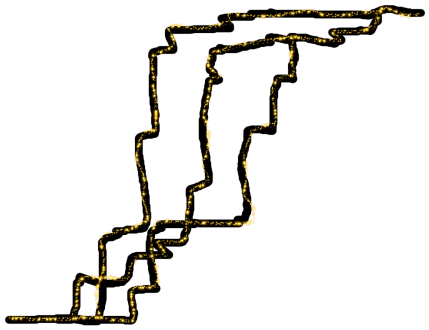
Explanatory of code mechanics of our Agent-based Evolutionary model

Department of chemistry and molecular biology

2017–03–28

Authors:
Forsberg, Timmy

E-mail:
Forsberg.Timmy@fuf.org
Forsberg.Timmy@live.com



Abstract

In this Manual: *Find descriptions of functions that constitutes the mechanics of an agent-based evolutionary growth model for yeast cells. The code is written in Python (2.7.6).*

1 Overview

The code grows yeast-cells that divide and mutate to adapt to an evolutionary pressure (a toxic environment). The growth happens in cycles where at the end of each cycle the population is reduced to then regrow in the next cycle. Over these cycles, agents collect mutations that can change their cell cycle time, making it divide faster or slower or neither. The size of such change is determined by already collected deletion data (i.e real measurements of yeast growth for loss of gene functions for each evolutionary pressure. The likelihood of such mutations is weighted by the yeast's number of Non-Synonymous nucleotides for each ORF and the number of stop-codons that can be created within each ORF from a mutation.

2 Model description

Model assumptions:

- Mutations are genetical,
- All mutations are loss of function mutations,
- No cells die, but can stop dividing.
- Mutations affect the duration of cell's cell cycle times.
- Mutational effects are additive and independent.
- A loss of gene function have the same effect as a gene deletion.
- A LoF mutation is a complete LoF mutation.
- A loss of gene functions is caused by either a stop-codon or non-synonymous mutation (in a conserved position, i.e SIFT < 0.5).

3 Function descriptions

3.1 def run()

For each cycle, iterate over time until the number of agents have grown to exceed the maximum allowed number of agents. This exit condition is checked in def. divide() which when fulfilled will set still_in_cycle to False. Then the relevant information is saved (line 28) and a new cycle can begin by randomly sampling some of the the individuals to repopulate the next cycle (line 30). In the new cycle all sampled agents goes through a lag phase again.

```
def run(func, const, data, environment, sim_env, mutation):

    # Initilize
    nr_alive = const.FOUNDER_COUNT

    for i_cycle in xrange(const.NR_CYCLES):
        t = 0
        lag = 1
        is_lagging = np.ones(nr_alive)
        still_in_cycle = 1

        while still_in_cycle:

            # Postpone division for lagging individuals
            lag, is_lagging, sim_env = \
                func.lag_f(const, sim_env, i_cycle, lag, is_lagging, t)
            save = func.save_growth(save,  nr_alive, t)

            # Make time tic
            t += const.TIME_STEP

            # Divide
            nr_alive, sim_env, mutation, still_in_cycle = \
                func.divide(const, data, mutation, sim_env, nr_alive, t, \
                    still_in_cycle, environment)

        # After a cycle
        save, distribution_gt = func.save_importants(const, save,  i_cycle,
    distribution_gt, nr_alive, t, mutation, sim_env)

        sim_env, mutation, nr_alive = func.sample_and_reset(const, mutation, sim_env)

    return save, mutation
```

3.2 def lag__f()

This function exists as a base if one would want to implement mutational lag effects. It works by postponing the division event (line 12) for each individual by one time step each iteration until that agent exits its lag phase. Since the time step overshoots the next division time, the next division time is readjusted by the size of the overshoot (line 16).

```
def lag_f(const, sim_env, i_cycle, lag, is_lagging, t):
    if lag:
        was_lagging = is_lagging
        if i_cycle == 0:
            is_lagging = (sim_env['lag_time'][:const.FOUNDER_COUNT] > t)
        else:
            is_lagging = (sim_env['lag_time'][:const.SAMPLE_COUNT] > t)

        lag = any(is_lagging)

        # Postpone next division
        sim_env['next_division_time'][is_lagging] += const.TIME_STEP

        # Readjust next division time for lag escapists
        lag_escapists = np.bool_(was_lagging * (1 - is_lagging))
        sim_env['next_division_time'][lag_escapists] -= \
        t - sim_env['lag_time'][lag_escapists]
    return lag, is_lagging, sim_env
```

3.3 def divide()

When an agents next division time is less than the current time it is time for that individual to divide. It is also possible to set a maximum number of divisions that an agent is allowed to make. After checking the cycle exit condition (line 11) the code moves on to copy the traits of all dividing agents. The daughter cells are then subjected to a mutational probability in the next function.

```
1  def divide(const, data, mutation, sim_env, nr_alive, t, still_in_cycle, environment):
2      to_divide = (sim_env['next_division_time'][:nr_alive] <= t) * \
3                  (sim_env['nr_divisions'][:nr_alive] <= const.MAXIMUM_NR_DIVISIONS)
4      nr_divide = np.sum(to_divide)
5
6      if nr_divide:
7          to_divide_nz = np.nonzero(to_divide)[0]
8          to_birth = np.arange(nr_alive, (nr_alive + nr_divide))
9
10         # Check if it's time to begin a new cycle.
11         if nr_alive + len(to_birth) >= const.MAXIMUM_NR_AGENTS:
12             nr_divide_reduced = const.MAXIMUM_NR_AGENTS - nr_alive
13             to_birth = np.arange(nr_alive, (nr_alive + nr_divide_reduced))
14             to_divide_nz = to_divide_nz[np.random.choice(nr_divide, nr_divide_reduced,
       replace=False)]
15             to_divide = to_divide_nz
16             nr_divide = nr_divide_reduced
17             sim_env['age'][:const.MAXIMUM_NR_AGENTS] += const.EXPERIMENT_TIME
18             still_in_cycle = 0
19             print 'About to exit cycle'
20
21         # Copy traits
22         sim_env['founder_id'][to_birth] = to_divide_nz
23         sim_env['lag_time'][to_birth] = sim_env['lag_time'][to_divide]
24         sim_env['cell_cycle_time'][to_birth] = sim_env['cell_cycle_time'][to_divide]
25         sim_env['next_division_time'][to_divide] += sim_env['cell_cycle_time'][
       to_divide]
26         sim_env['next_division_time'][to_birth] = sim_env['next_division_time'][
       to_divide]
27         sim_env['nr_divisions'][to_divide] += 1
28         mutation[to_birth] = mutation[to_divide]
29
30         sim_env, mutation = mutate(const, data, mutation, sim_env, nr_divide, to_birth,
        to_divide_nz, environment)
31         nr_alive += nr_divide
32     return nr_alive, sim_env, mutation, still_in_cycle
```

3.4 def mutate()

If a dividing daughter cell is mutating an ORF (represented by an index) is chosen. If that mutation already exist for that individual the mutation is rejected (line 12-18). To allow for 'unmutations' (i.e. mutations that restores a previous mutation), the target sizes for non-synonymous and stop-codon mutations for each ORF needs to be imported separately because of the target size for 'unmutations' is different then that of normal mutations.

After mutations have been stored (line 22), their affect on the cell cycle time is determined. Then the next division time is updated to eradicate for the change that was made in the previous function 'divide()'.

```
def mutate(const, data, mutation, sim_env, nr_divide, to_birth, to_divide_nz,
    environment):

    to_mutate = np.random.random(nr_divide) < const.MUTATION_PROB
    if any(to_mutate):
        # pick mutation
        mutation_cite = \
            np.random.random([sum(to_mutate), 1]) * (data.orf_target_size_cums[-1])
        # peek mutation
        orf_mutation = \
        np.searchsorted(data.orf_target_size_cums, mutation_cite, side='right')

        if ~const.OVERLAP_ALLOWED:
            overlap = (mutation[[to_birth[to_mutate]], :] == orf_mutation)[0]
            if np.any(overlap):
                overlapping_columns = np.nonzero(overlap)[0]
                orf_mutation = np.delete(orf_mutation, overlapping_columns)
                orf_mutation_nz = np.nonzero(to_mutate)[0]
                to_mutate[orf_mutation_nz[overlapping_columns]] = False

        # store mutation
        index_free_from_mutation = np.argmin(mutation[to_birth[to_mutate]], 1)
        mutation[to_birth[to_mutate], index_free_from_mutation] = orf_mutation.T[0]

        # change cell cycle time
        sim_env['cell_cycle_time'][to_birth[to_mutate]] = \
            sim_env['cell_cycle_time'][to_divide_nz[to_mutate]] + \
            data.gen_time[orf_mutation, environment] * const.MEAN_CELL_CYCLE_TIME

        # set next divition
        sim_env['next_division_time'][to_birth[to_mutate]] += \
            sim_env['cell_cycle_time'][to_birth[to_mutate]] - \
            sim_env['cell_cycle_time'][to_divide_nz[to_mutate]]

    return sim_env, mutation
```

3.5 Other functions

```

def save_growth(save,  nr_alive, t):
    save.append(('growth', nr_alive))
    save.append(('growth_time', t))
    return save


def save_importants(const, save,  i_cycle, distribution_gt, nr_alive, t, mutation,
    sim_env):
    save.append(('nr_haploid_types', calc_nr_haplotypes(const, mutation)))
    save.append(('mean_gt', np.mean(distribution_gt[:, i_cycle])))
    s = save_growth(save,  nr_alive, t)
    return save,  distribution_gt


def sample_and_reset(const, mutation, sim_env):

    # Sample for next cycle
    sample = \
        np.random.choice(const.MAXIMUM_NR_AGENTS, const.SAMPLE_COUNT, replace=False)
    sim_env[:const.SAMPLE_COUNT] = sim_env[sample]
    mutational_sample = mutation[sample]

    # (Re)set
```

```
24      nr_alive = const.SAMPLE_COUNT
25      mutation = - np.ones([const.MAXIMUM_NR_AGENTS, 5], dtype='int16')
26      mutation[:const.SAMPLE_COUNT] = mutational_sample
27      sim_env['next_division_time'][:const.SAMPLE_COUNT] = \
28          sim_env['cell_cycle_time'][:const.SAMPLE_COUNT]
29      return sim_env, mutation, nr_alive
```