# Documentary for BCNetwork simulation related Codes.

Purpose of this article is to create an index for the classes and modules defined in the projects. And guide the future development based on the work.

## Biochemical models storage and conversion

### BCNetwork Class Template

defined in "BCNetwork.h"

```
template<typename compType, typename rateType, typename powerType>
class BCNetwork
```

This module is the basic module to store the information necessary to describe a biochemical network. It is supposed to be able to handle all types of biochemical networks(from elemental to huge, both discrete and continuous.)

| type name | meaning |
|---|---|
| `compType` | data type of the amount of reactants |
| `rateType` | data type of the reactant constants |
| `powerType` | data type of both rate matrix and update matrix |

#### Public variable members

| `<type> variable_name` | meaning |
|---|---|
| `<double> t` | current time of the network |
| `<int> nComp` | number of reactants |
| `<compType> * comp` | pointer to datablock that store amount of each reactants |
| `<compType> * compBackup` | same as `comp`, a back up datablock for resetting purpose |
| `<int> nRate` | number of rate constants |
| `rateType * rate` | pointer to datablock that store each rate constants |
| `powerType * rateMatrix` | pointer to datablock for rate matrix. Format: `i*nComp+j` |
| `powerType * updateMatrix` | pointer to datablock for update matrix. Format, as above. |

#### Public function members

```
int loadParameter(vector<compType> &, vector<rateType> &, powerType *, powerType *);
```

Load parameter from a certain model. Four types of parameters required to fully describe a model are required. the four inputs are respectively : amount of reactants as a form of vector; amount of rate constants as a form of vector; rate matrix as a datablock of same format as described in variable members; update matrix as a datablock of same requirement.

loadParameter will import all the information obtained from the input into the inbuilt dataset. So the destruction of input source after the import will not disturb the funtion of this modules or any modules that inherit from it.

Importing a second network will overwrite the previous one stored in. The previous network will be automatically completely erased before a second network is written in, so backup is suggested before this action.

---

```
int fileOpen(string & condition);
int fileClose();
```

Open and close a data saving file. `fileOpen(string &)` will open a data file with name "./result/sim$(condition)". New file will be created if it is not exist. And older data willed be wiped in the file with a same name. Note that new folder won't be created if not exist. So a folder "result" created under the working directory before running the code is suggested.

Note that the folder name "./result/" is defined in "basicDef.h" as

```
#define RESULTFOLDER "./result/"
```

---

```
virtual void saveData();
```

Essentially this function will toss a copy of the current time and amount of each reactants into the data file opened by `int fileOpen(string &)`. Data will be saved in a file named "./result/simEmergency" if no file is opened. But open a file before saving the data is still suggested.

Note that this member function covers only the basic saving feature and changable in classes inherit this class template.

---

```
virtual void reset();
```

Restore the datablock that `* comp` pointing to with the datablock pointed by `* compBackup`. And again, this member function is preliminary and changable in classed derived from this class template.

### Constructor and Destructor

```
BCNetwork();
BCNetwork(vector<compType> & initComp, vector<rateType> & inputRate,
             powerType * inputRateMatrix, powerType * inputUpdateMatrix);
~BCNetwork();
```

`BCNetwork()` only setting private members to correct initial value for the class to function correctly. `BCNetwork(vector<compType> &, vector<rateType> &, powerType *, powerType *)` will call `int loadParameter(...)` to load a model when initiating the class. `~BCNetwork()` will free all memory that allocated from this class template before its destruction.

## modelLoadder Class Template

defined in "modelLoader.h"

```
template <typename compType, typename rateType, typename powerType >
class modelLoader
```

This class template is merely used to load a model from file. The use of this class is temporary before it's function incorporated into `BCNetwork` class template.

### public function members

```
void loadParameter(string & mName);
```

This is the only functional member. It's used to load model parameters stored in folder "./mName". The format of model parameters will be specified in a model creation/modification project. To completely load a model, four files will be read. The name of them are specified as macros stored in "basicDef.h":

```
#define RATEFILE "rate"     // reaction rate constants, first line is the number of
reactions
#define INITCONDFILE "initCond"     //initial condition for each reactants, first line is
the
                                    //number of reactants
#define RATEMATRIXFILE "rMatrix"    //rate matrix, same format as in datablock
#define UPDATEMATRIXFILE "updMatrix"    //update matrix, same format as in datablock
```

### Constructor/Destructor

```
modelLoader(string & mName);
~modelLoader();
```

`modelLoader(string &)` calls `loadParameter(string &)`. This is the only valid way of initiating modelLoader class. Other ways of initiating this class may involve unexpected behavior.

## ODENetwork Class

defined in "ODECommon.h"

```
class ODENetwork : public BCNetwork<double, double, int>
```

This is the basic class for biochemical networks described by ODEs. In addition to the features already defined in `BCNetwork`class template, the ODEs required for simulation and time variable will be defined in this class. But neither simulation nor analysis feature is defined in this class.

These chemical networks are defined in continuous space and are deterministic. Thus both reactant amount datatype and reaction rate datatype are double.

Note, the two matrices in most cases are integer, thus the powerType is defined so. For specific cases that require non-integer powerType, change the type accordingly.

### Protected member

| `<type> variable name` | meaning |
|---|---|
| `double h0` | initial size of time step |

### Public function members

```
void ODETimeDeri(double * timeDeri, double * component);
```

This is the ODEs generated from the biochemical network. The usage of the ODEs is to feed 2 pointers to the function. `* component` point to the datablock of current reactant amounts; `*timeDeri` point to the datablock that stores the time derivations of each reactants' amount.

Note that the datablock pointed by timeDeri will be changed after this function is called. Backup is suggested if there is anything you don't want to be erased there.

### Constructor/ Destructor

```
ODENetwork(vector<double> & initComp, vector<double> & inputRate,
            int * inputRateMatrix, int * inputUpdateMatrix, double
            initTimeStep) :
            BCNetwork<double,double,int>
            (initComp, inputRate, inputRateMatrix, inputUpdateMatrix);
```

Well, basically it's self explined. It does two things. It calls the constructor that loads a biochemical network, and give `h0` an initial value given by user.

# Algorithms

## Gillespie Algorithm

defined in "gillespie.h"

```
class gillespie: public BCNetwork<int, double, int>
```

This class defines both the algorithm and the simulation methods. Gillespie algorithm is an exact simulator for biochemical reactions. Continuum approximation won't work here. Thus the datatype of reactant amounts are integer. So far I didn't see the necessity of seperating them. But when there is a need, they might be seperated in the future.

### Public variable members

| `<type> variable name` | meaning |
|---|---|
| `double stoppingTime` | the stopping signal for the simulation |
| `long long int savePointInterval` | the interval of saving points by count of steps |
| `long long int nOfNewStep` | No. of steps since the last saving point |
| `double saveTimeInterval` | the interval of saving points measured by time interval |
| `double lastSavedTime` | the time point in which last saving action has taken places. |
| `bool saveMethod` | 0=save every `savePointInterval` steps;<br>1=save every saveTimeInterval of time; |
| `bool noSave` | If noSave signal is 1, no data saving is allowed |

### Public function members

```
void simulate();
```

This function will iterate the network with Gillespie algorithm till `BCNetwork::t` (defined in BCNetwork class template) surpasses `stoppingTime`. During the simulation, once the saving condition is satisfied, the current time and the current datablock that `BCNetwork:: *comp` points to will be saved into file specified by `BCNetwork::fileOpen(string &)`.

---

```
inline void reset();
```

This reset function reserves the function of `BCNetwork::reset()`. It also restore `BCNetwork::t`, `lastSavedTime` and `nOfNewStep` to 0.

### Constructor/Destructor

```
gillespie(vector<int> & initComp, vector<double> & inputRate,
        int * inputRateMatrix, int * inputUpdateMatrix,
        double runTime ,bool sMethod=0, double saveInterval=1)
```

It calls `BCNetwork::loadParameter(...)` to load a biochemical network. `stoppingTime` will be determined by `runTime`, `saveMethod` will be determined by `sMethod`. `saveInterval` will determine the value of either `savePointInterval` or `saveTimeInterval` considering which `saveMethod` user choose.

### Further note

The random number generator used by this class is rand48 given in gcc library. Involved function members are

```
inline void reseedRandom(int seed);
inline double popRandom();
```

Should other random number generator be used instead the current one. These two member functions can be changed accordingly.

## ODEIVPCommon Class Template

defined in "ODECommon.h"

```
template<typename modelClassType>
class ODEIVPCommon
```

This class is a basic class template for all algorithms created for ODE's Initial Value Problem simulation. Currently only Runge-Kutta based simulator is created from it.

`modelClassType` is a typename specified by the models that algorithms are applied upon. It is used by member function pointers of the algorithms for proper scope names.

## public variable members

| `<type> variable name` | meaning |
|---|---|
| `double ht` | current stepsize. For adaptive stepsize algorithms, this value is usually different from `ODENetwork::h0` |
| `int varNumber` | number of variables/ODEs. |
| `void (modelClassType::*ODEs)(double *, double *)` | member function pointer. Used to point to ODEs function provided by the specific model. |
| `int (*Normalizer)(double *)` | a normalizer, usually need to be specified for each different problem. Thus by definition it's not bound to the member class. Note that this is a function pointer, only static function can be pointed by it. |

## Public function members

N/A

## Constructor/Destructor

```
ODEIVPCommon();
ODEIVPCommon(int sysSize, double initTimeStep,
        void (modelClassType::*targetODEs)(double *, double *));
ODEIVPCommon(int sysSize, double initTimeStep,
        void (modelClassType::*targetODEs)(double *, double *),
        int (*targetNormalizer)(double *));
```

Three constructors are defined for this class template.

Default constructor does nothing.

`ODEIVPCommon(int , double, void (modelClassType::*)(double*, double*));` is used when no normalizer is required. This constructor will load the first parameter as `varNumber`, second parameter as the initial value for `ht`, and third parameter as the address function pointer `void (modelClassType::*ODEs)` pointing to. Normalizer will be specified to `int blankNormalizer(double *)` defined in "ODECommon.h". Normalizer can be otherwisely specified to other functions defined by user afterwards.

`ODEIVPCommon(int , double, void (modelClassType::*)(double*, double*), int (*)(double *));` is used when normalizer is required. First 3 parameters have the same meaning. And the last parameter is used to specify the pointer pointing to the normalizer function.

## RKmethod Class Template

defined in "rungeKutta.h"

```
template<typename modelClassType>
class RKmethod: public ODEIVPCommon<modelClassType>;
```

This is an algorithm class template using adaptive stepsize Runge-Kutta method for ODEs simulation of initial value problems. This class will do one thing, and one thing alone. Given current values of all components, their value of next time step and the time interval of current iteration will be returned. Users can implement this algorithm in their model for simulation and do any analysis they want with the returned data.

Note that due to the concern of reliable use of this algorithm, this class is not supposed to be directly modified by any means, thus most functions are kept private.

### Public function members

```
double iterator(double * var);
```

This function takes in the datablock storying current value of the variables pointed by `* var`. New value after one step of iteration will be directly written into the datablock `* var` pointing to. Time interval for this step of iteration will also be returned as a double value. User can use the time interval to determine the exact time value after this iteration.

### Constructor/Destructor

```
RKmethod();
RKmethod(int sysSize, double initTimeStep,
        void (modelClassType::*targetODEs)(double *, double *)) :
        ODEIVPCommon<modelClassType>::ODEIVPCommon(sysSize, initTimeStep, targetODEs);
RKmethod(int sysSize, double initTimeStep,
        void (modelClassType::*targetODEs)(double *, double *),
        void (*targetNormalizer)(double *)):
        ODEIVPCommon<modelClassType>::ODEIVPCommon(sysSize, initTimeStep,
        targetODEs,targetNormalizer);
```

The use of the three constructors given by this class template mirror the three constructors of `ODEIVPCommon`. The first one only initiate the private variables to their working states; Second constructor is used for the case that no normalizer is required; third constructor is used when a normalizer is required for a successful simulation.

# User facing modules for Biochemical Network simulation

Class `gillespie` described in algorithm classes also falls to this catagory.

## ODESimulate Class

defined in "ODEOperation.h"

```
class ODESimulate : public ODENetwork , public RKmethod<ODESimulate>
```

This class use Runge-Kutta algorithm to do simulation on IVP of ODEs defined by ODENetwork.

## Public variable members

| `<type> variable name` | meaning |
|---|---|
| `double stoppingTime` | the stopping signal for the simulation |
| `double saveTimeInterval` | the interval of saving points measured by time interval. |
| `double lastSavedTime` | the time point in which last saving action has taken places. |

## Public function members

```
void simulate(string & identifier);
```

This function will simulate the system and store the result in a file specified by `& identifier` (it automatically call the function `BCNetwork::fileOpen(string &)`).

---

```
void reset();
```

This function retains the function of `BCNetwork::reset()`. It also restore `lastSavedTime` to 0. And restore `ODEIVPCommon::ht` to the value specified by `ODENetwork::h0`.

## Constructor and Destructor

```
ODESimulate(vector<double> & initComp, vector<double> & inputRate,
        int * inputRateMatrix, int * inputUpdateMatrix,
        double initTimeStep, double runTime, double saveInterval) :
        ODENetwork(initComp, inputRate, inputRateMatrix, inputUpdateMatrix,
        initTimeStep), RKmethod<ODESimulate>(nComp, h0, &ODESimulate::ODETimeDeri);
```

Notice that no default constructor is defined. This constructor will load a typical biochemical network, and call the constructor of `RKmethod` that doesn't use normalizer. Two additional information is loaded to specify `stoppingTime` and `saveTimeInterval`.

```
    stoppingTime=runTime;
    saveTimeInterval=saveInterval;
```