nsga2 Manual

Important note: The nsga2 code depends entirely on the inputs given inside ip.py file. Be sure to check every input before running the simulations.

Note: indices in python start form 0 and not 1.

Basic instruction

Define the ip.py file and just run the nsga2.py file for generating results.

Extracting information from simulations.

Each generation creates matepop and updates oldpop as matepop. To access the final population, take oldpop and access the attributes.

Examples:

`oldpop.ind` gives list of individuals.

`oldpop.ind[0].xreal` gives array of real variables of individual 1

`oldpop.ind[3].fitness` gives array of fitness values of individual 4

Use list comprehensions to get entire list of DVs or fitness functions

```
dv = [oldpop.ind[i].xreal for i in range(popsize)]
fit = [oldpop.ind[i].fitness for i in range(popsize)]
```

Alternatively, use vstack to get arrays.

```
dv = oldpop.ind[0].xreal.T
for i in range(1,popsize):
      dv = np.vstack(dv, oldpop.ind[i].xreal.T)

fit = oldpop.ind[0].fitness.T
for i in range(1,popsize):
      fit = np.vstack(fit, oldpop.ind[i].fitness.T)
```

Append fit/dv in each generation into a list to keep track of all the solutions that are generated.

Detailed information on all the files.

ip

Parameters

nfunc : *int*
Number of objective functions to minimize
ncons : *int*
Number of constraints
func_name : *str*
Name of function for plots
nvar : *int*
Number of real variables
nchrom : *int*
Number of binary variables
lim_r : *ndarray*
Limits of real variables
lim_b : *ndarray*
Limits of binary variables
integers : *bool*
True if decision variables are integers.
popsize : *int, default =100*
Size of population
gener : *int, default = 100*
Number of generations
ans : *int*
Rigidness of limits. Use 1 if limits are specified. Else, use 0
seed : *float*
Seed for random number generation. (To reproduce results)
optype : *int, default =1*
Type of binary crossover. Use 1 for single point crossover, 2 for uniform crossover
vlen : *ndarray*
Length of each binary variable as array.
chrom : *int*
Total length of binary chromosome. sum(vlen)
pmut_r : *float*
Probability of real mutation. Use a value between 0 and 1/nvar
pmut_b : *float*
Probability of binary mutation. Use a value between 0 and 1/chrom
pcross : *float, default =1*
Probability of crossover. Use a value between 0.5 and 1.0
di : *int, default = 15*
Distribution index for SBX crossover operator (Real coded crossover). Use a value between 0.5 and 100
dim : *int, default = 20*
Distribution index for real coded mutation. Use a value between 0.5 and 500.
tol : *float, default = 1e-6*
Tolerance level for constraint violations.

Functions

f : input - x (ndarray)
Function which evaluates the objective functions.
Returns the objective values as an array.
Note: All the functions need to be coded as minimization functions. i.e., negate the maximization functions if any

cstr : input - x (ndarray)
Function which evaluates the constraint functions.
Returns the constraint values as an array
Note: All the constraints need to be coded as $g(x) \geq 0$

Note: Return functions and constraints as np.array([f1,f2,f3]) or np.array([c1,c2,c3])
You can specify nfunc, ncons, nvar, nchrom, lim_r and lim_b along with these functions.

Classes

individual( )

This class represents a solution candidate and related properties.

Attributes

- ***genes* : *(ndarray)***
  Array which stores the genes of the binary chromosome
- ***xreal* : *(ndarray)***
  Array which stores the real variables
- ***xbin* : *(ndarray)***
  Array which stores the decoded binary variables
- ***fitness* : *(ndarray)***
  Array which stores the values of objective functions
- ***constr* : *(ndarray)***
  Array which stores the constraint values
- ***error* : *(float)***
  Total constraint violation
- ***S* : *(list)***
  Set of dominated individuals
- ***n* : *(int)***
  Number of dominating individuals

population(popsize)

This class represents a population of solutions and related properties.

Attributes

- ***ind* : *(list)*** List of individual class instances (length = popsize)
- ***rankno* : *(ndarray)*** Array which stores the number of individuals in each rank level
- ***maxrank* : *(int)*** Array which stores the genes of the binary chromosome
- ***rankrat* : *(list)*** fraction of number of individuals in a rank level
- ***nondom* : *(ndarray)*** Array which stores the constraint values

init

Initiates binary coded population

Import from ip

- ***popsize***
- ***chrom***
- ***seed***

Input: pop (instance of population class)

Returns : pop *(population updated with binary coded initial population)*

With a probability of 0.5, genes of the total binary chromosome(chrom) of each individual in a population is filled with either 0 or 1 to generate initial population.

For probability 0.5, a random number between 0 and 1 is generated.

A simple algorithm is followed.

```
for all individuals in the population:
      for all genes in an individual:        # Fill until chrom
            if random>0.5:
                  pop.ind[i].genes[j] = 1
            else:
                  pop.ind[i].genes[j] = 0
return pop
```

decode

Decodes the binary codes to real variables.

Import from ip

- ***popsize***
- ***nchrom***
- ***lim_b***
- ***vlen***
- ***integers***

Input: pop (instance of population class)

Returns : pop *(population updated with decoded binary variables)*

Binary variables are stored as a single long chromosome. The length of each variable is stored in the array, vlen. The single chromosome is decoded sequentially with the lengths form vlen.

The decoded value x_i is given by,

$$x_i = x_i^{min} + \frac{x_i^{max} - x_i^{min}}{2^{l_i} - 1} DV(s_i)$$

l_i → string length

x_i^{max}, x_i^{min} → limits of binary variable x_i

$DV(s_i)$ → decoded value of string s_i

Note: The chromosome is all strings s_i, together as 1 string.

In this case, length of each s_i is stored in vlen

```
for all individuals in the population:
        for i in range nchrom:
                # This is DV(si) in the equation.
                sequentially decode the chromosome with length vlen[i]
                pop.ind[i].xbin[j] = final value using the equation
return pop
```

Note: the code uses a variable genlen. This variable ensures the code traverses the genes.

realinit

Initiates real coded population

Import from ip

- ***popsize***
- ***nvar***
- ***ans***
- ***lim_r***
- ***seed***

Input: pop (instance of population class)

Returns : pop *(population updated with real coded initial population)*

Initiates real coded population

For each variable of an individual, a random number 'a' (between 0 and 1) is generated. If the limits are rigid (ans=1), then the variables are calculated as follows

$$x_i = a * (x_i^{max} - x_i^{min}) + x_i^{min}$$

If the limits are not rigid, the random number is normalized between -1 and 1 {using 2*a-1). Now an inverse is taken to generate variables with real values.

```
for all individuals in the population:
        for all real variables:
                a = Generate random
                if ans == 1:
                        pop.ind[i].xreal[j] = calculated value
                else:
                       pop.ind[i].xreal[j] = 1/(2a – 1)
return pop
```

ranking

Import from ip

- ***tol***
- ***ncons***

indcmp

Compares 2 individuals for domination.

Input

- **fit1 (*ndarray)***
  Fitness values of 1st individual
- **fit2 (*ndarray)***
  Fitness values of 2nd individual

Returns

- **dom (*int, default =3)***
  Returns either 1,2 or 3.
  1 represents that fit1 dominates fit2.
  2 represents that fit2 dominates fit1
  3 represents mutually non dominating solutions

If constraints are involved, the if condition at the start checks for dominance based on errors. The solutions with more error are dominated by ones with lesser error.

A Boolean operator is used to check dominance. This function returns 3 by default. If either one of the solutions dominates the other, it returns either 1 or 2.

Python makes it simple to use Boolean operators on arrays as it returns Boolean arrays. A simple call of (fit1<=fit2) returns `True` or `False` at each location as an array. Now Boolean arrays can be summed up because `True` counts as 1 and `False` counts as 0.

After comparison, if the `sum == len(fit1)`, this means that the comparison operator is True at every location. Now if atleast one of the `fit1` is strictly less than `fit2`, `fit1` dominates `fit2`. This is the 2nd if condition.

```
if sum(fit1<=fit2) == len(fit1):
        if sum(fit1<fit2) > 0:
                dom = 1
```

dom is assigned 2 for the reverse case.

Note: This can be modified to `sum(fit1-fit2<=1e-6)` *to add tolerance.*

ranking

Ranks population on unconstrained problems

Input

- **pop (*population instance)***
- **popsize (*int)***
- **ret_arr *(bool, default =* `False`*)***
  pass `True` if rankarr should be returned

Returns

- **pop (*default*)**
  Population updated with ranks
- **rankarr *(if ret_arr = True)***
  Array representing individuals in each rank level in its rows.
  Note: `rankarr.size = (popsize,popsize)`
  Example – Say, there are 6 individuals. `[1,3,6]` *are rank 1.* `[4, 5]` *are rank 2 and* `2` *belongs to rank 3. The rankarr generated will be,*

```
array([[1., 3., 6., 0., 0., 0.],
       [4., 5., 0., 0., 0., 0.],
       [2., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

  This gives us a detailed map of all the individuals in different ranks.
  The individuals in rankarr start from 1 and not 0. To access a particular individual use `pop.ind[i-1]` *and not* `pop.ind[i]`

Note: For the above rankarr, the population attribute, rankno will be:

```
array([[3., 2., 1., 0., 0., 0.],
```

This is merely the number of individuals in each rank level

ranking generates 2 attributes for all individuals to finally calculate ranks. These are S and n mentioned in the individual class before. S will be a set/list which stores the individuals dominated and n stores the number of solutions that dominate a particular individual.

Initialize ranks, S and n of all individuals.
Initialize rankarr as a zero array.
Temparr will store the pareto fronts at each level.

Loop 1
Compare 2 individuals ind1 and ind2.

- If ind1 dominates ind2, add ind2 to S of ind1
- If ind2 dominates ind1, increase n of ind1 by 1.
- If mutually non dominating, continue

```
for each ind1 in population:
	for each ind2 in population:
		if unequal:
			val = indcmp(fit1,fit2)
			if val == 1:
				pop.ind1.S.append(ind2)
			if val == 2:
				pop.ind1.n += 1
	# If no ind2 dominates ind1, n will be 0 and this will belong to rank 1
	if pop.ind1.n = 0:
		pop.ind1.rank = 1
```

```
            temparr.append(ind1)
```

After this loop is completed, `temparr` will have the 1st pareto front.

Now, simply replace 1st row of `rankarr` by `temparr`.

`pop.rankno[0]` is also updated by `len(temparr)`

Loop 2

This loop finds all the other rank levels. We simply take the individuals in the 1st front, then take the individuals in their list **S**.

Now reduce n by 1 for all these individuals. If n comes out to be zero, they belong to the next front. This individual gets filled to `Q`. At the end `temparr` is updated as the new front which is `Q`. This is repeated until we find all the fronts.

The stopping criterion is decided using a while loop.

```
k =1
while temparr != [] and k<popsize:
       Q = []
       for each ind1 in temparr:
           for each ind2 in pop.ind1.S:
                 pop.ind2.n -= 1       # decrease n by 1

                 if pop.ind2.n == 0:   # Belongs to the next front.
                     Q.append(ind2)
                     pop.ind2.rank = k+1   # Update rank
       k = k+1
       temparr = Q     # new front
```

Now, simply replace `k-1`th row of `rankarr` by `temparr`.

`pop.rankno[k-1]` is also updated by `len(temparr)`

Finally, Update maxrank attribute of population by the highest rank.

func_con

Evaluates function values and constraint values

Import from ip

- ***popsize***
- ***ncons***
- ***f***

- *if ncons>0, import cstr*

Import form ranking

- *ranking*

Input: pop (instance of population class)

Returns : pop ***(population updated with real coded initial population)***

The real and binary variables are stacked to get a single array of all variables. This is passed to **f** and **cstr** (if consrained)

Constraint violations. Since constraints are of the form $g(x) \geq 0$, all negative values will be considered as errors and the sum of all negative values will be taken as the error for the individual.

Python makes it simple to use Boolean operators on arrays as it returns Boolean arrays. Error is calculated by using a Boolean operation on the array of constraint values. All those positions with negative values are summed up to get the total error. This is done using **np.any**

```
for all individuals in population:
	xx = horizontalstacking(pop.ind[i].xreal,pop.ind[i].xbin)
	pop.ind[i].fitness = f(xx)	# Evaluate fitness array using f

	if ncons>0
		pop.ind[i].constr = cstr(xx)
		violations = (pop.ind[i].constr < 0)
		# This returns a Boolean array
		if any in (violations) is True
			pop.ind[i].error = -sum(pop.ind[i].constr[violations])
if ncons>0:
	rankcon(pop,popsize)
else:
	ranking(pop,popsize)
```

Note: Initialize or reset highest ranks, errors of the population back to zero.

sselect

Tournament selection module

Import from ip

- *population*
- *popsize*
- *seed*

Input: curr_pop (instance of population class)

Returns: sel_pop *(selected population)*

Create a new instance of population (sel_pop). This population will be filled with winner candidates from tournament selection.

Select 2 random individuals from curr_pop using random indices.
Compare ranks and crowding distances(if necessary)
Add the winning candidate to sel_pop

```
sel_pop.ind[i] = winner
```

Note: Lower rank will be the winner. If ranks are equal, higher crowding distance will be the winner.

mut

Binary Mutation

Import from ip

- ***popsize***
- ***chrom***
- ***pmut_b***
- ***seed***

Input: pop (instance of population class)

Returns: pop *(mutated population)*

With mutation probability, **pmut_b**, the chromosome is mutated at each location.

```
for all individuals in the population:
      for all genes in the an individual:     # length - chrom
            Generate random
            if random <= pmut_b:              # mutate if True
                  pop.ind[i].genes[j] = 1 - pop.ind[i].genes[j]
                  # Changes 0 to 1 and 1 to 0
```

crossover

Binary single point crossover

Import from ip

- ***popsize***
- ***pcross***
- ***chrom***
- ***population***
- ***individual***
- ***seed***

Input:

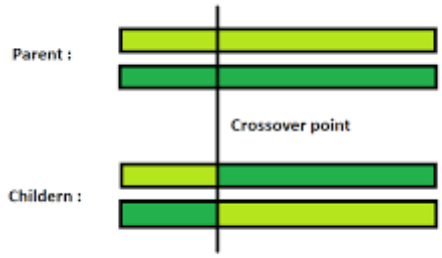
- **par_pop (parent population)**
- **ncross (number of crossovers)**

Returns:

- **child_pop *(child population)***
- **ncross (updated number of crossovers)**

Population and individual are imported (unlike mutation). This is to create a child population.

With crossover probability, **pcross**, 2 individuals are selected and then the mating site is decided. Now the chromosomes are exchanged at the mating site to create 2 new child chromosomes which are stored into individuals in child population defined using population( ) and individual( ).



If there is no crossover, parent chromosomes are simply copied and stored as individuals in the child population.

```
j,k = 0,0
for i in range(popsize/2):      # Since each step uses and creates 2 individuals
      Generate random
      if random <= pcross:            # crossover if True
            ncross = ncross + 1       # Increment of crossover number
            generate mating site
            create 2 child from par_pop.ind[k] and par_pop.ind[k+1]
            (child_pop[j] and child_pop[j+1])
      else:
            copy parents k and k+1 to child population
      j = j + 2           # Since each step uses and creates 2 individuals
      k = k + 2           # Since each step uses and creates 2 individuals
```

uniformxr

Binary uniform crossover

Import from ip

- ***popsize***
- ***pcross***
- ***chrom***
- ***population***

- ***individual***
- ***seed***

Input:

- **par_pop (parent population)**
- **ncross (number of crossovers)**

Returns:

- **child_pop *(child population)***
- **ncross (updated number of crossovers)**

Population and individual are imported (unlike mutation). This is to create a child population.

With crossover probability, **pcross**, a gene location is selected in the chromosome and genes are exchanged only at this location between 2 individuals to create 2 new child chromosomes which is stored into individuals in child population defined using population( ) and individual( ).

If there is no crossover, parent chromosomes are simply copied and stored as individuals in the child population.

```
j,k = 0,0
for i in range(popsize/2):       # Since each step uses and creates 2 individuals
       for n in range(chrom):
              Generate random
              if random <= pcross:             # crossover if True
                     ncross = ncross + 1       # Increment of crossover number
                     exchange genes only at n
                     (par_pop.ind[k].genes[n] and par_pop.ind[k+1].genes[n])
              else:
                     copy parents k and k+1 to child population
                     (child_pop[j] and child_pop[j+1])
       j = j + 2           # Since each step uses and creates 2 individuals
       k = k + 2           # Since each step uses and creates 2 individuals
```

realmut1

Real coded mutation

Import from ip

- ***popsize***
- ***nvar***
- ***lim_r***
- ***pmut_r***
- ***dim***
- ***seed***

- ***integers***

Input:

- **pop (instance of population class)**
- **nmut (number of mutations)**

Returns:

- **pop *(mutated population)***
- **nmut (updated number of mutations)**

With mutation probability, **`pmut_r`**, a real decision variable of the individual is selected and mutated.

Note: 'integers' is imported to return integer values if DVs are integers.

```
for all individuals in the population:
        for all real DVs in the individual:     # length - chrom
                Generate random
                if random <= pmut_r:            # mutate if True
                        mutate pop.ind.xreal
```

realcross2

Real coded SBX crossover

Import from ip

- ***popsize***
- ***pcross***
- ***nvar***
- ***lim_r***
- ***tol***
- ***di***
- ***population***
- ***individual***
- ***seed***
- ***integers***

Input:

- **par_pop (parent population)**
- **ncross (number of crossovers)**

Returns:

- **child_pop *(child population)***
- **ncross (updated number of crossovers)**

Population and individual are imported (unlike mutation). This is to create a child population.

With crossover probability, **`pcross`**, 2 individuals are selected and then the real decision variables are selected with a probability of 0.5 to create 2 new child decision variables which are stored into individuals in child population defined using population( ) and individual( ).

If there is no crossover, parent DVs are simply copied and stored as individuals in the child population.

Note: 'integers' is imported to return integer values if DVs are integers.

keepalive2

This module takes care of elitism and diversity, creating the global mating pool and preserving the best solutions.

Import from ip

- **popsize**
- **nfunc**
- ***population***

Import from ranking

- **ranking**

sort

Input:

- **arr (array for sorting)**

Returns:

- **arr (sorted array)**

The array should have 2 rows. The first row for the indices and the second row for the values. A simple comparison is done and the array is sorted according to the values.

gshare

To assign crowding distances.

Input:

- **globalpop (Global mating pool of both parent and child)**
- **rnk (rank level)**
- **rankarr (array obtained using ranking.py)**

Returns:

- **globalpop (updated global population)**

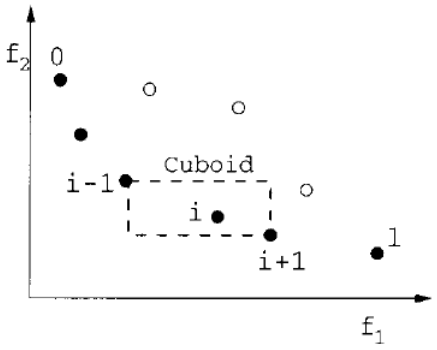


Fig. Crowding distance

Initialize length array which will store indices in the 1st row and crowding distances in the 2nd row.

Get the number of individuals (rankno) in the current rank level(rnk). Now select the individuals present in the current rank level using rankarr and rnk

For each objective, sort the candidates based on the objective function values. Set crowding distance using sorted[i-1] + sorted[i+1]. Assign really high values for the 1st and last candidate and store these values in length. Now update **pop.ind.cub_len** with this length value for each individual.

```
Select individuals to sort in current rank level
for all objectives:
        sort the individuals on fitness values
        length[i] = sorted[i-1] + sorted[i+1]
        length[0], length[last] = 100*maxvalue # Assign high value
        globalpop.ind.cub_len[i] += length[i]  # Assign crowding distance
```

gsort

Input:

- **globalpop (Global mating pool of both parent and child)**
- **rnk (rank level)**
- **rankarr (array obtained using ranking.py)**
- **sel (number of individuals to be selected)**
- **elite (non empty individual list to append the best solutions)**

Returns:

- **elite (updated list of individuals selected)**

Create an array with individual indices in the first row and the crowding distances in the second row. Sort this array using sort. Now select the last 'sel' number of individuals. (highest crowding distances).

Add these individuals to elite which is obtained from keepalive.

keepalive

Input:

- **pop1 (parent population)**
- **pop2 (child population)**

Returns:

- **pop3 (elite and diverse population)**

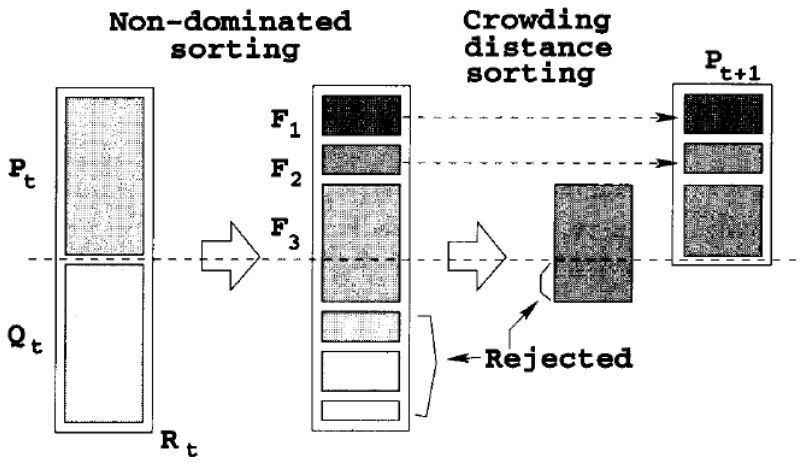


Fig. Non dominated crowding sort

Create globalpop by adding individuals from both pop1 and pop2. Rank globalpop using ranking and return rankarr for other funcrions.

Reset all crowding distances

Assign crowding distances using `gshare`

Now add individuals to pop3 from all rank levels until the size crosses popsize. For the last rank level which violates this, use gsort and get the best individuals and add them to pop3.

```
Create globalpop
Add pop1 and pop2 to globalpop
Ranking(globalpop)
Reset cub_len of all individuals
Assign cub_len
Add individuals to pop3 from each rank level
if pop3 size + rankno > popsize:
       sel = popsize - pop3 size
       Use gsort and select sel number of solutions
       pop3.ind.append(best solutions)
```

nsga2

Main file

Every module is imported here to implement the nsga2 algorithm.

Import

- *init*
- *realinit*
- *decode*
- *func_con*
- *sselect*
- *crossover*
- *uniformxr*

- ***realcross2***
- ***mut***
- ***realmut1***
- ***keepalive***

import from ip,

- ***popsize***
- ***nchrom***
- ***nvar***
- ***gener***
- ***optype***
- ***individual***
- ***population***
- ***nfunc***

Create oldpop and newpop. Populate oldpop with individuals, Initiate these individuals using init, decode or realinit. Evaluate functions using func_con. Now, for each generation, use tournament selection, use mutation and then crossover to get newpop. Evaluate functions for individuals in newpop using func_con. Use keepalive to get the best solutions from oldpop and newpop.

Graph is a list in which the fitness values of all individuals are appended at each generation.

```
create oldpop, newpop
# Initiate
if binary:
	init(oldpop)
	decode(oldpop)
else:
	realinit(oldpop)

# Evaluate functions
func_con(oldpop)

# Create an array of fitness values
term = oldpop.ind[0].fitness.T
for i in range(1,popsize):
	term = np.vstack(term, oldpop.ind[i].fitness.T)


# Add this array to the list graph
graph.append(term)

# Start generation
for i in range(gener):
	# Selection
	matepop = select(oldpop)
	# Crossover
	if binary:
```

```
        newpop = crossover(matepop)
        (or newpop = uniformxr(matepop))
else:
        newpop = realcross(matepop)

# Mutation
if binary:
        newpop = mut(newpop)
else:
        newpop = realmut(newpop)


decode(newpop) # Decode

func_con(newpop) # Evaluate


# Elitism and Diversity
matepop = keepalive(oldpop,newpop)

# All information stored in matepop.



# Update oldpop to complete algorithm
oldpop = newpop


# Create an array of new fitness values
term = oldpop.ind[0].fitness.T
for i in range(1,popsize):
        term = np.vstack(term, oldpop.ind[i].fitness.T)


# Add this new array to the list graph
graph.append(term)
```

Metrics

This module relies on the package pymoo to get metrics like hypervolume, GD, IGD, etc.

Refer to the commented code for more information.

Visit pymoo.org for more information.

Animate

This module animates the progression of the pareto front. The fitness values of all individuals are plotted at each generation and an animation is made.

This was stored in the list 'graph' using 'term' in nsga2.py

This part will automatically identify if the plots are 2D or 3D using nfunc.

Change the value of `frames` in `animation.FuncAnimation` to vary the number of generations plotted.

Change the value of `fps` in `ani.save` to change frame rate.

Refer to the commented code for more information.

Rotate

This part creates a rotating animation of the final pareto front generated.

The final 'term' from nsga2.py is used to plot the final pareto front. Change the value of 30 in `ax.view_init(30,i)` to experiment with angles and rotation. You may also put `i` in x position flor experiments.

Change the value of `frames` in `animation.FuncAnimation` to vary the angles. A value of 200 rotates the pareto front from 0 to 200 degrees.

Note: This part is only for 3D plots and 3 objective functions.

plot

This module has 2 functions to plot the fitness functions.

Plot2D

Input:

- **data (Array of fitness functions)**
- **name (function name)**
- **k (generation number)**

Returns:

- **plot**

Returns 2D plots in case of bi-objective problems. The input array data is 'term' from nsga2.py

Note: The columns of data array should be fitness values.

Plot3D

Input:

- **data (Array of fitness functions)**
- **name (function name)**
- **k (generation number)**

Returns:

- **plot**

Returns 3D plots in case of 3-objective problems. The input array data is 'term' from nsga2.py

Note: The columns of data array should be fitness values.