Greedy Algorithms

Introduction

The Greedy Heuristic method is one of the most natural and simple algorithms applied to optimization of functions and several other problems. In these algorithms, every iteration makes a decision based on the current information alone, regardless of this decision's effect in future.

Greedy algorithms build a solution piece by piece, in a way similar to Dynamic Programming, choosing the next piece in such a way that the picked piece results in an immediate cost-benefit. The previous decisions are not taken into account, but rather the current evaluation is used. Thus, the algorithm is a natural method for optimization problems. Greedy methods are easy to implement and quite efficient in most of the cases. If the underlying problem is **convex** then the greedy algorithm will result in a **global optimization**. Otherwise, the greedy approach may result in a **local optimum**. Thus, a greedy algorithm is an algorithmic paradigm based on the greedy-choice heuristic that follows local optimal choice at each step, hoping to find a globally optimal solution.

Note that the information about the convex/non-convex property of the problem at hand may not be available at the beginning or during. Thus, a drawback of greedy methods is that the approach might be greedy without the knowledge if the global optimum can ever be reached. As an example in data analytics or application of neural networks, the greedy heuristic optimizers are the main optimization tools, such as the gradient descent optimizer.

Though greedy algorithms do not guarantee a global optimum, they are always of polynomial complexity (or faster) algorithms.

Examples of non-convex problems that an optimum solution is not guaranteed by greedy heuristics (but still applied):

- Boolean Satisfiability (SAT)
- Traveling Salesman
- Graph Coloring
- Vertex Cover
- Hamiltonian Path
- Clique
- 0-1 Knapsack
- Job Scheduling

Convex Optimization Problems

An **optimization problem** has the form,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f_o(x) \\ \text{subject to} & f_i(x) \leq b_i, i = 1, \ldots m \end{array}$$

The vector $x = (x_1, \ldots, x_n)$ is the optimization variable of the problem, the function $f_o : \mathbb{R}^n \to \mathbb{R}$ is the **objective** function, the functions $f_i : \mathbb{R}^n \to \mathbb{R}, i = 1, \ldots, m$ are the **constraint** functions, and the constants $b_1, \ldots, b_m$ are the bounds for the constraints.

A convex optimization problem is one in which the **objective** f_o and **constraint** f_i functions are convex, or in other words, they satisfy the inequality,

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$.

Components of Greedy Algorithms

1. A candidate solution $x_o \in \mathbb{R}^n$
2. A feasibility function to determine whether a candidate can be used to contribute to the solution, i.e. $f_i(x_o) \leq b_i$ hold for $\forall i$
3. A selection function to pick the best candidate to be evaluated as a solution
4. An objective function f_o to assign a value to a solution x_o, i.e. evaluate x_o
5. A solution function to indicate whether a complete solution has been reached

Convex Sets

A set $C \subseteq \mathbb{R}^n$ is convex if $\alpha x_1 + (1 - \alpha)x_2 \in C$, for $x_1, x_2 \in C$ and any $\alpha \in [0, 1]$.

Convex Functions

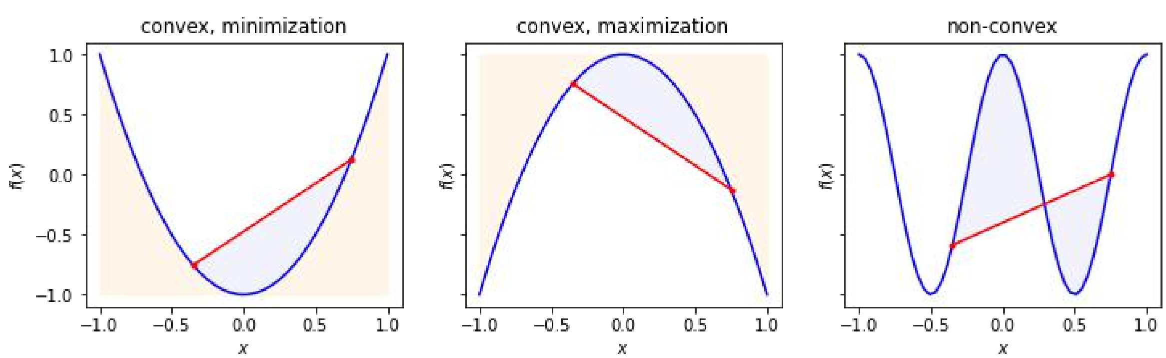
Geometrically, a function f is convex if the line segment drawn from any point (x_1, $f(x_1)$) to another point (x_2, $f(x_2)$) **always** lies on, above (minimization), or below (maximization) the graph of $f(x)$, as demonstrated in the cell below. This line segment is called the chord from x_1 to x_2.

Algebraically, a function f is convex if $f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2), \forall x_1, x_2 \in \mathbf{dom} f$, and $\forall t \in \mathbb{R}$ where $0 \leq t \leq 1$

A function is concave if $-f$ is convex, i.e. if the chord from x_1 to x_2 lies on or below the graph of $f(x)$. A linear

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

In [2]:
```
# plot
def plot(_ax, _x, _y, _x1, _x2, _y1, _y2, _title, region=0):
    _ax.plot(_x,_y, c='b')
    _ax.plot([_x1,_x2], [_y1,_y2], c='r', marker = '.')
    _ax.fill_between(np.linspace(_x1,_x2), f(np.linspace(_x1,_x2)), n
p.linspace(_y1,_y2),
                     color='lavender', alpha=0.5)
    if region == 0:  # minimization
        _ax.fill_between(_x, [min(_y)]*len(_x), _y,
                         color='papayawhip', alpha=0.5)
    elif region == 1:  # maximization
        _ax.fill_between(_x, _y, [max(_y)]*len(_x),
                         color='papayawhip', alpha=0.5)
    _ax.set_ylabel('$f(x)$')
    _ax.set_xlabel('$x$')
    _ax.set_title(_title)


fig, ax = plt.subplots(1, 3, figsize=(12, 3), sharey=True)

# x and f(x)
x = np.linspace(-1,1); x1, x2 = -0.35, 0.75

f = lambda _x: 2*_x**2-1
y = f(x); y1, y2 = f(x1), f(x2)
plot(ax[0], x, y, x1, x2, y1, y2, 'convex, minimization', region=0)

f = lambda _x: -2*(_x**2)+1
y = f(x); y1, y2 = f(x1), f(x2)
plot(ax[1], x, y, x1, x2, y1, y2, 'convex, maximization', region=1)

f = lambda _x: np.cos(2*np.pi*_x)
y = f(x); y1, y2 = f(x1), f(x2)
plot(ax[2], x, y, x1, x2, y1, y2, 'non-convex', region=2)

plt.show()
```



Gradient Descent Algorithm

Given the minimization problem, the minimum of a function $f(x)$, $x \in \mathbb{R}^n$, and $f : \mathbb{R}^n \to \mathbb{R}$. Denote the gradient of f by $g_k = g(x_k) = \nabla f(x_k)$.

The general idea behind most *greedy* optimization methods is to compute a *step* along a given search *direction,* d_k, such as,

$$x_{k+1} = x_k + \alpha_k d_k, \quad k = 0, 1, \ldots,$$

where the step length, α_k, is chosen (steepest descent) so that

$$\alpha_k = \arg\min_{\alpha} \phi_k(\alpha) = \arg\min_{\alpha} f(x_k + \alpha d_k)$$

Here $\arg\min$ refers to the argument of the minimum for the given function. For the gradient descent method, the search direction is given by $d_k = -\nabla f(x_k)$. Then the algorithm:

Given an initial x_o, and $d_o = -g_o$, and a convergence tolerance $\mathbf{tol}$,

```
for k in range(maxiter):
    a[k] = argmin(a) phi[k]
    x[k+1] = x[k] - a[k] * g[k]
    compute g[k+1]
    if norm(g[k+1]) <= tol:
        stop  # converged
```

Gradient descent requires the step length $\alpha_k \in \mathbb{R}^+$ and the gradient computation. One of the disadvantage of gradient descent is shown in the cells below where a convergence is never achieved, due to almost flat f surface causing the derivative being close to zero.

Note that Steepest Descent is a special case of gradient descent where the step length ($\in \mathbb{R}^+$) is chosen to minimize the objective function value. Gradient descent refers to any of a class of algorithms that calculate the gradient of the objective function, then move *downhill* in the indicated direction, so that the step length can be fixed, or estimated (e.g. line search).

In [3]:
```
def plot_func(_ax, _func, _title):
    _ax.plot(np.linspace(-30, 30), _func(np.linspace(-30, 30)), c='b'
)
    _ax.set_ylabel('$f(x)$')
    _ax.set_xlabel('$x$')
    _ax.set_title(_title)
```

In [4]:

```python
MAXITER = 50

# f function to be optimized, x0 initial condition, lr learning rate
 (step), tol to decide convergence
def gdescent(_f, x0, lr, tol=0.00001, binfo=False):
    x = xmin = x0
    e0 = np.abs(_f(x))                                    # LINE 3
    hist = []
    if binfo:  # debug info
        print(f"{'xpos':^9s} {'xmin':^9s} {'f(x)':^9s} {'grad':^11s}
{'learningrate':^9s}")
    for _ in range(MAXITER):
        g = (_f(x+tol)-_f(x))/tol  # gradient,              LINE 8
        # update x by step (learning rate) and direction from g
        x -= lr * (g / np.linalg.norm(g))                 # LINE 10
        # error - function evaluation
        e = _f(x)                                         # LINE 12
        if e > e0:  # check overshoot
            lr -= 0.1*lr  # reduce learning rate,           LINE 14
        else:
            xmin = x  # update the minimum,                 LINE 16
        # history used for debug info and animation
        hist += [x]
        if binfo:  # debug info
            print(f'{x:9.5f} {xmin:9.5f} {e:9.5f} {g:11.7f} {lr:9.5f}
')
        # check flat gradient
        if np.abs(g) < tol:                               # LINE 22
            break
        e0 = e
    return xmin, np.array(hist)
```

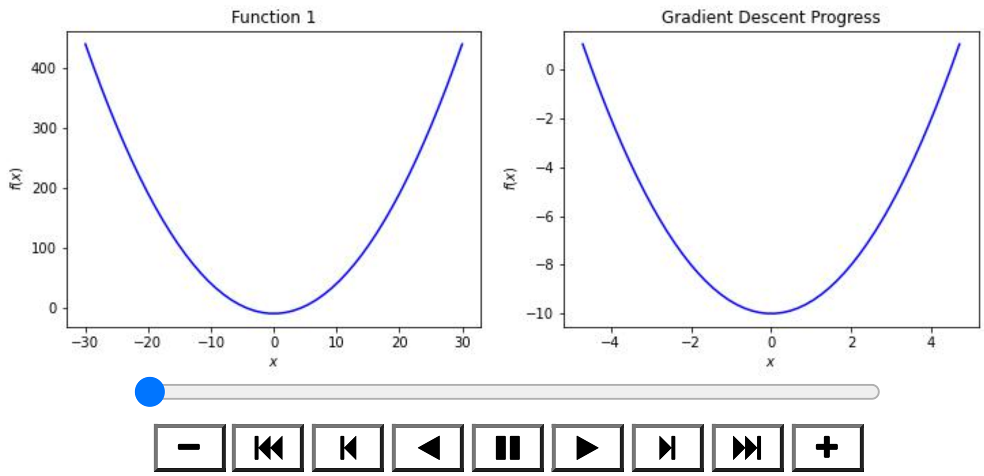
In [5]:
```
import matplotlib.animation as animation
from IPython.display import HTML

# Animate the optimization
def gd_info(_ax, _xpos, _f, _imax):
    x1, x2 = _xpos.min(), _xpos.max()
    x1, x2 = min(x1,-x2), max(x2,-x1)
    w = 0.1*np.linalg.norm([(x2-x1),(_f(x2)-_f(x1))])**0.3
    x = np.linspace(x1, x2); y = _f(x)

    _ax.clear()
    pt = _ax.plot(x, y, c='b')
    _ax.set_ylabel('$f(x)$'); _ax.set_xlabel('$x$'); _ax.set_title('G
radient Descent Progress')

    x1 = _xpos[0]
    i = 1
    while i < _imax and i < len(_xpos):
        x2 = _xpos[i]
        _ax.arrow(x1, _f(x1), x2-x1, _f(x2)-_f(x1), color='r',
                  head_width=w, head_length=2*w, alpha=1)
        x1 = x2
        i += 1
    #
    plt.close()
    #
    return pt


# Use func as function, history hist, and ax globally, to be defined
 later
def animate_gd(_i):
    return gd_info(ax, hist, func, _i)
```

In [6]:
```
# Example function
func = lambda _x: (0.5*_x**2 - 10)

# Gradient descent
x0 = 5; lr = 0.3
xmin, hist = gdescent(func, x0, lr, binfo=False)
print(f'Start from x0={x0}, min achieved at xmin={xmin:.5f}, f(xmin)=
{func(xmin):.5f}')
```

```
Start from x0=5, min achieved at xmin=0.00001, f(xmin)=-10.00000
```

In [7]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
plot_func(ax[0], func, 'Function 1')

ax = ax[1]
ani_gd = animation.FuncAnimation(fig, animate_gd, interval=1000, fram
es=len(hist))
HTML(ani_gd.to_jshtml())
```

Out[7]:



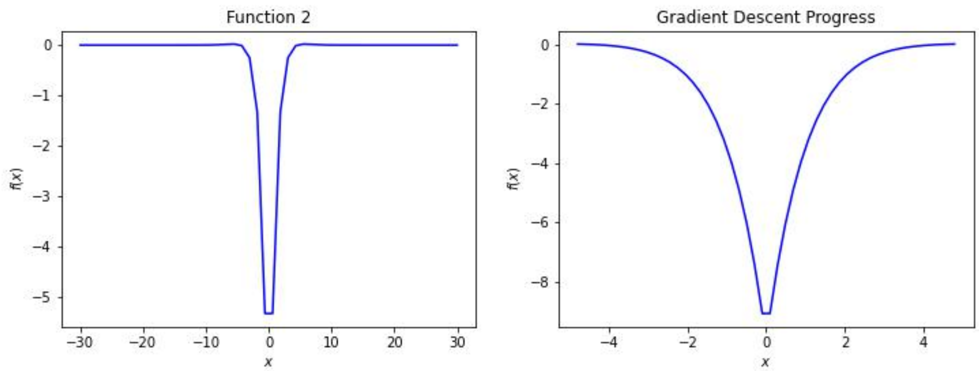
Once Loop Reflect

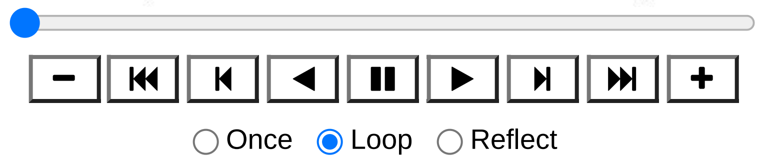
In [8]:
```
func = lambda _x: (0.5*_x**2 - 10) * np.exp(-np.abs(_x))

x0 = 5; lr = 0.2
xmin, hist = gdescent(func, x0, lr, binfo=False)
print(f'Start from x0={x0}, min achieved at xmin={xmin:.5f}, f(xmin)=
{func(xmin):.5f}')
```

```
Start from x0=5, min achieved at xmin=0.00003, f(xmin)=-9.99967
```

In [9]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
plot_func(ax[0], func, 'Function 2')

ax = ax[1]
ani_gd = animation.FuncAnimation(fig, animate_gd, interval=1000, fram
es=len(hist))
HTML(ani_gd.to_jshtml())
```

Out[9]:



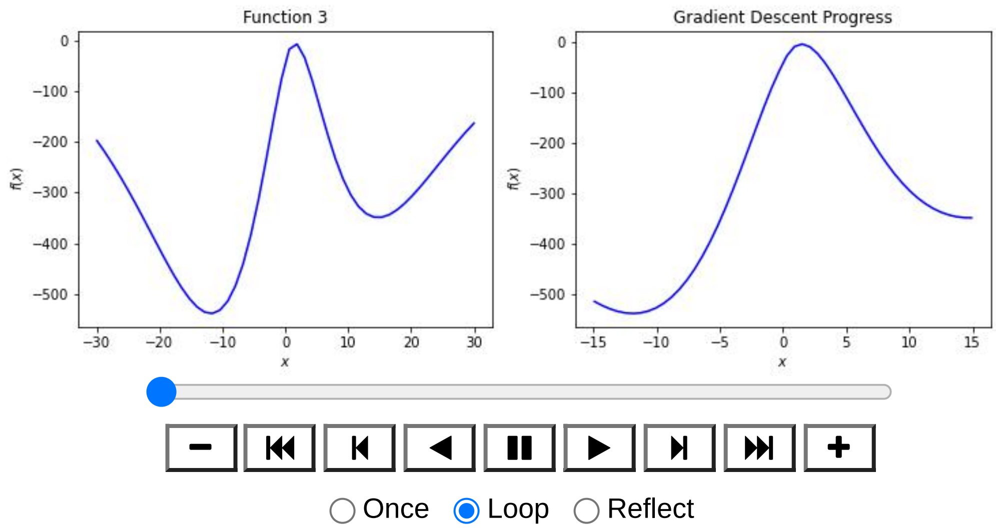


In [10]:
```
func = lambda _x: (2*_x**2 - 10 - 20*(_x-1.3)**2) * np.exp(-0.15*np.a
bs(_x))

x0 = 2; lr = 0.3
xmin, hist = gdescent(func, x0, lr, binfo=False)
print(f'Start from x0={x0}, min achieved at xmin={xmin:.5f}, f(xmin)=
{func(xmin):.5f}')
```

```
Start from x0=2, min achieved at xmin=14.81870, f(xmin)=-349.38260
```

In [11]:
```python
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
plot_func(ax[0], func, 'Function 3')

ax = ax[1]
ani_gd = animation.FuncAnimation(fig, animate_gd, interval=1000, fram
es=len(hist))
HTML(ani_gd.to_jshtml())
```

Out[11]:



Exercises

- Turn the `binfo` to `True` and monitor progress.
- Change the learning rate `lr` to see the behavior of gradient descent.
- Change the initial condition for Function 3 above to
  - `x0` $= 0$
  - `x0` $= -0.1$
  - `x0` $= 0.1$