Classes for simulating networks/ODEs

The idea is :

- a model is a list of variables and equations to update them
- it depends on a mother class called 'Modele', which performs more advanced functions (by calling it), such as printing or recording a time-course, managing a simulation (when it diverges for instance), reading a parameter set from a file, etc,
- finally, one can use more complex classes (look at 'Experiment' class), which use a model to perform different experiments, like different initial conditions, or doing the same experiment in different backgrounds.

1 - Coding a model (i.e. sub class of the Modele class) :

The mother class "Modele" provides tables to stores/work with parameters and variables for a time point. They don't need to be re-defined, and can be directly used in the sub-class.

```
    int t;                  // time in seconds
    vector<double> init;    // Initial variable values
    vector<double> params;  // Parameter values
    vector<double> val;     // value of the variables at the time point t
    vector<string> names;   // names of the variables
```

A model should implement (working on the tables provided by the mother class: init, params, val, ...):

- an ENUM of variable names and parameters. Note that it should provide the mother class with the number of variables and parameters when calling the 'Modele' constructor
- a function returning derivatives at time t from vector x, into vector dxdt

  virtual void *derivatives*(const vector<double> &x, vector<double> &dxdt, const double t);
- a function to initialize a simulation (doing t = 0 if necessary), and calling *initialiseDone();*

  the function is designed to be able to give an option, like a genetic background, for initializing

  virtual void *initialise*(*int option = 0*);
- a function to set basal parameter values, and calling *setBaseParametersDone();*

  virtual void *setBaseParameters*();
- give the name of the variables by filling names[]

Example of a minimalistic model, (population model with A --> B --> C --| A)

```
struct ModeleA1 : public Modele {
    ModeleA1();
    enum {A, B, C, NBVAR};                                  // List of variables
    enum {KF, KD1, KD2, KD3, K12, K23, KD, NBPARAM};        // List of parameters
    void initialise(int option = 0);
    void setBaseParameters();
    void derivatives(const vector<double> &x, vector<double> &dxdt, const double t);
};

ModeleA1::ModeleA1() : Modele(NBVAR, NBPARAM) {
    names[A] = string("PopA");
    names[B] = string("PopB");
    names[C] = string("PopC");
}

void ModeleA1::setBaseParameters(){
    params.clear();             params.resize(NBPARAM);     // to erase previous values
    params[KF]   = 1e-3;        params[KD1]  = 1e-4;
    params[KD2]  = 1e-6;        params[KD3]  = 1e-6;
    params[K12]  = 1e-6;        params[K23]  = 1e-5;
    params[KD]  = 0.1;
    setBaseParametersDone();
}

void ModeleA1::initialise(int option){
    val.clear();    val.resize(NBVAR, 0.0);                 // to erase previous values
    init.clear();   init.resize(NBVAR, 0.0);
    for(int i = 0; i < NBVAR; ++i){val[i] = init[i];}
    t = 0;
    initialiseDone();
}

void ModeleA1::derivatives(const vector<double> &x, vector<double> &dxdt, const double t){
    dxdt[A] = params[KF] * (1- x[C] / params[KD]) - (params[KD1] + params[K12]) * x[A];
    dxdt[B] = params[K12] * x[A] - (params[KD2] + params[K23]) * x[B];
    dxdt[C] = params[K23] * x[B] - params[KD3] * x[C];
}
```

Then, the basic use of the sub-class will be : (using the 'simule' fonction from the mother class). The *simulate* function is provided by the mother class 'Modele' and updates val[] by solving the ODEs for the given time (here, 36000 seconds)

```
Modele* M = new ModeleA1();
M->setBaseParameters();
M->initialise();
M->simulate(36000);
```

List of options provided by the 'Modele' class :

```
struct Modele {

/// The general constraints the submodel should follow are here :

    /// 0 - Calling the constructor : the sub-model calls says the nb of variables and parameters to allocate
    Modele(int _nbVars,  int _nbParams);

    /// 1 - the storage is provided by the mother class memory, (only accesssible by the sub-class but not outside)
    ///     note that the mother class only allocate them with a good size, but never operates/changes values in these 6 fields,
    ///     so the subclass should do everything
    vector<double> init;                                    /// for initial values of variables
    vector<double> params;                                  /// for parameter values
    vector<string> names;                                   /// for the names of the variables

    ///     for running a simulation, these variables will be used
    int t;                                                  /// updated during simulation / advised to do 't=0;' in the subclass function initialise
    vector<double> val;                                     /// for variables at time t

    /// 2 - The functions the sub-model HAS TO implement : time evolution for dt at time t, initialise and base parameter values
    virtual void derivatives(const vector<double> &x, vector<double> &dxdt, const double t);      /// computes the derivatives (at t) from position x
    virtual void setBaseParameters();                       /// gives a correct set of parameters, if you don't load other set.
    virtual void initialise(int _background);               /// initialise, (parameters should be set before initializing) - sets t = 0 if necessary.
                                                            /// to avoid confusion, this function should not change the parameter values !!
                                                            /// the background parameter allows to give options of simulation (such as deficient mice)

    /// 2b - This function is already implemented in the mother class, but as an option you can override it in the subclass
    virtual void loadParameters(string file_name);          /// reads parameters from a text file with format : "NB_PARAMS\tparam1\tparam2\tparam3 ..."

    /// 2c - the 'event' functions that the previous functions should call, so the mother class knows what's happening,
    void initialiseDone();
    void setBaseParametersDone();
    void loadParametersDone();

/// Now, the interests of using the Modele mother class comes :

/// 3 - functions for simulating (automatically calling the sub-class one-step function)

    void simulate(int _sec_max, Evaluator* E = NULL);       /// Simulates from t (not changed), to t + _sec_max, by calling one_step
                                                            /// the evaluator, if given, is a structure that says when to store data from the simulation, and avoids to store everything.
                                                            /// if the simulation diverges, it is stopped and a penalty is computed in the field penalities (of the mother class)

    public:    double dt;                                   /// minimum dt used for simulations. The tunable dt will start at dt*10 and be limited by dt*100
    public:    double penalities;                           /// automatically updated by the mother class : put to zero when initialiseDone() ; increased when the simulation diverges.

public:
/// 4 - I/O functions

/// 4a - to print the state of a simulation

    void print_header(int _t);      /// List of variables
    void print_state(int _t);       /// Values for each variables
    void print_parameters();        /// Values for each parameter

/// 4b - to get simulations as tables of values (kinetics) of simulation

    /// before a simulation, one has to activate the kinetic mode.
    /// Then, for every simulation, the simulation data will be put in a new table, that can be retrieved by getCinetique.
    /// Note that 'initialiseDone' clears the current table

    void setPrintMode(bool z, int _print_all_secs);         /// to set the 'recording mode' to ON/OFF, and the frequency of recording
    TableCourse getCinetique();                             /// then, each time initialiseDone() is called, the kinetics is cleared, and
    vector<double> getInit();
    double getT();

/// 4c - to get the value of a particular variable at a particular time point during a simulation,
```

```
/// then, the best way is to create an optimizer* E, give him the data points you want, and do the simulation with simulate(int _sec_max, E);
/// see the class Optimizer for that.

/// 4d to print most informations about the model

    void print();

/// 5 - Accessing variable's values (from an external name), and defining backgrounds
///     even if the model has its internal variable names, you might want to say that this vairable represents this cytokine,
///     in this case, a model can attribute an 'external' name to each variable
///     additionally, a model can declare the list of backgrounds it is able to simulate.
///     in that way, it is possible to interrogate what a model is able to simulate (variables, background).

///     vectors that can be filled by the subclass :
protected:
    vector<int> extNames;           /// for each variable, the model can give a global ID
    vector<int> backSimulated;      /// the model can list in here the IDs of backgrounds it is able to simulate

///     Then, the values of variables can be accessed with the 'external ID' with the following functions
///     these already implemented functions can be overrided (for instance if modifying a variable needs to be done in a particular way)
public:
    virtual void   setValue(int idGlobalVariable, double val);      /// to modify the value of a variable from the global ID of it
    virtual double getValue(int idGlobalVariable);                  /// to get the value of a variable from its global ID
    virtual bool   isSimuBack(int idGlobalBack);                    /// to know if a background can be simulated by the model
    virtual bool   isSimuVar(int idGlobalVariable);                 /// to know if a variable can be simulated by the model (from its global ID)
    virtual int    internValName(int idGlobalVariable);             /// to know the index of a variable inside the tables

/// 7 - Working directly with the parameters of a model (important for managing the interface with the optimizer (fitting class))

    int paramSize();
    void setParam(int i, double v);
    double getParam(int i);

/// 8 - to get information on the (internal) variables

    int getNbVars();
    string getName(int internalID);
    int getGlobalID(int internalID);


...
}
```

Example of additional things a model can do :

- provide 'external names' for its variables, i.e. whatever the variables are called inside, giving the 'official' name for certain variables, in the idea that one can ask the model the value of a variable with its external name without taking care which index it is inside the class. ex in the code below : extNames[IL12] = N::IL12
- give the list of backgrounds it is able to simulate, ex : backSimulated.push_back(Back::WT);
- re-implement the function to load parameters from a text file virtual void *loadParameters*(string file_name);
- have a variable 'background' which will allow to perform simulations on different contexts (deficient mice ...), and use the parameter of the initialize function to pass the background of the next simulation

Example of a 'full' model, with different backgrounds and external names/IDs for variables

```
namespace N
    {enum {IL12, STAT4P, TBET, IFNG, R12, RL12, STAT4, RIFN, IFNGR, STAT1, STAT1P, SOCS1, NB_GLOB_VARS};}
namespace Back
    {enum {WT, IFNGRKO, STAT4KO, TBETKO, NB_GLOB_BACK};}

struct Modele110L : public Modele {
    Modele110L();
    enum  {IL12, STAT4P, TBET, NBVAR};
    enum {KD12, KDS4P, KDTBET, S4PEQ, CS4P, TBETEQ, CTBET, CTCR, BS4P, BTBET, NBPARAM};
    int background;
    void initialise(int _background = Back::WT);
    void derivatives(const vector<double> &x, vector<double> &dxdt, const double t);
    void setBaseParameters();
};

Modele110L::Modele110L() : Modele(NBVAR, NBPARAM), background(Back::WT) {
    dt = 0.2;

    names[IL12] = string("IL12");
    names[STAT4P] = string("STAT4P");
    names[TBET] = string("TBET");

    extNames[IL12] = N::IL12;
    extNames[STAT4P] = N::STAT4P;
    extNames[TBET] = N::TBET;

    backSimulated.clear();
    backSimulated.push_back(Back::WT);
    backSimulated.push_back(Back::STAT4KO);
    backSimulated.push_back(Back::TBETKO);
}

void Modele110L::setBaseParameters(){
    params.clear();
    params.resize(NBPARAM);
    params[KD12] = 1.1e-4;
    params[KDS4P] = 2e-5;
    params[KDTBET] = 1e-5;
    params[S4PEQ] = 0.05;
    params[CS4P] = 1.0;
    params[TBETEQ] = 0.05;
    params[CTBET] = 1.0;
    params[CTCR] = 1.0;
    // params[BS4P] and params[BTBET] will be defined in initialize
    setBaseParametersDone();
}

void Modele110L::initialise(int _background){
    background = _background;
    val.clear();        val.resize(NBVAR, 0.0);
    init.clear();       init.resize(NBVAR, 0.0);

    // parameters that depend on other ones are better here in case one wants to use loadParameters
    params[BS4P]  = params[KDS4P] * params[S4PEQ];
    params[BTBET] = params[KDTBET] * params[TBETEQ] - params[CTBET] * params[S4PEQ] * params[TBETEQ] * (1 +
           params[CTCR]);
    if(params[BS4P]  < 0) params[BS4P] = 0;
    if(params[BTBET] < 0) params[BTBET] = 0;

    init[IL12]   = 0.0;
    init[TBET]   = params[TBETEQ];
    init[STAT4P] = params[S4PEQ];

    if(background == Back::STAT4KO) init[STAT4P] = 0.0;
    if(background == Back::TBETKO) init[TBET] = 0.0;

    for(int i = 0; i < NBVAR; ++i){val[i] = init[i];}
    t = 0;
}

void Modele110L::derivatives(const vector<double> &x, vector<double> &dxdt, const double t){
    dxdt[IL12] = - params[KD12] * x[IL12];
    dxdt[STAT4P] = - params[KDS4P] * x[STAT4P] + params[BS4P] + params[CS4P] * x[IL12] * x[STAT4P];
    dxdt[TBET] = - params[KDTBET] * x[TBET] + params[BTBET] + params[CTBET] * x[STAT4P] * x[TBET] * (1 +
           params[CTCR]);
}
```

How to use the model (see below for using kinetics and evaluators)

```
    Modele* M = new Modele110L();
    M->setBaseParameters();
    M->initialise(Back::WT);
    M->simulate(20);
    if(M->penalities > 0) cout << "diverged !" << "(t=" << M->getT() << ") penalty = " << M->penalities << endl;

    M->initialise(Back::WT);
    M->setValue(N::IL12, 0.01);
    M->setPrintMode(true, 20);
    M->simulate(10*3600);
    if(M->penalities > 0) cout << "diverged !" << "(t=" << M->getT() << ") penalty = " << M->penalities << endl;
    TableCourse T = M->getCinetique();
    T.print();

    cout << "Variables : ";
    for(int i = 0; i < M->getNbVars(); ++i){
        cout << M->getName(i) << "\t";
    }

    cout << "the value of IL12 at the end is " << M->getValue(N::IL12) << endl;
    cout << "Testing if backgrounds or variables are possible to simulate : ";
    if(M->isSimuBack(Back::WT)) cout << "Back::WT is simulated" << endl;
    else cout << "Back::WT is NOT simulated" << endl;
    if(M->isSimuBack(Back::IFNGRKO)) cout << "Back::IFNGRKO is simulated" << endl;
    else  cout << "Back::IFNGRKO is NOT simulated" << endl;
    if(M->isSimuBack(N::SOCS1)) cout << "N::SOCS1 is simulated" << endl;
    else  cout << "N::SOCS1 is NOT simulated" << endl;
```

A 'TableCourse' is a table with nbVar variables, and one line (vector) per time-point

```
struct TableCourse {
    vector<double> attribut;            // List of time points
    vector<string> headers;             // Names of the variables (columns)
    vector< vector<double> * > storage; // data (2D) storage[i][j] = value at time point i (t = attribut[i])
                                        // of variable j (whose name is headers[j+1])
    int nbVar;
    int nbLignes;                       // nb of time points

    TableCourse(int _nbVar);
    void reset();
    void setHeader(int i, string title);// starts at i=1 for variables. Header[0] = titre of attr. column
    void addSet(double attr, vector<double> &toCopy);

    void print();
    void save(string fileName, string title = string(""));
    vector<double> getTimeCourse(int var); // var = 0..nbVars-1
    vector<double> getTimePoints();
};
```

Example of use (alone) :

```
    TableCourse A(3);
    A.setHeader(0, string("Time"));
    A.setHeader(1, string("VarA"));
    A.setHeader(2, string("VarB"));
    A.setHeader(3, string("VarC"));
    vector<double> t1 = vector<double>(3, 0.0); // example of data for t1 at t=10sec
    t1[0] = 1.0; t1[1] = 0.5; t1[2] = 0.3;
    vector<double> t2 = vector<double>(3, 0.0); // example of data for t2 at t=20sec
    t2[0] = 0.8; t2[1] = 0.5; t2[2] = 0.4;
    A.addSet(10, t1);
    A.addSet(20, t2);

    A.print();
    A.save("EssaiTableCourse.txt", "Let's see if iut works");
    cerr << "Time course of variable B : ";
    vector<double> courseOfVarB = A.getTimeCourse(1);
    vector<double> timePoints = A.getTimePoints();
    if(courseOfVarB.size() != timePoints.size()) cerr << "ERR : the number of time points and the length
                                                          of a time course should be the same !!\n";
    for(int i = 0; i < courseOfVarB.size(); ++i){
        cerr << "t=" << timePoints[i] << ", B=" << courseOfVarB[i] << "\t" << endl;
    }
```

Example of use in the context of a model :

```
    Modele* M = new ModeleA1();
    M->setBaseParameters();
    M->initialise();
    M->print();
    M->simulate(100);
    M->initialise();
    M->setPrintMode(true, 1000);
    M->simulate(10000);
    TableCourse T = M->getCinetique();
    T.print();
    if(M->penalities > 0)
           cout << "The simulation diverged and was stopped ; penalty = " << M->penalities << endl;
```

Note that it can be hard to find good parameters, and most of the case with arbitrary parameters, it diverges ...

3 - Working with particular time points

To get the values of a simulation at particular time points for some variables, but without saving all the kinetics of a simulation, the Evaluator Class is used to make a 'wish list' of data points. Then, during the simulation of a model, the evaluator is automatically filled with the wished data.

Use : - Create an Evaluator,
- give the couples of (Variable, time point) to record, calling *getVal(variable ID, timePoint)*
- call *recordingCompleted(),*

For using 'manually', (the Modele::simulate function does it automatically during a simulation)
- for each time point, you can ask the evaluator if he needs data for this time point, by calling takeDataAtThisTime(i). If yes, speciesToUpdate gives the ID of the next variable.
- You give the value by setValNow(species, value), and call again speciesToUpdate, until it returns -1 meaning the evaluator doesn't need anymore data from this time point.
- Finally, your stored data can be retrieved by calling getVal(species, time point)

Note that, on purpose, the same function is used to give the 'wish list' before recordingCompleted is called, and for getting the recorded values. It allows to test what a function needs before a simulation and calling this function again after a simulation but with the simulated values.

Example :

```
    Evaluator E = Evaluator();
    E.getVal(1,15);
    E.getVal(3,20);
    E.getVal(1,20);
    E.getVal(5,50);

    E.recordingCompleted();
    //E.printState();
    for(int i = 0; i < 100; ++i){
        if(E.takeDataAtThisTime(i)){
            int z = E.speciesToUpdate();
            while(z >= 0){
                E.setValNow(z, (double) (i + z*100));
                z = E.speciesToUpdate();
            }
            //E.printState();
        }
    }

    cerr << "\n";
    cerr << E.getVal(1,15) << "\t";
    cerr << E.getVal(3,20) << "\t";
    cerr << E.getVal(1,20) << "\t";
    cerr << E.getVal(5,50) << "\n";
    E.printState();
```

Result :

```
115 320 120 550
Eva : 4 pts, nextTime=100000000
Sp=1 t=15, val=115, rec=Oui
Sp=3 t=20, val=320, rec=Oui
Sp=1 t=20, val=120, rec=Oui
Sp=5 t=50, val=550, rec=Oui
```

If used in the context of a model : (see the code for Modele110L in the model section)

```
    Modele* currentModel = new Modele110L();
    currentModel->setBaseParameters();
    currentModel->initialise(Back::WT);
    currentModel->loadParameters(
       string("C:/Users/parobert/Desktop/Optimisation/ThModeles/Modeles/BestSetM110.txt"));
    currentModel->print();
    Evaluator* E = new Evaluator();
    cout << "Before\n";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 10) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 20) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 30) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 40) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 10) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 20) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 30) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 40) << "\t";
    E->recordingCompleted();
    currentModel->setPrintMode(true, 10);
    currentModel->simulate(120, E);
    TableCourse T = currentModel->getCinetique();
    T.print();
    cout << "After\n";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 10) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 20) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 30) << "\t";
    cout << E->getVal(currentModel->internValName(N::STAT4P), 40) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 10) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 20) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 30) << "\t";
    cout << E->getVal(currentModel->internValName(N::TBET), 40) << "\t";
    cout << endl;
```