library [14]. The EO library provides a set of generic components that are used to build flexible evolutionary strategies. The aim is to facilitate maximum code reuse, adding, extending, and adapting existing features. Despite its promise, EO is mostly intended for advanced users as mentioned by their authors, and both PISA and ParadisEO-MOEO do not provide a graphical user interface. A further example, HeuristicLab (HL) [15] is a framework for heuristic and evolutionary algorithms. HL has a modular architecture built around a plugin-based system, which allows for extensions to be added to the tool; its design philosophy is sound in terms of software engineering, and it provides a comprehensive graphical user interface. However, HL only provides a classical MOEA, namely NSGA-II, which is to some extent, not suitable for dealing with multi-objective problems with four or more objectives [16]. This is because the primary selection mechanism used by NSGA-II relies on Pareto-dominance which becomes less effective at selecting solutions as the number of objectives increases. Another problem with HL is the fact that it is implemented in C# which is dependent on software patents owned by Microsoft. Although there are free implementations of C# (e.g. Portable.NET) that allow the software to be ported to different platforms, the issue relates to the fact that Microsoft could decide to enforce its software patents and force all free implementations of C# underground.[3]

In terms of programming language, a number of libraries have been implemented in Java, and examples are: ECJ [17], Opt4J [18], EvA2 [19], JCLEC-MOEA [20], MOEA Framework [21], and jMetal [22,23]. One of the main advantages of Java is its portability to different operating systems. However, given their reliance on a Java virtual machine they can be deemed less suitable for intensive computing tasks in general, due to lower runtime performance when compared with other libraries that can run directly on the computer's processor (e.g. C and C++). From the mentioned libraries, ECJ, JCLEC-MOEA, and jMetal do not provide a graphical user interface. Moreover, ECJ capabilities in evolutionary multi-objective optimization are somehow limited since it only offers two classical Pareto-based MOEAs, namely NSGA-II and SPEA2. More recently, a library known as PlatEMO [24] has been developed for the Matlab platform. This however makes it dependent on a user having to pay a licence to use it, since Matlab is not free software and there is currently no evidence, to the best of authors' knowledge, that this library could be used with free programming languages that offer some degree of compatibility with Matlab (e.g. GNU Octave).

[3] The development of free open source software with C# is currently discouraged by the *free software foundation* due to the potential threat by Microsoft to enforce its software patents as argued in https://www.fsf.org/news/dont-depend-on-mono.

3. Liger: the concept and architecture

Liger is an optimization software that makes use of open-source libraries, runs on different operating systems, and provides an easy to use environment with interactive decision-making capabilities. It includes a library of algorithms, operators and optimization data management utilities that are easily verifiable due to its open-source nature and allow for rapid algorithm prototyping. The software is mostly implemented in C++ and follows an objective-oriented paradigm with an emphasis on design principles that make the software design more understandable, flexible and maintainable. In summary the features of Liger are:

1. Diversified list of MOEAs that covers different paradigms in evolutionary optimization: a non-elitist and elitist Pareto-based MOEA, respectively, MOGA [4] and NSGA-II [25]; a decomposition-based MOEA, namely MOEA/D [26]; an indicator-based MOEA, namely SMS-EMOA [27]; and a surrogate based MOEA, namely ParEGO [28]. Two variants of the previous MOEAs are NSGA-II-PSA [29] a variant of NSGA-II, and sParEGO [30] a variant of ParEGO.
2. Rapid algorithm prototyping with an easy to use interface where an algorithm is simply a composite of different operators and new algorithms can be created by introducing new operators and/or recombining existing ones;
3. Performance assessment metrics for conducting a comparison between the performance of MOEAs (e.g. Hypervolume indicator [31]);
4. Interface for loading optimization problems from a number of programming languages, including support for using more than one programming language at a time in the implementation of the optimization model. Available programming languages are C++, Matlab and Python;
5. Support for the specification of different problem types: problems whose variables are a mix of different types (e.g. continuous and discrete), and problems with uncertainty.
6. Support for parallel evaluations. It is possible to take advantage of multi-core systems to speed up the optimization process.
7. Provision for advance visualization and data exploration tools that are suitable for dealing with high-dimensional datasets. In this context, the data corresponds to the solutions of optimization problems;
8. Support for real-time interactive decision-making, and progressive articulation of user preferences;
9. Plugin architecture for easy extensibility. The plugins provide a flexible and modular approach to control the functionality of Liger;
10. Suite of tests that demonstrate the code operation. For this, we employ a *continuous integration server*[4] that runs the tests following each request to change the code in our GitHub repository.

One of the key aspects of Liger is the optimization workflow. A workflow contains a sequence of operators that are to be executed in a sequence. Each operator is responsible for a particular task that is related to the optimization and decision-making process. This includes loading the optimization problem, initializing a population of solutions, running an optimizer, and displaying the obtained solutions in a plot. Due to its modular and flexible nature, a workflow does not need to be constructed with the sole aim of conducting an optimization run, since it can also

[4] *Travis CI* (https://travis-ci.org/) is our continuous integration server that offers a good integration with GitHub.

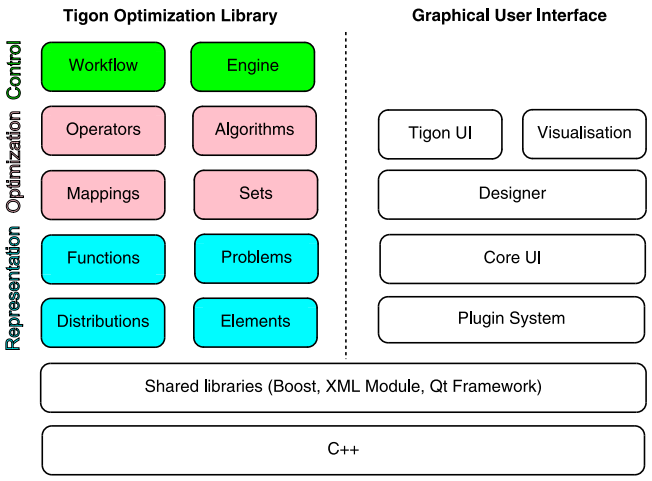


Fig. 1. Block diagram of the high-level architecture in Liger.

be used for other purposes, such as exploring the data generated by a previous optimization run, meaning that the workflow only needs to load the existing solutions from a file into the Liger environment.

In terms of architecture, the two main components of Liger are the Tigon optimization library and the GUI. These two are shown in the block diagram in Fig. 1, where each small block corresponds to a more specialized component, responsible for a particular functionality of the software. This diagram highlights some of the hierarchical dependencies between different components, in that each component depends on the component below and to the left of itself within each row. For example, *Distributions* is the most fundamental component in Tigon, and all other components depend on it; *Engine* is the most high-level component and depends on all other components. There are no dependencies between the components of Tigon and the GUI. This means that the Tigon library is not an integral part of the GUI, and can be replaced by a different optimization library if it is desirable. Also, the separation between Tigon and the GUI is made so that it is possible to create different front-ends for Tigon. Currently the Tigon front-end is based on the Qt toolkit, but if desirable, other GUIs could be developed using different toolkits for creating graphical user interfaces (e.g. GTK+[5]).

The architecture of the GUI is based on a subset of QtCreator, the integrated development environment created by The Qt Company.[6] The main reason for this choice has been expediency, as a large number of low-level utilities are already in place in QtCreator and have been extensively tested by the community over several years. For this reason, Liger shares several features with the architecture of QtCreator, most importantly, the plugin loader feature. In that, QtCreator consists of a plugin loader that loads and runs a set of plugins, and the incorporation of this concept into Liger allows for the existing functionality to be extended in a flexible manner, including complete control during the run-time of Liger. Moreover, the main plugins that exist in Liger are:

1. **Core User Interface (UI)**. The main plugin, upon which all other extensions depend. The Core plugin provides the necessary interfaces for fundamental tasks, such as communication, file input–output operations, handling of settings as well as classes implementing the main frame of the GUI. The Core UI is customized from QtCreator's CoreUI.
2. **Designer**. The plugin that defines a visual workflow composer, built-upon the Core UI. Note that the designer plugin only shares its name with QtCreator's designer.
3. **Tigon UI**. A plugin that implements the UI components in Designer and integrates the Tigon library with the user interface.
4. **Visualization**. The visualization plugin gives access to graphical plots for representing datasets generated by an optimization run for a given optimization problem. These plots facilitate data exploration, and provide a visual interface that supports preference elicitation to drive the search towards the most preferred solution by the DM. All plots are rendered by the Qt WebEngine web browsing module and their implementation is taken from the D3 library [32].

The interaction between the GUI and the Tigon optimization library is depicted in Fig. 2. The user interacts with the GUI to construct a workflow and once this is done he/she is in control of when to initiate the optimization process. During a typical start of the optimization process, the GUI communicates with the optimization library by sending the workflow via the Tigon UI plugin. After receiving the workflow, the optimization library invokes its own Engine to process the workflow operators, and following each iteration of the optimization algorithm, it communicates back to the Tigon UI, any generated results. This enables the Visualization plugin to update the plots in real-time during the optimization process.

4. Tigon optimization library

The design of Tigon is component-based, with a focus on flexibility and re-usability. It provides the user with a set of base classes, which can be used to interface complex problem structures, implement new operators, design/compose new algorithms, and to realize a complex optimization workflow. Tigon library is implemented in C++ and the code follows an object-oriented paradigm. Some of the dependencies of the Tigon library have already been shown in Fig. 1, this includes shared libraries such as the Boost library and the XML Module, and as well the standard C++ library. However, although the core functionality of the Tigon library is not dependent on the Qt Framework, the Tigon test suite currently is. We are working towards making Tigon less dependent on external libraries.

One of the key aspects of Tigon is the usage of the *Decorator* design pattern [33] to form a workflow. The decorator design pattern is a structural pattern that enables adding new functionality (to *decorate*) an existing object dynamically at run-time. This design principle results in a new design paradigm for evolutionary algorithms, which views operators as *decorators*. The use of this pattern also provides the flexibility to easily compose and alter a workflow. A useful property of the decorator design pattern is that it enforces the Single Responsibility Principle [34]. This principle states that a class should only have a single responsibility, to prevent having more than one reason for the class to change, in case the requirements change. To facilitate this process, each individual task conducted during the optimization process is the sole responsibility of a single operator.

More details about the Tigon library are provided in the following sections. This includes: the representation and evaluation of a solution for a given optimization problem in Section 4.1; the implementation of the decorator design pattern that forms the optimization architecture is described in Section 4.2; how to implement an optimization workflow is shown in Section 4.3; and the mechanism used to handle the information flow during the optimization process is described in Section 4.4.

[5] GTK+ https://www.gtk.org/.

[6] The Qt Company https://www.qt.io/.

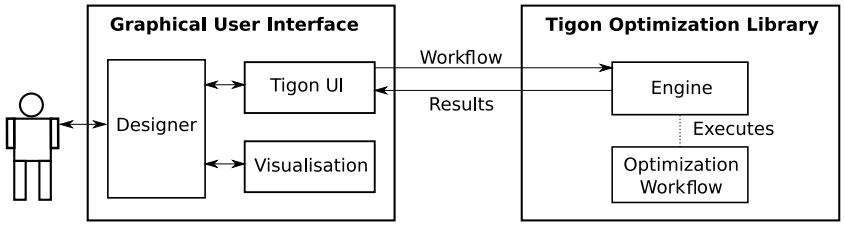


Fig. 2. Interactions between the GUI and the Tigon optimization library.

4.1. Representation and evaluation of a solution

The components of Tigon mentioned in this section are shown in a class diagram as depicted in Fig. 3. This type of diagram is used because Tigon implementation adopts an object oriented paradigm. Each class in the diagram is divided into three parts: the name on top, the attributes in the middle (their names often start with the *m_* prefix), and the operations (or methods) on the bottom. The lines connecting the classes represent two types of relations: association and realization. Association is depicted by a line without an arrow at the end, and is used to indicate if a class holds an instance (or object) of the other. For instance, the class ISet can have zero or more instances of the class IMapping, and in this case, these instances are stored inside the vector *m_mappings*. Realization is represented by a line with an arrow and it is used to indicate that one of the two related classes (the subclass) realizes (or implements) the other class (the superclass). For instance, the class IDistribution (a superclass) is meant to provide a generic interface that is common for all types of probability distributions, and all classes that provide an implementation for this interface are consider to be subclasses, such as NormalDistribution. In fact it can be said that NormalDistribution is a more specialized form of the class IDistribution.

The remainder of this section is organized as follows: handling different data types in Tigon is discussed in Section 4.1.1, the representation of a solution, problem and population are described in Section 4.1.2, and the way uncertainty is represented within Tigon is described in Section 4.1.3.

4.1.1. A unified interface for handling data types

A unified interface for handling data types is provided by the IElement class. IElements can be used as opposed to more simple types of variables, such as ints and doubles. The main characteristics are:

1. Support for cross-type operations, meaning that it handles arithmetic operations and relational operations between different types of variables;
2. Support for the specification of more specialized types, including automatic conversion from basic types;
3. The ability to define a variable as uncertain.

The data types currently supported are continuous and two discrete types, namely integer and nominal. The continuous type can take any value within an interval, the integer type is restricted to integer values, and the nominal type takes values from a set of categories without ordering relations being defined. The cross-type operations between IElements of different types, and also between an IElement and basic types such as int and double, includes: arithmetic operations such as addition, subtraction, multiplication and division, and also relational operations such as greater than and less than. The two main outcomes from this are: (i) that the operators are able to decouple their functionality, meaning that a separate routine can be created for handling each data type; and (ii) it is possible to specify optimization problems where the variables are a mix of different types.

Besides being defined by one of the three types above, an IElement can also be categorized as uncertain as opposed to deterministic. An uncertain IElement can be thought of as a random variable which may take a range of numerical outcomes, and the probability of sampling each outcome is defined by the underlying probability distribution. This is useful for the definition of optimization problems that have uncertainties. Section 4.1.3 provides more details about uncertainty specification in Tigon.

4.1.2. The solution, problem and population

This section describes the representation used in Tigon for a solution, the corresponding optimization problem, and also a population of solutions. These are represented in the class diagram in Fig. 3 and correspond respectively to the classes IMapping, Problem, and ISet. Since these classes are also dependent on others that are also depicted in Fig. 3, we start by describing the class IFunction.

The class IFunction represents any function (or a map) between a given number of inputs and a set of outputs. The implementation of the optimization model is conducted by a class that derives from IFunction, and due to its generic nature it can also be used for other purposes, such as to construct the surrogate of an hypothetical function based on a dataset (e.g. KrigingSurrogate), and there are even extensions to this class that enable the usage of other programming languages other than C++ in the function implementation (e.g. PythonFunction and MatlabFunction).

An example of a class that derives from IFunction is the class DTLZ1 that corresponds to a benchmark problem taken from the literature with the same name [35]. The implementation for this problem is conducted in C++, but if desirable, this could be done instead in Python or Matlab, via PythonFunction and MatlabFunction, respectively. This feature is useful for those users that are not familiar with C++ and wish to use a different programming language to implement their optimization models. Besides, it is then possible to access libraries and other modules that are not necessarily available in C++ (e.g. Simulink in Matlab, or scikit-learn in Python), which might be a requirement of the optimization model. *evaluate()* is a method from IFunction that is used to evaluate a function, meaning that a set of inputs are passed to the function for evaluation and the outputs are then generated. This method is able to evaluate only one set of inputs for each function call, but it is possible to evaluate multiple sets of inputs in one go by using the *batchEvaluate()* method. This is useful for functions that rely on external applications, such as Matlab, since the number of times that it is required to open and close a connection to the software can be severely reduced, besides it is also possible to speed up the function evaluation by making use of *vectorization*.[7] Another important feature of IFunction is its support for parallel evaluations. This takes advantage of multi-core systems by allowing solutions to be evaluated in parallel. Any function can be defined as parallelisable and the current implementation of this feature makes use of

[7] Vectorization is a programming technique that uses vector operations instead of element-by-element loop-based operations.

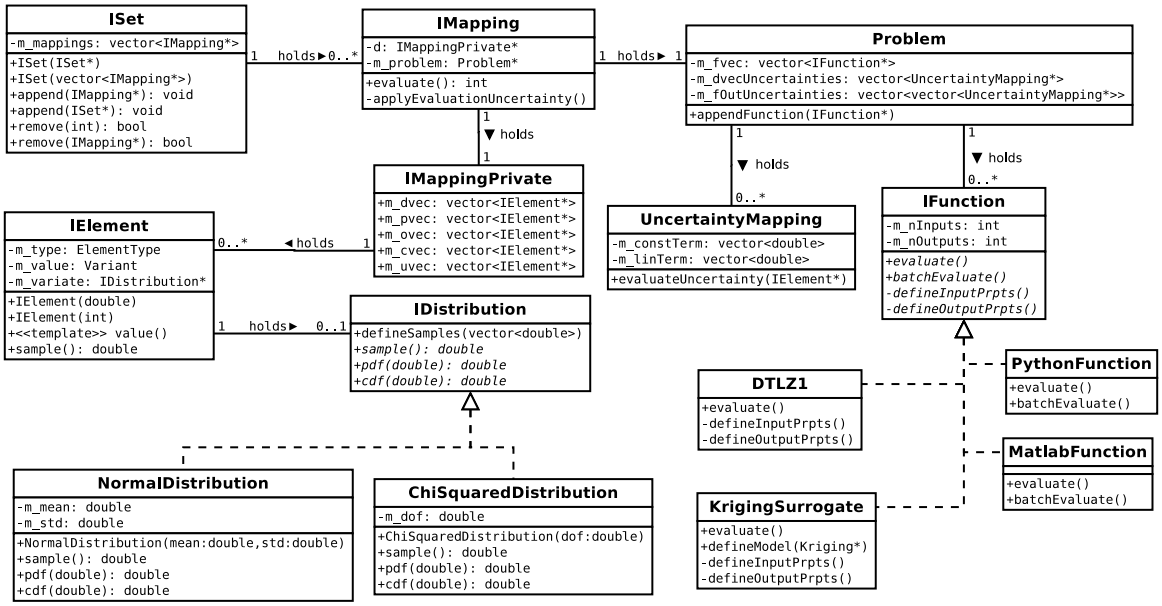


Fig. 3. The relationships between the classes in Tigon that represent the optimization problem (Problem), the corresponding solution (IMapping), and a population (ISet).

multi-threading capabilities of the C++ standard library. The term function will be used throughout this section to refer to any class, or function implementation, that derives from the IFunction class.

The Problem class is used to define all aspects of an optimization problem. This includes, and it is not limited to: bounds for decision variables, variable types, thresholds for constraints, and uncertainties in the problem model (more details about uncertainty are provided in Section 4.1.3). It is possible to use more than one function simultaneously in the problem model implementation, albeit that each function would contribute to the problem model with a separate set of objectives (and/or constraints). Although it is common practice to use only one function to implement the problem model[8] since a single function supports the specification of multiple inputs and outputs, there are situations where using multiple functions might be desirable:

1. The implementation can be divided into several files or modules, each implementing a different component of the optimization model. This is useful for the design of models that are too complex, such as multi-disciplinary optimization problems where it is a good practice to have a separate function for each discipline or engineering branch, which allows for different teams to work on them separately.
2. To take advantage of more than one programming language in the same optimization model. It is not possible to mix programming languages in the same function, therefore if the intention is to use libraries from, say C++ and Python programming languages together, then one function needs to be implemented in C++ and the other in Python.

The IMapping class represents a solution for an optimization problem. The name refers to the relationship that maps the model inputs (decision variables and parameters) to the outputs (constraints, objectives and unused outputs). The values of all inputs and outputs are stored in the instance of the IMappingPrivate class (namely *d*), and these are all of type IElement. Any instance of IMapping also keeps a reference to the optimization problem (*m_problem*), which is used to initialize many of the solution attributes (e.g. the number of inputs and their types), and allows access to the evaluation function of the optimization problem. Other attributes that form part of the IMapping class includes information about the solution feasibility, its evaluation status and whether or not it satisfies any user-defined preferences (e.g. goals in objective space), a weight vector that can be used by decomposition based optimizers, and also a fitness score which is useful for selection purposes. The method *evaluate()* in IMapping is used to call the evaluation function of the optimization problem. The procedure provides the inputs to the evaluation function and updates the outputs. Both the inputs and outputs can be either deterministic or uncertain. The deterministic values are stored in the attribute *m_value* of IElement, and can be obtained by calling the *value()* method. The uncertain values are obtained by calling the *sample()* method (more details about this are provided in Section 4.1.3).

The ISet class is a container of IMappings and therefore it can be used to represent a population of solutions. The IMappings are stored inside an ISet as pointers, and as a result, the same IMappings can co-exist in different sets at the same time. During the optimization process an evolutionary algorithm often evolves a population of solutions, and we refer to this population as the main optimization set. However, it is also possible for an optimizer to create more than one population during the optimization process. This is useful in particular for organizing the solutions (e.g. different dominance ranks) or even to define different type of structures (e.g. to define neighbourhoods depending on the distance between solutions). To facilitate the identification of a particular ISet (e.g. an ISet that contains only evaluated solutions) Tigon makes use of *tags*. The tags are nothing but simple unique strings that can be added to an ISet and they are essential for the information flow coordination during an optimization run. A more detailed explanation on the utility and usage of the tags in Tigon will be provided in Section 4.4.

4.1.3. Uncertainty

Tigon supports the specification of various sources of uncertainty, such as production variations, changing environmental conditions and model–method combination errors. The uncertainties in the inputs of the model (either tolerances in the design or changes in the environmental parameters) are described by

[8] The problem model of all benchmark test problems currently in the Tigon library are implemented using a single function.

a probability distribution. Let **X** and **P** be random variables that follow a certain distribution for decision variables and parameters, respectively. When a candidate solution with its inherent uncertainties is evaluated, all uncertainty factors for both **X** and **P** are sampled from their distributions, and the sampled values are processed by the evaluation function. The outcome of such evaluation, for each realization of the uncertainty, is a single performance (or constraint) vector **z**. The left side of Fig. 4 illustrates this procedure.

When models are used to simulate a physical phenomena, accuracy errors are likely to occur. This is handled by adding another layer of uncertainty as illustrated in the right side of Fig. 4. This is accommodated by requesting a domain expert to elicit the uncertainty of a simulation outcome **z** as a random variable **Z**. For every model evaluation, instead of using the raw evaluation **z**, a sample $\mathbf{z}'$ from the distribution **Z** is used.

As said before, an important feature of IElement is its ability to generate a sample according to a distribution. For this, the attribute *m_variate* of type IDistribution is able to store a probability distribution. IDistribution itself is an interface for all univariate distributions, either discrete or continuous. Examples of probability distributions that implement this interface are the normal distribution, uniform distribution, and Chi-squared distribution. Once the distribution has been defined, a sample can be generated by the *sample()* method.

Given that the inputs and outputs of the model are of type IElement, it is possible to define a probability distribution for each one of them, and treat them as random variables. The values that are sampled from the distributions are used during the evaluation process of a candidate solution as described in the first two paragraphs of this section. This process is straightforward for inputs of the model that are defined as parameters (e.g. environmental parameters), since all the samples can be generated directly from the probability distribution defined in IElement, that is, it is only required to invoke the *sample()* method each time to generate a new value. For design variables and model outputs we also need to take into account their nominal values, given that it is general practice to define the uncertainty as a function of the current design choices and their output values. This is achieved by letting the statistical parameters of the probability distribution be a function of the nominal values. This process is implemented in the UncertaintyMapping class where the *evaluateUncertainty(IElement*)* method modifies the statistical parameters of the IElement provided as input. The class Problem stores a reference to the UncertaintyMappings in the attributes *m_dvecUncertainties* (for decision variables) and *m_fOutUncertainties* (for model outputs).

4.2. Optimization architecture

We now describe the Tigon components that are involved in coordinating the optimization process, that jointly form the optimization architecture. These components are depicted in the class diagram shown in Fig. 5. Fundamentally, IPSetPrivate and IPSet are the two base classes (or interfaces) from which all other classes are derived. IPSetPrivate keeps track of multiple populations of solutions that are used during the optimization process, and also keeps a reference to the current optimization problem. IPSet contains an instance of IPSetPrivate and it provides an interface for processing (or evaluating) the optimization operators. More details about these components, and others that they depend on, are provided in the following subsections.

4.2.1. IOperator

Operators in Tigon are what constitute an algorithm and the optimization workflow. All operators operate on sets of Mappings (or solutions), and their operation may involve creating, reading, modifying, or even deleting any existing set, and their solutions. (For more information concerning the way Tigon coordinates the way operators process the sets, please refer to Section 4.4). Moreover, operators often expose properties that the user can modify based on the functionality of each operator (e.g. probability of solution crossover in SBX Crossover). There are five types of operators currently implemented in Tigon:

1. *Initialization*. All initialization operators derive from the IInitialisation interface. An initialization operator is essentially used to generate the initial population for an algorithm. For a population-based algorithm, it can represent a Design of Experiment (DoE) operation, Latin Hypercube Sampling (LHSInit), Hypercube Grid Sampling (HypergridInit), Uniform Sampling (RandomInit), for example.
2. *Evaluator*. Evaluators are designed to perform function evaluations. Essentially they manage when and how to call the *evaluate()* method in the Mappings and perform a sanity check of the outputs.
3. *Genetic*. These are operators used by evolutionary algorithms, such as mutation, filtration, non-dominated ranking, elite selection, etc.
4. *Formulation*. This operator manages the optimization problem formulation. It loads either a predefined optimization problem from the Tigon library or processes user-defined problems. The *ProblemGenerator* is a concrete implementation of this class.
5. *Convergence*. Performance metrics to evaluate the quality of a solution set. The scope is not limited to metrics that measure the convergence to the POF, and also includes those metrics that measure the distribution across the POF. One existing performance metric that can be used to estimate the convergence and distribution of the solution set is the hypervolume indicator [31].

New operator types can be easily added and incorporated in the current collection in a straightforward manner, since they would inherit from the IOperator class.

4.2.2. IAlgorithm

An algorithm contains a sequence of operators. Although initialization, formulation and evaluation operators can be part of an algorithm, they are usually defined outside the algorithm. This is done in order to simplify the workflow that is exposed to the user, since the initialization, formulation, and evaluation operators are always required to exist in any optimization algorithm workflow. So, in general, the algorithms implemented in Tigon contain genetic operators. That being said, it is also possible to add, for example, another evaluation operator into the workflow of an algorithm and, in fact, more than one if the need arises. All the algorithm classes derive from the IAlgorithm interface. Similar to an operator, the algorithm also exposes a set of properties that the user is able to modify. The user can simply add operators by appending them in the order of operation by using the *appendOperator()* method.

4.2.3. IPSet

The IPSet (Interface for Population Set) is a fundamental class in the architecture of Tigon. It is an abstraction for the data structure of an optimization process. All the information related to an optimization process, such as solution sets,[9] problem definition,

[9] The solutions generated during the optimization process are stored in the instance of IPSetPrivate, namely *d*.

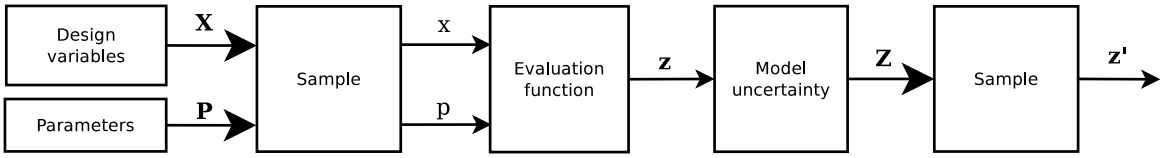


Fig. 4. Types of uncertainty and its propagation during the evaluation of a candidate solution. Random variables are marked with larger arrows.

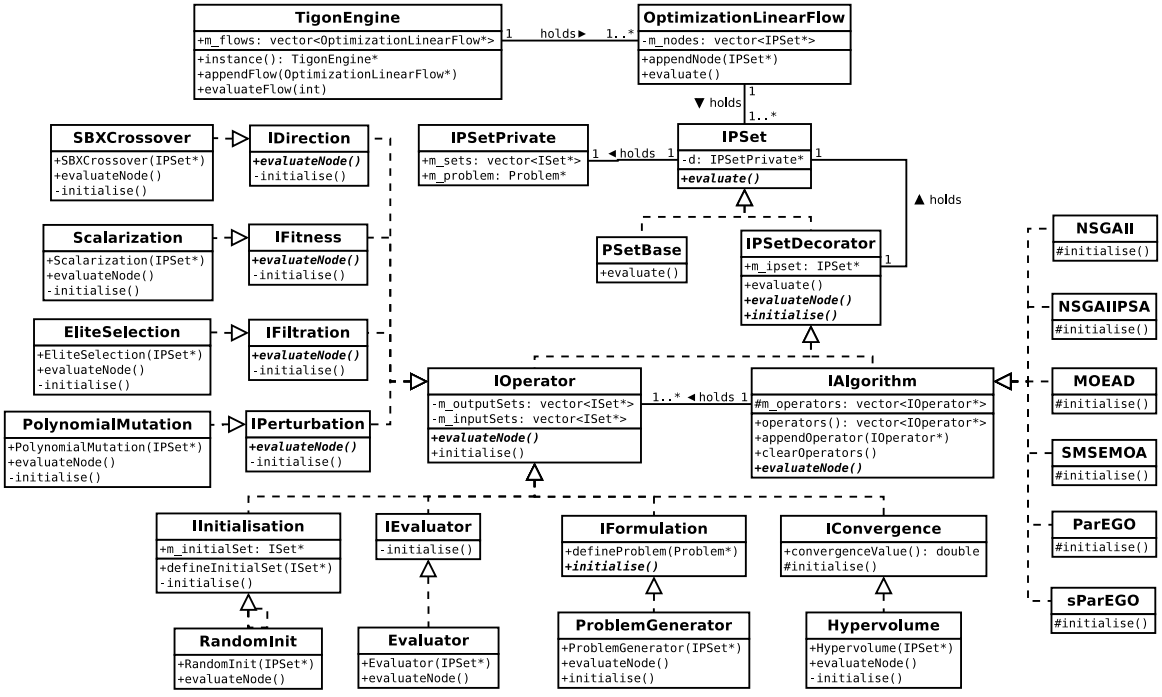


Fig. 5. The Tigon components that form the optimization architecture shown in a class diagram.

termination criteria etc., are encapsulated in the IPSet class. The derived classes from IPSet access and modify the data by using a protected interface. Certain aspects of this interface include the possibility for the user to specify the access policy of each operator to the existing sets. For instance, an operator might be only allowed to access sets that have a particular tag. More details about the mechanism that allows operators to access the sets is provided in Section 4.4.

IPSet contains the *evaluate()* method, which is implemented by the two derived classes, namely PSetBase and IPSetDecorator. This provides the control mechanism behind the decorator design pattern, and it is therefore responsible for coordinating how the operators access the data, and also in which sequence this happens. More details about this implementation will be provided in Section 4.2.5. The actual procedure conducted by the operators that are derived from IPSet is implemented in the method *evaluateNode()*. Note that this method is declared in the IPSetDecorator class as virtual, and each operator needs to provide its own implementation. This allows the operators to perform completely different actions on the sets.

4.2.4. PSetBase

The PSetBase class provides a concrete implementation of the evaluate method in IPSet. It does not perform any associated computation, that is, the *evaluate()* method is an empty function. It can be considered as the mark of the start of a workflow.

4.2.5. IPSetDecorator

As already mentioned, Tigon utilizes the *Decorator* design pattern to form a workflow, which can be used to setup the building blocks of an optimization algorithm. Alternatively, it is also possible to select a set of operators to conduct other tasks, such as load a population of solutions for visualization in a graphical front-end, generate a set of solutions based on some DoE approach, or to conduct post-optimization tasks (e.g. extract the non-dominated solutions from a population). A workflow in this context is composed of operators and each operator is used to decorate the instance (or object) that hold the workflow. The core implementation of this design pattern lies in the IPSetDecorator class, which is considered to be the base for all decorators. More details about the IPSetDecorator class are described as follows.

The IPSetDecorator class, like PSetBase, also provides an implementation to the *evaluate()* method from IPSet. This implementation makes use of recursive composition to produce a custom workflow based on a set of operators. The operators involved in a workflow are placed on a given order for execution, which is defined recursively, without the need to define this order by using a static approach.[10]

We now discuss some details about the implementation of the decorator design pattern and how this influences the construction of a workflow. To build a workflow, an order of execution for the operators needs to be defined. To define this order, we make use of the constructor of each operator, which can take an IPSet instance as an argument. Fig. 6 shows an excerpt from the code where a set of operators, and their order, is chained together by using their constructors.

[10] A static approach of composing a workflow in this context would require all possible combinations between the operators to be defined prior to compilation time, which can be impractical for a large number of operators.

```
PSetBase*          base        = new PSetBase();
ProblemGenerator* problem     = new ProblemGenerator(base);
RandomInit*        initialiser = new RandomInit(problem);
Evaluator*         evaluator   = new Evaluator(initialiser);

// Invoke evaluate() method
evaluator->evaluate();
```

Fig. 6. An illustration of operator evaluation calls.

The first operator is PSetBase and this indicates the base, and also the first execution point of our workflow. The following operators, that is, ProblemGenerator, RandomInit and Evaluator, execute a set of instructions in this particular order. First the optimization problem is loaded, then a set of solutions is initialized, and finally the objectives and constraint functions of all solutions are evaluated. To achieve this behaviour we make use of the IPSet instance (*m_ipset*) and the *evaluate()* method, both found in IPSetDecorator. What follows is a recursive composition to allow operators to be added to the IPSet instance and the implementation is shown in Fig. 7.

```
void IPSetDecorator::evaluate() {
    m_ipset->evaluate();
    evaluateNode();
}
```

Fig. 7. Implementation of the *evaluate()* method in IPSetDecorator.

In Fig. 6 the Evaluator operator invokes the *evaluate()* method. This creates a chain of events as illustrated in the sequence diagram shown in Fig. 8. Note that when the evaluate method of the evaluator is called, it backtracks all the way to the evaluate method in the base, which does nothing. After the evaluate method completes, control is passed to the ProblemGenerator operator, which invokes its own *evaluateNode()* method. Following this, both RandomInit and Evaluator also invoke their *evaluateNode()* methods in this particular order.

4.2.6. TigonEngine and OptimizationLinearFlow

The two top classes in Fig. 5 are TigonEngine and OptimizationLinearFlow. These two classes provide an interface to interact with the Tigon library, in a very simple way, since only the methods necessary to operate a workflow are exposed.

The OptimizationLinearFlow keeps track of a vector of operators that together form a workflow. The function *evaluate()* will automatically invoke the *evaluate()* function from the last operator added to the workflow, starting a chain of events, such as those described in Fig. 8. The TigonEngine class follows a Singleton design pattern [33], which defines a unique instance that holds the Tigon execution engine. This means that only one engine instance is allowed to exist during run-time. However, this does not prevent the user from adding multiple workflows to the same engine, by using the function *appendFlow(OptimizationLinearFlow*)*. Subsequently any workflow can be evaluated by calling the function *evaluateFlow(int)* where the input argument corresponds to the workflow index inside the engine.

4.3. Optimization workflow

An optimization procedure, including the problem definition and algorithm in Tigon, is described as a workflow. An optimization workflow is represented as a chain of operators. A complete workflow must contain a base set (PSetBase), a problem formulation operator, an initialization operator, an algorithm, and an evaluator. The base set marks the start of the workflow. The excerpt from the code in Fig. 9 shows a minimal workflow and its execution.

When a user runs an optimization algorithm, the operators are processed in a sequence and the existing sets are modified in a process that resembles an optimization task, where a population of solutions is evolved iteratively. Any solution during the optimization process is either created or modified by an operator. For instance, the set that contains the randomly initialized solutions is created by the RandomInit operator. It is then possible for other operators that search for this particular set to create sets based on the initial population and/or to directly modify the solutions of the set.

Knowing that multiple sets co-exist during the optimization process at the same time, it is important for the operators to be able to identify which sets they are supposed to operate on. To support the operators in identifying the required sets, a mechanism based on the usage of tags is implemented in Tigon. More details about this mechanism is provided in the following section.

4.4. Information flow

Each operator in Tigon contains tags, and these are used to help the operators to find the sets that they need to operate on. From the perspective of an operator the sets are categorized as either input or output sets. The input sets are read-only, meaning that they will not be modified, and the operators will only use them to access the information stored in their solutions, often to create or modify other sets. The output sets provide an indication to the operator that they can be modified, or in some cases they are even created directly by the operator itself. The way this is achieved is by assigning input and output tags to the operators. An operator will then search for those sets that contain the tags that it is looking for. The tags that are currently implemented in Tigon are used in a way that describes the operation with which the operator is associated and the operation that needs to be performed on a set. The tags are added to the operators before the start of the optimization process. When an operator is initialized it will be assigned with tags that are appropriate for its functionality. For example, the RandomInit operator creates a set that corresponds to the initial population of solutions, and tags this set with the tag *Main Optimization Set*. This is an output set for RandomInit because this set has been created by the operator. Other operators will be able to identify this set during the optimization process if they use the tag *Main Optimization Set*.

It is possible to add more than one tag to a set. For instance, the RandomInit operator, besides tagging the initial population

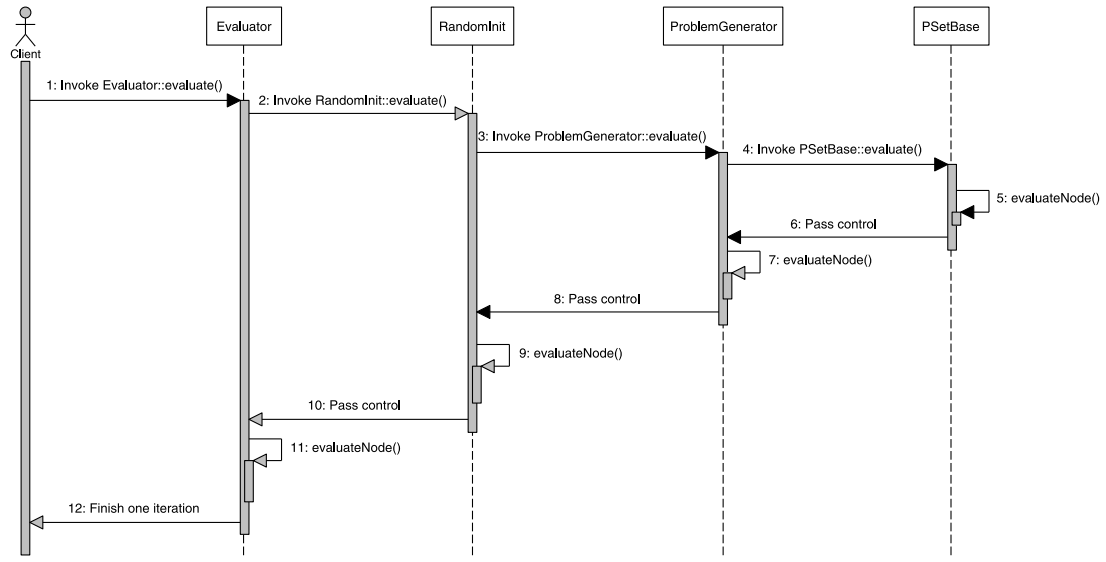


Fig. 8. An illustration of the chain of execution for example in Fig. 6.

```
1  // Define a workflow
2  PSetBase*      base    = new PSetBase();
3  ProblemGenerator* problem  = new ProblemGenerator(base);
4  RandomInit*    init    = new RandomInit(problem)
5  Evaluator*     evaluator = new Evaluator(init);
6  NSGAII*        algorithm = new NSGAII(evaluator);
7
8  // Run one iteration of the workflow
9  algorithm->evaluate();
```

Fig. 9. Source code for defining a minimal workflow in Tigon.

set with the *Main Optimization Set* tag, also adds the tag *For Evaluation*. This provides an indication that the solutions in this set have not been evaluated yet, and other operators, for instance those that derive from IEvaluation, are now able to identify this set in case it exists. Moreover, when searching for a set with a given tag, the operators make a distinction between an input tag and an output tag, that is:

1. If the tags in a set correspond to all input tags of an operator, the set is referred to as an input set, and the operator reads the solutions of the set but never modifies them.
2. Conversely, if the tags in a set correspond to all output tags of an operator, the set is referred to as an output set, and the operator can read and modify the solutions in the set.

In general, the operators have at least one output tag, and in some cases there are no input tags. For instance, the RandomInit operator only contains output tags and no input tags, since it does not need to read information from other sets in order to create the initial population. The following is a list of commonly used tags in Tigon operators and algorithms:

1. *Main Optimization Set*: The population-based optimization algorithms often have one population of solutions that is processed in every iteration (or generation); the set that contains such solutions is identified with this tag.
2. *For Evaluation*: This tag indicates that the solutions in the set need to be evaluated. This means that it is required to evaluate the objective functions (also constraints, if any). The operator Evaluator contains *For Evaluation* as an output tag.
3. *Fitness*: Some operators attribute fitness to the solutions, often in the form of a score, by using a performance criterion. This tag indicates that fitness has been added to the solutions in the set. One operator that attributes fitness to the solutions is *NonDominance Ranking*, and uses *Fitness* as an output tag.
4. *For Selection*: A set of solutions is tagged for selection to indicate that those solutions can undergo selection based on some criterion. This often means that the solutions in the set already have fitness, and it is now possible to select the best solutions. One example of a selection operator is the *NSGA-II Elite Selection*, which contains the output tag *For Selection*.

Consider the three operators and the respective tags shown in Fig. 10. The figure also shows the corresponding code that is required for implementing the desirable information flow between the operators in Tigon.[11] The operator at the top is always the first one to operate, which in this case corresponds to Random Initialization. This operator creates a population of solutions and the set that contains this population is initially tagged with *Main*

[11] Please note that many of the tags mentioned in Fig. 10 exist by default inside the operators and are only shown for illustration purposes. Hence, in most cases it is not required to add them while composing a workflow.

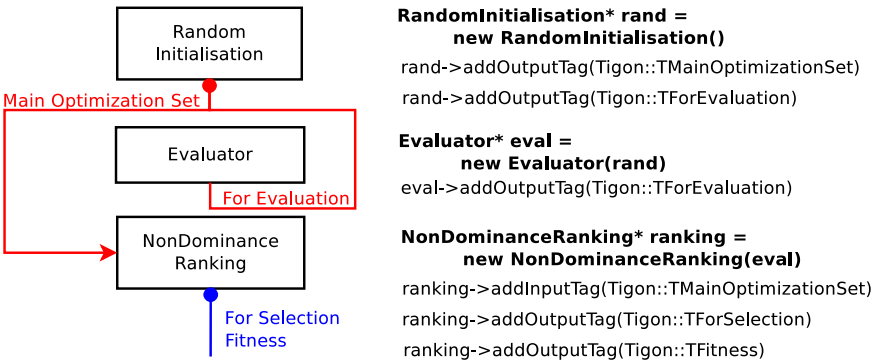


Fig. 10. An example showing three operators with their tags on the left, and the corresponding excerpt from the code in Tigon is on the right.

Optimization Set and *For Evaluation* tags. The circle shown below the Random Initialization and NonDominance Ranking operators in Fig. 10 indicates that new sets are created by the operators. That is:

1. The Random Initialization operator creates the main optimization set and attributes the tags *Main Optimization Set* and *For Evaluation*.
2. The Non-Dominance Ranking operator creates multiple sets, and each set is tagged with *Fitness* and *For Selection*.

Note that the Evaluator operator does not create any sets, since it only operates on existing solutions. Once the solutions in the main optimization set are evaluated, it is now possible to add fitness to the solutions by using some fitness operator, such as Non-Dominance Ranking. This operator reads from the main optimization set and divides the solutions into sets based on their dominance rank. Moreover, the tags added by the NonDominance Ranking operator lead to the following:

1. Fitness: the solutions have gained fitness by using the non-dominance principle as the fitness criterion.
2. For Selection: this tag indicates that selection operations can be applied to the solutions in the set. Knowing that fitness is now added to the solutions, it is now possible for other operators to conduct selection operations. In this case, the solution can be selected based on its rank in terms of the non-dominance principle.

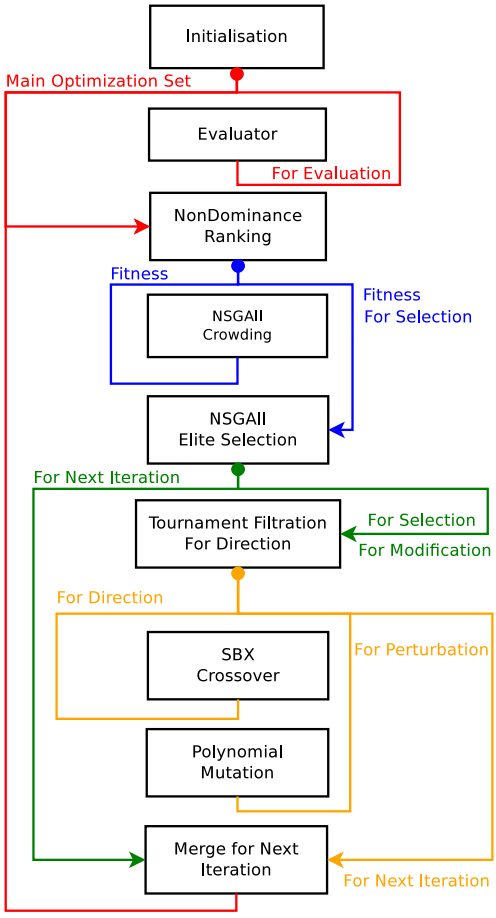
The sets created by the NonDominance Ranking operator contain the mappings that were originally created by the Random Initialization operator. These mappings that form the main optimization set have been simply rearranged into different sets according to their dominance rank. However, the sets only store pointers to the mappings, and not the mappings itself. This means that the mappings in the sets created by the NonDominance Ranking operator also exist in the main optimization set. Hence, any modification to any mapping in the main optimization set is also reflected on the sets created by the NonDominance Ranking operator, and vice-versa.

5. Composing an optimization algorithm

Tigon is designed such that it is straightforward to compose new algorithms from existing operators. In this section, we take NSGA-II [25] as an example to illustrate how to compose a new algorithm by using the Tigon operators together with the concept of tags. NSGA-II is a popular elitist MOEA that relies on Pareto-dominance for selection and crowding distance for maintaining a diverse set of solutions in the population. The steps of NSGA-II are as follows:

1. *Initialization*: The first step of NSGA-II is to initialize a population of solutions. This is often done by using a totally random approach, but any other more sophisticated DoE approach could be used instead (e.g. Latin hypercube sampling).
2. *Evaluation*: The initial population is then evaluated, which consists of evaluating all the objective functions, and constraints if any.
3. *Fitness*: It is now possible to apply some criterion to rank the solutions. The criterion adopted by NSGA-II is to rely on Pareto-dominance to induce an order between the solutions. The solutions are then organized into ranks, where rank 1 solutions are considered to be better than rank 2, etc. The ranks are also sorted based on crowding distance where the least crowded solutions appear first, and the more crowded ones appear later.
4. *Selection*: The next step is to select the elite population, and the criterion is to select the solutions starting from rank 1, and then 2 and so on, until at least half of the population is selected. The other half of the population is discarded.
5. *Filtration with Tournament*: The aim of this step is to select solutions from the elite population. The first iteration of the selection process is as follows. First, four unique solutions are chosen randomly from the population. Then, the first two and the last two solutions are paired together, and sent for tournament.[12] Finally, the selected solutions are those that win the tournament. This process is repeated until the number of selected solutions is equal to the elite population size.
6. *Recombination and Perturbation*: The selected solutions from the previous step are first paired together. Each pair is now recombined by using the SBX crossover, and in each case, two new solutions are generated. A perturbation operation is then applied to each new solution, often by polynomial mutation. Note that it is not expected for all solutions to undergo perturbation since this depends on some user-defined probability.
7. *Merge the population*: the new solutions are now combined with the elite population to generate the final population. Following this, one iteration (or generation) takes place. This process repeats itself by going back to step 2, and stops when some stopping criterion is satisfied (e.g. until a user-defined maximum number of iterations is exceeded).

[12] Consider two solutions, say solution a and b, and the tournament selection is as follows. Solution a wins the tournament if a dominates b or the crowding distance of a is higher than the crowding distance of b (meaning a is less crowded than b), otherwise b wins the tournament. In case the solutions are non-dominated (meaning that neither a dominates b nor b dominates a) and they both have the same crowding distance, then the solution that wins the tournament is picked randomly.




```
NonDominanceRanking* ranking =
        new NonDominanceRanking()
ranking->addInputTag(Tigon::TMainOptimizationSet)
ranking->addOutputTag(Tigon::TForSelection)
ranking->addOutputTag(Tigon::TFitness)

NSGAIICrowding* crowding =
        new NSGAIICrowding(ranking)
crowding->addOutputTag(Tigon::TFitness)

NSGAIIEliteSelection* elite =
        new NSGAIIEliteSelection(crowding)
elite->addInputTag(Tigon::TFitness)
elite->addIntputTag(Tigon::TForSelection)
elite->addOutputTag(Tigon::TForSelection)
elite->addOutputTag(Tigon::TForModification)

TournamentFiltrationForDirection* childs =
        new TournamentFiltrationForDirection(elite)
childs->addInputTag(Tigon::TForSelection)
childs->addInputTag(Tigon::TForModification)
childs->addOutputTag(Tigon::TForDirection)
childs->addOutputTag(Tigon::TForPerturbation)
childs->addOutputTag(Tigon::TForNextIteration)

SBXCrossOver* crossOver =
        new SBXCrossOver(childs)
crossOver->addOutputTag(Tigon::TForDirection)

PolynomialMutation* pm =
        new PolynomialMutation(crossOver)
pm->addOutputTag(Tigon::TForPerturbation)

MergeForNextIteration* nxt =
        new MergeForNextIteration(pm)
nxt->addInputTag(Tigon::TForNextIteration)
nxt->addOutputTag(Tigon::TMainOptimizationSet)
```

Fig. 11. NSGA-II operators and tags diagram (on the left) and corresponding code in Tigon (on the right).

The operators and tags used to compose NSGA-II are shown in Fig. 11. The figure also shows the corresponding code in Tigon to implement this example. The Initialization step is represented by the Initialization operator, which is responsible for initializing a population of solutions. This operator is just an interface and it will be replaced by other operator that implements it (e.g. RandomInit which initializes the solutions randomly). Notice that this operator is a set creator, and the new set is tagged with tags (in red) *Main Optimization Set* and *For Evaluation*. The Evaluation step is represented by the Evaluator operator, and the main optimization set is evaluated and treated as an output set. The Fitness step is represented by two operators, namely NonDominance Ranking and NSGA-II Crowding. The first is a set creator operator, and each new set corresponds to a different rank of solutions. Crowding distance then sorts the solutions inside each set. Since the solutions now have fitness, it is now possible to use selection operators, hence, the tags (in blue) *Fitness* and *For Selection*. The Selection step is conducted by NSGA-II Elite Selection operator, that reads from the sets created by the NonDominance Ranking operator. This operator is also a set creator since the elite population is created at this stage. The tags (in green) used in this case are as follows:

1. Knowing that the elite population will be selected into the next iteration, the set is therefore tagged with the tag *For Next Iteration*.
2. In order for TournamentFiltrationForDirection operator to pick only the elite population, and avoid the sets that have a lower rank, the tag *For Modification* is added to the elite set.
3. Given that the tag For Selection is an input tag of Tournament Filtration for Direction, we then need to add this tag to the elite set.

Moreover, the Filtration step with tournament is conducted by the operator Tournament Filtration for Direction. This operator creates multiple sets, where each set contains only two solutions taken from the elite population. The new sets are tagged (in orange) by the tags *For Direction*, *For Perturbation*, and *For Next Iteration*. The first ensures that the sets are found by the SBX CrossOver operator, the second does the same for Polynomial Mutation, and the third ensures that the new solutions are found by the Merge For Next Iteration operator. The recombination and perturbation step is then conducted by the SBX CrossOver and Polynomial Mutation operators, respectively. Finally, the new solutions are combined with the elite set by the Merge For Next Iteration operator, and the new final population replaces all solutions from the main optimization set. This process now completes one iteration, and this can be repeated again by using the new main optimization set in the next iteration. Note that the Initialization operator only operates during the first iteration, since it only generates new solutions in case these have not been created.

6. Graphical user interface

The GUI of Liger has been constructed by using the Qt toolkit and its architecture shares some features with the QTCreator software, most notably the plugins system. In this section we first describe the plugins that are responsible for the behaviour of the GUI in Section 6.1, how to create and run a simple optimization

workflow is described in Section 6.2, and the interactive decision-making capability of Liger are demonstrated on a real-world optimization problem in Section 6.3.

6.1. Plugins

The Liger plugin system is based on the plugin loader feature from the QtCreator software, which provides an interface for implementing functional modules to extend the capabilities of Liger. As previously mentioned in Section 3, some important plugins are: Core User Interface, Designer, Tigon User Interface, and Visualization. The Core User Interface is customized from QtCreator's CoreUI and provides interfaces for several fundamental tasks, including communication, file input–output operations, handling settings, and several others. The other plugins are described as follows.

6.1.1. Designer and Tigon user interface plugins

Designer defines a generic framework of a visual programming environment. Note that the Designer plugin is devised to be generic and is not coupled to any specific optimization library. Changes in the optimization library do not result in a change in the designer. In addition, this design decision enables the possibility of working with multiple optimization engines within a single structure. In software engineering practice, low coupling between libraries, or other software components, helps generate a well-structured software system and a good design, which in turn improves readability and maintainability.

The fundamental component of the Designer is the process node. Every process node extends the IProcessNode abstract class, which provides the necessary functionalities to allow the user to view or modify the properties of the underlying operators. A process node is comprised of zero or one *input port* and zero or one *output port*. Each process node holds a pointer to the underlying operator and has access to all of its properties. An optimization workflow consists of a list of process nodes and the associated links. Each workflow starts at the Master Start Node (MSN) and ends at the Master End Node (MEN). These two nodes are unique per design and provide a frame of reference for the start and end of an optimization loop. It is possible to add multiple process nodes between the MSN and MEN.

The IEngine within Designer is the part that coordinates the order of execution within an optimization workflow. Once the workflow is fully configured, the engine invokes *evaluate()* method to pass the workflow to the underlying optimizer and start the optimization process. Since the run time of the optimization process can take hours, or even days, to finish, to prevent the optimization from blocking the main application and for the sake of performance, *evaluate()* method will instantiate a new thread and execute the workflow on this thread. The communication between the optimization thread and the main application, i.e. the GUI, is accomplished via signals. When the optimization process starts, the engine will emit signals to inform the Designer so that no modification of the operators will be accepted. The optimizer performs its execution until either an error occurs or the termination criteria are satisfied. Upon termination, the engine emits signals to the Designer and sends data to the visualization plugin. Note that the IEngine in Designer defines fundamental methods to launch an optimizer. Any realization of the IEngine has to follow the interface defined in IEngine in order to be compatible with the rest of the GUI.

The Tigon User Interface plugin is a realization of the Designer plugin. It provides:

1. Process node implementations derived from IProcessNode, which exposes operators or algorithms from Tigon to the Designer. These process nodes enable the user to create or modify Tigon operators from the user interface.
2. Engine implementation derived from IEngine, which establishes the link between the GUI and the Tigon Engine. It passes the fully configured workflow to the engine and manages the evaluation of the workflow.

6.1.2. Visualization plugin

This section shares details about the visualization plugin which provides a set of graphical plots capable of visualizing solutions from an optimization problem. The graphical plots are accessible via process (visualization) nodes that can be drag and drop by the user into the Liger optimization workflow. One of the main difficulties in the development of this plugin has been the fact that a high-level C++ visualization library is not available under an open source licence that fits with the philosophy of Liger. This is why the visualization plugin of Liger uses a combination of the QtWebEngine and D3. D3 is a JavaScript based visualization library that creates *scalable vector graphics* and allows for user interaction. The performance of the library, so far, suits the requirements of Liger, although cannot be compared to a native C++ library that would make use of specialized hardware (graphics cards).

A class diagram in Fig. 12 provides an illustration of the visualization plugin architecture. There are currently three types of graphical plots available in Liger and they are implemented simply as process nodes. They are all derived from the QVizNode class, which, in turn, is derived from IProcessNode. This means that the plots are part of the workflow and they operate as a process node. It also makes each individual plot type, such as scatter plot, matrix scatter plot and parallel coordinates plot contain the same base functionality. This base functionality provides the capability to export the data (i.e. solutions shown in the graphical plots) to different formats, including in the form of an image (e.g. PNG, JPEG), or save the data in text human-readable formats, such as JSON[13] (short for JavaScript Object Notation). The base functionality also provides capabilities to handle information flow in both ways between the graphical plots and the optimization library Tigon. Thus, when the user interacts with some of the features of a plot they are fed back to the optimizer. Existing features that take advantage of this capability are:

1. Real-time update: the user is able to visualize changes being made to the population of solutions during the optimization run;
2. Interactive decision-making: a DM is able to provide his/her preferences (e.g. goals or target levels with respect to objectives) by interactively brushing the preferred region in the objective space directly in the graphical plots as a way to guide the optimization process towards the most preferred solution;
3. Data filtering: check boxes exist in the GUI that toggle between showing all solutions or just showing only those that are non-dominated, feasible (satisfy all constraints), or pertinent (satisfy all goals).

6.2. How to create, setup, and run a simple optimization workflow

This section demonstrates how to create a simple optimization workflow with a benchmark optimization problem, namely DTLZ1, and a multi-objective optimization algorithm, in this case NSGA-II is chosen. The optimizer is run for a number of iterations and the obtained results are shown in the visualization nodes.

The first step is to create an empty project by clicking on File, New, and then select "Liger Optimization Workflow". The process

[13] JSON is an open-standard file format that is language-independent and it is supported by many applications, including the existence of built-in functions in Matlab and Python for dealing with JSON files.

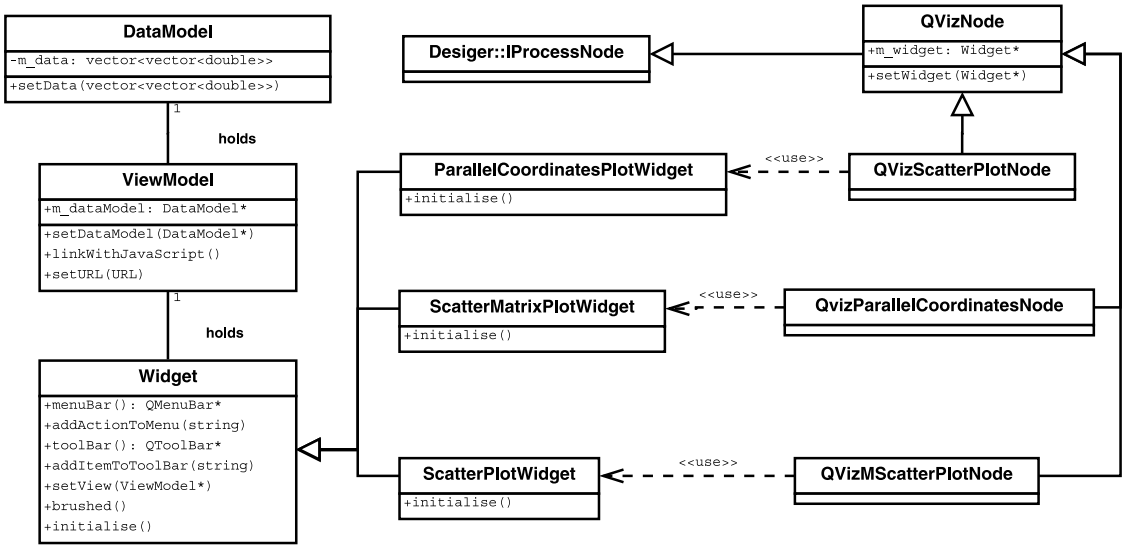


Fig. 12. Liger visualization plugin architecture.

nodes can now be added to the workflow by drag and drop from the left panel to the area in the middle as shown in Fig. 13. All nodes shown in the figure are then connected in series by having MSN (Green) as the first node, and MEN (Red) as the last one. The other nodes are as follows: *Prob*, loads the optimization problem; *RInit*, initializes a population of solutions randomly; *Eval*, evaluates the optimization model; *NSGA-II*, runs the optimizer; and the last three are visualization nodes corresponding to parallel coordinates plot, scatter plot, and matrix scatter plot.

The next step is to configure the process nodes, and in this example we only need to consider the problem formulation, random initialization, and MEN. The configuration of each node is presented in a window when a user double clicks the node as shown in Fig. 14. Consider the following:

1. In the problem formulation, DTLZ1 is loaded as a function with five input variables and three objectives. Given that DTLZ1 is an internal problem of Liger, it is accessible without the need for the user to provide an implementation of the problem model.
2. In the random initialization window the OptimizationSetSize property is set to 100. This means that the initial population will be initialized with a total of 100 solutions.
3. In the MEN we define the termination criterion for the optimization run. For this the maximum number of iterations property is set to 100.

Once the workflow has been configured, an optimization run is initiated by clicking on the green button (▶) shown in the bottom left corner of the Liger GUI. It is possible to stop at any time the optimization run by clicking on the red button (■) just below. Fig. 15 shows the graphical plots of the visualization nodes obtained at the end of the optimization run, in that: (i) the scatter plot (top-right) shows the solutions represented in the objective space for the first two objectives, although it is possible to select any other dimensions to be represented; (ii) the matrix scatter plot (bottom-left) shows a grid of scatter plots with all possible combinations between the objectives, it is also possible to extend this to the decision space; and (iii) the parallel coordinates plot (bottom-right) shows the solutions represented as connecting lines across all objectives, and decision variables. Objectives can be in total or partial conflict with each other, and this can be identified in a parallel coordinates plot, where conflict corresponds to regions where lines cross. For instance, in Fig. 15 the second objective is shown to be in conflict with the other two since most lines are shown crossing. Below the parallel coordinates plot there is a table that shows the values of the objectives and decision variables for all solutions. The data shown in the plots can now be saved as figures (e.g. PNG or JPEG) or exported into JSON format.

6.3. Demonstration of interactive decision-making on a real-world optimization problem

This section demonstrates the interactive decision-making capabilities of Liger on a real-world optimization problem. The problem model is implemented in Matlab. The term *interactive decision-making* has been mentioned in Section 6.1.2 as one of the features of the visualization plugin, where a DM can interact directly with the graphical plots to provide his/her preferences. The preferences take the form of goals with respect to the objectives, and an optimizer is tasked with finding solutions that satisfy them. For this, the optimizer uses an implementation of the preferability relation defined in [4] to induce an order between the solutions based on the provided goals. This feature is available in Liger for Pareto-based MOEAs such as NSGA-II and MOGA.

We consider a simplified control engineering problem where the aim is to design a digital feedback control system for a plant.[14] The problem model is available as one of the examples provided by the Liger software.[15] As is common in industrial control problems, the design engineer has the task of tuning a so-called *Proportional–Integral (PI)* controller. The tuning task involves selecting the values of the controller's two parameters—known as the *proportional gain*, K_p and the *integral gain*, K_i. There are nine performance criteria for the controller to meet:

1. Stability criteria (representing the extent to which the control system exhibits stable behaviour—these criteria are important for safety considerations):

[14] More details about the system model of the plant can be found in [36] page 265.

[15] Depending on where Liger has been installed (assuming to be in $LIGER), the model of the control system problem can found in the following folder $LIGER/share/liger/examples/matlab/Ogata/.

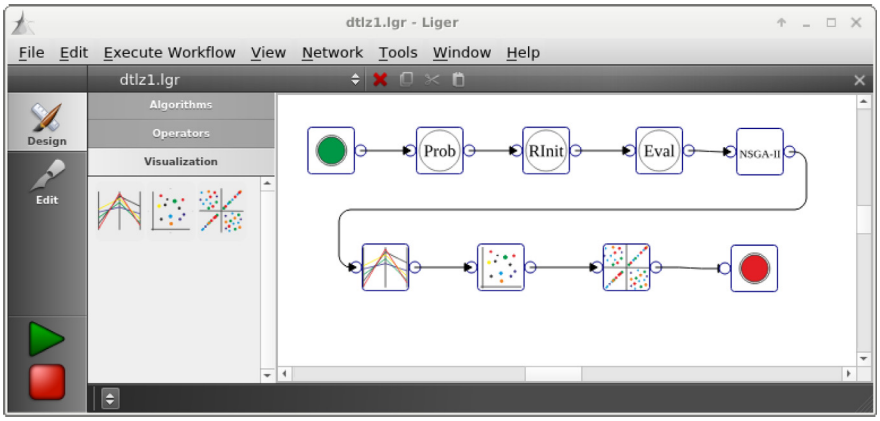


Fig. 13. Liger workflow.

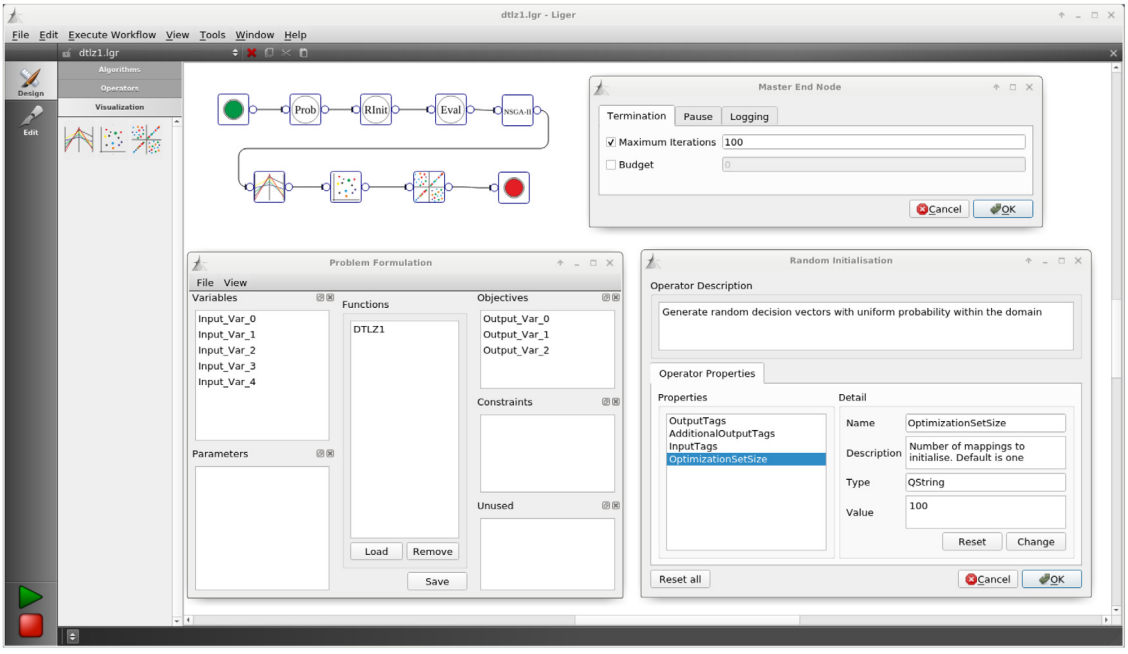


Fig. 14. Liger nodes configuration.

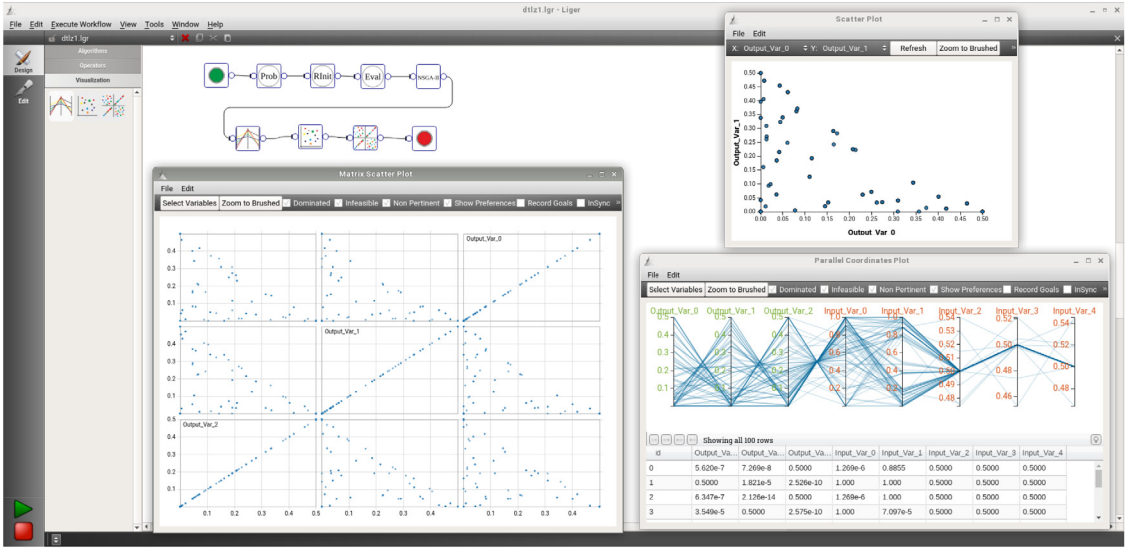


Fig. 15. Liger visualization nodes.

Table 1
Chief Engineer preferences for the digital control system.

Criterion	Direction	Goal	Priority
Largest closed-loop pole	Minimize	<1	Hard constraint
Gain margin	Maximize	10 dB	High
Phase margin	Range	$\geq 30°$ & $\leq 60°$	High
Rise time	Minimize	2 s	Moderate
Peak time	Minimize	10 s	Low
Maximum overshoot	Minimize	10%	Moderate
Maximum undershoot	Minimize	8%	Low
Settling time	Minimize	20 s	Low
Steady-state error	Minimize	1%	Moderate

(i) Magnitude of largest pole in the closed-loop transfer function (for a digital system, the pole magnitude must not exceed unity);
(ii) Gain margin;
(iii) Phase margin;

2. Transient criteria (relating to how satisfactorily the control system responds in the short-term to changes in demand):

(iv) Rise time (from 10% to 90% of the demanded value);
(v) Peak time;
(vi) Maximum overshoot;
(vii) Maximum undershoot;
(viii) Settling time (within 2% of the demanded value);

3. Steady-state criteria (relating to how satisfactorily the control system responds in the longer-term to changes in demand):

(ix) Steady-state error.

Note that the precise meanings of these criteria are not essential to the example—however the interested reader can find more details in [36]. The criteria are provided by the function *evaluateControlSystem* which takes a pair of controller gains (K_P, K_I) as input. The intention is to identify a set of controller gains that will meet the preferences of a Chief Engineer in relation to the performance criteria. These preferences are outlined in Table 1. The Chief Engineer is allowed to change the preferences during the optimization run based on any evidence presented to him/her that suggests, for instance, that these are unlikely to be satisfied, following which the Chief Engineer could make a decision about either relaxing or tightening the existing goals.

Fig. 16 shows a Liger workflow with the control engineering problem setup. The workflow is very similar to the one in Fig. 13, and the only difference is that instead of the node *Eval* there is *BEval*. The latter sends a batch of solutions to the evaluation function all at once, as opposed to calling the evaluation function several times, once for each solution. This is advantageous when interfacing with external programs like Matlab since it is faster to evaluate solutions in a batch as opposed to having to call the evaluation function several times. NSGA-II is chosen as the optimizer. This may come as a surprise given the earlier statement in Section 2 where NSGA-II, to some extent, is not suitable for dealing with multi-objective problems with four or more objectives. The main reason for this relates to the inefficacy of Pareto-dominance based primary selection, where all solutions can become non-dominated, leading to poor selection pressure for convergence to the POF. However, this difficulty is countered by exploiting, via inclusion of the preferability component, the preference order over the non-dominated solutions induced by the Chief Engineer's preferences.

The problem formulation in Fig. 16 shows the controller gains defined as variables, and each controller gain has been set to take any value from the interval [0.0001, 10] (defined on a logarithmic scale). The outputs of the problem function are defined as objectives, the only exception being the largest closed-loop pole (labelled in the workflow as CloseLoopStability) which is defined as a constraint. One of the current limitations of the problem formulation in Liger is that only one goal (or threshold) can be defined per output (either goals for objectives, or threshold for constraints). To address this issue, and knowing that the Chief Engineer wishes to restrict the phase margin criterion by a given range, we have exposed the phase margin output twice in the problem model, and we set one for minimization (PhaseMarginMin) and the other for maximization (PhaseMarginMax). Moreover, in the MEN the number of iterations to run the optimizer is set to 300, and in the tab Pause we have set two intervention points corresponding respectively to iterations 100 and 200. During each intervention point the obtained results are shown to the Chief Engineer, and a decision will be made if a change to the existing preferences is deemed desirable.

The results obtained after each intervention point, and also at the end of the optimization run, are shown in Fig. 17. For the first 100 iterations no goals have been set for the objectives to allow the optimization algorithm to explore all potential trade-offs, and all solutions obtained are shown in Fig. 17a. Considering the objectives with a high priority, namely gain margin and phase margin:

1. The upper bound on the phase margin restricts the maximum gain margin, where the maximum value found is around 8.6 dB. This means that it is expected to be difficult to satisfy simultaneously the upper bound of the phase margin and the 10 dB goal for the gain margin. This is shown in Fig. 17b.
2. Increasing gain margin beyond 10 dB is in conflict with minimizing rise time, peak time, and settling time, while both overshoot and undershoot are zero. On the other hand, satisfying the bounds for phase margin provides lower values for rise time, peak time and settling time, while overshoot and undershoot have higher than zero values both closer to their corresponding goals.

Given that relaxing the gain margin is likely to lead to solutions with values closer to the goals of rise time, peak time, and settling time, the decision is to relax the gain margin goal from 10 to 6 dB. The goals for the objectives with high priority are now set,[16] and the solutions found at iteration 200 are shown in Fig. 17c. Considering now the objectives with a moderate priority, namely rise time, overshoot, and steady-state error. Most solutions found satisfy the goals for overshoot and steady-state error. However, the lowest value for rise time is around 2.95 s which is higher than the goal of 2 s. There are also many solutions found that satisfy the goals for overshoot and steady-state error, and at same time deliver a gain margin close to 7 dB. The decision is to relax the rise time goal from 2 to 3 s, and to tighten the goal of gain margin from 6 to 7 dB. The goals for the objectives with a moderate priority are now set, and the solutions found at iteration 300 are shown in Fig. 17d. All solutions found satisfy the goals for peak time, undershoot and settling time. A decision is now made about which solution to pick given that the range of possible values is quite restricted: [0.2589, 0.2724] and [0.2457, 0.2557] for K_p and K_i respectively. Given that the gain margin has a high priority, we now assume that the Chief Engineer chooses the solution with the highest gain

[16] The goals for the objectives can be set directly in the parallel coordinates plot by brushing the objectives as shown in Fig. 17b, where the upper bound (for minimization) or lower bound (for maximization) of the brushed area is set as the current goal.

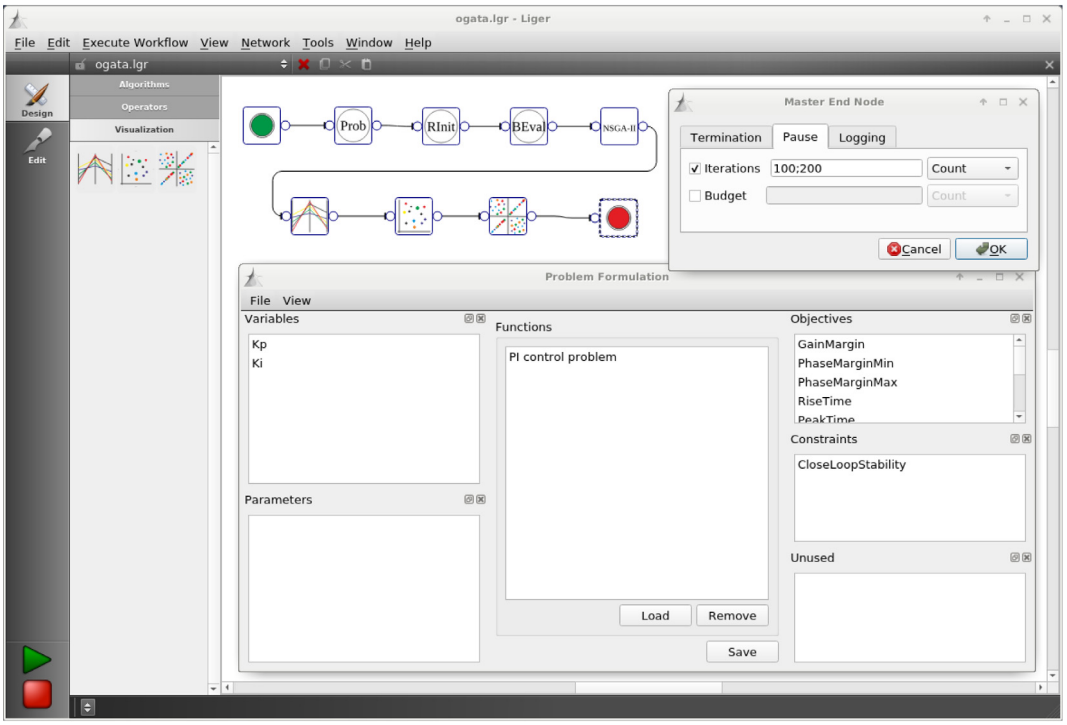


Fig. 16. Setup of the control engineering problem in Liger.

margin, which corresponds to controller gains $K_p = 0.2623$ and $K_i = 0.2473$, and delivers the following outputs: a gain margin of 7.362 dB, a rise time of 3 s, a peak time of 8 s, an overshoot of 7.959, a settling time of 10.23 s, and a steady state-error close to zero.

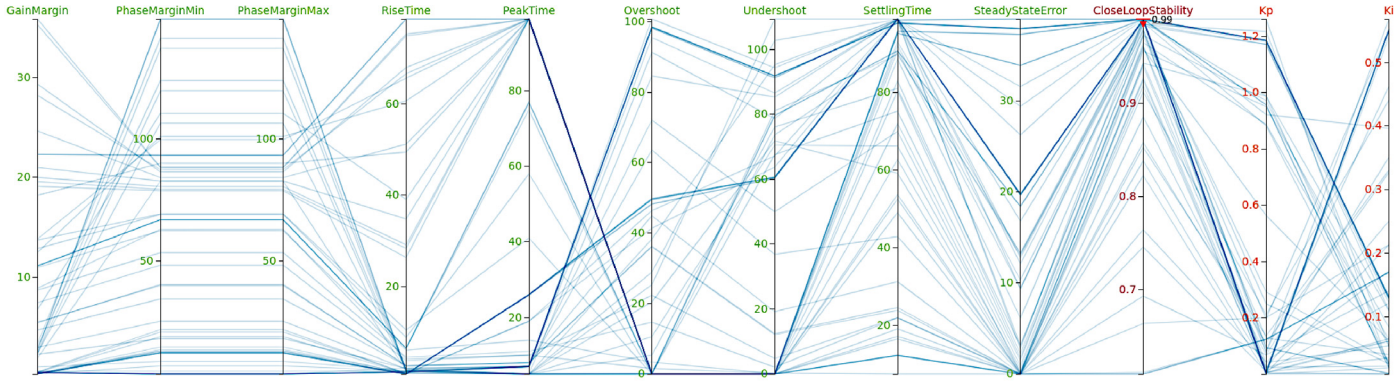
This example has shown a very important aspect of the interactive decision-making capabilities of Liger, which relates to how the tasks of optimization and decision-making are conducted. Traditionally in the field of Evolutionary Multi-objective Optimization (EMO), an MOEA is used to find a set of trade-off solutions, and then a DM is asked to select from amongst the available options the most desirable solution. The new paradigm introduced by the MCDM based MOEAs is to conduct optimization and decision-making interchangeably, giving more control to a DM (or multiple DMs) to guide the search to the solution most preferred by them. This also has the potential to inform the DM about some aspects of the problem model that he/she was not previously aware of (e.g. for the example problem, that by having an upper bound on the phase margin of 60° it is not possible to deliver a gain margin of 10 dB).

7. Summary

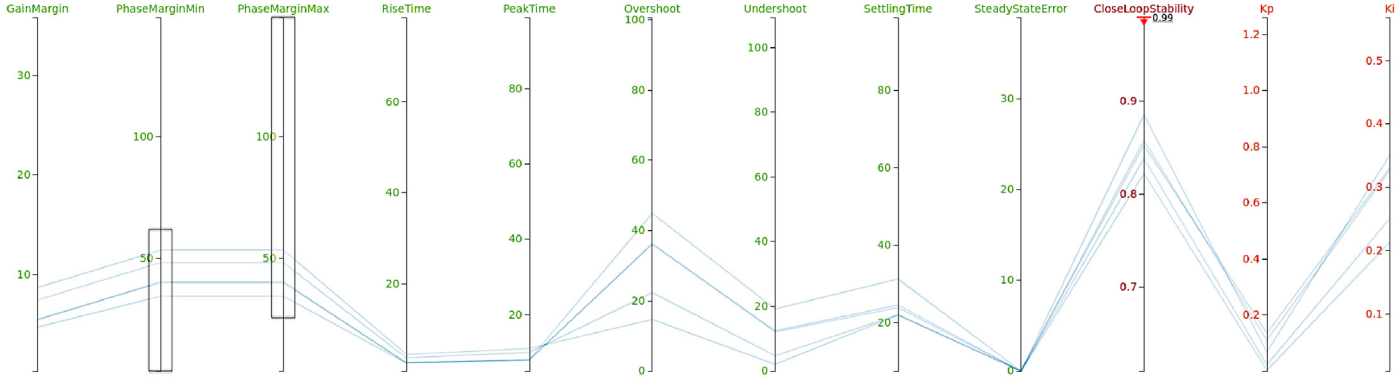
This paper marks a new direction in the development of Liger, extending its capabilities beyond those offered by the previous version. The new version (v1.0) is released under the LGPL licence and is publicly available as open-source in GitHub. The development of Liger has been steered by the need to solve real-world optimization problems found in industrial applications, many of those suggested by our industrial partners (e.g. Ford and Jaguar Land Rover). These optimization problems are often characterized for having constraints, multiple objectives, and a mix between different types of variables (e.g. continuous and discrete). There is also a need to account for various sources of uncertainty such as production variations, changing environmental conditions and model–method combination errors. In this process, we have learnt that many optimization practitioners inside different organizations are not necessarily optimization specialists, and they often use commercial software that cannot be tailored to their exact needs, or lack transparency about which optimization methods are being used under the hood. Therefore, the ideal optimization software for this target audience needs to be intuitive, easy-to-use and customizable. There is also the need to accommodate researchers that require an open, flexible, reusable and sustainable optimization environment to develop and maintain new algorithms. The previous version of Liger [9] relied on the jMetal optimization library which could not satisfy all these requirements, and more work needed to be done on the GUI to make it more intuitive and simpler to use.

To address the above, one major contribution of this work is a new optimization library, known as Tigon. The new library provides a new design paradigm for evolutionary algorithms based on the decorator design pattern. This allows for rapid algorithm prototyping where an algorithm is simply a composite of different operators, and new algorithms can be created by introducing new operators and/or recombining existing ones. We have shown how to use this concept to compose a popular elitism MOEA, namely NSGA-II. The implementation of other types of MOEAs covering different evolutionary computation paradigms are available in the Tigon library, and this demonstrates that the decorator design pattern provides a flexible approach for composing MOEAs. Other features found in the Tigon library include: support for progressive articulation of user preferences during the optimization run when using Pareto-based MOEAs; performance assessment metrics for comparing MOEAs; support for parallel evaluations to take advantage of multi-core systems to speed up the optimization process; support for a number of programming languages to implement the optimization problem; support for optimization problems with a mix of different types of variables (e.g. continuous, discrete, and also deterministic and uncertain).

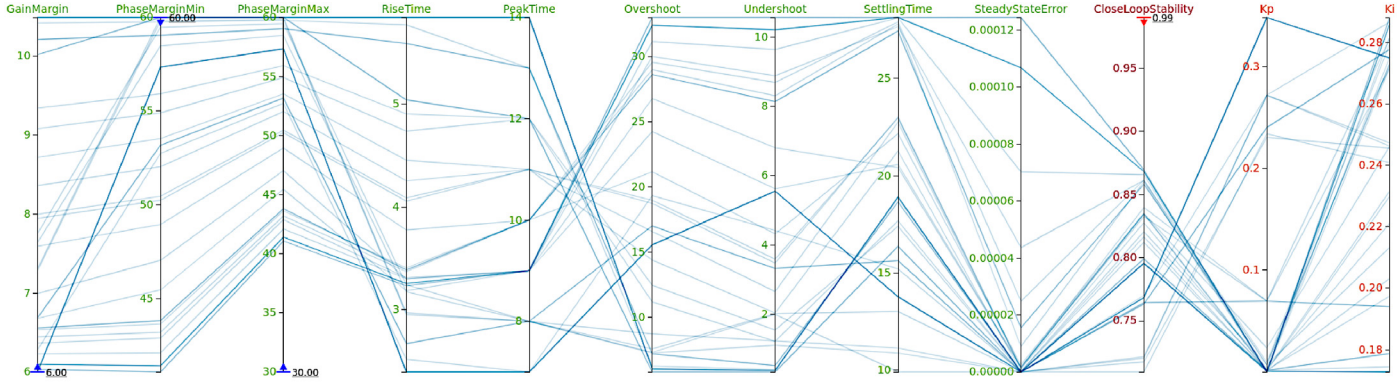
The GUI of Liger offers an intuitive interface where an optimization workflow can be constructed by placing operator nodes via a simple drag-and-drop functionality. During an optimization run, a user is able to inspect the obtained results by using



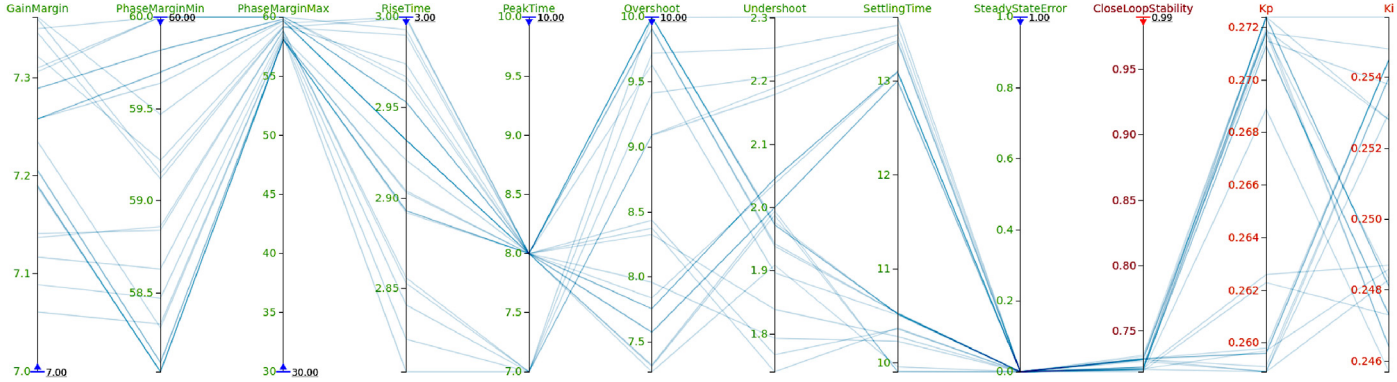
(a) Iteration 100: all solutions obtained without setting any goals for the objectives.



(b) Iteration 100: brushing the phase margin criterion to investigate the effects on gain margin and other objectives.



(c) Iteration 200: showing solutions after setting goals for high priority objectives.



(d) Iteration 300: showing solutions after setting goals for moderate priority objectives.

Fig. 17. Results of the control engineering problem showcasing the interactive decision-making capabilities of Liger.

state-of-the-art visualization tools tailored for multi-objective optimization. To facilitate the process of identifying the most preferred solution, a DM is able to provide his/her preferences in a form of a goal vector to express some form of target levels with respect to the objectives. It is possible to update the goal vector during an optimization run by interacting directly with the graphical plots offered by the visualization tools (e.g. via parallel coordinates plot). The preference articulation feature together with the existing visualization tools are highly valuable for dealing with real-world optimization problems since the initial preferences by a DM may be infeasible. It has been shown on a control engineering example problem that by conducting optimization and decision-making interchangeably, not just facilitates the search for the most desirable solution by the DM, but offers the potential to reveal the preference structure embedded in the optimization models to the DM. This is a new optimization paradigm promoted by the so called MCDM based MOEAs that is offered by the Liger software. Moreover, the functionality of the GUI can be easily extended in a flexible manner via a plugin system. This gives more freedom to the user to incorporate other features into Liger, which could be turned on or off on demand, and without having to modify any of the existing core libraries.

For future work, there are several research directions that are worth pursuing. For this, the authors endeavour to provide support for other types of problems. This includes Multidisciplinary Design Optimization (MDO) [37] problems that are characterized for having multiple interacting components, where each component can have its own (specialized) design task. This requires the development of new capabilities in Liger to support the construction of such problems, including the provision of suitable optimization approaches for dealing with them. Some examples currently found in the literature make use of bilevel or distributed approaches as described in [38]. Other features actively being pursued are:

1. preference articulation and constraint handling strategies for different types of MOEAs—currently both features are only available for Pareto-based MOEAs;
2. exploit other approaches to enable parallel evaluations (and perhaps parallelization at other algorithm levels)—current approach is to use the multi-threading capabilities of the C++ standard library but other libraries, such as OpenMP and MPI, might offer some advantages when used in modern High Performance Computing (HPC) facilities; and
3. consider the application of data-mining approaches to reveal interesting problem features from the optimization data that could be used to influence the decision-making process, for instance, make use of dimensionality reduction techniques, not just to reduce the problem complexity, but also to reveal the preference-structure of the objective functions.

CRediT authorship contribution statement

João A. Duro: Methodology, Software, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Yiming Yan:** Methodology, Software, Data curation, Writing - original draft, Visualization. **Ioannis Giagkiozis:** Conceptualization, Methodology, Software, Data curation, Writing - original draft. **Stefanos Giagkiozis:** Software, Writing - original draft. **Shaul Salomon:** Conceptualization, Methodology, Software, Visualization. **Daniel C. Oara:** Methodology, Software. **Ambuj K. Sriwastava:** Methodology, Software. **Jacqui Morison:** Conceptualization, Resources, Project administration. **Claire M. Freeman:** Conceptualization, Resources, Project administration, Funding acquisition. **Robert J. Lygoe:** Conceptualization, Resources, Project administration, Funding acquisition. **Robin C. Purshouse:** Conceptualization, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Peter J. Fleming:** Conceptualization, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to acknowledge financial support from EPSRC, United Kingdom and Jaguar Land Rover as part of the jointly funded Programme for Simulation Innovation (PSi) (EP/L025760/1), Innovate UK and Ford Motor Company as part of the Advanced Propulsion Centre UK project DYNAMO (grant 113130), Marie Curie International Research Staff Exchange Scheme Fellowship within the 7th European Community Framework Programme (grant agreement 295152), the University of Sheffield, United Kingdom, and the Institute of Digital Engineering (IDE), a spoke of the Advanced Propulsion Centre UK (grant J14921), and Daniel C. Oura acknowledges EPSRC scholarship support (EP/M508135/1 and EP/M506618/1). We would also like to acknowledge Samuele De-Guido from IDE for supporting the open-source release of Liger.

References

[1] K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley & Sons, Ltd, 2001.
[2] C.C. Coello, G.B. Lamont, D.A. van Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems, Kluwer Academic Publishers, 2002.
[3] L.C.T. Bezerra, M. López-Ibáñez, T. Stützle, Automatic component-wise design of multi-objective evolutionary algorithms, IEEE Trans. Evol. Comput. 20 (3) (2016) 403–417, http://dx.doi.org/10.1109/TEVC.2015.2474158.
[4] C.M. Fonseca, P.J. Fleming, Multiobjective optimization and multiple constraint handling with evolutionary algorithms—Part I: A unified formulation, IEEE Trans. Syst. Man Cybern. A 28 (1) (1998) 26–37, http://dx.doi.org/10.1109/3468.650319.
[5] K. Deb, A. Sinha, P.J. Korhonen, J. Wallenius, An interactive evolutionary multiobjective optimization method based on progressively approximated value functions, IEEE Trans. Evol. Comput. 14 (5) (2010) 723–739, http://dx.doi.org/10.1109/TEVC.2010.2064323.
[6] R. Wang, R.C. Purshouse, P.J. Fleming, "Whatever works best for you"- A new method for a priori and progressive multi-objective optimisation, in: Evolutionary Multi-Criterion Optimization: 7th International Conference, EMO 2013, Sheffield, UK, March 19-22, 2013. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 337–351, http://dx.doi.org/10.1007/978-3-642-37140-0_27.
[7] J.A. Duro, D. Kumar Saxena, K. Deb, Q. Zhang, Machine learning based decision support for many-objective optimization problems, Neurocomputing 146 (2014) 30–47, http://dx.doi.org/10.1016/j.neucom.2014.06.076, Bridging Machine learning and Evolutionary Computation (BMLEC) Computational Collective Intelligence.
[8] J. Branke, in: G. S., E. M., F. J. (Eds.), MCDA and Multiobjective Evolutionary Algorithms, in: International Series in Operations Research & Management Science, vol. 233, Springer, 2016, http://dx.doi.org/10.1007/978-1-4939-3094-4_23, Ch. Multiple Criteria Decision Analysis.
[9] I. Giagkiozis, R.J. Lygoe, P.J. Fleming, Liger: An open source integrated optimization environment, in: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, in: GECCO '13 Companion, ACM, New York, NY, USA, 2013, pp. 1089–1096, http://dx.doi.org/10.1145/2464576.2466801.
[10] S. Giagkiozis, R.J. Lygoe, I. Giagkiozis, P.J. Fleming, Diesel engine drive-cycle optimization with liger, in: International Conference on Evolutionary Multi-Criterion Optimization (EMO), Springer International Publishing, 2015, pp. 328–342, http://dx.doi.org/10.1007/978-3-319-15892-1_22, Part of the Lecture Notes in Computer Science book series (LNCS, volume 9019).
[11] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P.N. Suganthan, Q. Zhang, Multiobjective evolutionary algorithms: A survey of the state of the art, Swarm Evol. Comput. 1 (1) (2011) 32–49, http://dx.doi.org/10.1016/j.swevo.2011.03.001.

[12] S. Bleuler, M. Laumanns, L. Thiele, E. Zitzler, PISA — A platform and programming language independent interface for search algorithms, in: C.M. Fonseca, P.J. Fleming, E. Zitzler, K. Deb, L. Thiele (Eds.), Evolutionary Multi-Criterion Optimization (EMO 2003), in: Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 494–508.

[13] A. Liefooghe, L. Jourdan, E.-G. Talbi, A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO, European J. Oper. Res. 209 (2) (2011) 104–112, http://dx.doi.org/10.1016/j.ejor.2010.07.023.

[14] M. Keijzer, J.J.M. Guervós, G. Romero, M. Schoenauer, Evolving objects: A general purpose evolutionary computation library, in: Selected Papers from the 5th European Conference on Artificial Evolution, Springer-Verlag, London, UK, UK, 2002, pp. 231–244.

[15] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, M. Affenzeller, Advanced Methods and Applications in Computational Intelligence, in: Topics in Intelligent Engineering and Informatics, vol. 6, Springer, 2014, pp. 197–261.

[16] R.C. Purshouse, P.J. Fleming, On the evolutionary optimization of many conflicting objectives, IEEE Trans. Evol. Comput. 11 (6) (2007) 770–784, http://dx.doi.org/10.1109/TEVC.2007.910138.

[17] S. Luke, ECJ then and now, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, in: GECCO '17, ACM, New York, NY, USA, 2017, pp. 1223–1230, http://dx.doi.org/10.1145/3067695.3082467.

[18] M. Lukasiewycz, M. Glaß, F. Reimann, J. Teich, Opt4J: A modular framework for meta-heuristic optimization, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, in: GECCO '11, ACM, New York, NY, USA, 2011, pp. 1723–1730, http://dx.doi.org/10.1145/2001576.2001808.

[19] M. Kronfeld, H. Planatscher, A. Zell, The EvA2 Optimization framework, in: Learning and Intelligent Optimization: 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 247–250, http://dx.doi.org/10.1007/978-3-642-13800-3_27.

[20] A. Ramírez, J.R. Romero, S. Ventura, An extensible JCLEC-based solution for the implementation of multi-objective evolutionary algorithms, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, in: GECCO '15, ACM, New York, NY, USA, 2015, pp. 1085–1092, http://dx.doi.org/10.1145/2739482.2768461.

[21] D. Hadka, MOEA framework: A free and open source Java framework for multiobjective optimization, 2017, http://moeaframework.org, [Online; accessed 01-07-2019].

[22] J.J. Durillo, A.J. Nebro, Jmetal: A Java framework for multi-objective optimization, Adv. Eng. Softw. 42 (10) (2011) 760–771, http://dx.doi.org/10.1016/j.advengsoft.2011.05.014.

[23] A.J. Nebro, J.J. Durillo, M. Vergne, Redesigning the jMetal multi-objective optimization framework, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, in: GECCO '15, ACM, New York, NY, USA, 2015, pp. 1093–1100, http://dx.doi.org/10.1145/2739482.2768462.

[24] Y. Tian, R. Cheng, X. Zhang, Y. Jin, PlatEMO: A MATLAB platform for evolutionary multi-objective optimization, IEEE Comput. Intell. Mag. 12 (4) (2017) 73–87, http://dx.doi.org/10.1109/MCI.2017.2742868.

[25] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast elitist multi-objective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197, http://dx.doi.org/10.1109/4235.996017.

[26] Q. Zhang, H. Li, MOEA/D: a multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comput. 11 (6) (2007) 712–731, http://dx.doi.org/10.1109/TEVC.2007.892759.

[27] M. Emmerich, N. Beume, B. Naujoks, An EMO algorithm using the hypervolume measure as selection criterion, in: C. Coello Coello, A. Hernández Aguirre, E. Zitzler (Eds.), Evolutionary Multi-Criterion Optimization, in: Lecture Notes in Computer Science, vol. 3410, Springer Berlin / Heidelberg, 2005, pp. 62–76.

[28] J. Knowles, ParEGO: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems, IEEE Trans. Evol. Comput. 10 (1) (2006) 50–66, http://dx.doi.org/10.1109/TEVC.2005.851274.

[29] S. Salomon, C. Domínguez-Medina, G. Avigad, A. Freitas, A. Goldvard, O. Schütze, H. Trautmann, PSA based multi objective evolutionary algorithms, in: O. Schuetze, C.A. Coello Coello, A.-A. Tantar, E. Tantar, P. Bouvry, P.D. Moral, P. Legrand (Eds.), EVOLVE - a Bridge Between Probability, Set Oriented Numerics, and Evolutionary Computation III, Springer International Publishing, Heidelberg, 2014, pp. 233–259.

[30] J.A. Duro, R.C. Purshouse, S. Salomon, D.C. Oara, V. Kadirkamanathan, P.J. Fleming, SParEGO – A hybrid optimization algorithm for expensive uncertain multi-objective optimization problems, in: K. Deb, E. Goodman, C.A. Coello Coello, K. Klamroth, K. Miettinen, S. Mostaghim, P. Reed (Eds.), Evolutionary Multi-Criterion Optimization, Springer International Publishing, 2019, pp. 424–438.

[31] N. Beume, C.M. Fonseca, M. López-Ibáñez, L. Paquete, J. Vahrenhold, On the complexity of computing the hypervolume indicator, IEEE Trans. Evol. Comput. 13 (5) (2009) 1075–1082, http://dx.doi.org/10.1109/TEVC.2009.2015575.

[32] M. Bostock, V. Ogievetsky, J. Heer, D3: Data-driven documents, IEEE Trans. Vis. Comput. Graphics (Proc. InfoVis) (2011).

[33] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[34] R.J. Winter, Agile Software Development: Principles, Patterns, and Practices, Wiley Online Library, 2014.

[35] K. Deb, L. Thiele, M. Laumanns, E. Zitzler, Scalable test problems for evolutionary multi-objective optimization, in: A. Abraham, R. Jain, R. Goldberg (Eds.), Evolutionary Multiobjective Optimization: Theoretical Advances and Applications, Springer, 2005, pp. 105–145.

[36] K. Ogata, Discrete-Time Control Systems, second ed., Prentice-Hall, Inc, 1995.

[37] J.R.R.A. Martins, A.B. Lambe, Multidisciplinary design optimization: a survey of architectures, AIAA J. 51 (9) (2013) 2049–2075.

[38] K. Klamroth, S. Mostaghim, B. Naujoks, S. Poles, R. Purshouse, G. Rudolph, S. Ruzika, S. Sayın, M.M. Wiecek, X. Yao, Multiobjective optimization for interwoven systems, J. Multi-Criteria Decis. Anal. 24 (1–2) (2017) 71–81, http://dx.doi.org/10.1002/mcda.1598.