
CompuCell3D Developers Manual Documentation

Release 1.0.0

Maciek Swat, James A. Glazier

Apr 24, 2022

Introduction To CompuCell3D Core Objects

1	Funding	3
2	Setting up Windows computer for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++	5
3	Setting up Linux computer for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++	7
4	Setting up your Mac for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++	9
5	Setting up your Mac for CC3D compilation via conda	11
6	Configuring DeveloperZone Projects for compilation	13
7	Introduction to Core CC3D Objects	17
8	Building Steppable	33
9	Building Growth Steppable In the Developer Zone folder	49
10	Adding Python Bindings To Steppable in DeveloperZone	61
11	Computing Heterotypic Boundary Length	69
12	Attaching Custom Attributes To Cells	81
13	Debugging CC3D using GDB	97
14	Indices and tables	101

The focus of this manual is to explain internals of C++ code and provide all information you need start writing your own C++ extension modules for CompuCell3D

introduction

CHAPTER 1

Funding

From early days CompuCell3D was funded by science grants. The list of funding entities include

- National Institutes of Health (NIH)
- US Environmental Protection Agency (EPA)
- National Science Foundation (NSF)
- Falk Foundation
- Indiana University (IU)
- IBM

The development of CC3D was funded fully or partially by the following awards:

- National Institutes of Health, National Institute of Biomedical Imaging and Bioengineering, U24 EB028887, “Dissemination of libRoadRunner and CompuCell3D”, (09/30/2019 – 06/30/2024)
- National Science Foundation, NSF 1720625, “Network for Computational Nanotechnology - Engineered nanoBIO Node”, (09/1/2017-08/31/2022)
- National Institutes of Health, National Institute of General Medical Sciences, R01 GM122424, “Competitive Renewal of Development and Improvement of the Tissue Simulation Toolkit”, (02/01/2017 - 01/31/2021)
- Falk Medical Research Trust Catalyst Program, Falk 44-38-12, “Integrated in vitro/in silico drug screening for ADPKD”, (11/30/2014-11/29/2017)
- National Institutes of Health, National Institute of General Medical Sciences, National Institute of Environmental Health Sciences and National Institute of Biomedical Imaging and Bioengineering, U01 GM111243 “Development of a Multiscale Mechanistic Simulation of Acetaminophen-Induced Liver Toxicity”, (9/25/14-6/30/19)
- National Institutes of Health, National Institute of General Medical Sciences, R01 GM076692, “Competitive Renewal of MSM: Multiscale Studies of Segmentation in Vertebrates”, (9/1/05-8/31/18)
- U. S. Environmental Protection Agency, R835001, “Ontologies for Data & Models for Liver Toxicology”, (6/1/11-5/30/15)

- National Institutes of Health, National Institute of General Medical Sciences, R01 GM077138, “Competitive Renewal of Development and Improvement of the Tissue Simulation Toolkit”, (9/1/07-3/31/15).
- U. S. Environmental Protection Agency, National Center for Environmental Research, R834289, “The Texas-Indiana Virtual STAR Center: Data-Generating in vitro and in silico Models of Development in Embryonic Stem Cells and Zebrafish”, (11/1/09-10/31/13)
- Pervasive Technologies Laboratories Fellowship (Indiana University Bloomington) (12/1/03-11/30/04).
- IBM Innovation Institute Award (9/25/03-9/24/06)
- National Science Foundation, Division of Integrative Biology, IBN-0083653, “BIOCOMPLEXITY–Multiscale Simulation of Avian Limb Development”, (9/1/00-8/31/07)

Setting up Windows computer for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++

If you want to develop plugins and steppables under windows you will need to install free Visual Studio 2015 Community Version. The installation of this package is straightforward but you need to make sure that you are installing C/C++ compilers when the installer gives you options to select which programming languages you would like to have support for. The best way to download Visual Studio is to get it directly from Microsoft website. Current link to visual studio download page is here: <https://visualstudio.microsoft.com/vs/older-downloads/> . Just make sure you scroll down and find Visual Studio 2015. It has to be exactly this version. CompuCell3D compilation will likely not work with other versions. Once you download and install Visual Studio 2015 you are ready to start compiling Developer Zone projects and develop your own CompuCell3D plugins and steppables in C++.

Setting up Linux computer for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++

If you are using linux computer , most likely you do not need to do any compiler setup. CC3D on Linux comes prepackaged with all compilers and it does not matter if you installed CC3D using automated installer or installed directly using `conda install` command.

CHAPTER 4

Setting up your Mac for DeveloperZone compilation and developing new CC3D plugins and Steppables in C++

Starting with CC3D 4.3.0 when you install CC3D it will come with most of the tools needed to compile C++ plugins and steppables. The only thing that you need in addition to this is to install `xcode-select` package To install this from the terminal run the following:

```
xcode-select --install
```

This is it and you should be ready to compile custom plugins and steppables written in C++

Setting up your Mac for CC3D compilation via conda

Sometimes you may want to compile entire CC3D C++ code using conda build system. In general when compiling code via conda-build system you do not need to install anything - besides making sure that your conda-build system works properly. To ensure that conda build system works properly from your base conda environment (it is important that this is base environment or else things may not work properly) run

```
conda install conda-build
```

This will install all utilities you need to build CC3D. Tools like swig, cmake, compilers etc will be downloaded and installed automatically just in time for compilation. We will only mention that Current version of CC3D uses clang compilers version 12. On Linux we use gcc compilers and on Windows Visual Studio 2015 Community Version (free)

To compile CC3D on your Mac using conda-build system follow this procedure:

1. Install `xcode-select` - see above
2. install miniconda3 with Python 3.7 - https://repo.anaconda.com/miniconda/Miniconda3-py37_4.11.0-MacOSX-x86_64.sh . Once you install miniconda in the base environment of newly installed miniconda install conda-build package

```
conda install conda-build
```

3. Get MacOS SDK 10.10 - <https://github.com/phracker/MacOSX-SDKs> or directly from <https://github.com/phracker/MacOSX-SDKs/releases>. Here is direct link to the actual compressed folder: <https://github.com/phracker/MacOSX-SDKs/releases/download/11.3/MacOSX10.10.sdk.tar.xz> Once you unpack move the content to `/opt` folder of your Mac. You need to be admin to do this. You should be able to see the following folder `/opt/MacOSX10.10.sdk` after the copy is complete

4. Clone CC3D repository

```
git clone https://github.com/CompuCell3D/CompuCell3D.git
```

5. Go to CC3D repository's conda-recipes folder:

```
cd <CC3D repository dir>/conda=recipe
```

6. Start compilation by typing

```
conda build . -c conda-forge -c compucell3d
```

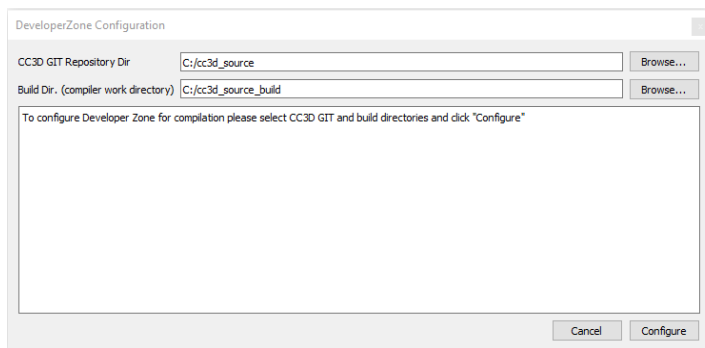
After a while you should have CC3D conda package ready

Configuring DeveloperZone Projects for compilation

From technical viewpoint `DeveloperZone` is a folder that contains source code for additional plugins and steppables written in C++. Depending on your needs, sometimes, you want to write high-performance CC3D module that runs much faster than equivalent Python code. Up until version 4.3.0 of CC3D developing C++ modules was a little bit involved because it required users to install and configure appropriate compilers that will work with provided binaries, performing CMake configuration - the challenge here was to make sure that all Cmake variables point to appropriate directories, that Python version identified by Cmake matches the one with which CC#D was compiled etc... In practice this process was often perceived as quite error-prone.

Starting with version 4.3.0 of CC3D we provide one-click configurator for “DeveloperZone” All that is required from the user is one time setup of compiler (on Windows you will install Visual Studio 2015 Community Edition, and on Mac you need to install xcode-select package - all described in sections above. On linux you will likely not need any additional setup).

Once you set up compilers (and install binaries for CC3D) open Twedit++ and go to CC3D C++ -> DeveloperZone This will open the following dialog:



Before going any further, make sure you have a working copy of the CC3D source code. The best way is to clone CC3D source code repository. If you have git installed on your system you are ready to go. If not you can easily do it by running `conda-shell` script from CC3D installation folder. Assuming your CC3D is installed into `c:\CompuCell3D` (on Windows) you would run the following:

```
conda install -c conda-forge git
```

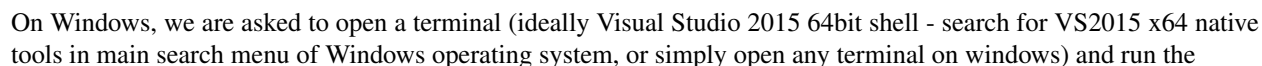
Let's clone CC3D source code now. In the terminal where you previously ran `conda-shell.bat`, do the following

```
cd c:\cc3d_source
git clone https://github.com/CompuCell3D/CompuCell3D.git .
```

Now let's make build directory. This is a directory where compilers will place temporary compilation objects:

```
cd c:\
mkdir cc3d_source_build
```

Now, fill in full path to CC3D repository (c:\cc3d_source) and to build folder (c:\cc3d_source_build) and click **Configure** button in the bottom right corner of the dialog. Configuration process will start. After it is done the dialog should display summary of what to do next:



```
c:\CompuCell3D\conda-shell.ba
```

Then you run the following:

```
cd C:\cc3d_source_build
```

Chapter 6. Configuring DeveloperZone Projects for compilation

(continued from previous page)

```
nmake
nmake install
```

First command changes directory to the directory that we designated for storing temporary compilation files. This is where the Makefile were generated to. Next, we run windows version of make called nmake.

```
(base) m@m-LENOVO C:\Users\m
> cd c:\cc3d_source_build

(base) m@m-LENOVO c:\cc3d_source_build
> nmake
```

Once compilation finishes

```
c:\cc3d_source\compuCell3d\core\compuCell3d\PluginManager.h(108): warning C4251: 'CompuCell3D::PluginManager<CompuCell3D::Step
pable>::libExtension': class 'std::basic_string<char,std::char_traits<char>,std::allocator<char>>' needs to have dll-interface
to be used by clients of class 'CompuCell3D::PluginManager<CompuCell3D::Stepable>'
c:\cc3d_source\compuCell3d\core\compuCell3d\Field3D\Field3DImpl.h(70): warning C4018: '<': signed/unsigned mismatch
c:\cc3d_source\compuCell3d\core\compuCell3d\Field3D\Field3DImpl.h(58): note: while compiling class template member function 'C
ompuCell3D::Field3DImpl<float>::Field3DImpl(const CompuCell3D::Dim3D,const T &)'
    with
    [
        T=float
    ]
c:\cc3d_source\CompuCell3D\core\CompuCell3D\Field3D\Array3D.h(133): note: see reference to function template instantiation 'Co
mpuCell3D::Field3DImpl<float>::Field3DImpl(const CompuCell3D::Dim3D,const T &)' being compiled
    with
    [
        T=float
    ]
c:\cc3d_source\CompuCell3D\core\CompuCell3D\Field3D\Array3D.h(129): note: see reference to class template instantiation 'Compu
Cell3D::Field3DImpl<float>' being compiled
[100%] Linking CXX shared module _CompuCellExtraModules.pyd
CompuCellExtraModulesPYTHON_wrap.cxx.obj : MSIL .netmodule or module compiled with /GL found; restarting link with /LTCG; add
/LTCG to the link command line to improve linker performance
LINK : warning LNK4075: ignoring '/INCREMENTAL' due to '/LTCG' specification
Creating library CompuCellExtraModules.lib and object CompuCellExtraModules.exp
Generating code
Finished generating code
CompuCellExtraModulesPYTHON_wrap.cxx.obj : MSIL .netmodule or module compiled with /GL found; restarting link with /LTCG; add
/LTCG to the link command line to improve linker performance
LINK : warning LNK4075: ignoring '/INCREMENTAL' due to '/LTCG' specification
Creating library CompuCellExtraModules.lib and object CompuCellExtraModules.exp
Generating code
Finished generating code
[100%] Built target CompuCellExtraModules

(base) m@m-LENOVO c:\cc3d_source_build
>
```

we install the compiled modules.

```
Finished generating code
[100%] Built target CompuCellExtraModules

(base) m@m-LENOVO c:\cc3d_source_build
> nmake install

Microsoft (R) Program Maintenance Utility Version 14.00.23026.0
Copyright (C) Microsoft Corporation. All rights reserved.

[ 16%] Built target CustomCellAttributeStepableShared
[ 33%] Built target HeterotypicBoundaryLengthShared
[ 50%] Built target GrowthStepableShared
[ 66%] Built target SimpleVolumeShared
[ 83%] Built target VolumeMeanShared
[ 88%] Built target CompuCellExtraModules_swig_compilation
[100%] Built target CompuCellExtraModules
Install the project...
-- Install configuration: "RelWithDebInfo"
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DCustomCellAttributeStepable.l
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DCustomCellAttributeStepable.d
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DHeterotypicBoundaryLength.lib
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DHeterotypicBoundaryLength.dll
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DGrowthStepable.lib
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DGrowthStepable.dll
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DPlugins/CC3DSimpleVolume.lib
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DPlugins/CC3DSimpleVolume.dll
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DVolumeMean.lib
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCell3DSteppables/CC3DVolumeMean.dll
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/_CompuCellExtraModules.pyd
-- Installing: c:/CompuCell3D/Miniconda3/Lib/site-packages/cc3d/cpp/CompuCellExtraModules.py

(base) m@m-LENOVO c:\cc3d_source_build
>
```

If you look carefully at the output screen you will see that the modules we compiled will get installed into subfolders of `c:\CompuCell3D\Miniconda3` which is Miniconda distribution that is part of CC3D installation. This is exactly what we want. In other words, with one click and few simple command line commands we were able to compile a

set of demo extensions modules written in C++. This is significant because this simple procedure allows you to easily add new C++ modules and significantly speedup your simulation. From now on you can focus on coding rather than figuring out of how to set up compilation

Introduction to Core CC3D Objects

- simulator
- potts

7.1 Simulator

Simulator is the key C++ module that sits at the root of each simulation run by CC3D. This is essentially a single class `Simulator` and it is responsible for orchestrating the flow of each CC3D simulation. Simulator object creates and manages other key objects such as `Potts3D` and ensures the integrity of the entire simulation. The code for the object is stored in `CompuCell13D\Simulator.h` and `CompuCell13D\Simulator.cpp`

Let us look at the header file of the `Simulator` to examine the responsibilities that `Simulator` when running CC3D simulations

```
namespace CompuCell13D {  
    class ClassRegistry;  
    class BoundaryStrategy;  
  
    template <typename Y> class Field3DImpl;  
    class Serializer;  
    class PottsParseData;  
    class ParallelUtilsOpenMP;  
  
    class COMPUCELLLIB_EXPORT Simulator : public Steppable {  
  
        ClassRegistry *classRegistry;  
  
        Potts3D potts;  
  
        int currstep;  
  
        bool simulatorIsStepping;  
    }  
}
```

(continues on next page)

(continued from previous page)

```

    bool readPottsSectionFromXML;
    std::map<std::string,Field3D<float>*> concentrationFieldNameMap;
    //map of steerable objects
    std::map<std::string,SteerableObject *> steerableObjectMap;

    std::vector<Serializer*> serializerVec;
    std::string recentErrorMessage;
    bool newPlayerFlag;

    std::streambuf * cerrStreamBufOrig;
    std::streambuf * coutStreamBufOrig;
    CustomStreamBufferBase * qStreamBufPtr;

    std::string basePath;
    bool restartEnabled;

public:

    ParserStorage ps;
    PottsParseData * ppdCC3DPtr;
    PottsParseData ppd;
    PottsParseData *ppdPtr;
    ParallelUtilsOpenMP *pUtils;
    ParallelUtilsOpenMP *pUtilsSingle; // stores same information as pUtils but
    ↪ assumes that we use only single CPU - used in modules for which user requests
    ↪ single CPU runs e.g. Potts with large cells

    double simValue;

    void setOutputRedirectionTarget(ptrdiff_t _ptr);
    ptrdiff_t getCerrStreamBufOrig();
    void restoreCerrStreamBufOrig(ptrdiff_t _ptr);

    void setRestartEnabled(bool _restartEnabled){restartEnabled=_restartEnabled;}
    bool getRestartEnabled(){return restartEnabled;}

    static PluginManager<Plugin> pluginManager;
    static PluginManager<Steppable> steppableManager;
    static BasicPluginManager<PluginBase> pluginBaseManager;
    Simulator();
    virtual ~Simulator();
    // PluginManager::plugins_t & getPluginMap(){return pluginManager.
    ↪ getPluginMap();}

    //Error handling functions
    std::string getRecentErrorMessage(){return recentErrorMessage;}
    void setNewPlayerFlag(bool _flag){newPlayerFlag=_flag;}
    bool getNewPlayerFlag(){return newPlayerFlag;}

    std::string getBasePath(){return basePath;}
    void setBasePath(std::string _bp){basePath=_bp;}

    ParallelUtilsOpenMP * getParallelUtils(){return pUtils;}
    ParallelUtilsOpenMP * getParallelUtilsSingleThread(){return pUtilsSingle;}

    BoundaryStrategy * getBoundaryStrategy();
    void registerSteerableObject(SteerableObject *);

```

(continues on next page)

(continued from previous page)

```

void unregisterSteerableObject(const std::string & );
SteerableObject * getSteerableObject(const std::string & _objectName);

void setNumSteps(unsigned int _numSteps) {ppdCC3DPtr->numSteps=_numSteps;}
unsigned int getNumSteps() {return ppdCC3DPtr->numSteps;}
int getStep() {return currstep;}
void setStep(int currstep) { this->currstep = currstep; }
bool isStepping(){return simulatorIsStepping;}
double getFlip2DimRatio(){return ppdCC3DPtr->flip2DimRatio;}
Potts3D *getPotts() {return &potts;}
Simulator *getSimulatorPtr(){return this;}
ClassRegistry *getClassRegistry() {return classRegistry;}

void registerConcentrationField(std::string _name,Field3D<float>* _fieldPtr);
std::map<std::string,Field3D<float>*> & getConcentrationFieldNameMap(){
    return concentrationFieldNameMap;
}
void postEvent(CC3DEvent & _ev);

std::vector<std::string> getConcentrationFieldNameVector();
Field3D<float>* getConcentrationFieldByName(std::string _fieldName);

void registerSerializer(Serializer * _serializerPtr){serializerVec.push_back(_
↪serializerPtr);}
virtual void serialize();

// Begin Steppable interface
virtual void start();
virtual void extraInit();///initialize plugins after all steppables have been
↪initialized
virtual void step(const unsigned int currentStep);
virtual void finish();
// End Steppable interface

//these two functions are necessary to implement proper cleanup after the
↪simulation
//1. First it cleans cell inventory, deallocating all dynamic attributes -
↪this has to be done before unloading modules
//2. It unloads dynamic CC3D modules - plugins and steppables
void cleanAfterSimulation();
//unloads all the plugins - plugin destructors are called
void unloadModules();

void initializePottsCC3D(CC3DXMLElement * _xmlData);
void processMetadataCC3D(CC3DXMLElement * _xmlData);

void initializeCC3D();
void setPottsParseData(PottsParseData * _ppdPtr){ppdPtr=_ppdPtr;}
CC3DXMLElement * getCC3DModuleData(std::string _moduleType,std::string _
↪moduleName="");
void updateCC3DModule(CC3DXMLElement *_element);
void steer();

};
};

```

Few things to notice:

1. All CompuCell3D classes are defined within CompuCell3D namespace:

```
namespace CompuCell3D {  
    class ClassRegistry;  
    ...  
    class COMPUCELLLIB_EXPORT Simulator : public Steppable {  
        ...  
    };  
};
```

2. Most CC3D objects are dynamically loaded. To make sure an object can be dynamically loaded on Windows we need to include `__declspec(dllexport)` and `__declspec(dllimport)` class decorators as introduced and required by Microsoft Visual Studio Compilers. Therefore the C++ macro you see above `-COMPUCELLLIB_EXPORT` contains required decorators on Windows and is an empty string on all other operating systems. You can find the details of the Microsoft decorators here:

- <https://stackoverflow.com/questions/14980649/macro-for-dllexport-dllimport-switch>

3. Simulator contains Potts3D object :

```
namespace CompuCell3D {  
    class ClassRegistry;  
    ...  
    class COMPUCELLLIB_EXPORT Simulator : public Steppable {  
        ClassRegistry *classRegistry;  
  
        Potts3D potts;  
  
        ...  
    };  
};
```

4. Simulator has dictionary of every concentration field used in the simulation

```
std::map<std::string,Field3D<float>*> concentrationFieldNameMap;
```

Those fields can be accessed by external code (e.g. Plugin or Steppable code) by using the following Simulator methods:

```
std::vector<std::string> getConcentrationFieldNameVector();  
Field3D<float>* getConcentrationFieldByName(std::string _fieldName);
```

where `getConcentrationFieldNameVector()` retrieves a vector of names of the fields used in the simulation and `Field3D<float>* getConcentrationFieldByName(std::string _fieldName)` returns a pointer to a field

5. Functions/class members related to streams e.g. `std::streambuf * cerrStreamBufOrig;` are related to redirecting output to either console or to a GUI. We will not discuss them here

6. Core simulator functionality, as far as the flow of the simulation is concerned, is implemented in the following functions:

```
void initializeCC3D();  
virtual void start();  
virtual void extraInit();///initialize plugins after all steppables have been  
↪initialized  
virtual void step(const unsigned int currentStep);  
virtual void finish();
```


- `void initializeCC3D()` initializes Potts3D object based on the CC3DML content, as well as loadable modules such as

Plugins and Steppables and it is the first Simulator function that is called after parsing of the CC3DML is complete

- `void extraInit()` is typically executed next and it calls `extraInit` method that is a member of every CompuCell3D

plugin. Think of this function as a way of performing a second round of initialization but in the situation where all necessary objects (plugins) are instantiated and properly located inside overseeing objects (Simulator / Potts3D)

- `void start()` function calls `start` method for all Steppables that were requested by current simulation.
- `void step(const unsigned int currentStep)` method executes a single Monte Carlo Step (MCS) by calling

`metropolis` method from Potts3D;

```
int flips = potts.metropolis(flipAttempts, ppdCC3DPtr->temperature);
```

and it also calls `step` method of every steppable requested by the simulation (including PDE solvers) by calling `step` method of a `classRegistry` member of the Simulator object. You may think about `classRegistry` as of a container that stores pointers to Steppable objects. Indeed, if we look at the `CompuCell3D\ClassRegistry.h` declarations we notice that `ClassRegistry` class is a collection of containers with extra functionality that simplify code calls from parent objects (*e.g.* from Simulator):

```
namespace CompuCell3D {
    class Simulator;

    class COMPUCELLLIB_EXPORT ClassRegistry : public Steppable {
        BasicClassRegistry<Steppable> steppableRegistry;

        typedef std::list<Steppable *> ActiveSteppers_t;
        ActiveSteppers_t activeSteppers;

        typedef std::map<std::string, Steppable *> ActiveSteppersMap_t;
        ActiveSteppersMap_t activeSteppersMap;

        Simulator *simulator;

        std::vector<ParseData *> steppableParseDataVector;

    public:
        ClassRegistry(Simulator *simulator);
        virtual ~ClassRegistry() {}

        Steppable *getStepper(std::string id);

        void addStepper(std::string _type, Steppable *_steppable);

        // Begin Steppable interface
        virtual void extraInit(Simulator *simulator);
        virtual void start();
        virtual void step(const unsigned int currentStep);
        virtual void finish();
        // End Steppable interface

        virtual void initModules(Simulator *_sim);
    };
};
```

- Finally the `void finish()` method is responsible finishing the simulation. This seemingly simple task involves

few critical steps: running few Monte Carlo Steps (of metropolis algorithm) with zero temperature - users specify number of those steps in the CC3DML code (in `<Anneal>` element), calling `finish` function of every steppable, unloading dynamically loaded modules (Plugins and Steppables) to ensure that subsequent simulations can run without restarting CC3D.

There are clearly more methods in the Simulator objects but the ones described perform most of the work.

7.2 Potts3D

Potts3D module (`Potts3D/Potts3D.cpp`, `Potts3D/Potts3D.h`) implements entire logic of the Potts algorithm. Moreover, this module is responsible for creating cell lattice and `Potts3D` class has methods that facilitate creation and destruction of cells. It is worth pointing out that creation and destruction of cells is not limited to calling `new` or `delete` operators but it also involves several steps necessary to ensure that cells created have all the attributes needed by requested by the user plugins. In CC3D cells' attributes are added dynamically and CC3D cells by default have only a small subset of attributes hard-coded. This is a design decision that has this nice consequence that when developing new plugin one does not have to modify `CellG` class but rather program the addition of cell's attributes entirely in the plugins code. We will cover this in detail in later section.

Let's examine the content of the `Potts3D` class (**Note:** we removed some of the code and are presenting only code snippets most relevant to current discussion. You are encouraged to look at the original code though as you go over the material presented here):

```
class Potts3D :public SteerableObject {
    WatchableField3D<CellG *> *cellFieldG;
    AttributeAdder * attrAdder;
    EnergyFunctionCalculator * energyCalculator;

    BasicClassGroupFactory cellFactoryGroup;           //creates aggregate of
    ↪objects associated with cell

    /// An array of energy functions to be evaluated to determine energy costs.
    std::vector<EnergyFunction *> energyFunctions;
    EnergyFunction * connectivityConstraint;

    std::map<std::string, EnergyFunction *> nameToEnergyFuctionMap;

    ...

    std::vector<BasicRandomNumberGeneratorNonStatic> randNSVec;

    /// An array of potts steppers. These are called after each potts step.
    std::vector<Stepper *> steppers;

    std::vector<FixedStepper *> fixedSteppers;
    /// The automaton to use. Assuming one automaton per simulation.
    Automaton* automaton;

    ...

    FluctuationAmplitudeFunction * fluctAmplFcn;

    /// The current total energy of the system.
```

(continues on next page)

(continued from previous page)

```

    double energy;

    std::string boundary_x; // boundary condition for x axis
    std::string boundary_y; // boundary condition for y axis
    std::string boundary_z; // boundary condition for z axis

    /// This object keeps track of all cells available in the simulations. It
    ↪ allows for simple iteration over all the cells
    /// It becomes useful whenever one has to visit all the cells. Without
    ↪ inventory one would need to go pixel-by-pixel - very inefficient
    CellInventory cellInventory;

    Point3D flipNeighbor;
    std::vector<Point3D> flipNeighborVec; //for parallel access

    double depth;
    //int maxNeighborOrder;
    std::vector<Point3D> neighbors;
    std::vector<unsigned char> frozenTypeVec; ///lists types which will remain
    ↪ frozen throughout the simulation
    unsigned int sizeFrozenTypeVec;

    ParallelUtilsOpenMP *pUtils;

public:

    Potts3D();
    Potts3D(const Dim3D dim);
    virtual ~Potts3D();

    void createCellField(const Dim3D dim);
    void resizeCellField(const Dim3D dim, Dim3D shiftVec = Dim3D());

    double getTemperature() const { return temperature; }

    void registerConnectivityConstraint(EnergyFunction * _connectivityConstraint);
    EnergyFunction * getConnectivityConstraint();

    bool checkIfFrozen(unsigned char _type);

...

    void initializeCellTypeMotility(std::vector<CellTypeMotilityData> & _
    ↪ cellTypeMotilityVector);
    void setCellTypeMotilityVec(std::vector<float> & _cellTypeMotilityVec);
    const std::vector<float> & getCellTypeMotilityVec() const { return
    ↪ cellTypeMotilityVec; }

    void setDebugOutputFrequency(unsigned int _freq) { debugOutputFrequency = _
    ↪ freq; }
    void setSimulator(Simulator *_sim) { sim = _sim; }

...

    Point3D getFlipNeighbor();

...

```

(continues on next page)

(continued from previous page)

```

    virtual void createEnergyFunction(std::string _energyFunctionType);
    EnergyFunctionCalculator * getEnergyFunctionCalculator() { return _
    ↪energyCalculator; }

    CellInventory &getCellInventory() { return cellInventory; }

    void clean_cell_field(bool reset_cell_inventory = true);

    virtual void registerAttributeAdder(AttributeAdder * _attrAdder);
    virtual void registerEnergyFunction(EnergyFunction *function);
    virtual void registerEnergyFunctionWithName(EnergyFunction *_function, _
    ↪std::string _functionName);
    virtual void unregisterEnergyFunction(std::string _functionName);

    /// Add the automaton.
    virtual void registerAutomaton(Automaton* autom);

    /// Return the automaton for this simulation.
    virtual Automaton* getAutomaton();
    void setParallelUtils(ParallelUtilsOpenMP *_pUtils) { pUtils = _pUtils; }

    virtual void setFluctuationAmplitudeFunctionByName(std::string _
    ↪fluctuationAmplitudeFunctionName);
    /// Add a cell field update watcher.

    /// registration of the BCG watcher
    virtual void registerCellGChangeWatcher(CellGChangeWatcher *_watcher);

    /// Register accessor to a class with a cellGroupFactory. Accessor will _
    ↪access a class which is a member of a BasicClassGroup
    virtual void registerClassAccessor(BasicClassAccessorBase *_accessor);

    /// Add a potts stepper to be called after each potts step.
    virtual void registerStepper(Stepper *stepper);
    virtual void registerFixedStepper(FixedStepper *fixedStepper, bool _front = _
    ↪false);
    virtual void unregisterFixedStepper(FixedStepper *fixedStepper);

    double getEnergy();

    virtual CellG *createCellG(const Point3D pt, long _clusterId = -1);
    virtual CellG *createCellGSpecifiedIds(const Point3D pt, long _cellId, long _
    ↪clusterId = -1);
    virtual CellG *createCell(long _clusterId = -1);
    virtual CellG *createCellSpecifiedIds(long _cellId, long _clusterId = -1);

    virtual void destroyCellG(CellG * cell, bool _removeFromInventory = true);

    BasicClassGroupFactory * getCellFactoryGroupPtr() { return &cellFactoryGroup; _
    ↪};

    virtual unsigned int getNumCells() { return cellInventory.
    ↪getCellInventorySize(); }

    virtual double changeEnergy(Point3D pt, const CellG *newCell, const CellG _
    ↪*oldCell);

```

(continues on next page)

(continued from previous page)

```

    virtual unsigned int metropolis(const unsigned int steps, const double temp);

    typedef unsigned int (Potts3D::*metropolisFcnPtr_t)(const unsigned int, const
↳double);

    metropolisFcnPtr_t metropolisFcnPtr;

    unsigned int metropolisList(const unsigned int steps, const double temp);

    unsigned int metropolisFast(const unsigned int steps, const double temp);
    unsigned int metropolisBoundaryWalker(const unsigned int steps, const double
↳temp);
    void setMetropolisAlgorithm(std::string _algName);

    virtual Field3D<CellG *> *getCellFieldG() { return (Field3D<CellG *>
↳*)cellFieldG; }
    virtual Field3DImpl<CellG *> *getCellFieldGImpl() { return (Field3DImpl<CellG
↳*> *)cellFieldG; }

    //SteerableObject interface
    virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag = false);
    virtual std::string steerableName();
    virtual void runSteppers();
    long getRecentlyCreatedClusterId() { return recentlyCreatedClusterId; }
    long getRecentlyCreatedCellId() { return recentlyCreatedCellId; }

};

```

Starting from the top of the file we notice that cell lattice (`WatchableField3D<CellG *> *cellFieldG;`) is owned by `Potts3D` and created by (`void createCellField(const Dim3D dim);`, `void resizeCellField(const Dim3D dim, Dim3D shiftVec = Dim3D());`).

The cell lattice is an instance of the `WatchableField3D` class (which strictly speaking is a template class). The cell lattice stores **pointers** to cell objects (type `CellG*`). This means that when a single cell single occupies several lattice sites we create one `CellG` object but store pointer to this object in all locations of `cellFieldG` that are assigned to this particular instance of `CellG` object. This way `CellG` objects do not get repeated for every pixel (this would cost too much memory) but rather are referenced from the cell lattice via pointers. The reason cell lattice field is called “Watchable” is because this class implements the observer design pattern. This means that any manipulation of the cell lattice (e.g. assigning cell to a given pixel) triggers calls to multiple registered observer objects that react to such change. For example, if I am extending a cell by assigning its pointer to the new lattice site one of the observer that will be called (we also refer to them as lattice monitors) is a module that tracks cell volume. The cell that gains new pixel will get its `volume` attribute increased by 1 and the cell that loses one pixel will get its volume decreased by 1. Similarly we could have another observer that updates center of mass coordinates, or one that monitors inertia tensor. The nice thing about using `WatchableField3D` template is that all those observers are called automatically when change in the lattice takes place. Observers are called in the order in which they were registered. Note, this may or may not be the order in which they were declared in the `CC3DCML`. `CC3D` sometimes requires certain lattice monitors to be loaded and registered before others and this happens automatically in the `CC3D` code. Let’s look at how `WatchableField3D` works in practice:

7.2.1 WatchableField3D

```

#ifdef WATCHABLEFIELD3D_H
#define WATCHABLEFIELD3D_H

```

(continues on next page)

(continued from previous page)

```

#include <vector>

#include "Field3DImpl.h"
#include "Field3DChangeWatcher.h"

#include <CompuCell3D/CC3DExceptions.h>

namespace CompuCell3D {

    template<class T>
    class Field3DImpl;

    template<class T>
    class WatchableField3D : public Field3DImpl<T> {
        std::vector<Field3DChangeWatcher<T>*> changeWatchers;

    public:
        /**
         * @param dim The field dimensions
         * @param initialValue The initial value of all data elements in the field.
         */
        WatchableField3D(const Dim3D dim, const T &initialValue) :
            Field3DImpl<T>(dim, initialValue) {}

        virtual ~WatchableField3D() {}

        virtual void addChangeWatcher(Field3DChangeWatcher<T> *watcher) {
            if (!watcher) throw CC3DException("addChangeWatcher() watcher cannot be_
↪NULL!");
            changeWatchers.push_back(watcher);
        }

        virtual void set(const Point3D &pt, const T value) {
            T oldValue = Field3DImpl<T>::get(pt);
            Field3DImpl<T>::set(pt, value);

            for (unsigned int i = 0; i < changeWatchers.size(); i++)
                changeWatchers[i]->field3DChange(pt, value, oldValue);
        }

        virtual void set(const Point3D &pt, const Point3D &addPt, const T value) {
            T oldValue = Field3DImpl<T>::get(pt);
            Field3DImpl<T>::set(pt, value);

            for (unsigned int i = 0; i < changeWatchers.size(); i++) {
                changeWatchers[i]->field3DChange(pt, value, oldValue);
                changeWatchers[i]->field3DChange(pt, addPt, value, oldValue);
            }
        }
    };
};
#endif

```

The `WatchableField3D<T>` template class inherits from `Field3DImpl<T>` template. The actual memory allocation takes place in the `Field3DImpl<T>` but we will not worry about it here. It is sufficient to mention that `Field3DImpl<T>` is the class that manages cell lattice memory. The important thing is to understand how

this automatic calling of lattice monitors is implemented. The `WatchableField3D<T>` class has a container `std::vector<Field3DChangeWatcher<T> *> changeWatchers;` that stores pointers to lattice monitors. The lattice monitor object is a class that inherits `Field3DChangeWatcher<T>` class. In CC3D case `T` is set to `CellG*`. The `BasicArray` is a thin wrapper around `std::vector` class and it is one of the legacies of the early CC3D implementations. So `WatchableField3D<T>` class has a collection of objects that react to the changes in the cell lattice. How do they react? If we look at the implementation of virtual void `set(const Point3D &pt, const T value)` function that modifies the lattice we can see that this function fetches old value stored in the lattice at location indicated by `Point3D pt` - in the case of cell lattice this will be pointers currently stored at this location. It then assigns new value to the field (new `CellG` pointer) and then it calls all registered lattice monitors:

```
for (unsigned int i = 0; i < changeWatchers.getSize(); i++)
    changeWatchers[i]->field3DChange(pt, value, oldValue);
```

In particular each lattice monitor (here referred to as `changeWatcher`) must define function called `field3DChange` that takes 3 arguments - location of the change `pt`, new value we assign to the field (e.g. new pointer to `CellG` object) and old value that was stored in the field before the assignment (e.g. pointer to the cell whose pixel gets overwritten).

This way the process of updating attributes of `CellG` object can be handled by appropriate `changeWatchers`. We will cover in detail examples of change watchers and things will become clearer then.

7.2.2 Energy Functions

Few lines below declaration of `cellField`, which as we know is an instance of `WatchableField3D<CellG *>` we find the declaration of containers associated with Energy function calculations. At this point we remind that the essence of Cellular Potts Model is in calculating **change of energy of the system due to randomly chosen lattice perturbation** (change of the single pixel). Pointers to energy functions objects are stored inside `Potts3D` object as follows:

```
/// An array of energy functions to be evaluated to determine energy costs.
std::vector<EnergyFunction *> energyFunctions;
EnergyFunction * connectivityConstraint;

std::map<std::string, EnergyFunction *> nameToEnergyFuctionMap;
```

All energy functions are actually objects and they all inherit base class `EnergyFunction`. `EnergyFunction` is defined inside `Potts3D/EnergyFunction.h` header file:

```
class EnergyFunction {
public:
    EnergyFunction() {}
    virtual ~EnergyFunction() {}

    virtual double localEnergy(const Point3D &pt){return 0.0;};

    virtual double changeEnergy(const Point3D &pt, const CellG *newCell, const_
↪CellG *oldCell)
    {
        if(1!=1);return 0.0;
    }
    virtual std::string toString()
    {
        return std::string("EnergyFunction");
    }
};
```

Each class that is responsible for calculating a **change in the overall system energy due to a proposed pixel copy** has to inherit `EnergyFunction`. The key function that has to be reimplemented in the derived class is virtual `double changeEnergy(const Point3D &pt, const CellG *newCell, const CellG *oldCell)`. After Metropolis algorithm function picks candidate for pixel overwrite it will then call `changeEnergy` for every element of the `energyFunctions` vector defined in class `Potts3D` (see above). The `pt` argument is a reference to a location of a pixel (specified as simple object `Point3D`) that would be overwritten as result of the pixel copy attempt. The `newCell` is pointer to a cell object that will occupy `pt` location of the `cellField` (if we accept pixel copy) and the `oldCell` is a pointer to a cell that currently occupies lattice location `pt`.

In CompuCell3D users declare which energy functions they want to use in their simulation so that the number of energy function in the `energyFunctions` vector will vary depending on what users specify in the CC3DML or in Python.

Later we will present detailed information on how to implement energy function plugins.

When we peek at the `metropolisFast` function of the `Potts3D` class we can see that the change of energy is calculated in a fairly straightforward way:

```
Point3D pt;

// Pick a random point
pt.x = rand->getInteger(sectionDims.first.x, sectionDims.second.x - 1);
pt.y = rand->getInteger(sectionDims.first.y, sectionDims.second.y - 1);
pt.z = rand->getInteger(sectionDims.first.z, sectionDims.second.z - 1);

CellG *cell = cellFieldG->getQuick(pt);

if (sizeFrozenTypeVec && cell) {///must also make sure that cell ptr is different 0;
↳Will never freeze medium
    if (checkIfFrozen(cell->type))
        continue;
}

unsigned int directIdx = rand->getInteger(0, maxNeighborIndex);

Neighbor n = boundaryStrategy->getNeighborDirect(pt, directIdx);

if (!n.distance) {
    //if distance is 0 then the neighbor returned is invalid
    continue;
}

Point3D changePixel = n.pt;

//check if changePixel refers to different cell.
CellG* changePixelCell = cellFieldG->getQuick(changePixel);

if (changePixelCell == cell) {
    //skip the rest of the loop if change pixel points to the same cell as pt
    continue;
}

if (sizeFrozenTypeVec && changePixelCell) {///must also make sure that cell ptr is
↳different 0; Will never freeze medium
    if (checkIfFrozen(changePixelCell->type))
        continue;
}
```

(continues on next page)

(continued from previous page)

```

++attemptedECVec[currentWorkNodeNumber];

flipNeighborVec[currentWorkNodeNumber] = pt;

/// change takes place at change pixel and pt is a neighbor of changePixel
// Calculate change in energy

double change = energyCalculator->changeEnergy(changePixel, cell, changePixelCell, i);

```

We first pick a random lattice location `pt` and retrieve pointer of a cell that occupies this location:

```
CellG *cell = cellFieldG->getQuick(pt);
```

We next make sure that the cell can move *i.e.* it is not frozen:

```

if (sizeFrozenTypeVec && cell) {///must also make sure that cell ptr is different 0;
    ↪Will never freeze medium
    if (checkIfFrozen(cell->type))
        continue;
}

```

Next we pick a random pixel out of set of neighbors of pixel `pt`:

We use `BoundaryStrategy` object pointed by `boundaryStrategy` to carry out all operations related to pixel neighbor operations. we will cover it later. For now it is important to remember that tracking and operating on pixel neighbors is usually done via `BoundaryStrategy` and this helps greatly when we have to deal with periodic boundary conditions pixels residing close to the edge of the lattice or classifying neighbor order of pixels. In this example we use boundary strategy to pick a neighbor `changePixel` of the `pt` and verify that this neighbor is a legitimate neighbor - if `(!n.distance)`. We next fetch cell that occupies `changePixel`:

```
CellG* changePixelCell = cellFieldG->getQuick(changePixel);
```

and verify that `changePixelCell` is different than `cell` at the location `pt`. We do this because overwriting pixel with the same cell pointer does not change lattice configuration at all. After also confirming that the `changePixelCell` is not frozen we compute change of energy if pixel `changePixel` currently occupied by `changePixelCell` were to be overwritten by `cell` currently residing at location `pt`. Or using `double changeEnergy(const Point3D &pt, const CellG *newCell, const CellG *oldCell)` terminology we can say that `pt <-> changePixel, newCell <-> cell` and `oldCell <-> changePixelCell` where we used `<->` symbol to illustrate how `changeEnergy` function arguments will be assigned in the call.

Interestingly, we call `changeEnergy` method of the object called `energyCalculator`:

```
double change = energyCalculator->changeEnergy(changePixel, cell, changePixelCell, i);
```

There is no magic here. If we look inside this function (`Potts3D/EnergyFunctionCalculator.cpp`) we see familiar summation over all values returned by `changeEnergy` of each `EnergyFunction` object:

```

double EnergyFunctionCalculator::changeEnergy(Point3D &pt, const CellG *newCell, const
    ↪CellG *oldCell, const unsigned int _flipAttempt){

    double change = 0;
    for (unsigned int i = 0; i < energyFunctions.size(); i++){
        change += energyFunctions[i]->changeEnergy(pt, newCell, oldCell);
    }
}

```

(continues on next page)

(continued from previous page)

```
    return change;
}
```

The reason we use `EnergyFunctionCalculator` object instead of implementing summation loop inside `metropolisFast` function is to handle additional tasks that might be associated with calculating energies - for example collecting information on every energy term associated with every pixel copy attempts. In this case we would use not `EnergyFunctionCalculator` but a more sophisticated version of this class called `EnergyFunctionCalculatorStatistics`

7.2.3 Steppers

A vector of Stepper objects - `std::vector<Stepper *> steppers;` is also a part of `Potts3D` object. Stepper objects all inherit from `Stepper` class defined in `Potts3D/Stepper.h` header file:

```
class Stepper {
public:
    virtual void step() = 0;
};
```

This is a very simple base class that defines only one function called `step`. More important is the question where and **why** we need this function. Steppers are called at the very end of the pixel copy attempt *i.e.* after all energy function calculation and if pixel copy was accepted after modifying `cellField`. Steppers are called always regardless whether pixel copy was accepted or not. A canonical example of the Stepper object is `VolumeTracker` declared and defined in `plugins/VolumeTracker/VolumeTrackerPlugin.h` and `plugins/VolumeTracker/VolumeTrackerPlugin.cpp`. `VolumeTracker` plugin tracks volume of each cell and ensures that cells' volume information is correct. It also removes dead cells *i.e.* those cells whose volume reached 0. In a sense it performs cleanup actions. However cleanup needs to be done as a very last action associated with pixel copy attempt. It would be a bad idea to do it earlier because we could remove cell object that might still be needed by other actions related to *e.g.* updating `cellField`.

7.2.4 Cell Inventory

`cellInventory` as its name suggest is an object that serves as a container for pointers to cell objects but it also allows fast lookups of particular cells. This is one of the most frequently accessed objects from Python (although we do it somewhat indirectly). Many of the Python modules you write for CC3D include the following loop:

```
for cell in self.cell_list:
    ...
```

What we are doing here is we iterate over every cell in the simulation. Internally the `self.cell_list` Python object accesses `cellInventory`. when we create a cell using `Potts3D`'s method `createCellG` we first construct cell object and then insert it into cell inventory. Similarly when we delete cell object using `destroyCellG` (method of `Potts3D`) we first remove the cell object from inventory and then carryout its destruction (which, as you know, is not just simple call to the C++ delete operator). It is worth knowing that in addition to cell inventories we track cell clusters and even links between cells (`FocalPointPlasticityPlugin`) via various "inventory" objects.

7.2.5 Acceptance Function and Fluctuation Amplitude Function

A key component of the Cellular Potts Model simulation is the so called acceptance function. It is the function that is responsible for the dynamic behavior of the simulation. It takes as an input a change in energy due to proposed pixel copy and outputs a probability with which this proposed pixel copy attempt will be accepted

Canonical formulation of the Cellular Potts Model acceptance function is as follows:

$$\begin{cases} P = e^{-(\Delta E - \delta)/kT} & \text{if } \Delta E > 0 \\ 1 & \text{if } \Delta E < 0 \\ 1/2 & \text{if } \Delta E = 0 \end{cases} \quad \text{where } \Delta E \text{ is a change in the energy due to proposed pixel copy attempt } T$$

is the the “temperature” which is a measure of cell membrane fluctuation amplitude and k is a constant which by default is set to 1 and δ is an energy offset by default set to 0

The higher the T is the higher the chance of accepting pixel copy attempts that result in higher energy. Those appear to be the “wrong” kind of attempts but it turns out that they often save the simulation from being stuck in a local minimum so ensuring some of them are accepted is essential.

The “temperature” or membrane fluctuation amplitude parameter can be set globally and many of the simulations using this convention. However, you can imagine that certain cells may have different membrane fluctuation amplitudes (different “temperatures”). To account for this fact and the fact that the two cells involved in pixel copy attempt may have different “temperatures” we use objects that derive from `FluctuationAmplitudeFunction` and whose goal is to compute effective “temperature” parameter associated with pixel copy based on the two “temperature” parameters that come from two cells involved in pixel copy. There are many possibilities here but the default strategy is to choose minimum of the two “temperatures”. The details can be found in `Potts3D/StandardFluctuationAmplitudeFunctions.h` and `Potts3D/StandardFluctuationAmplitudeFunctions.cpp`. We can also create new fluctuation amplitude functions depending on our needs.

Building Steppable

It is probably best to start discussing extension of CC3D by showing a relatively simple example of a steppable written in C++. In typical scenario steppables are written in Python. There are three main reasons for that **1)** No compilation is required. **2)** The code is more compact and easier to write than C++. **3)** Python has a rich set of libraries that make scientific computation easily accessible. However, writing a steppable in C++ is not that much more difficult, as you will see shortly, and you are almost guaranteed that your code will run orders of magnitude faster. Let me rephrase this last sentence - a **typical** code written in C++ is orders of magnitude faster than equivalent code written in pure Python. Since most of the steppable code consists of iterating over all cells and adjusting their attributes, C++ will perform this task much faster.

8.1 Getting started

Before you start developing CC3D C++ extension modules, you need to clone CC3D repository.

```
mkdir CC3D_DEVELOP
cd CC3D_DEVELOP
git clone https://github.com/CompuCell3D/CompuCell3D.git .
```

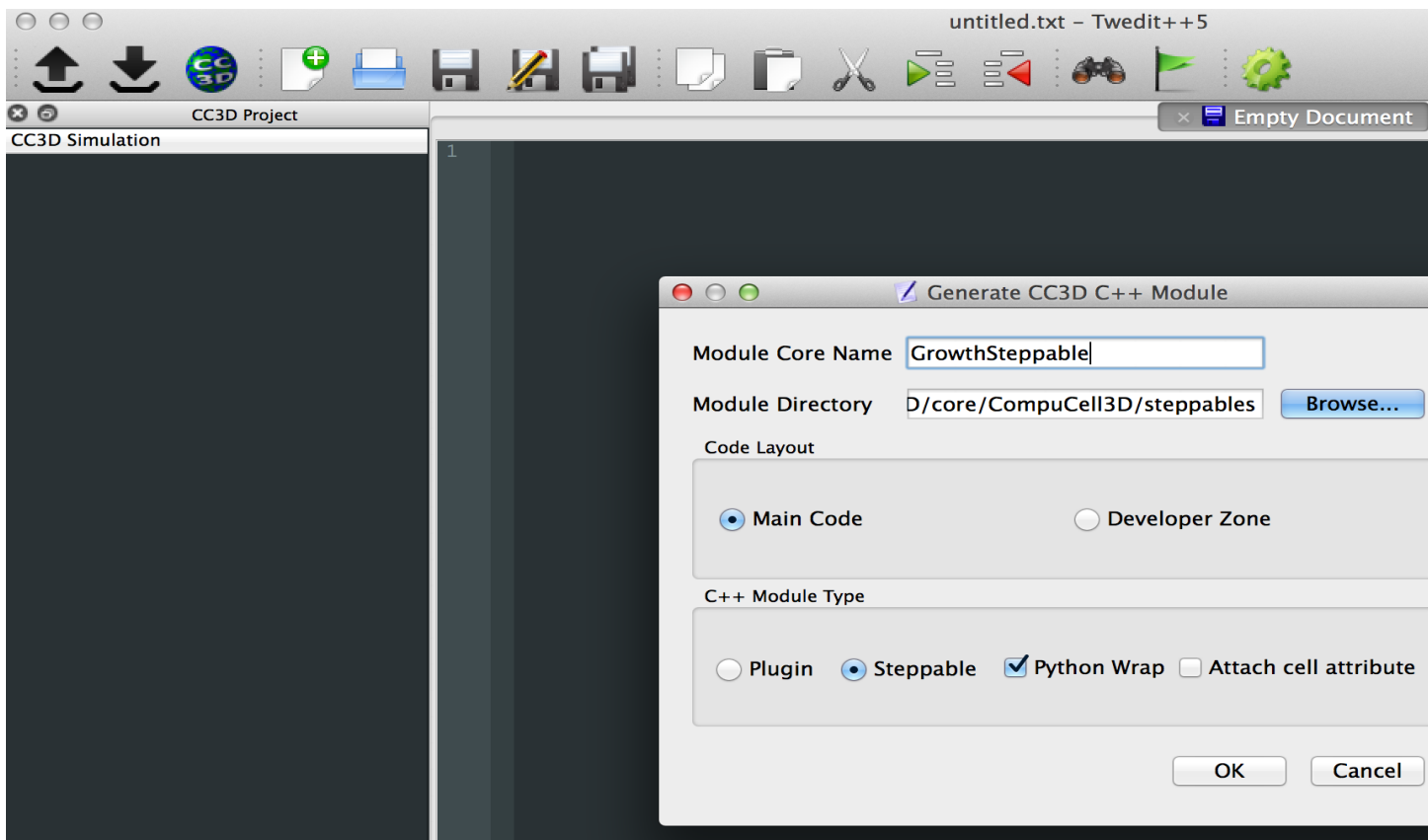
It is optional to checkout a particular branch of CC3D, but most often you will work with `master` branch. If , however, you want to checkout a branch - you would type something like this:

```
git checkout 4.0.0
```

```
1. mc [m@MacBook-Pro]:~/CC3D_developers_manual (bash)
bash-3.2$ mkdir CC3D_DEVELOP
bash-3.2$ git clone https://github.com/CompuCell3D/CompuCell3D.git .
fatal: destination path '.' already exists and is not an empty directory.
bash-3.2$ cd CC3D_DEVELOP/
bash-3.2$ git clone https://github.com/CompuCell3D/CompuCell3D.git .
Cloning into '.'...
remote: Enumerating objects: 309, done.
remote: Counting objects: 100% (309/309), done.
remote: Compressing objects: 100% (213/213), done.
remote: Total 30289 (delta 169), reused 175 (delta 95), pack-reused 29980
Receiving objects: 100% (30289/30289), 61.11 MiB | 561.00 KiB/s, done.
Resolving deltas: 100% (20277/20277), done.
Checking connectivity... done.
Checking out files: 100% (9990/9990), done.
bash-3.2$ █
```

At this point you have complete code in CC3D_DEVELOP directory. And in addition

Now we open Twedit++ - you need to have “standard” installation of CC3D on your machine available - and go to CC3D C++ menu and choose Generate New Module... entry and fill out the dialog box:



Note: In this example we show how to generate steppable code template in the “main” CC3D code. However, a more frequently used scenario is to Generate steppable code in “DeveloperZone” folder. We will show it in the subsequent chapter

This is exactly what we did:

1. We specify a name of a steppable as GrowthSteppable

2. We specify the location of when steppable code is stored `/Users/m/CC3D_DEVELOP/CompuCell3D/core/CompuCell3D/steppables`. Note, that we point to the cloned CC3D repository that we originally stored in CC3D_DEVELOP subdirectory. It happens that the path to this repository is `/Users/m/CC3D_DEVELOP`.
3. We select **Steppable** radio in the **C++ Module Type** panel. We also check `Python Wrap` checkbox to allow generation of Python bindings of this steppable.

When we press OK button Twedit++ will generate a complete set of template files that could be compiled as-is and the steppable will run. Obviously our goal is to modify template file to generate steppable we want. In current implementation of CC3D Twedit++ generates or modifies approximately 10 files.

```

1 Find_Package(OpenMP)
2
3 message(" OPEN MP FOUND " $OPENMP_FOUND)
4
5 INCLUDE_DIRECTORIES (
6     ${COMPUCELL3D_SOURCE_DIR}/core
7 )
8
9 if(${CMAKE_SYSTEM_NAME} STREQUAL Windows)
10     SET(STEPPABLE_DEPENDENCIES CompuCellLibShared XMLUtilsShared muParserStatic)
11 else(${CMAKE_SYSTEM_NAME} STREQUAL Windows)
12     SET(STEPPABLE_DEPENDENCIES CompuCellLibShared Potts3DShared BasicUtilsShared Field3DShared muParserShared)
13 endif(${CMAKE_SYSTEM_NAME} STREQUAL Windows)
14
15
16 MACRO(ADD_COMPUCCELL3D_STEPPABLE_HEADERS steppable)
17     # old sStyle command
18     # INSTALL_FILES(/include/CompuCell3D/CompuCell3D/steppables/${steppable} .h
19     # ${ARGN})
20
21     file(GLOB header_files "${CMAKE_CURRENT_SOURCE_DIR}/*.h")
22     # message(${steppable} "uses the following header files: " ${header_files})
23     INSTALL(FILES ${header_files} DESTINATION "${CMAKE_INSTALL_PREFIX}/include/CompuCell3D/CompuCell3D/steppables/${steppable}")
24
25 ENDMACRO(ADD_COMPUCCELL3D_STEPPABLE_HEADERS)
26
27
28 # #Recurse into subprojects
29 # #AutogeneratedModules - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO INSERT NEW PLUGIN DIRECTORY
30 ADD_SUBDIRECTORY(GrowthSteppable)
31 ADD_SUBDIRECTORY(BiasVectorSteppable)
32 ADD_SUBDIRECTORY(CleaverMeshDumper)
33 # # ADD_SUBDIRECTORY(CGALMeshDumper)
34
35 ADD_SUBDIRECTORY(PIFInitializer)
36 ADD_SUBDIRECTORY(PIFDumper)
37 ADD_SUBDIRECTORY(BlobFieldInitializer)
38 ADD_SUBDIRECTORY(UniformFieldInitializer)
39 ADD_SUBDIRECTORY(PDESolvers)
40 ADD_SUBDIRECTORY(FoamDataOutput)
41 ADD_SUBDIRECTORY(BoxWatcher)
42 ADD_SUBDIRECTORY(Dicty)
43 ADD_SUBDIRECTORY(Mitosis)
44 ADD_SUBDIRECTORY(ObjInitializer)
45 ADD_SUBDIRECTORY(RandomInitializers)
46

```

As you can see in the `CMakeLists.txt` file Twedit++ modified this file and added line `ADD_SUBDIRECTORY(GrowthSteppable)`

Now, let us focus on modifying template files and creating a steppable (`GrowthSteppable`) we specify growth rate in the XML and allow modification of this rate from Python.

Let's first examine the header of the `GrowthSteppable` class:

```

#ifndef GROWTHSTEPPABLESTEPPABLE_H
#define GROWTHSTEPPABLESTEPPABLE_H

#include <CompuCell3D/CC3D.h>
#include "GrowthSteppableDLLSpecifier.h"

```

(continues on next page)

(continued from previous page)

```

namespace CompuCell13D {

    template <class T> class Field3D;

    template <class T> class WatchableField3D;

    class Potts3D;
    class Automaton;
    class BoundaryStrategy;
    class CellInventory;
    class CellG;

    class GROWTHSTEPPABLE_EXPORT GrowthSteppable : public Steppable {

        WatchableField3D<CellG *> *cellFieldG;

        Simulator * sim;

        Potts3D *potts;

        CC3DXMLElement *xmlData;

        Automaton *automaton;

        BoundaryStrategy *boundaryStrategy;

        CellInventory * cellInventoryPtr;

        Dim3D fieldDim;

    public:

        GrowthSteppable ();

        virtual ~GrowthSteppable ();

        // SimObject interface

        virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);

        virtual void extraInit(Simulator *simulator);

        //steppable interface

        virtual void start();

        virtual void step(const unsigned int currentStep);

        virtual void finish() {}

        //SteerableObject interface

        virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);
    
```

(continues on next page)

(continued from previous page)

```

    virtual std::string steerableName();

    virtual std::string toString();

};

};

#endif

```

Each steppable defines virtual void start(), virtual void step(const unsigned int currentStep) and virtual void finish() functions. They have exactly the same role as analogous functions in Python scripting. The only difference is that C++ steppables will be called **before** Python steppables

Let us check the generated implementation file of the Steppable (the .cpp file):

```

#include <CompuCell3D/CC3D.h>
using namespace CompuCell3D;
using namespace std;
#include "GrowthSteppable.h"
GrowthSteppable::GrowthSteppable() : cellFieldG(0), sim(0), potts(0), xmlData(0),
↳boundaryStrategy(0), automaton(0), cellInventoryPtr(0) {}

GrowthSteppable::~GrowthSteppable() {}

}

void GrowthSteppable::init(Simulator *simulator, CC3DXMLElement *_xmlData) {

    xmlData=_xmlData;

    potts = simulator->getPotts();

    cellInventoryPtr=& potts->getCellInventory();

    sim=simulator;

    cellFieldG = (WatchableField3D<CellG *> *)potts->getCellFieldG();

    fieldDim=cellFieldG->getDim();

    simulator->registerSteerableObject(this);

    update(_xmlData,true);

}

void GrowthSteppable::extraInit(Simulator *simulator){

    //PUT YOUR CODE HERE

}

void GrowthSteppable::start() {

    //PUT YOUR CODE HERE

```

(continues on next page)

(continued from previous page)

```

}

void GrowthSteppable::step(const unsigned int currentStep) {

    //REPLACE SAMPLE CODE BELOW WITH YOUR OWN

    CellInventory::cellInventoryIterator cInvItr;

    CellG * cell=0;

    cerr<<"currentStep="<<currentStep<<endl;

    for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
    ↪cellInventoryEnd() ; ++cInvItr )

    {

        cell=cellInventoryPtr->getCell(cInvItr);

        cerr<<"cell.id="<<cell->id<<" vol="<<cell->volume<<endl;

    }

}

void GrowthSteppable::update(CC3DXMLElement *_xmlData, bool _fullInitFlag){

    //PARSE XML IN THIS FUNCTION

    //For more information on XML parser function please see CC3D code or lookup XML_
    ↪utils API

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
    ↪THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

    set<unsigned char> cellTypesSet;

    CC3DXMLElement * exampleXMLElem=_xmlData->getFirstElement("Example");

    if (exampleXMLElem) {

        double param=exampleXMLElem->getDouble();

        cerr<<"param="<<param<<endl;

        if(exampleXMLElem->findAttribute("Type")) {

            std::string attrib=exampleXMLElem->getAttribute("Type");

            // double attrib=exampleXMLElem->getAttributeAsDouble("Type"); //in case_
            ↪attribute is of type double

            cerr<<"attrib="<<attrib<<endl;

```

(continues on next page)

(continued from previous page)

```

    }

    }

    //boundaryStrategy has information about pixel neighbors

    boundaryStrategy=BoundaryStrategy::getInstance();
}

std::string GrowthSteppable::toString() {

    return "GrowthSteppable";

}

std::string GrowthSteppable::steerableName() {

    return toString();

}

```

The `step` and `start` functions are the first function we will modify. In its current implementation the generated `step` function already contains helpful code but `start` function will be rewritten. Let's take a look:

```

void GrowthSteppable::start() {

}

void GrowthSteppable::step(const unsigned int currentStep) {

    CellInventory::cellInventoryIterator cInvItr;

    CellG * cell=0;

    cerr<<"currentStep="<<currentStep<<endl;

    for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
    ↪cellInventoryEnd() ; ++cInvItr )

    {

        cell = cellInventoryPtr->getCell(cInvItr);

        cerr << "cell.id=" << cell->id << " vol=" << cell->volume << endl;

    }

}

```

The `for` loop iterates over inventory of cells and prints cell id and cell volume. To iterate over cell inventory we are using `cellInventoryPtr` which is a pointer to `CellInventory` object. The class for this object (`CellInventory`) is defined in `Potts3D/CellInventory.h` and implementation is in `Potts3D/CellInventory.cpp`. Internally, we are using STL(Standard Template Library - C++) maps to keep track of cells. The statement `cellInventoryPtr->cellInventoryBegin()` returns an iterator to cell in-

ventory. If you look closely at the implementation files the container we are using as a cell inventory is `std::map<CellIdentifier, CellG *>` and `CellIdentifier` contains cell id and cluster id to uniquely identify cells. Therefore iteration over cell inventory is simply iteration over STL map. If you are not familiar with concept of iterators and containers of STL we recommend that you look up basic C++ tutorials for example: https://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.

Let us now modify the above `start` and `step` functions and implement first version of growth steppable:

```
void GrowthSteppable::start() {

    CellInventory::cellInventoryIterator cInvItr;
    CellG * cell = 0;

    for (cInvItr = cellInventoryPtr->cellInventoryBegin(); cInvItr !=
    ↪cellInventoryPtr->cellInventoryEnd(); ++cInvItr)
    {

        cell = cellInventoryPtr->getCell(cInvItr);
        cell->targetVolume = 25.0;
        cell->lambdaVolume = 2.0;
    }
}

void GrowthSteppable::step(const unsigned int currentStep) {

    CellInventory::cellInventoryIterator cInvItr;

    CellG * cell=0;

    float growthRate = 1.0;

    for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
    ↪cellInventoryEnd() ; ++cInvItr )
    {

        cell = cellInventoryPtr->getCell(cInvItr);
        cell->targetVolume += growthRate ;

    }

}
```

When we create cells they all have `targetVolume` and `lambdaVolume` set to 0.0 and thus volume constraint does nothing. We fix it by setting those parameters for each cell in the `start` function.

If you are familiar with CC3D Python scripting you will quickly find analogies. The only thing we added was the following statement `cell->targetVolume += growthRate ;`

When we compile and run this example the cells' target volume will increase by amount hardcoded in the `growthRate` variable which in our case is 1.0.

Let's take it to the next level (slowly). Now we will write a code that increases target volume of cells but only for the first 100 MCS and only if cell type is equal to 1.

```
void GrowthSteppable::step(const unsigned int currentStep) {

    if (currentStep > 100)
```

(continues on next page)

(continued from previous page)

```

    return;

    CellInventory::cellInventoryIterator cInvItr;

    CellG * cell=0;

    float growthRate = 1.0;

    for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
    ↪cellInventoryEnd() ; ++cInvItr )

    {

        cell = cellInventoryPtr->getCell(cInvItr);
        if (cell->type == 1){
            cell->targetVolume += growthRate ;
        }

    }

}

```

First thing we do in this steppable is checking if current MCS is greater than 100 and if so we return. Inside the loop we added `if (cell->type == 1)` check that allows increase of target volume only if cell is of type 1. Small digression here. If you want to print cell type to the screen you need to use the following syntax:

```
cerr << "cell type=" << (int)cell->type <<endl;
```

As you can see we are performing type cast to `int`. This is because cell type (defined in `Potts3D/Cell.h`) is defined as `unsigned char`. Consequently CC3D allows only 256 cell types, which at first sight might look limiting but in practice is more than enough.

In the previous examples we hard-coded the value of growth rate using `float growthRate = 1.0;`. This is not an optimal solution. What if you want to run 5 simulations simultaneously each one with different value of growth rate. If you hard-code values you would need to have 5 distinct compilations of CC3D available. Clearly, hard-coding is not scalable. We need better solution. It is time to learn how to parse XML in C++ code

8.2 Parsing XML in C++

Building flexible code requires that we provide some sensible configuration mechanism via which users can customize their simulation without the need to recompile code. In CC3D we have two ways of achieving it **1) XML 2) Python** scripting. It is up to you which one you use and we will teach you how to use both approaches. For now let's start with XML parsing.

All C++ CC3D Plugins and Steppables define virtual function `update(CC3DXMLElement *_xmlData, bool _fullInitFlag)`. This function takes two arguments: pointer to XML element `_xmlData` (that CC3D initializes to be the root element of the particular Plugin or Steppable) and a flag `_fullInitFlag` that specifies if full initialization of the module is required or not.

Suppose that our XML will look as follows:

```

<Steppable Type="GrowthSteppable">
    <GrowthRate>1.0</GrowthRate>
</Steppable>

```

We would parse this XML in C++ using the following code:

```
void GrowthSteppable::update(CC3DXMLElement *_xmlData, bool _fullInitFlag){

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
↳THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

    set<unsigned char> cellTypesSet;

    CC3DXMLElement * growthElem = _xmlData->getFirstElement("GrowthRate");

    if (growthElem){

        this->growthRate = growthElem->getDouble();

    }

    //boundaryStrategy has information about pixel neighbors

    boundaryStrategy=BoundaryStrategy::getInstance();

}
```

As we mentioned before `_xmlData` points to `<Steppable Type="GrowthSteppable">`. We need to get the child of this element *i.e.* `<GrowthRate>1.0</GrowthRate>`. Since we know that there is only one child element (let's say we make such constraint for now - we will relax it later) we use the following code:

```
CC3DXMLElement * growthElem = _xmlData->getFirstElement("GrowthRate");
```

The `getFirstElement` method returns a pointer to a child element that is of the form

```
<GrowthRate ...>...</GrowthRate>
```

The returned pointer can be NULL if suitable child element cannot be found. This is why we add `if (growthElem)` check. Assuming that the `<GrowthRate>` child exist we read its `cdata` part. For any XML element, `cdata` part (`cdata` stands for character data) is the part that sits between closing `>` and opening `<` brackets of XML element. For example in

```
<GrowthRate>1.0</GrowthRate>
```

the `cdata` part is 1.0. The `CC3DXMLElement` has several methods that read and convert `cdata` to appropriate C++ type. Here we are using `getDouble()`

```
this->growthRate = growthElem->getDouble();
```

Obviously, `CC3DXMLElement` defines more methods to convert character data to required type (`getInt`, `getBool`, *etc...*) They are defined in `XMLUtils/CC3DXMLElement.h`

In order for this code to work we need to define `growthRate` inside `GrowthSteppable` class header - we can do it as follows:

```
class GROWTHSTEPPABLE_EXPORT GrowthSteppable : public Steppable {

    WatchableField3D<CellG *> *cellFieldG;
```

(continues on next page)

(continued from previous page)

```

Simulator * sim;

Potts3D *potts;

CC3DXMLElement *xmlData;

Automaton *automaton;

BoundaryStrategy *boundaryStrategy;

CellInventory * cellInventoryPtr;

Dim3D fieldDim;

public:

    GrowthSteppable ();

    virtual ~GrowthSteppable ();

    double growthRate;

    ...
}

```

With those changes we can rewrite our step function as:

It is almost the same implementation as before except we use `cell->targetVolume += this->growthRate;` instead of `cell->targetVolume += growthRate;`

The `this->growthRate` gets initialized based on the input provided in

```

<Steppable Type="GrowthSteppable">
  <GrowthRate>1.0</GrowthRate>
</Steppable>

```

If we change it to

```

<Steppable Type="GrowthSteppable">
  <GrowthRate>2.0</GrowthRate>
</Steppable>

```

and rerun the simulation the rate of increase of target volume will be 2.0. All the changes we make to the growth rate now do not require recompilation but only changes in the XML file, exactly how CC3D is designed to work. Next we will learn how to parse attributes of the XML elements. As a motivating example we will specify different growth rates for different cell types.

8.2.1 Parsing XMI Attributes

If we want our simulation to have different growth rates for different cell types we need to store them in *e.g.* STL map and we need to modify header of the `GrowthSteppable` to look as follows:

```

class GROWTHSTEPPABLE_EXPORT GrowthSteppable : public Steppable {

    WatchableField3D<CellG *> *cellFieldG;

```

(continues on next page)

(continued from previous page)

```

Simulator * sim;

Potts3D *potts;

CC3DXMLElement *xmlData;

Automaton *automaton;

BoundaryStrategy *boundaryStrategy;

CellInventory * cellInventoryPtr;

Dim3D fieldDim;

public:

    GrowthSteppable ();

    virtual ~GrowthSteppable ();

    std::map<unsigned int, double> growthRateMap;

    ...
}

```

We replaced `double growthRate` with `std::map<unsigned int, double> growthRateMap`; The key of the map is cell type and the value is growth rate. Now we need to design and parse XML that will allow users to specify required data. Let us try the following syntax:

```

<Steppable Type="GrowthSteppable">
  <GrowthRate CellType="1">1.3</GrowthRate>
  <GrowthRate CellType="2">1.7</GrowthRate>
</Steppable>

```

In case you wonder what I mean by “trying out syntax” it means that it is up to you to design XML syntax in such a way that it allows you to specify model in the way you want. The above example fulfills this requirement because we specify different growth rates for different cell types. However, we could also come up with a different way of specifying the same information:

```

<Steppable Type="GrowthSteppable">
  <GrowthRate CellType="1" Rate="1.3"/>
  <GrowthRate CellType="2" Rate="1.7"/>
</Steppable>

```

Both approaches are OK.

Let us write the update function that will parse first of the above XMLs:

```

void GrowthSteppable::update(CC3DXMLElement *_xmlData, bool _fullInitFlag) {

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
↳THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

```

(continues on next page)

(continued from previous page)

```

set<unsigned char> cellTypesSet;

CC3DXMLElementList growthVec = _xmlData->getElements("GrowthRate");

for (int i = 0; i < growthVec.size(); ++i) {
    unsigned int cellType = growthVec[i]->getAttributeAsUInt("CellType");
    double growthRateTmp = growthVec[i]->getDouble();
    this->growthRateMap[cellType] = growthRateTmp;
}

//boundaryStrategy has information about pixel neighbors
boundaryStrategy=BoundaryStrategy::getInstance();
}

```

The code is slightly different this time because we expect multiple entries of the type `<GrowthRate CellType="xxx" Rate="yyy"/>`. Therefore, by writing the code:

```
CC3DXMLElementList growthVec = _xmlData->getElements("Rate");
```

we ensure that CC3D will return a list (actually it is implemented as an STL vector) of XML element pointers that start with `<GrowthRate ...>`. Next, we iterate over the vector of XML element pointers and notice that `growthVec[i]` returns a pointer to XML element pointer and we query this element. First, we read and convert to unsigned int value of `CellType` attribute:

```
unsigned int cellType = growthVec[i]->getAttributeAsUInt("CellType");
```

The next line:

```
double growthRateTmp = growthVec[i]->getDouble();
```

should be familiar already because it reads the value of cdata of `<GrowthRate CellType="1">1.3</GrowthRate>`

Once we extracted cell type and actual growth rate from a single element we store those values in `this->growthRateMap` map:

```
this->growthRateMap[cellType] = growthRateTmp;
```

Note: We are not performing any error checks in the above code and assume that users enter reasonable values. In the production code we would monitor for possible errors but this extra code would make this introductory manual a bit too confusing

If we wanted to parse second syntax where we specify growth rate as and attribute rather than cdata :

```

<Steppable Type="GrowthSteppable">
    <GrowthRate CellType="1" Rate="1.3"/>
    <GrowthRate CellType="2" Rate="1.7"/>
</Steppable>

```

we would need to make only small modification:

```

void GrowthSteppable::update(CC3DXMLElement *_xmlData, bool _fullInitFlag){

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
↳THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

    set<unsigned char> cellTypesSet;

    CC3DXMLElementList growthVec = _xmlData->getElements("GrowthRate");

    for (int i = 0; i < growthVec.size(); ++i) {
        unsigned int cellType = growthVec[i]->getAttributeAsUInt("CellType");
        double growthRateTmp = growthVec[i]->getAttributeAsDouble("GrowthRate");
        this->growthRateMap[cellType] = growthRateTmp;
    }

    //boundaryStrategy has information about pixel neighbors
    boundaryStrategy=BoundaryStrategy::getInstance();

}

```

The code differs from previous parsing code by only one line:

```

double growthRateTmp = growthVec[i]->getAttributeAsDouble("GrowthRate");

```

As usual for a complete list of functions that read and convert XML attributes to concrete C++ types , check XMLUtils/CC3DXMLElement.h

In order to take advantage of the specification of growth rate on a per-cell-type basis we modify step function as follows:

```

void GrowthSteppable::step(const unsigned int currentStep){

    CellInventory::cellInventoryIterator cInvItr;

    CellG * cell=0;

    if (currentStep > 100)
        return;

    std::map<unsigned int, double>::iterator mitr;

    for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
↳cellInventoryEnd() ; ++cInvItr )
    {

        cell=cellInventoryPtr->getCell(cInvItr);

        mitr = this->growthRateMap.find((unsigned int) cell->type);

        if (mitr != this->growthRateMap.end()){
            cell->targetVolume += mitr->second;
        }

    }

}

```

(continues on next page)

(continued from previous page)

```
}
```

We declare an iterator to the `std::map<unsigned int, double>`. Hint: iterator is like a pointer and in the case of map iterator will have two components `mitr->first` which will be a key of `this->growthRateMap` map (in our case a key is a cell type) and `mitr->second` which will point to a value of the `this->growthRateMap` which in our case is a growth rate.

When we get a new cell first thing we do is to check if iterator pointing to a pair of (cell type, growth rate) exist:

```
mitr = this->growthRateMap.find((unsigned int)cell->type);
```

If such entry exists in the `this->growthRateMap` then this iterator will point to a value different than `this->growthRateMap.end()` and in such a case we know that `mitr->second` points to a growth rate for a cell type given by `cell->type`. We simply increase target volume of such cell by the growth rate. This logic is code up in the following if statement”:

```
if (mitr != this->growthRateMap.end()) {  
    cell->targetVolume += mitr->second;  
}
```

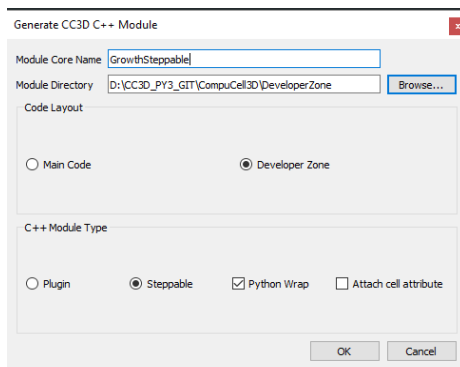
The presented example went over a theory of how to build a basic steppable and integrate it with main CC3D code. In the next tutorial we will present the same steppable but we will build it in the `DevelopeZone` folder of CC3D. The idea here is that this new steppable can live outside main CC3D code and still be accessible by the installed binaries.

Building Growth Steppable In the Developer Zone folder

Quite often you will want to build a steppable in a “non-intrusive” way *i.e.* without adding it to the main CC3D code-base. The way to do it is to utilize functionality of `DeveloperZone`.

This time we will use Windows system and our CC3D git repository is cloned to `D:\CC3D_PY3_GIT\CompuCell3D`

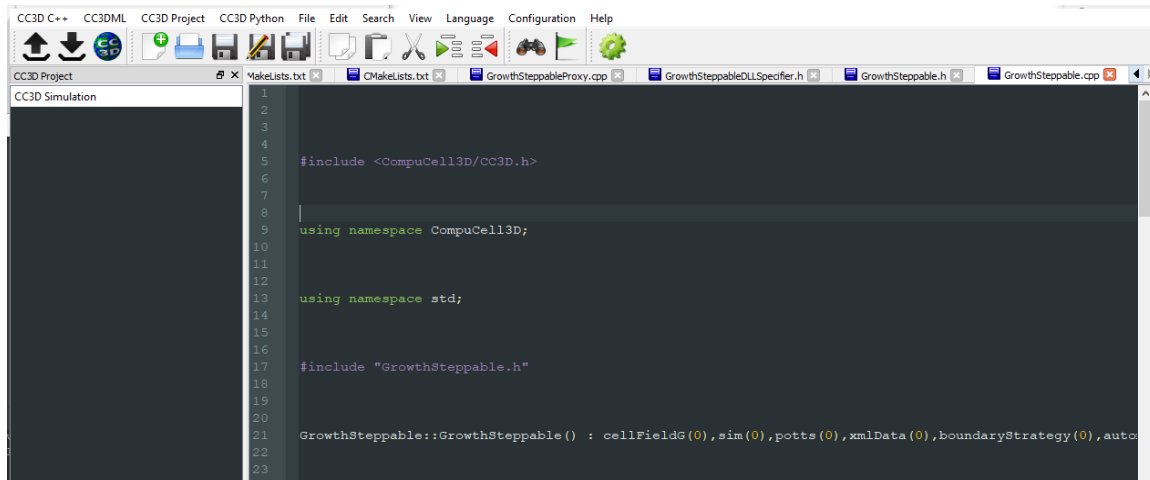
To add a steppable or plugin in the `DeveloperZone` you open up Twedit and from CC3D C++ menu select `Generate New Module....`



Notice that in the `Module Directory` in the dialog box we put `D:\CC3D_PY3_GIT\CompuCell3D\DeveloperZone`. Previously we put there a path to the `Steppable` folder in the main CC3D Code base (give where our repository is cloned this path would be `D:\CC3D_PY3_GIT\CompuCell3D\core\CompuCell3D\steppables\`)

Notice that we also checked `Python Wrap` option to generate Python bindings. We will show you how you can be creative here and leverage both XML and Python as a way to pass parameters to the Steppable. As you remember you do not have to generate Python bindings and it is perfectly OK to stick with C++ and XML.

After we press `OK` button Twedit++ will generate , a template Steppable code:



Now we copy code from our earlier example into appropriate files - we are only showing files that we modified: GrowthSteppable.h:

```

1 #ifndef GROWTHSTEPPABLESTEPPABLE_H
2 #define GROWTHSTEPPABLESTEPPABLE_H
3 #include <CompuCell3D/CC3D.h>
4 #include "GrowthSteppableDLLSpecifier.h"
5
6 namespace CompuCell3D {
7
8     template <class T> class Field3D;
9
10    template <class T> class WatchableField3D;
11
12    class Potts3D;
13
14    class Automaton;
15
16    class BoundaryStrategy;
17
18    class CellInventory;
19
20    class CellG;
21
22    class GROWTHSTEPPABLE_EXPORT GrowthSteppable : public Steppable {
23
24        WatchableField3D<CellG *> *cellFieldG;
25
26        Simulator * sim;
27
28        Potts3D *potts;
29
30        CC3DXMLElement *xmlData;
31
32        Automaton *automaton;
33
34        BoundaryStrategy *boundaryStrategy;
35
36        CellInventory * cellInventoryPtr;
37
38        Dim3D fieldDim;

```

(continues on next page)

(continued from previous page)

```

39
40 public:
41
42     GrowthSteppable ();
43
44     virtual ~GrowthSteppable ();
45
46     std::map<unsigned int, double> growthRateMap;
47
48     // SimObject interface
49
50     virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);
51
52     virtual void extraInit(Simulator *simulator);
53
54
55     //steppable interface
56
57     virtual void start();
58
59     virtual void step(const unsigned int currentStep);
60
61     virtual void finish() {}
62
63
64     //SteerableObject interface
65
66     virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);
67
68     virtual std::string steerableName();
69
70     virtual std::string toString();
71
72 };
73
74 };
75
76 #endif

```

and GrowthSteppable.cpp

```

1  #include <CompuCell3D/CC3D.h>
2  using namespace CompuCell3D;
3  using namespace std;
4  #include "GrowthSteppable.h"
5
6
7  GrowthSteppable::GrowthSteppable() :
8  cellFieldG(0), sim(0), potts(0), xmlData(0),
9  boundaryStrategy(0), automaton(0), cellInventoryPtr(0) {}
10
11  GrowthSteppable::~GrowthSteppable() {
12
13  }
14
15  void GrowthSteppable::init(Simulator *simulator, CC3DXMLElement *_xmlData) {
16

```

(continues on next page)

(continued from previous page)

```

17   xmlData=_xmlData;
18
19   potts = simulator->getPotts();
20
21   cellInventoryPtr=& potts->getCellInventory();
22
23   sim=simulator;
24
25   cellFieldG = (WatchableField3D<CellG *> *)potts->getCellFieldG();
26
27   fieldDim=cellFieldG->getDim();
28
29   simulator->registerSteerableObject (this);
30
31   update (_xmlData,true);
32 }
33
34 void GrowthSteppable::extraInit(Simulator *simulator){
35
36 }
37
38 void GrowthSteppable::start() {
39
40     CellInventory::cellInventoryIterator cInvItr;
41     CellG * cell = 0;
42
43     for (cInvItr = cellInventoryPtr->cellInventoryBegin(); cInvItr !=
44 ↪ cellInventoryPtr->cellInventoryEnd(); ++cInvItr)
45     {
46         cell = cellInventoryPtr->getCell(cInvItr);
47         cell->targetVolume = 25.0;
48         cell->lambdaVolume = 2.0;
49     }
50 }
51
52 void GrowthSteppable::step(const unsigned int currentStep) {
53
54     CellInventory::cellInventoryIterator cInvItr;
55
56     CellG * cell=0;
57
58     if (currentStep > 100)
59         return;
60
61     std::map<unsigned int, double>::iterator mitr;
62
63     for(cInvItr=cellInventoryPtr->cellInventoryBegin() ; cInvItr !=cellInventoryPtr->
64 ↪ cellInventoryEnd() ;++cInvItr )
65     {
66         cell=cellInventoryPtr->getCell(cInvItr);
67
68         mitr = this->growthRateMap.find((unsigned int) cell->type);
69
70         if (mitr != this->growthRateMap.end()) {
71             cell->targetVolume += mitr->second;

```

(continues on next page)

(continued from previous page)

```

72     }
73
74     }
75
76 }
77
78 void GrowthSteppable::update(CC3DXMLElement *_xmlData, bool _fullInitFlag){
79
80     automaton = potts->getAutomaton();
81
82     ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
↳THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)
83
84     set<unsigned char> cellTypesSet;
85
86     CC3DXMLElementList growthVec = _xmlData->getElements("GrowthRate");
87
88     for (int i = 0; i < growthVec.size(); ++i) {
89         unsigned int cellType = growthVec[i]->getAttributeAsUInt("CellType");
90         double growthRateTmp = growthVec[i]->getAttributeAsDouble("Rate");
91         this->growthRateMap[cellType] = growthRateTmp;
92     }
93
94     //boundaryStrategy has information about pixel neighbors
95     boundaryStrategy=BoundaryStrategy::getInstance();
96
97 }
98
99 std::string GrowthSteppable::toString(){
100
101     return "GrowthSteppable";
102 }
103
104 std::string GrowthSteppable::steerableName(){
105
106     return toString();
107 }

```

As you can see based on the previous discussion the update function where we parse XML is designed to handle the following syntax for the GrowthSteppable:

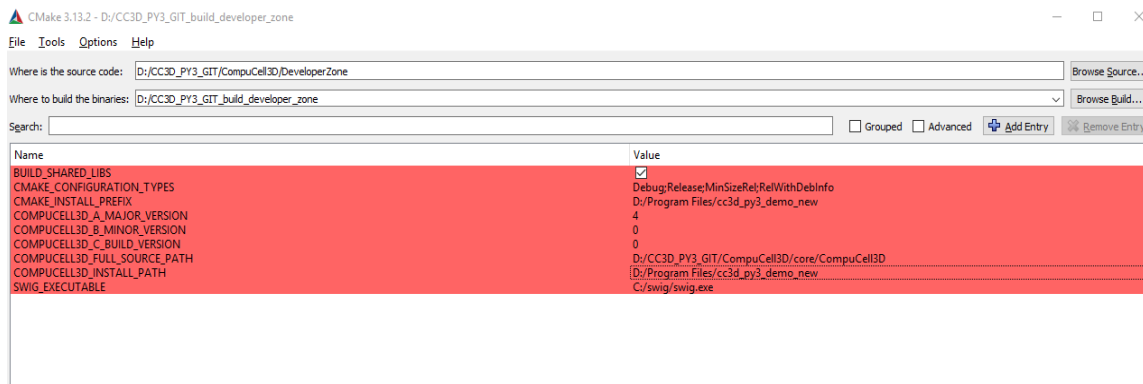
```

<Steppable Type="GrowthSteppable">
  <GrowthRate CellType="1" Rate="1.3"/>
  <GrowthRate CellType="2" Rate="1.7"/>
</Steppable>

```

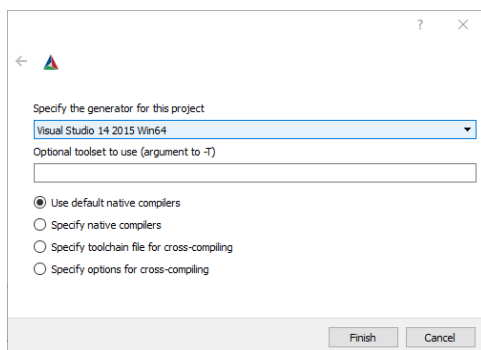
Note: Starting from version 4.3.0 of CC3D the DeveloperZone compilation setup (for any compiler) is is done automatically. All you need to do is to follow procedure outlined in [Configuring Developer Zone](#)

After we generated plugin code and added our modification to those two files, we are ready to begin compilation. We will show how to compile code on Windows. Compilation on Linux system is analogous up to CMake configuration part but then instead of using Visual Studio you will type `make` and `make install` in the terminal. For now let's stick with Windows compilation. After Twedit++ generated new files in the Developer Zone we need to use CMake tool (GUI - as we will go here, or console based tool) to configure our compilation. This is how CMake configuration looks in our case



First we point to the folder where DeveloperZone is (Where the source code is). In our case it is D:\CC3D_PY3_GIT\CompuCell3D\DeveloperZone and location for our Visual Studio project D:/CC3D_PY3_GIT_build_developer_zone (see Where to build the binaries)

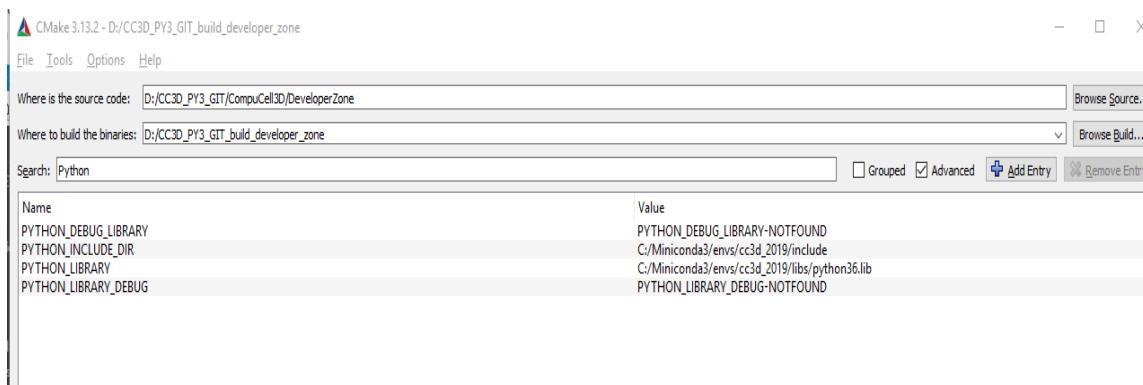
Then we after click Configure CMake will display the following dialog:



Make sure to select Visual Studio 14 2015 Win64 (we assume we are using 64-bit version of CC3D). If you are using 32-bit version then you would select Visual Studio 14 2015

Next, we set CMAKE_INSTALL_PREFIX and COMPUCELL3D_INSTALL_PATH to the folder where CC3D is installed - D:\Program Files\cc3d_py3_demo_new.

We also set where main CC3D code-base is COMPUCELL3D_FULL_SOURCE_PATH D:/CC3D_PY3_GIT/CompuCell3D/core/CompuCell3D Next, we set version number (4, 0, 0). We are almost done but since DeveloperZone also compiles Python module we must set Python paths as follows (you need to specify Python include directory and Python library path):



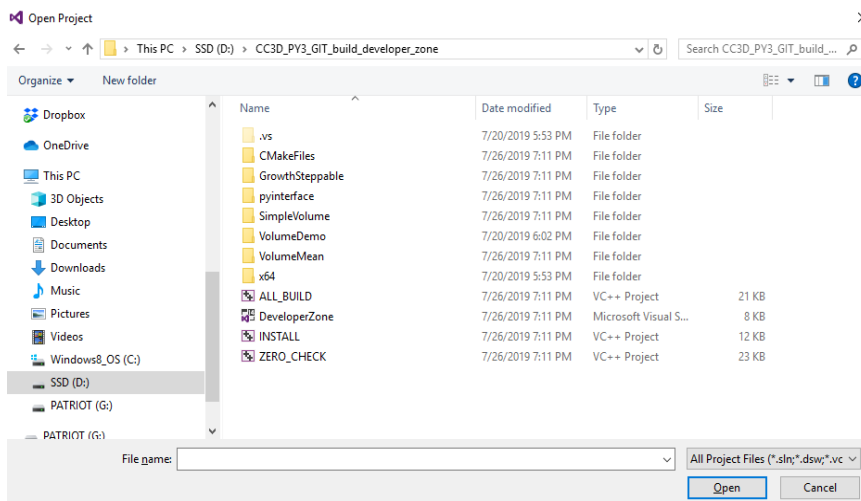
Note: It is perfectly fine to compile DeveloperZone modules without using Python. If this is what you would like

to do, just comment out line `add_subdirectory(pyinterface)` in `DeveloperZone/CMakeLists.txt`

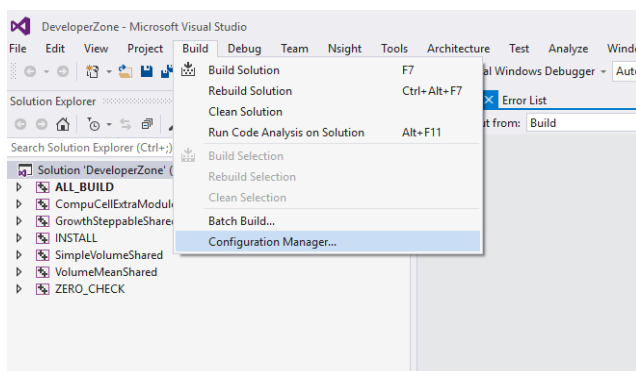
After we configured all paths in CMake GUI we press `Configure` button and then `Generate` button. The VisualStudio Project will be placed in `D:/CC3D_PY3_GIT_build_developer_zone` (see `Where to build the binaries` at the top of CMake GUI). We will open it next and will show you how to compile plugins and steppables in the `DeveloperZone`

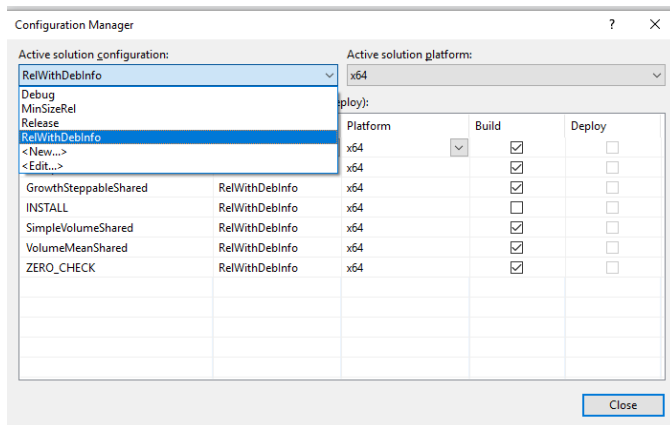
9.1 Compiling DeveloperZone in Visual Studio

Now that we created Visual Studio project for Developer Zone we will show you how to set up compilation. We open up Visual Studio and navigate to `File->Open->Project/Solution...` and in the File Open Dialog we go to `D:/CC3D_PY3_GIT_build_developer_zone` and select `ALL_BUILD.vcxproj`

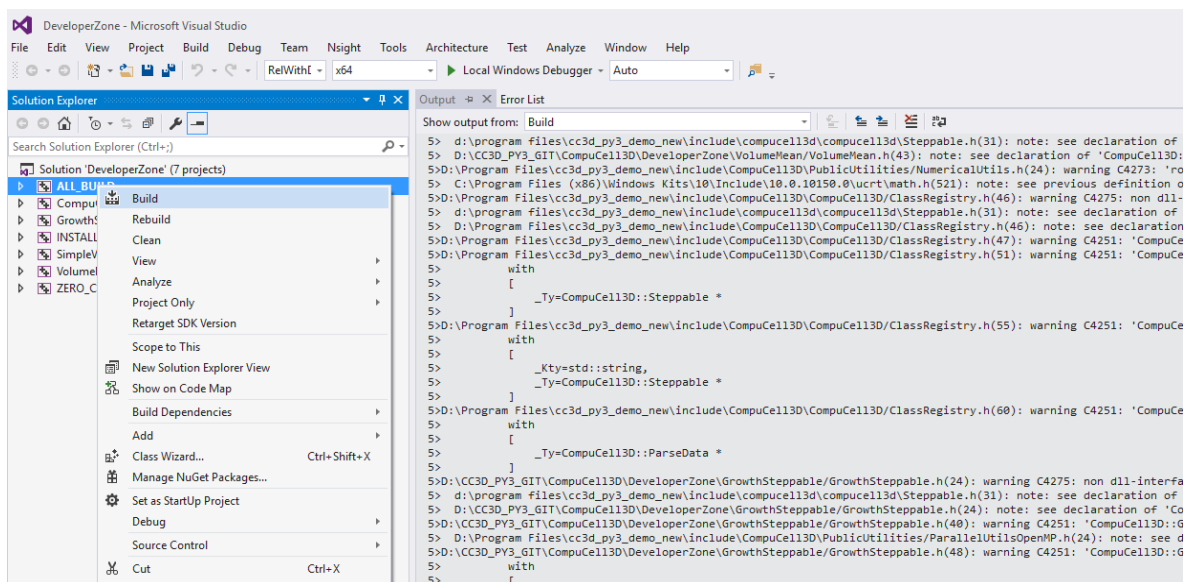


After `DeveloperZone` Visual Studio project gets loaded we go to `Build->Configuration Manager...` and from the pull down menu `Active Solution Configuration` (at the top of the dialog box) we select `RelWithDebInfo`:



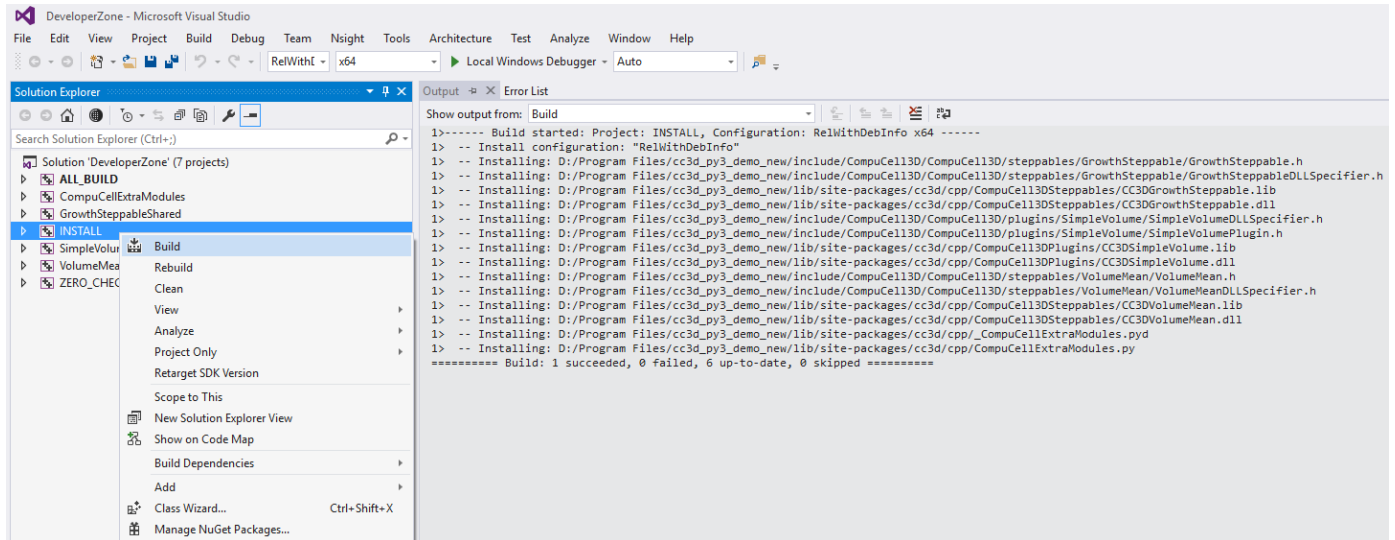


Next, to start compilation, we right-click on `ALL_BUILD` and from the context menu select `Build`:



Notice that there are additional modules in addition to our `GrowthSteppable`. Take a look at those. They show how to write simple modules (plugins or steppables).

After the compilation finished and there are no errors, we right-click at `INSTALL` subproject and from the context menu we select `Build`. This will install our newly created `GrowthSteppable` in the CC3D installation directory that we specified during CMake configuration (`D:/Program Files/cc3d_py3_demo_new`)



At this point we can build a simulation that will use newly created GrowthSteppable

9.1.1 Using DeveloperZone steppable in the simulation

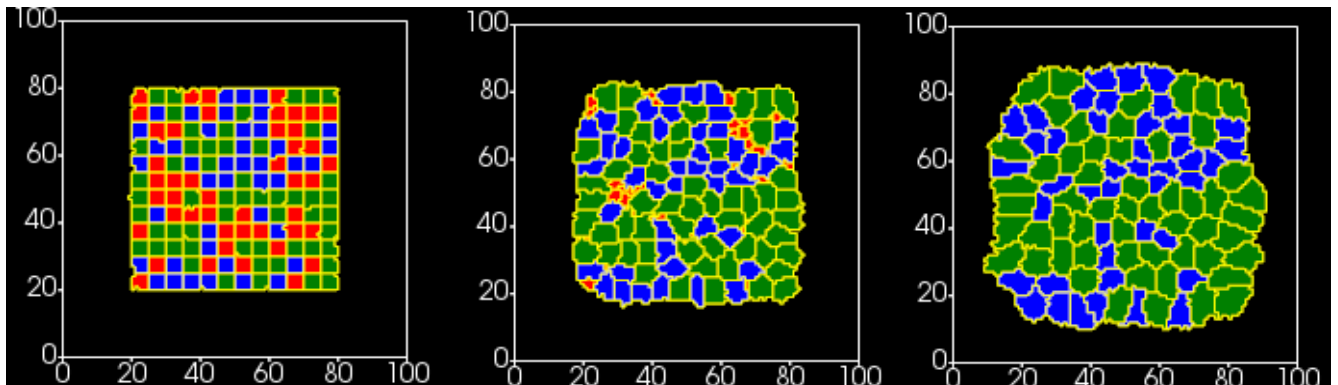
Writing C++ code and compiling it was a hard-part of the project. Using newly created steppable in the simulation is easy. In fact all we need to do is to add

```
<Steppable Type="GrowthSteppable">
  <GrowthRate CellType="1" Rate="1.3"/>
  <GrowthRate CellType="2" Rate="1.7"/>
</Steppable>
```

to any simulation where we want cell of type 1 to increase target volume at 1.3 pixels/MCS rate and for cells of type 2 the growth would be 1.7.

Note: The name of the steppable or plugin that we reference from XML is not based on module name but on the label encoded in the proxy file. In our case GrowthSteppableProxy.cpp has the following line `growthSteppableProxy("GrowthSteppable", ...)` and there we have label GrowthSteppable that we use in XML. If we changed this label to e.g. `growthSteppableProxy("MyGrowthSteppable", ...)` then we would need to change first line of XML for GrowthSteppable to `<Steppable Type="GrowthSteppable">`

Here are the results of the simulation at MCS 0, 20, and 40:



As you can see there are 3 cell types here but we specified growth rates for two of them As a result “red” cells are getting squashed by growing neighbors and at MCS 40 they disappear. Also notice that green cells are bigger than blue ones. This is what we expect when we have different growth rates.

Since we are modifying target volume we must use Volume plugin where we control all parameters for each cell individually - we use “local flex” version of Volume constraint `<Plugin Name="Volume"/>` where we only load Volume plugin but dont pass any parameters to it. Those parameters `targetVolume` `lambdaVolume` are set in C++ code:

```
<CompuCell3D Revision="20190604" Version="4.0.0">

  <Potts>

    <!-- Basic properties of CPM (GGH) algorithm -->
    <Dimensions x="100" y="100" z="1"/>
    <Steps>100000</Steps>
    <Temperature>10.0</Temperature>
    <NeighborOrder>1</NeighborOrder>
  </Potts>

  <Plugin Name="CellType">

    <!-- Listing all cell types in the simulation -->
    <CellType TypeId="0" TypeName="Medium"/>
    <CellType TypeId="1" TypeName="A"/>
    <CellType TypeId="2" TypeName="B"/>
    <CellType TypeId="3" TypeName="C"/>
  </Plugin>

  <Plugin Name="Volume"/>

  <Plugin Name="CenterOfMass">

    <!-- Module tracking center of mass of each cell -->
  </Plugin>

  <Plugin Name="Contact">
    <!-- Specification of adhesion energies -->
    <Energy Type1="Medium" Type2="Medium">10.0</Energy>
    <Energy Type1="Medium" Type2="A">10.0</Energy>
    <Energy Type1="Medium" Type2="B">10.0</Energy>
    <Energy Type1="Medium" Type2="C">10.0</Energy>
    <Energy Type1="A" Type2="A">10.0</Energy>
    <Energy Type1="A" Type2="B">10.0</Energy>
    <Energy Type1="A" Type2="C">10.0</Energy>
    <Energy Type1="B" Type2="B">10.0</Energy>
    <Energy Type1="B" Type2="C">10.0</Energy>
    <Energy Type1="C" Type2="C">10.0</Energy>
    <NeighborOrder>4</NeighborOrder>
  </Plugin>

  <Steppable Type="UniformInitializer">

    <!-- Initial layout of cells in the form of rectangular slab -->
    <Region>
      <BoxMin x="20" y="20" z="0"/>
      <BoxMax x="80" y="80" z="1"/>
      <Gap>0</Gap>
    </Region>
  </Steppable>
</CompuCell3D>
```

(continues on next page)

(continued from previous page)

```
<Width>5</Width>
<Types>A,B,C</Types>
</Region>
</Steppable>

<Steppable Type="GrowthSteppable">
  <GrowthRate CellType="1" Rate="1.3"/>
  <GrowthRate CellType="2" Rate="1.7"/>
</Steppable>

</CompuCell3D>
```

Note: We placed `GrowthSteppable` last. This is not coincidence. We must place it after steppable that creates cells `UniformInitializer`. If we reversed the order of those two steppables `GrowthSteppable` would be called first and in particular its `start` function would be called before `UniformInitializer` function and as a result the code that is supposed to set initial volume constraint parameters from `GrowthSteppable` (`start` function) would iterate over empty cell inventory. Therefore, listing `UniformInitializer` before `GrowthSteppable` is the right thing to do. Simply put order of appearances of steppables in the XML determines the order in which CC3D will call them

Note: Specified growth rates of 1.3 and 1.7 are very high and we used them for illustration purposes. In your simulation you should use much smaller rates to allow cells on the lattice to “equilibrate”

Full simulation can be downloaded [here](#) zip and full code for `GrowthSteppable` is [here](#) zip. You can also access both example and c++ code by going directly to `CompuCell3D/DeveloperZone`

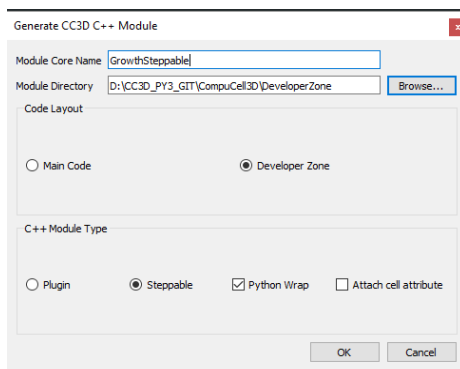
Adding Python Bindings To Steppable in DeveloperZone

In the previous example we controled entire simulation from C++. This is perfectly fine and will give you optimal performance. However sometimes it may make sense to add Python bindings to your module . Especially if the functions you wil call from python will not be called many times - functions calls in Python are much slower than in C++.

In addition to this if your entire code is in C++ every change you make to the code will require compilation and installation. This is is not a big deal but takes time and is more error prone than using well designed scripting interface. However, do not feel that you need to use Python bindings for your newly created C++ modules. They are optional and it is perfectly fine to operate in C++ space.

Nevertheless we would like to show you how to add and use Python bindings if you feel it will be beneficial for your simulation.

If you remember, the first step to generate steppable code using Twedit++ is to choose whether you like to add Python bindings or not.



In this first dialog box we checked Python Wrap option and therefore we already generated Python bindings. They are stored in SWIG file in DeveloperZone/pyinterface/CompuCellExtraModules/CompuCellExtraModules.i:

```

%module ("threads"=1) CompuCellExtraModules
#include "typemaps.i"
#include <windows.i>

%{
#include "ParseData.h"
#include "ParserStorage.h"
#include <CompuCell3D/Simulator.h>
#include <CompuCell3D/Potts3D/Potts3D.h>

#include <BasicUtils/BasicClassAccessor.h>
#include <BasicUtils/BasicClassGroup.h> //had to include it to avoid problems with
↳template instantiation

// ***** PUT YOUR PLUGIN PARSE DATA AND
↳PLUGIN FILES HERE *****

#include <SimpleVolume/SimpleVolumePlugin.h>

#include <VolumeMean/VolumeMean.h>

//AutogeneratedModules1 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//GrowthSteppable_autogenerated

#include <GrowthSteppable/GrowthSteppable.h>

// ***** END OF SECTION
↳*****

//have to include all export definitions for modules which are arapped to avoid
↳problems with interpreting by swig win32 specific c++ extensions...
#define SIMPLEVOLUME_EXPORT
#define VOLUMEMEAN_EXPORT
//AutogeneratedModules2 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//GrowthSteppable_autogenerated
#define GROWTHSTEPPABLE_EXPORT

#include <iostream>

using namespace std;
using namespace CompuCell3D;

%}

// C++ std::string handling
#include "std_string.i"

// C++ std::map handling
#include "std_map.i"

```

(continues on next page)

(continued from previous page)

```

// C++ std::map handling
#include "std_set.i"

// C++ std::vector handling
#include "std_vector.i"

//have to include all export definitions for modules which are arapped to avoid
↳problems with interpreting by swig win32 specific c++ extensions...
#define SIMPLEVOLUME_EXPORT
#define VOLUMEMEAN_EXPORT

//AutogeneratedModules3 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//GrowthSteppable_autogenerated
#define GROWTHSTEPPABLE_EXPORT

#include <BasicUtils/BasicClassAccessor.h>
#include <BasicUtils/BasicClassGroup.h> //had to include it to avoid problems with
↳template instantiation

#include "ParseData.h"
#include "ParserStorage.h"

// ***** PUT YOUR PLUGIN PARSE DATA AND
↳PLUGIN FILES HERE *****
// REMEMBER TO CHANGE #include to %include

#include <SimpleVolume/SimpleVolumePlugin.h>
// %include <SimpleVolume/SimpleVolumeParseData.h>

// THIS IS VERY IMORTANT STETEMENT WITHOUT IT SWIG will produce incorrect wrapper
↳code which will compile but will not work
using namespace CompuCell3D;

%inline %{
    SimpleVolumePlugin * reinterpretSimpleVolumePlugin(Plugin * _plugin){
        return (SimpleVolumePlugin *)_plugin;
    }

    SimpleVolumePlugin * getSimpleVolumePlugin(){
        return (SimpleVolumePlugin *)Simulator::pluginManager.get("SimpleVolume");
    }
%}

#include <VolumeMean/VolumeMean.h>

%inline %{
    VolumeMean * reinterpretVolumeMean(Steppable * _steppable){
        return (VolumeMean *)_steppable;
    }

    VolumeMean * getVolumeMeanSteppable(){
        return (VolumeMean *)Simulator::steppableManager.get("VolumeMean");
    }
}

```

(continues on next page)

(continued from previous page)

```

%}

//AutogeneratedModules4 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE_
↪INSERTION POINT
//GrowthSteppable_autogenerated

#include <GrowthSteppable/GrowthSteppable.h>
%inline %{

    GrowthSteppable * getGrowthSteppable() {

        return (GrowthSteppable *) Simulator::steppableManager.get("GrowthSteppable");
    }

%}

```

We are not going to explain how SWIG wrappers work here but if you look at the file and look for occurrences of `GrowthSteppable` you can see that adding your own steppable to the SWIG wrapper generator is fairly easy. On top of that if you use Twedit++ it will generate wrapper code for you.

Note: At the top of the wrapper file we find `%module ("threads"=1) CompuCellExtraModules`. This tells us that the Python module we develop will be called `CompuCellExtraModules`.

10.1 Adding Python-Accessible Method To GrowthSteppable

If we enable compilation of `CompuCellExtraModules` by uncommenting line `add_subdirectory(pyinterface)` in `DeveloperZone/CMakeLists.txt` we will already get `GrowthSteppable` bindings that we can access from `CompuCellExtraModules`. However, they are not particularly useful because all the functions accessible via Python are for functions that are already being used by the C++ code and, frankly it is best to leave it like that. We need to add additional function. A reasonable choice is a function that changes growth rate for a given cell type.

First we need to add `void setGrowthRate(unsigned int cellType, double growthRate);` ``function to ```GrowthSteppable` header - see below:

```

#ifndef GROWTHSTEPPABLESTEPPABLE_H
#define GROWTHSTEPPABLESTEPPABLE_H
#include <CompuCell3D/CC3D.h>
#include "GrowthSteppableDLLSpecifier.h"

namespace CompuCell3D {

    template <class T> class Field3D;
    template <class T> class WatchableField3D;

    class Potts3D;
    class Automaton;
    class BoundaryStrategy;
    class CellInventory;
    class CellG;

    class GROWTHSTEPPABLE_EXPORT GrowthSteppable : public Steppable {

```

(continues on next page)

(continued from previous page)

```

WatchableField3D<CellG *> *cellFieldG;

Simulator * sim;

Potts3D *potts;

CC3DXMLElement *xmlData;

Automaton *automaton;

BoundaryStrategy *boundaryStrategy;

CellInventory * cellInventoryPtr;

Dim3D fieldDim;

public:

    GrowthSteppable ();

    virtual ~GrowthSteppable ();

    std::map<unsigned int, double> growthRateMap;

    // SimObject interface

    virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);

    virtual void extraInit(Simulator *simulator);

    // Python wrapper functions

    void setGrowthRate(unsigned int cellType, double growthRate);

    //steppable interface

    virtual void start();

    virtual void step(const unsigned int currentStep);

    virtual void finish() {}

    //SteerableObject interface

    virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);

    virtual std::string steerableName();

    virtual std::string toString();

};

};

#endif

```

Next, we add implementation of this function in the `GrowthSteppable.cpp`:

```
void GrowthSteppable::setGrowthRate(unsigned int cellType, double growthRate){  
    cerr<<"CHANGING GROWTH RATE FOR CELL TYPE "<<cellType<<" TO "<<growthRate<<endl;  
    this->growthRateMap[cellType] = growthRate;  
}
```

The implementation of this function is pretty straightforward - it is a function that takes two arguments (`unsigned int cellType` and `double growthRate` and prints out message to the screen that it is about to change growth rate for a given cell type and then assigns a growth rate to a given entry in the `this->growthRateMap`.

Why does this function make sense to be implemented in Python? If you think about a simulation where you want to run many simulations that need to modify growth rate at a particular MCS but you don't know which MCS it will be you can write a simple code where you could try many time points at this you change growth rate and see if the outcome matches your expectations. Obviously, you could do it all in c++ but then you would need to pass more parameters to the XML making XML harder and harder to understand or you could hard-code everything in C++ but then you would need to recompile `DeveloperZone` every time you run the simulation. You quickly realize that Python provides convenient platform for handling situations like this. This is why , it makes perfect sense to add Python bindings to your C++ modules.

At this point we can recompile the `DeveloperZone` but before we do it it is essential that we “touch” `DeveloperZone/pyinterface/CompuCellExtraModules/CompuCellExtraModules.i` by e.g. add or remove empty line in this file and re-saving it.

Warning: If you add new method to header file and want this method be accessible in Python bindings you must force SWIG to re-generate bindings. One way of doing so is by “refreshing” the file but making adding (or removing) extra empty line and saving it. In the future we will write better CMake code to avoid this manual step but for now you should be aware of this limitation.

After we compiled and installed `DeveloperZone` modules we can rerun the simulation. Now, however, we will add Python code where we show you how to access new C++ steppable from Python.

Here is python Steppable code (`GrowthSteppablePythonModules`):

```
from cc3d.core.PySteppables import *  
from cc3d.cpp import CompuCellExtraModules  
  
class GrowthSteppablePython(SteppableBasePy):  
    def __init__(self, frequency=1):  
        SteppableBasePy.__init__(self, frequency)  
        self.growth_steppable_cpp = None  
  
    def start(self):  
        self.growth_steppable_cpp = CompuCellExtraModules.getGrowthSteppable()  
  
    def step(self, mcs):  
        if mcs == 10:  
            self.growth_steppable_cpp.setGrowthRate(1, -1.2)
```

At the top of the file we import `CompuCellExtraModules`. This is the module that SWIG generated for us In `__init__` constructor we create a variable that will hold a reference to the C++ `GrowthSteppable`. In `start` function we access C++ `GrowthSteppable` by typing:

```
self.growth_steppable_cpp = CompuCellExtraModules.getGrowthSteppable()
```

If you look at the end of the `DeveloperZone/pyinterface/CompuCellExtraModules/CompuCellExtraModules.i` you will see `getGrowthSteppable()` declared there. In other words `getGrowthSteppable()` function will become a function of the `CompuCellExtraModules` and therefore we access it as `CompuCellExtraModules.getGrowthSteppable()`. Now, we can get creative, because we can access every publicly defined function of the C++ `GrowthSteppable`. This is exactly what we do in the `step` function. We call our newly added function

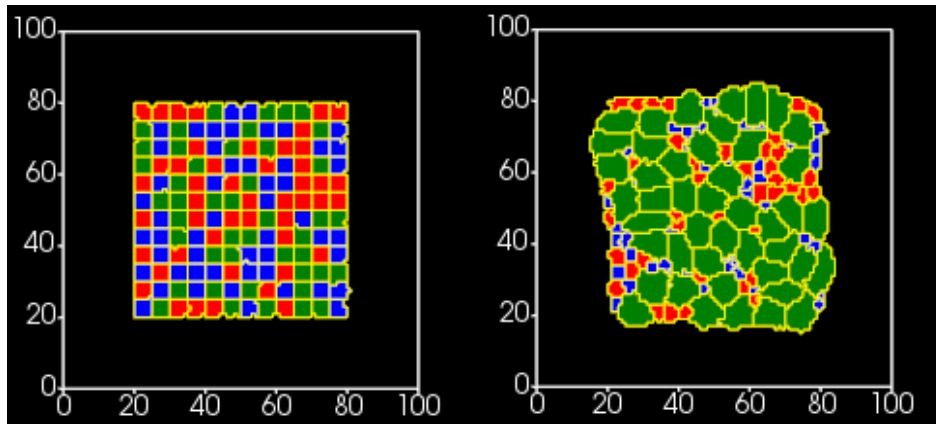
```
self.growth_steppable_cpp.setGrowthRate(1, -1.2)
```

This call at `MCS=10` changes growth of cells of type `1` into shrinking rate.

When we run the simulation at `mcs==10` the text output will look as follows:

```
total number of pixel copy attempts=10000
Number of Attempted Energy Calculations=910
Step 8 Flips 210/10000 Energy -803.2 Cells 144 Inventory=144
Metropolis Fast
total number of pixel copy attempts=10000
Number of Attempted Energy Calculations=870
Step 9 Flips 236/10000 Energy -1074.4 Cells 144 Inventory=144
Metropolis Fast
total number of pixel copy attempts=10000
Number of Attempted Energy Calculations=859
Step 10 Flips 236/10000 Energy -1244 Cells 144 Inventory=144
CHANGING GROWTH RATE FOR CELL TYPE 1 TO -1.2
```

You can see there our C++ printout being triggered by calling `setGrowthRate` from Python level. And the simulation configuration at `MCS 0` and `30` respectively will look as follows:



Notice that the blue cells almost disappeared. This is the result of the negative growth rate we we set by calling `self.growth_steppable_cpp.setGrowthRate(1, -1.2)`.

The C++ code for this example can be found in `DeveloperZone/GrowthSteppable`, python bindings are in `DeveloperZone/pyinterface/CompuCellExtraModules/CompuCellExtraModules.i` and the simulation example is in `CompuCell3D/DeveloperZone/Demos/GrowthSteppablePython`

Computing Heterotypic Boundary Length

Heterotypic boundary surface (or length in 2D) is total surface of contact between all cells of two types. For example when you have 2 cells of type 1 and 100 cells of type 2 the heterotypic surface between the two will be a sum of all contact surfaces between the two types. In this example we are not going to show every step how we generate the steppable using Twedit. We have shown this earlier and here we will concentrate on the actual code.

This example is a bit more advanced but we will explain clearly every line of code.

The module that we generated is called `HeterotypicBoundaryLength`. We then click `Configure` and `Generate` in the CMake Gui and start writing actual code. We will first implement function that walks over entire lattice and computes heterotypic surface (or length in 2D) between all cells of different types.

All C++ files can be found in `DeveloperZone/Demos/HeterotypicBoundarySurface` and Python bindings are , as usual in `DeveloperZone/pyinterface/CompuCellExtraModules/CompuCellExtraModules.i`. The simulation example that uses our newly created module is in `DeveloperZone/Demos/HeterotypicBoundarySurface`

Here is the header file for our new steppable:

```
#ifndef HETEROTYPICBOUNDARYLENGTHSTEPPABLE_H
#define HETEROTYPICBOUNDARYLENGTHSTEPPABLE_H
#include <CompuCell3D/CC3D.h>
#include "HeterotypicBoundaryLengthDLLSpecifier.h"

namespace CompuCell3D {

    template <class T> class Field3D;
    template <class T> class WatchableField3D;

    class Potts3D;
    class Automaton;
    class BoundaryStrategy;
    class CellInventory;
    class CellG;

    class HETEROTYPICBOUNDARYLENGTH_EXPORT HeterotypicBoundaryLength : public Steppable
    {
```

(continues on next page)

(continued from previous page)

```

WatchableField3D<CellG *> *cellFieldG;

Simulator * sim;
Potts3D *potts;
CC3DXMLElement *xmlData;
Automaton *automaton;
BoundaryStrategy *boundaryStrategy;
CellInventory * cellInventoryPtr;
Dim3D fieldDim;

public:

HeterotypicBoundaryLength ();

virtual ~HeterotypicBoundaryLength ();

// new methods and members

std::map<unsigned int, double> typePairHTSurfaceMap;

unsigned int typePairIndex(unsigned int cellType1, unsigned int cellType2);
void calculateHeterotypicSurface();
double getHeterotypicSurface(unsigned int cellType1, unsigned int cellType2);

// SimObject interface

virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);

virtual void extraInit(Simulator *simulator);

//steppable interface

virtual void start();

virtual void step(const unsigned int currentStep);

virtual void finish() {}

//SteerableObject interface

virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);

virtual std::string steerableName();

    virtual std::string toString();

};

};

#endif

```

we added few methods and one class member there:

```
// new methods and members

std::map<unsigned int, double> typePairHTSurfaceMap;

unsigned int typePairIndex(unsigned int cellType1, unsigned int cellType2);
void calculateHeterotypicSurface();
double getHeterotypicSurface(unsigned int cellType1, unsigned int cellType2);
```

The typePairHTSurfaceMap is a dictionary (map) that will store heterotypic boundary surface between different cell types. Notice that we will encode pair of cell types as a single unsigned integer (hence a key to the dictionary is unsigned integer). To do this we will use convenience function unsigned int typePairIndex(unsigned int cellType1, unsigned int cellType2) that takes as its arguments two unsigned integers that denote cell type 1 and cell type 2. Here is the implementation of this function:

```
unsigned int HeterotypicBoundaryLength::typePairIndex(unsigned int cellType1,
↳ unsigned int cellType2) {
    return 256 * cellType2 + cellType1;
}
```

we take advantage of the fact that the number of cell types in CC3D is limited to 256 and the index this function returns looks analogous to the index you would use to access a matrix if you were to store a matrix as 1D array.

Next we have two functions calculateHeterotypicSurface() that computed actual total heterotypic surface between all cell types and double getHeterotypicSurface(unsigned int cellType1, unsigned int cellType2) that given two types it returns a boundary between them.

Let's start analyzing code for calculateHeterotypicSurface function:

```
1 void HeterotypicBoundaryLength::calculateHeterotypicSurface() {
2
3     unsigned int maxNeighborIndex = this->boundaryStrategy->
↳ getMaxNeighborIndexFromNeighborOrder(1);
4     Neighbor neighbor;
5
6     CellG *nCell = 0;
7
8     this->typePairHTSurfaceMap.clear();
9
10    // note: unit surface is different on a hex lattice. if you are runnign
11    // this steppable on hex lattice you need to adjust it. Remember that on hex_
↳ lattice unit length and unit surface have different values
12    double unit_surface = 1.0;
13
14    cerr << "Calculating HTBL for all cell type combinations" << endl;
15
16    for (unsigned int x = 0; x < fieldDim.x; ++x)
17        for (unsigned int y = 0; y < fieldDim.y; ++y)
18            for (unsigned int z = 0; z < fieldDim.z; ++z) {
19                Point3D pt(x, y, z);
20                CellG *cell = this->cellFieldG->get(pt);
21
22                unsigned int cell_type = 0;
23                if (cell) {
24                    cell_type = (unsigned int)cell->type;
25                }
26
27                for (unsigned int nIdx = 0; nIdx <= maxNeighborIndex; ++nIdx) {
```

(continues on next page)

(continued from previous page)

```

28      neighbor = boundaryStrategy->getNeighborDirect (const_cast<Point3D&
    ↪>(pt), nIdx);
29      if (!neighbor.distance) {
30          //if distance is 0 then the neighbor returned is invalid
31          continue;
32      }
33
34      nCell = this->cellFieldG->get(neighbor.pt);
35      unsigned int n_cell_type = 0;
36      if (nCell) {
37          n_cell_type = (unsigned int)nCell->type;
38      }
39
40      if (nCell != cell) {
41          unsigned int pair_index_1 = typePairIndex(cell_type, n_cell_
    ↪type);
42          unsigned int pair_index_2 = typePairIndex(n_cell_type, cell_
    ↪type);
43          this->typePairHTSurfaceMap[pair_index_1] += unit_surface;
44          if (pair_index_1 != pair_index_2) {
45              this->typePairHTSurfaceMap[pair_index_2] += unit_surface;
46          }
47      }
48  }
49
50  }
51
52  }
53
54  }

```

We will be iterating over lattice pixels. Every lattice pixel has neighbors of different order but 1-st order neighbors are simply adjacent pixels. `BoundaryStrategy` is an object that facilitates iteration over pixel neighbors and it also keeps track of boundary conditions, pixels, adjacent to the boundary etc. so that you can write a simpler code. All we need to do to iterate over 1-st order pixel neighbors is to know what is the maximum number of them and this is what we do in this line:

```

unsigned int maxNeighborIndex = this->boundaryStrategy->
    ↪getMaxNeighborIndexFromNeighborOrder(1);

```

We get maximum index of a 1-st order pixel (`BoundaryStrategy` keeps them in a vector and we are getting max index of this vector). On 2D. cartesian lattice there could be up to 4 such neighbors hence the max vector index is 3 (we start counting from 0).

We next clear the map where we store our results because each time we call this function it will be incrementing the values so if we did not clear we would be starting counting from different value than zero.

At line 16 we start triple loop where we iterate over all lattice pixels. This might not be the most efficient method but it is the simplest to code.

In lines 19 and 20 we get a cell that resides at a given pixel. If the cell pointer returned is `NULL` we are dealing with `Medium` and cell and this is why in lines 23-25 we check if the cell is different than `NULL` (`if (cell)`) before accessing its type. If it is null that we do not execute line 24 and the cell type is 0 as it should be for the `Medium`.

At line 27 we start iterating over neighbors of the current pixel (`Point3D pt(x, y, z)`). This is where we do actual calculations. Code in line 28 fetches one of the neighbor of pixel `pt(x, y, z)`. In line 30 we check if this neighbor is a valid one (e.g. if you are at the edge of the lattice we may get pixel that is outside of the lattice and then

if `neighbor.distance` is zero we know we are dealing with invalid pixel), hence in the line 31 we skip the rest of the loop. If, however, the pixel is valid then we get a cell that resides at the neighboring pixel (line 34):

In lines 35-38 we extract cell type of neighbor cell , again we have to be mindful of `Medium` as we did in lines 22-25.

Finally, lines 41-45 contain actual code that increments boundary surface between two cell types. This code runs only if `nCell` and `cell` *i.e.* cells belonging to adjacent pixels are different cells. In this case we compute index for type of `nCell` and type of `cell` (

```
unsigned int pair_index_1 = typePairIndex(cell_type, n_cell_type);
```

and increment appropriate entry in the `this->typePairHTSurfaceMap` - lines 43. Notice that we also permute cell types in call to `typePairIndex` - line 44-45. so that when we access boundary length between cell type 1 and 2 it will give us the same value as between cell types 2 and 1. But we do this only when the two types are different

Obviously, we are double-counting and we correct this in the function that returns heterotypic surfaces:

```
double HeterotypicBoundaryLength::getHeterotypicSurface(unsigned int cellType1,
↳ unsigned int cellType2) {
    unsigned int pair_index = typePairIndex(cellType1, cellType2);

    double heterotypic_surface = this->typePairHTSurfaceMap[pair_index]/2.0;

    return heterotypic_surface;
}
```

11.1 Running the Simulation with Heterotypic Surface Calculator

The simulation code is quite easy to write as it follows the same pattern that we encountered in the previous chapter where we introduced Python bindings to the C++ steppable. We start with an XML file:

```
1 <CompuCell3D Revision="20190604" Version="4.0.0">
2   <Potts>
3     <!-- Basic properties of CPM (GGH) algorithm -->
4     <Dimensions x="256" y="256" z="1"/>
5     <Steps>100000</Steps>
6     <Temperature>10.0</Temperature>
7     <NeighborOrder>1</NeighborOrder>
8   </Potts>
9
10  <Plugin Name="CellType">
11    <!-- Listing all cell types in the simulation -->
12    <CellType TypeId="0" TypeName="Medium"/>
13    <CellType TypeId="1" TypeName="A"/>
14    <CellType TypeId="2" TypeName="B"/>
15  </Plugin>
16
17  <Plugin Name="Volume">
18    <VolumeEnergyParameters CellType="A" LambdaVolume="2.0" TargetVolume="50"/>
19    <VolumeEnergyParameters CellType="B" LambdaVolume="2.0" TargetVolume="50"/>
20  </Plugin>
21
22  <Plugin Name="CenterOfMass">
23    <!-- Module tracking center of mass of each cell -->
24  </Plugin>
25
26  <Plugin Name="Contact">
```

(continues on next page)

(continued from previous page)

```

27      <!-- Specification of adhesion energies -->
28      <Energy Type1="Medium" Type2="Medium">10.0</Energy>
29      <Energy Type1="Medium" Type2="A">10.0</Energy>
30      <Energy Type1="Medium" Type2="B">10.0</Energy>
31      <Energy Type1="A" Type2="A">10.0</Energy>
32      <Energy Type1="A" Type2="B">10.0</Energy>
33      <Energy Type1="B" Type2="B">10.0</Energy>
34      <NeighborOrder>4</NeighborOrder>
35  </Plugin>
36
37  <Steppable Type="UniformInitializer">
38      <!-- Initial layout of cells in the form of rectangular slab -->
39      <Region>
40          <BoxMin x="51" y="51" z="0"/>
41          <BoxMax x="204" y="204" z="1"/>
42          <Gap>0</Gap>
43          <Width>7</Width>
44          <Types>A,B</Types>
45      </Region>
46  </Steppable>
47
48  <Steppable Type="HeterotypicBoundaryLength"/>
49
50 </CompuCell3D>

```

It is Twedit-generated XML file that has basic energy terms (Volume and Contact Constraints) plus initializer and at the end in line 48 we add our new HeterotypicBoundaryLength steppable. Notice that this is a one-line call because we are not really passing any parameters to the steppable from the XML and our update (CC3DXMLElement *_xmlData, bool _fullInitFlag=false) method does not contain any code that parses XML.

Note: It is important that every module (steppable, plugin) that you develop in C++ be instantiated in XML. Otherwise it will not be loaded and you will not be able to use it from Python. You can, however, write Python code that will properly load and initialize your module but this approach is way more complex than adding a simple line or lines in the XML.

In our example even if we add <Steppable Type="HeterotypicBoundaryLength"/> to the XML we will not see any calculations being done. Why? Because start and step functions are empty:

```

void HeterotypicBoundaryLength::start() {
}

void HeterotypicBoundaryLength::step(const unsigned int currentStep) {
}

void HeterotypicBoundaryLength::update(CC3DXMLElement *_xmlData, bool _fullInitFlag) {

    //PARSE XML IN THIS FUNCTION

    //For more information on XML parser function please see CC3D code or lookup XML_
    ↪utils API

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
    ↪THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

```

(continues on next page)

(continued from previous page)

```

//boundaryStrategy has information about pixel neighbors
boundaryStrategy=BoundaryStrategy::getInstance();

}

```

We left those implementations empty on purpose. We wanted to show you how you can use steppable to implement functionality that gets called on-demand from Python code. Let us now look at the Python code:

```

1  from cc3d.core.PySteppables import *
2  from cc3d.cpp import CompuCellExtraModules
3
4
5  class HeterotypicBoundarySurfaceSteppable(SteppableBasePy):
6
7      def __init__(self, frequency=1):
8          SteppableBasePy.__init__(self, frequency)
9          self.htbl_steppable_cpp = None
10
11     def start(self):
12         self.htbl_steppable_cpp = CompuCellExtraModules.getHeterotypicBoundaryLength()
13
14     def step(self, mcs):
15         self.htbl_steppable_cpp.calculateHeterotypicSurface()
16
17         print(' HTBL between type 1 and 2 is ',
18               self.htbl_steppable_cpp.getHeterotypicSurface(1, 2))
19
20         print(' HTBL between type 2 and 1 is ',
21               self.htbl_steppable_cpp.getHeterotypicSurface(1, 2))
22
23         print(' HTBL between type 1 and 1 is ',
24               self.htbl_steppable_cpp.getHeterotypicSurface(1, 1))
25
26         print(' HTBL between type 0 and 1 is ',
27               self.htbl_steppable_cpp.getHeterotypicSurface(0, 1))
28
29         print('THIS ENTRY DOES NOT EXIST. HTBL between type 3 and 20 is ',
30               self.htbl_steppable_cpp.getHeterotypicSurface(3, 20))

```

At the top we import `CompuCellExtraModules` that SWIG generates. This Python contains contains Python-wrapper for our `HeterotypicBoundaryLength` C++ steppable. In line 12 we fetch a reference to the steppable and store it in class-accessible `self.htbl_steppable_cpp` variable. In Python steppable `step` function we call C++ function that calculates heterotypic surface (line 15). The next series of `print` statements fetches results for the calculations - see lines 18 21, *etc...*. The output looks as follows:

```
Calculating HTBL for all cell type combinations
HTBL between type 1 and 2 is 3261.0
HTBL between type 2 and 1 is 3261.0
HTBL between type 1 and 1 is 1804.0
HTBL between type 0 and 1 is 321.0
THIS ENTRY DOES NOT EXIST. HTBL between type 3 and 20 is 0.0
Metropolis Fast
total number of pixel copy attempts=65536
Number of Attempted Energy Calculations=3640
Step 2 Flips 367/65536 Energy 1300 Cells 484 Inventory=484
Calculating HTBL for all cell type combinations
HTBL between type 1 and 2 is 3268.0
HTBL between type 2 and 1 is 3268.0
HTBL between type 1 and 1 is 1814.0
HTBL between type 0 and 1 is 326.0
THIS ENTRY DOES NOT EXIST. HTBL between type 3 and 20 is 0.0
```

Note that heterotypic surface between types 1 and 2 is the same as between 2 and 1. This is why, earlier in the C++ code we included two pair indexes:

```
unsigned int pair_index_1 = typePairIndex(cell_type, n_cell_type);
unsigned int pair_index_2 = typePairIndex(n_cell_type, cell_type);
```

Notice also that if we try to access heterotypic surface between types 3 and 20 (none of those types exist in our simulation) we get back 0.0. Why? The answer has to do with the behavior of C++ `std::map` container. If we try to use a key that does not exist in the map the C++ will add this key and initialize the value of the key value-pair to the whatever default constructor of the value type is. In our case our map container has the following type : `std::map<unsigned int, double> typePairHTSurfaceMap` so that key is of unsigned int type and value is of double type. Any modern C++ compiler will put 0.0 as a default value for objects of type double. If you are familiar with Python `defaultdict` class that is a member of standard collections package than you can see similarities. C++ `std::map` behaves in similar way to the `defaultdict`. Therefore when we access `typePairHTSurfaceMap` using key that does not exist C++ will insert this key into `typePairHTSurfaceMap` and set value to 0.0. This is also a reason why the following code works at all:

```
this->typePairHTSurfaceMap[pair_index_1] += unit_surface
```

This brings us to the last remark we want to make regarding the C++ code. Why are we using `unit_surface` and not 1.0? It has to do with various lattices that CC3D supports. For Cartesian 2D or 3D lattice unit surface has value 1.0. However, hexagonal lattices are constructed in such a way that the volume of a voxel is constrained to be 1.0. Therefore from geometry constraints it follows that in 2D on hex lattice unit surface (or length) is $\sqrt{2.0 / (3.0 * \sqrt{3.0})}$ which is approx equal to 0.6204 and in 3D it is $8.0 / (12.0 * \sqrt{2.0} * \text{pow}(9.0 / (16.0 * \sqrt{3.0}), 1.0 / 3.0) * \text{pow}(9.0 / (16.0 * \sqrt{3.0}), 1.0 / 3.0))$ which is approx equal to 0.445

For more information please see http://www.compuCell3d.org/BinDoc/cc3d_binaries/Manuals/HexagonalLattice.pdf as well as in the code of the `BoundaryStrategy` class method `LatticeMultiplicativeFactors` `BoundaryStrategy::generateLatticeMultiplicativeFactors(LatticeType _latticeType, Dim3D dim)` in `CompuCell3D/core/CompuCell3D/Boundary/BoundaryStrategy.cpp`

For completeness we also show SWIG file that was used to generate the wrapper :


```

%module ("threads"=1) CompuCellExtraModules

#include "typemaps.i"

#include <windows.i>

%{

#include "ParseData.h"
#include "ParserStorage.h"
#include <CompuCell3D/Simulator.h>
#include <CompuCell3D/Potts3D/Potts3D.h>

#include <BasicUtils/BasicClassAccessor.h>
#include <BasicUtils/BasicClassGroup.h> //had to include it to avoid problems with
↳template instantiation

// ***** PUT YOUR PLUGIN PARSE DATA AND
↳PLUGIN FILES HERE *****

#include <SimpleVolume/SimpleVolumePlugin.h>
#include <VolumeMean/VolumeMean.h>

//AutogeneratedModules1 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//HeterotypicBoundaryLength_autogenerated

#include <HeterotypicBoundaryLength/HeterotypicBoundaryLength.h>

//GrowthSteppable_autogenerated

#include <GrowthSteppable/GrowthSteppable.h>

// ***** END OF SECTION
↳*****

//have to include all export definitions for modules which are arapped to avoid
↳problems with interpreting by swig win32 specific c++ extensions...
#define SIMPLEVOLUME_EXPORT
#define VOLUMEMEAN_EXPORT
//AutogeneratedModules2 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//HeterotypicBoundaryLength_autogenerated
#define HETEROTYPICBOUNDARYLENGTH_EXPORT
//GrowthSteppable_autogenerated
#define GROWTHSTEPPABLE_EXPORT

#include <iostream>

using namespace std;
using namespace CompuCell3D;

%}

```

(continues on next page)

(continued from previous page)

```

// C++ std::string handling
#include "std_string.i"

// C++ std::map handling
#include "std_map.i"

// C++ std::map handling
#include "std_set.i"

// C++ std::vector handling
#include "std_vector.i"

//have to include all export definitions for modules which are arapped to avoid
↳problems with interpreting by swig win32 specific c++ extensions...
#define SIMPLEVOLUME_EXPORT
#define VOLUMEMEAN_EXPORT

//AutogeneratedModules3 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE
↳INSERTION POINT
//HeterotypicBoundaryLength_autogenerated
#define HETEROTYPICBOUNDARYLENGTH_EXPORT
//GrowthSteppable_autogenerated
#define GROWTHSTEPPABLE_EXPORT

#include <BasicUtils/BasicClassAccessor.h>
#include <BasicUtils/BasicClassGroup.h> //had to include it to avoid problems with
↳template instantiation
#include "ParseData.h"
#include "ParserStorage.h"

// ***** PUT YOUR PLUGIN PARSE DATA AND
↳PLUGIN FILES HERE *****
// REMEMBER TO CHANGE #include to %include
#include <SimpleVolume/SimpleVolumePlugin.h>
// %include <SimpleVolume/SimpleVolumeParseData.h>
// THIS IS VERY IMORTANT STETEMENT WITHOUT IT SWIG will produce incorrect wrapper
↳code which will compile but will not work
using namespace CompuCell3D;

%inline %{
    SimpleVolumePlugin * reinterpretSimpleVolumePlugin(Plugin * _plugin){
        return (SimpleVolumePlugin *)_plugin;
    }

    SimpleVolumePlugin * getSimpleVolumePlugin(){
        return (SimpleVolumePlugin *) Simulator::pluginManager.get("SimpleVolume");
    }
%}

#include <VolumeMean/VolumeMean.h>

%inline %{
    VolumeMean * reinterpretVolumeMean(Steppable * _steppable){
        return (VolumeMean *)_steppable;
    }
%}

```

(continues on next page)

(continued from previous page)

```

    VolumeMean * getVolumeMeanSteppable(){
        return (VolumeMean *) Simulator::steppableManager.get("VolumeMean");
    }
%}

//AutogeneratedModules4 - DO NOT REMOVE THIS LINE IT IS USED BY TWEDIT TO LOCATE CODE_
↪ INSERTION POINT
//HeterotypicBoundaryLength_autogenerated
#include <HeterotypicBoundaryLength/HeterotypicBoundaryLength.h>
%inline %{

    HeterotypicBoundaryLength * getHeterotypicBoundaryLength(){
        return (HeterotypicBoundaryLength *) Simulator::steppableManager.get (
↪ "HeterotypicBoundaryLength");
    }
%}

//GrowthSteppable_autogenerated
#include <GrowthSteppable/GrowthSteppable.h>
%inline %{

    GrowthSteppable * getGrowthSteppable(){
        return (GrowthSteppable *) Simulator::steppableManager.get("GrowthSteppable");
    }
%}

// ***** END OF SECTION_
↪ *****

```


Attaching Custom Attributes To Cells

Cells in CompuCell3D are represented by CellG class - see CompuCell3D/core/CompuCell3D/Potts3D/Cell.h

```
#ifndef CELL_H
#define CELL_H

#ifndef PyObject_HEAD
struct _object; //forward declare
typedef _object PyObject; //type redefinition
#endif

class BasicClassGroup;

namespace CompuCell3D {

/**
 * A Potts3D cell.
 */

class CellG{
public:
    typedef unsigned char CellType_t;
    CellG():
        volume(0),
        targetVolume(0.0),
        lambdaVolume(0.0),
        surface(0),
        targetSurface(0.0),
        lambdaSurface(0.0),
        clusterSurface(0.0),
        targetClusterSurface(0.0),
        lambdaClusterSurface(0.0),
        type(0),
```

(continues on next page)

(continued from previous page)

```

    xCM(0),yCM(0),zCM(0),
    xCOM(0),yCOM(0),zCOM(0),
    xCOMPprev(0),yCOMPprev(0),zCOMPprev(0),
    iXX(0), iXY(0), iXZ(0), iYY(0), iYZ(0), iZZ(0),
    lX(0.0),
    lY(0.0),
    lZ(0.0),
    lambdaVecX(0.0),
    lambdaVecY(0.0),
    lambdaVecZ(0.0),
    flag(0),
    id(0),
    clusterId(0),
    fluctAmpl(-1.0),
    lambdaMotility(0.0),
    biasVecX(0.0),
    biasVecY(0.0),
    biasVecZ(0.0),
    connectivityOn(false),
    extraAttribPtr(0),
    pyAttrib(0)

{}
long volume;
float targetVolume;
float lambdaVolume;
double surface;
float targetSurface;
float angle;
float lambdaSurface;
double clusterSurface;
float targetClusterSurface;
float lambdaClusterSurface;
unsigned char type;
unsigned char subtype;
double xCM,yCM,zCM; // numerator of center of mass expression (components)
double xCOM,yCOM,zCOM; // numerator of center of mass expression (components)
double xCOMPprev,yCOMPprev,zCOMPprev; // previous center of mass
double iXX, iXY, iXZ, iYY, iYZ, iZZ; // tensor of inertia components
float lX,lY,lZ; //orientation vector components - set by MomentsOfInertia_

↪Plugin - read only
    float ecc; // cell eccentricity
    float lambdaVecX,lambdaVecY,lambdaVecZ; // external potential lambda vector_

↪components
    unsigned char flag;
    float averageConcentration;
    long id;
    long clusterId;
    double fluctAmpl;
    double lambdaMotility;
    double biasVecX;
    double biasVecY;
    double biasVecZ;
    bool connectivityOn;
    BasicClassGroup *extraAttribPtr;

```

(continues on next page)

(continued from previous page)

```

    PyObject *pyAttrib;
};

class Cell {
};

class CellPtr{
public:
    Cell * cellPtr;
};
};
#endif

```

As you can see CellG has a number of “standard” attributes. But very often you would like to add new attributes. For example you would like to keep last 50 center of mass positions of each cell to be able to plot recent cell trajectory. How would you do this? A simple approach would be to attach *e.g.* `std::queue` to the `CellG` class. This is a valid approach but it has one major disadvantage. It will require you to recompile almost entire C++ code because `CellG` class is a core class that is used by virtually every single CompuCell3D module. Also, if you would like to share the code with your colleague he would also need to recompile his or her copy of CC3D. Hence while this simple approach would certainly work it is not the most convenient way of adding attributes. What about Python then? Yes, adding new attribute in Python is very simple:

```

cell.dict['cell_x_positions'] = [0.0]*50
cell.dict['cell_y_positions'] = [0.0]*50
cell.dict['cell_z_positions'] = [0.0]*50

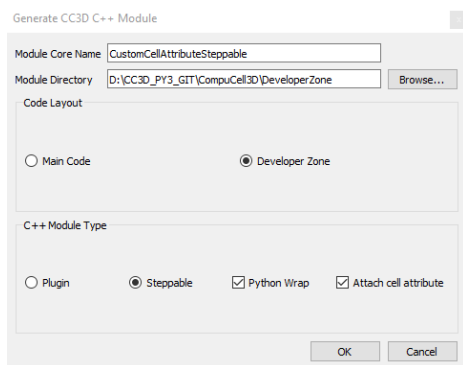
```

Here, we added 3 attributes each one representing last 50 positions x, y, or z coordinates of center of mass. We initialized them to be 0.0 hence the code `[0.0]*50`. In Python when you multiply list by an integer it will return a list that contains multiple copies of the list you originally multiplied (in our case we will get a list with 50 zeros).

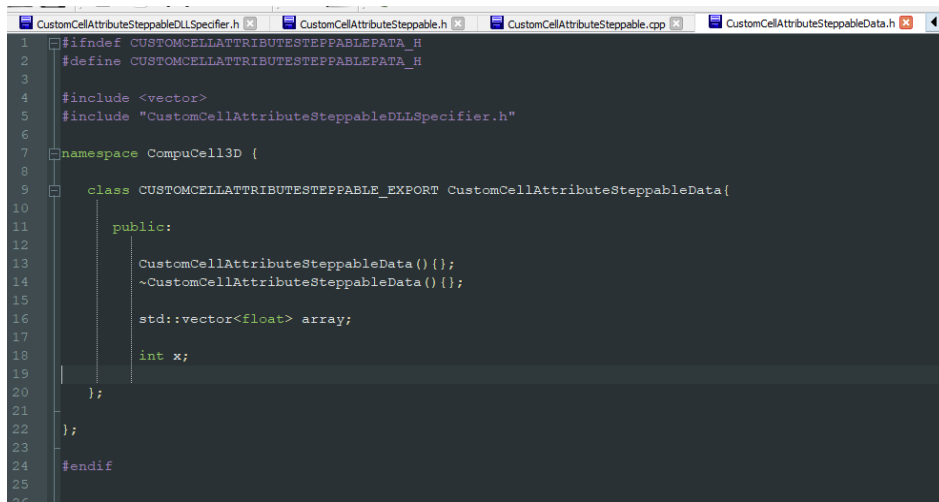
Python approach would certainly work, but what if, for efficiency reasons, you want to stay in C++ world. There is a solution for this that scales nicely i.e. it does not require recompilation of entire code and it allows to attach any C++ class as a cell attribute. This is what we will be teaching you next.

12.1 Constructing Steppable with Custom Class Attached to Each Cell

We begin the usual way - open Twedit++, go to CC3D C++ menu and choose `Generate New Module...` from the menu. There, as before we fill out steppable (we call it `CustomCellAttributeSteppable`) details - making sure to check `Developer Zone` radio button, but in addition to this we also check `Attach Cell Attribute` check box. This ensures that the code that Twedit++ generates contains code that will inform CC3D cell factory object to attach additional cell attribute.



We press OK button and the steppables code with additional attribute will get generated and the code will open in Twedit++ tabs:



The class shown in the editor window will be used during cell construction to create object of this class and attach it to each cell. In other words, once the steppable we have just created gets loaded it will tell CC3D to attach to each cell an object of class CustomCellAttributeSteppableData

```

#ifndef CUSTOMCELLATTRIBUTESTEPPABLEPATA_H
#define CUSTOMCELLATTRIBUTESTEPPABLEPATA_H

#include <vector>
#include "CustomCellAttributeSteppableDLLSpecifier.h"

namespace CompuCell3D {

    class CUSTOMCELLATTRIBUTESTEPPABLE_EXPORT CustomCellAttributeSteppableData{

    public:

        CustomCellAttributeSteppableData() {};
        ~CustomCellAttributeSteppableData() {};

        std::vector<float> array;

        int x;

    };

```

(continues on next page)

(continued from previous page)

```
};

#endif
```

If we look into CustomCellAttributeSteppable init function (this function is called during steppable initialization) we can see a line `potts->getCellFactoryGroupPtr()->registerClass(&customCellAttributeSteppableDataAccessor);`. This line is responsible for telling cell factory object that each new cell should have an object of type CustomCellAttributeSteppableData attached.

```
void CustomCellAttributeSteppable::init(Simulator *simulator, CC3DXMLElement *_xmlData) {
    xmlData=_xmlData;

    potts = simulator->getPotts();

    cellInventoryPtr=& potts->getCellInventory();

    sim=simulator;

    cellFieldG = (WatchableField3D<CellG *> *)potts->getCellFieldG();

    fieldDim=cellFieldG->getDim();

    ExtraMembersGroupAccessorBase *accessorPtr = &
    customCellAttributeSteppableDataAccessor;
    potts->getCellFactoryGroupPtr()->registerClass(accessorPtr);

    simulator->registerSteerableObject(this);

    update(_xmlData,true);
}
```

How do we know that CustomCellAttributeSteppableData is the class whose objects will get attached to each cell? We look into steppable header file and see the following line: `“ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> customCellAttributeSteppableDataAccessor;”`.

This line creates special accessor object that given a pointer to a cell it will fetch attached object of type CustomCellAttributeSteppableData. The exact details of how this is done are beyond the scope of this manual but if you follow the pattern you will be able to attach arbitrary C++ objects to cc3d cells. The pattern is as follows:

1. Add ExtraMembersGroupAccessor member to your module - steppable or a plugin - `ExtraMembersGroupAccessor<ClassYouWantToAttach>`. In our case we add `“ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> customCellAttributeSteppableDataAccessor;”`.
2. Add a function that accesses a pointer to this ExtraMembersGroupAccessor member - in our case we add (see code below) `ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> * getCustomCellAttributeSteppableDataAccessorPtr() {return & customCellAttributeSteppableDataAccessor;}`
3. Register ExtraMembersGroupAccessor object with cell factory (we do it in the init function) of the steppable or plugin - see full init function above:

```
potts->getCellFactoryGroupPtr()->registerClass(&customCellAttributeSteppableDataAccessor);
```

```

#ifndef CUSTOMCELLATTRIBUTESTEPPABLESTEPPABLE_H
#define CUSTOMCELLATTRIBUTESTEPPABLESTEPPABLE_H
#include <CompuCell3D/CC3D.h>
#include "CustomCellAttributeSteppableData.h"
#include "CustomCellAttributeSteppableDLLSpecifier.h"

namespace CompuCell3D {

    template <class T> class Field3D;
    template <class T> class WatchableField3D;

    class Potts3D;
    class Automaton;
    class BoundaryStrategy;
    class CellInventory;
    class CellG;

    class CUSTOMCELLATTRIBUTESTEPPABLE_EXPORT CustomCellAttributeSteppable : public CustomCellAttributeSteppableDataAccessor {

        ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> customCellAttributeSteppableDataAccessor;

        WatchableField3D<CellG *> *cellFieldG;

        Simulator * sim;

        Potts3D *potts;

        CC3DXMLElement *xmlData;

        Automaton *automaton;

        BoundaryStrategy *boundaryStrategy;

        CellInventory * cellInventoryPtr;

        Dim3D fieldDim;

    public:

        CustomCellAttributeSteppable ();

        virtual ~CustomCellAttributeSteppable ();

        // SimObject interface

        virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);

        virtual void extraInit(Simulator *simulator);

        ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> *getCustomCellAttributeSteppableDataAccessorPtr() {return &customCellAttributeSteppableDataAccessor;}

        //steppable interface

```

(continues on next page)

(continued from previous page)

```

virtual void start();

virtual void step(const unsigned int currentStep);

virtual void finish() {}

//SteerableObject interface

virtual void update(CC3DXMLElement *_xmlData, bool _fullInitFlag=false);

virtual std::string steerableName();

virtual std::string toString();

};

};

#endif

```

Now that we know basic rules of adding custom attributes to cells. Let's write a little bit of code that makes use of this functionality. First we will cleanup function that parses XML (we do not need any XML parsing in our example and then we will modify `step` function to store a product of cell id and current MCS in the variable `x` CustomCellAttributeSteppableData object (remember objects of this class will be attached to cell). We will also store x-coordinates of 5 last center of mass positions of each cell.

Here is implementation of the update function where we remove XML parsing code since we are not doing any XML parsing in this particular case:

```

void CustomCellAttributeSteppable::update(CC3DXMLElement *_xmlData, bool _
↳fullInitFlag) {

    //PARSE XML IN THIS FUNCTION

    //For more information on XML parser function please see CC3D code or lookup XML_
↳utils API

    automaton = potts->getAutomaton();

    ASSERT_OR_THROW("CELL TYPE PLUGIN WAS NOT PROPERLY INITIALIZED YET. MAKE SURE_
↳THIS IS THE FIRST PLUGIN THAT YOU SET", automaton)

    //boundaryStrategy has information about pixel neighbors
    boundaryStrategy = BoundaryStrategy::getInstance();

}

```

The implementation of step function is a bit more involved but not by much:

```

1 void CustomCellAttributeSteppable::step(const unsigned int currentStep) {
2

```

(continues on next page)

(continued from previous page)

```

3      CellInventory::cellInventoryIterator cInvItr;
4
5      CellG * cell = 0;
6
7      for (cInvItr = cellInventoryPtr->cellInventoryBegin(); cInvItr !=
↪ cellInventoryPtr->cellInventoryEnd(); ++cInvItr)
8
9      {
10
11         cell = cellInventoryPtr->getCell(cInvItr);
12
13         CustomCellAttributeSteppableData * customCellAttrData =
↪ customCellAttributeSteppableDataAccessor.get (cell->extraAttribPtr);
14
15         //storing cell id multiplied by currentStep in "x" member of the
↪ CustomCellAttributeSteppableData
16         customCellAttrData->x = cell->id * currentStep;
17
18
19
20         // storing last 5 xCOM positions in the "array" vector (part of
↪ CustomCellAttributeSteppableData)
21         std::vector<float> & vec = customCellAttrData->array;
22         if (vec.size() < 5) {
23             vec.push_back (cell->xCOM);
24         }
25         else
26         {
27             for (int i = 0; i < 4; ++i) {
28                 vec[i] = vec[i + 1];
29             }
30             vec[vec.size() - 1] = cell->xCOM;
31         }
32     }
33
34
35     //printouts
36     for (cInvItr = cellInventoryPtr->cellInventoryBegin(); cInvItr !=
↪ cellInventoryPtr->cellInventoryEnd(); ++cInvItr) {
37         cell = cellInventoryPtr->getCell(cInvItr);
38         CustomCellAttributeSteppableData * customCellAttrData =
↪ customCellAttributeSteppableDataAccessor.get (cell->extraAttribPtr);
39
40         cerr << "cell->id=" << cell->id << " mcs = " << currentStep << " attached x
↪ variable = " << customCellAttrData->x << endl;
41
42         cerr << "----- up to last 5 xCOM positions ----- for cell->id " << cell-
↪ id << endl;
43         for (int i = 0; i < customCellAttrData->array.size(); ++i) {
44             cerr << "x_com_pos[" << i << "]= " << customCellAttrData->array[i] << endl;
45         }
46     }
47
48 }

```

Lines 7–11 should be familiar. We iterate over all cells in the simulation and fetch a cell pointer from inventory and store it in local variable `cell`.

In line 13 we make use of our accessor object. Here we are actually fetching object of type `CustomCellAttributeSteppableData` that is attached to each cell. Note that `customCellAttributeSteppableDataAccessor.get` function takes as an input special pointer that is a member of every cell object `cell->extraAttribPtr` and returns a pointer to the object that accessor is associated with in our case it returns a pointer to `CustomCellAttributeSteppableData`.

In line 16 we assign `x` variable of the object of class `CustomCellAttributeSteppableData` to be a product of current cell `id` and current `MCS`.

In lines 21–33 we append current `xCOM` position of current cell to the vector array. We only keep last 5 positions and therefore in the `else` portion lines 25–31 we last 4 positions of the vector to the “front” of the vector and write `xCOM` in the last position of the vector - line 30. Note that the `else` part gets executed only if we determine that vector has already 5 elements. As you can see our attached attribute can store variable number of elements - because we append to vector. In general we can have vectors, lists, maps, queues of arbitrary objects. In fact instead of using `std::vector` it would be better to use `queue` because `queue` container makes it much easier to remove and add elements to and from the beginning and end of the container.

Warning: One thing to remember that computer has a finite memory and if you keep appending you may actually exhaust all operating system memory.

Note: Unlike in Python where we can store arbitrary objects in the list or dictionary, in C++ we need to declare which types we want to store. It makes C++ less flexible but you recoup this minor inflexibility in much faster speed of code execution

The full code for this example can be found in `CompuCell3D/DeveloperZone/Demos/CustomCellAttributesCpp` directory

12.2 Using Python scripting to modify custom C++ attributes

Sometimes you may end up in situation where in addition to modifying custom attributes in C++ you may want to modify them also in Python. In this part of the tutorial we will show you how to do it. If all we want to do is to access `x` variable from `CustomCellAttributeSteppableData` we should be “pre-wired”. Well, almost. You see that when we access objects of `CustomCellAttributeSteppableData` class from within C++ steppable where we declared the accessor object we simply type:

```
CustomCellAttributeSteppableData * customCellAttrData =  
↳ customCellAttributeSteppableDataAccessor.get (cell->extraAttribPtr)
```

However, note that `customCellAttributeSteppableDataAccessor` is declared in the “private” section of `CustomCellAttributeSteppable`. Therefore, it is not “visible” from outside of C++ `CustomCellAttributeSteppable` class. At this point we have three potential solutions:

1. Make the accessor public - not ideal, this is a low-level object that should remain hidden
2. Make a public function that returns a pointer to accessor - again, not ideal because then in Python or in other C++ module we would need to perform a fairly complex fetching of the `CustomCellAttributeSteppableData`
3. Declare a public function that takes a pointer to a cell object and returns attached `CustomCellAttributeSteppableData` object. This solution seems like the cleanest of all three options

Let’s modify a code and add the function that returns pointer to `CustomCellAttributeSteppableData` object. We first modify header file for the steppable class:

```

class CUSTOMCELLATTRIBUTESTEPPABLE_EXPORT CustomCellAttributeSteppable : public
↳Steppable {

    ExtraMembersGroupAccessor<CustomCellAttributeSteppableData>
↳customCellAttributeSteppableDataAccessor;

    WatchableField3D<CellG *> *cellFieldG;

    Simulator * sim;

    // ... we skipped par t fo the code here for brevity
public:

    CustomCellAttributeSteppable ();

    virtual ~CustomCellAttributeSteppable ();

    // SimObject interface

    virtual void init(Simulator *simulator, CC3DXMLElement *_xmlData=0);

    virtual void extraInit(Simulator *simulator);

    ExtraMembersGroupAccessor<CustomCellAttributeSteppableData> *
↳getCustomCellAttributeSteppableDataAccessorPtr() {return &
↳customCellAttributeSteppableDataAccessor;}

    CustomCellAttributeSteppableData * getCustomCellAttribute(CellG * cell);

    // ... we skipped par t fo the code here for brevity

};

```

Now, we add implementation of the `getCustomCellAttribute` function to implementation file

```

CustomCellAttributeSteppableData *
↳CustomCellAttributeSteppable::getCustomCellAttribute(CellG * cell) {

    CustomCellAttributeSteppableData * customCellAttrData =
↳customCellAttributeSteppableDataAccessor.get(cell->extraAttribPtr);
    return customCellAttrData;
}

```

Note: Each time you modify header file for a C++ class that you are wrapping in Python . Make sure you also “refresh” SWIG .i file. It can be as simple as adding extra empty line to `CompuCell3D\DeveloperZone\pyinterface\CompuCellExtraModules\CompuCellExtraModules.i`

At this point we should be able access `CustomCellAttributeSteppableData` objects from “the outside” of the steppable class. Let us now add Python steppable where we can access `CustomCellAttributeSteppable` and `CustomCellAttributeSteppableData`:

```

1 from cc3d.core.PySteppables import *
2 from cc3d.cpp import CompuCellExtraModules
3

```

(continues on next page)

(continued from previous page)

```

4
5 class CustomCellAttributePythonSteppable(SteppableBasePy):
6
7     def __init__(self, frequency=1):
8         SteppableBasePy.__init__(self, frequency)
9         self.custom_attr_steppable_cpp = None
10
11     def start(self):
12         self.custom_attr_steppable_cpp = CompuCellExtraModules.
↪getCustomCellAttributeSteppable()
13
14     def step(self, mcs):
15
16         print ('mcs=', mcs)
17
18         for cell in self.cell_list:
19             custom_cell_attr_data = self.custom_attr_steppable_cpp.
↪getCustomCellAttribute(cell)
20             print('custom_cell_attr_data=', custom_cell_attr_data)
21             print('custom_cell_attr_data.x=', custom_cell_attr_data.x)
22             custom_cell_attr_data.x = cell.id * mcs ** 2
23
24             print('after modification custom_cell_attr_data.x=', custom_cell_attr_
↪data.x)
25             break

```

In line 12 we get access to C++ steppable object and store it in it a class variable `self.custom_attr_steppable_cpp`. In case you are wondering where `getCustomCellAttributeSteppable()` comes from, look into `CompuCell3D\DeveloperZone\pyinterface\CompuCellExtraModules\CompuCellExtraModules.i`. This SWIG wrapper file declares this function and it returns C++ steppable object. This function is generated automatically by Twedit++:

```

%inline %{
    CustomCellAttributeSteppable * getCustomCellAttributeSteppable() {
        return (CustomCellAttributeSteppable *) Simulator::steppableManager.get (
↪"CustomCellAttributeSteppable");
    }
}

```

Coming back to our Python code we see that inside for loop we print to the screen the `CustomCellAttributeSteppableData` object (line 20) and also print `x` member of this object. Later we modify and print to the screen the `x` variable of the object and we only do it for the first cell we encounter during iteration over all cells to make output more concise (see `break` statement at the end of the loop)

The output looks encouraging:

```
x_com_pos[2]=72.0625
x_com_pos[3]=72
cell->id=81 mcs = 3 attached x variable = 243
----- up to last 5 xCOM positions ----- for cell->id 81
x_com_pos[0]=79
x_com_pos[1]=78.8936
x_com_pos[2]=78.8
x_com_pos[3]=78.7273
mcs= 3
custom_cell_attr_data= <cc3d.cpp.CompuCellExtraModules.CustomCellAttributeSteppableData; proxy of <Swig Object of type 'CompuCell3D::CustomCellAttributeSteppableData *' at 0x000001CCAED37330> >
custom_cell_attr_data.x= 3
after modification custom_cell_attr_data.x= 9
CALLING CLOSE EVENT FROM SIMTAB
EXITING WITH ERROR CODE= 0

Process finished with exit code 0
```

We can see - look at the lines:

```
custom_cell_attr_data.x= 3
after modification custom_cell_attr_data.x= 9
```

that we can access and modify x variable of the CustomCellAttributeSteppableData object that is attached to each cell.

What about the array member of CustomCellAttributeSteppableData. Remember, in C++ it is of type `std::vector<float>`. Can we access it? Can we modify it? Let's us try:

```
1 from cc3d.core.PySteppables import *
2 from cc3d.cpp import CompuCellExtraModules
3
4
5 class CustomCellAttributePythonSteppable(SteppableBasePy):
6
7     def __init__(self, frequency=1):
8         SteppableBasePy.__init__(self, frequency)
9         self.custom_attr_steppable_cpp = None
10
11     def start(self):
12         self.custom_attr_steppable_cpp = CompuCellExtraModules.
↳ getCustomCellAttributeSteppable()
13
14     def step(self, mcs):
15         print('mcs=', mcs)
16
17         for cell in self.cell_list:
18             custom_cell_attr_data = self.custom_attr_steppable_cpp.
↳ getCustomCellAttribute(cell)
19             print('custom_cell_attr_data=', custom_cell_attr_data)
20             print('custom_cell_attr_data.x=', custom_cell_attr_data.x)
21
22             custom_cell_attr_data.x = cell.id * mcs ** 2
23
24             print('after modification custom_cell_attr_data.x=', custom_cell_attr_
↳ data.x)
25
26             print('custom_cell_attr_data.array=', custom_cell_attr_data.array)
27             print('custom_cell_attr_data.array[0]=', custom_cell_attr_data.array[0])
28
29             if len(custom_cell_attr_data.array) < 5:
30                 custom_cell_attr_data.array.push_back(100.0)
31             print('custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] =
↳ ,
```

(continues on next page)

(continued from previous page)

```

32         custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1])
33
34     break

```

In lines 26–27 we print the type of `custom_cell_attr_data.array` as well as the first element. Later, in lines 29–32 we are appending elements to the vector using `push_back` C++ function (because `array` is a C++ object wrapped in Python). Notice that we are doing double append for first cell. First append (`push_back`) happens in C++ and in Python we are doing a second one. This, somewhat artificial example shows how to access and modify custom attributes from C++ and from Python in a single simulation.

Here is the output:

```

----- up to last 5 xCOM positions ----- for cell->id 81
x_com_pos[0]=78.8936
x_com_pos[1]=78.8936
x_com_pos[2]=78.7333
x_com_pos[3]=78.3404
mcs= 3
custom_cell_attr_data= <cc3d.cpp.CompuCellExtraModules.CustomCellAttributeSteppableData; proxy of <Swig Object of type 'CompuCell3D::CustomCellAttributeSteppableData' at 0x00000196312DF090> >
custom_cell_attr_data.x= 3
after modification custom_cell_attr_data.x= 9
custom_cell_attr_data.array= <cc3d.cpp.PlayerPython.Vectorfloat; proxy of <Swig Object of type 'std::vector< float,std::allocator< float > >' at 0x00000196312DF0F0> >
custom_cell_attr_data.array[0]= 100.0
custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] = 22.934782028198242
CALLING CLOSE EVENT FROM SIMTAB
EXITING WITH ERROR CODE= 0
Process finished with exit code 0

```

12.2.1 Adding a complex type to attached attribute and accessing it from Python

So far things worked as a charm. We were able to access simple type variables (`x`), STL vectors `array`. So, perhaps we can try adding something more complex to the `CustomCellAttributeSteppableData`, for example let us add `std::map<long int, std::vector<int> >` which is C++ dictionary (map) that uses long integers as keys and stores vectors of type integer:

```

#ifndef CUSTOMCELLATTRIBUTESTEPPABLEDATA_H
#define CUSTOMCELLATTRIBUTESTEPPABLEDATA_H

#include <vector>
#include "CustomCellAttributeSteppableDLLSpecifier.h"

namespace CompuCell3D {

    class CUSTOMCELLATTRIBUTESTEPPABLE_EXPORT CustomCellAttributeSteppableData{

    public:

        CustomCellAttributeSteppableData(){};
        ~CustomCellAttributeSteppableData(){};

        std::vector<float> array;
        std::map<long int, std::vector<int> > simple_map;

        int x;

    };

};

#endif

```

After we recompile (remember to refresh `CompuCellExtraModules.i`) and try running the following Python code:

```

1 from cc3d.core.PySteppables import *
2 from cc3d.cpp import CompuCellExtraModules
3
4
5 class CustomCellAttributePythonSteppable(SteppableBasePy):
6
7     def __init__(self, frequency=1):
8         SteppableBasePy.__init__(self, frequency)
9         self.custom_attr_steppable_cpp = None
10
11     def start(self):
12         self.custom_attr_steppable_cpp = CompuCellExtraModules.
↳ getCustomCellAttributeSteppable()
13
14     def step(self, mcs):
15         print('mcs=', mcs)
16
17         for cell in self.cell_list:
18             custom_cell_attr_data = self.custom_attr_steppable_cpp.
↳ getCustomCellAttribute(cell)
19             print('custom_cell_attr_data=', custom_cell_attr_data)
20             print('custom_cell_attr_data.x=', custom_cell_attr_data.x)
21
22             custom_cell_attr_data.x = cell.id * mcs ** 2
23
24             print('after modification custom_cell_attr_data.x=', custom_cell_attr_
↳ data.x)
25
26             print('custom_cell_attr_data.array=', custom_cell_attr_data.array)
27             print('custom_cell_attr_data.array[0]=', custom_cell_attr_data.array[0])
28
29             if len(custom_cell_attr_data.array) < 5:
30                 custom_cell_attr_data.array.push_back(100.0)
31             print('custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] =
↳ ',
32                 custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1])
33
34             simple_map = custom_cell_attr_data.simple_map
35
36             print('simple_map.size()=', simple_map.size())

```

we will get an error when we try to get number of elements stored in the map (should be 0):

```

Traceback (most recent call last):
  File "D:\CC3D_PY3_GIT\cc3d\CompuCellSetup\sim_runner.py", line 77, in run_cc3d_
↳ project
    exec(code, globals(), locals())
  File "D:\CC3D_PY3_
↳ GIT\CompuCell3D\DeveloperZone\Demos\CustomCellAttributesPython\Simulation\CustomCellAttributesPython
↳ .py", line 6, in <module>
    CompuCellSetup.run()
  File "D:\CC3D_PY3_GIT\cc3d\CompuCellSetup\simulation_setup.py", line 117, in run
    main_loop_fcn(simulator, simthread=simthread, steppable_registry=steppable_
↳ registry)
  File "D:\CC3D_PY3_GIT\cc3d\CompuCellSetup\simulation_setup.py", line 583, in main_
↳ loop_player

```

(continues on next page)

(continued from previous page)

```

    steppable_registry.step(cur_step)
File "D:\CC3D_PY3_GIT\cc3d\core\SteppableRegistry.py", line 169, in step
    steppable.step(_mcs)
File "D:\CC3D_PY3_
→GIT\CompuCell3D\DeveloperZone\Demos\CustomCellAttributesPython\Simulation\CustomCellAttributesPython
→py", line 36, in step
    print('simple_map.size()=', simple_map.size())
AttributeError: 'SwigPyObject' object has no attribute 'size'

```

Why the error? Simply put we did not tell SWIG about the complex types we are using for member `simple_map`. You may ask how come before when we had `std::vector<int> array;` things worked. They worked because elsewhere in the CompuCell3D main python wrapper we told SWIG about template `std::vector<int>`. However now that we are dealing with `std::map<long int, std::vector<int> > simple_map;` we need to tell SWIG how to make those object available. It is actually quite easy to do. We add the following lines to `CompuCellExtraModules.i`:

For each template we are using in the our extra attribute we give it a name (e.g. `vector_int`) and list its type -
`%template (vector_int) std::vector<int>;`

After this fix when we try to run the earlier Python code we would get the following output:

```

mcs= 0
custom_cell_attr_data= <cc3d.cpp.CompuCellExtraModules.CustomCellAttributeSteppableData; proxy of <Swig Object of type 'CompuCell3D::CustomCellAttributeSteppableData' at 0x0000026A3506F720> >
custom_cell_attr_data.x= 0
after modification custom_cell_attr_data.x= 0
custom_cell_attr_data.array= <cc3d.cpp.PlayerPython.vectorfloat; proxy of <Swig Object of type 'std::vector< float,std::allocator< float >' at 0x0000026A3506F750> >
custom_cell_attr_data.array[0]= 23.0
custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] = 100.0
simple_map.size()= 0

```

As we can see the size of the map comes up as zero because we did not put any elements in it. Let's add a code that puts something in the map:

```

1  from cc3d.core.PySteppables import *
2  from cc3d.cpp import CompuCellExtraModules
3
4
5  class CustomCellAttributePythonSteppable(SteppableBasePy):
6
7      def __init__(self, frequency=1):
8          SteppableBasePy.__init__(self, frequency)
9          self.custom_attr_steppable_cpp = None
10
11      def start(self):
12          self.custom_attr_steppable_cpp = CompuCellExtraModules.
→getCustomCellAttributeSteppable()
13
14      def step(self, mcs):
15          print('mcs=', mcs)
16
17          for cell in self.cell_list:
18              custom_cell_attr_data = self.custom_attr_steppable_cpp.
→getCustomCellAttribute(cell)
19              print('custom_cell_attr_data=', custom_cell_attr_data)
20              print('custom_cell_attr_data.x=', custom_cell_attr_data.x)
21
22              custom_cell_attr_data.x = cell.id * mcs ** 2
23
24              print('after modification custom_cell_attr_data.x=', custom_cell_attr_
→data.x)

```

(continues on next page)

(continued from previous page)

```

25
26     print('custom_cell_attr_data.array=', custom_cell_attr_data.array)
27     print('custom_cell_attr_data.array[0]=' , custom_cell_attr_data.array[0])
28
29     if len(custom_cell_attr_data.array) < 5:
30         custom_cell_attr_data.array.push_back(100.0)
31     print('custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] =
↪ ',
32           custom_cell_attr_data.array[len(custom_cell_attr_data.array) - 1])
33
34     simple_map = custom_cell_attr_data.simple_map
35
36     print('simple_map.size()=' , simple_map.size())
37     vec = CompuCellExtraModules.vector_int()
38     vec.push_back(20)
39     vec.push_back(30)
40     simple_map[cell.id] = vec
41
42     print('simple_map[cell.id]=' , simple_map[cell.id])
43
44     break

```

In line 37 we create a C++ vector of integers using “CompuCellExtraModules.vector_int()” call. Remember, vector_int is precisely template identifier we added in SWIG CompuCellExtraModules.i file. Now we are simply invoking constructor for this type. In the next two lines 38–39 we push back two integers to the newly created vector and finally in line 40 we store this vector in the map that is part of the object that is attached to a cell. To check if we can retrieve the stored vector we use code from line 42. The output is as follows:

```

mcs= 1
custom_cell_attr_data= <cc3d.cpp.CompuCellExtraModules.CustomCellAttributeSteppableData; proxy of <Swig Object of type 'CompuCell3D::CustomCellAttributeSteppableData **' at 0x0000026A35D04480> >
custom_cell_attr_data.x= 1
after modification custom_cell_attr_data.x= 1
custom_cell_attr_data.array= <cc3d.cpp.PlayerPython.vectorfloat; proxy of <Swig Object of type 'std::vector< float,std::allocator< float > > **' at 0x0000026A35D04540> >
custom_cell_attr_data.array[0]= 22.9375
custom_cell_attr_data.array[len(custom_cell_attr_data.array)-1] = 100.0
simple_map.size()= 1
simple_map[cell.id]= (20, 30)

```

12.3 Summary

In this section we learned how to attach C++ attribute to each cell, how to modify it from C++ and how to interact with complex types that are part of the attached attribute at the Python level.

CHAPTER 13

Debugging CC3D using GDB

Sometimes when you execute simulation and encounter software crash it is useful to do a quick introspection to see what went wrong. IN this section we will show you how to inspect CC3D call trace using GDB.

Note: Provided recipe works only on OSX and Linux

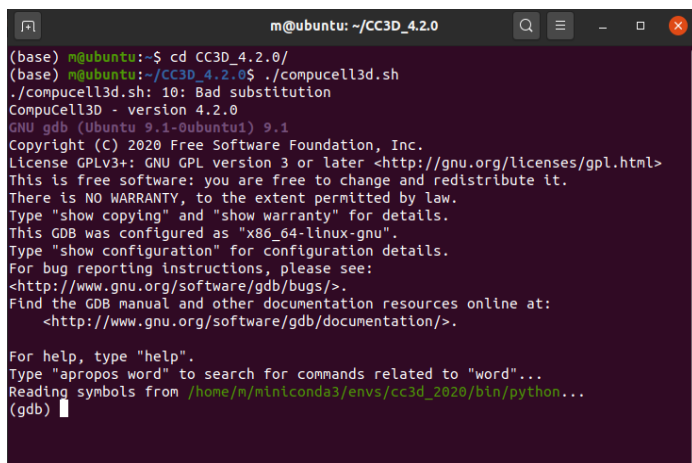
First it is useful to create a copy of CC3D run scripts because we will be modifying those. This way you will have a copy to revert to after you are done with debugging. LEt us start with modifications to `compuce113d.sh` (on OSX `compuce113d.command`) script. This script launches Player and allows you to run simulation. When we open `compuce113d.sh` in editor, towards the end of the file you will see a line that looks as follows:

```
${PYTHON_EXEC} ${PREFIX_CC3D}/lib/site-packages/cc3d/player5/compuce113d.pyw $* --  
↪currentDir=${current_directory}
```

we will replace this line with

```
gdb ${PYTHON_EXEC}
```

Save the script and run it. As a result we will be dropped to gdb shell that is setup to debug Python scripts (`${PYTHON_EXEC}` points to Python interpreter)

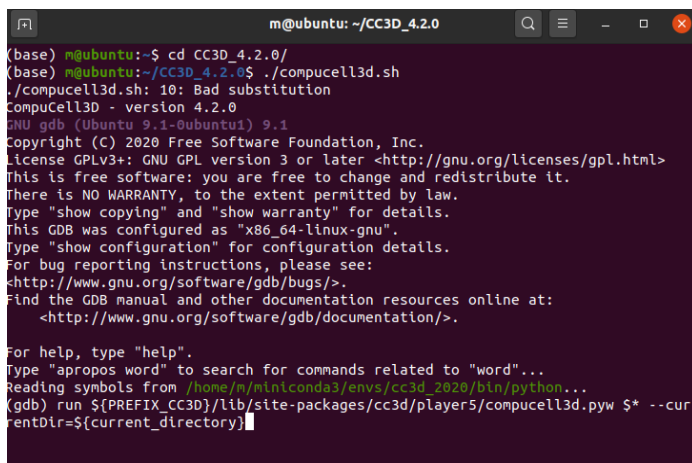


```
m@ubuntu: ~/CC3D_4.2.0
(base) m@ubuntu:~$ cd CC3D_4.2.0/
(base) m@ubuntu:~/CC3D_4.2.0$ ./compucell3d.sh
./compucell3d.sh: 10: Bad substitution
CompuCell3D - version 4.2.0
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /home/n/miniconda3/envs/cc3d_2020/bin/python...
(gdb)
```

Next, in Python shell we will run actual player by typing

```
run ${PREFIX_CC3D}/lib/site-packages/cc3d/player5/compucell3d.pyw $* --currentDir=${current_directory}
```

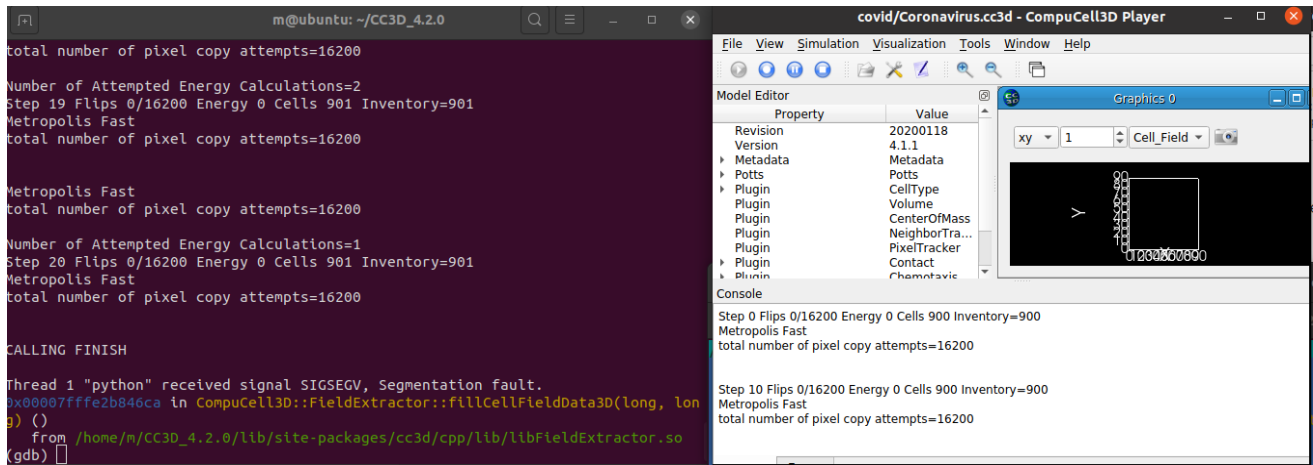


```
m@ubuntu: ~/CC3D_4.2.0
(base) m@ubuntu:~$ cd CC3D_4.2.0/
(base) m@ubuntu:~/CC3D_4.2.0$ ./compucell3d.sh
./compucell3d.sh: 10: Bad substitution
CompuCell3D - version 4.2.0
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /home/n/miniconda3/envs/cc3d_2020/bin/python...
(gdb) run ${PREFIX_CC3D}/lib/site-packages/cc3d/player5/compucell3d.pyw $* --currentDir=${current_directory}
```

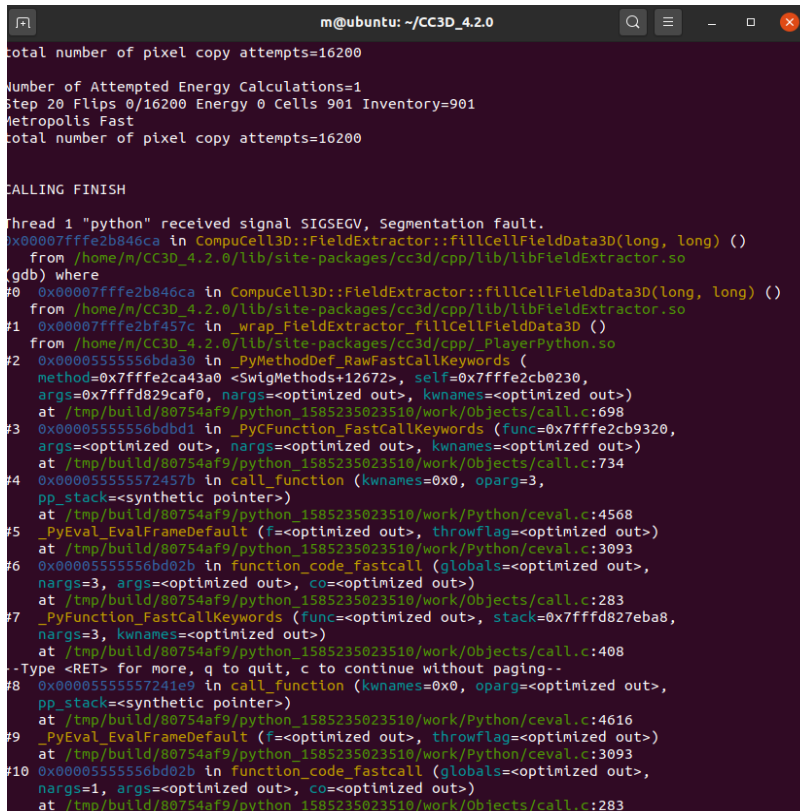
the run command tells gdb to start running the program that we pass to gdb when we invoked gdb shell. In our case this program is a Python interpreter, exactly what we want. The remaining arguments of the run line are the arguments we are passing to the program we are debugging. In our case we pass `${PREFIX_CC3D}/lib/site-packages/cc3d/player5/compucell3d.pyw $* --currentDir=${current_directory}` which means that Python interpreter will run `player5/compucell3d.pyw` executable script that takes `$* --currentDir=${current_directory}` as arguments.

After the player pops up we load simulation and run it as if it were a normal CC3D run. This time however we are running in the debugger shell and as you can see in the left panel we are getting debug output. In this case we see a crash occurring:



The crash happened at the end of the simulation (we turned off thread synchronization code to trigger crash)

As you can see, the crash happened in `fillCellFieldData3D` function. To get full call stack trace we can type `where` in the gdb shell to get more information



When you experience CC3D crash it is useful to take this extra step and get more information to figure out where the actual problem occurs. Sending this information to developers can fast-track the software patch

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`