Continuous Optimization

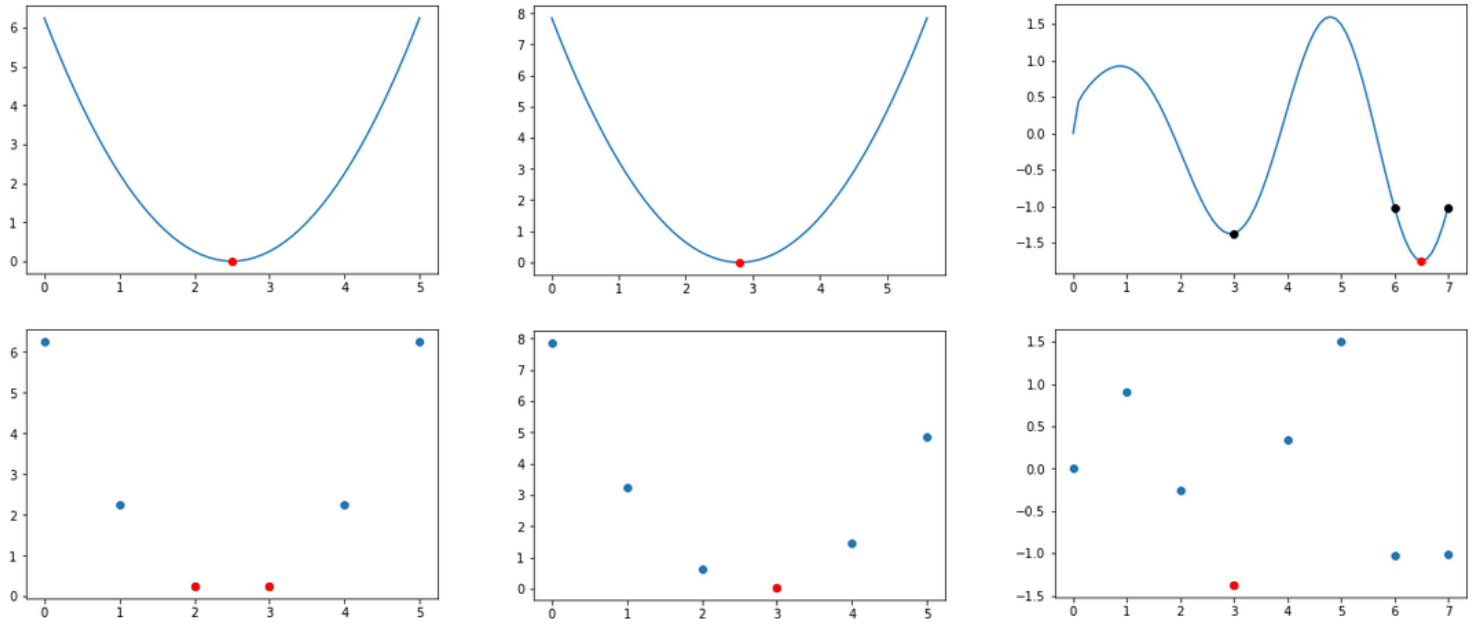
Continuous Optimization

Continuous Optimization Problems: domains are dense (usually subsets of $\mathbb{R}$ or $\mathbb{Q}$)

Continuous Optimization Problems arise in

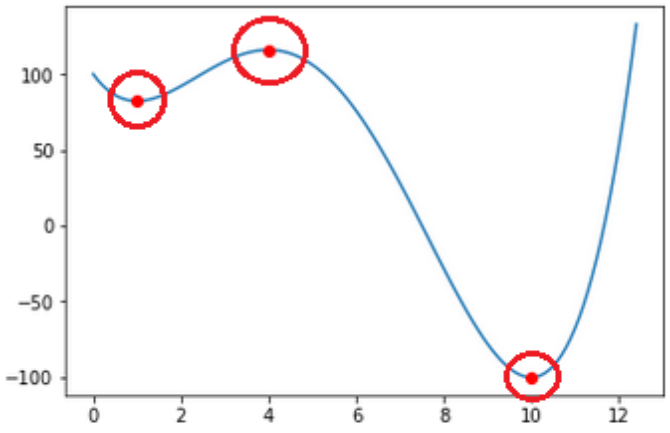
- Machine Learning (minimize loss function)
- Probability Theory (compute probability bounds)
- Natural Sciences (physical and chemical models)
- Engineering (optimal design and control problems)
- Economics (maximize utility)

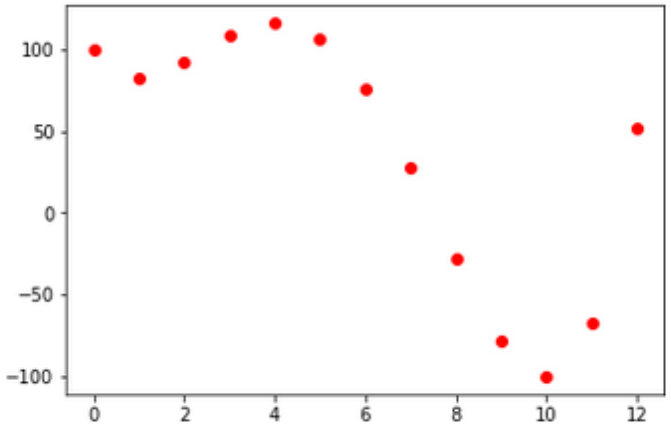
- in continuous optimization, we can move continuously through space
- this makes continuous optimization often easier than discrete optimization



- in n dimensions, the global optimum of a continuous function is surrounded by 2^{n+1} candidates for discrete optima

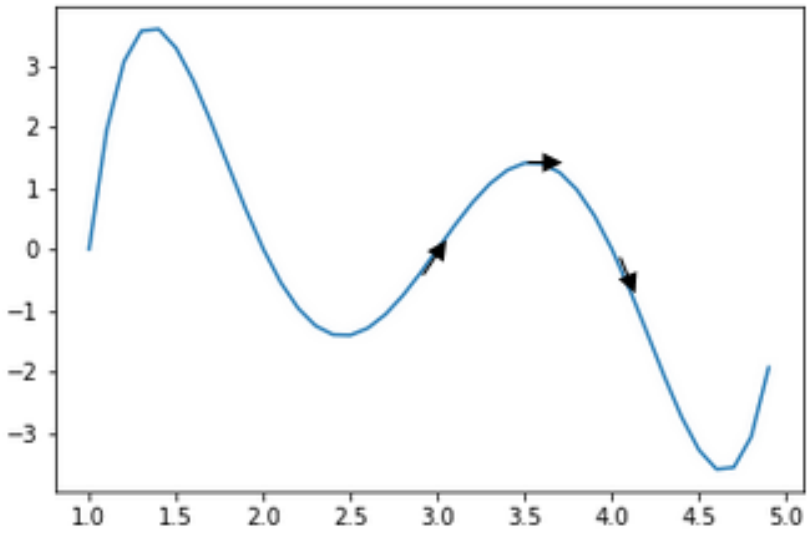
Local Optimality





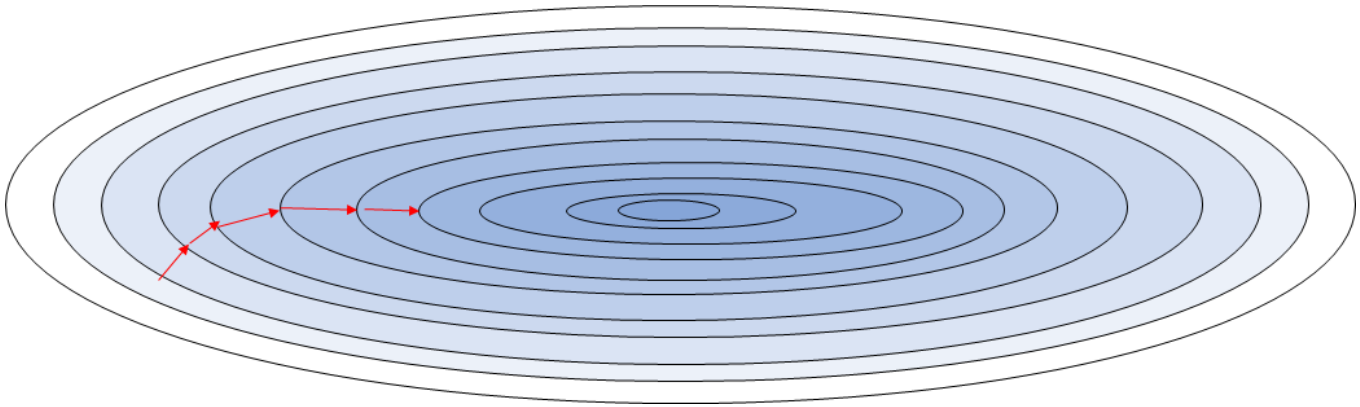
- in discrete spaces, we have to define a neighborhood in order to talk about local optimality
- in continuous spaces, local optimality is naturally defined with respect to ϵ-neighborhoods
- so we can always talk about local optimality of algorithms
- however, in discrete spaces, this notion of optimality becomes meaningless (in $\mathbb{Z}$, every point is locally optimal for $\epsilon = 0.5$)

Gradient



- derivative at a point, corresponds to slope at this point
- in multiple dimensions, gradient corresponds to direction of steepest ascent of function

Gradient-based Optimization



- intuitively, gradient ascent is continuous-space analogue of hill-climbing
- however, instead of looking at the whole neighborhood, gradient can be computed immmediately if it exists

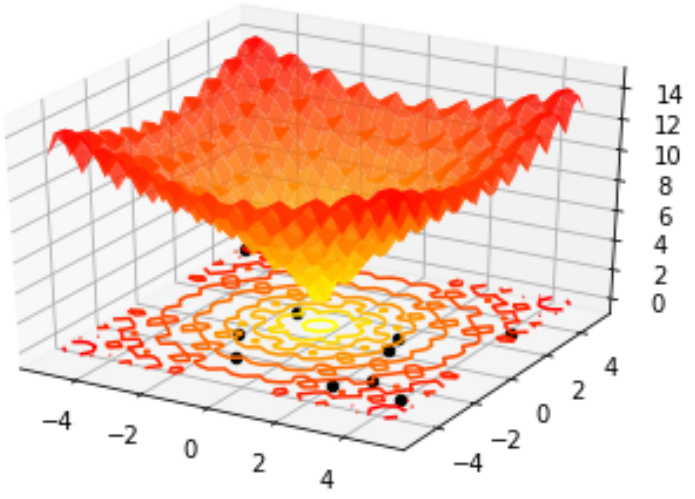
Why Meta-Heuristics?

- Gradient-based algorithms often find local optima quickly
- however, Gradient descent may converge to poor local optima
- we may also be unable to compute gradient
  - does not exist
  - too complicated

Differential Evolution

Differential Evolution: Idea

- high-level idea bears strong resemblance to genetic algorithms
- DE also evolves population of solutions (chromosomes)
- again, adaptation based on mutation and crossover
- actual mechanics are quite different, though



Basic DE Algorithm

population $\leftarrow$ initialize(N)

do

  next_population $\leftarrow \emptyset$

  for each x *in* population

    $v \leftarrow$ generateDonor(population, x) *(Mutation)*

    $u \leftarrow$ generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population $\leftarrow$ next_population

until *termination condition reached*

return best(population)

Initialization

population $\leftarrow$ initialize(N)

do

  next_population $\leftarrow \emptyset$

  for each x *in* population

    $v \leftarrow$ generateDonor(population, x) *(Mutation)*

    $u \leftarrow$ generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population $\leftarrow$ next_population

until *termination condition reached*

return best(population)

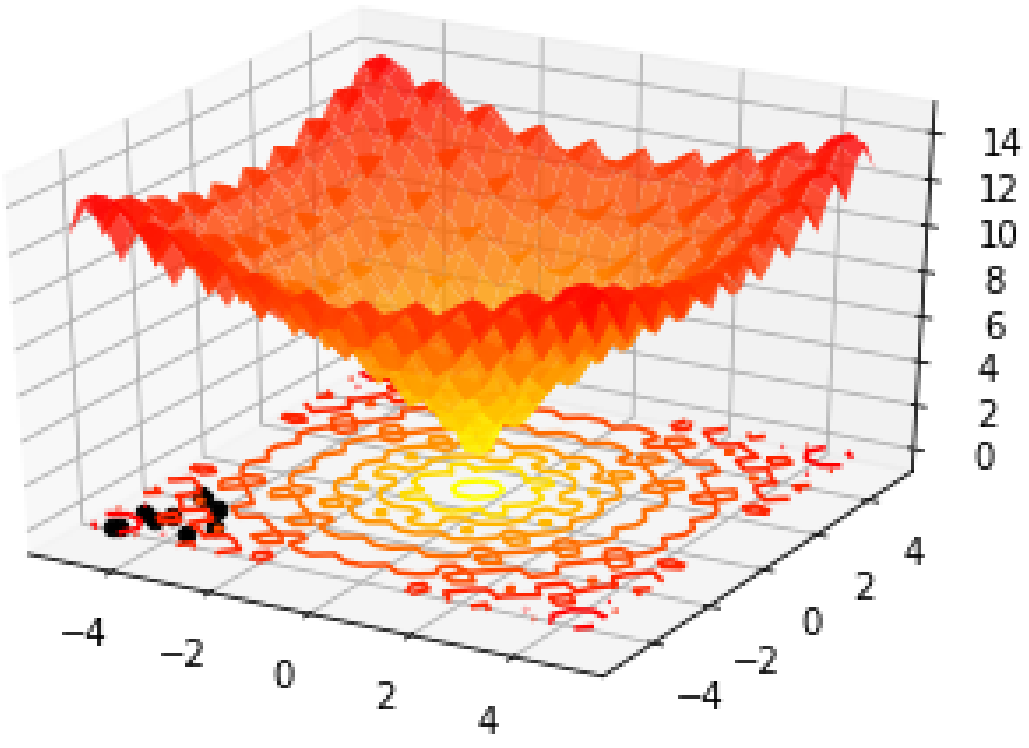
Initialization

- as usual, population size N should not be chosen too small to allow for good exploration
- similarly, initial points should be widely spread over interesting region
- consider range $[l_i, u_i]$ for i-th dimension and initialize x by

$$x_i = l_i + r \cdot (u_i - l_i),$$

where r is a random number from $[0, 1]$

Poor Initialization Example



Donor Generation

population ← initialize(N)

do

  next_population ← ∅

  for each x *in* population

    v ← generateDonor(population, x) *(Mutation)*

    u ← generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population ← next_population

until *termination condition reached*

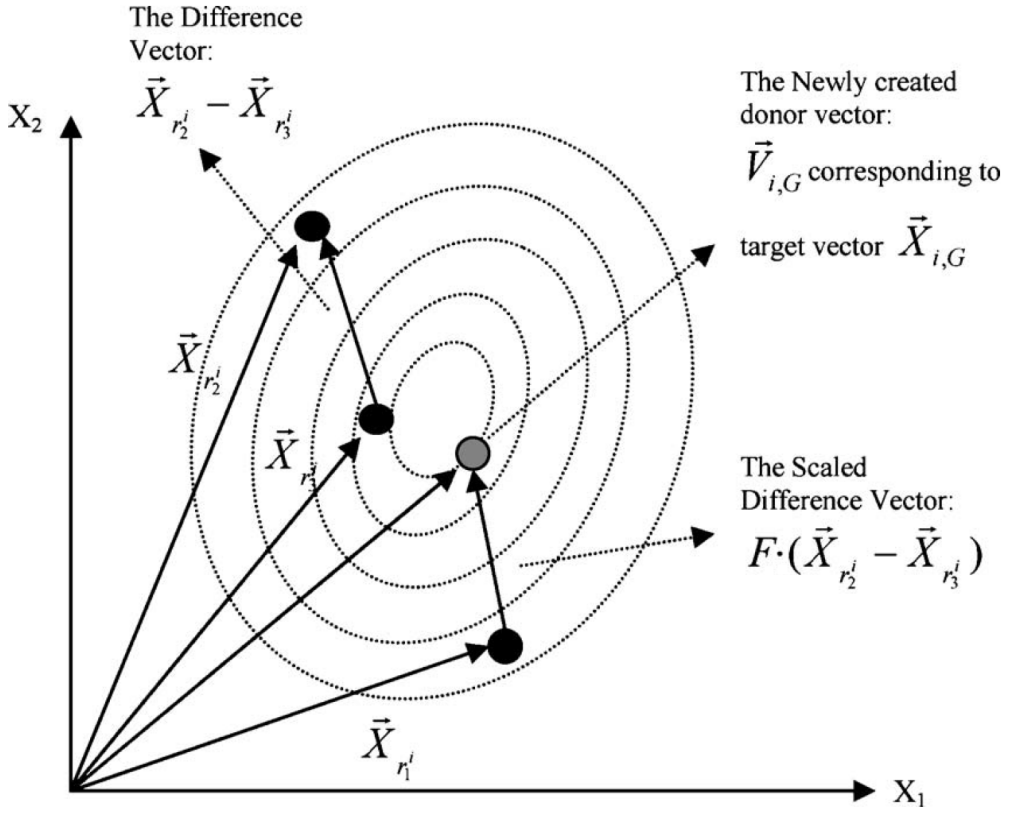
return best(population)

Basic Donor Generation (Mutation)

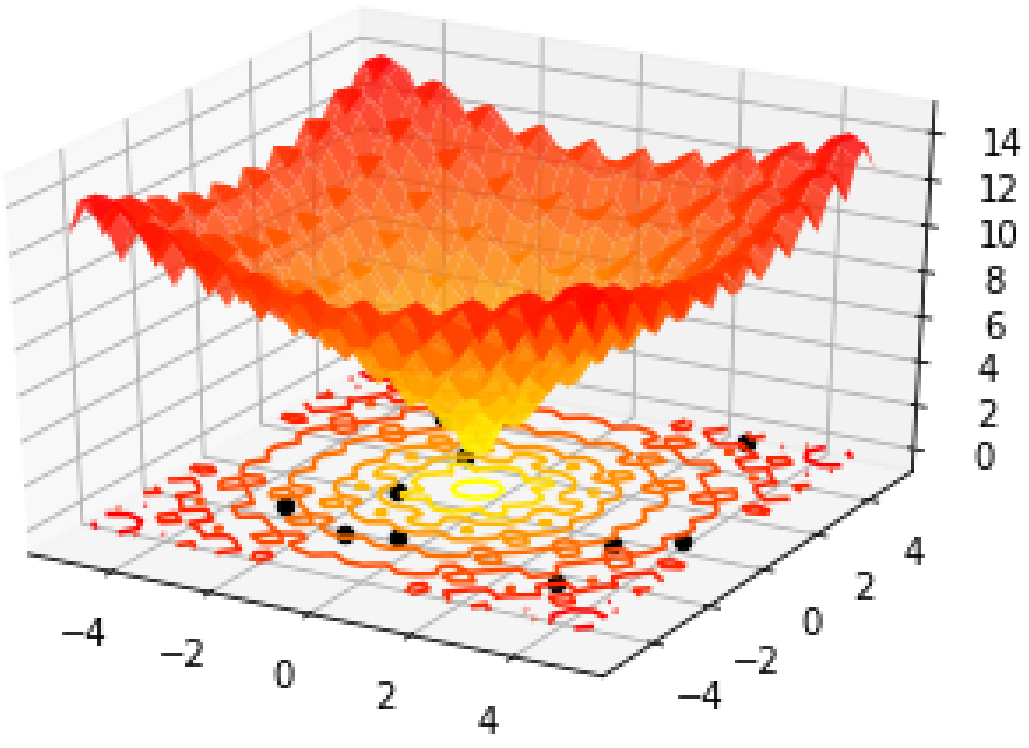
- select three other distinct vectors x_1, x_2, x_3
- donor vector v is defined as

$$v = x_1 + F \cdot (x_2 - x_3),$$

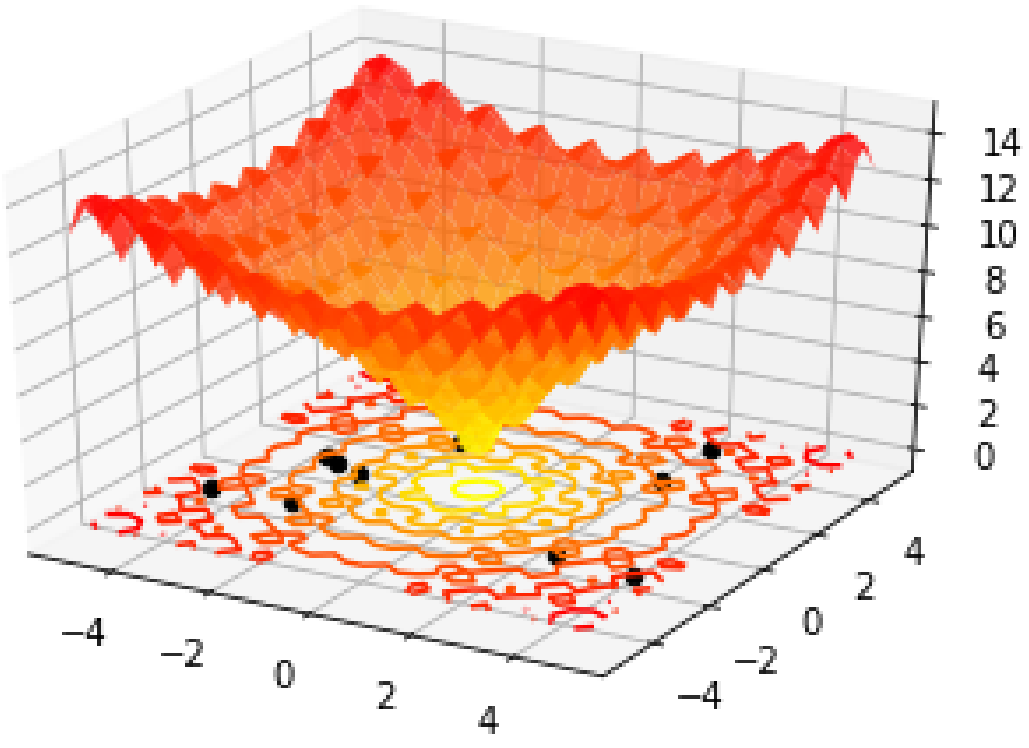
where F is a scaling parameter usually chosen from $[0.4, 1]$



F Too Small Example, F=0.01



F Too Large Example, F=5



Trial Generation

population ← initialize(N)

do

  next_population ← ∅

  for each *x* *in* population

    $v \leftarrow$ generateDonor(population, x) *(Mutation)*

    $u \leftarrow$ generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population ← next_population

until *termination condition reached*

return best(population)

Basic Trial Generation (Binomial/Uniform Crossover)

- when in $\mathbb{R}^D$, pick random number r from $\{1, \ldots, D\}$
- set $u_r = v_r$ (pick at least one element from donor)
- for all other dimensions j, independently set

$$u_j = \begin{cases} v_j & \textit{with probability } C \\ x_j & \textit{with probability } 1 - C, \end{cases}$$

  where $C \in [0, 1]$ is the crossover parameter

- extreme cases:
  - $C = 0$: only $u_r = v_r$ (perhaps too stable)
  - $C = 1$: $u = v$ (perhaps too unstable)

Selection

population ← initialize(N)

do

  next_population ← ∅

  for each x *in* population

    v ← generateDonor(population, x) *(Mutation)*

    u ← generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population ← next_population

until *termination condition reached*

return best(population)

Selection Alternatives

Basic Selection

- select u unless it is worse than x

Other selection schemes are similar to genetic algorithms

- fitness-proportionate selection
- tournament selection
- truncated selection
- ...

Termination

population $\leftarrow$ initialize(N)

do

  next_population $\leftarrow \emptyset$

  for each x *in* population

    $v \leftarrow$ generateDonor(population, x) *(Mutation)*

    $u \leftarrow$ generateTrial(x, v) *(Crossover)*

    next_population.add(select(x, u))

  population $\leftarrow$ next_population

until *termination condition reached*

return best(population)

Termination Conditions

as for genetic algorithms, you may consider

- maximum number of generations
- maximum number of unchanged generations
- maximum number of non-improving generations
- time limit
- fitness threshold reached

Parameters

population ← initialize(N)

do

  next_population ← ∅

  for each *x* *in* population

    v ← generateDonor(population, *x*, *F*) *(Mutation)*

    u ← generateTrial(*x*, *v*, *C*) *(Crossover)*

    next_population.add(select(*x*, *u*))

  population ← next_population

until *termination condition reached*

return best(population)

- population size N
- scaling factor F
- crossover rate C

The Familiy DE/x/y/z

Donor Generation Variants

- our basic donor generation looked like this

$$v = \underbrace{x_1}_{b} + \underbrace{F \cdot (x_2 - x_3)}_{s}$$

- more generally, we can consider a base vector b and a shift vector s and define the donor as

$$v = b + s$$

- there are several options for creating
  - the base vector b
  - the shift vector s

Base Vector Variants

- Random: b is a random vector from population
- Best: b is the best vector x_b from population
- Target-To-Best: b is the target (the current) vector shifted in the direction of the best vector x_b

$$b = x + F \cdot (x_b - x)$$

Shift Vector Variants

- instead of considering one direction, we can consider k directions
- pick $2k$ random vectors $x_1, \ldots, x_{2k}$ from population and let

$$s = F \cdot \sum_{i=1}^{k} (x_{2i-1} - x_{2i})$$

- other variants:
  - (dis)allow target vector in sample
  - sample with(out) replacement

Exponential (Two-Point Modulo) Crossover

- start crossover at random index r between 1 and D
- pick element at index r from donor
- keep picking elements from donor until random number from $[0, 1]$ is greater than C
- remaining elements are taken from target

The Familiy DE/x/y/z

The Family DE/x/y/z is defined by

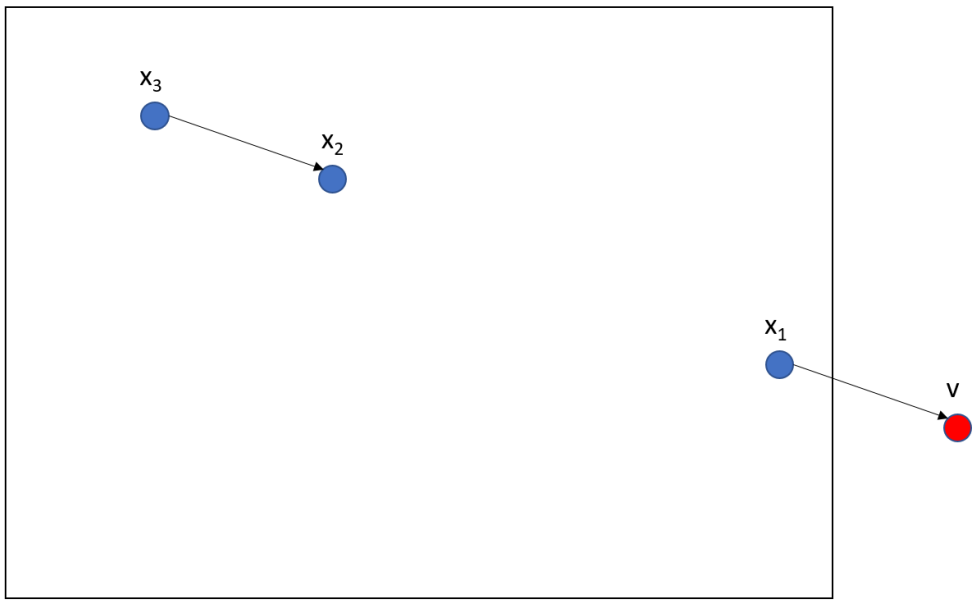
- base vector scheme x
  (Rand/Best/Target-To-Best)
- number of shifting directions y
- crossover scheme z (bin/exp)

Our basic algorithm corresponds to DE/Rand/1/bin

Constrained Optimization

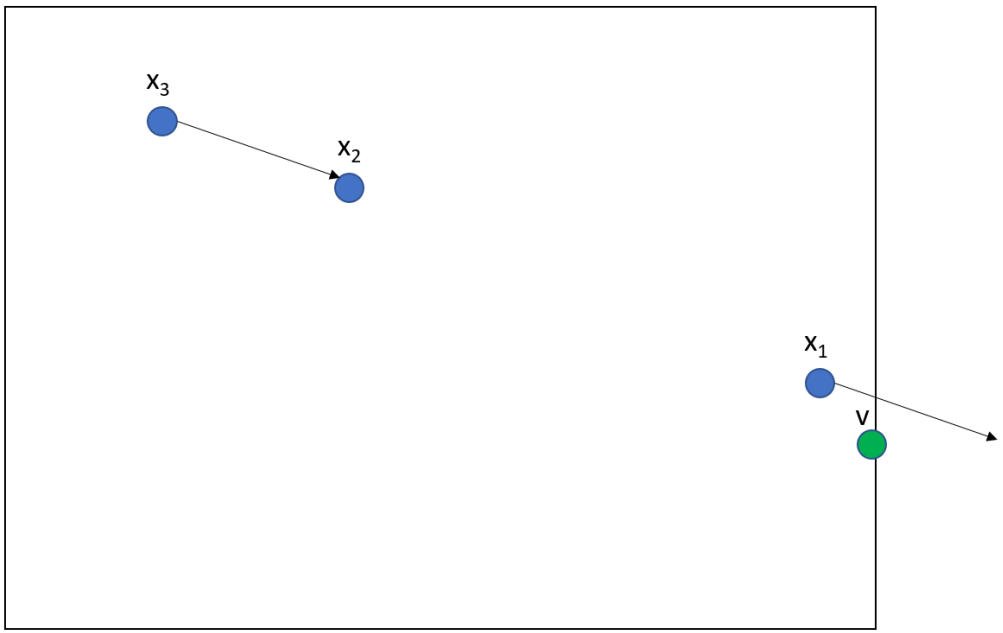
Bound Constraints

- often, some variables must take values in interval $[l_i, u_i]$
  - Non-negativity of values
  - simple resource constraints
- such constraints are called bound constraints
- donor may violate constraints



Cutoff

- there is a simple solution for this problem
- Cutoff: if $v_i < l_i$, set $v_i = l_i$
  if $v_i > u_i$, set $v_i = u_i$



More General Constraints

- what about more general constraints like

$$\begin{pmatrix} \sum_{i=1}^{D} c_i \cdot x_i \\ cos(x_2) \\ \exp(0.1 \cdot x_1 + 0.2 \cdot x_2) \end{pmatrix} \leq \begin{pmatrix} 3 \\ 0.5 \\ 10 \end{pmatrix}$$

- the optimization literature provides sophisticated ways to deal with general constraints by adapting the objective function

- for DE, it can be sufficient to modify crossover procedure:
  - if trial u is infeasible
  - replace component of trial u with random component of target x until u is feasible (sample without replacement)

  however, this may or may not work well

Programming Task Differential Evolution

Power Plant Optimization

- suppose, you work for an electric utility service provider
- you can generate energy with different plane types with different costs
- you can sell energy at different markets at different prices (e.g. business, consumer)



- Problem: determine how much energy to generate with which plane type, how much energy to sell to which market and at what price in order to maximize profit

High-Level Profit Model

- Profit = Revenue - Cost

High-Level Profit Model

- Profit = Revenue - Cost
- Revenue = SoldQuantity * Price

High-Level Profit Model

- Profit = Revenue - Cost
- Revenue = SoldQuantity * Price
- Cost = ProductionCost + PurchasingCost

High-Level Profit Model

- Profit = Revenue - Cost
- Revenue = SoldQuantity * Price
- Cost = ProductionCost + PurchasingCost
- Production Cost = GeneratedQuantity * CostFactor

High-Level Profit Model

- Profit = Revenue - Cost
- Revenue = SoldQuantity * Price
- Cost = ProductionCost + PurchasingCost
- Production Cost = GeneratedQuantity * CostFactor
- Purchasing Cost
  = max(SoldQuantity - GeneratedQuantity, 0) * CostPrice

High-Level Profit Model

- Profit = Revenue - Cost
- Revenue = SoldQuantity * Price
- Cost = ProductionCost + PurchasingCost
- Production Cost = GeneratedQuantity * CostFactor
- Purchasing Cost
  = max(SoldQuantity - GeneratedQuantity, 0) * CostPrice

(if you sell more energy than you can provide, you have to buy energy from another provider)

Plant Cost Model

We describe each plant type by three parameters

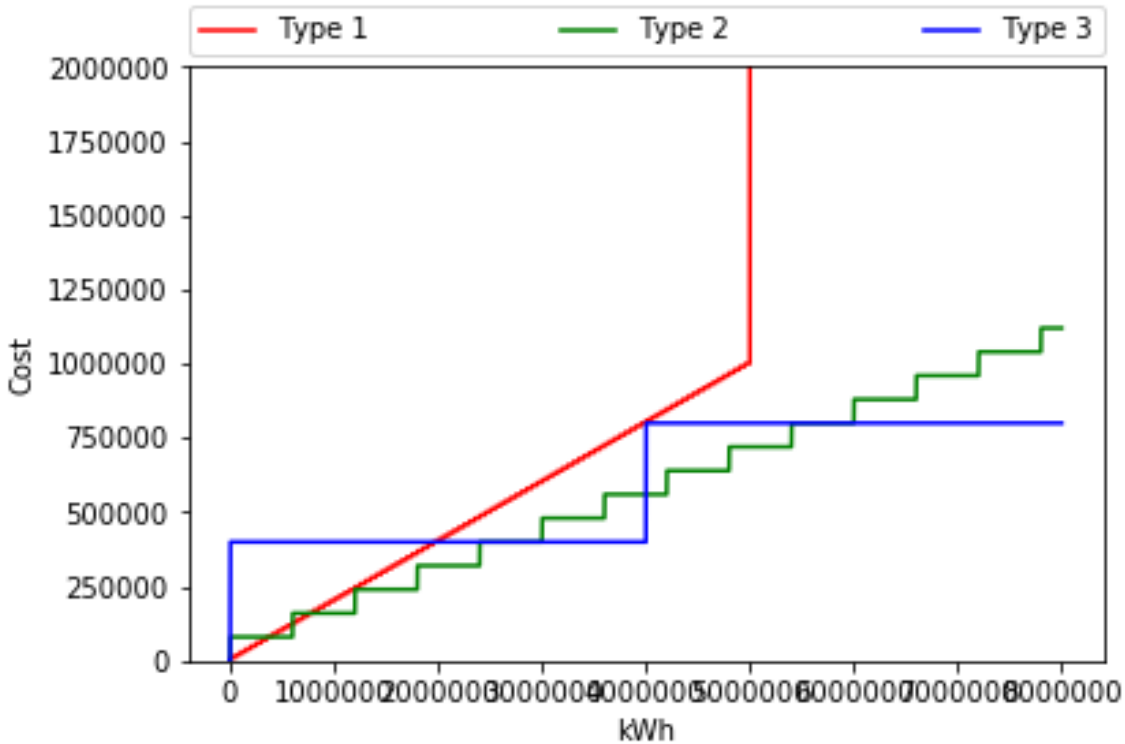
- k - kWh per plant
- c - cost per plant
- m - maximum number of plants that can be used

```
def cost(x, kwhPerPlant, costPerPlant, maxPlants):

    #if x is non-positive, return 0
    if(x <= 0):
        return 0

    #if x is greater than what can be generated return prohibitively large value
    if(x > kwhPerPlant * maxPlants):
        return LARGE

    #otherwise determine number of plants needed to generate x
    plantsNeeded = math.ceil(x / kwhPerPlant)

    #cost is number of plants needed times cost per plant
    return plantsNeeded * costPerPlant
```

Plant Types

We consider three plant types

- Type 1: k = 50,000, c = 10,000, m = 100
- Type 2: k = 600,000, c = 80,000, m = 50
- Type 3: k = 4,000,000, c = 400,000, m = 3



Market Model

We describe each market by two parameters

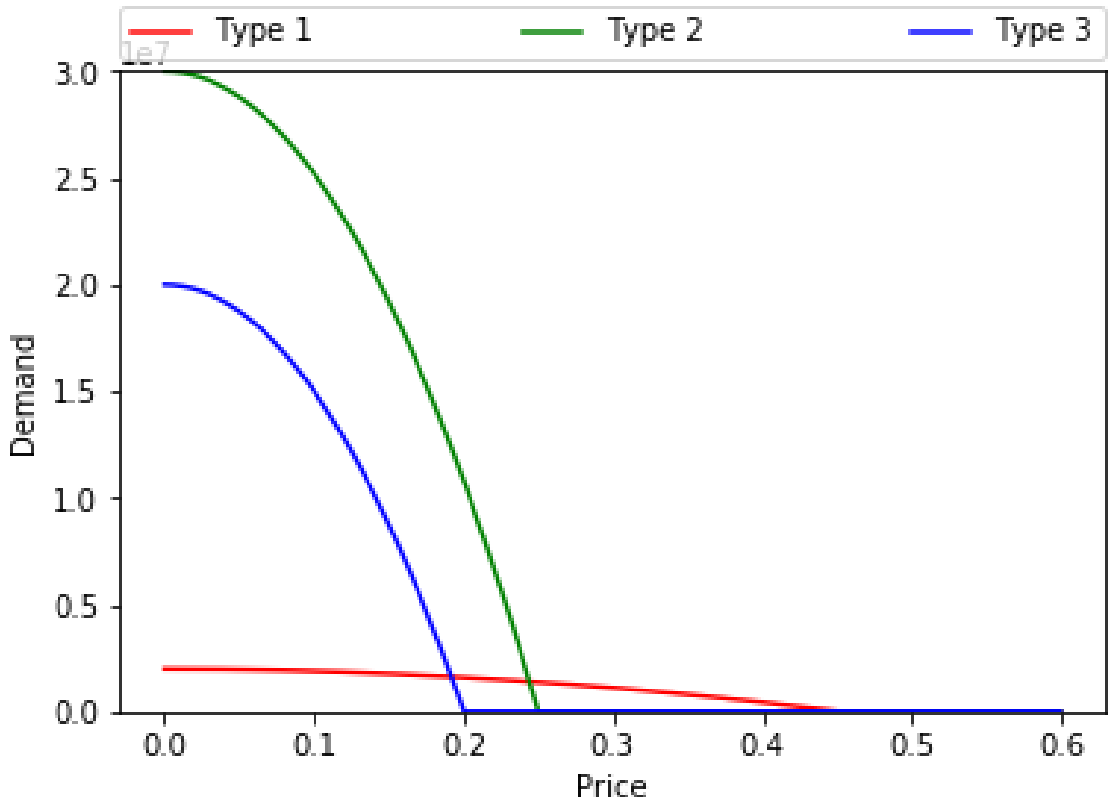
- p - maximum price at which customers buy
- d - maximum demand

```
def demand(price, maxPrice, maxDemand):

    #if price is greater than max price, return 0
    if(price > maxPrice):
        return 0

    #if product is free return maxDemand (ignore negative price)
    if(price <= 0):
        return maxDemand

    #else determine demand based on price
    demand = maxDemand - price**2 * maxDemand / maxPrice**2

    return demand
```

Market Types

We consider three market types

- Type 1: p = 0.45, d = 2,000,000
- Type 2: p = 0.25, d = 30,000,000
- Type 3: p = 0.2, d = 20,000,000



Profit Model

9 Variables $(e_1, e_2, e_3, s_1, s_2, s_3, p_1, p_2, p_3)$

1. e_1, e_2, e_3: energy produced with plants of type i
2. s_1, s_2, s_3: planned amount of energy sold to market of type i
3. p_1, p_2, p_3: price for market of type i

Enforce non-negativity of all variables (and perhaps other bounds)

Profit model

- Profit = Revenue - Cost
- Revenue = $\sum_{i=1}^{3} \min\left(demand_i(p_i), s_i\right) \cdot p_i$
- Cost = ProductionCost + PurchasingCost
- Production Cost = $\sum_{i=1}^{3} cost_i(e_i)$
- Purchasing Cost = $\max\left(\left(\sum_{i=1}^{3} s_i\right) - \left(\sum_{i=1}^{3} e_i\right), 0\right) \cdot 0.6$

DE Programming Task

- solve Power Plant Problem using Basic DE
- as usual, decompose your implementation
  - Initialization (parameter for population size)
  - Donor Generation (parameter for scale factor)
  - Trial Generation (parameter for crossover rate)
  - Selection
- document what scheme you use for which component (try several if you have time)
- divide work among group members

DE Programming Task

- make some experiments with different parameter settings
- document your findings and prepare a small presentation (5-10 minutes)
- upload a compressed archive containing
  1. slides (structure findings in table or other visualization)
  2. source files/ notebook
  3. assignment of tasks to group members

Remarks

- there are other (and perhaps better) ways to model this problem equivalently
- if you have a better idea, feel free to try an alternative model
- if you have time, also try different (and more) plant and market types
- note that you are completely flexible in the choice of your objective function (non-linear, non-differentiable, ...)

Some Results

Problem	Plants			Markets		Cost Price	Profit
	k	c	m	p	d		Achieved
P1	50k	10k	100	0.45	2M	0.6	1,514,312
	600k	80k	50	0.25	30M		
	4M	400k	3	0.2	20M		
P2	50k	10k	100	0.45	2M	0.1	1,818,406
	600k	80k	50	0.25	30M		
	4M	400k	3	0.2	20M		
P3	50k	10k	100	0.5	1M	0.6	404,041
	600k	80k	50	0.3	5M		
	4M	400k	3	0.1	5M		