UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1666

A SOFTWARE FRAMEWORK FOR INTERACTIVE VISUALIZATION OF OPTIMIZATION ALGORITHMS

Marija Kalebota Kodžoman

Zagreb, June 2018

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 15 March 2018

MASTER THESIS ASSIGNMENT No. 1666

Student:	**Marija Kalebota Kodžoman (0036472084)**
Study:	Computing
Profile:	Computer Science

Title:	**A Software Framework for Interactive Visualization of Optimization Algorithms**

Description:

Describe the most common deterministic optimization algorithms, taking into account the number of variables, the availability of the gradient of the function and the presence of constraints. Develop the structure of a software system for implementing optimization algorithms of choice for arbitrary target functions. Develop the software system using different modules for optimization, for displaying the steps and results of the algorithms and for the user interface. Implement the functionality for optimization of functions of several variables with the possibility of using gradients and defining additional constraints. Prepare examples that demonstrate the functionalities of the implemented software. In addition to the thesis, provide the source code of the software, the obtained results with additional explanations and list the references.

Issue date:	16 March 2018
Submission date:	29 June 2018

Mentor:

[illegible]

Full Professor Domagoj Jakobović, PhD

Committee Secretary:

[illegible]

Assistant Professor Tomislav Hrkać, PhD

Committee Chair:

[illegible]

Full Professor Siniša Srbljić, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 15. ožujka 2018.

DIPLOMSKI ZADATAK br. 1666

Pristupnik:	**Marija Kalebota Kodžoman (0036472084)**
Studij:	Računarstvo
Profil:	Računarska znanost

Zadatak: **Programski okvir za interaktivnu vizualizaciju algoritama optimizacije**

Opis zadatka:

Opisati klasične postupke determinističke optimizacije s obzirom na broj varijabli, dostupnost gradijenta i postojanje ograničenja. Osmisliti strukturu programskog sustava za ostvarenje proizvoljnog postupka optimizacije uz proizvoljnu funkciju cilja. Ostvariti programski sustav korištenjem zasebnih modula za optimizaciju, prikaz rada, prikaz rezultata i komunikaciju s korisnikom. Ostvariti funkcionalnost optimizacije funkcija više varijabli uz mogućnost korištenja gradijenta i postojanje ograničenja. Izraditi primjere korištenja navedenih postupaka uz potrebne upute. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku:	16. ožujka 2018.
Rok za predaju rada:	29. lipnja 2018.

Mentor:

[illegible]

Prof. dr. sc. Domagoj Jakobović

Djelovođa:

[illegible]

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

[illegible]

Prof. dr. sc. Siniša Srbljić

I would like to thank my mentor Domagoj Jakobović for facilitating an inviting environment for the pursuit of knowledge.

I would also like to thank Marko Đurasević for always having the right answer to all of my questions.

Finally, I would like to thank my family for their continuous support and encouragement. My accomplishments would not have been possible without them.

Thank you.

Table of contents

1. Introduction

Optimization algorithms are widely used in various engineering applications, such as machine learning and scheduling. Some algorithms work better than others on certain optimization problems, but it can be difficult to understand why that is, especially for more complex methods. Understanding how optimization methods work can be significantly eased by using visualization techniques. Visualizations provide a more abstract view of an optimization method, which can allow users to focus more on understanding its underlying idea.

The purpose of this thesis is to develop a modular software framework which would provide both students and educators with an interactive and practical way of visualizing how optimization algorithms iterate through possible solutions during their execution. It should be easy to extend the framework with new components, so that users can experiment with various new algorithms and optimization problems.

The first chapter contains an overview of basic terms which are commonly encountered when dealing with optimization. The second chapter describes some of the technologies used to develop the framework. The third chapter describes the implementation of the framework in more detail. The fourth chapter offers practical examples of how the framework can be used, and the fifth chapter shows how new modules can be added to it. The sixth, and final, chapter offers a brief conclusion.

2. Optimization

As this thesis primarily deals with the optimization of mathematical functions, the following chapters offer theoretical overview of most commonly used terms.

2.1 Mathematical optimization problems

In mathematics, a *minimization* problem is the problem of *minimizing* a function – finding the point x at which the value of the function $f(x)$ reaches its lowest possible value – its *minimum* [1]. Similarly, a *maximization* problem is the problem of *maximizing* a function – finding the point x in which the value of the function $f(x)$ reaches its highest possible value – its *maximum*. When referring to either of these cases in their respective contexts, the terms *optimum* and *optimization* are also used. In these problems, the function $f(x)$ is called the *objective function*, because it is usually a representation of a practical problem – an objective that needs to be reached. The objective is reached by finding that function's optimum, which is achieved through the process of optimization.

Optimization problems are conventionally posed as minimization problems, but the same methods of solving them also apply to maximization problems. In order to follow convention, one can easily pose a maximization problem as a minimization problem by negating the objective function (using $-f(x)$ instead of $f(x)$). Therefore, all future references to optimization will be describing minimization, but it should be noted that they all equally apply to maximization.

The simplest form of an optimization problem is:

$$\underset{x}{\text{minimize}}\, f(x) : \mathbb{R}^n \to \mathbb{R}$$

In this case, the whole domain of the function $f(x)$ is the *feasible area* for finding the solution. That means that the minimum of the function could be found at any possible point x.

2.1.1 Unimodality

One important property of functions is whether they are *unimodal* or *multimodal*. A function is unimodal if it only has a single local optimum. If more local optima exist, it is multimodal. Figure 1 shows the graph of a unimodal function, $f(x) = (x - 3)^2$. This function has only one minimum which is located at the point $x = 3$.

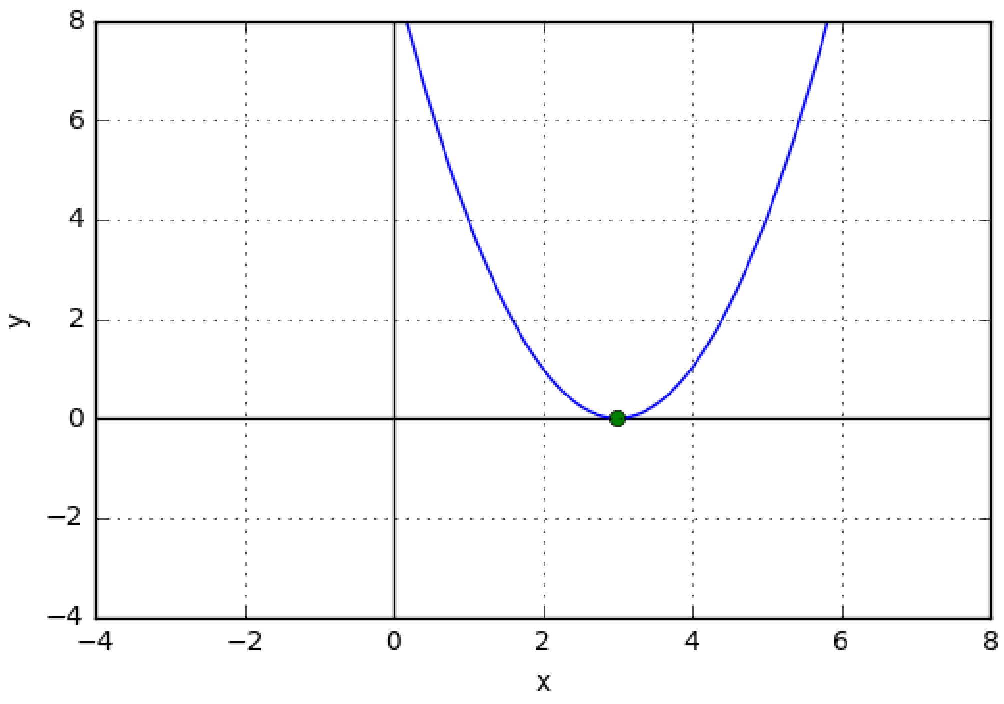


Figure 1 - The graph of the function $f(x) = (x - 3)^2$. The green dot represents the global optimum.

Figure 2 shows the function $f(x) = x^4 - x^2 + \frac{x}{10}$, which has two local optima but only one global optimum, and Figure 3 shows the function $f(x) = x * cos(x)$ which has many local optima, but no global optimum. Both of these functions are multimodal.

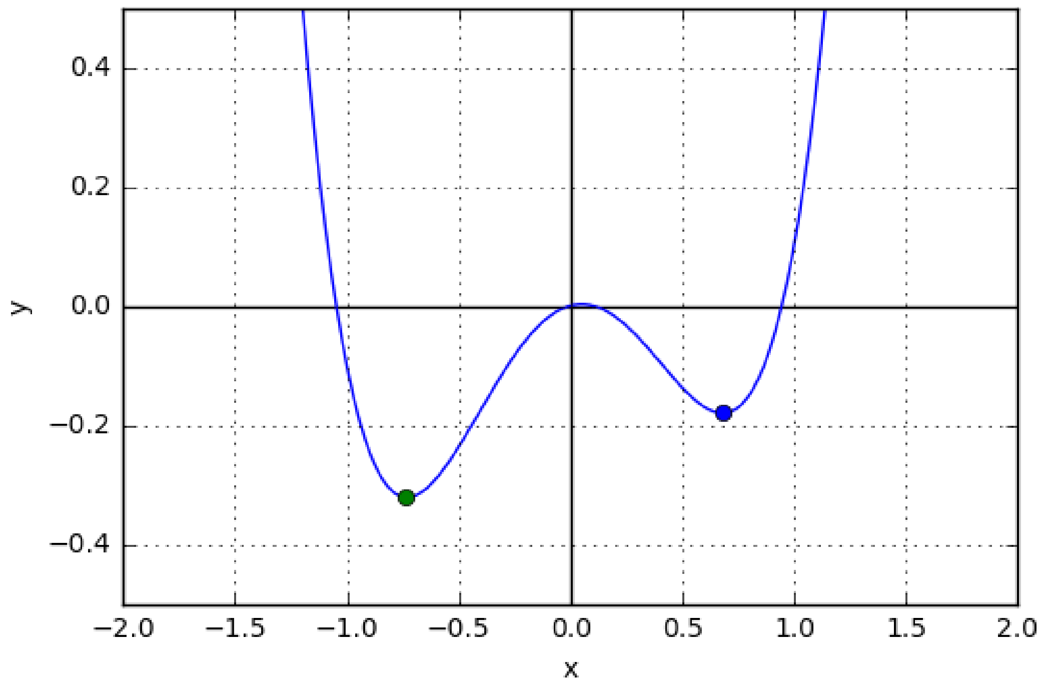


Figure 2 - The graph of the function $f(x) = x^4 - x^2 + \frac{x}{10}$*. The blue dot represents a local optimum, while the green dot represents the global optimum.*

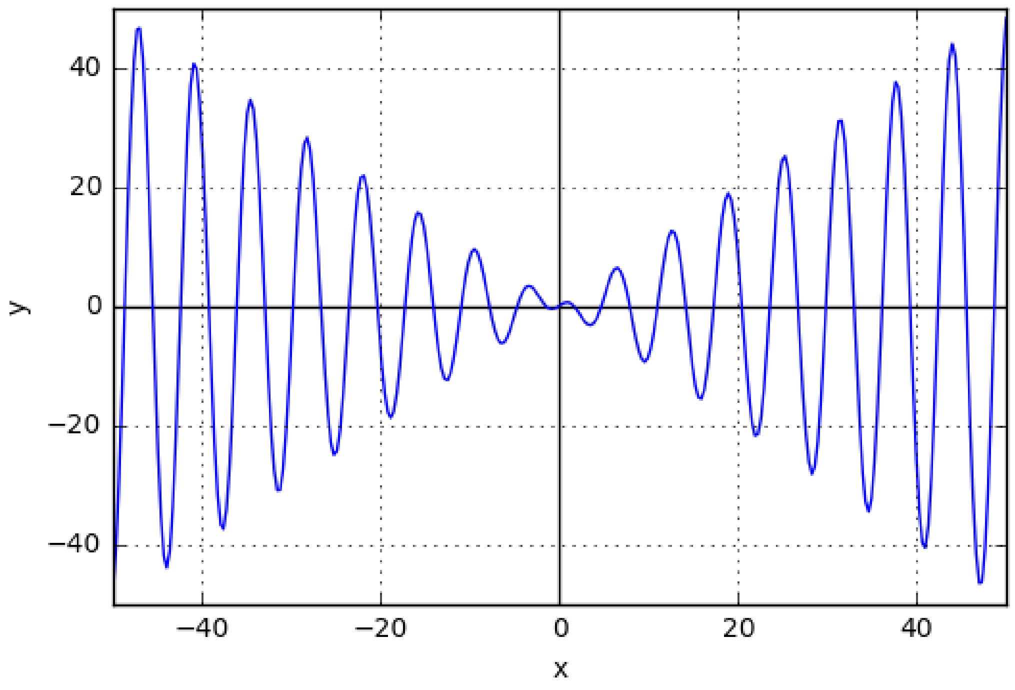


Figure 3 - The graph of the function $f(x) = x * cos(x)$

The modality of a function significantly impacts the ways in which its optimum can be found. Unimodal functions can usually be optimized much more easily than their multimodal counterparts. This is because the presence of local optima can lead optimization algorithms to mistakenly resolve that they have found the global optimum. In many such cases, it is difficult or even impossible to know whether the solution found by the algorithm is indeed the best one. Therefore, solving optimization problems with multimodal functions is usually quite challenging, and more advanced optimization algorithms need to be developed as a result.

2.1.2 Constraints

Conditions which can restrict the feasible area to a smaller subset of the domain can also be added to the problem. These conditions are called *constraints*. There are two types of constraints – *explicit* constraints and *implicit* constraints.

Explicit constraints are defined as:

$$\vec{x}_l \leq \vec{x} \leq \vec{x}_u$$

Therefore, explicit constraints simply define an upper and lower bound to the values that point x can become. They constrain the domain from $\mathbb{R}$ to $[\vec{x}_l, \vec{x}_u]$.

Implicit constraints can be divided into two groups – *inequality* implicit constraints and *equality* implicit constraints. They are defined in the following way:

$g_i(x) \leq 0$ – inequality constraints

$h_j(x) = 0$ – equality constraints

They constrain the feasible area *implicitly*, by posing conditions on certain dimensions of the point x, and stating that they should either equal zero (equality constraints) or that they should be less than or equal to zero (inequality constraints).

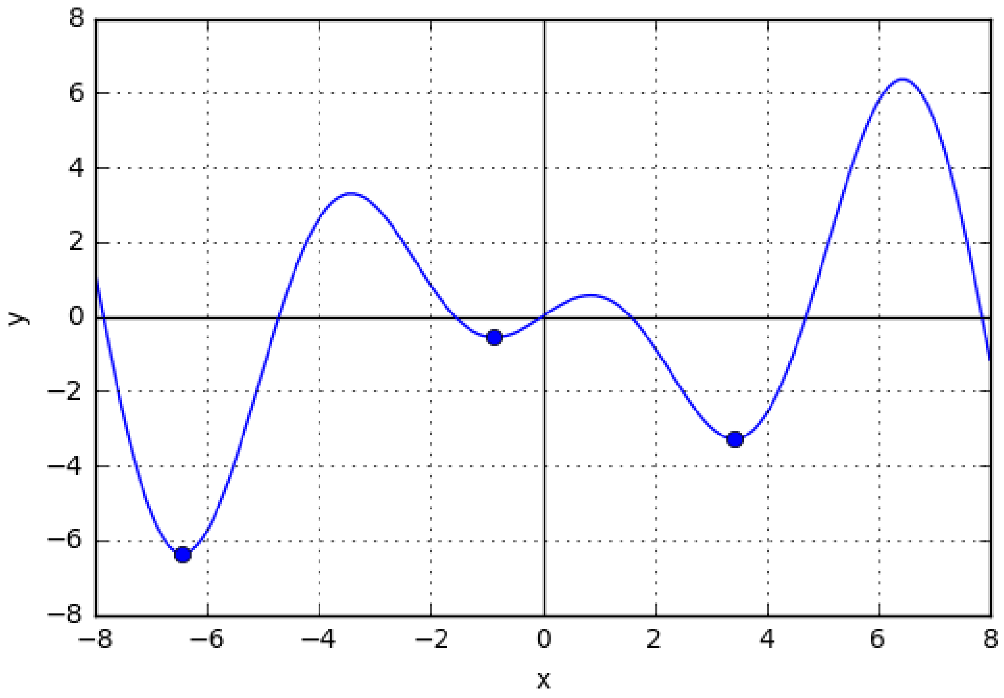


Figure 4 - The graph of the function $f(x) = x * cos(x)$. The blue dots represent local optima.

For example, let the objective function be $f(x) = x * cos(x)$ (Figures 3 and 4). By adding the explicit constraint

$$-4 \leq x \leq 4$$

to the problem, the feasible area is reduced to the interval $[-4,4]$, and the objective function can only obtain the values shown in blue in Figure 5. The optimization problem did not have a global optimum before, but one does exist when it is constrained in this way (marked with a green dot in Figure 5).

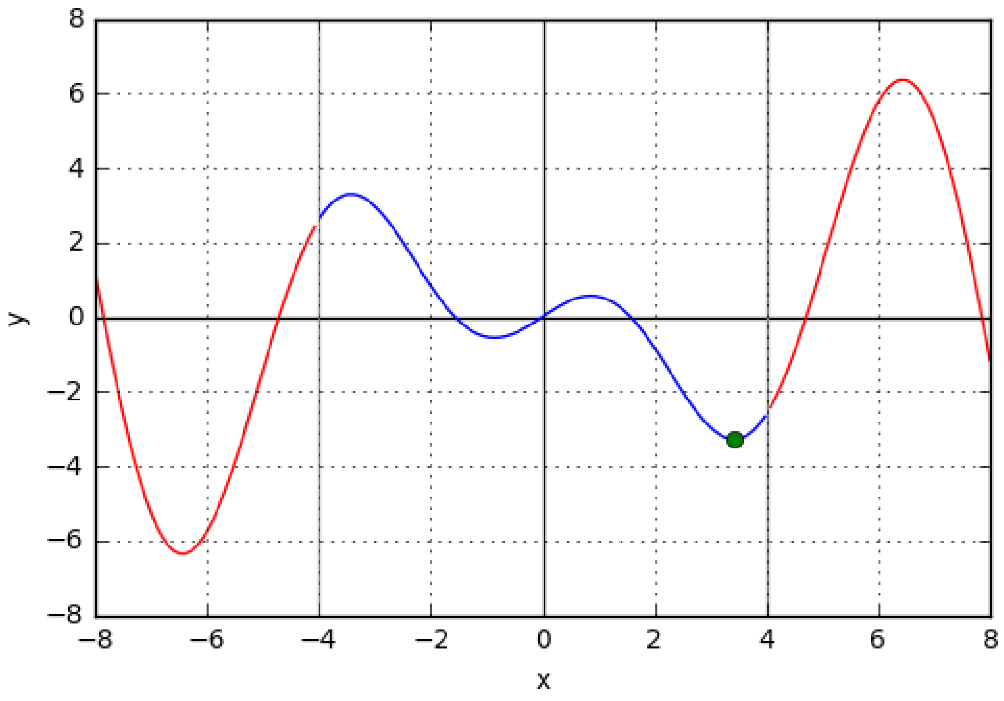


Figure 5 - The graph of the function $f(x) = x * cos(x)$) with an explicit constraint

By adding the inequality implicit constraint

$$x^2 - 4 \leq 0,$$

the feasible area is reduced further, to the interval $[-2,2]$, as shown in Figure 6. It is interesting to note that the global optimum is now at the edge of the feasible area – this is a common occurrence with constrained optimization problems.

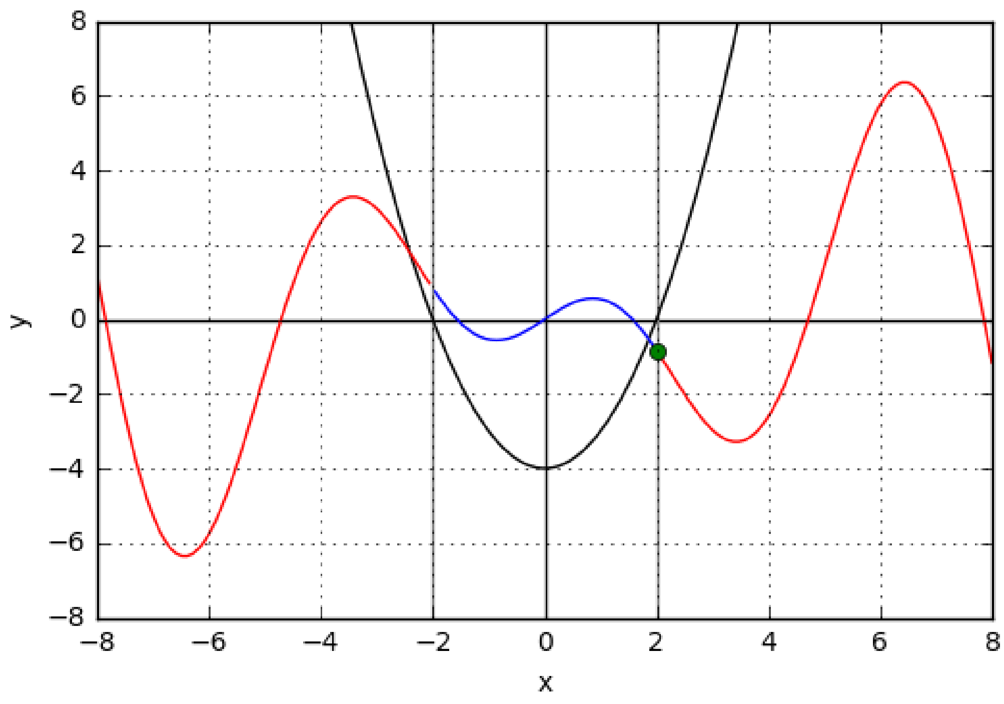


Figure 6 - The graph of the function $f(x) = x * cos(x)$ *with the implicit constraint* $x^2 - 4 \leq 0$

Finally, by adding the equality implicit constraint

$$x^2 - \frac{\pi^2}{4} = 0,$$

the feasible area is reduced to only two points, as shown in Figure 7.

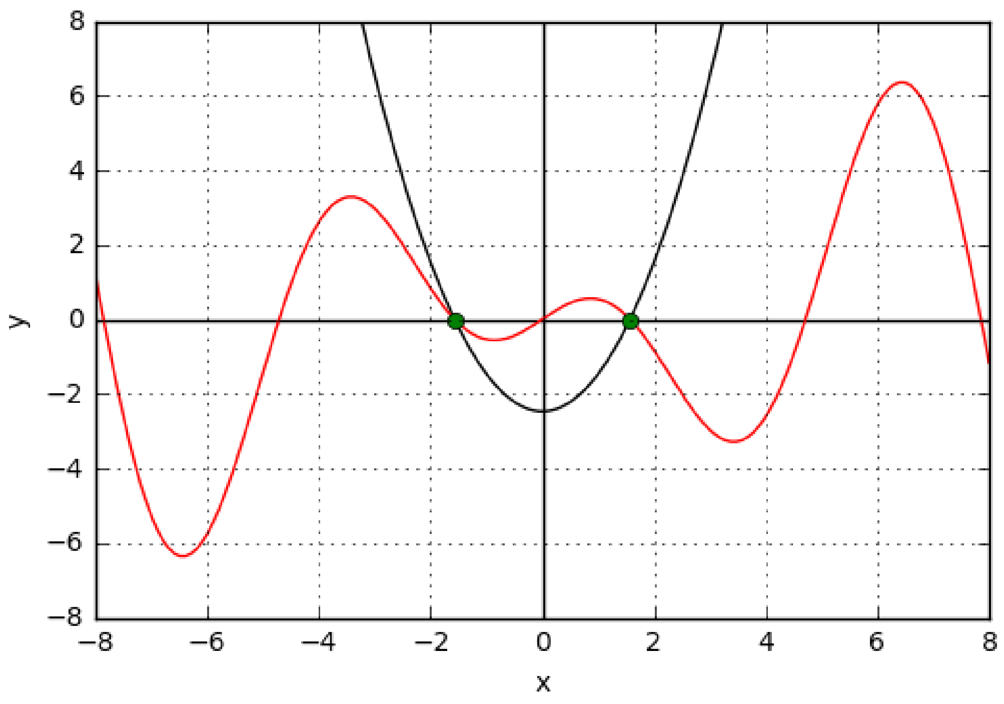


Figure 7 - The graph of the function $f(x) = x * cos(x)$ *with the implicit constraint* $x^2 - \frac{\pi^2}{4} = 0$

2.2 Optimization algorithms

Methods for solving optimization problems are called *optimization algorithms*. One of the most significant challenges surrounding optimization problems is dealing with the fact that their domain is $\mathbb{R}^n$, a set of infinite cardinality. Because of this, the number of potential solutions to a problem is infinite, so it is very difficult to find the best one. Humans may be able to quickly find the minimum of $f(x) = (x-3)^2$, but functions in optimization problems are generally much more complex than that. With complex and time-consuming problems, it is only natural that we would utilize any tools available to us in order to solve them more quickly and more efficiently. The most convenient such tool available to us is the computer.

Even though computers could try out the possible solutions and compare the resulting function values significantly more quickly than humans, with a feasible area of infinite cardinality, they would still never be able to give a definite solution to the problem – there would always be another point at which to evaluate the function, to see if the value at that point is lower than any of the ones found previously.

Therefore, methods for solving optimization problems need to work in ways that find the best solution to the problem in a finite number of steps. Different algorithms work in different ways, but the goal of all of them is the same – to find the solution to the problem with as little evaluations of the objective function as possible.

No matter what type of problem an algorithm specializes for, most of them have several things in common: they try to find the solution to the problem iteratively (step by step), and after each iteration, there is a point that the algorithm believes to be the best solution. The algorithm continues to iterate until it reaches its termination criterion, which is different for every algorithm. Examples of termination criteria are:

- Exceeding a given number of iterations
- Not having found a significantly different solution for a given number of iterations
- Exceeding a given number of evaluations of the objective function

3. Technologies

The primary idea when developing the framework was for it to be used and shared easily and efficiently, both as a demonstration tool by educators and as an educational aid for students. Jupyter Notebooks were seen as the most fitting platform for fulfilling these objectives, because results of executing code can be viewed instantaneously when using them, and they can be opened in web browsers, so there are ways of using them that do not require any installation.

The framework was developed in the Python programming language. It was chosen because it is the programming language most commonly used with Jupyter Notebooks, and because it provides many existing modules which are particularly useful for visualization of optimization algorithms, like NumPy for data manipulation and Matplotlib for plotting graphs. Additionally, the module which was perhaps the most integral to achieving interactivity within the framework is ipywidgets. It offers a wide range of different *widget* elements - interactive graphical elements that can be used to interface with the framework.

3.1 Jupyter Notebooks

The Jupyter Notebook combines three components [2]:

1. **The notebook web application** - a web application for writing and running code interactively, and authoring notebook documents
2. **Kernels** - “computational engines” that execute the code contained in a Notebook document. The web application starts a new kernel for each open notebook document. There are kernels for many different programming languages, but the default one is for Python.
3. **Notebook documents** - documents that contain a representation of all content visible in the notebook web application They are simply files on the user’s computer which can be moved between folders or shared with others.

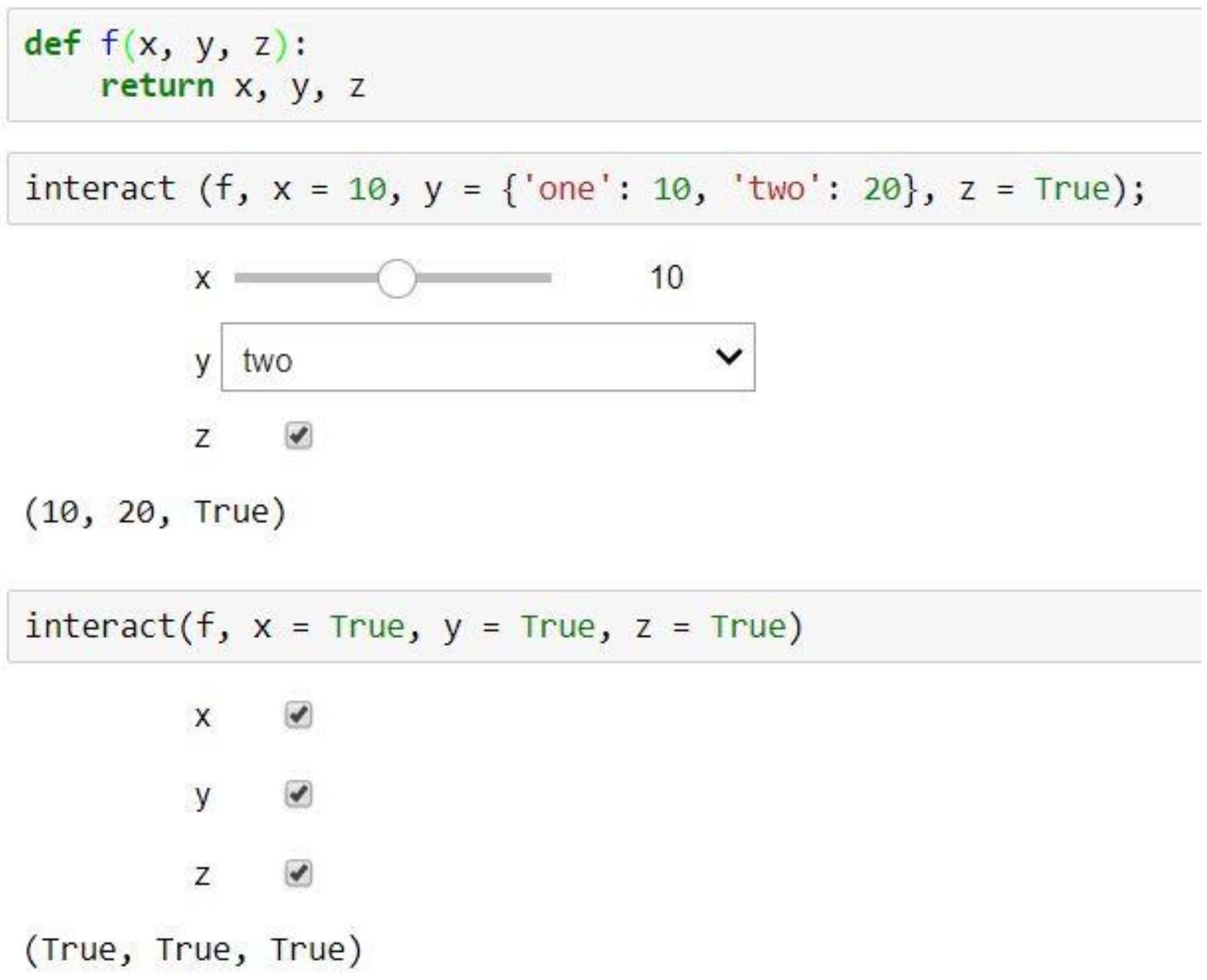
Jupyter Notebooks consist of a series of cells, and programming code contained inside can be executed cell-by-cell. The cells can output HTML, images, video and plots, and all generated output is saved with the notebook document, so it is a complete record of a computation.

3.2 Ipywidgets

The ipywidgets module offers many different ways of creating widgets, which can be used for adjusting the input of methods or functions. This is achieved using the *interact* method [3].

Its first argument is always an instance of a Python method or function. Depending on the types of the other arguments it is given, the *interact* method displays their corresponding widgets: checkboxes for boolean types, sliders for numbers, text boxes for strings, dropdown menus for dictionaries and so on. The values obtained from these interactive elements are used as arguments for the method which was given as the first argument, and the method is re-executed with new arguments whenever one of the displayed widgets is modified.

An example of this can be seen in Figure 8, where the *interact* method is used on the function *f* which takes three arguments. Depending on the types of arguments given to the *interact* method, different widgets are created for giving input to the same function. The first call to the interact method provides a slider, a dropdown menu and a checkbox. The second call, in which all the arguments are boolean values, provides three checkboxes.



```
def f(x, y, z):
    return x, y, z
```

```
interact (f, x = 10, y = {'one': 10, 'two': 20}, z = True);
```

```
(10, 20, True)
```

```
interact(f, x = True, y = True, z = True)
```

```
(True, True, True)
```

Figure 8 – The behaviour of the interact method

Widget objects can also be created independently and stored in variables, which allows for adjustments to their parameters. For example, executing the line

```
slider = widgets.IntSlider(min = 0, max = 20, value=10)
```

creates a slider with a selection range of [0,20] and an initial value of 10. This slider can then be passed as an argument to the *interact* method. When creating an interactive application, this gives the programmer a lot of control over the choices that a user can make.

4. Implementation

The framework was developed in Python, and the main focus when designing it was to maintain *modularity* and *readability*. It was not designed to be finished and never be edited again, as fewer and fewer programming projects are. Throughout the entire development process, there was a focus on adhering to the guidelines of clean code, and on developing a framework that is not only functional but also written in a way that could later be upgraded with ease.

Figure 9 shows the class diagram of the framework. The following is an overview of its most significant classes.

Point – a class representing a mathematical point of arbitrary dimensionality. The user sets the value of every dimension, and the class offers the functionality of addition, subtraction, multiplication, and operations applicable to mathematical points.

IFunction – a class designed to be inherited by implementations of specific mathematical functions. It offers the ability to evaluate the function at feasible points, to return the gradient and the Hessian at a certain point, and to keep a counter of the number of evaluations of the function.

Classes representing constraints

- **IConstraint** – a class designed to be inherited by implementations of specific mathematical constraints. It offers the ability to check whether the constraint is satisfied at a given point, and also to return the value or the gradient at that point.
- **IImplicitConstraint** – a class which inherits the IConstraint class, and which is inherited by the classes IEqualityImplicitConstraint and IInequalityImplicitConstraint
- **IEqualityImplicitConstraint** and **IInequalityImplicitConstraint** – classes that represent the two types of implicit constraints. They directly inherit the IImplicitConstraint class.

- **ExplicitConstraintForOneDimension** – a class that represents an explicit constraint for one dimension. It *does not* inherit the IConstraint class due to the significant differences in the nature of these two types of constraints.

Logger – a class designed to log all the iterations of an optimization algorithm. This log contains generic information like the current solution and the value of the objective function at the current solution, but it can also contain any information specific to the algorithm. It was designed to hold all the relevant information about the execution of the algorithm, so that its behaviour can easily be reconstructed and visualised.

Iteration – a class designed to log all the relevant information about an iteration of an algorithm's execution. The Logger holds, among other things, a list of instances of the Iteration class.

IAlgorithm – a class designed to be inherited by a specific implementation of an optimization algorithm. When it runs, a complete log of all its iterations is created, and stored in an instance of the Logger class.

Drawer – a class designed to plot static graphs based on different mathematical elements like points, functions, and constraints. It offers the functionality of plotting two-dimensional plots, contour plots and three-dimensional plots.

Animator – a class designed to create all the interactive elements of the visualization, like checkboxes, sliders, text boxes or different kinds of buttons. The Presenter uses methods from this class to create the elements it needs in order to present the final visualization to the user.

Presenter – a class designed to present the final interactive visualization of an algorithm's execution to the user. It is responsible for interpreting the data stored in the Logger and presenting it to the user by utilizing the capabilities of the Drawer and Animator classes.

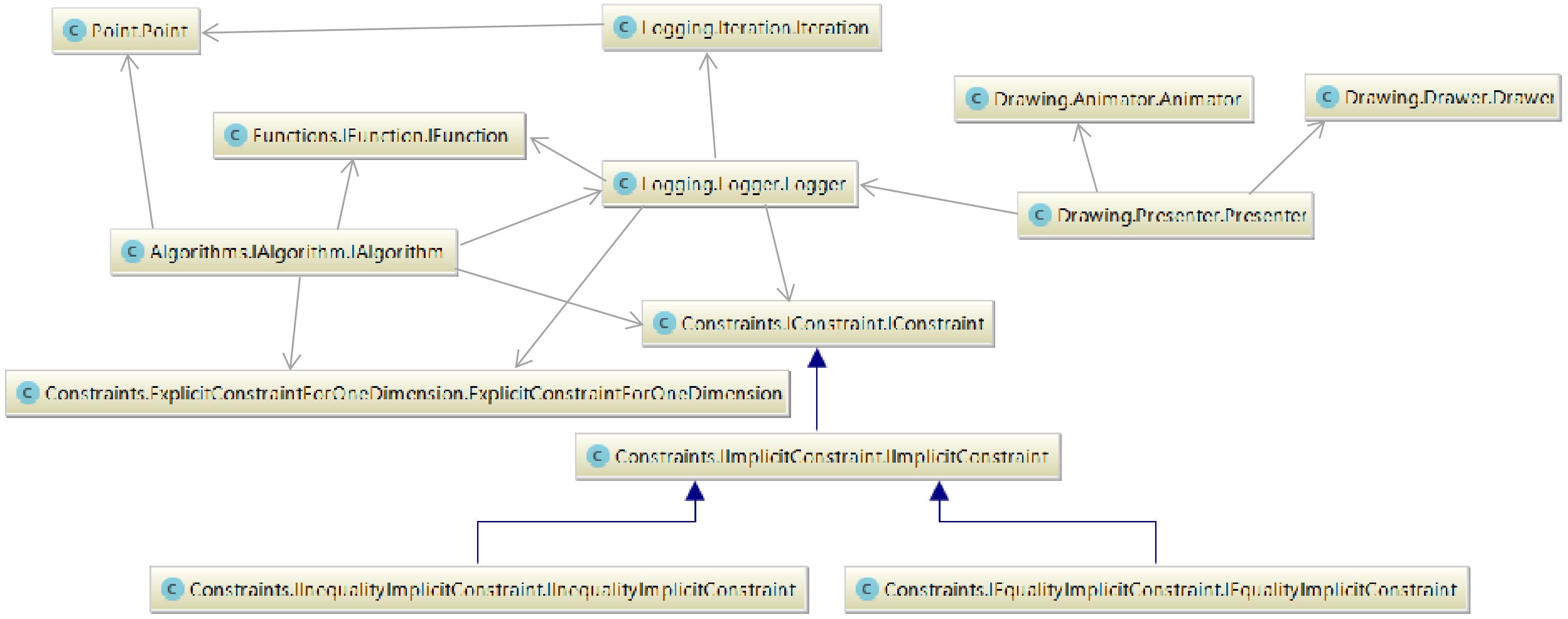

Point.Point
Logging.Iteration.Iteration
Drawing.Animator.Animator
Drawing.Drawer.Drawer
Functions.IFunction.IFunction
Logging.Logger.Logger
Drawing.Presenter.Presenter
Algorithms.IAlgorithm.IAlgorithm
Constraints.IConstraint.IConstraint
Constraints.ExplicitConstraintForOneDimension.ExplicitConstraintForOneDimension
Constraints.IImplicitConstraint.IImplicitConstraint
Constraints.IInequalityImplicitConstraint.IInequalityImplicitConstraint
Constraints.IEqualityImplicitConstraint.IEqualityImplicitConstraint

Figure 9 – The class diagram of the framework

5. Using the framework

The framework was designed to be used to visualize the way optimization algorithms behave when optimizing different mathematical functions. The following chapters offer an overview of the currently available use cases.

5.1 One-dimensional functions

This example will show how the framework can be used to visualize the behaviour of an algorithm solving an optimization problem in which the objective function is one-dimensional. The algorithm used in the example is the Hooke-Jeeves algorithm, and the objective function is $f(x) = (x-3)^2$.

The user must first import all the necessary modules of the framework:

```
from FIVOA import *
from FIVOA.Functions import *
from FIVOA.Constraints import *
from FIVOA.Algorithms import *
from FIVOA.Drawing import *
```

Next, an instance of the class *IFunction*, which will represent the objective function of the problem, needs to be created. The framework already contains a class that inherits IFunction and represents the function $f(x) = (x-3)^2$ – it is called *F3OneDimensional*. Therefore, the following line should be executed:

```
function = F3OneDimensional.F3OneDimensional()
```

After creating the function, the user needs to create an instance of the class *IAlgorithm* which will represent the algorithm that will be used to solve the problem. The framework contains the class *HookeJeeves* which inherits the class *IAlgorithm* and implements the behaviour of the Hooke-Jeeves optimization algorithm. In order to create it, the user should execute the following line:

```
hj_algorithm = HookeJeeves.HookeJeeves(function = function,
step = 1, factor = 0.1, epsilon = 1E-6)
```

The parameters `step`, `factor` and `epsilon` are specific to the algorithm, and they are presented here with example values. Other algorithms would have different parameters.

The Hooke-Jeeves algorithm needs to be given an initial point from which to start its search. A representation of the point $x = 10$ is created by executing the following line:

```
point = Point.Point(elements = [10])
```

Finally, the algorithm can be run using the following line:

```
solution_hj, logger_hj = hj_algorithm.run(point)
```

The results of running the algorithm are the solution to the optimization problem (`solution_hj`), which is an instance of the *Point* class, and a complete log of its iterations saved in an instance of the *Logger* class (`logger_hj`).

Now that the log has been created, it can be used to create an interactive visualization of the steps that the algorithm took.

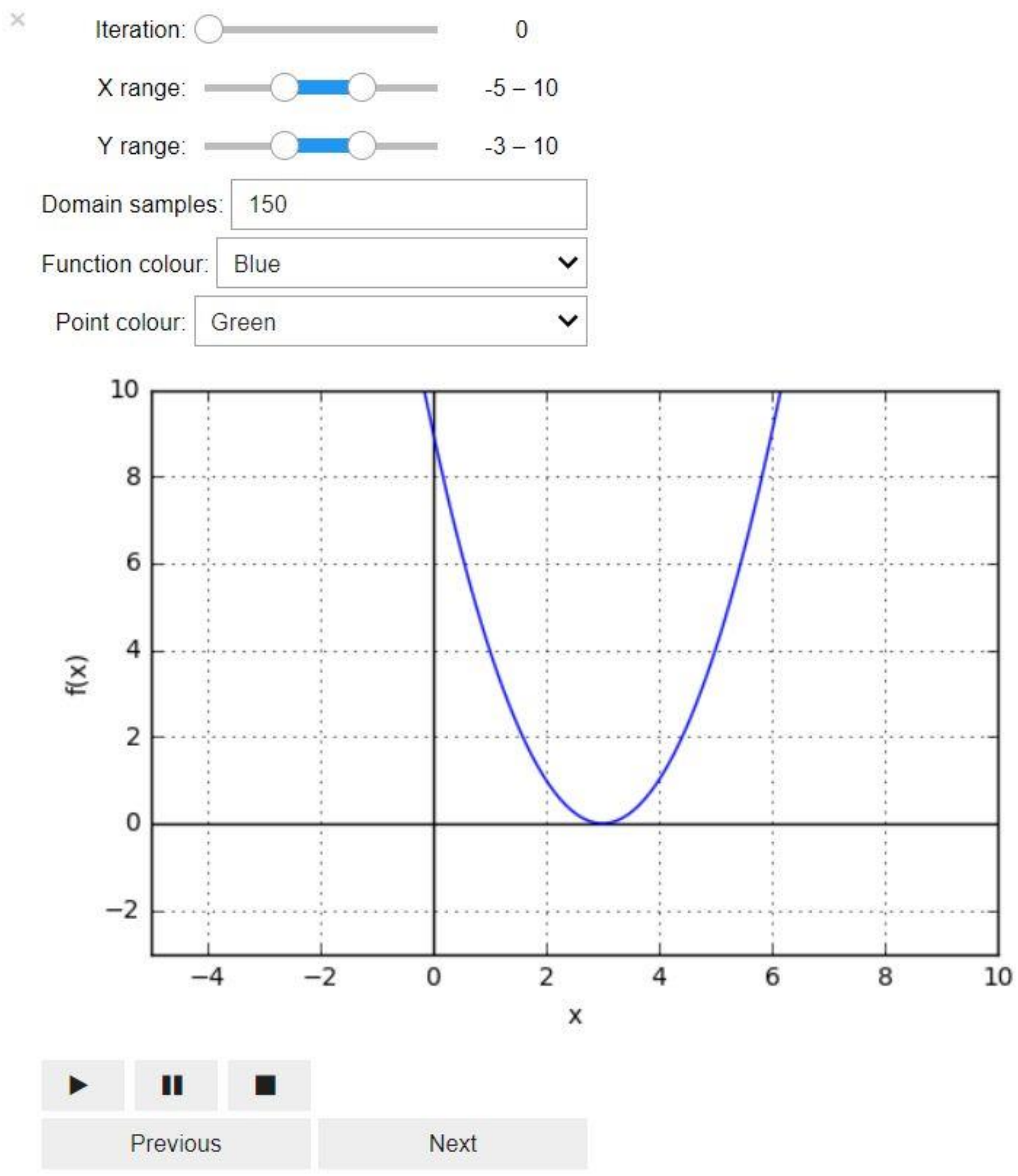
The *Presenter* is the class designed to create the interactive visualizations. Upon its creation, in addition to the logger, it also needs to be given instances of the *Drawer* and *Animator* classes:

```
presenter = Presenter.Presenter(logger = logger_hj, drawer = 
Drawer.Drawer(), animator = Animator.Animator())
```

Finally, the user should execute the line

```
presenter.present_2D()
```

in order to be presented with all the elements needed to achieve an interactive visualization, as shown in Figure 10.


Iteration: 0

X range: -5 – 10

Y range: -3 – 10

Domain samples: 150

Function colour: Blue

Point colour: Green

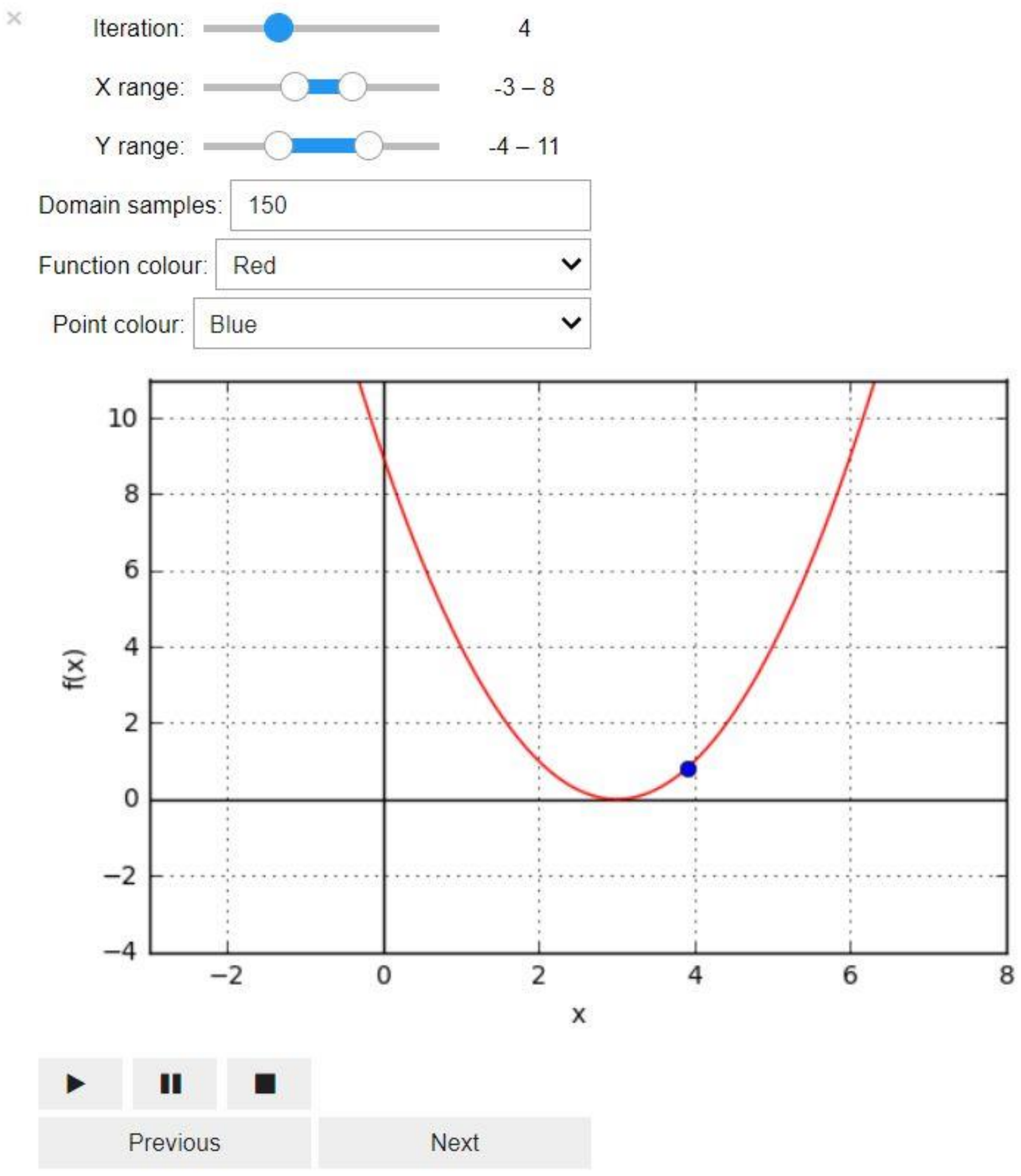
Figure 10 - Elements created by the Presenter for an interactive visualization of an algorithm for optimizing a one-dimensional function

The user is presented with the following options:

- To start, pause and stop an animation which iterates through all the steps of the algorithm, showing the solution in each of them

- To go through all the steps manually, by either dragging the slider which represents the iteration number, or by clicking the “Previous” and “Next” buttons
- To choose the area displayed by the graph by dragging the “X range” or “Y range” sliders
- To choose the accuracy of the graph by typing in the number of the samples of the domain which are used to plot the function
- To choose the colours of the function and the point marking the solution at the chosen step

After interacting with some of the given elements, the graph could change to the one shown in Figure 11.



Iteration: 4

X range: -3 – 8

Y range: -4 – 11

Domain samples: 150

Function colour: Red

Point colour: Blue

Figure 11 – The elements created by the Presenter for an interactive visualization of an algorithm for optimizing a one-dimensional function, after some interaction

5.2 Two-dimensional functions

When the objective function is two-dimensional, two possible visualizations of it are contour plots and three-dimensional plots. The following examples will show how the framework can be used to visualize the behaviour of an algorithm used to solve a constrained optimization problem with a two-dimensional objective function, using those two visualization methods.

The optimization algorithm used is the Box algorithm. The objective function of the optimization problem is $f(x) = (1 - x_1)^2 + 100 * (x_2 - {x_1}^2)^2$ (Rosenbrock's function). The explicit constraints are

$$x_1 \in [-100, 100]$$

$$x_2 \in [-100, 100]$$

and the implicit constraints are

$$x_2 - x_1 \geq 0$$

$$2 - x_1 \geq 0$$

5.2.1 Contour plots

In order to use the framework to visualize the algorithm's execution using contour plots, the following steps should be taken.

First, after importing all the necessary modules of the framework (as described previously), the user needs to create an instance of the *F1RosenbrockBananaFunction* class which inherits the *IFunction* class:

```
function =
F1RosenbrockBananaFunction.F1RosenbrockBananaFunction()
```

Next, the user needs to create instances of classes which will represent the necessary constraints. The explicit constraints are created in the following way:

```
explicit_constraint_on_x1 =
ExplicitConstraintForOneDimension.ExplicitConstraintForOneDim
ension(-100.0, +100.0)
```

```
explicit_constraint_on_x2 =
ExplicitConstraintForOneDimension.ExplicitConstraintForOneDim
ension(-100.0, +100.0)
```

Representations of the implicit constraints defined in the problem already exist within the framework – they are called *InequalityImplicitConstraint1* and *InequalityImplicitConstraint2*. Therefore, they can be easily created using the following lines:

```
implicit_constraint_1 =
InequalityImplicitConstraint1.InequalityImplicitConstraint1()

implicit_constraint_2 =
InequalityImplicitConstraint2.InequalityImplicitConstraint2()
```

Next, an instance of the *BoxAlgorithm* class which inherits the *IAlgorithm* class needs to be created. In addition to some other parameters, constraints need to be given to it in lists, so those lists also need to be created:

```
explicit_constraints =
     [explicit_constraint_on_X1, explicit_constraint_on_X2]
implicit_constraints =
     [implicit_constraint_1, implicit_constraint_2]
box_algorithm = BoxAlgorithm.BoxAlgorithm(
     function = function,
     explicit_constraints = explicit_constraints,
     implicit_constraints = implicit_constraints,
     epsilon = 1E-6,
     alpha = 1)
```

Now the algorithm should be run. Seeing as an initial point is needed for that, one must be created:

```
point = Point.Point([-1.9, 2])
solution_box, logger_box = box_algorithm.run(point)
```

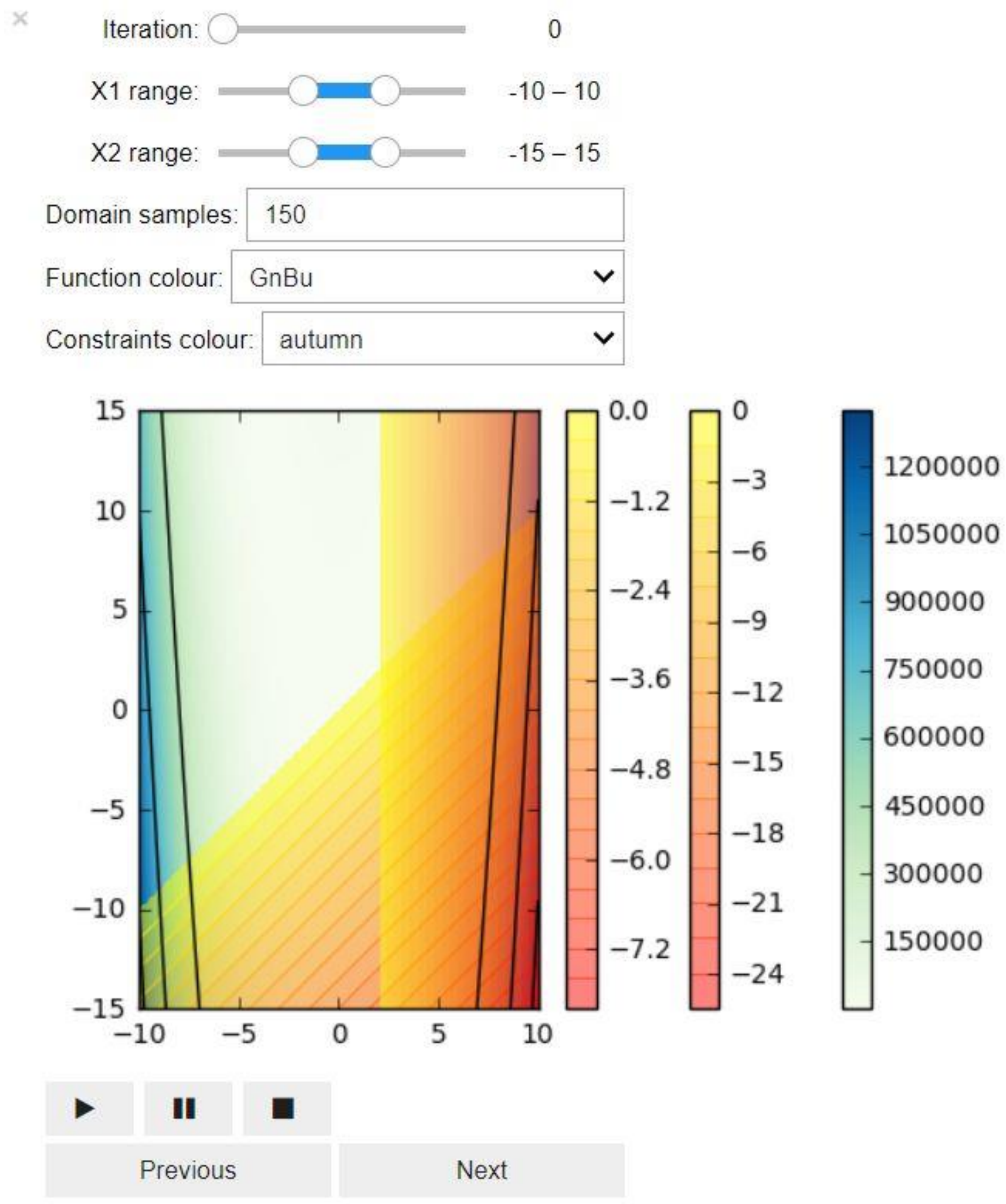
The results of running the algorithm are the solution to the optimization problem (`solution_box`), and a log of its iterations saved in an instance of the *Logger* class (`logger_box`). The log can now be given to a *Presenter* so that the final interactive visualization can be displayed:

```
presenter = Presenter.Presenter(logger = logger_box, drawer =
Drawer.Drawer(), animator = Animator.Animator())
```

Executing the line

```
presenter.present_contour()
```

presents the user with elements for an interactive visualization which uses contour plots, as shown in Figure 12.


Iteration: 0

X1 range: -10 – 10

X2 range: -15 – 15

Domain samples: 150

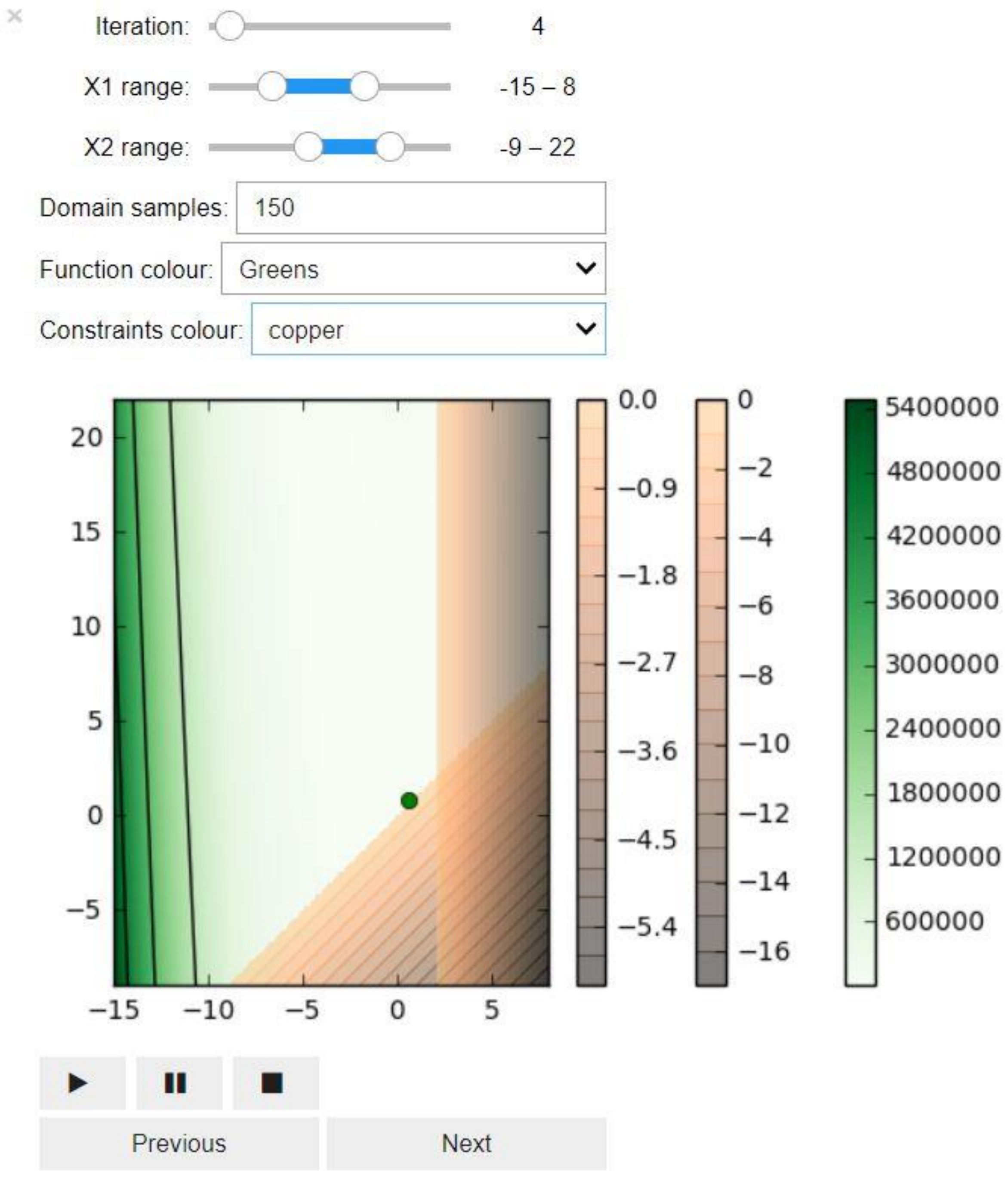
Function colour: GnBu

Constraints colour: autumn

Figure 12 – An interactive visualization which uses contour plots

The available options are very similar to the ones mentioned previously. The constraints are visualised by the orange shading of the infeasible area - the parts of the domain which could never be “reached” by the algorithm because they are excluded from the set of possible solutions by the constraints.

After interacting with some of the given elements, the graph could change to the one shown in Figure 13.



Iteration: 4

X1 range: -15 – 8

X2 range: -9 – 22

Domain samples: 150

Function colour: Greens

Constraints colour: copper

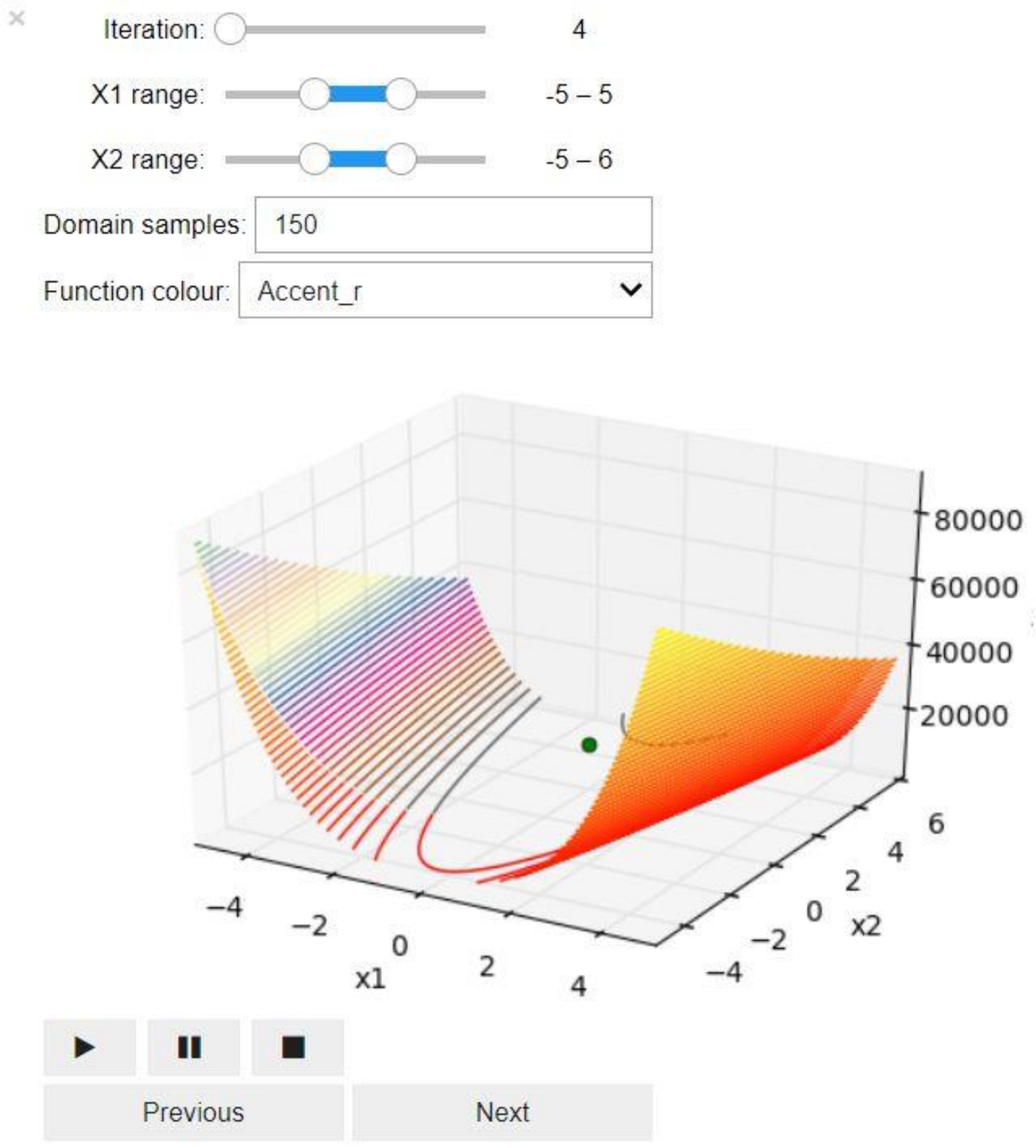
Figure 13 - An interactive visualization which uses contour plots, after some interaction

5.2.2 Three-dimensional plots

The framework can be used to visualise the same problem using three-dimensional plots simply by executing the following line:

```
presenter.present_3D()
```

The user is then presented with the elements shown in Figure 14.



Iteration: 4

X1 range: -5 – 5

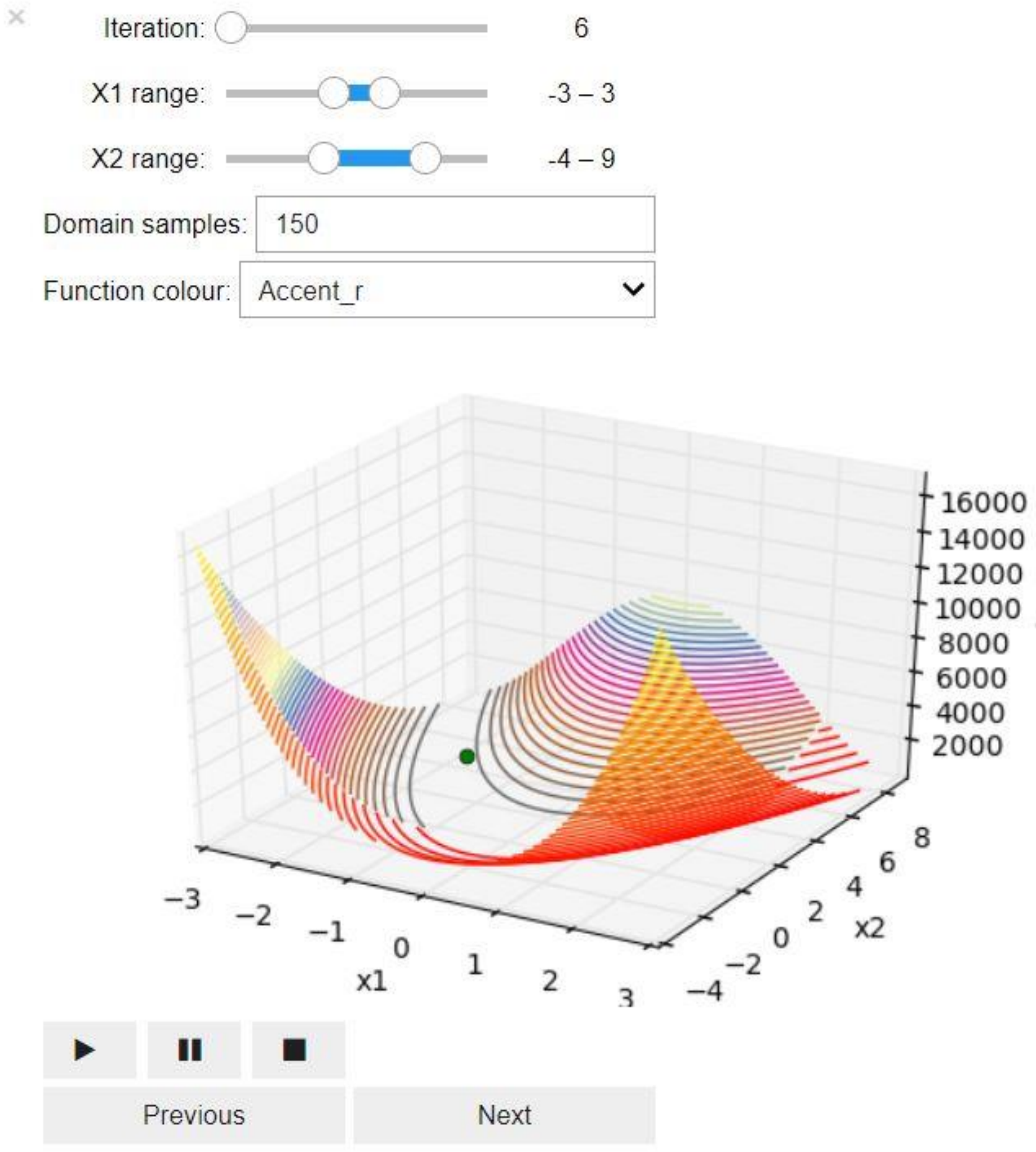
X2 range: -5 – 6

Domain samples: 150

Function colour: Accent_r

Figure 14 - An interactive visualization which uses three-dimensional plots

The constrained area is shown in shades of orange again. After some interaction, the graph could change to the one shown in Figure 15.


Iteration: 6

X1 range: -3 – 3

X2 range: -4 – 9

Domain samples: 150

Function colour: Accent_r

Figure 15 - An interactive visualization which uses three-dimensional plots, after some interaction

6. Adding new modules

One of the most defining features of the framework is the ability to add new modules that would inherit the existing interfaces. The following chapters offer examples of how that can be done. It should be noted that, in addition to fully integrating the classes into the framework as is shown here, the user could also simply declare the classes inside a Jupyter notebook and use them only for a particular computational session.

6.1 Adding functions

This example will show how to add the function $f(x) = x^4 - x^2 + \frac{x}{10}$ to the framework. It will be called *F4TwoLocalOptima*. The first step is to create a new Python file in the *Functions* folder of the framework and call it *F4TwoLocalOptima*. Next, a class of the same name needs to be created inside the folder, and it has to implement the methods *value_at*, *gradient_at* and *hessian_at*. The following lines of code show what the file should contain:

```
from IFunction import *
class F4TwoLocalOptima(IFunction):
     def value_at(self, point):
          self.increment_number_of_calls()
          # TODO implement rest of value_at
     def gradient_at(self, point):
          # TODO implement gradient
     def hessian_at(self, point):
          # TODO implement hessian
```

The most important parts are shown in bold. The class must inherit the *IFunction* class if it is to be used in the framework. In order to do this, the line "`from IFunction import *`" needs to be included in the file. It is also important to include the line "`self.increment_number_of_calls()`" in the *value_at* method. Furthermore, the `point` argument in all the above methods is an instance of the *Point* class, which should be taken into account when using it in the implementations. The following would, therefore, be a fully implemented *value_at* method for the current example:

```
def value_at(self, point):
    self.increment_number_of_calls()
    value_at_point = point.get_value_at_dimension(0) ** 4 -
        point.get_value_at_dimension(0) ** 2 +
        point.get_value_at_dimension(0) / 10
    return value_at_point
```

When all the necessary lines of code have been properly added to the file and when all the required methods are implemented, the function can be used as an integral part of the framework.

6.2 Adding algorithms

The process of adding a new algorithm to the framework will be shown here. For the purposes of this example, a fictional algorithm named *NewAlgorithm* will be used. First, a file named *NewAlgorithm* should be created in the *Algorithms* folder. A class named *NewAlgorithm* should be defined inside it, and the following lines of code show what it should contain:

```
from IAlgorithm import *
from Logging import *
class NewAlgorithm(IAlgorithm):
    def __init__(self, function,
                 algorithm_specific_parameters):
        self.function = function
        self.logger = Logger(self.function)
        self.parameters = algorithm_specific_parameters
    def run(self, initial_point):
        # TODO implement run
    return solution_point, self.logger
```

The most important parts are shown in bold. The algorithm should inherit the class *IAlgorithm*. As arguments in its constructor, it should take an instance of the

IFunction class (`function`) and use it to create a *Logger*, and it should also take any other parameters specific to it (`algorithm_specific_parameters`). Additionally, it should implement the *run* method, which takes an instance of the *Point* class as an argument. The *run* method should fill the logger with information about the algorithm's execution. It should do this by creating an instance of the *Iteration* class in each of its iterations, and adding it to the log. When the algorithm finishes executing, it should return its solution (an instance of the *Point* class) and the instance of the *Logger* containing a comprehensive log of the execution. An expanded implementation of the *run* method with some steps described in more detail is:

```
Iteration_number = 0
while(!termination_criterion_met):
     # TODO algorithm-specific implementation - finds the
               current solution
     # log all the relevant information in an instance of the
               Iteration class:
     current_iteration = Iteration(
          iteration_number = iteration_number,
          function_value_at_current_solution =
          self.function.value_at(current_solution),
          current_solution_point = current_solution,
          additional_data =
     algorithm_specific_additional_data,
          number_of_function_calls =
          self.function.get_number_of_calls())
     self.logger.add_iteration(current_iteration)
     iteration_number++
solution_point = current_solution
return solution_point, self.logger
```

6.3 Adding implicit constraints

New implicit constraints can be added to the framework following the same principles used when adding functions or algorithms. Similarly to what the case was with them, the new class should be added to the folder *Constraints*, in a file of the same name as the class. Regardless of whether an equality or inequality implicit constraint is being added, they have to implement the same methods – *is_satisfied*, *value_at* and *get_gradient*. It is only important that inequality constraints inherit the class *IinequalityImplicitConstraint* and equality constraints inherit the class *IEqualityImplicitConstraints*. This is necessary because these constraints need to be treated differently when being drawn, and all the other modules in the framework are designed to use these classes to distinguish between them.

6.4 Adding presenters

The following example will show the creation of a simple presenter, which would only provide the option of selecting the iteration of the algorithm using a slider, and which would display a plot that shows only the graph of the function and the algorithm's solution in the selected iteration.

It will be created by taking an instance of the *Logger*, *Drawer* and *Animator* classes, so its constructor will be defined as:

```
def __init__(self, logger, drawer, animator):
     self.logger = logger
     self.drawer = drawer
     self.animator = animator
```

Its most important method will be the *present* method, which will consist of several parts.
First, it will use the *Logger* to find the function which it will present, and it will add it to the *Drawer*:

```
function = self.logger.get_function()
self.drawer.add_function(function)
```

It will also find the number of iterations that the algorithm took:

```
number_of_iterations = self.logger.get_number_of_iterations()
```

Next, it will use the *Animator* to create a slider for selecting the displayed iteration. The number of iterations is the maximum value that the slider should offer, so it will be created with the following line:

```
iteration_slider = self.animator.create_int_slider(
     min = 0,
     max = number_of_iterations - 1,
     step=1)
```

The desired functionality is for the displayed graph to change depending on the selected iteration – it should always display the graph of the function, but the displayed solution should be the one corresponding to the iteration number. This will be done using the *interact* method, but to achieve it, a new method needs to be defined – one that takes the iteration number as a parameter and displays its corresponding graph.

This method will first remove all the points that are present in the *Drawer*, then update it with the algorithm's solution in the chosen iteration. Finally, it will display the desired graph. The code of that method is the following:

```
def draw_simple_iteration(self, iteration_number):
     self.drawer.clear_points()
     self.drawer.add_point(self.logger.get_iteration(iteratio
     n_number).get_current_solution())
     self.drawer.draw_2D_graph()
```

After this method is defined, the presenter will have all the components necessary in order to present the user with an interactive visualization. The line

```
interact(draw_simple_iteration,
     iteration_number=iteration_slider)
```

will call the *draw_simple_iteration* method, and the value passed to it as the `iteration_number` argument will be the value obtained from the slider, which will be controlled by the user.

Therefore, the complete code for the *SimplePresenter* class is the following:

```
from Animator import *
class SimplePresenter:
    def __init__(self, logger, drawer, animator):
        self.logger = logger
        self.drawer = drawer
        self.animator = animator
    def draw_simple_iteration (self, iteration_number):
        self.drawer.clear_points()
        self.drawer.add_point(
            self.logger.get_iteration(iteration_number).ge
            t_current_solution())
        self.drawer.draw_2D_graph()
    def present(self):
        function = self.logger.get_function()
        self.drawer.add_function(function)
        number_of_iterations =
            self.logger.get_number_of_iterations()
        iteration_slider =
            self.animator.create_int_slider(
                min=0,
                max=number_of_iterations - 1,
                step=1)
        interact(self. draw_simple_iteration,
            iteration_number=iteration_slider)
```

7. Conclusion

The developed software framework offers the ability of visualizing the execution of optimization algorithms in an interactive way. It offers visualizations of one-dimensional objective functions using two-dimensional plots, as well as visualizations of two-dimensional objective functions using both contour plots and three-dimensional plots. Furthermore, its modular design allows for the extension and adaptation of almost all of its components.

This modular design is one of the framework's most significant benefits. The ease with which it can be extended makes it more likely that it will continue to be upgraded over time, and that it will grow into a sought-after educational resource.

Another great benefit of the framework is that it poses very few restrictions on how it can be used. It is not necessary that additional modules be built into it - they can simply be defined when they are needed and then used in conjunction with the framework's existing capabilities. Whichever function, constraint, algorithm or even visualization method a user might need, it can be created right inside a Jupyter Notebook with only a few lines of code, and then used instantly. This makes the framework suitable for demonstration purposes, because examples can not only be easily prepared, but also given to users to experiment with. This can help them to get a better understanding of the execution of the algorithms when they are applied to different optimization problems.

8. Bibliography

[1] Budin, L., Analiza i projektiranje računalom - skripta s predavanjima, Zagreb

[2] Jupyter Team, What is the Jupyter Notebook?, http://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/What%20is%20the%20Jupyter%20Notebook.html, 28 June 2018

[3] Project Jupyter, ipywidgets: User Guide, https://ipywidgets.readthedocs.io/en/latest/index.html, 28 June 2018

A Software Framework for Interactive Visualization of Optimization Algorithms

Abstract

A software framework for interactive visualization of optimization algorithms has been developed as part of this thesis. It offers visualizations of one-dimensional objective functions using two-dimensional plots, as well as visualizations of two-dimensional objective functions using both contour plots and three-dimensional plots. It is modular and can be extended with ease. This thesis provides details about its implementation and the ways it can be used. Instructions and examples for extending the framework are also provided.

Keywords: optimization algorithms, optimization problems, interactivity, visualization, modularity, framework, Python, Jupyter Notebook, ipywidgets

Programski okvir za interaktivnu vizualizaciju algoritama optimizacije

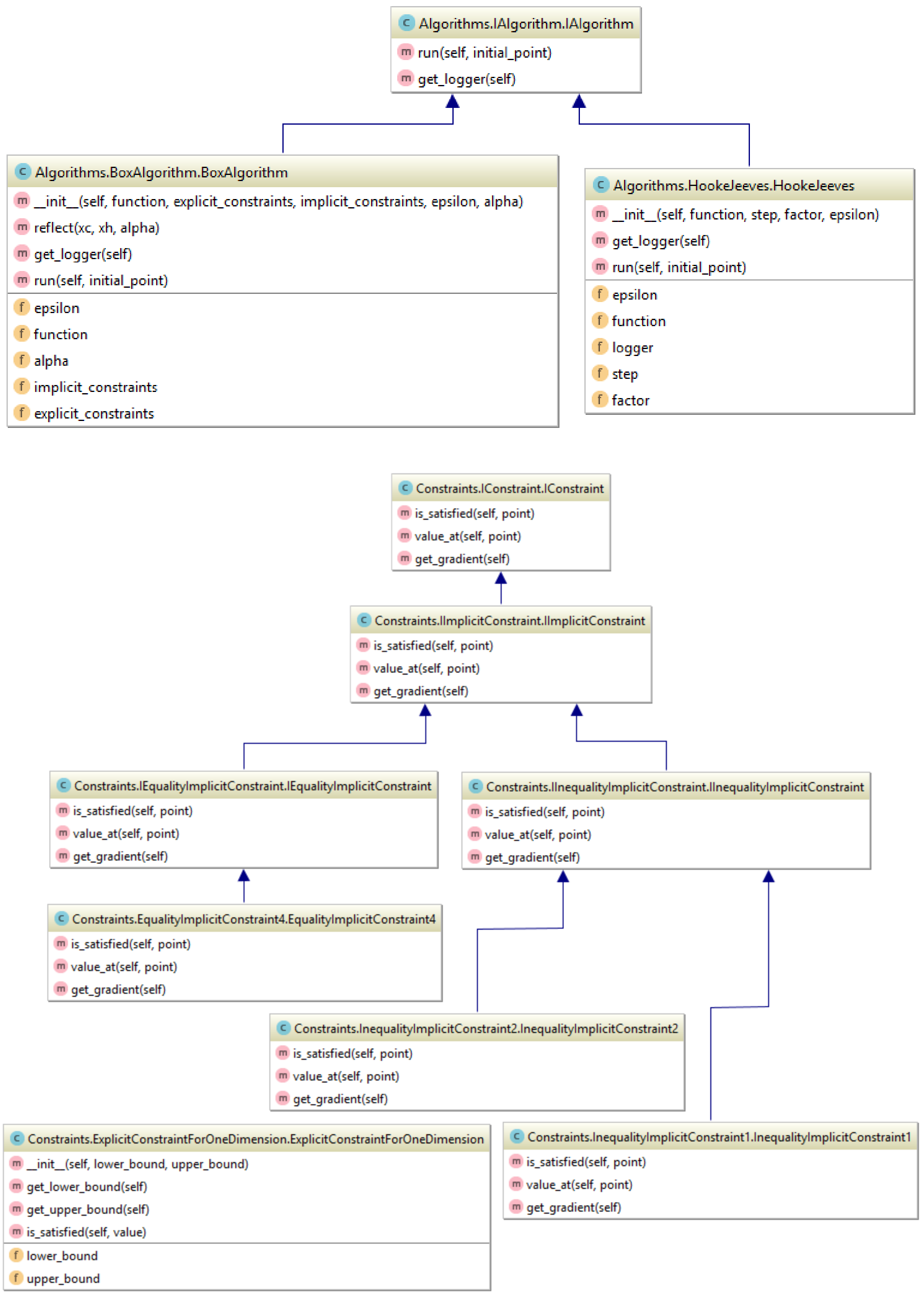
Sažetak

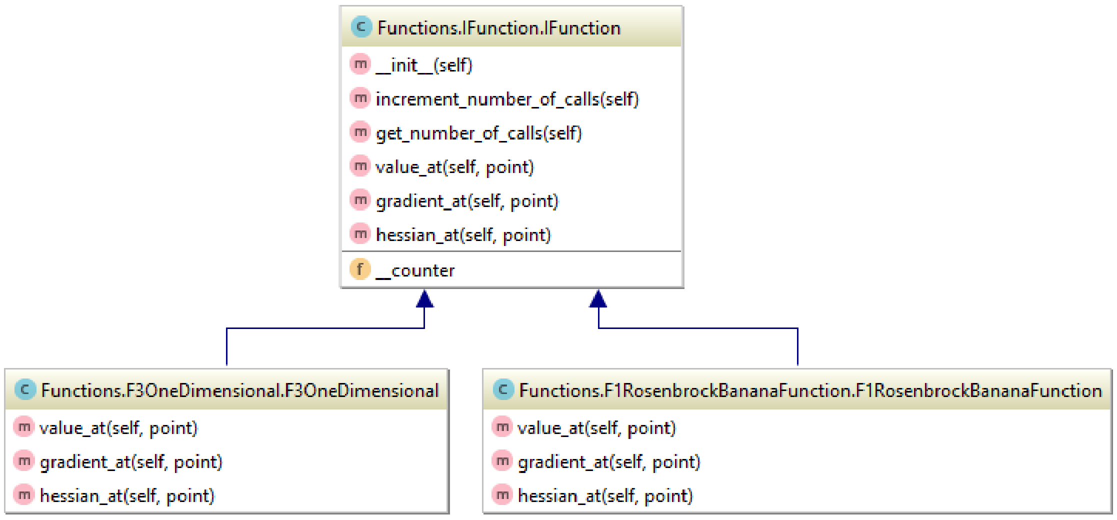
U sklopu ovog rada razvijen je programski okvir za interaktivnu vizualizaciju algoritama optimizacije. Nudi mogućnost vizualizacije ciljnih funkcija jedne dimenzije koristeći dvodimenzionalne grafove i mogućnost vizualizacije ciljnih funkcija dvije dimenzije koristeći grafove kontura i trodimenzionalne grafove. Modularan je i može se lako nadograđivati. U radu je podrobnije objašnjena njegova implementacija i načini na koje se može koristiti. Također su ponuđene upute i primjeri za njegovu nadogradnju.

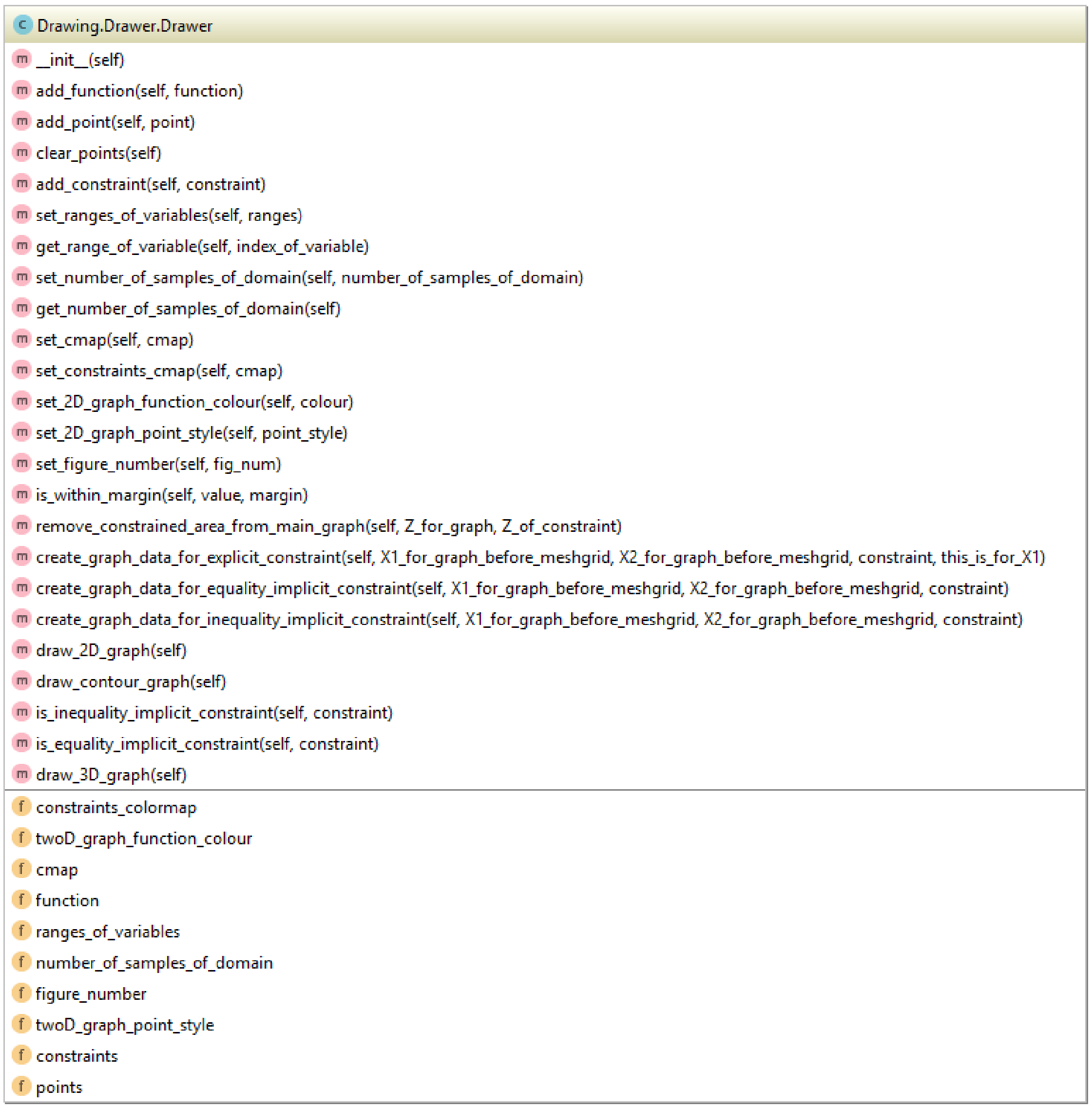
Ključne riječi: algoritmi optimizacije, problemi optimizacije, interaktivnost, vizualizacija, modularnost, programski okvir, Python, Jupyter Notebook, ipywidgets

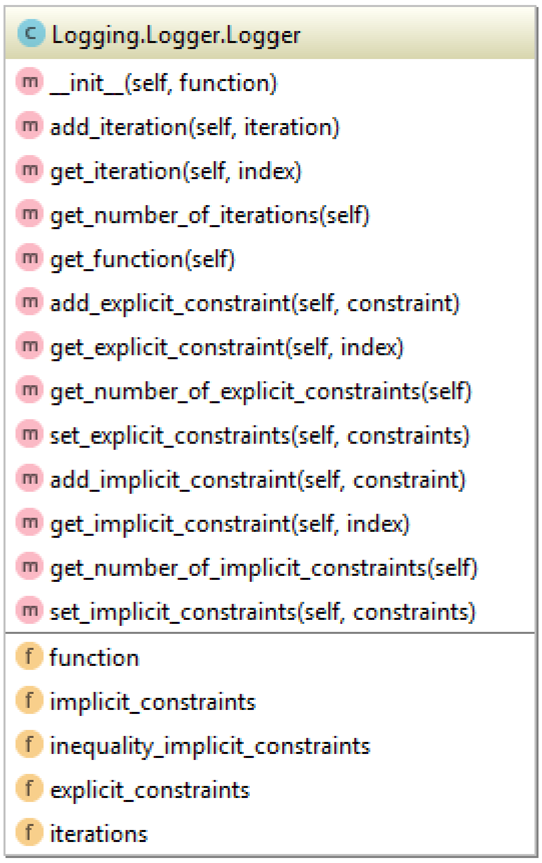
Appendix A

Detailed class diagram of the framework


Algorithms.IAlgorithm.IAlgorithm
run(self, initial_point)
get_logger(self)
Algorithms.BoxAlgorithm.BoxAlgorithm
__init__(self, function, explicit_constraints, implicit_constraints, epsilon, alpha)
reflect(xc, xh, alpha)
get_logger(self)
run(self, initial_point)
epsilon
function
alpha
implicit_constraints
explicit_constraints
Algorithms.HookeJeeves.HookeJeeves
__init__(self, function, step, factor, epsilon)
get_logger(self)
run(self, initial_point)
epsilon
function
logger
step
factor
Constraints.IConstraint.IConstraint
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.IImplicitConstraint.IImplicitConstraint
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.IEqualityImplicitConstraint.IEqualityImplicitConstraint
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.IInequalityImplicitConstraint.IInequalityImplicitConstraint
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.EqualityImplicitConstraint4.EqualityImplicitConstraint4
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.InequalityImplicitConstraint2.InequalityImplicitConstraint2
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)
Constraints.ExplicitConstraintForOneDimension.ExplicitConstraintForOneDimension
__init__(self, lower_bound, upper_bound)
get_lower_bound(self)
get_upper_bound(self)
is_satisfied(self, value)
lower_bound
upper_bound
Constraints.InequalityImplicitConstraint1.InequalityImplicitConstraint1
is_satisfied(self, point)
value_at(self, point)
get_gradient(self)


Functions.IFunction.IFunction
__init__(self)
increment_number_of_calls(self)
get_number_of_calls(self)
value_at(self, point)
gradient_at(self, point)
hessian_at(self, point)
__counter
Functions.F3OneDimensional.F3OneDimensional
value_at(self, point)
gradient_at(self, point)
hessian_at(self, point)
Functions.F1RosenbrockBananaFunction.F1RosenbrockBananaFunction
value_at(self, point)
gradient_at(self, point)
hessian_at(self, point)

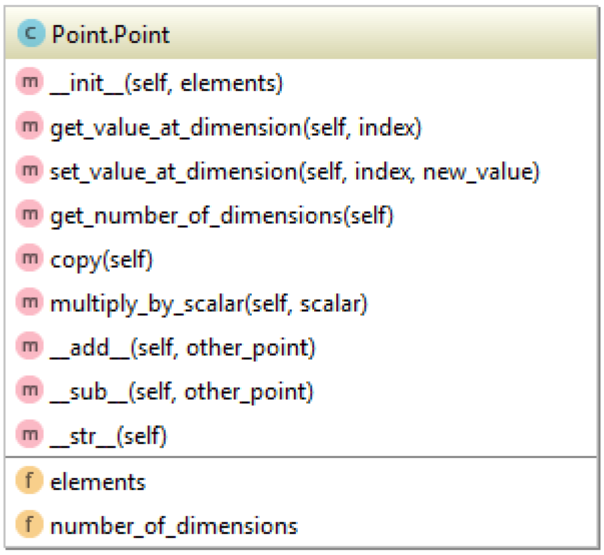

Drawing.Drawer.Drawer
__init__(self)
add_function(self, function)
add_point(self, point)
clear_points(self)
add_constraint(self, constraint)
set_ranges_of_variables(self, ranges)
get_range_of_variable(self, index_of_variable)
set_number_of_samples_of_domain(self, number_of_samples_of_domain)
get_number_of_samples_of_domain(self)
set_cmap(self, cmap)
set_constraints_cmap(self, cmap)
set_2D_graph_function_colour(self, colour)
set_2D_graph_point_style(self, point_style)
set_figure_number(self, fig_num)
is_within_margin(self, value, margin)
remove_constrained_area_from_main_graph(self, Z_for_graph, Z_of_constraint)
create_graph_data_for_explicit_constraint(self, X1_for_graph_before_meshgrid, X2_for_graph_before_meshgrid, constraint, this_is_for_X1)
create_graph_data_for_equality_implicit_constraint(self, X1_for_graph_before_meshgrid, X2_for_graph_before_meshgrid, constraint)
create_graph_data_for_inequality_implicit_constraint(self, X1_for_graph_before_meshgrid, X2_for_graph_before_meshgrid, constraint)
draw_2D_graph(self)
draw_contour_graph(self)
is_inequality_implicit_constraint(self, constraint)
is_equality_implicit_constraint(self, constraint)
draw_3D_graph(self)
constraints_colormap
twoD_graph_function_colour
cmap
function
ranges_of_variables
number_of_samples_of_domain
figure_number
twoD_graph_point_style
constraints
points



Logging.Logger.Logger

__init__(self, function)
add_iteration(self, iteration)
get_iteration(self, index)
get_number_of_iterations(self)
get_function(self)
add_explicit_constraint(self, constraint)
get_explicit_constraint(self, index)
get_number_of_explicit_constraints(self)
set_explicit_constraints(self, constraints)
add_implicit_constraint(self, constraint)
get_implicit_constraint(self, index)
get_number_of_implicit_constraints(self)
set_implicit_constraints(self, constraints)

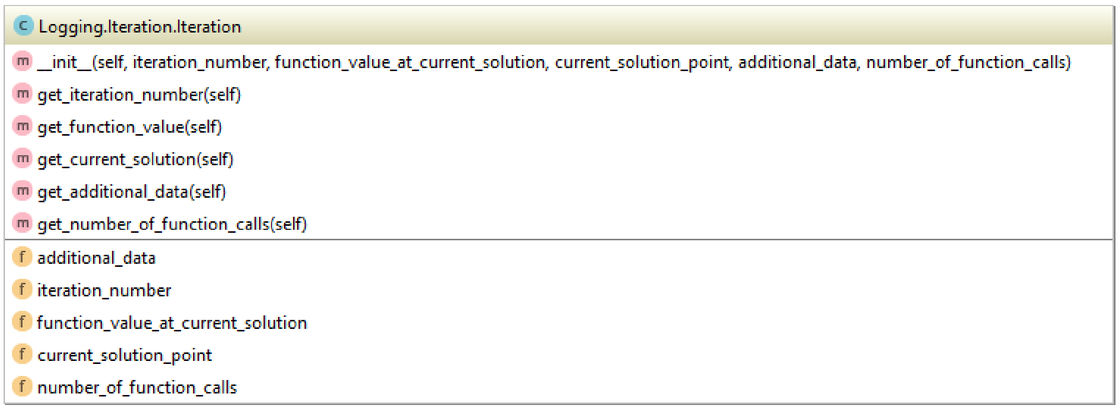
function
implicit_constraints
inequality_implicit_constraints
explicit_constraints
iterations



Point.Point

__init__(self, elements)
get_value_at_dimension(self, index)
set_value_at_dimension(self, index, new_value)
get_number_of_dimensions(self)
copy(self)
multiply_by_scalar(self, scalar)
__add__(self, other_point)
__sub__(self, other_point)
__str__(self)

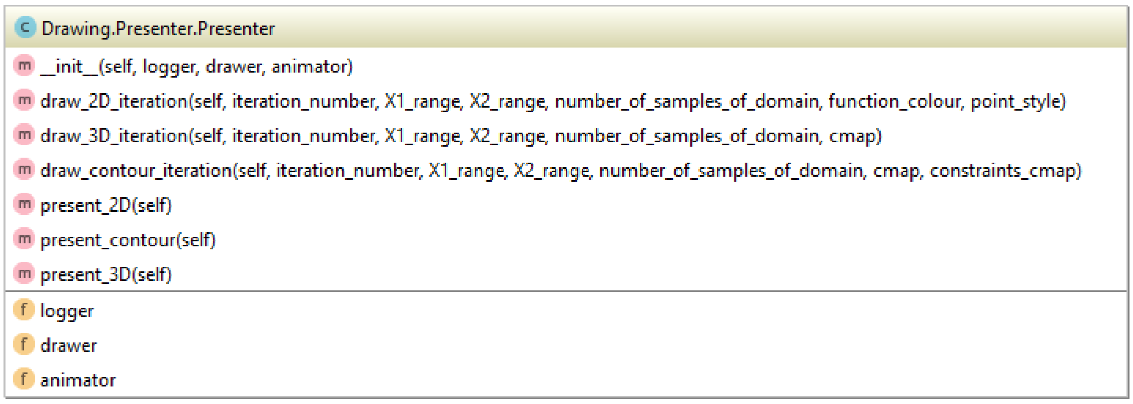
elements
number_of_dimensions



Logging.Iteration.Iteration

- __init__(self, iteration_number, function_value_at_current_solution, current_solution_point, additional_data, number_of_function_calls)
- get_iteration_number(self)
- get_function_value(self)
- get_current_solution(self)
- get_additional_data(self)
- get_number_of_function_calls(self)

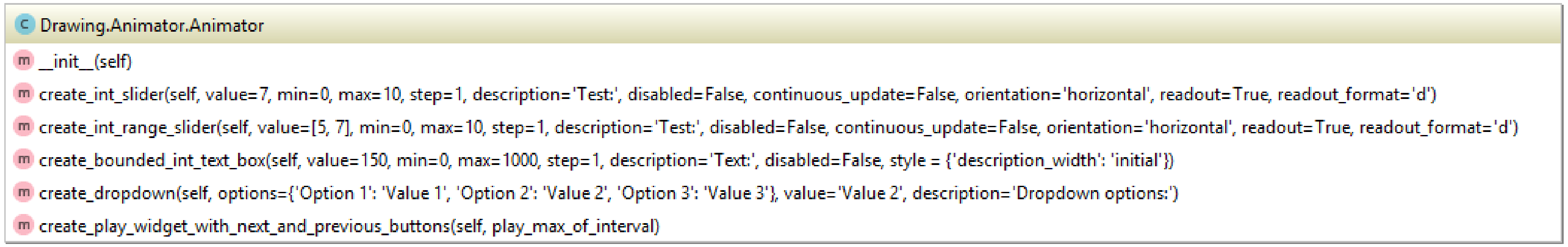
- additional_data
- iteration_number
- function_value_at_current_solution
- current_solution_point
- number_of_function_calls



Drawing.Presenter.Presenter

- __init__(self, logger, drawer, animator)
- draw_2D_iteration(self, iteration_number, X1_range, X2_range, number_of_samples_of_domain, function_colour, point_style)
- draw_3D_iteration(self, iteration_number, X1_range, X2_range, number_of_samples_of_domain, cmap)
- draw_contour_iteration(self, iteration_number, X1_range, X2_range, number_of_samples_of_domain, cmap, constraints_cmap)
- present_2D(self)
- present_contour(self)
- present_3D(self)

- logger
- drawer
- animator



```
Drawing.Animator.Animator
__init__(self)
create_int_slider(self, value=7, min=0, max=10, step=1, description='Test:', disabled=False, continuous_update=False, orientation='horizontal', readout=True, readout_format='d')
create_int_range_slider(self, value=[5, 7], min=0, max=10, step=1, description='Test:', disabled=False, continuous_update=False, orientation='horizontal', readout=True, readout_format='d')
create_bounded_int_text_box(self, value=150, min=0, max=1000, step=1, description='Text:', disabled=False, style = {'description_width': 'initial'})
create_dropdown(self, options={'Option 1': 'Value 1', 'Option 2': 'Value 2', 'Option 3': 'Value 3'}, value='Value 2', description='Dropdown options:')
create_play_widget_with_next_and_previous_buttons(self, play_max_of_interval)
```