One Dimensional Optimization

Release 1.0.0

Victor Barbarich

Mar 23, 2022

CONTENTS

CHAPTER ONE

PROBLEM STATEMENT

1. **Problem initializing:**

   We have function $f(x)$ and interval $[\,a, b\,]$.
   It's necessary to create a function that finds the minimum of function on interval. In addition need to create an application for work with function.

2. **Minimizing need to doing by 4 methods:**

   1. Golden section search
   2. Successive parabolic interpolation
   3. Brent's method
   4. Broyden–Fletcher–Goldfarb–Shanno algorithm

3. **Requirements:**

   1. Program needs to return $x_{\min}$, $f_{\min}$, information about algorithm steps.
   2. Application needs to get a function, bounds, etc., to print work success message, to animate every steps of algorithm's work.

CHAPTER

TWO

MATHEMATICAL MODEL

1. **Requirements to $f(x)$:**
    1. $f:\mathbb{R}\to\mathbb{R}$
    2. f is uni-modal function on $[\,a,b\,]$
    3. $f\in\mathbf{C}[\,a,b\,]$
    4. $f(x)$ has $\min$ on the interval $[\,a,b\,]$
2. **Algorithms:**
    1. **Golden section search**
        1. Set a $f(x),a,b,e$ - function, left and right bounds, precision
        2. $x_1=\dfrac{b-(b-a)}{\varphi}\qquad x_2=\dfrac{a+(b-a)}{\varphi}$
        3. if $f(x_1)>f(x_2)$ (for min) [$f(x_1)<f(x_2)$ (for max)]
        then $a=x_1$ else $b=x_2$
        4. Repeat $2,3$ steps while $|a-b|\geq e$
    2. **Successive parabolic interpolation**
        1. Set $x_0,x_2,x_1,f=f(x),e$ and calculate $f_0=f(x_0),f_1=f(x_1),f_2=f(x_2)$
        2. Arrange x_0,x_1,x_2 so that $f_2\leq f_1\leq f_0$
        3. Calculate x_{i+1} with the formula below
        $x_{i+1}=x_i+\dfrac{1}{2}\left[\dfrac{(x_{i-1}-x_i)^2(f_i-f_{i-2})+(x_{i-2}-x_i)^2(f_{i-1}-f_i)}{(x_{i-1}-x_i)(f_i-f_{i-2})+(x_{i-2}-x_i)(f_{i-1}-f_i)}\right]$
        4. Repeat step 2-3 until then $|x_{i+1}-x_i|\geq e$ or $|f(x_{i+1})-f(x_i)|\geq e$
    3. **Brent's algorithm. Minimizer(**$f,a,b,e,t=10^{-9}$**)**

        0\. Set:

        0.1 "Parabolic step": $u=x+\dfrac{p}{q}=\dfrac{(x-u)^2\cdot(f(x)-f(w))-(x-w)^2\cdot(f(x)-f(v))}{2\cdot((x-v)\cdot(f(x)-f(w))-(x-w)\cdot(f(x)-f(v)))}$

        0.2 "Golden step": if $x<\dfrac{a+b}{2}$: $u=x+\dfrac{\varphi-1}{\varphi}\cdot(b-x)$ else: $u=x+\dfrac{\varphi-1}{\varphi}\cdot(a-x)$
        0.3 Set tolerance $=e\cdot|x|+t$

        1. Set $f(x),a,b,e$ - function, left and right bounds, precision
        2. There are three variables x,w,v : x is the point, where $f(x)$ is the least of all 3 points, w and $f(w)$ has a middle value and v and $f(v)$ has the largest value.

3. Set $x = w = v = a + \dfrac{\varphi - 1}{\varphi} \cdot (b - a)$
4. Let r be the previous remainder. (The remainder is the value we add to x step by step).
5. Check **4 conditions**
    1. $|r| >$ tolerance
    2. $q \neq 0$
    3. $x + \dfrac{p}{q} \in [\,a, b\,]$
    4. $\dfrac{p}{q} < \dfrac{r}{2}$
6. If 4 conditions are satisfied do **"Parabolic step"** else **"Golden step"**
7. Rearrange u, x, w, v to x, w, v by rule in step 2.
8. Repeat 4-7 until $|x - \dfrac{a+b}{2}| < 2\cdot$ tolerance $- \dfrac{b-a}{2}$

4. **BFGS**

   Wright and Nocedal, 'Numerical Optimization', 1999, pp. 56-60 - alpha search; pp.136-140 BFGS algorithm. The algorithm will be here later…

CHAPTER

THREE

ONE DIMENSIONAL OPTIMIZATION

3.1 algorithms

3.1.1 golden_section_search

golden_section_search(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*, ***kwargs*)

Golden-section search

Algorithm: $\varphi = \dfrac{(1+\sqrt{5})}{2}$

1. a, b - left and right bounds
2. $x_1 = b - \dfrac{b-a}{\varphi}$

   $x_2 = a + \dfrac{b-a}{\varphi}$
3. if $f(x_1) > f(x_2)$ (for min) $[f(x_1) < f(x_2)$ (for max) $]$
   then $a = x_1$ else $b = x_2$
4. Repeat $2, 3$ steps while $|a-b| > e$

If optimization fails golden_section_search will return the last point

Code example:

```
>>> def func(x): return 2.71828 ** (3 * x) + 5 * 2.71828 ** (-2 * x)
>>> point, data = golden_section_search(func, (-10, 10), type_optimization='min',␣
↪keep_history=True)
```

Parameters

- **function** (*Callable[[numbers.Real, Any], numbers.Real]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **bounds** (*Tuple[numbers.Real, numbers.Real]*) – tuple with two numbers. This is left and right bound optimization. [a, b]
- **epsilon** (*numbers.Real*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – 'min' / 'max' - type of required value
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs

- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*OneDimensionalOptimization.algorithms.support.Point*, *OneDimensionalOptimization.algorithms.support.HistoryGSS*]

3.1.2 successive_parabolic_interpolation.py

successive_parabolic_interpolation(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*, ***kwargs*)

Successive parabolic interpolation algorithm

Algorithm:

1. Set x_0, x_2, x_1 and calculate $f_0 = f(x_0), f_1 = f(x_1), f_2 = f(x_2)$
2. Arrange x_0, x_1, x_2 so that $f_2 \leq f_1 \leq f_0$
3. Calculate x_{i+1} with the formula below
4. Repeat step 2-3 until then $|x_{i+1} - x_i| \geq e$ or $|f(x_{i+1}) - f(x_i)| \geq e$

$$x_{i+1} = x_i + \frac{1}{2}\left[\frac{(x_{i-1} - x_i)^2 (f_i - f_{i-2}) + (x_{i-2} - x_i)^2 (f_{i-1} - f_i)}{(x_{i-1} - x_i)(f_i - f_{i-2}) + (x_{i-2} - x_i)(f_{i-1} - f_i)}\right]$$

Example

```
>>> def func1(x): return x ** 3 - x ** 2 - x
>>> successive_parabolic_interpolation(func1, (0, 1.5), verbose=True)
Iteration: 0    |       x2 = 0.750      |       f(x2) = -0.891
Iteration: 1    |       x2 = 0.850      |       f(x2) = -0.958
Iteration: 2    |       x2 = 0.961      |       f(x2) = -0.997
Iteration: 3    |       x2 = 1.017      |       f(x2) = -0.999
Iteration: 4    |       x2 = 1.001      |       f(x2) = -1.000
...
```

```
>>> def func2(x): return - (x ** 3 - x ** 2 - x)
>>> successive_parabolic_interpolation(func2, (0, 1.5), type_optimization='max',
↪verbose=True)
Iteration: 0    |       x2 = 0.750      |       f(x2) = 0.891
Iteration: 1    |       x2 = 0.850      |       f(x2) =  0.958
Iteration: 2    |       x2 = 0.961      |       f(x2) =  0.997
Iteration: 3    |       x2 = 1.017      |       f(x2) =  0.999
...
```

Parameters

- **function** (*Callable[[numbers.Real, Any], numbers.Real]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **bounds** (*Tuple[numbers.Real, numbers.Real]*) – tuple with two numbers. This is left and right bound optimization. [a, b]
- **epsilon** (*numbers.Real*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – 'min' / 'max' - type of required value

- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*OneDimensionalOptimization.algorithms.support.Point*, *OneDimensionalOptimization.algorithms.support.HistorySPI*]

3.1.3 brent.py

brent(*function*, *bounds*, *epsilon=1e-05*, *type_optimization='min'*, *max_iter=500*, *verbose=False*, *keep_history=False*, ***kwargs*)

Brent's algorithm. Brent, R. P., Algorithms for Minimization Without Derivatives. Englewood Cliffs, NJ: Prentice-Hall, 1973 pp.72-80

Parameters

- **function** (*Callable[[numbers.Real, Any], numbers.Real]*) – callable that depends on the first positional argument. Other arguments are passed through kwargs
- **bounds** (*Tuple[numbers.Real, numbers.Real]*) – tuple with two numbers. This is left and right bound optimization. [a, b]
- **epsilon** (*numbers.Real*) – optimization accuracy
- **type_optimization** (*Literal['min', 'max']*) – 'min' / 'max' - type of required value
- **max_iter** (*int*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Variables

- **gold_const** – b - (b - a) / phi = a + (b - a) * gold_const
- **type_opt_const** – This value unifies the optimization for each type of min and max

Returns tuple with point and history.

Return type *Tuple*[*OneDimensionalOptimization.algorithms.support.Point*, *OneDimensionalOptimization.algorithms.support.HistoryGSS*]

update_history(*history*, *values*)

Updates brent history :param history: HistoryBrent object in which the update is required :param values: Sequence with values: 'iteration', 'f_least', 'f_middle', 'f_largest', 'x_least',

'x_middle', 'x_largest', 'left_bound', 'right_bound', 'type_step'

Returns updated HistoryBrent

Parameters

- **history** (OneDimensionalOptimization.algorithms.support.HistoryBrent) –
- **values** (*Sequence[Any]*) –

Return type *OneDimensionalOptimization.algorithms.support.HistoryBrent*

3.1.4 combine_function

solve_task(*algorithm='Golden-section search'*, ***kwargs*)

A function that calls one of 4 one-dimensional optimization algorithms from the current directory, example with Golden-section search algorithm:

```
>>> def f(x): return x ** 2
>>> solve_task('Golden-section search', function=f, bounds=[-1, 1])
({'point': -7.538932043742175e-17, 'f_value': 5.6835496360162564e-33},
 {'iteration': [0], 'middle_point': [0], 'f_value': [], 'left_point': [0], 'right_
↪point': [0]})
```

Parameters

- **algorithm** (*Literal['Golden-section search', 'Successive parabolic interpolation', "Brent's method", 'BFGS algorithm']*) – name of type optimization algorithm
- **kwargs** – arguments requested by the algorithm

Returns tuple with point and history.

Return type *Tuple*[*OneDimensionalOptimization.algorithms.support.Point*, *OneDimensionalOptimization.algorithms.support.HistoryGSS*]

3.1.5 support

class HistoryBFGS

Bases: TypedDict

function: List[numbers.Real]

iteration: List[numbers.Real]

point: List[Tuple]

class HistoryBrent

Bases: TypedDict

Class with an optimization history of Brant's algorithm

f_largest: List[numbers.Real]

f_least: List[numbers.Real]

f_middle: List[numbers.Real]

iteration: List[numbers.Integral]

left_bound: List[numbers.Real]

right_bound: List[numbers.Real]

type_step: List

x_largest: List[numbers.Real]

x_least: List[numbers.Real]

x_middle: List[numbers.Real]

class HistoryGSS
Bases: `TypedDict`

Class with an optimization history of GSS

f_value: List[numbers.Real]

iteration: List[numbers.Integral]

left_point: List[numbers.Real]

middle_point: List[numbers.Real]

right_point: List[numbers.Real]

class HistorySPI
Bases: `TypedDict`

Class with an optimization history of SPI

f_value: List[numbers.Real]

iteration: List[numbers.Integral]

x0: List[numbers.Real]

x1: List[numbers.Real]

x2: List[numbers.Real]

class Point
Bases: `TypedDict`

Class with an output optimization point

f_value: numbers.Real

point: numbers.Real

class PointNd
Bases: `TypedDict`

Class with an output optimization point

f_value: numbers.Real

point: Tuple[numbers.Real]

3.2 drawing

3.2.1 gss_visualizer

gen_animation_gss(*func*, *bounds*, *history*, ***kwargs*)
Generates an animation of the golden-section search on *func* between the *bounds*

Parameters

- **func** (*Callable*) – callable that depends on the first positional argument
- **bounds** (*Tuple[numbers.Real, numbers.Real]*) – tuple with left and right points on the x-axis
- **history** (`OneDimensionalOptimization.algorithms.support.HistoryGSS`) – a history object. a dict with lists. keys iteration, f_value, middle_point, left_point, right_point

Returns go.Figure with graph

Return type plotly.graph_objs._figure.Figure

transfer_history_gss(*history*, *func*)

Generate data for plotly express with using animation_frame for animate

```
>>> from OneDimensionalOptimization.algorithms.golden_section_search import␣
↪golden_section_search
>>> _, hist = golden_section_search(lambda x: x ** 2, (-1, 2))
>>> data_for_plot = transfer_history_gss(hist, lambda x: x ** 2)
>>> data_for_plot[::50]
    index  iteration    type        x              y  size
0       0          0  middle  0.50000  2.500000e-01     3
50     24         24    left -0.00001  9.302363e-11     3
```

Parameters

- **history** (OneDimensionalOptimization.algorithms.support.HistoryGSS) – a history object. a dict with lists. keys iteration, f_value, middle_point, left_point, right_point
- **func** (*Callable[[numbers.Real, Any], numbers.Real]*) – the functions for which the story was created

Returns pd.DataFrame for px.scatter

Return type pandas.core.frame.DataFrame

3.2.2 simple_plot

gen_lineplot(*function*, *bounds*, *found_point*)

Generates a graph of the function between the bounds

```
>>> def f(x): return x ** 2
>>> gen_lineplot(f, (-1, 2), ([0], [0]))
```

Parameters

- **function** (*Callable*) – callable that depends on the first positional argument
- **bounds** (*Tuple[numbers.Real, numbers.Real]*) – tuple with left and right points on the x-axis
- **found_point** (*Tuple[Sequence[numbers.Real], Sequence[numbers.Real]]*) – points that was found by the method. A tulpe with two list / np.ndarray / tuple

Returns go.Figure with graph

Return type plotly.graph_objs._figure.Figure

3.3 parser

3.3.1 sympy_parser

logarithm_replace(*string*)

Replace logN(A) by log(A, N), where N is the sequence of symbols before '('. A is the symbols between '(' and ')', including other '(', ')' symbols at lower levels, example:

```
>>> logarithm_replace('log3(x) + 2 * log5(4)')
log(x, 3) + 2 * log(4, 5)

>>> logarithm_replace('logA(log5(4 * x + 1)) + 8')
'log(log(4 * x + 1, 5), A) + 8'
```

Parameters

- **string** (*AnyStr*) – A sequence of symbols. For exmaple some function
- **string** –

Returns A string of symbols with correnct logarithms.

Return type *AnyStr*

parse_func(*function_string*)

Convert the string to sympy.core.expr.Expr:

```
>>> parse_func('log2(x) + e ** x')
exp(x) + log(x)/log(2)

>>> parse_func('tg(sqrt(x)) + log2(loge(x))')
log(log(x))/log(2) + tan(sqrt(x))
```

Parameters **function_string** (*AnyStr*) – a string with function that is written by python rules

Returns function as sympy Expression

Return type sympy.core.expr.Expr

ru_names_to_sympy(*string*)

Replace russian names of function like tg to tan, example:

```
>>> ru_names_to_sympy("tg(x**2)")
'tan(x**2)'
```

Note: This bag of translation words will be updated.

Parameters **string** (*AnyStr*) – A string with some mathematical expression

Returns A string with correct names to sympy

Return type *AnyStr*

sympy_to_callable(*function_sympy*)

Convert sympy expression to callable function, for example:

```
>>> f = parse_func('x**2')
>>> f = sympy_to_callable(f)
>>> f
<function_lambdifygenerated(x)>
>>> f(2)
4
```

Parameters **function_sympy** (*sympy.core.expr.Expr*) – sympy expression

Except AssertionError. If function depends on more one variable

Returns callable function from one argument

Return type *Callable*