Customizing Models

Thomas O. McDonald and Franziska Michor

September 7, 2016

Models can be customized by rewriting chunks of code and recompiling. Function placeholders were created to make it easy to locate and change code in these locations. The goal is to illustrate how to create unique models built around the infinite-allele framework.

Each section discusses a different way to customize the code starting with the most basic.

Changing Fitness Distributions

The fitness, mutation rate, and number of new mutations are each random variables that are generated upon a new mutation appearing in the population given the correct model parameters. The functions for generating these random variables exist in `rvfunctions.cpp`. The current distribution for fitness is hardcoded as a modified double exponential with an atom at zero.

The code in the simulation can easily be changed to support any two-parameter distribution with an additional parameter two define the probability of no fitness change (or the atom at zero). The parameters are inputted as alpha and beta, so if the current distribution is replaced with a normal, $N(\alpha, \beta)$, then we simply replace the function `double GenerateFitness(FitnessParameters fit_params)` to

```
double GenerateFitness(FitnessParameters fit_params)
{
  double fitness;
  double z = gsl_ran_flat(gp.rng, 0, 1);


  if( z < fit_params.pass_prob )
  {
    return 0;
  }
  else
  {
    // GSL uses gaussian with mean zero and standard deviation as 2nd parameter
    fitness = gsl_ran_gaussian(gp.rng, double beta) + gp.alpha;
    return fitness;
  }
}
```

Note: Rewriting this and any other function requires recompiling the code again. run `make` in Terminal in the **SIApop** directory to do so.

Time-Dependent Functions

Time-dependent models have a few more constraints on them, but custom functions can still be easily written. The predefined time-dependent functions are selected using an array of function pointers where an integer input (birth_function or death_function) determines which function is used. A custom function is built into the array associated with an input of 4. Modification of the function requires changing the code in `ratefunctions.cpp`. Any number of parameters are accepted for this function and are associated with the input parameter `td_birth_params` or `td_death_params` in the ancestor or input text files.

As an example, if we wish to use a cosine function (for whatever reason), rewrite the function `double RateFunctions::custom(double t, void *p)` to:

```
double RateFunctions::custom(double t, void *p)
{
  struct TimeDependentParameters *params = (struct TimeDependentParameters *)p;
  double additional_rate = params->additional_rate;

  double a = params->coefs[0];
  double b = params->coefs[1];
  double c = params->coefs[2];
  // currently a cosine function just for fun
  double y = additional_rate + a + b * cos(c * t);
  return y;
}
```

The argument `p` is a pointer to a structure containing model parameters. The only member of the structure to worry about is the double array `coefs` which should contain all parameters.

Recompiling the program is necessary after any changes are made. See the manual for how to use the new function and parameters.

Important notes: Functions must be bounded, nonnegative, and continuous almost everywhere. Piecewise functions can be used. Some functions may cause convergence issues and return an error. This will be fixed in the future.

New Mutation Effects

Finally, the ability to customize the effect of a mutation on new clones can be modified. The code is written so that each clone in the model is an element of a doubly linked list ordered by the time of appearance. When an individual splits and mutates, a new clone is formed as a node to be inserted at the end of the linked list. The clone takes all parameters from its parent by default. Adjustments to this clone (generating a new fitness, burst of mutations, etc.) can all be made to the clone before it is inserted at the end of the linked list via the function `InsertNode`.

`NewCloneFunction` is an abstract base class of functions meant to alter a new mutant clone that appears in the population. An instance of NewCloneFunction is created and a member function is assigned in `main` based on the value of certain parameters in the input. The fitness, punctuation, and epistatic models all use members of `NewCloneFunction` that adjust the parameters of the new clone. `NewCloneCustom` is included to allow customization of the model when a new clone appears. However, modifying this function in `clonelist.cpp` may require some modification of the header file `clonelist.h` and `main` as well. Any member of a clone structure can be modified.

The only required line of code in the function is

```
  cl.InsertNode(new_clone, parent_clone, 1);
```

attached to the end which inserts the node, new_clone, to the end of the linked list and links it to its parent, parent_clone.

As an example, suppose we wish to create a model where every mutant has a geometric number of mutations and when 3+ mutations arise, the death rate of the mutant increases by 10%.

```
void CloneList::NewCloneCustom::operator()(struct clone *new_clone, struct clone *parent_clone)
{
  // generate number of new mutations from geometric with p = 0.8
  int new_mutations =  gsl_ran_geometric(gp.rng, 0.8);
```

```
  if(new_mutataions >= 3)
  {
    new_clone->death_rate = new_clone->death_rate * 1.1;
  }
  else
  {
    new_clone->birth_rate = new_clone->birth_rate * 1.005;
  }

  // End with this piece - Insert new clone with new_mutations for number of mutations
  cl.InsertNode(new_clone, parent_clone, new_mutations);
}
```

For this example, this should be enough to run properly. If we wanted to include parameters from any of the other models, it would required including those structures in the function declaration in the header file. As an example, if we wanted to use the fitness parameters in `NewCloneCustom`, we would rewrite its declaration in `clonelist.h` to include the associated structure.

```
class NewCloneCustom : public NewCloneFunction
{
  public:
    NewCloneFitMut(CloneList& cl_, FitnessParameters fit_params_) : cl(cl_), fit_params(fit_params_)
    {
    }
    ~NewCloneFitMut(){};
    CloneList& cl;
    void operator()(struct clone *new_clone, struct clone *parent_clone);

  private:
    FitnessParameters fit_params;
};
```

and the assignment of NewClone in `main` should be changed to

```
if (gp.is_custom_model)
{
  NewClone = new CloneList::NewCloneCustom(population, fit_params);
}
```

IMPORTANT NOTE: an input parameter `is_custom_model` should be set to 1 in the text file to indicate using the function NewCloneCustom.

Again, recompiling the program is required after this modification.