JIOP: A JAVA INTELLIGENT OPTIMISATION AND MACHINE LEARNING FRAMEWORK

L. I. Hatledal, F. Sanfilippo, H. Zhang

Department of Maritime Technology and Operations
Aalesund University College
Postbox 1517, 6025 Aalesund, Norway

KEYWORDS

Optimisation methods; Machine learning; Object oriented programming; Inverse kinematics

ABSTRACT

This paper presents an open source, object-oriented machine learning framework, formally named *Java Intelligent Optimisation* (*JIOP*). While *JIOP* is still in the early stages of development, it already provides a wide variety of general learning algorithms that can be used.

Initially designed as a collection of existing learning methods, *JIOP* aims to emphasise commonalities and dissimilarities of algorithms in order to identify their strengths and weaknesses, providing a simple, coherent and unified view. For this reason, *JIOP* is suitable for pedagogical purposes, such as for introducing bachelor and master degree students to the concepts of intelligent algorithms.

The problems that *JIOP* aims to solve are initially discussed to demonstrate the need for such a framework. Later on, the design architecture and the current functions of the framework are outlined. As a validating case study, a real application where *JIOP* is used to minimise the cost function for solving the inverse kinematics (IK) of a *KUKA* industrial robotic arm with six degrees of freedom (DOF) is also presented. Related simulations are carried out to prove the effectiveness of the proposed framework.

INTRODUCTION

Machine learning refers to the ability of a computer system, or more generally, a machine, to learn from examples (Simon 2013). Essentially, this ability concerns the task of computing a mathematical rule that generalises a relationship initially provided by a finite sample of real data, without being explicitly programmed but by following some kind of training process. This idea is based on the fundamental concepts of representation, evaluation and generalisation. Representation of data instances and evaluation of the same instances by using some kind of assessing function are essential steps for processing the information carried out by the data. Generalisation is a fascinating property that guarantees good system performance on unseen data instances. The set of all possible outputs when given all possible inputs is too large to be covered by the set of observed data. Hence, the system must generalise from the given set of examples, so as to be able to produce useful output in new cases.

Machine learning is highly pervasive today and it is employed in a wide variety of tasks and successful applications in several fields, such as computer vision, natural language processing, pattern recognition, search engines, bio-informatics, robotics, and more generally, for a wide variety of optimisation processes. Optimisation problems often represent very complex tasks and non-heuristic methods are greatly limited in finding proper solutions (Pluhacek et al. 2013). A vast number of different algorithms have already been presented in previous literature and they can be classified into a taxonomy based on the type of input available during the training process (Hormozi et al. 2012). Supervised learning algorithms are trained on labelled examples and and their main objective consists of generating a function that maps inputs to desired outputs (Kotsiantis et al. 2007). Unsupervised learning algorithms operate on unlabelled examples and attempt to discover some kind of structure in the data (Alpaydin 2004). Semi-supervised learning combines both labelled and unlabelled examples to generate an appropriate function or classifier (Zhu 2006). Transduction methods try to predict new outputs based on specific and fixed test cases from observed specific training cases (Alpaydin 2004). Reinforcement learning is concerned with learning how to act given an observation of the environment to maximise some notion of reward (Alpaydin 2004). Learning to learn is a model of inductive bias learning based on previous experience (Evgeniou & Pontil 2004).

These machine learning algorithms are often quite complex to implement from scratch and to use properly and efficiently for students, especially considering the limited amount of time that they can spend on this task during their bachelor or master courses of study. As such, it would be very useful to dispose of some kind of developing tool that is easy to use and at the same time gives students a chance to experience the benefit of using machine learning algorithms in practical applications. In particular, a software framework that collects different machine learning methods would greatly help students to emphasise commonalities and dissimilarities of algorithms in order to identify their strengths and weaknesses.

For these reasons, an open source object-oriented machine learning framework, formally named *Java Intelligent Optimisation* (*JIOP*) has been developed at Aalesund University College to help bachelor and master degree students use existing machine learning algorithms, combine them, extend them or even experiment with new ones. The object-oriented approach was justified by the need to create a clear modular structure for the framework. Moreover, this choice makes it

easy to maintain and modify software. Consequentially, *Java* was chosen as the language of development because it is object-oriented, easy to learn and platform-independent. *JIOP* already provides the following algorithms: Genetic Algorithm (GA) (Deb et al. 2002), Simulated Annealing (SA) (Aarts & Korst 1988), Differential Evolution (DE) (Storn & Price 1997), Particle Swarm Optimisation (PSO) (Kennedy 2010) and Artificial Bee Colony (ABS) (Karaboga & Basturk 2007). However, new methods can be easily developed and added to the framework. The framework is available under a Berkeley Software Distribution (BSD) license and can be retrieved from the following website: https://github.com/aauc-mechlab/JIOP.

RELATED RESEARCH WORK

In recent years, the machine learning community has developed a notable number of different libraries. However, most of the time, these libraries are specifically designed for a particular algorithm and for a defined application. Moreover, they are typically written by using different languages and only a few of them are publicly available. Very few comprehensive collections of different algorithms are freely available to developers and students.

In (Kohavi et al. 1994), Kohavi et al. introduced *MLC++*, a library of C++ classes for supervised machine learning. *MLC++* (up to version 1.3.X) is in the public domain and is still distributed as such by the *Silicon Graphics International* (SGI) Corporation. SGI *MLC++* (V2.0 and higher) includes improvements to the original *MLC++*. However, even if these improvements are available in both source and object code formats, they are only in the research domain. In (Abeel et al. 2009), Abeel et al. presented *Java-ML*, a collection of machine learning and data mining algorithms for both software developers and research scientists. The interfaces for each type of algorithm are quite clear and algorithms strictly follow their respective interface. In (Heaton & Reasearch 2010), Heaton outlined an advanced machine learning framework, formally named *Encog*. This framework supports a variety of advanced algorithms, as well as support classes to normalise and process data. Most *Encog* training algorithms are multi-threaded and scale well to multi-core hardware. *Encog*, which is available for Java, .Net and C/C++, can also make use of a Graphical Processing Unit (GPU) to improve processing time.

It should be noted that these libraries are often created to be used by professional developers and as such, do not have a very strong pedagogical orientation.

SYSTEM ARCHITECTURE

The *JIOP* design architecture aims to provide simplicity and flexibility. The general idea is that the user should be able to use the already-implemented algorithms or even implement new algorithms with the least amount of effort. In order to accomplish this, the framework relies on a number of abstract classes and well defined interfaces to do most of the work. Moreover, the framework supports generic types, denoted by an $<E>$ in this paper, which allows the user to choose how to represent the variables to optimise. Thanks to generics, the user is not only in control of whether or not the variables should be stored in an array, a list, or some other user defined type, but also if the variables should be stored as doubles, floats, strings, etc. Also, *JIOP* provides multi-threading support on a selected set of functions, more specifically functions that creates and evaluates multiple candidates in a single method call. A Unified Modelling Language (UML) class diagram showing the software architecture is available in Fig. 1. The general idea is that all algorithms must extend the base class, the *MLAlgorithm*, whose goal is to optimise its candidates' variables. A candidate is an *Object* with a set of variables stored in an encoding instance and a cost which relates to the fitness of the variables. A *CandidateFactory* instance may be used to create new candidates. The following subsections gives a more detailed description of the *JIOP* classes.

Base classes

The abstract *MLAlgorithm* class, described in Table I, is the base class of the framework and defines a single abstract method that subclasses must implement in order to function as a *JIOP* optimisation class. The purpose of the abstract *internalIteration()* function in all subclasses is to optimise *Candidate* instances in a single step. Furthermore, this class keeps a reference to the best *Candidate* found and also keeps a history of the performance using the *MLHistory* class.

The abstract *PopulationBasedMLAlgorithm* class is a subclass of *MLAlgorithm* and adds functionality to store a population of *Candidate* instances, as well as keeping a *MLHistory* of the average performance.

The *Candidate* class, described in Table II, is a wrapper around an *Encoding* instance and a corresponding cost. The cost is a measure of performance, where a low cost is desirable. Moreover, as the *Candidate* class implements the *Comparable* interface from the Java standard library, any array or *Collection* of *Candidates* can be sorted based upon its affiliated cost. A related subclass is the final *EvaluatedCandidate* class, which has additional information on time used and the number of iterations that were necessary to find the candidate solution.

The *Encoding* interface is the basic template for any class used to hold variables. The methods that need to be implemented are shown in Table III. More specifically, classes that implement this interface must store the actual variables used in the optimisation process. The user may choose any Java *Object* to represent the variables due to generics and should implement this interface accordingly. However, some implementations are included in the framework. Currently, these are implementations for *double[]*, *float[]*, *List<Double>* and *List<Float>* encodings. This general interface is extended by another interface, *ParticleEncoding*, which is a special case used for PSO optimisation, as this optimisation technique has a velocity associated with its variables (Kennedy 2010).

The abstract *EncodingFactory* class defines a set of factory methods, all of which returns an *Encoding* instance. These methods must be implemented by subclasses, and are subsequently used by the *CandidateFactory* class, described in Table V, to create new *Candidate* instances.

The *CandidateContainer* class extends the *ArrayList* class from the Java standard library and is used to store candidates. In addition to the standard list functionality, this class provides functions for sorting, printing and getting the average score of the contained candidates.

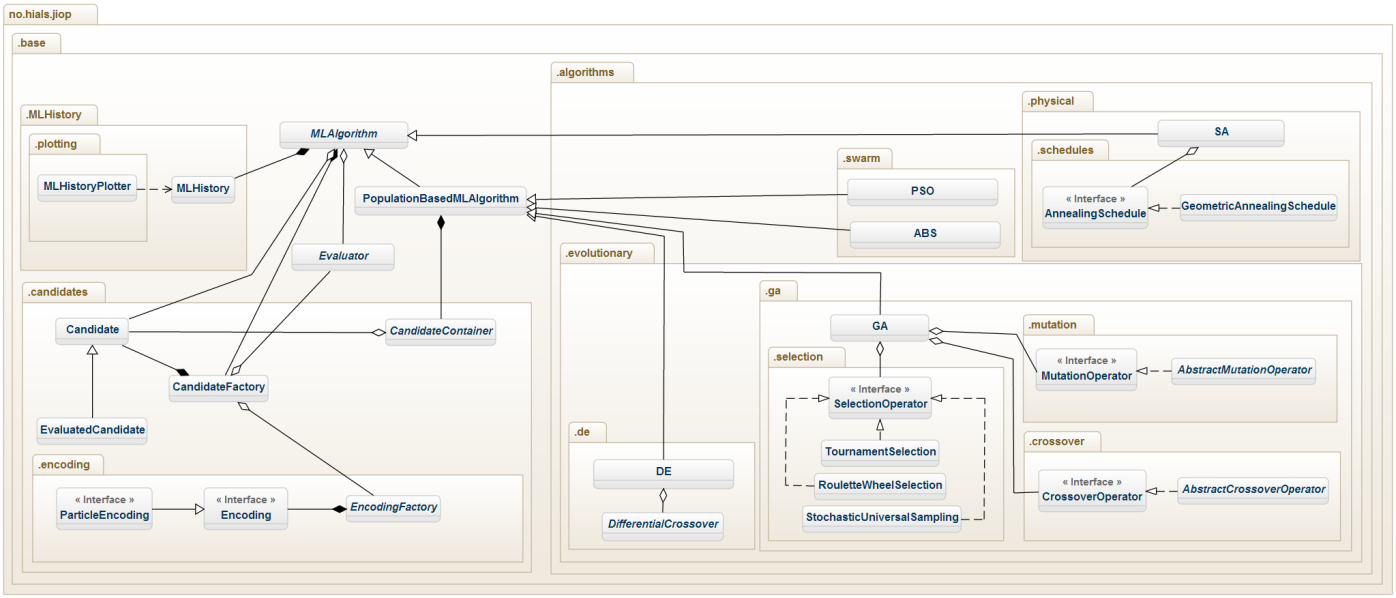

Fig. 1. A UML class diagram depicting the JIOP core classes in their respective software package

TABLE I. THE MLALGORITHM CLASS

Abstract class MLAlgorithm<E>	
Data members	
-Candidate<E> bestCandidate	A reference to the best found candidate
-AbstractEvaluator<E> evaluator	A reference to the evaluator
-CandidateFactory factory	A reference to the candidate factory
-MLHistory history	A reference to the optimisation history
-ExecutorService pool	Thread pool used for multi-threading
Functions	
+EvaluatedCandidate<E> runfor(long millis)	Runs the algorithm for the specified amount of time
+EvaluatedCandidate<E> runFor(double error)	Runs the algorithm until the error falls below the specified error
+EvaluatedCandidate<E> runFor(int iterations)	Runs the algorithm for the specified number of iterations
+void iteration()	Invokes *internalIteration()* and updates the *MLHistory*
+void reset(boolean clearHistory)	Resets the algorithm and populates it with random candidates
+void reset(boolean clearHistory, List<E> seed)	Resets the algorithm and populates it with new candidates based on the seed.
+void writeHistoryToFile(String location)	Writes the history to a text-file
+void submit(List<Runnable> jobs)	Uses multi-threading to finish the submitted jobs
Setters and getters omitted...	
Abstract functions	
+void internalIteration()	Does a single optimisation step, implemented in a subclass

TABLE II. THE CANDIDATE CLASS

Class Candidate<E> implements Comparable	
Data members	
-BasicEncoding<E> encoding	The candidate encoding.
-double cost	The candidate cost
Functions	
+Encoding<E> getEncoding()	Returns the encoding
+E getVariables()	Returns the encodings variables
+double getCost(double cost)	Returns the cost of the candidate
+void setCost()	Updates the cost
+int compareTo(Candidate c)	Used to sort the candidates

TABLE III. THE ENCODING INTERFACE

Interface Encoding<E>	
E getVariables()	The variables - double[], String[] etc.
int size()	The number of variables
Encoding<E> copy()	A unique copy of the instance

The abstract *Evaluator* class, described in Table IV, defines a single abstract method that subclasses must implement in order for the evaluation process to work. Furthermore, it has two extra convenience methods for evaluation.

The *MLHistory* class is a basic class in charge of storing a historic perspective of a *MLAlgorithm*. The class constructor takes the iteration number, time-stamp and the cost as a input, and adds the data to a list, which can be used later for plotting.

TABLE IV. THE EVALUATOR CLASS

Abstract class Evaluator<E>	
Functions	
+double evaluate(Candidate<E> candidate)	Evaluates the candidate
+double evaluate(Encoding<E> encoding)	Evaluates the encoding
Abstract Functions	
+double evaluate(E variables)	Evaluates the variables

The *MLHistoryPlotter* uses a 3rd party library, *JMathPlot* (μ-labs, 2012), and derives from the *Plot2DPanel* from said library. This class populates a line graph depicting the method's performance with regards to time or iterations, based on the *MLHistory* data from a supplied *MLAlgorithm*.

Optimisation classes

The *DE* class is an implementation of the Differential Evolution algorithm. It optimises a problem by maintaining a population of *Candidate* instances and creates new ones by

TABLE V. THE CANDIDATEFACTORY CLASS

Class CandidateFactory<E>	
Data members	
-ExecutorService pool	Thread pool used for multi-threading
-Evaluator<E> evaluator	A reference to the evaluator
-EncodingFactory<E> factory	A reference to the encoding factory
Functions	
+Candidate<E> random()	Returns a candidate with random variables
+Candidate<E> toCandidate(E e)	Returns a candidate based on the variables
+Candidate<E> neighbour(Candidate<E> c)	Returns a neighbour candidate
+List<Candidate<E>> randomCandidates(int n)	Returns a List of n candidates (multi-threaded operation)
+List<Candidate<E>> toCandidates(List<E> e)	Creates a list of candidates from the list of variables (multi-threaded operation)
+List<Candidate<E> neighbourCandidates(Candidate<E> original, int n)	Returns a list of n neighbour candidates (multi-threaded operation)

combining existing ones. The behaviour of the algorithm can be influenced by modifying the size of the population *NP*, the weighting factor *F* and the crossover weight *CR*.

The *PSO* class is an implementation of the Particle Swarm optimisation algorithm using a swarm of *Candidate* instances. This method implements the *ParticleEncoding* interface as it allows the *Candidate* to be updated according to the PSO scheme. The behaviour of the algorithm can be influenced by modifying the size of the swarm, the inertia weight ω and the learning factors c_1 and c_2.

The *ABS* class is an implementation of the Artificial Bee Colony algorithm, which is based on the intelligent behaviour of a honey bee swarm. The behaviour of the algorithm can be influenced by setting the size of the colony and the number of bees employed as scouts.

The *GA* class is an implementation of a continuous Genetic algorithm. It holds a population of *Candidate* instances and optimises a problem by mimicking natural evolution using *elitism*, *selection*, *crossover* and *mutation*. These operators can be implemented by the user by way of the *SelectionOperator*, *CrossoverOperator* and *MutationOperator* respectably. The behaviour of the algorithm can be greatly influenced depending on the selection and mutation operators as well as the population size, crossover rate, mutation rate and elitism variables.

The *SA* class is an implementation of the Simulated Annealing algorithm. It tries to mimic the annealing process in metallurgy. It does not rely on a population of *Candidate* instances, but uses a current *Candidate* and generates neighbours of this instance. Furthermore, it has a starting temperature and uses an annealing schedule to regulate it, defined by the *AnnealingSchedule* interface, and a standard acceptance probability function.

CASE STUDY

The case study focuses on finding a solution to the IK problem, which consists of determining the joint parameters that provide the desired position and orientation of the *KUKA* robot's end-effector. A solution to this problem can be found using classical approaches, such as analytically using the Jacobian matrix or through geometrics. However, the geometric approach does not scale well with the number of DOFs to be controlled, as the complexity of the calculations increases. Furthermore the Jacobian approach is known to have stability issues around singular configurations due to matrix inversions. Therefore, modifications to the classical Jacobian must be introduced. The main advantage of using machine learning algorithms is to save the user from having to hard-code geometric equations or deriving the Jacobian matrix from the model's forward kinematics (FK). Moreover, singular configurations are not ill-posed as no matrix inversions are performed. In order to conduct this case study, the same framework that our own research group introduced in (Sanfilippo et al. 2013), is used. More specifically, *JIOP* is used within this framework.

Description of the evaluation function

The evaluator measures the cost of the candidates, and is made up of three components:

1) Positional error.
2) Orientation error.
3) The change in joint angles between two consecutive IK solutions.

The sum of these components gives the cost of a proposed candidate solution, where a lower score is better.

In this case study, the variables in the *Candidate* instances are stored as a *double[]*, which is an array of double precision floating point numbers. A single value represents a joint angle θ_n, whereas the whole array represents a complete joint configuration $[\theta_1, \theta_2, ...]$ for the *KUKA* robot. FK is utilised to compute the resulting end-effector position and orientation given a set of angles.

The position cost is found using the euclidean norm:

$$a = \sqrt{(p_{1_x} - p_{2_x})^2 + (p_{1_y} - p_{2_y})^2 + (p_{1_z} - p_{2_z})^2} \quad (1)$$

where p_1 and p_2 are the normalised desired and candidate position vectors.

The orientation cost is also found using the euclidean norm:

$$b = \sqrt{(o_{1_x} - o_{2_x})^2 + (o_{1_y} - o_{2_y})^2 + (o_{1_z} - o_{2_z})^2} \quad (2)$$

where o_1 and o_2 are the normalised desired and candidate orientation vectors.

The cost related to the changes in joint angles between two successive solutions is given by (3) and is the sum of the per-element absolute difference between the previous solution $\hat{\theta}(t-1)$ and the candidate solution $\hat{\theta}(t)$.

$$c = |\hat{\theta}\text{(t-1)} - \hat{\theta}(t)| \quad (3)$$

This cost is added to encourage similar solutions, because the 6-DOF *KUKA* robot is redundant and can have multiple valid

TABLE VI. ALGORITHM SPECIFIC PARAMETERS

Genetic Algorithm		Particle Swarm Optimisation		Differential Evolution		Artificial Bee Colony		Simulated Annealing	
Population Size	100	Swarm size	40	Population size	30	Swarm Size	30	Starting temperature, t_0	100
Selection size	.5	Inertia weight, ω	.9	Weighting factor, F	.8	Scouts	6	Annealing Schedule	Geometric
Mutation rate	.5	Local bias, c_1	.9	Crossover rate, CR	.9				
Elitism	.1	Global bias, c_2	.9						
Selection	SUS								

solutions.
The cost returned by the cost function is then given by:

$$cost = \alpha\, a + \beta\, b + \gamma\, c \tag{4}$$

where α, β and γ are weighting factors. In this case study, these are: $\alpha = 1$, $\beta = 1$ and $\gamma = 0.03$, where α and β were chosen to consider the position and orientation error equally, while γ was chosen by trial and error.

Algorithm parameters

The algorithm-specific parameters used in this case study are shown in Table VI. For the GA algorithm, the crossover rate is the probability of recombination, the mutation rate is the probability of a mutation occurring at the gene level, the elitism is the percentage of the population that survives unaltered into the next generation and the population size is the number of individuals to use. The selection type used by the GA is an implementation of *Stochastic Universal Sampling* (SUS).

The parameters used in the PSO algorithm are the inertia weight, the learning factors and the swarm size. The inertia weight controls the velocity, the learning factors are biases toward the local and global best positions respectively and the swarm size is the number of particles to use.

For the DE algorithm, the weighting factor controls the amplification of differential variation, the crossover weight probabilistically controls the amount of recombination while the population size is the number of parameter vectors to use.

The ABS algorithm parameters are the number of bees in the colony, and the number of bees employed as scouts. The scouts are responsible of looking for promising food sources and communicate findings to the rest of the swarm.

Finally, the starting temperature for the SA algorithm is the initial temperature of the system. The SA uses a *GeometricAnnealingSchedule* with a constant decay rate of 0.9 to iteratively cool the temperature.

Optimisation using the JIOP framework

After extending the *Evaluator* class and implementing the *evaluate(E variables)* function, an instance of this class is passed on to the constructor of an *MLAlgorithm* along with a *CandidateFactory* and additional algorithm-specific methods and parameters. If the algorithm is population based, then a *CandidateContainer* instance is also required. After instantiating a *MLAlgorithm* instance, the user may call one of the *runFor()* methods, which returns an *EvaluatedCandidate* with the result. For a graphical representation of the result, the user can initiate an *MLHistoryPlotter* and pass the respective algorithm's *MLHistory* instance to it. The resulting plot can then be shown in a graphical window. Alternatively, the user may write the data to a text-file, using the *writeHistoryToFile()*, for plotting in some external software.

SIMULATION RESULTS

In this section, the performance of the machine learning algorithms currently found in the *JIOP* framework are presented. In particular, these are a DE, a PSO, a ABS, a GA and a SA implementation. In order to simulate a real operational scenario of the *KUKA* robot using position control, a set of points that defines a possible trajectory for the end-effector was chosen. In this operational scenario, couples of adjacent points do not differ much from each other statistically. Moreover, the algorithms have no more than a 50 ms time frame to produce a solution in order to maintain a real-time control scenario. Starting from the beginning of the second solution, the initial population of the respective algorithms is injected with a fraction of the previously best found candidate solutions, which is a feature of the *MLAlgorithm* and mimics elitism in GAs. Fig. 3, 4, 5, 6 and 7 shows a plot of the cost versus time of the different algorithms. The data used to produce the plots is gathered from the respective algorithms' *MLHistory* and saved as a text file, using the *dumpDataTofile()* function, and then plotted using MATLAB. Table VIII shows the resulting position and orientation of the end-effector. It is clear that the machine learning algorithms implemented in *JIOP* and presented in this paper are able to find the solution to the given problem quickly and accurately. It should be noted that a cost of zero is not possible because of the third component in the cost function, given by (4). Furthermore, Fig. 2 shows the resulting poses of the calculations.

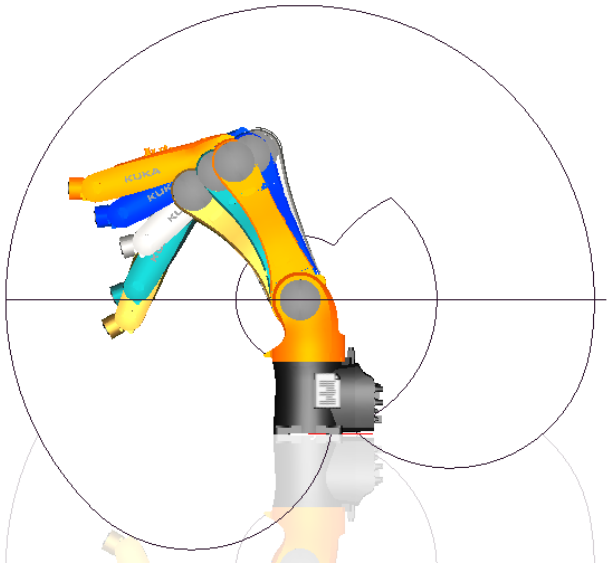

Fig. 2. Visualisation of the five resulting *KUKA* poses, found using the *JIOP* framework.

The computation was done on a computer running Windows 7 with a quad core Intel Core i7-3820QM processor. The result of utilising the processor's multiple threads is given in Table VII, and shows the total number of iterations that the different algorithms were able to complete in the given time-frame. It is clear that utilising multiple threads is highly beneficial to the performance of the algorithms. Note that the Simulated Annealing implementation runs on a single thread. The result for four threads is therefore undefined.

TABLE VII. MULTI-THREADING PERFORMANCE

Algorithms	Iterations	
	1 thread	4 threads
Differential Evolution	1508	4229
Particle Swarm Optimisation	1095	3375
Artificial Bee Colony	871	2181
Genetic Algorithm	581	1852
Simulated Annealing	42305	-

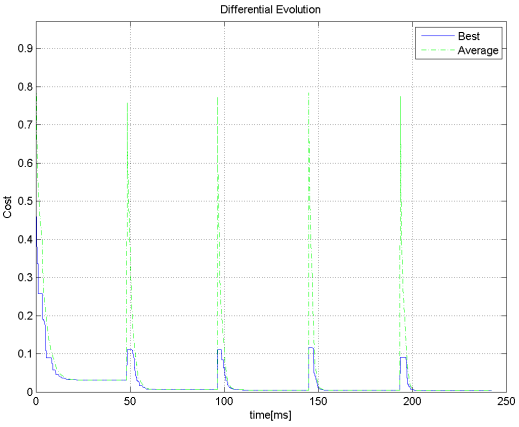

Fig. 3. Cost versus time using the Differential Evolution algorithm from the *JIOP* framework solving five consecutive IK solutions

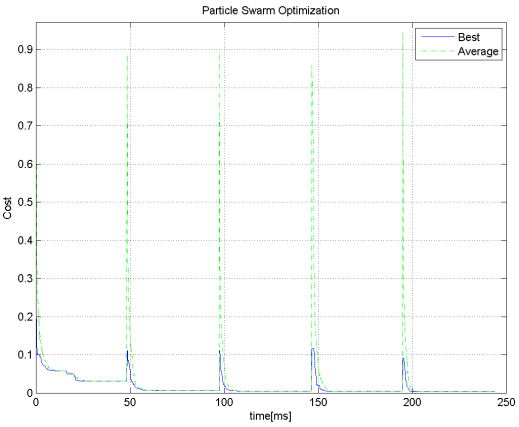

Fig. 4. Cost versus time using the Particle Swarm Optimisation algorithm from the *JIOP* framework solving five consecutive IK solutions

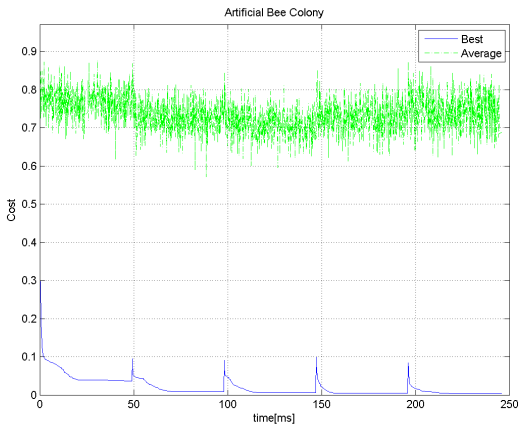

Fig. 5. Cost versus time using the Artificial Bee Colony algorithm from the *JIOP* framework solving five consecutive IK solutions

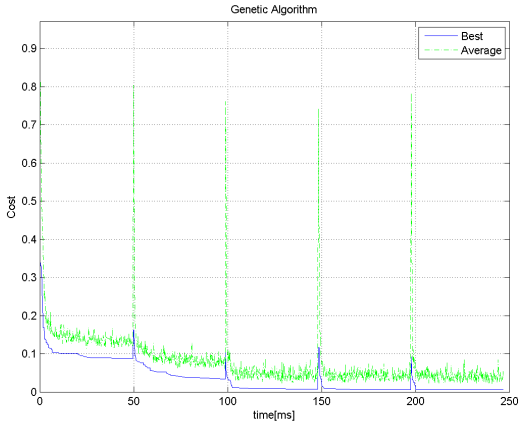

Fig. 6. Cost versus time using the Genetic Algorithm from the *JIOP* framework solving five consecutive IK solutions

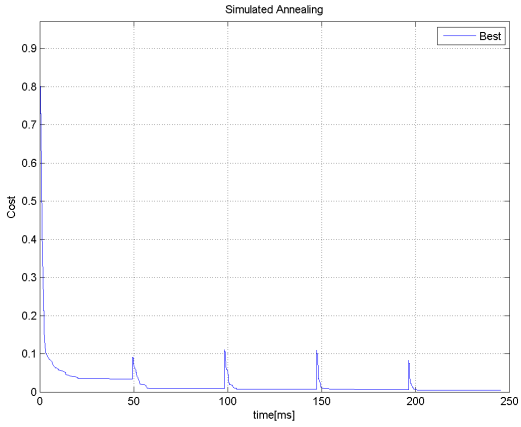

Fig. 7. Cost versus time using the Simulated Annealing from the *JIOP* framework solving five consecutive IK solutions

TABLE VIII. SIMULATION RESULTS

	Position $[m]$	Orientation$[deg]$	
Desired	$[0.800, 0.000, 0.600]$	$[0.000, 0.000, 0.000]$	
	$[0.700, 0.000, 0.500]$	$[0.000, 7.500, 0.000]$	
	$[0.600, 0.000, 0.400]$	$[0.000, 15.000, 0.000]$	
	$[0.600, 0.000, 0.250]$	$[0.000, 22.500, 0.000]$	
	$[0.600, 0.000, 0.150]$	$[0.000, 30.000, 0.000]$	Cost
DE	$[0.800, -0.000, 0.600]$	$[0.000, 0.000, -0.000]$	0,0314
	$[0.700, 0.000, 0.500]$	$[-0.000, 7.500, -0.000]$	0,0066
	$[0.600, -0.000, 0.400]$	$[-0.000, 15.000, -0.000]$	0,0052
	$[0.600, 0.000, 0.250]$	$[-0.000, 22.500, -0.000]$	0,0044
	$[0.600, -0.000, 0.150]$	$[-0.000, 30.000, -0.000]$	0,0030
PSO	$[0.800, -0.000, 0.600]$	$[0.000, -0.000, 0.000]$	0,0314
	$[0.700, 0.000, 0.500]$	$[0.000, 7.500, -0.000]$	0,0066
	$[0.600, 0.000, 0.400]$	$[-0.000, 15.000, -0.000]$	0,0052
	$[0.600, -0.000, 0.250]$	$[0.000, 22.500, -0.000]$	0,0044
	$[0.600, 0.000, 0.150]$	$[0.000, 30.000, 0.000]$	0,0030
ABS	$[0.801, -0.000, 0.601]$	$[-0.130, -0.136, -1.236]$	0,0370
	$[0.700, 0.000, 0.500]$	$[-0.048, 7.489, -0.076]$	0,0083
	$[0.600, 0.000, 0.400]$	$[0.001, 15.027, -0.021]$	0,0055
	$[0.600, 0.000, 0.250]$	$[-0.002, 22.513, -0.014]$	0,0046
	$[0.600, 0.000, 0.150]$	$[0.015, 29.987, -0.004]$	0,0031
GA	$[0.881, 0.019, 0.667]$	$[-0.031, 0.031, -0.025]$	0,0887
	$[0.702, 0.006, 0.501]$	$[-0.066, 7.591, -0.357]$	0,0332
	$[0.600, 0.000, 0.400]$	$[-0.093, 14.950, -0.273]$	0,0071
	$[0.600, 0.002, 0.252]$	$[-0.011, 22.544, -0.055]$	0,0061
	$[0.599, 0.002, 0.155]$	$[-0.008, 30.002, -0.007]$	0,0054
SA	$[0.801, 0.003, 0.602]$	$[-0.178, 0.006, -0.010]$	0,0340
	$[0.700, 0.001, 0.501]$	$[-0.419, 7.522, -0.097]$	0,0086
	$[0.601, 0.001, 0.401]$	$[0.238, 15.006, -0.081]$	0,0070
	$[0.601, -0.001, 0.248]$	$[0.131, 22.577, 0.054]$	0,0064
	$[0.600, 0.001, 0.151]$	$[-0.143, 29.871, 0.114]$	0,0046

CONCLUSION AND FUTURE WORK

JIOP demonstrates an object-oriented machine learning framework to anyone with an interest in machine learning algorithms and the Java programming language. This is especially true for students that are in a need of a compact, easy to use and highly configurable framework that also provides visual feedback. In this case, *JIOP* delivers a effective and lightweight environment for using and creating machine learning algorithms. A continuous effort will be made to streamline and expand the framework with even more algorithms and configuration options.

REFERENCES

Aarts, E. & Korst, J. (1988), 'Simulated annealing and boltzmann machines'.

Abeel, T., Van de Peer, Y. & Saeys, Y. (2009), 'Java-ml: A machine learning library', *The Journal of Machine Learning Research* **10**, 931–934.

Alpaydin, E. (2004), *Introduction to machine learning*, MIT press.

Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002), 'A fast and elitist multiobjective genetic algorithm: Nsga-ii', *IEEE Transactions on Evolutionary Computation* **6**(2), 182–197.

Evgeniou, T. & Pontil, M. (2004), Regularized multi–task learning, *in* 'Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining', ACM, pp. 109–117.

Heaton, J. & Reasearch, H. (2010), 'Encog java and dotnet neural network framework', *Heaton Research, Inc., Retrieved on July* **20**, 2010.

Hormozi, H., Hormozi, E. & Nohooji, H. R. (2012), 'The classification of the applicable machine learning methods in robot manipulators', *International Journal of Machine Learning and Computing* **2**, 560–563.

Karaboga, D. & Basturk, B. (2007), 'A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm', *Journal of global optimization* **39**(3), 459–471.

Kennedy, J. (2010), Particle swarm optimization, *in* 'Encyclopedia of Machine Learning', Springer, pp. 760–766.

Kohavi, R., John, G., Long, R., Manley, D. & Pfleger, K. (1994), Mlc++: A machine learning library in c++, *in* 'Proceedings of the Sixth International Conference on Tools with Artificial Intelligence', IEEE, pp. 740–743.

Kotsiantis, S. B., Zaharakis, I. & Pintelas, P. (2007), 'Supervised machine learning: A review of classification techniques'.

Pluhacek, M., Senkerik, R., Zelinka, I. & Davendra, D. (2013), Multiple choice strategy for pso algorithm - performance analysis on shifted test functions., *in* 'ECMS', European Council for Modeling and Simulation, pp. 393–397.

Sanfilippo, F., Hatledal, L. I., Schaathun, H. G., Pettersen, K. Y. & Zhang, H. (2013), A universal control architecture for maritime cranes and robots using genetic algorithms as a possible mapping approach, *in* 'Proceeding of the IEEE International Conference on Robotics and Biomimetics (ROBIO) Shenzhen, China, December 2013', IEEE, pp. 322–327.

Simon, P. (2013), *Too Big to Ignore: The Business Case for Big Data*, Wiley. com.

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

Zhu, X. (2006), 'Semi-supervised learning literature survey', *Computer Science, University of Wisconsin-Madison* **2**, 3.

AUTHOR BIOGRAPHIES

LARS IVAR HATLEDAL received a bachelor's degree in Automation from Aalesund University College, Norway. Here, Hatledal joined the Department of Maritime Technology and Operations as a Project Leader in June 2013.
Email: laht@hials.no

FILIPPO SANFILIPPO is a PhD candidate in Engineering Cybernetics at the Norwegian University of Science and Technology, and a research assistant at the Department of Maritime Technology and Operations, Aalesund University College, Norway. He obtained his Master's Degree in Computer Engineering at the University of Siena, Italy.
Email: fisa@hials.no

HOUXIANG ZHANG received Ph.D. degree in Mechanical and Electronic Engineering in 2003. From 2004, he worked as Postdoctoral Fellow at the Institute of Technical Aspects of Multimodal Systems (TAMS), Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, University of Hamburg, Germany. Dr. Zhang joined the Department of Maritime Technology and Operations, Aalesund University College, Norway in April 2011, where he is a Professor in Robotics and Cybernetics.
Email: hozh@hials.no