# A C++ Template for Writing Medical Decision Models

Ayman Ali

Source code can be found at:
`https://github.com/namyaila/microsim_base` .

## Contents

## List of Figures

## List of Tables

# 1 Setup

## 1.1 Development

## 1.2 Common Issues with Building and Compiling

### 1.2.1 Windows

Probably the most common issue will be that Visual Studio can't find `<boost/file.hpp>`. If this happens, then make sure that you've downloaded boost. Once you have, make sure that you installed it in the include search path of whatever Visual Studio version you're working on. You can also just install boost on the desktop and add that to your include path. Doesn't really matter.

If you get an error building the library that deals with not being able to find functions that are in the standard library (cmath, vector, etc.) then all that's happening is the project is looking in the wrong spot for the includes. So, to fix this, go to the solution explorer and right click on the project. Click properties, and where it says Windows SDK Version, choose whatever version of Windows you're on.

# 2 Parameters

## 2.1 Configuration

Configuration parameters are set in a class that is initiated automatically at run-time and available as an external class called settings. Including `"configuration.h"` makes the class available and accessible through `parameters::settings`. The class is only used for parameters that are intended to remain constant for the duration of the program. The method reads configuration files located in `input\`.

### 2.1.1 Input File Format

The file format is meant to be simple, although admittedly fragile. Comments (lines beginning with a #) and newlines are ignored; lines containing input settings are tokenized by comma. The parser for this is located in the header file `"parse_aa.h"`. Assume that your configuration file reads:

```
# Simple example of a configuration file.
# Input the gender to run (0 = male, 1 = female, 2 = all), a double for step size, and a boolean for ca

1, 0.01231, 1
```

The code to parse this would be as simple as:

```
std::vector<std::string> tokenized_line;
// the parse function will automatically ignore comments
tokenized_line = parse::get_tokenized_line(input_file, comment_, delim);
// Need to staticly cast integers to enumerator values.
gender_ = static_cast<Gender>(std::stoi(tokenized_line[0]));
population_ = std::stod(tokenized_line[1]);
calibration_ = parse::string_to_bool(tokenized_line[2]);
```

#### 2.1.2 Overall Settings

#### 2.1.3 Calibration Settings

## 2.2 Literature Parameters

Literature parameters are defined as those that are model inputs, such as costs, transitions, rates of disease recurrence, or sensitivity and specificity of tests. Recently there has been a trend towards more complete sensitivity analyses for parameters estimated from literature; one and two-way sensitivity analyses are not enough. The current most common way to address this is with a probabilistic sensitivity analysis (PSA) in which we take into account the uncertainty in each parameter and assess the joint variance. In order to do so, each parameter estimated from literature must be assumed to follow a certain distribution; common ones are normal or lognormal distributions for costs, and beta distributions for probabilities. Therefore, literature parameters are constructed in this template such that subsequent PSA's are simple to set up and conduct; there is even code to generate PSA sets.

#### 2.2.1 Setting Literature Parameters in Model

The following types of literature parameter distributions are currently supported: beta, normal, gamma, lognormal, and uniform. The current parameter classifications are: probabilities, relative risks, utilities, and costs. Each literature parameter distribution derives from a base class which has the following important attributes:

1. `std::string key_` : An attribute to represent the name of the parameter, always a constructor arguments.
2. `double current_value_` : A current value to be used for the simulation (literature parameters are constant for every simulation, but are variable during PSAs). Note that more complex parameters, such as those that depend on previous attributes, would require you to write your own function to call a lookup table or whatever else you'd need.
3. `ParamType param_type_` : An attribute to represent the type of parameter, such as cost etc. listed above.
4. `virtual double standard_deviation()` : An overloaded function dependent on the distribution type. If you're using the distributions that I've provided, such as uniform, I've already overloaded them.
5. `virtual double pdf(const double value)`: Same as above.

Setting a literature parameter is quite simple and is hard-coded into the constructor (an unnecessary hassle to read in as an input) and, as an example,

```
Literature::Literature()
{
    using namespace distributions::literature;
    add_parameter(BETA, PROBABILITY, "DRINKING_TEA", 5, 10);
    // Note that the 5 and the 10 above refer to a probability of 5/10.
    // The first two parameters are enumerators that you'll see in the code
    // for distribution type and for parameter type
    add_parameter(GAMMA, COST, "SALARY", 100, 1500);
    // hint hint ^^ (but actually, shape and scale for the parameters)
}
```

Note that currently the function to add a parameter takes only two arguments; this is intentional because all of the distributions that I have currently implemented as literature parameters are uniquely defined by exactly two such arguments. In the future, if this has to change, it's a simple matter of overloading the function to accept more parameters.

### 2.2.2 Calling Values

Each parameter is by default set to the mean; in order to call the value, however, simply call the default current value. For example,

```
if (modeling::distributions::uniform(patient->generator, 0, 1) <
    patient->literature.params_["P_EAT_TOO_MUCH_SUGAR"].get_current_value())
{
    patient.stroke = true;
}
```

I tend to keep a shared pointer to a generator and to literature parameters in the patient or person object running through the simulation. There are other methods.

## 2.3 Calibrated Parameters

Calibrated parameters are those that are unknown from literature, cannot be easily estimated into an input parameter, and must be derived from repeated model iteration fits to targets. For example, the rate at which a normal cell may become cancerous given certain conditions (age, sex, body-mass-index, calendar year, and so forth) is completely unknown, but the amount of people that develop cancer at a certain age and year is a target that we can hit. Typically, we estimate these parameters with one of three models: a constant parameter, a linear one, and logistic.

Calibrated parameters work much differently then literature because of the added complexity of reading parameters into the model on a regular basis (we often read configuration files for different parameter sets). Therefore, when we initialize a calibrated parameter object, the constructor requires a string for the input file name. However, there is a default, which is `../input/calibrated_params`.

### 2.3.1 Setting Calibrated Parameters in Model

There are two things that you must do when you create and set a calibrated parameter in the model. The first is to hard-code the parameter name and type of parameter into the model. To do this, navigate to `calibrated_params.cpp` and find the function `void Calibrated::set_up_default()`. In this function you must add the parameter and the type of parameter to the class member `params_` which is the member that we always access calibrated values from. For example:

```
void Calibrated::set_up_default()
{
    using distributions::calibration::Constant;
    using distributions::calibration::Linear;
    using distributions::calibration::Logistic;

    params_["BATHROOM_AFTER_COFFEE"] = Constant();
    params_["NEED_HELP_NOW"] = Logistic();
}
```

Once that's set up, then you must navigate to the input file that you call. Recall that in most cases, that'll be the default file mentioned in the previous section. Once there, you must add the parameter in the exact way described in the default file. The instructions, for documentation reasons, are provided in Figure 1.

The functions that parse the file are all written and so you don't have to worry about error-checking there. However, because this is a sensitive part to the program, any issues here will cause a helpful (hopefully) error and subsequent program termination.

Figure 1: Format of Calibrated Parameter Input Files

```
# The format for this file is very specific, so pay attention closely.
# Spelling and capitalization really matters ... especially for the
# type of the parameter.
# The first thing to type is the parameter name exactaly how you will
# be calling it in the model code: for example, "COUCH_TO_5K".
# Then, type the type of the parameter as either: CONSTANT, LINEAR, or
# LOGISTIC.
# Then, you must type the current value, lower bound, and upper bound of
# EACH DISTINCT DEFINING VALUE OF THE CALIBRATED VALUE.
# A constant has only one, since it's constant; a linear has two, which is
# the intercept and slope; a logistic has three, which is
# horizontal asymtote, shift, and steepness.
# The ORDER OF DEFINING VALUES MATTERS, and it's the same as described
# above. Again, the capitalization of keys also matters (alot).
# The program will terminate (with a nice error) if anything is missing.
# Note that everything must be comma separated. An example parameter is below.
# Add as many parameters as you'd like. Comments (#), newlines, and
# whitespace are ignored.

BATHROOM_AFTER_COFFEE, CONSTANT, 0.5, 0, 1
NEED_HELP_NOW, LOGISTIC, 1, 0, 1,    1, 0 , 1,    1, 0, 1
```

### 2.3.2 Calling Values

Implementation of this is slightly tricky, but using it should be trivial. For a constant parameter, simply call the member function `get_value()`. Linear and logistic parameters, however, depend on a current value. Because I don't assume that it's an integer or a double (age or BMI, for example), there is a template'd function that is simply called with `get_value(x)`. This way, the program will not crash depending on the type of x. As examples, using the parameters we have been using above:

```
if (modeling::distributions::uniform(patient->generator, 0, 1) <
    patient->calibrated.params_["NEED_HELP_NOW"].get_value())
{
    patient.help_now = true;
}

if (modeling::distributions::uniform(patient->generator, 0, 1) <
    patient->calibrated.params_["BATHROOM_AFTER_COFFEE"].get_value(
        patient.current_age)) // can use current age or BMI and no issues
{
    patient.bathroom = true;
}
```