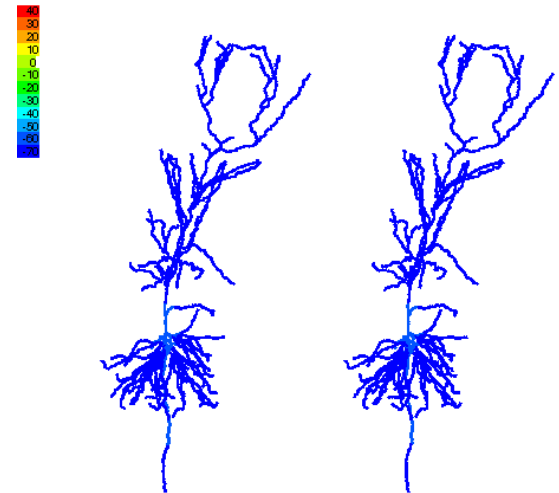
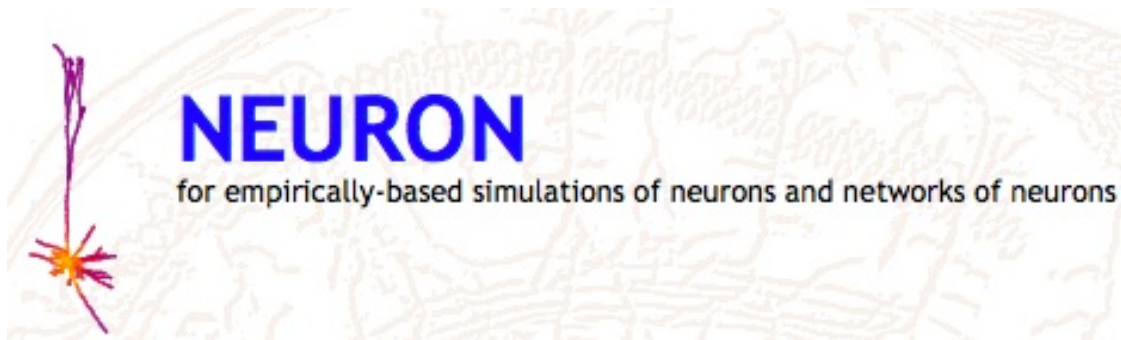


Computational Neuroscience

Lab 11:

Evolutionary optimization of parameters



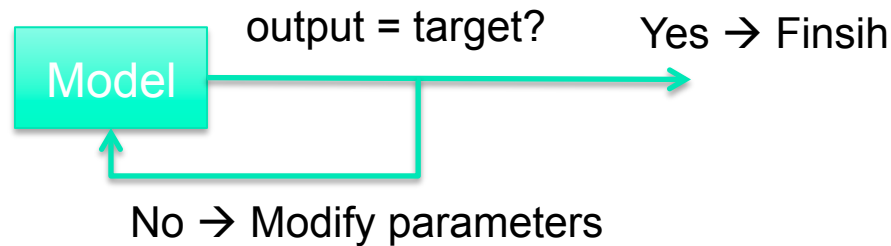
Instructor: Salvador Dura-Bernal



Overview

- ❑ Why do we need it? Examples:
 - Single cell params
 - Network firing rates
 - Behavioral performance
- ❑ Approaches
 - Hand tuning
 - Brute force
 - Search Algorithms
- ❑ Evolutionary algorithms
- ❑ Mutation rates and elites
- ❑ Optimizing the firing rate of a network

Parameter optimization/tuning/fitting



- ❑ Finding the optimum **parameter** values to achieve a **target**

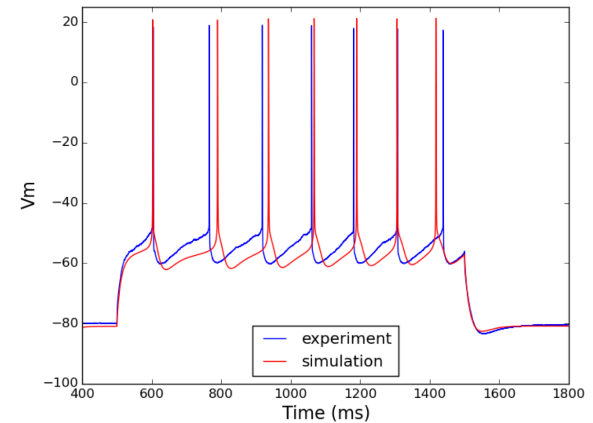
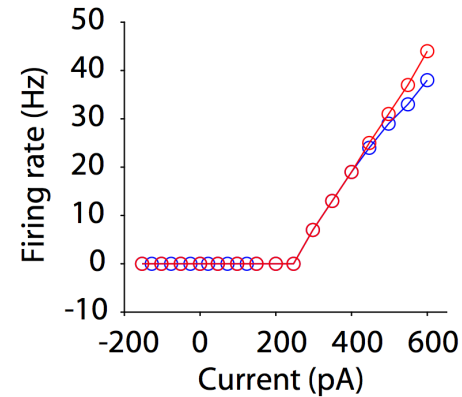
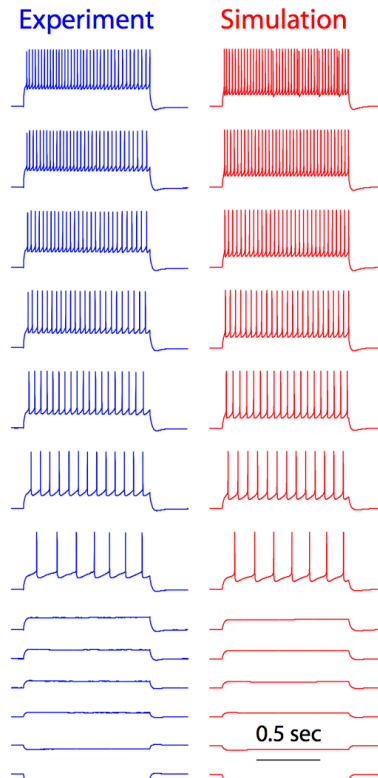
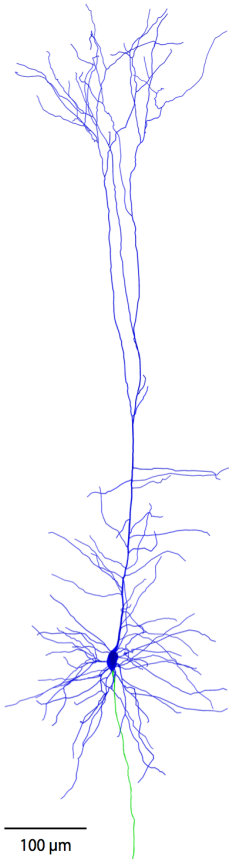
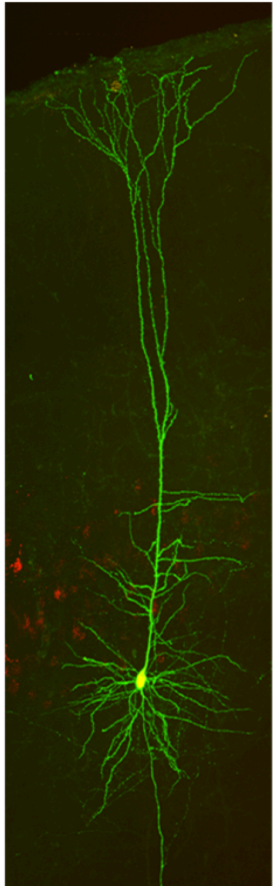
- ❑ Target typically based on experimental data, e.g.:
 - Single cell intrinsic properties (f-I curve, ...)
 - Network properties (firing rates, oscillations, ...)
 - Behavioral task (hand distance to target, ...)

- ❑ Parameters of model adjusted within realistic range, e.g.:
 - Density of ionic channels
 - Connection weights, background inputs
 - Reinforcement learning interval

Parameter optimization: cell

Parameters: Capacitance, leak, Ra, and density of channels HCN, Kd, Na, Kdr, Ka, Ca, KCa

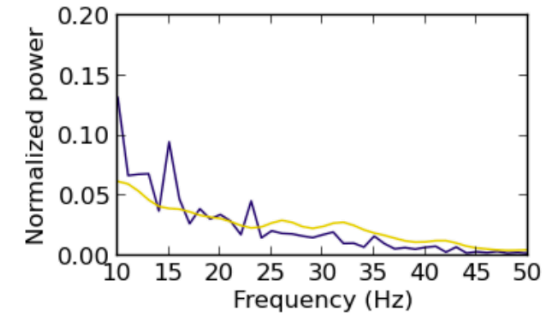
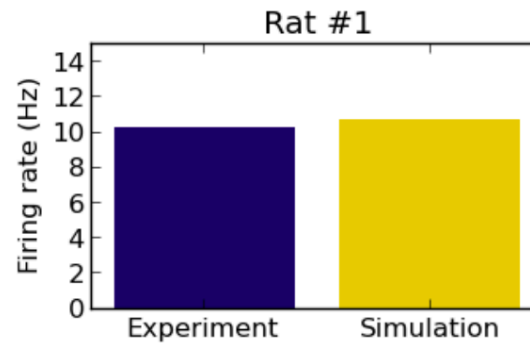
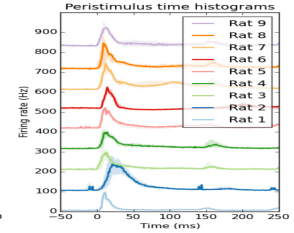
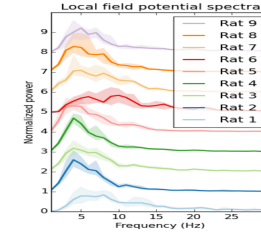
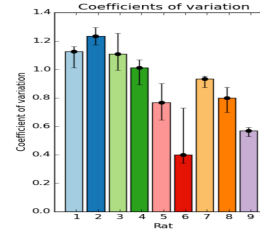
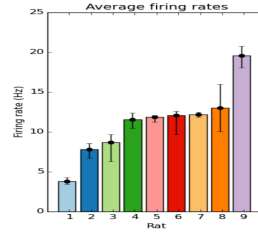
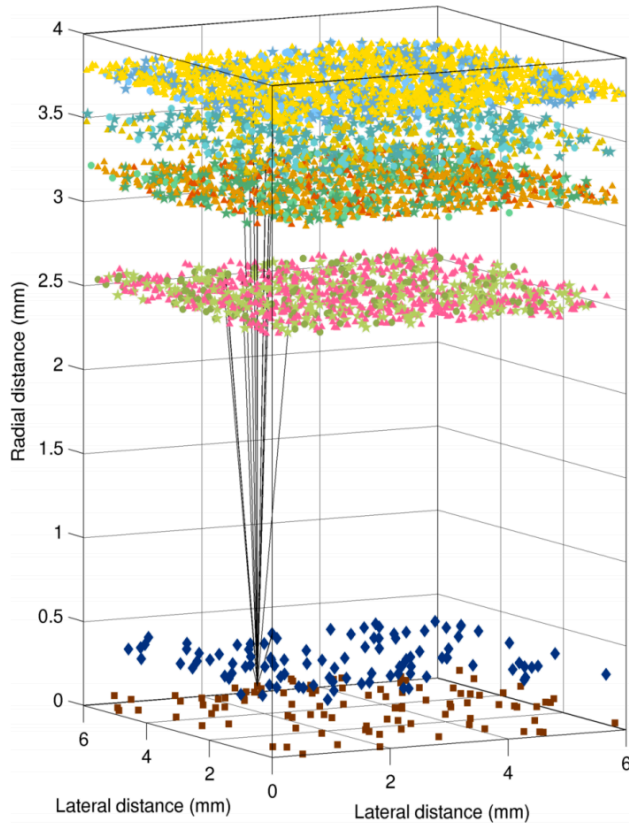
Target: Firing rates (I-f curve), spike shape, subthreshold voltage shape



Parameter optimization: network

Parameters: Connection weights and background input rate

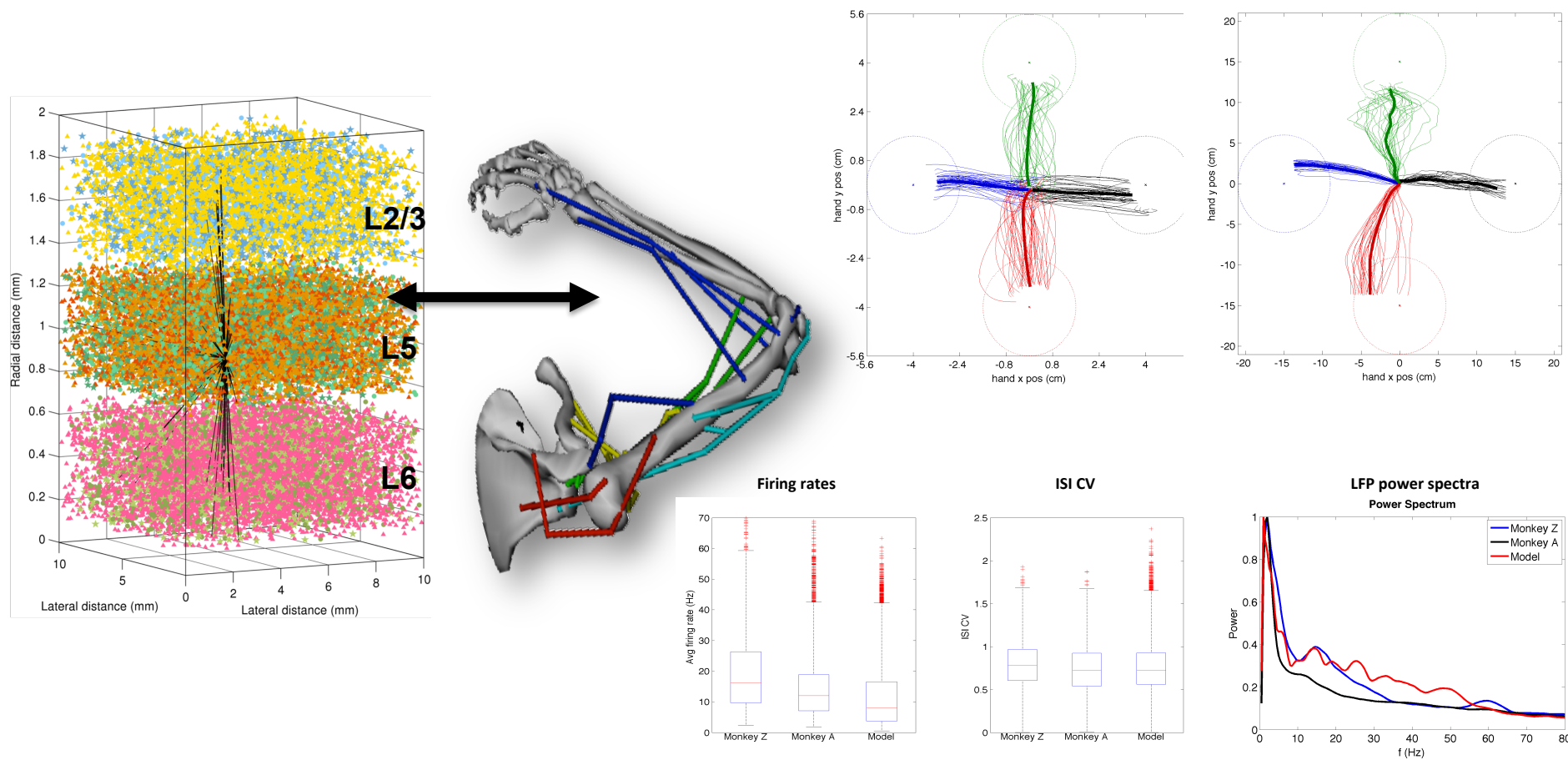
Target: Avg firing rates, coefficient of variation, LFP spectra, PSTH



Parameter optimization: behavior

Parameters: Training metaparameters (RL interval, exploratory movements, ...)

Target: Final distance to target, distance from ideal trajectory

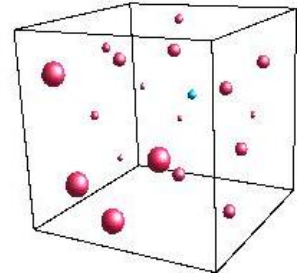


Parameter optimization

□ 3 main approaches

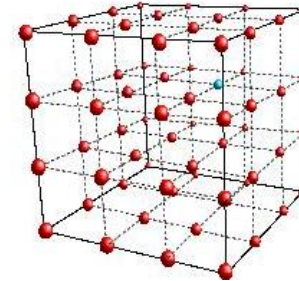
■ Hand tuning:

- Example: Purkinje cell (1994), 1 year of work
- Intelligent and focused but limited search of parameter space



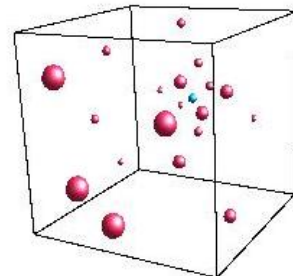
■ Brute force

- Incomplete or complete but sparse sampling
- Computation, storage and analysis challenges



■ Search algorithms

- Find “best model” based on some *fitness* measure
- Many methods: gradient descent, simulated annealing, genetic/evolutionary algorithms, ...



Evolutionary Algorithms

I talked about the survival
of the fittest



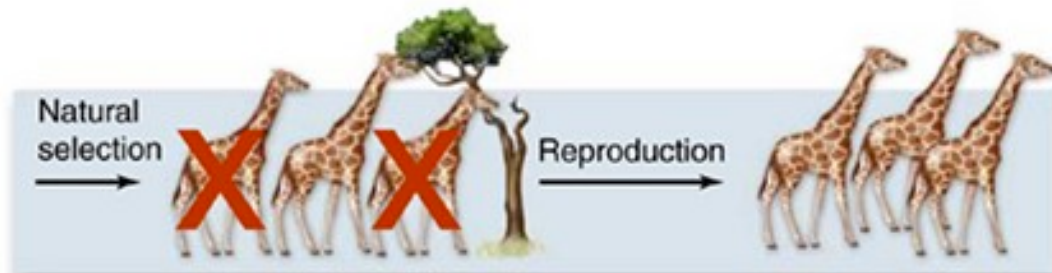
Before the Hunger Games



Some individuals born happen to have longer necks due to genetic differences.



Individuals pass on their traits to next generation.

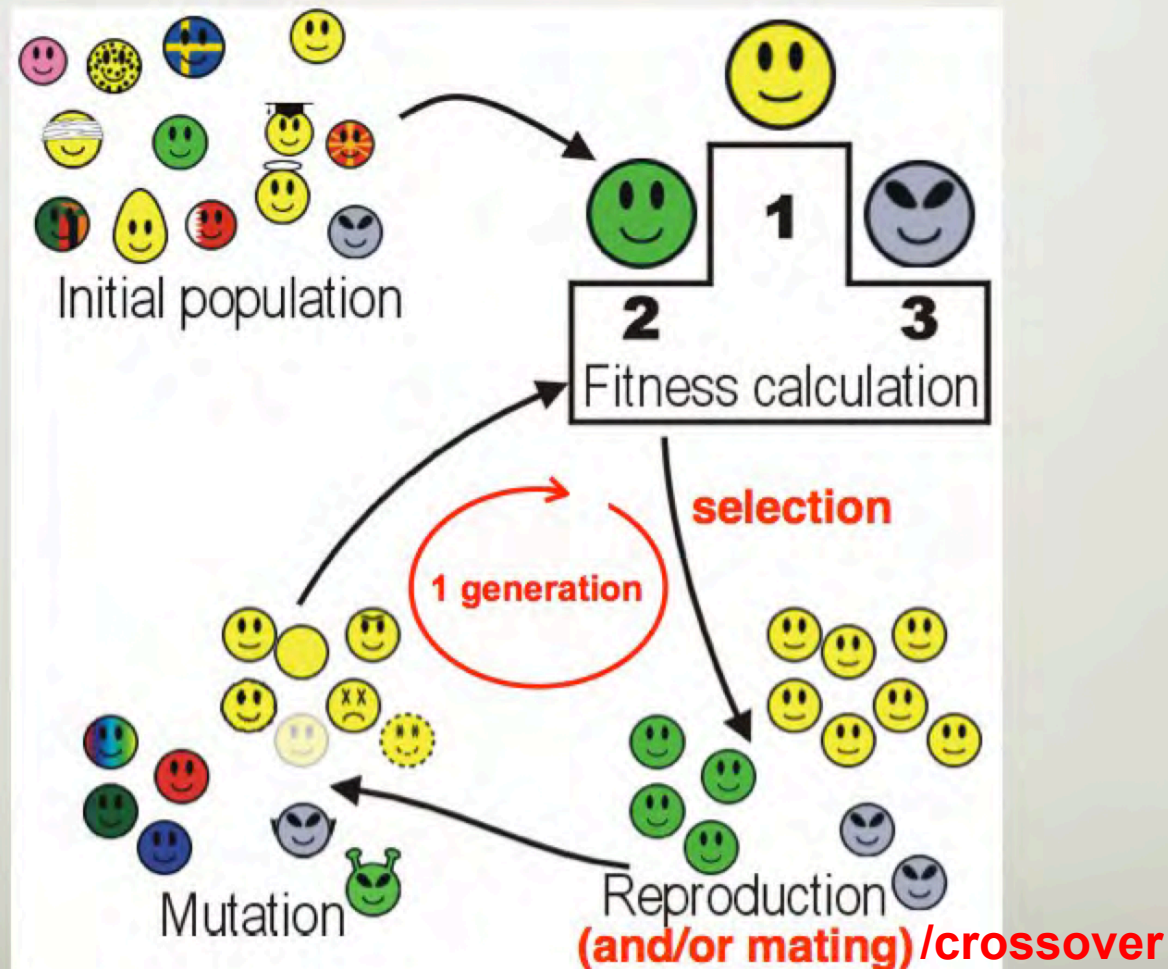


Over many generations, longer-necked individuals are more successful, perhaps because they can feed on taller trees, and pass the long-neck trait on to their offspring.

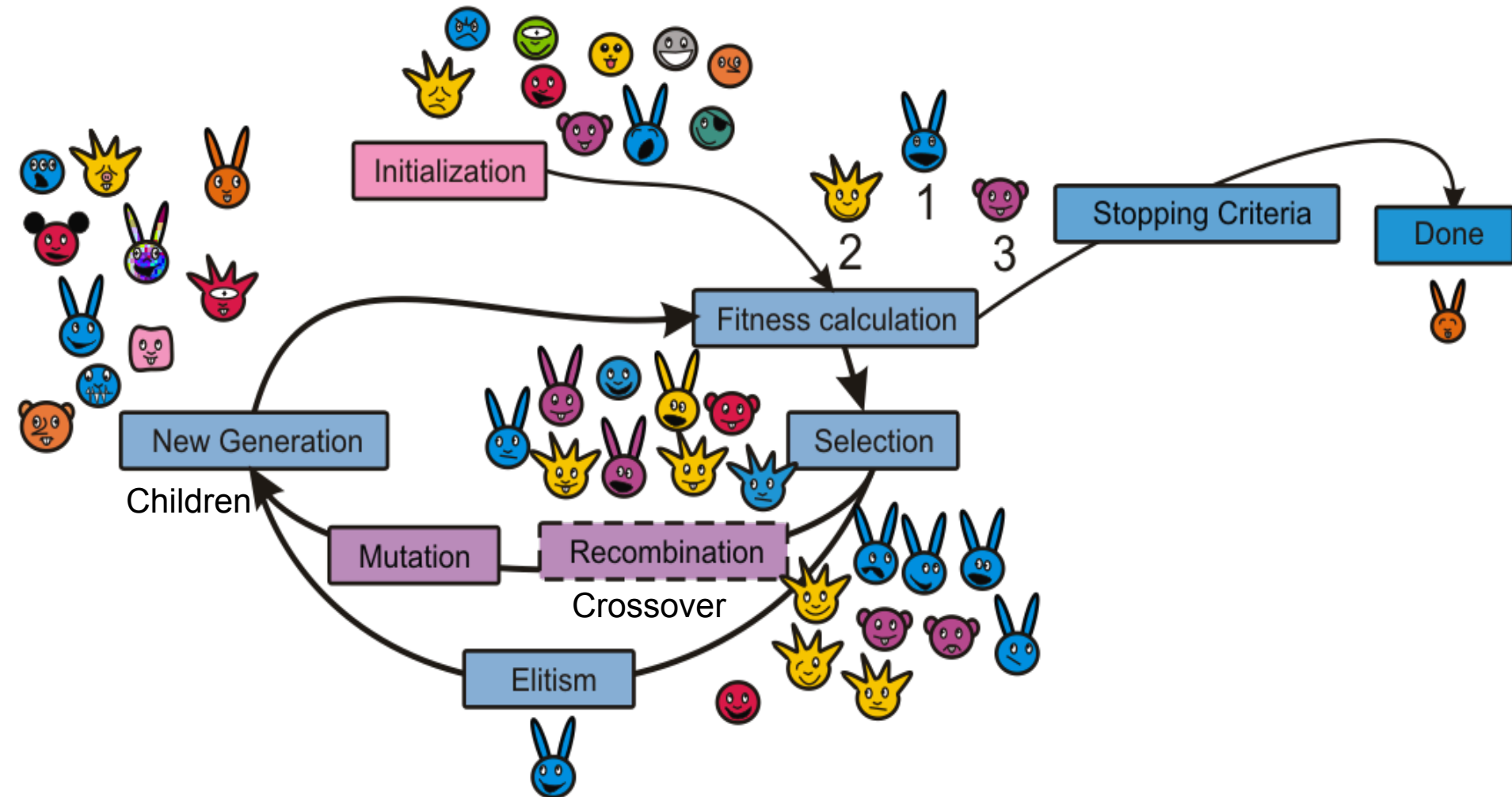
b. Darwin's theory: natural selection or genetically-based variation leads to evolutionary change.

Evolutionary/genetic algorithms

Mimics biology: successive generations of a population become fitter through selection, combined with mating and mutation.

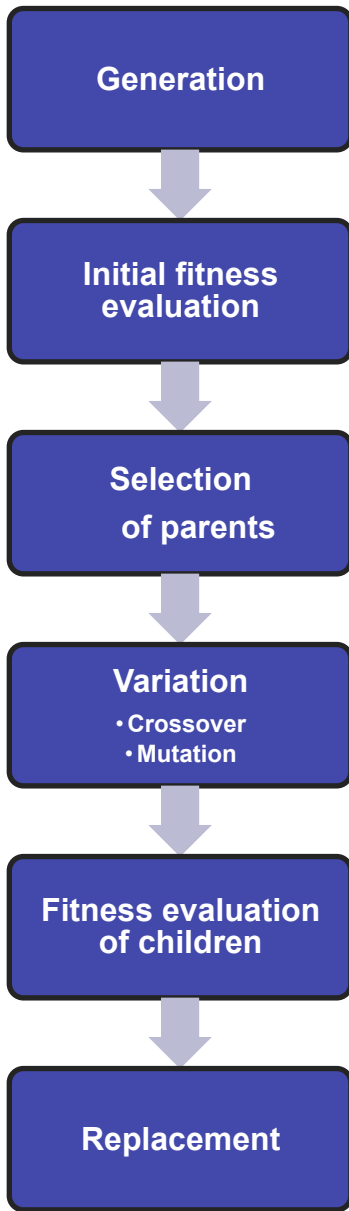


Evolutionary algorithms

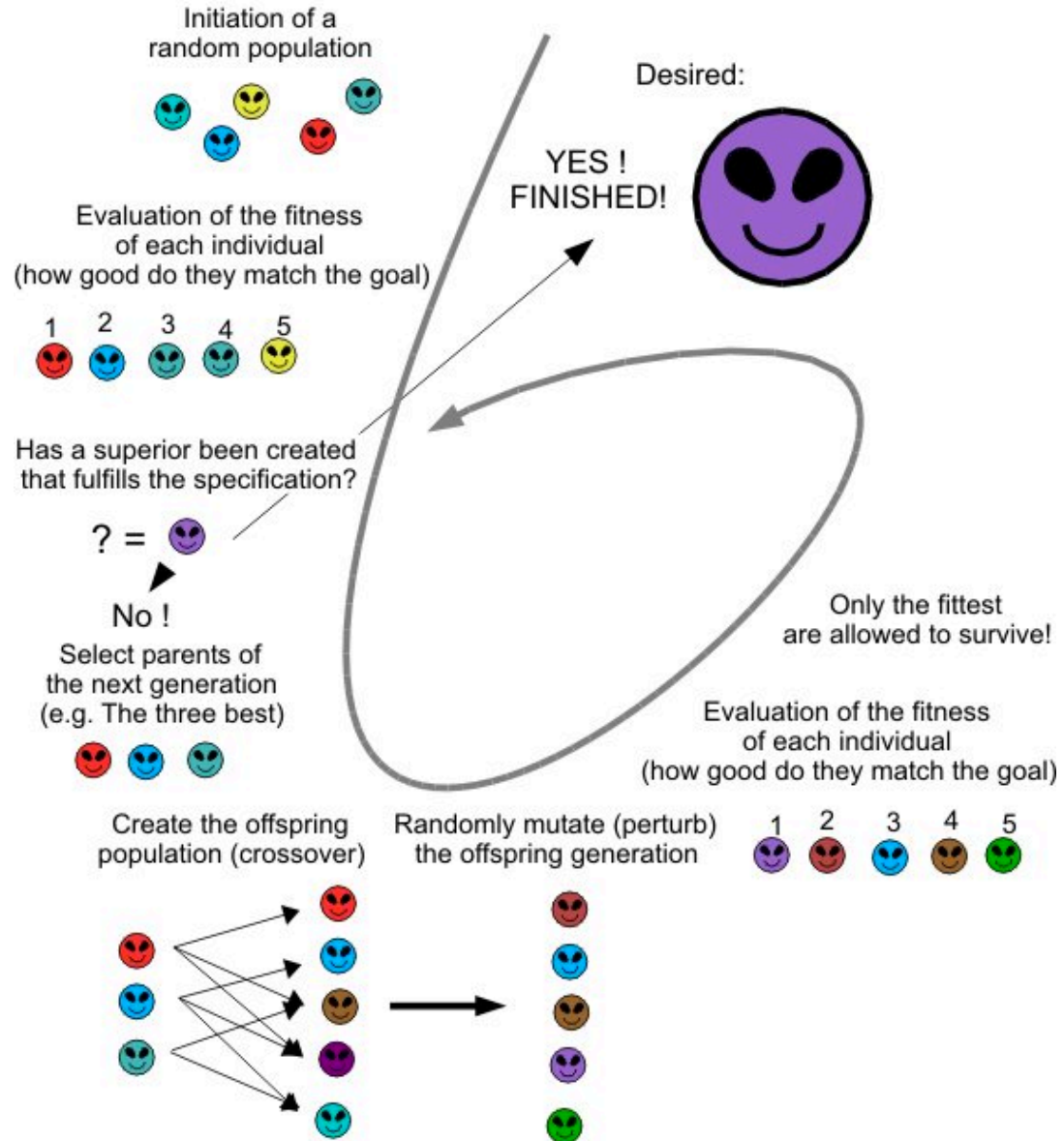


Evolutionary algorithms

New generation



How does an Evolutionary Algorithm (EA) work ?





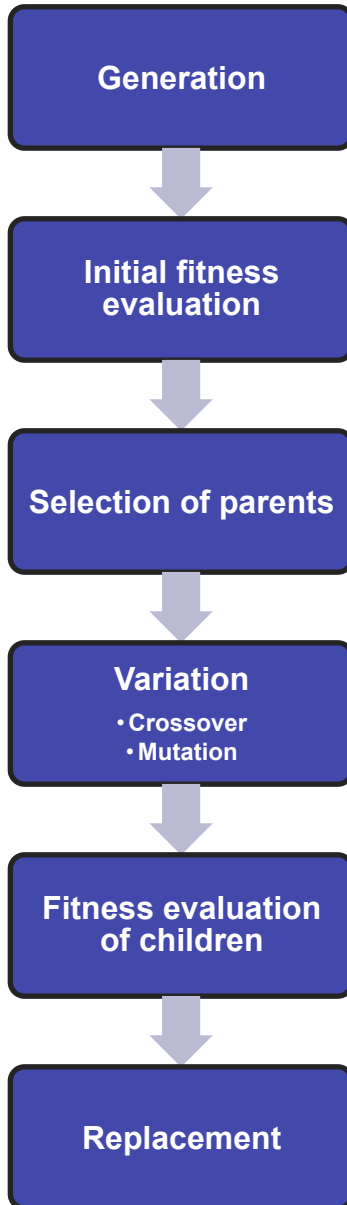
Inspyred

inspyred: Bio-inspired Algorithms in Python

- Python library for bio-inspired algorithms
- Includes evolutionary algorithms and many others
- Each stage of the algorithm is customizable
- <http://pythonhosted.org/inspyred/index.html>
- Installation: *pip install inspyred*

- Overview
 - Bio-inspired Computation
 - Design Methodology
 - Installation
 - Getting Help
- Tutorial
 - The Rastrigin Function
 - The Generator
 - The Evaluator
 - The Evolutionary Computation
 - Evolving Polygons
 - The Generator
 - The Evaluator
 - The Bounder
 - The Observer
 - The Evolutionary Computation
 - Lunar Explorer
 - The Generator
 - The Evaluator
 - The Evolutionary Computation
- Examples
 - Standard Algorithms
 - Genetic Algorithm
 - Evolution Strategy
 - Simulated Annealing
 - Differential Evolution Algorithm
 - Estimation of Distribution Algorithm
 - Pareto Archived Evolution Strategy (PAES)
 - Nondominated Sorting Genetic Algorithm (NSGA-II)
 - Particle Swarm Optimization
 - Ant Colony Optimization
 - Customized Algorithms
 - Custom Evolutionary Computation
 - Custom Archiver
 - Custom Observer
 - Custom Replacer
 - Custom Selector
 - Custom Terminator
 - Custom Variator

Inspired



Problem-specific components

- A **generator** that defines how solutions are created
- An **evaluator** that defines how fitness values are calculated for solutions

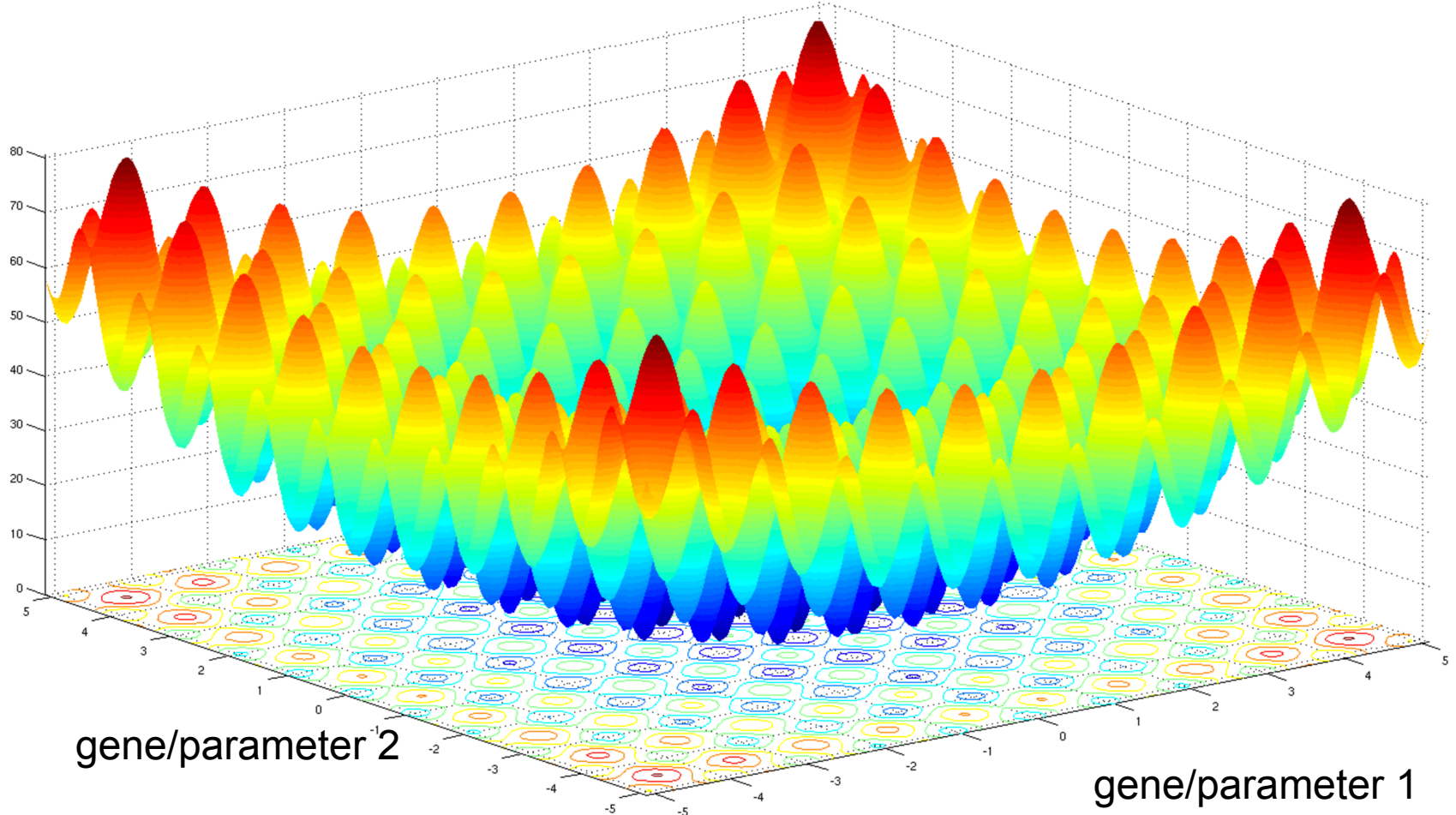
Algorithm-specific evolutionary operators

- A **terminator** that determines whether the evolution should end
- A **selector** that determines which individuals should become parents
- A **variator** that determines how offspring are created from existing individuals
- A **replacer** that determines which individuals should survive into the next generation

Inspyred example

Rastrigin example

Fitness function (minimize)

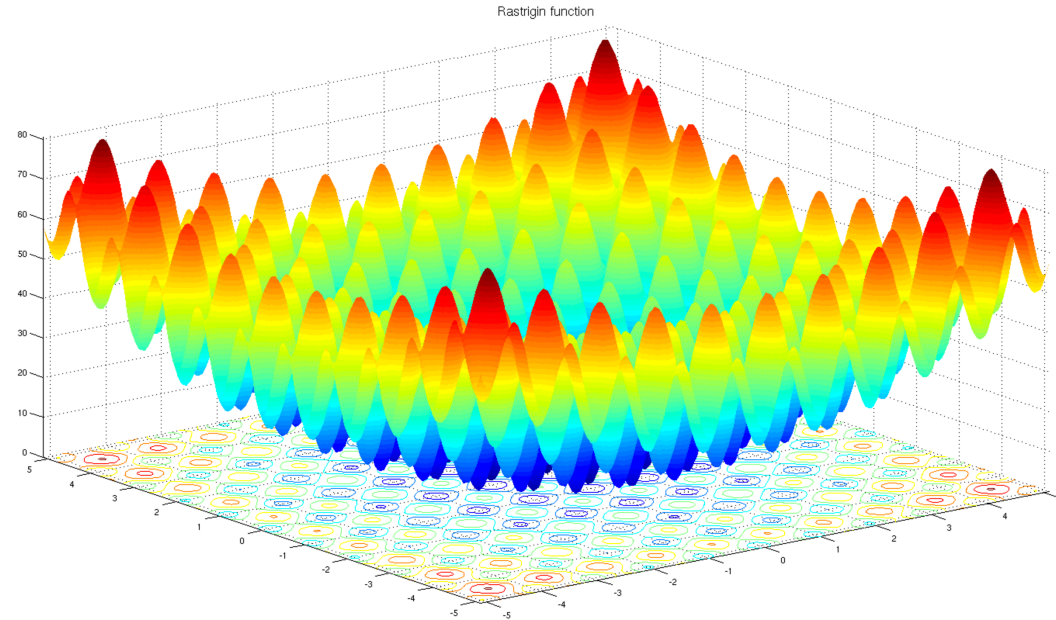


Inspired example

Rastrigin example

The Rastrigin function is a non-convex function used as a performance test problem for optimization algorithms.

Finding the minimum of this function is a fairly difficult problem due to its large search space and its large number of local minima.



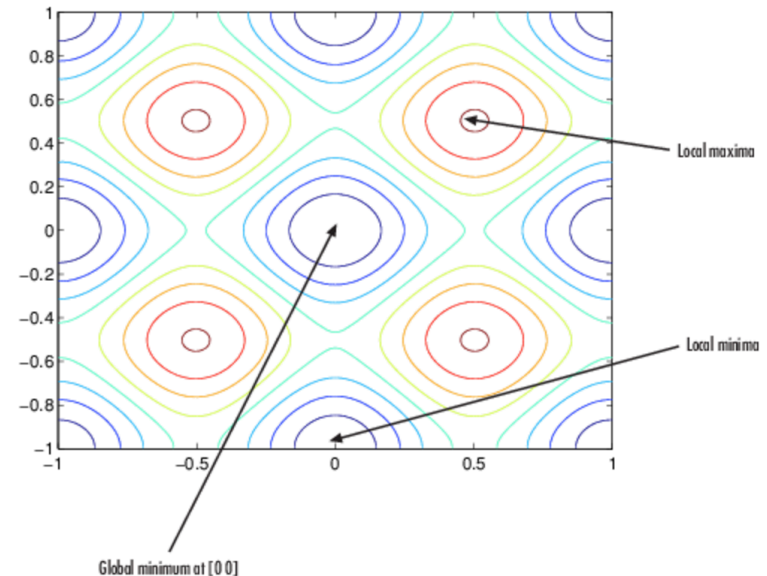
Fitness function:

Minimize

$$10n + \sum_{i=1}^n ((x_i - 1)^2 - 10 \cos(2\pi(x_i - 1)))$$

for $x_i \in [-5.12, 5.12]$.

Parameters: x_1, x_2





Optimization Example

```
rand = Random()  
rand.seed(int(time()))  
  
my_ec = ec.EvolutionaryComputation(rand)
```

- Create evolutionary computation object that encapsulates the components of a generic evolutionary computation.
- These components are the *selection* mechanism, *the variation* operators, the *replacement* mechanism, the *terminators*, and the *s*.
- Requires random value since needs to generate random crossover and mutations

Optimization Example

```
my_ec.selector = ec.selectors.tournament_selection
```

- Select which parents will reproduce

`inspyred.ec.selectors.uniform_selection(random, population, args)`

Return a uniform sampling of individuals from the population.

This function performs uniform selection by randomly choosing members of the population with replacement.

Optional keyword arguments in args:

- `num_selected` – the number of individuals to be selected (default 1)

`inspyred.ec.selectors.tournament_selection(random, population, args)` ¶

Return a tournament sampling of individuals from the population.

This function selects `num_selected` individuals from the population. It selects each one by using random sampling without replacement to pull `tournament_size` individuals and adds the best of the tournament as its selection. If `tournament_size` is greater than the population size, the population size is used instead as the size of the tournament.

Optional keyword arguments in args:

- `num_selected` – the number of individuals to be selected (default 1)
- `tournament_size` – the tournament size (default 2)

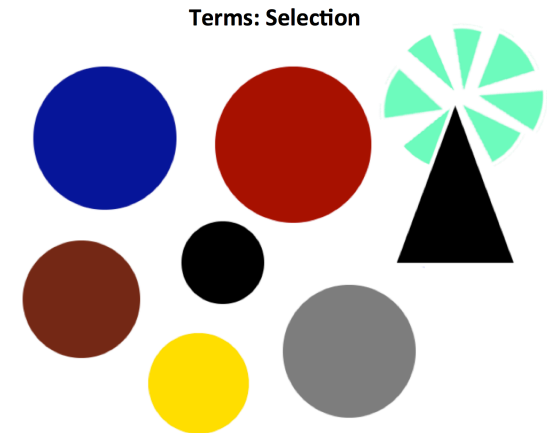
`inspyred.ec.selectors.truncation_selection(random, population, args)`

Selects the best individuals from the population.

This function performs truncation selection, which means that only the best individuals from the current population are selected. This is a completely deterministic selection mechanism.

Optional keyword arguments in args:

- `num_selected` – the number of individuals to be selected (default `len(population)`)

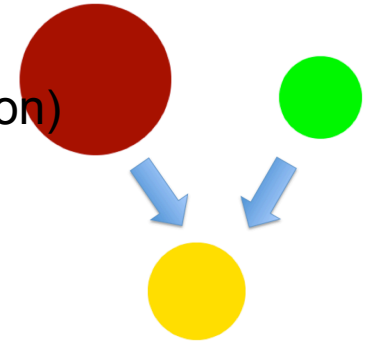


Optimization Example

```
my_ec.variator = [ec.variators.uniform_crossover,  
ec.variators.gaussian_mutation]
```

- Select variations to apply during reproduction (crossover+mutation)

Terms: Crossover/Recombination



`inspyred.ec.variators.uniform_crossover(random, candidates, args)` ¶

Return the offspring of uniform crossover on the candidates.

This function performs uniform crossover (UX). For each element of the parents, a biased coin is flipped to determine whether the first offspring gets the 'mom' or the 'dad' element. An optional keyword argument in `args`, `ux_bias`, determines the bias.

Optional keyword arguments in `args`:

- `crossover_rate` – the rate at which crossover is performed (default 1.0)
- `ux_bias` – the bias toward the first candidate in the crossover (default 0.5)

`inspyred.ec.variators.n_point_crossover(random, candidates, args)`

Return the offspring of n-point crossover on the candidates.

This function performs n-point crossover (NPX). It selects n random points without replacement at which to 'cut' the candidate solutions and recombine them.

Optional keyword arguments in `args`:

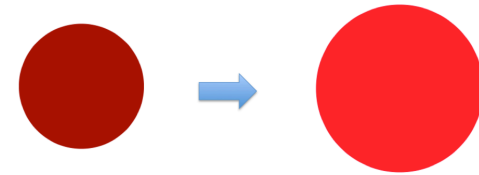
- `crossover_rate` – the rate at which crossover is performed (default 1.0)
- `num_crossover_points` – the number of crossover points used (default 1)

Optimization Example

```
my_ec.variator = [ec.variators.uniform_crossover,  
ec.variators.gaussian_mutation]
```

Terms: Mutation

- Select variations to apply during reproduction (crossover+mutation)



`inspyred.ec.variators.gaussian_mutation(random, candidates, args)`

Return the mutants created by Gaussian mutation on the candidates.

This function performs Gaussian mutation. This function makes use of the boulder function as specified in the EC's `evolve` method.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)
- *gaussian_mean* – the mean used in the Gaussian function (default 0)
- *gaussian_stdev* – the standard deviation used in the Gaussian function (default 1)

The mutation rate is applied on an element by element basis.

`inspyred.ec.variators.scramble_mutation(random, candidates, args)`

Return the mutants created by scramble mutation on the candidates.

This function performs scramble mutation. It randomly chooses two locations along the candidate and scrambles the values within that slice.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)

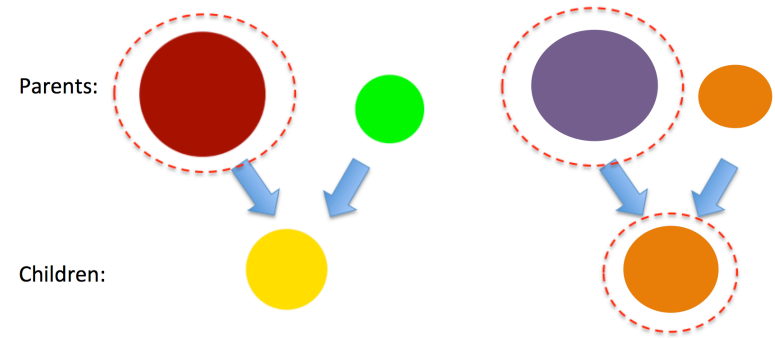
The mutation rate is applied to the candidate as a whole (i.e., it either mutates or it does not, based on the rate).

Optimization Example

```
my_ec.replacer = ec.replacers.steady_state_replacement
```

- Select what set of parents and children to keep for next generation

Terms: Replacement



```
inspyred.ec.replacers.steady_state_replacement(random, population, parents, offspring, args)
```

Performs steady-state replacement for the offspring.

This function performs steady-state replacement, which means that the offspring replace the least fit individuals in the existing population, even if those offspring are less fit than the individuals that they replace.

```
inspyred.ec.replacers.generational_replacement(random, population, parents, offspring, args)
```

Performs generational replacement with optional weak elitism.

This function performs generational replacement, which means that the entire existing population is replaced by the offspring, truncating to the population size if the number of offspring is larger. Weak elitism may also be specified through the *num_elites* keyword argument in *args*. If this is used, the best *num_elites* individuals in the current population are allowed to survive if they are better than the worst *num_elites* offspring.

Optional keyword arguments in *args*:

- *num_elites* – number of elites to consider (default 0)



Optimization Example

```
my_ec.terminator = ec.terminators.evaluation_termination
```

- Select criteria to stop evolutionary optimization

`inspyred.ec.terminators.evaluation_termination(population, num_generations, num_evaluations, args)`

Return True if the number of function evaluations meets or exceeds a maximum.

This function compares the number of function evaluations that have been generated with a specified maximum. It returns True if the maximum is met or exceeded.

Optional keyword arguments in args:

- `max_evaluations` – the maximum candidate solution evaluations (default `len(population)`)

`inspyred.ec.terminators.generation_termination(population, num_generations, num_evaluations, args)`

Return True if the number of generations meets or exceeds a maximum.

This function compares the number of generations with a specified maximum. It returns True if the maximum is met or exceeded.

Optional keyword arguments in args:

- `max_generations` – the maximum generations (default 1)

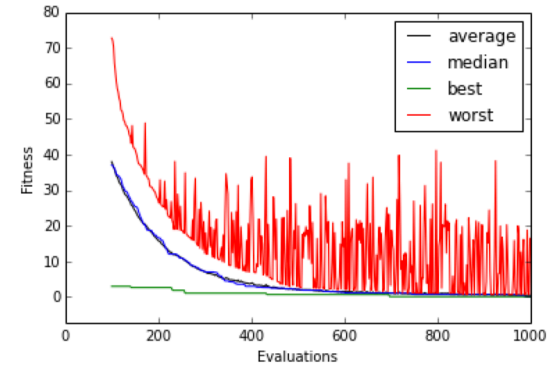
Optimization Example

```
my_ec.observer = [ec.observers.stats_observer,  
ec.observers.plot_observer,  
ec.observers.best_observer]
```

- Select what information to observe/display during the evolutionary computations

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
450	1000	16.4311490	0.09074930	0.10785487	0.47232884	1.62516803

Best Individual: [1.0207958718596968, 0.9949433488753152] : 0.0907493057544



`inspyred.ec.observers.stats_observer(population, num_generations, num_evaluations, args)`
Print the statistics of the evolutionary computation to the screen.

This function displays the statistics of the evolutionary computation to the screen. The output includes the generation number, the current number of evaluations, the maximum fitness, the minimum fitness, the average fitness, and the standard deviation.

`inspyred.ec.observers.best_observer(population, num_generations, num_evaluations, args)`
Print the best individual in the population to the screen.

This function displays the best individual in the population to the screen.

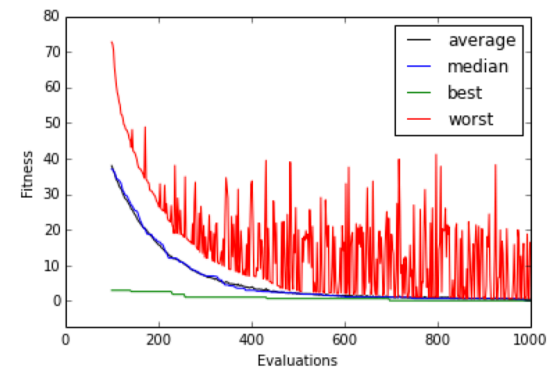
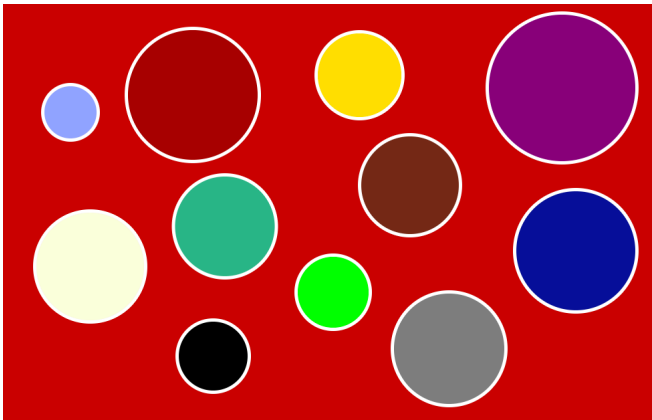
`inspyred.ec.observers.plot_observer(population, num_generations, num_evaluations, args)`
Plot the output of the evolutionary computation as a graph.

This function plots the performance of the EC as a line graph using the pylab library (matplotlib) and numpy. The graph consists of a blue line representing the best fitness, a green line representing the average fitness, and a red line representing the median fitness. It modifies the keyword arguments variable 'args' by including an entry called 'plot_data'.

Optimization Example

```
final_pop = my_ec.evolve(generator=generate_rastrigin,  
                        evaluator=evaluate_rastrigin,  
                        pop_size=100,  
                        maximize=False,  
                        bounder=ec.Bounder(-5.12, 5.12),  
                        max_evaluations=1000,  
                        num_selected=2,  
                        mutation_rate=0.25,  
                        num_inputs=2)
```

- Select properties of evolutionary algorithm





Understanding population size

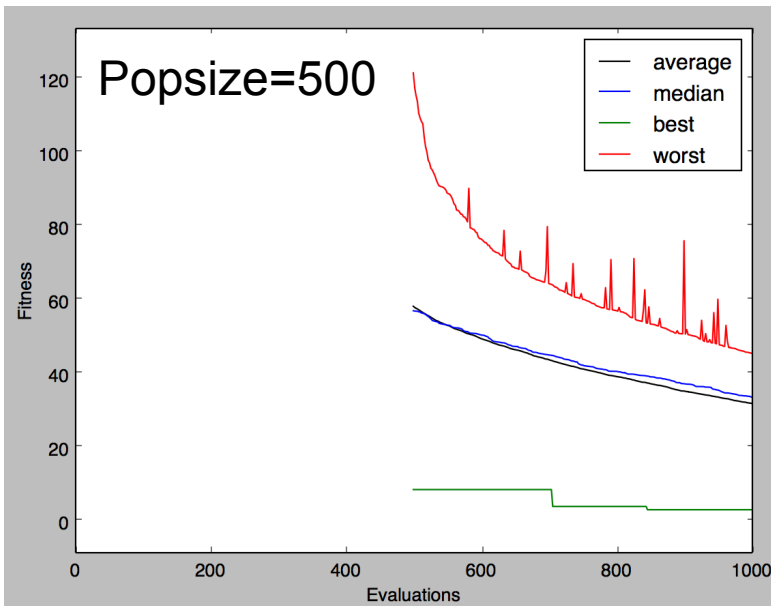
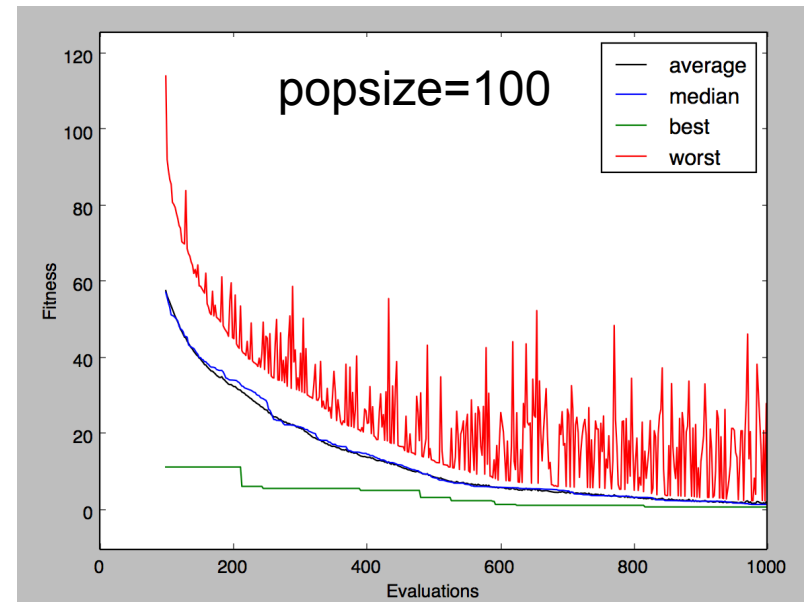
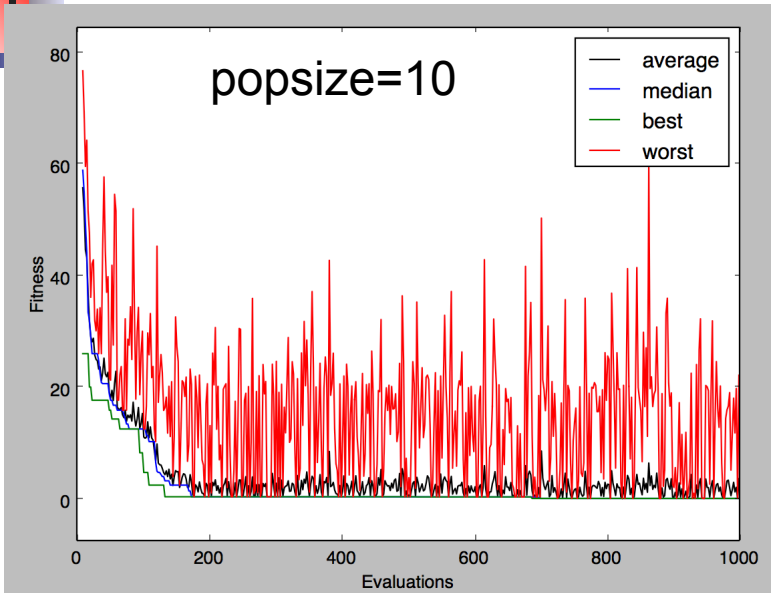
Start from lab10_optim1.py (Rastrigin example) make the following changes:

- 1) Change the population size (`pop_size`) from 100 to a) 10 individual and b) 500 individuals -- what effect does it have?

Note 1: Population size is constant through generations, but only new individuals need to be evaluated; ie. all individuals are evaluated once at the beginning, but after that only offspring.

Note 2: The number of new individuals (offspring) in each generation is determined by the parameter `num_selected` (the number of offspring will be equal to the number of individuals selected for reproduction, ie. 1 child / parent -- not completely intuitive but that's how it works)

pop_size



- ❑ Why is 'worst' higher at start in 100 and 500?
- ❑ Why is 'best' lower at start in 500?
- ❑ Why is 10 faster to converge?



Understanding selection

2) Using `pop_size=100`, change the number of individuals selected for reproduction (`num_selected`) from 2 to

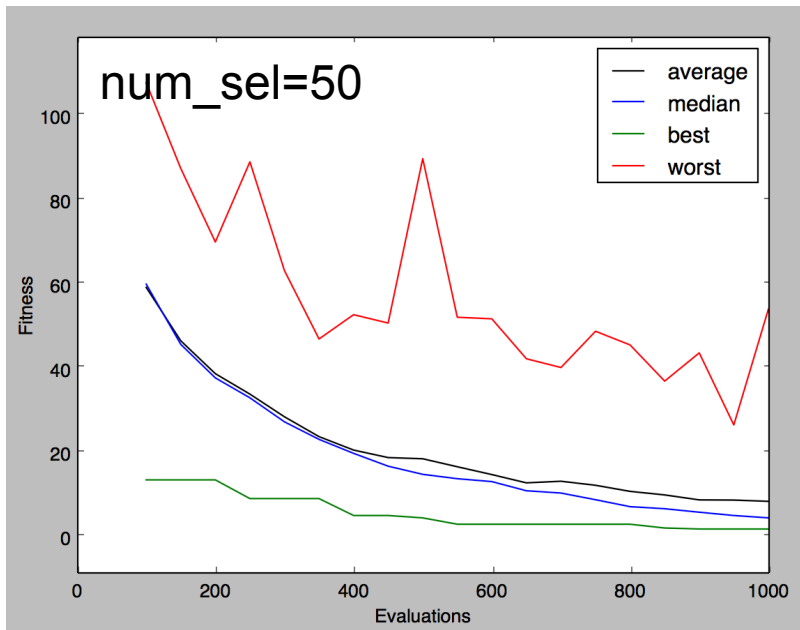
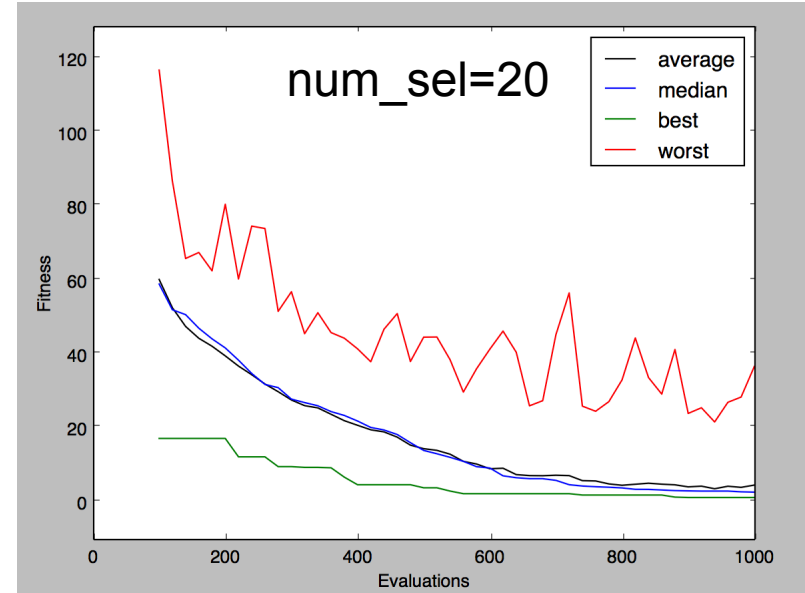
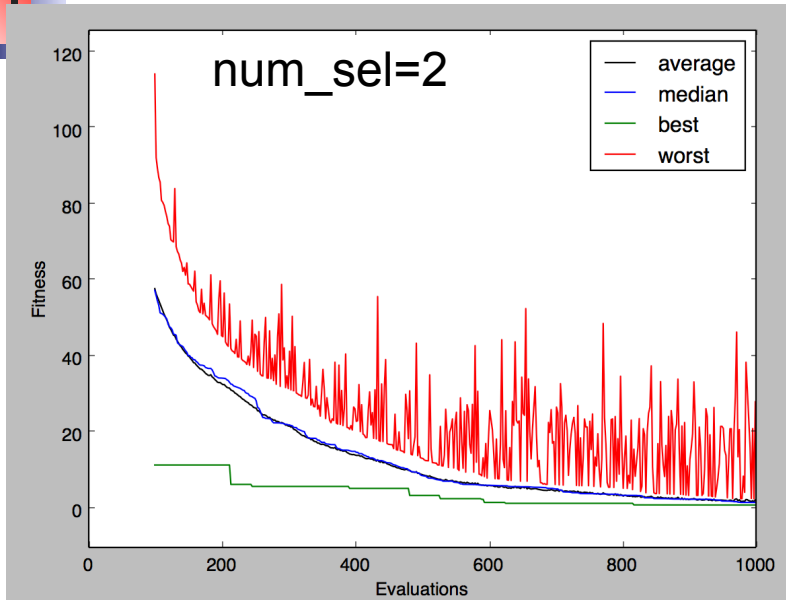
a) 20 and

b) 100

What is the effect?

How would the graph look if the x-axis showed num generations instead of evaluations?

num_selected



- Why do the 'worst' lines look so different?
- How would the graph look if x-axis was *Generations* ?
(hint: write down the total number of generations in each case)



Understanding mutation

3) Using `pop_size=100` and `num_selected=2`, remove the `gaussian_mutation` variator.

What is the effect in convergence?

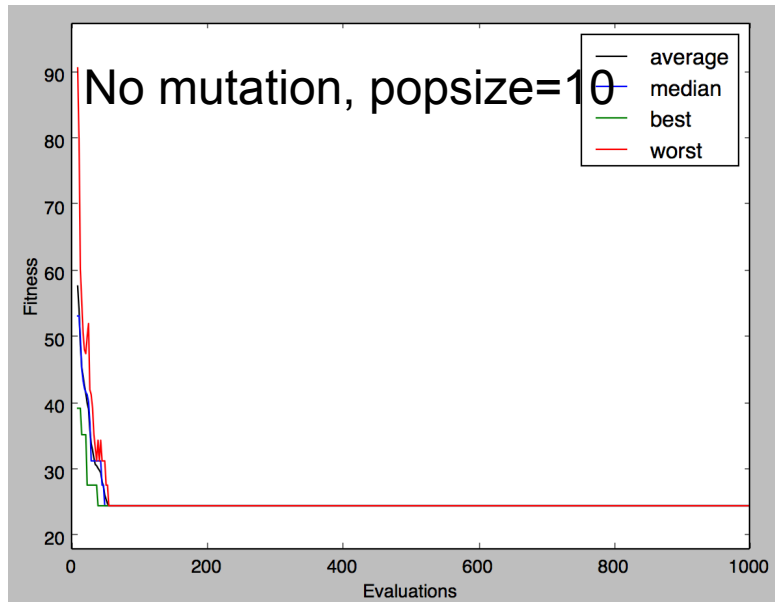
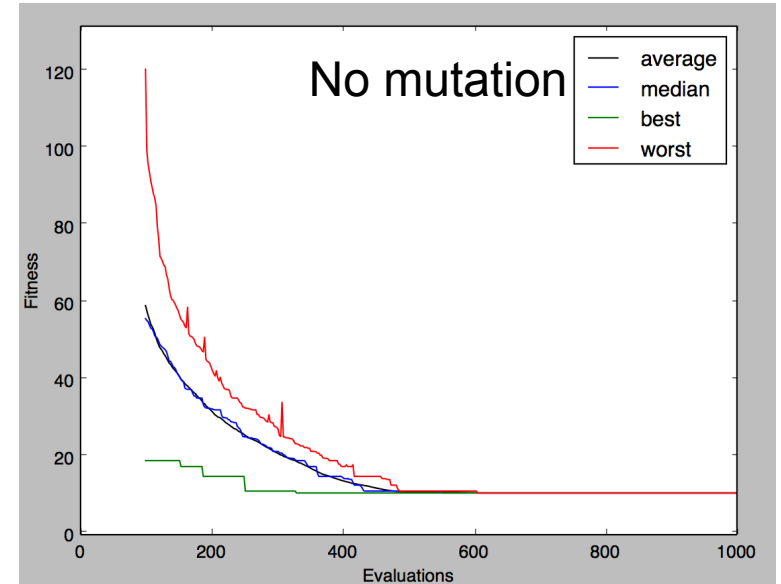
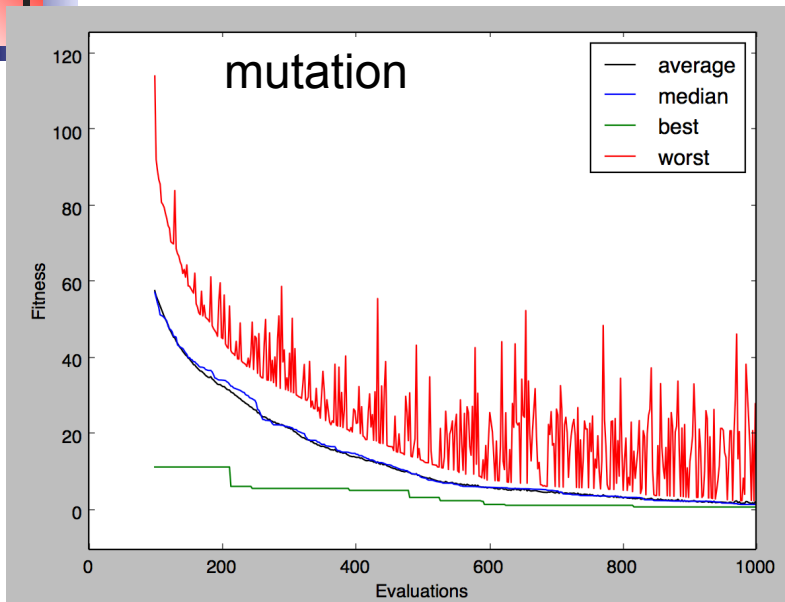
How does the final fitness solution compare to previous ones?

a) What happens if you now reduce `pop_size=10`?

b) Using `pop_size=10`, `num_selected=10`, put back the `gaussian_mutation` variator; and test the following 3 `mutation_rate` values: 0, 0.1 and 2 -- what's going on?

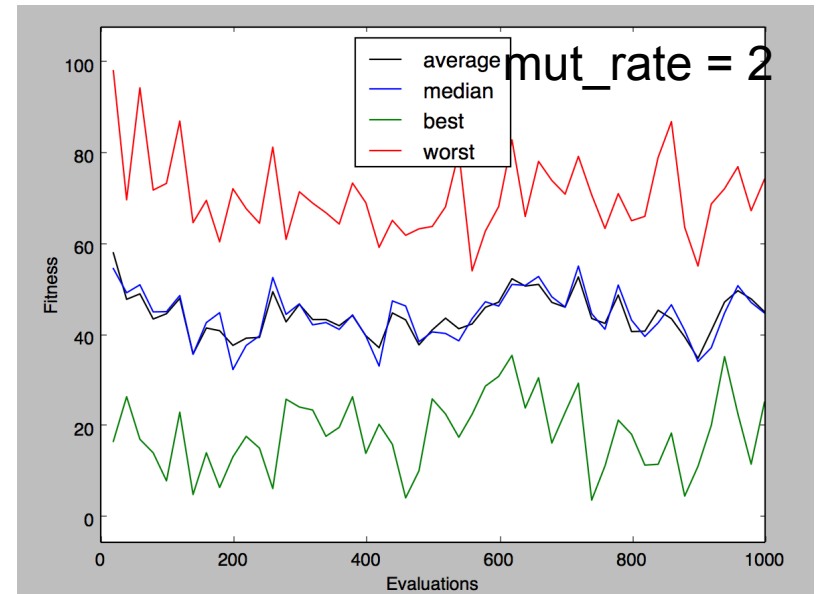
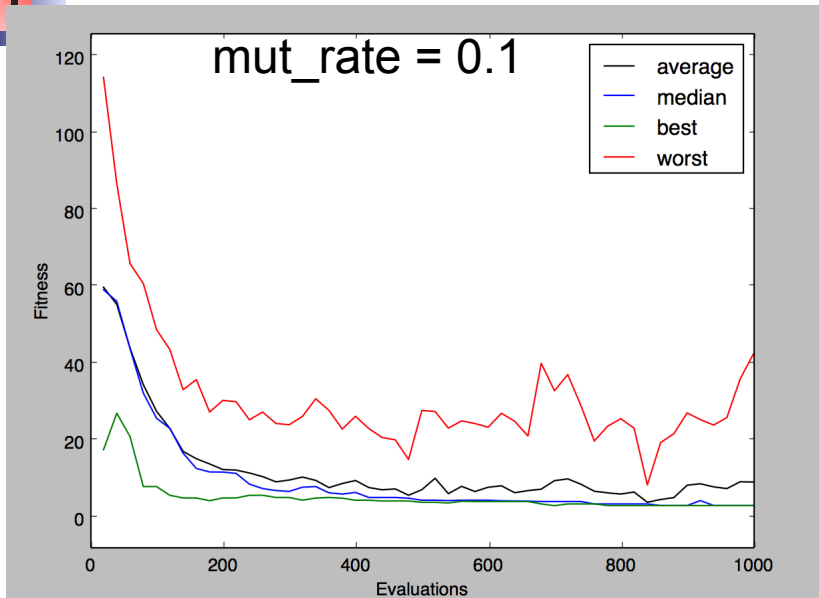
Note: This example should highlight the importance of mutation.

mutation

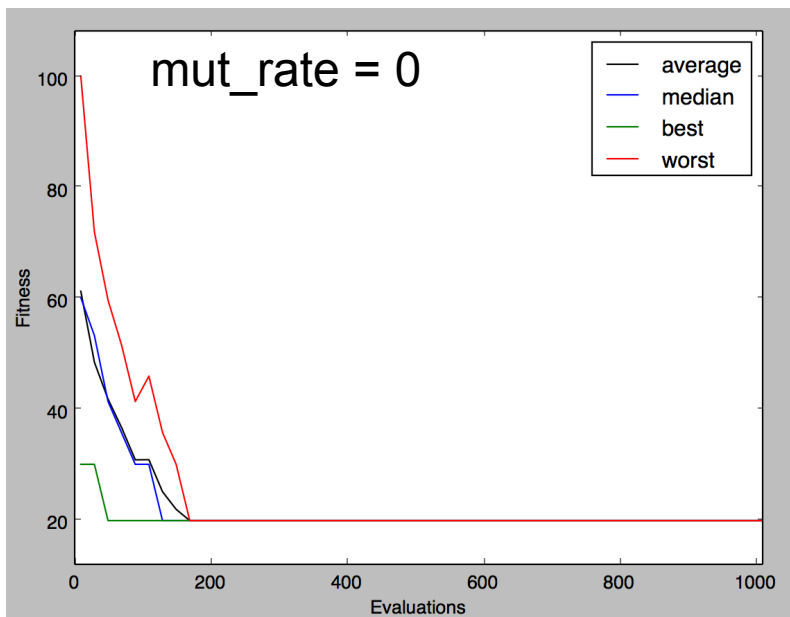


- What is the effect in convergence?
- How does the final fitness solution compare?
- Why is the effect stronger for smaller pop size?

mutation rate



- ❑ Why doesn't the mut_rate=2 converge/decrease?
- ❑ What happens when mut_rate=0?





Understanding elites

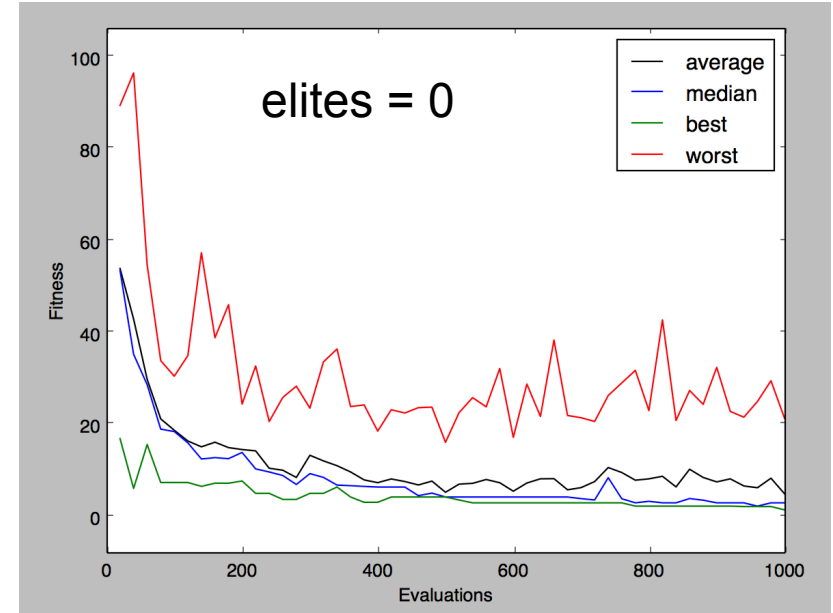
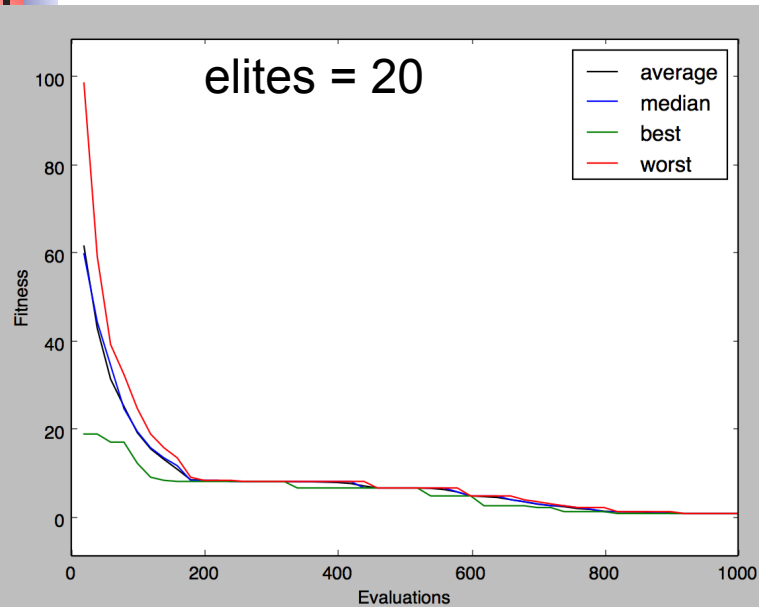
4) Using `pop_size=20` and `num_selected=20`, `mutation_rate=0.1`, change the current survivor replacement method (`steady_state_replacement`) for the generational replacement method (<http://pythonhosted.org/inspyred/reference.html#replacers-survivor-replacement-methods>)

Add the argument `num_elites=0` to the function call `my_ec.evolve(...)`. Compare the output for the following values of `num_elites`: 0, 1 and 20.

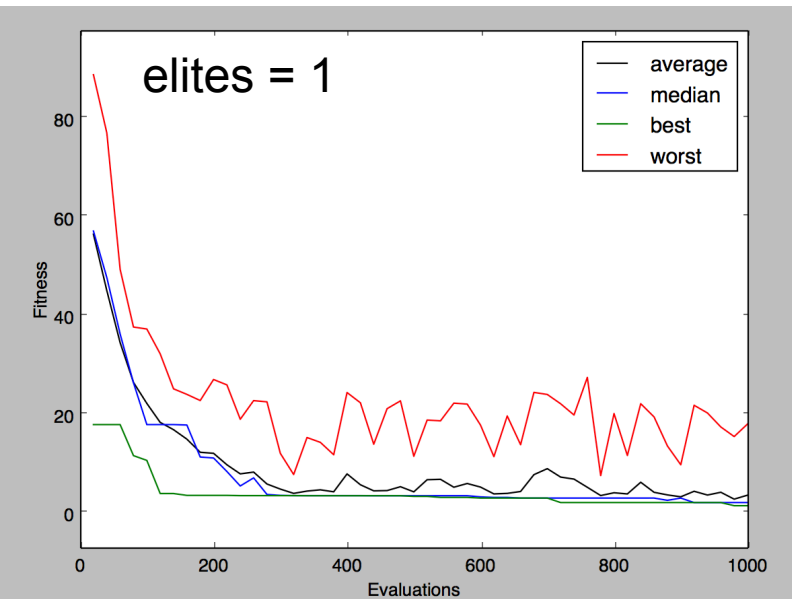
What's the effect? Notice any difference between 0 and 1?

Note: the best *num_elites* individuals in the current population are allowed to survive if they are better than the worst *num_elites* offspring.

num elites



- Why doesn't 'worst' line increase with 20 elites?
- Difference between 'best' line of 0 vs 1 elites?

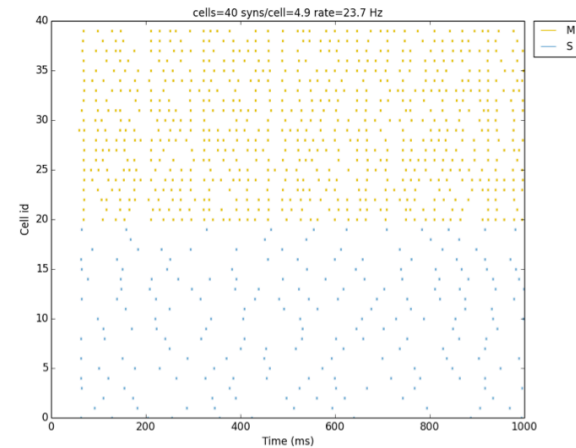
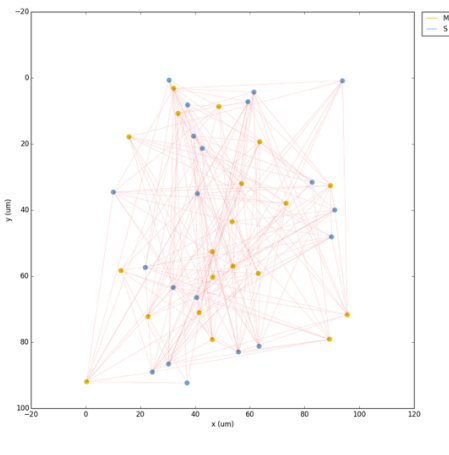


Network optimization

- *Aim*: optimize the network connectivity parameters to obtain a specific firing rate in the *tut2.py* example.

- *Parameters*:

- Probability (S->M conns)
- Weight (S->M conns)
- Delay (S->M conns)



- *Target (fitness measure)*: mean firing rate per cell (eg. 17 Hz)

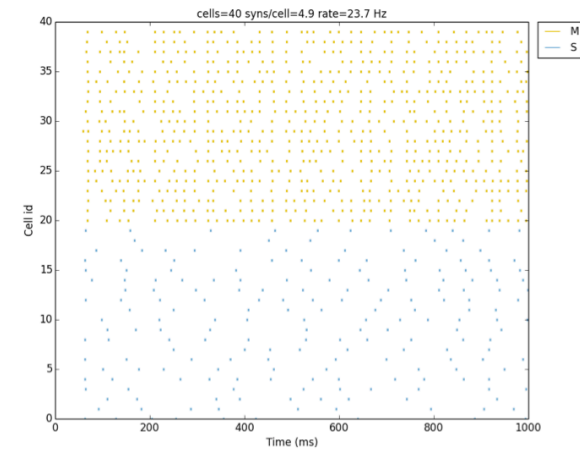
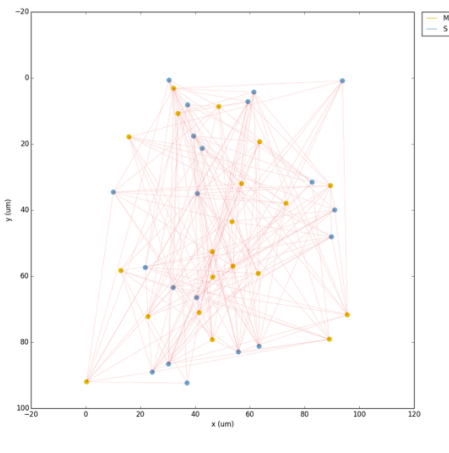
How to calculate fitness function? Minimize or maximize?

Network optimization

- *Aim*: optimize the network connectivity parameters to obtain a specific firing rate in the *tut2.py* example.

- *Parameters*:

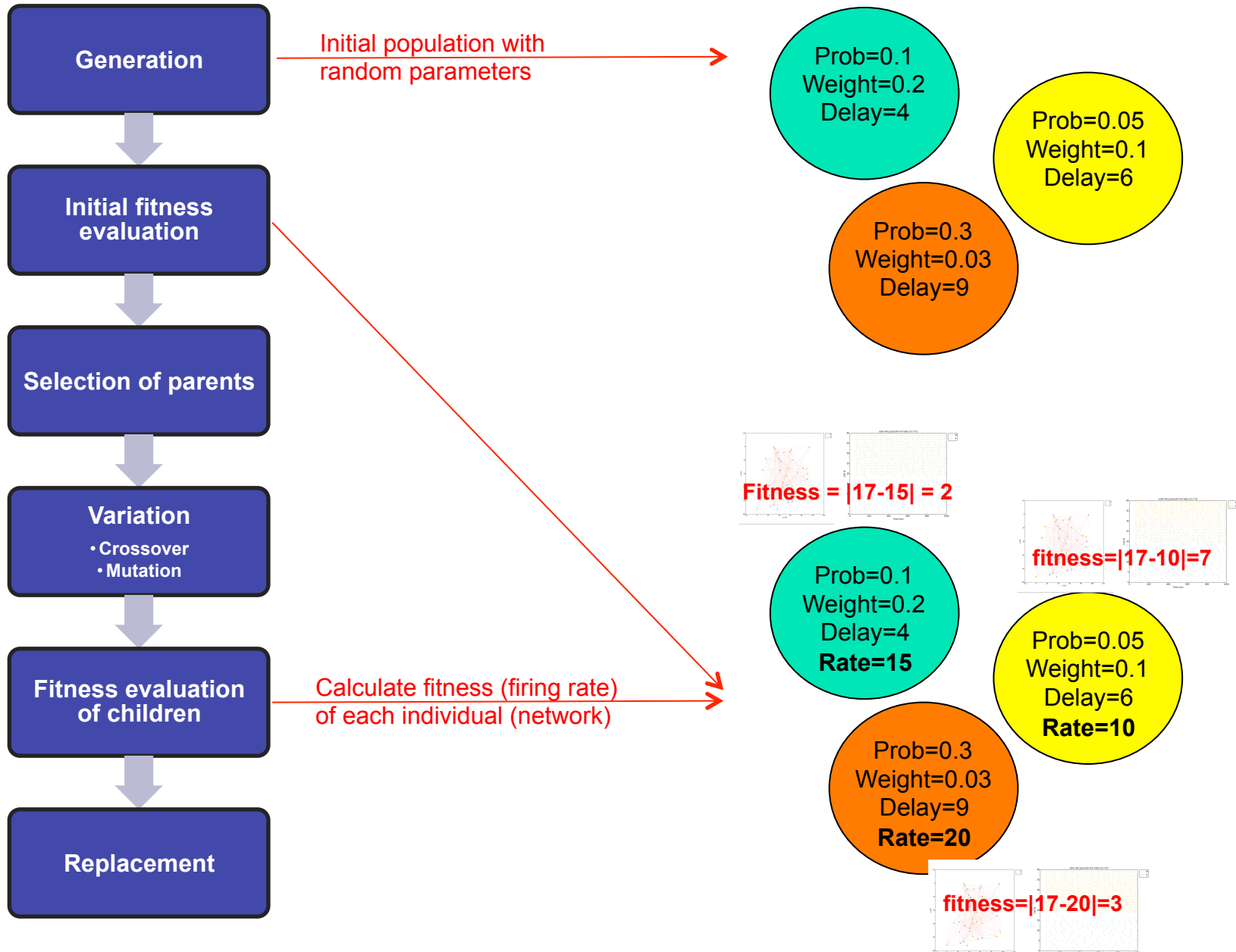
- Probability (S->M conns)
- Weight (S->M conns)
- Delay (S->M conns)



- *Target (fitness measure)*: mean firing rate per cell (eg. 17 Hz)

Fitness = | target rate – actual rate | (minimize!)

Network optimization





Network optimization

1) We will use the network model in *tut2.py* but need to change the following things:

- Since we are going to run the model many times for each generation of the evolutionary algorithm, we don't want plots showing up for each one.

Therefore, change the *simConfig* options so that NO plots are generated (set plotting of raster, cells, and 2d map to *False*) –

- Remove the *createSimulateAnalyze()* call from the end of the file – we will decide when to run the model from the optimization algorithm

- Change the duration of the simulation to 0.5 sec – since we are going to run the model many times we need to make it a bit faster

2) Now lets adapt the parameter optimization code. Start from *lab10_optim1.py* save as *lab10_optim2.py*. We will begin by changing the fitness evaluation function, so it creates and runs the neural network:

- add *import tut2* so we can use the network we defined there

- add *from netpyne import sim* so we can use netpyne to run the network

- Replace the function *evaluate_rastrigin* with *evaluate_netparams*, which should:

- Create an empty list called *fitnessCandidates*
- for each *candidate* create and simulate the *tut2* network
- calculate a *fitness* value for each candidate – for now just set this to a fixed value of 1 (we'll fix later)
- Add the *fitness* if each candidate value to *fitnessCandidates*

3) In the optimization algorithm options set in *my_ec.evolve*, modify the following:

- Set *evaluator=evaluate_netparams* so we make use of the newly defined fitness evaluator functions

- Set *pop_size=10, max_evaluations=50, num_selected=10, mutation_rate=0.2*

Network optimization

Creating simulation of 3 cell populations for 0.5 s on 1 hosts...

Number of cells on node 0: 40
Done; cell creation time = 0.00 s.

Making connections...

Number of connections on node 0: 197
Done; cell connection time = 0.01 s.

Running...

Done; run time = 0.33 s; real-time ratio: 1.51.

Gathering spikes...

Done; gather time = 0.00 s.

Analyzing...

Run time: 0.33 s
Simulated time: 0-s; 40 cells; 1 workers
Spikes: 441 (22.05 Hz)
Connections: 197 (4.92 per cell)
Done; plotting time = 0.00 s

Total time = 0.34 s

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
4	50	1	1	1.0	1.0	0.0

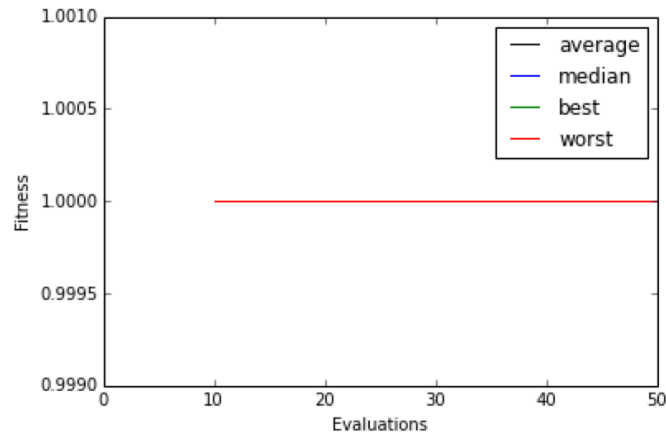
Best Individual: [1.1721428848721431, -5.12] : 1

/u/salvador/anaconda/lib/python2.7/site-packages/matplotlib/axes.py:2760: UserWarning:

Attempting to set identical bottom==top results
in singular transformations; automatically expanding.

bottom=1.0, top=1.0

+ 'bottom=%s, top=%s') % (bottom, top))



What are we missing??

Network optimization

Creating simulation of 3 cell populations for 0.5 s on 1 hosts...

Number of cells on node 0: 40
Done; cell creation time = 0.00 s.

Making connections...

Number of connections on node 0: 197
Done; cell connection time = 0.01 s.

Running...

Done; run time = 0.33 s; real-time ratio: 1.51.

Gathering spikes...

Done; gather time = 0.00 s.

Analyzing...

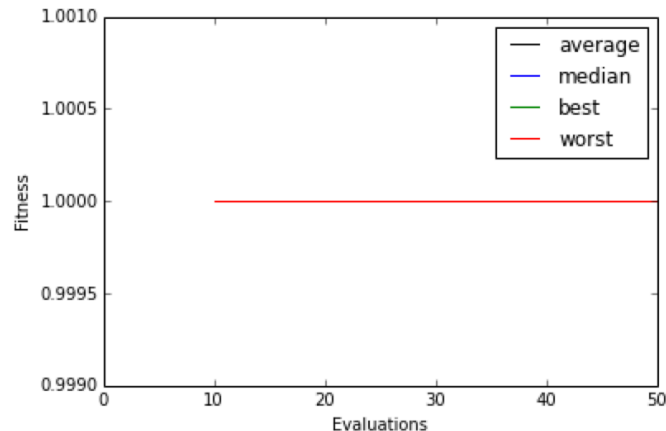
Run time: 0.33 s
Simulated time: 0-s; 40 cells; 1 workers
Spikes: 441 (22.05 Hz)
Connections: 197 (4.92 per cell)
Done; plotting time = 0.00 s

Total time = 0.34 s

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
4	50	1	1	1.0	1.0	0.0

Best Individual: [1.1721428848721431, -5.12] : 1

```
/u/salvador/anaconda/lib/python2.7/site-packages/matplotlib/axes.py:2760: UserWarning:  
Attempting to set identical bottom==top results  
in singular transformations; automatically expanding.  
bottom=1.0, top=1.0  
+ 'bottom=%s, top=%s') % (bottom, top))
```



What are we missing??

- Generate initial population with value within range
- Modify the network parameters based on individual evaluated
- Calculate fitness function correctly



Network optimization

1) In the main code, add a variable *targetFiring* to store the average target firing rate we want to obtain in the network, and set it to 18 Hz.

2) Add the minimum and maximum values for each of the 3 parameters (probability, weight and delay). These are needed during the generation of the initial population, and during the crossover and mutation phases – to make sure the genes of new children are within the allowed range:

- Create a list *minParamValues* to store minimum allowed values: 0.01 (for probability), 0.001 (for weight) and 1 (for delay)
- Create a list *maxParamValues* to store maximum allowed values: 0.5 (for probability), 0.1 (for weight) and 20 (for delay)

3) Replace the rastrigin generation function with *generate_netparams* using the following code:

```
def generate_netparams(random, args):  
    size = args.get('num_inputs')  
    initialParams = [random.uniform(minParamValues[i], maxParamValues[i]) for i in range(size)]  
    return initialParams
```

Make sure you also select it in the main code: *generator=generate_netparams*

4) Modify the bouncer (which makes sure parameter values are within the allowed range) so it makes use of the newly define parameter values: *bouncer=ec.Bouncer(minParamValues, maxParamValues)* ; and modify the number of inputs to 3: *num_inputs=3*

5) Modify the tut2 network parameters (prob, weight, delay) before creating and simulating it. The parameter values (prob, weight, delay) for each new candidate/child are stored in the list *cand*:

- the probability values can be accessed via *tut2.netParams.connParams['S->M']['probability']* and should be set to *cand[0]*
- the weight values can be accessed via *tut2.netParams.connParams['S->M']['weight']* and should be set to *cand[1]*
- the delay values can be accessed via *tut2.netParams.connParams['S->M']['delay']* and should be set to *cand[2]*



Network optimization

6) Calculate the firing rate of the network and fitness of each individual:

- Calculate the number of spikes, by finding the length of the list containing all the spike times: `sim.simData['spkt']`
- Calculate the number of cells, by finding the length of the list containing all the spike times: `sim.net.cells`
- Calculate the sim duration in seconds, using the `tut2.simConfig.duration` (which gives the duration in ms)
- Make sure the above values are stored as floats not integers (ie. can have decimals) – to convert any value to a float use the `float(value)` function
- Calculate the average network firing rate of the network by dividing the number of spikes, by the number of cells and the duration.
- Calculate the *fitness* value for this candidate as the absolute difference (use `abs()`) between the target firing rate (use the variable we defined in prev section) and the network firing rate (just calculated above)
- Store the candidate's *fitness* value in the list `fitnessCandidates`

7) For each candidate, print a message showing the candidate number (*icand*), the 3 parameter values of the candidate (prob, weight, delay), the firing rate of the resulting candidate network, and the fitness value of the candidate network, eg.

```
CHILD/CANDIDATE 9: Network with prob:0.01, weight:0.00, delay:10.5  
firing rate: 9.1, FITNESS = 8.95
```


Network optimization

```

Creating simulation of 3 cell populations for 0.5 s on 1 hosts
  Number of cells on node 0: 40
  Done; cell creation time = 0.00 s.
Making connections...
  Number of connections on node 0: 1
  Done; cell connection time = 0.00 s.

Running...
  Done; run time = 0.34 s; real-time ratio: 1.45.

Gathering spikes...
  Done; gather time = 0.00 s.

Analyzing...
  Run time: 0.34 s
  Simulated time: 0-s; 40 cells; 1 workers
  Spikes: 181 (9.05 Hz)
  Connections: 1 (0.03 per cell)
  Done; plotting time = 0.00 s

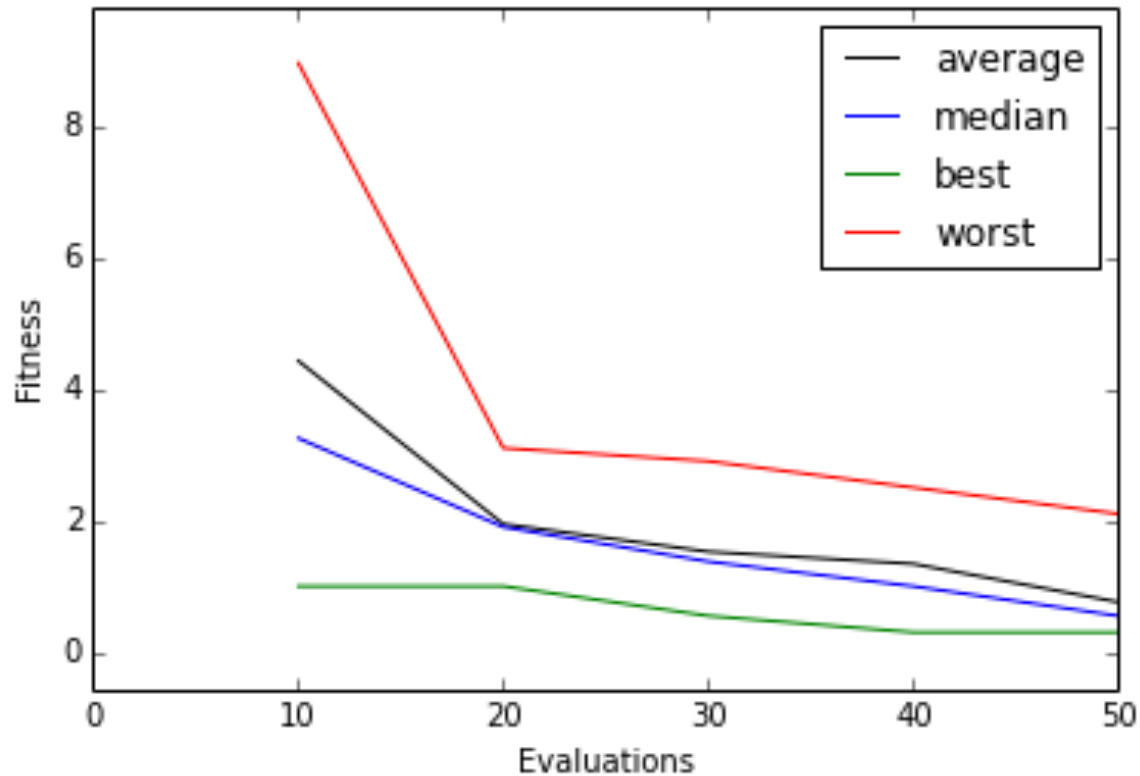
Total time = 0.35 s
  
```

```

CHILD/CANDIDATE 9: Network with prob:0.01, weight:0.00, delay:10.5
  firing rate: 9.1, FITNESS = 8.95
  
```

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
4	50	2.1	0.3	0.55	0.76	0.53329166

Best Individual: [0.27529211166881334, 0.06583998948192707, 18.171594793221356] : 0.3

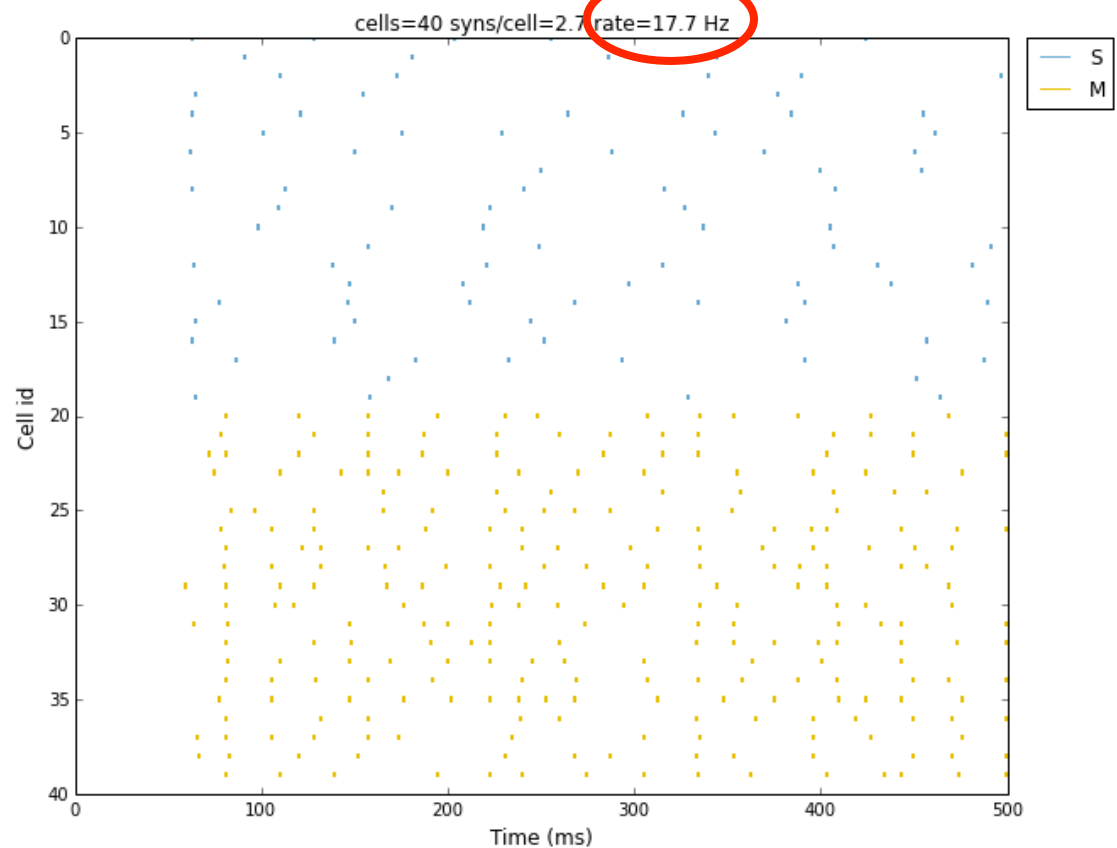


Network Optimization

8) Lets now check how the network with the optimized parameters found by the evolutionary algorithm looks! At the end of the main code, add the following:

- store in *bestCand* the parameters of the best individual (optimum solution) using the following code:

```
final_pop.sort(reverse=True)  
bestCand = final_pop[0].candidate
```
- Set the tut2 network parameters to those in *bestCand* (same as inside the evaluation function)
- Set the tut2 option to plot the raster: `tut2.simConfig['plotRaster'] = True`
- Create and simulate the tut2 network



Network optimization (Assessment)

1) Start from *lab10_optim2.py* and replace the fitness function so that instead of the target being the average firing rate, the target is now to maximize spike synchrony (ie. the synchrony of spikes of the different cells in the network):

- Add the following generic function to calculate spike synchrony.

```
def syncMeasure(spikeTimes, duration):  
    t0=-1  
    width=1  
    cnt=0  
    for spkt in spikeTimes:  
        if (spkt>=t0+width):  
            t0=spkt  
            cnt+=1  
    return 1-cnt/(duration/width)
```

- Modify the fitness evaluator function so that the fitness value for each candidate is equivalent to the synchrony of the network. Note you will need to use the *syncMeasure* above and pass the appropriate params: a list with the spike times of the network, and the duration of the simulation (you can find how to obtain both values looking at the previous examples)
- Since we want to maximize the synchrony of the network, make sure that the relevant evolutionary computation options is set to try to *maximize* the fitness (and not minimize it as before).
- Modify *tut2.py* so that you plot synchrony bars and the synchrony value for the network: *simConfig[plotSync] = True*

2) Replace the *delay* parameter (currently one of the 3 being optimized) with the decay time constant (*tau2*) of the network synaptic mechanism. Use the previous examples to figure out how to access this parameter in *tut2*. Set the min and max allowed values for this new parameter to 2 and 9.

3) Change the print statement for each candidate to reflect the above changes.

4) Replace the current *tournament selection* selector with *truncation selection* (only best individuals are selected).

Network optimization (Assessment)

CHILD/CANDIDATE 8: Network with prob:0.12, weight:0.05, tau2:5.5
sync: 0.71, FITNESS = 0.71

Creating simulation of 3 cell populations for 0.5 s on 1 hosts...

Number of cells on node 0: 40
Done; cell creation time = 0.00 s.

Making connections...

Number of connections on node 0: 143
Done; cell connection time = 0.01 s.

Running...

Done; run time = 0.09 s; real-time ratio: 5.38.

Gathering spikes...

Done; gather time = 0.00 s.

Analyzing...

Run time: 0.09 s
Simulated time: 0-s; 40 cells; 1 workers
Spikes: 473 (23.65 Hz)
Connections: 143 (3.58 per cell)
Done; plotting time = 0.00 s

Total time = 0.11 s

CHILD/CANDIDATE 9: Network with prob:0.36, weight:0.04, tau2:2.2
sync: 0.70, FITNESS = 0.70

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
4	50	0.706	0.804	0.75	0.7546	0.03292476

Best Individual: [0.5, 0.1, 8.344059017154711] : 0.804

