Search Algorithms: Object-Oriented Implementation (Part C)

Contents

- Conventional vs. AI Algorithms
- Local Search Algorithms
- ~~Genetic Algorithm~~
- Implementing Hill-Climbing Algorithms
- Defining ‘Problem’ Class // 지난 강의 (Part C-1)
- Adding Gradient Descent // 오늘 강의 (Part C-2)
- Defining ‘HillClimbing’ Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments

Part C-1

Class implementation

MIGRATING TO CLASSES

variables

functions

Defining Classes

problem.py source code define three classes

Problem

solution
value
numEval

__init__
storeResult
report

Numeric

expression
domain
delta = 0.01

__init__
setVariables
getDelta (new)
randomInit
evaluate
mutants
mutate
randomMutant
describe
report
coordinate

Tsp

numCities
locations
distanceTable

__init__
setVariables
calcDistanceTable
randomInit
evaluate
mutants
inversion
randomMutant
describe
report
tenPerRow

Code outside of problem.py

steepest ascent(tsp).py

```
def main():
  p = Tsp()
  ...
def steepestAscent(p):
  ...
def bestOf(neighbors,p):
  ...
def displaySetting():
  ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables() # Set its class var
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algorithm set
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

first-choice(tsp).py

```
def main():
  p = Tsp()
  ...
def firstChoice(p):
  ...
def displaySetting():
  ...

main()
```

```
def main():
    # Create an object for TSP
    p = Tsp()          # Create a problem
    p.setVariables() # Set its class va
    # Call the search algorithm
    firstChoice(p)
    # Show the problem to be solved
    p.describe()
    displaySetting()
    # Report results
    p.report()
```

Code outside of problem.py

steepest ascent(n).py

```
def main():
  p = Numeric()
  ...
def steepestAscent(p):
  ...
def bestOf(neighbors,p):
  ...
def displaySetting(p):
  ...

main()
```

```
def main():
    # Create a Problme object for
    p = Numeric()      # Create a p
    p.setVariables() # Set its cl
    # Call the search algorithm
    steepestAscent(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

first-choice(n).py

```
def main():
  p = Numeric()
  ...
def firstChoice(p):
  ...
def displaySetting(p):
  ...

main()
```

```
def main():
    # Create a Problme object for
    p = Numeric()      # Create a p
    p.setVariables() # Set its cl
    # Call the search algorithm
    firstChoice(p)
    # Show the problem and algori
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```

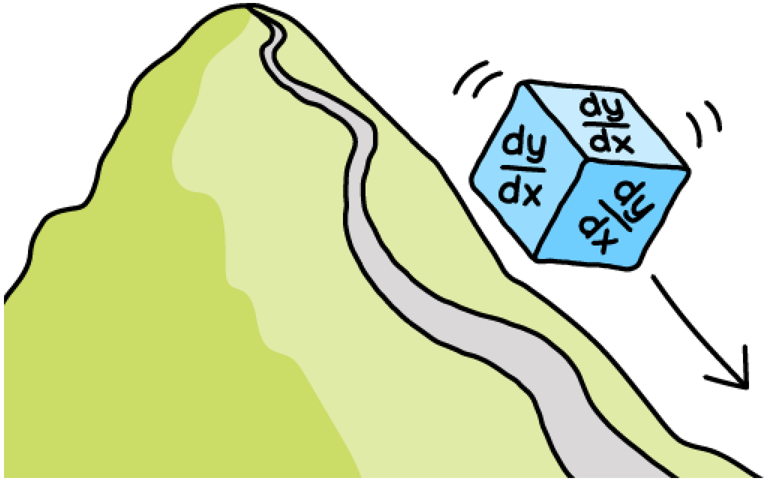
Code outside of problem.py

```
gradient descent.py
```

```
def main():
    p = Numeric()
    ...
def gradientDescent(p):
    ...
def displaySetting(p):
    ...

main()
```

Today's topic

```
def main():
    # Create a Problme object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables() # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```



Solution for solving continuous variable problems

GRADIENT DESCENT ALGO.

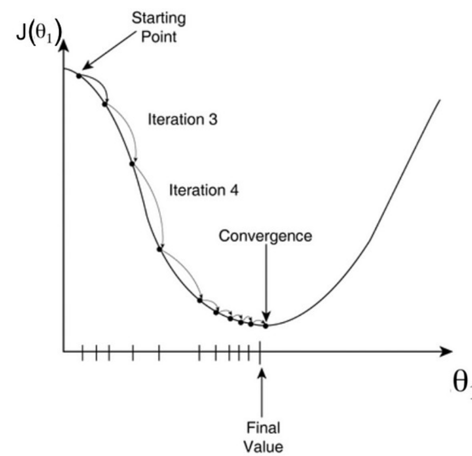
Gradient Descent Algorithm

- General Descent Algorithm (When the objective is to **minimize**)
  - Finds the optimal **solution** by iteratively moving the variable's value slightly in the direction that makes the **objective function smaller**.
  - Find the direction **d** that reduces the objective function and change the variable's value by the **step size**.

```
x = random() # decision variable init
Repeat:
    x = x − α × d, α:step size (learning rate)
Until (Stop condition)
Return x
```

- Primarily used in **Optimization**, **Machine Learning**, and **Deep Learning**.
- Depending on how the **initial value** is set, the algo rithm may converge to a **global optimum** (전역 해) or a **local optimum** (지역 해).

- Primarily used in **Optimization**, **Machine Learning**, and **Deep Learning**.
- Depending on how the **initial value** is set, the algo-rithm may converge to a **global optimum** (전역 해) or a **local optimum** (지역 해).



Gradient Descent Algorithm

- **Gradient Descent** is referred to as a **first-order iterative algorithm** and is a representative technique for finding a **local optimum**.
  - Note: in case of convex optimization, local opt = global opt
  - Note: The search direction is determined using the **first derivative** (1 차 미분 = **gradient**).
  - https://en.wikipedia.org/wiki/Gradient_descent
  - This lecture assumes a **differentiable objective function**.

```
x = random() # decision variable init
Repeat:
    x = x − α × d, α:step size
Until (Stop condition)
Return x
```

The Gradient Descent algorithm uses the **gradient** (derivative) to find the direction ***d*** that reduces the objective function.

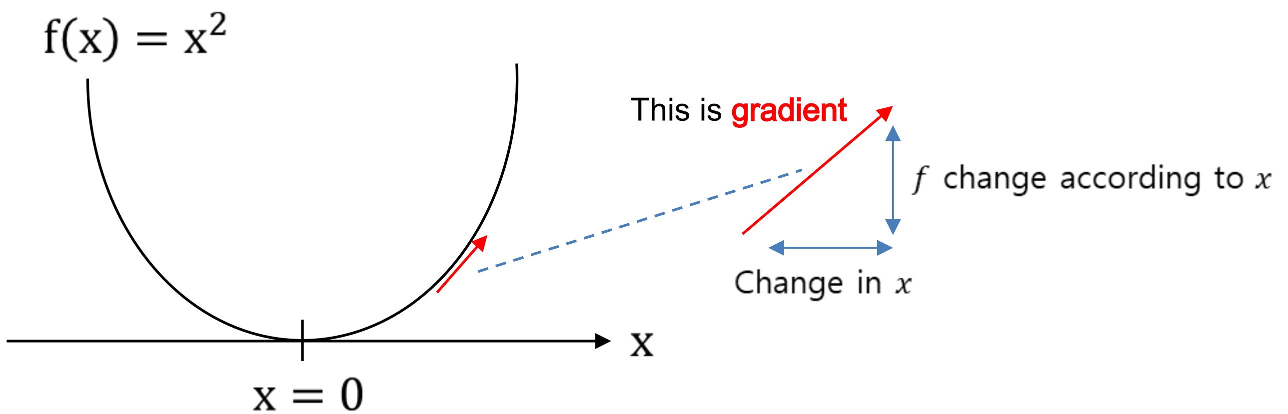

```
x = random() # decision variable init
Repeat:
    x = x − α × d, α:step size
Until (Stop condition)
Return x
```

The Gradient Descent algorithm uses the **gradient** (derivative) to find the direction ***d*** that reduces the objective function.

Gradient Descent Algorithm

- The Direction to Decrease the Objective Function?
  - 1st derivative = **gradient** :

$$f'(x) = \lim_{\Delta \to 0} \frac{f(x+\Delta) - f(x)}{\Delta} = \frac{\text{change in f}}{\text{change in x}}$$



  - The **Gradient** indicates the direction in which the function's value **increases**. Therefore, the **opposite direction of the gradient** ($\text{grad} \times -1$) is the direction in which the **objective function decreases**!

Gradient Descent Algorithm

- A general algorithm for finding solutions to optimization problems: Descent Algorithm
  - Searches for the solution by moving a certain **step size** in the **opposite direction of the Gradient**.

x = random() # decision variable init
Repeat:
    $x = x + \alpha \times (-1 * \nabla_x f_0(x))$, α:step size, ∇:1^{st} derivative
Until (Stop condition)
Return x

The "Stopping condition" can be set in various ways. It can be implemented to terminate the process if the result of plugging the newly calculated x into the objective function is not better than the previous result.

x = random() # decision variable init
Repeat:
    $x = x + \alpha \times (-1 * \nabla_x f_0(x))$, α:step size, ∇:1^{st} derivative
Until (Stop condition)
Return x

The "Stopping condition" can be set in various ways. It can be implemented to terminate the process if the result of plugging the newly calculated x into the objective function is not better than the previous result.

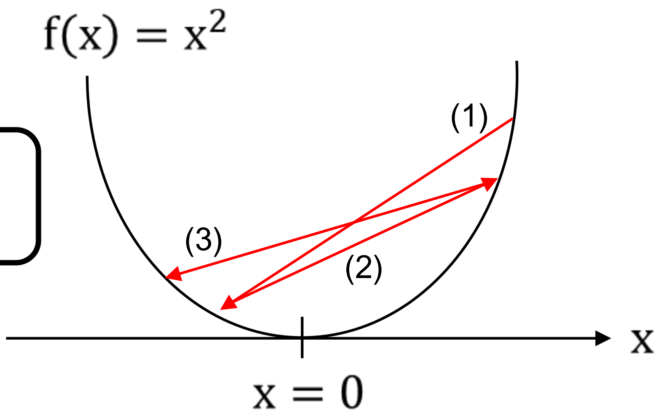
  - The **step size** determines how much the decision variable will move in each iteration,
    In Machine and Deep Learning, this is called the **learning rate (lr)**.

```
keras.optimizers.Adam(
    learning_rate=0.001,
```

Gradient Descent Algorithm

- If the step size(learning rate) is too small, it takes a long time to find the optimal solution
- If the step size(learning rate) is too large? over-shooting may occurs:

Algorithm keep overshooting Optimal solution and cannot reach it.



- So… how should the step size be?

Gradient Descent Algorithm

- How should the step size be set? (here are the options…)
  - Select small value "appropriately" between (0,1): fixed value of around 0.01~0.001 are commonly used
  - Starting with a large value and gradually decreasing (ex: 1/iter_count)

$$\alpha^t = \frac{1+m}{t+m} \text{ for } m \in \mathbb{R}_+$$

  - In general, stochastic gradient converges to a stationary point if
    - Ratio of sum of squared step-sizes over sum of step-sizes converges to 0 (*1)
    - $\frac{\sum_{t=1}^{\infty}(\alpha_t)^2}{\sum_{t=1}^{\infty}\alpha_t} = 0$
  - Note: The method primarily used in **Deep Learning** is to dynamically update the step size (learning rate) using optimizers such as *AdaGrad*, *RMSProp*, and *Adam*

(*1) https://www.cs.ubc.ca/~fwood/CS340/lectures/L24.pdf

Gradient Descent Algorithm

- The Algorithm for Iteratively Updating the Variable's Value:

$$\mathbf{x}_{\mathtt{new}} \leftarrow \mathbf{x}_{\mathtt{current}} - \alpha \nabla \mathtt{f}(\mathbf{x}_{\mathtt{current}})$$

- While you can directly derive the **derivative** (∇f) of the objective function, you can also calculate an **approximation** of the derivative value using a simple operation.

$$\frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Define the Objective Function and its Derivative Function

```python
def f(x): # 목적 함수
    return 2 * (x**2)

def derivative(x): # 목적 함수의 미분
    return 4*x
```

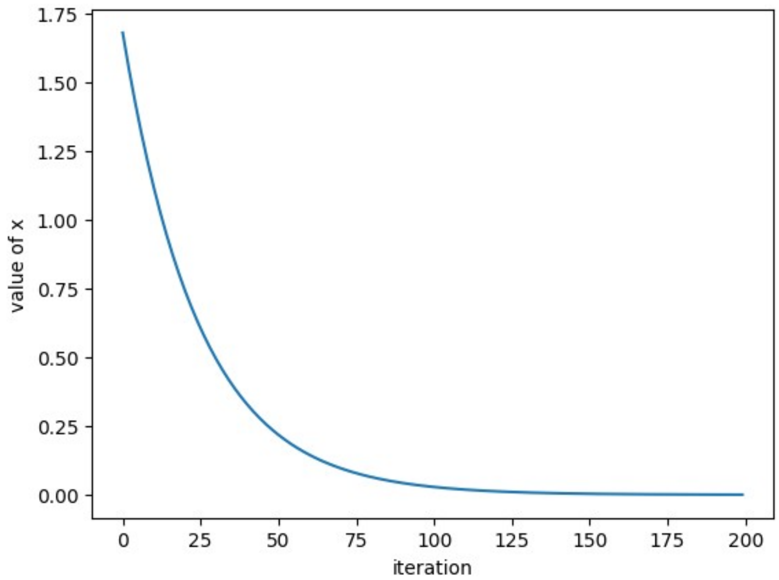
Gradient Descent Algorithm

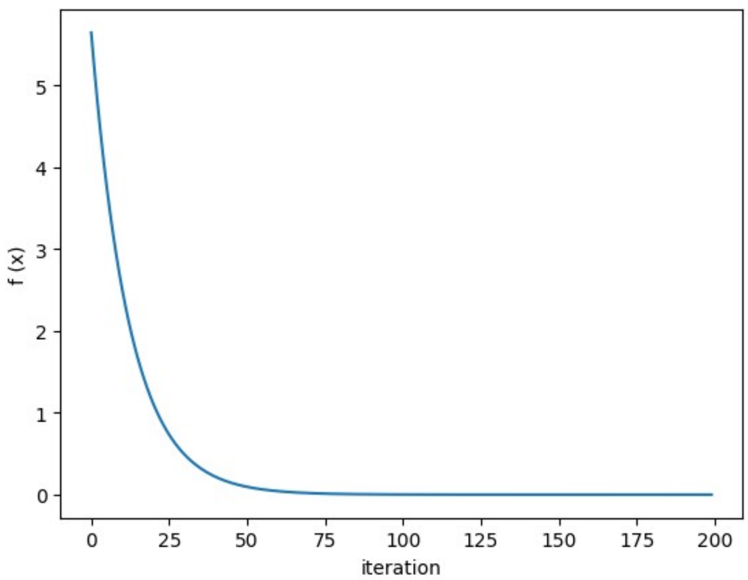
- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Implementing GD algorithm

```
import random

learning_rate = 0.01
#x_current = random.random()*4 - 2  # 무작위 시작점: [-2.0, +2.0)
x_current = 1.75  # 공평한 비교 실험을 위해, 시작점을 고정함

def GD(x_current):
    x_new = x_current - learning_rate * derivative(x_current)
    return x_new, f(x_new)

xs, fs, ITER_MAX = [], [], 200
for i in range(ITER_MAX):
    x_new, f_new = GD(x_current)
    xs.append(x_new)
    fs.append(f_new)
    x_current = x_new
```

Gradient Descent Algorithm

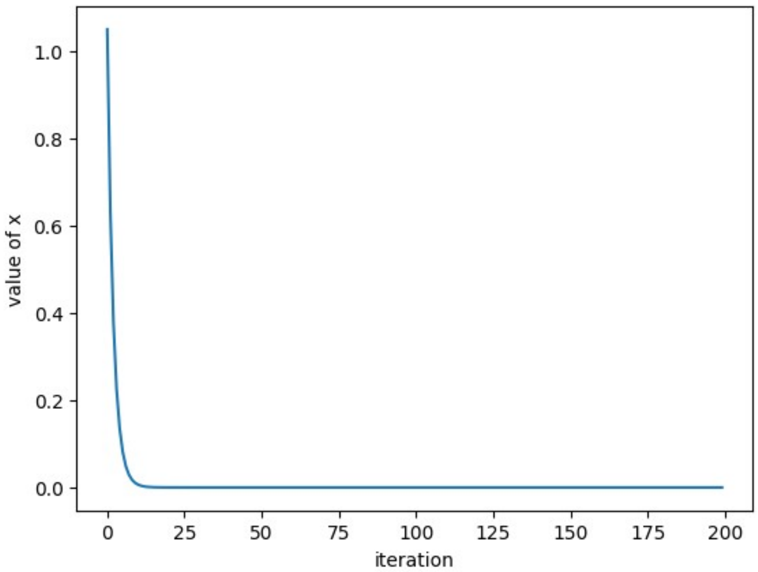
- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.01)

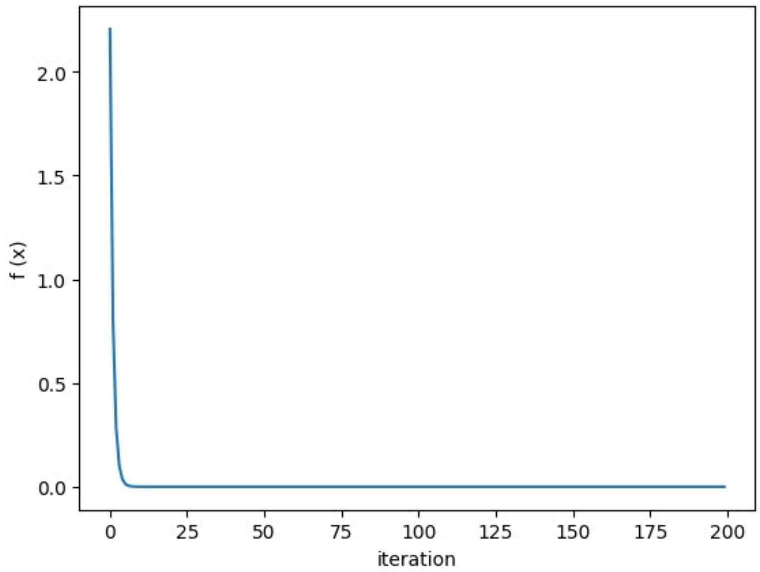




Gradient Descent Algorithm

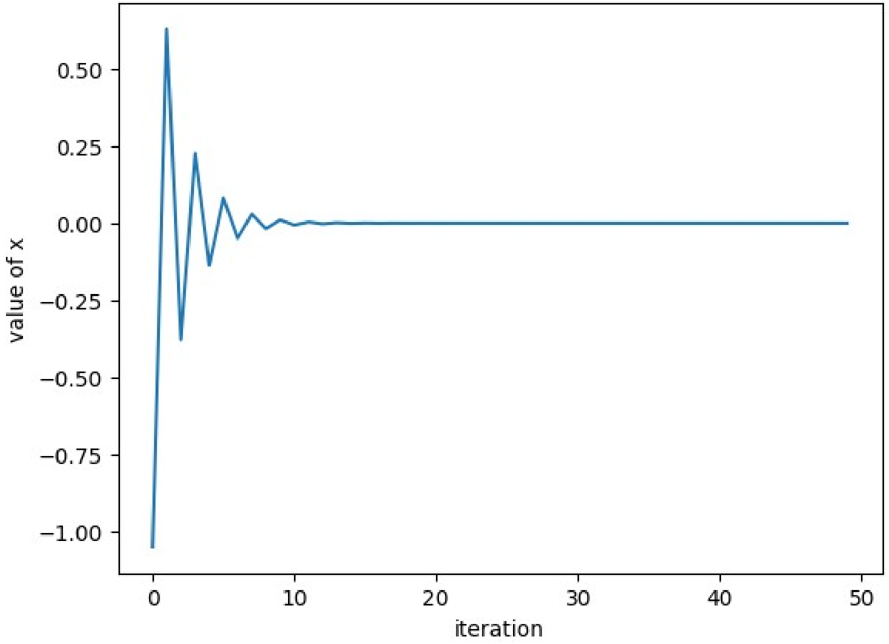
- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.1)
      - ✓ Lr is increased, converging quicker

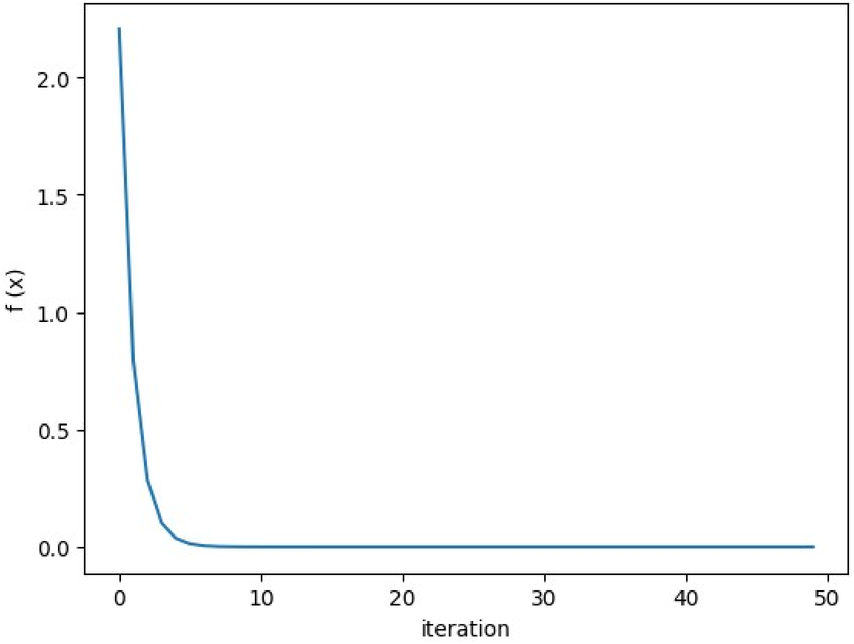




Gradient Descent Algorithm

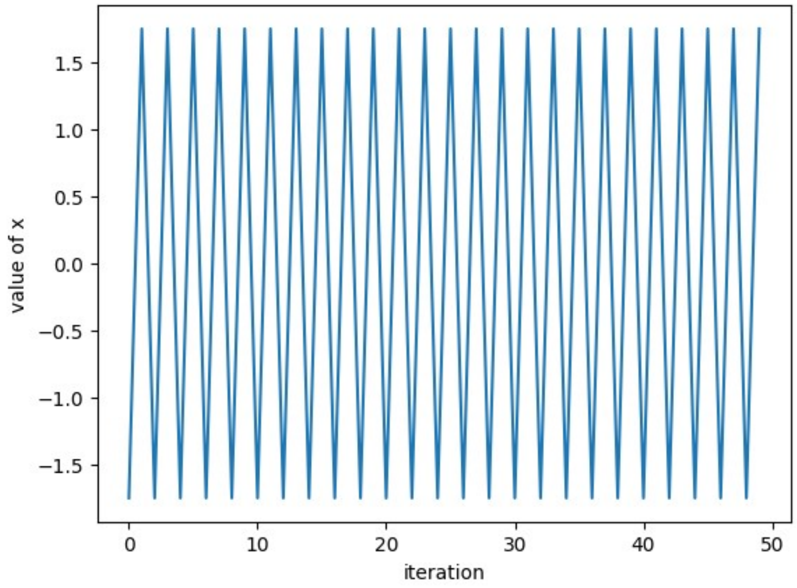
- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.4, ITER_MAX=50)
      - ✓ Unstable convergence at the start

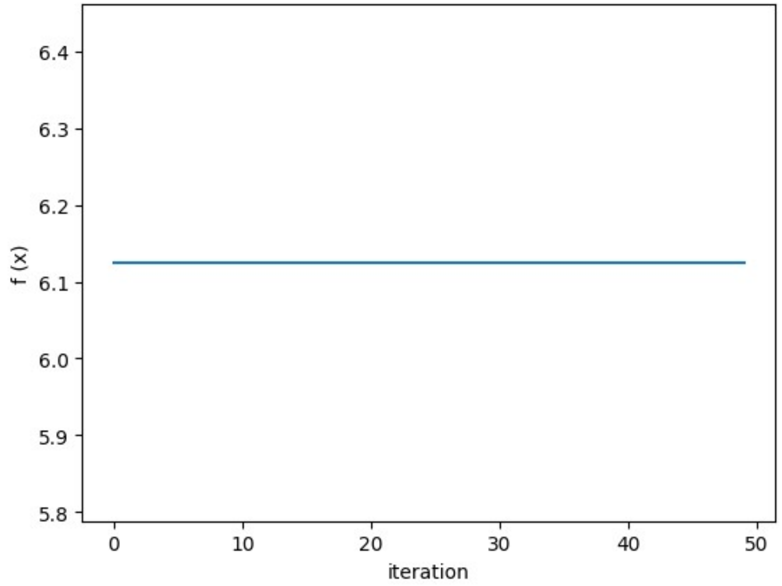




Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.5, ITER_MAX=50)
      - ✓ Zig-zag over minimum value

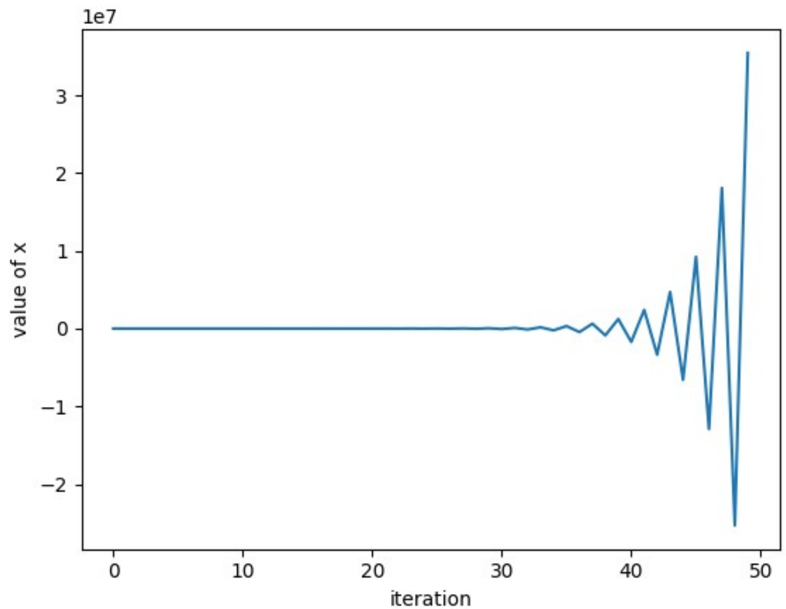


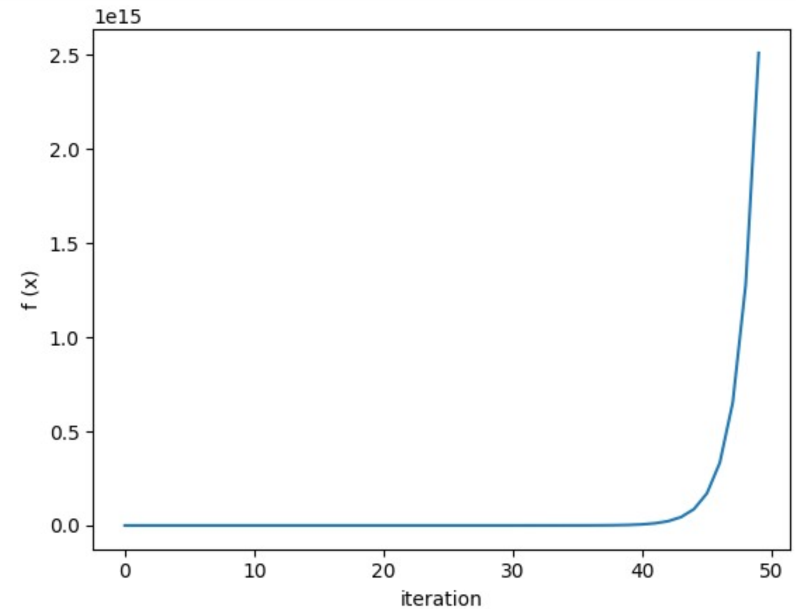


Gradient Descent Algorithm

- Python practice 1
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.6, ITER_MAX=50)
      - ✓ Diverged, fails to converge

The **learning_rate is too large**. As the iterations proceed, the value moves further away from the optimum $(x = 0)$. Consequently, the **gradient $\nabla f(x)$ keeps getting larger**, which causes the entire step size $\alpha \times \nabla f(x)$ to continually increase, leading to **divergence**.





Gradient Descent Algorithm

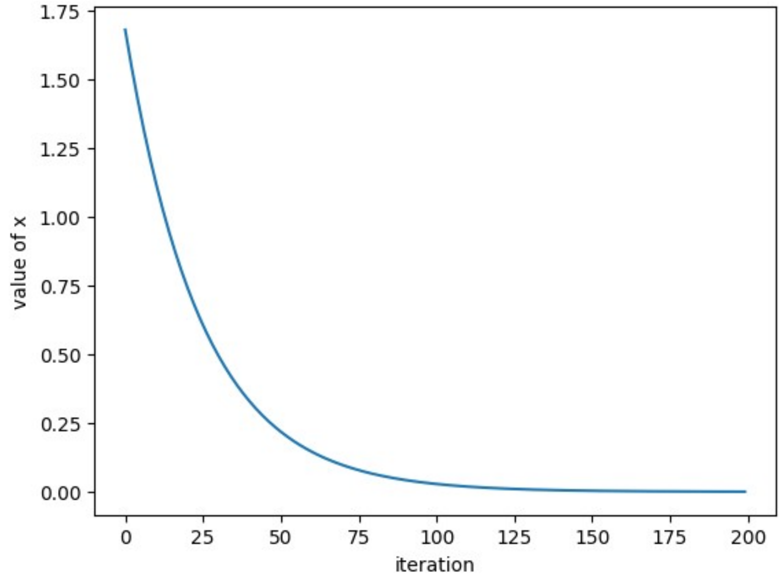
- Python practice 2
  - Using the GD algorithm to minimize f(x)=2x^2
    - Using the derivative value based on the limit definition instead of direct derivative

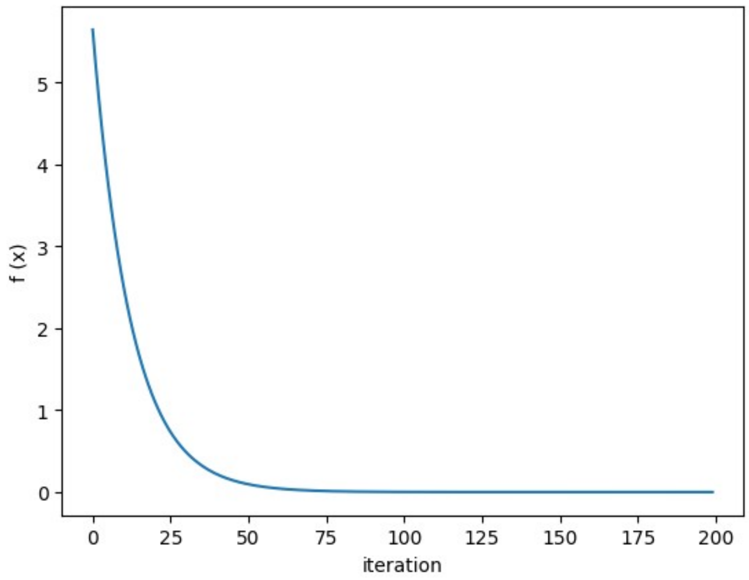
$$\frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

```python
def f(x): # 목적 함수
    return 2 * (x**2)

def derivative(x): # 목적 함수의 미분
    delta = 0.0001
    return (f(x+delta) - f(x)) / delta
```

Gradient Descent Algorithm

- Python practice 2
  - Using the GD algorithm to minimize f(x)=2x^2
    - Experiment result (learning_rate = 0.01, ITER_MAX=200)
      - ✓ **The result is (almost) identical** to the result obtained by calculating and using the analytical derivative.





Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize $f(x)=2x_0$^2+4(x_1-1)^2
  - Vector variable x = [x_0, x_1]

```python
def f(x): # 목적 함수
    # 2 x0^2 + 4 (x1-1)^2
    return 2 * (x[0]**2) + 4 * ((x[1]-1)**2)

def derivative_x0(x): # 목적 함수의 x0에 대한 미분
    return 4*x[0] + 0

def derivative_x1(x): # 목적 함수의 x1에 대한 미분
    return 0 + 8*(x[1]-1)
```

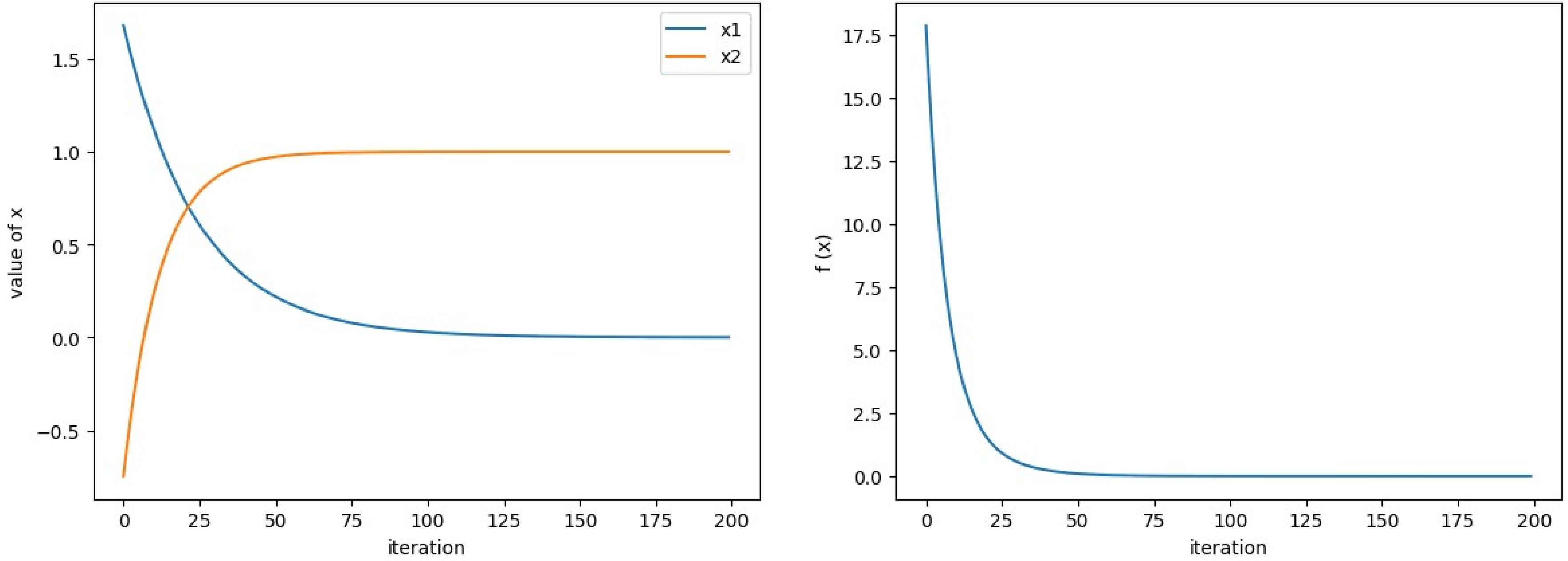
Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize $f(x)=2x_0\text{^}2+4(x_1-1)\text{^}2$
  - Implement GD to take account of the vector variable

```
import random

learning_rate = 0.01
#x_current = [random.random()*4 - 2, random.random()*4 - 2]  # 무작위 시작점: [-2.0, +2.0)
x_current = [1.75, -0.9]  # 공평한 비교 실험을 위해, 시작점을 고정함

def GD(x_current):
    x_new = [x_current[0] - learning_rate * derivative_x0(x_current),
             x_current[1] - learning_rate * derivative_x1(x_current)]
    return x_new, f(x_new)

x1s, x2s, fs, ITER_MAX = [], [], [], 200
for i in range(ITER_MAX):
    x_new, f_new = GD(x_current)
    x1s.append(x_new[0])
    x2s.append(x_new[1])
    fs.append(f_new)
    x_current = x_new
```

Gradient Descent Algorithm

- Python practice 3
  - Using the GD algorithm to minimize f(x)=2x_0^2+4(x_1-1)^2
  - Experiment result (learning_rate = 0.01)



Adding Gradient Descent (for Numeric Optimization problems)

- Based on the content covered so far, let's learn how to **implement Gradient Descent** for a given **Numerical Optimization problem**!

Adding Gradient Descent (for Numeric Optimization problems)

- Gradient descent is the same as the steepest-ascent except the way a next point is created from the current point
  - Gradient descent generates only one neighbor
    - c.f.) Steepest ascent generates m neighbors from which to select a successor to move to (m evaluations are needed)
  - Gradient descent computes gradient at the current point and apply the gradient update rule to calculate the next point
    - n evaluations are needed to calculate partial derivatives in all the dimensions, where n is the dimension of the objective function (i.e., n = number of variables)
    - One more evaluation is needed to evaluate the next point obtained by applying the update rule using the gradient
- Gradient descent is applicable only to numerical optimization

Adding Gradient Descent (for Numeric Optimization problems)

- Two variables are newly added to the **`Numeric`** subclass:
  - **`alpha`**: update rate for gradient descent $\boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$
    - Set to a default value of 0.01 for the time being
    - Referenced by the method **`takeStep`**
  - **`dx`**: size of the increment used when calculating derivative
    - Set to a default value of 10^{-4} for the time being
    - Referenced by the method **`gradient`**

$$\frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

Simple method for calculating the Gradient: Calculate the increase in $f(x)$ when $\mathbf{x}$ increases infinitesimally.

Adding Gradient Descent (for Numeric Optimization problems)

- Also, following methods are newly added to the **`Numeric`** subclass:
  - **`takeStep(self, x, v)`**:
    - Computes the gradient (**`gradient`**) of the current point **`x`** whose objective value is **`v`**

$$\nabla f(\mathrm{x}) = \left(\frac{\partial f(\mathrm{x})}{\partial x_1}, \frac{\partial f(\mathrm{x})}{\partial x_2}, \cdots, \frac{\partial f(\mathrm{x})}{\partial x_n} \right)^T$$

    - Makes a copy of **`x`** and changes it to a new one by applying the gradient update rule as long as the new one is within the domain (**`isLegal`**)

isLegal refers to the case where all individual components x_i that constitute the vector **x** do not exceed their designated bounds (domain).

$$\boldsymbol{x}_i \leftarrow (\mathrm{x} - \alpha \nabla f(\mathrm{x}))_i = x_i - \alpha \frac{\partial f(\mathrm{x})}{\partial x_i}$$

$\Rightarrow$

```
if x_new is legal, return
x_new
else return x
```

Performs the update for each $\mathbf{x_i}$ and collects them to form $\mathbf{x_{new}}$.

Adding Gradient Descent (for Numeric Optimization problems)

- `gradient(self, x, v)`
  - Calculates partial derivatives at **`x`**

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = \frac{f(\boldsymbol{x}') - f(\boldsymbol{x})}{\delta}$$

$$\boldsymbol{x}' = (x_1, \ldots, x_{i-1}, x_i + \delta, x_{i+1}, \ldots, x_d)^T$$

  - Returns the gradient $\nabla f(\boldsymbol{x})$
- `isLegal(self, x)`
  - Checks if **`x`** is within the domain for each x_i in x
- `getAlpha(self)`
- `getDx(self)`

- **`getAlpha`** and **`getDx`** are called from **`displaySetting`** of the main program when reporting the update rate and the increment size for calculating derivative

Adding Gradient Descent (for Numeric Optimization problems)

- Gradient Descent Algorithm (Overall Flow)
  1. Generate a **random starting point** ($\mathbf{x}_{\mathbf{current}}$) and calculate its evaluated result ($\mathbf{v}_{\mathbf{current}}$).
  2. Generate a new $\mathbf{x}_{\mathbf{new}}$ according to the **GD rule** and calculate its evaluated result ($\mathbf{v}_{\mathbf{new}}$).
  3. If $\mathbf{v}_{\mathbf{new}}$ is **better** than $\mathbf{v}_{\mathbf{current}}$, update $\mathbf{x}_{\mathbf{current}} = \mathbf{x}_{\mathbf{new}}$ and $\mathbf{v}_{\mathbf{current}} = \mathbf{v}_{\mathbf{new}}$, and return to **step 2**. Otherwise, terminate the process.

- `Numeric.gradient` method
  - Calculates the partial derivative $\frac{\partial f(\mathbf{x})}{\partial x_i}$ for each x_i and collects them to return the gradient $\nabla f(\mathbf{x})$.
- `Numeric.takeStep` method
  - Uses the $\frac{\partial f(\mathbf{x})}{\partial x_i}$ calculated by the **Numeric.gradient method** to perform the **update for each x_i**, and collects them to return $\mathbf{x}_{\mathbf{new}}$.

variables

functions

Final Class Hierarchy

Numeric

expression domain delta = 0.01	alpha dx = 10 ** (-4)
__init__ setVariables getDelta (new) randomInit evaluate mutants mutate randomMutant describe report coordinate	getAlpha getDx takeStep gradient isLegal

Gradient Descent implementation part

GD Implementation Code File

```
gradient descent.py
```

```
def main():
  p = Numeric()
   ...
def gradientDescent(p):
   ...
def displaySetting(p):
   ...

main()
```

```
def main():
    # Create a Problme object for numerical optimization
    p = Numeric()      # Create a problem object
    p.setVariables() # Set its class variables (expression, domain)
    # Call the search algorithm
    gradientDescent(p)
    # Show the problem and algorithm settings
    p.describe()
    displaySetting(p)
    # Report results
    p.report()
```